How to Extract the Month from a Date in PySpark

Authored by

stats writer

February 5, 2026

Extracting the month from a date field is a foundational requirement in almost any time-series or transactional data analysis. In the realm of big data processing, particularly when leveraging PySpark, this operation must be performed efficiently across massive datasets. The ability to obtain the numerical representation of the month, or even its full name, is crucial for subsequent operations like filtering, grouping, aggregation, and generating business intelligence reports. PySpark provides highly optimized, built-in functions designed specifically for date and time manipulation, ensuring that complex data transformations are handled with speed and accuracy. This detailed guide will walk you through the essential methods for extracting month information from date columns within a PySpark DataFrame, focusing on clarity, efficiency, and real-world application.

PySpark Essentials: Extracting Time Components from DataFrames

When working with temporal data in PySpark, the primary structure utilized is the DataFrame. PySpark extends the capabilities of traditional SQL and Pandas by offering a dedicated module, `pyspark.sql.functions`, which houses powerful utility functions for data manipulation. To extract the numerical month from a column containing date values, we primarily rely on the dedicated `month` function. This approach is highly efficient because the processing is pushed down to the optimized Spark execution engine, benefiting from distributed computing.

The general syntax for incorporating the extracted month into your existing DataFrame involves using the `withColumn` method. This method allows the creation of a new column without altering the structure or content of the original dataset columns. Below demonstrates the fundamental structure required to import the necessary function and perform the transformation:

from pyspark.sql.functions import month

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

This snippet utilizes the `month` function, imported from the SQL functions module, to calculate the month value based on the input column (here generalized as `date_column`). The result is stored in a new column, `month_number`, which is attached to the existing DataFrame, `df`, resulting in the new DataFrame, `df_new`.

Detailed Walkthrough: Setting Up the Environment and Data

Before performing any transformations, it is essential to establish a PySpark session and define the source data. For demonstration purposes, we will simulate a transactional dataset containing sales records. This dataset includes a `date` column, which is the focus of our extraction task, and a corresponding `sales` column. Setting up the environment properly ensures that Spark is initialized

and ready to handle distributed operations.

Initialization begins with importing the `SparkSession`, which is the entry point for using Spark functionality. We then define the sample data structure using Python lists, specifying both the date (formatted as YYYY-MM-DD strings, which Spark interprets as dates or timestamps) and the corresponding sales volume. This meticulous setup mirrors real-world scenarios where data is loaded from sources like CSV, Parquet, or databases, often requiring subsequent time-series analysis.

The following code block demonstrates the necessary steps to define the data, specify the schema (column names), and materialize the initial PySpark DataFrame, which will serve as the source for our month extraction procedures:

from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MonthExtraction").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|
+----------+-----+
|2023-04-11| 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|
+----------+-----+

As observed in the output, the initial DataFrame, `df`, contains two columns: `date`, holding the temporal information, and `sales`, holding the associated numeric values. Our immediate objective is to derive the numerical month value (e.g., 4 for April, 5 for May) from the `date` column.

Method 1: Extracting the Numerical Month using the `month()` Function

The most straightforward and highly recommended way to extract the numerical month index (1 through 12) from a date column in PySpark is by employing the dedicated `month` function. This function is part of the robust `pyspark.sql.functions` library and is designed to handle Spark's internal date and timestamp types seamlessly. Using this function ensures correct handling of time zone complexities and null values inherent in large-scale datasets.

To integrate this extraction into our workflow, we use the `withColumn` transformation. This technique allows us to specify the name of the new column we wish to create (in this case, `month`) and define its calculation based on existing columns. The syntax is concise and declarative, reflecting Spark's functional programming principles. We apply the `month` function directly to the column reference, `df`.

Executing this transformation creates a derived column containing the numerical month representation. This format is particularly useful for numerical computations, such as calculating average sales per month or defining time-based windows for machine learning features. Observe the implementation and the resulting DataFrame structure below, confirming that the month indices are correctly derived from the source dates:

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 resulting `df_new` DataFrame successfully includes the new `month` column. The values (4, 5, 10) correspond accurately to April, May, and October, respectively. This newly derived column is now ready to be used for complex aggregations or joins, enabling powerful time-based data segmentation within your PySpark pipeline.

Method 2: Extracting the Month Name using the `date_format()` Function

While numerical month extraction is vital for computation, there are numerous analytical and reporting situations where the full textual name of the month is preferred for readability and presentation. PySpark caters to this requirement through the highly versatile `date_format` function. Unlike the `month` function, which only returns an integer, `date_format` allows developers to specify output patterns, converting the date object into a formatted string.

The power of `date_format` lies in its reliance on standard SQL date formatting patterns. To extract the full month name, we use the pattern code `MMMM`. If only the abbreviated month name is required (e.g., Jan, Feb), the pattern code `MMM` would be used instead. This flexibility makes `date_format` an indispensable tool for customizing output formats based on specific reporting needs.

Implementation involves importing all functions from `pyspark.sql.functions` (using `import *` for brevity, although importing specific functions is often better practice) and then applying `date_format` within the `withColumn` context, providing the input column and the desired format pattern as arguments. This results in a textual column that is ideal for final user-facing reports or categorical grouping analyses.

from pyspark.sql.functions import*#extract month name from date column
df_new = df.withColumn('month_name', date_format('date', 'MMMM'))

#view new DataFrame
df_new.show()

+----------+-----+----------+

```
| date|sales|month_name|
+----------+-----+----------+
|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_name` column clearly contains the textual representation of the month derived from the original date column. This representation is much more intuitive for human analysts reviewing the data. Note that we utilized the `withColumn` function again, demonstrating its power in adding computed columns to the DataFrame seamlessly.

Understanding PySpark Date Format Patterns

To effectively utilize the `date_format` function, one must be familiar with the standard formatting codes used by Spark SQL, which are largely inherited from Java's SimpleDateFormat standard. These codes dictate exactly how the date object is parsed and represented as a string. Understanding these patterns is critical, as a small change in capitalization or repetition can drastically alter the output.

For month extraction specifically, there are three primary codes that data engineers frequently employ, depending on the required level of detail and formatting:

`MM`: This pattern returns the numerical representation of the month, zero-padded (e.g., "04" for April, "10" for October). While the `month()` function provides the non-padded integer, `date_format(col, 'MM')` provides a string representation, which can be useful when merging with other string identifiers.

`MMM`: This pattern returns the abbreviated textual name of the month (e.g., "Apr," "May," "Oct"). This is often used in condensed dashboards or charts where space is limited but readability is still important.

`MMMM`: This pattern, as demonstrated in Method 2, returns the full, unabbreviated textual name of the month (e.g., "April," "May," "October"). This is preferred for formal reporting.

The choice between these formats should be driven by downstream requirements. If the data will

be used for numerical sorting or mathematical operations, the integer output from the `month` function is optimal. If the column is strictly for presentation or categorical grouping, the string outputs from `date_format` provide superior clarity.

Advanced Applications: Grouping and Filtering by Month

The true value of extracting the month component lies in its application for analytical tasks. Once the month is extracted--either numerically or textually--it can be used as a key for powerful aggregation operations. This allows us to move beyond row-level detail and analyze performance, trends, or anomalies across monthly boundaries. For example, a common requirement is calculating the total sales achieved in each month across the entire dataset.

To calculate total monthly sales, we first perform the month extraction (using the numerical representation for easier sorting/comparison), and then apply the standard PySpark grouping mechanism using `groupBy`, followed by an aggregation function like `sum`. This capability is fundamental to building summary statistics and key performance indicators (KPIs) in data processing pipelines.

Furthermore, the extracted month column simplifies conditional filtering. Suppose an analyst only needs to examine data corresponding to Q2 (April, May, June). By using the numerical month column, a simple filter operation (`df_new.filter("month >= 4 AND month <= 6")`) can isolate the necessary subset efficiently. This targeted querying is essential for optimizing query performance on massive DataFrames.

Efficiency Considerations and Best Practices

When dealing with large volumes of data characteristic of Apache Spark environments, optimizing performance is paramount. Using PySpark's built-in functions, such as `month()` and `date_format()`, is considered a best practice because these functions are highly optimized and executed natively within the JVM (Java Virtual Machine) via Spark SQL catalyst optimizer, minimizing the overhead associated with Python interpretation.

It is crucial to ensure that the input column being processed is recognized by Spark as a proper Date or Timestamp type. If the date column is mistakenly treated as a generic String type, operations like `month()` may fail or produce incorrect results. If necessary, always explicitly cast or convert string date columns using functions like `to_date()` before performing extraction.

Finally, when choosing between numerical extraction (`month()`) and string extraction (`date_format('MMMM')`), consider the subsequent steps. Numeric columns are generally faster to sort and filter than string columns. If the goal is purely analytical or computational, favor the numeric output. If the column is solely for final output formatting, the string format provides better

human readability.

Summary of PySpark Date Extraction Functions

PySpark offers a wealth of date and time functions beyond just month extraction. These tools enable comprehensive manipulation of temporal data, ensuring that every time component--year, day, hour, etc.--can be isolated and analyzed independently. Mastery of these functions is key to becoming a proficient PySpark data engineer.

Here is a quick reference of related functions often used alongside `month()`:

`year(col)`: Extracts the year as an integer.

`dayofmonth(col)`: Extracts the day of the month (1-31).

`quarter(col)`: Extracts the quarter of the year (1, 2, 3, or 4).

`weekofyear(col)`: Extracts the week number of the year (1-53).

`to_date(col, format)`: Converts a string column to a Date type using a specified format.

By leveraging these functions, data practitioners can build resilient and flexible data models capable of handling complex time-series data challenges in a distributed environment. Extracting the month, whether numerically or as a formatted string, remains a fundamental step in this process.