Calculate the Mean of a Column in PySpark

Authored by

stats writer

November 16, 2025

Why Calculate the Mean in PySpark?

In the realm of big data processing, accurately calculating descriptive statistics is fundamental. The arithmetic average, or Mean, serves as a crucial measure of central tendency, offering immediate insights into the typical value within a dataset. When working with massive, distributed datasets, tools like PySpark DataFrame become indispensable for efficient parallel computation. Understanding how to compute the mean across specific columns allows data scientists and engineers to quickly characterize data distribution, identify potential outliers, and prepare features for machine learning models.

PySpark, the Python API for Apache Spark, provides highly optimized functions for performing these statistical calculations across distributed clusters. Unlike traditional Python libraries that struggle with data volumes exceeding local memory capacity, PySpark leverages its architecture to handle petabytes of data seamlessly. Specifically, the calculation of the mean is achieved using built-in SQL functions optimized for execution across Spark executors, ensuring both speed and scalability. This distributed computing approach is essential for modern data workflows where datasets are often too large to fit on a single machine.

This guide details the two primary, highly efficient methods available in PySpark for determining the average value of numerical columns. Whether you need the average of a single metric or need to concurrently compute averages across dozens of variables, PySpark offers concise and powerful syntax to achieve your goals, utilizing the built-in efficiency of the Spark engine. These techniques are cornerstones of data analysis and preliminary exploration in any Spark environment.

Essential Methods for Mean Calculation

PySpark offers flexibility in performing column-wise statistical aggregation. The choice between methods often depends on the desired output format and the quantity of columns being analyzed. The first approach, utilizing the `.agg()` method, is particularly suited for returning a scalar value or performing complex, chained aggregations. The second approach, leveraging the `.select()` transformation alongside imported functions, is ideal for generating a new DataFrame containing multiple summary statistics.

The `.agg()` method treats the entire DataFrame as a single group, applying the specified statistical function (like `mean()`) across the selected column. This method is highly recommended when the goal is simply to retrieve the single statistical result, often followed by a local retrieval command like `.collect()`. It is robust, clear, and adheres closely to standard SQL aggregation practices.

Conversely, the `.select()` method, paired with the `mean` function imported from

pyspark.sql.functions, allows users to define the output schema explicitly. This is crucial for calculating the mean across several columns simultaneously, resulting in a new DataFrame row where each column holds the average of its corresponding input column. Both methods rely on the optimized backend functions provided by PySpark.

Method 1: Calculate Mean for One Specific Column

from pyspark.sql import functions as F

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

Method 2: Calculate Mean for Multiple Columns

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()

Setting Up the PySpark Environment and Sample Data

Before executing any statistical commands, we must establish a working Spark session and load the data into a PySpark DataFrame. The data structure utilized in the following examples represents hypothetical scores from multiple games recorded for various teams. This structured format is typical for analytical tasks where we need to find the average performance across different metrics (games).

The setup involves importing the necessary `SparkSession`, defining the raw data as a list of lists, and specifying the column names. Crucially, the `spark.createDataFrame(data, columns)` command converts this local Python data structure into a distributed, schema-aware PySpark object, ready for distributed processing. Once the DataFrame (named `df`) is created, displaying it using `df.show()` verifies that the data has been loaded correctly and the schema inferred as intended.

It is essential to understand that all subsequent operations will be lazy transformations until an action, such as `.show()` or `.collect()`, is called. The following examples rely on this established DataFrame structure for demonstrating the two distinct methods of calculating the mean.

The following examples show how to use each method in practice with the following PySpark DataFrame:

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


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

The `.agg()` method is the canonical approach in PySpark for performing a single, whole-dataset aggregation. When focusing on a specific metric, such as the scores in **game1**, this method provides the cleanest path to obtaining the final scalar value. We import the required functions module, typically aliased as `F`, for succinct access to statistical utilities like `F.mean()`.

In this example, `df.agg(F.mean('game1'))` first calculates the average score for the specified column across all records in the DataFrame. This transformation yields a new DataFrame with a single row and a single column containing the resulting mean. Since we require the actual

numerical value for immediate use in our local Python environment, we chain the `.collect()` action.

The `.collect()` action retrieves the result from the distributed environment back to the driver program as a Python list of Row objects. Accessing then extracts the numerical value from the first (and only) row and the first (and only) column of the resulting DataFrame, providing the final, precise average.

from pyspark.sql import functions as F

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

19.333333333333332

The calculated mean for the **game1** column is confirmed to be approximately **19.333**. This result is derived by applying the aggregation function across the entire column partition. This method is highly effective for focused statistical analysis requiring immediate scalar output.

Understanding PySpark Aggregation and Data Retrieval

Understanding the mechanism behind `.agg()` is crucial for optimizing PySpark workflows. The aggregation operation requires shuffling data across the cluster to compute the global average, making it a computationally intensive action. However, Spark's query optimizer ensures this process is executed with maximum efficiency, often leveraging memory caching and optimized execution plans.

The use of `.collect()`, while necessary here to retrieve the single mean value, should be approached with caution in production environments. `.collect()` forces all distributed data results back onto the driver node. For single statistical results, this is fine, but if applied carelessly to large datasets, it can lead to driver memory overflow (an OutOfMemoryError). The standard practice is to use actions like `.collect()` only when the expected result size is small, as is the case when retrieving a single aggregated metric.

The alternative output method, as seen in Example 2, uses `.show()`, which is safer as it only displays a limited preview of the resulting DataFrame, rather than attempting to load the entire result set into the driver program's memory. Developers must always be mindful of the difference between DataFrame transformations (like `.agg()` or `.select()`) and DataFrame actions (like `.collect()` or `.show()`).

Example 2: Calculating the Mean Across Multiple Columns using select()

While `.agg()` is excellent for single-value retrieval, calculating descriptive statistics for numerous columns simultaneously is often achieved more elegantly using the `.select()` transformation combined with the imported `mean` function. This approach treats the calculation as a projection, creating a new, highly specialized DataFrame containing the results.

By importing `mean` directly from pyspark.sql.functions, we can apply it directly within the `.select()` statement for each column we wish to analyze (`df.game1`, `df.game2`, `df.game3`). This generates three new columns in the resulting DataFrame, each containing the average score for the respective game. The resulting column names are automatically prefixed with `avg()` unless aliasing is explicitly used.

This method is preferable when the user requires the results to remain within the PySpark ecosystem for further chaining operations, such as joining the summary statistics back to the original data or writing the results to a distributed file system like HDFS or S3. The use of `.show()` here provides a clean, formatted table output demonstrating the simultaneous calculation across the three performance metrics.

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 from the pyspark.sql.functions approach provides the averages for all specified columns in a structured table.

The mean of values in the **game1** column is **19.333**.
The mean of values in the **game2** column is **15.167**.
The mean of values in the **game3** column is **16.5**.

Addressing Null Values and Data Quality in Mean Calculation

A critical consideration when calculating statistical metrics on real-world data is how the function handles missing data. In SQL-based environments like PySpark, missing data is typically

represented by null values. Understanding PySpark's default behavior regarding these nulls is essential for ensuring data integrity and accurate statistical reporting.

Note: If there are null values in the column, the **mean** function will ignore these values by default.

This default behavior, known as "skip nulls," means that the calculation only considers non-null data points. For instance, if a column has 10 rows but 2 are null, the average will be calculated by summing the 8 non-null values and dividing by 8, not 10. This is generally the desired behavior in descriptive statistics, as nulls represent unknown or irrelevant data points rather than a zero value that should factor into the average.

If, however, the intent is to impute or replace null values before calculation (e.g., replacing them with zero or the column median), the user must explicitly apply cleaning transformations using methods like `df.na.fill()` before calling `.mean()`. Failure to preprocess nulls when zero values are expected could lead to an inflated average that does not accurately represent the true data distribution, especially when dealing with sparse datasets. Always verify data quality before performing statistical analysis.

Summary of PySpark Aggregation Techniques

The methods demonstrated--using `.agg()` for focused scalar retrieval and `.select()` for multi-column DataFrame output--provide powerful, scalable ways to calculate the mean in PySpark. Both rely on the highly efficient core functions within the pyspark.sql.functions module, ensuring that even petabyte-scale datasets can be processed rapidly across a distributed cluster.

The key takeaway is choosing the method appropriate for your workflow: use `.agg()` combined with `.collect()` when you need the final average value immediately in the driver program, and use `.select()` combined with `F.mean()` when you need the results to remain within the Spark execution plan for subsequent transformations or when calculating statistics for many columns simultaneously. Mastering these foundational aggregation techniques is crucial for advanced data manipulation and statistical modeling within the PySpark ecosystem.