How to Find the Mode of a Column in PySpark

Authored by

stats writer

February 5, 2026

Calculating the mode of a column is a fundamental operation in exploratory data analysis and statistics. The mode represents the value that occurs most frequently within a dataset. While languages like R and Python's Pandas library often provide direct, simple functions for this calculation, determining the mode in PySpark requires a slightly more procedural approach leveraging Spark's distributed processing capabilities. This is primarily because PySpark is optimized for high-volume data operations and typically does not include a standalone `mode()` function directly applicable to DataFrame columns in the same way.

Although some specialized packages or older versions might imply a built-in `mode()` function, the robust and standard method in modern PySpark relies on combining transformation functions like `groupBy`, `count`, and `orderBy`. This combination allows us to tally the frequency of every unique value and then identify the record with the highest count. It is important to remember that if multiple values share the highest frequency (a multimodal dataset), this standard approach will deterministically return only one of those values--specifically, the first one encountered after sorting. Furthermore, this methodology inherently handles **null values** efficiently by excluding them from the frequency counts, ensuring the calculated mode is based solely on valid data points.

By mastering these techniques, data scientists can seamlessly obtain the most common value in any DataFrame column, preparing the data for sophisticated analysis or necessary imputation strategies, regardless of whether the data is **numerical** or **categorical**.

Calculating the Mode in a PySpark DataFrame

To effectively calculate the mode of one or more columns within a PySpark DataFrame, two primary methods utilizing aggregation and ordering are employed. These methods offer clarity and efficiency, whether you need the mode for a single variable or an overview of the modes across the entire dataset.

The following sections detail the steps and syntax required for both scenarios, offering practical examples using a consistent sample dataset.

Method 1: Calculating the Mode for a Single Specific Column

This approach is ideal when the analytical focus is narrow, requiring the most frequent value only from a designated column. It involves grouping the data by the target column, counting the occurrences, and then extracting the top record after sorting in descending order of frequency. This pipeline is highly efficient because Spark's groupBy operation is optimized for distributed calculations.

The resulting output of this method is the single modal value as a raw Python object, suitable for immediate use in subsequent operations like filtering or imputation.

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

Method 2: Calculating Modes Across All Columns Efficiently

When the goal is to perform a rapid quality check or preliminary analysis on a broad DataFrame, calculating the mode for every column simultaneously is necessary. This powerful technique employs Python's **list comprehension** combined with the core groupBy logic demonstrated above, allowing iteration over the DataFrame schema's list of column names.

Using list comprehension significantly simplifies the code required for repetitive tasks across numerous columns, transforming what might otherwise be a lengthy loop into a single, compact expression. This method returns a list of sub-lists, where each element contains the column name and its corresponding mode, providing a structured output summarizing the entire dataset's central tendency.

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

Setting Up the PySpark Environment and Sample Data

Before executing the mode calculation methods, we must initialize a Spark session and define a sample DataFrame. This ensures a reproducible environment for testing the code snippets provided. We utilize the `SparkSession` entry point, which is standard practice when working with PySpark applications and distributed computing tasks.

The sample data provided below contains a mix of categorical columns ('team', 'conference') and numerical columns ('points', 'assists'), allowing us to demonstrate the mode calculation across different data types effectively, ensuring the methodology is robust regardless of the column schema.

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

As observed in the displayed output, the sample DataFrame `df` consists of six records. We can already anticipate the modes: for 'team', 'A' appears three times; for 'conference', 'East' appears three times; and for 'points' and 'assists', certain values appear twice, setting up clear test cases for our aggregation methods.

Example 1: Demonstrating Mode Calculation for the 'Conference' Column

In this first practical example, we apply Method 1 to determine the most frequent conference affiliation within the dataset. We specifically target the `conference` column, which is a **categorical variable**, using the robust groupBy aggregation pipeline previously introduced.

The sequence of operations--grouping, counting, and sorting--forces PySpark to calculate the exact frequency distribution across the distributed nodes, ultimately bringing the result back to the driver program as a single value via the `first()` action.

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

'East'

The execution confirms that `'East'` is the modal value for the `conference` column. By observing the input data, we confirm this is accurate, as 'East' appears three times, while 'West' appears only two times. This value, representing the most frequently occurring category, can be instrumental in feature engineering or defining the majority class in classification problems.

Example 2: Calculating Modes Across the Entire DataFrame

The second example showcases the efficiency of Method 2, where we iterate through all defined columns to calculate the mode for each one. This provides a comprehensive statistical summary in a single computation, minimizing boilerplate code.

By using the concise list comprehension structure, we avoid repetitive code blocks, achieving high readability and maintainability, which is crucial in large-scale data processing workflows where hundreds of columns may exist.

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

, , ,]

The resulting output is a list detailing the mode for every field in the DataFrame. This structural output is highly useful for automated data reporting or passing results to subsequent functions within a larger data pipeline.

The output clearly summarizes the central tendency for each variable:

The mode of the **team** column is 'A' (Appearing three times).
The mode of the **conference** column is 'East' (Appearing three times).
The mode of the **points** column is 6 (Appearing twice).
The mode of the **assists** column is 4 (Appearing twice).

Understanding the Underlying PySpark Aggregation Logic

In both methodological approaches, the core mechanism relies entirely on the combination of `groupBy` and `count` functions. These functions are the distributed computing engine's way of creating a frequency table across partitions. The `groupBy` operation partitions the data based on unique values in the target column, and `count` tallies the number of records in each partition, resulting in a temporary DataFrame with two columns: the unique value and its frequency count.

The subsequent steps, `orderBy('count', ascending=False)` and `first()`, serve to isolate the single value with the highest frequency. Ordering ensures the highest count is at the top of the

sorted result set. The `first()` action retrieves that top row (which is a `Row` object), and finally, indexing with extracts only the column value itself (the mode), cleanly discarding the associated count for a concise result.

This specific sequence is highly optimized for performance in a distributed environment, ensuring that the necessary shuffles and aggregations are managed efficiently across the cluster nodes before the final result is materialized.

Handling Edge Cases: Multimodal Data and Null Values

A critical consideration when calculating the mode is handling situations where two or more values share the exact same highest frequency--known as a **multimodal distribution**. Because Spark's execution plan relies on deterministic ordering (often ASCII/lexicographical sorting when counts are equal), the standard method presented (using `orderBy` and `first()`) will always select the value that appears first based on that internal tie-breaker rule. This selection, while consistent for a single query execution, only returns one mode.

If all modes are required, a slightly more complex query involving a **Window function** or filtering based on the maximum count would be necessary to capture all tied values. However, for standard imputation or summary statistics, returning a single mode is usually considered acceptable. Regarding **null values**, the aggregation method inherently ignores them. When you group by a column, only non-null unique values form groups, ensuring the calculated mode accurately reflects the central tendency of the observed, valid data points, aligning with standard statistical definitions of the mode.

Summary of PySpark Mode Calculation Techniques

Calculating the mode in PySpark is achieved not through a dedicated function, but through a powerful sequence of distributed transformations that mimic the functionality of a frequency calculation. Whether focusing on a single column using the `groupBy().count().orderBy().first()` pipeline or iterating across all columns using list comprehension for bulk calculations, these methods ensure accurate and scalable results across massive datasets.

These fundamental aggregation techniques are essential building blocks for tackling complex analytical challenges within the Spark ecosystem, providing necessary statistical insights into data distribution and central tendency. Mastering this pattern allows data engineers to efficiently prepare data for subsequent machine learning models or reporting dashboards.

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