How to Calculate Lag by Group in PySpark DataFrames

Authored by

stats writer

January 19, 2026

Introduction to Calculating Lag in PySpark

Analyzing time-series data often requires referencing previous records within specific groups--a critical operation known as calculating the **lagged value**. This is a fundamental technique used widely in predictive modeling, financial forecasting, and trend analysis. When operating on large-scale datasets, the PySpark library provides the necessary tools for performing these advanced calculations efficiently across distributed clusters. To successfully compute lagged values partitioned by grouping attributes (such as 'store' ID or 'category'), we must effectively utilize PySpark's specialized **Window functions**.

The primary challenge in calculating lag across groups lies in defining the context for the calculation. Spark needs explicit instructions on how to segment the data (partitioning) and how to sequence the records within those segments (ordering). By correctly specifying the **Window specification**, we precisely dictate to Spark which previous row should be retrieved for the current record, ensuring the calculation respects the boundaries of each group and the underlying temporal order of the data points.

The following syntax illustrates the fundamental structure required to calculate lagged values by group using a PySpark DataFrame. This approach necessitates importing the `Window` class and the `lag` function from the `pyspark.sql` modules.

```
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 code snippet successfully generates a new column named **lagged_sales**. This column is populated with the lagged values from the **sales** column, with the calculations strictly segmented according to the values present in the **store** column. The `orderBy` clause ensures that the lag is calculated chronologically based on the `day` column within each store's partition.

The Role of Window Functions in Time Series Analysis

Window functions are foundational components in PySpark for performing calculations across a set of table rows that are related to the current row. Unlike standard aggregate functions (like SUM or AVG), which collapse rows into a single summary output, window functions retain the original row structure while adding a new calculated column. This capability is absolutely essential for time

series operations.

To calculate lag, we must define the computational window using the `Window.partitionBy()` and `Window.orderBy()` methods. The `partitionBy` method determines the groups across which the calculation should operate independently (e.g., separating sales data by store), while `orderBy` establishes the sequence, which is crucial for determining which row is "previous" to the current one.

If the `partitionBy` clause is omitted, the lag calculation would treat the entire DataFrame as one single group, leading to inaccurate results if the data contains multiple independent time series (such as sales data from different geographic locations or stores). Therefore, correct partitioning is mandatory when dealing with group-level time features.

Essential PySpark Functions: Lag and Window

The two core functions used here are the `Window` definition and the Lag function. The Window function acts as a scoping mechanism, defining the subset of data the lag calculation will look at. The `lag(column, offset)` function then retrieves the value of `column` that is `offset` rows before the current row within the defined window boundaries.

In most standard time series applications, the offset is set to 1, as we usually aim to retrieve the value from the immediately preceding time step (the prior day, month, or transaction). However, this offset can be adjusted to any integer value required for broader temporal analysis, such as looking at sales from the previous week (offset 7) or the previous quarter.

Using `.over(w)` links the specific analytical function (`lag`) to the previously defined **Window object** (`w`). This combination ensures that the calculation is executed efficiently and accurately according to the specified groupings and sequence across all distributed nodes utilized by PySpark.

Practical Example: Setting up the Sales Data

To demonstrate this powerful mechanism, we will first construct a sample DataFrame. This dataset simulates daily sales data collected over five consecutive days from two independent entities: Store A and Store B. This data structure precisely mirrors the scenario where partitioned lag analysis is essential, as the sales history of Store A should never influence the calculated lag for Store B.

We begin by establishing a Spark Session, defining our raw sales data, and mapping it to the appropriate column names: 'store', 'day', and 'sales'. These column names will serve as the anchors for our window specification.

The following script handles the initialization and creation of the sample 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()
```

The resulting DataFrame is structured correctly for implementing our windowing logic, where 'store' will be used for partitioning and 'day' will be used for ordering.

Implementing the Lag Calculation Across Groups

Once the sample DataFrame `df` is prepared, we apply the **Window function** logic. The sequence of operations involves importing the necessary functions, defining the window specification `w` (partitioned by `store` and ordered by `day`), and then using `withColumn` along with the Lag function to generate the new feature.

The syntax below applies the calculation, ensuring that the previous day's sales (lag 1) are retrieved strictly within the boundaries of each individual store's time series. This method is highly scalable and ensures computational accuracy even when dealing with billions of records.

```
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 Lagged Results

The output table reveals the newly created **lagged_sales** column, which accurately displays the previous day's sales for each corresponding store. For example, for Store A on Day 2 (sales of 33), the lagged sales value is 18, correctly referencing Store A's sales on Day 1. This demonstrates successful partitioning and ordering.

A key aspect of interpreting these results is observing how the calculation handles the boundary conditions. Notice that the first record within each group (Store A, Day 1 and Store B, Day 1) contains a **null** value in the **lagged_sales** column.

For the initial row of any partition, the Lag function cannot find a preceding record within that group, thus assigning **null** by default.

The reset between partitions is also evident: When the data shifts from Store A (Day 5) to Store B

(Day 1), the lagged value resets to **null**, proving that the `partitionBy('store')` clause prevented cross-contamination of time series data.

Understanding the significance of these boundary **null** values is crucial for preparing the data for subsequent modeling steps.

Handling Missing Values (NULLs) with fillna

While the presence of **null** values at the start of each partition is technically correct from a computational standpoint, they can pose challenges for analytical tasks, especially when integrating with machine learning models that expect numerical inputs. A standard data engineering practice is to impute these starting **nulls** with a meaningful placeholder. In sales analysis, replacing the initial lag value with zero (0) often makes sense, as it implies zero sales occurred before the start of the recorded period for that specific store.

PySpark's `fillna` function offers a straightforward way to manage this imputation. By targeting the **lagged_sales** column and specifying the replacement value as 0, we can refine our feature set without impacting the correctly calculated lagged values in the middle of the time series.

We apply the `fillna` function to the `df_new` 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|
+-----+---+-----+------------+
```

As shown above, the initial **null** values in the **lagged_sales** column for both stores have been successfully converted to zero. The resulting DataFrame is now robust, clean, and ready for further

statistical or predictive modeling.

Advanced Window Function Customization

While this example focused on a lag of 1, the flexibility of the Lag function allows for complex temporal analysis. For instance, analyzing seasonal data often requires looking back several periods (e.g., `lag(df.sales, 30)` for monthly lag in daily data). This modification requires no change to the underlying Window function definition, only an adjustment to the offset parameter.

Moreover, the structure defined by `partitionBy` and `orderBy` is universal for all advanced windowing operations in PySpark, including calculating dense rank, row numbers, or complex rolling statistics like moving averages (which would use the `rowsBetween` clause alongside the partitioning).

Mastering these window specifications is the gateway to highly efficient, distributed analysis of sequential data in PySpark environments, allowing users to move beyond simple aggregations and delve into time-dependent feature engineering at scale.

Conclusion and Further Resources

Calculating lagged features grouped by specific attributes is a routine and necessary task in data science. PySpark provides a robust, scalable, and elegant solution through the synergy of the `Window` specification and the Lag function. By ensuring that the data is correctly partitioned and ordered, data professionals can reliably generate these crucial time-series features even when processing massive distributed datasets.

For comprehensive details on all available parameters and variations, it is recommended to consult the official PySpark documentation for both window definition and function implementation.

Note: You can find the complete documentation for the PySpark **lag** function here: PySpark Lag Function Documentation.

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