PySpark: Do a Left Join on Multiple Columns

Authored by

stats writer

November 16, 2025

Understanding the Need for Multi-Column Joins in PySpark

In advanced data processing workflows, it is frequently necessary to merge two **DataFrame** objects based on complex criteria. Unlike simple merges that rely on a single primary key, many **relational database** operations require matching records across multiple attributes simultaneously to ensure data integrity and uniqueness. This is particularly true when dealing with composite keys or ensuring that specific contextual dimensions align between the two datasets being combined within the **PySpark** environment. Mastering the multi-column join is essential for effective data manipulation and transformation using distributed computing resources.

The core challenge arises when simple equality across one column is insufficient to uniquely identify a matching row. For instance, in transactional data, you might need to match both a **user ID** and a **transaction date**. If only the user ID is matched, the join operation could inadvertently link multiple transactions from different days, leading to incorrect aggregation or analysis. Therefore, explicitly defining multiple join conditions ensures precision in merging, guaranteeing that only records satisfying all specified criteria are paired together, which is the foundational principle for accurate big data processing.

The syntax provided by PySpark for handling these complex join requirements is both powerful and expressive, allowing developers to define these logical conditions directly within the `.join()` method call. This approach utilizes column expressions as the join predicate, offering fine-grained control over how the two datasets are reconciled. This capability is paramount when migrating complex legacy **SQL** logic into a distributed PySpark environment.

The Core Syntax for PySpark Left Joins on Multiple Keys

To successfully perform a **left join** in PySpark using multiple corresponding columns, the syntax requires passing a list of equality expressions to the `on` parameter within the `.join()` function. This method is highly recommended when the column names across the two DataFrames differ, as it allows for explicit mapping of the keys.

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

In the illustrative syntax above, the resulting **DataFrame**, named `df_joined`, is created by merging `df1` with `df2`. The join operation is executed specifically on the composite key defined by the pair of columns: `df1.col1` matching `df2.col1`, AND `df1.col2` matching `df2.col2`. Crucially, the `how='left'` argument specifies that this is a **left outer join**, meaning all rows originating from the left DataFrame (`df1`) will be preserved in the output, irrespective of whether a match is found in the right DataFrame (`df2`).

Understanding the implications of the **left join** is vital. Every row from `df1` is guaranteed to be returned in the final result set. For rows in `df1` where no corresponding match exists in `df2` based on the two defined conditions, the columns originating from `df2` will be populated with **null** values. Conversely, only those rows from `df2` that meet the strict criteria of having identical values across both `col1` and `col2` pairs will contribute their data to the output row. This mechanism ensures that the resulting dataset retains the full structure and volume of the primary dataset (`df1`) while enriching it with associated data from the secondary dataset (`df2`).

Setting Up the Environment and Initial DataFrames

To demonstrate this mechanism practically, we must first initialize a **SparkSession** and define our two source DataFrames, `df1` and `df2`. These DataFrames represent typical structured data where a composite key--in this case, a combination of team and position identifiers--is required for accurate linking. We use the `SparkSession.builder.getOrCreate()` method to ensure a robust environment for executing **PySpark** operations, which is the standard practice for setting up a distributed context.

Our primary DataFrame, `df1`, contains performance metrics (points) indexed by `team` and `pos`. This DataFrame will serve as the left side of our join operation, meaning its structure and row count will dictate the final output structure.

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

#define data for df1: team, pos, points
data1 = ,
,
,
]

#define column names for df1
columns1 =

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

#view dataframe
df1.show()

+----+---+------+
|team|pos|points|

```
+----+---+------+
| A| G| 18|
| A| F| 22|
| B| F| 19|
| B| G| 14|
+----+---+------+
```

Next, we define `df2`, which contains supplementary data (assists). Notice that the column names in `df2`--`team_name` and `position`--are intentionally different from those in `df1`, mimicking a common real-world scenario where standardization is not perfect. Furthermore, `df2` includes a team 'C' which does not exist in `df1`, and `df1` includes a combination ('B', 'G') that is missing from `df2`. These discrepancies are crucial for observing the behavior of the **left join** operation.

#define data for df2: team_name, position, assists
data2 = ,
,
,
,
]

#define column names for df2
columns2 =

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

#view dataframe
df2.show()

```
+---------+--------+-------+
|team_name|position|assists|
+---------+--------+-------+
| A| G| 4|
| A| F| 9|
| B| F| 8|
| C| G| 6|
| C| F| 5|
+---------+--------+-------+
```

Defining Complex Join Conditions

The primary objective of this operation is to enrich the performance data in `df1` with the assistance data from `df2`. Since the column names are heterogeneous (`team` vs. `team_name` and `pos` vs. `position`), we must explicitly define the column-level equivalence within the join clause. This explicit definition prevents ambiguity and ensures that the distributed Spark engine knows exactly how to partition and compare the records across the cluster.

We require a match on both the organizational unit (team identification) and the functional role (position). Failure to match on both criteria simultaneously would lead to either an erroneous join or a missing match, depending on the data. The multi-column join ensures this logical **AND** operation is strictly enforced, guaranteeing that the combined key uniquely identifies the row pair.

The required join conditions can be summarized clearly before implementation. By breaking down the complex operation into discrete, logical requirements, we ensure the correctness of the subsequent **PySpark** code.

The **team** column from **df1** must precisely match the **team_name** column from **df2**.
The **pos** column from **df1** must precisely match the **position** column from **df2**.

Executing the Multi-Column Left Join Operation

With the DataFrames prepared and the specific join keys identified, we proceed to execute the **left join**. The chosen syntax utilizes a list of boolean expressions within the `on` parameter, linking the columns across the two DataFrames using the `==` operator. This method is highly flexible and scalable for big data applications handled by the Spark framework, allowing us to specify exactly how the distributed join should be performed.

#perform left join using two conditions
df_joined = df1.join(df2, on=, how='left')

#view resulting DataFrame
df_joined.show()

+----+---+------+---------+--------+-------+
|team|pos|points|team_name|position|assists|
+----+---+------+---------+--------+-------+
A	G	18	A	G	4
A	F	22	A	F	9
B	F	19	B	F	8
B	G	14	null	null	null

```
+----+---+------+---------+--------+-------+
```

The resulting `df_joined` **DataFrame** demonstrates the successful merging of the two datasets based on the composite key. It is crucial to observe the output closely to confirm that the semantics of the **left join** have been correctly applied. Since `df1` had four rows, the resulting DataFrame also contains exactly four rows, maintaining the integrity of the left dataset, regardless of whether matches were found in the right dataset.

Interpreting the Results and Handling Null Values

The output clearly illustrates how the join operation handles both successful matches and non-matching keys. For the first three rows--(A, G), (A, F), and (B, F)--exact matches were found in `df2` based on the two specified conditions, and the corresponding `assists`, `team_name`, and `position` columns were populated correctly. This confirms that the composite key logic performed the expected linkage across the distributed partitions.

The crucial point of observation is the fourth row: (B, G). While the team 'B' exists in `df2`, the combination ('B', 'G') does not exist (only 'B', 'F' exists). Since the join requires BOTH `df1.team == df2.team_name` AND `df1.pos == df2.position` to be true, this specific record from `df1` finds no match in `df2`. Consequently, all columns originating from `df2` (namely `team_name`, `position`, and `assists`) are filled with **null** values. This behavior is the defining characteristic of a **left outer join**, ensuring that no data from the primary DataFrame is discarded simply because supplementary data is unavailable.

Furthermore, it is worth noting the data from `df2` relating to team 'C'. Because 'C' did not exist in the left DataFrame, `df1`, these rows were implicitly excluded from the final result set. Had we performed a **full outer join** or a **right join**, these unmatched rows from `df2` would have been included, demonstrating the necessity of selecting the appropriate join type based on the analytical objective. The left join methodology prioritizes the preservation of the primary dataset's structure, which is a common requirement in data warehousing and reporting.

Refining the Output: Dropping Redundant Columns

Upon successful execution of a multi-column join where heterogeneous key names are used (e.g., `team` and `team_name`), the resulting DataFrame often contains duplicate or redundant key columns. In our `df_joined` DataFrame, the columns `team_name` and `position` essentially replicate the information already present in `team` and `pos` for all matched rows. Maintaining these redundant columns is unnecessary, consumes memory, and can lead to confusion during downstream processing or analysis, especially when working with wide datasets typical in big data applications.

To streamline the dataset, we can use the `.drop()` method, which allows us to remove one or more specified columns from the **DataFrame** efficiently. We target the secondary key columns that were brought over from `df2` (`team_name` and `position`). This step ensures the final data structure is clean, concise, and optimized for subsequent analytical tasks or storage, aligning with best practices for data engineering pipelines.

#drop 'team_name' and 'position' columns from joined DataFrame
df_joined.drop('team_name', 'position').show()

```
+----+---+------+-------+
|team|pos|points|assists|
+----+---+------+-------+
| A| G| 18| 4|
| A| F| 22| 9|
| B| F| 19| 8|
| B| G| 14| null|
+----+---+------+-------+
```

The final output DataFrame, as displayed above, represents the culmination of a successful multi-column **left join** operation, stripped down to its essential components. This refined structure is now ready for aggregation, machine learning feature engineering, or export. The process demonstrates the robust capabilities of **PySpark** in handling complex data integration requirements while maintaining efficient use of resources and data clarity.

Summary and Best Practices for PySpark Joins

Successfully performing joins on **multiple columns** is a fundamental requirement in advanced big data processing using Spark. This tutorial highlighted the specific syntax necessary when joining conditions involve multiple key pairs, especially when column names differ between the two source datasets. By utilizing a list of explicit column equality expressions in the `on` parameter, developers gain precision and clarity in their data manipulation logic, which is crucial for maintaining performance and correctness across distributed systems.

When dealing with large-scale DataFrames, several best practices should be considered. Firstly, always ensure that the key columns used for joining are properly typed and cleaned before the operation, as mismatches can lead to spurious null results. Secondly, be highly explicit about the join type (e.g., `how='left'`, `how='inner'`, `how='right'`) to avoid unexpected data loss or inflation in the result set. Thirdly, if column names are identical across both DataFrames, an alternative, more concise syntax can be used by passing a simple list of column names to the `on` parameter, though the expression syntax shown here is far more versatile and less prone to

ambiguity during complex data pipeline construction, especially in production environments where clarity is paramount.

This structured approach to multi-column joining ensures that data integration is performed accurately, leveraging the full power of the distributed Spark engine.