How to Group a PySpark DataFrame by Week for Weekly Trend Analysis

Authored by

stats writer

February 4, 2026

Analyzing data based on time periods is a fundamental requirement in data science, especially when dealing with transactional or streaming datasets. When working with large volumes of data using the **PySpark DataFrame** structure, grouping records by the week provides crucial insights into short-term cyclical behavior and performance metrics. This methodology, integral to **Time Series Analysis**, allows data professionals to smooth out daily noise and clearly identify patterns, seasonality, and significant deviations on a reliable weekly timescale.

The process of grouping by week is highly practical. It involves deriving the week number from a date column and then applying various aggregate functions, such as calculating the **sum**, **mean**, **count**, or **standard deviation**, across all records belonging to that specific week. This level of Aggregation is essential for generating meaningful business intelligence reports, facilitating easy comparisons between different operational periods, and ultimately enabling prompt, data-driven strategic decisions based on established weekly trends.

Introduction: The Value of Weekly Time Series Analysis in PySpark

For organizations tracking key performance indicators (KPIs) like sales, website traffic, or resource utilization, daily fluctuations can often obscure the underlying long-term patterns. By leveraging the power of **PySpark**, which is designed for processing massive datasets, we can systematically transform raw, timestamped data into actionable weekly metrics. This transformation moves the analysis from granular transaction logs to high-level summaries, offering a panoramic view of operational efficiency or market response over rolling weekly windows. This is particularly valuable in environments where data volume necessitates distributed computing capabilities.

PySpark provides a highly optimized set of functions within its `pyspark.sql.functions` module that make temporal grouping straightforward and efficient. Unlike manual SQL approaches that might require complex substring operations or calendar calculations, PySpark abstracts this complexity, allowing developers to focus purely on the analytical outcome. The key to this weekly grouping is the use of the `weekofyear() function`, which standardizes date conversion across the distributed cluster.

Implementing weekly grouping is a crucial step in conducting thorough **Time Series Analysis**. By collapsing data into weekly buckets, we gain a clearer perspective on seasonality--for instance, identifying sales spikes every fourth week due to payroll cycles--or tracking the impact of short-term marketing campaigns more effectively than relying solely on monthly summaries.

Prerequisites and Essential PySpark Functions for Date Manipulation

Before executing any time-based grouping operation in PySpark, it is crucial to ensure that the necessary functions are imported and that the date column in your **PySpark DataFrame** is correctly cast as a Date or Timestamp type. Although Spark is often intelligent enough to handle

common string formats implicitly, explicit type casting is a best practice to prevent unexpected errors during distributed processing, especially when using date-specific functions across large clusters.

The core functions required for this transformation are found within the `pyspark.sql.functions` library. Specifically, we rely on `weekofyear()`, which extracts the week number (1-53) from a given date column based on the calendar year. Additionally, the standard aggregation functions like `sum()`, `avg()`, or `count()` must also be imported. By combining these functions with the standard `groupBy()` and `agg()` DataFrame methods, we construct a concise yet powerful pipeline for weekly data reduction, ensuring computational efficiency.

Proper function selection is vital for accurate weekly Aggregation. The `weekofyear()` function returns the week number of the year, where weeks start on Sunday and the first week of the year contains January 1st. Understanding this definition is essential for interpreting the output correctly, particularly when comparing results against different calendar standards, such as ISO 8601, which uses a Monday start day and a different definition for Week 1.

The Core Syntax: Grouping DataFrames Using weekofyear()

The primary method for achieving weekly grouping is by first generating a new column representing the week number using the `weekofyear() function` within the `groupBy()` clause. This ensures that the subsequent aggregation steps process rows based on this derived temporal category. It is standard practice to use the `alias()` function immediately after defining the week column to give it a descriptive name, such as 'week', thereby improving readability and integration into subsequent analysis steps.

Once the grouping key (the week number) is established, the `agg()` method is applied. This method takes one or more aggregate function calls (e.g., `sum('sales')`), optionally aliased for clarity. This chained command structure efficiently performs the weekly summary calculation across the entire distributed dataset. The resulting output is a new **PySpark DataFrame** where each row represents a unique week and the corresponding aggregated value.

The fundamental syntax for grouping rows by week in a **PySpark DataFrame** and calculating a summary metric is demonstrated below. This structure serves as the template for all weekly aggregation tasks, emphasizing clarity and computational efficiency:

from pyspark.sql.functions import weekofyear, sum

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

This powerful command specifically groups all records in the DataFrame based on the week

number extracted from the `date` column. Subsequently, it computes the total (sum) of values found in the `sales` column for each identified week. This two-step process--deriving the group key and then applying the metric calculation--is the standard pattern for complex temporal Aggregation in PySpark.

Step-by-Step Example: Defining and Viewing the Source DataFrame

To fully illustrate the practical application of this syntax, consider a scenario involving daily sales data collected over several months. We must first initialize the Spark environment and define a representative dataset. This step ensures that we have a suitable **PySpark DataFrame** containing a date column and a numerical value column (sales) upon which to perform our weekly analysis. Initialization of the Spark context requires creating a **SparkSession**, which is the entry point for all PySpark functionality and manages the distributed resources.

The example below defines the data, specifies the schema (column names), and then creates and displays the resulting DataFrame. Notice how the dates are intentionally scattered across different weeks and months, simulating real-world data collection where transactions are sporadic or non-sequential.

Suppose we have the following **PySpark DataFrame** that contains transactional information about daily sales figures across various dates:

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

This initial DataFrame, `df`, provides the raw transactional data points necessary for analysis. Our immediate goal is to compress this daily information into weekly totals, allowing us to quickly assess weekly performance without being overwhelmed by eight separate daily entries. This shift from granularity to summary is the core utility of weekly grouping in **Time Series Analysis**.

Implementing Weekly Aggregation: Calculating Sum of Sales

We now proceed to execute the grouping operation using the syntax introduced earlier. The objective is to calculate the total sum of sales for every distinct week present in the **`date`** column. This calculation efficiently identifies high-performing and low-performing weeks, providing actionable metrics for sales managers and financial analysts.

The implementation requires importing `weekofyear` and the `sum` functions from `pyspark.sql.functions`. We then apply the chaining logic: `df.groupBy(...)` determines the groups, and `.agg(...)` executes the specific calculation requested, resulting in a concise, aggregated output. The resulting DataFrame will clearly show the week number alongside the corresponding summed sales total, ready for visualization or reporting.

We use the following specific syntax to calculate the sum of the **`sales`** column, grouped by week derived from the **`date`** column:

```
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 the Results of Weekly Grouping

The resulting DataFrame above provides a succinct summary of the original data. Instead of eight rows of daily data, we now have five rows, each representing a consolidated week of activity. Interpreting these results involves mapping the derived week number back to the original calendar period, though for analytical purposes, the numeric identifier is often sufficient for tracking sequential performance and creating comparison charts.

For enhanced clarity, let us break down the meaning of the resulting weekly aggregates:

The resulting DataFrame, containing columns **`week`** and **`sum_sales`**, displays the total sales volume achieved during that 52-week annual cycle.
The **`sum_sales`** value for week 15 is **36**. This corresponds to the sales from April 11th (22) and April 15th (14), which both fall within the 15th week of the year 2023, based on the default PySpark calendar calculation.
Week 16 recorded a total sales figure of **12**, corresponding solely to the transaction on April 17th.
The highest performance is clearly visible in Week 43, which accumulated **124** in sales (45 + 32 + 47), indicating a strong performance period likely encompassing the last few days of October.

This aggregated view drastically simplifies trend analysis. We can immediately identify that Week 43 represents a significant peak compared to the earlier weeks (15, 16, 20, 21). Such visualization of weekly totals is vastly more informative and easier to manage than analyzing the raw daily figures alone, especially when dealing with hundreds of thousands of daily records.

Extending Aggregation: Analyzing Weekly Frequency (Count)

The flexibility of the PySpark grouping mechanism allows us to substitute the `sum()` function with any other standard aggregation metric, depending on the analytical goal. For example, instead of calculating the total sales value, we might be interested in the frequency of transactions--that is, how many sales events occurred within each week.

To achieve this, we simply replace `sum('sales')` with `count('sales')` within the `agg()` function. This reveals the distribution of transaction volume, which can be useful for monitoring operational load, assessing staff efficiency, or analyzing customer activity patterns, independent of the monetary value of those transactions.

The following syntax demonstrates how to calculate the total count of sales transactions, grouped by week:

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 DataFrame, showing **`cnt_sales`**, confirms that Weeks 15 and 43 were the busiest in terms of transaction frequency, hosting two and three transactions respectively. This reinforces the finding that Week 43 was the peak period, based on both volume (sum) and frequency (count), providing a robust conclusion about that time period.

Advanced Considerations: Handling Missing Data and ISO Weeks

While the `weekofyear() function` is highly useful, it is important for expert users to understand its underlying calendar definition. As mentioned, it typically treats the first week of the year as starting on January 1st, regardless of the day of the week. For highly rigorous financial or logistical analyses, adhering to the internationally recognized **ISO 8601 standard** (where Week 1 is the first week with four or more days in the new year, starting on Monday) may be necessary.

PySpark offers alternative functions to handle ISO standards, such as `iso_weekofyear()` or, even more commonly, `date_trunc('week', date_col)`, which provides the actual start date of the week rather than just a number. Choosing the appropriate function depends strictly on the specific requirements of the organization's official calendar standards. Furthermore, effective data cleaning is critical: handling null or invalid date entries prior to grouping is essential, as the date functions

will not process corrupted data correctly, potentially leading to incomplete weekly summaries.

Using `date_trunc` is often preferred by advanced analysts because it returns an actual timestamp representing the start of the week (e.g., '2023-04-10 00:00:00'), which makes the resulting groups instantly recognizable without needing external calendar lookups. This method offers greater clarity for subsequent joining operations or visualization tasks in **Time Series Analysis**.

from pyspark.sql.functions import date_trunc, sum

#calculate sum of sales by week, using date_trunc for clarity
`df.groupBy(date_trunc('week', 'date').alias('week_start_date')).agg(sum('sales').alias('sum_sales')).show()`

While the original example focuses on `weekofyear()` for simplicity, recognizing and employing these more precise temporal functions ensures the highest standard of accuracy in professional data engineering environments.

Conclusion: Leveraging Weekly Insights for Data-Driven Decisions

The ability to efficiently group and aggregate large datasets by temporal units like the week is a cornerstone of modern data processing using **PySpark**. By utilizing the `weekofyear() function` in conjunction with robust Aggregation methods, data engineers can quickly transform daily transactional noise into meaningful weekly summaries, regardless of the scale of the input data.

These weekly summaries enable organizations to monitor performance trends, identify critical sales periods, and react swiftly to cyclical changes in the market or operational environment. Whether calculating the sum of monetary values, the count of transactions, or the average customer rating, the structured methodology of PySpark ensures that this vital **Time Series Analysis** is performed accurately and at scale, unlocking the true potential of big data insights. Mastering these grouping techniques is fundamental for any data professional working within the Apache Spark ecosystem.

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