Learn How to Calculate Time Differences in PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learn How to Calculate Time Differences in PySpark DataFrames*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=16589

Calculating the time difference between two Timestamp columns is a fundamental operation when performing time-series analysis or tracking event durations within a DataFrame. In the PySpark environment, this process requires careful handling of data types to ensure accurate, granular results. The standard approach involves converting the timestamp fields into a numerical format, specifically the Epoch time representation, which simplifies subtraction and subsequent conversion into familiar units like seconds, minutes, or hours.

The following syntax demonstrates the efficient method for calculating and deriving the difference between two time fields within a PySpark DataFrame. This technique relies on the built-in functions available in the `pyspark.sql.functions` module, ensuring high performance across distributed datasets characteristic of Spark processing.

```
from pyspark.sql.functions import col

df_new = df.withColumn('seconds_diff', col('end_time').cast('long') - col('start_time').cast('long'))
.withColumn('minutes_diff', (col('end_time').cast('long') - col('start_time').cast('long'))/60)
.withColumn('hours_diff', (col('end_time').cast('long') - col('start_time').cast('long'))/3600)
```

This specific implementation is designed to calculate the duration between the timestamps stored in the **start_time** and **end_time** columns. By performing the calculation in **seconds** first, we establish a precise base unit which can then be easily scaled up to **minutes** and **hours** through simple division (by 60 and 3600, respectively). Understanding the underlying conversion mechanism is key to debugging and optimizing time-based calculations in PySpark.

Understanding the Epoch Time Conversion

When dealing with time differences in Spark, direct subtraction of two Timestamp objects does not always yield a simple numerical result suitable for algebraic manipulation across all Spark versions or configurations. To guarantee a reliable, numerical difference, we utilize the concept of Epoch time. Epoch time, or Unix time, is defined as the number of seconds that have elapsed since January 1, 1970 (UTC).

The critical step in the provided code snippet is the use of the `.cast('long')` function. When applied to a PySpark Timestamp column, this function converts the timestamp value into its corresponding Epoch time representation, which is an integer representing the total number of seconds since the Epoch. Once both the start and end times are represented as long integers (seconds), simple subtraction yields the exact duration in seconds. This mathematical operation is highly efficient and perfectly suited for distributed computation across a large DataFrame.

This method ensures consistency regardless of time zones or daylight savings adjustments, as the underlying representation (seconds since Epoch) remains standardized. The resulting difference in

seconds serves as the foundation for deriving all other required time units, such as minutes and hours, by dividing the total seconds by the appropriate conversion factors.

Preparing the PySpark DataFrame Example

To illustrate this technique, let us construct a sample DataFrame containing various start and end timestamps. When loading data into PySpark, timestamp fields are often initially ingested as strings. It is crucial that these string representations are explicitly converted into the native Timestamp data type before any temporal arithmetic can be accurately performed. This conversion ensures that Spark recognizes the column contents as time points rather than simple text strings.

We begin by initializing the Spark Session and defining the raw data, followed by applying the conversion logic using the `F.to_timestamp` function. This step is vital for ensuring that the subsequent `.cast('long')` operation correctly interprets the data. The format specified (`'yyyy-MM-dd HH:mm:ss'`) must exactly match the format of the strings in the source data.

The following example demonstrates the necessary setup steps, including defining the data, creating the base DataFrame, and performing the essential string-to-timestamp conversion before proceeding with the difference calculation.

```
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 columns to timestamp
df = df.withColumn('start_time', F.to_timestamp('start_time', 'yyyy-MM-dd HH:mm:ss'))
.withColumn('end_time', F.to_timestamp('end_time', 'yyyy-MM-dd HH:mm:ss'))

#view DataFrame
```

```
df.show()

+-------------------+-------------------+
| start_time| end_time|
+-------------------+-------------------+
|2023-01-15 04:14:22|2023-01-18 04:15:00|
|2023-02-24 10:55:01|2023-02-24 11:14:30|
|2023-07-14 18:34:59|2023-07-14 18:35:22|
|2023-10-30 22:20:05|2023-11-02 07:55:00|
+-------------------+-------------------+
```

The output confirms that our input data has been successfully transformed into a PySpark DataFrame where both `start_time` and `end_time` columns are of the Timestamp type. This prepared structure is now ready for the calculation of the time differences across all records in a vectorized and distributed manner.

Calculating Duration in Multiple Units

With the DataFrame properly prepared, we can now apply the core logic utilizing the withColumn function. The withColumn function is essential in PySpark for adding new columns or transforming existing ones without modifying the original DataFrame in place, adhering to Spark's immutable data structure principles. We chain multiple calls to `withColumn` to create three distinct difference columns simultaneously.

For maximum precision, the calculation always starts by finding the difference in seconds. This is achieved by subtracting the casted `start_time` (long integer) from the casted `end_time` (long integer).

seconds_diff: This column is the raw difference calculated after casting both timestamps to `long` (Epoch seconds). This provides the most granular measurement of the duration.

minutes_diff: This is calculated by taking the raw seconds difference and dividing it by 60. Note that the result will be a floating-point number, representing the duration accurately, including partial minutes.

hours_diff: Calculated by dividing the raw seconds difference by 3600 (60 seconds * 60 minutes). This also yields a precise floating-point value suitable for further analysis or aggregation.

The following snippet executes this calculation, creating `df_new` which contains the original timestamp columns alongside the three newly derived duration columns:

```
from pyspark.sql.functions import col

#create new DataFrame with time differences
df_new = df.withColumn('seconds_diff', col('end_time').cast('long') - col('start_time').cast('long'))
.withColumn('minutes_diff', (col('end_time').cast('long') - col('start_time').cast('long'))/60)
.withColumn('hours_diff', (col('end_time').cast('long') - col('start_time').cast('long'))/3600)

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

```
+-------------------+-------------------+------------+-------------------+--------------------+
| start_time| end_time|seconds_diff| minutes_diff| hours_diff|
+-------------------+-------------------+------------+-------------------+--------------------+
|2023-01-15 04:14:22|2023-01-18 04:15:00| 259238| 4320.633333333333| 72.01055555555556|
|2023-02-24 10:55:01|2023-02-24 11:14:30| 1169| 19.483333333333334| 0.32472222222222225|
|2023-07-14 18:34:59|2023-07-14 18:35:22| 23|0.38333333333333336|0.006388888888888889|
|2023-10-30 22:20:05|2023-11-02 07:55:00| 207295| 3454.9166666666665| 57.581944444444446|
+-------------------+-------------------+------------+-------------------+--------------------+
```

Interpreting the Resulting Duration Columns

The resulting DataFrame, `df_new`, successfully provides the duration between the start and end events for each row, expressed in three distinct, yet mathematically linked, units. Analyzing the output confirms the effectiveness of the Epoch time conversion approach in PySpark. For instance, examining the first row, we see a duration spanning multiple days, resulting in a large number of seconds (259,238).

The newly created columns are defined as follows:

seconds_diff: Represents the precise difference between the start and end Timestamp in total seconds.

minutes_diff: Represents the difference expressed in minutes, calculated as `seconds_diff / 60`.

hours_diff: Represents the difference expressed in hours, calculated as `seconds_diff / 3600`.

It is important to note that the use of floating-point division ensures that the fractional parts of minutes and hours are preserved, providing an exact measure of elapsed time. Should integer results be required (e.g., for reporting only whole hours), explicit casting to an `integer` type would

need to be incorporated into the withColumn chain using `.cast('integer')` before the final division, although this is generally less recommended for analytical accuracy.

Alternatives and Advanced Time Manipulation

While the `.cast('long')` method is robust and universal for obtaining time differences in seconds, PySpark offers other powerful functions for handling temporal calculations, especially when dealing with specific date or interval components. Functions such as `datediff` (for differences in whole days) or `timestamp_seconds` (to convert seconds back into a timestamp/interval) can be employed for different use cases.

Another relevant function is `F.unix_timestamp()`, which explicitly returns the Unix time (Epoch seconds) from a timestamp column, serving a similar purpose to `.cast('long')` but often preferred for its semantic clarity regarding time conversion. For example, the core calculation could be rewritten using `F.unix_timestamp(col('end_time')) - F.unix_timestamp(col('start_time'))`. Choosing between `.cast('long')` and `F.unix_timestamp()` often comes down to stylistic preference, as both achieve the necessary conversion to numerical seconds for arithmetic operations.

When working with PySpark, utilizing vectorized operations like those demonstrated with withColumn and built-in functions ensures that the calculations are executed efficiently across the distributed cluster, maximizing performance when dealing with massive datasets.

The complete documentation for the PySpark withColumn function provides further details on customizing data transformations.