How to Count Values in a PySpark Column with a Condition

Authored by

stats writer

February 7, 2026

Analyzing large datasets efficiently is the core strength of **PySpark**. A frequent requirement in data processing workflows involves determining the frequency of specific data points--that is, counting the number of rows that satisfy one or more criteria within a particular column. This operation, often trivial in smaller datasets, requires specialized distributed computing techniques when dealing with Petabytes of information.

To accurately and efficiently count the number of values in a column of a DataFrame while applying specific conditional logic, the primary method in PySpark leverages the powerful combination of the filter function and the count function. The **filter** transformation restricts the dataset to only include rows meeting the desired criteria, while the **count** action returns the total number of resulting rows. While the `agg` function is powerful for general aggregation, using `filter().count()` provides the most direct and idiomatic solution for this specific task, ensuring you efficiently count the values in a column while also considering a specific condition.

PySpark: Efficiently Counting Values in a Column Based on Conditions

Defining the PySpark Methodology for Conditional Counting

When working with a PySpark DataFrame, there are two primary patterns for applying conditional counts. Both rely on the efficient parallel processing capabilities of Spark. The choice between methods usually depends on the complexity of the criteria: whether you are filtering based on a single specific value or testing against a list of acceptable values. These methods are foundational for advanced data analysis and preparation, ensuring that counts are derived accurately across all partitions of the distributed dataset. Understanding the nuances of the `filter` syntax is crucial for writing clean and performant Spark code.

The core principle involves minimizing data shuffling and leveraging Spark's optimized predicate pushdown. By placing the conditional logic within the filter function, we ensure that only the necessary rows are processed before the final **count** action is executed. Below, we detail the implementation syntax for the two most common conditional counting requirements.

Method 1: Counting Values That Meet a Single, Explicit Condition

This is the simplest form of conditional counting. It uses standard comparison operators (such as equality `==`, greater than `>`, or less than `<`) within the filter function. The expression `df.column_name == 'value'` evaluates to a boolean column, which `filter` then uses to keep only the matching rows. This method is highly recommended when dealing with exact matches or simple threshold checks against a column's contents.

The syntax below illustrates how to count records where a specific column value matches 'C'. Note the idiomatic use of direct column access `df.team` for comparison within the filter transformation.

#count values in 'team' column that are equal to 'C'
df.filter(df.team == 'C').count()

Method 2: Counting Values That Meet One of Several Conditions (Using `isin`)

When the requirement is to count rows where a column's value matches any element within a defined list (an 'OR' condition), using the `isin()` method is significantly cleaner and more efficient than chaining multiple `|` (OR) operators. The `isin()` method requires the column reference to be explicitly imported using the pyspark.sql.functions module, specifically the `col` function, although DataFrame column access via bracket notation (`df`) can also often be used. This approach allows for checking multiple membership criteria simultaneously, which is highly optimized within the Spark engine for distribution.

This pattern is crucial for data validation and segment analysis where records need to be grouped based on belonging to a predefined set of categories or codes. Using `isin()` ensures the conditional logic remains readable even when the membership list contains dozens of items.

from pyspark.sql.functions importcol#count values in 'team' column that are equal to 'A' or 'D'
df.filter(col('team').isin()).count()

Establishing the PySpark Environment and Sample Data

To demonstrate the utility and syntax of the conditional counting methods described above, we will first instantiate a **SparkSession** and create a sample DataFrame. This dataset models fictional information regarding various basketball players, including their team designation, conference affiliation, and points scored. The code snippet below details the necessary steps for initialization and data population, resulting in a structured DataFrame ready for analysis.

This initial setup is standard practice in PySpark, allowing us to define a local execution environment for demonstration purposes. The resultant DataFrame, `df`, provides a clear structure, allowing us to focus specifically on conditional counts applied to categorical columns like `team` and `conference`, and potentially numerical columns like `points`.

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|
+----+----------+------+
| A| East| 11|
| A| East| 8|
| A| East| 10|
| B| West| 6|
| B| West| 6|
| C| East| 5|
| C| East| 15|
| C| West| 31|
| D| West| 24|
+----+----------+------+
```


Example 1: Applying Conditional Counting for a Single Criterion

In this first scenario, we aim to calculate the total number of players associated with Team 'C'. This task requires using the fundamental approach involving the filter function followed immediately by the count function. The filtering logic `df.team == 'C'` creates a temporary, filtered DataFrame containing only the rows where the team column satisfies the equality condition.

It is important to recognize the two distinct parts of this chain: `filter()` is a **transformation**, meaning it lazily defines how the data should be reshaped, while `count()` is an **action**, which triggers the actual computation across the Spark cluster and returns the result to the driver program. This structure ensures minimal data movement and efficient processing of the conditional logic.

#count values in 'team' column that are equal to 'C'
df.filter(df.team == 'C').count()

3

Upon execution, the result confirms that a total of **3** records within the `team` column meet the specified criterion of being equal to 'C'. This method is highly effective for isolating and quantifying specific occurrences within any categorical column, serving as the backbone for basic data aggregation tasks in PySpark.

Example 2: Counting Based on Membership in a Set of Values

When the condition requires checking if a column's value belongs to a list of potential matches, the `isin()` method provides a concise and readable solution, superior to chaining multiple `|` (OR) conditions. For this demonstration, we seek to count the total number of players belonging to either Team 'A' or Team 'D'.

To use `isin()`, we must leverage functions available within pyspark.sql.functions. We import `col` to reference the column object explicitly and then apply the `isin()` method, passing the list of desired values () as the argument. This approach scales gracefully when the list of required values grows large, maintaining code clarity and computational speed.

from pyspark.sql.functions importcol#count values in 'team' column that are equal to 'A' or 'D'
df.filter(col('team').isin()).count()

4

The resulting output of **4** indicates that there are four rows where the `team` value is either 'A' or 'D'. This showcases the efficiency of the `isin()` operator for set membership testing across distributed DataFrame columns, and is a key technique for simplifying complex boolean logic.

Applying Conditions on Numerical Columns

The conditional counting techniques demonstrated so far primarily focused on categorical string data (the `team` column). However, the same principles apply seamlessly to numerical data, such as the `points` column in our example DataFrame. Instead of equality checks against strings, we utilize relational operators like greater than (>), less than (<), or ranges (e.g., `>= 10 & df.points <= 20`).

For example, if a business requirement dictates counting all players who scored more than 10 points, the syntax remains straightforward, leveraging the implicit column reference:

#count players who scored more than 10 points
df.filter(df.points > 10).count()

4

This illustrates the versatility of the filter function. When dealing with complex numerical filtering, such as excluding nulls or handling floating-point precision, ensuring data quality checks are performed before the count operation is a critical best practice in **PySpark** processing pipelines. This approach is fundamental for segmenting data based on quantitative measures.

Advanced Technique: Using `sum` and `when` for Multiple Counts

While the `filter().count()` method is ideal for finding a single conditional count, sometimes analysts need to calculate counts for several different, potentially mutually exclusive, conditions simultaneously within a single operation. This can be achieved efficiently using the combination of `pyspark.sql.functions.when()` and `pyspark.sql.functions.sum()`.

The `when()` function evaluates a condition and returns a specified value if true (typically 1) and another value if false (typically 0). Since Spark treats boolean values as 1 (True) and 0 (False) implicitly during summation, summing these conditional values provides the total count for that condition. This technique avoids multiple scans of the DataFrame, which enhances performance, particularly on massive datasets.

For instance, one could simultaneously calculate the count of players in the 'East' conference and the count of players who scored less than 10 points using a single aggregate call. This method is preferred when generating summary tables that require various statistics derived from the same source data.

Summary of PySpark Conditional Counting Best Practices

Conditional counting is a fundamental operation in data analysis using **PySpark**. By mastering the usage of the filter function in conjunction with the `count()` action, users can derive precise insights

from large-scale data quickly and reliably. Whether applying a single condition using comparison operators or handling complex set memberships using the `isin()` function, PySpark provides the tools necessary for distributed conditional aggregation.

Developers are encouraged to use `isin()` over complex chained OR statements, and to explore alternative aggregation techniques like `sum(when())` for generating multiple distinct counts in a single pass. For detailed technical specifications, always refer to the official documentation. The powerful features within pyspark.sql.functions offer numerous additional ways to construct complex, highly optimized conditional logic for all your big data processing needs.