How to Add a Column from Another DataFrame in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 9, 2026

Welcome to this detailed guide on merging data from two separate DataFrame objects within PySpark. When the need arises to consolidate disparate datasets, adding a column from one DataFrame into another is a common operation. The most straightforward approach involves using the built-in join function, which merges the two data structures based on a common, unique identifier column. This process generates an entirely new DataFrame that incorporates the desired column.

Alternatively, the withColumn function can sometimes be leveraged to create a new column, particularly when the relationship between the two DataFrames is already established or can be mapped using complex expressions. Regardless of the method chosen, successfully executing these operations demands a strong grasp of PySpark's core DataFrame mechanics and syntax, especially concerning distributed operations.

PySpark: Add Column from Another DataFrame

The Strategy: Joining DataFrames Without a Natural Key

When two DataFrames need to be merged but lack a natural, common key (like a customer ID or primary index), a common and robust technique is to generate a temporary, synthetic key using row numbers. This key allows us to perform an inner join, effectively aligning the rows based on their relative position in the original, sorted DataFrames. The following syntax outlines the necessary steps to create this temporary 'id' column using the row_number function within a Window specification.

```
from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

# Define a window specification. We order by a constant 'A' to treat the entire DataFrame as one partition,
# ensuring sequential row numbering from 1 to N.
w = Window().orderBy(lit('A'))

# Add a temporary column 'id' to each DataFrame containing sequential row numbers.
df1 = df1.withColumn('id', row_number().over(w))
df2 = df2.withColumn('id', row_number().over(w))

# Join the two DataFrames using the newly created 'id' column as the join key.
final_df = df1.join(df2, on=).drop('id')
```

Understanding this methodology is crucial. By applying the row_number function over a non-

partitioned Window specification, we assign a unique, ordered index to every row in both DataFrames. This index then serves as the common denominator for the join function, allowing us to accurately combine the data based on positional alignment before finally dropping the temporary 'id' column to clean up the result.

Practical Example: Merging Basketball Data

To illustrate the effectiveness of this index-based join function, we will walk through a concrete example. Suppose we have two separate DataFrames: one containing team names and another containing scoring data, but neither DataFrame contains a linking key. Our goal is to merge the 'points' column into the 'team' DataFrame, assuming that the rows are already correctly aligned based on their implicit order.

We begin by defining and displaying our first DataFrame, named df1, which holds a single column of basketball team identifiers. We must first establish a SparkSession to enable DataFrame creation and manipulation, a prerequisite for any PySpark job.

Defining the First DataFrame: Team Names (df1)

The following Python code initializes the Spark environment and creates df1, demonstrating the structure of our team name data. This DataFrame is simple, containing only the team names as strings, ready to be augmented with scoring metrics.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data1 = ,
,
,
,
,
]

#define column name
columns =

#create dataframe using data and column name
df1 = spark.createDataFrame(data1, columns)

#view dataframe
```

df1.show()

+-------+
| team|
+-------+
| Mavs|
| Nets|
| Nets|
|Blazers|
| Heat|
| Heat|
|Thunder|
+-------+

Defining the Second DataFrame: Point Values (df2)

Next, we define our second DataFrame, df2, which contains the corresponding point values. It is imperative that the order of the values in df2 aligns perfectly with the order of the teams in df1 for the upcoming join operation to be meaningful and accurate.

#define data
data2 = ,
,
,
,
,
]

#define column name
columns2 =

#create dataframe using data and column name
df2 = spark.createDataFrame(data2, columns2)

#view dataframe
df2.show()

+------+
|points|
+------+

| 22|
| 25|
| 41|
| 17|
| 32|
| 50|
| 18|
+------+

Executing the Merge Operation and Viewing the Result

With both source DataFrames defined, we can now apply the previously introduced indexing and joining logic. We import the necessary functions--row_number and Window--to create the temporary 'id' key. Once this key is established in both DataFrames, the final join function merges the rows based on sequential numbering, successfully appending the 'points' column from df2 to df1.

from pyspark.sql.functions import row_number,lit
from pyspark.sql.window import Window

# Add temporary 'id' column using row numbers for alignment
w = Window().orderBy(lit('A'))
df1 = df1.withColumn('id', row_number().over(w))
df2 = df2.withColumn('id', row_number().over(w))

# Join DataFrames and drop the temporary key
final_df = df1.join(df2, on=).drop('id')

# View the final merged DataFrame
final_df.show()

+-------+------+
| team|points|
+-------+------+
Mavs	22
Nets	25
Nets	41
Blazers	17
Heat	32
Heat	50

|Thunder| 18|
+-------+------+

As demonstrated by the output of the final DataFrame, the points column from the second DataFrame (df2) has been successfully and accurately appended to the initial DataFrame (df1) using this powerful indexing strategy.

Conclusion and Further PySpark Operations

The method of generating sequential row numbers and utilizing a Window specification for merging DataFrames is a vital technique in PySpark when standard key-based joins are not feasible. This ensures data integrity by matching rows positionally. Mastering the join function in combination with window functions opens up numerous possibilities for complex data manipulation in distributed computing environments.

For those looking to expand their knowledge, the following tutorials explain how to perform other common data manipulation tasks in PySpark:

PySpark: How to Add New Column with Constant Value