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

Authored by

**stats writer**

February 4, 2026

# PySpark: Splitting Strings in a Column and Extracting the Final Element

This detailed guide provides a formal, efficient methodology for manipulating string data within a PySpark DataFrame. Specifically, we address the common requirement of splitting a string column based on a defined delimiter and subsequently extracting the final element from the resulting array or list. This technique is indispensable for data cleaning and feature engineering tasks where contextual information is often concatenated into single strings.

To achieve this complex operation efficiently within the distributed computing framework of PySpark, we leverage powerful built-in functions from the `pyspark.sql.functions` module. The core of the process involves three key steps: first, transforming the string into a column of arrays using the `split()` function; second, determining the length of that array dynamically using the `size()` function; and finally, using array indexing to retrieve the element at the index corresponding to the last position. This approach ensures high performance and scalability across large datasets, avoiding costly User Defined Functions (UDFs).

## The Fundamental Technique: Combining Split and Indexing

The challenge in extracting the last item lies in the fact that the resulting arrays, after the split operation, may not have a uniform length across all rows. Traditional indexing (e.g., ) would only work if all strings contained the same number of components. PySpark addresses this beautifully by allowing us to calculate the required index dynamically using its powerful SQL functions.

The syntax below illustrates the necessary imports and the chaining of column operations required to perform the split and extraction in a single, fluent operation. We utilize `.withColumn()` twice: first to perform the split, creating a temporary array column, and second, to access the final element of that newly created array column.

**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 of splitting the input string column, named `employees`, using a standard space character as the partitioning delimiter. Once the string is partitioned into a list of elements, the index of the last item is calculated by taking the total size of the array and subtracting one (since array indexing is zero-based). Although the code utilizes

`'new'` for the column name in intermediate steps, in practice, you should rename the final column to something descriptive, such as `last_name` or `extracted_value`, to maintain clarity in your data structure.

## Prerequisites and Environment Setup

Before executing any data transformation, it is essential to ensure the PySpark environment is properly configured and the required libraries are imported. All operations involving column manipulation and SQL functions necessitate the use of the `pyspark.sql.functions` module. Furthermore, a valid SparkSession must be initialized to interact with the underlying Spark cluster.

A DataFrame is the core structure for handling tabular data in PySpark, providing an optimized way to process data in parallel. The structure we are targeting for this demonstration contains employee names, which are stored as single strings, and corresponding sales figures.

The following code snippet demonstrates the initialization of the Spark context and the creation of our sample dataset. This ensures reproducibility and provides a concrete foundation upon which to apply our string splitting logic.

**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|
+-----------------+-----+

As observed in the output, the `employees` column contains varying numbers of names separated by spaces. Row 2, for instance, contains two names, while Row 4 contains four names. This variability is precisely why relying on the `size()` function for dynamic array indexing is crucial for a robust solution.

## Detailed Implementation of the Transformation

Our objective is clear: we need to separate the names in the `employees` column and extract only the final name (which we assume represents the last name or family name) into a new column. We will utilize the imported functions `split`, `col`, and `size` to perform this intricate array manipulation.

The transformation process is executed via method chaining on the DataFrame. The first use of `withColumn` converts the string into an array. If the string is 'Ian John Ken Liam', the result of the split operation using a space delimiter is the array .

The second, and most critical, operation redefines the column. We access the column containing the newly created array and use the array indexing syntax (). Inside the index brackets, we calculate the required index: `size('new') - 1`. The size function determines the total number of elements, and subtracting one accounts for Python's zero-based indexing system, effectively targeting the last element regardless of the array's length.

We can use the following syntax to apply this logic and view the resulting DataFrame:

**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'))

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

Upon reviewing the final output, observe the newly generated column, labeled `last`. This column successfully contains only the final name extracted from the list of names in the `employees` column for every row. Crucially, the process was able to correctly handle strings with two, three, and four names, confirming the effectiveness of dynamic indexing via the `size()` function.

## Deconstructing the PySpark Functions Used

Understanding the specific roles of the utilized PySpark functions is paramount for effective data engineering. The solution relies entirely on high-performance built-in functions, which are optimized for Spark's distributed architecture.

The `split(str, pattern)` function, available in `pyspark.sql.functions`, is responsible for the initial transformation. It takes the target column (a string) and a regular expression pattern (our delimiter) and returns a new column of type `Array`. This is the foundation upon which subsequent array operations are performed. For instance, `split function('employees', ' ')` turns the string into an addressable array of names.

The `size(col)` function is critical for dynamic indexing. It accepts an array or map type column reference and returns an integer representing the number of elements it contains. Because array indices begin at zero, if an array has a size of `N`, the last element is always located at index `N - 1`. Thus, `size function('new') - 1` provides the exact index of the required item, regardless of list variation.

Finally, the `col(name)` function is used to reference columns explicitly. While referencing a column by its string name is often sufficient in PySpark methods, using `col()` is crucial when chaining operations or applying specific array indexing, as seen in `col('new')`. This syntax leverages Spark's ability to treat the array column like a list, allowing retrieval of elements based on the calculated index.

## Handling Complex Delimiters and Edge Cases

While the example above used a simple space delimiter, the `split()` function is powerful because it accepts regular expressions. This allows us to handle complex string separation requirements, such as splitting by commas, tabs, or multiple special characters.

For instance, if the employee names were separated by a semicolon and a space (e.g., 'Andy;

Bob; Chad'), the pattern used in the split function would need to be updated to `';\s*'` to correctly handle the separator and any surrounding whitespace. It is important to remember that PySpark's `split` uses Java regular expression syntax, requiring careful handling of escape characters.

Two common edge cases must be considered:

**Empty or Null Strings:** If a row contains a `null` value in the input column, the resulting array will also be `null`, and the final extracted item will propagate the `null` value, which is generally desired behavior. If the string is empty (`""`), the split function usually returns an array containing a single empty string (). Using `size() - 1` would still correctly index this array, though the extracted value might be an empty string. Robust solutions often incorporate `when().otherwise()` clauses to handle these empty strings explicitly before splitting.

**Delimiter Not Found:** If the specified delimiter does not exist in the string (e.g., splitting "John Smith" by a comma), the resulting array will contain the entire original string as its only element (e.g., ). The size function will return 1, and the index `1 - 1 = 0` will correctly retrieve the full string.

## Performance Considerations and Alternatives

Using built-in PySpark functions, as demonstrated here, is the best practice for performance. These functions are highly optimized, compiled directly into the underlying Java Virtual Machine (JVM) code, and executed efficiently across the distributed cluster.

An alternative approach, often tempting for users familiar with native Python, is the use of User Defined Functions (UDFs). A UDF would involve writing a Python function that splits the string and returns the last item, then registering and applying that function to the column. While conceptually simpler, UDFs introduce serialization overhead between the Python executor and the JVM, leading to significantly reduced performance, especially on large-scale DataFrame operations.

Therefore, whenever complex string and array manipulations are required in PySpark, data engineers should prioritize finding combinations of native functions (like `split`, `size`, `element_at`, or array indexing) before resorting to UDFs. The chained approach shown here represents the optimal solution for extracting elements based on position.

## Conclusion and Further Learning

Mastering efficient string manipulation techniques is a cornerstone of effective data processing in PySpark. By combining the `split()` function with dynamic array indexing calculated using `size() - 1`, we can reliably and performantly extract the last item from a variable-length string column. This structured approach ensures that the data transformation scales linearly with data volume and complexity.

To continue building expertise in PySpark data wrangling, explore the complete documentation for related functions.

**Note**: You can find the complete documentation for the PySpark split function on the official Apache Spark website.

## Related PySpark Tutorials

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

Understanding the use of `explode()` for array columns.

Techniques for handling missing values and null propagation across columns.

Advanced aggregation using window functions in PySpark SQL.