How to Group a PySpark DataFrame by Date for Analysis

Authored by

stats writer

February 5, 2026

Analyzing transactional or observational data often requires grouping records based on temporal characteristics. When working with PySpark, a powerful tool for large-scale data processing, achieving accurate date-based aggregation requires careful handling of column types, particularly when dealing with timestamps.

The fundamental method for grouping a DataFrame by date involves utilizing the built-in `groupBy()` function. However, because data containing timestamps usually includes hours, minutes, and seconds, we must first convert the complex timestamp structure down to a simpler date structure. This conversion ensures that all records occurring on the same calendar day are treated as a single group, irrespective of the precise time of day they occurred.

This technique is essential for processing time-series data, enabling critical tasks such as calculating daily totals, averages, or counts. By combining `groupBy()` with standard aggregation functions like `sum()` or `count()`, users can efficiently summarize massive datasets, transforming raw transactional logs into actionable business metrics.

Group by Date in PySpark DataFrame

Prerequisites for Date Grouping in PySpark

When dealing with time data in PySpark, it is crucial to recognize that timestamps (which contain date and time information) must be explicitly converted into a date format before grouping. If you attempt to use `groupBy()` directly on a column of type TimestampType, PySpark will group by the combination of year, month, day, hour, minute, and second. Since most transaction timestamps are unique down to the second, this would result in almost no effective grouping.

To solve this, we leverage the casting mechanism available within the PySpark SQL functions. Casting allows us to change the data type of a column. Specifically, we cast the timestamp column to the DateType. This conversion effectively truncates the time components, leaving only the year, month, and day. This truncated value then serves as a robust grouping key.

The standard process involves three steps: first, ensuring the column is in a valid timestamp format (often converting from an initial string); second, applying the `cast(DateType())` operation; and third, applying the `groupBy()` and subsequent aggregation functions. Understanding this explicit type conversion is the single most important concept for performing accurate date-based aggregations in PySpark.

Implementing the Core Syntax

The core mechanism relies on using the `cast()` function to transform the column data type within

the `groupBy()` call itself. This operation is dynamic and does not require modifying the original DataFrame structure unless desired. The syntax below shows how to calculate the sum of a metric (e.g., sales) after grouping the rows based purely on the date component extracted from a timestamp column named `ts`.

Here is the fundamental structure demonstrating how to group rows by date using the casting mechanism:

from pyspark.sql.types import DateType

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

In this specific example, the rows of the DataFrame are grouped by date based on the values in the **ts** column. The `cast(DateType())` function extracts the date part, and `alias('date')` names this new grouping column. Following the grouping, the `agg()` function is used to apply the `sum('sales')` operation, calculating the total sales value for each unique date group.

This syntax is highly efficient because the conversion happens directly within the optimized Spark query plan. It allows for flexible aggregation, meaning you can replace `sum()` with other functions like `avg()`, `min()`, or `count()`, depending on the required business metric.

Setting Up the Sample DataFrame

To illustrate this process clearly, let us define a sample DataFrame containing transactional sales data. This data includes a timestamp column (`ts`) and a numeric column representing the sales amount (`sales`). This scenario is typical in business intelligence and data warehousing environments where events are recorded precisely down to the second.

First, we must initialize the SparkSession and define our raw data. Crucially, the timestamp column often starts as a simple string type. We must convert this string column into a proper PySpark timestamp format (TimestampType) using `F.to_timestamp` before we can reliably cast it to a DateType for grouping.

The setup code below initializes the environment, defines the data, converts the necessary string column, and displays the initial, pre-grouped DataFrame:

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

```
from pyspark.sql import functions as F

#define data
data = ,
,
,
,
]

#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#convert string column to timestamp
df = df.withColumn('ts', F.to_timestamp('ts', 'yyyy-MM-dd HH:mm:ss'))

#view dataframe
df.show()

+-------------------+-----+
| ts|sales|
+-------------------+-----+
|2023-01-15 04:14:22| 20|
|2023-01-15 10:55:01| 30|
|2023-01-15 18:34:59| 15|
|2023-01-16 21:20:25| 12|
|2023-01-16 22:20:05| 15|
|2023-01-17 04:17:02| 41|
+-------------------+-----+
```


As observed in the output, the `ts` column now contains timestamps, where the rows recorded on January 15, 2023, have different times of day (04:14:22, 10:55:01, 18:34:59). Our goal is to consolidate these three records into a single row representing the total activity for 2023-01-15. This is the exact challenge that date-based grouping resolves.

Calculating the Sum of Sales by Date

With the DataFrame initialized and properly typed, we can now proceed to apply the grouping logic

to calculate the total sales metric per calendar day. This involves importing the necessary DateType and chaining the `groupBy()` and `agg()` functions. The process effectively collapses multiple timestamp entries into a single daily summary record.

The following command executes the aggregation. Note how the column `ts` is converted to a date, used for grouping, and then the `sum()` function is applied to the `sales` column, creating a new metric column named `sum_sales`:

from pyspark.sql.types import DateType

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

+----------+---------+
| date|sum_sales|
+----------+---------+
|2023-01-15| 65|
|2023-01-16| 27|
|2023-01-17| 41|
+----------+---------+

The resulting DataFrame provides the summarized totals, drastically reducing the number of rows while preserving the essential daily information. We can verify the results against the initial dataset:

For 2023-01-15, the original sales were 20, 30, and 15. The summed result is **65**.
For 2023-01-16, the original sales were 12 and 15. The summed result is **27**.
For 2023-01-17, the original sale was 41. The summed result is **41**.

Performing Alternative Aggregations

The flexibility of the `groupBy()` function combined with the `agg()` function allows for easy switching between different statistical measures. While calculating the sum is a common requirement, analysts frequently need to know the volume or frequency of transactions. For this purpose, we can use the `count()` function instead of `sum()`.

Using `count()` helps determine how many individual sales records occurred on each specific day, providing insights into transaction volume rather than monetary value. This is particularly valuable for performance monitoring or capacity planning.

The following syntax demonstrates how to calculate the count of sales records, grouped by date.

Notice that only the aggregation function within `agg()` is changed from `sum('sales')` to `count('sales')`, highlighting the ease of customization in PySpark:

from pyspark.sql.types import DateType

#calculate count of sales by date
df.groupBy(df.cast(DateType()).alias('date'))
.agg(count('sales').alias('count_sales')).show())

+----------+-----------+
| date|count_sales|
+----------+-----------+
|2023-01-15| 3|
|2023-01-16| 2|
|2023-01-17| 1|
+----------+-----------+

The resulting DataFrame now clearly shows the transaction frequency. On January 15th, there were 3 sales records; on January 16th, there were 2; and on January 17th, there was 1. This confirms that the date grouping correctly consolidated all records corresponding to the truncated date value.

The key takeaway is the power of the `agg()` function after grouping. It accepts various built-in functions, allowing users to define complex aggregation rules--such as calculating the average sales price (using `avg()`) or finding the maximum sales value (using `max()`)--all based on the unified daily grouping key derived from the timestamp column.

Summary of the PySpark Grouping Workflow

Mastering date-based aggregation in PySpark is fundamental for anyone working with structured or semi-structured temporal data. The necessity arises from the common need to reduce high-granularity timestamp information (down to the second) into manageable daily metrics.

The entire workflow hinges on two crucial operations:

Data Type Preparation: Ensuring the timestamp column is first a valid TimestampType (if starting from strings), and then explicitly converting it to the `DateType` within the grouping operation using `.cast(DateType())`.
Grouping and Aggregation: Applying the `groupBy()` function using the newly cast date column as the key, followed immediately by the `agg()` function to compute the desired metrics (sum, count, average, etc.).

By following these steps, you can efficiently handle large volumes of time-series data and transform it into meaningful daily, weekly, or monthly reports. This method ensures that your analytical results are accurate, relying on the inherent optimization and scalability of the Spark engine.

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