Learning to Extract the Last Element from a Split String Column in PySpark

Authored by

Mohammed looti

November 11, 2025

The Challenge of Semi-Structured Data in PySpark

PySpark, the powerful Python API for Apache Spark, is the industry standard for executing large-scale distributed data processing tasks, often within complex ETL pipelines. A frequent hurdle faced by data engineers is managing raw, semi-structured information where multiple logical data points are concatenated into a single string column. Examples include fully qualified file paths, combined names, or aggregated identifier keys separated by a consistent delimiter. To unlock the analytical potential of this data, we must first break these aggregated strings into discrete, manageable elements.

This process, known as string splitting, is foundational to ensuring data integrity and usability. Analytical efficiency hinges on the principle of data atomicity--the necessity that each cell in a DataFrame should contain only one discrete value. When a column holds multiple values separated by a space, comma, or pipe character, transformation is mandatory. Crucially, in many real-world scenarios, the objective is not just to split the string, but to isolate only a specific component, such as retrieving the file name from a full path or isolating the last name from a full employee record for grouping purposes.

The core technical difficulty arises when attempting to index the resulting array of strings. While standard Python lists support convenient negative indexing (e.g.,) to fetch the last item, PySpark DataFrames operate under a distributed model that demands explicit, column-based calculations. We cannot rely on static indexing when the length of the split array varies across rows. Therefore, retrieving the final element requires calculating the array size dynamically for each row and applying that value to access the correct index, ensuring performance and compatibility with Spark's optimized execution engine.

Leveraging Native PySpark Functions for String Manipulation

To efficiently split a column and dynamically retrieve its final element, developers must utilize the highly optimized, built-in functions available in the `pyspark.sql.functions` module. Relying on these native functions is a critical best practice in Spark development, as they are optimized by the Catalyst query optimizer, yielding significant performance advantages over custom user-defined functions (UDFs). The three primary functions essential for this operation are `split`, `col`, and `size`.

The process begins with the split function. This function accepts the target column and the specific delimiter string, transforming the original string column into a column containing an array datatype. Each element in the resulting array corresponds to a segment of the input string that was separated by the specified delimiter. Because this function is vectorized, it is executed across the entire dataset in parallel, which is fundamental to Spark's distributed efficiency. This intermediate array column is the necessary stepping stone for element extraction.

After the string is converted to an array, the challenge becomes locating the last item dynamically. Given that array lengths can differ (e.g., a path might have two segments in one row and five in the next), a fixed index is insufficient. This is where the `size` function plays its crucial role. The `size` function calculates and returns the total number of elements within the array for each corresponding row. By utilizing the output of the `size` function, we can calculate the precise index of the last element, which is always derived by subtracting one from the total size (`size(array_column) - 1`) due to zero-based indexing conventions common across computing systems, including PySpark.

The Dynamic Indexing Solution: Combining Split and Size

The most robust and efficient implementation strategy involves chaining operations using the `withColumn` method within a DataFrame transformation. While it is possible to compress the logic into a single complex expression, a two-step approach is often preferred for clarity, maintainability, and ease of debugging intermediate results, especially in complex pipelines.

The first logical step is the application of the split function, which converts the string column into an intermediate array column. The second, and most critical, step involves dynamically indexing this intermediate array. We use `withColumn` again, referencing the array column and applying the indexing logic: `col('array_column')`. This concise pattern ensures that we retrieve the final string segment, regardless of how many segments the original string contained.

The following snippet demonstrates this highly optimized methodology. Here, we create a temporary column named `new` to hold the split array before immediately overwriting that column with the extracted final element:

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 code assumes the source string is in the **employees** column and uses a space as the delimiter. By chaining these two operations, we leverage Spark's internal optimizations entirely. The repeated use of the column name `'new'` is acceptable because the second call to `withColumn` overwrites the datatype of the column, transforming it from an array of strings into a simple string containing only the desired last element. This method entirely bypasses the need for resource-intensive operations like UDFs.

Practical Demonstration: Isolating Employee Last Names

To solidify understanding, consider a common scenario: analyzing employee data where names are stored as full strings. Our objective is to reliably extract only the last name for subsequent aggregation or joining, acknowledging that some employees might have two names (First Last) while others might have three or four (First Middle Initial Last). This variability makes the dynamic indexing technique indispensable.

We begin by initializing the SparkSession, the entry point for all Spark functionality, and defining a small sample dataset. Observe the heterogeneity in the `employees` column: 'Doug Eric' has two tokens, while 'Ian John Ken Liam' has four. This variation perfectly illustrates why a static index (like or) would lead to erroneous results.

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 goal now is to apply the calculated transformation: splitting the strings in the **employees** column using the space delimiter and then extracting the final element of the resulting array for

each record. We will use a descriptive column name, `last`, for the final output, clearly differentiating it from the source data and any intermediate array structure. This concise yet powerful solution demonstrates the declarative nature of PySpark transformations.

Applying the Transformation and Validating Results

We execute the transformation logic on the sample DataFrame, `df`. In this final implementation, we ensure that the resulting column is named `last` immediately, clearly defining the output structure. The transformation remains a two-step chained process, leveraging the internal power of `split` and `size` without introducing performance overhead.

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

#create new column that contains only last item from employees column
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 output table, `df_new`, confirms the successful and accurate extraction. The new `last` column contains only the final token from the respective **employees** string. For example, the four-token name 'Ian John Ken Liam' yields 'Liam', and the three-token name 'Andy Bob Chad' yields 'Chad'. This validation confirms the key takeaway: the combination of `size` and indexing provides a highly robust solution for handling variable-length data resulting from string transformations. This adaptive capability is crucial for processing real-world data at scale, where data consistency is often imperfect.

Performance Optimization and Advanced Alternatives

The primary strength of the method demonstrated here lies in its performance profile within the Spark ecosystem. By relying exclusively on native SQL functions (`split`, `size`, `col`), the entire

operation is executed efficiently within the Java Virtual Machine (JVM) layer, managed by the Spark engine. This avoids the severe performance penalties associated with user-defined functions (UDFs), which require costly serialization and deserialization of data between Python and the JVM for every record processed.

While the `split` and `size` approach is optimal for simple, delimiter-based extraction, PySpark provides powerful alternatives for more complex pattern matching. If the extraction required matching the last token based on a sophisticated pattern--for instance, ensuring the final token is purely numeric or follows a specific filename convention--the `regexp_extract` function might be a superior choice. However, for the common task of extracting the last segment based on a simple separator, the `split` method offers the best balance of efficiency, readability, and maintainability. Mastering the application of these optimized, built-in functions is fundamental to effective large-scale data manipulation in a distributed environment.

Expanding Your PySpark Data Wrangling Toolkit

Effective data engineering using PySpark extends far beyond string manipulation. Proficiency requires mastering a comprehensive suite of data preparation techniques, including handling complex datatypes, temporal data, and various aggregation methods. The foundational knowledge of column transformations and function optimization gained from this string splitting example serves as a launchpad for more advanced tasks.

Developers should continually explore and implement the rich set of functions available in `pyspark.sql.functions` to build highly scalable and maintainable data pipelines. Key areas to focus on for continued development include:

Techniques for handling and imputing null or missing values across massive datasets.
Utilizing array functions like `explode` to transform array columns into multiple rows for one-to-many analysis.
Implementing advanced concepts such as window functions for calculating moving averages, cumulative sums, or analytical rankings.
Optimizing join operations, particularly understanding when to use broadcast joins for efficiency.

By integrating these skills, developers can ensure their data pipelines are optimized for high-volume, distributed computation, delivering reliable and fast data transformation results.