PySpark: Add Months to a Date Column

Authored by

Mohammed looti

November 11, 2025

Mastering Date Arithmetic in PySpark

Working with time-series data or logs often requires precise manipulation of date fields within a large-scale data processing framework. In the world of big data, PySpark provides robust tools for handling these operations efficiently. One common requirement is adjusting dates by a specific number of months, whether looking forward (adding) or backward (subtracting). This guide focuses specifically on utilizing the built-in add_months function within a DataFrame context to achieve precise date calculation results.

The ability to perform date arithmetic directly within the distributed environment of Spark is incredibly powerful, as it avoids the need to collect data locally or perform complex user-defined functions (UDFs) for simple temporal shifts. By leveraging optimized PySpark functions, we ensure that these transformations are executed quickly and scalably across the entire cluster, maintaining performance even with massive datasets.

The Core Function: `add_months()` Syntax

The primary tool for this operation is the `add_months` function, which belongs to the `pyspark.sql.functions` module. This function requires two main arguments: the target date column and the integer value representing the number of months to add. A positive integer will move the date forward, while a negative integer will move it backward, allowing for a single function to cover both addition and subtraction requirements.

To utilize this function, you must first import the necessary modules, typically aliasing `pyspark.sql.functions` as `F` for convenience and readability. The following syntax illustrates how you would implement this logic to calculate a new date column based on an existing date field within your DataFrame.

from pyspark.sql import functions as F

df.withColumn('add5months', F.add_months(df, 5)).show()

In this introductory example, we create a new column named `add5months`. This column calculates the date five months after the value found in the original **date** column. The `withColumn` function is essential here, as it allows us to append the result of our transformation as a new column to the existing DataFrame without modifying the original data structure, promoting immutability and clarity in data pipelines.

Setting Up the PySpark Environment and Sample Data

Before applying the date transformation, we must initialize a Spark environment and load the data

into a DataFrame. Establishing the `SparkSession` is the foundational step for any PySpark operation. Once the session is active, we can define our sample data, which in this case represents sales transactions occurring on various dates throughout a year.

For demonstration purposes, let us assume we have a simple dataset containing two columns: `date` (formatted as YYYY-MM-DD) and `sales` (representing the transaction value). This structure is typical for time-series analysis where calculating future or retrospective dates based on transaction dates is a necessary step.

The following code snippet demonstrates the necessary setup, including the creation of the `SparkSession` and the subsequent creation and display of our sample DataFrame:

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|
+----------+-----+
|2023-01-15| 225|
|2023-02-24| 260|
|2023-07-14| 413|
|2023-10-30| 368|
|2023-11-03| 322|
|2023-11-26| 278|
+----------+-----+

Practical Application: Adding a Positive Number of Months

Now that the data is prepared, the requirement is to calculate a future date by adding a specific period--in this demonstration, five months--to every date entry. This calculation might be necessary for forecasting future contract renewal dates, projecting revenue five months out, or determining warranty expiration periods.

To perform this calculation, we again use the `withColumn` transformation along with `F.add_months`. We pass the original **date** column reference and the integer `5` as arguments. PySpark handles the complexities of calendar shifts, including year boundaries (e.g., adding months to a date in October might result in a date in the following year, as seen in the output below).

Executing the transformation yields a new DataFrame where the calculated date field, `add5months`, is appended. Notice how the calculation correctly crosses the year boundary for the sales recorded late in 2023, resulting in dates in early 2024.

from pyspark.sql import functions as F

#add 5 months to each date in 'date' column
df.withColumn('add5months', F.add_months(df, 5)).show()

+----------+-----+----------+
| date|sales|add5months|
+----------+-----+----------+
|2023-01-15| 225|2023-06-15|
|2023-02-24| 260|2023-07-24|
|2023-07-14| 413|2023-12-14|
|2023-10-30| 368|2024-03-30|
|2023-11-03| 322|2024-04-03|
|2023-11-26| 278|2024-04-26|
+----------+-----+----------+

The result clearly shows the successful addition of five months to each corresponding entry in the **date** column. This demonstrates the efficiency and accuracy of using the dedicated PySpark functions for temporal calculations, which is crucial for maintaining data integrity in analytical workflows.

Handling Date Subtraction Using `add_months()`

A significant advantage of the `add_months` function is its versatility in handling subtraction. Instead

of requiring a separate function like `subtract_months`, PySpark simplifies the process by accepting negative integers as the second argument. If, for instance, we needed to look retrospectively at the date five months prior to the recorded sale, we would simply pass `-5` to the function.

This approach maintains code consistency and reduces the complexity of the data processing script. By passing a negative value, we instruct the function to calculate a date that precedes the original date by the absolute value of the integer provided.

The following code demonstrates how to subtract five months from the original **date** column. Note the use of the integer `-5` inside the `add_months()` call, and the resulting column name is changed to `sub5months` to reflect the operation clearly:

from pyspark.sql import functions as F

#subtract 5 months from each date in 'date' column
df.withColumn('sub5months', F.add_months(df, -5)).show()

+----------+-----+----------+
| date|sales|sub5months|
+----------+-----+----------+
|2023-01-15| 225|2022-08-15|
|2023-02-24| 260|2022-09-24|
|2023-07-14| 413|2023-02-14|
|2023-10-30| 368|2023-05-30|
|2023-11-03| 322|2023-06-03|
|2023-11-26| 278|2023-06-26|
+----------+-----+----------+

As expected, the new column `sub5months` reflects the date five months earlier than the original sale date. Importantly, this calculation correctly handles the transition backward across the year boundary, as seen in the first two rows where 2023 dates are shifted back into 2022. This seamless handling of both addition and subtraction reinforces the utility of the `add_months` function for comprehensive date modification tasks.

Understanding the Role of `withColumn` in Data Transformation

In all the examples above, the `withColumn` method is central to the transformation process. In PySpark, dataframes are immutable, meaning they cannot be changed in place. Instead, methods like `withColumn` return a completely new DataFrame reflecting the applied transformation.

The `withColumn` function takes two arguments: the name of the new column (or the existing column to overwrite) and the expression that defines the column's values. In our case, the expression is always `F.add_months(...)`. This approach guarantees that the original data remains safe and unaltered, which is a fundamental principle of functional programming and distributed data processing.

The use of `withColumn` is highly flexible; if you were to pass the name of an existing column, that column would be overwritten with the result of the new expression. However, for auditability and clarity, it is generally recommended to create a new column when performing temporal calculations unless you explicitly intend to replace the source field. The official documentation for the PySpark `withColumn` function offers comprehensive details on its usage and parameters.

Summary of Key Concepts and Best Practices

Successfully manipulating date columns in PySpark hinges on understanding and correctly applying the `add_months` function alongside the `withColumn` transformation. By using dedicated PySpark functions, data engineers can ensure highly efficient, distributed date calculations that automatically manage complex calendar logic, such as leap years and month-end variations.

Key takeaways for efficient date manipulation include:

`F.add_months(column, integer)` handles both addition (positive integer) and subtraction (negative integer).

Always import `pyspark.sql.functions`, typically aliased as `F`.

The transformation is non-destructive, requiring `withColumn` to append the result to a new DataFrame.

This streamlined approach is essential for large-scale data preparation and feature engineering in time-series analysis within the Spark ecosystem.

Additional Resources

For those looking to delve deeper into PySpark's extensive capabilities for date and time handling, the official Apache Spark documentation provides thorough explanations of all available functions and best practices.