How to Calculate Lag by Group in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 4, 2026

Analyzing sequential dependencies within large datasets is a critical task in modern data processing. When working with distributed systems like PySpark, calculating the difference between a current row's value and a previous row's value--known as the **lag**--must be performed efficiently across partitioned groups. This calculation is indispensable for financial analysis, operational trend identification, and especially for preparing time-series data for forecasting models.

To achieve precise lag calculations in a scalable manner, PySpark leverages powerful constructs called Window functions. These functions allow analysts to perform calculations across a set of table rows that are somehow related to the current row, without collapsing the individual rows into a single aggregated output. The key advantage of using Window functions is their ability to partition the data, ensuring that the lag calculation restarts independently for each defined group.

This detailed guide explores the methodology for calculating lag by group within a DataFrame using PySpark's dedicated windowing API. By mastering the combination of partitioning, ordering, and the specific lag function, users can transform raw sequential data into features rich with historical context, significantly enhancing subsequent analytical processes and statistical modeling efforts.

Calculating Lagged Values by Group in PySpark

Understanding PySpark Window Functions for Sequential Analysis

The core mechanism enabling group-specific calculations in PySpark is the Window functions API, derived from standard SQL windowing concepts. Unlike traditional aggregate functions (like SUM or AVG) which reduce the number of output rows, window functions retain the original number of rows while adding contextual data derived from surrounding rows in the defined window. This is precisely what is needed when calculating a **lagged value**: we need to see the sales from yesterday while still retaining today's sales row.

To properly define a window, two primary clauses are necessary: Window functions require both a definition of how the data should be grouped (the partitioning) and how the rows within those groups should be sequenced (the ordering). When calculating lag, correct ordering is absolutely vital. If the data is not ordered chronologically or sequentially, the preceding row will not be the intended historical record, leading to incorrect calculations and flawed analysis.

The syntax for defining and applying a window operation in PySpark is concise yet powerful. It involves importing the necessary classes and functions, defining the window specification, and then using a function like `lag()` applied over that specification. The following structure outlines the fundamental approach required to introduce lagged features into your DataFrame efficiently.

Core Components: Partitioning and Ordering Data

Defining the window specification is the most critical step in calculating lag by group. This specification is built using the `Window` class, which requires two key method calls: `partitionBy()` and `orderBy()`. The `partitionBy()` method specifies the columns that define the distinct groups. For instance, if calculating sales lag by store, the 'store' column must be used for partitioning. This ensures that when the calculation reaches a new store, the sequence starts fresh.

The `orderBy()` method dictates the sequence within each partition. Since lag inherently relies on time or sequence, this step usually involves ordering by a date, timestamp, or sequence ID column (like 'day' in a simplified example). If `orderBy()` is omitted or incorrectly defined, the output of the lag function will be arbitrary, as Spark cannot guarantee the sequence in a distributed environment without explicit instruction.

Once the window object (`w` in common examples) is created using both partitioning and ordering definitions, it encapsulates all the rules necessary for Spark to execute the sequential calculation correctly across the distributed partitions of the cluster. This abstraction simplifies the complex task of distributed sequential processing into a single, reusable object.

Implementing the Lag Function: Syntax and Explanation

The actual calculation of the preceding value is handled by the lag function, which is imported from `pyspark.sql.functions`. The `lag()` function typically accepts three arguments: the column to lag, the offset (how many rows back to look, usually 1 for immediate lag), and an optional default value for rows where no prior value exists (which defaults to `null`).

The function must then be explicitly applied "over" the predefined window specification using the `.over(w)` method call. This tells PySpark to apply the `lag()` operation not globally across the entire DataFrame, but specifically within the boundaries set by the partition (e.g., within Store A, then within Store B) and according to the specified sequence (ordered by day).

You can use the following syntax to calculate lagged values by group in a PySpark DataFrame, which creates a new column using the `withColumn` transformation:

```
from pyspark.sql.window import Window
from pyspark.sql.functions import lag

#specify grouping and ordering variables
w = Window.partitionBy('store').orderBy('day')

#calculate lagged sales by group
df_new = df.withColumn('lagged_sales', lag(df.sales,1).over(w))
```

This particular example creates a new column called **lagged_sales** that contains the lagged values from the **sales** column in the DataFrame, grouped by the values in the **store** column and ordered by the **day** column. We are requesting an offset of `1`, meaning we look back exactly one row within that specific store's sequence.

Practical PySpark Example: Setting Up the Sales Data

To illustrate this process clearly, consider a typical business scenario involving daily sales records captured across multiple retail locations. We need to determine yesterday's sales figure for each store independently to calculate daily changes or ratios, making this a perfect use case for grouped lag calculation.

Suppose we have the following PySpark DataFrame that contains information about sales made during consecutive days at two different stores (Store A and Store B). The dataset is simple yet effective for demonstrating the segregation enforced by the partitioning clause.

The initial setup requires initializing a Spark session, defining the schema, and creating the foundational DataFrame. This step is crucial for working with any data processing task in the PySpark environment:

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()

Running the `df.show()` command confirms the structure: a sequence of days and corresponding sales values, clearly categorized by the 'store' identifier. This dataset structure is typical for sequential or time-series data analysis where temporal relationships need to be established within distinct entities.

Applying the Lag Calculation and Interpreting Results

With the source DataFrame prepared, we now apply the window specification defined earlier. We partition by `'store'` to ensure Store A's lag calculations only reference other rows from Store A, and we order by `'day'` to guarantee that the prior day is correctly identified. The lag function uses an offset of 1 to look back at the immediate previous day's sales figure.

The following code block executes the window calculation and displays the resulting DataFrame, `df_new`, which now includes the new feature column, `lagged_sales`. Careful examination of the output confirms that the lagged values transition smoothly within each store partition but reset to `null` whenever a new partition (a new store) begins.

from pyspark.sql.window import Window
from pyspark.sql.functions import lag

#specify grouping and ordering variables
w = Window.partitionBy('store').orderBy('day')

#calculate lagged sales by group
df_new = df.withColumn('lagged_sales', lag(df.sales,1).over(w))

#view new DataFrame
df_new.show()

+-----+---+-----+------------+
|store|day|sales|lagged_sales|
+-----+---+-----+------------+
A	1	18	null
A	2	33	18
A	3	12	33
A	4	15	12
A	5	19	15
B	1	24	null
B	2	28	24

B	3	40	28
B	4	24	40
B	5	13	24
+-----+---+-----+------------+

Interpreting the new column, **lagged_sales**, reveals two crucial points about Window functions. Firstly, for any row (e.g., Store A, Day 2), the `lagged_sales` value (18) correctly matches the `sales` value of the immediately preceding row in that partition (Store A, Day 1). Secondly, the first row of every partition (Store A, Day 1, and Store B, Day 1) results in a **null** value because, by definition, there is no preceding row within that partition to reference.

Advanced Handling: Dealing with Null Values and Edge Cases

The appearance of **null** values in the first row of every partition is expected behavior for the lag function when the default value parameter is not supplied. In many analytical applications, particularly when training machine learning models or calculating percentage changes, these nulls must be addressed, as they can cause errors or skew statistical results.

A common practice when dealing with sequential data where the first entry lacks historical context is to replace these nulls with a meaningful substitute, such as zero (0), or potentially the mean or median of the sales column if appropriate for the analysis. PySpark provides the `fillna` DataFrame function for this purpose, offering a straightforward method to clean up the newly created feature column.

To replace the null values in the **lagged_sales** column with zero, ensuring a complete numerical dataset for subsequent processing, the following syntax is applied. Note that this operation occurs after the window calculation has finished, treating the `lagged_sales` column as any other column in the DataFrame.

#replace null values with 0 in lagged_sales column
df_new.fillna(0, 'lagged_sales').show()

+-----+---+-----+------------+
|store|day|sales|lagged_sales|
+-----+---+-----+------------+
A	1	18	0
A	2	33	18
A	3	12	33
A	4	15	12
A	5	19	15

B	1	24	0
B	2	28	24
B	3	40	28
B	4	24	40
B	5	13	24
+-----+---+-----+------------+

The resulting DataFrame confirms that the initial null values have been successfully replaced with zero, providing a clean, complete, and highly informative dataset suitable for rigorous analysis of time-series data trends within each store group. Understanding how to define, apply, and clean the output of Window functions is foundational for mastering advanced data manipulation in PySpark.

Further Learning Resources

For further detailed information and alternative parameters, you can consult the official documentation for the PySpark **lag** function and other related window operations.

These related tutorials explain how to perform other common sequential and analytical tasks in PySpark:

Calculating Rolling Averages in PySpark

Using Lead Function for Future Projections in PySpark

Ranking Data within Groups using PySpark Window Functions