Calculate the Minimum by Group in PySpark

Authored by

stats writer

November 16, 2025

Introduction to Minimum Aggregation in PySpark

The ability to calculate descriptive statistics across subsets of data is fundamental in modern data processing. When working with large-scale datasets, tools like PySpark are essential for efficient parallel computation. One common requirement is determining the minimum value of a specific column, partitioned by one or more grouping keys. This process, known as minimum aggregation, allows data analysts to quickly identify low outliers or baseline values within distinct categories.

In PySpark, this operation is performed using a combination of the `groupBy()` transformation and the `agg()` action, utilizing built-in functions from `pyspark.sql.functions`. The efficiency of Spark ensures that even highly complex datasets can be processed rapidly. We will explore two primary methods for calculating the minimum value by group: grouping by a single column and grouping by multiple columns.

Understanding how to correctly apply these methods is crucial for mastering data manipulation in a distributed environment. Whether you are seeking the lowest sales record per region or the minimum score per team, the principles remain the same. The examples provided below utilize a sample DataFrame containing athlete statistics to illustrate these powerful aggregation techniques.

Method 1: Calculating Minimum Grouped by a Single Column

To calculate the minimum value based on a single categorical criterion, we use the `groupBy()` function, specifying the grouping column, followed by the `agg()` function, which applies the minimum function, `F.min()`, to the target numerical column. This approach is straightforward and highly effective for generating simple summary statistics.

The syntax requires importing the necessary functions module, typically aliased as `F`. The result is a new DataFrame where each row represents a unique value from the grouping column, along with the corresponding minimum value from the aggregated column.

import pyspark.sql.functions as F

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

This streamlined code ensures that the distributed computation engine efficiently processes the data partitions, identifying the smallest value within the specified column for every unique key in the grouping column. This is the standard and most efficient way to perform single-column minimum aggregation within PySpark.

Method 2: Calculating Minimum Grouped by Multiple Columns

When greater granularity is required--for instance, finding the minimum value based on combinations of categories--we extend the `groupBy()` function to include multiple column names. This creates composite keys for the aggregation. Every unique combination of values across the specified columns will form a distinct group, and the minimum calculation will be applied independently to each group.

This method is vital for segmenting data across several dimensions simultaneously, allowing for highly specific analytical insights. For example, if we are analyzing player performance, grouping by both `team` and `position` gives us the minimum score achieved by players in a specific role within a specific team.

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 power of this technique lies in its ability to handle complex groupings efficiently. By defining multiple grouping keys, the resulting DataFrame provides a detailed breakdown of the minimum values, ensuring that the statistics are relevant to the fine-grained data partitions defined by the user.

Prerequisites: Setting up the PySpark Environment and DataFrame

Before executing the aggregation methods, it is necessary to initialize a `SparkSession` and create the source DataFrame. Our sample dataset includes information about basketball players, specifically tracking their `team`, `position`, and `points` scored. This setup block demonstrates the preparation required to load data into the Spark context.

We use a list of lists (`data`) to hold the records and define the column schema explicitly (`columns`). The `spark.createDataFrame()` method converts this local collection into a distributed Spark DataFrame, ready for processing. This step is foundational for any subsequent PySpark operation.

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


As observed in the output, the resulting DataFrame `df` is correctly structured with ten records, ready for the minimum value calculations across the different teams and positions.

Example 1: Calculating Minimum Grouped by Team

Our first example applies Method 1 to find the minimum points scored by any player within each unique team (A, B, and C). This involves using `df.groupBy('team')` to partition the data, followed by `agg(F.min('points'))` to compute the desired statistic.

This calculation aggregates all player records belonging to Team A together, all records for Team B, and so forth. The result clearly demonstrates the lowest scoring performance recorded for each team in the dataset. Note that `F.min()` automatically names the aggregated column `min(points)`, although it is best practice in production environments to alias this column using `.alias('min_points')` for clarity.

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 table simplifies the original data into a concise summary of the minimum performance metric per group. This approach is highly useful for initial exploratory data analysis.

Detailed Analysis of Single Column Aggregation Results

Analyzing the output from the single column aggregation reveals specific minimum performance benchmarks for each team. These results provide immediate insights into the lowest recorded performance within the grouping category.

For **Team A**, the minimum recorded score across all players and positions is **8** points. Reviewing the source data confirms that one of the Guards on Team A scored 8 points, which is the lowest value for that team (lower than 11 and 22).

For **Team B**, the minimum recorded score is **7** points. This value belongs to a Forward on Team B, which is lower than the other scores recorded for that team (14 and 13).

For **Team C**, the minimum recorded score is **5** points. This represents the absolute minimum score in the entire dataset and belongs to a Forward on Team C.

This detailed breakdown demonstrates how the `groupBy()` operation successfully condensed the ten original records into three summary statistics, each representing the lowest boundary for its respective team.

Example 2: Calculating Minimum Grouped by Team and Position

In this advanced example, we apply Method 2, partitioning the data based on two dimensions: `team` and `position`. This allows us to calculate the minimum score achieved by Guards on Team A separately from Forwards on Team A, and so on for all unique combinations.

By passing both column names to `groupBy('team', 'position')`, PySpark creates composite keys such as ('A', 'Guard'), ('A', 'Forward'), ('B', 'Guard'), etc. The `F.min()` function then computes the minimum points within each of these six distinct groups, providing a much finer level of statistical detail.

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

This resulting table offers significantly richer context compared to the single-column aggregation, allowing analysts to compare the minimum performance across specific roles within the organization of each team. This approach is highly scalable and forms the backbone of complex data warehousing tasks involving dimensional analysis.

Interpreting Multi-Column Aggregation Results

The multi-column output provides granular statistics essential for performance benchmarking. We can now precisely define the minimum metric for a specific role and team combination.

For Guards on **Team A**, the minimum points value is **8**. This contrasts with the minimum for Forwards on Team A, which is **22**. This disparity shows that while the team minimum is 8, the minimum performance level for Forwards is much higher.

For Guards on **Team B**, the minimum points value is **13**. Comparing this to the minimum for Forwards on Team B (**7**), we see that the Guards generally have a higher floor score than the Forwards, indicating different performance distributions within the team structure.

For Guards on **Team C**, the minimum points value is **8**, while the minimum for Forwards is **5**. The calculation correctly identifies the minimum score of 5 points as belonging specifically to the ('C', 'Forward') group.

The successful application of the multi-column `groupBy()` operation showcases the precision and analytical depth achievable when leveraging PySpark's distributed aggregation capabilities.

Advanced Considerations for Grouped Minimum Calculation

While the basic application of `F.min()` is sufficient for most tasks, data practitioners should be aware of several advanced considerations. Firstly, handling null values is critical. By default, Spark ignores `null` values during aggregation. If a group consists entirely of `null` values, the result for that group's minimum will also be `null`.

Secondly, performance can be optimized by minimizing shuffles. The `groupBy()` operation is a wide transformation, meaning it requires data shuffling across the cluster nodes. For very large DataFrames, analysts may consider using window functions if they need the minimum value to be calculated relative to the group but retained alongside the original record, rather than collapsing the data into an aggregated summary. However, for generating pure summary statistics, the `groupBy().agg()` pattern remains the standard.

Finally, when defining the aggregation, users should always consider renaming the output column using `.alias('custom_min_name')` to prevent the default `min(column_name)` output, which can be verbose or confusing when multiple aggregations are applied simultaneously. This practice enhances the readability and usability of the resulting summary DataFrame.

Conclusion

Calculating the minimum value by group is a core data analysis task efficiently handled by PySpark. Whether grouping by a single column (e.g., `team`) or complex composite keys (e.g., `team` and `position`), the combination of `groupBy()` and `agg(F.min())` provides a powerful, scalable solution for deriving granular summary statistics from massive datasets.

By following these established methods, data professionals can ensure accurate and highly performant determination of minimum boundaries within their segmented data. Mastery of these aggregation techniques is foundational for advanced data manipulation and reporting within the Apache Spark ecosystem.