Count by Group in PySpark (With Examples)

Authored by

stats writer

November 16, 2025

Introduction to Grouped Counting in PySpark

When performing large-scale data analysis, one of the most fundamental operations is aggregation, which often involves counting records based on specific criteria. In the PySpark environment, managing and summarizing data stored within a distributed DataFrame is achieved efficiently using a combination of the `groupBy()` and `count()` functions. These methods are essential for generating summary statistics, identifying category distributions, and gaining initial insights into the structure of your dataset. Mastering these techniques is crucial for any data engineer or analyst working with big data pipelines, as it allows for the transformation of raw records into meaningful, actionable statistics. This guide will walk you through the precise syntax and practical implementation of these methods, demonstrating how to effectively count values when grouping by either a single column or multiple columns simultaneously in a PySpark DataFrame.

The core functionality relies on the distributed nature of PySpark. When you invoke the `groupBy()` operation, Spark internally shuffles the data across the cluster nodes so that all rows sharing the same key (the column or columns specified in the grouping) are brought together. Subsequently, applying the `count()` aggregation function computes the number of records within each of these newly formed groups. This two-step process--grouping followed by aggregation--is the standard paradigm for calculating group totals. We will explore two primary scenarios: grouping by a single categorical variable and grouping by a composite key formed by multiple variables.

Method 1: Counting Values Grouped by One Column

This approach is used when you wish to tally the frequency of occurrences for unique values found within a single specified column. It provides a simple distribution count, often used for exploratory data analysis (EDA) to understand the cardinality and balance of categorical features. The syntax is concise and highly readable, making it the most common form of simple aggregation within a DataFrame.

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

Here, `col1` represents the singular column used as the grouping key. The resultant DataFrame will contain two columns: the grouping column (`col1`) and the aggregate count column (typically named `count`).

Method 2: Counting Values Grouped by Multiple Columns

When your analysis requires counting occurrences based on the unique combinations of values across several columns, you employ multi-column grouping. This is particularly useful for

segmenting data where the context of one column depends on another--for instance, counting how many players belong to a specific position *within* a particular team. The process remains syntactically similar to the single-column method, but the `groupBy()` function accepts multiple column names as arguments, effectively creating a composite key for aggregation.

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

The resulting groups are defined by the unique combinations of entries found across both `col1` and `col2`. While functionally straightforward, it is important to remember that grouping by a large number of columns can significantly increase the shuffling overhead in a distributed environment, potentially impacting performance.

Setting Up the Illustrative PySpark DataFrame

To properly illustrate these counting techniques, we will initialize a PySpark DataFrame containing sample data related to basketball players. This dataset includes three key fields: `team` (categorical identifier), `position` (player role), and `points` (a numerical score). This realistic example allows us to demonstrate how counts can reveal the composition and structure of teams within the dataset. Before executing any transformations, establishing a `SparkSession` is the prerequisite step, serving as the entry point to all PySpark functionality.

The following initialization code defines the data structure and converts the local Python data into a distributed PySpark DataFrame. Notice the use of `SparkSession.builder.getOrCreate()`, which ensures that an existing session is reused or a new one is created if necessary, managing the cluster resources efficiently. The resulting DataFrame, `df`, is then displayed using `df.show()` to verify its structure and content before proceeding to the aggregation exercises.

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

This initial DataFrame, `df`, now serves as our working example. We observe that it contains 10 records distributed across three teams (A, B, and C) and two positions (Guard and Forward). Our goal in the subsequent examples is to summarize these 10 records based on the categorical columns to reveal the headcount for each grouping.

Example 1: Counting Players per Team (Single Grouping)

In this inaugural example, our objective is to determine the total number of players associated with each unique team identifier (A, B, or C). This is a classical frequency counting problem, optimally solved using the single-column grouping technique. We apply the `groupBy('team')` method to partition the data, followed immediately by the `count()` aggregation. The resulting DataFrame will clearly show the total number of records that fall under each specific team designation, providing a quick summary of team sizes within our dataset.

The simplicity of the syntax masks the underlying complexity of the distributed computation that

Spark performs. When the `groupBy()` is executed, data rows are shuffled across the cluster so that all members of Team A reside together, all members of Team B reside together, and so on. The subsequent `count()` function then operates locally on these grouped partitions before the final result is collected and displayed using `show()`. This process ensures scalability even with datasets far larger than our small example.

We utilize the following syntax to count the number of rows in the PySpark structure, grouping exclusively by the values found in the **team** column:

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

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

Analyzing the output provides immediate insight into the distribution of players across the teams. Team A and Team B are relatively balanced in size, while Team C has the smallest representation in this sample dataset. Specifically, we can deduce the following from the aggregation result:

There are **4** players distinctly associated with team **A**, indicating a robust representation.
There are **4** players assigned to team **B**, matching the size of Team A.
There are only **2** players recorded for team **C**, suggesting potential data scarcity or a smaller squad size for this team.

Example 2: Counting Players by Team and Position (Multi-Grouping)

While counting by a single column is useful for overall totals, a more nuanced analysis often requires understanding distributions across multiple categorical variables simultaneously. This is where multi-column grouping excels. For this example, we aim to determine the exact composition of each team, specifically calculating how many players on each team fill the **Guard** role and how many fill the **Forward** role. This requires grouping the data based on the unique combination of both the `team` and `position` columns.

By passing both column names into the `groupBy()` function--i.e., `groupBy('team', 'position')`--we instruct PySpark to treat the pair of values as a single composite key. For instance, ('A', 'Guard') is treated as distinct from ('A', 'Forward') and distinct from ('B', 'Guard'). This

level of detail provides granular insight into team structure that a simple single-column count cannot offer, allowing for comparisons of positional strength across different teams.

We utilize the following syntax to calculate the row counts based on the combined unique 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|
+----+--------+-----+
```

The resulting table provides a clear breakdown of positional representation across all teams. For instance, we can immediately see that Team B is heavily skewed towards Guards (3 players), whereas Team A exhibits a perfectly balanced composition (2 Guards, 2 Forwards). This detailed output facilitates complex statistical comparisons and resource planning based on specific data segments.

Team **A** maintains an equal distribution: **2** players fulfilling the **Guard** position and **2** players fulfilling the **Forward** position.
Team **B** prioritizes Guards, featuring **3** players in the **Guard** role and only **1** player categorized as **Forward**.
Team **C**, despite having only two total players, maintains a balance of **1** Guard and **1** Forward.

Alternative Aggregations and Performance Considerations

While `count()` is the simplest aggregation available after grouping, the `groupBy()` method is not limited solely to counting. It is a precursor to a wide array of powerful aggregate functions, such as `sum()`, `avg()`, `min()`, `max()`, and `agg()`. If, for instance, you wanted to calculate the average points scored by each position within each team, you would replace the `count()` call with `agg({'points': 'avg'})`. Understanding this flexibility is key to performing comprehensive analytical tasks using PySpark SQL structures. The choice of aggregation function determines the

specific metric calculated on the partitioned groups.

It is also crucial to consider performance, especially when dealing with massive datasets. The `groupBy()` operation is considered a 'wide' transformation because it necessitates shuffling data across the network. Excessive shuffling, typically caused by grouping on high-cardinality columns or too many columns simultaneously, can become a significant bottleneck. Data partitioning strategies, pre-filtering large DataFrames, and ensuring that the keys used for grouping are evenly distributed are best practices for mitigating performance degradation in production environments.

Furthermore, Spark offers alternative ways to achieve counts, such as using `df.cube()` or `df.rollup()`, which generate hierarchical summaries and grand totals in addition to the standard grouped counts. For simple frequency counts, the `groupBy().count()` method remains the most direct and idiomatic PySpark approach, but for complex, multi-dimensional analysis, exploring these advanced grouping functions can save significant development time and simplify the resulting code base.

Summary and Final Thoughts on PySpark Grouping

The ability to accurately and efficiently count records based on specific groupings is fundamental to data processing in SparkSession environments. Through the provided examples, we have clearly demonstrated how to implement both single-column and multi-column grouping using the powerful `groupBy().count()` combination. Whether you are conducting initial exploratory analysis or preparing aggregated metrics for downstream machine learning models, these methods offer the necessary functionality to transform raw, individual records into structured, summarized data.

The key takeaway is the flexibility offered by the `groupBy()` operator. It provides a highly declarative way to express complex partitioning logic, allowing the Spark engine to handle the challenging, distributed aspects of data shuffling and aggregation behind the scenes. Developers should always aim for clarity and efficiency when selecting their grouping keys, prioritizing the smallest set of columns necessary to achieve the desired analytical granularity, thereby optimizing cluster resource utilization.

By integrating these grouping techniques into your workflow, you enhance your capacity to derive meaningful statistics from large-scale data, making the transition from raw data ingestion to insightful reporting seamless and scalable. Continue experimenting with different aggregation functions following the `groupBy()` operation to unlock the full analytical potential of PySpark.