Learning PySpark: Calculating Grouped Means in DataFrames

Authored by

Mohammed looti

November 11, 2025

Understanding Grouped Aggregation in PySpark DataFrames

Calculating statistical aggregates across specific subsets of data is an indispensable requirement in modern, large-scale data processing. When dealing with massive datasets distributed across computing clusters, PySpark provides an exceptionally fast and scalable framework for these operations. Specifically, determining the statistical mean, or average value, based on distinct categorical attributes, is crucial for deriving actionable business intelligence and generating meaningful insights. This comprehensive tutorial is dedicated to leveraging the powerful PySpark DataFrame API to execute these grouped calculations efficiently, ensuring clarity and peak performance within your analytical workflows. We will detail two foundational approaches: grouping data based on a single categorical identifier and grouping by a complex combination of multiple columns, offering detailed, practical examples for both methodologies.

The core functionality that enables this powerful aggregation is the `groupBy()` method. This function intelligently organizes rows that share identical values in the specified column(s) into distinct, logical partitions. Once the data is partitioned, a wide range of aggregation functions can be immediately applied, such as `mean()`, `sum()`, `min()`, or `count()`, targeting the numerical columns within those defined groups. Achieving accurate data summarization in a distributed computing environment requires a solid understanding of how to correctly sequence and chain these methods, which is central to the methodology employed by PySpark. The subsequent sections will meticulously detail the required syntax to execute mean calculations effectively using a relatable sample dataset containing hypothetical basketball player statistics.

To successfully compute the average value across groups within a PySpark DataFrame, two key components must first be identified: the grouping key (the categorical column or columns used for partitioning) and the target column (the numerical column upon which the average calculation will be performed). The fundamental structural pattern involves invoking the `groupBy()` method on the source DataFrame, supplying the desired column name(s) as arguments, and subsequently calling the `mean()` method, explicitly naming the column to be averaged. This concise syntax is the bedrock of complex data reduction in Spark.

Method 1: Grouping by a Single Categorical Variable

The simplest and most frequently used technique involves calculating the average of a specific metric based on the unique values present in only one categorical column. Imagine a scenario where you possess comprehensive data detailing points scored by numerous players spread across various teams. A common analytical requirement would be to determine the average points scored for each team individually. This aggregation process efficiently reduces the DataFrame's size, replacing potentially millions of individual detail rows with a single, summarized row corresponding to each unique group identifier. This method provides a clear, high-level summary

view of the data distribution.

 # Calculate the mean of 'points' grouped exclusively by 'team'
df.groupBy('team').mean('points').show()

The provided code snippet perfectly encapsulates this methodology. We begin by invoking `df.groupBy('team')`, which logically segments the entire dataset according to the values found in the 'team' column. Immediately thereafter, we apply the aggregation function `.mean('points')`, instructing PySpark to compute the arithmetic mean across the values in the 'points' column, operating distinctly within each of the defined team groups. The final `.show()` action triggers the computation and displays the resulting aggregated DataFrame. This output will inherently contain one row per unique team and feature a new column that presents the calculated average points for that team.

Method 2: Advanced Grouping by Multiple Columns

Frequently, data analysis demands a much finer level of detail, necessitating the calculation of the mean based on the synergistic combination of values across two or more separate columns. This advanced technique allows for highly specific analyses, such as determining the average points scored by players categorized simultaneously by both their team affiliation and their specific positional role (e.g., 'Team A Guard'). Grouping by multiple columns ensures that a unique group is generated for every distinct permutation of values found across all specified grouping columns, providing highly granular summary statistics.

 # Calculate mean of 'points' grouped by both 'team' and 'position'
df.groupBy('team', 'position').mean('points').show()

In this more complex application, the `groupBy()` function is provided with an ordered sequence of column names--in this case, 'team' and 'position'. PySpark subsequently constructs groups based on the concatenation of these two factors (e.g., only rows where 'Team' is 'A' AND 'Position' is 'Guard' form a single group). The critical subsequent step, `.mean('points')`, computes the average points scored exclusively within these narrower, highly refined groups. This technique is invaluable when analyzing complex, hierarchical data structures or when the goal is to break down performance statistics by several categorical variables concurrently to uncover hidden trends or biases.

Setting Up the PySpark Environment and Sample Data Preparation

To effectively demonstrate these powerful aggregation methods in a practical setting, the initial step requires the initialization of a DataFrame populated with suitable sample data. Our example

utilizes synthesized information about several basketball players, specifically tracking their assigned team, their defined position, and the total points they scored. Establishing a **SparkSession** is the absolute foundational prerequisite for initiating any and all PySpark operations, as it serves as the unified entry point for interacting with the underlying Spark cluster and its distributed computing capabilities.

The code block below meticulously handles all necessary setup procedures. This includes importing required libraries, initializing the Spark context via the session object, defining the raw dataset (structured as a Python list of lists), specifying the precise column schema, and ultimately constructing the functional DataFrame instance. It is absolutely essential to review the structure and content of this initial DataFrame before proceeding, as it establishes the definitive baseline data for all subsequent, critical aggregation examples.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define the raw dataset structure
data = ,
,
,
,
,
,
,
,
]

Define the column schema
columns =

# Create the PySpark DataFrame
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure
df.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|

A	Guard	8
A	Forward	22
A	Forward	22
B	Guard	14
B	Guard	14
B	Guard	13
B	Forward	7
C	Guard	8
C	Forward	5
+----+--------+------+

Practical Example 1: Calculating Average Points per Team

This section applies the first grouping method discussed, focusing exclusively on determining the aggregate average points achieved by players belonging to each unique team identifier (A, B, and C) within our dataset. We execute this by utilizing the **`groupBy()`** method, specifically targeting the **team** column as the key, and then instructing PySpark to compute the mean of the numerical **points** column. This process yields a clear, high-level overview of the overall offensive performance distributed across the various teams represented in the sample data.

The syntax required for this calculation is remarkably concise and highly readable, a hallmark of the expressive, functional programming interface provided by PySpark's API. By executing this singular, chained command, we effectively harness Spark's distributed processing power, ensuring the averages are computed rapidly and efficiently, regardless of the immense size of the underlying dataset. The resulting aggregated DataFrame structure clearly presents the team identifier alongside its corresponding calculated average score.

 # Calculate mean of 'points' grouped by 'team'
df.groupBy('team').mean('points').show()

+----+-----------+
|team|avg(points)|
+----+-----------+
A	15.75
B	12.0
C	6.5
+----+-----------+

From the resulting output DataFrame, which automatically names the newly aggregated column as

`avg(points)`, we can draw immediate conclusions regarding performance metrics for each group:

The average points value for players on **team A** is **15.75**, signifying the highest overall offensive output among the three competing teams.
The average points value for players on **team B** is exactly **12.0**, positioning them squarely in the intermediate performance range.
The average points value for players on **team C** is **6.5**, demonstrating the lowest average performance in this particular statistical metric.

Practical Example 2: Granular Averages by Team and Position

To achieve a more profound understanding of the data--specifically, the source and distribution of the points scored--we now implement the second methodology: grouping simultaneously by both the **team** column and the **position** column. This dual-criteria refinement allows us to isolate and compare the performance metrics of specific player roles within the context of their respective teams. The number of resulting rows in the aggregated DataFrame will correspond to the total number of unique team-position combinations present in the original data.

This sophisticated grouping approach proves indispensable for complex analytical tasks where controlling for multiple categorical variables is essential, perhaps to mitigate the risks associated with phenomena like Simpson's Paradox or to ensure accurate, apples-to-apples comparisons across distinct sub-populations. By supplying both column names into the **groupBy()** method, we guarantee that the mean calculation is executed only on those rows that precisely satisfy both team and position criteria.

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

```
+----+--------+------------------+
|team|position| avg(points)|
+----+--------+------------------+
| A| Guard| 9.5|
| A| Forward| 22.0|
| B| Guard|13.666666666666666|
| B| Forward| 7.0|
| C| Forward| 5.0|
| C| Guard| 8.0|
+----+--------+------------------+
```

A careful review of this detailed output strongly validates the necessity of multi-column

aggregation. We observe significant disparities in scoring based on positional roles, particularly evident within Team A, where Forwards maintain a much higher scoring average than Guards. It is worth noting that PySpark defaults to retaining the full precision of floating-point numbers in the output unless explicit formatting or rounding functions are applied.

The average points value for **Guards on team A** is precisely **9.5**.
The average points value for **Forwards on team A** is **22.0**, clearly identifying them as the primary scoring role.
The average points value for **Guards on team B** is approximately **13.67** (13.666...).
The average points value for **Forwards on team B** is exactly **7.0**.

Conclusion and Next Steps in PySpark Aggregation

Achieving proficiency with the **groupBy()** function, coupled with essential aggregation methods such as `mean()`, is absolutely fundamental for conducting effective and scalable data analysis using PySpark. Whether your analytical goal demands a high-level summary based on a single variable or an intricate, detailed breakdown across multiple categorical dimensions, these DataFrame techniques offer the flexibility, efficiency, and robust performance necessary for successfully manipulating big data volumes. Always remember that the integrity of your aggregation results hinges on two critical steps: correctly identifying the appropriate grouping columns and ensuring that the target column for the calculation is strictly numerical.

For data professionals interested in expanding their knowledge beyond simple averages, the PySpark ecosystem supports a wealth of other common aggregation tasks. Further exploration might include calculating statistical metrics like standard deviation (using `stddev()`), finding maximum or minimum values (`max()` / `min()`), or performing highly customized aggregations using advanced features such as User-Defined Functions (UDFs).

The following resources provide excellent pathways for continuing your mastery of advanced PySpark DataFrame operations: