How to Duplicate a Column in a PySpark DataFrame

Authored by

stats writer

February 6, 2026

In advanced PySpark SQL operations, the need often arises to create an exact copy of an existing column within a PySpark DataFrame. This fundamental task of data manipulation is easily achieved using the powerful `withColumn()` transformation. By invoking this function, users can specify a new column name and reference the existing column they wish to clone. The resulting output is a DataFrame containing the original column alongside a new column that holds identical values. This technique is invaluable for several key scenarios, such as preparing data for complex transformations where intermediate results are required, ensuring the preservation of the original dataset during intensive computation, or simply setting up standardized columns for subsequent feature engineering processes without risking the integrity of source data.

Creating a Duplicate Column Efficiently in a PySpark DataFrame

Understanding the `withColumn()` Transformation

The core mechanism for generating a duplicate column in PySpark DataFrames relies entirely on the `withColumn()` method. This method is a crucial component of the DataFrame API, specifically designed to add a new column or replace an existing column in a non-mutating fashion. Since DataFrames in PySpark are immutable, this function returns a completely new DataFrame object, ensuring that the original structure remains untouched. When utilizing `withColumn()` for duplication, we essentially instruct PySpark to create a new column (the first argument) and populate it with the values derived directly from an existing column (the second argument, typically a Column object).

This approach offers significant advantages in large-scale distributed data processing environments. Because Spark utilizes lazy evaluation, defining a new column this way is merely the addition of a step to the logical plan until an action is triggered. By referencing an existing column object as the expression for the new column, we achieve perfect replication of the data, metadata (including the data type and nullability properties), without incurring the cost of deep copying or complex data re-evaluation. Understanding how `withColumn()` handles column references is therefore essential for optimized data manipulation.

Fundamental Syntax for Column Duplication

To successfully generate a duplicate column, the required syntax is both straightforward and highly expressive. It necessitates calling the `withColumn()` method directly on the existing DataFrame, providing the chosen name for the new duplicate column as a string, and finally, specifying the original column object that acts as the source data. This explicit referencing mechanism guarantees clarity regarding the source data being copied, which is vital for maintaining robust and

reproducible codebases across various data pipelines.

The following structure demonstrates the foundational syntax used across virtually all PySpark implementations for achieving column duplication:

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

In the above example, `df` represents the source DataFrame, while `'my_duplicate_column'` is the mandatory string literal defining the name of the new column to be inserted into the schema. Crucially, `df` is the column object reference that instructs Spark to copy the values verbatim. It is important to remember that the output, `df_new`, is a distinct DataFrame object containing the augmented structure.

Detailed Example: Setting Up the Sample DataFrame

To comprehensively illustrate the practical application of this duplication technique, we must first establish a sample PySpark DataFrame. We will use a dataset containing hypothetical statistics for basketball players, including identifiers for their team and position, alongside performance metrics such as points scored and assists made. This dataset provides a clear, numerical context for demonstrating the mechanics of column replication.

The following code snippet handles the necessary initial setup, beginning with the crucial task of importing and initializing the SparkSession--the primary gateway to all features within the Apache Spark ecosystem--and subsequently defining the structure and population of our sample DataFrame:

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 DataFrame, `df`, displayed above, consists of ten rows and four descriptive columns. For the core demonstration of duplication, our attention will be focused on the numerical `points` column, showing how easily its data can be replicated for use in subsequent transformations without corrupting the source figures.

Executing the Column Duplication Process

With the initial sample data established, we can now proceed to utilize the `withColumn()` method to achieve the desired column duplication. The objective is clearly defined: duplicate the existing values from the `points` column and assign them to a new, distinctly named column called `points_duplicate`. This naming strategy is intentional, designed to maximize clarity and avoid any potential ambiguity in the DataFrame schema.

The following detailed code block executes the duplication operation. Following the transformation, we immediately invoke the `show()` action, forcing Spark to compute and display the newly generated DataFrame, thereby confirming the success of the applied transformation and the

resulting data structure:

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

As clearly evidenced by the output, the resulting DataFrame, `df_new`, now possesses five columns, an increment of one over the source DataFrame. The newly appended column, `points_duplicate`, contains values that are an exact, row-by-row match of those in the original `points` column. This successful transformation validates the use of `withColumn()` as the primary mechanism for direct and reliable column duplication in PySpark SQL environments.

Understanding the Immutability of DataFrames

The successful creation of the `points_duplicate` column highlights a core principle of Apache Spark: immutability. When we applied the `withColumn()` transformation, the system did not modify the original DataFrame object, `df`, in memory. Instead, it calculated and returned a completely new DataFrame, `df_new`, which incorporates the newly defined column. This guaranteed immutability is essential for achieving fault tolerance and consistent results in large-scale distributed computing.

Furthermore, when duplicating a column using its object reference, the newly created column inherits all critical metadata, including the original column's precise data type and handling characteristics for null values. This metadata inheritance ensures that the duplicate is truly

functional and ready for analysis immediately, eliminating the need for explicit type casting or verification before proceeding with complex calculations or aggregations.

The Critical Constraint of Column Naming

A non-negotiable requirement when employing `withColumn()` to introduce new data elements is that the name provided for the new column must be unique within the context of the target DataFrame's schema. If a user attempts to reuse the name of an existing column, the function does not generate a duplicate column appended to the end; rather, it executes an update operation. Specifically, it attempts to overwrite or redefine the existing column using the expression provided in the second argument.

In the specific case of duplication, where the expression provided is merely the column itself (e.g., attempting to redefine `points` using `df`), this results in a silent failure to create a new column. Since the existing column is being redefined by its own data, no visible change occurs in the data or the schema, and the intended duplication objective is not met. Developers must strictly adhere to using unique identifiers to expand the DataFrame.

The following example demonstrates the outcome when the naming constraint is violated by attempting to reuse the original column name, `points`, as the target name for the new column:

#attempt to create duplicate points column using the same name
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|
+----+--------+------+-------+
```

The output above confirms that the DataFrame structure remains identical to the original; specifically, no fifth column was added to the schema. This unequivocally illustrates that `withColumn()` prioritizes replacement over addition when a name conflict occurs, reinforcing the need for distinct naming when the goal is schema expansion.

Practical Applications and Advanced Use Cases

The ability to duplicate columns is a cornerstone of preparatory steps in data science and engineering workflows using PySpark SQL. One of the most common requirements is creating a safety baseline for feature engineering. By duplicating a column like `points` to `points_processed`, analysts can apply aggressive transformations such as standardization, normalization, or non-linear scaling to the duplicate, while reliably retaining the raw, original values for auditing, logging, or comparison purposes.

A more advanced application involves sophisticated window functions. Often, a dataset requires applying two or more distinct window specifications (e.g., one calculation partitioned by `team` and another calculated over the entire dataset) that reference the same source data. Duplicating the source column allows these separate, potentially conflicting calculations to proceed without ambiguity. Furthermore, column duplication is a quick method for creating intermediate backups of critical data before performing operations that might introduce unexpected data loss or null values, thereby acting as an efficient form of data safety mechanism.

Further PySpark Data Manipulation Tutorials

The capability to efficiently duplicate columns using the `withColumn()` transformation is an essential foundational skill in PySpark. Mastering this and similar DataFrame operations empowers developers to manipulate and manage large-scale data structures effectively, preparing them for robust and scalable analysis. For those seeking deeper knowledge, the official documentation for the PySpark `withColumn` function provides comprehensive details on its capabilities beyond simple duplication, including handling complex expressions and incorporating conditional logic (like `when().otherwise()`).

To continue building proficiency in PySpark, explore related tutorials that cover other common data manipulation tasks, many of which rely on foundational DataFrame transformations similar to `withColumn()`:

Tutorial 1: How to Rename a Column in PySpark
Tutorial 2: Dropping Multiple Columns in a PySpark DataFrame
Tutorial 3: Using Window Functions for Advanced Aggregations