How can PySpark be used to perform Groupby Agg on Multiple Columns?

Authored by

stats writer

January 20, 2026

The Power of PySpark for Distributed Data Analysis

PySpark, the Python API for the powerful distributed computing system Apache Spark, has become indispensable in the realm of big data processing. It allows data professionals to harness the speed and scalability of Spark using familiar Python syntax. At the heart of data preparation and analysis lies the need to summarize large volumes of transactional data into meaningful metrics, a process often accomplished through grouping and subsequent statistical aggregation. This capability is paramount for tasks ranging from operational reporting to advanced machine learning feature engineering, requiring the system to efficiently handle massive datasets spread across numerous nodes in a cluster. The ability of PySpark to execute a **GroupBy Aggregation** across multiple columns is a fundamental feature that distinguishes it as a leading tool in modern data science stacks, ensuring that complex summary statistics can be derived quickly and reliably, regardless of data volume.

When working with a DataFrame in PySpark, data is often partitioned based on key categorical variables. For instance, in sales data, one might need to group by `region` and `product_category` simultaneously. Once these distinct groups are formed, aggregation functions are applied to calculate collective metrics--such as the total sales amount (sum), the average transaction size (mean), or the total number of records (count)--for each unique group combination. This multi-column grouping is crucial for layered analysis, providing deeper insights than simple single-column summaries. PySpark's optimized internal execution engine ensures that this complex operation is performed with minimal latency, distributing the computational load effectively across the cluster resources, thereby maintaining high performance even when dealing with petabytes of information.

The syntax used to achieve this powerful functionality is both concise and highly expressive, leveraging the rich set of built-in functions available within the `pyspark.sql.functions` module. Understanding the proper structure for chaining the `groupBy()` and `agg()` methods is the key to unlocking advanced data summarization capabilities. This structure mandates specifying the grouping columns first, followed by a dictionary or a list of aggregation functions applied to specific target columns. Mastery of this pattern allows analysts to transform granular data into actionable summary tables, forming the basis for business intelligence reports and strategic decision-making processes.

Core Syntax for Multi-Column GroupBy Aggregation

To effectively group data based on defined criteria and simultaneously apply multiple aggregation functions to different columns, PySpark provides a streamlined method chain. This specific sequence ensures that the grouping logic is executed first, followed immediately by the calculation of the desired statistics. The general methodology involves selecting the grouping columns, and

then explicitly defining which aggregation function (e.g., `sum`, `mean`, `count`) should be applied to which value column, along with assigning a new, descriptive alias for the resulting summary column.

The following canonical syntax demonstrates how to group a PySpark DataFrame and perform several different aggregation tasks concurrently. Note the use of the `.alias()` function, which is critical for renaming the resulting aggregated columns to ensure clarity and usability in subsequent analysis steps. Without proper aliases, PySpark would generate default column names that might be verbose or confusing.

You can use the following syntax to group by and perform aggregations on multiple columns in a PySpark DataFrame:

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 particular example groups the rows of the DataFrame by the **team** column, resulting in a single row for every unique team identifier. It then performs three distinct aggregations simultaneously, summarizing the quantitative columns related to scoring and playmaking:

Calculates the **sum** of the **points** column across all records within that team group, naming the resulting column **sum_pts**.
Calculates the **mean** (average) of the **points** column for that group, labeling it **mean_pts**.
Calculates the **count** of non-null values in the **assists** column for that group, resulting in the column **count_ast**.

The effective use of the `agg()` function allows for this comprehensive, parallel calculation of diverse metrics, providing a powerful, summarized view of the underlying raw data grouped by the specified category.

Setting Up the PySpark Environment

Before any data manipulation can occur, it is necessary to initialize the Spark context and create a SparkSession, which serves as the entry point to programming Spark with the DataFrame API. The SparkSession is crucial as it manages the connection to the underlying Spark infrastructure and allows for the creation, registration, and manipulation of DataFrames. Establishing this session is the first, mandatory step in any PySpark script designed for data processing, ensuring that the

necessary distributed resources are allocated and ready for computation.

For our practical example, we must first define the raw data structure representing basketball player statistics. This structured data, often sourced from external files like CSVs or Parquet, is then converted into a PySpark DataFrame. This conversion step transforms the static data into a highly optimized, distributed collection of rows organized into named columns, ready for high-performance operations like GroupBy Aggregation. Defining clear column names (or schemas) ensures that the subsequent queries and aggregations are applied correctly and logically.

The following code block demonstrates the complete setup process, including importing the necessary `SparkSession` class, initializing the session, defining the sample data, specifying column headers, and finally constructing and displaying the initial DataFrame structure, which serves as the foundation for our analysis.

Practical Demonstration: Creating the Sample DataFrame

To illustrate how to use this syntax in practice, we will assume the existence of a PySpark DataFrame containing detailed information about various basketball players. This dataset includes metrics such as which **team** they belong to, their **position**, the **points** they scored, and the number of **assists** they recorded. This type of structured data is highly amenable to GroupBy operations, allowing us to summarize player performance at the team level.

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

The resulting DataFrame, `df`, provides the raw inputs necessary for calculating team-level performance metrics. Notice that Team A and Team B each have four recorded player entries, while Team C has only two. Our goal is to consolidate these individual player statistics into a single, summarized row for each of the three distinct teams (A, B, and C), using the GroupBy Aggregation operation. This process will effectively reduce the number of rows while simultaneously enriching the data with aggregated insights.

Executing the Multi-Column Aggregation

We now apply the core PySpark logic to transform our detailed player data into summarized team statistics. The operation involves calling the `groupBy()` method on the `df` DataFrame, specifying the **team** column as the grouping key. This action logically segments the dataset. Following this, the `agg()` method is chained to calculate the sum and mean of the **points**, and the total count of **assists** for every single team group identified in the initial step.

This step is fundamental because it moves beyond simple filtering and ordering; it performs a structural reduction and analytical calculation simultaneously across the distributed system. By requesting the sum and mean of points, we gain insight into both the total offensive production and the average scoring capability per player within that team. The count of assists, when aggregated, provides an accurate measure of the total recorded playmaking actions linked to each team,

serving as a powerful indicator of data completeness or activity level.

We use the following syntax to group the rows by the values in the **team** column and then calculate several aggregate metrics, demonstrating the efficiency and clarity of the PySpark DataFrame API:

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
+----+-------+--------+---------+

Interpreting the Results of the GroupBy Operation

The output of the aggregation is a new PySpark DataFrame that contains only three rows, one for each unique team (A, B, and C). This resultant DataFrame effectively summarizes the entire input dataset based on our chosen metrics: `sum_pts`, `mean_pts`, and `count_ast`. Analyzing this output allows us to quickly compare the performance profiles of the three teams based on these core statistics, moving from detailed row-level data to crucial, high-level summaries.

For instance, Team A shows the highest overall total points (63), indicating high combined offensive output across its players. However, Team C, despite having fewer players (indicated by `count_ast` = 2), exhibits the highest average score per player (26.5). Team B presents a balanced performance profile, positioned between A and C in terms of total points and mean points. Such comparative analysis would be cumbersome to perform on the raw, unaggregated data, highlighting the true value and efficiency of the **GroupBy Aggregation** technique in PySpark.

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, confirming that the aggregation functions were applied correctly across the grouped data partitions.

For example, a detailed breakdown of the statistics reveals:

The sum of points for team A is **63** (11 + 8 + 22 + 22).
The mean of points for team A is **15.75** (63 / 4 players).
The count of assists for team A is **4**, confirming four records were aggregated for this team.

Similarly, for Team C, the sum of points is 53 (23 + 30), and since there are only 2 records, the mean points is 26.5 (53 / 2). These calculations validate that the aggregation has been applied correctly to the distinct partitions defined by the team identifier, offering a reliable summary statistic set.

Extending Aggregation: Grouping by Multiple Keys

While the primary example focused on grouping by a single column (`team`), the power of PySpark's GroupBy operation is fully realized when grouping by multiple dimensions. Had we chosen to group by both `team` and `position` (e.g., `df.groupBy('team', 'position')`), the resulting DataFrame would contain unique combinations of these two fields--such as 'A', 'Guard' and 'A', 'Forward'--and the aggregations would be calculated independently for each of these smaller, more granular segments. This technique is often used in business intelligence to drill down into specific segments of performance data.

The primary benefit of grouping by multiple columns lies in creating highly specific analytical cuts. For instance, an analyst might want to know the average assists specifically among guards on Team B, versus the average assists among forwards on Team B. A multi-column GroupBy operation handles this request efficiently by creating the necessary partitions before performing the computational summary. This detailed approach is critical for diagnostics and fine-grained performance monitoring in complex data structures.

Furthermore, PySpark supports a vast array of aggregation functions beyond simple sum and mean. Functions like `stddev()`, `min()`, `max()`, and statistical functions like `corr()` can all be included within the `agg()` clause. This flexibility allows data professionals to perform highly complex statistical analysis in a single, efficient distributed operation, transforming raw data into sophisticated metrics ready for machine learning model input or advanced reporting.

Conclusion: The Efficiency of PySpark GroupBy

The ability to perform robust GroupBy aggregation on multiple columns is a cornerstone of effective data handling in PySpark. This technique not only simplifies complex data summarization tasks but also leverages the distributed nature of Apache Spark to execute these computations at scale, making it suitable for even the largest big data environments. By mastering the `groupBy()` and `agg()` syntax, data scientists can efficiently transform raw, transactional data into insightful, summarized metrics that drive informed decisions.

The demonstrated example, moving from individual player records to consolidated team performance statistics, illustrates a foundational data transformation pattern common across almost all industries, from finance to healthcare. PySpark ensures that this process is executed with maximum performance, consolidating its position as the preferred tool for high-volume, analytical workloads requiring detailed and multi-dimensional aggregation.

For those looking to further enhance their PySpark skills, exploring advanced topics such as window functions--which allow for calculations across related rows without fully collapsing the data--or user-defined aggregation functions (UDAFs) is highly recommended. These tutorials explain how to perform other common tasks in PySpark: