How to Sort PySpark Data in Descending Order Using Window.orderBy()

Authored by

stats writer

February 3, 2026

PySpark, the Python API for Apache Spark, offers incredibly powerful tools for large-scale data manipulation, among which are its specialized Window functions. These functions enable complex calculations across a defined group of rows that are related to the current row, without collapsing the groups, similar to how aggregate functions work in standard SQL but retaining individual rows. One essential component for defining how data is processed within these windows is the ability to order the data.

The **`Window.orderBy()`** function is precisely designed for this purpose within the PySpark DataFrame environment. It facilitates the crucial step of sorting the data within each specified partition of the window. While it naturally defaults to ascending order, using it in conjunction with other PySpark functions allows for highly efficient organization of data in a specific column in a **descending fashion**, based on a set of criteria defined by the data analyst.

Mastering the appropriate use of **`Window.orderBy()`** is fundamental for advanced data analysis tasks, such as calculating rolling averages, ranking records, or determining differences between consecutive rows. By carefully specifying the target column and the sorting logic, analysts gain a potent tool for accurately arranging massive datasets, making complex data transformations both efficient and manageable within the distributed computing context of Spark.

PySpark: How to Utilize Window.orderBy() for Descending Data Sorting

Core Syntax for Descending Order Sorting in PySpark

To effectively leverage **`Window.orderBy()`** to achieve a descending sort within a DataFrame, it is essential to import the necessary helper function from the `pyspark.sql.functions` module. Specifically, we must utilize the **`desc()`** function, which explicitly tells the sorting mechanism to reverse the natural ascending order. The following syntax illustrates the standard approach for defining a window specification that incorporates grouping using `partitionBy()` and sorting using `orderBy()` with the required descending modification.

from pyspark.sql.functions import row_number, desc
from pyspark.sql.window import Window

# Specify the window criteria
w = Window.partitionBy('team').orderBy(desc('points'))

# Add a new column ('id') containing sequential row numbers based on the defined window
df = df.withColumn('id', row_number().over(w))

In this highly functional example, the process first defines a window object named `w`. This window specification dictates that rows are grouped by the values found in the `team` column via **`partitionBy()`**. Crucially, within each of these teams (partitions), the records are then ordered based on the `points` column using **`orderBy()`**, which wraps the column name inside the **`desc()`** function to ensure the highest point totals appear first.

Following the window definition, a new column called `id` is generated using **`withColumn()`**. This column utilizes the **`row_number()`** window function, which assigns a sequential integer to each row within the window (partition). Because the window `w` specifies descending order by `points`, the resulting `id` column represents a descending rank or sequence number based on the scores achieved within each respective team.

Practical Example: Setting Up the DataFrame

To demonstrate the practical application of **`Window.orderBy()`** in descending mode, let us construct a simple PySpark DataFrame containing hypothetical data about basketball players. This dataset includes player information organized by team, position, and the number of points scored. This setup is typical for scenarios where relative ranking within distinct groups is required.

We begin by initializing a **SparkSession** and defining the raw input data as a list of lists. The column names are also explicitly defined to ensure clarity and proper structure upon DataFrame creation. This standardized approach guarantees that the subsequent window operations are applied to a correctly formatted structure, allowing us to accurately track how the descending order sorting affects the row numbering process across different teams.

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

# Define the input data for players and scores
data = ,
,
,
,
,
,
,
,
]

Define column names
columns =

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

# Display the initial DataFrame structure
df.show()
```

```
+----+--------+------+
|team|position|points|
+----+--------+------+
| A| Guard| 11|
| A| Guard| 8|
| A| Forward| 21|
| A| Forward| 22|
| A| Forward| 30|
| B| Guard| 14|
| B| Guard| 14|
| B| Forward| 13|
| B| Forward| 7|
+----+--------+------+
```

The initial DataFrame successfully loads the nine records, showcasing players across Team A and Team B, each with varying point totals. The objective now is to assign a rank or sequential identifier to each player within their respective team based on their score, prioritizing the highest scores by sorting in **descending order**.

Defining the Window Specification for Grouped Sorting

The critical step in achieving the desired descending order ranking is defining the Window specification correctly. We require a mechanism that resets the sequence count every time the `team` changes, which is handled by **`partitionBy('team')`**. Furthermore, we must ensure that the sorting within those partitions is based on the `points` column, arranged from highest to lowest.

Therefore, the task is to add a new column, conventionally named `id` for demonstration purposes, that provides row numbers for every record, grouped by the `team` column and ordered by the values in the `points` column in the specified **descending order**. This operation effectively assigns the rank 1 to the top scorer on Team A, and independently, assigns rank 1 to the top scorer(s) on Team B, illustrating the power of partitioned window functions.

Applying the Window Function and Reviewing Results

We implement the window definition and subsequent column creation using the precise syntax detailed earlier. This involves importing `row_number` and `desc`, defining the window variable `w` using `partitionBy` and `orderBy(desc(...))`, and finally applying `row_number().over(w)` to the DataFrame using `withColumn`.

```
from pyspark.sql.functions import row_number, desc
from pyspark.sql.window import Window

# Specify window: Partition by team, order by points in descending order
w = Window.partitionBy('team').orderBy(desc('points'))

# Add column called 'id' that contains row numbers
df = df.withColumn('id', row_number().over(w))

# View the resulting DataFrame
df.show()

+----+--------+------+---+
|team|position|points| id|
+----+--------+------+---+
| A| Forward| 30| 1|
| A| Forward| 22| 2|
| A| Forward| 21| 3|
| A| Guard| 11| 4|
| A| Guard| 8| 5|
| B| Guard| 14| 1|
| B| Guard| 14| 2|
| B| Forward| 13| 3|
| B| Forward| 7| 4|
+----+--------+------+---+
```


The resultant DataFrame clearly shows the successful execution of the descending sort within the window partitions. The newly created column named `id` contains sequential row numbers, which are reset for Team B. More critically, the values within each partition are sorted based on the `points` column, ensuring that the highest score receives the lowest row number (i.e., rank 1).

Detailed Analysis of the Descending Row Numbering

A closer inspection of the output confirms that the use of `Window.orderBy(desc('points'))`

achieved the intended ranking within groups. The sequential numbering provided by `row_number()` now directly corresponds to the descending performance ranking of players based on their scores within their respective teams.

For instance, observe the results for Team A. The player with 30 points is assigned an ID of 1, followed by the player with 22 points (ID 2), and so on, until the lowest score of 8 points receives an ID of 5. This progression confirms the descending order sort was correctly applied to the points column.

Similarly, for Team B, the highest point totals (two players tied at 14 points) receive the first available row numbers. Note that because `row_number()` assigns sequential integers, tied values will receive unique, consecutive ranks. The two players scoring 14 points are assigned `id` values of **1** and **2**, demonstrating that the ordering process correctly handled the tie by maintaining the descending sequence before moving to the next highest score (13 points, ID 3).

To summarize the outcome based on the partitioning and ordering rules:

The row with the largest points value for Team A (30 points) receives an **id** value of **1**, marking it as the top scorer in that partition.

The row with the next largest points value for Team A (22 points) receives an **id** value of **2**, continuing the descending sequence.

For tied scores, such as the two players on Team B scoring 14 points, `row_number()` assigns the next consecutive values, **1** and **2**, based on the internal ordering of the tied rows.

Key Considerations and Alternatives

When implementing window functions, understanding the role of the `desc()` function is paramount. Had we omitted the `desc()` wrapper within the `orderBy()` function, the default behavior of PySpark would have been triggered. The row numbers would have been assigned based on the values in the `points` column in **ascending order** instead, meaning the lowest score would have received the rank 1, which is often not the intended result when calculating performance metrics or rankings.

Furthermore, while this tutorial focuses on `row_number()`, the same descending order principle applies when using other ranking window functions, such as `rank()` or `dense_rank()`, which handle ties differently. The integrity of the sorting operation always relies on the correct application of the `desc()` function alongside `Window.orderBy()`. For developers seeking further technical specifications and alternative parameters, the complete documentation for the PySpark Window.orderBy function is the definitive resource.