How to Find the Mode of a Column in PySpark DataFrames

Authored by

stats writer

January 19, 2026

Understanding the Mode in PySpark DataFrames

The Mode, a key measure of central tendency in statistics, identifies the value that occurs with the greatest frequency within a dataset. In the context of big data processed by PySpark, calculating the mode is crucial for exploratory data analysis (EDA), helping analysts understand the typical or most common attribute state in a given feature column, regardless of whether that column contains numerical or categorical data. While standard SQL or Pandas approaches work for small datasets, distributed frameworks require specialized techniques to perform this calculation efficiently across partitions.

Calculating the mode in a distributed environment like Apache Spark requires aggregating data based on unique values and then counting their occurrences. Since the PySpark DataFrame does not have a dedicated built-in `mode()` function, we must implement a custom solution utilizing powerful aggregation tools. The core strategy involves applying the groupby function combined with the count aggregation to determine frequency distributions. This approach ensures that the calculation is performed in parallel, maximizing performance on large-scale data.

This document presents two highly efficient methods for determining the mode of a column in a PySpark DataFrame. The first method is highly optimized for retrieving the mode of a single, specified column, offering simplicity and clarity for targeted analysis. The second method provides a more generalized, programmatic approach using list comprehensions to iterate through all columns in the DataFrame and return the mode for each one, proving invaluable when seeking a quick overview of central tendencies across an entire schema.

Methodology Overview: Targeting Single vs. Multiple Columns

Method 1: Calculate Mode for One Specific Column

This method focuses precision, targeting a single column using a sequence of chained operations. It groups the data by the unique values in the target column, counts the occurrences of each group, orders these counts in descending fashion, and finally extracts the first row, which corresponds to the value with the highest frequency--the mode. This is the recommended approach when the analyst knows exactly which feature they need to summarize.

#calculate mode of 'conference' column
df.groupby('conference').count().orderBy('count', ascending=False).first()

Method 2: Calculate Mode for All Columns

For a broader analysis, this method employs a list comprehension across the DataFrame's columns. It programmatically applies the same groupby, count, and ordering logic to every column

iteratively. The result is a list of lists, where each inner list pairs the column name with its corresponding mode value, providing a comprehensive statistical summary of the dataset in a concise format. This approach is highly efficient for data quality checks or initial statistical profiling.

#calculate mode of each column in the DataFrame
] for i in df.columns]

Setting up the PySpark Environment and Sample Data

To practically demonstrate these two methods, we will first establish a PySpark environment by initializing a SparkSession and then construct a sample DataFrame. This small, representative dataset simulates real-world sports data, containing both categorical columns (`team`, `conference`) and numerical columns (`points`, `assists`). The presence of different data types ensures that our mode calculation methods are robust across various column schemas.

The following code block outlines the necessary steps: importing the requisite modules, creating the Spark entry point, defining the sample data structure, and finally generating the DataFrame (`df`). This setup is foundational, as all subsequent mode calculations will operate directly on this distributed data structure. Note the explicit steps for defining both the raw data rows and the column names to ensure clarity and proper schema definition when using the `createDataFrame` function.

Observing the output of `df.show()` confirms the successful creation and structure of the dataset. We can already visually inspect the data to predict which values might be the Mode for certain columns, for instance, 'East' appears frequently in the `conference` column, and 'A' in the `team` column. This initial visualization is helpful for validating the results of the automated mode calculation that follow.

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|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
| B| West| 6| 12|
| B| West| 6| 4|
| C| East| 5| 2|
+----+----------+------+-------+
```

Example 1: Calculating the Mode for One Specific Column

When the goal is to specifically identify the most frequently occurring value within a single attribute, such as the `conference` column, we apply Method 1. This highly targeted approach uses a sequence of chained transformations optimized for single-column aggregation. The process begins with the `df.groupby('column_name')` function, which organizes the distributed dataset by the unique values present in the specified column. Immediately following this grouping, the `.count()` method calculates the total number of records belonging to each unique group, resulting in a temporary DataFrame with two columns: the unique value and its corresponding frequency count.

The subsequent steps are critical for isolation: `.orderBy('count', ascending=False)` sorts the results so that the value with the highest frequency appears first in the dataset. Since the Mode is defined as the most frequent value, this sorting operation places our desired result at the top. Finally, we use `.first()` to retrieve only the top row containing the mode and its count, and then we access the zeroth element of the resulting row object (), which corresponds to the column value itself (the actual mode), discarding the frequency count for the final output.

For our sample DataFrame, applying this logic to the `conference` column yields 'East'. This means that the Eastern conference appears more often than any other conference in our dataset, representing the most common categorical assignment. This entire process is highly efficient because Spark executes these chained operations as a single optimized query plan across the cluster.

#calculate mode of 'conference' column
df.groupby('conference').count().orderBy('count', ascending=False).first()

'East'

The calculated Mode of the **conference** column is **East**.

Deep Dive into the Single Column Mode Logic

The effectiveness of Method 1 stems from the efficient handling of the groupby and ordering operations by Spark's underlying execution engine. When dealing with enormous datasets where the sheer volume of unique values might exceed available memory on a single machine, this distributed approach becomes indispensable. Spark intelligently shuffles the data only when necessary--during the groupby and count phase--to gather identical keys together for accurate frequency aggregation.

It is important to understand the role of the `.first()` action. In PySpark, actions trigger computation, and `.first()` retrieves the first row as a `Row` object. Since we have already sorted the data, this row is guaranteed to contain the Mode. Accessing of the `Row` object is necessary because the row structure includes both the value (index 0) and the count (index 1). If a column has multiple values tied for the highest frequency (a multimodal distribution), this syntax will arbitrarily return only one of those values--the one that happens to appear first after the distributed sorting operation, which is standard behavior for simple mode retrieval.

This streamlined code snippet effectively abstracts away the complexities of distributed frequency calculation. It is a powerful example of how PySpark allows analysts to perform sophisticated Statistical analysis using concise, readable syntax, transforming what could be a multi-step iterative process in other languages into a single, efficient chain of operations.

Example 2: Calculating the Mode Across All Columns

To profile an entire DataFrame quickly, Method 2 uses a Python list comprehension combined with the inherent capabilities of the DataFrame object. This approach iterates through `df.columns`, applying the exact same frequency calculation logic detailed in Method 1 to every column in the dataset, regardless of its data type. This programmatic efficiency avoids writing redundant code blocks for each column, making it ideal for wide datasets or scripts where the schema might change dynamically.

The structure `for i in df.columns]` ensures that for each column name `i`, we execute the aggregation, sorting, and extraction steps, and then pair the column name `i` with the resulting mode value. This results in a comprehensive list where every element is a list containing the

column name and its corresponding mode. This output structure is clean, making it simple to parse the results for visualization or further processing.

The following syntax demonstrates how to calculate the mode in each column of the DataFrame (`team`, `conference`, `points`, and `assists`). Notably, this method works seamlessly across both categorical columns (like `team` and `conference`) and numerical columns (like `points` and `assists`), identifying the most frequent item in each case. The consistency of this approach highlights its utility in generalized data profiling tasks.

#calculate mode of each column in the DataFrame
] for i in df.columns]

, , ,]

Interpreting the Results of the Multi-Column Mode Calculation

The resulting output from Method 2, `, , , ]`, provides an immediate statistical snapshot of the dataset. This format is easily convertible into a dictionary or another structured data type if needed for downstream applications. The clarity of the output allows data users to immediately identify the most typical observation for every attribute without requiring manual iteration or inspection.

For example, in the `points` column, the mode is 6. This signifies that the score of 6 points occurs more frequently than any other score in the dataset, even though `points` is a numerical field. Similarly, the mode of 4 for the `assists` column indicates its highest frequency. This finding is valuable because the Mode often reveals structural biases or common data entry patterns that might be obscured when only reviewing measures like the mean, which are susceptible to extreme outliers. The generalized approach is essential for large-scale data validation and model feature selection.

We can summarize the findings in a simple list format for readability:

The mode of the **team** column is 'A'.
The mode of the **conference** column is 'East'.
The mode of the **points** column is 6.
The mode of the **assists** column is 4.

Technical Summary: The Aggregation Engine

It is important to consolidate the underlying mechanism driving both mode calculation examples. In essence, both methods rely fundamentally on the `groupby` and `count` functions. The `groupby` function initiates the aggregation process, organizing the data by unique column values, while

`count` calculates the frequency of each unique group. The remainder of the chain (`orderBy` and `first`) is simply a mechanism for extracting the specific value associated with the maximum frequency.

The choice between Method 1 and Method 2 depends heavily on the analytical objective. Method 1 is preferred when focusing on a specific feature, offering maximum clarity and often slightly better performance due to the lack of iterative overhead. Method 2, while offering breadth by covering all columns, involves repeated execution of the aggregation logic within a Python loop, making it a better fit for comprehensive, high-level dataset profiling rather than deep-dive analysis into a single variable.

Understanding these PySpark DataFrame operations is foundational for advanced data manipulation. The ability to perform complex statistical summaries, like mode calculation, using simple chained syntax underscores the power and efficiency of the Spark framework in big data analytics pipelines. Mastering aggregation techniques is a cornerstone of effective distributed data processing.

Conclusion and Further Reading in PySpark

We have successfully demonstrated two powerful and flexible methods for calculating the Mode of columns within a PySpark DataFrame. Whether you need a targeted result for a single column or a comprehensive summary across your entire dataset, the combination of `groupby`, `count`, and ordering provides a robust solution for frequency analysis in a distributed computing environment. These techniques are standard practice for data scientists working with Apache Spark.

To further enhance your skills in PySpark and data manipulation, consider exploring other common statistical and transformation tasks. Understanding how to calculate other central tendencies (like median or trimmed mean), handling missing values, or performing advanced window functions will build upon the foundational aggregation skills demonstrated here. Continued study of the PySpark API will unlock the full potential of distributed processing for complex analytical requirements.

The following resources explain how to perform other common tasks in PySpark: