Do a Left Join in PySpark (With Example)

Authored by

stats writer

November 17, 2025

Performing data transformations, such as joining multiple datasets, is fundamental in modern data engineering, particularly when working with large-scale distributed systems. PySpark, the Python API for Apache Spark, provides robust and highly optimized methods for handling these operations. Among the most common join types is the left join (or left outer join), which is critical for enriching a primary dataset without losing any of its original records, even if matching data is unavailable in the secondary dataset. This guide provides an in-depth explanation of how to execute a left join effectively in PySpark, complete with practical, runnable examples.

The concept behind the left join is straightforward yet powerful: we designate one DataFrame as the "left" side (the base structure) and another as the "right" side (the augmentation source). The resulting combined DataFrame will contain every single row from the left DataFrame. When a row in the left DataFrame finds a match in the right DataFrame based on the specified key, the corresponding columns from the right dataset are appended. Crucially, if no matching entry exists in the right DataFrame, the columns contributed by the right dataset are populated with null values, ensuring that the integrity and count of the primary dataset are maintained throughout the merging process.

Core Syntax for Performing a Left Join in PySpark

In PySpark, the primary method for merging two DataFrames is the built-in .join() transformation. This method requires three essential arguments: the secondary DataFrame, the column(s) on which to join, and the type of join to execute. To specify a left join, we simply set the how parameter to 'left'. The syntax is remarkably concise and follows standard SQL conventions, making it easy to translate existing relational database logic into your distributed computing environment.

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

Analyzing this snippet, df1 represents the left DataFrame, which dictates the output structure and row count. df2 is the right DataFrame being joined onto the first. The on= argument specifies that the join operation must match rows based on identical values found in the column named team across both DataFrames. Finally, how='left' explicitly instructs Spark to perform a left join, ensuring that all records from df1 are included in the final df_joined result, regardless of whether a corresponding record exists in df2. This powerful and flexible mechanism allows data scientists and engineers to integrate disparate datasets seamlessly while maintaining data completeness.

Setting Up the PySpark Environment and DataFrames

To demonstrate the left join in a runnable context, we first need to initialize a SparkSession, which serves as the entry point to programming Spark with the Dataset and DataFrame API. For this

practical example, we will simulate a scenario involving basketball team statistics. Our left DataFrame (df1) will hold basic scoring data, and the right DataFrame (df2) will contain assist metrics. By designing these datasets to have both matching and non-matching keys, we can clearly observe the specific behavior of the left join operation.

Creating these sample DataFrames from raw Python lists is standard practice for localized testing and prototyping in PySpark. The crucial step is the use of spark.createDataFrame(), which efficiently converts local Python data structures into distributed Spark DataFrames, making them ready for large-scale operations. It is important to ensure that the column designated as the join key (in this case, team) exists in both DataFrames and contains consistent data types, although Spark is generally robust enough to handle simple schema inference.

Defining the Left DataFrame (df1)

The first dataset, df1, represents our primary collection of records. This dataset includes various basketball teams along with their accumulated points. Because df1 is positioned as the left DataFrame in the join operation, every record present here is guaranteed to appear in the final output. This is a crucial concept to grasp when choosing the order of DataFrames in a left join; the structural integrity of the left side is always preserved. We initialize the Spark session here for environment setup.

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
+--------+------+

We can clearly see that df1 contains six distinct records. Notably, teams like "Hawks" and "Warriors" are present in this dataset. When we proceed to define the right DataFrame, we will deliberately omit some of these teams to illustrate how the left join mechanism handles missing data. This careful construction ensures that the subsequent join operation accurately demonstrates the insertion of null values where matches are not found, which is the defining characteristic of this type of join.

Defining the Right DataFrame (df2)

The secondary dataset, df2, serves to enrich the primary data in df1 by providing additional metrics, specifically the number of assists for each team. This DataFrame is defined as the right side of our join. It is designed to have some overlap with df1 (e.g., Mavs, Nets, Kings, Suns) but also features missing teams (Hawks, Warriors are absent) and an exclusive team ("Grizzlies"). The inclusion of "Grizzlies" is intentional, as a left join dictates that this record will be excluded from the final output, since its matching key does not exist in the left DataFrame.

#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
+---------+-------+

This DataFrame provides the specific values that the join transformation will attempt to append to df1. The structure of the join key (team) must align perfectly for a match to occur. It is good practice to visually inspect both DataFrames using .show(), especially in debugging complex join logic, to confirm that the key columns are correctly defined and populated before execution. Understanding the initial state of both datasets is key to predicting the final output of the left join.

Executing the Left Join on the Common Column

With both DataFrames (df1 and df2) successfully defined in the SparkSession context, we can now execute the primary operation. We call the .join() method on df1 (the left side), passing df2 (the right side) and specifying the key () and the type of join (how='left'). This single line of code leverages Spark's optimized distributed processing engine to efficiently merge potentially massive datasets.

The resulting DataFrame, df_joined, demonstrates the successful combination of the points column from df1 and the assists column from df2, linked by the shared team column. The use of .show() immediately displays the resulting structure and content in the console, allowing for immediate verification of the join's integrity and logic. This real-time visibility is invaluable during the iterative development process in PySpark.

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

+--------+------+-------+
| team|points|assists|
+--------+------+-------+
| Mavs| 11| 4|
| Hawks| 25| null|

| Nets| 32| 7|
| Kings| 15| 7|
|Warriors| 22| null|
| Suns| 17| 8|
+--------+------+-------+

Interpreting the Results: Handling Mismatches and Nulls

Upon reviewing the output of the left join, several key characteristics confirm its successful execution according to the definition of a left outer join. First and foremost, the resulting DataFrame contains exactly six rows, matching the row count of the left DataFrame (df1). This confirms that no primary records were dropped during the augmentation process. Furthermore, the combined schema now includes all columns from both initial datasets: team, points, and assists.

The most critical aspect to analyze is the handling of non-matching keys. Observe the records for "Hawks" and "Warriors." Both teams exist in df1 but were absent from df2. For these rows, the left join correctly preserves their data from df1 (points 25 and 22, respectively) while inserting the special value null into the corresponding assists column, which originated from df2. This behavior is essential when dealing with incomplete or sparse secondary datasets, as it explicitly flags where enrichment data was missing. Conversely, the "Grizzlies" record, which was present in df2 but not in df1, is entirely excluded from the result, adhering strictly to the left join rule.

Understanding the placement of null values is crucial for subsequent data cleaning and analysis. The presence of null often necessitates post-join steps, such as imputation, filtering, or using the PySpark fillna() function to replace missing values with a default or statistically calculated measure. For example, if we knew that missing assist data should default to 0, we would chain a .fillna(0, subset=) operation immediately after the join to clean the resulting DataFrame for downstream processing.

Advanced Considerations and Multi-Column Joins

While our example used a single column (team) for the join key, PySpark fully supports joining on multiple columns, often necessary when the join key is a composite of several fields. To execute a multi-column left join, the syntax remains nearly identical; one simply lists all required columns within the on parameter array. For instance, if records needed to match both team and year, the join clause would be on=.

Another common scenario involves joining DataFrames where the key columns have different names. If df1 used team_name and df2 used franchise, we cannot use the simplified list syntax. In this case, the on parameter must be replaced with a SQL-style expression or a condition specifying

the equality of the two columns, such as df1.team_name == df2.franchise. When using this conditional syntax, it is vital to explicitly select the columns afterward to manage duplicate key columns that may otherwise appear in the final output.

Performance is a significant factor in distributed computing. When performing large-scale joins, it is important to consider data partitioning and caching. Spark optimizes joins by partitioning data based on the join keys. If one DataFrame is substantially smaller than the other (often defined as being small enough to fit into the memory of a single worker node), consider using a broadcast hash join. By explicitly broadcasting the smaller DataFrame using pyspark.sql.functions.broadcast(), you can often achieve massive performance improvements by avoiding a costly data shuffle across the cluster, leading to much faster join completion times.

Practical Applications and Best Practices

The left join is essential in data warehousing and ETL processes, typically used for dimension table lookups or data augmentation. A classic use case is joining a large fact table (the left DataFrame, holding transaction records) with a smaller dimension table (the right DataFrame, holding descriptive customer or product details). Using a left join ensures that all original transactions are preserved, regardless of whether the corresponding descriptive data is present, which is crucial for auditing and completeness.

A key best practice when working with joins in PySpark involves robust column management. Before executing the join, ensure that only necessary columns are retained in the right DataFrame. Dropping or selecting specific columns prior to the join minimizes data transfer overhead and reduces the chance of schema conflicts, especially if both DataFrames contain columns with identical names that are not the join key. If identical non-key column names exist, the resulting DataFrame will contain duplicate column names, making subsequent access ambiguous unless aliasing or renaming is performed proactively.

Finally, always remember the distinction between the null value produced by a failed join and a value that is inherently zero or empty string in the source data. The null generated by the join indicates a lack of matching information. If your business logic dictates that missing assist data implies zero assists, you must explicitly handle the null conversion. Relying on Spark's native null handling provides a clear audit trail of where data enrichment failed, which is generally preferable to masking the missing values entirely.

Summary of PySpark Left Join Mastery

Mastering the left join is a foundational skill for anyone working with data processing in PySpark. The mechanism guarantees that the entire dataset designated as the left DataFrame remains intact, while non-matching data from the right DataFrame is gracefully represented by null markers.

This property makes the left join indispensable for merging information where data completeness is paramount and losing records is unacceptable.

We demonstrated the fundamental syntax using the .join() function with the how='left' parameter, illustrating its immediate application to merge basketball statistics. We also showed how the resulting DataFrame clearly highlights missing information by inserting null values in the newly added columns for teams like "Hawks" and "Warriors." This explicit handling of missing data is a powerful feature that simplifies debugging and downstream data quality checks.

By applying these techniques, you can confidently integrate diverse data sources within the distributed computing power of PySpark, ensuring both computational efficiency and high data fidelity. Remember to always consider the data types and column names, particularly when dealing with complex join conditions or when optimizing performance for extremely large-scale DataFrames by using techniques like broadcasting the smaller dataset.