How to Sort a PySpark Pivot Table by Column Values

Authored by

stats writer

February 5, 2026

PySpark, the Python API for Apache Spark, provides robust tools for large-scale data processing and analysis. Among the most essential tools are **pivot tables**, which enable users to transform rows into columns, offering a succinct, summarized view of complex datasets. These structures are instrumental in data aggregation, allowing analysts to group information based on specific dimensions and calculate summarized metrics, such as sums, averages, or counts. However, merely creating a pivot table is often insufficient for comprehensive data interpretation.

While a pivot table efficiently summarizes data, the order in which the results are presented significantly impacts the ease of analysis. When analyzing aggregated metrics--for example, total sales per region or total points scored per position--it is crucial to rank these results. If the data is not ordered, identifying the top performers or key outliers requires manual scanning, which becomes infeasible with large DataFrames. Therefore, the ability to sort the resulting pivot table based on the aggregated values within a specific column is a fundamental requirement for gaining actionable insights and making data-driven decisions swiftly.

This guide details the precise methodology for sorting a pivot table generated in PySpark. We will focus on utilizing the built-in sorting capabilities of the DataFrame API, specifically the `orderBy` function. Understanding this function's application post-pivot is key to structuring analytical outputs effectively, ensuring that the most critical information--the highest or lowest values in a given metric--is presented immediately and clearly to the analyst.

PySpark: Sort Pivot Table by Values in Column

Understanding the PySpark orderBy Function

The core mechanism for sorting data in PySpark is the `orderBy` function, which is available on all DataFrame objects. This function is an alias for the more general `sort` function but is often preferred for its clarity. It instructs the Spark engine to reorganize the rows of the DataFrame based on the values found in one or more specified columns. Since the output of a pivoting operation is itself a DataFrame, this sorting mechanism applies directly and seamlessly to the aggregated results.

When sorting a standard DataFrame, one might order by an existing dimension column (e.g., 'date' or 'name'). However, when dealing with a pivot table, the relevant columns for sorting are usually the newly created aggregated columns (e.g., 'Sum_of_Sales' or 'Guard' points). The orderBy function handles this structure naturally; one simply passes the name of the desired aggregated column as an argument. PySpark's sorting logic is designed to efficiently handle distributed data sorting, optimizing the shuffle operations necessary to reorder the results across the cluster nodes.

A key advantage of using `orderBy` is its scalability across vast datasets. Regardless of whether

the resulting pivot table contains tens of rows or millions, Spark manages the sorting process by distributing the workload. By default, `orderBy` sorts in **ascending order**. If a descending sort is required--typically to show the highest values first--an optional parameter must be explicitly passed to reverse the order of the resultant dataset.

Syntax for Sorting Pivot Tables in PySpark

The syntax for applying the sorting operation is straightforward, assuming the pivot table has already been generated and assigned to a DataFrame variable, which we will call `df_pivot` in our examples. The fundamental structure involves calling `orderBy` directly on this pivoted DataFrame object.

To sort the rows based on the values within a specified column--let's use a placeholder name `my_column`--the basic command is as follows. This syntax automatically results in an ascending sort, placing the lowest values of `my_column` at the top of the resulting table display.

The general structure required to implement this sorting action, followed immediately by the display command, is:

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

This command executes the sort operation on the DataFrame named **df_pivot**. The rows are rearranged according to the numerical or lexicographical values present in the column identified by the string **my_column**. Note that in the context of a pivot table, **my_column** would represent one of the aggregated measure columns created during the pivoting process.

Example: How to Sort a Pivot Table in PySpark

We will now illustrate this functionality using a concrete example based on basketball player data. Suppose we have the following PySpark DataFrame that contains information about the points scored by various basketball players across different teams and positions. We must first initialize the Spark session and define our sample data.

The following code block outlines the setup and display of the initial, unsorted, raw data DataFrame, which will serve as the input for our pivoting operation:

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
+----+--------+------+

This initial DataFrame, `df`, contains granular point totals. Our goal is to transform this into a wide-format pivot table that aggregates the total points scored by each team, separating the totals by player position.

Generating the Aggregated Pivot Table

We utilize the `groupBy` and `pivot` functions to restructure the data. We use the **team** column as the grouping dimension (rows), the **position** column for the new column headers, and the sum of the **points** column as the aggregated values within the table cells. This creates the DataFrame `df_pivot`, which is the structure we need to sort.

The execution of the pivot operation and the display of the resulting aggregated data structure are shown below:

#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. As noted earlier, the current row order is arbitrary ('B', 'C', 'A'). We will now apply the sorting logic to order these teams based on a specific metric.

Sorting the Pivot Table in Ascending Order

To sort the rows of the pivot table in **ascending order** based on the values in the **Forward** column, we apply the `orderBy` function directly to `df_pivot`, specifying 'Forward' as the target column. Since no direction parameter is given, the sort defaults to ascending (lowest values first).

This operation is critical for quickly identifying the team with the minimum performance in the specified metric.

#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. Team C, having the lowest total of 12 forward points, is now positioned at the top of the table.

Sorting the Pivot Table in Descending Order

If the goal is to identify top performers or maximum values, we must sort the rows in **descending order**. To achieve this, we use the optional argument `ascending=False` within the orderBy function call.

Sorting in descending order is typically the most common requirement when analyzing aggregated performance data, as it ensures that the most impactful results are presented immediately at the start of the output. We apply this modification to the same 'Forward' column.

#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. Team A, with 34 points, is correctly identified as the leader in this metric.

Advanced Sorting and Data Type Considerations

The `orderBy` function is highly flexible, supporting more complex ranking criteria, such as sorting by multiple columns. If tie-breaking logic is necessary, multiple column names can be provided to the function. Furthermore, specialized functions like `desc()` or `asc()` (imported from `pyspark.sql.functions`) can be used to specify different sorting directions for individual columns within the same operation.

For instance, sorting first by 'Guard' descending, and then by 'Forward' ascending for tie-breaking would require: `df_pivot.orderBy(desc('Guard'), asc('Forward')).show()`. This advanced approach ensures complete control over the ranking hierarchy of the aggregated data.

It is also essential to confirm that the sorting column contains appropriate data types. Since pivot aggregations like `sum` produce numerical data, the sort operates numerically. If, however, the target column contained string data, the sort would be lexicographical, potentially leading to misleading results if numeric interpretation was intended. Always verify the schema of the resulting DataFrame after the pivot operation to guarantee expected sort behavior.

Summary of PySpark Pivot Table Sorting

The ability to generate comprehensive pivot tables is a key strength of PySpark for large-scale data aggregation. However, the true analytical value is unlocked when these results are presented in a meaningful order. By leveraging the `orderBy` function, analysts can seamlessly sort the resulting DataFrame based on any aggregated metric column produced during the pivoting process.

Whether the goal is to identify the lowest values using the default ascending sort or to highlight the top performers using `ascending=False`, the orderBy function provides the necessary precision and efficiency required in a distributed computing environment. Mastery of this simple command ensures that complex aggregation results are always immediately interpretable, significantly enhancing the speed and quality of data analysis workflows.

For detailed specifications and further optional parameters, users are encouraged to consult the official documentation for the PySpark **orderBy** function.

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