How to Group and Aggregate Data by Date in a PySpark DataFrame

Authored by

stats writer

January 2, 2026

Analyzing time-series data efficiently is a cornerstone of modern data engineering and business intelligence. When working with large datasets in the Apache Spark ecosystem, specifically using PySpark, it is frequently necessary to distill high-granularity data (like timestamps) down to daily, weekly, or monthly summaries. Grouping records by date within a DataFrame is achieved primarily through the combination of the powerful `groupBy` function and subsequent aggregation methods like `agg`.

This technique allows developers and analysts to shift from row-level detail to meaningful summaries. By aggregating data based on a date column, you can perform essential statistical operations on each resulting group. These operations typically include calculating the **sum**, **average** (mean), or **count** of specific values contained within those groups. Mastering date grouping is fundamental for powerful data analysis, visualization, and reporting workflows executed within the PySpark environment.

Understanding the Need for Date Grouping

In real-world scenarios, datasets often contain transaction timestamps (`ts`) that capture events down to the second or millisecond. While this level of detail is necessary for transactional logs, it is often too granular for macro-level analysis, such as tracking daily sales performance or monitoring website traffic patterns over time. To derive business insights, we must normalize these timestamps into standard dates.

The primary challenge in PySpark is that the grouping column must have a consistent value for all records that should belong to the same group. A timestamp column, even for events occurring on the same day, will have unique time components. Therefore, the crucial preprocessing step involves extracting or converting the timestamp field into a pure date field, effectively discarding the time component. Once this transformation is complete, the standard groupBy function can be applied to isolate all records sharing the exact same calendar date.

Effective date grouping unlocks numerous possibilities for time-series feature engineering. For instance, it allows the calculation of daily totals necessary for creating rolling averages or year-over-year comparisons. This simplification of the time dimension makes complex analytical tasks far more manageable and efficient when dealing with terabytes of distributed data handled by the Spark engine.

The PySpark Grouping Syntax Explained

To successfully group rows by date in a PySpark DataFrame, you must utilize the `cast()` function to convert the original timestamp column into a `DateType` before applying the grouping logic. This ensures that only the year, month, and day are considered for the grouping key. The overall syntax structure is concise yet powerful, leveraging chained methods typical of the functional

programming paradigm found in PySpark DataFrames.

You can use the following fundamental syntax template to group rows by date in a PySpark DataFrame:

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()

This particular example demonstrates a complete sequence of operations. It first targets the timestamp column, typically named `ts`, and explicitly converts its data type to `DateType`. It then renames this new date column using `alias('date')`, which provides a clean grouping key. Finally, the `agg` function calculates the **sum** of the values in the `sales` column, resulting in the total sales for each unique calendar date identified by the grouping operation.

The Importance of Type Casting to DateType

When dealing with time-related data in PySpark, data types are critically important. If your data is initially loaded as strings, it must first be converted to a proper `TimestampType` before attempting to extract the date component. Assuming a proper timestamp (like `TimestampType`) is already present, the subsequent step is utilizing the `cast(DateType())` method.

The `DateType` in PySpark represents a physical date value, containing only year, month, and day information, without time zone or time-of-day details. By casting the timestamp column to this type directly within the `groupBy` clause, PySpark efficiently truncates the time part (hours, minutes, seconds) for every record. This truncation is what ensures that all events occurring, for example, on January 15, 2023, are treated as identical entries for the purpose of aggregation.

It is best practice to apply an `alias()` immediately after the cast operation. This step not only adheres to good programming conventions by providing a descriptive name (like 'date') for the new grouping column but also prevents potential confusion if the original DataFrame contains multiple time-related fields. Using `cast` ensures the date isolation process is performed dynamically and optimized by the Spark execution engine.

Step-by-Step Practical Implementation

To illustrate this process clearly, we will construct a sample DataFrame that simulates sales data recorded at various timestamps. This setup phase ensures that we are working with the appropriate data types before attempting the aggregation step. We assume a scenario where a

company records sales transactions throughout the day, and we wish to analyze their daily performance metrics.

The initial setup requires importing the necessary Spark components, defining the raw data, and creating the DataFrame. Crucially, if the timestamp column is initially a string (which is common when loading CSV or JSON files), we must explicitly convert it to a `TimestampType` using PySpark SQL functions, such as `F.to_timestamp`, before proceeding with date grouping.

Example: Setting Up the PySpark DataFrame

Suppose we have the following PySpark DataFrame that contains detailed information about sales made at various timestamps:

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 contains precise time information, making direct aggregation impossible if we desire daily totals. Our goal now is to aggregate the monetary sales amounts based strictly on the day the transaction occurred, collapsing the multiple entries for January 15th and January 16th into single summary rows.

Calculating Aggregate Metrics using Sum

Now that the DataFrame is prepared and the timestamp column is correctly formatted, we can apply the date grouping logic to calculate the total sales per day. We explicitly import `DateType` and then chain the `groupBy` and `agg` functions.

The `agg()` function accepts PySpark SQL aggregation functions (like `sum()`, `avg()`, `min()`, etc.) and calculates these metrics across the groups defined by the preceding `groupBy` call. In this instance, we are using the `sum()` function on the `sales` column and naming the resulting aggregated column `sum_sales`.

We use the following syntax to calculate the sum of the sales, grouped by date:

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, displayed above, provides a clean, summarized view. Each row now

represents an entire day, and the `sum_sales` column reflects the cumulative sales recorded across all transactions that occurred on that specific date. For clarity, we can verify the manual calculations:

The sum of sales for **2023-01-15** is 20 + 30 + 15 = **65**.
The sum of sales for **2023-01-16** is 12 + 15 = **27**.
The sum of sales for **2023-01-17** is **41** (only one entry).

Performing Alternative Aggregations

While calculating the sum is often the most common requirement, the power of the `agg` function lies in its flexibility to apply any number of standard aggregation functions. For instance, instead of summing the sales amounts, an analyst might be interested in the number of transactions that occurred on a given day. This requires using the `count()` aggregation function.

The structure remains identical, ensuring the grouping key is still the date derived from the timestamp column. Only the function passed into `agg()` changes, along with the output column alias, which we update to `count_sales` for clarity.

For example, you could use the following syntax to calculate the **count** of sales transactions, grouped by date:

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 volume per day. We see that on 2023-01-15, three distinct sales transactions occurred, while on 2023-01-17, only one transaction was recorded in our sample data. This metric provides a different perspective on performance compared to the sum, useful for understanding operational loads or frequency metrics.

Advanced Considerations and Performance

While the combination of `cast(DateType)` and `groupBy` is the canonical method for date aggregation, performance considerations are critical when dealing with massive PySpark DataFrames. The `groupBy` operation inherently involves a shuffle, where data must be redistributed across the Spark cluster nodes based on the grouping key (the date).

To optimize this shuffle process, ensure that the preceding operations, such as the initial conversion from string to timestamp, are performed efficiently. If performance is a significant concern, consider using the built-in PySpark SQL functions like `F.date_trunc('day', col('ts'))` instead of `cast(DateType)`. While both achieve the same result--truncating the time component--`date_trunc` is often optimized at the SQL level and can sometimes yield minor performance improvements in specific Spark versions and configurations, especially when dealing with high-volume real-time streams.

Furthermore, if you need to perform multiple aggregations simultaneously (e.g., calculate sum, count, and average in one go), the `agg` function supports passing a dictionary or multiple arguments, enabling all calculations to occur within a single pass over the grouped data. This minimizes computational overhead and is highly recommended for complex analytical tasks.

Summary of PySpark Date Grouping Workflow

The process of grouping data by date in PySpark is a critical skill for time-series analysis. It involves a three-stage pipeline that ensures data fidelity and efficient computation:

Data Preparation: Ensure the time column is of `TimestampType`. If it is a string, use `F.to_timestamp()` for conversion.
Grouping Key Definition: Use `.cast(DateType()).alias('date')` within the `groupBy()` function to create a new column that serves as the precise, time-independent grouping key.
Aggregation: Apply the desired metric calculations (e.g., `sum()`, `count()`, `avg()`) using the `agg()` function to compute summary statistics for each unique date group.

By following these steps, you can effectively transform granular transactional data into high-level daily metrics, enabling clear analysis and visualization of temporal trends within large datasets.

The following tutorials explain how to perform other common tasks in PySpark, building upon the foundational knowledge of DataFrame manipulation and aggregation: