How to Groupby and Aggregate Multiple Columns in PySpark

Authored by

stats writer

February 6, 2026

The `groupBy().agg()` methodology is a cornerstone feature within the **PySpark** framework, essential for transforming raw, granular data into meaningful, summarized insights. This powerful combination allows developers and data analysts to segment large datasets based on one or more categorical columns and then apply various **aggregate functions**--such as calculating sums, averages, or counts--across the resulting groups. Utilizing `groupBy().agg()` on multiple columns simultaneously is critical for complex data analysis, enabling the computation of diverse statistics in a single, efficient pass.

This approach significantly streamlines the data processing pipeline, offering a high degree of flexibility and performance when dealing with massive datasets managed by **PySpark DataFrames**. The ability to specify multiple aggregation rules tailored to different output columns ensures that analysts can derive comprehensive summaries without needing iterative processing steps. By consolidating multiple statistical calculations into one operation, we not only improve code readability but also leverage the distributed processing capabilities of **PySpark** to maximum effect, making data preparation and analysis tasks robust and extremely scalable.

Mastering PySpark: Using GroupBy and Aggregation on Diverse Columns

The Foundational Syntax of GroupBy Aggregation

To effectively harness the power of distributed computing offered by **PySpark**, it is vital to understand the precise syntax required for grouping and simultaneous aggregation. The core structure involves selecting the grouping keys, followed immediately by the application of the `.agg()` method. This method accepts a variable number of aggregation expressions, each targeting a specific column and using a standard SQL-like function.

When executing complex data transformations, the clarity of the code is paramount. The `.groupBy()` method takes one or more column names as arguments, which define the level at which the data should be segmented. Subsequently, the `.agg()` method defines the specific calculations to be performed on the grouped data. Crucially, each aggregation must be aliased using `.alias()` to provide a distinct and meaningful name for the new resulting columns in the output **DataFrame**, ensuring that the final output is intuitive and ready for subsequent analysis.

You can use the following syntax to group by and perform aggregations on multiple columns in a **PySpark DataFrame**. Note that importing `pyspark.sql.functions` is required to access the necessary aggregate functions such as `sum`, `mean`, and `count`.

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()

Dissecting the Aggregate Functions Used

The example above demonstrates the versatility of the aggregation step by applying three distinct functions to two different input columns. Understanding what each function calculates is key to interpreting the output. While the grouping column (`team`) remains constant, the aggregation step provides flexibility to summarize various metrics, offering a complete profile of each group.

The specific aggregation definitions used in this structure ensure that we derive precise statistical measures for each unique team segment. This particular example groups the rows of the **DataFrame** by the `team` column, then performs the following aggregations:

Calculates the **sum** of the **points** column and uses the alias **sum_pts**. This gives the total points contributed by all players within that team.
Calculates the **mean** (average) of the **points** column and uses the alias **mean_pts**. This reveals the average scoring efficiency for players on that specific team.
Calculates the **count** of the **assists** column and uses the alias **count_ast**. Because `count()` tallies non-null values, this implicitly tells us the number of records (players or games) associated with each team.

This powerful combination means that we are simultaneously deriving totals, central tendencies, and record counts across our data partitioned by team, all within a single, optimized operation. This significantly reduces the computational overhead compared to running these calculations sequentially.

Example: How to Use Groupby Agg On Multiple Columns in PySpark

Setting Up the Environment and Sample DataFrame

To provide a clear, practical demonstration, we first need to establish a working PySpark environment and define a sample dataset. For this example, we will simulate a dataset containing basketball player statistics, including their team, position, points scored, and assists recorded. This dataset is small enough to be easily visualized but complex enough to showcase the multi-column aggregation technique effectively.

We begin by initializing a `SparkSession`, which is the entry point for using **PySpark** functionality. Following the session creation, we define the raw data list and the corresponding column names. The columns defined are `team` (our grouping key), `position`, `points` (a metric for summation and averaging), and `assists` (a metric for counting records).

The subsequent step involves creating the **DataFrame** itself using `spark.createDataFrame(data, columns)`. Viewing the initial data structure through `df.show()` confirms that our source data is correctly structured and ready for the grouping operation. This preparation phase is crucial, as the quality and structure of the input data directly impact the reliability of the aggregation results.

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

Executing the Multi-Column Aggregation Query

Once the source **DataFrame** is prepared, the next step is applying the grouped aggregation logic. We utilize the same concise syntax introduced earlier, ensuring that all necessary aggregation functions are imported from `pyspark.sql.functions`. Our goal here is to group all player records by the `team` identifier and calculate the total points, the average points, and the count of records for each team in one streamlined process.

The command structure `df.groupBy('team').agg(...)` efficiently distributes this workload across the cluster. PySpark handles the necessary shuffle operations--moving data with the same team identifier to the same partition--before applying the specified functions. This internal optimization is what makes PySpark suitable for massive scale computations, performing aggregations far more quickly than traditional single-node databases or scripting environments.

We can use the following syntax to group the rows by the values in the **team** column and then calculate several aggregate metrics simultaneously, producing a clean summary table:

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 Aggregated Results

The resulting **DataFrame**, shown in the output above, condenses the ten original records into three rows, one for each unique team (A, B, and C). Each row now contains the summarized statistical output derived from the aggregation functions applied across the entire group. Analyzing this output is where the real insights are gained, comparing the performance metrics across the different segments.

For instance, by examining Team A, we immediately gain three key pieces of information: the total scoring output, the average efficiency per record, and the total number of records contributing to these totals. If the objective was to compare overall team performance, this aggregated view provides a clear basis for comparison that the raw data could not offer without manual calculation.

The resulting DataFrame clearly shows the sum of the points values, the mean of the points values, and the count of assists values for each team. For example, we can observe the summarized metrics for Team A:

The sum of points for team A is **63**, representing the total points scored by all players within that team across the recorded entries.
The mean of points for team A is **15.75**, indicating the average points scored per player or record for Team A.
The count of assists for team A is **4**, confirming that four records (players or game observations) contributed to the statistics for Team A.

Advantages of Multi-Metric Aggregation

Performing multiple aggregations within a single `.agg()` call is not merely a syntactic convenience; it is a fundamental optimization technique in distributed computing. When PySpark executes a grouping operation, it must perform an expensive data shuffle to collect all related records onto the same cluster nodes. If we were to run three separate queries (one for sum, one for mean, and one for count), the system would have to perform three separate shuffle operations.

By using `df.groupBy().agg(func1, func2, func3)`, we instruct PySpark to perform the grouping and shuffle just once. All defined aggregate functions are then computed locally on the partitioned data, dramatically reducing network I/O and overall processing time. This single-pass efficiency is paramount when working with petabyte-scale datasets where shuffles represent the most significant performance bottleneck.

Further Applications and PySpark Flexibility

While this example focused on basic numeric statistics (sum, mean, count), the `.agg()` method

supports a vast array of sophisticated functions available in `pyspark.sql.functions`. Users can calculate standard deviations, variances, minimums, maximums, and even complex window functions within the aggregation context. Furthermore, the grouping key itself is highly flexible; one could group by multiple columns, such as `df.groupBy('team', 'position')`, to get granular statistics broken down by both team and player role.

This flexibility allows data scientists to move beyond basic reporting and delve into complex exploratory data analysis (EDA). For instance, grouping by geographic region and product type while calculating median sales and percentile distribution provides immediate insights into localized market performance. Mastering the integration of multiple aggregate functions within the GroupBy structure is thus a critical skill for advanced data manipulation in **PySpark**.

Related PySpark Tutorials for Common Data Tasks

The following tutorials explain how to perform other common tasks related to data transformation and analysis in PySpark, building upon the foundational knowledge of aggregation and grouping:

Learning to calculate percentile rankings within grouped data structures.
Understanding how to pivot DataFrames to transform rows into columns for easier reporting.
Techniques for joining large PySpark DataFrames efficiently using appropriate join strategies.