Calculate the Mean by Group in PySpark

Authored by

stats writer

November 16, 2025

Introduction: Calculating Grouped Means in PySpark

Analyzing data often requires calculating descriptive statistics across specific subgroups. In the realm of big data processing, PySpark provides powerful and efficient tools for performing these essential aggregation operations, particularly when dealing with large-scale datasets that necessitate distributed computation. Calculating the mean value--or average--of a numerical column, grouped by categorical variables, is one of the most common tasks performed by data engineers and data scientists. This technique allows us to derive critical insights, such as the average performance metric per category or the typical cost per region. Understanding how to execute a group by operation effectively in PySpark is fundamental to harnessing the power of the Spark framework.

This detailed guide serves as a comprehensive resource for experts seeking to understand practical PySpark techniques for data summarization. We will explore the primary methods available for computing the mean of a column based on one or more grouping keys within a PySpark DataFrame. The examples provided utilize standard PySpark syntax, illustrating how the powerful combination of the `groupBy()` and `mean()` functions enables rapid and scalable data summarization. We emphasize clarity and efficiency in the code examples, ensuring they are directly applicable to production environments where optimized performance is paramount.

The methods discussed below utilize the core components of the PySpark SQL API to achieve reliable statistical computations. Whether you need to group data based on a single variable, such as a team identifier, or combine multiple variables, such as team and position, the structured approach of the PySpark DataFrame facilitates straightforward implementation. Before diving into the specific syntax, we will first establish the foundational DataFrame used throughout our analysis, focusing on basketball player statistics to provide a relatable context for these powerful data manipulation concepts.

Establishing the PySpark Environment and DataFrame

Before executing any aggregation calculations, we must first initialize a SparkSession and define the source DataFrame. The SparkSession is the entry point to all PySpark functionality, managing the connection to the underlying cluster infrastructure. For demonstration purposes, we will create a small, localized DataFrame containing data related to basketball players, including their team affiliation, position, and achieved points. This controlled dataset allows us to clearly verify the results of our grouped mean calculations.

The dataset is intentionally designed to contain multiple entries for each combination of 'team' and 'position', making it ideal for demonstrating how aggregation functions correctly consolidate and summarize data across these defined subgroups. The columns are 'team' (categorical), 'position' (categorical), and 'points' (numerical). Our goal is to calculate the average points scored, grouped

by either 'team' alone, or by the combination of 'team' and 'position'. This setup mirrors real-world analytical scenarios where metrics need to be calculated across various dimensions simultaneously.

The following code snippet demonstrates the necessary steps for initializing the Spark environment, defining the schema, and viewing the resulting DataFrame structure. It is critical to ensure that the numerical column (**'points'** in this case) is correctly typed for arithmetic operations to be performed accurately. We use standard list-of-lists data initialization, which is a common practice when generating small test datasets directly within the PySpark environment.

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	Guard	13
B	Forward	7
C	Guard	8
C	Forward	5
+----+--------+------+

Method 1: Grouping by a Single Categorical Column

The most straightforward aggregation task involves grouping the data based on the unique values present in a single categorical column. In our example, we want to determine the overall average points scored by players within each individual team (A, B, and C). This operation is achieved by chaining the `groupBy()` function, specifying the grouping column, followed immediately by the `mean()` function, specifying the numerical column on which the calculation should be performed. This syntax is highly expressive and mirrors typical SQL aggregation structures.

The `df.groupBy('team')` part of the command logically partitions the DataFrame based on the **'team'** values. Subsequently, the `.mean('points')` function calculates the arithmetic mean for the **'points'** column within each of those partitions. The output of this operation is a new DataFrame, where each row represents a unique group key (the team identifier), and a new column--typically named `avg(points)`--holds the calculated mean value for that group. Note that the original DataFrame remains unaltered, as PySpark operations are inherently immutable, always returning a new DataFrame result.

This approach is highly efficient because PySpark leverages the distributed nature of the Spark engine. When `groupBy()` is called, the data is typically shuffled across the cluster executors so that all related rows (e.g., all rows belonging to Team A) reside together before the mean calculation is performed. This distributed process minimizes network overhead and ensures fast processing, even with billions of records. Below is the precise syntax required to calculate the mean points grouped solely by the team affiliation.

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

Example 1: Analyzing Single Column Aggregation Results

Upon executing the syntax for single column grouping, the output provides a concise summary of the average performance for each team. This result is immensely useful for high-level comparisons

across organizational units. For instance, comparing the average points across teams A, B, and C immediately highlights which team, on average, has the highest scoring ability among the measured players in the dataset, aiding quick decision-making and performance evaluation.

The generated output confirms the efficiency of the PySpark aggregation pipeline. The new DataFrame contains only three rows, corresponding to the three unique teams present in our sample data. The automatically generated column name, `avg(points)`, clearly indicates the function applied and the source column, although professional practice often involves renaming this output column for improved readability and subsequent processing steps, a technique we will discuss later.

Let us examine the specific results derived from running the single-column group aggregation against our basketball dataset to understand the quantitative insights produced:

#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 analysis of this summary table, we can extract the following quantitative observations:

The average points value for players on **team A** is **15.75**. This suggests a relatively high overall scoring average for this group compared to the others.

The average points value for players on **team B** is exactly **12.0**. Team B falls into the moderate range concerning average performance.

The average points value for players on **team C** is **6.5**. This represents the lowest average scoring rate among the three observed teams.

Method 2: Grouping by Multiple Columns for Granular Analysis

Often, a single grouping criterion is insufficient for deriving meaningful insights. Data analysis frequently requires calculating metrics across combinations of attributes--a process known as multi-dimensional aggregation. For instance, we may need to calculate the average points scored

not just by team, but specifically by the intersection of 'team' and 'position' (e.g., Guards on Team A vs. Forwards on Team A). PySpark handles this complexity seamlessly by extending the arguments passed to the `groupBy()` function.

To group by multiple columns, we simply pass a comma-separated list of column names (as strings) into the `groupBy()` function. The syntax `df.groupBy('team', 'position')` instructs PySpark to create a distinct group for every unique combination found in these two columns. All subsequent aggregation functions, such as `mean('points')`, will then operate independently on these newly formed, smaller partitions. This granular level of analysis is crucial for identifying performance differences within subgroups that might be masked by a broader single-column average.

Executing multi-column grouping increases the granularity of the analysis significantly, providing a deeper understanding of the underlying data distribution. For large datasets, this operation is still highly optimized by Spark's execution engine, which manages the complex shuffling and partitioning required to bring all rows belonging to a specific ('team', 'position') tuple together before calculating the average points for that unique group. This method represents a powerful and flexible approach to data segmentation and statistical summarization within the distributed computing environment.

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

Example 2: Analyzing Multi-Dimensional Aggregation Results

When the multiple grouping columns (**'team'** and **'position'**) are applied, the resulting DataFrame expands to include rows for every unique combination present in the source data. This allows for detailed performance profiling. For instance, we can now clearly see that while Team A's overall average was 15.75 (from Example 1), there is a significant discrepancy between the scoring averages of their Guards and their Forwards, providing a far richer context than the aggregated team metric alone.

A noticeable characteristic of this result set, especially when dealing with floating-point arithmetic in distributed systems, is the potential for highly precise, long decimal values (e.g., 13.6666...). While PySpark maintains high precision internally, it is often necessary to round or format these numerical outputs for presentation and reporting purposes. Techniques like using the `round()` function in combination with `agg()` can refine the display of these averages for easier human consumption.

The final structured output clearly links the aggregated mean points to the specific team and

position combination, offering maximum detail for segmentation analysis. It highlights the power of PySpark's SQL API in turning raw, transactional data into actionable summary statistics through flexible grouping mechanisms that are essential for deep-dive analysis.

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

Reviewing the results from the multi-column grouping yields granular insights into player performance:

The average points value for **Guards on team A** is **9.5**.

The average points value for **Forwards on team A** is significantly higher at **22.0**, indicating that Team A's overall high average is heavily influenced by its Forwards.

The average points value for **Guards on team B** is approximately **13.67** (rounded for reporting purposes).

The average points value for **Forwards on team B** is **7.0**, showcasing internal variations within Team B.

Advanced Techniques: Renaming and Multiple Aggregations with `agg()`

While using the chaining method `df.groupBy().mean()` is functional and quick for simple tasks, it has two primary drawbacks in production environments: the resulting column name is generic (e.g., `avg(points)`), and it only allows for a single aggregation type per chain. For more complex analytical needs, especially when calculating multiple statistics simultaneously (e.g., mean, standard deviation, and count) or when custom column names are required, the `agg()` function should be employed.

The `agg()` method provides robust control over the output schema and aggregation logic. Instead of chaining `.mean()`, you chain `.agg()` after `.groupBy()`, passing a dictionary or a list of required aggregation functions along with their desired output column names. For instance, to calculate the mean points and rename the column to **average_points**, the syntax would utilize the `pyspark.sql.functions` module (often imported as `F`). This practice significantly improves code readability and maintainability when processing complex statistical requirements.

Adopting the `agg()` function for statistical summaries is considered a best practice in professional PySpark development. It allows analysts to not only calculate the mean but also to incorporate other critical descriptive statistics, such as the minimum, maximum, and standard deviation, within the same single pass over the grouped data. This minimizes unnecessary shuffling operations and enhances computational efficiency, which is vital when dealing with petabyte-scale data lakes. Mastering this combination of `groupBy()` and `agg()` is key to becoming proficient in data summarization using the PySpark framework.

Conclusion

Calculating the mean value by group is a core data manipulation task, and PySpark offers two highly effective methods--one for single-column grouping and one for multi-column grouping--that leverage the distributed computational power of the Spark engine. These operations are critical for summarizing large datasets and extracting meaningful insights, such as average performance metrics categorized by specific attributes like team, location, or product type.

By utilizing the sequential application of the `groupBy()` and `mean()` DataFrame methods, users can quickly transform raw data into aggregated summaries. We demonstrated how the syntax scales seamlessly from grouping by a single categorical variable (**'team'**) to grouping by a compound key (**'team'** and **'position'**). Furthermore, understanding the advanced usage of the `agg()` function ensures that analytical results are not only accurate but also formatted professionally with descriptive column names, meeting the stringent requirements of enterprise-level data reporting.

Proficiency in these aggregation techniques is essential for anyone working with distributed data processing. They form the foundation for complex feature engineering, dashboard creation, and statistical modeling in high-volume data pipelines. The ability of PySpark to handle these operations efficiently and scalably solidifies its position as the premier tool for modern big data analysis.