Learning PySpark: How to Duplicate a Column in a DataFrame

Authored by

Mohammed looti

November 11, 2025

Introduction to Data Manipulation in PySpark

In the realm of big data processing and analysis, PySpark serves as the essential Python API for Apache Spark, offering powerful, distributed tools for handling massive datasets. A fundamental operation in data preparation, especially during ETL (Extract, Transform, Load) processes and feature engineering, is the ability to efficiently manipulate columns within a DataFrame. Duplicating an existing column is a common requirement; this might be necessary if you plan to apply transformations, such as normalization or type casting, but wish to preserve the original feature set for comparison, auditing, or subsequent operations.

The core philosophy of PySpark DataFrames emphasizes immutability, meaning that transformations like column duplication do not modify the original DataFrame in place. Instead, these operations return a new DataFrame containing the results of the transformation. Understanding this non-destructive approach is crucial for writing efficient and predictable data pipelines. The primary method utilized for adding or modifying columns in PySpark is the **withColumn** transformation, which provides a declarative way to define the logic for the new column based on existing data.

To create a precise, identical copy of an existing column, you leverage the **withColumn** method by passing the desired unique name for the new column and referencing the contents of the original column as the definition expression. This technique is straightforward, highly optimized, and ensures that even large-scale datasets can be processed quickly and reliably within the distributed Spark environment. The following command structure represents the most basic and standard approach for achieving column duplication in your PySpark DataFrame.

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

Understanding the `withColumn` Transformation

The `withColumn` function is one of the foundational tools for column-level operations in PySpark SQL. It is designed with dual functionality: it can either introduce a new column to a DataFrame if the specified name is unique, or it can replace (overwrite) an existing column if the name matches a column already present in the schema. When our intention is pure duplication, we utilize the 'add a new column' capability by ensuring the provided column name is distinct.

The structure of the **withColumn** function requires two critical arguments. The first argument is a string specifying the name of the column you wish to create (the duplicate). The second argument is a column expression that defines the content for that new column. In the context of duplication, this expression is simply a direct reference to the original column. For example, referencing a column named 'sales' would use the expression `df`. This simple assignment instructs Spark to

copy the values from the source column into the destination column across all partitions of the data.

This transformation method is invaluable for preserving data integrity while preparing features. Duplication serves as an essential preliminary step before applying more complex column manipulations, such as performing arithmetic calculations, implementing conditional logic (using `when` and `otherwise` clauses), or applying custom business rules via User Defined Functions (UDFs). By first creating a duplicate, the original source feature remains pristine and available, acting as an immutable reference point for validation, auditing, or comparison against the transformed feature.

Setting up the PySpark Environment and Sample Data

To effectively demonstrate column duplication, we must first establish a functional Spark environment and define a representative sample dataset. All operations in PySpark require an active SparkSession, which acts as the primary gateway to access Spark functionality. We begin by initializing this session and then defining a small dataset. For this tutorial, we will use a DataFrame containing hypothetical statistics about basketball players, including their team, position, total points scored, and assists provided.

The dataset is structured as a list of lists, where each inner list represents a row of data. We then define a corresponding list of column names, which provides the schema. This method of using `spark.createDataFrame(data, columns)` is standard practice for creating small, in-memory DataFrames used for prototyping and demonstration purposes. After creation, we immediately display the DataFrame using the `df.show()` action to confirm its schema and contents, ensuring that the starting point for our transformation is correctly established.

The complete setup code, including the necessary imports and the definition of the initial DataFrame, is provided in the block below. Pay close attention to the structure of the data, as we will target the 'points' column for our duplication exercise in the subsequent steps.

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

Implementing Column Duplication: A Step-by-Step Guide

Now that our initial DataFrame is ready, the next step is to execute the duplication command. We will specifically focus on duplicating the **points** column. This duplication might be necessary if, for example, we plan to calculate a derived metric, such as a rolling point average, but we need the original, raw point totals to remain intact for later aggregation or reporting purposes. By naming the duplicate column **points_duplicate**, we clearly distinguish the new feature while ensuring the original data remains unaltered.

The implementation is executed by applying the `withColumn` transformation to our existing DataFrame, `df`. The result is stored in a new object, `df_new`, which inherits all columns from `df`

and includes the newly copied column. It is vital to remember that this operation is a transformation; it updates the execution plan in Spark's Catalyst Optimizer but does not immediately compute the data. The computation is triggered only when an action, such as `show()` or `collect()`, is called.

Upon reviewing the output of the transformed DataFrame, you will notice that the schema has expanded from four columns to five. The newly added column, **points_duplicate**, contains an exact, element-by-element match of the values found in the source **points** column. This confirms the successful and efficient completion of the duplication process, providing a robust, duplicated column ready for further analysis or manipulation within the distributed PySpark framework.

#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
+----+--------+------+-------+----------------+

The output clearly demonstrates that the **points_duplicate** column holds values identical to those in the original **points** column. This validates that the transformation executed correctly, and we now have a safe, secondary column to use for complex data wrangling tasks without jeopardizing the integrity of the source data.

Addressing Column Naming Constraints and Overwriting Issues

A fundamental concept to grasp when using the `withColumn` transformation relates to how

PySpark handles naming conflicts. The function's design dictates that if the name provided for the new column already exists within the DataFrame's schema, PySpark will interpret the command not as a request for duplication, but as a request to overwrite or update the existing column with the contents of the expression.

Consequently, a stringent requirement for successfully creating a duplicate column is that the new column must be assigned a name that is unique within the schema. If you attempt to use the exact same column name for both the target column and the source column expression, PySpark will simply replace the existing column with its own values. This results in no structural change to the DataFrame, and crucially, no new column is generated. Understanding this behavioral constraint is vital to avoid logical errors in data pipelines and is a common pitfall for those new to PySpark.

To illustrate this point, let us intentionally attempt to duplicate the 'points' column while specifying 'points' as the name of the new column. As demonstrated in the code block below, the command executes without raising an error, but the resulting DataFrame, `df_new`, remains structurally identical to the original `df`, containing only four columns. This outcome confirms that the operation did not result in duplication but rather a self-overwrite, which is computationally redundant for simple duplication tasks.

#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 clearly shows that no fifth column was created. This solidifies the rule: for successful column duplication in PySpark, the name provided to the `withColumn` function must be unique relative to the existing columns in the target DataFrame.

Use Cases and Practical Applications of Duplicated Columns

The ability to quickly and reliably duplicate a column is more than just a convenience; it is a foundational prerequisite for several advanced data engineering and machine learning workflows. One prevalent application is comparative feature engineering. For instance, if you are working with a continuous feature like 'income', you might duplicate it into 'income_standardized' and 'income_robust_scaled'. By maintaining both scaled versions derived from the same original source, data scientists can directly evaluate which scaling technique yields better performance when training various machine learning models, all while preserving the original 'income' column for interpretability and reporting.

Another critical use case is centered around data governance, debugging, and audit trails. Complex data pipelines often involve multiple stages of cleaning, imputation, and aggregation, which can significantly alter the values of a column. In such scenarios, maintaining an original copy of a critical feature is essential for data provenance. If data integrity issues or anomalies arise downstream, having the original column readily available prevents the need to rerun the entire pipeline, offering an immediate point of comparison and vastly improving the efficiency of quality assurance and error tracing. This practice of preserving source features is highly recommended in regulated environments.

Furthermore, duplication streamlines scenarios that require applying multiple, distinct window functions or aggregations to the same feature. Consider needing to calculate both a running total of 'sales' across the dataset and a daily average of 'sales' grouped by 'region'. Instead of relying on complex subqueries or temporary views that can often complicate the execution plan, duplicating the 'sales' column allows one copy to be dedicated to the window function calculation and the other to a simple aggregation. This isolation simplifies the code logic and ensures that operations remain clear and optimized within the Spark distributed environment.

Conclusion and Further Learning

Mastering fundamental column manipulation techniques, such as duplication, is essential for any professional working extensively with PySpark DataFrames. The **withColumn** function provides a versatile and highly efficient mechanism for both deriving new features and creating exact copies of existing columns. By strictly adhering to the constraint of providing a unique name for the new column, developers can successfully execute duplication, thereby enabling safe feature engineering practices and robust data integrity throughout their data processing lifecycle.

The techniques demonstrated here form the bedrock of columnar data manipulation. For those aspiring to deepen their expertise in Spark transformations, it is highly beneficial to explore the full potential of the `withColumn` function. This includes combining it with PySpark SQL functions, implementing sophisticated conditional logic, and integrating User Defined Functions (UDFs) for executing highly customized, non-standard data transformations.

The following resources provide complete documentation and further guidance on related PySpark tasks:

The complete documentation for the PySpark **withColumn** function can be found on the official PySpark API documentation site.

Additional Resources

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