How to Calculate the Median by Group in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 8, 2026

Calculating the median by group is a fundamental operation in advanced data processing, particularly when dealing with massive datasets. In the context of PySpark, this technique involves efficiently finding the central value--the 50th percentile--for distinct subsets of data defined by one or more grouping keys. This capability is essential because the median, unlike the mean, is resistant to outliers, providing a robust statistical measure of central tendency that is crucial for accurate data analysis and machine learning feature engineering tasks. By leveraging PySpark's distributed computing power, we can apply the aggregation function across grouped data partitions, ensuring high performance even with petabytes of information.

The process generally involves using the built-in DataFrame operations, specifically the `groupBy()` transformation followed by the `agg()` action, where the `F.median()` function from `pyspark.sql.functions` is applied. This structured approach allows developers and data scientists to move beyond simple total medians and delve into the characteristics of granular data segments, leading to more insightful and actionable business intelligence.

Calculate the Median by Group in PySpark

The Necessity of Grouped Median Calculation in PySpark

In large-scale data processing using PySpark, data is rarely homogenous; it usually contains categorical variables that define natural subgroups. To truly understand the distribution of a metric, it is often insufficient to calculate a single overall median. Instead, we must calculate the median within these specific groups. For example, when analyzing sales figures, calculating the median sales per region or per product category provides much deeper insights than calculating the overall median sale amount across the entire company.

PySpark provides the necessary functions to perform this operation efficiently, leveraging the clustered nature of Apache Spark. The core principle involves partitioning the DataFrame based on the grouping keys, allowing the subsequent aggregation (median calculation) to occur in parallel across the cluster. This parallel execution is the cornerstone of Spark's performance advantage when handling massive data volumes, distinguishing it from traditional sequential processing methods.

The primary methods detailed below demonstrate how to utilize the `pyspark.sql.functions` module, aliased as `F`, to access the optimized `median()` aggregation function. We will explore scenarios involving both single-column grouping and multi-column composite grouping, showcasing the versatility of PySpark's aggregation framework when performing complex statistical computations.

The standard syntax pattern for calculating a grouped median in a PySpark DataFrame (`df`)

involves these two approaches:

Method 1: Calculate Median Grouped by One Column

import pyspark.sql.functions as F

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

Method 2: Calculate Median Grouped by Multiple Columns

import pyspark.sql.functions as F

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

Establishing the PySpark Environment and Sample Data

Before executing the median calculation, it is necessary to initialize a SparkSession and prepare a sample DataFrame. This foundational setup allows us to simulate a real-world dataset scenario, in this case, a collection of basketball player statistics including their team affiliation, position, and accrued points. This structured approach ensures that the examples provided are fully reproducible and demonstrate the required initialization steps for any PySpark script.

The following code block outlines the steps required to create the SparkSession, define the data structure, assign meaningful column names, and finally, instantiate the DataFrame (`df`). The resulting DataFrame structure is crucial for understanding how the subsequent grouping operations will segment the data.

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 `df` contains ten records, categorized by three teams (A, B, C) and two positions (Guard, Forward), with the numerical variable `points` being the metric we intend to analyze. This structure is perfectly suited for demonstrating both simple and complex grouping scenarios using the median calculation.

Method 1: Calculating Median Grouped by a Single Attribute

The most common scenario involves calculating the median of a numerical column based on a single categorical grouping variable. In our example, we aim to determine the typical scoring output, represented by the median of the `points` column, segmented solely by the `team` column. This calculation immediately provides a quick comparison of central performance across the different teams, filtering out the effect of potentially skewed high or low scores.

To execute this, we use the `df.groupBy('team')` command to logically group all rows belonging

to the same team. Subsequently, the `.agg()` method is called, applying the `F.median('points')` aggregate function to each resulting group. PySpark handles the sorting of the points within each group and extracts the middle value, whether it is a single data point (for odd counts) or the average of the two central data points (for even counts).

The following syntax provides a clear demonstration of this process, generating a new DataFrame that contains the team identifier and the calculated median points for that team:

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

The output is a concise summary of team performance, highlighting significant differences in central scoring tendency. For instance, Team A shows a considerably higher median points value than Team C, suggesting overall better performance in the scoring metric when outliers are mitigated by the use of the median.

Method 2: Calculating Median Grouped by Composite Attributes

Often, data analysis requires finer granularity than a single grouping column can provide. When we need to segment the data by the combination of two or more attributes--for instance, determining the median score for a specific position within a specific team--we employ composite grouping. This approach is instrumental in uncovering interaction effects between variables that might be obscured when grouping by only one attribute.

In this example, we calculate the median value of the `points` column based on the unique combinations of both the `team` and `position` columns. The DataFrame is partitioned first by team, and then each team partition is further subdivided by position. The aggregation function is then applied to these smaller, more specific subgroups.

The syntax is extended by simply adding the second grouping column, `'position'`, to the `groupBy()` method:

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

This result provides a highly detailed statistical breakdown. We can now observe that while Team A's overall median was 16.5, this value masks a significant disparity: Forwards on Team A have a very high median score (22.0), while Guards on Team A have a much lower median score (9.5). This level of detail is paramount in targeted performance analysis and data segmentation tasks.

Interpreting Results and Statistical Implications

Understanding the output of the grouped median calculation is as important as the execution itself. The median value provides a robust measure of the typical observation within a group. When analyzing the results from both single and composite grouping, we gain specific, actionable statistical insights:

For Example 1 (Grouped by Team):

The median points value for players on **team A is 16.5**. This is derived from the sorted points . Since there are four observations (an even count), the median is the average of the two middle values (11 and 22), resulting in 16.5.
The median points value for players on **team B is 13.5**. This is derived from the sorted points . The median is the average of the two middle values (13 and 14).
The median points value for players on **team C is 6.5**. This is derived from the sorted points . The median is the average of the two values (5 and 8).

For Example 2 (Grouped by Team and Position), the segregation is even more telling, revealing performance patterns specific to positional roles:

The median points value for Guards on team A is **9.5**. The sorted values are .
The median points value for Forwards on team A is **22.0**. The sorted values are . This perfect median indicates highly consistent scoring at the median level for this sub-group.
The median points value for Guards on team B is **14.0**. The sorted values are . Since there are three observations (an odd count), the middle value is the median.

These calculations highlight the power of the median in providing an accurate representation of central performance, especially when dealing with smaller subsets of data where a single extreme outlier could heavily skew the mean. Using PySpark ensures that these complex sort and aggregation steps are optimized for distributed execution, regardless of the overall scale of the underlying data.

Advanced Considerations and Further PySpark Tasks

While the standard `F.median()` function is robust and accurate, data practitioners working with extremely large datasets should be aware of certain performance implications. Calculating the exact median requires sorting the data within each group, which can be computationally intensive and memory-demanding when groups are massive, even in a distributed environment like Spark.

For scenarios where absolute precision is not critical but performance is paramount, PySpark offers the `approx_quantile()` function. This function calculates the approximate median (or any specified quantile) using reservoir sampling techniques, significantly reducing computation time and resource usage by avoiding a full sort. When dealing with operational reporting or exploratory data analysis on vast datasets, approximate calculations are often preferred for their speed and scalability.

The techniques demonstrated here--grouping and aggregation--are foundational to advanced PySpark workflows. Mastering the combination of `groupBy()` and `agg()` is essential for performing virtually any complex statistical measure, including calculating standard deviation, variance, quartiles, and custom aggregations defined by user-defined functions (UDFs). Integrating these statistical insights derived from grouped medians allows data teams to build more powerful and accurate predictive models.

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