How to Group by Multiple Columns in PySpark and Perform Aggregations

Authored by

stats writer

January 20, 2026

RECOMMENDED CITATION

stats writer (2026). *How to Group by Multiple Columns in PySpark and Perform Aggregations*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=126700

Introduction to PySpark Grouping

Analyzing large datasets often requires partitioning the data based on specific characteristics before performing mathematical calculations. In PySpark, this crucial operation is handled efficiently using the **groupBy** function, a staple tool in any data engineer's toolkit. While grouping by a single column is straightforward, real-world data analysis frequently demands grouping across two or more dimensions simultaneously. This article provides an expert guide on how to effectively utilize the **groupBy** method on multiple columns within a DataFrame, ensuring clean aggregation results.

The ability to group by multiple fields allows for granular analysis, such as calculating total sales aggregated by both region and product category, or, as we will demonstrate, totaling points scored based on both team and player position. Mastering this technique is fundamental for anyone working with large-scale distributed data processing using the Apache Spark engine accessed via its Python API, **PySpark**. We will walk through the exact syntax, provide a clear, practical example, and show how to manage the resulting column names for optimal readability.

The Core Syntax for Multiple Column GroupBy

The syntax for executing a multi-column group-by operation in PySpark is surprisingly intuitive, maintaining consistency with SQL-like operations. It involves chaining the **groupBy** method, followed by the desired aggregation function, such as **sum**, **count**, or **avg**. The key difference when grouping by multiple columns is simply supplying all relevant column names as arguments to the **groupBy** function.

When preparing to group your PySpark DataFrame, remember that the order in which you list the columns within the **groupBy** parentheses does not affect the final result set, only how the rows might be organized internally before the aggregation step. However, for clarity and maintainability, it is generally recommended to list the primary grouping column first.

The following concise code snippet illustrates the fundamental structure required to group by two columns (`team` and `position`) and subsequently calculate the sum of values in a third column (`points`). This serves as the blueprint for nearly all multi-column aggregation tasks in PySpark:

df.groupBy('team', 'position').sum('points').show()

In this powerful one-liner, the operation calculates the sum of the numerical values present in the **points** column, applying this calculation separately for every unique combination found across the **team** and **position** columns within the source DataFrame. This results in a much-condensed DataFrame where each row represents a unique combination of the grouping keys along with the

resulting aggregate value.

Understanding the Aggregation Process

Before diving into the full example, it is essential to appreciate what happens internally during the **groupBy** and subsequent aggregation stages. When you invoke `df.groupBy('col1', 'col2')`, PySpark initiates a shuffle operation across the cluster. This shuffle ensures that all rows sharing the exact same values for `col1` and `col2` are moved to the same partition on the same executor node. This co-location is critical because the aggregation function (e.g., **sum**) must operate on the entire set of relevant records simultaneously.

Once the data is correctly partitioned and co-located, the aggregation function takes over. For instance, using `.sum('points')` tells the executor to iterate through all points values associated with that specific group (e.g., 'Team A' and 'Guard') and calculate their total. If the grouping columns defined a unique row identifier, the aggregation would return the original value, but since we are grouping by categorical variables, the result is a summary metric that collapses many rows into one.

By including multiple column names in the groupBy function, you are effectively creating a compound key. Only records that match across **all specified columns** will be placed into the same group. This high level of specificity allows analysts to generate complex, multi-dimensional summaries of their data, providing deeper insights than simple, single-variable aggregations.

Practical Example: Setting Up the DataFrame

To solidify this concept, let us work through a concrete, runnable example using synthetic data pertaining to basketball player statistics. Our objective is to calculate the total points scored, categorized by both the player's team and their position on the court. This setup requires defining the session, the raw data, and the schema for our initial PySpark DataFrame.

We begin by initializing the Spark session, which is necessary to perform any data manipulation in PySpark. Following this, we define our sample data, which is structured as a list of lists, where each inner list contains the team designation, the position (Guard or Forward), and the points scored in a game. Finally, we map these raw data points to the appropriate columns to construct the DataFrame.

Observe the data structure provided below. It clearly illustrates the need for multi-column grouping, as Team A has multiple Guards and multiple Forwards, each with individual scores. We want to condense these eight rows into four distinct aggregated records.

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|
+----+--------+------+
A	Guard	11
A	Guard	8
A	Forward	22
A	Forward	22
B	Guard	14
B	Guard	14
B	Forward	13
B	Forward	7
+----+--------+------+

Implementing Multi-Column Grouping and Summation

With our source DataFrame successfully loaded, we can now apply the **groupBy** function using the compound key formed by `team` and `position`. This step is where the power of PySpark's distributed processing shines, consolidating the raw individual records into meaningful summarized groups. We will utilize the **sum** aggregation function, as our goal is to find the cumulative score for

each distinct team-position pairing.

The command below demonstrates the precise implementation. Notice how the column names `team` and `position` are passed sequentially within the **groupBy** method, followed immediately by the `.sum('points')` operation. This streamlined syntax efficiently instructs the Spark engine on exactly how the data transformation should be performed.

#calculate sum of points, grouped by team and position
df.groupBy('team', 'position').sum('points').show()

```
+----+--------+-----------+
|team|position|sum(points)|
+----+--------+-----------+
| A| Guard| 19|
| A| Forward| 44|
| B| Guard| 28|
| B| Forward| 20|
+----+--------+-----------+
```

Interpreting the Grouped Results

The resulting DataFrame is significantly smaller and presents a clean, aggregated view of the data. Instead of eight rows detailing individual scores, we now have four rows, each representing a unique combination of team and position. The newly created column, automatically named `sum(points)` by PySpark, holds the aggregated value for that specific group.

By carefully examining the output, we can draw immediate and actionable conclusions regarding the performance metrics summarized in the DataFrame. The grouping effectively isolates the contributions of players based on their role within their respective teams. For instance, comparing the total points of Guards on Team A versus Guards on Team B is now a direct row-by-row comparison, simplifying statistical reporting.

We can break down the results derived from the multi-column grouping operation as follows:

The total score accumulated by all **Guards** on **Team A** is **19**. (Calculated from 11 + 8 in the original data).
The total score accumulated by all **Forwards** on **Team A** is **44**. (Calculated from 22 + 22 in the original data).
The total score accumulated by all **Guards** on **Team B** is **28**. (Calculated from 14 + 14 in the original data).
The total score accumulated by all **Forwards** on **Team B** is **20**. (Calculated from 13 + 7 in the

original data).

Renaming Aggregate Columns Using the Alias Function

A common requirement after performing an aggregation is to rename the resulting column. PySpark's default naming convention (e.g., `sum(points)`) is functional but often lacks professional clarity for reporting or subsequent processing steps. To address this, we use the alias function in conjunction with the more flexible **agg** function instead of the simpler **sum** method.

The **agg** function allows us to specify the aggregation calculation and immediately assign an alias (a new name) to the resulting column. This requires importing the specific aggregation function (like **sum**) from `pyspark.sql.functions`. This approach provides fine-grained control over the output schema, which is vital when building complex data pipelines where column names must adhere to strict naming conventions.

The following code snippet demonstrates how to achieve the exact same aggregation result as before, but this time, the output column will be clearly named `points_sum`, making the DataFrame output much cleaner and easier to use downstream.

from pyspark.sql.functions import sum

#calculate sum of points, grouped by team and position
df.groupBy('team', 'position').agg(sum('points').alias('points_sum')).show()

+----+--------+----------+
|team|position|points_sum|
+----+--------+----------+
A	Guard	19
A	Forward	44
B	Guard	28
B	Forward	20
+----+--------+----------+

As evidenced by the output, the resulting DataFrame maintains the correct aggregated values for each unique combination of team and position. Crucially, the aggregate column is now named **points_sum**, fulfilling the requirement for a descriptive and custom column name specified within the **alias** function. This practice is strongly recommended for production environments.

Exploring Alternative Aggregation Metrics

While our comprehensive example focused on the **sum** function, the multi-column **groupBy**

technique is not limited to simple summation. PySpark offers a wide variety of robust aggregation metrics that can be applied to the grouped data, allowing for diverse analytical tasks. The choice of metric depends entirely on the analytical question being addressed.

It is important to remember that after the **groupBy** operation is established, you can substitute `.sum('column')` with any other appropriate aggregation function. The mechanism for grouping remains identical regardless of whether you are calculating totals, averages, counts, or extreme values.

Some of the most frequently used alternative aggregation metrics available in PySpark include:

count: Calculates the number of items (rows) within each group. This is useful for determining group size.
mean or **avg**: Computes the arithmetic average of the values in the specified column for each group.
max: Finds the maximum value in the specified column within each group.
min: Determines the minimum value in the specified column within each group.
stddev: Calculates the standard deviation of the values in the specified column per group, providing insight into data variability.

Summary of Best Practices

Grouping by multiple columns in a PySpark DataFrame is a powerful and necessary technique for deep data analysis. To ensure optimal performance and code readability, always prioritize using the **agg** function in conjunction with **alias** when the output column name needs customization. Furthermore, be mindful that **groupBy** triggers a data shuffle, which can be computationally expensive; thus, it should be reserved for necessary aggregation tasks.

By following the syntax outlined--passing multiple column names to **groupBy**--and selecting the appropriate aggregation metric (whether **sum**, **count**, or **mean**), you gain the ability to accurately summarize complex datasets, moving beyond simple univariate analysis into sophisticated multi-dimensional reporting.

For users looking to perform other common tasks in PySpark, exploring related tutorials covering data filtering, joining, and window functions is highly recommended to build a comprehensive data manipulation skillset.