How to Add Time to a Datetime Column in PySpark

Authored by

stats writer

February 4, 2026

Manipulating datetime object fields is a fundamental requirement in large-scale data analysis, especially when working with time series data or event logs. In the PySpark environment, adding specific time intervals to a timestamp column is efficiently achieved using robust built-in functions.

The primary method involves utilizing the withColumn transformation. This function is essential for creating a new column in a PySpark DataFrame by applying a specified expression to existing data. While functions like `date_add` or `date_sub` handle whole days, calculating precise time increments (hours, minutes, seconds) requires leveraging SQL expressions within PySpark, ensuring accurate and versatile manipulation of timestamp fields for complex processing tasks.

PySpark: Add Time to Datetime

The Core Mechanism: Using `withColumn` and `F.expr`

To successfully add granular time components (such as hours, minutes, or seconds) to a datetime object within a PySpark DataFrame, the most effective technique combines the `withColumn` transformation with the specialized F.expr function. The F.expr function allows us to inject standard SQL interval syntax directly into the PySpark logic, which is critical for handling time components smaller than a day.

The standard syntax involves selecting the target timestamp column and then applying an addition operation using `+ F.expr('INTERVAL ...')`. The interval string itself must precisely define the duration you intend to add. This approach provides maximum flexibility, allowing developers to specify arbitrary durations combining years, months, days, hours, minutes, and seconds as needed for complex transformations.

Syntax Deep Dive: Adding Specific Time Intervals

The following example illustrates the basic structure required to add a composite time interval--specifically 3 hours, 5 minutes, and 2 seconds--to an existing timestamp column named `ts`. Note how the SQL syntax for the interval is enclosed within the F.expr call:

from pyspark.sql import functions as F

df = df.withColumn('new_ts', df.ts + F.expr('INTERVAL 3 HOURS 5 MINUTES 2 SECONDS'))

This particular transformation creates a new column, conventionally named `new_ts`, where each value is calculated by applying the specified increment (3 hours, 5 minutes, and 2 seconds) to the corresponding original datetime object found in the `ts` column. This method is highly performant and the recommended pattern for timestamp arithmetic in PySpark.

Important Consideration: Subtraction: Should your requirement be to subtract time, the process remains identical. You simply replace the addition sign (`+`) with the subtraction sign (`-`) immediately preceding the F.expr call.

Practical Demonstration: Setting Up the PySpark DataFrame

To showcase this functionality, we will use a sample PySpark DataFrame containing sales records, each associated with a specific timestamp. First, we need to initialize a SparkSession and create the sample data, ensuring the timestamp column is correctly cast to the appropriate type before arithmetic operations can be performed.

The setup code below defines a small dataset and converts the initial string representation of time into the proper timestamp format using `F.to_timestamp`. This preparation step is crucial, as time arithmetic cannot be performed on plain string columns.

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|
+-------------------+-----+

Implementing Complex Time Addition (Hours, Minutes, Seconds)

With our PySpark DataFrame initialized and the `ts` column correctly formatted as a timestamp, we can now apply the core logic to shift the time. We aim to create a new column, `new_time`, which represents the original sales timestamp plus exactly 3 hours, 5 minutes, and 2 seconds. This demonstrates the precise control offered by the SQL interval expression.

We use the withColumn function combined with `F.expr` to perform the calculation across every row simultaneously, which is characteristic of the optimized execution in the PySpark architecture:

from pyspark.sql import functions as F

#add 3 hours, 5 minutes and 2 seconds to each datetime in 'ts' column
df = df.withColumn('new_time', df.ts + F.expr('INTERVAL 3 HOURS 5 MINUTES 2 SECONDS'))

#view updated DataFrame
df.show()

+-------------------+-----+-------------------+
| ts|sales| new_time|
+-------------------+-----+-------------------+
|2023-01-15 04:14:22| 225|2023-01-15 07:19:24|
|2023-02-24 10:55:01| 260|2023-02-24 14:00:03|
|2023-07-14 18:34:59| 413|2023-07-14 21:40:01|
|2023-10-30 22:20:05| 368|2023-10-31 01:25:07|
+-------------------+-----+-------------------+

The resulting `new_time` column clearly shows the time shift applied. For instance, the first entry shifts from `04:14:22` to `07:19:24`. Crucially, notice the final entry (`2023-10-30 22:20:05`) correctly rolls over into the next day (`2023-10-31 01:25:07`). This demonstrates that the interval addition correctly handles date boundaries and time zone considerations inherent in Spark's internal timestamp representation.

Simplifying Interval Operations (Adding Only Hours)

While the interval syntax is powerful for combined units, it is equally straightforward to use it for adding a single unit, such as only hours. This simplified approach requires only specifying the necessary unit within the `F.expr` statement. This is often necessary when applying simple time zone offsets or calculating lead/lag times in hours.

The following modification shows how to add only 3 hours to the existing timestamp column `ts`, creating a new column `new_time`:

from pyspark.sql import functions as F

#add 3 hours to each datetime in 'ts' column
df = df.withColumn('new_time', df.ts + F.expr('INTERVAL 3 HOURS'))

#view updated DataFrame
df.show()

```
+-------------------+-----+-------------------+
| ts|sales| new_time|
+-------------------+-----+-------------------+
|2023-01-15 04:14:22| 225|2023-01-15 07:14:22|
|2023-02-24 10:55:01| 260|2023-02-24 13:55:01|
|2023-07-14 18:34:59| 413|2023-07-14 21:34:59|
|2023-10-30 22:20:05| 368|2023-10-31 01:20:05|
+-------------------+-----+-------------------+
```

Now the `new_time` column shows each time from the `ts` column with only 3 hours added to it. This confirms the flexibility of the `F.expr` approach: you can include any combination of time units (YEARS, MONTHS, DAYS, HOURS, MINUTES, SECONDS) or use a single unit based on your specific transformation needs. Feel free to use this syntax to add or subtract as much time as you'd like from a datetime column.

Summary and Best Practices for Time Arithmetic

The technique of using withColumn combined with `F.expr` and SQL `INTERVAL` syntax is the definitive method for adding or subtracting precise time durations in PySpark. This method is preferred over converting timestamps to Unix epoch time for arithmetic because it preserves date boundaries and avoids potential overflow errors associated with large integer manipulations.

When implementing time arithmetic in production environments, always ensure the following best

practices are followed:

Type Validation: Always verify that the column being manipulated is of the `TimestampType`. Attempting to add an interval to a string or date type without time components will result in errors or inaccurate outputs.

Clarity in Intervals: Use clear, descriptive interval strings (e.g., `'INTERVAL 1 DAY 12 HOURS'`) rather than trying to calculate total seconds, enhancing code readability and maintainability.

Handling Time Zones: Be aware of the time zone settings of your Spark cluster and your data source. Spark generally handles timestamps internally in UTC, and interval arithmetic is applied based on this internal representation.

Further Exploration in PySpark Datetime Operations

The following tutorials explain how to perform other common tasks in PySpark data processing, building upon the foundational knowledge of manipulating datetime objects:

How to calculate the difference between two timestamps in seconds.
Methods for extracting year, month, or day components from a timestamp.
Using window functions to analyze time-series data efficiently.