How to Duplicate a Column in a PySpark DataFrame

Authored by

stats writer

January 1, 2026

Working with large datasets often requires sophisticated data manipulation techniques, and in the world of big data, the PySpark DataFrame is the foundational structure for data processing. A common requirement during the feature engineering or data preparation phase is the need to create a copy, or duplicate, of an existing column. This duplication is often necessary for tasks such as creating backup fields before applying destructive transformations, preparing inputs for machine learning models that require features derived from the same source but processed differently, or simply for validation and comparison purposes within analytical workflows.

The primary and most efficient tool for achieving this duplication within PySpark is the built-in withColumn() function. This highly versatile method is specifically designed for adding new columns or replacing existing ones based on calculations or expressions derived from other columns in the DataFrame. When leveraged for duplication, it allows us to map the values and data type of an existing column directly onto a newly named column instantaneously, utilizing the underlying parallel processing capabilities of the Spark engine.

The efficiency of using withColumn() stems from its native integration with the underlying Apache Spark architecture, ensuring that these operations are performed in a highly distributed and optimized manner across the cluster. This method accepts two key arguments: the first defines the symbolic name of the output column, and the second defines the expression that dictates its content, which, in the case of duplication, is a direct reference to the source column. This streamlined approach makes it the standard, high-performance method for column manipulation in distributed environments.

Dissecting the Mechanism of the withColumn() Function

The withColumn() function operates fundamentally on the principle of immutability, which is a core tenet of PySpark architecture. When this function is invoked, it does not modify the existing DataFrame object in place. Instead, it returns an entirely new DataFrame instance that incorporates the specified column addition or modification. This design choice is vital for ensuring Spark's robust fault tolerance, enabling complex directed acyclic graphs (DAGs) of transformations, and guaranteeing data consistency across distributed operations.

When used for duplicating data, **withColumn()** requires that the expression provided--the second argument--is simply a direct call to the existing column data. For instance, if you are duplicating a column named **'Revenue'**, the expression would typically be **df**. Spark's Catalyst Optimizer interprets this instruction as a requirement to map the exact contents of the source column, including its data type and values, into the newly named target column.

It is crucial to understand the dual nature of **withColumn()**: it acts as a creator if the column name specified in the first argument does not exist in the DataFrame's schema, and it acts as an

updater/replacer if the column name already exists. Therefore, successful duplication, which requires retaining both the original and the copied data, relies entirely on providing a unique, novel name for the first argument. If the names conflict, PySpark will perform an overwrite operation, which prevents the creation of a true parallel duplicate.

Implementing the Basic Syntax for PySpark Duplication

The implementation of column duplication is straightforward, adhering to the clean, functional API design of PySpark. This process involves chaining the **withColumn()** method onto the DataFrame object and specifying the two required parameters. This single-line operation encapsulates the complex distributed computation necessary to copy data across potentially thousands of partitions.

The syntax below provides the template necessary for any duplication task. The placeholders clearly indicate the two necessary components: the new, unique identifier for the duplicated field and the reference pointing back to the data source column. It is often recommended to use the standard bracket notation (**df**) or the **col()** function from **pyspark.sql.functions** to correctly reference the column object within the expression argument.

You can use the following basic syntax to create a duplicate column in a PySpark DataFrame:

df_new = df.withColumn('my_duplicate_column', df)

This syntax results in the generation of a new DataFrame, **df_new**, which structurally includes all the columns from the original **df**, plus the new **my_duplicate_column** containing data mirrored from **original_column**. This is the cornerstone technique for efficient data manipulation when copies are required.

Establishing the Context: Initial DataFrame Creation

The following example shows how to use this syntax in practice. Before demonstrating the duplication, we must first establish a sample dataset. We will create a PySpark DataFrame containing performance statistics for several fictional basketball players. This example dataset is manageable yet sufficiently detailed to illustrate the transformation process clearly.

The initial setup involves invoking the **SparkSession.builder.getOrCreate()** method, which initializes or retrieves the necessary Spark execution environment. Defining the data array and the corresponding column names separately ensures that the schema is correctly applied when **spark.createDataFrame()** is called. Our dataset includes attributes such as **team**, **position**, **points**, and **assists**, providing a mix of categorical and numerical data types.

We specifically target the numerical **points** column for duplication, as numerical features are most

commonly copied prior to scaling, imputation, or complex mathematical transformations required for subsequent machine learning model training. Displaying the original DataFrame (**df.show()**) verifies that our baseline data is correctly loaded and ready for the duplication operation.

Example: Initializing the Player Data DataFrame

The code block below outlines the complete setup sequence, from importing the necessary Spark component to the final display of the source DataFrame. This standardized approach ensures that all subsequent operations are performed within a stable and defined Spark context.

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|assists|
+----+--------+------+-------+
A	Guard	11	5
A	Guard	8	4
A	Forward	22	3
A	Forward	22	6

B	Guard	14	3
B	Guard	14	5
B	Forward	13	7
B	Forward	14	8
C	Forward	23	2
C	Guard	30	5
+----+--------+------+-------+

The resulting table, displayed above, confirms the schema and content of our initial DataFrame, **df**. This four-column structure is the starting point for our manipulation, clearly showing the **points** column that we intend to duplicate.

Performing the Successful Column Duplication

With the source DataFrame established, we proceed with the core operation: creating a duplicate column named **points_duplicate**. This action is executed via the withColumn() function, ensuring that the result, stored in **df_new**, is a complete copy of the original DataFrame plus the appended field.

The critical step here is the second argument: **df**. By referencing the existing column object, we instruct PySpark to calculate the values of the new column based on the exact values in the source column. Since **'points_duplicate'** is a new column name, the function correctly interprets the command as an insertion operation, expanding the DataFrame schema.

The output of **df_new.show()** provides immediate visual confirmation of the successful duplication. We can observe the original four columns, followed by the newly added **points_duplicate** column. A row-by-row comparison confirms that the data values are perfectly mirrored, demonstrating the effectiveness and simplicity of the **withColumn()** approach for creating exact copies.

We can use the following code to create a duplicate of the **points** column and name it **points_duplicate**:

```
#create duplicate of 'points' column
df_new = df.withColumn('points_duplicate', df)

#view new DataFrame
df_new.show()

+----+--------+------+-------+----------------+
|team|position|points|assists|points_duplicate|
+----+--------+------+-------+----------------+
```

A	Guard	11	5	11
A	Guard	8	4	8
A	Forward	22	3	22
A	Forward	22	6	22
B	Guard	14	3	14
B	Guard	14	5	14
B	Forward	13	7	13
B	Forward	14	8	14
C	Forward	23	2	23
C	Guard	30	5	30
+----+--------+------+-------+----------------+

Notice that the **points_duplicate** column contains the exact same values as the **points** column, successfully achieving the data replication.

The Importance of Unique Identifiers and Overwriting

The requirement that the duplicate column must possess a name different from the original is absolute. If a user provides an existing column name as the first argument to **withColumn()**, the operation changes from creating a new field to modifying the existing field's definition. Even if the expression provided is a self-reference (i.e., copying the column onto itself), PySpark will not introduce a new column to the schema.

In data engineering, this functionality is usually desirable when one intends to apply a transformation and overwrite the original data (e.g., converting a column from Celsius to Fahrenheit, storing the result back in the original column name). However, when the goal is duplication--the simultaneous existence of both the raw and the derived data--using the same name results in a silent failure to duplicate, as the DataFrame's structure remains unaltered.

This behavior is a crucial characteristic of the functional programming paradigm underlying Spark. Since the transformation simply redefines the column using an identical source expression, the logical plan does not result in a structural change to the DataFrame. Therefore, strict adherence to unique naming conventions is necessary to guarantee successful column addition and achieve true duplication.

Demonstrating the Structural Failure of Non-Unique Naming

To highlight the consequences of reusing a column name, we execute the operation again, this time intentionally setting the new column name to **'points'**, which already exists. This will demonstrate how **withColumn()** handles updates versus additions when the expression is self-

referential.

When the following code runs, the resulting DataFrame, **df_new**, is structurally identical to the original **df**. The column count is still four, and no **points_duplicate** field appears. This confirms that the operation was interpreted as an attempt to overwrite the **points** column with itself, yielding no observable structural change, and failing to create the desired parallel copy.

This outcome underscores the distinction between duplication and modification in PySpark. When performing data manipulation tasks, always ensure the new field is uniquely named unless the explicit goal is to replace the values of an existing field. This clarity prevents confusing results and maintains the integrity of the data pipeline.

For example, if we attempt to use the following code to create a duplicate column using the existing name, it won't work:

```
#attempt to create duplicate points column
df_new = df.withColumn('points', df)

#view new DataFrame
df_new.show()

+----+--------+------+-------+
|team|position|points|assists|
+----+--------+------+-------+
| A| Guard| 11| 5|
| A| Guard| 8| 4|
| A| Forward| 22| 3|
| A| Forward| 22| 6|
| B| Guard| 14| 3|
| B| Guard| 14| 5|
| B| Forward| 13| 7|
| B| Forward| 14| 8|
| C| Forward| 23| 2|
| C| Guard| 30| 5|
+----+--------+------+-------+
```

No duplicate column was created; the DataFrame retains its original four-column structure.

Conclusion: Mastering Column Duplication in PySpark

The creation of a duplicate column in a PySpark DataFrame is a foundational task efficiently

handled by the **withColumn()** function. By understanding its dual role--adding a new column if the name is unique, or replacing an existing one otherwise--users can confidently manage their data transformations at scale. The key to successful duplication is pairing the **withColumn()** method with a clear reference to the source column and, most importantly, a distinct name for the new field.

This technique is essential for maintaining robust data pipelines, allowing data scientists to isolate raw data from processed features. Whether preparing data for complex aggregations or feature selection, the ability to quickly and efficiently duplicate columns without incurring significant memory or computational overhead is a testament to the power of the Spark framework.

For those interested in exploring further capabilities of PySpark transformations, the official documentation for the **withColumn** function offers comprehensive details on advanced usage, including conditional logic, type casting, and integration with other Spark SQL expressions.

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

How to filter rows based on multiple conditions.

Performing joins across multiple DataFrames efficiently.

Applying User Defined Functions (UDFs) for custom transformations.