Learning PySpark: How to Calculate the Maximum Value by Group

Authored by

Mohammed looti

November 11, 2025

Mastering Grouped Aggregation in PySpark

Calculating the maximum value within various subgroups is a fundamental and often critical operation in modern Big Data analysis, especially when dealing with distributed datasets. This process, known as grouped aggregation, allows data scientists and engineers to summarize vast quantities of information by extracting key metrics relevant to specific categories. In the realm of Apache Spark, specifically using the Python API, PySpark DataFrames offer robust, distributed methods to perform this task with high efficiency and scalability. This comprehensive tutorial outlines the precise techniques required to determine the maximum value of a specific numerical column, grouped effectively by either a single categorical key or a combination of multiple columns, utilizing the powerful, built-in functions provided by the framework.

The ability to perform these high-speed distributed computations is what sets Apache Spark apart from traditional single-machine processing tools like Pandas. When working with terabytes or petabytes of data, relying on Spark's optimized engine ensures that the grouping and aggregation steps are parallelized across a cluster of machines. This parallel execution minimizes latency and enables rapid data exploration and reporting. Understanding the underlying mechanisms of how Spark handles grouping--namely, shuffling data across nodes--is key to writing truly efficient and scalable code.

To achieve max-by-group calculation in PySpark DataFrames, we primarily rely on a chain of two core methods, representing the standard pattern for summary statistics generation: the **`groupBy`** transformation, which defines the partitions for aggregation by specifying the categorical keys, and the **`agg`** action, which applies an aggregate function, such as `max()`, to the grouped data. Mastering how these fundamental functions interact is absolutely critical for efficient and effective distributed data processing and transformation within the Spark ecosystem. The proper selection of grouping keys directly influences the granularity and utility of the final summary output.

Method 1: Calculating Max Grouped by One Column

The first and most common scenario encountered in data analysis involves calculating the maximum value relative to a single categorical variable. This technique is typically used for generating simple, high-level summaries, such as finding the maximum sales per region or the highest score per team. This process efficiently reduces the size of the original PySpark DataFrame by collapsing all rows that share the same grouping key into a single summary row, which then contains the highest numeric value observed within that specific group.

Executing this single-key aggregation requires specifying only one column name within the `groupBy` method. PySpark internally initiates a shuffle operation across the cluster to bring all rows with identical key values (e.g., all rows belonging to 'Team A') onto the same physical worker node.

Once partitioned, the `agg()` function applies the `max()` operation locally on each partition, ensuring that the maximum value is correctly identified in a distributed manner.

import pyspark.sql.functions as F

# Calculate the maximum value of 'points' grouped exclusively by 'team'
df.groupBy('team').agg(F.max('points')).show()

This concise code block demonstrates the power of the PySpark API. The `groupBy` method effectively partitions the dataset based on the unique categorical values found in the `team` column. Subsequently, the `agg` function is invoked, applying `F.max()`--which is derived from the `pyspark.sql.functions` module--to the `points` column. This results in the generation of a new, summarized DataFrame containing only the grouping key (team) and the resultant maximum value for that specific category, providing an immediate snapshot of peak performance per team.

Method 2: Calculating Max Grouped by Multiple Columns

When granular analysis is required, data scientists often need to segment the data based on combinations of multiple features rather than just one. This approach allows for highly detailed aggregation, identifying the maximum value only when all specified grouping criteria are identical. For instance, determining the highest score achieved by a specific player position within a specific team requires grouping by both the `team` and the `position` columns simultaneously. This multi-key grouping strategy refines the level of detail significantly, moving from a broad team summary to a performance metric specific to a role within that team.

This technique is essential for sophisticated segmentation analysis where the context provided by multiple variables is necessary for accurate and meaningful summary statistics. By introducing multiple grouping columns, we force Spark to create separate buckets (partitions) for every unique intersection, ensuring that the maximum calculation is performed independently within those distinct intersections. This prevents the highest score in one position from skewing the perceived maximum of another position within the same team.

import pyspark.sql.functions as F

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

In this advanced scenario, the `groupBy` method accepts multiple column names (`team` and `position`). Apache Spark then calculates the maximum of the `points` column for every unique combination of team and position found in the input DataFrame. This necessitates a more complex

shuffle operation than single-column grouping, as Spark must ensure that all rows sharing the exact same combination of keys are processed together, leading to a much finer-grained result set that is crucial for segmentation analysis.

Setting Up the PySpark Environment and Sample Data

Before executing these distributed aggregation methods, a foundational step involves properly initializing a Spark session and constructing a representative sample PySpark DataFrame. The Spark session acts as the entry point to the entire underlying cluster functionality, managing resource allocation and task scheduling. The following setup code defines a simple, yet illustrative dataset containing basketball player information, including their team affiliation, specific position, and points scored during a hypothetical game.

This data preparation step is absolutely essential for achieving reproducible and verifiable results. We first define the raw data as a Python list of lists, where each inner list represents a record. Crucially, we then specify the column schema (`team`, `position`, `points`) as a separate list. This schema definition allows Spark to correctly interpret the data types, which is vital for numerical aggregations like `max()`. Finally, we use `spark.createDataFrame()` to transform this local Python data structure into a distributed, schema-aware PySpark object, which is then referenced as `df` in all subsequent code examples.

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

# Define raw data representing player performance metrics
data = ,
,
,
,
,
,
,
,
,
]

Define column names (schema)
columns =

# Create the distributed PySpark DataFrame
df = spark.createDataFrame(data, columns)

```
# Display the initial DataFrame structure and content
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 Maximum Points Grouped by Team

For the first practical demonstration, we focus exclusively on identifying the highest score achieved by any player within each unique team partition (A, B, and C). This exercise represents the most straightforward application of grouped aggregation, providing a high-level performance benchmark for comparison. This requires grouping the entire distributed dataset based solely on the **team** column and then applying the maximum aggregation function to the **points** column. The result is a concise summary of peak offensive performance across the defined teams in the dataset.

In execution, PySpark first executes the `groupBy('team')` operation, which triggers a shuffle to organize the data by team across the cluster nodes. Once the data is partitioned, the `agg(F.max('points'))` step calculates the largest numerical value in the `points` column within each partition. We utilize the `pyspark.sql.functions` module, aliased as `F`, to access the optimized `max()` function. The resulting DataFrame will contain two columns: the grouping key (`team`) and the calculated maximum value, automatically labeled by PySpark as `max(points)`.

```
import pyspark.sql.functions as F

# Execute single-key maximum aggregation
df.groupBy('team').agg(F.max('points')).show()

+----+-----------+
```

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

From the resulting output, we can draw the following conclusions regarding the maximum points scored across the grouped teams, providing immediate insights into the maximum performance recorded for each team:

The max points value recorded for players on **team A** is **22**. This value represents the highest score achieved by any player associated with team A, irrespective of their specific position on the court.
The max points value recorded for players on **team B** is **14**. This indicates that 14 was the peak performance score registered by any player on this team within the dataset, providing a clear benchmark for their individual scoring ceiling.
The max points value recorded for players on **team C** is **8**. This demonstrates the highest individual scoring performance recorded for players belonging to team C, highlighting their top offensive output.

Example 2: Calculating Maximum Points Grouped by Team and Position

The second example demonstrates how to achieve a significantly more granular breakdown by grouping the data based on two distinct categorical variables: **team** and **position**. This multi-key technique is invaluable for analysts seeking to dissect performance metrics within specific operational roles, offering fine-grained insights into which positions yield the highest scores within each team structure. We use the same aggregation logic, but specify both columns within the `groupBy` function, enabling precise contextual analysis.

By grouping on two keys, PySpark generates a maximum point total for every unique pair (e.g., Team A, Guard; Team A, Forward). This partitioning strategy ensures that the maximum calculated score is highly contextual, reflecting the best performance only within that highly specific subgroup defined by the intersection of the two grouping columns. This level of detail is indispensable for performance benchmarking and segmentation reporting across large-scale datasets managed by PySpark.

import pyspark.sql.functions as F

Execute multi-key maximum aggregation

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

Analyzing the output from the multi-column aggregation, we observe detailed performance metrics that were previously hidden in the broader team summary:

The maximum points value for **Guards on team A** is **11**. This score is significantly lower than the team's overall maximum (22), clearly indicating that a Forward player achieved the team's highest score.
The maximum points value for **Forwards on team A** is **22**. This confirms that the highest individual score for team A came specifically from a player operating in the Forward position.
The maximum points value for **Guards on team B** is **14**. This score matches the overall team maximum found in Example 1, demonstrating that a Guard held the top scoring slot for team B, differentiating its performance profile.

Advanced Considerations for PySpark Aggregation

While `max()` is a simple and fundamental aggregate function, the `pyspark.sql.functions` module supports a remarkably wide array of other crucial operations that can be deployed within the `agg()` method, including `min()`, `avg()`, `sum()`, and `count()`. Furthermore, PySpark allows users to apply multiple aggregations simultaneously within a single `agg()` call, calculating multiple summary statistics for each group in one pass.

A crucial best practice for production-ready code involves enhancing readability and usability by renaming the aggregated column output. By default, PySpark assigns non-descriptive names like `max(points)`. To remedy this and ensure descriptive column headers, the `alias()` function must be used immediately after the aggregation function call, which is often recommended for production code. This practice simplifies subsequent data manipulation steps, such as filtering or joining, which rely on clear column identifiers.

For example, if you wished to rename the output column in Example 1 for better clarity, the code would be modified as follows: `df.groupBy('team').agg(F.max('points').alias('Max_Team_Points')).show()`. This practice enhances readability and simplifies subsequent operations performed on the resulting DataFrame. The core principles of using **`groupBy`** followed by **`agg()`** remain the standard, most robust, and most efficient pattern for complex distributed data summarization in the PySpark DataFrame API.