Calculate the Max by Group in PySpark

Authored by

stats writer

November 16, 2025

Introduction to Grouped Aggregation in PySpark

Analyzing large datasets often requires calculating summary statistics based on specific categories or groups. When working with distributed processing frameworks, such as PySpark, efficiently performing these grouped calculations is paramount for data science and engineering workflows. This comprehensive guide details the essential methods used to determine the maximum value (**max**) within defined groups across a PySpark DataFrame. Mastering the groupBy() function combined with aggregation operators is fundamental for extracting meaningful insights from massive, distributed data structures.

The process of computing a maximum value per group involves partitioning the DataFrame based on one or more categorical columns, applying the aggregation function (maximum) to a target numerical column within each partition, and finally combining these results into a new, summarized DataFrame. This operation is highly optimized within the Apache Spark engine, leveraging its distributed nature to handle scales of data impractical for traditional, single-node databases. We will explore two primary scenarios: grouping by a single key and grouping by compound keys, demonstrating how to achieve precise data roll-ups using native PySpark methods.

The methodologies presented here rely heavily on the powerful utilities provided by the pyspark.sql.functions module, specifically the `max()` function. Understanding how to import and alias these functions--conventionally using `F`--streamlines the code and makes the intent clear. By utilizing `df.groupBy().agg(F.max(...))`, we execute the aggregation efficiently, ensuring high performance even when dealing with petabytes of information across clusters managed by PySpark.

Prerequisites: Setting Up the PySpark DataFrame

Before diving into the aggregation methods, it is essential to establish a representative dataset. For demonstration purposes, we will create a sample DataFrame containing basketball player statistics, including their team affiliation, position, and accrued points. This DataFrame serves as the foundation upon which we will apply the maximum grouped calculation techniques. This setup process involves initializing a PySpark `SparkSession`, defining the raw data, specifying column schemas, and finally constructing the distributed data structure.

The sample data includes three columns: `team` (categorical identifier), `position` (another categorical descriptor), and `points` (the numerical column we intend to aggregate). The heterogeneity of the data across these teams and positions is crucial for demonstrating how the `groupBy` operation correctly isolates data subsets before calculating the maximum score achieved within each subset. This initialization step ensures that all subsequent code examples are immediately runnable and verifiable.

To ensure reproducibility and clarity, the following PySpark code block illustrates the exact steps required to instantiate the `df` DataFrame. This foundational step is mandatory for running the subsequent examples successfully within any distributed Spark environment. The resulting table displays the initial state of our data before any aggregations are performed.

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

# Define sample data for basketball players
data = ,
,
,
,
,
,
,
,
,
]

Define column names (schema)
columns =

# Create DataFrame using the defined data and schema
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure and contents
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|
+----+--------+------+
```

Core Techniques for Calculating Grouped Maximums

Calculating the maximum value per group fundamentally relies on the sequential application of two PySpark DataFrame API methods: `groupBy()` and `agg()`. The `groupBy()` function dictates how the dataset is logically partitioned, specifying the key columns that define the groups. The subsequent `agg()` function then applies one or more aggregate functions--in our case, `F.max()`--to the remaining columns within those defined groups, generating a single summary row for each unique group combination.

We focus on two principal patterns for achieving this maximum aggregation. The first, and simplest, involves grouping by a single categorical column, such as finding the maximum points scored strictly by **team**. The second, more granular approach, utilizes multiple columns for grouping, allowing us to find the maximum points scored not just by **team**, but broken down further by **position**. Both methods utilize the same fundamental syntax structure but differ critically in the number of arguments provided to the `groupBy()` method.

The general structure for both methods is highly efficient and scalable, leveraging Spark's distributed architecture. Note the essential import necessary for both techniques: `import pyspark.sql.functions as F`. This alias is standard practice in the PySpark ecosystem and grants immediate access to the necessary aggregation operator.

Method 1: Calculating Maximums Based on a Single Column Key

When the analytical goal is to summarize a metric across the broadest categorical level, grouping by a single column is the most direct and simplest approach. In our basketball dataset context, this translates to determining the highest points scored by any player within each specific **team**, irrespective of their position. This method simplifies the resultant output, providing a concise summary where each row corresponds to a unique value in the grouping column.

The implementation is straightforward: we first call `df.groupBy('team')` to define the grouping key. This action triggers the crucial step of data shuffling across the cluster nodes, ensuring all records belonging to the same team are processed together on the same executor. Following the grouping, we invoke `.agg()` and pass the `F.max('points')` function as an argument. The resultant output DataFrame will invariably contain two columns: the grouping key (`team`) and the calculated aggregate (`max(points)`).

This technique is widely used in exploratory data analysis (EDA) to quickly gauge the highest

performing entity based on a key metric. Below is the specific PySpark syntax required for this single-column grouped maximum calculation.

import pyspark.sql.functions as F

# Calculate max of 'points' grouped solely by 'team'
df.groupBy('team').agg(F.max('points')).show()

+----+-----------+
|team|max(points)|
+----+-----------+
A	22
B	14
C	8
+----+-----------+

Detailed Analysis of Single-Column Grouping Results

Upon reviewing the output generated by the single-column aggregation, we can observe clear, definitive statistics for each team represented in the dataset. This summarized view demonstrates the effectiveness of the `groupBy` operation in collapsing detailed records into crucial performance indicators. The column `max(points)` represents the highest score achieved by any single player belonging to that respective team partition.

Specifically, the analysis of the results reveals the maximum point totals:

The **max points** value for players on team A is **22**. This value is derived from examining all associated records for Team A and selecting the highest points score.
The **max points** value for players on team B is **14**. This value is extracted from the maximum points scored among all players affiliated with Team B.
The **max points** value for players on team C is **8**. This represents the peak performance recorded for Team C within the dataset.

This simple aggregation provides significant value by identifying the peak achievement across large sets of data, serving as a rapid metric for performance benchmarking across the defined organizational categories. It offers a macro-level view of performance potential within the dataset.

Method 2: Calculating Maximums Across Multiple Grouping Keys

Often, a more granular analysis is necessary, requiring the aggregation to be segmented by more than one dimensional attribute. To achieve this level of detail, we extend the groupBy() function to

accept multiple column names. For our example, we seek to find the maximum points scored not just by `team`, but specifically by the intersection of `team` and `position`. This action creates much finer, more precise partitions within the data before the maximum function is applied.

By specifying `df.groupBy('team', 'position')`, the Spark engine creates unique compound keys (e.g., 'A', 'Guard'; 'A', 'Forward'; 'B', 'Guard', etc.). The maximum calculation (`F.max('points')`) is then executed independently within each of these distinct, multi-key groups. This yields a richer, more descriptive summary, allowing analysts to compare performance across specific roles within organizations. This technique is indispensable for detailed segmentation analysis.

This multi-column grouping is crucial for analyses where the metric being evaluated is highly dependent on subcategories. For instance, determining whether Guards or Forwards contribute higher maximum scores requires this two-level grouping structure. The resultant DataFrame will contain all grouping keys plus the aggregated metric. The following code demonstrates the required syntax and the corresponding summarized output table.

import pyspark.sql.functions as F

# Calculate max of 'points' grouped by both 'team' and 'position'
df.groupBy('team', 'position').agg(F.max('points')).show()

+----+--------+-----------+
|team|position|max(points)|
+----+--------+-----------+
A	Guard	11
A	Forward	22
B	Guard	14
B	Forward	7
C	Forward	5
C	Guard	8
+----+--------+-----------+

The output table clearly breaks down the maximum points based on the combination of team and position. We can now precisely identify the highest performance for specific roles within specific teams, revealing nuances that the single-column aggregation obscured.

The **max points** value for Guards on team A is **11**.
The **max points** value for Forwards on team A is **22**.
The **max points** value for Guards on team B is **14**.

Advanced Context and Performance Considerations

While `groupBy().agg()` is the standard and most performant method for calculating maximums across groups that result in a reduced dataset, data professionals should be aware of context-specific alternatives, particularly when the requirement is to calculate the maximum value per group while retaining all original rows in the DataFrame. This requirement often necessitates the use of Window functions. Unlike standard aggregation, which collapses rows, Window functions allow for calculating the maximum value over a defined partition (group) and then joining that maximum value back to every single row within that partition, maintaining the original dataset size for subsequent analysis stages.

From a performance perspective in PySpark, aggregation operations like `groupBy().agg()` are highly optimized but necessitate a critical step known as data shuffling. Shuffling is the costly process where data records are physically redistributed across the cluster nodes based on the grouping keys (`team` or `team` and `position`). Minimizing the number of columns used for grouping helps reduce the computational overhead and the network traffic associated with shuffling, thereby significantly improving overall execution speed for large-scale data operations.

Furthermore, a key optimization technique involves executing multiple aggregations simultaneously within a single `.agg()` call (e.g., calculating max points, average points, and count in one operation). Spark's Catalyst Optimizer can process all requested functions during a single pass of the shuffled data, avoiding redundant shuffling steps that would occur if separate `groupBy` statements were executed for each aggregate metric. This methodology is strongly recommended for complex reporting requirements.