PySpark: Add Column from Another DataFrame

Authored by

Mohammed looti

November 11, 2025

The Challenge of Adding Columns by Position in PySpark

As data professionals frequently working with large datasets, we often encounter scenarios where we need to combine columns from two separate DataFrame structures. While this task is straightforward in single-machine environments like Pandas, merging columns strictly by position in a distributed system like PySpark requires a specific strategy. This is because PySpark DataFrames are inherently unordered collections; simply attempting to append one column to another without a common key or index can lead to unpredictable results due to the distributed nature of the data processing.

To reliably transfer a column from a source DataFrame to a target DataFrame based on row sequence, we must first impose a temporary, consistent ordering on both datasets. This technique involves generating a unique, sequential row number for every record in both DataFrames. Once both DataFrames share a common index--an artificial primary key--we can then execute a standard Join operation to align the columns correctly row-by-row.

The core syntax for achieving this sequential alignment and subsequent column addition relies heavily on Spark's powerful analytic functions, specifically the Window function and the row_number() function. The following block illustrates the necessary imports and the logic required to synchronize the two DataFrames using a temporary identifier column named 'id'.

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

#add column to each DataFrame called 'id' that contains row numbers from 1 to n
w = Window().orderBy(lit('A'))
df1 = df1.withColumn('id', row_number().over(w))
df2 = df2.withColumn('id', row_number().over(w))

#join together both DataFrames using 'id' column
final_df = df1.join(df2, on=).drop('id')

Understanding the Core PySpark Technique: Using Row Numbers for Alignment

The technique demonstrated above, while powerful, requires careful consideration of its components. The fundamental goal is to ensure that the Nth row of the source DataFrame (df2) is correctly paired with the Nth row of the target DataFrame (df1). We leverage the concept of a Window function to achieve this, as it allows us to perform calculations across a set of DataFrame rows that are related to the current row, such as assigning sequential numbers.

In the provided code snippet, the creation of the window specification, `w = Window().orderBy(lit('A'))`, is critical yet subtle. By leaving the partition specification empty (`Window()`) and providing a constant literal value ('A') for the `orderBy` clause, we effectively force the entire DataFrame into a single, temporary partition and impose an arbitrary, but consistent, order across all rows. This setup allows the row_number() function to assign a unique, sequential integer ID (from 1 to N) to every record within both `df1` and `df2`.

Once the temporary 'id' column is successfully added to both DataFrames via the `withColumn` operation, the stage is set for the actual merging process. This index acts as the common key, guaranteeing that the rows align perfectly during the subsequent join. This approach is essential in scenarios where the two DataFrames are known to be in the correct corresponding order but lack a natural key to link them, effectively simulating an ordered zip operation in a distributed environment.

Step-by-Step Implementation: Generating Test DataFrames

To illustrate this powerful column addition method, we will define two distinct PySpark DataFrames. The first DataFrame, `df1`, will serve as our primary dataset, containing categorical data--in this case, basketball team names. The second DataFrame, `df2`, will hold the numerical data--the corresponding points scored--that we wish to append to `df1`. It is crucial to remember that for this positional join technique to work correctly, the number of rows and the desired sequence of records in both `df1` and `df2` must match perfectly.

First, we initiate a Spark session and construct `df1`, which contains a single column named 'team'. Notice how the team names are listed sequentially, representing our initial dataset structure.

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

#define data
data1 = ,
,
,
,
,
,
]

#define column name
columns1 =

#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 Source Data for the New Column (df2)

Following the creation of `df1`, we define `df2`, which holds the numerical values we intend to merge. This DataFrame contains the 'points' column, where each value corresponds positionally to a team in `df1`. The data definition process for `df2` follows the exact same pattern as `df1`, guaranteeing that both DataFrames start with the same number of rows (seven in this example). This careful setup is fundamental to ensuring a successful join based on sequential indexing.

The 'points' data represents the column that will be appended to the right side of the existing `df1` structure. By keeping the creation and inspection processes clean and separated, we confirm the initial state of our source column before initiating the complex PySpark operations needed to create the join keys.

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

With both DataFrames initialized and confirmed to be structurally compatible (same row count), we can proceed to the critical merging step. This involves applying the windowing logic defined earlier to generate the necessary 'id' column in preparation for the Join operation.

Executing the Indexed Join Operation in PySpark

The final phase involves applying the row-number generation and performing the join. This sequence of operations effectively transforms the two independent DataFrames into a single, unified DataFrame where the 'points' data is correctly aligned with the 'team' data. This is arguably the most complex step, as it leverages distributed computing concepts to solve a seemingly simple column concatenation problem.

The code below first imports the necessary functions, defines the window specification `w`, and then uses `withColumn` to create the sequential 'id' key in both `df1` and `df2`. After both DataFrames are indexed, the `join` method is called using this temporary 'id' column as the joining criterion. Finally, the temporary 'id' column is removed using the `drop` function, leaving only the desired columns ('team' and 'points') in the resulting `final_df`.

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

```
#add column to each DataFrame called 'id' that contains row numbers from 1 to n
w = Window().orderBy(lit('A'))
df1 = df1.withColumn('id', row_number().over(w))
df2 = df2.withColumn('id', row_number().over(w))

#join together both DataFrames using 'id' column
final_df = df1.join(df2, on=).drop('id')

#view final DataFrame
final_df.show()
```

```
+-------+------+
| team|points|
+-------+------+
| Mavs| 22|
| Nets| 25|
| Nets| 41|
|Blazers| 17|
| Heat| 32|
| Heat| 50|
|Thunder| 18|
+-------+------+
```

Reviewing the Final Merged PySpark DataFrame

The output of the `final_df.show()` command confirms the success of the indexed join strategy. The 'points' column, sourced entirely from `df2`, has been accurately appended to `df1`, aligning perfectly with the corresponding 'team' records. For instance, the first record 'Mavs' is correctly paired with '22', and the second 'Nets' is paired with '25', and so on, following the precise sequence established during the initial creation of the DataFrames.

This method highlights a fundamental aspect of working with distributed DataFrames in PySpark: whenever positional consistency is required, a mechanism must be introduced to force that ordering explicitly across the distributed partitions. The use of the row_number() over a full-partition window provides a robust and reliable way to handle such column addition tasks, transforming two separate DataFrame structures into a cohesive, enriched dataset ready for further analysis.

Additional Resources for PySpark DataFrame Manipulation

For developers seeking to expand their knowledge of PySpark, mastering various DataFrame

manipulation techniques is essential. The following resources offer further guidance on common data engineering tasks within the Spark ecosystem, building upon the foundational knowledge of column addition and indexing demonstrated here:

PySpark: How to Add New Column with Constant Value

PySpark Documentation on Joins and Window Functions

Tutorials on optimizing Spark joins for performance