How to Duplicate Rows in a PySpark DataFrame

Authored by

stats writer

January 3, 2026

The ability to manipulate data structures efficiently is central to large-scale data processing using Apache Spark. When working with analytical models or performance testing, data engineers often encounter scenarios where they need to deliberately increase the volume of data by replicating existing records. This technical guide explores the most effective and idiomatic method for replicating rows within a PySpark DataFrame, focusing on techniques that leverage Spark's highly optimized columnar operations.

While a simple approach might involve using the `union()` function to join a DataFrame with copies of itself--thereby causing an immediate duplication of all records--this method is generally less flexible and can be inefficient for selective or scaled replication tasks. For instance, if you merely need to double the size, `union(df, df)` suffices, but for replicating rows a specific, high number of times, or for integration into complex transformation pipelines, a specialized approach is required.

The superior method involves combining several powerful SQL functions available within the `pyspark.sql.functions` module: namely, `array_repeat` and `explode`, orchestrated through the `withColumn` operation. This methodology allows for programmatic control over the replication factor and integrates seamlessly into existing DataFrame transformations, maintaining the distributed nature of the data processing inherent to Spark. This technique is particularly valuable in machine learning workflows requiring synthetic data generation or data augmentation.

Introduction to PySpark DataFrame Replication

A PySpark DataFrame represents a distributed collection of data organized into named columns, serving as the fundamental structure for Spark SQL operations. When preparing data for sophisticated analyses, especially in areas like statistical modeling or distributed testing, it is often necessary to scale up the data volume without generating completely new synthetic data. Replication allows us to increase the dataset size by duplicating existing, high-quality records.

The primary challenge in achieving efficient row replication in a distributed environment is ensuring that the operation leverages Spark's parallel processing capabilities. Relying on iterative methods or local memory operations would completely negate the benefits of using Apache Spark. Therefore, the most robust solutions involve using Catalyst-optimized transformations that can execute the duplication logic simultaneously across all cluster nodes.

While the `union()` approach can achieve simple doubling, it is cumbersome for larger replication factors. The preferred strategy leverages the conversion of a scalar value into an array of repeated values, followed by the expansion of that array back into multiple rows, preserving the rest of the original row's attributes. This technique offers precise control over the multiplication factor and excellent performance characteristics.

Understanding the Core Replication Functions

The powerful replication method relies on three core components: the `withColumn` transformation, the `expr` function, and the combination of `array_repeat` and `explode` SQL functions. Understanding how these elements interact is crucial for implementing row multiplication correctly and efficiently.

The `array_repeat(column, N)` function takes a specified column and an integer N, generating a new column containing an array where the value from the original column is repeated N times. If we select an arbitrary existing column (e.g., 'ID' or 'name'), this new array column will hold N identical elements for every original row. This transformation does not change the number of rows, but it introduces a complex data type (an array) suitable for the next step.

Following the array generation, the `explode(array_column)` function is applied. Explosion is the critical mechanism that turns the array column into new rows. For every element within the array, `explode` generates a distinct row, carrying along the values of all other columns from the original record. Since our array contains N identical elements, `explode` effectively generates N new rows, thereby achieving the desired replication factor across the entire DataFrame.

We utilize the `expr` function to execute these complex SQL operations directly within the `withColumn` API call. `expr` allows us to pass a SQL expression string, such as `explode(array_repeat(column, N))`, enabling powerful chained functionality without the need to define temporary columns or use complex Python function calls, resulting in a cleaner and more readable transformation.

The Standard Syntax for Controlled Row Duplication

To perform controlled row replication, we define a transformation using `withColumn` that overwrites an existing column by applying the combined logic of `array_repeat` and `explode`. The key is that the column being modified is only used as a reference point for the replication factor; the resulting values in that column after the explosion are secondary to the goal of structural row multiplication.

The following syntax illustrates the concise code required to replicate each row in a PySpark DataFrame a specified number of times. In this example, we aim for a replication factor of **3**, and we arbitrarily choose the column named 'team' to host the transformation.

You can use the following syntax to replicate each row in a PySpark DataFrame a certain number of times:

from pyspark.sql.functions import expr

df_new = df.withColumn('team', expr('explode(array_repeat(team, 3))'))

This particular example replicates each row in the DataFrame **3** times. If the original DataFrame contained 100,000 rows, the resulting DataFrame, `df_new`, would contain exactly 300,000 rows. The integrity of the data within the non-modified columns is perfectly preserved across the newly generated records.

Note: We used the **team** column within the **array_repeat** function but you can use any column name that exists in the DataFrame and the result will be the same. The crucial factors are the integer value (the replication factor) and the application of the **explode** function.

Practical Demonstration: Setting up the Sample Data

The following example shows how to use this syntax in practice. Suppose we have the following PySpark DataFrame that contains information about various basketball players. We must first initialize the Spark session and define our source data.

We begin by importing the necessary `SparkSession` and defining a small list of data rows along with column names. This creates a baseline dataset of four distinct records, allowing for clear verification of the replication outcome.

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|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| West| 15| 7|

| B| West| 6| 12|
| C| East| 5| 2|
+----+----------+------+-------+

The PySpark DataFrame `df` currently has 4 total rows. Our goal is to expand this small dataset to a total of 12 rows, maintaining the exact data integrity of the original columns across the newly created replicas. This setup provides a clear visual demonstration of the multiplication effect.

Executing the Replication Logic

We now proceed with the core replication operation. We import `expr` and apply the chained functions `explode(array_repeat())` via `withColumn`. We specifically target a multiplication factor of **3**.

The transformation is executed lazily, meaning Spark constructs an optimal execution plan before running the computation. When the `show()` action is called, the plan is executed: first, a temporary array column is implicitly created containing three duplicates of the 'team' value for each row. Second, the `explode` operation expands these arrays, tripling the number of records.

We can use the following syntax to replicate each row **3** times so the resulting DataFrame will have a total of 12 rows:

from pyspark.sql.functions import expr

#replicate each row in DataFrame 3 times
df_new = df.withColumn('team', expr('explode(array_repeat(team, 3))'))

#view new DataFrame
df_new.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
A	East	11	4
A	East	11	4
A	East	11	4
A	West	15	7
A	West	15	7
A	West	15	7
B	West	6	12
B	West	6	12

```
| B| West| 6| 12|
| C| East| 5| 2|
| C| East| 5| 2|
| C| East| 5| 2|
+----+----------+------+-------+
```

Analyzing the Results and Technical Explanation

Notice that each row in the original DataFrame has been replicated **3** times. The resultant DataFrame, `df_new`, successfully contains 12 records, confirming the multiplication factor. For instance, the original record for Team 'A' in the 'East' conference is now represented by three identical rows.

The efficiency of this method stems from its functional programming approach. We used the **array_repeat** function to create an array containing the team column repeated 3 times. This step is a column-wise operation and is highly optimized. Then, we utilized the **explode** function, which is designed to convert an array or map column into multiple rows, thereby achieving the desired multiplication.

The end result is that each row is repeated three times, and this transformation is achieved using native Spark SQL functions, making it a highly performant and scalable solution suitable for processing massive datasets across distributed cluster resources. This pattern should be prioritized over non-vectorized or loop-based alternatives whenever possible in Apache Spark programming.

Summary of PySpark Data Transformation Tasks

Mastering PySpark involves familiarity with various functions and patterns beyond simple replication. Engineers frequently leverage advanced features such as window functions, complex joins, and UDFs (User Defined Functions) to handle intricate data transformations. Replication, as demonstrated, falls under the category of structural transformation, preparing data for downstream analytical or modeling tasks where controlled data volume is essential.

The following is a list of common structural and analytical tasks frequently encountered when utilizing PySpark for large-scale data processing:

Filtering DataFrames based on complex criteria using boolean logic.
Performing efficient joins between multiple DataFrames using different join types (inner, outer, left anti).
Implementing custom aggregations using Spark SQL to summarize data.

Handling schema evolution and data type conversions efficiently.

By integrating robust methods like `explode(array_repeat())` into your toolkit, you ensure that your data engineering workflows are both efficient and scalable, fully utilizing the distributed computation capabilities provided by the Spark ecosystem.