How to Convert a Date Column to a String in PySpark

Authored by

stats writer

February 4, 2026

The need to convert a column from a date format to a string format is a common requirement in data processing, especially when preparing data for reporting, serialization, or integration with external systems that rely on specific text representations of time data. In the context of PySpark, this conversion is efficiently handled using specialized functions available within the `pyspark.sql.functions` module. Unlike simple casting, which might result in an undesired default string representation, using functions like `date_format` allows the user granular control over the output format.

By transforming date values into strings formatted according to precise specifications (e.g., "YYYY-MM-DD" or "MM/dd/yyyy"), we enable easier manipulation, analysis, and standardization of the data. For instance, once a date column is converted to a string, it can be concatenated with other columns or processed using standard PySpark string functions, which often proves more flexible for certain business logic requirements than working directly with specialized date objects. This flexibility is vital when dealing with large-scale DataFrames and complex ETL pipelines.

PySpark: Convert Column from Date to String

Introduction: The Necessity of Data Type Conversion in PySpark

Working with heterogeneous datasets in a distributed environment like Apache Spark necessitates careful management of column data types. While the native `DateType` in Spark is optimized for date arithmetic and temporal indexing, there are numerous scenarios where a string format is required. Common use cases include generating fixed-width files for legacy systems, exporting data to non-SQL environments, or simply enforcing a consistent display format for reporting dashboards.

The primary method for achieving a controlled date-to-string transformation in PySpark is through the `date_format` function. This powerful function allows developers to specify precisely how the date components (year, month, day) should be rendered into the resulting text string, ensuring compliance with global or internal date presentation standards, such as ISO 8601 or US-specific formats. This level of control is essential for maintaining data integrity and ensuring interoperability across different data platforms.

By leveraging the built-in functions provided by the `pyspark.sql.functions` module, we can perform this conversion efficiently across vast amounts of data without compromising performance, which is crucial for large-scale data engineering tasks. The following sections will detail the specific syntax and provide a comprehensive, runnable example demonstrating this fundamental transformation within a DataFrame structure.

The Primary Tool: Utilizing the date_format Function

To perform this transformation, PySpark developers rely on the `date_format` function. This function takes two mandatory arguments: the column containing the date values you wish to convert, and a format pattern string that defines the desired output structure. The format string uses specific character sequences--such as `'yyyy'` for the four-digit year, `'MM'` for the two-digit month, and `'dd'` for the two-digit day--to dictate the exact representation of the date components.

It is important to understand that `date_format` does not merely change the underlying data type; it processes the temporal value and reconstructs it as a new string format based on the pattern provided. This distinction is critical compared to a simple `.cast("string")` operation, which generally yields the default ISO 8601 string representation (e.g., "YYYY-MM-DD") regardless of user preference. By using `date_format`, we gain the ability to output formats like "30-Oct-2023" or "10/30/2023", tailoring the data precisely for its end use.

The function is typically used in conjunction with the `withColumn` transformation, which allows for the creation of a new column in the DataFrame while preserving the original date column, thereby maintaining traceability and providing flexibility during intermediate steps of a pipeline. The following syntax block illustrates the standard implementation structure for this essential operation.

Syntax Implementation for Date-to-String Conversion

The conversion is executed by importing the necessary function from `pyspark.sql.functions` and then applying it within a `withColumn` call on the target DataFrame. The core logic involves specifying the input column name and the desired output pattern, ensuring the new column captures the date information in the required string format.

You can use the following syntax to convert a column from a date to a string in PySpark, creating a new column to hold the formatted output:

from pyspark.sql.functions import date_format

df_new = df.withColumn('date_string', date_format('date', 'MM/dd/yyyy'))

This particular example demonstrates converting the dates residing in the column named **date** to strings, placing them into a new column called **date_string**. Crucially, the specified pattern `'MM/dd/yyyy'` ensures that the dates are rendered in the common US format, with the two-digit month followed by the day and the four-digit year. This pattern string is highly customizable and represents the core control mechanism for the output appearance of the date data.

Practical Example Setup: Initializing the PySpark Environment

To fully illustrate the conversion process, we will begin by establishing a Spark session and creating a sample DataFrame containing sale records, where the date column is explicitly defined using the native Spark `DateType`. This foundational step mirrors real-world scenarios where data is ingested from sources like databases or CSV files, often retaining its temporal data type.

Suppose we have the following PySpark DataFrame that tracks sales figures recorded on various days for a hypothetical company. Note the use of Python's datetime module to ensure the initial data is correctly instantiated as date objects:

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

#define data
data = ,
,
,
,
]

#define column names
columns =

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

#view dataframe with full column content
df.show()

+----------+-----+
| date|sales|
+----------+-----+
|2023-10-30| 136|
|2023-11-14| 223|
|2023-11-22| 450|
|2023-11-25| 290|
|2023-12-19| 189|
+----------+-----+

This resulting DataFrame, `df`, provides the perfect foundation for demonstrating the conversion process. Although the dates appear in the standard "YYYY-MM-DD" format in the output of `df.show()`, their internal representation is still a Spark `DateType`, which we will confirm in the next step.

Verifying the Initial Data Types

Before proceeding with any transformation, it is standard practice to inspect the schema of the DataFrame. Understanding the current data types is essential for selecting the correct conversion function and ensuring the operation targets the intended column. In PySpark, the `dtypes` attribute of the DataFrame provides this schema information as a list of tuples, where each tuple contains the column name and its corresponding data type string.

We use the following command to check the data type of each column in our initial DataFrame:

#check data type of each column
df.dtypes

The output clearly indicates that the **date** column currently holds a data type of **date**, confirming it is ready for conversion using `date_format`. The **sales** column is correctly identified as a **bigint**. This verification step is a crucial checkpoint in any data pipeline, preventing runtime errors that arise from attempting to apply date functions to string columns, or vice versa.

Executing the Conversion and Reviewing Results

Now that we have confirmed the initial data type, we proceed with the core operation. We utilize the `withColumn` transformation to introduce a new column, `date_string`, whose values are derived by applying the `date_format` function to the existing `date` column, specifying the desired `'MM/dd/yyyy'` string format.

This approach ensures that the original data is preserved while the transformed data is added, adhering to best practices for non-destructive data manipulation:

from pyspark.sql.functions import date_format

#create new column that converts dates to strings
df_new = df.withColumn('date_string', date_format('date', 'MM/dd/yyyy'))

#view new DataFrame
df_new.show()

```
+----------+-----+-----------+
| date|sales|date_string|
+----------+-----+-----------+
|2023-10-30| 136| 10/30/2023|
|2023-11-14| 223| 11/14/2023|
|2023-11-22| 450| 11/22/2023|
|2023-11-25| 290| 11/25/2023|
|2023-12-19| 189| 12/19/2023|
+----------+-----+-----------+
```

Upon viewing `df_new`, we can clearly observe the successful creation of the `date_string` column. The values, such as '2023-10-30', have been correctly converted to the text representation '10/30/2023', demonstrating the precise control afforded by the format pattern. This new column is now suitable for use in applications or systems that specifically require date information to be presented as a formatted string.

Post-Conversion Data Type Confirmation and Flexibility

The final step in this process is to re-verify the schema to ensure that the newly created column has, in fact, been assigned the `StringType` data type, thereby confirming the success of our conversion operation. This verification guarantees that subsequent string-based operations can be performed on `date_string` without error.

We use the `dtypes` attribute once again to view the schema of the newly transformed DataFrame, `df_new`:

```
#check data type of each column
df_new.dtypes
```

As confirmed by the output, the `date_string` column now officially possesses the **string** data type, validating that we have successfully created a string column derived from a date column, formatted according to the specification. It is crucial to remember that the format string used (e.g., `MM/dd/yyyy`) is entirely customizable; users may choose any valid date format specifiers based on regional standards or output system requirements, such as `yyyy-MM-dd HH:mm:ss` if time components were also present in the original data.

This process highlights the robust capabilities of PySpark for handling essential data manipulation tasks in a structured and scalable manner. We have successfully navigated the transformation from a date object designed for computation back into a textual representation suitable for diverse

consumption patterns.

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