How can I group a PySpark DataFrame by year?

Authored by

stats writer

January 20, 2026

Introduction: Grouping Data by Time in PySpark

Analyzing time-series data is a fundamental requirement in modern data processing. When utilizing PySpark DataFrames, analysts frequently need to aggregate metrics based on temporal boundaries, such as calculating totals or averages for a specific year, quarter, or month. Grouping data by year simplifies trend analysis and provides a high-level overview of performance metrics across different operational periods. This process requires leveraging built-in functions provided by the PySpark SQL functions module, specifically those designed for date manipulation.

The core challenge involves extracting the year component from a date column and using that extracted value as the key for the aggregation operation. Since PySpark operates on distributed data, efficiency is paramount. Using optimized functions ensures that the operations are executed effectively across the cluster. We will explore the precise syntax and methodology required to achieve clean, yearly aggregations, transforming granular transactional data into actionable summary statistics.

The Core Syntax for PySpark Grouping by Year

To effectively group rows by year within a PySpark DataFrame, we rely on the year function and the standard groupBy and `agg` methods. The `year()` function extracts the year from a specified date or timestamp column, allowing it to serve as the grouping key. Following the grouping operation, the `agg()` function permits the application of one or more aggregation functions, such as `sum`, `count`, `avg`, or `max`, to the remaining numeric columns.

The following syntax provides a clean and standard approach to calculating the sum of a metric (e.g., sales) grouped by the year extracted from a date column:

from pyspark.sql.functions import year, sum

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

In this structure, the column named `date` is processed by the year function, and the resulting column of years is aliased simply as `year`. Subsequently, the sum function calculates the total values found in the `sales` column for each unique year, outputting the result in a column aliased as `sum_sales`. This demonstrates a robust mechanism for summarizing large datasets based on annual intervals.

Prerequisites: Setting Up the Sample PySpark DataFrame

Before executing the aggregation logic, we must establish a representative PySpark DataFrame. For this demonstration, we will create a DataFrame containing transactional records, specifically

daily sales figures linked to specific dates over several years (2021, 2022, and 2023). This setup ensures we have sufficient chronological variation to test the yearly grouping mechanism accurately.

The construction involves initializing a Spark Session, defining the raw data, specifying the column schema (`date` and `sales`), and finally, creating the DataFrame using `spark.createDataFrame`. Note that the `date` column must be in a format that PySpark can correctly interpret as a date type for the `year()` function to operate successfully, which is typically handled automatically when using standard date strings like 'YYYY-MM-DD'.

Here is the practical code snippet for generating the sample data, followed by a visualization of the initial DataFrame:

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

#define data containing daily sales records
data = ,
,
,
,
,
,
,
]

#define column names (schema)
columns =

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

#view dataframe structure and content
df.show()

+----------+-----+
| date|sales|
+----------+-----+
|2021-04-11| 22|
|2021-04-15| 14|
|2021-04-17| 12|
|2022-05-21| 15|

|2022-05-23| 30|
|2023-10-26| 45|
|2023-10-28| 32|
|2023-10-29| 47|
+----------+-----+

The DataFrame `df` now holds eight records across three distinct years. Our objective is to consolidate these records, calculating the total sales attributed to each year present in the `date` column.

Implementation Example 1: Calculating the Sum of Sales Per Year

The most common use case for time-based grouping is calculating cumulative metrics, such as total revenue or total transactions, over the specified period. We will now apply the core syntax discussed earlier to calculate the sum of sales for each year in our sample DataFrame. This process involves importing the necessary functions, defining the grouping key using the year function, and executing the aggregation via the sum function.

It is important to ensure that the column used for aggregation (`sales` in this case) is numeric, as aggregation functions like `sum` and `avg` cannot operate on string or complex types. Furthermore, aliasing the resulting columns (`year` and `sum_sales`) enhances readability and makes the resulting DataFrame easier to utilize in subsequent operations.

Observe the executed code and its resulting output, which clearly summarizes the annual performance:

from pyspark.sql.functions import year, sum

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

+----+---------+
|year|sum_sales|
+----+---------+
|2021| 48|
|2022| 45|
|2023| 124|
+----+---------+

The resulting DataFrame, `+----+---------+`, confirms the successful yearly aggregation. For

instance, by manually reviewing the input data, we confirm that the total sales recorded in 2023 (45 + 32 + 47) indeed sum up to 124. This calculated DataFrame is significantly more concise and provides the macro-level insights needed for business reporting.

Specifically, the aggregated results demonstrate the following annual totals:

The total sum of sales recorded for the year 2021 is exactly **48**.
The total sum of sales recorded for the year 2022 is **45**.
The total sum of sales recorded for the year 2023 is **124**.

Understanding the Key Components of the Aggregation Query

A deep understanding of the two primary functions used--`groupBy()` and `agg()`--is crucial for mastering complex PySpark operations. The groupBy method initiates the grouping process, partitioning the DataFrame rows based on the unique values generated by the expression provided within its parentheses: `year('date').alias('year')`. If we had skipped the `year()` function and simply grouped by `'date'`, no aggregation would occur beyond daily totals, as every date in the input is unique.

The `.agg()` method takes the result of the grouping operation and applies the defined aggregation logic. It receives key-value pairs where the key is the aggregation function applied to a column (e.g., `sum('sales')`), and the value is the desired alias for the new aggregated column (e.g., `.alias('sum_sales')`). PySpark allows multiple aggregations to be performed simultaneously within a single `agg()` call, enabling the calculation of total sales, average sales, and maximum sales all in one pass.

When working with time components, the flexibility of PySpark functions extends beyond just `year()`. Functions like `month()`, `dayofmonth()`, or even `date_trunc()` can be used interchangeably within the `groupBy()` clause to achieve aggregations at different temporal resolutions, such as monthly or quarterly summaries. This versatility makes the approach highly adaptable to various analytical requirements.

Implementation Example 2: Counting Records Per Year

While calculating the sum of sales provides insight into overall magnitude, calculating the count of records provides insight into activity frequency. For example, a sharp increase in sales totals might be due to a single large transaction, whereas an increase in the count of sales indicates greater transactional volume or frequency. To calculate the number of transactions per year, we simply substitute the sum function with the count function within the `agg()` method.

This approach maintains the same grouping key (the year extracted from the `date` column) but

alters the resulting metric. The `count()` function tallies the number of non-null records for the specified column (in this case, `sales`) within each group defined by the year. This yields a precise measure of how many transactions occurred annually.

The syntax for calculating the total count of sales, grouped by year, is demonstrated below:

from pyspark.sql.functions import year, count

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

+----+---------+
|year|cnt_sales|
+----+---------+
|2021| 3|
|2022| 2|
|2023| 3|
+----+---------+

The resulting DataFrame clearly indicates the transactional frequency: 3 sales were recorded in 2021, 2 sales in 2022, and 3 sales in 2023. This count metric, when analyzed alongside the total sales metric, provides a more complete picture of the underlying business dynamics. The ability to swap aggregation functions effortlessly showcases the modularity of the PySpark aggregation API.

Conclusion: Flexibility in Time-Based Aggregation

Grouping a PySpark DataFrame by year is an essential operation for time-series analysis, easily accomplished using the combination of the specialized `year()` function and the standard `groupBy().agg()` pattern. Whether the requirement is to calculate sums, counts, averages, or standard deviations, PySpark provides the necessary functions to handle large-scale distributed calculations efficiently.

Mastering this technique allows data professionals to quickly generate high-level summaries from massive, granular datasets, paving the way for advanced analytical tasks, forecasting models, and comprehensive performance reporting. The principles demonstrated here--extracting a temporal component, defining the group key, and applying the desired aggregation--are universally applicable across many PySpark grouping scenarios.

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