Do an Inner Join in PySpark (With Example)

Authored by

stats writer

November 16, 2025

Understanding Joins in PySpark and Big Data

Performing a join operation is fundamental when working with relational datasets, allowing analysts and data engineers to combine information from multiple sources based on common keys. In the realm of PySpark, which is the Python API for Apache Spark, joins are executed efficiently across massive, distributed datasets, a core requirement for modern Big Data processing. When dealing with gigabytes or terabytes of information stored across a cluster, the ability to quickly and accurately merge data stored in separate DataFrames is paramount to deriving meaningful insights. The syntax for achieving this in PySpark is designed to be intuitive, leveraging similar concepts to traditional SQL while optimizing execution for parallel processing.

The standard approach for combining two PySpark DataFrames, say **df1** and **df2**, involves utilizing the built-in `.join()` function. This function requires defining three critical parameters: the second DataFrame to be joined (df2), the column or list of columns to join **on** (known as the join key), and the **how** parameter, which specifies the type of join to execute. Although there are several join types--including left, right, full, and semi-joins--the Inner Join remains one of the most frequently used operations because of its precise filtering mechanism. Understanding how to correctly specify these parameters is the first step toward effective data manipulation in a Spark environment, ensuring that the merged dataset contains only the intersection of records from both sources.

Specifically, the inner join operation retrieves only those records that have matching values in both DataFrames being merged. This means that if a row in **df1** does not have a corresponding entry in **df2** based on the specified join key, that row will be excluded from the resulting merged DataFrame. Conversely, if a key exists only in **df2**, it too will be excluded. This characteristic makes the inner join invaluable for cleaning data, ensuring data integrity across combined datasets, and focusing analysis exclusively on the intersection of the available information. For developers moving from pandas or traditional SQL environments, the transition to the PySpark join syntax is straightforward, although the underlying execution engine--optimized for distributed computing--offers massive performance improvements when scaling up to handle enterprise-level data volumes.

The Mechanics of the PySpark Inner Join

The core syntax for executing an inner join in PySpark is concise and highly readable. It involves chaining the `.join()` method onto the first DataFrame (the left side) and passing the necessary arguments to define the relationship. The explicit definition of `how='inner'` tells the Spark engine precisely which logical operation to perform on the distributed data partitions. This specific command is essential because, while Spark often defaults to an inner join if the `how` parameter is omitted, best practice dictates specifying it clearly for code robustness and maintainability, especially in complex pipelines involving multiple types of joins.

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

In this powerful one-liner, we are defining a new DataFrame called **df_joined**. This new object is the result of combining **df1** with **df2**. The crucial element `on=` specifies that the combination must occur based on matching values found exclusively in the column named **team**. If a 'team' value exists in df1 but not in df2, that record is dropped. Conversely, if a 'team' value exists in df2 but not in df1, that record is also dropped. The final `.show()` action triggers the execution of the distributed computation and prints the resulting DataFrame to the console, allowing immediate inspection of the merged data schema and content.

It is important to note the difference between specifying the join key using the on parameter (which assumes the column names are identical in both DataFrames) versus using a full join expression specified within the `.join()` function. While using `on=` is cleaner when the keys share the same name--as demonstrated here with 'team'--you must use the explicit expression syntax (e.g., `df1.col_a == df2.col_b`) if the join columns have different names in the two source DataFrames. However, for an introductory inner join operation, ensuring consistent column naming across the key fields simplifies the syntax considerably, promoting efficient and less error-prone coding practices within the PySpark environment.

Prerequisites: Setting Up the PySpark Environment and DataFrames

Before executing any join operations, it is necessary to establish the computing environment by initializing a SparkSession. The SparkSession serves as the entry point to all functionality in Spark, managing the connection to the underlying cluster resources and providing the necessary context for creating and manipulating distributed DataFrames. Without a properly instantiated SparkSession, subsequent PySpark commands will fail to execute, as they lack the distributed engine required for processing data efficiently across a cluster. The standard boilerplate code using `SparkSession.builder.getOrCreate()` ensures that Spark is ready to handle our subsequent data manipulation tasks.

Once the session is established, the next crucial step is defining the source data. In this example, we manually define two small datasets, **data1** and **data2**, which simulate raw records derived from Python lists. These lists of lists represent the data structure we intend to load into distributed DataFrames. Although in a real-world Big Data scenario, data is usually loaded from external sources like Parquet files, CSVs, or dedicated data warehouses, creating sample data manually allows us to clearly illustrate the inner join logic without the complexity of external input/output operations. Each dataset requires corresponding column names, defined by **columns1** and **columns2**, ensuring that the DataFrames are properly structured and schema-validated upon creation.

The creation of the DataFrames themselves is achieved using the `spark.createDataFrame()` method. This method takes the raw Python data and the defined column schema, converting the local Python data structure into a distributed, immutable DataFrame object. This object is the fundamental unit of data processing in PySpark, optimized for performance and parallelism. For our example, we create **df1** and **df2**, which represent two separate tables holding related but non-identical information about basketball teams. We explicitly define one common column, 'team', which will serve as our essential join key, linking the points data in df1 to the assists data in df2.

Defining DataFrames for the Inner Join Example (df1 and df2)

To demonstrate the inner join in a practical context, we begin by constructing our two source DataFrames, which contain non-matching records intentionally. Our first DataFrame, **df1**, contains records linking team names to total points scored. This DataFrame acts as the left side of our upcoming join operation. We define the data structures and then call `df1.show()` to visualize the resulting distributed table structure, confirming the presence of six distinct team records along with their respective points tallies.

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

Following the creation of **df1**, we introduce the second DataFrame, **df2**, which represents the right side of the join. This DataFrame holds related information--in this case, the total assists recorded for various teams. It is crucial to observe that while some team names overlap between **df1** and **df2** (Mavs, Nets, Kings, Suns), other names are unique to each DataFrame (Hawks, Warriors in df1; Grizzlies in df2). This non-matching structure is key to demonstrating how the inner join correctly filters the data and isolates only the intersecting records.

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

The preparation step of defining these two distinct, yet related, DataFrames highlights the utility of the join operation. We have successfully simulated a common scenario in Big Data processing where critical metrics are stored in separate silos, requiring a robust mechanism to synthesize them based on a common identifier--the 'team' name. Our goal now is to use the inner join to produce a final table containing only records for teams for which we have both points data (from df1) and assists data (from df2), effectively excluding any incomplete entries.

Executing the PySpark Inner Join Syntax

With the source DataFrames **df1** and **df2** defined and populated, the next step is applying the core PySpark logic to combine them using the inner join strategy. This operation is executed through a single, powerful line of code that encapsulates the source DataFrames, the join key, and the specific join type. We are instructing the distributed engine to map the 'team' column across both datasets and retain only the rows where the values perfectly align, generating a composite record for those teams present in both sources.

The syntax below utilizes the DataFrame method `.join()`, specifying **df2** as the right-hand table. The parameter `on=` explicitly designates the 'team' column as the unique identifier used for matching. Most importantly, `how='inner'` ensures that the resulting DataFrame, **df_joined**, retains only the intersection of the two datasets. This command triggers the optimized execution plan within PySpark, distributing the data shuffling and matching process across the cluster nodes efficiently.

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

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

Upon execution, the distributed computation returns the concise, aggregated result shown in the output block. The resulting table includes the common join key ('team') followed by the columns retained from df1 ('points') and the columns retained from df2 ('assists'). This result confirms that the inner join successfully combined the relevant metrics, creating a unified view of the data. The subsequent analysis of this result set is crucial, as it validates the behavior of the inner join and

confirms which records were successfully mapped across the two disparate sources based on the common key.

Analyzing the Results of the Inner Join

A critical step after performing any join operation in PySpark is verifying the resulting data structure and content. By examining the output of **df_joined**, we can clearly see the fundamental definition of the inner join in action. The resulting DataFrame contains only four rows, corresponding precisely to the four team names that were present in both the original **df1** and **df2** DataFrames: **Kings**, **Mavs**, **Nets**, and **Suns**.

To understand the exclusions, consider the teams that were dropped. **df1** contained data for 'Hawks' and 'Warriors'. Since neither of these team names appeared in the 'team' column of **df2**, the inner join logically excluded them from the final result, as the operation requires a match on both sides. Similarly, **df2** contained a record for 'Grizzlies'. Because 'Grizzlies' was not present in **df1**, that record was also excluded. This stringent matching requirement means the inner join always produces a subset of the Cartesian product of the two tables, specifically focusing only on the intersection of the primary keys.

This behavior stands in clear contrast to other join types. For instance, a left join would have kept all six rows from **df1**, filling the 'assists' column with null values for 'Hawks' and 'Warriors'. A full outer join would have kept all unique team names (seven rows in total), resulting in nulls where data was missing from either side. The precision of the inner join in yielding a complete, non-null set of matched records across the join key is what makes it highly suitable for many data integration tasks where incomplete data rows must be explicitly discarded in Big Data pipelines.

Common Use Cases and Advantages of Inner Joins

The inner join is not merely a technical function; it is a critical analytical tool used across numerous data science and engineering workflows. One of its most common use cases is filtering transactional data against master data lists. For example, if you have a DataFrame of recent sales transactions (df1) and a DataFrame of verified, active customer IDs (df2), an inner join on the customer ID will efficiently produce a resulting DataFrame containing only sales made by currently active customers, automatically excluding erroneous, inactive, or test data. This capability drastically reduces the data volume that subsequent complex computations must handle, improving overall pipeline efficiency.

Another powerful application is enforcing referential integrity across distributed datasets in PySpark. While traditional relational databases often rely on strict foreign key constraints, Spark DataFrames are inherently schema-on-read and lack these built-in constraints. By performing an inner join between a facts table and a dimensions table (e.g., combining orders with product

details), we guarantee that every record in the output has a valid, corresponding entry in the dimension table. If a record fails the inner join (meaning the foreign key is invalid or missing), it is automatically excluded, ensuring high data quality downstream without requiring explicit manual filtering logic.

Furthermore, in scenarios involving A/B testing or feature engineering for machine learning models, the inner join is indispensable. Suppose we generate two sets of features derived from separate data sources (e.g., demographic data and usage statistics). To train a model, we only want to use subjects for whom we have both complete sets of features. An inner join ensures that the final training DataFrame is perfectly balanced and complete regarding the required inputs, preventing the introduction of sparse or null values that could negatively impact model performance. This focus on completeness, inherent to the inner join, saves significant time and computational resources in large-scale analysis.

Alternative Join Types in PySpark (Brief Comparison)

While the inner join is ideal for finding commonalities, PySpark supports several other join types that address different requirements for data combination and retention. Understanding these alternatives is crucial for selecting the optimal strategy based on the analytical goal. The primary alternatives include the Left Outer Join, Right Outer Join, Full Outer Join, and the Semi/Anti Joins. Each type uses the same `.join()` syntax but modifies the `how` parameter accordingly (e.g., `how='left'` or `how='full'`).

The **Left Outer Join** (or simply Left Join) is used when you need to retain all rows from the left DataFrame (df1 in our example) and match them with corresponding rows from the right DataFrame (df2). If a match is not found in df2, the columns belonging to df2 are filled with null values for that row. Conversely, the **Right Outer Join** (Right Join) ensures all rows from df2 are retained, potentially introducing nulls for columns originating from df1. These types are essential when preservation of one dataset's integrity is prioritized over the intersection of data, allowing analysts to retain context even when associated data is missing.

The **Full Outer Join** is the most permissive, retaining all rows from both DataFrames, regardless of whether a match is found in the other DataFrame. If a key exists only in df1, df2's columns are null. If a key exists only in df2, df1's columns are null. This join is typically used for comprehensive data audits where identifying all unmatched records is as important as identifying matched ones, providing a holistic view of the data landscape. Furthermore, specialized joins like the **Left Anti Join** are highly efficient for exclusion filtering in complex Big Data workflows, quickly identifying which records in the left table are missing from the right without returning any columns from the right table.