How to Calculate Time Differences in PySpark

Authored by

stats writer

February 7, 2026

To effectively calculate the difference between two temporal measurements in a large-scale data environment, especially within a PySpark context, analysts rely on specialized built-in functions provided by the PySpark SQL library. These functions are designed to handle distributed time series data efficiently, making time manipulation a core capability for tasks ranging from latency measurement in logs to session duration analysis. Key functions like `datediff` and the indispensable `unix_timestamp` enable the accurate manipulation and comparison of time values stored within a Spark DataFrame. By leveraging these powerful tools, the temporal difference between two points can be precisely calculated and integrated into sophisticated data processing pipelines. This approach ensures highly efficient and accurate time-based calculations, solidifying PySpark's position as a valuable framework for time-sensitive data analysis.

Understanding the underlying mechanism is crucial for successful implementation. PySpark, like many data processing systems, converts date-time objects into a standardized numerical format--specifically, the number of seconds elapsed since the Unix epoch (January 1, 1970). This conversion allows for simple arithmetic subtraction, which is significantly faster and more reliable than complex date object manipulation in a distributed setting. The result of this subtraction is the time difference expressed in total seconds, which can then be trivially converted into minutes, hours, or days using standard division operations. This methodology ensures that time difference calculations remain consistent and scalable, even when dealing with petabytes of data distributed across numerous clusters.

This document provides a comprehensive, step-by-step guide detailing the syntax and practical application necessary to calculate time differences, focusing specifically on obtaining results in seconds, minutes, and hours. We will explore how to set up the appropriate columns, use the necessary cast functions, and finally, generate a new DataFrame that clearly presents the elapsed time metrics derived from your temporal columns. Adherence to these steps guarantees clean, performant, and reliable time difference calculations within any PySpark environment.

PySpark: Calculate Difference Between Two Times

Core Mechanism for Calculating Time Differences in PySpark

To perform accurate time difference calculations within a PySpark DataFrame, the fundamental process involves converting the existing timestamp columns into a numerical type that represents time in seconds. This conversion is achieved using the `cast('long')` function. Once both the start and end timestamp columns are cast to the `long` type, simple subtraction yields the difference in total seconds. This numerical result forms the basis for all subsequent temporal metrics, such as minutes and hours.

The syntax below illustrates the direct application of this technique using the withColumn function,

which is essential for adding new computed columns to your DataFrame without altering the original structure. We utilize the `col` function to reference the existing columns and perform the necessary casting operation within the expression. This method ensures that the operation is executed efficiently across the distributed cluster resources managed by Spark.

The following code snippet demonstrates the required transformation. It assumes that `start_time` and `end_time` are already properly formatted timestamp columns. If they are initially strings, they must first be converted to the appropriate `TimestampType`, a crucial step we will cover in the example section. Note how the core calculation--the subtraction of the casted columns--is reused and scaled (divided by 60 and 3600) to obtain the required units.

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 particular configuration calculates the duration between the timestamps stored in the **`start_time`** and **`end_time`** columns. The results are explicitly categorized into three new columns: `seconds_diff`, `minutes_diff`, and `hours_diff`. The efficiency of this method stems from its reliance on simple numerical subtraction after converting the complex temporal objects into simple numerical representations of seconds since the epoch.

Understanding the PySpark Functions for Temporal Arithmetic

To successfully execute time difference calculations, a precise understanding of the involved PySpark functions is required. The primary function responsible for the transformation is `.cast('long')`. When applied to a column of Spark's `TimestampType`, this method converts the timestamp into a 64-bit integer, where the value represents the count of seconds passed since the Unix epoch. This conversion is essential because direct subtraction between two native `TimestampType` columns is not automatically defined to return a numerical duration in seconds; instead, it might return an interval type, which is less flexible for standard mathematical operations required for unit conversion.

The withColumn transformation is fundamental for generating derived metrics. It allows users to define a new column based on a calculated expression applied row-wise across the DataFrame. In our case, we chain multiple `withColumn` calls to sequentially add the seconds, minutes, and hours difference columns. Chaining these operations ensures that the resulting DataFrame, `df_new`, maintains all original columns while incorporating the three newly calculated duration metrics efficiently.

Furthermore, the use of `col('column_name')` imported from `pyspark.sql.functions` is mandatory to refer to columns symbolically within transformation expressions. This abstraction ensures that the calculation is performed on the underlying column data type. The repeated expression `(col('end_time').cast('long') - col('start_time').cast('long'))` forms the backbone of the calculation, providing the difference in seconds. The subsequent division by `60` (for minutes) and `3600` (for hours) utilizes standard arithmetic operations on the numerical result, producing floating-point results necessary for precise time duration representation.

Practical Example: Setting Up the DataFrame

To illustrate this method concretely, let us establish a sample DataFrame containing start and end times for various activities. This example simulates real-world telemetry or log data where the duration of an event is critical for analysis. Before calculating the difference, it is vital to ensure that the time data is correctly interpreted by Spark as `TimestampType`. If the source data is ingested as strings (which is common, as shown below), explicit conversion must occur.

We begin by initializing a `SparkSession` and defining our sample data. This data includes four rows, each containing a start time and an end time represented as strings in the format 'YYYY-MM-DD HH:mm:ss'.

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 critical step within this setup is using `F.to_timestamp`. This function parses the string representation of the date and time, converting it into Spark's internal `TimestampType`, which is necessary before any numerical temporal arithmetic can be performed. Failing to correctly cast string columns results in errors when attempting the `.cast('long')` operation later, as the latter requires a valid timestamp type as its input.

Step-by-Step Implementation of Time Difference Calculation

Once the DataFrame `df` contains correctly formatted timestamp columns, we can proceed with the calculation of the time difference. This involves applying the transformation logic defined earlier, using the withColumn method sequentially to derive the required duration metrics in seconds, minutes, and hours. This process leverages the distributed nature of PySpark, ensuring that the heavy lifting of calculation is spread across the cluster.

We begin by importing the necessary `col` function. We then define the new DataFrame, `df_new`, by taking `df` and applying three chained withColumn transformations. The first transformation calculates the base difference in seconds by casting both the `end_time` and `start_time` columns to `long` integers and subtracting them. Since the difference between two Unix timestamps (represented as long integers) is inherently the difference in seconds, this column is straightforward.

The subsequent transformations reuse this core calculation: the difference in seconds. To obtain the difference in minutes, we divide the seconds difference by 60. To obtain the difference in hours, we divide the seconds difference by 3600 (60 seconds * 60 minutes). It is crucial that the division operation is performed after the subtraction to maintain accuracy across the distributed dataset.

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

Analyzing the Resulting Output and Data Precision

The output DataFrame, `df_new`, successfully incorporates the three newly computed duration columns alongside the original timestamp columns. Reviewing this output provides critical insight into the duration of each activity recorded in the dataset. The resulting DataFrame structure is clean and ready for further analytical processing or aggregation.

The resulting DataFrame contains the following three new duration columns derived from the calculation:

`seconds_diff`: This column provides the exact numerical difference between the start and end time, measured purely in seconds. This is a `long` integer value, representing the raw output of the casted subtraction operation.

`minutes_diff`: This column shows the difference in minutes. Since the calculation involves division by 60, this column is typically of `DoubleType`, resulting in floating-point precision which is necessary to capture fractions of a minute accurately.

`hours_diff`: Similarly, this column displays the difference in hours. Division by 3600 also results in a `DoubleType`, ensuring that even very short durations are accurately represented as fractional hours.

It is important to acknowledge the precision inherent in these calculations. Since we are dealing

with time, the `seconds_diff` provides the most granular, integer-based metric. The `minutes_diff` and `hours_diff` columns are essential for reporting and high-level analysis, but analysts must be aware that these are floating-point numbers. If integer-only results (e.g., floor or ceiling results) are required for display purposes, additional PySpark functions such as `round()`, `floor()`, or `ceil()` must be applied to the calculated duration columns.

Summary of PySpark Time Manipulation Tools

The methodology outlined above--using `.cast('long')` for direct numerical subtraction--is highly effective for calculating differences measured in seconds, minutes, or hours. However, PySpark offers a much broader suite of functions for temporal manipulation within its SQL library, which are useful depending on the specific analytical requirement.

For instance, if the requirement is simply to calculate the number of calendar days between two dates, ignoring time components, the `datediff(end_date, start_date)` function is more appropriate and simpler to use. Similarly, functions like `date_sub` and `add_months` are invaluable for feature engineering tasks where you need to shift dates forward or backward by specific calendar intervals. While the `cast('long')` method provides maximum precision for duration calculations down to the second, being aware of these alternatives ensures that the most appropriate tool is selected for every unique time-based analytical challenge within PySpark.

Mastering these temporal functions, particularly the reliable conversion via `.cast('long')`, empowers data professionals to conduct complex time series analysis and duration measurements efficiently on massive, distributed datasets managed by the Spark framework.

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