Learn How to Count Distinct Values in PySpark DataFrames: A Comprehensive Guide

Authored by

Mohammed looti

November 11, 2025

Introduction to Counting Distinct Values in PySpark

In modern **data analysis** and preparation, especially when navigating massive datasets, the ability to rapidly determine the number of unique elements is absolutely fundamental. For processing big data at scale, PySpark stands as the essential Python API, granting users access to the powerful, distributed computation framework of Apache Spark. A core aspect of initial data profiling involves understanding the **cardinality** of various fields--that is, calculating the total count of unique values present in a column. This information is critical for diverse tasks, ranging from effective **feature engineering** and rigorous data quality validation to optimizing complex downstream machine learning models. For instance, a column exhibiting high cardinality demands different processing strategies than one with low cardinality.

This comprehensive tutorial outlines three distinct and highly efficient methodologies for calculating the count of unique values within a DataFrame using specialized PySpark SQL functions. The methods presented here are tailored to address varying analytical requirements: whether your goal is to ascertain the cardinality of a single column, simultaneously profile all columns, or determine the total number of unique rows across the entire dataset. Working effectively with PySpark DataFrames requires leveraging the highly optimized functions found within the `pyspark.sql.functions` module, ensuring operations are performed efficiently within Spark's **distributed environment** rather than being confined to local memory.

The choice of technique for counting distinct values directly impacts performance and the scope of your analysis. Are you focused primarily on identifying **data sparsity** within a single field, or are you executing a full-scale, comprehensive data profile across dozens of schema columns? Each of the three approaches we detail utilizes Spark's distributed computation model effectively, but they target different granularities of distinctness measurement. We will meticulously explore the necessary syntax and provide practical applications for each method, ensuring you select the most appropriate and performant tool for your specific data task.

It is crucial to understand the underlying mechanism: calculating distinct values in PySpark often triggers a shuffle operation. A shuffle is a computationally expensive process where data must be redistributed across network partitions to group identical records together. Therefore, writing high-performance Spark code involves minimizing unnecessary shuffles. Functions like `countDistinct` are specifically engineered to execute these complex aggregations efficiently. We recommend always initiating your session using `SparkSession.builder.getOrCreate()` to establish a consistent and optimized entry point to Spark functionality, as demonstrated in the setup phase below.

The following three methods are essential techniques for counting distinct values in a PySpark DataFrame:

Method 1: Count Distinct Values in One Column

from pyspark.sql.functions import col, countDistinct

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

Method 2: Count Distinct Values in Each Column

from pyspark.sql.functions import col, countDistinct

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

Method 3: Count Number of Distinct Rows in DataFrame

df.distinct().count()

Setting Up the PySpark Environment and Sample Data

Before we dive into the implementation details of distinct counting, it is necessary to establish a functional PySpark environment and construct a sample **DataFrame** that we can use for demonstration. This foundational setup involves two primary steps: first, initializing the **SparkSession**--which serves as the mandatory unified entry point for all Spark functionalities--and second, defining the specific structure and content of our sample dataset. Our chosen example utilizes basketball player statistics, providing a clear, concrete context for observing how unique values distribute across various categories, such as team affiliation, player position, and points scored. The methodical preparation of this data simulates a very common initial task in a data science workflow: loading raw data and requiring immediate profiling before any deep analysis can begin.

The code block below handles the initialization of Spark and the creation of our demonstration DataFrame. It is important to note the clear, structured separation within the code: defining the raw data list, specifying the column **schema**, and finally invoking the `spark.createDataFrame()` method. This deliberate approach ensures both clarity and full reproducibility for readers attempting to replicate the results. The resulting DataFrame, which contains eight rows, has been intentionally engineered to include explicit duplications in key columns (like 'team' and 'position') and full row duplications. This design allows us to thoroughly test and validate the accuracy of the three distinct counting methodologies we are about to explore.

This generated DataFrame will serve as the immutable foundation for all subsequent code examples in this guide. We highly recommend that readers execute this setup code within their

own Spark environment to follow along precisely and confirm the input data state. Pay careful attention to the output generated by `df.show()`; this visual representation of the data is invaluable for confirming the counts produced by the distinct functions. For instance, a visual inspection reveals only two unique entries in the 'team' column ('A' and 'B'), while the 'points' column shows a greater variation of scores, providing immediate and intuitive validation points for our upcoming cardinality calculations.

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
+----+--------+------+

Method 1: Counting Distinct Values in a Single Column

The most common requirement is often calculating the **cardinality** of a specific, targeted column. This approach is highly recommended for efficiency, as it minimizes computational overhead by restricting the required data processing and shuffle operation only to the data contained within that single field. To execute this targeted count, we strategically combine two core PySpark SQL components: the `countDistinct` function and the `.agg()` method. The `countDistinct` function, which must be imported from `pyspark.sql.functions`, is designed to accept a column reference, typically provided either by wrapping the column name in `col()` or by using a string literal.

The `.agg()` function is indispensable in DataFrame operations, acting as the mechanism for executing aggregate functions--functionally similar to the familiar GROUP BY clause in traditional SQL, but without requiring explicit grouping when calculating a total count across the dataset. By nesting the `countDistinct` function call within `.agg()`, we instruct PySpark to compute the precise total number of unique, non-null values for the specified column across the entirety of the DataFrame. Furthermore, adopting the best practice of using `.alias()` to rename the resulting aggregated column is vital. This ensures the output is clean, professional, and unambiguously indicates the column to which the distinct count corresponds, which is essential when complex aggregations are combined.

We will now apply this powerful method to determine the number of unique entries solely within the `team` column of our sample data. Based on our initial visual inspection during the setup phase, we anticipate an exact count of **2** (representing Team A and Team B). This technique delivers a fast, reliable mechanism for profiling individual categorical variables, enabling data scientists to quickly diagnose potential issues such as inconsistent data entry or unexpected levels of variation. The final output is always a new, single-row DataFrame clearly displaying the calculated distinct count under the designated aliased column name.

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 our expectation: there are exactly **2** distinct values present in the `team`

column. This result validates the effectiveness and scalability of Method 1, which should be the primary choice when profiling only a limited subset of columns.

Method 2: Counting Distinct Values Across All Columns Simultaneously

While targeted profiling of a single column is useful, the requirement often shifts toward generating a holistic data profile, demanding the distinct count for every column within the DataFrame in one streamlined operation. Manually constructing individual aggregation expressions for a dataset containing dozens or hundreds of columns is tedious and highly susceptible to error. PySpark offers an elegant, idiomatic Python solution that leverages a concise list comprehension combined with the unpacking operator (`*`) used inside the `.agg()` function.

This technique exploits Python's inherent capability to iterate over the DataFrame's column names, which are easily accessed via the `df.columns` attribute. For every column name `c` encountered during the iteration, we dynamically construct the required aggregation expression: `countDistinct(col(c)).alias(c)`. This process generates a comprehensive list of aggregate expressions. Critically, the unpacking operator (`*`) is then utilized within `df.agg()` to expand this list, effectively passing every individual distinct count expression as a separate argument to the aggregation function. This results in the execution of a single, highly optimized aggregation job that calculates all necessary distinct counts concurrently across the entire **distributed environment**.

Method 2 is immensely practical for Exploratory Data Analysis (EDA), where a swift, high-level overview of feature variability and cardinality is essential. It consolidates all results into a single output row, where each column reports its respective distinct count. Although this operation inherently involves a more extensive internal data shuffle operation compared to the single-column method, its primary benefit lies in minimizing the total number of code execution steps and producing a compact, easily digestible summary output. Below is the implementation demonstrating how to count the distinct values for 'team', 'position', and 'points' concurrently.

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|
+----+--------+------+

The resulting output provides a clear, concise summary of the **cardinality** for all fields within the

DataFrame:

There are **2** unique values in the `team` column.

There are **2** unique values in the `position` column.

There are **6** unique values in the `points` column.

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

Moving beyond analyzing individual column cardinality, a highly critical step in maintaining data quality is accurately determining the existence and quantity of fully duplicate rows. A fully duplicate row is defined as one where the values across every single column exactly match another row within the dataset. Accurately counting these fully unique rows provides crucial insight into data integrity and is an essential prerequisite for any subsequent **deduplication** efforts. PySpark provides a remarkably straightforward and powerful method for achieving this through the sequential chaining of the `.distinct()` transformation followed by the `.count()` action.

The `.distinct()` transformation is conceptually simple: when applied to a DataFrame, it guarantees that only rows unique across all columns are retained, filtering out all exact copies. However, due to its nature, this operation is computationally intensive in a large-scale distributed setting. Identifying uniqueness across all fields requires comparing every record with every other record, which necessitates a full, expensive shuffle operation of the data across all cluster partitions. Consequently, `.distinct()` is classified as a **wide transformation** and must be used with careful consideration on extremely large datasets.

Once the `.distinct()` transformation has successfully been executed, the resulting DataFrame contains only the unique observations. The final step is applying the `.count()` action, which triggers the execution of all preceding transformations (including the shuffle) and returns the final numerical count of the unique rows. Reviewing our sample data, we previously identified two explicit duplicate rows: the row for appears twice, and also appears twice. Since the original DataFrame contains 8 total rows, we logically expect the final distinct count to be 6 (8 total rows minus the 2 duplicate pairs).

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

6

The output confirms precisely that there are **6** distinct rows in the DataFrame. This method is the definitive, gold-standard way to measure the true number of unique observations in your dataset

before proceeding with complex analytical modeling or feature processing.

Summary and Performance Considerations

We have successfully navigated and implemented three robust methods for calculating distinct values in PySpark DataFrames, with each technique serving a specific and valuable analytical purpose. **Method 1** (using `countDistinct` on a single column via `.agg()`) is the preferred strategy for quick, targeted **cardinality** checks and offers the best performance if only one column is needed, as it significantly minimizes the scope of the necessary data shuffle. **Method 2** (the dynamic application of `countDistinct` across all columns) provides a rapid, high-level overview of data variation, trading a slight performance hit for high efficiency in comprehensive profiling by consolidating all counts into a single Spark job. Finally, **Method 3** (the use of `.distinct().count()`) is essential for thorough data quality assessment, specifically for identifying and quantifying the exact presence of fully duplicated records across all fields.

When operating in a massive-scale distributed environment, **performance optimization** must always be a paramount concern. It is important to reiterate that both `countDistinct` and `.distinct()` operations are categorized as "wide transformations" under Apache Spark because they fundamentally necessitate a full shuffle operation--redistributing the entire dataset across the cluster partitions to group identical keys. This shuffling is notoriously resource-intensive, consuming substantial network bandwidth and CPU time. For scenarios involving extremely high-cardinality columns or DataFrames so massive that an exact count is impractical, PySpark offers a valuable alternative: the `approx_count_distinct` function. This function utilizes sophisticated **HyperLogLog algorithms** to provide a highly accurate, yet dramatically faster, estimate of the distinct count, all while maintaining bounded memory usage.

However, for routine data quality tasks and standard Exploratory Data Analysis (EDA), where precision is paramount, utilizing the exact counting methods detailed here remains the standard, most reliable practice. Always ensure your code is clean and manageable by importing only the necessary functions (such as `col` and `countDistinct`). A thorough mastery of these fundamental aggregation techniques is indispensable for any professional working with big data, forming the bedrock upon which more complex analytical pipelines and advanced transformations are built using the PySpark API.

Further Resources for PySpark Mastery

To continue deepening your expertise in PySpark and related big data technologies, we strongly recommend dedicated exploration of the official **Apache Spark documentation** and comprehensive tutorials that detail the usage of PySpark SQL functions. Achieving proficiency with complex transformations, understanding the mechanics of window functions, and mastering

optimization techniques will enable you to fully leverage the immense power of the **SparkSession** and the underlying distributed computation engine. The following resources offer detailed explanations on how to execute other common and complex data manipulation tasks efficiently within the distributed framework.

Mastering advanced aggregation operations--such as grouping, summing, and averaging--is a natural and necessary progression from counting distinct values. These operations frequently rely on the same fundamental underlying principles of data shuffling, aggregation planning, and memory management. We strongly encourage exploring tutorials that cover essential topics like robust handling of null values, performing complex joins (e.g., left anti-joins and full outer joins), and developing and utilizing **User Defined Functions (UDFs)** to precisely customize your data processing workflow. Continuous learning in these areas is the definitive key to maintaining high-performance, scalable data pipelines in any big data environment.

The following tutorials explain how to perform other critical data manipulation tasks in PySpark:

How to filter rows based on multiple conditions.

Best practices for efficiently joining two DataFrames using different join types.

Implementing window functions for ranking, sequencing, and advanced analytical calculations.