How to Combine Rows with Matching Values in a PySpark DataFrame

Authored by

stats writer

February 5, 2026

PySpark: Combine Rows with Same Column Values

Introduction to Combining Rows in PySpark

When working with large-scale data processing using the PySpark library, a common requirement is to consolidate records based on shared identifiers. This process, often referred to as grouping and aggregation, is fundamental for summarizing transaction data, calculating metrics per entity, or preparing feature sets for machine learning models. Effectively combining rows that share identical values in one or more columns allows data analysts and engineers to transform raw, granular transactional data into meaningful, aggregated summaries.

In the world of big data, managing duplicated or segmented records belonging to the same entity (such as a customer ID, product number, or employee code) requires robust tooling. DataFrames in PySpark provide the necessary structure and functions to perform these complex operations efficiently across distributed computing clusters. The core mechanism involves identifying the common key and then applying specific aggregation logic to the remaining columns.

This guide will provide a deep dive into the specific PySpark methods required to achieve this consolidation, focusing on the core functions used for grouping and defining the necessary aggregation logic. Mastery of these techniques is essential for anyone aiming to leverage the full power of Apache Spark for data transformation tasks.

The PySpark `groupBy()` and `agg()` Functions

The standard approach for combining rows in a PySpark DataFrame utilizes a powerful combination of two sequential methods: `groupBy()` and `agg()`. The `groupBy()` function partitions the data into groups based on the unique values found within the specified column(s). Once the data is logically grouped, the resulting GroupedData object cannot be directly displayed; it must be followed by an aggregation step.

The `agg()` function is the mechanism used to define how the grouped columns should be summarized. This function accepts various aggregation expressions from `pyspark.sql.functions`, such as `sum`, `avg`, `count`, `max`, `min`, `collect_list`, and others. Crucially, every column that is not used for grouping must have an aggregation function explicitly applied to it.

Selecting the correct aggregation function is vital, as it determines the nature of the resulting summary statistic. For numerical fields like sales or returns, functions like `sum()` or `avg()` are appropriate. For categorical or identifying fields, such as employee names or descriptive labels, functions like `first()`, `last()`, or `collect_set()` are necessary to select a representative value

for the new, combined row.

Syntax for Combining Rows by Column Value

The generalized syntax for combining and aggregating rows in a PySpark DataFrame is straightforward yet flexible. It involves chaining the grouping operation with the specific aggregation definitions, ensuring clear naming conventions are used for the resulting columns.

You can use the following standard structure to combine rows with the same column values in a PySpark DataFrame:

```
from pyspark.sql.functions import*#create new DataFrame by combining rows with same ID values
df_new = df.groupBy('ID').agg(first('employee').alias('employee'),
sum('sales').alias('sum_sales'),
sum('returns').alias('sum_returns'))
```

In this powerful example, the DataFrame is partitioned based on the unique values present in the **ID** column using `groupBy('ID')`. Following the grouping, the `agg()` function is invoked to define the summary logic for the remaining columns. We calculate the sum of values for both the **sales** and **returns** columns. Furthermore, we use the `alias()` function to rename the resulting aggregated columns, promoting clarity in the output schema.

For the non-numeric column, **employee**, we use the `first()` aggregation function. Since all rows belonging to the same ID typically share the same employee name, `first()` efficiently selects the name found in the first record of that group, thus preserving the necessary identifying information while consolidating the numerical data.

Detailed Practical Example Setup

To illustrate this grouping process in a practical setting, let us first establish a sample PySpark DataFrame. This DataFrame simulates transactional records containing sales and returns data attributed to specific employees, identified by a unique **ID**. Notice that several IDs, such as 101 and 103, have multiple corresponding records, which is typical for transactional data requiring consolidation.

We start by importing the necessary `SparkSession` and defining the data structure:

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

```
#define data
data = ,
,
,
,
,
]

#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+---+--------+-----+-------+
| ID|employee|sales|returns|
+---+--------+-----+-------+
|101| Dan| 4| 1|
|101| Dan| 1| 2|
|102| Ken| 3| 2|
|103| Rick| 2| 1|
|103| Rick| 5| 3|
|103| Rick| 3| 2|
+---+--------+-----+-------+
```


As seen in the initial output, the DataFrame `df` contains six records. We observe that Employee Dan (ID 101) has two entries, and Employee Rick (ID 103) has three entries, while Employee Ken (ID 102) has only one. Our primary objective is to collapse these multiple records into a single summary row per employee ID, aggregating the associated sales and returns figures.

This preparation step is crucial, as it clearly defines the input structure and the aggregation keys (**ID**). We are seeking to answer the question: What are the total sales and total returns associated with each unique employee identifier?

Executing the Grouping and Aggregation

Now that the data is prepared, we can apply the core PySpark logic using the `groupBy()` and `agg()` functions to achieve the desired row combination and summation. We import all necessary

functions and then execute the transformation on our DataFrame `df`.

We use the following syntax to group by **ID** and calculate the aggregated sums:

from pyspark.sql.functions import*#create new DataFrame by combining rows with same ID values
df_new = df.groupBy('ID').agg(first('employee').alias('employee'),
sum('sales').alias('sum_sales'),
sum('returns').alias('sum_returns'))

#view new DataFrame
df_new.show()

```
+---+--------+---------+-----------+
| ID|employee|sum_sales|sum_returns|
+---+--------+---------+-----------+
|101| Dan| 5| 3|
|102| Ken| 3| 2|
|103| Rick| 10| 6|
+---+--------+---------+-----------+
```

The resulting DataFrame, `df_new`, successfully consolidates the six original records into three distinct rows, one for each unique **ID**. This transformation effectively summarizes the performance metrics per employee. For instance, for ID 101 (Dan), the aggregated sales (4 + 1) resulted in 5, and the aggregated returns (1 + 2) resulted in 3. Similarly, for ID 103 (Rick), the total sales (2 + 5 + 3) reached 10, and total returns (1 + 3 + 2) reached 6.

Understanding the Aggregation Functions: `first()` and `sum()`

When performing data aggregation, it is essential to understand why specific functions are chosen for different column types. In our example, we used two primary aggregation functions: `sum()` for numerical metrics and `first()` for categorical identifiers.

The `sum()` function is a standard arithmetic operation designed for quantitative columns. Its role is straightforward: it calculates the total cumulative value of all entries within a given group. When applied to **sales** and **returns**, it provides the comprehensive metric of overall activity associated with that employee ID. This is the most common aggregation for transaction data.

The `first()` function, in contrast, is used for columns where we only need a representative value, such as an employee's name, which remains constant across all records for a single ID. It simply returns the value of that column from the first record encountered within the grouping key. This is a

highly efficient way to carry forward non-aggregated identifying attributes into the resulting summary row. If the data quality guarantees that the employee name is consistent for a given ID, `first()` is the ideal choice. Alternatively, one could use `last()` or `min()`/`max()` on strings, though `first()` is generally the most descriptive choice for this purpose.

Enhancing Readability with the `alias()` Function

A crucial element of generating clean and interpretable DataFrames is ensuring that the resulting columns are clearly named. Aggregation functions like `sum('sales')` typically produce a default column name that includes the function name (e.g., `sum(sales)`), which can be verbose and difficult to use in subsequent operations.

To mitigate this, we utilize the `alias()` function immediately after defining the aggregation expression. The `alias()` function allows the user to specify a concise and descriptive name for the output column, such as transforming `sum(sales)` into `sum_sales`.

Note #1: We used the `first` function to return the representative name of the employee associated with a particular ID, ensuring that the identifying text field is present in the summarized result.

Note #2: We used the `alias` function to specify the user-friendly names (e.g., `sum_sales`, `sum_returns`) to use for the columns in the resulting DataFrame, improving data schema clarity.

Key Considerations and Alternatives for Grouping

While grouping by a single column (like **ID**) is the most straightforward approach, PySpark offers flexibility for more complex aggregation scenarios. It is important to understand the capabilities and limitations of the `groupBy()` function, especially when dealing with hierarchical data or multi-key grouping.

For scenarios requiring deeper segmentation, you can group by multiple columns simultaneously. For example, grouping by `groupBy('ID', 'Region')` would create unique summary rows for every combination of employee ID and geographical region, aggregating sales data specific to that pairing. The core syntax of `agg()` remains the same, but the resulting groups are finer-grained.

If the goal is merely to pivot the data without complex aggregation, alternative functions like `pivot()` might be considered, although `groupBy()` remains the standard and most performant method for creating summary statistics. Furthermore, be mindful of the data types when selecting aggregation functions. Using `sum()` on non-numeric columns will result in an error, reinforcing the need to use functions appropriate for the column's data type, such as `first()` or `collect_list()` for strings.

Conclusion and Further PySpark Resources

Combining rows based on common column values is a fundamental data manipulation task in distributed computing environments. By effectively utilizing the `groupBy()` method followed by the versatile `agg()` function, data professionals can transform raw, extensive datasets into concise, aggregated reports ready for analysis. The key to successful aggregation lies in choosing the correct grouping key and applying appropriate aggregation functions for both numerical and categorical data fields.

This technique ensures that the resulting summary data is accurate, meaningful, and optimized for speed and efficiency inherent in the Apache Spark framework. Continued practice with various aggregation functions will deepen proficiency in PySpark data transformation workflows.

The following resources and tutorials explain how to perform other common and advanced tasks in PySpark: