How to Calculate the Minimum Value by Group in PySpark

Authored by

stats writer

February 9, 2026

The process of calculating the minimum value by group in PySpark is a fundamental operation in large-scale data processing. It requires segmenting a given dataset, specifically a DataFrame, based on one or more grouping columns, and subsequently applying an aggregation function, such as minimum calculation, to the resulting groups. This is efficiently achieved using the groupBy() transformation followed by the agg() function. The result is a concise output dataset that summarizes the lowest observed value for the specified numeric column within each category defined by the grouping key. This technique is indispensable for effective data analysis, enabling users to rapidly identify outliers, base thresholds, or simply determine the floor value across disparate segments of vast datasets in a high-performance, distributed computing environment.

Understanding the interplay between grouping and aggregation is crucial for mastering data manipulation in Spark. Unlike traditional single-machine processing tools, PySpark leverages parallelism; when the groupBy() operation is executed, Spark triggers a shuffle operation across the cluster. This shuffle ensures that all rows belonging to the same group key are physically moved to the same worker node partition. Once co-located, the aggregation step--the minimum calculation in this case--can be computed locally on that partition, leading to optimized performance and resource utilization. This two-phase process (shuffle and aggregate) is the backbone of calculating statistics like minimums, sums, or counts efficiently across massive scale data.

The efficiency of this approach is particularly evident when dealing with petabytes of data where calculating statistics sequentially would be infeasible. By distributing the workload, Spark ensures that the computation for finding the minimum value for 'Team A' happens simultaneously with the computation for 'Team B' and so forth. The final stage involves collecting these individual minimum results back into a single, cohesive result DataFrame, ready for subsequent processing or visualization. Mastery of this pattern is a prerequisite for advanced ETL (Extract, Transform, Load) tasks and sophisticated analytical modeling within the Spark ecosystem.

Calculate the Minimum by Group in PySpark

To accurately determine the minimum value within defined subgroups of your DataFrame, PySpark offers highly flexible methods utilizing the chaining of functional transformations. These methods are designed for scalability and readability, making complex data segregation tasks straightforward. We will explore two primary scenarios: grouping based on a single categorical column, and grouping based on a combination of multiple categorical columns, which allows for finer granularity in the resulting aggregation.

The core syntax relies heavily on importing the necessary functions from `pyspark.sql.functions`, typically aliased as `F` for brevity and standard practice. This module contains essential aggregation functions such as `min()`, `max()`, `avg()`, and `count()`. Using the

aliased namespace ensures clean code and avoids naming conflicts, which is especially important in large-scale scripting environments where numerous libraries might be imported concurrently. Furthermore, the selection of the correct aggregation function, `F.min()` in this context, directly dictates the summary statistic returned for each partition following the grouping operation.

Method 1: Calculating Minimum Grouped by One Column

When the requirement is to find the minimum value of a metric based solely on a single attribute--for example, calculating the lowest salary achieved within each department--this method is employed. The syntax is concise and follows the standard Spark pattern: defining the grouping keys first, and then specifying the aggregation to be performed on the target column. This approach is highly efficient because it minimizes the complexity of the shuffle phase, as data only needs to be grouped according to a single criterion.

The following structure illustrates how to perform this operation, targeting the 'points' column and grouping based on the 'team' column. Note the use of the groupBy() method which is a transformation, creating a `GroupedData` object, followed by the agg() function which is an action, triggering the computation.

import pyspark.sql.functions as F

#calculate minimum of 'points' grouped by 'team'
df.groupBy('team').agg(F.min('points')).show()

The resulting DataFrame will contain two columns: the grouping column ('team') and the resulting aggregated minimum value ('min(points)'). This output structure is crucial for readability and subsequent joining operations, should you need to merge these summary statistics back into the original dataset or other tables for enrichment.

Method 2: Calculating Minimum Grouped by Multiple Columns

Often, a broader analytical context requires grouping data based on the intersection of multiple categorical variables. For instance, determining the minimum sales achieved by a specific salesperson within a specific region requires grouping by both 'salesperson' and 'region'. This provides a much more granular view of the data, allowing analysts to pinpoint minimums under highly specific conditions. While this increases the complexity of the grouping key (a composite key), Spark handles this transparently during the shuffle phase by creating unique combinations of the specified columns.

This technique is essential for multilayered data analysis where simple, single-dimensional grouping might mask important patterns or anomalies. By utilizing two or more columns in the

groupBy() method, we instruct Spark to treat the unique pairing of those column values as the key for aggregation. For example, ('A', 'Guard') is treated as a distinct group separate from ('A', 'Forward').

import pyspark.sql.functions as F

#calculate minimum of 'points' grouped by 'team' and 'position'
df.groupBy('team', 'position').agg(F.min('points')).show()

The output structure in this case expands to include all columns used in the grouping definition ('team', 'position') alongside the calculated minimum ('min(points)'). This structure facilitates immediate interpretation of the results, clearly associating the calculated minimum with the exact combination of attributes that produced it. Proper handling of composite keys is a hallmark of robust distributed data processing, ensuring accuracy even when the number of unique combinations becomes exceptionally large.

Defining the Sample PySpark DataFrame

To demonstrate these methodologies effectively, we must first establish a sample dataset. This DataFrame simulates basketball player statistics, containing categorical data (team, position) and numerical data (points). The creation process begins by initializing a `SparkSession`, which is the entry point for all Spark functionality, and then defining the raw data structure and schema (column names).

This example highlights the typical workflow of data creation in PySpark: defining the nested list structure for the rows, defining a corresponding list of column headers, and finally using `spark.createDataFrame()` to instantiate the distributed data structure. This structured setup ensures that subsequent transformations and aggregations, such as the minimum calculation, operate on a well-defined schema, optimizing Spark's Catalyst optimizer performance.

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

The resulting DataFrame, displayed above, serves as our baseline. It clearly shows the distribution of players across three teams (A, B, C) and two positions (Guard, Forward), along with their respective points. This data structure facilitates precise testing of the minimum aggregation logic, allowing us to manually verify that the calculated minimums for each group are correct.

Example 1: Calculating Minimum Grouped by a Single Column

We now execute the first method, focusing on calculating the minimum score achieved solely on a team-by-team basis. This requires grouping the entire dataset by the 'team' column. Spark will internally partition the data such that all records for Team A are together, all records for Team B are together, and so on. The `F.min('points')` operation is then applied to the 'points' column within each of these segregated groups.

This operation is fundamental to comparative data analysis, providing a quick statistical summary of performance boundaries across the primary grouping variable. If we were concerned with identifying the lowest performer threshold for each team organizationally, this approach provides the necessary metric directly and efficiently, demonstrating the power of the groupBy() and agg() function chain.

import pyspark.sql.functions as F

#calculate minimum of 'points' grouped by 'team'
df.groupBy('team').agg(F.min('points')).show()

```
+----+-----------+
|team|min(points)|
+----+-----------+
| A| 8|
| B| 7|
| C| 5|
+----+-----------+
```

The resulting output is a concise summary table. It represents the lowest score observed among all players belonging to a specific team. This result is achieved through the distributed shuffle and subsequent localized aggregation process inherent to the PySpark architecture, ensuring accuracy regardless of the input data size.

Analyzing the Output of Single-Column Grouping

A detailed examination of the output confirms the effectiveness of the single-column grouping strategy. By manually inspecting the original dataset, we can verify that the calculated minimums are accurate representations of the lowest 'points' value associated with each unique 'team' key. This verification step is vital in any data processing pipeline to ensure transformation correctness.

The interpretation of these results provides immediate, actionable insights into team performance floors. Specifically, the minimum value serves as a baseline for the scoring range within that group. If the data represented quality metrics, the minimum value would denote the worst observed quality standard for that category. Understanding the magnitude of this minimum value relative to the overall dataset minimum can inform decisions regarding performance improvement or resource allocation.

The minimum points value for players on **team A** is **8**. (Original scores for A: 11, 8, 22, 22).
The minimum points value for players on **team B** is **7**. (Original scores for B: 14, 14, 13, 7).
The minimum points value for players on **team C** is **5**. (Original scores for C: 8, 5).

This simple aggregation provides a powerful comparative tool, instantly revealing Team C as having the lowest individual scoring floor among the three teams in this dataset. This level of statistical insight, derived from minimal code, demonstrates why grouped aggregation is a cornerstone of modern data analysis using Spark.

Example 2: Calculating Minimum Grouped by Multiple Columns

In scenarios requiring higher analytical resolution, grouping by multiple columns--'team' and 'position'--allows us to identify the minimum score for a specific position within a specific team. This creates six distinct groups in our dataset (e.g., Team A Guards, Team A Forwards, Team B Guards, etc.). This compound grouping dramatically refines the statistical output, offering insight into departmental or role-specific performance minima.

This method is executed by passing a list or sequence of column names to the groupBy() method. The internal Spark optimization ensures that the necessary data shuffling respects both keys simultaneously, creating granular partitions that accurately reflect the unique combinations. The efficiency of the agg() function remains consistent, applying the minimum calculation to each of these refined subgroups.

import pyspark.sql.functions as F

#calculate minimum of 'points' grouped by 'team' and 'position'
df.groupBy('team', 'position').agg(F.min('points')).show()

```
+----+--------+-----------+
|team|position|min(points)|
+----+--------+-----------+
| A| Guard| 8|
| A| Forward| 22|
| B| Guard| 13|
| B| Forward| 7|
| C| Forward| 5|
| C| Guard| 8|
+----+--------+-----------+
```

The resulting DataFrame now provides a minimum score for every unique combination of team and position found in the original data. This level of segmentation is often necessary for statistical modeling or generating reports that must reflect specialized performance metrics across different functional units.

Interpreting Multi-Column Aggregation Results

Interpreting the multi-column aggregation results reveals insights that were previously hidden in the single-column grouping. For instance, while we knew Team A's overall minimum was 8, the multi-column grouping clarifies that this minimum belongs specifically to the Guards of Team A, while the Forwards of Team A have a much higher performance floor (22). This specificity is invaluable for targeted intervention or performance reviews.

The clarity provided by this granular grouping mechanism demonstrates the robust capabilities of PySpark in handling complex relational analysis within a distributed computing framework. Each row in the output represents a summary metric derived from potentially hundreds or thousands of input records, all belonging to that specific composite key.

The minimum points value for **Guards** on **team A** is **8**. (Original Guard A scores: 11, 8).
The minimum points value for **Forwards** on **team A** is **22**. (Original Forward A scores: 22, 22).
The minimum points value for **Guards** on **team B** is **13**. (Original Guard B scores: 14, 14, 13).
The minimum points value for **Forwards** on **team B** is **7**. (Original Forward B scores: 7).
The minimum points value for **Forwards** on **team C** is **5**. (Original Forward C scores: 5).
The minimum points value for **Guards** on **team C** is **8**. (Original Guard C scores: 8).

This granular data confirms, for example, that Team B's lowest score (7) belongs exclusively to one of its Forwards, rather than being an issue spanning both positions. Such detailed statistical breakdowns are fundamental to high-stakes decision-making processes based on empirical data.

Conclusion and Further Explorations in PySpark Aggregation

Calculating the minimum value by group in PySpark is a powerful yet straightforward task, executed primarily through the coupling of the groupBy() transformation and the `F.min()` aggregation within the agg() function. Whether the analysis requires broad categorization via a single column or precise segmentation using multiple composite keys, PySpark provides the scalable tools necessary for efficient distributed computation.

These methods are highly versatile and can be easily adapted to calculate other key metrics by simply replacing `F.min()` with functions like `F.max()`, `F.avg()`, or `F.sum()`. Furthermore, the `agg()` function permits simultaneous calculation of multiple statistics within a single grouping operation, such as finding the minimum and maximum points per team concurrently, dramatically reducing computational overhead and speeding up the data analysis cycle.

Mastering grouped aggregation is a cornerstone of effective big data processing using Spark. The seamless integration of Python syntax with Spark's optimized execution engine makes these complex, distributed tasks accessible and performant, paving the way for advanced analytical

workflows, machine learning feature engineering, and high-volume report generation across enterprise datasets.

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