How to Count Distinct Values in PySpark Using groupBy

Authored by

stats writer

February 6, 2026

When dealing with vast amounts of structured data, analyzing summary statistics across defined subgroups is a fundamental requirement in data engineering and data analysis. The PySpark framework, built on the robust Apache Spark engine, provides incredibly efficient tools for handling this task, particularly through its **DataFrame** API. One of the most frequently employed operations is **grouping** data based on categorical variables and then performing an aggregation function on the resulting groups. This process allows data professionals to transform raw, granular records into actionable insights, such as calculating totals, averages, or, as we will explore here, counting the number of unique occurrences within each segment.

The core mechanism enabling this efficient subgroup analysis is the groupBy transformation. This operation is indispensable when you need to answer questions like: "How many unique products did each region sell?" or "What is the average salary per department?" While grouping itself establishes the boundaries for analysis, it requires a subsequent aggregation function to finalize the output. Understanding how to seamlessly combine the power of **groupBy** with specific statistical functions is key to leveraging the full potential of distributed computing offered by PySpark.

Our specific focus here is on counting distinct values within groups. This requirement often arises during exploratory data analysis (EDA) when attempting to understand the diversity or cardinality of a specific feature relative to another. For instance, if you have a dataset detailing user activity, grouping by user ID and counting the distinct pages visited provides a measure of engagement diversity. This article will provide a comprehensive guide on integrating the countDistinct function with the groupBy method, offering practical examples that illustrate its implementation and interpretation.

PySpark: Use groupBy with Count Distinct

Understanding the groupBy Transformation in PySpark

The groupBy function is a crucial component of the PySpark DataFrame API, serving as a transformation that logically partitions the dataset based on the values in one or more specified columns. It does not immediately return a resultant DataFrame but instead returns a `GroupedData` object. This object is essentially a promise--it holds the grouping structure and awaits an aggregate operation to be applied across those defined groups before generating the final aggregated output.

When you invoke **groupBy**, you are instructing Apache Spark to shuffle the data across the cluster such that all rows sharing the same key (the grouping column values) reside together. While this shuffling operation is computationally intensive, Spark's optimized engine minimizes its impact. The primary benefit is that once the data is co-located, the subsequent aggregation--whether summing, averaging, or counting distinct elements--can be performed efficiently in parallel on the

partitioned data, dramatically speeding up complex analytical queries involving large-scale datasets. This separation of the grouping definition from the aggregation execution provides flexibility and efficiency in the data pipeline.

For effective analysis, selecting the correct grouping column is paramount. This column must typically be a categorical feature or an identifier. Once the grouping is established, all subsequent aggregation functions operate within the scope of that group. If you group by the 'City' column, any count or average calculated afterwards will be unique to each distinct city present in the dataset. This foundational understanding is necessary before incorporating specific aggregation functions like **countDistinct**, which further refine the type of summary statistic being calculated.

The Role of countDistinct in Data Aggregation

While standard counting functions (like `count()`) simply tally the number of rows within each group, the countDistinct function is specifically designed to calculate the cardinality, or the number of unique non-null values, within a column for each defined group. This function is extremely valuable when analyzing feature diversity. For example, if you are tracking transactions, grouping by 'Customer_ID' and using **countDistinct** on 'Product_SKU' reveals how many different types of products each customer purchased, regardless of how many total transactions they made.

It is important to import the **countDistinct** function from `pyspark.sql.functions` before use, as it is not a native method of the `GroupedData` object but rather a high-level function designed for SQL-like operations within PySpark. When applied after the groupBy operation, it requires one or more column names as arguments--these are the columns whose distinct values will be counted across the established groups. This distinction between a standard count and a distinct count is critical for accurate aggregation and interpretation of dataset characteristics.

Unlike some other complex approximate counting methods available in Spark, **countDistinct** provides an exact count of unique values. While calculating exact distinct counts can be resource-intensive in a distributed environment (often requiring more shuffling than approximate counts), it is necessary for business logic that demands precision. By integrating **countDistinct** within the `.agg()` structure following the **groupBy** operation, we achieve a powerful, concise, and accurate way to derive group-level cardinality metrics, enabling robust exploratory data analysis.

Core Syntax: Combining groupBy and countDistinct

To count the number of distinct values in one column of a PySpark DataFrame, grouped by another column, you must follow a standard pattern involving three key steps: importing the necessary function, defining the grouping, and applying the aggregation using the `.agg()` method. The `.agg()` method takes the result of the **groupBy** operation and applies the defined aggregation functions, transforming the `GroupedData` object back into a final, summarized DataFrame.

The standard syntax is remarkably clean and mirrors SQL paradigms, which aids readability. First, you import `countDistinct`. Second, you call `.groupBy()` on your DataFrame (`df`) specifying the column(s) you wish to group by. Finally, you chain the `.agg()` method, passing the `countDistinct()` function, referencing the column you want to count uniquely. The following structure illustrates this fundamental operation:

from pyspark.sql.functions import countDistinct

df.groupBy('team').agg(countDistinct('points')).show()

This particular example calculates the number of distinct values found in the **points** column, with the results segmented and summarized based on the unique values present in the **team** column. The output DataFrame will contain two columns: the grouping column ('team') and the calculated aggregate count (which PySpark typically names automatically, e.g., `count(points)`). Mastering this structure allows for quick, powerful analysis across any pair of categorical and numerical features within your dataset.

Setting Up the Example DataFrame

To demonstrate the practical application of combining **groupBy** and countDistinct, we will create a sample DataFrame containing fictional basketball player data. This dataset includes three fields: the 'team' the player belongs to (the grouping key), their 'position' (an additional descriptive column), and the 'points' they scored (the column whose distinct values we wish to count per team). The creation process starts by initializing a SparkSession, which is the entry point for using Spark functionality.

The following detailed code block defines the necessary imports, initializes the Spark environment, prepares the raw data as a list of lists, defines the column schema, and finally creates and displays the resulting PySpark DataFrame. Notice the inclusion of repeated 'points' values within the same 'team' (e.g., Team A having two entries for 22 points, and Team B having two entries for 14 points). This redundancy is deliberate, as it will highlight how **countDistinct** filters out these duplicates when grouped by team.

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| Forward| 13|
| B| Forward| 14|
| C| Forward| 23|
| C| Guard| 30|
+----+--------+------+
```

This resulting DataFrame, `df`, serves as our operational dataset. We now have ten records across three distinct teams (A, B, and C). Our goal in the next section is to calculate precisely how many unique point values are represented within the records belonging to Team A, Team B, and Team C, respectively, demonstrating a core use case for PySpark aggregation functions.

Practical Application: Counting Distinct Points Per Team

With our sample DataFrame prepared, we can now execute the primary calculation: finding the count of distinct points scored, grouped by team. This operation immediately demonstrates the efficiency and clarity of the PySpark API. We import `countDistinct` and apply it within the `.agg()`

function following the `.groupBy('team')` definition. This tells Spark to look at all rows associated with Team A, then Team B, and so on, and count only the unique values found in the 'points' column within that specific subset.

Executing the code yields a new DataFrame that compactly summarizes the desired cardinality metric. Note how the resulting column name defaults to `count(points)`, reflecting the function applied. This output is crucial for understanding the variety of scores achieved by each unit (the team) in our data model. The ability to perform such complex calculations across millions or billions of rows is what makes distributed computing frameworks like Spark essential for modern data processing.

from pyspark.sql.functions import countDistinct

#calculate distinct values in points column, grouped by team column
df.groupBy('team').agg(countDistinct('points')).show()

+----+-------------+
|team|count(points)|
+----+-------------+
B	2
C	2
A	3
+----+-------------+

Interpreting the resultant DataFrame provides immediate insights into the dataset's structure:

There are **2** distinct values in the points column for team B (13, 14).
There are **2** distinct values in the points column for team C (23, 30).
There are **3** distinct values in the points column for team A (11, 8, 22).

This result confirms that the countDistinct function operates exactly as intended, providing an accurate measure of diversity within the specified column, segregated by the grouping key.

Refining Output: Renaming Aggregate Columns Using alias

While the default column naming convention (e.g., `count(points)`) is functional, it often lacks semantic clarity, especially when multiple aggregations are performed simultaneously or when the resulting DataFrame is integrated into downstream reporting tools. To enhance readability and maintain clear naming conventions, PySpark provides the `.alias()` function. This function allows you to assign a custom, descriptive name to the result of any aggregation expression, making the output easier to consume and interpret.

To implement column renaming, the `.alias()` function is chained directly after the aggregation function call within the `.agg()` method. By naming the output column appropriately--in this case, `distinct_points`--we make the meaning of the aggregated value explicit. This practice is considered a best practice in data engineering, ensuring that schema modifications are immediately understandable to anyone reviewing the code or the resulting output.

Below shows the revised syntax incorporating the `.alias()` function, followed by the cleaner, more professional output:

from pyspark.sql.functions import countDistinct

#calculate distinct values in points column, grouped by team column
df.groupBy('team').agg(countDistinct('points').alias('distinct_points')).show()

+----+---------------+
|team|distinct_points|
+----+---------------+
B	2
C	2
A	3
+----+---------------+

The resulting DataFrame now clearly shows the number of unique points values for each team, identified by the meaningful column name **distinct_points**, exactly as specified using the **alias** function. This final step optimizes the output for integration into other parts of a larger data pipeline or for direct use in visualization tools, solidifying the process of performing grouped distinct counts in PySpark.

Further Considerations and Best Practices

While **groupBy** combined with countDistinct is highly effective, it is crucial to be aware of performance considerations in a large-scale environment. Calculating exact distinct counts inherently requires high network utilization (shuffling) across the Spark cluster, as all data points corresponding to a key must be brought together before uniqueness can be determined. For extremely large datasets where absolute precision is not mandatory, data scientists often opt for approximate distinct counting methods, such as HyperLogLog implementations available via functions like `approx_count_distinct()` in PySpark, which trade a small amount of accuracy for significant performance gains.

Additionally, remember that the **groupBy** function can accept multiple columns. If we had grouped by both `'team'` and `'position'`, the distinct points count would have been calculated for every

unique combination of team and position (e.g., Team A, Guard; Team A, Forward; etc.). This multi-column grouping capability allows for highly granular analytical breakdowns, providing even deeper insights into data distribution and relationships.

Finally, always consult the official PySpark documentation for the most current details regarding function behavior and best practices for performance tuning. Effective utilization of these powerful DataFrame operations ensures that large-scale data processing remains efficient, accurate, and scalable for any modern data challenge.