# Do a Right Join in PySpark (With Example)

Authored by

**stats writer**

November 16, 2025

## Introduction to Data Joining in PySpark

Data manipulation is a core requirement in modern data engineering and analysis. When working with large datasets distributed across a cluster, tools like PySpark become essential. PySpark, the Python API for Apache Spark, provides robust functionalities for merging disparate data sources, primarily through the use of join operations. Understanding how to correctly implement these joins is fundamental to preparing complex data for machine learning models or business intelligence reporting. This guide focuses specifically on executing a Right Join, a critical technique for preserving integrity of records from one dataset while enriching it with matching information from another.

Joining operations in PySpark involve combining two DataFrame objects based on common key columns, much like standard SQL joins. The choice of join type--whether it be inner, left, right, or full outer--dictates which rows are preserved in the final combined DataFrame. For data analysts who need to ensure that all records from a primary reference table are maintained, regardless of whether a match exists in the secondary table, the Right Join is the preferred method. Mastering this function ensures data completeness and prevents accidental data loss during transformation pipelines.

## Understanding the Right Join Operation

A Right Join, often referred to as a RIGHT OUTER JOIN, is a type of join operation that returns all rows from the right-hand DataFrame (the second DataFrame specified in the join function call). Concurrently, it returns only the matching rows from the left-hand DataFrame. If a row exists in the right DataFrame but has no corresponding match in the left DataFrame, the columns originating from the left DataFrame will be populated with Null values. This characteristic makes it fundamentally useful when the integrity of the secondary dataset must be preserved entirely.

In PySpark, performing a Right Join is straightforward using the built-in `.join()` method available on the DataFrame object. The crucial element is specifying the `how='right'` argument within the method call. The general syntax dictates the order of the DataFrames, the shared key column(s), and the join type. The right DataFrame is conceptually the one that supplies all rows to the final result set, ensuring that no record from that source is discarded due to a lack of matching keys.

The basic structure for executing this operation is intuitive and highly readable:

**df_joined = df1.join(df2, on=, how='right ').show()**

In this specific syntax, `df1` is the left DataFrame, and `df2` is the right DataFrame. The join is executed based on the values found in the column named `team`. The output, `df_joined`, will

contain every row from `df2`. Only rows from `df1` that possess a corresponding value in the `team` column will be successfully merged. Rows from `df2` without a match in `df1` will still appear, but their corresponding columns derived from `df1` will display Null.

## Contrasting Right Joins with Other Join Types

To fully appreciate the utility of the Right Join, it is helpful to compare it briefly with the other common join types available in PySpark. The main distinction lies in how they handle non-matching records.

**Inner Join:** This is the most restrictive join. It only returns rows where keys match in **both** the left and right datasets. Any record existing exclusively in one dataset is discarded entirely.

**Left Join (or Left Outer Join):** This is the inverse of the Right Join. It guarantees that all rows from the left dataset are preserved, and only matching rows from the right dataset are included. Non-matching fields from the right side are filled with Null.

**Full Outer Join:** This is the least restrictive. It returns all rows from **both** the left and right datasets. If a match does not exist in the counterpart dataset, the resulting columns are filled with Null values.

Choosing the Right Join specifically indicates an architectural decision that privileges the integrity and completeness of the secondary dataset (the right-hand side). This is typical when appending supplementary data, such as statistical metrics or descriptive labels, to a core reference list that must remain exhaustive. If you mistakenly used an Inner Join when a Right Join was required, you would inadvertently drop critical records from your secondary source if they lacked a key match in the primary source.

## Practical Implementation: Defining Initial DataFrames

To demonstrate the functionality of the right join in a practical context, we will first establish a PySpark environment and define two sample DataFrame objects. These examples simulate combining basketball team statistics, where one DataFrame holds points scored and the other holds assists recorded. We initiate the process by creating a SparkSession, which serves as the entry point to utilizing Spark functionality in Python.

The first dataset, referred to as `df1` (the left DataFrame in the upcoming join operation), contains team names and total points scored. Notice that this dataset contains information for six distinct teams, including "Warriors" and "Hawks," which we will later observe may not exist in our second dataset.

The code snippet below shows the setup of `df1` and its contents:

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

#define data
data1 = ,
,
,
,
,
]

#define column names
columns1 =

#create dataframe using data and column names
df1 = spark.createDataFrame(data1, columns1)

#view dataframe
df1.show()

+--------+------+
| team|points|
+--------+------+
| Mavs| 11|
| Hawks| 25|
| Nets| 32|
| Kings| 15|
|Warriors| 22|
| Suns| 17|
+--------+------+



## Defining the Right-Hand DataFrame (df2)

Next, we define the second dataset, `df2`, which will act as the right DataFrame. This dataset contains the team names and their total assists. Crucially, `df2` contains a record for the "Grizzlies," a team absent from `df1`, and lacks records for "Hawks" and "Warriors," which were present in `df1`. This specific asymmetry is intentional, as it clearly illustrates how the Right Join handles non-matching keys by preserving all records from this dataset.

The use of `SparkSession.createDataFrame()` method is a standard way in PySpark to ingest data from native Python structures (like lists of tuples) into the distributed DataFrame format, allowing Spark to efficiently manage and process the data across the cluster.

Here is the definition and view of `df2`:

**#define data**
**data2 = ,**
**,**
**,**
**,**
**]**

#define column names
columns2 =

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

#view dataframe
df2.show()

```
+---------+-------+
| team|assists|
+---------+-------+
| Mavs| 4|
| Nets| 7|
| Suns| 8|
|Grizzlies| 12|
| Kings| 7|
+---------+-------+
```



## Executing the Right Join

With both `df1` (points data, the left side) and `df2` (assists data, the right side) defined, we can now execute the Right Join. We specify the join condition using the `on=` parameter, which tells PySpark to look for equivalent values in the `team` column of both DataFrames. The critical instruction is `how='right'`, which enforces the preservation of all records originating from `df2`.

The resulting DataFrame, `df_joined`, will seamlessly merge the `points` and `assists` columns where the `team` names match. For teams present in `df2` but absent in `df1`, the values for the

`points` column will be automatically inserted as Null by the Spark engine.

Below is the syntax and the output generated by performing the Right Join:

**#perform right join using 'team' column**
**df_joined = df1.join(df2, on=, how='right').show()**

```
+---------+------+-------+
| team|points|assists|
+---------+------+-------+
| Mavs| 11| 4|
| Nets| 32| 7|
| Suns| 17| 8|
|Grizzlies| null| 12|
| Kings| 15| 7|
+---------+------+-------+
```

## Analyzing the Joined Result Set

The output clearly illustrates the mechanics of the Right Join. We observe five rows in the final result, mirroring the five rows present in the right DataFrame (`df2`). This confirms that every record from the right source was successfully retained.

For rows where the `team` name existed in both DataFrames (Mavs, Nets, Suns, Kings), the data from `df1` (points) and `df2` (assists) were combined perfectly. For example, the "Mavs" row correctly shows 11 points (from `df1`) and 4 assists (from `df2`).

The most instructive row is "Grizzlies." Since "Grizzlies" appeared in `df2` but not in `df1`, the row was preserved in the join, but the column derived from `df1` (`points`) was populated with the value `null`. This handling of missing data is the defining feature of any outer join and demonstrates the successful execution of the right join policy. Conversely, teams present in `df1` but absent in `df2` (e.g., "Hawks," "Warriors") were completely excluded from the final result, as the Right Join does not guarantee their inclusion unless a match exists in the right DataFrame.

## Handling Null Values and Performance Considerations

The appearance of the `null` value is a critical detail. In PySpark, `null` signifies the absence of a value, typically indicating that no corresponding record was found during the join process. When integrating datasets, data engineers often need to post-process these Null values using functions like `fillna()` or `coalesce()` to replace them with meaningful default values (e.g., 0 for numerical

data) before analysis can proceed. Failing to address `null` values can lead to unexpected errors or incorrect statistical outputs in downstream applications.

While conceptually simple, performing joins on massive datasets in PySpark can be resource-intensive, often requiring significant data shuffling across the cluster nodes. When executing any outer join, including the Right Join, the Spark optimizer works to distribute the data efficiently. To maximize performance, it is generally recommended to ensure the key columns used for joining are highly selective and that data skew is minimized. Furthermore, utilizing broadcast joins (if one DataFrame is small enough to fit into the driver memory) can drastically reduce shuffling overhead and improve execution speed for Spark jobs involving right joins.

Effective use of the SparkSession configuration and careful planning of data partitioning are crucial for ensuring that joins execute quickly and reliably in a production environment. Always verify the resulting row count and schema after a complex join to confirm that the desired data retention policy--in this case, preserving all rows from the right side--has been correctly applied.