How to Calculate the Mean of a Column in PySpark Easily

Authored by

stats writer

February 8, 2026

Calculating the mean (or average) of a specific column is a fundamental operation in PySpark, essential for performing basic data analysis and understanding the central tendency of a dataset. PySpark, the Python API for Apache Spark, provides highly optimized methods within the DataFrame API to handle these calculations efficiently, even when dealing with massive datasets distributed across a cluster. Mastering these techniques is crucial for anyone involved in large-scale data processing and statistical analysis.

This guide will explore the primary techniques available for computing the mean of one or multiple columns. We will focus specifically on two powerful methods: utilizing the built-in aggregate function (`agg`) and employing the `select` transformation combined with standard SQL functions. These approaches ensure both accuracy and performance when analyzing numerical data stored within a DataFrame structure.

Calculate the Mean of a Column in PySpark

Overview of PySpark Aggregation Methods

When working with Spark, the core data structure is the DataFrame, which offers several statistical functions natively. The most direct way to calculate the mean is through the use of aggregation operations. An aggregation reduces a collection of values (a column) into a single summary value, representing a crucial step in deriving descriptive statistics.

The two primary mechanisms for achieving this calculation efficiently in PySpark are:

Using the **`agg()`** method: This powerful method allows you to apply one or more aggregation functions (like `mean`, `sum`, `max`) across specified columns of the entire DataFrame, returning a single-row result DataFrame containing the calculated averages. This is typically the cleanest approach for calculating a simple average and extracting the result as a scalar value.

Using **`select()`** combined with SQL functions: For calculating multiple column means simultaneously, especially if you wish to retain the output as a compact DataFrame, importing and using the specific statistical function (e.g., `F.mean()`) directly within a `select` statement is highly efficient and flexible.

Alternatively, the `describe()` function can also be employed to quickly profile numerical columns. While `describe()` is optimized for generating comprehensive descriptive statistics (including count, standard deviation, min, and max), it does include the calculated mean value, making it an excellent tool for initial data exploration.

Setting Up the Example PySpark DataFrame

To demonstrate these calculation methods practically, we must first establish a sample DataFrame. This DataFrame simulates scoring results across multiple games for various sports teams. We utilize the built-in PySpark `SparkSession` to manage the distributed environment and load the structured data from a Python list of lists.

The following code block shows the necessary imports, data definition, and the creation of the Spark DataFrame, which will serve as our foundation for all subsequent mean calculations.

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 successfully created DataFrame contains six records and four columns, providing the numerical data points (`game1`, `game2`, `game3`) required to execute the aggregation functions demonstrated in the subsequent examples.

Method 1: Calculating the Mean for a Single Column using `agg()`

To calculate the average score for a single column, such as `game1`, the most expressive and direct approach is to use the aggregate function (`agg`) combined with the `mean` function imported from `pyspark.sql.functions` (often aliased as `F`). This method is preferred when the objective is to calculate and extract a single statistical value quickly.

The syntax below applies the mean calculation to the `game1` column. Since `df.agg()` returns a new single-row DataFrame, we must use `.collect()` to access the underlying scalar value, converting the distributed result back into a usable Python float for reporting or further computation.

from pyspark.sql import functions as F

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

19.333333333333332

The result shows the average score for the **game1** column is precisely **19.333333333333332**. This high-precision result is characteristic of Spark's internal handling of floating-point arithmetic.

Verification and Mathematical Accuracy

It is standard practice in data analysis to verify statistical results, especially those derived from complex distributed systems like PySpark. We can manually confirm the mean for the `game1` column by applying the basic arithmetic mean formula: sum of all values divided by the count of values.

The data points in the `game1` column are (25, 22, 14, 30, 15, 10), totaling six observations.

The steps for manual calculation are as follows:

Summation: 25 + 22 + 14 + 30 + 15 + 10 = 116.

Division: 116 / 6 = 19.333333333333332.

The manual result confirms the output provided by the aggregate function, demonstrating the fidelity of Spark's statistical implementation. The mean of values in **game1** is indeed **19.333**.

Method 2: Calculating Means for Multiple Columns Simultaneously

When the analytical requirement is to calculate the average across several numerical columns at once, using the `select()` transformation in combination with imported SQL functions is often the most readable approach. This method avoids the need for chained `agg()` statements or complex output parsing.

We import the specific `mean` function and pass multiple mean expressions to the `select` method, treating each calculation as a new derived column in the resulting DataFrame. This is ideal for generating quick summary statistics for a feature set.

from pyspark.sql.functions import mean

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

```
+-----------------+-----------------+----------+
| avg(game1)| avg(game2)|avg(game3)|
+-----------------+-----------------+----------+
|19.333333333333332|15.166666666666666| 16.5|
+-----------------+-----------------+----------+
```

The output table displays the averages for all three game columns in a concise format. Spark automatically aliases the resulting column names using the convention `avg(columnName)`.

From this output, we can extract the specific calculated averages:

The mean of values in the **game1** column is **19.333**.

The mean of values in the **game2** column is approximately **15.167**.

The mean of values in the **game3** column is exactly **16.5**.

Handling Missing Data (Null Values) During Mean Calculation

A crucial aspect of calculating statistical measures in production environments is understanding how the system manages missing data. In PySpark, missing values are represented by `null`.

By default, all statistical aggregation functions, including the `mean` function, are designed to automatically exclude `null` values from their computation. This ensures that the calculated average accurately reflects the average of only the existing, valid numerical entries within the column. For instance, if a column has 10 values but 2 are null, the denominator for the mean

calculation will be 8 (the count of non-null values), not 10.

Note: If there are `null` values in the column, the **mean** function will ignore these values by default. If the column contains only `null` values, the resulting mean calculation will also yield a `null` value. If analysts require a different treatment (e.g., treating missing scores as zero), an explicit data manipulation step, such as using `df.fillna(0, subset=)`, must be applied before the aggregation is executed.

Conclusion and Further Exploration of Descriptive Statistics

Calculating the mean is a foundational step in any data exploration process. PySpark offers streamlined, powerful methods--specifically `agg()` and `select()` with SQL functions--to achieve this calculation rapidly and accurately, scaling effortlessly to big data environments.

Beyond the mean, the PySpark SQL functions library includes numerous other valuable statistical operations that contribute to robust descriptive statistics. Analysts are encouraged to explore functions like `stddev` (standard deviation), `min`, `max`, and `count` to gain a much richer understanding of the distribution, variability, and structure of their datasets.

The ability to perform these efficient, distributed calculations is what makes PySpark an indispensable tool for modern data processing.

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