Learning PySpark: Extracting the Month from Date Columns in DataFrames

Authored by

Mohammed looti

November 11, 2025

Mastering Date Extraction in PySpark

Processing temporal data is a fundamental requirement in nearly all data engineering and analysis pipelines. When working within the distributed computing framework of PySpark, efficiently handling date and time structures stored within a DataFrame is essential for deriving meaningful insights. One of the most common transformation tasks is extracting specific components from a date field, such as the month, year, or day. This article focuses specifically on how to extract the month number from a date column using built-in PySpark SQL functions, providing both the concise solution and detailed examples to facilitate robust data preparation.

The primary utility we leverage for this operation is the dedicated **month** function, found within the PySpark DataFrame API. This function offers a straightforward, optimized approach to isolating the month component, returning it as an integer value (1 through 12). Understanding the correct syntax for importing and applying this function is crucial for leveraging the full power of Spark's distributed capabilities, ensuring that date transformations are performed efficiently across large datasets.

To implement this extraction, you must first import the required function from `pyspark.sql.functions`. The resulting transformation typically involves creating a new column in your existing DataFrame using the powerful **withColumn** method. This method allows for the creation of derived columns based on calculations applied to existing fields without modifying the source data structure. The core syntax for extracting the month from a date column in a DataFrame looks like this:

from pyspark.sql.functions import month

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

This specific instruction creates a new column named **month** within the `df_new` DataFrame. It extracts the month value directly from the source column, here assumed to be named **date**, demonstrating the simplicity and efficiency of using the built-in PySpark functions for time series feature engineering.

Practical Implementation: Setting Up the Data Environment

Before performing any transformation, it is necessary to establish a working PySpark session and define the source DataFrame. In a real-world scenario, this DataFrame would likely be loaded from a large data source such as Parquet, Delta Lake, or CSV files. For the purpose of this tutorial, we will instantiate a simple example DataFrame containing fictional sales data linked to specific transaction dates. This dataset mimics typical business data where temporal analysis is required.

The process begins by initializing the SparkSession, which serves as the entry point for using

Spark functionality. Once the session is active, we define our sample data--a list of rows containing a date string and a corresponding sales figure. We then define the schema by specifying the column names: **date** and **sales**. Finally, we use the `createDataFrame` method to materialize the data into a schema-aware DataFrame named `df`.

The following code block demonstrates the necessary steps to create and display our initial dataset, which will be the basis for our month extraction task. Note the careful definition of data types implied by the input structure, although PySpark often infers string types for the date column initially, which is suitable for the **month** function to operate on, provided the string format is standard ('YYYY-MM-DD').

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

# Define sample data: date and sales figures
data = ,
,
,
,
,
,
,
]

Define column names for the DataFrame
columns =

# Create the DataFrame
df = spark.createDataFrame(data, columns)

# View the resulting DataFrame structure
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|
+----------+-----+

Executing the Extraction: Applying the `month` Function

With the source DataFrame successfully created, the next step is applying the transformation required to extract the month number. The goal is to iterate through every row in the **date** column and calculate the corresponding month value, appending this result as a new field. This operation is typically performed using the **withColumn** method because it maintains the immutability of the original DataFrame (`df`) and generates a new transformed DataFrame (`df_new`).

The **month** function works seamlessly with PySpark's handling of date strings or explicit DateType columns. When supplied with the name of the column containing the date information, it parses the date structure and returns the month index (1 for January, 12 for December). This is an essential step for operations such as grouping sales data by month, calculating monthly averages, or performing time-series forecasting where monthly periodicity is a critical feature.

Observe the execution below. We explicitly import the **month function** and then call withColumn, passing 'month' as the name of the new column and `month(df)` as the calculation expression. This illustrates the clean, declarative syntax that PySpark encourages for large-scale data manipulation, ensuring that the computation is optimized and distributed across the cluster.

from pyspark.sql.functions import month

```
# Apply the month function to the date column and create a new column
df_new = df.withColumn('month', month(df))

# View the new DataFrame with the extracted month column
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|
+----------+-----+-----+

As visible in the resulting output, the new **month** column successfully contains the numerical month extracted from the **date** column. This transformed DataFrame, `df_new`, is now ready for aggregation or further analysis based on monthly periods.

Advanced Extraction: Retrieving Month Names Using `date_format`

While extracting the numerical month is often sufficient for internal calculations, sometimes data visualization or reporting requires the full name of the month (e.g., "April" instead of "4"). PySpark provides the highly versatile **date_format** function specifically for controlling the output format of temporal fields. This function accepts two arguments: the column containing the date and a pattern string defining the desired output format.

To achieve the full month name, we utilize the formatting pattern `MMMM`. Standard SQL date formatting patterns, such as those used in Java's SimpleDateFormat, are supported by PySpark. Using `MMMM` ensures that the full textual representation of the month is returned, which significantly enhances the readability of reports and user-facing dashboards. For abbreviated month names (e.g., "Apr"), the pattern `MMM` would be used instead, offering flexibility based on the analytical requirement.

The implementation below demonstrates how to substitute the **month** function with the more versatile **date_format** function, again leveraging withColumn to append the newly formatted column. We use the wildcard import `import *` for brevity, although importing only **date_format** is generally recommended practice in production environments for code clarity. Note how the resulting output structure remains the same, but the data type and content of the **month** column are now textual.

from pyspark.sql.functions import *

# Extract month name using date_format with 'MMMM' pattern
df_new = df.withColumn('month', date_format('date', 'MMMM'))

# View the 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|
+----------+-----+-------+

This resulting DataFrame clearly shows the month names corresponding to the dates in the original column. This demonstrates the powerful capabilities of date_format for customizing temporal feature engineering based on specific analytical or reporting needs, moving beyond simple numerical extraction.

Deep Dive into `withColumn`: The Foundation of Transformations

Central to both examples provided is the use of the withColumn method. Understanding its role is critical for anyone performing data manipulation in PySpark. The primary purpose of withColumn is to add a new column to an existing DataFrame or replace an existing column with the same name, based on the results of a specified expression. In our context, we used it to calculate the month value (either numerical or textual) and assign it to the new **month** column, preserving the original **date** and **sales** columns unchanged.

The functional programming paradigm of Spark means that withColumn does not mutate the original DataFrame. Instead, it returns a new, transformed DataFrame. This principle of immutability is fundamental to ensuring fault tolerance and predictability in distributed processing environments. If we had chosen a column name that already existed, withColumn would automatically overwrite the content of that column with the new calculation, allowing for easy updates or corrections to existing features.

The syntax is straightforward: `df.withColumn(colName, col)`, where `colName` is a string defining the new column name, and `col` is a Column object representing the calculation (e.g., `month(df)` or `date_format('date', 'MMMM')`). Mastering this function unlocks the ability to chain multiple transformations together efficiently, allowing data engineers to build complex ETL pipelines using clear, expressive code. For detailed documentation on all parameters and capabilities, refer to the official PySpark withColumn documentation.

Conclusion and Further Resources

Extracting temporal components like the month from date columns is a routine yet vital task in data preparation using PySpark. By utilizing specialized functions such as **month** for numerical output or **date_format** for textual output, coupled with the foundational withColumn method, data professionals can efficiently enrich their datasets for downstream analytical models and reporting tools. The distributed nature of PySpark ensures that these date transformations scale effectively, regardless of the size of the input data.

To continue mastering advanced date and time manipulations in PySpark, exploring related functions is highly recommended. Functions such as `year()`, `dayofmonth()`, `quarter()`, and `to_date()` offer comprehensive tools for handling various temporal requirements. Understanding how to correctly apply these SQL functions within the DataFrame API is crucial for building robust and scalable data processing solutions.

Additional Resources for PySpark Development

For readers interested in expanding their knowledge of PySpark and complex data transformations, the following resources provide authoritative guidance:

Official Apache Spark Documentation: Comprehensive guides on DataFrame operations and SQL functions.
PySpark API Reference: Detailed documentation for functions like date_format and month function.
Tutorials on Distributed Computing: Resources focusing on optimization and performance tuning of Spark jobs.