# Do an Outer Join in PySpark (With Example)

Authored by

**stats writer**

November 16, 2025

## Understanding the PySpark Outer Join

The ability to combine disparate datasets is fundamental to modern data processing, and within the Apache Spark ecosystem, this is primarily achieved through join operations on DataFrames. When working with PySpark, selecting the correct join type--whether it be an inner, left, right, or outer join--is critical for achieving the desired analytical result. An **Outer Join**, often referred to as a **Full Outer Join**, is specifically designed to ensure that no records are lost from either of the joining tables during the aggregation process. This is particularly valuable in scenarios where comprehensive data coverage is necessary, even if matching keys do not exist across all datasets.

Unlike an **Inner Join**, which only returns rows where the join key exists in both the left and right DataFrames, the Outer Join provides a complete union of the records. If a row in the first DataFrame has no corresponding match in the second DataFrame based on the specified join column, the row from the first DataFrame is still included in the final result. Correspondingly, the columns derived from the second DataFrame will contain placeholder values for the missing data. This symmetric inclusion guarantees that every record from both input datasets contributes to the final output, making it an indispensable tool for reconciliation and holistic data auditing.

The core functionality of the full outer join is to maximize data retention. For any rows that match on the specified key, the output DataFrame combines the corresponding fields from both inputs. For rows unique to one DataFrame, the output includes the fields from the source DataFrame and populates the fields of the unmatched DataFrame with Null values. Understanding this mechanism is essential for interpreting the results, especially when dealing with missing or incomplete source data. The formal implementation of this operation in PySpark maintains simplicity and efficiency, adhering closely to standard relational database join semantics.

## Core Syntax for Full Outer Joins in PySpark

The syntax for executing a join operation in PySpark is highly intuitive, leveraging the method chaining capabilities inherent in the DataFrame API. To perform a full outer join, the primary method used is `.join()`, which takes several critical arguments: the second DataFrame to be joined, the common column(s) on which the join should occur, and the type of join to execute, specified via the `how` parameter. Specifying the join type as `'full'` or `'fullouter'` instructs Spark to perform the desired full outer join operation, ensuring all records are preserved.

The general structure involves calling the `.join()` method on the first DataFrame (`df1`), passing the second DataFrame (`df2`) as the first positional argument. Subsequently, the join condition is defined using the `on` parameter, which typically accepts a list of column names acting as the join keys. Finally, the critical component is setting the `how` parameter to `'full'`. This simple, yet powerful, combination of parameters clearly defines the intent and execution logic for the

distributed computing environment provided by Spark.

Here is the foundational syntax required to perform a full outer join in PySpark. This example illustrates how two DataFrames, conventionally named `df1` and `df2`, are merged based on a common field, represented here by the placeholder column `team`, resulting in the joined DataFrame `df_joined`.

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

In this specific implementation, the operation joins the DataFrames `df1` and `df2` using the column named `team` as the linking key. The defining characteristic of the `how='full'` parameter is that every single row from both DataFrames will be included in the resulting output. Where matches are found, the data is concatenated horizontally; where matches are absent, the missing fields are automatically populated with **Null** values, which is the standard behavior for data preservation in an Outer Join context.

## Prerequisites: Setting up DataFrames in PySpark

Before any join operation can be executed, the foundational components--the DataFrames--must be instantiated within a running Spark environment. This initialization process typically begins with the creation of a SparkSession, which acts as the entry point to programming Spark with the DataFrame API. The **SparkSession** handles resource coordination and execution context across the cluster, making it the essential prerequisite for all PySpark operations. Once the session is active, data can be loaded from various sources--such as CSV files, databases, or in our simplified example, from local Python lists--and converted into the highly optimized distributed collection known as the DataFrame.

The DataFrames used in a join must share at least one common column that serves as the join key. While the column names do not strictly have to be identical in the source DataFrames, using identical names (as is the case with our `team` column) simplifies the `on` parameter specification in the `.join()` method. If the column names were different, the syntax would require an explicit definition of the column pairs involved in the linkage. Defining the data structure clearly, including column names and their respective data types, ensures that the join operation executes without schema mismatch errors.

For the purpose of demonstrating the outer join, we will establish two distinct DataFrames. The first DataFrame, `df1`, represents one set of metrics (e.g., points scored), and the second DataFrame, `df2`, represents a different set of metrics (e.g., assists made). Crucially, these two datasets must have slightly mismatched keys--some teams exist in `df1` but not `df2`, and vice versa--to fully illustrate how the full outer join handles non-matching rows by introducing Null values. This setup

provides a perfect scenario to observe the behavior of the `how='full'` parameter.

## Creating the Initial PySpark DataFrames (df1 and df2)

To perform the practical demonstration, we must first define and display our input DataFrames. The following code block initializes the Spark context using `SparkSession.builder.getOrCreate()` and then constructs `df1`, which contains data regarding basketball teams and their points. This DataFrame serves as the initial left-hand side of our join operation. Note the necessity of defining both the raw data (as a list of lists) and the specific column names before creating the distributed DataFrame using `spark.createDataFrame()`.

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

# Define data for df1 (Teams and Points)
data1 = ,
,
,
,
,
]

# Define column names for df1
columns1 =

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

# View the contents of df1
df1.show()

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

Following the definition of the first dataset, we define `df2`, which represents the second set of metrics (assists). It is crucial to observe the differences in the team names present in this DataFrame compared to `df1`. While some teams overlap (Mavs, Nets, Kings, Suns), other teams are unique to `df1` (Hawks, Warriors) or unique to `df2` (Grizzlies). This deliberate mismatch is key to demonstrating the comprehensive preservation behavior of the full outer join, ensuring that all unique records are retained, regardless of which dataset they originate from.

**# Define data for df2 (Teams and Assists)**
**data2 = ,**
**,**
**,**
**,**
**]**

# Define column names for df2
columns2 =

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

# View the contents of df2
df2.show()

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

With both DataFrames initialized and containing intentionally divergent data, we are now ready to execute the join operation. The common column, `team`, will serve as the connection point, allowing PySpark to merge these two disparate sets of statistics into a single, comprehensive dataset using the full outer join logic. This preparation ensures that the subsequent execution phase accurately reflects real-world scenarios involving incomplete or fragmented data records.

## Executing the Full Outer Join Operation

The execution of the Outer Join in PySpark is performed by applying the established syntax to our defined DataFrames, `df1` and `df2`. The objective is to combine all information available about the teams, regardless of whether they appear in both the points DataFrame (`df1`) and the assists DataFrame (`df2`). This is achieved by explicitly setting the join type to `'full'`, signaling to Spark that rows lacking a match should still be included in the output.

The join command below merges the two datasets on the common key, `team`. The resulting DataFrame, `df_joined`, will seamlessly integrate the `points` column from `df1` and the `assists` column from `df2`. The use of the `.show()` action triggers the computation across the Spark cluster and displays the final, consolidated result set directly to the console. This immediate visualization allows for prompt verification that the join logic has been executed correctly according to the full outer strategy.

We use the following specific syntax to perform the full outer join between `df1` and `df2`, linking the records based solely on the values found within the **team** column. Observe how the resulting schema now contains the union of all columns from both input DataFrames: `team`, `points`, and `assists`.

**# Perform full outer join using 'team' column**
**df_joined = df1.join(df2, on=, how='full').show()**

+---------+------+-------+
| team|points|assists|
+---------+------+-------+
|Grizzlies| null| 12|
| Hawks| 25| null|
| Kings| 15| 7|
| Mavs| 11| 4|
| Nets| 32| 7|
| Suns| 17| 8|
| Warriors| 22| null|
+---------+------+-------+

## Detailed Analysis of the Joined Result Set

A careful examination of the resulting DataFrame, `df_joined`, confirms the successful execution of the full outer join. The most significant observation is that the resultant DataFrame contains a total of seven records. This number is derived from the five unique teams in `df2` and the six unique

teams in `df1`, where four teams overlapped (Mavs, Nets, Kings, Suns). The final count (6 + 5 - 4 = 7) accurately reflects the union of all unique keys present across both initial datasets, which is the defining principle of an Outer Join.

For the teams that successfully matched in both DataFrames--specifically, Mavs, Kings, Nets, and Suns--the resulting row contains valid, non-Null values across all three columns (`team`, `points`, and `assists`). For instance, the row for 'Mavs' shows 11 points (from `df1`) and 4 assists (from `df2`), demonstrating a complete match and successful horizontal concatenation of records based on the shared key. These rows represent the intersection that an Inner Join would have returned, but they are seamlessly integrated into the comprehensive outer join structure.

The power of the full outer join is most evident when analyzing the rows that did not have a match in one of the input DataFrames. Consider the 'Grizzlies' row: this team existed exclusively in `df2` (the assists DataFrame). Consequently, the `points` column, derived from `df1`, is populated with the value `null`, indicating the absence of corresponding data in the first source. Conversely, for the teams 'Hawks' and 'Warriors', which were unique to `df1`, the `assists` column (derived from `df2`) displays `null` values. This systematic use of `null` ensures that the data structure remains consistent while explicitly marking data gaps, thus providing a complete picture of all available team records.

## Handling Mismatched Keys and Null Values

The proper management and interpretation of Null values are central to working with full outer joins in PySpark. When the join operation cannot find a corresponding record for a given key in one of the DataFrames, **Spark** automatically inserts `null` into the columns belonging to the unmatched DataFrame. This is not an error state, but rather a deliberate mechanism to maintain row alignment and preserve data integrity across the union of keys.

As observed in the result set, if a team name did not appear in both input DataFrames, a value of **null** appeared in the column associated with the missing dataset. Specifically, for 'Hawks', which had 25 points in `df1` but was absent in `df2`, the resulting row received a **null** in the **assists** column. This explicit marking distinguishes genuinely missing data from zero values or other placeholder metrics, which is crucial for subsequent data cleaning and analytical steps. Data practitioners must be aware of this behavior, as `null` values often require specific handling--such as imputation or filtering--before performing aggregate functions or statistical modeling.

In summary, the full outer join operation using the `how='full'` parameter provides the highest degree of data retention among all join types. It ensures that all unique keys are present, bridging data from multiple sources into a single, consolidated DataFrame. This capability is vital in complex distributed processing environments where combining fragmented data sets is a common

requirement for comprehensive reporting and analysis. Mastering this join type allows users to reliably merge incomplete datasets without sacrificing any underlying information from the input sources.