How to Extract the Month from a Date in PySpark

Authored by

stats writer

January 19, 2026

Introduction to Date Manipulation in PySpark

Handling temporal data is a fundamental requirement in almost any significant data processing pipeline. When working with large-scale datasets, the analytical power of PySpark, the Python API for Apache Spark, is invaluable. Extracting specific components from a date field, such as the month, is a common operation necessary for tasks like time series aggregation, seasonal analysis, or creating features for machine learning models. This guide provides a comprehensive overview of how to efficiently extract the month from a date column within a PySpark DataFrame, covering both numerical and textual representations of the month.

The core functionality for date and time operations in PySpark resides within the `pyspark.sql.functions` module. This module provides specialized functions that are highly optimized for distributed processing, ensuring performance even when dealing with petabytes of data. For extracting the month as an integer, the dedicated `month()` function is the standard tool.

Understanding how to correctly import and apply these functions is key to successful data preparation. We will explore the specific syntax required, demonstrate practical examples using a sample dataset, and discuss alternative methods for scenarios where the month name (e.g., 'January' instead of '1') is preferred for reporting or visualization purposes. Mastering these techniques will significantly enhance your capabilities in working with complex date structures in the PySpark environment.

Utilizing the PySpark `month()` Function for Numerical Extraction

The most straightforward method to extract the month number from a date column in a PySpark DataFrame is by utilizing the built-in `month()` function. This function is designed specifically for temporal extraction, accepting a column containing date or timestamp data and returning the month index as an integer (1 for January, 12 for December). This is particularly useful when performing mathematical aggregations or filtering based on the month index.

To implement this functionality, you must first import the function from `pyspark.sql.functions`. The operation is typically performed in conjunction with the `withColumn` transformation, which allows for the creation of a new column based on calculations applied to existing data, preserving the integrity of the original dataset.

The following snippet illustrates the essential syntax required to define a new column named 'month' by applying the `month()` function to a specified date column, often named 'date' or 'timestamp'. This particular example creates a new column called **month** that extracts the month from the date in the **date** column.

from pyspark.sql.functions import month

df_new = df.withColumn('month', month(df))

In this structure, the `df_new` DataFrame is created. It includes all existing columns from the original `df`, plus a new column named 'month'. The value in this new column is derived by applying the `month()` extraction function to the values found within the source column, which is specified here as `df`. This method ensures clean and declarative data manipulation within your distributed processing workflow.

Setting Up the PySpark Environment and Sample Data

Before demonstrating the extraction process, we must establish a working PySpark environment and create a sample DataFrame. Our example scenario involves sales data recorded over several days, where the goal is to group or analyze sales based on the month they occurred. We initiate the **SparkSession**, define our sample data, and structure the DataFrame accordingly. The following example shows how to use this syntax in practice.

Suppose we have the following PySpark DataFrame that contains information about the sales made on various days at some company. The sample data contains two fields: 'date' (representing the transaction date) and 'sales' (the numerical sales amount). It is crucial that the 'date' column is correctly interpreted by PySpark as a date or timestamp type for the `month()` function to operate successfully.

The following code block demonstrates the necessary steps for initialization and DataFrame creation, resulting in a structured dataset ready for transformation:

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

# Define sample sales data (date, sales amount)
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|
+----------+-----+
```

The displayed output confirms the successful creation of our base DataFrame, `df`. The 'date' column contains the raw date string, which we now intend to process to isolate the month component for downstream analysis. Suppose we would like to extract the month from each date in the **date** column. We can use the following syntax to do so.

Example: Extracting the Month as an Integer Index

Our primary goal is to append a new column containing the numerical month index derived from the existing 'date' column. This transformation is achieved using the `month()` function combined with the `withColumn` method, as described earlier. This process is highly efficient because PySpark executes the operation in parallel across the cluster.

When working with PySpark transformations, it is important to remember that they are immutable; they always return a new DataFrame rather than modifying the original one in place. We capture this result in `df_new`. The resultant month values will range from 1 to 12, corresponding to the month of the date.

Executing the following command block demonstrates the extraction and the subsequent display of the transformed DataFrame, clearly showing the new 'month' column populated with the correct integer values:

```
from pyspark.sql.functions import month
```

```
# extract month from date column
df_new = df.withColumn('month', month(df))

# view new DataFrame
df_new.show()

+----------+-----+-----+
| date|sales|month|
+----------+-----+-----+
|2023-04-11| 22| 4|
|2023-04-15| 14| 4|
|2023-04-17| 12| 4|
|2023-05-21| 15| 5|
|2023-05-23| 30| 5|
|2023-10-26| 45| 10|
|2023-10-28| 32| 10|
|2023-10-29| 47| 10|
+----------+-----+-----+
```

The new **month** column contains the month of each date in the **date** column. As observed in the result, the new 'month' column accurately reflects the numerical month index for each record. This numerical representation is ideal for subsequent data manipulation, such as calculating average sales per quarter or comparing month-over-month performance metrics.

Alternative Formatting: Extracting the Month Name using `date_format()`

While numerical extraction is useful for computational tasks, data visualization and reporting often require the full name of the month (e.g., 'April', 'May') for better human readability. Also note that you could use the `date_format` function if you'd like to return the name of the month instead. This function allows users to specify an output format pattern for any date or timestamp column.

The `date_format()` function takes two primary arguments: the column to be formatted and a string specifying the desired output pattern. To retrieve the full month name, we use the format specifier `'MMMM'`. If you needed the abbreviated month name (e.g., 'Oct'), you would use `'MMM'`. This flexibility makes it a powerful tool for tailored date presentation.

The syntax for using `date_format()` maintains the standard structure of transformations within a DataFrame. It requires importing the necessary functions from `pyspark.sql.functions` and applying the operation via `withColumn` to generate the resulting textual column, providing a streamlined approach compared to manual string manipulation.

Example: Generating the Full Month Name

To demonstrate the extraction of the month name, we reuse the original `df` and apply the `date_format()` function. For convenience, we import all functions using the wildcard `*` from `pyspark.sql.functions`.

The format string `'MMMM'` is the critical component here, instructing PySpark to convert the numerical month index into the corresponding full English month name. This results in a column of type string, replacing the integer representation we achieved with the `month()` function.

Observe the transformation below, where the `df_new` DataFrame now features month names instead of month numbers:

from pyspark.sql.functions import *

# extract month name from date column
df_new = df.withColumn('month', date_format('date', 'MMMM'))

# view new DataFrame
df_new.show()

```
+----------+-----+-------+
| date|sales| month|
+----------+-----+-------+
|2023-04-11| 22| April|
|2023-04-15| 14| April|
|2023-04-17| 12| April|
|2023-05-21| 15| May|
|2023-05-23| 30| May|
|2023-10-26| 45|October|
|2023-10-28| 32|October|
|2023-10-29| 47|October|
+----------+-----+-------+
```

The new **month** column now contains the name of the month for each date in the **date** column. This type of textual output is invaluable for creating human-readable reports and ensures that downstream users can immediately understand the temporal context of the data without needing to reference a numerical key.

Deep Dive: The Importance of `withColumn`

The operation of extracting the month, regardless of whether you use `month()` or `date_format()`, relies fundamentally on the `withColumn` transformation method. Note that we used the **withColumn** function to add a new column called **month** to the DataFrame while keeping all existing columns the same.

The primary purpose of `withColumn` is to add a new column to a DataFrame or replace an existing one, based on a defined column expression. When defining a new column, `withColumn` takes two arguments: the name of the target column and the column expression that defines its values. By passing the result of a function like `date_format(df, 'MMMM')` as the expression, we ensure that the calculation is executed efficiently across all partitions of the data.

Using `withColumn` adheres to the immutable nature of PySpark DataFrames, ensuring that the original data is preserved while a new transformed DataFrame is returned. This is essential for traceable and repeatable data processing pipelines. If a column name already exists, `withColumn` gracefully handles the replacement, making schema updates straightforward.

For developers seeking deeper understanding, the official PySpark documentation for the **withColumn** function provides comprehensive details on its usage and optimization strategies.

Conclusion: Choosing the Right Date Extraction Method

Extracting the month from a date field in PySpark is a fundamental yet essential operation, made simple by the rich library of optimized functions available in `pyspark.sql.functions`. The choice between methods depends entirely on the required output format and the subsequent use case for the extracted data.

The two primary methods offer distinct advantages:

For Numerical Output (1-12): Use the `month()` function. This is preferred for aggregations, mathematical comparisons, or features requiring integer input, as it results in a highly efficient numeric data type.

For Textual Output (Month Name): Use the `date_format()` function with the `'MMMM'` pattern. This is ideal for user-facing reports or data quality checks where readability is paramount, resulting in a string data type.

By leveraging these built-in functions in conjunction with the powerful `withColumn` transformation, data analysts can efficiently prepare and enrich their temporal data for complex distributed analysis within the PySpark DataFrame structure.