Learning Time-Series Analysis: Grouping Data by Week in PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

The Crucial Role of Time-Series Aggregation in PySpark

Analyzing data across defined temporal windows--such as daily, weekly, or monthly periods--is a foundational requirement for modern data science, Business Intelligence, and large-scale operational reporting. When dealing with massive, distributed datasets, the robust performance and parallel processing capabilities of PySpark are essential. Grouping data by week provides analysts with a crucial level of granularity, allowing them to effectively smooth out daily noise, observe short-term behavioral trends, identify cyclical patterns (e.g., in sales or website traffic), and perform consistent weekly performance comparisons across various metrics. This powerful process hinges on accurately transforming raw date columns into standardized weekly identifiers before standard aggregation functions are applied.

The primary technical hurdle in performing time-series aggregation within the PySpark framework is ensuring the precise extraction and formatting of date information. While traditional SQL environments often handle date extraction implicitly, PySpark mandates the use of specialized, high-performance functions available within the `pyspark.sql.functions` module. This module provides the tools necessary to manipulate date and timestamp fields efficiently across a distributed cluster.

Specifically, we rely on the `weekofyear` function, which is designed to translate a date field into its corresponding week number (an integer typically ranging from 1 to 53) within the given year. Once this transformation is successfully executed, we can seamlessly apply standard and powerful aggregation methods, such as calculating the `sum` or `count`, to derive meaningful and actionable weekly summaries from the raw transactional data.

Essential PySpark Functions for Date Manipulation

This tutorial provides a comprehensive guide to achieving precise weekly grouping and summarization within a PySpark DataFrame. The process requires establishing a clear pipeline that converts a date column into an effective grouping key. We will focus on the essential syntax and demonstrate practical examples, ensuring complete clarity on how to manipulate date fields effectively to produce the accurate weekly reports necessary for informed, data-driven decision-making.

To successfully group rows by week, you must first import the necessary date and aggregation functions. The core operation is a standard grouping and aggregation pipeline, which relies on two critical components: the weekofyear function, which generates the required grouping key, and an appropriate aggregation function (like `sum` or `count`), which dictates how the numerical columns will be summarized within each defined group.

The overall workflow can be broken down into three logical steps: first, importing the required

functions (e.g., weekofyear and `sum`); second, applying the DataFrame's `groupBy` method using the output of the weekofyear function on the target date column; and finally, utilizing the `agg` method to calculate the desired metric on the relevant numerical column. It is highly recommended to use the `.alias()` method within the aggregation step. This practice ensures that the resulting summary columns have descriptive and meaningful names, greatly improving readability and downstream analysis.

Implementing the Core Weekly Grouping Syntax

The following syntax provides the blueprint for extracting the week number from a column named `date`, grouping the entire DataFrame based on this new weekly identifier, and calculating the total sum of values found in the `sales` column for that period. This concise code block represents the backbone of weekly reporting in any PySpark environment.

from pyspark.sql.functions import weekofyear, sum

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

In this implementation, the rows are systematically grouped based on the integer week number derived from the **date** column. Immediately following the grouping operation, the sum function is applied. This calculates the total value present in the **sales** column for every unique week observed across the entire dataset. This straightforward, yet highly efficient, approach is fundamental to time-series analysis in distributed computing environments.

Practical Walkthrough: Setting Up the Sample Data

To demonstrate this robust functionality in a real-world context, we must first construct a sample PySpark DataFrame. This synthetic DataFrame is designed to simulate typical transactional data, containing records of daily sales made over several non-consecutive months throughout a year. The structure includes two essential fields: the transaction `date` (the grouping key) and the volume of `sales` recorded on that specific day (the value to be aggregated).

We begin by initializing a `SparkSession`, which acts as the entry point for all PySpark functionality. Next, we define the raw data list and specify the corresponding column names. Finally, we invoke the `spark.createDataFrame` method to convert this local data structure into a distributed, schema-aware DataFrame, which is necessary for all subsequent PySpark processing. This preparatory step is vital, as all grouping and aggregation operations rely on the data being correctly formatted and loaded into the Spark execution context.

The following code snippet illustrates the creation of our sample DataFrame and provides a visual

confirmation of the initial dataset structure before any transformations are applied, ensuring we start with valid input data:

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

Calculating Total Sales Using GroupBy and SUM

Our primary analytical objective is to shift the focus away from daily granularity and calculate the

cumulative sum of sales for each observed week. This transformation provides a necessary higher-level view of performance, effectively smoothing out daily fluctuations and allowing underlying weekly trends to become apparent. To achieve this, we will apply the grouping logic introduced earlier, utilizing the weekofyear function in conjunction with the powerful `groupBy` and `agg` methods intrinsic to PySpark.

The following syntax executes the required transformation: it first identifies the corresponding week number for every date entry in the raw data, proceeds to group all records sharing the same week number together, and then applies the sum function to aggregate the corresponding sales figures within those groups. The result is a highly condensed and informative summary of weekly performance.

from pyspark.sql.functions import weekofyear, sum

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

+----+---------+
|week|sum_sales|
+----+---------+
15	36
16	12
20	15
21	30
43	124
+----+---------+

Interpreting Weekly Metrics: Understanding ISO Week Numbers

The resulting DataFrame represents a concise weekly summary, providing the total sales figure (`sum_sales`) corresponding to each unique calendar week (`week`) found within the original dataset. Analyzing this output allows stakeholders to rapidly identify performance peaks and troughs across the year without the tedious necessity of manually processing individual daily records. This high level of efficiency and scalability is a defining characteristic of effective big data analysis achieved using PySpark.

It is essential to recognize that the week numbers displayed (15, 16, 20, 21, 43) are derived directly from the date entries according to ISO week numbers standards. For example, the daily records of '2023-04-11' (sales: 22) and '2023-04-15' (sales: 14) both fall squarely within the 15th week of 2023. When these records are grouped, their sales figures are correctly combined (22 +

14 = 36), yielding the resulting `sum_sales` for week 15. The effectiveness of the weekofyear function is powerfully demonstrated here by its ability to automatically and accurately collapse multiple granular daily records into a single, standardized weekly summary row.

To ensure absolute clarity regarding the aggregation process, we can break down the results to confirm the accuracy of the calculation:

The cumulative sum of sales for the **15th week** of the year was calculated as **36** (22 + 14 from the original daily data).
The sum of sales for the **16th week** of the year was precisely **12**.
The total sum of sales for the **20th week** of the year totaled **15**.
The **43rd week**, which included three separate transactions (2023-10-26, 2023-10-28, and 2023-10-29), showed the highest aggregated sales total of **124** (45 + 32 + 47).

Advanced Aggregation: Shifting from SUM to COUNT

While calculating the total sum of a value (like sales revenue) is often the primary objective, the inherent flexibility of the PySpark framework allows us to easily pivot and calculate other essential metrics using the exact same grouping logic. For instance, instead of finding the monetary total, an analyst might be more interested in the frequency of transactions--that is, the total count of sales records--that occurred within each specific week.

The only modification required to shift metrics is substituting the sum function with the count function within the `agg` statement. The grouping mechanism, which is based on the immutable weekofyear column, remains perfectly identical. This elegantly demonstrates the modularity of the PySpark aggregation framework: the grouping key remains constant, while the metric applied to the value column changes according to the precise analytical requirement.

To calculate the total count of sales transactions recorded, grouped by week, we would utilize the following syntax. Note that the output column is appropriately renamed to `cnt_sales` for clear identification of the new metric:

from pyspark.sql.functions import weekofyear, count

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

+----+---------+
|week|cnt_sales|
+----+---------+
| 15| 2|
| 16| 1|

```
| 20| 1|
| 21| 1|
| 43| 3|
+----+---------+
```

The resulting table now clearly shows the number of discrete entries or transactions that occurred during each weekly period. For instance, week 43 had 3 separate sales entries, indicating high activity, while weeks 16, 20, and 21 each had only one. This metric is crucial for tracking activity levels, determining staff scheduling needs, or understanding transactional volatility independent of the sales volume itself. Furthermore, other aggregation functions such as `avg`, `min`, `max`, or `stddev` can be easily substituted here to provide a complete statistical profile of the weekly data distribution.

Conclusion and Further PySpark Resources

Mastering weekly time-series aggregation in PySpark is an indispensable skill for any data engineer or analyst working extensively with large datasets. By expertly combining the weekofyear function with the standard `groupBy` and `agg` operations, we can rapidly transform granular daily data into insightful, summarized weekly reports. This technique is highly scalable, ensuring efficient processing across distributed computing clusters managed by Spark. The ability to seamlessly swap out aggregation functions--whether using the sum function for calculating totals or the count function for frequency analysis--provides the necessary flexibility for generating complex business intelligence reports.

For professionals seeking to further expand their knowledge of PySpark's capabilities in handling complex data transformations, especially those involving date and time-series analysis, exploring the official documentation and advanced tutorials is strongly recommended. A deep understanding of functions related to date formatting, rolling averages, and windowing operations will significantly enhance your distributed data processing toolkit and allow for the creation of even more sophisticated analytical workflows.

The following resources offer additional guidance on performing other common and advanced tasks within the PySpark ecosystem: