How to Group a PySpark DataFrame by Week and Calculate Weekly Sums

Authored by

stats writer

January 19, 2026

The Power of Time-Based Analysis in DataFrames

Analyzing data based on temporal components, such as weeks, months, or quarters, is fundamental in business intelligence and data science. When working with large datasets distributed across a cluster, PySpark provides robust tools to handle these time-series operations efficiently. Grouping data by week allows analysts to identify underlying trends, assess cyclical performance, and measure weekly productivity without the computational overhead of processing every single row individually. This technique is particularly valuable for tracking sales figures, website traffic, or resource utilization over distinct, easily comparable periods.

The Core PySpark Syntax for Weekly Grouping

To aggregate data within a DataFrame based on the calendar week, we rely on specialized functions available within the `pyspark.sql.functions` module. Specifically, the weekofyear function is essential as it extracts the week number (1-53) from a date column, transforming the timestamp data into a manageable grouping key. Once this key is established, the standard `groupBy()` operation is applied, followed by an appropriate aggregation function like `sum()` or `count()`.

The standard syntax for grouping rows by week in a PySpark DataFrame involves importing the necessary functions and chaining the operations. Review the following foundational code block which demonstrates this powerful operation:

from pyspark.sql.functions import weekofyear, sum

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

This concise command executes a multi-step process: first, it applies the weekofyear function to the `date` column, assigning the resulting weekly index an alias of `week`. Second, it uses `groupBy()` to organize all records sharing that weekly index. Finally, the `agg()` method calculates the sum of all values found in the `sales` column for each group, naming the resulting column `sum_sales`. This pattern forms the backbone of temporal Aggregation within Spark's distributed computing environment.

Prerequisites: Setting up the PySpark Environment

Before executing any data transformation logic, it is crucial to initialize a SparkSession. The PySpark DataFrame API depends entirely on this session object to communicate with the underlying Spark cluster. The session acts as the entry point for utilizing all Spark functionality. In a local development environment, the `builder.getOrCreate()` method ensures that either an

existing session is reused or a new one is instantiated if none is currently active.

For our practical demonstration, we will first define the necessary imports and establish this core session. Following initialization, we construct a sample dataset simulating daily sales transactions, which includes a date string and the corresponding sales amount. This structured approach ensures reproducibility and clarity throughout the example.

Creating and Inspecting the Sample DataFrame

Let us define a scenario where we have daily sales records from a company spanning several non-contiguous weeks throughout 2023. This sample data structure is defined as a list of lists, where the first element is the **date** (formatted as YYYY-MM-DD) and the second element is the corresponding **sales** figure. The creation process involves defining the data, specifying the column names, and finally calling `spark.createDataFrame()` to materialize the distributed data structure.

The following code block sets up the environment and creates the initial DataFrame (`df`). Note how the `show()` action triggers the execution and displays the first twenty rows, confirming the data integrity before further processing.

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|
+----------+-----+

It is important to acknowledge that when creating a DataFrame using string representations of dates, PySpark automatically infers the schema. While our subsequent use of the weekofyear function implicitly handles the date conversion for the calculation, for complex time operations, explicitly casting the column to a `DateType` or `TimestampType` using `withColumn` and `to_date` is usually recommended practice.

Performing the Weekly **Aggregation**: Summing Sales

Our primary objective is to calculate the total sales volume for each calendar week represented in the dataset. This requires the combined power of three chained methods: `weekofyear()` for extraction, `groupBy()` for partitioning, and `agg(sum())` for the final reduction operation. By grouping on the extracted week number, we effectively consolidate all daily sales figures that fall within the same 7-day period.

The use of `alias()` is a best practice, ensuring the resulting column names are descriptive (e.g., `week` and `sum_sales`). This clarity enhances the readability of the resulting schema and facilitates downstream analysis or reporting. Observe how the following code block executes this transformation and displays the consolidated weekly totals:

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 the Weekly Summation

The resulting DataFrame provides a clear, aggregated view of sales performance organized by the calendar week number. Each row represents a unique week, and the associated `sum_sales` column reflects the cumulative sales for all dates falling within that week. This output immediately facilitates time-series comparisons and performance benchmarking.

For instance, we can draw immediate conclusions regarding the distribution of sales activity across the year 2023, based on the provided sample data:

The sum of sales recorded during the **15th week** of the year was **36**. This aggregate value combines the sales from April 11th (22) and April 15th (14), which both fall into the same ISO week definition.
Sales during the **16th week** were lower, totaling **12**, corresponding solely to the transaction on April 17th.
The most significant activity occurred in the **43rd week**, registering a total sale amount of **124**, demonstrating a peak period in late October.

This structured output simplifies the process of identifying high- and low-performing periods without needing to manually cross-reference individual daily records.

Alternative Aggregation: Counting Weekly Transactions

While summing sales is a common requirement, the grouping methodology is flexible enough to accommodate various other aggregation metrics. Often, data analysts need to determine the frequency or count of transactions rather than their monetary value. This is achieved by replacing the `sum()` function within the `agg()` method with the `count()` function. This modification yields a count of how many records contributed to each weekly group.

To calculate the total count of sales transactions, grouped by week, we simply adjust the imported function and the aggregation operation, as demonstrated below. This provides insight into transaction volume--a metric often distinct from total sales value, useful for analyzing operational tempo or customer activity.

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

In this context, `cnt_sales` indicates the number of distinct records (or days, in our simple dataset) that contributed data to that specific week. For instance, week 43 had **3** recorded sales entries, which aligns with the three dates (October 26, 28, and 29) defined in the source data.

Considerations for Date and Time Functions in PySpark

When working with time-based data in PySpark, it is critical to understand the nuances of the functions being used. The weekofyear function determines the week number based on the definition used by the Spark cluster's underlying configuration, which typically follows the ISO 8601 standard or an internal calendar system depending on the environment setup. Analysts should confirm which day is considered the start of the week (Sunday vs. Monday) if week boundaries are critical to their reports.

Furthermore, PySpark offers numerous other time-based functions that can enhance this grouping methodology. For example, if grouping by month were required, `month()` could be used instead of weekofyear. For more advanced analysis, functions like `window()` allow for time-series grouping based on sliding or fixed intervals, which provides capabilities beyond simple calendar unit aggregation. This flexibility ensures that PySpark can handle complex analytical requirements efficiently.

Conclusion: Enhancing Data Analysis Capabilities

The ability to perform distributed, time-based Aggregation is a cornerstone of effective big data processing using PySpark. By leveraging the `groupBy()` method in conjunction with specialized temporal functions like weekofyear, users can quickly transform vast transactional datasets into meaningful summaries. This approach not only optimizes performance by executing computations across a cluster but also provides immediate, actionable insights into weekly trends and

performance metrics. Mastering this pattern is essential for any data professional working in the Spark ecosystem.

For those seeking to expand their knowledge of PySpark, several related tutorials explaining how to perform other common tasks are highly recommended: