Calculate the Median of a Column in PySpark

Authored by

stats writer

November 16, 2025

Introduction: Understanding Median Calculation in PySpark

The ability to calculate descriptive statistics is fundamental when performing data analysis, especially within large-scale data environments like those managed by PySpark. The Median, a robust measure of central tendency, is often preferred over the mean when dealing with skewed distributions or outliers. Determining the median of a specific column within a PySpark DataFrame is a straightforward task, though it requires leveraging the appropriate built-in functions optimized for distributed processing.

In this comprehensive guide, we explore two primary methods available for computing the median value(s) in your dataset. Whether you need to find the median for a single numerical column or calculate it across several fields simultaneously, PySpark provides efficient and optimized functions designed for distributed computation. We will detail the specific syntax, explain the underlying logic, and demonstrate practical applications using a sample dataset designed for clarity.

You can utilize the following methods to calculate the median of a column or columns in a PySpark DataFrame:

Method 1: Calculating the Median for a Single Specific Column using agg()

The most efficient approach for calculating the median of just one column involves using the built-in aggregation function available in the functions module. This method requires importing the `functions` module, typically aliased as `F`, and applying the `F.median()` function within the DataFrame's `agg()` transformation. This sequence performs a single aggregation across the entire dataset to return the result efficiently.

This approach is particularly useful when you need to extract a single numerical result--a scalar value--for immediate reporting or for subsequent calculations outside the Spark environment. The use of `.collect()` is necessary to pull the distributed result back to the driver program, where it can be indexed to isolate the final median value.

from pyspark.sql import functions as F

#calculate median of column named 'game1'
df.agg(F.median('game1')).collect()

Method 2: Calculating Medians for Multiple Columns Simultaneously using select()

When the analytical requirement is to find the central tendency across several different numerical

variables and display these results as a new structure within the Spark execution environment, utilizing the `select()` transformation combined with imported statistical functions is the recommended path. This method allows for parallel calculation of medians across the specified columns, returning a resultant DataFrame rather than a single scalar value.

Unlike the first method, which relies on the `agg()` function for general aggregation, this technique specifically imports the `median` function directly from `pyspark.sql.functions` and applies it column-wise inside a `select()` call. The result, which is a one-row DataFrame, is typically displayed immediately using `.show()` for rapid diagnostic checks.

from pyspark.sql.functions import median

#calculate median for game1, game2 and game3 columns
df.select(median(df.game1), median(df.game2), median(df.game3)).show()

Prerequisites: Setting Up the PySpark Environment and Sample Data

Before diving into the practical examples, we must establish a working Spark session and define the DataFrame that will serve as our sample data. This setup ensures reproducibility and allows us to clearly illustrate how the methods detailed above operate on real data. We are simulating a small dataset tracking performance metrics across several fictional teams.

The provided examples will utilize this sample PySpark DataFrame. Note the structure: we have an identifier column ('team') and three numerical score columns ('game1', 'game2', 'game3') that represent the variables upon which we will compute the Median. Correct data typing is crucial here, as statistical functions require numerical input.

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|game1|game2|game3|
+-------+-----+-----+-----+
| Mavs| 25| 11| 10|
| Nets| 22| 8| 14|
| Hawks| 14| 22| 10|
| Kings| 30| 22| 35|
| Bulls| 15| 14| 12|
|Blazers| 10| 14| 18|
+-------+-----+-----+-----+
```

This resulting DataFrame, named `df`, is now properly initialized and ready for statistical processing. The consistent numerical data across the score columns ensures that the statistical aggregation functions will execute smoothly and produce accurate results.

Example 1: In-Depth Single Column Analysis and Verification

As detailed in Method 1, calculating the median for a single column requires the use of the `agg()` function combined with `F.median()`. This operation is highly optimized in Spark for large datasets. Because `df.agg()` returns a new DataFrame, we must use `.collect()` to extract the resulting median value as a primitive Python number, making it immediately usable for external reports or control flow logic.

We utilize the following syntax to calculate the median of values specifically contained within the **game1** column of the DataFrame. This method is the most precise way to obtain a single descriptive statistic efficiently across a potentially massive dataset, optimizing for minimum computation overhead.

```
from pyspark.sql import functions as F

#calculate median of column named 'game1'
df.agg(F.median('game1')).collect()

18.5
```

The execution of this script yields the precise value: **18.5**. This output represents the calculated median of all scores recorded in the **game1** column. PySpark's implementation of the median calculation is robust, correctly handling situations where the total count of records is an even number by averaging the two central values.

To ensure the accuracy of the result, we can manually verify the calculation based on the definition of the Median, which requires sorting the data points.

The raw scores for the **game1** column are: 25, 22, 14, 30, 15, 10. When sorted in ascending order, these values become: 10, 14, **15**, **22**, 25, 30. Since there are six (an even number) data points, the median is calculated as the average of the two middle values, 15 and 22. Calculating $(15 + 22) / 2$ results in **18.5**, confirming the correctness of the PySpark function output.

Example 2: Analyzing Medians Across Multiple Columns Concurrently

When analyzing datasets with multiple metrics, calculating multiple medians concurrently provides a quick comparative overview of the central tendencies of different variables. Method 2, utilizing `df.select()` in combination with the imported `median` function from the functions module, is ideal for this purpose. This approach yields a resulting DataFrame containing the summary statistics for each specified column, making cross-metric comparisons intuitive.

We apply the following syntax to calculate the median values simultaneously for the **game1**, **game2**, and **game3** columns. Notice that unlike the previous example, when using `select()`, we reference the DataFrame columns directly using dot notation (`df.gameX`) rather than string literals, which is standard practice when applying functions from the functions module.

from pyspark.sql.functions import median

#calculate median for game1, game2 and game3 columns
df.select(median(df.game1), median(df.game2), median(df.game3)).show()

```
+-------------+-------------+-------------+
|median(game1)|median(game2)|median(game3)|
+-------------+-------------+-------------+
| 18.5| 14.0| 13.0|
+-------------+-------------+-------------+
```

The resulting output is a temporary DataFrame with a single row, displaying the calculated median for each column analyzed. This structure is highly beneficial for quick inspection and direct comparison of the central tendencies of different metrics within the dataset, demonstrating efficiency in parallel processing.

The interpretation of these aggregated results provides immediate statistical insights:

The median of values in the **game1** column is **18.5**, confirming the result from Example 1.
The median of values in the **game2** column is **14.0**.
The median of values in the **game3** column is **13.0**.

This approach highlights the power of Spark's vectorized operations, allowing for rapid parallel aggregation across numerous columns without requiring iterative loops or complex manual coding, which is crucial for big data processing environments where time is a critical factor.

Statistical Considerations and Handling Null Values

Understanding how the median function handles data quality issues, such as missing values, is essential for accurate analysis. The Median function in PySpark is designed to be robust against missing data points, following standard statistical practice for aggregation functions in data processing frameworks.

Note: If there are null values (NaN or None) present in the column being processed, the **median** function will automatically ignore these values by default during the calculation. This behavior is critical because it prevents missing data from arbitrarily skewing the measure of central tendency. Only valid, non-null numerical entries contribute to the final sorted list used for determining the middle element. If all values in a column are null, the median will return null.

The median's inherent resistance to outliers makes it superior to the mean in many business and scientific contexts. While the mean can be significantly affected by a few extreme scores, the median remains a stable representation of the 50th percentile of the data distribution. This robustness is why data scientists frequently rely on the median when reporting typical performance metrics, especially in datasets prone to measurement errors or heavy skewness.

Conclusion and Best Practices

Calculating the median in PySpark is a fundamental and straightforward step in exploratory data analysis and data profiling. By leveraging the optimized functions within the `pyspark.sql.functions` module, users can efficiently determine this central metric across vast, distributed datasets. The choice between Method 1 (using `agg()` for a single column scalar result) and Method 2 (using `select()` for a multi-column DataFrame result) depends entirely on the required output format and subsequent steps in your analytical workflow.

As a final best practice, always ensure that the columns used for median calculation are of an appropriate numerical type (Integer, Long, or Double). Attempting to calculate the median on string or non-numeric columns will result in runtime errors, halting the Spark job. Adherence to these

methods ensures that your statistical aggregation on your DataFrame is accurate, performant, and aligned with distributed computing best practices. Understanding how PySpark handles nulls by default simplifies data cleaning preprocessing when focusing on central tendency measures.