How to Count Unique Values in PySpark Like Pandas value_counts()

Authored by

stats writer

January 21, 2026

The transition from single-machine data analysis tools like Pandas to distributed computing frameworks such as PySpark often requires finding equivalents for familiar operations. A frequently used function in data profiling and exploration is `value_counts()`, which provides a concise summary of the frequency of unique values within a column. While PySpark does not offer a direct, identically named function, the equivalent functionality is achieved by chaining the **`groupBy()`** and **`count()`** methods. This combination is fundamental for performing aggregation tasks across large-scale distributed DataFrames.

Understanding how to replicate this essential operation is vital for analysts and data scientists leveraging the scalability of Apache Spark. By utilizing the PySpark API, we can efficiently group records based on a specified column and then calculate the total frequency for each distinct group, effectively replicating the behavior of value_counts(). This article provides a comprehensive guide to implementing and refining this technique using various sorting and aggregation strategies within the PySpark environment.

Understanding value_counts() in Pandas

Before diving into the PySpark implementation, it is helpful to briefly recap the utility of the `value_counts()` function in Pandas. This method, usually applied to a Series (a single column of a DataFrame), rapidly returns a Series containing counts of unique values. The resulting output is typically sorted in descending order by count, making it instantaneous to identify the most frequent occurrences in the dataset. This immediate frequency analysis is crucial for initial data cleaning and understanding categorical distributions.

The primary strength of `value_counts()` lies in its simplicity and default sorting behavior. It provides an immediate ranking of categories, which is often the first step in exploratory data analysis (EDA). When migrating data workflows to PySpark, the goal is not just to replicate the calculation, but also to maintain this critical analytical structure--the ability to easily rank categories by frequency.

You can use the **value_counts()** function in pandas to count the occurrences of each unique value in a given column of a DataFrame.

The PySpark Equivalent: groupBy() and count()

In a distributed environment like Spark, operations are designed for parallel execution across a cluster. PySpark achieves the functionality of `value_counts()` by leveraging two primary transformation and action methods: **`groupBy()`** and **`count()`**. The `groupBy()` transformation partitions the PySpark DataFrame based on the unique values in the specified column. This crucial step prepares the data for aggregation, ensuring that all identical values are grouped together across the cluster nodes before counting can commence.

Following the grouping, the `count()` action is applied. This method operates on the resulting GroupedData object produced by groupBy() and calculates the number of rows within each group. The outcome is a new DataFrame containing two columns: the original grouping column and a new column, typically named 'count', representing the frequency of occurrence for that value. This two-step process--grouping followed by counting--is the canonical method for frequency analysis in Spark SQL and PySpark DataFrames, prioritizing scalability and efficiency.

You can use the following methods to replicate the **value_counts()** function in a PySpark DataFrame:

Method 1: Count Occurrences of Each Unique Value in Column

The most straightforward implementation involves chaining the `groupBy()` and `count()` methods directly. This approach calculates the frequency of each unique value in the target column. It is important to note that, unlike the default behavior of Pandas' `value_counts()`, the output of this direct chain in PySpark is typically sorted lexicographically by the grouping column itself, not by frequency count.

This base method is highly performant because it avoids a costly distributed sort operation, which is often a bottleneck in large-scale processing. It is ideal when you primarily need the raw counts and plan to use the results for subsequent filtering or joining processes where immediate visual ranking is not paramount. The efficiency of groupBy() in Spark ensures that this calculation scales robustly.

Method 1: Count Occurrences of Each Unique Value in Column

#count occurrences of each unique value in 'team' column
df.groupBy('team').count().show()

Method 2: Count Occurrences of Each Unique Value in Column and Sort Ascending

While Method 1 provides the raw counts, data analysis often requires results to be ordered based on frequency. To introduce sorting, we must append the `orderBy()` method to the chain. Method 2 specifically demonstrates how to sort the resulting frequency table in ascending order based on the 'count' column. This sorting reveals the least frequent values first, which can be crucial for identifying sparse categories or potential data anomalies that occur rarely in the dataset.

The `orderBy()` function in PySpark is highly optimized for distributed sorting, though it does incur performance costs proportional to the data size. By specifying the 'count' column (which is

automatically generated by the preceding count() action) without any additional parameters, the sort defaults to ascending order. This produces a clear view of the data distribution, moving from the lowest frequency to the highest frequency group. This perspective is vital when managing data quality and category sparsity.

Method 2: Count Occurrences of Each Unique Value in Column and Sort Ascending

#count occurrences of each unique value in 'team' column and sort ascending
df.groupBy('team').count().orderBy('count').show()

Method 3: Count Occurrences of Each Unique Value in Column and Sort Descending

The most common use case, mirroring the default behavior of Pandas' `value_counts()`, is sorting the frequency counts in descending order. This immediately highlights the most dominant categories within the column, providing immediate insight into the distribution skew and the most frequent occurrences. This is invaluable for feature engineering or understanding which categories require the most resource allocation or memory consumption.

To achieve descending order, the `orderBy()` method must explicitly receive the `ascending=False` parameter. This modification instructs PySpark to invert the sorting logic, placing the highest counts at the top of the resulting DataFrame. This method is the preferred way to replicate the full analytical utility of the original Pandas function in a distributed environment, ensuring that primary data trends are instantly visible for high-level summaries and reports.

Method 3: Count Occurrences of Each Unique Value in Column and Sort Descending

#count occurrences of each unique value in 'team' column and sort descending
df.groupBy('team').count().orderBy('count', ascending=False).show()

Setting Up the Environment and Sample DataFrame

To illustrate these three methods practically, we will utilize a small sample dataset representing basketball player statistics. This example requires initializing a **SparkSession**, defining the schema and data, and creating the PySpark DataFrame. This setup ensures that we have a concrete basis for demonstrating how the `groupBy()` and count() operations function across different categorical values, specifically focusing on the 'team' column for frequency analysis.

The following code snippet demonstrates the necessary imports and DataFrame construction. This initial step is critical for transitioning any data source into the structure required for distributed

computation in Apache Spark, which relies on the `SparkSession` entry point to manage cluster resources and execute distributed logic.

The following examples show how to use each method in practice with the following PySpark DataFrame that contains information about various basketball players:

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	11
A	Guard	30
B	Forward	22
B	Forward	22
B	Guard	14
B	Guard	10
C	Forward	13
D	Forward	7
D	Forward	16

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

Example 1: Basic Count Replication Walkthrough

Example 1 demonstrates the fundamental replication of the frequency counting functionality without any explicit sorting. We apply the **`groupBy('team')`** method, which aggregates all records belonging to the same team identifier. Following this, the **`count()`** method calculates the total number of records for each aggregated team group, resulting in a new DataFrame showing the unique teams and their frequencies.

The command below illustrates this sequence. Notice that the resulting output is sorted by the 'team' column alphabetically (A, B, C, D), rather than by the numerical frequency count. Team B has the highest count (4), but appears second in the list due to the alphabetical ordering default imposed by PySpark when no explicit `orderBy()` clause is specified. This is a crucial detail when interpreting output directly.

We can use the following syntax to count the number of occurrences of each unique value in the **team** column of the DataFrame:

#count occurrences of each unique value in 'team' column
df.groupBy('team').count().show()

+----+-----+
|team|count|
+----+-----+
A	2
B	4
C	1
D	2
+----+-----+

The output displays the count of each unique value in the **team** column.

By default, the rows are sorted in alphabetical order by the unique values in the **team** column.

Example 2: Ascending Frequency Sort Walkthrough

For data diagnostics, identifying the least frequent categories is often as crucial as finding the most frequent ones. Example 2 achieves this by appending `.orderBy('count')` to the aggregation chain. This command sorts the resultant frequency table based on the 'count' column in ascending

order, bringing the rare occurrences to the top of the output.

As shown in the output, Team C, with only 1 entry, appears first. Teams A and D, both having 2 entries, are tied. In this tie scenario, the tie-breaking rule uses the grouping key ('team') alphabetically, resulting in Team A appearing before Team D. This detailed sorting control is essential for thorough exploratory data analysis, especially for identifying and addressing data imbalance.

We can use the following syntax to count the number of occurrences of each unique value in the **team** column of the DataFrame and sort by count ascending:

#count occurrences of each unique value in 'team' column and sort ascending
df.groupBy('team').count().orderBy('count').show()

+----+-----+
C	1
A	2
D	2
B	4
+----+-----+

The output displays the count of each unique value in the **team** column, sorted by count in ascending order.

Example 3: Descending Frequency Sort Walkthrough

This final example provides the standard, analysis-ready output for determining data distribution popularity. By adding `.orderBy('count', ascending=False)`, we force the PySpark engine to arrange the results from the highest frequency count down to the lowest. This is the closest operational mirror to the default behavior of Pandas' `value_counts()`, ensuring maximum compatibility for analysts migrating existing workflows.

The descending sort confirms that Team B is the most represented group in the sample dataset (count: 4). This structure is preferred for immediate insight generation, enabling analysts to quickly identify majority stakeholders or primary data sources in large-scale data manipulation tasks. The combined use of `groupBy()`, `count()`, and `orderBy()` provides the full flexibility required for robust distributed frequency counting.

We can use the following syntax to count the number of occurrences of each unique value in the **team** column of the DataFrame and sort by count descending:

#count occurrences of each unique value in 'team' column and sort descending
df.groupBy('team').count().orderBy('count', ascending=False).show()

```
+----+-----+
| B| 4|
| A| 2|
| D| 2|
| C| 1|
+----+-----+
```

The output displays the count of each unique value in the **team** column, sorted by count in descending order.

Further Reading on PySpark Data Manipulation

Mastering frequency counting is just the start of efficient data manipulation in PySpark. For those transitioning from Pandas, exploring other fundamental equivalences--such as how to perform joins, apply custom functions (UDFs), or execute window functions--is highly recommended to fully leverage Spark's distributed capabilities.

Successfully bridging the gap between familiar single-node commands and distributed operations requires patience and a deep understanding of how Spark optimizes data shuffling and aggregation. The `groupBy().count()` pattern is a fundamental building block for many complex analytical processes.

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

Exploring data distributions using histograms in PySpark.
Implementing efficient data sampling strategies for large DataFrames.
Converting between RDDs, DataFrames, and Pandas DataFrames in Python.