How to Conditionally Sum a Column in PySpark

Authored by

stats writer

February 6, 2026

Working with large datasets often requires complex analysis, and one of the most common requirements is performing conditional aggregations. In the distributed computing environment of PySpark, summing a column based on specific criteria is a fundamental skill. While standard SQL queries handle this using `CASE WHEN` statements, PySpark offers highly efficient programmatic methods utilizing built-in functions.

This tutorial focuses on achieving conditional sums within a DataFrame using the powerful combination of the `filter` transformation and the `sum` aggregation function. We will explore three distinct scenarios, ranging from simple single-condition filtering to complex logic involving multiple criteria linked by logical operators like AND (`&`) and OR (`|`). Understanding these methods is crucial for anyone performing advanced data manipulation and reporting using PySpark.

PySpark: Sum Column Based on a Condition

To efficiently calculate the sum of values in a specific column of a PySpark DataFrame that satisfy one or more conditions, developers commonly employ the following structured approaches:

Method 1: Applying Summation Based on a Single Criterion

The simplest form of conditional summing involves isolating rows that meet one specific criterion before applying the aggregation. This is achieved using the `filter` transformation, which acts similar to the `WHERE` clause in SQL. The `filter` operation reduces the DataFrame to only the relevant records, and then the `sum` function is applied via `agg` (aggregate) on the targeted column. This technique is highly optimized for filtering large volumes of data distributed across a cluster.

This method provides a clean and readable way to perform basic conditional aggregates. We import the required `sum` function from `pyspark.sql.functions` and chain the operations: first filtering the DataFrame, then aggregating the specified column. The result is typically extracted using `.collect()` for immediate use in Python, especially when the resulting aggregation is a single scalar value.

from pyspark.sql.functions import sum

#sum values in points column for rows where team column is 'B'
df.filter(df.team=='B').agg(sum('points')).collect()

Method 2: Handling Multiple Conditions with Logical AND (&)

Often, data analysis requires imposing stricter limits, where multiple criteria must be satisfied

simultaneously for a row to be included in the sum calculation. In PySpark, when using the `filter` method on column expressions, we must use the logical AND operator, represented by the ampersand symbol (`&`), to combine conditions. Crucially, each condition must be enclosed in parentheses to ensure correct operator precedence, as standard Python boolean operators (like `and`) are not overloaded for use with PySpark Column types.

This approach allows for highly targeted conditional aggregations. For instance, calculating the total points scored only by 'Guards' on 'Team B' requires both conditions to be true. The logical AND operator ensures that only the intersection of rows satisfying both criteria is passed through to the final aggregation step, resulting in a precise measurement.

from pyspark.sql.functions import sum

#sum values in points column for rows where team is 'B' and position is 'Guard'
df.filter((df.team=='B') & (df.position=='Guard')).agg(sum('points')).collect()

Method 3: Combining Conditions with Logical OR (|)

In contrast to the restrictive nature of the AND operator, data analysts sometimes need to calculate totals based on records meeting any one of several specified criteria. This is achieved using the logical OR operator, represented by the pipe symbol (`|`), within the `filter` clause. Like the AND operation, it is essential to wrap each individual condition in parentheses to maintain the integrity of the expression parsing within the PySpark engine.

Using the OR operator expands the scope of the aggregation, including rows that satisfy Condition A, Condition B, or both. This is useful for grouping data from disparate categories that share a common analytical interest. For example, summing points scored by all players on 'Team B' OR all players who are 'Guards', regardless of their team. This results in a comprehensive summation across multiple distinct groups.

from pyspark.sql.functions import sum

#sum values in points column for rows where team is 'B' or position is 'Guard'
df.filter((df.team=='B') | (df.position=='Guard')).agg(sum('points')).collect()

Setting Up the PySpark Environment and Sample Data

To demonstrate these methods practically, we must first initialize a **SparkSession** and create a sample DataFrame. This DataFrame models basketball player statistics, containing columns for

`team`, `position`, and `points`. This setup is crucial for ensuring the subsequent code examples execute correctly and produce verifiable results. The use of `SparkSession.builder.getOrCreate()` ensures that we either reuse an existing session or start a new one if necessary.

The data defined below provides a balanced mix of records spanning two teams ('A' and 'B') and two positions ('Guard' and 'Forward'), allowing us to test all three conditional scenarios effectively. The creation of the DataFrame requires defining the raw data structure and assigning appropriate column names, making the data readily accessible for manipulation and aggregation using the functions detailed previously.

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

Practical Demonstration: Summing Based on Team 'B' (Example 1)

In our first practical example, we utilize Method 1 to calculate the total points scored exclusively by players on **Team B**. This straightforward aggregation demonstrates the efficiency of using the `filter` operation to quickly subset the data. We specify the condition `df.team == 'B'`, which instructs PySpark to only retain rows where the value in the `team` column matches 'B'.

Once the DataFrame is filtered, the `agg` function applies the `sum` aggregation specifically to the `points` column. The output is a new, single-row DataFrame containing the result, which we extract using `.collect()` for display.

from pyspark.sql.functions import sum

#sum values in points column for rows where team column is 'B'
df.filter(df.team=='B').agg(sum('points')).collect()

48

By reviewing the original DataFrame, we can manually verify this result: Team B players scored 14 + 14 + 13 + 7, totaling **48** points. This confirms the correct application of the single-condition filter and subsequent summation.

Practical Demonstration: Refining Aggregation with AND (Example 2)

Example 2 requires a more granular calculation: summing the points only for players who are both on **Team B** *and* play the **Guard** position. This is where Method 2, utilizing the logical AND operator (`&`), becomes necessary. The syntax ensures that a row must satisfy two simultaneous criteria: `df.team == 'B'` and `df.position == 'Guard'`.

This highly specific filter is enclosed within the `df.filter()` clause, requiring careful use of parentheses around each condition. The resulting subset is much smaller, focusing exclusively on the intersection of the two defined groups. This illustrates the power of combining conditions for precise analytical insights within conditional aggregations.

from pyspark.sql.functions import sum

#sum values in points column for rows where team is 'B' and position is 'Guard'
df.filter((df.team=='B') & (df.position=='Guard')).agg(sum('points')).collect()

28

The resulting sum is **28**. Checking the data confirms that Team B Guards scored 14 points and 14 points (14 + 14 = 28). This demonstrates effective use of the `&` operator to narrow the scope of the summation.

Practical Demonstration: Broadening Selection with OR (Example 3)

Our final example showcases Method 3, which uses the logical OR operator (`|`) to include points from players who are either on **Team B** *or* hold the **Guard** position. This expands the selection pool compared to the previous two examples, as a player only needs to meet one of the two requirements to be included in the sum.

The filter condition `(df.team == 'B') | (df.position == 'Guard')` includes three distinct groups of players: (1) all players on Team B, (2) all Guards on Team A, and (3) all Guards on Team B (the overlap). This type of wide selection is often required when calculating totals that span overlapping, yet related, categories within the data.

from pyspark.sql.functions import sum

#sum values in points column for rows where team is 'B' or position is 'Guard'
df.filter((df.team=='B') | (df.position=='Guard')).agg(sum('points')).collect()

67

The calculated total is **67**. Let's verify this using the original data:

Team B players: 14 + 14 + 13 + 7 = 48 points.
Team A Guards: 11 + 8 = 19 points.
Total sum (Team B OR Guard): 48 + 19 = **67** points.

The use of the `|` operator successfully aggregated all points from these combined subsets, proving its effectiveness for broad, inclusive conditional summing.

Summary of Techniques and Alternative Approaches

The combination of the `filter` transformation and the `sum` aggregation provides the most idiomatic and often the most performant way to achieve conditional summation in PySpark when only a few specific criteria are needed. Because `filter` reduces the dataset size early in the execution plan, it often minimizes the data shuffled across the cluster, improving efficiency.

While this article focused on the `filter().agg(sum())` approach, it is worth noting that conditional summing can also be achieved using the `when()` function within the `agg()` block, particularly useful for summing based on multiple distinct conditions within a single aggregation step, or when incorporating the sum into a larger group-by operation. However, for simple row selection followed by aggregation, the `filter` method remains preferred for its clarity and performance characteristics.

Mastery of these fundamental conditional aggregations is essential for performing sophisticated data preparation and reporting tasks in a distributed environment. Choosing the correct logical operator (AND or OR) and understanding the need for parentheses are key aspects of writing reliable and efficient PySpark code.

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