How to Find the Maximum Value in Each Group with PySpark

Authored by

stats writer

February 8, 2026

Analyzing large-scale datasets often requires calculating aggregate statistics across defined subsets of data. In the context of PySpark, the efficient calculation of the maximum value within each group is a fundamental operation. This process leverages the powerful combination of the groupBy() function to partition the data based on categorical columns, followed by the agg() function, which applies the relevant aggregation--in this case, the standard `max()` function--to the target numerical column. Adopting this vectorized approach is critical for high performance in a distributed computing environment.

The core advantage of using PySpark's DataFrame API methods, such as `groupBy()` and `agg()`, lies in their optimization by the underlying Apache Spark engine. Spark handles the complex shuffle operations necessary for grouping data across different nodes efficiently. Furthermore, relying on highly optimized PySpark built-in functions (like `max()`, `sum()`, or `avg()`) is always recommended over writing custom User Defined Functions (UDFs) in Python, as UDFs introduce serialization overhead that significantly hampers performance in large-scale operations. Understanding these foundational principles is key to mastering data aggregation tasks.

Calculating the Maximum Value by Group in PySpark DataFrames

The following sections detail the primary methodologies for calculating maximum values within a PySpark DataFrame, structured by the complexity of the grouping key:

Method 1: Calculating Max Grouped by a Single Column

When analyzing data, the most straightforward aggregation scenario involves grouping records based on the values of a single categorical identifier. This method is exceptionally common for tasks such as finding the top sale amount per region or, as demonstrated here, the maximum score achieved by a sports team. This approach requires specifying only one column within the groupBy() function, simplifying the necessary shuffle operation across the cluster nodes.

Implementing this method utilizes the `pyspark.sql.functions` module to access the native `max` function. This ensures that the operation remains highly efficient, leveraging the vectorized nature of the Spark execution engine. The output is a new DataFrame where each row represents a unique value from the grouping column and the corresponding maximum value from the aggregated column.

import pyspark.sql.functions as F

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

The code snippet above imports the necessary functions from `pyspark.sql.functions`, aliased

as `F`. It then takes the DataFrame `df`, groups it by the `'team'` column, and applies the maximum function (`F.max`) to the `'points'` column within the resulting grouped data structure. The `show()` action triggers the computation and displays the resulting aggregated DataFrame, ensuring a high level of performance by delegating the computation to Spark's optimized core.

Method 2: Calculating Max Grouped by Multiple Columns

More granular analysis often requires partitioning the data based on combinations of attributes. If you need to find the maximum value corresponding to unique combinations of two or more columns--for example, the maximum points scored by a specific player position within a specific team--you must include all relevant columns in the grouping operation. This allows PySpark to perform a multi-dimensional aggregation, delivering precise metrics for specific subgroups.

When grouping by multiple columns, the composite key (e.g., Team + Position) dictates the resulting output rows. This operation is computationally more intensive than single-column grouping because the shuffle operation must correctly route data based on a concatenated key hash, demanding careful management of data locality and serialization. Despite the increased complexity, the DataFrame API handles this seamlessly, maintaining high performance.

import pyspark.sql.functions as F

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

In this scenario, the groupBy() function accepts two arguments: `'team'` and `'position'`. PySpark calculates the maximum value of `'points'` for every distinct combination of team and position found in the dataset. This level of granularity is essential when analyzing hierarchical or segmented data distributions, ensuring that aggregations are precise to the defined subgroups and providing deep analytical insights.

Setting Up the Example PySpark DataFrame

To illustrate these aggregation techniques in practice, we will utilize a sample PySpark DataFrame containing typical sports statistics. This dataset records information about various basketball players, including their team assignment, playing position, and points scored. The setup process involves initializing a **SparkSession**, defining the data schema, and creating the DataFrame, which is the standard procedure for any PySpark workflow.

A crucial first step in any PySpark operation is ensuring the SparkSession is properly initialized, as this serves as the entry point to all Spark functionality. Once the session is active, we define the raw data list and the corresponding column names, maintaining clarity and structure before

ingestion into the distributed DataFrame object. The resulting DataFrame structure allows for straightforward demonstration of the aggregation methods, serving as a reliable source for validating the subsequent grouping calculations.

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, `df`, provides a clear view of the underlying data, making it easy to manually verify the results of the subsequent aggregation operations. We can now proceed to apply the grouping logic defined previously to calculate the maximum points achieved based on differing levels of grouping complexity.

Example 1: Aggregating Max Points by Team

This first example demonstrates the application of Method 1, focusing solely on the `'team'` column to define the groups. The objective is to identify the highest score recorded by any player belonging to each respective team (A, B, and C). This is a simple but powerful aggregation that summarizes the maximum potential output achieved by each group identifier, often serving as a high-level performance metric.

The syntax employs the standard PySpark aggregation pattern: `df.groupBy('team').agg(F.max('points'))`. By selecting only one grouping column, the shuffle operation is minimized relative to multi-column grouping, optimizing resource usage. The output DataFrame will contain two columns: the grouping key (`team`) and the resulting aggregated maximum value (`max(points)`), demonstrating the peak performance level for each organization.

import pyspark.sql.functions as F

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

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

Analyzing the computed output allows us to draw immediate conclusions about the scoring capabilities across teams:

The max points value achieved by any player on **Team A** is **22**.
The max points value achieved by any player on **Team B** is **14**.
The max points value achieved by any player on **Team C** is **8**.

Example 2: Aggregating Max Points by Team and Position

To perform a more detailed analysis, we now implement Method 2, grouping the data based on two distinct categorical columns: `'team'` and `'position'`. This technique provides insights into the maximum performance achieved within specific roles across the teams, allowing for performance comparisons between Guards on Team A versus Guards on Team B, for instance, which is crucial for role-specific evaluations.

When using the agg() function following a multi-column group, the resulting DataFrame will list every unique combination of the grouping columns and the corresponding maximum score. This process requires PySpark to perform a more complex shuffle to ensure all records belonging to the same (Team, Position) tuple are routed to the same executor for accurate aggregation, highlighting the power of Spark's distributed architecture.

import pyspark.sql.functions as F

#calculate max of 'points' grouped by '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|
+----+--------+-----------+
```

Interpretation of this granular output confirms the maximum points attained within each subgroup, providing a rich, role-specific performance breakdown:

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**.
The max points value for **Forwards** on Team B is **7**.
The max points value for **Guards** on Team C is **8**.
The max points value for **Forwards** on Team C is **5**.

Optimizing PySpark Aggregations for Performance

While `groupBy()` and `agg()` provide the necessary functionality, achieving optimal performance in large-scale distributed computing requires careful consideration of optimization techniques. The primary bottleneck in grouping operations is often the data shuffle--the physical movement of data across the network to bring records with the same group key onto the same machine. Minimizing this shuffle is paramount for efficiency.

One highly effective optimization strategy involves using specialized functions when applicable. For instance, if performing multiple aggregations, listing them all within a single agg() function call (e.g., `df.groupBy().agg(F.max('col1'), F.sum('col2'))`) ensures that the expensive grouping and shuffling operations occur only once. In contrast, running multiple separate `groupBy()` operations for different metrics forces Spark to repeatedly perform the shuffle, severely diminishing performance.

Furthermore, developers should always leverage the Catalyst Optimizer by using DataFrame API operations and PySpark built-in functions. These built-in functions execute optimized Scala/Java code under the hood, avoiding the need for Python serialization/deserialization that is characteristic of UDFs. For maximum calculation specifically, the `F.max()` function is inherently non-blocking and highly optimized for columnar data processing, guaranteeing swift execution even on massive datasets.

Conclusion: The Power of Vectorized Aggregation

The ability to efficiently calculate maximum values within defined groups is essential for exploratory data analysis and feature engineering in big data environments. By strategically employing the `groupBy()` method coupled with the `agg(F.max())` function in PySpark, practitioners can ensure that their statistical summaries are generated quickly and reliably, even when processing petabytes of data.

The choice between grouping by a single column versus multiple columns depends entirely on the required analytical granularity. Regardless of the complexity of the grouping key, the principles of relying on DataFrame API optimizations and native Spark functions remain constant, ensuring robust and scalable data pipelines. Mastering these aggregation techniques forms a cornerstone of effective data manipulation using the Apache Spark framework.

Further PySpark Tutorials

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