Convert Timestamp to Date in PySpark (With Example)

Authored by

Mohammed looti

November 11, 2025

Introduction: The Necessity of Temporal Data Simplification in PySpark

Handling temporal data forms the backbone of modern data engineering, especially when processing massive datasets using distributed frameworks like PySpark. In nearly every analytical workflow, raw transaction records or log files contain precise timestamps--detailed values that include date, hour, minute, and second information. While this high precision is valuable for forensic analysis, it often becomes cumbersome when the goal is higher-level aggregation, such as calculating daily sales metrics, tracking monthly trends, or performing calendar-based filtering operations. Simplifying these complex temporal structures into pure date formats is a critical step in standardizing data for downstream processing pipelines.

The distinction between a timestamp and a date is fundamental: a timestamp captures a specific moment in time (e.g., 2023-10-30 22:20:05), whereas a date captures only the calendar day (e.g., 2023-10-30). For business intelligence and reporting, analysts frequently require this simplification to group related records effectively. Attempting to group by the full timestamp would result in highly fragmented data, treating transactions that occurred seconds apart as entirely distinct categories. Therefore, mastering the efficient conversion from timestamp to date within a PySpark DataFrame is essential for achieving clean, actionable insights from big data.

This expert guide details the precise, optimized mechanism for performing this temporal conversion using PySpark's built-in functionalities. We will focus on utilizing the robust casting features provided by the Spark SQL library, specifically the combination of the `withColumn` transformation and the `cast()` function. This approach is highly performant because the operation is executed natively within the distributed Spark environment, ensuring scalability even when handling petabytes of information. We will walk through the required imports, syntax, and a complete, practical example to ensure a seamless transformation.

Understanding the underlying mechanism is crucial: Spark treats data types strictly. The conversion process leverages the explicit instruction to change the column's schema from `TimestampType` to `DateType`. When applied, the Spark engine automatically truncates the time component, maintaining data integrity while delivering the standardized date format (YYYY-MM-DD) necessary for straightforward temporal analysis.

The Foundational Method: Leveraging `cast()` and `DateType`

The most reliable and efficient way to transform a column containing a timestamp to one containing only the date in PySpark is through explicit data type casting. This method requires importing the target schema definition, which is the `DateType` class, sourced from the `pyspark.sql.types` module. The `DateType` is Spark's internal specification for representing dates without time components, making it the perfect target for our conversion. Using explicit casting ensures that the

data transformation adheres strictly to Spark's schema rules, minimizing ambiguity and potential runtime errors.

The core operation is executed using the `cast()` method applied directly to the column object within the context of a DataFrame modification. We typically employ the `withColumn` transformation, which is instrumental for adding a new column derived from existing ones or replacing an existing column entirely. Best practice suggests creating a new date column to preserve the original high-resolution timestamp data, which can be invaluable for debugging or future, more granular analyses.

The syntax below represents the standard, idiomatic approach used by data professionals when performing this specific data manipulation task. It assumes the existence of a DataFrame named `df` and a source timestamp column named `my_timestamp`. Notice the crucial import statement that defines the target schema for the casting operation.

from pyspark.sql.types import DateType

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

In this sequence, a new column, `my_date`, is generated. The input data, sourced from `my_timestamp`, is processed by the `cast(DateType())` function, which systematically handles the removal of the hour, minute, and second components. This operation is not merely a string truncation; it is a type conversion enforced by Spark's Catalyst optimizer, making it highly optimized for distributed processing. The resulting column will hold values conforming to the ISO 8601 date standard (YYYY-MM-DD), ready for immediate use in date-based queries and aggregations.

Preparing the Environment: Setting Up a PySpark Session and Sample Data

To demonstrate the conversion process effectively, we must first establish a controlled PySpark environment and construct a sample DataFrame that accurately simulates real-world transactional data. Our simulated dataset tracks sales activities, linking a precise execution time to a sales amount. This setup requires several foundational steps: initializing the Spark session, defining the raw data, and critically, ensuring the source column is properly recognized as a timestamp by Spark.

Often, raw timestamps are ingested as simple strings (e.g., 'YYYY-MM-DD HH:MM:SS'). Before we can cast this string to a date, it must first be converted into Spark's internal `TimestampType`. We leverage the powerful `pyspark.sql.functions.to_timestamp` function for this initial preparation. This preparatory conversion guarantees that Spark accurately interprets the time components and handles any implicit time zone assumptions before the final date extraction

occurs.

The following comprehensive code block initiates the `SparkSession`, defines the sample data, creates the initial DataFrame, and performs the mandatory string-to-timestamp conversion. This results in a DataFrame where the source column, `ts`, is correctly formatted as a genuine timestamp, making it suitable for the subsequent casting operation.

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

from pyspark.sql import functions as F

#define data
data = ,
,
,
]

#define column names
columns =

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

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

#view 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|
+-------------------+-----+

The resultant DataFrame, `df`, is now properly structured. The `ts` column, having successfully undergone the preparatory conversion, holds the necessary precision, allowing us to proceed to the core transformation. This meticulous setup prevents common errors where Spark might fail to

interpret the source string correctly during a direct cast attempt, underlining the importance of proper data type handling upfront.

Schema Validation: Confirming Data Types Before Transformation

In any robust data engineering pipeline, validating the schema before and after major transformations is not merely a suggestion--it is a mandatory step for quality assurance. Before we apply the conversion logic, we must explicitly confirm that the source column, `ts`, is indeed a `timestamp` data type. If the column were still recognized as a string or an integer, the subsequent `cast(DateType())` operation would either fail or produce incorrect, null results.

In PySpark, the schema can be quickly inspected using the `df.dtypes` attribute, which returns a list of tuples, each containing the column name and its current assigned data type. This simple check provides immediate feedback on the data structure we are interacting with, allowing us to verify the success of the preparatory steps we took earlier (converting the string to a timestamp).

Executing the schema inspection command on our prepared DataFrame yields the following output:

#check data type of each column
df.dtypes

The output confirms our setup: the `ts` column is correctly identified as `timestamp`, meaning it contains both date and time information. The associated `sales` column is a `bigint`, appropriate for numerical values. This successful verification confirms that the DataFrame's structure is precisely what is required for the next phase: applying the `cast(DateType())` function to strip away the time elements, thereby ensuring that the conversion will operate as intended, based on Spark's internal temporal logic.

Executing the Conversion: Applying `withColumn` for Date Extraction

With the schema confirmed, we are now ready to execute the core transformation. As established, we utilize the imported `DateType` and the `withColumn` function. A crucial best practice in large-scale data manipulation is immutability; we avoid overwriting the source `ts` column and instead create a new column, `new_date`, to hold the simplified date values. This preservation strategy maintains the original data fidelity, allowing for flexible analysis later.

The operation `df.cast(DateType())` triggers Spark's highly efficient casting mechanism. When a timestamp is cast to a date, Spark inherently truncates the time part (hours, minutes, seconds) while respecting the underlying time zone settings used during the timestamp's creation or

ingestion. This ensures that the derived date is accurate relative to the recorded moment in time, a critical consideration when dealing with global datasets. This operation is executed in parallel across the cluster, making it extremely fast even for enormous data volumes.

The following code snippet demonstrates the seamless application of this technique and displays the resulting updated DataFrame, now enriched with the new, simplified date column:

```
from pyspark.sql.types import DateType

#create date column from timestamp column
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|
+-------------------+-----+----------+
```

Inspection of the output confirms the success of the transformation. The `new_date` column contains only the date component (YYYY-MM-DD), stripped of the time details present in the `ts` column. This result is perfectly suited for analytical operations that depend solely on calendar days, such as calculating daily totals, performing joins based on the day, or creating simple time-series visualizations. This transformation is a cornerstone of effective temporal data management in big data environments.

Conclusion and Best Practices for Efficient Temporal Processing

We have successfully detailed and executed the conversion of high-resolution timestamp data into a simplified date format using the optimized casting facilities in PySpark. This methodology, centered around `.cast(DateType())`, is scalable, reliable, and crucial for cleaning and preparing temporal features for analytical modeling and reporting. Before concluding, we must perform one final validation to confirm the resulting column's schema.

Running `df.dtypes` one last time ensures that the visual representation in `.show()` accurately reflects the underlying schema:

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

The final output confirms that `new_date` is indeed of the `date` data type. This structural confirmation is essential, as the data type governs how subsequent Spark operations interpret and handle the column, ensuring correct date arithmetic and filtering capabilities.

To maintain robust and high-quality data pipelines, adhere to these key best practices when working with temporal data conversions:

Data Preservation is Paramount: Always utilize `withColumn` to create a new column (e.g., `new_date`) for the converted data. Overwriting the original timestamp column (`ts`) destroys the high-granularity time information, which might be critical for future, more detailed analyses, such as calculating event durations or analyzing hourly trends.

Time Zone Awareness: Time zone handling is complex in distributed systems. While `cast(DateType())` usually handles the truncation correctly, developers must be mindful of the session's default time zone configuration, especially when working with timestamps that fall near midnight. If timestamps originate from various time zones, consider standardizing them to UTC before performing any date extraction to prevent off-by-one day errors.

Schema Verification Iteration: Make schema validation (using `.dtypes` or `.printSchema()`) a routine check after every significant transformation step. This practice immediately identifies discrepancies in data type conversion, saving significant debugging time later in the pipeline.

This streamlined approach guarantees efficient and accurate temporal data preparation, empowering data scientists and engineers to derive meaningful insights from their distributed datasets.

Additional Resources

For readers seeking to deepen their expertise in PySpark temporal manipulation, further exploration of related functions is highly recommended. These resources provide context for handling other common time-related challenges:

Official Apache Spark Documentation on Date and Time Functions, detailing the full suite of temporal transformation tools available in Spark SQL.

Guides on converting Unix Timestamps (epoch time) into standard human-readable formats, which requires the use of functions like `from_unixtime` or `to_timestamp`.

In-depth tutorials focusing on `pyspark.sql.functions` for complex time-series operations, including window functions and date arithmetic.