How to GroupBy and Count Distinct Values in PySpark

Authored by

stats writer

January 1, 2026

The ability to perform robust data aggregation is fundamental to modern data processing, especially when dealing with large-scale datasets. In the context of PySpark, two exceptionally powerful functions enable precise data analysis: **groupBy** and **countDistinct**. These functions are often used in tandem to calculate the uniqueness metrics across various subsets of your data. The groupBy function allows users to partition a DataFrame based on the shared values of one or more specified columns, creating logical groups. Once these groups are defined, you can apply various aggregation operations.

The utility of grouping data becomes apparent when seeking insights into category-specific distributions. For instance, if you are analyzing sales data, grouping by 'Region' allows you to apply calculations that are specific to North America versus Europe. Following the grouping operation, countDistinct provides the mechanism to determine the total number of unique values within a target column for each of these defined groups. This combination is essential for tasks such as calculating the number of unique customers per product line or, as we will demonstrate, finding the distinct performance metrics across different teams. Understanding how to correctly implement **groupBy** and **countDistinct** in PySpark is a cornerstone skill for scalable data science, ensuring that derived metrics accurately reflect the diversity within each subset of the data.

This guide will explore the specific syntax required to leverage these functions efficiently, providing practical, actionable examples rooted in statistical analysis. We will walk through the process of importing necessary libraries, structuring the DataFrame, and executing the aggregation pipeline, ensuring a clear understanding of both the inputs and the resulting aggregated output. Mastering this technique allows for profound insights into data heterogeneity, which is critical for accurate reporting and complex analytical model preparation. The ability to perform precise counts on unique attributes after partitioning is crucial for robust data quality checks and subsequent feature engineering.

Implementing Grouped Distinct Counting in PySpark

To calculate the number of distinct values present in one column of a PySpark DataFrame, partitioned according to the values found in another column, you must utilize the **agg()** function following the **groupBy()** operation. This structure ensures that the distinct counting is applied contextually within each group defined by the grouping column. This pattern is highly efficient in distributed computing environments because Spark can perform the counts in parallel across all worker nodes, speeding up the overall aggregation process significantly compared to non-distributed alternatives. This chaining of methods--from DataFrame to GroupedData to AggregatedData--is characteristic of the PySpark API design.

The general approach involves importing the required functions, specifically countDistinct from `pyspark.sql.functions`, applying the groupBy method to the DataFrame, and then chaining the

agg() method where the **countDistinct()** function is passed as an argument. The argument within **countDistinct()** specifies the column whose unique values we wish to quantify. The following syntax block illustrates the necessary Python code structure for this common analytical task, where 'team' represents the grouping variable and 'points' is the target column for distinct counting:

```
from pyspark.sql.functions import countDistinct

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

This particular execution calculates the number of unique point totals recorded in the **points** column, with the results segmented by the corresponding values in the **team** column. It is important to remember that countDistinct only counts unique occurrences; duplicates within a group are counted only once. This methodology provides a concise and powerful tool for obtaining summary statistics that describe the variability within key categorical subsets of your dataset, offering a critical layer of analytical depth beyond simple total row counts. Moreover, the efficiency of this operation scales well due to Spark's underlying architecture, making it suitable for petabyte-scale data analysis.

Setting Up the PySpark Environment and Sample Data

Before executing any PySpark commands, it is essential to initialize a **SparkSession**, which serves as the entry point to programming Spark with the DataFrame API. Once the session is established using `SparkSession.builder.getOrCreate()`, we can define our sample data. For demonstration purposes, we will use a dataset representing scores achieved by basketball players across different teams and positions. This dataset is intentionally structured with duplicate entries and varied scores to clearly illustrate how the **countDistinct** function operates within groups, particularly how it handles redundant information efficiently.

The data is defined as a list of rows, where each row contains the team identifier, the player position, and the points scored. We then define the corresponding column names: 'team', 'position', and 'points'. This meticulous preparation ensures the resulting DataFrame is correctly structured for the subsequent groupBy analysis. Defining schemas clearly upfront is a recommended practice in Spark to avoid inference issues and to ensure optimal performance during distributed processing.

The following example demonstrates the complete setup required to create and display the initial DataFrame, confirming the data integrity and structure before proceeding to the aggregation phase:

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

Executing the GroupBy and CountDistinct Aggregation

With the DataFrame initialized and structured correctly, the next step is to execute the core aggregation logic. Our goal remains consistent: to determine how many unique scoring

performances (distinct values in the **points** column) are associated with each basketball team (grouped by the **team** column). This is a common requirement in performance analysis where understanding the variety of outcomes is as important as the average outcome. This approach provides insight into scoring consistency versus diversity across different organizational units.

To achieve this, we first call `df.groupBy('team')`, which logically separates the DataFrame rows into distinct segments: Team A, Team B, and Team C. Subsequently, the `.agg(countDistinct('points'))` function is applied to each of these segments independently. Crucially, the **countDistinct** function evaluates the 'points' within Team A only, then within Team B only, and so on, ensuring that a score of '14' in Team B is counted separately from the unique scores found in Team A.

We use the following compact syntax, which showcases the fluent API design of PySpark, allowing complex operations to be chained seamlessly. This is the most direct and idiomatic way to express this requirement in PySpark SQL functions:

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 Grouped Distinct Count Results

The resultant DataFrame provides a clear summary of the heterogeneity of scores within each team. The output contains two columns: the grouping column (**team**) and the newly aggregated column, automatically named **count(points)** by default PySpark behavior when using countDistinct without an explicit alias. Analyzing this output allows us to quickly identify which teams exhibit the greatest variety in individual player scores, which can be an indicator of diverse player roles or varied strategic outcomes.

By reviewing the original data alongside the aggregated results, we can verify the correctness of the **countDistinct** operation, confirming that the shuffles and counts performed by Spark were accurate:

For **Team A**, the recorded points are 11, 8, 22, and 22. When counting distinct values, 22 is only counted once. Therefore, the unique scores are 11, 8, and 22, resulting in a count of **3**. This indicates a high variance in scoring results for Team A.
For **Team B**, the recorded points are 14, 14, 13, and 14. The unique scores are 13 and 14. This yields a count of **2** distinct values in the points column for team B. Despite having four records, there were only two unique score values.
For **Team C**, the recorded points are 23 and 30. Both scores are unique, resulting in **2** distinct values in the points column for team C.

The resulting DataFrame accurately reflects these calculations, confirming the functionality of the groupBy and **countDistinct** combination in segmenting the data and calculating unique metrics efficiently across large datasets. This type of analysis is invaluable in quality control, anomaly detection, and exploratory data analysis where understanding underlying data distributions is key.

Renaming the Output Column using the Alias Function

While PySpark provides a sensible default name for the aggregated column (e.g., `count(points)`), in production environments or complex analytical pipelines, it is standard practice to rename columns to be more descriptive, clearer, and easier to reference later in subsequent transformations. The alias function is specifically designed for this purpose, allowing users to assign a custom name to the output of a column transformation, such as the result of **countDistinct()**. Using descriptive names like `distinct_points` avoids ambiguity and prevents potential conflicts with other columns.

To implement custom naming, the **alias()** method is chained directly to the aggregation function within the **agg()** call. We pass the desired output name, for example, `distinct_points`, as the argument to **alias()**. This simple addition drastically improves the readability of the resulting schema and makes integrating the summary statistics into final reports much cleaner. This step is highly recommended for building production-ready, maintainable code, adhering to rigorous software engineering principles.

The revised syntax incorporating the alias function looks like this, demonstrating how the column expression is constructed before being passed to `agg()`:

from pyspark.sql.functions import countDistinct

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

+----+---------------+
|team|distinct_points|

```
+----+---------------+
| B| 2|
| C| 2|
| A| 3|
+----+---------------+
```

The resulting DataFrame successfully displays the aggregated counts, but with the key difference that the output column is now clearly labeled **distinct_points**, exactly as defined using the **alias** function. This practice ensures data cleanliness and improves collaboration among analysts, making the output intuitive and self-documenting.

Advanced Considerations: Performance and Alternatives

While countDistinct is highly effective for exact calculations, it is important to understand its performance characteristics within a distributed environment. Calculating exact distinct counts requires extensive data shuffling across the network to ensure that all identical values are brought together before counting. This shuffle operation can be computationally intensive and costly in terms of network overhead, especially for columns with high cardinality (many unique values). Data professionals must balance the need for precision against the cost of execution.

When dealing with truly massive datasets where absolute precision is not paramount, PySpark offers alternatives. One such alternative is the **approx_count_distinct** function. This function uses the HyperLogLog algorithm to provide a highly accurate estimate of the distinct count while significantly reducing the memory footprint and the need for expensive data shuffling. For use cases like dashboard monitoring, real-time metrics, or trend analysis, where a small margin of error is acceptable (typically less than 5%), leveraging **approx_count_distinct** can provide substantial performance gains over the exact **countDistinct** method, particularly when grouped over many partitions using groupBy. Choosing between the exact and approximate method depends entirely on the specific precision requirements and latency tolerances of the analytical task at hand.

Further Exploration: Combining Multiple Aggregations

It is important to note that the **agg()** function is not limited to a single operation. Analysts frequently need to calculate multiple summary statistics simultaneously after grouping. For instance, in addition to finding the distinct number of scores, one might also want the average score (using `avg()`) and the maximum score (using `max()`) for each team. The **agg()** method easily accommodates this by accepting multiple aggregation functions as arguments, often using dictionaries or named arguments for clarity, allowing all metrics to be computed efficiently in parallel within the same grouped context.

This multi-aggregation capability significantly streamlines the data processing pipeline, eliminating the need for multiple separate **groupBy** operations and ensuring that all summary metrics are computed in a single pass over the grouped data. This efficiency is critical when working with performance-sensitive distributed systems like PySpark, maximizing resource utilization and reducing execution time. By mastering the integration of functions like countDistinct with other analytical tools, such as `sum`, `avg`, and `max`, users gain full control over their data summarization needs in a concise and optimized manner.

Note: You can find the complete documentation for the PySpark **groupBy** function here.