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

Authored by

stats writer

February 5, 2026

PySpark is a powerful and essential framework for modern Big Data processing, enabling users to manage, manipulate, and analyze massive datasets with remarkable efficiency and scalability. A frequently encountered analytical requirement involves identifying the record associated with the maximum value within predefined sub-groups of the data. This capability is vital for tasks such as pinpointing top sales performers in each region, identifying the highest temperature reading at various sensors, or, as demonstrated here, determining the top scorer on every team. While standard grouping methods often return only the aggregated maximum value, the challenge lies in retrieving the entire corresponding row of data. By strategically employing PySpark's specialized Window function capabilities, developers can elegantly solve this problem, extracting valuable insights that drive informed decisions through convenient and efficient data analysis.

PySpark: Select Row with Max Value in Each Group

Understanding the Necessity of Window Functions for Group Maximums

When working with large-scale data in PySpark, a common operation is finding the maximum value of a specific column. However, simply using the standard `groupBy()` followed by an `agg(max())` function will collapse the groups, returning only the grouping column and the maximum aggregated value, thereby discarding all other columns associated with that maximum row. To retrieve the complete row--which often contains critical identifying information or related metrics--we must employ a more sophisticated technique. This is where the powerful concept of the Window function comes into play.

A Window function operates on a set of rows related to the current row, known as a 'window,' and computes a result for each row based on that window. Unlike traditional aggregation functions that reduce the number of rows, Window functions return a result for every input row, making them ideal for analytical tasks such as ranking, calculating running totals, or, in this case, comparing a row's value against the maximum value found across its peers within a specific partition. This approach maintains the original structure of the DataFrame while allowing for complex group-level calculations.

The core strategy involves two primary steps: first, calculating the maximum value for each group using the Window function and storing it temporarily as a new column; and second, filtering the original DataFrame based on where the original value equals the newly calculated maximum group value. This comparison effectively isolates the exact row or rows (in case of ties) that contain the maximum value for that group, achieving the desired result without loss of detail.

Prerequisites and Essential Imports for Window Operations

To successfully implement this group-wise maximum selection logic in PySpark, two essential

components must be imported from the `pyspark.sql` library. These imports provide the necessary tools for defining the scope of the calculation and applying the aggregate functions within that scope.

The `Window` Class: This class is crucial as it allows us to define the boundaries and behavior of the window operation. The most important method we utilize is `Window.partitionBy()`, which specifies which column or columns define the groups over which the aggregation (finding the maximum) will occur. Think of `partitionBy()` as the equivalent of the `GROUP BY` clause in standard SQL, but applied non-destructively.

`pyspark.sql.functions`: We typically import this module and alias it as `F`. This module contains a vast collection of built-in functions, including aggregation functions like `max()`, `min()`, and `avg()`, which we will apply over the defined window. Using the alias `F` makes the code cleaner and easier to read.

The syntax for defining the window involves calling `Window.partitionBy('column_name')`. This creates a window specification (often stored in a variable like `w`) that tells PySpark to divide the entire DataFrame into distinct partitions based on the unique values in the specified column. All subsequent window functions applied using `.over(w)` will operate independently within these partitions.

The Canonical Syntax for Maximum Value Selection by Group

The following syntax provides a robust and highly scalable method for selecting the row corresponding to the maximum value within a specified group in a PySpark DataFrame. This pattern leverages the efficiency of Window function processing, which Spark optimizes internally.

```
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()
```

Let us break down the execution steps within this code block. First, the `w = Window.partitionBy('team')` line defines the segmentation rule, ensuring that the aggregation

respects the boundaries of each unique team. Second, the `df.withColumn('maxPoints', F.max('points').over(w))` clause is critical: it iterates through every row, calculates the maximum value of the `points` column within that row's respective partition (team), and creates a temporary column called `maxPoints` containing this group maximum. This value is identical for every row belonging to the same team.

The third stage, `.where(F.col('points') == F.col('maxPoints'))`, acts as the filter. Since the `maxPoints` column holds the calculated group maximum, we only retain the rows where the player's individual `points` value exactly matches the group's maximum value. This comparison effectively selects only the top player(s) from each team. Finally, `.drop('maxPoints')` ensures that the temporary helper column, which is no longer needed after the filtering step, is removed, leaving a clean, concise result DataFrame that contains the rows with the maximum value in the **points** column for each unique value in the **team** column.

Setting Up the Practical PySpark Example Dataset

To illustrate the application of this powerful syntax, consider a scenario involving basketball player statistics. We aim to identify the player who achieved the highest point total for each distinct team recorded in our dataset. This example requires initial setup of a PySpark DataFrame, which involves establishing a Spark Session and defining the raw data and schema.

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

The initial dataset, shown above, contains three columns: **team** (the grouping variable), **points** (the variable we are maximizing), and **assists** (additional relevant player data). Notice that within team 'A', points range from 12 to 33. For team 'B', points range from 19 to 28. For team 'C', points range from 13 to 40. Our objective is to execute a query that efficiently determines the row corresponding to 33 for Team A, 28 for Team B, and 40 for Team C, while retaining the associated assists values and team identifiers.

Applying the Window Function to Identify Team Scorers

Using the established Window function methodology detailed previously, we can now apply the necessary transformations to the `df` DataFrame to select the desired rows. This process demonstrates the precision and non-destructive nature of PySpark Window function operations.

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

The execution begins by defining the window `w` based on the **team** column. When `F.max('points').over(w)` is calculated, Spark internally identifies 33 as the max for all Team A rows, 28 for all Team B rows, and 40 for all Team C rows. By filtering the DataFrame where the individual **points** column matches this calculated group maximum, we effectively isolate the top performer's row for each partition. This single, streamlined chain of operations replaces complex iterative loops or less performant SQL subqueries.

Analyzing the Resulting DataFrame and Efficiency

The resulting DataFrame confirms the successful extraction of the rows with the maximum point values per team. For instance, the max points value among players on team A was **33**, achieved by the player who also recorded 5 assists. Similarly, the highest score for team C was **40**, associated with 7 assists. The final output is concise, containing only three rows--one representative row for each unique team in the original dataset.

This approach is highly favored in PySpark due to its adherence to Spark's core principles of distributed computing. When we use the Window function combined with `partitionBy`, Spark executes the grouping and aggregation logic across the cluster efficiently. This method avoids the need for a costly secondary join (which would be required if we first aggregated and then attempted to join back to the original DataFrame to retrieve the remaining columns), making it the optimal pattern for this type of analytical query on massive datasets.

Alternative Group Maximum Selection Method: Row Number and Ranking

While the `max()` and filter approach is effective, an alternative and equally common method for selecting the top record within a group involves using ranking Window functions like `row_number()`, `rank()`, or `dense_rank()`. This method is particularly useful when you need to select the top N rows per group, not just the single maximum.

To use a ranking function, the window specification must include an `orderBy()` clause in addition to `partitionBy()`. For instance, to select the highest points, you would partition by team and order by points in descending order. The `row_number()` function then assigns a rank of 1 to the highest row within each partition. You would then filter for rows where the rank is equal to 1.

Although both the `max()` filter method and the `row_number()` method achieve the same result for selecting the single maximum row, the `max()` filter is often slightly more intuitive when the goal is strictly finding the maximum value, whereas the ranking functions offer more flexibility for complex top-N scenarios. Mastering both techniques ensures comprehensive proficiency in advanced PySpark data manipulation.

Further PySpark Data Manipulation Tutorials

The principles of partitioning, aggregating, and filtering demonstrated here are foundational to advanced data processing in PySpark.

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