Calculate Percentiles in PySpark (With Examples)

Authored by

stats writer

November 16, 2025

Introduction: Mastering Percentile Calculation in PySpark

The ability to calculate statistical measures efficiently is paramount when dealing with large datasets. In the realm of big data processing, PySpark serves as a powerful engine for executing complex analytical operations. One fundamental statistical requirement is the calculation of percentiles, which are essential for understanding the distribution and spread of data within a column.

A percentile defines the value below which a given percentage of observations falls. For instance, the 25th percentile (or first quartile) indicates that 25% of the data points are less than or equal to that value. This measure is indispensable for **outlier detection**, summarizing data distributions, and performing quartile analysis across distributed computing environments.

This detailed guide explores the precise methods available in PySpark for calculating percentiles. We will demonstrate two primary approaches: calculating overall percentiles for a single column across the entire dataset, and calculating percentiles grouped by specific categories. Mastery of these techniques ensures you can derive meaningful insights from massive volumes of data stored in a PySpark DataFrame.

Overview of PySpark Percentile Methods

PySpark provides flexible mechanisms to perform percentile calculations using its SQL functions module. Regardless of the complexity of the desired calculation, the core method involves the use of the `percentile` expression within a DataFrame aggregation.

The following two principal methods allow users to tailor percentile calculation based on analytical needs, whether requiring a dataset-wide statistic or a segmented breakdown. Understanding the distinction between these methods is crucial for accurate data interpretation and efficient code execution in a distributed environment.

Method 1: Calculate Percentiles for One Column. This method provides a single statistical summary across the entire numerical column, ignoring any categorical grouping.

Method 2: Calculate Percentiles for One Column, Grouped by Another Column. This method segment the calculation by a categorical field (e.g., 'team'), resulting in a percentile value for each unique group.

Let's examine the foundational syntax for each method, highlighting how the `F.expr()` function is leveraged alongside the `percentile` function and the necessity of extracting the scalar result from the returned array.

Method 1: Syntax for Overall Percentile Calculation

import pyspark.sql.functions as F

#calculate 25th percentile for 'points' column
df.agg(F.expr('percentile(points, array(0.25))').alias('%25')).show()

Method 2: Syntax for Grouped Percentile Calculation

import pyspark.sql.functions as F

#calculate 25th, 50th and 75th percentile of 'points', grouped by 'team'
df_new = df.groupby('team').agg(F.expr('percentile(points, array(0.25))').alias('%25'),
F.expr('percentile(points, array(0.50))').alias('%50'),
F.expr('percentile(points, array(0.75))').alias('%75'))

Setting Up the PySpark Environment and Sample Data

To demonstrate these methods practically, we must first establish a functional SparkSession and load our sample data into a DataFrame. Our dataset contains information about various basketball players, crucial for illustrating how to analyze statistical measures across different categories like 'team' and 'position'.

The following setup code initializes the Spark environment, defines the raw data containing player information (team, position, points), and converts this data into a schema-defined DataFrame. This preparation step ensures the subsequent code examples run smoothly and provides context for the percentile outputs.

Pay close attention to the structure of the resulting DataFrame shown below, as it contains the **points** column--the numerical feature we will be calculating percentiles against--and the categorical **team** column necessary for our grouped 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|position|points|
+----+--------+------+
| A| Guard| 11|
| A| Guard| 8|
| A| Forward| 22|
| A| Forward| 22|
| B| Guard| 14|
| B| Guard| 14|
| B| Guard| 13|
| B| Forward| 7|
| C| Guard| 8|
| C| Forward| 5|
+----+--------+------+
```


Example 1: Calculating the 25th Percentile Across the Entire Dataset

When aiming for a high-level summary of the data distribution, we calculate the percentile without any grouping. This calculation treats the **points** column as a single, large pool of values distributed across all partitions of the cluster. We use the `agg()` function to apply the statistical transformation.

To calculate the 25th percentile, we specify `0.25` in the array argument of the `percentile` expression. Since the `percentile` function returns an array, we must use the array index `` immediately after the expression to extract the singular resulting value. We then assign a clear

alias, `%25`, to the new column using the `alias()` method.

Executing the code below provides a quick, robust measure of the first quartile for scoring performance across all players in the dataset, irrespective of their team affiliation. This is a crucial step in understanding the overall minimum scoring expectations.

import pyspark.sql.functions as F

#calculate 25th percentile for 'points' column
df.agg(F.expr('percentile(points, array(0.25))').alias('%25')).show()

```
+----+
| %25|
+----+
|15.0|
+----+
```

From this output, we observe that the 25th percentile of values in the **points** column is **15.0**. Statistically, this means that 25% of the observed points totals fall at or below 15 points.

Example 2: Grouped Percentile Calculation for Team Comparison

To perform comparative analysis, we need to segment our data before calculating the statistics. We use the `groupby('team')` method to create groups based on team affiliation. Subsequent aggregation is then applied independently to each group, allowing us to compare score distributions across different teams.

In this example, we calculate three crucial percentiles--the 25th, 50th (median), and 75th percentiles--all within a single `agg()` call. Each quartile requires its own `F.expr()` definition, ensuring that the results are distinctly separated and labeled in the final output DataFrame. This method is highly effective for calculating statistical summaries like the **Five-Number Summary** for segmented data.

This approach is particularly valuable for performance benchmarking, identifying which teams have a higher overall median score, or analyzing the spread (Interquartile Range) of scores within individual teams. The resulting DataFrame, `df_new`, will have one row per unique team, containing the calculated quartiles for that team's points.

import pyspark.sql.functions as F

#calculate 25th, 50th and 75th percentile of 'points', grouped by 'team'
df_new = df.groupby('team').agg(F.expr('percentile(points, array(0.25))').alias('%25'),

```
F.expr('percentile(points, array(0.50))').alias('%50'),
F.expr('percentile(points, array(0.75))').alias('%75'))

#view new DataFrame
df_new.show()

+----+----+----+----+
|team| %25| %50| %75|
+----+----+----+----+
| A|15.0|22.0|22.0|
| B|15.0|28.0|31.0|
| C| 6.5| 7.5| 8.0|
+----+----+----+----+
```

Interpreting Grouped Results and Statistical Insights

The output table from Example 2 provides valuable comparative statistics. By examining the quartiles for each team, we can draw immediate conclusions about the central tendency and spread of their performance data. This segmented analysis moves beyond simple averages, offering a more nuanced view of the score distribution.

For **Team A**:

The 25th percentile (Q1) is **15.0**.
The 50th percentile (Median/Q2) is **22.0**.
The 75th percentile (Q3) is **22.0**.

The fact that Q2 and Q3 are identical for Team A suggests that 50% of their scores fall between 15 and 22, but scores greater than 22 are clustered closely, or there is heavy data repetition around the 22-point mark. The Interquartile Range (IQR = Q3 - Q1) is 7.0, indicating a moderate spread in the central 50% of the data.

For **Team B**:

The 25th percentile (Q1) is **15.0**.
The 50th percentile (Median/Q2) is **28.0**.
The 75th percentile (Q3) is **31.0**.

Team B shows a significantly higher median (28.0 vs 22.0), indicating better overall scoring performance. The IQR for Team B is 16.0, suggesting a much wider spread in the central 50% of their scores compared to Team A. This points to greater variability in performance among Team

B's players. Team C's results (Q1=6.5, Q2=7.5, Q3=8.0) indicate a very low scoring team with minimal variance.

Advanced Considerations: Accuracy vs. Performance

When dealing with truly massive datasets, the computational cost of calculating exact percentiles can become prohibitive due to the need for global sorting. Spark offers mechanisms to address this tradeoff between accuracy and execution speed.

While the standard `percentile` function aims for exactness, the function `percentile_approx` is available in PySpark for scenarios where exact results are not mandatory but speed is critical. `percentile_approx` uses T-Digest or other sampling methods to estimate the percentile within a specified relative error range (e.g., `percentile_approx(col, array(0.50), relativeError)`).

Choosing the appropriate function is a strategic decision in big data analysis. For financial data or regulatory reporting, exactness is often paramount, justifying the use of the standard `percentile` function. For general exploratory analysis or visualization purposes, the speed benefits gained by using the approximate function often outweigh the minor statistical error introduced. Furthermore, optimization of the underlying Spark configuration, particularly concerning memory and shuffle operations during the `groupby` and aggregation steps, remains critical for maintaining performance.