How to Select the Row with the Maximum Value in Each Group Using PySpark

Authored by

stats writer

January 19, 2026

1. Introduction: The Challenge of Grouped Maximums

When working with large datasets in PySpark, a common requirement is to identify the entire row corresponding to the maximum value of a specific column, partitioned across different categories or groups. For instance, if you have sales data, you might need to find the specific transaction details (the entire row) that resulted in the highest sale amount for each regional manager. Achieving this efficiently requires specialized techniques beyond simple aggregation, as standard grouping methods often discard the detailed record information.

The most robust and performant way to solve this challenge in PySpark involves leveraging the power of **Window functions**. Unlike standard aggregation which collapses rows, Window functions perform calculations over a set of rows related to the current row, preserving the detail level of the original DataFrame. This approach ensures that we can identify the maximum value within a group and subsequently filter the original records based on that calculated maximum.

2. The Core PySpark Solution using Window Functions

To select the full record associated with the maximum value grouped by a specified column in a DataFrame, we utilize the Window function in conjunction with aggregation functions like max. The general pattern involves three critical steps: defining the window specification (the grouping logic), calculating the maximum value within that window, and finally filtering the original data based on that temporary maximum column.

The following syntax template provides the clean and efficient method for solving this problem. In this example, we aim to find the row with the highest value in the `points` column, categorized by the `team` column.

You can use the following syntax to select the row with the max value by group in a PySpark DataFrame:

```
from pyspark.sql import Window
import pyspark.sql.functions as F

#specify column to group by
w = Window.partitionBy('team')

#find row with max value in points column by team
df.withColumn('maxPoints', F.max('points').over(w))
.where(F.col('points') == F.col('maxPoints'))
.drop('maxPoints')
.show()
```

This code snippet effectively returns a DataFrame containing only the rows where the value in the `points` column matches the calculated maximum value for its respective `team` (the defined group). We temporarily create an auxiliary column, `maxPoints`, to hold this group maximum before filtering and then dropping it to maintain a clean output structure.

3. Understanding the Window Specification (partitionBy)

The core of this solution lies in defining the **Window** object, often aliased as `w` in examples. A Window function requires a definition specifying which rows are grouped together for calculation, known as the window frame. The `Window.partitionBy()` method dictates how the data should be segmented. In our case, `Window.partitionBy('team')` instructs PySpark to treat all rows belonging to the same team as a single partition.

When the aggregation function `F.max('points').over(w)` is applied, PySpark calculates the maximum value of the `points` column independently within each specified partition (each team). Crucially, this maximum value is then repeated across all rows belonging to that partition in the newly created `maxPoints` column. This repetition is what allows for the subsequent comparison and filtering step.

4. Setting Up the Example DataFrame

To demonstrate this functionality in a practical setting, let us first construct a sample DataFrame. This dataset contains simulated basketball player statistics, including their team affiliation, points scored, and assists made. Our goal will be to identify which player achieved the highest point total for each team (A, B, and C).

We initialize the SparkSession and define the raw data and column structure necessary to create the initial DataFrame. This setup is crucial for verifying the accuracy of the grouped maximum selection method.

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|points|assists|
+----+------+-------+
| A| 18| 3|
| A| 33| 5|
| A| 12| 8|
| A| 15| 10|
| B| 19| 4|
| B| 24| 4|
| B| 28| 2|
| C| 40| 7|
| C| 24| 3|
| C| 13| 4|
+----+------+-------+
```

5. Applying the Window Function for Maximum Value Selection

With our sample data prepared, we can now execute the core logic. We instantiate the Window function by partitioning by `team`. The chained operations `withColumn`, `where`, and `drop` ensure that we calculate the maximum, filter based on that maximum, and clean up the result in a single, fluent expression. This chaining of DataFrame transformations is idiomatic PySpark practice.

The syntax below demonstrates how to return a filtered DataFrame that contains the entire row associated with the max value in the `points` column for each unique team group:

from pyspark.sql import Window
import pyspark.sql.functions as F

```
#specify column to group by
w = Window.partitionBy('team')

#find row with max value in points column by team
df.withColumn('maxPoints', F.max('points').over(w))
.where(F.col('points') == F.col('maxPoints'))
.drop('maxPoints')
.show()

+----+------+-------+
|team|points|assists|
+----+------+-------+
| A| 33| 5|
| B| 28| 2|
| C| 40| 7|
+----+------+-------+
```

6. Step-by-Step Breakdown of the Code Logic

Understanding the flow of transformations is essential for debugging and optimization. The process executes sequentially through the chained methods:

Window Definition: `w = Window.partitionBy('team')` creates the window specification, defining the boundaries for the subsequent aggregation.

Calculation: `.withColumn('maxPoints', F.max('points').over(w))` calculates the maximum `points` within each team partition and adds this value to every row of that team under the new `maxPoints` column.

Filtering: `.where(F.col('points') == F.col('maxPoints'))` performs the crucial filtering operation. It retains only those original rows where the individual player's `points` score is exactly equal to the calculated group maximum held in `maxPoints`. If multiple players tied for the maximum points, all those rows would be retained.

Cleanup: `.drop('maxPoints')` removes the temporary auxiliary column used for the comparison, ensuring the output DataFrame maintains the structure of the original dataset.

This sequence guarantees that we isolate the specific rows that hold the maximum value based on the partition key, providing an efficient alternative to traditional `groupBy` followed by a join operation.

7. Analyzing the Final Result

The final output DataFrame is concise and contains only the records representing the highest scoring player for each respective team.

For Team A, the maximum score observed was **33**, corresponding to the row where assists were 5.

For Team B, the maximum score was **28**, paired with 2 assists.

For Team C, the highest score was **40**, associated with 7 assists.

The resulting DataFrame successfully achieved the goal: isolating the entire record linked to the maximum value within each distinct group, while preserving all original columns. This confirms the effectiveness of the Window function methodology for group-wise maximum retrieval in large-scale data processing.

8. Conclusion and Further Reading

Using **Window functions** with `partitionBy` is the standard, high-performance method in PySpark for solving complex analytical problems that require calculations across related rows while maintaining dataset granularity. While the `max().over(w)` approach detailed here is excellent for finding the maximum value record, alternative analytic functions like `row_number()` or `rank()` can be used when the requirement is to select the top N rows per group, offering even greater flexibility.

Mastering Window functions is a crucial step for any data engineer utilizing PySpark for advanced data transformation. These functions streamline complex operations that would otherwise require multiple steps involving grouping, aggregation, and expensive joins.

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