Learning PySpark: Implementing Pandas value_counts() Functionality

Authored by

Mohammed looti

November 11, 2025

Bridging Pandas and PySpark for Frequency Analysis

When migrating data processing workflows from single-node environments to large-scale, distributed systems, analysts often seek direct equivalents for familiar functions. In the world of data manipulation using Pandas, the highly useful **value_counts()** function is indispensable. This function quickly calculates the frequency of each unique item within a specified column (or Series) of a DataFrame, providing immediate insights into data distribution and cardinality.

However, when transitioning data analysis workloads to distributed frameworks like PySpark, a direct, single-function equivalent for **value_counts()** does not exist. This is due to the fundamental architectural differences between Pandas, which operates in memory on a single machine, and PySpark, which uses lazy evaluation and distributed computation across a cluster. To achieve the same frequency analysis goal in PySpark, we must explicitly leverage the core aggregation capabilities of the Spark engine.

The core functionality of counting unique values is achieved by combining the `groupBy` and `count` operations. This combination forces a distributed shuffle of the data across the cluster partitions, grouping identical values together before performing the final aggregation step. Understanding this multi-step process is crucial for writing efficient and scalable frequency counting logic in PySpark, ensuring that the results accurately reflect the distribution of data across the entire distributed dataset.

The Core Concept: Replicating value_counts() in PySpark

To effectively replicate the behavior of the Pandas **value_counts()** function in a PySpark DataFrame, we utilize the specific transformation and action sequences provided by the Spark SQL API. Instead of relying on a single convenience function, we chain together methods that explicitly define the aggregation logic. The primary tools used for this task are **groupBy()**, which initiates the grouping process based on the unique values of a selected column, and **count()**, which executes the aggregation, returning the number of records within each group.

Once the counts are generated, the resulting DataFrame contains two columns: the original grouping column (e.g., 'team') and a new column named 'count'. This output mirrors the fundamental structure provided by the Pandas equivalent, but in a distributed format suitable for massive datasets. Furthermore, PySpark offers flexibility that extends beyond simple counting, allowing us to immediately apply sorting and ordering operations to the resulting frequency table, which is often a necessary step for data validation and top-N analysis.

The following three methodologies represent the most common and robust ways to calculate and present frequency counts in a PySpark DataFrame. These methods build upon the core **groupBy().count()** pattern, adding sophisticated sorting mechanisms to tailor the output

presentation. We will examine each approach, demonstrating how to achieve unsorted, ascending sorted, and descending sorted frequency tables.

Implementation Method 1: Simple Frequency Count (Unsorted)

The most straightforward way to calculate the frequency of unique values is by applying the **groupBy().count()** chain without any explicit sorting instructions. This method is highly efficient as it minimizes unnecessary operations, focusing solely on the aggregation required for counting. The resulting output DataFrame will contain the unique values from the grouped column and their corresponding frequencies.

It is important to note that while the output of the `show()` action may appear sorted (often alphabetically by the key column), this sorting is not guaranteed across different Spark environments or execution plans unless an `orderBy` transformation is explicitly applied. For most exploratory data analysis (EDA) where the exact order is not critical, this method provides the fastest path to the frequency data.

To count the occurrences of each unique value in a specific column, such as 'team', you would use the following concise syntax. This is the PySpark foundation for replicating **value_counts()**:

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

Implementation Method 2: Sorting Results in Ascending Order

In many analytical scenarios, seeing the least frequent items first can be as important as seeing the most frequent ones. Method 2 addresses this requirement by incorporating the **orderBy()** transformation immediately after the counting aggregation. By default, when no arguments are passed to **orderBy()** (or if the column name is specified without direction), PySpark sorts the resulting DataFrame in ascending order based on the specified column--in this case, the 'count' column.

Applying **orderBy()** on the 'count' column is particularly useful for identifying rare categories or outliers within the dataset. This transformation ensures that the final presented results are explicitly ordered, facilitating immediate interpretation of the data distribution, starting from the smallest counts and moving towards the largest.

To count occurrences of each unique value in the 'team' column and then sort the resultant frequencies from lowest to highest, the syntax is extended as follows:

#count occurrences of each unique value in 'team' column and sort ascending

df.groupBy('team').count().orderBy('count').show()

Implementation Method 3: Sorting Results in Descending Order

The most common use case for frequency counting is identifying the categories that appear most often. This directly mimics the default behavior of the Pandas **value_counts()** function, which sorts results in descending order. To achieve this in PySpark, we must explicitly set the `ascending` parameter within the **orderBy()** function to `False`.

The descending sort operation is essential for calculating mode or for performing analyses that require ranking categories by popularity or volume. By specifying `ascending=False`, we instruct Spark to order the aggregated rows such that the highest count values appear at the top of the output table. This method provides the clearest view of the dominant categories within the dataset.

To count occurrences of each unique value in the 'team' column and sort the frequencies from highest to lowest, ensuring the most frequent items are immediately visible, use the following syntax:

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

Practical Demonstration: Setting Up the PySpark DataFrame

To illustrate these three methodologies in action, we first need to define and initialize a sample DataFrame. This DataFrame will simulate real-world data, containing information about various basketball players, including their team designation, position, and points scored. This setup ensures that we have categorical columns ('team' and 'position') suitable for frequency analysis.

We begin by initializing the **SparkSession**, which serves as the entry point for all Spark functionality. We then define a list of lists representing our raw data and a corresponding list of column names. Finally, the `spark.createDataFrame()` method converts this local data into a distributed DataFrame (`df`), making it ready for PySpark transformations and actions.

The code block below outlines the steps for creating and displaying the sample **PySpark DataFrame** used in the subsequent examples. Notice the structure of the data, specifically the varying counts for 'team' (e.g., Team B appears 4 times, Team C appears 1 time), which will be the basis of our frequency analysis.

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: Demonstrating Simple Frequency Count

Applying the first method, we execute the fundamental aggregation chain: `df.groupBy('team').count().show()`. This operation calculates how many rows exist for each unique value present in the 'team' column (A, B, C, D). As discussed, this method provides the raw frequency data without guaranteeing a specific order, although the results often appear sorted by

the key column itself (Team A, then B, C, D).

The resultant table clearly shows the distribution of players across the teams. Team B has the highest count (4), while Team C has the lowest (1). This immediate feedback is highly valuable for initial data quality checks and understanding dataset skewness. Note that the output schema is automatically defined by the aggregation, resulting in the original grouping column ('team') and the new aggregated column ('count').

We 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
+----+-----+

Example 2: Demonstrating Ascending Order Frequency Count

To demonstrate Method 2, we introduce the **orderBy('count')** transformation to sort the results based on the frequency column in ascending order. This rearrangement highlights the teams with the fewest players, immediately placing Team C (count 1) at the top of the output.

This approach is crucial when analysts are searching for rare events or categories that might require special handling or investigation due to their low representation in the dataset. By default, the `orderBy` function assumes ascending sorting, simplifying the syntax required to achieve this specific ordering.

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

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

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

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

Example 3: Demonstrating Descending Order Frequency Count

Finally, we apply Method 3, which is the most direct analogue to the default Pandas **value_counts()** output. By including `ascending=False` within the **orderBy()** function, we force the output to display the highest counts first, effectively ranking the teams by player volume.

This descending sort is the standard requirement for identifying the modal category or the most frequent items, which is often the starting point for feature engineering or understanding data dominance. Team B (count 4) is correctly positioned at the top of the list, providing the most impactful statistical summary first.

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

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

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

The output clearly displays the count of each unique value in the **team** column, sorted by count in descending order, successfully replicating the primary functionality of Pandas' **value_counts()** in a distributed PySpark environment.

Additional Resources

Mastering frequency analysis is just one step in utilizing the full power of distributed computing. PySpark offers extensive capabilities for advanced aggregation, data transformation, and machine learning at scale. While `groupBy().count()` handles basic frequency counting efficiently, more complex aggregation needs might require alternative functions like `cube()` or `rollup()` for generating subtotals and hierarchical summaries.

These foundational tutorials explain how to perform other common data wrangling and aggregation tasks in PySpark, allowing you to further expand your toolkit for handling large-scale datasets.

Exploring the official Apache Spark documentation is recommended for detailed information on optimization techniques, dealing with skewed data during shuffling, and other advanced configurations related to aggregation operations.