How to Combine Rows with Same Column Values in PySpark Using groupBy() and agg()

Authored by

stats writer

January 2, 2026

Data processing in large-scale environments often requires the consolidation of records based on shared attributes. In the world of PySpark DataFrame manipulation, this process--known as data aggregation--is crucial for transforming raw, granular data into meaningful summaries. Combining multiple rows that share the same value in a specific column is a fundamental operation necessary for reporting, analytical summary generation, and data reduction.

To effectively manage this task within the PySpark DataFrame architecture, developers primarily utilize two powerful methods working in tandem: the groupBy() method and the agg() method. The combination of these two operations facilitates the creation of a new, summarized DataFrame where redundant rows are merged, and chosen metrics (such as sums, counts, minimums, or maximums) are calculated across the newly defined groups.

The resulting structure is inherently smaller than the original source data, offering a concise view. Essentially, for every unique combination of values found within the specified grouping column(s), the aggregation pipeline generates exactly one output row. Mastering this technique is vital for anyone working with distributed data processing using the PySpark DataFrame API.

Understanding the Core Mechanics: `groupBy()` and `agg()`

The efficiency of merging rows in PySpark stems from its optimized execution of the split-apply-combine strategy. This paradigm is handled internally by the distributed nature of Apache Spark, ensuring that even extremely large datasets can be grouped and aggregated rapidly across a cluster of machines. The process begins with the groupBy() method, which acts as the initial grouping mechanism.

When you invoke groupBy(), you specify one or more column names that define the desired groups. PySpark analyzes the DataFrame and conceptually splits the data into distinct sets of rows, where all rows within a set share identical values for the specified grouping key(s). This operation returns a special `GroupedData` object, which is an intermediate state awaiting the application of an aggregation function.

Following the grouping step, the agg() method is applied to the `GroupedData` object. This is where the actual mathematical or statistical operations take place. The agg() method takes one or more aggregation expressions--such as `sum()`, `count()`, `min()`, or `avg()`--and applies them across all rows within each defined group. It then combines the results of these calculations back into a new, consolidated DataFrame, completing the split-apply-combine cycle.

It is important to note the distinction: groupBy() defines the boundaries, while agg() executes the calculations. Without the agg() function, the grouped data remains unprocessed. This modular approach provides immense flexibility, allowing users to define complex aggregation logic using various built-in PySpark SQL functions.

Standard Syntax for Aggregation and Combining Rows

The standard methodology for combining rows based on shared column values follows a clear, two-part structure implemented via method chaining. This syntax requires importing necessary functions from `pyspark.sql.functions` to ensure access to essential aggregation tools like `sum`, `first`, and `alias`.

Below is the foundational syntax used to combine rows in a PySpark DataFrame, followed by a detailed explanation of its components:

The following structure illustrates how to group a DataFrame and apply multiple aggregation functions simultaneously:

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'))

This specific block of code performs several critical actions. First, it identifies all rows where the value in the **ID** column is identical, establishing those groups. Next, it applies three distinct aggregation expressions within the agg() method. Specifically, it uses the sum() function to calculate the total values for both the **sales** and **returns** columns, effectively collapsing multiple transactional records into single summary rows.

Furthermore, it utilizes the first() function on the **employee** column. Since employee names are non-numeric and typically consistent within an ID group, a simple aggregation like `sum()` is inappropriate. The first() function guarantees that one representative value (in this case, the employee's name) is retained for the group. Finally, the use of the alias() function ensures that the resulting columns in the aggregated DataFrame are clearly and descriptively named (e.g., `sum_sales`).

Deep Dive into PySpark Aggregation Functions

PySpark provides a rich library of functions within `pyspark.sql.functions` that are essential for complex data consolidation. Choosing the correct aggregation function is paramount to deriving accurate results from the grouping operation. While `sum()`, `count()`, `min()`, and `max()` are the most commonly used, dealing with non-numeric or identifying columns requires specialized functions.

For columns that hold categorical or identifying information (like names, timestamps, or unique identifiers) which do not require arithmetic aggregation, functions like first() function or `last()` are necessary. These functions select a single representative value from the grouped column. For instance, if grouping sales transactions by a customer ID, you might use `first('customer_name')` to retain the customer's name in the summarized output.

Other powerful aggregation functions include `collect_list()` and `collect_set()`, which are used when you need to retain all individual values from a column within the group, rather than summarizing them. `collect_list()` returns a list containing all values, including duplicates, while `collect_set()` returns a list containing only unique values. This is invaluable when transforming wide data structures into nested, complex types, such as arrays or structs.

Furthermore, managing null values during aggregation is critical. Most standard aggregation functions (like sum() function or `avg()`) automatically skip nulls. However, specific requirements might necessitate handling nulls explicitly, perhaps by using functions that count non-null values (`count()`) or by employing window functions in more complex scenarios where cumulative totals are needed alongside grouping.

Practical Example Setup: Creating the Source DataFrame

To fully demonstrate the row combination process, consider a scenario involving sales data for various employees, where individual transactions are logged daily. Our goal is to consolidate these daily records into a summary view showing the total sales and returns for each employee ID. First, we must initialize a SparkSession and create the initial, granular DataFrame.

We begin by defining the data structure, which includes employee ID, name, sales amount, and returns amount. Note that certain IDs, such as `101` and `103`, appear multiple times, reflecting multiple transactions that need to be merged.

The following code snippet sets up the necessary environment and creates the source PySpark DataFrame:

Example: Combine Rows with Same Column Values in PySpark

Suppose we have the following PySpark DataFrame that contains detailed transactional information about sales and returns for several salesmen within an organization:

```
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 evident in the output, the source DataFrame `df` contains multiple entries for employees Dan (ID 101) and Rick (ID 103). Our objective is to reduce this table such that only one row exists for each unique **ID**, with the sales and returns figures aggregated accordingly. This transformation is necessary for producing summary reports that measure performance at the employee level rather than the transactional level.

Executing the Aggregation Logic

Now that the source data is defined, the next step involves applying the transformation logic. We aim to combine all rows sharing the same value in the **ID** column and subsequently aggregate the values in the remaining relevant columns. This utilizes the powerful combination of groupBy() and agg(), alongside carefully selected aggregation functions.

We specifically need to preserve the employee name while calculating the total sum of sales and

the total sum of returns. To ensure the employee name is accurately represented in the resulting row, we use the first() function. For the numerical columns, the sum() function is deployed to accumulate the values. Finally, the alias() function provides clarity by renaming the aggregated output columns to `sum_sales` and `sum_returns`, adhering to best practices for data presentation.

We can use the following concise syntax to execute the grouping and aggregation:

```
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|
+---+--------+---------+-----------+
```

Analyzing the Aggregated Result

The output DataFrame, `df_new`, successfully achieves the objective of consolidating the transactional data. By comparing the output to the original dataset, we can observe the following critical transformations, confirming the successful operation of groupBy() and agg().

For ID 101 (Dan), the original sales figures (4 and 1) are combined to produce a `sum_sales` of 5, and returns (1 and 2) are combined to yield a `sum_returns` of 3. For ID 103 (Rick), the three sales entries (2, 5, and 3) total 10, and the returns (1, 3, and 2) total 6. The employee Ken (ID 102) had only one entry, which remains unchanged in its aggregated form, demonstrating that grouping works correctly even for unique keys.

This resulting PySpark DataFrame now represents a clear, summarized view of the employee performance data, making it immediately ready for downstream reporting or further analytical processing. It preserves the core identifying column (ID), retains a representative value for the employee name, and provides the necessary summarized metrics.

Advanced Considerations: Using `first()` and `alias()`

Two functions used in the example--first() function and alias() function--warrant closer examination due to their importance in crafting clean and complete aggregated datasets.

The necessity of the first() function arises because, during a groupBy() operation, any column that is not part of the grouping key must be subjected to an aggregation. If we had simply omitted `first('employee')`, the resulting DataFrame would lack the employee name entirely. Since, in this scenario, all rows within the same ID group share the same employee name, selecting the first encountered name is a safe and effective way to include this descriptive information in the summary row. Alternative functions, such as `max()` or `min()`, could technically be used on strings, but `first()` more clearly communicates the intent to retain a representative non-aggregated value.

The alias() function is purely cosmetic but crucial for maintainability and readability. When an aggregation function like sum() function is applied, the resulting column name defaults to a concatenation of the function name and the source column (e.g., `sum(sales)`). By using `.alias('sum_sales')`, we override this cumbersome default name with a concise and professional column label, making the output DataFrame immediately understandable to analysts and consumers of the data.

In summary, these functions ensure that the final DataFrame is not only mathematically sound but also structurally complete and highly readable. Ignoring these quality-of-life additions can lead to ambiguous or incomplete result sets.

Summary and Best Practices for PySpark Aggregation

Combining rows with identical column values in PySpark is a robust and efficient process centered around the use of the groupBy() and agg() methods. This technique is indispensable for generating summarized views from large, distributed datasets. Adherence to best practices ensures optimal performance and accurate results.

Key takeaways for successful PySpark aggregation include:

Define Keys Clearly: Ensure the grouping key(s) passed to groupBy() accurately represent the desired granularity of the summary output.
Use Appropriate Functions: Always match the aggregation function to the data type and analysis requirement (e.g., use sum() function for totals, `avg()` for means, and first() function for representative non-numeric values).
Rename Columns: Utilize the alias() function extensively to create descriptive column names, enhancing the usability of the final DataFrame.

By following these guidelines and leveraging the power of PySpark DataFrame API, data engineers can reliably transform complex, transactional data into clean, concise summary reports necessary for business intelligence and decision-making processes.

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