PySpark: Add Column from Another DataFrame

Authored by

stats writer

November 16, 2025

Introduction: The Challenge of Column Addition in PySpark

In traditional sequential programming environments, merging data based purely on row order is straightforward. However, when working with distributed computing frameworks like PySpark DataFrames, which leverage parallel processing across multiple nodes, simply assuming that the row order remains consistent across different operations or between two distinct DataFrames can lead to unpredictable and incorrect results. This fundamental difference necessitates a robust, explicit method for aligning rows when attempting to add a column from one DataFrame to another when no natural key exists. The standard method involves leveraging powerful Window functions to generate a temporary, consistent identifier that facilitates a reliable, ordered join operation.

When we need to transfer data--specifically, the values from a column--from a source DataFrame (DF2) onto a target DataFrame (DF1), and we must ensure that the alignment is based on the existing row sequence, we cannot rely on the implicit indexing often found in tools like Pandas. PySpark requires an explicit join key. Since row order itself is not a join key, we must manufacture one using a technique that consistently assigns a sequence number to every row in both DataFrames. This approach ensures that the first row of DF1 joins correctly with the first row of DF2, the second with the second, and so forth, regardless of how Spark partitions the data internally.

The following prescribed syntax provides a bulletproof way to perform this row-wise column addition. It utilizes standard PySpark SQL functions, ensuring high performance and compatibility within the distributed environment. It involves creating a monotonic sequence for both DataFrames, using that sequence as a temporary join key, executing the merge, and finally dropping the temporary key to yield the final, merged structure. This structure ensures that the distributed nature of Spark does not compromise the intended sequential alignment of the data.

Step-by-Step Syntax for PySpark Column Transfer

To successfully integrate a column from a source DataFrame into a destination DataFrame based solely on row position, we must execute three critical steps: define a Window specification, generate a row number identifier in both DataFrames, and perform the inner join. This mechanism guarantees alignment across the clusters. We employ the `row_number()` function, which is ideal for assigning sequential ranks starting from one within a defined window.

The required PySpark syntax is concise yet powerful. We first import necessary utilities: row_number for sequencing and lit (literal) for creating a dummy ordering column within the window specification. The Window object itself is imported to define the partitioning boundaries. Although we are not partitioning the data here, we must define an ordering within the window to ensure the row numbers are generated sequentially. The use of `lit('A')` serves this purpose--it provides a constant value that forces the ordering mechanism to process the rows consistently

across the entire DataFrame, treating it as a single partition for numbering purposes.

The process is summarized in the following PySpark code block. Observe how the temporary column named 'id' is generated independently on both DataFrames (`df1` and `df2`) using the identical window definition (`w`). This symmetry is crucial for the subsequent join operation. After the join is executed on this new 'id' column, we immediately use the `.drop('id')` method to clean up the resulting DataFrame, returning only the desired columns: the original columns from `df1` plus the transferred column(s) from `df2`.

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

# Define a Window specification ordered by a constant literal value.
# This ensures row numbering treats the entire DataFrame as one group.
w = Window().orderBy(lit('A'))

# Add the temporary 'id' column to each DataFrame using row_number()
df1 = df1.withColumn('id', row_number().over(w))
df2 = df2.withColumn('id', row_number().over(w))

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

Why Direct Indexing Fails in Distributed Systems

Understanding why the approach above is necessary requires grasping the nature of distributed processing in frameworks like Spark. Unlike data structures that reside entirely in memory on a single machine (like a Pandas Series or list in Python), a PySpark DataFrame is partitioned across several worker nodes. The concept of a global, inherent row index is fundamentally incompatible with this architecture, as rows are processed in parallel and their physical order is not guaranteed across operations unless explicitly enforced.

If one were to attempt to rely on the underlying RDD indexing (e.g., using `zipWithIndex`), the results can become non-deterministic, especially after transformations that shuffle the data. Any operation that requires shuffling, such as grouping or aggregation, destroys the original ordering. Furthermore, even if the data hasn't been shuffled, Spark prioritizes efficiency, meaning the rows might be processed in the most efficient order across partitions, which rarely corresponds to a human-readable sequence from 1 to N. Therefore, to ensure that we correctly map the first record of DF1 to the first record of DF2, we must explicitly calculate that position using a window function that forces a sequential calculation over the entire dataset.

The use of the Window function combined with `orderBy(lit('A'))` is the accepted standard pattern. The `orderBy` clause specifies the order in which the row_number() function assigns its values. By ordering by a literal constant, we bypass dependency on any existing data columns, effectively forcing the numbering sequence to run over the entire set of rows sequentially. This deterministic assignment ensures that both DataFrames receive identical identifiers for the corresponding records, setting the stage for a safe and accurate inner join.

Example: How to Add Column from Another DataFrame in PySpark

Practical Example Setup: Defining Source DataFrames

To illustrate this technique, let us define two distinct DataFrames. The goal is to append the 'points' column from the second DataFrame (`df2`) to the rows of the first DataFrame (`df1`), maintaining the row correspondence exactly as the data was initially defined.

Our first DataFrame, named **df1**, contains a list of basketball team names. Note that some teams appear multiple times, which is perfectly acceptable as we are joining purely by row index, not by team name. The definition of the Spark Session and the DataFrame creation process are detailed below. It is standard practice to initialize the Spark Session first, which is the entry point for all Spark functionality.

This initial setup shows the creation of `df1` using predefined lists of data and column headers. The resulting output demonstrates the simple structure of `df1`, which consists of a single column labeled 'team'.

Defining the Target DataFrame (df1)

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

# Define the base data for DataFrame 1 (team names)
data1 = ,
,
,
,
,
,
]

Define the column name for df1
columns1 =

```
# Create DataFrame 1
df1 = spark.createDataFrame(data1, columns1)

# View the structure and content of df1
df1.show()

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

Defining the Source DataFrame (df2)

Next, we define our second DataFrame, **df2**, which contains the numerical data we wish to append--the 'points' values. It is critical that the number of rows in `df2` exactly matches the number of rows in `df1`. If the row counts are mismatched, the row_number approach will still execute, but the resulting join will only align records where corresponding IDs exist, potentially dropping or duplicating records incorrectly depending on the size discrepancy. For this demonstration, both DataFrames have seven records, ensuring a perfect one-to-one mapping.

The structure of `df2` is simple, containing only the 'points' column. The data generation process for `df2` mirrors that of `df1`. By using separate data lists and column definitions, we treat these as completely independent datasets residing within the distributed environment, setting the stage for the necessary alignment process.

This source DataFrame represents the column content that needs to be transferred based on sequential order. The data types are implicitly inferred by Spark upon creation, but typically, when transferring data, one must ensure type compatibility if complex transformations were involved, although in this case, simple integer transfer is expected.

```
# Define the data for DataFrame 2 (point values)
data2 = ,
,
```

,
,
,
,
]

Define the column name for df2
columns2 =

# Create DataFrame 2
df2 = spark.createDataFrame(data2, columns2)

# View the structure and content of df2
df2.show()

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

Final Execution and Validation of the Result

We can now execute the full integration sequence using the syntax established earlier. This process demonstrates the successful movement of the 'points' data from `df2` and its addition to `df1` based on the corresponding row sequence.

The code below performs the critical steps: generating the temporary 'id' column on both DataFrames using the global Window ordered by a literal, performing the join operation on the 'id', and then showing the final, merged result. The symmetry in applying the `row_number().over(w)` calculation is what guarantees the accurate one-to-one correspondence between the rows of the two distributed datasets.

The resulting DataFrame, `final_df`, clearly shows the successful alignment. For instance, the first record ('Mavs') correctly receives the first points value (22), the second record ('Nets') receives 25,

and so on. This validation confirms that the use of the row_number function within a constant-ordered Window specification is the correct and reliable methodology for performing row-order based column addition in PySpark.

```
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 (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 both DataFrames using 'id' column 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|
+-------+------+
```

Notice that the **points** column from **df2** has been successfully added to the DataFrame named **df1**, confirming the correctness of the distributed row alignment technique.

Performance Considerations and Alternatives

While using the row_number function within a single, global window is highly effective for ensuring accurate row-based alignment, it is important to understand its performance implications. By defining a window without a partition clause (`Window()`) and ordering by a literal, Spark is essentially forced to collect all data onto a single executor before assigning the sequence numbers. This operation, often referred to as a global sort, is computationally expensive and can become a significant bottleneck when dealing with extremely large DataFrames (hundreds of

gigabytes or terabytes).

In scenarios where performance optimization is paramount, alternatives must be considered, although they often come with limitations. One alternative is the RDD function `zipWithIndex()`. This function assigns sequential indices, but it operates on the underlying RDD structure and lacks the robustness and deterministic guarantees of a SQL window function, especially when preceded by complex transformations. Another approach might involve utilizing monotonically_increasing_id, but this function only guarantees that generated IDs increase within each partition and does not guarantee strict sequence numbering (1, 2, 3...) or global uniqueness, making it unsuitable for a strict row-by-row join.

Therefore, the Window function method, despite the global shuffle overhead, remains the most recommended and reliable technique when the objective is a perfectly ordered, row-based transfer of data between two DataFrames that originally lacked a common key. Data engineers should weigh the cost of the global sort against the absolute necessity of maintaining strict row alignment based on the initial input order. For smaller to moderately sized DataFrames, this approach is the clear winner in terms of correctness and simplicity.

Summary of Best Practices

When working in PySpark and facing the requirement to merge data based on implicit row position, the critical best practice is to always transform that implicit order into an explicit, deterministic join key. The combination of the Window function and row_number provides this essential key.

Remember the key components:

Initialization: Import `row_number`, lit, and Window.
Window Definition: Define the window `w = Window().orderBy(lit('A'))` to enforce a global order.
Key Generation: Apply `df.withColumn('id', row_number().over(w))` symmetrically to both source and target DataFrames.
Joining: Execute the inner join on the temporary 'id' column.
Cleanup: Drop the 'id' column post-join to ensure a clean final schema.

Adhering to these steps guarantees that your data operations remain accurate and reliable within the scalable, distributed architecture of Apache Spark.

PySpark: How to Add New Column with Constant Value