Learning PySpark: Counting Values by Group in DataFrames with Examples

Authored by

Mohammed looti

November 11, 2025

Introduction to Grouped Counting in PySpark

In the realm of large-scale data processing, the ability to summarize and aggregate information based on categorical variables is indispensable. PySpark, the Python API for Apache Spark, offers highly efficient, distributed methods for performing these crucial aggregation tasks. These operations mirror the familiar functionality of the standard SQL `GROUP BY` clause, allowing users to efficiently count records within specific groups. This capability is foundational for various analytical tasks, including frequency distribution analysis, cardinality estimation, and the generation of essential key performance indicators (KPIs).

We primarily execute these grouping operations using the DataFrame structure. The DataFrame serves as the primary distributed collection of data, organized into named columns, making it ideal for structured queries. Counting values by group within a DataFrame involves chaining two fundamental methods: the `groupBy()` transformation, which logically segments the data based on one or more specified columns, and the `count()` action, which calculates the total number of records within each resulting group segment across the distributed cluster.

This comprehensive guide will demonstrate the two primary syntax patterns used to achieve grouped counts in PySpark. We will cover grouping by a single column for straightforward frequency analysis and grouping by multiple columns to conduct nuanced, multi-dimensional data segmentation. Mastering these techniques is absolutely essential for any data professional working with distributed computing environments and large datasets utilizing the Apache Spark ecosystem.

Core Mechanism: Understanding Transformations and Actions

The mechanism for counting by group is deeply rooted in the transformation and action paradigm inherent to the Apache Spark framework. The `groupBy()` function is categorized as a transformation; it does not trigger immediate computation. Instead, it returns a GroupedData object, which prepares the data structure for subsequent aggregation logic. Crucially, the actual aggregation process--which involves the necessary data shuffling and calculation across all worker nodes in the cluster--is only initiated when an action, such as `count()`, is explicitly called upon the grouped object. This deferred execution is what gives Spark its efficiency and scalability.

We categorize the application of this aggregation into two common patterns, determined by the complexity required for the desired data segmentation:

Method 1: Counting Values Grouped by a Single Column

This highly common method is employed whenever a simple frequency distribution is required based on the unique entries found within just one column. The output of this operation is a new Data Frame containing the original grouping column and a new column, typically named `count`,

that holds the total number of records found for each unique group key. This is the simplest way to get a quick overview of data distribution.

df.groupBy('col1').count().show()

Method 2: Counting Values Grouped by Multiple Columns (Composite Keys)

When the analysis requires a more detailed, segmented breakdown--for instance, counting occurrences based on the intersection of two or more attributes--the composite key approach is necessary. To achieve this, you simply pass multiple column names, separated by commas, to the `groupBy()` function. This instructs PySpark to form unique groups based on the combined values across all specified columns, yielding a much finer-grained summary of the data distribution.

df.groupBy('col1', 'col2').count().show()

Prerequisites: Setting up the PySpark Environment and Sample Data

To begin executing any PySpark transformations or actions, the environment must first be initialized. A SparkSession is the unified entry point into all Spark functionality, acting as the necessary gateway for creating, reading, and manipulating DataFrames. For the purposes of this demonstration, we will construct a small, reproducible Data Frame that contains simulated data about basketball players, specifically tracking their team affiliation, playing position, and points scored.

The setup code provided below first initializes the required SparkSession instance. It then defines our sample dataset, which represents a common real-world scenario where aggregations are crucial for understanding underlying team structures and performance distributions. We will use this established DataFrame, named `df`, throughout the following examples to clearly illustrate how grouped counting functions in practice.

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| 8|
| A| Forward| 22|
| A| Forward| 22|
| B| Guard| 14|
| B| Guard| 14|
| B| Guard| 13|
| B| Forward| 7|
| C| Guard| 8|
| C| Forward| 5|
+----+--------+------+
```

Example 1: Counting Values Grouped by a Single Column

Our initial objective is to quickly and accurately determine the total number of players rostered under each distinct team (A, B, and C). To achieve this, we must group the entire Data Frame exclusively based on the unique values present in the **team** column. This simple, single-column aggregation is paramount for gaining an immediate understanding of the size and distribution of our data across this main categorical variable.

The implementation is concise and clear: we invoke the `groupBy()` method, explicitly specifying `'team'` as the grouping key. This transformation is immediately followed by the `count()` action, which triggers the distributed computation. The resulting output, detailed below, provides a highly

succinct summary of the player roster size for every team defined in our initial Data Frame, demonstrating the efficiency of PySpark's aggregation functions.

#count number of values by team
df.groupBy('team').count().show()

```
+----+-----+
|team|count|
+----+-----+
| A| 4|
| B| 4|
| C| 2|
+----+-----+
```

The resulting Data Frame clearly isolates the size of each unique group defined by the **team** column. Based on this highly summarized output, we can derive the following precise conclusions regarding the structure and composition of the team rosters:

Team **A** contains a total of **4** players.

Team **B** contains a total of **4** players.

Team **C** contains a total of **2** players.

Example 2: Counting Values Grouped by Multiple Columns

Although single-column aggregation offers an excellent high-level overview, real-world data analysis frequently necessitates deeper, more complex segmentation. If, for instance, our goal is to understand the precise distribution of player positions **within** each team, we must group the data using a composite key. This key combines both the **team** and the **position** columns, providing a significantly more granular view of the dataset than was possible with the previous example.

By passing both `'team'` and `'position'` arguments to the `groupBy()` method, PySpark is instructed to identify every unique combination of team and position found in the data. Following the grouping, the `count()` action is applied to each of these unique combinations. This sophisticated technique is highly effective for generating contingency tables or pivot-like summaries directly within the distributed computing environment, ensuring accurate and efficient categorization of records.

#count number of values by team and position
df.groupBy('team', 'position').count().show()

```
+----+--------+-----+
|team|position|count|
+----+--------+-----+
| A| Guard| 2|
| A| Forward| 2|
| B| Guard| 3|
| B| Forward| 1|
| C| Forward| 1|
| C| Guard| 1|
+----+--------+-----+
```

This multi-dimensional result clearly reveals the specific breakdown of player positions for every team, offering critical insights into team composition that a simple, single-column count could never provide. We can interpret the results below, illustrating the detailed roster breakdown based on the composite key:

Team **A** has **2** players categorized as **Guard**.

Team **A** has **2** players categorized as **Forward**.

Team **B** has **3** players categorized as **Guard**.

Team **B** has **1** player categorized as **Forward**.

Team **C** has **1** player categorized as **Forward**.

Team **C** has **1** player categorized as **Guard**.

Advanced Considerations: Alternatives and Performance

While the straightforward chaining of `groupBy()` followed by `count()` is the most intuitive method for frequency calculation, Apache Spark provides highly flexible alternatives, particularly when complex or multiple aggregations are required simultaneously. The powerful `agg()` function, for instance, enables users to perform several aggregations--such as counting, summing, or averaging--immediately after the grouping transformation. If the requirement is to count only the non-null values within a specific column, the syntax `df.groupBy('col1').agg({'col2': 'count'})` might be used, which is more specific than the general `count()` action that tallies all rows in the group regardless of column nullity.

A critical aspect of performance in Spark is the process known as data shuffling. When a DataFrame executes the `groupBy()` transformation, the infrastructure must ensure that all data

records sharing the same grouping key are physically moved to the same worker node across the cluster before the aggregation can be computed. This required network transfer is computationally demanding and can easily become a major performance bottleneck, especially when dealing with grouping columns that have extremely high cardinality or when the data distribution is severely imbalanced (known as data skew).

To effectively mitigate these scaling challenges, data engineers should prioritize selecting the most efficient grouping columns and, when necessary, carefully adjust Spark's internal configurations. Specifically, modifying the `spark.sql.shuffle.partitions` setting can optimize the parallel processing of grouped data. A deep understanding of the underlying physical query plan generated by PySpark commands is paramount for effectively debugging and tuning distributed group-by operations at massive scale.

Conclusion and Further Resources

Counting values by group is an absolutely fundamental data manipulation technique within PySpark, providing the capability for rapid, distributed frequency analysis. By effectively chaining the `groupBy()` transformation with the `count()` action, users are empowered to quickly summarize vast datasets based on single or multiple categorical dimensions. This essential functionality is key for preparing data for analytical models, generating mandatory business reports, and performing effective exploratory data analysis (EDA) on high-volume distributed platforms.

For professionals seeking to expand their mastery of PySpark and SparkSession operations beyond basic counting, we highly recommend consulting the official Apache Spark documentation. Further exploration of tutorials covering more complex aggregation scenarios will solidify your skills in distributed data processing.

To continue your development in data aggregation techniques, consider exploring the following related tutorials:

Calculating Averages: Detailed instructions on using the `avg()` function immediately after grouping to find mean values for numerical columns.

Applying Multiple Aggregations: A guide to utilizing the flexible `agg()` function for simultaneous calculations (including count, sum, minimum, and maximum) across groups.

Handling Nulls: Essential techniques for dealing with missing values during the grouping process to guarantee accurate and reliable counts.