Convert Timestamp to Date in PySpark (With Example)

Authored by

stats writer

November 16, 2025

Introduction: The Necessity of Type Conversion in PySpark

Working with large-scale datasets often requires meticulous attention to data types, especially when dealing with temporal information. In the context of big data processing, such as within an PySpark environment, managing time series data accurately is paramount for reliable analytics and reporting. A common requirement is the conversion of a timestamp, which typically includes both date and time components (down to seconds or milliseconds), into a simpler date format, which only retains the year, month, and day.

This conversion is crucial for operations such as grouping data by day, aggregating daily sales figures, or partitioning data storage efficiently. While a timestamp provides granular detail, many business intelligence tasks only need the date component. Using the appropriate data type ensures that subsequent transformations and functions operate correctly, avoiding potential errors and improving performance within the distributed computing framework provided by Apache Spark.

This guide provides a comprehensive, step-by-step methodology for converting a timestamp column into a date column within a PySpark DataFrame. We will focus on the most reliable and idiomatic PySpark syntax, ensuring clarity and efficiency for data engineers and analysts alike.

Understanding PySpark Data Types for Date and Time

Before diving into the conversion process, it is essential to understand how PySpark manages temporal data. PySpark utilizes specific internal types mapped to standard SQL types to ensure consistency and optimization across the cluster. The two primary types relevant here are **TimestampType** and **DateType**.

The **TimestampType** stores points in time, representing an instant defined relative to the epoch (usually January 1, 1970, 00:00:00 UTC). It typically includes precision up to microseconds, making it suitable for logging, event tracking, and detailed time-series analysis where the exact time of occurrence is critical. In contrast, the DateType is a simpler type designed to store calendar dates, omitting the time component entirely. It simplifies comparison and grouping operations based purely on the day.

The process of converting from **TimestampType** to **DateType** inherently involves truncation--dropping the hour, minute, and second components while preserving the calendar date. This distinction is vital, as attempting to apply date-specific functions to a timestamp column, or vice versa, without explicit conversion can lead to unexpected results or execution failures.

The Essential Conversion Method: Using .cast()

The most direct and foundational method for altering the data type of a column in PySpark is by

utilizing the built-in `.cast()` function. This function allows a column to be explicitly converted from one data type to another, provided the conversion is logically supported. When converting a timestamp to a date, the `.cast()` function handles the necessary truncation seamlessly.

To perform this operation, you must import the specific data type class you wish to cast the column to, which in this case is `DateType` from the `pyspark.sql.types` module. The operation is typically executed using the withColumn method, which allows you to either create a new column or overwrite an existing one with the result of the cast operation.

The core syntax for converting a timestamp column to a date column within a DataFrame is demonstrated below. Note the use of the `.cast()` method applied directly to the existing timestamp column:

from pyspark.sql.types import DateType

df = df.withColumn('my_date', df.cast(DateType()))

In this powerful yet concise example, we instruct PySpark to create a new column named **my_date**. The values for this new column are derived by taking the values from the existing **my_timestamp** column and explicitly casting them into the desired DateType. This methodology ensures that the time component is correctly discarded, leaving behind only the date information suitable for date-based analysis.

Practical Demonstration Setup: Creating the Initial DataFrame

To illustrate the conversion process effectively, we will construct a sample DataFrame containing simulated sales data, complete with a timestamp column tracking the exact time of each transaction. This demonstration mirrors a typical scenario encountered in real-world data engineering tasks where raw time data needs standardization.

First, we must initialize a **SparkSession**, which is the entry point to using PySpark functionality. We then define our raw data, which initially contains time information stored as a string, and define the column schema. Crucially, we use the `F.to_timestamp()` function to explicitly convert the initial string column into a proper **TimestampType** column. This step is often necessary when data is loaded from sources like CSV or JSON, where temporal data might initially be parsed as plain text strings.

The following code block sets up our environment, defines the sample data, and performs the initial necessary string-to-timestamp conversion, making the data ready for the final date conversion step.

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

from pyspark.sql import functions as F

# Define data containing timestamp strings and sales figures
data = ,
,
,
]

# Define column names
columns =

# Create DataFrame using data and column names
df = spark.createDataFrame(data, columns)

# Convert string column 'ts' to proper timestamp type
df = df.withColumn('ts', F.to_timestamp('ts', 'yyyy-MM-dd HH:mm:ss'))

# View the resulting DataFrame
df.show()

+-------------------+-----+
| ts|sales|
+-------------------+-----+
|2023-01-15 04:14:22| 225|
|2023-02-24 10:55:01| 260|
|2023-07-14 18:34:59| 413|
|2023-10-30 22:20:05| 368|
+-------------------+-----+
```

As seen in the output, the `ts` column now holds the full timestamp value, representing the exact moment of each sale. Our goal is to isolate only the calendar date from these values.

Initial Data Inspection and Type Validation

Before proceeding with the conversion, verifying the current data types is a critical debugging step. Ensuring that the source column is correctly identified as a timestamp guarantees that the subsequent conversion operation will execute as intended. In PySpark, we can use the `.dtypes` attribute of the DataFrame to quickly inspect the schema and data types of all columns.

Executing the `.dtypes` command on our newly created DataFrame confirms the current structure:

```
# Check data type of each column
df.dtypes
```

The inspection confirms that the `ts` column is correctly recognized as a **timestamp** type, while the `sales` column is an integer type (`bigint` in Spark SQL terms). Having validated the source type, we are now ready to apply the date conversion logic. This proactive validation helps prevent runtime errors that might occur if the column were still mistakenly treated as a string or another incompatible type.

Implementing the Timestamp to Date Conversion

The conversion process is executed by combining the withColumn method with the `.cast(DateType())` expression. We choose to create a new column, `new_date`, to preserve the original timestamp data for comparison and potential further analysis. This is generally considered a best practice in data transformation workflows, ensuring data lineage and reversibility.

The transformation logic is straightforward: we reference the existing `ts` column and apply the .cast() function, instructing it to convert the data into the DateType. The PySpark engine efficiently performs this operation across the distributed partitions of the DataFrame.

```
from pyspark.sql.types import DateType

# Create date column from timestamp column using .cast(DateType())
df = df.withColumn('new_date', df.cast(DateType()))

# View updated DataFrame
df.show()
```

```
+-------------------+-----+----------+
| ts|sales| new_date|
+-------------------+-----+----------+
|2023-01-15 04:14:22| 225|2023-01-15|
|2023-02-24 10:55:01| 260|2023-02-24|
|2023-07-14 18:34:59| 413|2023-07-14|
|2023-10-30 22:20:05| 368|2023-10-30|
+-------------------+-----+----------+
```

The resulting output clearly demonstrates the success of the conversion. The new `new_date`

column retains the original calendar date but entirely omits the time components (hours, minutes, seconds), which were present in the source `ts` timestamp column. This new column is now optimally formatted for date-based aggregations.

Verifying the New Data Type

The final step in any transformation workflow is to confirm that the output column possesses the expected data type. We use the `.dtypes` attribute once more to inspect the schema of the modified DataFrame, focusing specifically on the `new_date` column.

This verification confirms that our intent--to assign the **DateType**--was successfully executed by the Spark engine, ensuring that this column is treated as a date in all subsequent transformations and queries.

 # Check data type of each column after conversion
df.dtypes

As confirmed by the schema output, the `new_date` column now proudly holds the desired **date** data type. We have successfully completed the objective of converting a timestamp column to a date column in PySpark using the robust and explicit `.cast(DateType())` method.

Alternative Conversion Methods: Utilizing the to_date Function

While the `.cast()` method is universally applicable and highly explicit regarding schema changes, PySpark also provides dedicated SQL functions for temporal manipulation, which often offer a more fluent syntax for certain operations. The `F.to_date()` function, available via `pyspark.sql.functions`, is specifically designed to handle conversions to the **DateType**.

When applied to a **TimestampType** column, `F.to_date()` achieves the exact same result as casting to DateType: it extracts and returns only the date component. This method can sometimes feel cleaner, as it avoids the need to explicitly import `DateType` from `pyspark.sql.types`.

The equivalent syntax using the `F.to_date()` function would look like this: `df = df.withColumn('new_date_alt', F.to_date(df))`. Both methods are efficient and widely used, and the choice between them often comes down to stylistic preference or integration within a larger SQL expression. However, understanding the `.cast()` method remains fundamental for general schema conversions beyond just date and time.

Best Practices for Handling Time Series Data in PySpark

Effective manipulation of time series data goes beyond simple type conversion. Adhering to certain best practices ensures performance, reliability, and ease of maintenance in big data pipelines. First and foremost, always ensure your source data is in the native **TimestampType** or **DateType** as early as possible in the ETL process. Operating on string representations of time is inefficient and prone to formatting errors, especially across different locales.

Furthermore, when converting from a higher precision type (timestamp) to a lower precision type (date), be mindful of the potential loss of information. If the time component is required later, it is highly recommended to create a new column for the date rather than overwriting the original timestamp column, as demonstrated in our example. This preserves the original data fidelity.

Finally, leverage Spark's optimization capabilities. Using built-in functions like .cast() or dedicated functions from `pyspark.sql.functions` (like `to_date`) ensures that the operations are executed efficiently using Catalyst Optimizer and Tungsten engine, minimizing overhead compared to custom Python UDFs (User-Defined Functions) for similar date transformations.