How to Count Rows by Group in PySpark DataFrames

Authored by

stats writer

February 8, 2026

Mastering Grouped Counting in PySpark (Comprehensive Guide and Examples)

The ability to perform a group-level count is fundamental in modern data analysis, especially when handling large-scale datasets characteristic of Apache Spark environments. In PySpark, the combination of the `groupBy()` and `count()` transformations allows data engineers and analysts to efficiently summarize data by aggregating rows based on shared values in specified columns. This operation is essential for various analytical tasks, such as calculating frequency distributions, identifying cardinality within categories, or determining the representation of distinct groups within a massive dataset.

By leveraging this powerful functionality, users can shift from looking at raw transactional data to generating meaningful, summarized insights. For instance, this approach enables immediate identification of patterns--whether it's counting the volume of customer transactions per region, assessing the prevalence of different product categories, or analyzing resource allocation across various project teams. Ultimately, the PySpark Count by Group function provides a robust, scalable, and convenient mechanism for executing complex data aggregation tasks, crucial for efficient data processing and large-scale data intelligence.

Prerequisites: Understanding PySpark DataFrames and Grouping Logic

Before diving into the implementation, it is vital to grasp the concept of the PySpark DataFrame. A PySpark DataFrame is a distributed collection of data organized into named columns, conceptually equivalent to a table in a relational database or a DataFrame in R/Python's Pandas library. The distributed nature of the DataFrame is what gives Spark its immense power for processing big data. When we apply the `groupBy()` operation, we are instructing the Spark engine to shuffle the data across the cluster so that all rows sharing the same value (or combination of values) in the specified grouping columns are brought together onto the same partition. This preparatory step is critical for ensuring that subsequent aggregation functions, like `count()`, operate accurately across the entire group.

The grouping logic follows standard relational algebra principles, mirroring the `GROUP BY` clause found in SQL. The result of the `groupBy()` transformation is not yet a standard DataFrame but a `GroupedData` object, which awaits an aggregation function. By appending `.count()`, we execute the aggregation, which calculates the total number of rows associated with each unique group identified by the grouping keys. Understanding this separation--the grouping step followed by the aggregation step--is key to harnessing the efficiency of PySpark for complex analytical workflows. This process ensures that computationally intensive counting operations are performed in parallel across the distributed cluster infrastructure.

Core Methodology: Syntax for Grouped Counting

PySpark offers highly readable and concise syntax for performing grouped counts, typically involving chaining the `groupBy()` transformation directly with the `count()` action. Depending on the complexity of the desired aggregation, the method requires specifying either a single column or a list of multiple columns that define the distinct groups. Both approaches return a resulting DataFrame that contains the original grouping column(s) and a new column, usually named `count`, which holds the row tally for that specific combination of values.

The primary benefit of using the combined `groupBy().count()` approach is its simplicity and optimized execution plan generation by the Catalyst Optimizer within Spark. When Spark recognizes this pattern, it can efficiently handle the shuffling and aggregation phases, minimizing unnecessary data movement. While alternative methods exist (such as using `agg(F.count('*'))`), the dedicated `count()` method after grouping is the most straightforward way to achieve a simple row enumeration per group, making it the preferred method for quick descriptive statistics and exploratory data analysis.

You can use the following methods to count values by group in a PySpark DataFrame:

Method 1: Counting Values Grouped by a Single Column

This method is used when the analysis requires summarizing the dataset based on the distinct values present in just one attribute. This is perhaps the most frequent use case for grouped counting, ideal for generating histograms or frequency tables. We provide the column name as a string argument to the `groupBy()` function, followed immediately by the `count()` action to trigger the computation. The output will immediately show the distribution of records across the unique categories found in that single column.

Method 1: Count Values Grouped by One Column

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

Method 2: Counting Values Grouped by Multiple Columns

For more granular analysis, it is often necessary to count the records based on the unique combinations of values found across two or more columns. This multi-column grouping is crucial for segmenting data, such as counting unique events per user per day, or in our upcoming example, assessing player distribution across specific teams and positions simultaneously. When passing multiple columns to `groupBy()`, these columns must be provided as separate string arguments within the function call.

The Spark engine treats the combination of values across all specified columns as a single key. For example, if grouping by ('Region', 'Product'), a record with ('East', 'Laptop') will be grouped separately from ('East', 'Monitor'). This powerful segmentation capability allows for deep, multi-dimensional analysis that goes beyond simple univariate statistics, providing comprehensive visibility into the dataset's structure.

Method 2: Count Values Grouped by Multiple Columns

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

Implementation Step 3: Setting Up the PySpark Environment and Data

To demonstrate these methods practically, we must first establish a working PySpark environment and define a sample dataset. We initiate a `SparkSession`, which serves as the entry point to communicate with the core Spark functionality. Our example uses a structured dataset relating to basketball players, capturing their team affiliation, playing position, and points scored. This dataset provides clear categorical variables (`team`, `position`) perfect for demonstrating grouped counting operations.

The construction of the DataFrame involves defining the raw data as a list of rows and specifying the corresponding column names. This ensures that the schema is correctly established before any transformations are applied. Observing the initial DataFrame structure is essential for verifying the input data integrity before proceeding with the complex aggregation logic. The following code block details the initialization process and displays the resulting input DataFrame.

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| 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 Player Totals Grouped by Team (Single Column Aggregation)

In this first demonstration, our goal is to determine the total number of players associated with each distinct team (A, B, and C). We apply the `groupBy()` function exclusively to the `team` column. This instructs PySpark to collect all records sharing the same team identifier and then calculate the size of that collective group using `count()`. This provides a direct summary of team sizes within our roster data.

This technique is indispensable for quickly assessing distributional balance or identifying outliers in group sizes. The resulting DataFrame clearly shows the team identifier alongside the calculated count, making the summary easily interpretable. Note that the resulting column holding the aggregated value is automatically named 'count' unless explicitly aliased using a more complex aggregation method.

We use the following syntax to count the number of rows in the DataFrame grouped solely by the values in the **team** column:

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

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

From the output, which provides a clean frequency table, we can definitively observe the composition of the roster:

There are **4** players associated with team **A**.
There are **4** players associated with team **B**, indicating parity in size with team A.
There are **2** players rostered for team **C**, demonstrating that team C is the smallest in this dataset.

Example 2: Counting Player Distribution Grouped by Team and Position (Multi-Column Aggregation)

To gain a deeper understanding of the team compositions, we now perform a multi-dimensional count by grouping simultaneously on both the `team` and `position` columns. This complex grouping provides visibility into how players are distributed across different roles within each team, a much more granular view than the previous example. The keys in the resulting aggregated DataFrame now consist of unique combinations, such as ('A', 'Guard') and ('A', 'Forward'), rather than just the team name alone.

This type of segmentation is critical for reporting and resource management, allowing analysts to immediately see, for example, which teams are heavily weighted towards certain player positions. As datasets scale, the efficiency of PySpark in handling the necessary data shuffling for this two-level aggregation becomes paramount, ensuring results are returned quickly even from petabytes of source data. This example showcases the true power of granular data aggregation within the Spark framework.

We use the following syntax to count the number of rows in the DataFrame grouped by the values in the **team** and **position** columns:

#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|
+----+--------+-----+
```

Analyzing this detailed output allows us to derive precise structural insights:

There are **2** players on team **A** designated with a position of **Guard**.
There are **2** players on team **A** designated with a position of **Forward**, indicating an even split in positional roles for Team A.
There are **3** players on team **B** who are **Guard** positions, suggesting Team B favors this role slightly more than others.
Crucially, we also see that Team C has an equal split of one **Forward** and one **Guard** player.

Advanced Considerations and Performance Optimization

While the `groupBy().count()` method is robust, users dealing with extremely large DataFrames should be mindful of performance implications. Grouped operations in Spark inherently involve a data shuffle, which moves data across the network to collocate records that belong to the same group. This shuffle is often the most resource-intensive step in any Spark job.

To mitigate performance bottlenecks, especially when dealing with high-cardinality grouping keys, consider the following optimization strategies. First, ensure that the columns used for grouping are of an appropriate data type; using complex or overly long string keys can increase shuffle overhead. Second, consider pre-filtering the DataFrame to reduce the volume of data before the aggregation occurs. Finally, for counts that also require additional aggregations (e.g., counting and summing points simultaneously), it might be more efficient to use the `agg()` function explicitly, allowing Spark to execute multiple aggregations in a single pass over the grouped data.

Understanding the interplay between grouping keys, partitioning, and the resulting shuffle is key to writing high-performance PySpark code. The simple `count()` is efficient for basic frequency

analysis, but complex pipelines may benefit from deeper understanding of the Spark execution plan to ensure optimal resource utilization.

Conclusion: The Efficiency of PySpark Aggregation

The `groupBy()` followed by `count()` functions in PySpark provides an exceptionally efficient and clear method for performing group-level row counts, whether focusing on a single column or creating complex, multi-dimensional segmentations. This functionality is foundational for nearly every analytical task performed on distributed data, translating raw transactional entries into actionable summarized statistics.

By mastering these fundamental aggregation techniques, data professionals can unlock the full potential of Apache Spark for large-scale data processing. The examples provided--from defining the initial data structures to executing single and multi-column counts--demonstrate the straightforward yet powerful syntax that makes PySpark the tool of choice for big data analysis. The tutorials below offer pathways to explore further common tasks and advanced data manipulations within the PySpark ecosystem.

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