How to Calculate Cumulative Sums in PySpark Easily

Authored by

stats writer

February 8, 2026

PySpark is a robust and highly scalable open-source framework specifically designed for managing and processing big data environments using the Python programming language. As the Python API for PySpark, it provides data scientists and engineers with a powerful toolset, including various built-in functions and methods, that enable efficient manipulation, transformation, and complex analysis of massive datasets. One particularly vital analytical operation required in time-series analysis, financial calculations, and performance tracking is the calculation of a cumulative sum, also frequently referred to as a running total.

The cumulative sum function calculates the sum of all preceding values up to the current row in a dataset, providing a dynamic view of accumulation over a defined sequence. In PySpark, this is elegantly achieved using specialized Window functions. These functions operate on a defined group of rows related to the current row, allowing for powerful aggregate calculations without necessitating a full data shuffle across the cluster, which is common in standard grouping operations. This article will serve as a comprehensive guide, detailing the methods and necessary code implementation for calculating cumulative sums efficiently in a PySpark DataFrame, covering both simple running totals and partitioned calculations.

Calculate a Cumulative Sum in PySpark

The Role of Window Functions in Cumulative Calculations

To successfully calculate a cumulative sum in PySpark, we must utilize the powerful framework provided by Window functions. Unlike standard aggregation functions (like `sum()` or `avg()`) which collapse rows, Window functions return a value for every input row. They require the definition of a "window"--a set of rows over which the computation is performed. Defining this window involves three key components: partitioning, ordering, and framing.

For cumulative operations, the definition of the window frame is critical. We use `Window.orderBy()` to ensure the rows are processed in the correct sequence, typically by a date or sequential identifier like a day number. Furthermore, we define the boundaries of the calculation using `rowsBetween(Window.unboundedPreceding, 0)`. This specific frame tells Spark to include all rows from the very start of the partition (or dataset, if unpartitioned) up to and including the current row (represented by `0`). This structure is essential for generating an accurate cumulative sum.

Method 1: Calculating Cumulative Sum Across the Entire DataFrame

The simplest application of the cumulative sum is calculating a running total across the entire dataset, without breaking the calculation based on categories. This method is ideal when analyzing aggregate performance over time, such as tracking total company sales or website traffic since

inception. The definition of the window only requires an ordering column to establish the sequence of accumulation.

We import the necessary classes, `Window` and `functions as F`, from `pyspark.sql`. The window specification `my_window` uses `Window.orderBy('day')` to set the sequence. The `rowsBetween` clause ensures that for any given row, the summation includes all previous rows in the ordered sequence. Finally, the calculation is applied using the `F.sum('sales').over(my_window)` expression within the `withColumn` transformation, creating a new column with the running total.

You can use the following methods to calculate a cumulative sum in a PySpark DataFrame:

Implementation Structure for Method 1: Unpartitioned Sum

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

# Define window for calculating cumulative sum across all rows, ordered by 'day'
my_window = (Window.orderBy('day')
.rowsBetween(Window.unboundedPreceding, 0))

# Create new DataFrame that contains cumulative sales column by applying the window function
df_new = df.withColumn('cum_sales', F.sum('sales').over(my_window))
```

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

In real-world data analysis, it is frequently necessary to calculate running totals independently for subsets of the data. For instance, calculating cumulative sales separately for each distinct store, region, or product category. This requires the use of the `Window.partitionBy()` clause. The inclusion of `partitionBy` effectively segments the data into logical groups, and the cumulative calculation resets to zero (or the first value) every time a new partition begins.

This approach maintains the efficiency of Window functions while providing the analytical depth of grouped aggregates. Within each partition, the `orderBy` clause maintains the correct sequence, and the `rowsBetween` clause ensures the running total accumulates only within that specific partition. This transformation is exceptionally useful for comparative analysis between different groups over time.

Implementation Structure for Method 2: Partitioned Sum

```
from pyspark.sql import Window
```

```
from pyspark.sql import functions as F

# Define window for calculating cumulative sum, partitioned (grouped) by 'store' and ordered by 'day'
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))
```

The difference between Method 1 and Method 2 lies entirely in the inclusion of `Window.partitionBy('store')`. This small addition dictates a fundamental shift in how the cumulative sum is computed across the cluster, ensuring that the running total is local to the specified grouping key.

Example 1: Practical Implementation of Unpartitioned Cumulative Sum

To demonstrate the functionality of Method 1, let us initialize a PySpark DataFrame containing sequential daily sales data. We must first establish a Spark context using a SparkSession, which is the entry point for all Spark functionality. The data represents sales figures over ten consecutive days at a single grocery store.

The initial step involves defining the data structure and creating the DataFrame. Notice that since we are calculating an unpartitioned cumulative sum, we only need the sequential index (`day`) and the value to be summed (`sales`). This preparation ensures the data is correctly ingested into the distributed computing environment of PySpark before applying the complex window operation.

Suppose we have the following PySpark DataFrame that contains information about the sales made during 10 consecutive days at some grocery store:

```
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
+---+-----+

We can use the following syntax to calculate the cumulative sum of the values in the **sales** column, demonstrating the running total calculation across all rows:

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

# Define window for calculating cumulative sum, ordered by the 'day' column
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 resulting DataFrame now contains a new column named **cum_sales**. Observe how the value in this column for any given day is the sum of the current day's sales and all preceding sales values. For example, on Day 2, the cumulative sales value (19) is the sum of Day 1 sales (11) and Day 2 sales (8). This confirms the correct implementation of the unpartitioned running total.

Example 2: Practical Implementation of Partitioned Cumulative Sum

For our second example, we introduce a categorical column: **store**. We now have sales data distributed across two different grocery stores, 'A' and 'B'. The goal is to calculate the running total of sales for store 'A' independently, and likewise for store 'B', resetting the count when the store identifier changes. This requires partitioning the data by the **store** column before ordering by **day** and calculating the cumulative sum.

Initializing this data structure requires defining three attributes: the store identifier, the sequential day, and the sales amount. This scenario perfectly illustrates the power of `Window.partitionBy()` in handling complex, group-specific calculations efficiently across a distributed system like PySpark.

Suppose we have the following PySpark DataFrame that contains information about the sales made during 5 consecutive days at two different grocery stores:

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 can use the following syntax to calculate the cumulative sum of the values in the **sales** column, ensuring the running total is calculated independently within each **store** partition:

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

# Define window for calculating cumulative sum, partitioned by 'store'
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|
+-----+---+-----+---------+
```

Interpreting the Partitioned Results

A close examination of the resulting DataFrame highlights the effect of the `partitionBy('store')` clause. When the calculation reaches the transition point from Store A to Store B (Day 6), the **cum_sales** value resets. For Store A, the running total peaks at 33. For Store B, the cumulative calculation begins anew with Day 6 sales (8), and the running total progresses independently, ending at 32. This partitioning is vital for business intelligence applications where group-specific metrics are required.

Understanding the syntax of Window functions--specifically the combination of partitioning, ordering, and the frame definition (`rowsBetween`)--allows for precise control over complex analytical calculations performed on massive scale data using PySpark.

Further Applications and Considerations

While this guide focused on the simple cumulative sum using `F.sum()`, Window functions are versatile and can be used for many other related calculations, such as cumulative average, calculating running maximums, or determining moving averages. The core principles of defining the window (partitioning and ordering) remain constant, only the aggregate function changes (e.g., `F.avg()` instead of `F.sum()`).

Furthermore, while we used `Window.unboundedPreceding` to start the sum from the beginning of the partition, the frame definition can be customized. For instance, calculating a 3-day rolling sum would involve changing the frame to `rowsBetween(-2, 0)`, which instructs Spark to sum the values from the current row (0) and the two preceding rows (-2). Mastering these parameters is key to leveraging the full analytical power of PySpark for time-series analysis and sequential metrics.

Conclusion

Calculating a cumulative sum is a foundational requirement in many data processing tasks, especially those involving sequential or time-series data. PySpark provides an optimized and highly expressive way to perform this operation using Window functions. By mastering the distinction between unpartitioned (global) and partitioned (grouped) window specifications, analysts can efficiently derive meaningful running totals on large-scale datasets, transforming raw transactional data into actionable analytical insights.

Further Reading

For those interested in exploring advanced PySpark techniques, the following tutorials explain how to perform other common tasks:

How to calculate moving averages in PySpark.
Using ranking functions (like `row_number()`) with Window specifications.
Optimizing PySpark performance when dealing with complex joins and aggregations on big data.