How to Group a PySpark DataFrame by Month

Authored by

stats writer

February 6, 2026

Analyzing time-series data often requires grouping records based on temporal units like days, weeks, or months. In the context of big data processing using PySpark DataFrames, grouping data by month is a frequent and crucial operation for calculating periodic metrics. To achieve this, we harness the power of specific pyspark.sql.functions, combining the groupBy function to structure the aggregation and the built-in `month()` function to accurately extract the monthly identifier from a timestamp or date field. This article serves as a comprehensive guide to implementing robust monthly aggregations in PySpark.

Group by Month in PySpark DataFrame

Understanding PySpark DataFrame Grouping

A PySpark DataFrame is fundamentally a distributed collection of data organized into named columns, analogous to a table in a relational database. When performing analytical tasks, we often need to consolidate rows that share a common characteristic. The primary mechanism for achieving this data summarization is the use of the groupBy transformation, which divides the DataFrame into groups based on the values of one or more specified columns.

When dealing with temporal data, such as sales records or server logs, the grouping key must be derived from the existing date or timestamp column. PySpark simplifies this process immensely by offering specialized functions within the `pyspark.sql.functions` module that allow developers to extract specific time components--such as year, quarter, or, in our case, the month--directly within the grouping operation. This approach avoids the need for creating intermediate columns and streamlines the analytical pipeline, making it efficient for large-scale data processing.

After the data has been logically grouped, the next critical step is applying an aggregation. PySpark uses the agg function to apply statistical calculations--such as sums, counts, averages, or minimums--to the grouped data. By combining `groupBy`, the `month()` extractor, and `agg`, we create an efficient and scalable method for calculating monthly performance metrics across vast datasets distributed across a cluster.

The Core Syntax for Monthly Aggregation

The standard pattern for aggregating a PySpark DataFrame by month involves invoking the `groupBy` method on the DataFrame, utilizing the `month()` function to transform the date column into a monthly integer identifier, and concluding with the agg() function to specify the desired mathematical operation. This concise syntax is highly readable and optimized for Spark's distributed architecture.

The following snippet illustrates the foundational code required to group sales data by the month

found within the `date` column and subsequently calculate the total sales value for each resulting month. Note the use of `.alias()` to rename the resulting columns for clarity and integration into subsequent analytical steps.

from pyspark.sql.functions import month, sum

df.groupBy(month('date').alias('month')).agg(sum('sales').alias('sum_sales')).show()

In this powerful one-liner, we instruct Spark to first extract the month number (1-12) from the column named 'date' and label this new grouping column 'month'. We then calculate the summation of all values contained within the 'sales' column for every distinct month group, aliasing the result as 'sum_sales'. This methodology is central to effective time-series analysis in PySpark, providing immediate monthly summaries.

Prerequisites: Setting up the PySpark Environment and Data

Before we can execute the monthly grouping logic, we must ensure a working SparkSession is initialized and that we have a sample DataFrame containing the necessary date and metric columns. For demonstration purposes, we will create a small dataset representing sales transactions across several different months in 2023. This setup phase is crucial for ensuring the reproducibility of the example.

The data structure is simple, consisting of a `date` column, which must adhere to a valid date or timestamp format (such as the standard YYYY-MM-DD used here), and a `sales` column, which contains the numerical metric we intend to aggregate. It is essential that the date column is properly recognized by Spark, typically as a `DateType` or `TimestampType`, for the `month()` function to operate correctly and avoid unexpected null values or errors during extraction.

The following code block demonstrates the necessary setup, including importing `SparkSession`, defining the sample data, specifying the column names, and finally creating and displaying the initial DataFrame `df`.

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()

+----------+-----+
| date|sales|
+----------+-----+
|2023-04-11| 22|
|2023-04-15| 14|
|2023-04-17| 12|
|2023-05-21| 15|
|2023-05-23| 30|
|2023-10-26| 45|
|2023-10-28| 32|
|2023-10-29| 47|
+----------+-----+

As shown in the output, our sample DataFrame `df` contains records distributed across three distinct months: April (4), May (5), and October (10). This distribution provides an excellent foundation for demonstrating how the monthly grouping mechanism correctly partitions the data for subsequent calculations based on the values in the date column.

Step-by-Step Implementation: Calculating Total Sales by Month

Our primary objective is to aggregate the daily sales figures into monthly totals. This involves applying the groupBy method to the extracted month identifier and using the standard `sum()` function provided by `pyspark.sql.functions` within the agg function. This is the core transformation that produces the desired time-series summary.

The critical step lies in the expression `month('date').alias('month')`. The `month()` function takes the values from the `date` column and returns an integer representing the month number (1 for January, 12 for December). This integer then serves as the unique key for the grouping

operation, ensuring that all records sharing the same month number are processed together.

Once the DataFrame is grouped by this monthly key, the agg function computes the sum of the `sales` column across all records belonging to that specific month. The resulting DataFrame will be significantly smaller, featuring only one row per unique month present in the original dataset, along with the calculated total sales for that period.

We execute the following code to perform the aggregation and display the resulting monthly sales summary:

from pyspark.sql.functions import month, sum

#calculate sum of sales by month
df.groupBy(month('date').alias('month')).agg(sum('sales').alias('sum_sales')).show()

+-----+---------+
|month|sum_sales|
+-----+---------+
4	48
5	45
10	124
+-----+---------+

Analyzing the Results of the Sum Aggregation

The output DataFrame, consisting of the columns `month` and `sum_sales`, provides an immediate and clear summary of the total sales activity segmented by the calendar month. This resulting structure is a typical outcome of time-series aggregation and is highly valuable for reporting and visualization purposes, enabling business intelligence dashboards to track monthly performance trends easily and reliably.

By examining the output, we can directly verify the calculations against the source data to confirm the integrity of the grouping and summation process. The results show a clear progression of sales across the sampled months:

The sum of sales for dates falling in April (month 4) is correctly calculated as **48** (22 + 14 + 12 from the original data).
The sum of sales for dates falling in May (month 5) is **45** (15 + 30).
The sum of sales for dates falling in October (month 10) is **124** (45 + 32 + 47).

This verification confirms the accuracy of the combined `month()` and `sum()` operation,

demonstrating that PySpark efficiently handles the partition and calculation of data across the distributed cluster. The resulting output simplifies large, transactional datasets into actionable, time-based summaries, which is the cornerstone of operational analytics.

Extending Aggregation: Calculating the Count of Records per Month

While calculating the sum of sales is often the primary metric, analysts frequently need to know the volume or frequency of transactions within a given period. This requires applying a different aggregation function: `count()`. The architecture of the aggregation process remains identical, leveraging the same monthly grouping key; only the specific function passed to the agg method changes.

By substituting `sum('sales')` with `count('sales')`, we can determine how many individual sales records contributed to each monthly group. This metric is crucial for understanding operational throughput, assessing data completeness, and identifying potential data sparsity or irregularities in the input data based on the date column distribution.

The following code snippet demonstrates how to modify the previous logic to calculate the total count of sales transactions grouped by month, naming the resulting column `cnt_sales` for clarity:

from pyspark.sql.functions import month, count

#calculate count of sales by month
df.groupBy(month('date').alias('month')).agg(count('sales').alias('cnt_sales')).show()

+-----+---------+
|month|cnt_sales|
+-----+---------+
4	3
5	2
10	3
+-----+---------+

The resulting DataFrame clearly indicates the number of rows processed for each month. We see that April and October both contained 3 daily records, while May contained 2. This flexibility highlights the power of the PySpark aggregation framework, allowing users to switch metrics (e.g., sum, count, average, min, max) simply by changing the function imported from pyspark.sql.functions and applying it within the agg call.

Choosing the Right PySpark Functions for Time-Series Analysis

PySpark provides a comprehensive set of functions designed specifically for handling temporal data, making complex time-series analysis straightforward and highly efficient. While we focused on `month()` for this specific grouping task, it is important to recognize the availability of related functions that can adjust the granularity of aggregation based on business needs.

For instance, if the goal were to group by year and month combination (essential when data spans multiple years), one might use a combination of `year()` and `month()` functions concatenated together, or utilize the `date_trunc()` function to truncate the date directly to the beginning of the month. This latter approach often provides a more robust and unique grouping key that intrinsically includes the year component.

When preparing to analyze temporal data in PySpark DataFrames, always ensure the source column is cast to the appropriate data type--typically `DateType` or `TimestampType`--to guarantee that the extraction functions, like `month()`, execute correctly and efficiently across the cluster. Selecting the appropriate function based on the required analytical granularity is key to successful big data processing and accurate reporting.

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