How to Join PySpark DataFrames with Different Column Names

Authored by

stats writer

January 20, 2026

One of the common challenges when working with large-scale data processing using PySpark is merging two related datasets when their respective join keys--or column names--do not match. Unlike traditional SQL, directly joining on mismatched column names in DataFrames requires an intermediate step. Fortunately, PySpark provides an elegant solution by utilizing the withColumn function to standardize the keys before executing the join operation.

To successfully join two DataFrames (`df1` and `df2`) based on different column names (e.g., `team_id` in `df1` and `team_name` in `df2`), you must first rename both columns temporarily to a common key, such as `id`. The following consolidated syntax demonstrates this powerful technique:

df3 = df1.withColumn('id', col('team_id')).join(df2.withColumn('id', col('team_name')), on='id')

Understanding the Renaming and Joining Process

The method detailed above ensures a clean, explicit join without modifying the original structure of the source DataFrames permanently. We employ the `withColumn` function combined with the `col` function from `pyspark.sql.functions` (assuming it is imported) to achieve this necessary standardization.

This process is sequential and crucial for alignment. The temporary key, **id** in this case, acts as the unifying field that allows the join operation to proceed correctly. Note that because we use withColumn, the original columns (`team_id` and `team_name`) are retained in the final result `df3` alongside the newly created common `id` column.

Here is a breakdown of the specific actions executed by the syntax:

First, the expression `df1.withColumn('id', col('team_id'))` creates a new column named **id** in `df1`, populated by the values from the existing **team_id** column.
Next, the expression `df2.withColumn('id', col('team_name'))` performs an identical operation on `df2`, creating a new **id** column using values from the **team_name** column.
Lastly, the join method is executed, merging `df1` and `df2` based on shared values found within the newly created **id** columns across both DataFrames.

Setting Up the Example DataFrames for Demonstration

To illustrate this technique practically, let us define two sample DataFrames representing basketball team statistics. We will assume we have already initialized our SparkSession, which is essential for any PySpark operation. The goal is to combine these two datasets, even though the team identifiers are labeled differently.

Our first dataset, `df1`, contains points scored and uses the identifier **team_ID**:

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

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

#define column names
columns =

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

#view dataframe
df1.show()

+-------+------+
|team_ID|points|
+-------+------+
Mavs	18
Nets	33
Lakers	12
Kings	15
Hawks	19
Wizards	24
Magic	28
+-------+------+

Next, we define our second dataset, `df2`, which contains assist statistics but uses the identifier **team_name**. Notice the crucial difference in the naming convention of the primary key used for identification between the two DataFrames.

#define data
data = ,
,
,

,
,
,
]

#define column names
columns =

#create dataframe using data and column names
df2 = spark.createDataFrame(data, columns)

#view dataframe
df2.show()

+---------+-------+
|team_name|assists|
+---------+-------+
Hawks	4
Wizards	5
Raptors	5
Kings	12
Mavs	7
Nets	11
Magic	3
+---------+-------+

Implementing the Cross-Key Join Operation

With our sample DataFrames established, we can now apply the previously discussed technique. We execute an inner join (the default type when only the `on` parameter is specified) between `df1` and `df2`. This requires temporarily projecting the original keys (`team_ID` and `team_name`) onto the consistent key **id**. This guarantees that only records where the team names match are included in the final result, `df3`.

The key insight here is that the withColumn operation is chained directly into the join operation, making the code highly efficient and readable. The resulting DataFrame, `df3`, will contain all columns from both source DataFrames, plus the new common key column.

#join df1 and df2 on different column names
df3 = df1.withColumn('id', col('team_id')).join(df2.withColumn('id', col('team_name')), on='id')

```
#view resulting DataFrame
df3.show()

+-------+-------+------+---------+-------+
| id|team_ID|points|team_name|assists|
+-------+-------+------+---------+-------+
| Hawks| Hawks| 19| Hawks| 4|
| Kings| Kings| 15| Kings| 12|
| Magic| Magic| 28| Magic| 3|
| Mavs| Mavs| 18| Mavs| 7|
| Nets| Nets| 33| Nets| 11|
|Wizards|Wizards| 24| Wizards| 5|
+-------+-------+------+---------+-------+
```

Analyzing the Complete Joined Result

The resulting DataFrame, `df3`, successfully merges the records where **team_ID** from `df1` matched **team_name** from `df2`. We can observe that teams present in only one dataset, such as 'Lakers' (only in `df1`) and 'Raptors' (only in `df2`), were excluded because a default inner join was performed.

Crucially, the resulting schema of `df3` now includes five columns: the new common join key (**id**), the original key from `df1` (**team_ID**), the payload from `df1` (**points**), the original key from `df2` (**team_name**), and the payload from `df2` (**assists**). This redundancy of having three columns representing the team name (`id`, `team_ID`, `team_name`) is typical when using withColumn before the join, but it ensures transparency and verification of the join key.

Refining the Output with the `select` Transformation

While the initial join is successful, it often results in redundant columns, particularly the duplicated key fields (`team_ID` and `team_name`) alongside the temporary `id` key. For practical analysis and storage efficiency, it is often necessary to clean up the resulting schema. PySpark allows us to chain the join operation directly with the `select` function to specify exactly which columns we wish to retain.

By appending a `.select()` clause to the complex join statement, we can streamline `df3` to contain only the necessary data points--in this case, the standardized team identifier (**id**), the **points**, and the **assists**. This simplifies the final dataset dramatically, removing clutter and optimizing downstream processing.

The following syntax demonstrates how to achieve this streamlined result, ensuring that we only display the columns relevant to our aggregate statistics:

#join df1 and df2 on different column names
df3 = df1.withColumn('id', col('team_id')).join(df2.withColumn('id', col('team_name')), on='id')
.select('id', 'points', 'assists')

#view resulting DataFrame
df3.show()

+-------+------+-------+
| id|points|assists|
+-------+------+-------+
Hawks	19	4
Kings	15	12
Magic	28	3
Mavs	18	7
Nets	33	11
Wizards	24	5
+-------+------+-------+

Notice that only the **id**, **points**, and **assists** columns are shown in the joined DataFrame, providing a clean and targeted final result.

Conclusion and Further PySpark Exploration

Handling mismatched column names during a join operation in PySpark is a common requirement in data engineering workflows. By strategically using the withColumn function to establish a temporary, common join key, we can reliably merge disparate datasets. This method is highly flexible and scalable, performing efficiently even on massive distributed datasets typical of the Spark ecosystem.

Remember that while this example used an inner join, the same renaming technique is applicable regardless of the specific join type (e.g., left, right, or full outer join). The fundamental requirement remains the creation of a consistently named key across both DataFrames before the join condition is evaluated.

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

(Related tutorial on data filtering)
(Related tutorial on window functions)

(Related tutorial on aggregation)