Learning PySpark: Calculating the Median by Group

Authored by

Mohammed looti

November 11, 2025

Introduction to Grouped Median Calculation in PySpark

Analyzing large datasets often requires calculating descriptive statistics segmented by specific categories. This process, known as grouped aggregation, is central to effective PySpark data analysis, particularly when dealing with massive, distributed data volumes. While the mean (average) is a common metric, it suffers from a critical drawback: high sensitivity to outliers, which can skew the perception of central tendency. For producing statistically robust summaries, the median--the middle value of an ordered dataset--provides a far more reliable measure, reflecting the typical value without being unduly influenced by extreme observations. Understanding how to calculate this metric efficiently across groups is an indispensable skill for data professionals working with distributed systems.

This comprehensive guide details the mechanism for efficiently computing the median value for distinct groups within a DataFrame using PySpark's powerful and optimized aggregation tools. The core methodology relies on a specific sequence of operations: first, leveraging the **groupBy()** transformation to logically partition the data based on categorical variables; second, invoking the **agg()** action; and finally, applying the statistical function **F.median()**, which is imported from the `pyspark.sql.functions` module. This approach is engineered for maximum performance in distributed computing environments, crucial when handling Big Data volumes where standard, non-distributed methods would fail due to memory constraints or excessive processing time.

Mastering this technique is essential for analysts and engineers who require precise, segmentation-based metrics across complex data structures. We will systematically explore two primary methodologies in the sections that follow: initially, calculating the median based on a single grouping criterion for broad summary views, and subsequently, extending the calculation to include multiple criteria to unlock deeper, more granular insights. These sophisticated techniques are widely applicable across diverse domains, ranging from meticulous financial modeling and risk assessment to high-stakes sports analytics, as demonstrated through our practical examples using player performance data.

Setting Up the PySpark Environment and Sample Data

Before commencing any data manipulation or grouped calculations in a distributed manner, the foundational step involves initializing the processing environment. This mandates the creation of a **SparkSession**, which serves as the entry point for interacting with the underlying Spark functionality and coordinating distributed tasks across the cluster. We must also precisely define the input data structure that the session will manage. Our illustrative sample dataset is designed to model basketball player performance, featuring three key columns: the player's `team`, their functional `position` (a second categorical variable), and their accumulated `points` (the numerical column on which we will calculate the median). This structure deliberately incorporates categorical

variables to demonstrate how performance metrics can be reliably segmented and summarized.

The following canonical code block handles the crucial creation and initialization of the sample PySpark DataFrame. It is critical to note the standard practice of using `SparkSession.builder.getOrCreate()`. This method ensures that the application either retrieves an existing, active SparkSession if one is already running in the environment, or it gracefully starts a new session if none is available. This practice is standard for robust and repeatable PySpark development, preventing redundant resource allocation and guaranteeing consistent execution context.

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

The resulting DataFrame, named `df`, represents the foundational structure for all subsequent analysis. It is now ready for distributed processing. The next stage involves demonstrating the exact PySpark syntax necessary to invoke the calculation of the median for the `points` column, dynamically segmented based on various levels of aggregation complexity. This preparation ensures that we move from conceptual understanding directly into practical application using a well-defined dataset.

Core Syntax for Calculating Median by a Single Grouping Column

The most elementary form of aggregated analysis involves summarizing a metric, such as the median, based exclusively on the distinct values present in a single categorical column. This process is of paramount importance for generating quick, high-level summaries or comparing overall characteristics across major categories within the dataset. In the PySpark framework, this is efficiently executed by systematically chaining the **groupBy()** transformation with the **agg()** action. Specifically, the operation requires specifying the column(s) that define the groups and then passing the `F.median()` function, targeting the numerical column for which the middle value must be computed.

When the data is grouped by a single column--for example, the `team` column--PySpark logically partitions the entire DataFrame into distinct subsets, one for every unique team identifier. The chosen aggregation function, in this scenario, the median, is then applied independently and in parallel to the data within each logical partition. This distributed application is what grants Spark its immense speed when processing petabytes of data. It is a mandatory prerequisite for this process to ensure that the necessary functions alias, conventionally `F`, is imported from the `pyspark.sql.functions` module, as this provides access to all the optimized aggregation utilities, including F.median().

The following code snippet precisely outlines the technical syntax utilized to calculate the median points, grouping the data solely by the `team` column. This particular calculation yields a rapid summary of the intrinsic scoring ability across the different teams. Crucially, by selecting the median instead of the mean, we effectively mitigate the disproportionate influence of any single player's extraordinarily high or low score, providing a more reliable indicator of the typical team

performance.

import pyspark.sql.functions as F

#calculate median of 'points' grouped by 'team'
df.groupBy('team').agg(F.median('points')).show()

Practical Example 1: Median Grouped by 'Team'

We now proceed to apply the single-column grouping methodology directly to our previously defined sample basketball dataset, `df`. The objective of this first practical exercise is straightforward: to accurately determine the median scoring performance, represented by the `points` column, for every unique `team` identifier present in the dataset (Teams A, B, and C). The resultant aggregated DataFrame offers a clear, high-level, and statistically robust comparison of performance across the teams, providing insights that are minimally affected by anomalous data points.

The operational steps involve initializing the aggregation with the **groupBy('team')** function, which defines the boundaries for the grouping operation. Following this transformation, we apply the **agg(F.median('points'))** function. This action computes the median score independently within the confines of each defined team group. The output generated is a new, concise DataFrame, which structurally contains two columns: the team identifier used for grouping and the corresponding calculated median value for the points scored by players on that team.

Executing the specified PySpark code generates the output shown below. This result is crucial as it explicitly illustrates the measure of central tendency for scoring capabilities across the three distinct teams, facilitating immediate statistical comparison and interpretation.

import pyspark.sql.functions as F

#calculate median of 'points' grouped by 'team'
df.groupBy('team').agg(F.median('points')).show()

```
+----+--------------+
|team|median(points)|
+----+--------------+
| A| 16.5|
| B| 13.5|
| C| 6.5|
+----+--------------+
```

A systematic analysis of these results allows us to derive precise statistical summaries concerning the performance profile of each team. Specifically:

The **median points value** for players affiliated with **Team A** is calculated as **16.5**. This figure is derived by ordering the four scores (8, 11, 22, 22); since the count is even, the median is the average of the two central values (11 and 22).

The **median points value** for players on **Team B** stands at **13.5**. Team B also has four scores (7, 13, 14, 14). The median is the arithmetic mean of the two central values (13 and 14), thus resulting in 13.5.

The **median points value** for players associated with **Team C** is **6.5**. Team C has only two scores (5, 8); the median is calculated as the average of these two values, demonstrating the flexibility of the median function even with small groups.

Extending the Calculation: Grouping by Multiple Columns

While single-column grouping provides foundational summaries, it frequently lacks the necessary level of detail for sophisticated data analysis. In real-world scenarios, particularly those requiring granular insights--such as evaluating performance segmented not just by team but also by a player's specific functional role--it becomes imperative to implement grouped data calculations across multiple columns simultaneously. PySpark is designed to handle this complexity seamlessly by allowing the data scientist to list all necessary grouping columns within the **groupBy()** method.

Grouping by multiple columns, often referred to as composite grouping, leads to the creation of significantly more logical partitions within the distributed system. This process enables the isolation of highly specific data subsets based on the unique combination of the chosen categorical variables. For example, by simultaneously grouping the data by both `team` and `position`, we can facilitate a direct and highly insightful comparison between the median scoring of Guards on Team A and the median scoring of Guards on Team B. This nuanced, segmented approach offers far greater analytical depth than merely comparing the overall performance of Team A versus Team B, where roles might be mixed.

This multi-dimensional grouping methodology ensures that the resulting statistical measures are highly relevant and context-specific, reflecting the distinct characteristics defined by the combination of categorical variables. Consequently, this technique is absolutely indispensable for advanced segmentation tasks, complex performance benchmarking, and intricate business intelligence applications where broad averages obscure important differences. The syntax required to execute this advanced grouping remains fundamentally similar to the single-column approach, requiring only a simple extension of the arguments supplied to the `groupBy()` function:

import pyspark.sql.functions as F

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

Practical Example 2: Median Grouped by 'Team' and 'Position'

To powerfully illustrate the capability and analytical benefit of multi-column aggregation, we apply the aforementioned syntax to calculate the median points based on the combined criteria of `team` and `position`. This composite grouping operation yields a total of six distinct median calculations, corresponding precisely to every unique pairing of team (A, B, C) and position (Guard, Forward) present within our sample dataset.

This attained level of detail is profoundly critical for rigorous performance evaluation, as it correctly and statistically separates the metrics for players based on their specific, functional role within the team hierarchy. Performance expectations, scoring distributions, and responsibilities for Guards, for instance, are inherently different from those of Forwards. Calculating the median separately for each segment effectively addresses and accounts for these fundamental differences, providing truly meaningful comparative statistics.

The resultant DataFrame, generated when grouping by both criteria, is presented below. Observe how the median values now reflect the specific context of both team and position, offering targeted insights into role-based performance:

import pyspark.sql.functions as F

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

```
+----+--------+-------------+
|team|position|median(points)|
+----+--------+-------------+
| A| Guard| 9.5|
| A| Forward| 22.0|
| B| Guard| 14.0|
| B| Forward| 7.0|
| C| Forward| 5.0|
| C| Guard| 8.0|
+----+--------+-------------+
```

The output successfully reveals highly specific and actionable median scores for each player segment defined by the composite key:

The median points value for **Guards on Team A** is **9.5**, derived from the ordered scores 8 and 11.

The median points value for **Forwards on Team A** is exactly **22.0**, calculated from the identical scores 22 and 22.

The median points value for **Guards on Team B** is **14.0**, representing the middle value of the three ordered scores 13, 14, and 14.

Summary and Further Resources

The capability to calculate statistically sound metrics, particularly the median, when grouped by categorical variables in PySpark, forms a foundational cornerstone of reliable distributed data analysis. By harnessing the combined power of the **groupBy()** transformation and the efficient **F.median()** function, data professionals are empowered to rapidly generate accurate, outlier-resistant statistics across even the most massive datasets. This methodology allows for flexibility, whether the goal is to group by a single, broad categorical variable or to combine multiple attributes for intensely detailed segmentation and comparative analysis. This approach is demonstrably critical for both accurate organizational reporting and for facilitating insightful, data-driven decision-making processes when managing large-scale data aggregation tasks.

Achieving proficiency in grouped aggregation is a vital developmental step for anyone seeking mastery of PySpark, as it enables the execution of complex data transformations far beyond simple data filtering and selection. We strongly encourage readers to extend their exploration to other related statistical aggregation tasks. These include calculating quantiles, which provide a breakdown of the data distribution, or computing measures of spread like variance and standard deviation, all of which adhere to a similar, consistent syntax pattern utilizing the versatile `agg()` function.

For continued learning and to robustly expand your PySpark skillset into advanced domains, we recommend diligently reviewing the official documentation and engaging with tutorials focused on related and specialized topics:

Exploring the computation of other essential statistical measures, such as standard deviation and variance, using the built-in PySpark functions.

Investigating advanced concepts like window functions for calculating rolling medians, cumulative statistics, or percentile ranks over defined partitions without explicit grouping.

Learning best practices for optimizing `groupBy()` operations specifically for performance

enhancement in highly demanding, large-scale cluster environments to ensure scalability and efficiency.