How to Calculate a Rolling Mean in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 8, 2026

Calculating a rolling mean in PySpark refers to the process of finding the average value of a given set of data points over a specific number of time periods. This is often used in time series analysis to smooth out the data, eliminate short-term fluctuations, and identify underlying trends or cycles. The rolling mean, also known as the moving average, provides a localized average that shifts forward through the dataset as new data points become available. Understanding this technique is fundamental for data engineers and analysts working with sequential data, especially within the context of big data environments where efficiency is paramount.

To accurately and efficiently calculate a rolling mean in PySpark, you leverage the powerful concept of the Window function in conjunction with the aggregation function, typically `F.avg()`. This combination allows you to define a specific partition (or window frame) over which the aggregation should be computed. By defining the window based on time or sequential order and specifying the range of rows to include (the preceding and following rows), PySpark can calculate the local average within each frame. This scalable methodology ensures accurate calculation of rolling means, even when dealing with extremely large, distributed datasets that characterize modern data processing workflows.

Calculating the Rolling Mean Efficiently in PySpark

Understanding the PySpark Window Function Concept

Before diving into the specific code implementation, it is essential to grasp the mechanism behind Window functions in Spark SQL. A Window function performs calculations across a set of table rows that are somehow related to the current row, without collapsing the rows into a single output row (unlike standard aggregation functions like `GROUP BY`). When calculating a moving average, the window defines the specific subset of rows (e.g., the preceding three days and the current day) over which the `avg()` operation will apply. This structured approach allows for sophisticated analytical calculations across partitions of data, which is crucial for handling complex time-based metrics and sequential data analysis.

The definition of a window requires three primary components: the partitioning clause (similar to `GROUP BY`, but optional for rolling metrics), the ordering clause (`orderBy`), and the frame specification (`rowsBetween` or `rangeBetween`). For calculating a standard rolling mean, the ordering clause is critical, typically based on a sequential or date column (like 'day'). The frame specification then dictates how far back (and potentially forward) the calculation should look. This meticulous control over the data frame ensures that the rolling average accurately reflects the desired lag period specified by the analyst, providing a powerful tool for smoothing noisy data.

The process leverages PySpark's optimized distributed architecture. By defining the window operation, we instruct Spark to manage the necessary data shuffling and grouping efficiently

across the cluster, ensuring that even extremely large datasets can be processed quickly without requiring excessive memory overhead on any single node. This efficiency is the core reason why the Window function approach is preferred over traditional iterative or manual aggregation methods.

Core Syntax for Calculating the Rolling Average

To execute the rolling mean calculation on a PySpark DataFrame, you must first import the necessary modules: `Window` for defining the analytical window and `functions` (aliased as `F`) for accessing aggregation methods like `avg`. The following syntax demonstrates the standard pattern used to define and apply a window function for calculating the rolling average across a time-ordered column. This structure is highly efficient as it leverages Spark's optimized execution engine for distributed computations.

The definition of the window object, here named `w`, uses `Window.orderBy('day')` to specify the strict order of calculation, which is non-negotiable for sequential metrics. Crucially, `rowsBetween(-3, 0)` defines the exact window frame: it starts three rows before the current row (`-3`, indicating a three-day look-back) and includes the current row itself (`0`). Thus, for any given row, the window encompasses four total rows, resulting in a 4-day rolling mean. Once the window is defined, the `withColumn` transformation is used to add the new metric, applying the average function (`F.avg('sales')`) over the defined window (`.over(w)`).

You can use the following syntax to calculate a rolling mean in a PySpark DataFrame:

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

# Define the window specification for calculating the rolling mean
w = (Window.orderBy('day').rowsBetween(-3, 0))

# Create a new DataFrame adding the calculated 4-day rolling mean column
df_new = df.withColumn('rolling_mean', F.avg('sales').over(w))
```

This particular implementation creates a new column named `rolling_mean` that contains the 4-day rolling average of values found in the **sales** column of the input DataFrame. Note that the window includes the current row (indicated by the `0`), which is conventional for calculating trailing moving averages used in financial or operations data analysis. The usage of `rowsBetween` provides granular control over the aggregation boundaries, ensuring precise measurement across sequential records within the ordered dataset.

Practical PySpark Example: Data Initialization

To solidify the conceptual understanding, we will now walk through a complete, practical example. Suppose we are analyzing daily sales data from a grocery store over a 10-day period. The first step in any PySpark analysis is to initialize a Spark session and prepare the raw data structure into a usable DataFrame. This initial setup is crucial for ensuring that the subsequent window calculations can correctly reference the necessary columns, such as the sequential 'day' column required for proper ordering and frame definition.

The dataset we are working with consists of two key columns: 'day', representing the sequential index from 1 to 10, and 'sales', representing the monetary value of goods sold on that specific day. This simple structure is perfect for demonstrating how the rolling average smooths out the daily variations in sales performance. We utilize the `SparkSession.builder.getOrCreate()` method to initialize the environment, ensuring that the necessary infrastructure for distributed computation is available, followed by `spark.createDataFrame` to instantiate the data structure defined by the provided list of tuples and column names.

The following code snippet demonstrates the complete initialization process, resulting in the source DataFrame used for the subsequent rolling mean calculation. Viewing the DataFrame using `df.show()` confirms the successful loading and structuring of the input data before applying any analytical transformations. This step validates the schema and ensures data quality before computational resources are dedicated to the window function execution.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define the raw data, structured as (day, sales)
data = ,
,
,
,
,
,
,
,
,
]

Define the schema column names
columns =

```
# Create the PySpark DataFrame
df = spark.createDataFrame(data, columns)

# Display the initial 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|
+---+-----+
```

Executing the PySpark Rolling Mean Transformation

With the input DataFrame successfully created and visualized, the next critical step is to apply the calculated 4-day rolling mean transformation. We reuse the `Window` definition syntax previously introduced, which specifically orders the data by the 'day' column and includes the current row along with the three preceding rows. This calculation is performed efficiently across all rows simultaneously due to Spark's parallel processing capabilities, avoiding iterative loops that are inefficient in a distributed computing paradigm.

It is important to note how PySpark handles the crucial boundary conditions--specifically, the first few rows where a full 4-day history is not yet available. On Day 1, the calculation only uses Day 1's sales (1 row); on Day 2, it uses Days 1 and 2 (2 rows); and on Day 3, it uses Days 1, 2, and 3 (3 rows). Only starting from Day 4 does the window encompass the full four rows defined by `rowsBetween(-3, 0)`. PySpark automatically manages these partial windows, calculating the average based only on the available data points within the defined frame, resulting in accurate initialization of the rolling average column instead of returning null values.

The following comprehensive code block imports the required libraries, defines the trailing window, applies the transformation using `F.avg()` over the window, and then displays the resulting

DataFrame containing the newly calculated `rolling_mean` column, illustrating the smoothed data output.

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

# Define the 4-day trailing window
w = (Window.orderBy('day').rowsBetween(-3, 0))

# Apply the average function over the defined window
df_new = df.withColumn('rolling_mean', F.avg('sales').over(w))

# View the resulting DataFrame with the rolling mean
df_new.show()
```

```
+---+-----+-----------------+
|day|sales| rolling_mean|
+---+-----+-----------------+
| 1| 11| 11.0|
| 2| 8| 9.5|
| 3| 4|7.666666666666667|
| 4| 5| 7.0|
| 5| 5| 5.5|
| 6| 8| 5.5|
| 7| 7| 6.25|
| 8| 7| 6.75|
| 9| 6| 7.0|
| 10| 4| 6.0|
+---+-----+-----------------+
```


Interpreting the Results and Manual Verification

The resulting DataFrame, `df_new`, now includes the appended column, `rolling_mean`, which clearly demonstrates the calculated moving average of the sales values over the most recent four days. The presence of this metric provides a smoother representation of sales performance compared to the raw daily figures, making it significantly easier to spot underlying trends or deviations from the smoothed baseline, which is the primary utility of applying time series analysis techniques in practical data processing.

To ensure the calculation is correct and to build confidence in the analytical output, it is highly recommended practice to manually verify a couple of data points derived from the window function.

By checking the calculation for Day 4, which is the first row where the window fully spans four data points, we can confirm the exact logic of the `rowsBetween(-3, 0)` setting.

For example, the rolling mean of values in the sales column on **Day 4** is calculated using the sales from Days 1, 2, 3, and 4:

Rolling Mean = (11 + 8 + 4 + 5) / 4 = **7**

Similarly, we can verify the calculation for the next sequential period, Day 5. This check demonstrates the "rolling" nature of the calculation, as the window shifts forward one step, dropping Day 1 and incorporating Day 5 into the four-day frame.

Rolling Mean = (8 + 4 + 5 + 5) / 4 = **5.5**

These manual calculations confirm that the Window function correctly interprets the frame specification and produces the expected trailing average, providing high confidence in the output used for downstream predictive or descriptive analytical modeling.

Customizing the Window Frame: Adjusting the Lag Period

A significant advantage of using PySpark Window functions is the inherent flexibility in defining the calculation frame. While the example calculated a 4-day rolling mean using `rowsBetween(-3, 0)`, analysts frequently need to adjust the duration of the rolling period to capture different cyclical trends--a 7-day average for weekly seasonality, a 30-day average for monthly trends, or even custom periods based on business cycles. Changing the span requires only a simple modification to the first value in the `rowsBetween` parameter.

The first value in the `rowsBetween(start, end)` function determines how many rows precede the current row. If you wish to calculate an N-day rolling average, you should set the start value to `-(N-1)`, ensuring the window size includes N total elements (N-1 preceding plus 1 current element). The end value remains `0` to ensure the current row is always included in the calculation. For instance, if you require a 5-day rolling average, the calculation must include the current day and the four preceding days (5 total rows); therefore, you would use `-4` as the start parameter.

Therefore, to calculate a 5-day rolling average, you would modify the window definition to use **rowsBetween(-4, 0)** instead. This ability to easily parameterize the window size makes PySpark an extremely versatile tool for dynamic time series analysis across varying frequencies and granularities, all without modifying the core logic of the Spark operation.

Advanced Considerations: Handling Gaps and Partitions

While the provided example used sequential day IDs, real-world data often involves irregularities

such as time gaps or requires calculations partitioned by specific categories (e.g., store ID, product type). If there are significant time gaps in the sequential data, using `rowsBetween` might incorrectly count non-sequential days as adjacent rows, which could skew time-based averages. For strictly time-based windows, where the physical elapsed time matters more than the number of recorded events, data engineers should consider using `rangeBetween`. This function calculates the frame based on value ranges (like numeric timestamps or epoch time) rather than absolute row counts, providing a time-aware window definition.

Furthermore, if the source data included sales from multiple physical stores, the rolling mean would need to be calculated independently for each store. This necessity requires introducing a partitioning clause to the window definition using `Window.partitionBy('store_id')`. This ensures that the window calculation is reset at the boundary of each partition, preventing cross-contamination of averages between different entities. This partitioning capability is what transforms a simple SQL concept into a powerful, scalable engine suitable for complex analytical tasks in distributed computing environments like PySpark.

Conclusion and Further PySpark Analytical Techniques

The calculation of a rolling mean in PySpark is a foundational skill for analysts and data scientists dealing with sequential data. By mastering the efficient application of the Window function, data professionals can efficiently generate smoothed metrics that reveal crucial trends and cyclical patterns within immense datasets. This methodology is highly extensible; the windowing framework is not limited to simple averages but can be applied to calculate rolling sums, minimums, maximums, or more complex rank-based statistics, such as percentiles or rank orderings, all within the same highly optimized paradigm.

The flexibility, scalability, and performance offered by PySpark's windowing capabilities make it the preferred choice for advanced analytical tasks in distributed computing environments. Leveraging tools like `rowsBetween` or `rangeBetween` allows for precise control over the analytical scope, driving deeper, more reliable insights from complex time-dependent data structures.

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