Learn How to Calculate Percentiles in PySpark with Examples

Authored by

Mohammed looti

November 11, 2025

The Importance of Percentiles in Big Data Analysis

Calculating percentiles represents a foundational statistical requirement in contemporary data analysis workflows. These metrics are crucial for gaining a deep understanding of the underlying data distribution, identifying potential statistical outliers that deviate significantly from the norm, and facilitating comprehensive quantile analysis, such as determining quartiles or deciles. Fundamentally, a percentile defines the value below which a specified percentage of observations in a given dataset falls. For example, understanding that the 25th percentile (the first quartile) is a particular value tells us precisely that 25% of all observed data points lie at or below that value. This capability moves beyond simple averages, providing a robust view of data spread and asymmetry.

When organizations transition from small, local datasets to massive, large-scale data volumes--a common reality in modern analytics--relying on traditional computing methods becomes computationally prohibitive. This is where the powerful, distributed computing framework offered by **Apache Spark** becomes indispensable. Utilizing the Python API for Spark, universally known as PySpark, data scientists and engineers can execute highly complex statistical operations with remarkable efficiency, regardless of the scale of the input data. Specifically, PySpark allows for the efficient and accurate calculation of percentiles across petabytes of data distributed across a cluster, a task impossible for a single machine.

PySpark is equipped with robust functionality housed within the `pyspark.sql.functions` module, which provides the necessary tools for complex statistical aggregation. This module allows analysts to precisely specify the required quantile levels, offering versatility in application. Calculations can be performed either globally across an entire numerical column or conditionally, grouped by specific categorical variables. This guide will meticulously detail the two primary, most effective methodologies for calculating percentiles within a PySpark DataFrame, techniques essential for anyone needing to summarize numerical distributions quickly and accurately within a distributed environment.

Initializing the PySpark Environment and Preparing Sample Data

The prerequisite for any successful PySpark operation, including the calculation of percentiles, is the initialization of a SparkSession. This session acts as the definitive entry point for programmers interacting with Spark using the Dataset and DataFrame APIs. The SparkSession manages the connection to the cluster resources and provides the necessary context for all subsequent distributed operations. Only once this session is successfully established and active can we proceed to define and load the target data that will undergo statistical analysis.

For demonstrative clarity and to provide a practical context, we will utilize a concise, illustrative dataset. This dataset contains structured information regarding basketball players, tracking key

attributes such as their assigned **team**, their specific playing **position**, and the number of **points** they scored in a theoretical game. This structure is deliberately chosen as it allows us to showcase two distinct but equally important analytical tasks: calculating percentiles across all players globally (ungrouped) and calculating them conditionally based on their team affiliation (grouped).

The initial coding step involves defining the raw data structure, typically represented as a Python list of lists, and simultaneously defining the corresponding column schema (names). Supplying both the raw data and the schema allows PySpark to accurately infer and interpret the data types and names of the fields upon the creation of the distributed DataFrame. The resulting structure, conventionally named `df`, becomes the foundational element upon which all subsequent percentile calculation operations will be performed, ensuring the data is correctly partitioned and ready for distributed processing.

The following comprehensive code block details the complete setup procedure: defining the initial data, creating the necessary PySpark DataFrame, and finally, displaying the resulting structure for verification. It is important to note that the sample data employed here is structured to ensure realistic and easily interpretable output for demonstrating both global and conditional percentile metrics.

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|
| A| Guard| 32|
| B| Guard| 14|
| B| Guard| 14|
| B| Forward| 13|
| B| Guard| 7|
| B| Forward| 15|
+----+--------+------+
```

Method 1: Calculating Global Percentiles Using Aggregation (Single Column)

The most direct and frequently employed methodology involves computing a specific percentile across all values contained within a single numerical column of the PySpark DataFrame. This calculation disregards any potential grouping variables and treats the entire column as one continuous statistical sample. This process is effectively achieved by utilizing the powerful aggregation function, `df.agg()`, which is designed to compute summary statistics over the entire DataFrame or a specific set of columns. To access the advanced percentile computation logic, we pair `df.agg()` with the highly flexible built-in SQL function `percentile`, which is accessed via `F.expr()`. Using `F.expr()` is a vital technique, as it allows developers to execute complex Spark SQL functions directly within the Python DataFrame API, leveraging highly optimized code that may not have a simple, direct wrapper in the standard PySpark functions module.

To successfully calculate, for instance, the 25th percentile, we must pass a precisely structured expression string to `F.expr()`. This SQL expression requires two mandatory arguments: first, the target column name (e.g., `'points'`) whose distribution we are analyzing; and second, an array specifying the desired quantile(s). These quantiles must be represented as floating-point numbers ranging between 0.0 and 1.0 (e.g., `array(0.25)`). Because the underlying `percentile` function is inherently designed to return an array of results (even if the user only requests a single percentile), a crucial post-processing step is necessary: we must index the result using immediately following the `F.expr()` call to extract the single calculated scalar value from the returned array.

This technique is fundamentally important for generating initial, high-level summary statistics of the

entire dataset's distribution. The outcome is a single, definitive value that accurately describes the cutoff point at the specified percentage rank. Establishing this overall distribution metric is often the necessary initial phase in any comprehensive exploratory data analysis (EDA), helping analysts quickly gauge central tendency and spread before diving into sub-group comparisons.

Deconstructing the `F.expr` and `percentile` Function Syntax

A thorough understanding of the specific syntax--`F.expr('percentile(column, array(quantiles))')`--is paramount for effective percentile calculation in PySpark. The complexity arises because Spark's `percentile` function is not a simple scalar function but rather an advanced aggregate function engineered to handle concurrent computation of multiple quantiles with optimal efficiency. The requirement that quantiles must be passed as an array (even if it contains only one value, such as `0.25`) ensures that the function maintains its inherent flexibility and scalability. For instance, if an analysis required the 25th, 50th (median), and 75th percentiles simultaneously, the array would simply be expanded to `array(0.25, 0.50, 0.75)`, executing all calculations in a single, efficient pass.

The strategic use of `F.expr()` is non-negotiable in this context because it provides the mechanism to execute raw, optimized Spark SQL expressions directly within the Python DataFrame API workflow. This capability effectively bridges the gap between the native Python environment and the vast library of powerful, highly optimized SQL functions available within the Spark ecosystem. This integration ensures that even when dealing with extremely large datasets, the calculation maintains high performance and utilizes Spark's internal optimizations effectively. The expression string itself must be meticulously formatted, ensuring precise referencing of the target column and the required array of quantiles using correct SQL syntax conventions.

Finally, the mandatory requirement of appending after the `F.expr(...)` segment addresses the output format constraint. As previously noted, the `percentile` aggregate function returns its results encapsulated within an array. Therefore, if the analyst requested only one specific percentile (e.g., 0.25), the resulting output array will contain exactly one element. The application of serves the essential purpose of unpacking this single-element array, extracting the scalar percentile value, which is then ready to be assigned a clean, descriptive alias (e.g., `'%25'`) using the `.alias()` method, significantly improving the readability and clarity of the final output DataFrame.

Implementation Example: Single Column Percentile Calculation

In this first practical example, we will apply Method 1 to calculate the 25th percentile for the **points** column using our established basketball player DataFrame. Executing this calculation globally will yield the score value below which 25% of all players in the dataset fall, irrespective of their

assigned team or playing position. This statistic provides a key benchmark for the lower performance quartile across the entire sample population.

We utilize the required `F.expr` syntax combined with the `.agg()` method, ensuring that the resulting column is clearly aliased as `%25` to label the output statistic unambiguously. The final `.show()` action is what triggers the distributed computation across the Spark cluster and displays the resulting single-row DataFrame, which contains the calculated global percentile value.

The following precise syntax is used to compute the 25th percentile for the values contained within the **points** column:

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|
+----+
|12.5|
+----+

Based on the resulting output, we can definitively state that the 25th percentile of scores in the **points** column is **12.5**. This result carries significant meaning: it quantifies that exactly 25% of all players included in our sample dataset scored 12.5 points or fewer. This figure serves as a rapid, quantitative measure of the performance level of the lower quartile across the entire aggregated dataset, establishing a baseline for subsequent comparative analysis.

Method 2: Grouped Percentile Calculation for Comparative Analysis

While calculating percentiles across an entire dataset provides a necessary global benchmark, this approach is often insufficient for complex analytical needs. In real-world data science, it is far more common to require a comparison of the metric distribution (such as 'points' scored) across distinct, meaningful categories (such as 'team' or 'position'). This comparative requirement mandates the use of the `.groupby()` method immediately before the aggregation step. This powerful technique ensures that the percentile calculation is executed entirely independently and separately for every unique group defined by the specified grouping column(s).

When the `.groupby()` method is invoked, the subsequent `.agg()` function applies the specified aggregated expression--in this case, the sophisticated `percentile` function--separately to each logical partition created by the grouping key. This capability enables highly powerful and granular

comparative statistics. For instance, analysts can easily determine the median score (the 50th percentile) achieved by Team A and contrast it directly with the median score achieved by Team B, revealing internal performance disparities that would be masked by a global calculation.

Furthermore, when performing grouped aggregation, it is best practice and extremely common to calculate multiple quantiles simultaneously. This typically involves computing the 25th, 50th (median), and 75th percentiles. Collecting these three statistics provides the analyst with the core components of the five-number summary (excluding the minimum and maximum values), offering a comprehensive view of the distribution's central tendency, spread, and potential skewness for each distinct group. This requires chaining multiple `F.expr` statements within the single `.agg()` call, with each resulting calculation being assigned a distinct and meaningful alias for clarity in the final output.

Implementation Example: Grouped Percentiles and Interpretation

For our second and more advanced example, the objective is to calculate three critical percentile values--the 25th, 50th, and 75th--for the **points** column, while explicitly grouping the results by the categorical values present in the **team** column. This segregated analysis allows us to precisely assess the scoring distribution statistics independently for the players belonging to Team A and the players belonging to Team B, facilitating a direct, comparative performance evaluation.

We initiate the process by utilizing the `df.groupby('team')` method, immediately followed by the `.agg()` function. Within the aggregation block, we provide three separate `F.expr` calculations. Each calculation targets a distinct quantile (0.25, 0.50, and 0.75) and is appropriately aliased as %25, %50, and %75, respectively. The resultant DataFrame, named `df_new`, will consequently contain one dedicated row for each unique team, showcasing the three calculated percentiles specifically for the 'points' scored by players affiliated with that team.

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| 9.5|22.0|27.0|
| B|13.0|14.0|14.5|
+----+----+----+----+

A careful analysis of this segregated output provides a clear and statistically sound comparison of the scoring distributions between the two basketball teams, yielding actionable insights:

The 25th percentile (Lower Quartile) of points for **Team A is 9.5**, whereas for **Team B it is 13.0**. This distinct difference suggests that the lowest-scoring quartile of players within Team A generally exhibits a lower performance baseline compared to the lowest-scoring quartile within Team B.
The 50th percentile (Median) of points for **Team A is 22.0**, which is substantially higher than the median for **Team B, which is 14.0**. This pronounced difference strongly indicates that Team A demonstrates a significantly higher overall central scoring tendency and performance level than Team B.
The 75th percentile (Upper Quartile) for **Team A is 27.0**, compared to only **14.5** for Team B. This metric confirms that Team A possesses stronger top-tier individual performers who pull the upper distribution significantly higher than those on Team B.

By successfully implementing the grouped percentile calculation methodology in PySpark, data analysts are empowered to rapidly identify critical performance differences, accurately assess internal spread and variability within sub-groups, and inform strategic decisions based on highly reliable, segregated statistical summaries, even when processing massive distributed PySpark DataFrames. This technique is indispensable for robust comparative analytics in a big data environment.