How to Easily Calculate Summary Statistics with PySpark

Authored by

stats writer

January 3, 2026

PySpark serves as the crucial Python API for Apache Spark, providing analysts and data scientists with a robust and scalable framework for big data processing. Before diving into complex modeling or deep learning, understanding the fundamental characteristics of your dataset is paramount. This initial exploration relies heavily on calculating summary statistics, which offer quick insights into the distribution, central tendency, and variability of the data. While PySpark offers granular functions like `select()` and `agg()` for custom statistical computations, the built-in `.summary()` method provides an efficient, all-in-one approach to descriptive analysis on a DataFrame.

This guide will walk through three distinct yet highly useful methods for generating descriptive statistics for columns within a PySpark DataFrame. We will illustrate how to retrieve a comprehensive default set of metrics, how to focus on specific quantiles and extreme values, and crucially, how to isolate and analyze only the numerical features, preventing misinterpretations associated with applying mathematical operations to string data types. Mastering these techniques is fundamental for efficient data quality assessment and feature engineering in a distributed computing environment.

The PySpark DataFrame and Descriptive Analysis

A PySpark DataFrame is essentially a distributed collection of data organized into named columns. Its structure is similar to a traditional relational database table, but it is optimized for parallel processing across a cluster of machines. When performing descriptive analysis, the goal is often to quickly determine metrics like the mean, median, minimum, and maximum values for quantitative columns, while also understanding the count and distribution of categorical fields.

The core utility for automated descriptive analysis in PySpark is the `.summary()` method. Unlike the lower-level `.describe()` function, `.summary()` offers a slightly richer set of outputs, including percentiles (quartiles) by default, which are essential for identifying skewness and potential outliers without needing complex custom aggregations. This is particularly valuable when dealing with massive datasets where manually inspecting individual records is impractical or impossible. Understanding which method to use depends entirely on the required level of detail and the data types involved in the analysis.

You can leverage the following standardized patterns to generate summary statistics across different scopes within your PySpark DataFrame:

Method 1: Calculate Comprehensive Summary Statistics for All Columns. This provides the default set of metrics for every column, regardless of data type.
Method 2: Calculate Specific Summary Statistics for All Columns. This allows the user to specify a subset of metrics (often quantiles) to be calculated.
Method 3: Calculate Summary Statistics for Only Numeric Columns. This involves filtering the

DataFrame columns based on their data type before applying the statistical calculation, yielding cleaner and more meaningful results for quantitative analysis.

Prerequisites: Setting Up the PySpark Environment

To demonstrate the functionality of the `.summary()` method, we must first initialize a SparkSession and create a sample DataFrame. Our example dataset contains information about various basketball players, including their team affiliation, conference, points scored, and assists made. Note the mixture of string (categorical) and numeric (quantitative) data types, which is crucial for illustrating the differences between the three summary methods.

The following code snippet handles the creation of the Spark session, definition of the raw data (a list of lists), definition of the column schema, and the final construction of the distributed DataFrame. We then display the contents of the DataFrame to verify the structure before proceeding with the statistical analysis.

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|conference|points|assists|
+----+----------+------+-------+
A	East	11	4
A	East	8	9
A	East	10	3

B	West	6	12
B	West	6	4
C	East	5	2
+----+----------+------+-------+

This DataFrame, `df`, is now ready for analysis. We observe two string columns (`team` and `conference`) and two integer columns (`points` and `assists`). The challenge, particularly when dealing with heterogeneous datasets, is ensuring that statistical functions are applied appropriately based on the underlying data type. The following methods demonstrate different strategies to achieve accurate descriptive reporting.

Method 1: Calculating Comprehensive Summary Statistics (`df.summary().show()`)

The simplest application of descriptive statistics in PySpark is calling the `.summary()` function without any arguments. This instructs Spark to calculate a default, predefined set of metrics for every column present in the DataFrame. This method is excellent for a rapid, initial inspection of the entire dataset.

We execute this calculation using the syntax shown below. Notice that the operation generates a new DataFrame where the first column, named `summary`, contains the names of the calculated statistics (e.g., count, mean, min). Subsequent columns correspond to the original fields in the dataset.

df.summary().show()

Applying this method to our basketball player DataFrame yields the following output, providing a comprehensive overview of both numerical and categorical fields:

#calculate summary statistics for each column in DataFrame
df.summary().show()

+-------+----+----------+-----------------+------------------+
|summary|team|conference| points| assists|
+-------+----+----------+-----------------+------------------+
count	6	6	6	6
mean	null	null	7.666666666666667	5.666666666666667
stddev	null	null	2.422120283277993	3.9327683210007005
min	A	East	5	2
25%	null	null	6	3

50%	null	null	6	4
75%	null	null	10	9
max	C	West	11	12
+-------+----+----------+-----------------+------------------+

Interpreting the Full Summary Output

The output table from the default `.summary()` method provides eight key metrics. It is crucial to understand how these metrics apply differently based on the column's data type. For numerical columns (`points` and `assists`), all statistics are meaningful and accurately calculated. For categorical or string columns (`team` and `conference`), only the `count`, `min`, and `max` values have sensible interpretations, while aggregate metrics like `mean`, `stddev`, and percentiles (25%, 50%, 75%) are returned as `null`.

The statistics provided include:

count: Represents the total number of non-null records in the column, verifying completeness.
mean: The arithmetic average of the values (only calculated for numeric types).
stddev: The standard deviation, indicating the dispersion of the data points around the mean (only calculated for numeric types).
min: The absolute smallest value found in the column. For strings, this is the alphabetically smallest value.
25%: The first quartile (Q1), meaning 25% of the data falls below this value.
50%: The median or second quartile (Q2), meaning 50% of the data falls below this value. This is the central tendency of the dataset, less sensitive to outliers than the mean.
75%: The third quartile (Q3), meaning 75% of the data falls below this value.
max: The absolute largest value found in the column. For strings, this is the alphabetically largest value.

For our numerical columns, we can quickly observe that the average `points` scored is approximately 7.67, with a standard deviation of 2.42, indicating moderate variability. The 50th percentile (median) for both `points` (6) and `assists` (4) is slightly lower than the mean, suggesting a potential slight positive skew. The string columns, on the other hand, show that teams range from 'A' to 'C' and conferences from 'East' to 'West' based on alphabetical sorting.

Method 2: Calculating Specific Quantiles and Extremes (Custom Input)

While the default `.summary()` method is useful, data analysts often require only a specific subset of metrics, typically focused on the distribution boundaries and quartiles, especially during exploratory data analysis (EDA). PySpark allows users to pass a list of desired statistics as string

arguments directly to the `.summary()` function.

When providing arguments, the available statistics generally include `count`, `mean`, `stddev`, `min`, `max`, and any custom percentile specified as a string percentage (e.g., `'25%'`, `'90%'`). If you only request quantiles and extremes, the output is much cleaner and focuses strictly on the distribution spread.

In the following example, we specifically request the minimum, maximum, and the three standard quartiles (25%, 50%, and 75%).

df.summary('min', '25%', '50%', '75%', 'max').show()

Executing this tailored command significantly reduces the output noise, focusing entirely on the requested distribution markers:

#calculate specific summary statistics for each column in DataFrame
df.summary('min', '25%', '50%', '75%', 'max').show()

```
+-------+----+----------+------+-------+
|summary|team|conference|points|assists|
+-------+----+----------+------+-------+
| min| A| East| 5| 2|
| 25%|null| null| 6| 3|
| 50%|null| null| 6| 4|
| 75%|null| null| 10| 9|
| max| C| West| 11| 12|
+-------+----+----------+------+-------+
```

While this approach is more targeted, it still includes the string columns (`team` and `conference`), resulting in `null` values for the percentile calculations. For true quantitative analysis, filtering the DataFrame to only include numeric fields is the most professional and accurate method, as demonstrated in the final example.

Method 3: Focusing Analysis on Numeric Columns (Data Type Filtering)

In most real-world data science tasks involving PySpark, the primary requirement is to calculate metrics like mean and standard deviation exclusively for numerical features. Applying these mathematical statistics to categorical or string fields results in meaningless `null` outputs, cluttering the results and potentially leading to errors if the output is piped into subsequent processing stages.

To address this, we first need to dynamically identify which columns in the DataFrame are numeric. This is achieved by inspecting the DataFrame schema using the `df.dtypes` attribute, which returns a list of (column name, data type) tuples. We then use a Python list comprehension to filter this list, excluding columns that start with the data type 'string' (or any other non-numeric type). Once the list of `numeric_cols` is generated, we use the `select()` method to subset the DataFrame before applying the `.summary()` calculation.

This two-step process ensures that the statistical computation is executed solely on relevant quantitative features:

#identify numeric columns in DataFrame
numeric_cols =

#calculate summary statistics for only the numeric columns
df.select(*numeric_cols).summary().show()

```
+-------+------------------+------------------+
|summary|            points|           assists|
+-------+------------------+------------------+
|  count|                 6|                 6|
|   mean| 7.666666666666667| 5.666666666666667|
| stddev|2.422120283277993|3.9327683210007005|
|    min|                 5|                 2|
|    25%|                 6|                 3|
|    50%|                 6|                 4|
|    75%|                10|                 9|
|    max|                11|                12|
+-------+------------------+------------------+
```

Notice that summary statistics are displayed exclusively for the two quantitative columns in the DataFrame: the **points** and **assists** columns. This filtered output is clean, actionable, and suitable for further automated processing or reporting dashboards. This technique represents the best practice for generating descriptive statistics within large, complex PySpark environments.

Conclusion and Further Resources

The ability to quickly and accurately calculate descriptive statistics is a core skill for anyone working with distributed data using PySpark. The `.summary()` method offers a versatile and scalable tool for initial data assessment, whether you require a full overview of all fields or a focused analysis solely on numerical dimensions.

We have reviewed three crucial methods: the standard comprehensive summary (Method 1), the quantile-focused custom summary (Method 2), and the critical technique of filtering for numeric columns prior to calculation (Method 3). Utilizing the filtering technique (Method 3) ensures data integrity and clarity, which is essential when preparing data for machine learning or detailed statistical modeling.

For users seeking more advanced statistical calculations, PySpark provides flexibility through the `pyspark.sql.functions` module, allowing for custom aggregations using functions like `mean()`, `min()`, `max()`, and `stddev()` combined with the `.agg()` method. However, for rapid, standardized descriptive analysis, the `.summary()` function remains the most efficient choice. We encourage readers to consult the official documentation for a deeper understanding of all parameters and capabilities of the PySpark **summary** function.

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