How to Calculate the Mean by Group in PySpark

Authored by

stats writer

February 8, 2026

Calculating the mean, or arithmetic average, across specific subsets of data is a fundamental operation in statistical analysis. When dealing with vast quantities of information in big data environments, tools like PySpark become essential. PySpark facilitates complex grouped calculations efficiently by leveraging distributed processing capabilities. The core methodology involves utilizing the powerful groupBy() transformation, which partitions the data based on categorical columns, followed by an aggregation function like `mean()` or `agg()`. This approach ensures that the average is computed independently for each predefined group, delivering focused and actionable insights into the underlying structure of the dataset. This technique is indispensable for identifying critical patterns, performance differences, or trends across distinct categories within large-scale datasets, such as performance metrics grouped by geographic region or product type.

Calculating the Mean by Group in PySpark

Introduction to Grouped Aggregation in PySpark

When analyzing structured data, it is often necessary to move beyond simple statistics calculated across the entire dataset. Aggregations, especially calculating the central tendency metrics like the mean, gain significant analytical power when applied contextually within subgroups. PySpark, the Python API for Apache Spark, offers robust SQL functions tailored for this exact purpose. The efficiency of PySpark stems from its ability to parallelize these calculations across a cluster, making grouped aggregation feasible even for massive, petabyte-scale data volumes. Understanding the proper use of the aggregation pipeline--specifically the groupBy() transformation--is foundational for any data engineer or scientist working within the Spark ecosystem.

The process relies entirely on the inherent structure of the PySpark DataFrame. A DataFrame is a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database or an R/Python data frame. By selecting one or more categorical columns for grouping, we instruct Spark to collect all rows sharing identical values in those columns into a single logical group. Once grouped, any subsequent aggregation function, such as `mean()`, is executed on the numerical column within the confines of that specific group, producing one result row per unique group combination.

While this guide focuses explicitly on calculating the average, the principles outlined here apply universally to other aggregate statistics supported by PySpark, including standard deviation, sum, minimum, maximum, and count. The flexibility provided by Spark's aggregation framework allows for complex data transformation tasks, enabling users to generate summary statistics that condense millions of records into highly informative statistical tables.

Setting Up the PySpark Environment and Sample Data

Before executing any grouped calculations, a connection to the Spark environment must be established, typically done using the SparkSession object. The SparkSession serves as the entry point to programming Spark with the Dataset and DataFrame API. For demonstrative purposes, we will first define and instantiate a sample DataFrame containing basketball player metrics, categorized by team and position, along with their associated points scored.

The construction of the sample data is crucial for illustrating the methods accurately. Our dataset contains three columns: `team` (categorical identifier), `position` (further categorical refinement), and `points` (the numerical column on which the mean calculation will be performed). This setup mimics real-world scenarios where performance statistics need to be compared across various organizational structures or groups.

The following code snippet demonstrates the necessary steps to initialize Spark and create the sample DataFrame. Note the explicit definition of the schema via the column names, ensuring clarity and structure for the subsequent aggregation operations.

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

Core Technique: Calculating the Mean Grouped by a Single Column

The most straightforward grouping operation involves partitioning the data based on a single categorical variable. In our running example, we want to determine the average points scored by players, separated by their respective teams. This calculation provides immediate feedback on the overall performance disparity between the teams, irrespective of player position. The syntax is concise and highly readable, leveraging the chainable nature of DataFrame transformations.

The fundamental sequence involves calling the groupBy() function on the DataFrame, specifying the grouping column (`'team'`). This operation returns a `GroupedData` object, which is then ready to accept an aggregation. We immediately chain the `mean()` function, passing the name of the numerical column we wish to average (`'points'`). The final `show()` action triggers the execution and displays the resulting aggregated DataFrame.

This approach is highly efficient because Spark optimizes the grouping and aggregation steps, utilizing internal mechanisms like shuffling to move related data partitions onto the same executor before the calculation of the mean occurs. The resulting output DataFrame will contain the grouping column and a new column, typically named `avg(column_name)`, holding the calculated average for each group.

Method 1: Syntax for Single Column Grouping

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

Example 1: Analyzing Team Performance Averages

Using the syntax outlined above, we apply the single-column grouping to our sample dataset to compare the average points achieved across teams A, B, and C. This step transitions from conceptual syntax to practical output, demonstrating how the raw data is summarized effectively.

We calculate the mean value in the points column, partitioned exclusively by the values found in the team column. The result clearly shows a concise summary of the average scoring ability for each team.

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

Interpreting the output generated by the distributed calculation reveals distinct performance levels:

The average points value for players on **team A** is **15.75**, suggesting the highest overall scoring contribution among the three teams.
The average points value for players on **team B** is **12.0**, placing them in the middle of the performance spectrum.
The average points value for players on **team C** is **6.5**, indicating a significantly lower collective scoring mean compared to teams A and B.

Advanced Grouping: Calculating the Mean Across Multiple Columns

Often, data analysis requires a finer level of granularity than what a single grouping column can provide. To achieve sub-segmentation--for instance, determining the average points scored by players based on both their team and their specific position--we utilize multi-column grouping. This technique involves passing a list of column names to the groupBy() function.

When multiple columns are specified, PySpark creates unique groups based on the concatenation of the values across all specified columns. For example, 'Team A' combined with 'Guard' forms a distinct group separate from 'Team A' combined with 'Forward'. This allows for deep, nested analytical comparisons, exposing variations in performance that might be masked by simple one-

dimensional averages.

The complexity of the calculation increases slightly, but the syntax remains intuitive: simply list the categorical columns in the groupBy() call. This powerful feature is critical for analyzing complex hierarchies, such as sales grouped by product category and region, or employee performance grouped by department and tenure level.

Method 2: Syntax for Multi-Column Grouping

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

Example 2: Detailed Analysis of Positional Averages

Applying the multi-column grouping method allows us to drill down into the data and isolate the performance contribution of specific player types within each team. This provides a crucial insight that the overall team average (from Example 1) cannot provide, highlighting, for instance, which positions contribute disproportionately to the team's total score.

We calculate the mean value in the points column, now segmented by both team and position. The output expands the results dramatically, showing six distinct group averages where previously there were only three.

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

The granular results provide a comprehensive view of player contributions:

The average points value for **Guards** on team A is **9.5**, which is considerably lower than the team

average of 15.75, suggesting Guards are not the primary scorers.
The average points value for **Forwards** on team A is **22.0**, which is the highest individual group average in the dataset, driving Team A's overall high performance.
The average points value for **Guards** on team B is approximately **13.67**, indicating solid performance from that position within the team structure.
The average points value for **Forwards** on team B is **7.0**, highlighting a specific area of low scoring contribution within Team B.

Deep Dive into the PySpark groupBy() and agg() Functions

While the `df.groupBy().mean()` syntax is highly convenient, it is structurally a shorthand notation for a more generic and powerful method using `agg()`. The `agg()` function allows users to perform multiple simultaneous aggregations, rename output columns, or apply custom functions post-grouping, providing maximum flexibility in data summarization. Understanding the relationship between these two approaches is vital for complex analytical tasks.

Specifically, `df.groupBy('team').mean('points')` is equivalent to `df.groupBy('team').agg({'points': 'mean'})`. Furthermore, using `agg()` allows for custom naming of the output column, preventing the default `avg(points)` label. For instance, `df.groupBy('team').agg(F.mean('points').alias('average_points'))`, where `F` refers to functions imported from `pyspark.sql.functions`, provides a cleaner, more production-ready output structure.

The flexibility of `agg()` means that a single grouping operation can calculate the mean, minimum, and count simultaneously, minimizing the number of shuffles and passes over the data required. This feature dramatically improves performance when numerous summary statistics are needed for the same groups, solidifying the importance of mastering both the simple `mean()` shorthand and the robust `agg()` function in PySpark programming.

Conclusion and Further PySpark Applications

The ability to calculate the mean by group in PySpark is a cornerstone of distributed data analysis. By employing the groupBy() method, whether targeting a single column or a combination of columns, data professionals can transform raw, extensive datasets into concise, statistically rich summaries. This structured approach not only accelerates insight generation but also ensures that computations are performed efficiently and scalably across the Spark cluster architecture.

Mastering these aggregation techniques opens the door to numerous advanced analytical tasks within the DataFrame API, including window functions, cumulative aggregations, and complex statistical modeling preparation. The grouped mean calculation demonstrated here serves as a template that can be easily adapted for calculating other crucial metrics, such as variance or

skewness, which are essential for deeper data exploration and machine learning feature engineering.

For those seeking to further enhance their proficiency in distributed data processing, exploring additional tutorials on aggregation functions, such as counting distinct values, calculating standard deviations, or using SQL expressions within PySpark, is highly recommended.

The following tutorials explain how to perform other common tasks in PySpark: