How to Extract the Last Item from a String Column in PySpark

Authored by

stats writer

January 18, 2026

Mastering String Manipulation in PySpark

The ability to efficiently manipulate and transform complex data structures is fundamental to large-scale data processing. In environments leveraging PySpark, a common requirement involves parsing column data, specifically dealing with string fields that contain multiple delimited values. Often, data scientists need to isolate a specific element from these strings, such as retrieving the last name from a full employee name list or extracting a final path component from a URL structure. This task, while seemingly simple, requires careful utilization of specialized functions available within the PySpark SQL library to ensure performance and correctness across massive datasets. Understanding how to correctly employ array indexing after splitting a string is key to mastering this particular data transformation challenge.

This guide provides an expert-level walkthrough on how to use PySpark functions to effectively split a string column within a DataFrame and precisely extract the last item resulting from that separation. We will focus on combining the power of the **split function** with dynamic indexing capabilities, allowing the logic to handle variable-length strings seamlessly. This method is highly recommended over less efficient iterative approaches, as it leverages Spark's optimized vectorized operations.

To achieve this precise extraction, we rely on a combination of built-in functions: `split`, `col`, and `size`. The core syntax involves creating a temporary column holding the array of split elements, and then referencing the last position of that array using dynamic indexing (array size minus one). This technique ensures robust data handling, regardless of whether the original string contains two words or twenty. This approach is standardized, scalable, and fully integrated with the PySpark framework, making it the preferred solution for production environments.

The Core Challenge: Splitting Strings and Accessing Elements

When working with messy or concatenated data, the first step is often normalization. If a column contains multiple pieces of information joined by a consistent delimiter--such as a space, comma, or pipe--we must first break that single string into an array or list of its component parts. PySpark provides the powerful `split` function specifically for this purpose. However, simply splitting the string is only half the battle; the resulting structure is a complex type (an array) that must then be correctly navigated to pull out the desired element.

Traditional programming languages might allow simple negative indexing (e.g., `array`) to access the last item. Unfortunately, PySpark's native array indexing for SQL expressions requires explicit knowledge of the array's boundaries. Since the length of the string--and thus the resulting array--can vary row-by-row, we cannot use a fixed index like or . We must instead dynamically calculate the maximum index for each row. This is where the `size` function becomes indispensable,

providing the count of elements in the newly created array column.

The core challenge, therefore, lies in this dynamic calculation: accessing the last element requires knowing the total size of the array and subtracting one (since array indexing is zero-based). By combining the split function, which turns the string into an array, with the `size` function to determine the array's length, we can construct a single, highly efficient chained operation that achieves our goal. This method avoids the need for defining complex User Defined Functions (UDFs) for such a common transformation, thereby maximizing performance.

Essential PySpark Functions for String Operations

Achieving the goal of splitting and extracting the last item relies on three specific functions imported from `pyspark.sql.functions`. Understanding the role of each is critical for constructing the robust transformation logic.

The **split function**: This function takes a column and a delimiter string as arguments. It returns a new column containing an array of strings, where the input string has been broken up based on the provided delimiter. For instance, if the delimiter is a space (' '), a string like "First Middle Last" will become the array . This is the foundational step in our process.

The **`size` function**: This utility function is designed to operate on array columns. It calculates and returns the number of elements contained within the array for each row. We need the size to calculate the correct zero-based index of the last element. If an array has three elements, the size is 3, meaning the last element is at index 3 - 1 = 2.

The **`col` function**: Used to reference an existing column by its name, particularly when performing complex operations or renaming. While often implicit in simple column selections, using `col()` explicitly helps maintain clarity, especially when performing array indexing operations on a newly created or temporary column.

By chaining these functions using the `withColumn` operation on the DataFrame, we can define the required transformation in a highly declarative and expressive manner. The initial application of `split` creates the necessary intermediate array structure, which is then immediately used in the second `withColumn` call alongside `size` to perform the targeted extraction. This two-step process, performed within a single transformation block, is the standard practice for efficient string manipulation in large-scale Apache Spark jobs.

Step-by-Step Implementation of the Solution (The Code Breakdown)

The following concise code block represents the optimal method for splitting a string column, such as an `employees` field, and retrieving only the last component. This solution is designed for

maximum efficiency by leveraging PySpark's native SQL functions rather than Python UDFs.

from pyspark.sql.functions import split, col, size

# Step 1: Split the 'employees' string column by a space delimiter.
# The result is temporarily stored in a new column named 'new' as an array type.
df_temp = df.withColumn('new', split('employees', ' '))

# Step 2: Access the last element of the array in 'new' using dynamic indexing: size('new') - 1.
# The resulting value replaces the array in the 'new' column.
df_new = df_temp.withColumn('new', col('new'))

In the original implementation shown below, the two steps are efficiently chained together, reusing the intermediate column name (which is good for minimizing variable definitions).

from pyspark.sql.functions import split, col, size

#create new column that contains only last item from employees column
df_new = df.withColumn('new', split('employees', ' '))
.withColumn('new', col('new'))

This particular example demonstrates the process using the `employees` column and a single space character (' ') as the delimiter. The first `withColumn` call transforms the input string into an array. The second `withColumn` call then performs the critical array indexing. It calculates the total number of elements using `size('new')`, subtracts 1 to get the zero-based index of the last item, and uses this dynamic index to retrieve the final element. This result is then stored back into the column, effectively replacing the array with the desired scalar value.

Practical Example: Setting Up the Sample DataFrame

To fully illustrate the functionality, let us define a simple DataFrame containing employee identifiers and associated sales figures. The `employees` column is structured such that each employee name might contain two, three, or even four space-separated components, simulating real-world variability where name complexity differs across records.

Before we apply the splitting logic, we must ensure our Spark environment is correctly initialized and the data is loaded into the DataFrame structure. We initialize a `SparkSession` and then define the sample data. Notice the intentional variability in the employee names: "Andy Bob Chad" has three components, while "Ian John Ken Liam" has four. This variability is essential to demonstrate the robustness of the `size` function in accurately identifying the last element regardless of the string's length.

```
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()

+-----------------+-----+
| employees|sales|
+-----------------+-----+
| Andy Bob Chad| 200|
| Doug Eric| 139|
| Frank Greg Henry| 187|
|Ian John Ken Liam| 349|
+-----------------+-----+
```

The initial DataFrame now clearly shows the challenge: we need to isolate "Chad," "Eric," "Henry," and "Liam" into a dedicated column. If we attempted to use a static index (e.g.,), we would fail for the "Doug Eric" record, which only has elements up to index 1, leading to errors or null values. This setup confirms the necessity of using the dynamic indexing approach powered by the `size` function.

Executing the Transformation: Splitting and Extracting the Final Value

Now that the data is prepared, we apply the PySpark transformation syntax developed earlier. Our objective is to split the strings in the `employees` column and populate a new column, which we will name `last`, with the last name component from each record. This execution leverages the optimized nature of PySpark SQL functions, ensuring high throughput for large-scale data processing.

We import the necessary functions--`split`, `col`, and `size`--and execute the transformation chain. We explicitly name the new column `last` in the transformation below to match the desired output schema shown in the `df_new.show()` result.

from pyspark.sql.functions import split, col, size

```
#create new column that contains only last item from employees column (named 'last')
df_new = df.withColumn('last', split(col('employees'), ' '))
.withColumn('last', col('last'))

#view new DataFrame
df_new.show()
```

```
+-----------------+-----+-----+
| employees|sales| last|
+-----------------+-----+-----+
| Andy Bob Chad| 200| Chad|
| Doug Eric| 139| Eric|
| Frank Greg Henry| 187|Henry|
|Ian John Ken Liam| 349| Liam|
+-----------------+-----+-----+
```

The resulting DataFrame, `df_new`, successfully incorporates the new `last` column. Crucially, the transformation correctly isolated "Chad" from the three-component string "Andy Bob Chad" and "Liam" from the four-component string "Ian John Ken Liam." This success confirms that our approach, using `size` for dynamic indexing, is fundamentally sound and robust against variations in the input string length. This capability is vital for production systems dealing with non-standardized user input or legacy datasets.

Analyzing the Results and Conclusion

The transformation's output confirms the efficacy of combining the split function and the `size` function. This approach ensures that the array indexing is always correct, regardless of the number of elements resulting from the split. This eliminates the risk of index-out-of-bounds errors that are common when attempting fixed indexing on variable-length arrays. Since `split` returns an array of strings, the subsequent indexing operation correctly yields a single string value, making the new column suitable for standard relational operations, filtering, or grouping.

To summarize, when seeking to retrieve the last component of a delimited string in a PySpark DataFrame, the recommended, production-ready technique is to use chained `withColumn` calls utilizing `split` and `size`. The core principle is simple yet powerful: calculate the length of the split

array dynamically and access the index at `Length - 1`. This method stands superior to custom UDFs in terms of performance and scalability across distributed clusters.

For developers seeking deeper understanding, the official documentation provides comprehensive details on the PySpark **split function** and related array manipulation tools. Mastering this type of operation is foundational to complex data cleansing and preparation within the PySpark environment.

Further PySpark Tutorials

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