How to Sort a PySpark Pivot Table by Column Values

Authored by

stats writer

January 19, 2026

PySpark is an incredibly powerful unified analytics engine designed for large-scale data processing. One of its most common applications involves transforming raw data into meaningful summaries using aggregation techniques, often resulting in a pivot table. A pivot table reshapes a DataFrame, allowing users to aggregate data across specified rows and columns. While creating a pivot table using the PySpark dataframe API is straightforward, the subsequent step of ordering the resulting data structure based on the aggregated values within a specific column often requires careful application of the sorting functions.

This article provides a comprehensive guide on how to effectively sort a generated pivot table in PySpark. We will demonstrate how to utilize the built-in sorting capabilities, specifically the **orderBy** function, to resequence the rows based on the calculated values contained in any designated column. This capability is essential for generating reports that require ranked or ordered views, ensuring that the most relevant data points are immediately visible for subsequent analysis or visualization workflows.

Core Syntax: Sorting Pivot Tables with orderBy()

When working with PySpark, the fundamental method for sorting any DataFrame--including those resulting from pivot operations--is through the use of the orderBy method. This function is attached directly to the DataFrame object and takes one or more column names as arguments, specifying the criteria by which the rows should be ordered. Understanding this basic syntax is the first step toward effective data manipulation after aggregation.

The core syntax required to sort the rows in a PySpark pivot table (which is itself a standard DataFrame) based on the values in a specific column is outlined below. It is important to remember that the column name referenced must correspond exactly to one of the newly created columns in the pivoted result.

You can use the following syntax to sort the rows in a pivot table in PySpark based on values in a specific column:

df_pivot.orderBy('my_column').show()

This particular example sorts the rows in the pivot table called **df_pivot** based on the values present in the column named **my_column**. By default, PySpark performs an ascending sort, meaning it orders the values from smallest to largest or alphabetically from A to Z. We will later explore how to reverse this default behavior to achieve descending order.

Practical Demonstration: Setting Up the PySpark Environment

To illustrate the process of sorting a pivot table, we must first establish a reproducible environment

and create a sample dataset. All operations in PySpark begin with a SparkSession, which serves as the entry point to communicate with the underlying Spark functionality. This session allows us to define and manipulate DataFrames.

Suppose we have the following PySpark DataFrame that contains information about the points scored by various basketball players. This dataset is small enough for demonstration purposes yet complex enough to require effective pivoting and sorting for meaningful insight extraction.

The following code snippet demonstrates how to initialize the Spark session, define the raw data, specify the column headers, and finally construct the PySpark DataFrame, displaying the initial structure for confirmation:

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	14
A	Guard	4
A	Forward	16

A	Forward	18
B	Guard	9
B	Forward	5
B	Forward	25
C	Forward	12
C	Guard	14
C	Guard	23
+----+--------+------+

As shown in the output, the resulting DataFrame, named `df`, provides granular player data, listing individual point totals. This raw structure is the foundation upon which we will build our aggregated pivot table.

Step 1: Generating the Pivot Table

The process of pivoting involves grouping the data by one or more columns (the row keys) and then transforming unique values from another column (the pivot column) into new columns. Concurrently, an aggregation function is applied to summarize the values within the intersection of the new rows and columns. In our basketball example, we are interested in summing the total points scored by each team, broken down by player position (Forward vs. Guard).

We achieve this aggregation by first using the `groupBy('team')` method, specifying that 'team' will form the rows of our pivot table. Next, we use the `pivot('position')` method, instructing PySpark to convert the unique values within the 'position' column (Guard and Forward) into new columns. Finally, we apply the `sum('points')` function to calculate the total points for each team-position intersection.

We can use the following syntax to create a pivot table using **team** as the rows, **position** as the columns and the sum of **points** as the values within the pivot table:

#create pivot table that shows sum of points by team and position
df_pivot = df.groupBy('team').pivot('position').sum('points')

#view pivot table
df_pivot.show()

+----+-------+-----+
|team|Forward|Guard|
+----+-------+-----+
| B| 30| 9|
| C| 12| 37|

```
| A| 34| 18|
+----+-------+-----+
```

The resulting pivot table shows the sum of the points values for each team and position. The current row order is arbitrary, reflecting how PySpark processed the teams, making immediate comparison difficult.

Step 2: Sorting the Pivot Table in Ascending Order

Once the pivot table (`df_pivot`) is created, sorting it requires calling the orderBy function. When sorting is performed without any additional parameters, PySpark defaults to ascending order. This means that if we sort by the 'Forward' column, the team with the lowest total points scored by forwards will appear first.

Ascending sorts are useful when you want to identify minimum values quickly or find entries that fall below a certain threshold. In our scenario, sorting ascendingly by 'Forward' points will rank the teams from the lowest total 'Forward' points to the highest.

We can use the following syntax to sort the rows of the pivot table in ascending order based on the values in the **Forward** column:

#sort rows of pivot table by values in 'Forward' column in ascending order
df_pivot.orderBy('Forward').show()

```
+----+-------+-----+
|team|Forward|Guard|
+----+-------+-----+
| C| 12| 37|
| B| 30| 9|
| A| 34| 18|
+----+-------+-----+
```

Notice that the rows in the pivot table are now sorted in ascending order (12, 30, 34) based on the values in the **Forward** column. This demonstrates the default sorting behavior of the orderBy function.

Step 3: Sorting the Pivot Table in Descending Order

In many data analysis tasks, users prefer to see the highest values first--for example, ranking teams from best to worst performance. To achieve this, we must explicitly instruct PySpark to sort

in descending order. This is done by passing the optional parameter `ascending=False` within the orderBy function call.

A descending sort arranges the data from the largest value to the smallest, effectively highlighting the top performers immediately. This is particularly useful for leaderboards or performance reports where quick identification of maximum values is necessary.

If you would instead like to sort the rows in descending order, you can use the argument **ascending=False** as follows:

#sort rows of pivot table by values in 'Forward' column in descending order
df_pivot.orderBy('Forward', ascending=False).show()

```
+----+-------+-----+
|team|Forward|Guard|
+----+-------+-----+
| A| 34| 18|
| B| 30| 9|
| C| 12| 37|
+----+-------+-----+
```

The rows in the pivot table are now sorted in descending order (34, 30, 12) based on the values in the **Forward** column. This simple modification provides the desired ranking for performance comparison.

Further Resources and PySpark Tutorials

Understanding the PySpark **orderBy** function is critical for producing clean, ordered results from complex aggregations like pivot tables. The ability to specify ascending or descending order allows for flexible reporting and visualization tailored to specific business or analytical requirements. Remember that `orderBy` can also accept multiple column arguments for hierarchical sorting (e.g., sort by Column A, then by Column B to break ties).

For users interested in exploring the complete range of parameters and features available for the sorting operation, including detailed handling of null values and complex expressions, the official documentation remains the ultimate reference.

Note: You can find the complete documentation for the PySpark **orderBy** function on the Apache Spark official website.

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