Learning Cumulative Sum Calculation in PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

Understanding Cumulative Sums in Data Analysis

The calculation of a cumulative sum, frequently referred to as a running total, is a foundational operation indispensable across various analytical domains, particularly in **time-series analysis** and complex financial tracking. This metric enables analysts to accurately monitor the total accumulation of a specific measure up to any given point in a sequence or over time. For organizations dealing with large volumes of data--the definition of modern big data--relying on powerful, distributed computing frameworks such as PySpark becomes absolutely essential for achieving efficient and scalable data processing.

Within the PySpark environment, deriving these crucial running totals inside a DataFrame is primarily executed through the use of advanced Window functions. These functions are highly versatile, designed to facilitate complex aggregations that operate on a defined group of rows related to the current row without collapsing the dataset. This sophisticated mechanism permits calculations like running totals, ranking, and moving averages. We will thoroughly investigate the two primary methodologies for calculating the cumulative sum in PySpark: the **global approach**, which treats the entire dataset as a single stream, and the **partitioned approach**, which calculates running totals independently within defined groups.

The PySpark Mechanism: Mastering Window Functions

Before proceeding to specific implementation examples, it is vital to gain a deep understanding of the core component driving these calculations: the `Window` object in PySpark SQL. The Window object serves as the blueprint, explicitly defining the set of rows upon which an aggregation function (such as summation, averaging, or ranking) will operate. When our objective is to compute a cumulative sum, three distinct components of the Window specification become necessary to ensure correctness and determinism.

Ordering (`orderBy()`): This clause is strictly mandatory for any cumulative or sequential calculation. Because the running total is entirely dependent on the sequence of data points, proper ordering ensures that the results are consistent and logical. Without explicit ordering, the results generated by a distributed framework like Spark will be non-deterministic. We must specify the column that dictates the sequence of accumulation (e.g., a timestamp or a day index).

Framing (`rowsBetween()`): The framing clause defines the specific boundaries of the rows included in the calculation for the current row's output. For a true, accurate cumulative sum calculation, we must specify the frame boundaries to span from the very beginning of the partition or dataset, represented by `Window.unboundedPreceding`, up to and including the current row, which is always represented by the index `0`.

Partitioning (`partitionBy()`): This component is optional but crucial for grouped analysis. The

`partitionBy()` clause dictates the column(s) based on which the running total should reset. If this clause is omitted, the entire DataFrame is inherently treated as a single, massive group, resulting in a continuous, global cumulative total.

The actual aggregation is performed using the functions module, conventionally imported as `F`. This module provides the necessary function, typically F.sum, which is then dynamically applied over the rigorously defined window using the fluent `.over()` method. This streamlined combination of definition and application is what makes PySpark's approach highly efficient and scalable for processing large-scale data transformations.

Method 1: Implementing a Global Cumulative Sum (Ungrouped)

The global cumulative sum approach involves calculating the running total across the entirety of the DataFrame, strictly based on a designated ordering column. This method is perfectly suited for scenarios where the underlying data represents a **single, cohesive stream**, such as tracking the total daily website traffic or the consolidated sales figures for an entire organization without segregation by geographical region or product line.

The defining characteristic of this implementation is the absence of the `partitionBy()` clause in the Window definition. By omitting partitioning, we implicitly instruct PySpark to view all rows as belonging to one continuous group. The vital component remains the specification of the window frame, ensuring it spans from the start of the dataset (unbounded preceding) up to the current row, thereby accumulating all previous values into the running 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))
```

Practical Walkthrough: Global Sum Example

To effectively solidify the foundational concepts introduced in Method 1, we will now execute a complete, end-to-end example. Our goal is to calculate the global cumulative sum of sales recorded over ten consecutive days. Since we are interested in the aggregated running total across the entire period, there is no requirement for partitioning the data. The first step involves initializing a PySpark session and generating a representative sample DataFrame, which we name

`df`. This DataFrame tracks essential daily sales figures, and the inherent ordering provided by the `day` column is critical for our subsequent Window specification.

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
+---+-----+

To accurately compute the required running total, we proceed by defining a Window object named `my_window`, which explicitly specifies ordering by the `day` column. By defining the frame boundaries using `rowsBetween(Window.unboundedPreceding, 0)`, we explicitly instruct the PySpark engine to include every preceding row within the dataset up to, and including, the row currently being processed (represented by the index 0). Following the definition, we apply the F.sum aggregation function to the `sales` column, utilizing the fluent `.over(my_window)` method to perform the calculation.

```
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 resulting DataFrame, `df_new`, now features a newly generated column named `cum_sales`. A careful inspection reveals that for every row, the value recorded in `cum_sales` represents the running summation of all preceding `sales` values, meticulously including the sales figure for the current day. For illustrative purposes, on Day 4, the cumulative sum is correctly calculated as 11 +

8 + 4 + 5 = 28, unequivocally confirming the accurate functionality of the global cumulative calculation based on the defined window.

Method 2: Calculating a Partitioned Cumulative Sum (Grouped)

In the realm of complex analytical scenarios, it is frequently necessary for the cumulative sum calculation to reset based on a specific grouping variable, such as a distinct product category, geographic region, or individual store identifier. For instance, when analyzing multi-store sales data, the running total for Store A must remain entirely separate from the running total for Store B. This sophisticated, grouped accumulation is achieved in PySpark by partitioning the DataFrame using the `partitionBy()` clause within the Window function specification.

By invoking `partitionBy()`, we instruct Spark to logically divide the massive dataset into smaller, independent partitions based on the unique values in the specified column. The cumulative calculation then operates autonomously within each of these distinct partitions. Critically, the running total is guaranteed to reset every time the partitioning column's value changes, ensuring that the cumulative sum is meaningful and specific to its respective group.

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))

Practical Walkthrough: Partitioned Sum Example

To demonstrate the practical application of partitioning, let us consider an example tracking sales data across two distinct entities: 'Store A' and 'Store B'. The core requirement here is that the running total of sales must strictly restart whenever the store identifier transitions. This scenario perfectly highlights why the `partitionBy()` clause is an indispensable tool in the Window function arsenal. We initiate the process by creating an updated DataFrame structure that now includes a `store` identifier column, along with the existing `day` and `sales` columns. This structure provides the necessary grouping dimension for PySpark to accurately distribute and calculate the independent running totals.

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

The fundamental syntactical change in this example is the inclusion of the `Window.partitionBy('store')` command. This explicit instruction directs PySpark to rigorously treat all rows belonging to 'Store A' as one distinct, independent set and all rows associated with 'Store B' as a completely separate set. Within the boundaries of each partition, the cumulative

calculation proceeds sequentially, ordered strictly by the `day` column, spanning from `unboundedPreceding` up to the current row. We then apply the F.sum function to accurately compute the running total of sales specific to each respective store 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))

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

Reviewing the resulting `cum_sales` column definitively confirms that the calculation has been partitioned and executed correctly. For all records associated with Store A, the sum accumulates sequentially, concluding at a total of 33. Crucially, when the data transitions to Store B (starting on Day 6), the cumulative sum immediately resets to 8 (the sales value for Day 6), and subsequently continues accumulating only the sales figures strictly associated with Store B, ultimately concluding at 32. This example clearly illustrates the exceptional power and flexibility inherent in PySpark's Window functions when applied to complex, grouped aggregation tasks.

Summary and Further Learning

Mastering the calculation of a cumulative sum in PySpark is acknowledged as a critical, fundamental skill for any data engineer or analyst specializing in working with sequential or time-series data at scale. By gaining proficiency in utilizing the `Window` function and its essential components--specifically the defining `orderBy()`, the boundary-setting `rowsBetween()`, and the optional yet strategically vital `partitionBy()`--users are empowered to efficiently and reliably derive running totals across truly massive DataFrame objects. These specific techniques are intrinsically scalable and represent the established, industry-standard approach for performing sophisticated analytical transformations within the robust Spark environment.

We have successfully demonstrated and executed the two most powerful and widely used approaches for running totals: calculating a continuous, global cumulative sum and calculating a precise, partitioned cumulative sum that resets based on categorical variables. These two methods collectively encompass the vast majority of real-world scenarios that demand accurate running total computations.

To continue advancing your expertise in PySpark SQL and distributed data manipulation, we highly recommend exploring tutorials that cover other advanced Window functions, such as the techniques required for calculating moving averages or for efficiently ranking data points within defined partitions.

Additional Resources

The following tutorials explain how to perform other common data manipulation and analysis tasks using PySpark: