Learning PySpark: How to Combine Rows in a DataFrame by Grouping on Column Values

Authored by

Mohammed looti

November 11, 2025

Mastering Data Aggregation in PySpark

In the realm of large-scale data processing, efficiently combining and summarizing data is a fundamental requirement. When working with PySpark DataFrames, analysts frequently encounter scenarios where multiple rows pertain to the same entity, necessitating an operation to consolidate these records. This process, known as aggregation, is critical for tasks ranging from calculating total sales per employee to determining average metrics across specific organizational units. Apache Spark, through its Python API, provides highly optimized functions to handle these complex transformations efficiently across distributed clusters.

The primary mechanism for grouping and aggregating data in PySpark DataFrames is the combination of the `groupBy` and `agg` functions. The `groupBy` method partitions the DataFrame based on the unique values found in one or more specified columns, preparing the data for subsequent aggregation. The `agg` function then applies one or more aggregation functions (like sum, count, average, or first) to the resulting groups, producing a single summary row for each unique group key. Understanding how to properly chain these operations is essential for unlocking the analytical power of Spark.

The following syntax illustrates the standard approach used to combine rows that share identical values within a designated column. This pattern allows for explicit control over how each non-grouped column is summarized, ensuring that the resulting PySpark DataFrame is meaningful and aggregated according to specific business logic.

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 specific example, the transformation successfully groups the rows in the source PySpark DataFrame using the unique values present in the **ID** column. Following the grouping, the calculation aggregates the data by applying the `first` function to the **employee** column and the `sum` function to both the **sales** and **returns** columns. This methodology ensures that we retain the employee name (by taking the first occurrence) while accurately calculating the total transaction values associated with that unique **ID**.

The Role of `groupBy` and `agg` in Distributed Computing

Data aggregation is significantly more challenging in a distributed environment like Apache Spark

compared to single-machine tools like Pandas. When a PySpark DataFrame is distributed across multiple nodes in a cluster, the `groupBy` operation initiates a critical step known as the **shuffle**. The shuffle mechanism is responsible for physically moving data between the nodes so that all records belonging to the same group key (in our case, the same **ID**) are brought together onto the same partition or worker node. This crucial step is necessary before the aggregation functions, specified in the `agg` clause, can compute the final summary statistics accurately.

The efficiency of the grouping operation is paramount, as shuffles are typically resource-intensive due to network latency and disk I/O. Therefore, when defining the grouping columns using `groupBy`, it is beneficial to select columns that yield a balanced number of groups. Once the data is co-located, the `agg` function applies the specified logic--whether calculating a **sum**, **average**, **count**, or other statistical metrics--to produce the final, consolidated output. The separation of grouping and aggregation allows Spark to optimize the execution plan for maximum performance across the cluster.

It is vital to recognize that after using `groupBy`, the resulting object is a **GroupedData** object, not a standard DataFrame. This object must be followed by an aggregation function (like `agg`, `count()`, or `sum()`) to transform it back into a usable PySpark DataFrame. The flexibility of the `agg` function, which accepts multiple column-function pairs, allows for complex, multi-faceted summary calculations in a single, clean line of code, significantly streamlining data analysis workflows.

Practical Implementation Example: Combining Sales Records

To solidify our understanding, let us walk through a complete, runnable example. Suppose we are tasked with analyzing sales performance data where individual transactions are logged, resulting in multiple entries for the same employee or sales unit ID. Our objective is to consolidate these records to obtain the total sales and returns associated with each unique employee identifier. The following steps demonstrate the creation of the initial DataFrame and the application of the aggregation logic.

We begin by initializing a SparkSession and defining the raw data structure. This preliminary step ensures that the environment is set up correctly and that we have a tangible dataset to manipulate. The dataset includes four columns: **ID**, **employee**, **sales**, and **returns**. Note how the IDs 101 and 103 appear multiple times, necessitating the consolidation process.

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

With the initial DataFrame prepared, the next step involves applying the core transformation logic. Our objective is to consolidate all rows sharing the same **ID** value and then aggregate the quantitative fields (sales and returns). The crucial decision here is determining how to handle the non-numeric column, **employee**, since direct summation is not possible. For categorical or identifying columns like names, we often use functions such as `first`, `last`, or `collect_set` to ensure the resulting aggregated row retains the necessary identity information.

Applying the Aggregation Syntax

We now apply the `groupBy` operation, specifying the **ID** column as the key for consolidation. Immediately following this, the `agg` function is invoked to specify the precise aggregation logic for each column we wish to retain in the final output. This chaining of commands is the standard pattern for complex data transformation in Apache Spark.

The following code snippet executes the aggregation, yielding a new DataFrame, `df_new`, where each row represents a unique employee ID and contains the summarized metrics. Notice the

necessity of importing functions from `pyspark.sql.functions` to use **`first`** and **`sum`** directly within the **`agg`** call.

```
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 aggregates the data. For **ID 101**, the original sales records (4 and 1) are summed to 5, and returns (1 and 2) are summed to 3. Similarly, for **ID 103**, the sales records (2, 5, and 3) total 10, and returns (1, 3, and 2) total 6. This consolidated view provides the high-level summary needed for performance analysis.

Dissecting PySpark Aggregation Functions: `first`, `sum`, and `alias`

The power of this operation lies in the specific aggregation functions used within the **`agg`** clause. When grouping data, every column that is not part of the **`groupBy`** key must be accounted for using an aggregation function. Two key functions used here are **`first`** and **`sum`**, complemented by the utility function **`alias`** for clarity.

The `first` Function: Since the **employee** name associated with a specific **ID** (like 'Dan' for 101) is constant across all grouped rows, we used the **`first`** function. This function simply returns the value of the specified column from the first row encountered within that particular group. This is a common pattern for retaining identifying information that doesn't require mathematical aggregation.

The `sum` Function: The **`sum`** function is a standard mathematical aggregator, calculating the total value of all entries in the specified column for each group. It is the ideal choice here for consolidating transactional volumes like **sales** and **returns**. PySpark offers many other

aggregation functions, such as `avg`, `min`, `max`, and `count`, allowing for diverse analytical requirements.

Finally, the **`alias`** function is critically important for maintaining readability and clarity in the resulting DataFrame. After an aggregation function is applied, the output column often inherits a complex name (e.g., `sum(sales)`). By chaining **`.alias('new_name')`** to the end of the function call (e.g., `sum('sales').alias('sum_sales')`), we specify clean, descriptive column names for the final output, making the resulting DataFrame immediately usable for downstream analysis or reporting tools.

Additional Resources for PySpark Data Wrangling

Mastering data aggregation is just one step in becoming proficient with Apache Spark and PySpark DataFrames. For those looking to explore further capabilities, the official documentation and community resources offer extensive guidance on complex window functions, joins, and optimization techniques.

The following tutorials explain how to perform other common tasks in PySpark, building upon the foundational knowledge of grouping and aggregation demonstrated here:

Understanding Window Functions for advanced calculations.
Performing efficient Joins between multiple DataFrames.
Optimizing Spark configuration settings for large-scale shuffles.

Continued exploration of PySpark's SQL and DataFrame API is highly recommended for any data engineer or scientist working in a big data environment.