Count Distinct Values in PySpark (3 Methods)

Authored by

stats writer

November 16, 2025

Analyzing the unique values, or distinct values, within datasets is a fundamental task in data cleaning and exploratory data analysis. When working with massive datasets, the powerful distributed processing capabilities of Apache Spark, accessed via the Python API, are essential for efficiency. Determining cardinality helps identify data quality issues, understand feature distributions, and optimize storage when utilizing the PySpark DataFrame structure.

In PySpark, there are three primary, highly efficient techniques to calculate distinct counts, ranging from targeting a single column to inspecting the uniqueness of entire rows within your data. Choosing the correct method depends on the specific analytical goal: whether you need the cardinality of one field, an overview of all fields, or an accurate row count after deduplication.

Method 1: Counting Distinct Values in a Single Column

The most straightforward requirement is often determining the number of unique entries for a specific column. This task is crucial for feature engineering or verifying categorical distributions before running models. PySpark facilitates this using the combination of the agg function and the specialized countDistinct SQL function.

The agg function allows you to apply aggregation operations (like counts, sums, or distinct counts) across your entire PySpark DataFrame, returning a new single-row DataFrame containing the results. When coupled with `countDistinct(col('column_name'))`, it efficiently computes the count of distinct values for the specified column by performing the aggregation logic in parallel across the cluster.

Using `.alias()` ensures that the resulting aggregated column is clearly labeled with the name of the column you analyzed, maintaining clarity in the output. This method minimizes data movement compared to collecting unique values, making it highly scalable for finding single-column cardinality.

from pyspark.sql.functions import col, countDistinct

df.agg(countDistinct(col('my_column')).alias('my_column')).show()

Method 2: Counting Distinct Values in Every Column

Often, initial data exploration requires a quick survey of the cardinality for every field in the dataset to identify potential identifiers, quasi-identifiers, or fields with very low variability. Manually applying Method 1 to dozens or hundreds of columns is inefficient and error-prone. PySpark allows us to dynamically generate aggregation expressions for all columns simultaneously.

This technique leverages Python's generator expressions coupled with the splat operator (*) inside

the agg function. The generator expression iterates through `df.columns`, the built-in attribute that holds all column names of the DataFrame. For each column `c`, it dynamically creates a named aggregation: `countDistinct(col(c)).alias(c)`.

The resulting expressions are unpacked directly into the `agg()` call, executing all distinct counts concurrently. This results in a single aggregated row displaying the distinct count for every column in the input DataFrame. This pattern is considered a best practice for comprehensive cardinality checks on wide datasets due to its brevity and efficiency.

from pyspark.sql.functions import col, countDistinct

df.agg(*(countDistinct(col(c)).alias(c) for c in df.columns)).show()

Method 3: Counting the Number of Distinct Rows in the Entire DataFrame

The final crucial metric is determining the total number of unique rows in the dataset, treating the entire row combination across all columns as a single entity. This differs fundamentally from Methods 1 and 2, which operate on columns individually. Counting distinct rows is essential for detecting exact duplicate records, which can corrupt statistical analysis or lead to biased machine learning models if not removed.

PySpark provides the built-in `distinct()` transformation. When executed, this returns a new PySpark DataFrame containing only the unique rows, eliminating any exact duplicates. Since `distinct()` is a transformation, it is lazily evaluated. To force the computation and retrieve the final numerical count, we must chain it with the `count()` action.

It is important to understand that `distinct()` is a wide transformation, meaning it requires shuffling the data across the cluster partitions to group and compare similar rows. While highly accurate, this operation can be resource-intensive and time-consuming on extremely large-scale data and should be used judiciously, typically only when confirming data integrity or performing large-scale deduplication.

df.distinct().count()

Setting Up the Sample PySpark DataFrame for Demonstration

To provide concrete examples of these three methods in action, we must first initialize a sample PySpark environment and populate a DataFrame. We will use a small dataset containing information about basketball players, which includes intentional duplicate rows and repeated values across the `team`, `position`, and `points` columns. This setup allows us to verify that each

method produces the expected cardinality metric.

The setup involves initiating a `SparkSession`--the entry point to PySpark functionality--and then structuring the raw data and column names before calling `spark.createDataFrame()`. Pay close attention to the raw data structure: the row appears twice, as does . The total initial row count is 8, but we expect the unique row count to be lower.

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	7
+----+--------+------+

Example 1: Applying Single Column Count to 'team'

We begin by implementing Method 1, focusing specifically on determining the cardinality of the `team` column. This is a crucial step in understanding the distribution of players across different teams. We utilize the agg function combined with `countDistinct` to execute this calculation across the distributed dataset.

The code below targets the `team` column, calculates the number of unique entries (A and B), and presents the result as a new, concise DataFrame. This approach is highly optimized for performance, as it only needs to calculate the unique values for one field without interacting with the data in other columns.

from pyspark.sql.functions import col, countDistinct

#count number of distinct values in 'team' column
df.agg(countDistinct(col('team')).alias('team')).show()

+----+
|team|
+----+
| 2|
+----+

The output confirms that there are exactly **2** distinct values in the `team` column. This outcome is expected, given that the data only includes teams 'A' and 'B'.

Example 2: Analyzing Distinct Counts Per Column

Next, we apply Method 2 to obtain a holistic view of the cardinality across all three columns--`team`, `position`, and `points`--with a single, dynamic command. This calculation is vital for profiling the data, especially when dealing with schemas containing a large number of columns.

By using the generator expression technique, we ensure that PySpark processes the unique counts for all fields in parallel, minimizing the total execution time compared to querying each column sequentially. The result is a single row that serves as an essential summary statistic for the dataset's distribution.

from pyspark.sql.functions import col, countDistinct

#count number of distinct values in each column
df.agg(*(countDistinct(col(c)).alias(c) for c in df.columns)).show()

```
+----+--------+------+
|team|position|points|
+----+--------+------+
| 2| 2| 6|
+----+--------+------+
```

Analyzing the results provides detailed insights into the data distribution:

There are **2** unique values in the `team` column (A, B).
There are **2** unique values in the `position` column (Guard, Forward).
There are **6** unique values in the `points` column (11, 8, 22, 14, 13, 7). Although the total row count is 8, two point values (22 and 14) are repeated across different players, confirming the distinct count is six.

Example 3: Determining the Total Number of Distinct Rows

Finally, we execute Method 3 to check for row-level duplication. This step determines the definitive size of our unique dataset by counting how many distinct combinations of `(team, position, points)` exist. This is the gold standard for data cleaning.

We apply the `distinct()` transformation to identify and eliminate the two known duplicate rows, followed immediately by the `count()` action to return the numerical result. Since the original DataFrame had 8 rows, and we know two are duplicates, we anticipate a final count of 6 unique records.

#count number of distinct rows in DataFrame
df.distinct().count()

6

The resulting count is **6**. This confirms that of the initial 8 rows, 2 were exact duplicates (and). This metric is critical for ensuring that subsequent analyses operate on a clean, deduplicated dataset, thereby preserving statistical integrity.