Learning PySpark: Grouping and Aggregating Data Across Multiple Columns

Authored by

Mohammed looti

November 11, 2025

Introduction to PySpark GroupBy and Aggregation

When working with large datasets, the ability to summarize and analyze data based on specific categories is fundamental. In PySpark, the Python API for Apache Spark, this crucial operation is handled efficiently through the combination of the `groupBy()` and `agg()` methods. While `groupBy()` partitions the data based on the unique values in one or more grouping columns, the `agg()` function then applies one or more statistical aggregation functions--such as sum, average, minimum, maximum, or count--to the resulting grouped data. This powerful combination allows users to transition raw, row-level data into meaningful summary statistics, which is essential for reporting and analytical processing within the distributed computing framework of Spark.

The true power emerges when we need to calculate multiple diverse metrics simultaneously across different columns for each defined group. For instance, an analyst might need to find the total sales, the average transaction value, and the count of unique customers, all grouped by region. Attempting to calculate these metrics separately would require multiple passes over the DataFrame, leading to performance bottlenecks. Utilizing `groupBy()` followed by a single, comprehensive `agg()` call optimizes this process, leveraging Spark's inherent ability to perform these operations in parallel across the cluster. Understanding the precise syntax for applying multi-column aggregations is key to writing clean, efficient, and scalable data pipelines in PySpark.

The structured query language (SQL) paradigm heavily influences PySpark's DataFrame API. The `groupBy` operation is analogous to the `GROUP BY` clause in SQL, and the `agg` operation handles the various functions defined in the `SELECT` statement. Mastering this pattern ensures data consistency and operational efficiency, especially when dealing with petabytes of data. The following section introduces the specific Python syntax required to execute a sophisticated aggregation workflow that targets multiple columns simultaneously, demonstrating how to rename the resulting aggregated columns for clarity and accessibility in the final output DataFrame.

Understanding the Syntax for Multi-Column Aggregation

To effectively group and aggregate data on multiple metrics within a DataFrame, PySpark provides a streamlined syntax. The general approach involves chaining the `groupBy()` method, specifying the grouping column(s), and then calling the `agg()` method, passing a series of aggregation functions imported from pyspark.sql.functions. The use of `.alias()` is highly recommended within the `agg()` function to assign descriptive, unique names to the newly created aggregated columns, preventing confusion and enhancing code readability.

The standard syntax for performing multi-column aggregation is demonstrated in the code block below. This particular example showcases how to group rows based on the `team` column while simultaneously calculating three distinct metrics: the sum of `points`, the mean (average) of

`points`, and the count of `assists`. Note how the column names within the aggregation functions (e.g., `'points'`, `'assists'`) are passed as string literals, and the final output columns (e.g., `'sum_pts'`, `'mean_pts'`, `'count_ast'`) are defined using the `.alias()` method.

from pyspark.sql.functions import *

#group by team column and aggregate using multiple columns
df.groupBy(df.team.alias('team')).agg(sum('points').alias('sum_pts'),
mean('points').alias('mean_pts'),
count('assists').alias('count_ast')).show()

This specific command executes a series of parallel computations across the grouped data. It first identifies all unique values in the `team` column, and for each unique team, it applies the specified functions. The resulting structure is a much narrower DataFrame where each row represents a unique team and the subsequent columns contain the calculated summary statistics. Specifically, the aggregations performed are:

Calculates the `sum` of the **points** column, naming the resultant column **sum_pts**.
Calculates the `mean` (average) of the **points** column, naming the resultant column **mean_pts**.
Calculates the `count` of non-null values in the **assists** column, naming the resultant column **count_ast**.

Setting Up the Example PySpark DataFrame

To illustrate the practical application of the multi-column aggregation syntax, we will construct a sample PySpark DataFrame containing hypothetical basketball player statistics. This dataset includes information on the player's team, their position, the points they scored, and the number of assists they achieved. This structure mimics real-world scenarios where data analysts frequently need to summarize performance metrics across different categorical groups, such as teams or departments.

The creation process involves initiating a `SparkSession`--the entry point for using Spark functionality--defining the raw data as a list of lists, and specifying the column schema. The data includes records for three teams (A, B, and C) and two positions (Guard and Forward). The subsequent code block details the necessary PySpark setup steps required to transform this raw Python data structure into a distributed, schema-enforced Spark DataFrame, ready for complex analytical operations. This foundational step is critical before any data manipulation, including grouping and aggregation, can be performed.

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()

+----+--------+------+-------+
|team|position|points|assists|
+----+--------+------+-------+
| A| Guard| 11| 5|
| A| Guard| 8| 4|
| A| Forward| 22| 3|
| A| Forward| 22| 6|
| B| Guard| 14| 3|
| B| Guard| 14| 5|
| B| Forward| 13| 7|
| B| Forward| 14| 8|
| C| Forward| 23| 2|
| C| Guard| 30| 5|
+----+--------+------+-------+
```

This initial DataFrame, `df`, provides the raw data points for ten individual player performances. Our objective now is to use the `groupBy()` and `agg()` methods to condense these ten records into

three summary rows--one for each team--calculating the total points scored, the average points per player, and the total number of assist records available for each team. This transformation simplifies the data structure significantly, making team-level comparisons straightforward and efficient.

Executing the GroupBy and Aggregation Operation

With the PySpark DataFrame successfully initialized, we can proceed to apply the multi-column aggregation technique. We aim to group all rows belonging to the same team and then calculate the essential performance metrics for that team across the `points` and `assists` columns. This single operation encapsulates what would typically require multiple SQL queries or iterative loops in less optimized environments, showcasing the efficiency of the PySpark DataFrame API.

We import all necessary functions from pyspark.sql.functions (using `import *` for brevity in this example) to access standard statistical operations like `sum()`, `mean()`, and `count()`. The resulting query uses the `groupBy(df.team)` method to segment the data and the `agg()` method to define the specific calculations applied within each segment. This is the culmination of the data preparation, translating the analytical requirement--"What are the combined metrics per team?"--into executable, distributed code.

from pyspark.sql.functions import *

#group by team column and aggregate using multiple columns
df.groupBy(df.team.alias('team')).agg(sum('points').alias('sum_pts'),
mean('points').alias('mean_pts'),
count('assists').alias('count_ast')).show()

+----+-------+--------+---------+
|team|sum_pts|mean_pts|count_ast|
+----+-------+--------+---------+
A	63	15.75	4
B	55	13.75	4
C	53	26.5	2
+----+-------+--------+---------+

The output clearly demonstrates the successful application of the multi-column aggregation. The original ten rows have been collapsed into three summary rows, one for each unique team identifier (A, B, and C). The columns `sum_pts`, `mean_pts`, and `count_ast` contain the derived statistics, providing a concise performance overview of each team. This efficient data reduction is central to big data analysis, allowing analysts to focus on high-level trends rather than individual

data points.

Interpreting the Results

The resulting DataFrame, which contains the aggregated metrics, is the primary output of this operation. It effectively summarizes the entire dataset based on the groupings defined by the `team` column. Interpreting this summary is straightforward, as the new column names (`sum_pts`, `mean_pts`, `count_ast`) clearly define the calculated metric for the corresponding team. This structure facilitates immediate comparative analysis between groups.

For instance, examining the row corresponding to Team A, we can quickly extract its overall performance based on the input data. The resulting DataFrame shows the sum of the points values, the mean of the points values, and the count of assists values for each team. This allows for immediate, clear data interpretation without the need to manually calculate or re-sort the initial dataset.

The sum of points for team A is **63**, meaning all players on Team A combined scored 63 points in the recorded performances.
The mean of points for team A is **15.75**, indicating the average points scored per recorded performance by a Team A player.
The count of assists for team A is **4**, which tells us that there were four recorded instances of player performance data for Team A in the original dataset (or four non-null assist values).

Similarly, we observe that Team C, while having the highest average points (26.5), only has a `count_ast` of **2**, suggesting that Team C had fewer recorded player performances than Teams A and B in the original dataset. These quick insights demonstrate the utility of calculating multiple aggregate statistics in a single, consolidated view, providing immediate context regarding volume (count) versus magnitude (sum/mean).

Advanced Considerations for PySpark Aggregation

While the example focuses on a single grouping column (`team`), the `groupBy()` method in PySpark can accept multiple columns to define more granular groups. For example, grouping by `df.team` and `df.position` would yield summary statistics for 'Team A - Guard', 'Team A - Forward', and so on. This hierarchical grouping is essential for drilling down into specific sub-categories within the data. Furthermore, users are not limited to standard functions like `sum` and `mean`; the pyspark.sql.functions module provides numerous specialized functions, including `stddev()`, `min()`, `max()`, and percentile functions, allowing for complex statistical analysis.

A crucial best practice when using `groupBy()` and `agg()` is managing the computational cost associated with shuffling. Grouping data requires Spark to shuffle data across the network to

ensure that all records belonging to the same group land on the same worker node. For very large DataFrames, minimizing the number of distinct groups or utilizing efficient partitioning strategies can drastically reduce shuffle overhead, thereby improving performance. Developers should strive to use efficient data types for grouping columns (e.g., integers instead of complex strings) and be mindful of data skew, where certain groups might dominate the dataset, leading to workload imbalance.

Finally, the combination of `groupBy().agg()` can often be replaced by the more concise `rollup()` or `cube()` operations when dealing with reporting requirements that involve calculating subtotals and grand totals across different dimensions. While `groupBy().agg()` provides the base group-level results, `rollup()` generates results for the specified grouping columns plus all possible intermediate subtotals, and `cube()` generates results for all possible combinations of the grouping columns, including the grand total. Choosing the correct aggregation pattern based on the desired output structure is a hallmark of efficient PySpark development.

Additional Resources for PySpark Mastery

The following tutorials explain how to perform other common and advanced tasks in PySpark, building upon the foundational knowledge of `groupBy` and `agg` operations. Mastering these techniques is fundamental for any data engineer or analyst utilizing the power of distributed computing offered by Apache Spark.

Further exploration into the pyspark.sql.functions module will reveal powerful tools for cleaning, transforming, and analyzing data at scale.