How to Convert Epoch Values to Datetime in PySpark

Authored by

stats writer

February 5, 2026

The conversion of an Epoch time value into a readable **datetime format** is a frequent requirement in data engineering, especially when processing machine logs or time-series data. In the context of large-scale data processing using PySpark, handling time-based data efficiently is paramount. PySpark offers robust built-in functions that streamline this conversion process, allowing data scientists and analysts to move quickly from raw numeric timestamps to meaningful temporal representations.

The primary methods for achieving this conversion involve utilizing specialized functions within the `pyspark.sql.functions` module, specifically `from_unixtime` and `to_timestamp`. These tools are designed to operate on large distributed datasets managed by Spark, ensuring performance and scalability. Understanding how to correctly apply these functions, along with proper casting of data types, is essential for accurate time-based manipulation and analysis within the PySpark environment.

PySpark: Converting Epoch Time to Datetime Format

Understanding Epoch Time and PySpark Context

Before diving into the conversion syntax, it is crucial to establish a clear understanding of what an **Epoch time** value represents. Also known as Unix time or POSIX time, Epoch time is defined as the total number of seconds that have elapsed since the Unix Epoch--January 1, 1970, 00:00:00 Coordinated Universal Time (UTC). This standardized numerical format simplifies data storage and calculation across different systems but is not human-readable. Converting these large integer or long values into a structured **datetime object**, such as `YYYY-MM-DD HH:MM:SS`, enables effective data visualization and complex temporal filtering.

In PySpark, time data is typically handled within a DataFrame structure. To perform time transformations, we rely on the extensive library of SQL functions available through `pyspark.sql.functions`. When dealing with DataFrame columns, standard Python time operations are ineffective because Spark operates lazily and in a distributed manner. Therefore, we must use Spark SQL functions that can execute these operations across all nodes in the cluster efficiently, maintaining the integrity and schema of the data.

The choice between the primary functions, `to_timestamp` and `from_unixtime`, often depends on the specific scenario and the desired output format, although both achieve the fundamental conversion goal. Both methods require the input column containing the **Epoch value** to be correctly identified and often require explicit casting to ensure Spark interprets the numeric value as seconds since the Epoch, rather than a standard integer.

The Primary Conversion Method: Using `to_timestamp`

The most direct and often recommended approach for converting Epoch seconds to a datetime format in PySpark involves the use of the `to_timestamp` function combined with explicit type casting. Since `to_timestamp` is primarily designed to convert string representations of time into timestamps, when used with numeric Epoch values, we must first cast the numeric column to the desired time type to guide Spark's interpretation.

The standard syntax involves retrieving the DataFrame column containing the Epoch data, casting it to `TimestampType`, and then applying `to_timestamp`. This ensures that the numeric value is treated as an absolute measure of time rather than merely a large number, thus facilitating the correct conversion to the UTC-based standard timestamp format used internally by Spark.

The following syntax demonstrates how to convert an existing column named `epoch` into a new column named `datetime` within a DataFrame:

from pyspark.sql import functions as f
from pyspark.sql import types as t

df.withColumn('datetime', f.to_timestamp(df.epoch.cast(dataType=t.TimestampType())))

This code snippet is efficient and clear. It utilizes the `withColumn` operation to create the new column `datetime`. Inside this function, `f.to_timestamp()` is applied to the `epoch` column after it has been explicitly cast using `df.epoch.cast(dataType=t.TimestampType())`. This particular example successfully converts a raw Epoch time value, such as **1655439422**, into a precise PySpark datetime string like **2022-06-17 00:17:02**, demonstrating the effectiveness of type handling in data transformation.

Setting Up the PySpark Environment for Conversion

To execute any data transformation within PySpark, a properly initialized Spark session is required, along with the necessary imports for SQL functions and data types. This setup ensures that we have access to the distributed computing capabilities and the specific utility functions needed for our conversion task. We typically need to import `SparkSession` to start the environment, `functions` (aliased as `f`) for operations like `to_timestamp`, and `types` (aliased as `t`) for defining schema or performing explicit casting, such as to `TimestampType`.

Defining the input data structure--the DataFrame--is the next crucial step. In practical scenarios, this data might be loaded from files (CSV, Parquet, JSON) or databases. For demonstration purposes, we often define a small dataset in memory. It is vital to confirm that the column containing the **Epoch value** is stored as a numeric type (`LongType` or `IntegerType`), as this is the

format expected for conversion from seconds since the Epoch.

Consider a practical scenario where we track sales data, logging the transaction time only as an Epoch time stamp. Our goal is to convert these numeric stamps into a readable datetime format to enable time-series analysis and reporting. The following example outlines the complete setup, from initializing Spark to viewing the original source data.

Step-by-Step Example: Creating the Source DataFrame

Suppose we manage a log of sales transactions, where each entry records the timestamp as an **Epoch value** and the quantity of items sold. We begin by initializing the Spark Session and defining the raw data structure. This process is mandatory for executing any distributed computation operations against the data structure in Spark.

We define the data rows using a list of lists, where the first element is the Epoch time and the second is the sales count. We also explicitly define the column names for clarity, ensuring that the **epoch** column is correctly labeled for subsequent use in the conversion syntax.

The following PySpark code block illustrates the creation and display of the initial source DataFrame that contains information about sales made on various epoch times at a hypothetical company:

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

#define data
data = ,
,
,
,
,
]

#define column names
columns =

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

#view dataframe
df.show()

```
+----------+-----+
| epoch|sales|
+----------+-----+
|1655439422| 18|
|1655638422| 33|
|1664799422| 12|
|1668439411| 15|
|1669939422| 19|
|1669993948| 24|
+----------+-----+
```

The resulting DataFrame, `df`, clearly shows the raw numeric `epoch` column. These numbers, while precise, lack immediate human interpretability regarding the day, hour, and minute of the transaction. The subsequent step involves applying the conversion logic developed earlier to transform this raw data into a usable time format.

Executing the Conversion Logic

Once the source DataFrame (`df`) is prepared, we apply the `withColumn` transformation. This operation is non-mutating; it returns a new DataFrame (`df_new`) that includes the newly calculated `datetime` column alongside the original data. The core of the operation lies in the seamless combination of casting and the `to_timestamp` function.

By explicitly casting the `epoch` column to `TimestampType`, we instruct Spark to treat the numeric values as timestamp objects representing seconds since the Epoch. The `to_timestamp` function then correctly processes this input and formats it into the standard Spark SQL timestamp string.

The following syntax creates a new DataFrame, `df_new`, containing a column called **datetime** that converts each time in the **epoch** column to a recognizable datetime format:

```
from pyspark.sql import functions as f
from pyspark.sql import types as t

#create new column called 'datetime' that converts epoch to datetime
df_new = df.withColumn('datetime', f.to_timestamp(df.epoch.cast(dataType=t.TimestampType())))

#view new DataFrame
df_new.show()
```

```
+----------+-----+-------------------+
| epoch|sales| datetime|
```

```
+----------+-----+-------------------+
|1655439422| 18|2022-06-17 00:17:02|
|1655638422| 33|2022-06-19 07:33:42|
|1664799422| 12|2022-10-03 08:17:02|
|1668439411| 15|2022-11-14 10:23:31|
|1669939422| 19|2022-12-01 19:03:42|
|1669993948| 24|2022-12-02 10:12:28|
+----------+-----+-------------------+
```

The output clearly illustrates the successful transformation. The new **datetime** column contains properly formatted timestamps, making the data instantly useful for chronological querying, filtering, and aggregation based on specific dates and times.

Analyzing the Results and Data Types

Upon viewing the results in `df_new`, it is immediately apparent that the numeric Epoch time values have been successfully translated into understandable time strings. For example, the Epoch value **1655439422** is precisely mapped to **2022-06-17 00:17:02**. This conversion is crucial for downstream analytical tasks where time granularity is important.

We can verify the transformation by inspecting a few specific rows:

The Epoch time **1655439422** is equivalent to **2022-06-17 00:17:02**.
The Epoch time **1655638422** is equivalent to **2022-06-19 07:33:42**.
The Epoch time **1664799422** is equivalent to **2022-10-03 08:17:02**.

In addition to readability, the data type of the new column is important. When `to_timestamp` is used, the resulting column type is `TimestampType`. This is a fundamental Spark data type optimized for time operations, allowing you to use other functions like `date_format`, `hour`, or `year` directly on the `datetime` column without further conversions. Using native Spark types ensures maximum performance when running operations across the distributed cluster.

Alternative Method: Using `from_unixtime`

While `to_timestamp` combined with casting is highly effective, PySpark also provides the dedicated function `from_unixtime`, which is often considered the most semantic choice for converting Unix/Epoch timestamps. The `from_unixtime` function converts a Unix epoch seconds value (usually a numeric or string representation) to a string representation of the timestamp in a specified format.

The key difference is that `from_unixtime` returns a **string** by default, formatted according to a user-specified pattern (e.g., `'yyyy-MM-dd HH:mm:ss'`), whereas the method using `to_timestamp` returns a native `TimestampType` object that Spark handles internally as a date/time structure. If the end goal is to have the output immediately available for display or integration into a non-Spark system that expects a formatted string, `from_unixtime` might be preferred.

If you were to use `from_unixtime` on our example DataFrame, the syntax would look like this:

df_new_string = df.withColumn('datetime_string', f.from_unixtime(f.col("epoch"), 'yyyy-MM-dd HH:mm:ss'))

This approach is simpler in terms of code structure, avoiding the explicit import and use of `TimestampType` for casting. However, if further complex time arithmetic (like calculating duration between two timestamps) is required, having a native `TimestampType` (as provided by the `to_timestamp` method) is usually more advantageous.

Handling Time Zone Considerations

A critical consideration when converting Epoch time is the handling of time zones. The Epoch standard (Unix time) is inherently based on Coordinated Universal Time (UTC). When PySpark converts an Epoch value into a `TimestampType`, it internally stores this value as UTC.

However, when PySpark displays or formats the output of a `TimestampType` column (e.g., when using `df.show()`), it automatically applies a time zone conversion. This conversion typically defaults to the local time zone configured on the machine or cluster running the Spark driver. This is an important distinction: the underlying data stored is UTC, but the displayed output is localized.

Therefore, when interpreting results such as **2022-06-17 00:17:02**, users should be aware that this time has been adjusted to their local time zone from the original UTC Epoch value. If data needs to be presented consistently in a specific time zone (e.g., EST regardless of the cluster's location), additional PySpark functions, such as `f.to_utc_timestamp` or `f.withTimezone`, should be employed to explicitly manage the time zone offset during the conversion or formatting stage.

Conclusion: Selecting the Right PySpark Tool

Converting Epoch values to a datetime format is a foundational operation in processing temporal data with PySpark. Both `to_timestamp` (with casting) and `from_unixtime` offer reliable, distributed solutions for this task. The choice often comes down to the desired final data type: `to_timestamp` provides a native `TimestampType` essential for downstream analytical operations, while `from_unixtime` offers immediate string formatting.

The most robust method involves using `f.to_timestamp(df.epoch.cast(dataType=t.TimestampType()))`, as it ensures that the resulting column is correctly typed for all subsequent complex temporal aggregations and calculations native to the Spark framework. Mastery of these built-in functions ensures high efficiency and accuracy when dealing with time-series data at scale within any Spark application.

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