Calculate a Cumulative Sum in PySpark

Authored by

stats writer

November 16, 2025

Introduction to Cumulative Sums in Data Analytics using PySpark

Calculating a cumulative sum, or running total, is a fundamental operation in time-series analysis, financial modeling, and business intelligence reporting. This aggregation allows analysts to track the total value accumulated up to a specific point in a sequence, which is essential for monitoring metrics like total sales over a quarter or inventory used over a month. When dealing with the massive datasets typical in modern data pipelines, highly scalable frameworks like PySpark are necessary to perform these operations efficiently across distributed clusters.

In PySpark, cumulative sums are calculated using advanced analytical functions known as Window Functions. These functions operate on a defined group of rows (the "window") related to the current row, enabling complex statistical calculations without requiring expensive data shuffling operations like traditional

GROUP BY

clauses often demand. The key to successful running total calculation lies in correctly defining the window specification, which includes ordering the data and setting precise boundaries for the aggregation frame.

This guide introduces the two primary methods for generating a cumulative sum on a PySpark DataFrame. The first method covers simple accumulation across the entire dataset, while the second demonstrates how to partition the data to calculate separate running totals for different categorical groups. Understanding the nuances of the window frame definition--specifically the use of

unboundedPreceding

--is critical for mastering this technique.

Method 1: Calculating Cumulative Sum of One Column (Non-Partitioned)

The simplest implementation of a running total calculates the sum sequentially across all rows in the DataFrame. This method is used when the data naturally belongs to a single sequence (e.g., global daily website traffic). The essential components of the Window definition here are the ordering clause and the frame boundary definition.

We rely on

Window.orderBy('day')

to establish the chronological order necessary for accumulation. Crucially, the frame is defined using

.rowsBetween(Window.unboundedPreceding, 0)

. This parameter setup specifies an expanding window:

unboundedPreceding

ensures the calculation starts at the first row of the window, and

0

indicates that the summation ends at the current row. As the function iterates through the rows, the window expands, thus achieving the running total effect.

The following code snippet demonstrates how to import the necessary classes, define the non-partitioned window specification, and apply the

F.sum()

function over this defined window to create a new column containing the cumulative sales total.

```
from pyspark.sql import Window
from pyspark.sql import functions as F

#define window for calculating cumulative sum
my_window = (Window.orderBy('day')
.rowsBetween(Window.unboundedPreceding, 0))

#create new DataFrame that contains cumulative sales column
df_new = df.withColumn('cum_sales', F.sum('sales').over(my_window))
```

Method 2: Calculating Cumulative Sum Grouped by Another Column (Partitioned)

In real-world data analysis, running totals often need to be calculated independently for subsets of the data--a scenario perfectly suited for partitioning. If, for example, we are tracking sales across multiple distinct stores, the cumulative count must restart for each store to provide accurate localized figures. This is achieved by introducing the

partitionBy()

clause.

The

Window.partitionBy('store')

command tells PySpark to logically segment the DataFrame based on the values in the 'store' column. Within each resulting partition, the cumulative calculation proceeds independently. Every time the processor crosses a boundary between stores, the cumulative sum automatically resets to zero, starting the count anew for the next group.

As with Method 1, the crucial

orderBy('day')

clause is required to ensure that the summation within each store partition happens chronologically. The window frame (

rowsBetween

) remains the same, maintaining the expanding cumulative behavior within the confines of the partition.

```
from pyspark.sql import Window
from pyspark.sql import functions as F

#define window for calculating cumulative sum
my_window = (Window.partitionBy('store').orderBy('day')
.rowsBetween(Window.unboundedPreceding, 0))

#create new DataFrame that contains cumulative sales, grouped by store
df_new = df.withColumn('cum_sales', F.sum('sales').over(my_window))
```

Example 1: Step-by-Step Simple Cumulative Sum Implementation

To solidify the non-partitioned approach, let us work through a practical example using simulated daily sales data from a single location over ten days. This is a common requirement for generating basic performance metrics over a continuous time period.

First, we generate the sample DataFrame. This step involves initializing the

SparkSession

, defining the data array, specifying column names ('day' and 'sales'), and then creating and displaying the resulting distributed dataset.

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()

+---+-----+
|day|sales|
+---+-----+
1	11
2	8
3	4
4	5
5	5
6	8
7	7

8	7
9	6
10	4
+---+-----+

Next, we apply the Window function definition (Method 1) to calculate the running total. Notice how the

orderBy('day')

establishes the sequence, ensuring that the sales are summed in ascending order of the day index.

from pyspark.sql import Window
from pyspark.sql import functions as F

#define window for calculating cumulative sum
my_window = (Window.orderBy('day')
.rowsBetween(Window.unboundedPreceding, 0))

#create new DataFrame that contains cumulative sales column
df_new = df.withColumn('cum_sales', F.sum('sales').over(my_window))

#view new DataFrame
df_new.show()

+---+-----+---------+
|day|sales|cum_sales|
+---+-----+---------+
1	11	11
2	8	19
3	4	23
4	5	28
5	5	33
6	8	41
7	7	48
8	7	55
9	6	61
10	4	65
+---+-----+---------+

The resultant DataFrame includes the new

cum_sales

column. By inspecting the output, we confirm the calculation logic: the value for Day 5 (33) is the sum of all sales from Day 1 through Day 5. Since no partition was defined, the running total spans the entire dataset, ending with the grand total of 65 on Day 10.

Example 2: Implementing a Grouped Cumulative Sum with Partitioning

This example demonstrates how to use Method 2 when the data requires segmented running totals. We use sales data spanning two different stores, 'A' and 'B', where the cumulative sum must restart when the store identifier changes, reflecting the power of partitioning.

The initial data setup is similar to Example 1, but we introduce the categorical 'store' column. This column will serve as the partition key, defining where the cumulative calculation must begin and end.

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()

```
+-----+---+-----+
|store|day|sales|
+-----+---+-----+
| A| 1| 11|
| A| 2| 8|
| A| 3| 4|
| A| 4| 5|
| A| 5| 5|
| B| 6| 8|
| B| 7| 7|
| B| 8| 7|
| B| 9| 6|
| B| 10| 4|
+-----+---+-----+
```

We now execute the cumulative sum using the partitioned window. The key distinction is the inclusion of

Window.partitionBy('store')

alongside the

orderBy('day')

clause. This ensures that the cumulative calculation is performed locally within the scope of each store.

```
from pyspark.sql import Window
from pyspark.sql import functions as F

#define window for calculating cumulative sum
my_window = (Window.partitionBy('store').orderBy('day')
.rowsBetween(Window.unboundedPreceding, 0))

#create new DataFrame that contains cumulative sales, grouped by store
df_new = df.withColumn('cum_sales', F.sum('sales').over(my_window))

#view new DataFrame
df_new.show()
```

```
+-----+---+-----+---------+
|store|day|sales|cum_sales|
+-----+---+-----+---------+
| A| 1| 11| 11|
| A| 2| 8| 19|
| A| 3| 4| 23|
| A| 4| 5| 28|
| A| 5| 5| 33|
| B| 6| 8| 8|
| B| 7| 7| 15|
| B| 8| 7| 22|
| B| 9| 6| 28|
| B| 10| 4| 32|
+-----+---+-----+---------+
```

The output confirms the success of the partitioning strategy. While Store A's cumulative sales ends at 33 (Day 5), the calculation for Store B begins fresh at Day 6, resetting the cumulative sum to 8 (the sales value for Day 6), and then continues to accumulate up to 32. This method is fundamental for sophisticated cross-sectional analysis in large-scale data systems.

Conclusion and Key Takeaways for PySpark Aggregations

The calculation of a cumulative sum is highly efficient in PySpark when utilizing the Window Functions framework. Whether you need a simple global running total or complex intra-group aggregations, the process remains consistent: define the window, ensure proper ordering, and select the correct frame boundaries.

Always remember these critical design principles when implementing cumulative calculations:

Order is Non-Negotiable: Without

orderBy()

, the cumulative result is mathematically meaningless as Spark cannot guarantee the sequence of summation across nodes.

Choose the Right Frame: For true cumulative sums (running totals), the frame definition must be

rowsBetween(Window.unboundedPreceding, 0)

. Using other boundaries, such as

rangeBetween

or fixed offsets, will yield different analytical results (e.g., sliding window averages).

Partitioning for Isolation: If the running count must reset based on a category, the

partitionBy()

clause is the necessary mechanism to isolate these segments within the distributed computation environment.

By mastering these techniques, developers can leverage the distributed power of PySpark to generate powerful analytical features with optimal performance and reliable accuracy.