How to Calculate Quartiles Easily with PySpark

Authored by

stats writer

January 20, 2026

In the realm of statistics and data analysis, **quartiles** are fundamental measures used to characterize the distribution of a numerical feature. They represent values that effectively split a sorted **dataset** into four segments, each containing an equal number of observations. Understanding these division points is crucial for performing effective exploratory data analysis (EDA) and identifying potential outliers or skewness in the data distribution.

When we analyze a statistical distribution, we are primarily interested in three specific quartile values that mark the boundaries of these four segments:

First Quartile (**Q1**): This value is located at the 25th **percentile**. It signifies that 25% of the data points fall below this value.
Second Quartile (**Q2**): This is the central value, located at the 50th percentile. It is synonymous with the **median** of the dataset.
Third Quartile (**Q3**): This value is located at the 75th percentile, meaning 75% of the data points are less than or equal to this value.

Calculating these statistics efficiently in large-scale, distributed computing environments like **PySpark** requires specialized tools. PySpark provides the powerful `approxQuantile` function, which allows for the rapid estimation of these boundary points even when dealing with massive datasets distributed across a cluster.

How PySpark Calculates Quartiles Using approxQuantile

Due to the inherent complexity and high computational cost of calculating exact quantiles in a distributed manner, PySpark employs the `approxQuantile` method. This function computes approximate quantiles based on a high-performance algorithm that provides a highly accurate estimate while minimizing shuffle operations across the cluster. This approximation method is crucial for maintaining scalability and performance when processing **PySpark DataFrames**.

The function requires three key parameters: the name of the column you wish to analyze, an array specifying the required quantile probabilities (e.g., for the three quartiles), and a relative error parameter (epsilon). The relative error parameter controls the trade-off between accuracy and computation time. A smaller error value results in a more precise estimate but takes longer to compute. Setting this error to zero (0) attempts to compute the exact quantiles, although this is generally discouraged for performance reasons in large-scale production environments.

You can use the following syntax structure to calculate the standard quartiles for a specified numerical column within your PySpark DataFrame:

 # Calculate quartiles of the 'points' column
df.approxQuantile('points', , 0)

Detailed Example: Calculating Quartiles in PySpark

To demonstrate the practical application of the `approxQuantile` function, let us work with a sample DataFrame. Suppose we have compiled data detailing the scores achieved by various basketball teams during a recent set of games. This DataFrame, which we will name `df`, contains team identifiers and their corresponding scores, stored in the `points` column.

The following code snippet illustrates the necessary steps to initialize a Spark Session, define the data structure, and create the PySpark DataFrame for our analysis. This setup is a prerequisite for any data manipulation tasks executed within the Spark ecosystem.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define the sample data for basketball scores
data = ,
,
,
,
,
,
,
,
]

Define column names: 'team' (string) and 'points' (numeric)
columns =

# Create the PySpark DataFrame
df = spark.createDataFrame(data, columns)

# Display the resulting DataFrame structure and contents
df.show()

+-------+------+
| team|points|
+-------+------+
Mavs	18
Nets	33
Lakers	12

| Kings| 15|
| Hawks| 19|
|Wizards| 24|
| Magic| 28|
| Jazz| 40|
|Thunder| 24|
| Spurs| 13|
+-------+------+

Applying the approxQuantile Function to the Data

Once the DataFrame is successfully initialized, we can proceed to calculate the quartiles for the `points` column. This calculation provides immediate insight into the distribution of scores, showing us where the bulk of the data lies and identifying the central tendency without relying solely on the mean, which can be sensitive to extreme values.

We utilize the `approxQuantile` function, specifying `'points'` as the target column. We request the 25th, 50th, and 75th percentiles using the array . For this small example, we set the relative error to `0`, seeking the exact calculation, although in massive real-world data applications, a non-zero error tolerance (e.g., 0.01) is often preferred for performance gains.

Executing the following command yields the three crucial quartile values:

 # Calculate the three main quartiles of the 'points' column
df.approxQuantile('points', , 0)

Interpreting the Quartile Results

The resulting list, , directly corresponds to Q1, Q2, and Q3, respectively, based on the order of probabilities provided in the input array. Interpreting these results provides a comprehensive summary of the score distribution across the teams in our sample data.

The first **quartile** (Q1) is **15.0**. This means that 25% of the teams scored 15 points or fewer.
The second quartile (Q2), or the median, is **19.0**. Exactly half (50%) of the teams scored below 19 points, and half scored above it.
The third **quartile** (Q3) is **28.0**. This indicates that 75% of the observed scores fall at or below 28 points.

Calculating the Interquartile Range (IQR)

A critical statistic derived directly from the quartiles is the **Interquartile Range (IQR)**. The IQR measures the spread of the middle 50% of the data and is calculated simply as the difference between the third quartile (Q3) and the first quartile (Q1). It is a robust measure of statistical dispersion, meaning it is less susceptible to the influence of extreme data points or outliers compared to the standard deviation.

In our example, the IQR is calculated as: IQR = Q3 - Q1 = 28.0 - 15.0 = 13.0. This value tells us that the central half of the basketball scores spans a range of 13 points. A smaller IQR suggests the data points are tightly clustered around the median, while a larger IQR indicates greater variability in the central distribution.

Advanced Considerations for PySpark Quantiles

When working with massive data in a production environment, data scientists rarely set the relative error parameter to zero. The strength of the `approxQuantile` function lies in its ability to trade off minor accuracy for massive performance gains. If you set the relative error (epsilon) to 0.01, for instance, the resulting quartile estimates are guaranteed to be accurate within 1% of the true value, which is usually sufficient for large-scale statistical modeling and business intelligence reporting.

It is essential to understand that this function returns an array of estimated quantile values. Since the input probabilities are provided as an array, the output maintains the same sequential order. For example, if you request the 10th and 90th **percentile**s (), the first element of the output array will be the 10th percentile, and the second will be the 90th percentile. This consistency is vital for programmatic access and robust data pipeline construction using **PySpark DataFrames**.

The use of quartiles and related descriptive statistics provides a quick yet powerful summary of any numerical variable in a distributed system, enabling fast detection of data anomalies or shifts in distribution over time.

Further PySpark Resources and Documentation

For deep technical reference on the statistical methods employed by Apache Spark, particularly concerning distributed quantile calculation, consult the official documentation. The specifics regarding the implementation and performance characteristics of the `approxQuantile` function are detailed within the **PySpark** API documentation.

Understanding how to leverage functions like `approxQuantile` is crucial for any data engineer or scientist working with big data to ensure both computational efficiency and statistical integrity.

Note: You can find the complete documentation for the PySpark **approxQuantile** function here.

Related PySpark Tutorials

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