How to Explode Array Columns into Rows in PySpark

Authored by

stats writer

January 18, 2026

Introduction: Understanding Array Expansion in PySpark

When working with large-scale data processing using PySpark, it is common to encounter nested data structures, particularly within columns of a DataFrame. Arrays, which hold multiple values within a single cell, are frequently used to store related pieces of information, such as lists of historical metrics, transaction IDs, or, as in our running example, scores from multiple games. However, many subsequent data analysis or machine learning tasks require that each item within this array be treated as an individual, atomic record. This normalization process demands a powerful and efficient mechanism to flatten the structured data, converting complex column types into standard row-based data points.

The technique of converting array elements into distinct rows is formally known as "exploding" the column. PySpark provides a dedicated and highly optimized explode function within its library of SQL functions designed specifically for this purpose. This function is critical for reshaping data, making it compliant with standard relational models where each individual observation occupies a single row. Utilizing the built-in PySpark functions ensures optimal performance compared to custom solutions, as they are implemented deep within Spark's core engine, leveraging its underlying distributed computing capabilities for maximum efficiency and speed.

Mastering the `explode` operation is essential for data engineers and scientists who routinely handle complex, semi-structured datasets in the Spark environment. Without this ability, accessing or aggregating data stored in nested structures becomes convoluted and computationally expensive. This article will thoroughly detail the required syntax, provide a practical, hands-on example using basketball score data, and explore the crucial implications of applying this fundamental transformation on your PySpark DataFrame structure and schema.

The Necessity of Exploding Arrays in Data Processing

Data schemas often involve complex types to minimize storage redundancy and logically group related information. Consider a typical data warehousing scenario where customer order history is stored: rather than repeating the customer ID for every item purchased across potentially hundreds of rows, a single row might contain the customer details and an array listing all purchased products or services. While this initial structure is concise and space-efficient, it severely limits direct analytical capabilities. Standard aggregation functions (like calculating the average score across all games globally) or filtering operations cannot be easily or efficiently applied directly to the nested array elements without first breaking them out.

The primary motivation for using the explode function is to transition from a one-to-many relationship stored vertically (one row containing many values in an array) to a traditional relational structure where the one-to-many relationship is spread horizontally (many rows, each containing one value from the original array). This transformation is indispensable when preparing data for

sophisticated downstream tasks, such as feeding it into machine learning models which generally require single features per column, or for detailed reporting systems that necessitate atomic event tracking.

Furthermore, exploding arrays drastically simplifies subsequent data manipulation, including joins and filtering logic. Once the array elements are promoted to their own rows, standard SQL functions and join conditions become directly applicable. Attempting to process nested data without the built-in `explode` function would force developers to resort to complex user-defined functions (UDFs) or iterative processing methods. These alternative approaches are typically much slower and less efficient within the distributed PySpark framework, contradicting Spark's core philosophy of maximizing parallelism and minimizing expensive data shuffling.

Implementing the Core Explode Syntax

The procedure for exploding a column containing arrays in a PySpark DataFrame is exceptionally straightforward, primarily requiring the import of the specific `explode` function from the `pyspark.sql.functions` module. The core implementation relies on the `withColumn` method, a versatile tool that allows users to either create a new column based on a transformation or, as is often the case with exploding, overwrite an existing column with the results of that transformation.

To effectively flatten the array column, the explode function is applied directly to the target column. The function acts as a generator, taking the array column as input and returning a sequence of rows, one for each element in the array. By chaining this operation with `withColumn` and specifying the original array column name, we achieve the desired transformation in place, where the non-array columns (such as 'team' or 'position' in our example) are automatically duplicated to maintain context for each new exploded row.

The required syntax for exploding a column, such as one named **points**, is demonstrated below. This snippet represents the foundational command for transforming complex column types into rows suitable for granular analysis, and it is the starting point for any array flattening procedure in Spark:

from pyspark.sql.functions import explode

#explode points column into rows
df_new = df.withColumn('points', explode(df.points))

Setting Up the Demonstration Environment

To provide a tangible illustration of the practical application of the `explode` function, we will first construct a sample PySpark DataFrame. This artificial dataset simulates basketball player

performance tracking, recording essential metadata--the team affiliation, the player's position, and crucially, an array of scores achieved across three different games. Establishing a reproducible setup is a critical first step in thoroughly understanding how data transformation operates in the distributed environment of PySpark.

The setup involves several standard PySpark procedures: initializing a Spark session using `SparkSession.builder.getOrCreate()`, defining the raw data structure (which is typically represented as a list of lists in standard Python), and explicitly specifying the corresponding column names. The core challenge encapsulated in this initial structure is that the **points** column contains three distinct scores grouped together in a single unit, which inherently prevents us from calculating straightforward statistics like the overall average points per game across all players easily without first performing the necessary separation of these values.

The following code block demonstrates the complete setup required to create our initial, nested DataFrame, followed by the output showing the compact, array-based structure before any array transformation has been applied:

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|position| points|
+----+--------+------------+
A	Guard	
A	Forward	
B	Guard	
B	Forward	

+----+--------+------------+

A crucial technical observation here, often confirmed by inspecting the DataFrame schema using `df.printSchema()`, is that the **points** column is currently of a complex type, typically `ArrayType(IntegerType, True)`. This array structure confirms that multiple scores are contained within a single cell, which validates the necessity of using the explode function to separate these values into individual, measurable rows for effective analytical processing.

Step-by-Step Example: Transforming Array Data

With the necessary data setup complete, the core objective is to execute the `explode` function on our sample DataFrame (`df`). By importing the required function from `pyspark.sql.functions` and applying it via the `withColumn` expression, we precisely instruct PySpark to iterate over the elements within the **points** array for every existing row and subsequently generate a new row for each element it encounters. This is the moment the data structure is fundamentally reshaped.

During this operation, when the explode function is called, it preserves the values in all other, non-array columns (in this case, **team** and **position**) and duplicates them as needed to maintain context for every new row created during the expansion process. For example, the original row corresponding to Team A, Guard, which contained three scores , will be logically replaced by three distinct rows. Each of these new rows will be associated with the team 'A' and position 'Guard', but the **points** column will now carry only one of the scores (11, 8, or 25) individually.

The following code block executes the entire transformation and displays the resulting DataFrame, which we have named `df_new`. This output clearly demonstrates how the nested array structure has been successfully flattened, making the dataset immediately ready for standard statistical analysis and detailed reporting, achieving the goal of vertical normalization:

from pyspark.sql.functions import explode

#explode points column into rows
df_new = df.withColumn('points', explode(df.points))

#view new DataFrame
df_new.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|
| A| Guard| 8|

A	Guard	25
A	Forward	14
A	Forward	20
A	Forward	22
B	Guard	21
B	Guard	30
B	Guard	6
B	Forward	22
B	Forward	12
B	Forward	34
+----+--------+------+

Analyzing the Results of the Explode Operation

The transformation from the initial DataFrame (`df`) to the exploded DataFrame (`df_new`) manifests a profound change in the underlying data structure. The original four input rows, each representing a unique player/position combination that contained an array of scores, have now been systematically expanded into twelve resulting rows. This precise multiplicative increase is calculated because each of the four initial arrays contained exactly three elements (scores), yielding a total of 4 multiplied by 3, resulting in 12 total records in the newly structured DataFrame.

The new structure is vertically normalized, meaning it conforms to the first normal form of relational databases. Every single score now occupies its own dedicated row, paired correctly and unambiguously with its corresponding static metadata, which includes the team and position. This atomic structure is absolutely essential for calculating accurate statistics such as the mean score per position, determining the standard deviation of scores, or efficiently identifying the highest individual score recorded across all games--analytical tasks that were either cumbersome or mathematically impossible to perform using standard SQL functions before the array explosion.

It is crucial for professional data practitioners to acknowledge the inevitable memory and performance implications of this operation. Exploding arrays, particularly large ones, inherently increases the overall row count of the DataFrame, sometimes by orders of magnitude. While this normalization is necessary for analytical deep dives, users dealing with extremely large, highly nested arrays must remain acutely mindful of the resulting increase in dataset size, as it directly impacts subsequent processing time, network bandwidth usage, and memory consumption across the distributed cluster environment managed by PySpark.

Handling Nulls and Empty Arrays: Variations of Explode

While the standard `explode` function is highly robust and performs the necessary flattening

effectively, users must be keenly aware of its default behavior when encountering missing or empty data. If a cell within the array column contains a **null** value, the standard `explode` function operates under an implicit inner join logic and will entirely drop that corresponding row from the resulting DataFrame. This is also true if a cell contains an **empty array**--the row will be silently discarded from the output. This behavior is often desirable in specific analytical contexts, such as when processing scores, where a player with no recorded scores contributes no meaningful data points to the analysis.

However, if the primary goal is to preserve all rows from the original DataFrame, irrespective of whether the array field is null or empty, PySpark provides specialized alternative functions to achieve this outcome. Specifically, `posexplode` is used when the index (or position) of the array element is also a required piece of output data, and `explode_outer` or `posexplode_outer` are available to perform a left-join style array explosion. This "outer" variant retains the original non-array columns even if the array field is null or empty, filling the new exploded column with explicit null values, thus preserving the row count integrity.

Understanding these variations allows for precise control over data integrity and row preservation:

explode: Acts like an inner join on the array. It drops input rows where the array column value is null or contains an empty array.

explode_outer: Acts like a left outer join. It preserves all input rows; if the array is null or empty, the new exploded column will contain a single row with a null value.

posexplode: Similar to `explode`, but creates an additional column containing the index (position, starting from 0) of the element in the original array, which is crucial for ordered datasets.

posexplode_outer: Combines the functionality of `explode_outer` and `posexplode`, preserving all input rows while providing both the element value and its index, handling null or empty arrays gracefully.

Conclusion and Further Resources

The ability to efficiently explode arrays into individual rows is recognized as a fundamental and non-negotiable operation in advanced data preparation using PySpark. By consistently leveraging the highly optimized, built-in explode function, data practitioners are empowered to seamlessly convert complex, nested structures into a flat, vertically normalized format that is fully compatible with modern downstream analytics, reporting dashboards, and highly demanding machine learning pipelines.

The detailed, step-by-step example provided clearly illustrates how a compact dataset representing multi-game scores can be transformed and normalized into a transactional view, where every

single score constitutes a unique, measurable record. This normalization significantly enhances the efficiency and ease of querying, aggregation, and statistical inference, adhering rigorously to best practices in large-scale, distributed data modeling within the complex Spark ecosystem.

For users seeking deeper knowledge, comprehensive details, and exhaustive technical specifications regarding all array manipulation capabilities, it is highly recommended to consult the official documentation concerning PySpark's array and SQL functions. This documentation thoroughly covers all nuanced behaviors, including advanced topics like handling map types and array structures with differing nested complexities.

Note: You can find the complete documentation for the PySpark **explode** function here.

Related PySpark Tutorials

To further enhance your expertise in distributed data manipulation and advanced PySpark techniques, the following related tutorials explain how to perform other common and essential tasks:

Tutorial 1: Aggregating Data Using Window Functions in PySpark

Tutorial 2: Converting Columns to Different Data Types

Tutorial 3: Understanding and Applying User-Defined Functions (UDFs) for Custom Logic