How to Convert Epoch Time to Datetime in PySpark

Authored by

stats writer

January 19, 2026

Epoch time, often referred to as Unix time, is a crucial concept in computing, representing a point in time as the number of seconds that have elapsed since the Unix Epoch (January 1, 1970, 00:00:00 UTC). While this format is highly efficient for data storage and manipulation within distributed systems like **Apache Spark**, it is not practical for human interpretation or analytical reporting. Therefore, data engineers and analysts frequently face the task of converting these numeric timestamps into a standard, recognizable Datetime format within PySpark. This detailed guide demonstrates the canonical method for performing this conversion, ensuring data integrity and optimizing the process for large-scale datasets.

Converting a column of integer or long values representing Epoch time into a proper Datetime object in **PySpark** requires leveraging the powerful built-in functions available in the `pyspark.sql.functions` module. The solution must be vectorized to efficiently handle the massive volumes of data processed by **Spark** clusters. The method described below is the standard, reliable approach that maintains high performance across various computational environments.

The Core Syntax for Epoch to Datetime Conversion

The conversion process is streamlined by utilizing the `to_timestamp` function coupled with the `cast` method. This combination first ensures that the numeric Epoch column is properly interpreted as a temporal data type before the conversion function applies the necessary formatting rules. It is essential to import the necessary modules, specifically `functions` and `types`, to access these critical tools within the **Spark DataFrame** context.

You can use the following syntax to convert epoch time to a recognizable Datetime in PySpark, generating a new column labeled `datetime`:

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 specific code snippet performs a crucial two-step transformation. First, the `df.epoch.cast(dataType=t.TimestampType())` command explicitly tells Spark to treat the raw integer values in the `epoch` column as actual timestamps. Second, the `f.to_timestamp()` function then formats these internal timestamp representations into the standard, human-readable Datetime string format. This technique is highly effective and ensures that the data maintains the necessary precision for subsequent time-series analysis.

Understanding the Conversion Mechanism

The `withColumn` operation is fundamental in **PySpark** for adding or replacing columns in a DataFrame. Here, we are creating a new column named `datetime`. The core of the operation relies on the interplay between casting and the to_timestamp function. When dealing with numeric epoch values (which are typically `LongType` or `IntegerType`), Spark needs guidance on how to interpret these numbers temporally.

The `cast(dataType=t.TimestampType())` function is the mechanism that converts the raw numeric data into Spark's internal TimestampType format, which inherently understands the number of seconds since the epoch reference point. Once the column has been cast correctly, `to_timestamp` is applied to provide the final, formatted output. For instance, this precise syntax will accurately convert an epoch time value of **1655439422** into the corresponding **PySpark Datetime** format: **2022-06-17 00:17:02**.

While some legacy methods or non-Spark environments might use functions like `from_unixtime`, the combination of `cast` and `to_timestamp` is generally preferred for its clarity and efficiency when manipulating datatypes directly within a **PySpark DataFrame** structure. This high level of explicit data type handling is crucial when managing large, complex schemas.

Setting Up the PySpark Environment for Testing

To demonstrate this conversion in a concrete scenario, we must first establish a minimal PySpark environment and create a sample DataFrame. We will simulate a scenario where a company records sales transactions, storing the time of sale exclusively as an Epoch timestamp for simplicity and storage efficiency.

The setup involves initializing a `SparkSession`, which is the entry point for all **Spark** functionality, and then defining both the raw data and the column names (schema). This preparation ensures that the data is correctly ingested and structured before any transformations are applied. The raw data provided below uses six arbitrary epoch timestamps paired with corresponding sales figures.

Example: How to Convert Epoch to Datetime in PySpark

Suppose we have the following **PySpark DataFrame** that contains information about sales made at various epoch times. Note how the initial `epoch` column consists solely of large integer values, which are meaningless without conversion:

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 initial DataFrame, `df`, successfully holds the raw Epoch timestamps. Our next step is to apply the conversion logic discussed earlier. We will use the `withColumn` method to introduce `df_new`, which will contain the human-readable timestamps alongside the original data. This practice is recommended as it preserves the original raw data while providing the transformed output for analytical work.

We utilize the imported functions `f` (for functions) and `t` (for types) to construct the conversion expression. This approach is cleaner and more concise when writing complex transformation pipelines in **PySpark**. The result demonstrates the smooth transition from raw numeric timekeeping to standardized calendar time.

```
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|
+----------+-----+-------------------+
```

Analyzing the Converted Datetime Results

Upon viewing the resulting DataFrame, `df_new`, the efficiency of the conversion method is immediately apparent. The new **datetime** column now contains structured, recognizable dates and times, which greatly facilitates reporting, filtering based on time periods, and calculating time differences. This conversion is an essential step in preparing data for any serious time-series analysis within a distributed computing framework.

We can verify several converted rows to confirm the accuracy of the process. Each numeric Epoch value is correctly mapped to its corresponding Gregorian calendar date and time representation:

The epoch time **1655439422** is successfully converted to **2022-06-17 00:17:02**.

The epoch time **1655638422** is successfully converted to **2022-06-19 07:33:42**.

The epoch time **1664799422** is successfully converted to **2022-10-03 08:17:02**.

The subsequent rows follow the same accurate transformation pattern, validating the effectiveness of using `cast(TimestampType)` combined with `to_timestamp`.

This new Datetime column enables complex operations such as aggregation by day, week, or month, and allows for seamless integration with visualization tools that require standard date formats. The ability to perform these conversions at scale is one of the primary advantages of working with **PySpark**.

Understanding Timezone Considerations in PySpark

A critical factor to remember when performing timestamp conversions in **PySpark** is how the system handles timezones. Internally, **Apache Spark** stores all TimestampType values in **UTC** (Coordinated Universal Time). However, when Spark displays the data--such as when running `df_new.show()`--it automatically converts these internal UTC values to the local timezone configured on your machine or the **Spark Session** configuration.

This automatic conversion is often convenient but requires careful consideration, especially when analysts are collaborating across different geographical regions or when data sources originate from various timezones. If your original Epoch time implicitly represents a time in a specific local timezone (not UTC), you must account for this offset before or after conversion to ensure temporal accuracy.

If explicit timezone handling is required, **PySpark** offers additional functions such as `from_utc_timestamp` and `to_utc_timestamp`, which allow developers to explicitly specify the source and target timezones, overriding the default behavior. For standard Epoch conversion, where the input is typically assumed to be UTC (as per the Unix standard), the method shown above is sufficient, provided the user understands that the display output will be localized.

Conclusion

Converting Epoch timestamps to a recognizable Datetime format is a standard requirement in data processing, and **PySpark** provides an elegant and highly performant solution. By utilizing the `withColumn` method in combination with the `cast(TimestampType)` and `to_timestamp` functions, large datasets can be quickly transformed, preparing them for detailed analytical operations.

Mastering this conversion technique is fundamental for any data scientist or engineer working with large-scale time-series data on the **Spark** platform. This method ensures both high performance and adherence to standard SQL data type specifications, making the resulting data reliable and easy to integrate into downstream reporting tools.