How to Find the Time Difference Between Two Columns in PySpark

Authored by

stats writer

January 1, 2026

Analyzing temporal data is a fundamental requirement in data engineering and analytics. When working with large datasets distributed across a cluster, PySpark provides robust tools for handling date and time calculations efficiently. Specifically, determining the duration, or the difference between two time points, often requires converting time-based data types into a numerical format that allows for standard arithmetic operations. This approach ensures high performance when processing millions of records within a DataFrame, which is crucial for calculating metrics like session duration, latency, or interval lengths across various business processes.

The primary method for calculating the difference between two times in PySpark involves utilizing the underlying numerical representation of Timestamp columns. By casting these timestamps to a long integer type, we effectively convert the time value into the total number of seconds elapsed since the Unix epoch (January 1, 1970). Once converted to these numerical seconds, simple subtraction yields the time difference in seconds, which can then be scaled up to minutes, hours, or days as needed. While PySpark offers dedicated functions like F.datediff() for differences in days, the direct subtraction method provides granular control down to the second level, making it versatile for precision analysis.

This detailed guide explores the precise syntax and methodology required to accurately compute these temporal differences. We will focus on leveraging the powerful withColumn transformation combined with casting operations to generate new columns representing duration in seconds, minutes, and hours, thereby maximizing the utility of your time-series data within the Spark ecosystem. Understanding this method is key to advanced temporal data manipulation in PySpark, allowing data scientists and engineers to derive meaningful insights from sequential events.

The Importance of Casting Timestamps to Long

In the Apache Spark environment, the most reliable and efficient way to perform time difference calculations is by converting the native Spark Timestamp type into a numerical format. This is achieved using the cast() function, which transforms the timestamp into a long integer. The resulting long integer represents the total number of seconds since the Unix epoch (1970-01-01 00:00:00 UTC). This conversion is essential because, unlike dedicated Python datetime objects that support direct subtraction yielding a timedelta object, PySpark DataFrame columns are designed for distributed, SQL-style operations, where numerical subtraction is the standard approach for measuring scalar differences.

When you subtract one timestamp (as a long integer) from another, the result is the duration expressed directly in seconds. This numerical outcome is easily scalable and integrable into subsequent arithmetic calculations without the overhead of complex temporal functions for simple subtraction. For instance, if you are calculating the time elapsed between a transaction initiation time and a transaction completion time, casting both columns to 'long' provides the necessary

numerical foundation for a quick and accurate calculation. It also avoids potential issues related to time zones or complex calendar arithmetic that dedicated time functions might introduce when only a simple duration is needed.

Furthermore, relying on primitive numerical types like long improves performance. PySpark is optimized for column-based arithmetic operations, and performing basic subtraction on large numerical columns is significantly faster than using more complex built-in duration functions, particularly when dealing with massive datasets typical of big data processing. This method aligns perfectly with Spark's fundamental goal of providing high-speed distributed computation across clusters, ensuring that time calculations do not become a bottleneck in data processing pipelines.

The Standard Syntax for Duration Calculation

To implement this casting and subtraction method, we utilize the withColumn transformation, which allows us to add new columns to an existing DataFrame based on the results of an expression. The expression involves selecting the desired timestamp columns using the col function, casting them to 'long', and then performing the subtraction. To derive differences in units other than seconds, such as minutes or hours, we simply divide the resulting second difference by the appropriate conversion factor (60 for minutes, 3600 for hours).

The following syntax demonstrates how to calculate the difference between the times stored in the `end_time` and `start_time` columns, generating three new columns representing the duration in seconds, minutes, and hours, respectively. This structure is highly repeatable and forms the core logic for temporal difference analysis in PySpark:

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 pattern is widely used in real-world PySpark applications because it explicitly defines the calculation logic and utilizes highly optimized Spark functions. Notice the chained withColumn calls; this functional approach builds the new DataFrame incrementally without mutating the original one, adhering to best practices in distributed data manipulation. This clean structure allows for easy debugging and maintenance, even when many complex transformations are applied sequentially.

Example: Setting Up the PySpark Environment and Data

To demonstrate this technique in a practical scenario, consider a dataset tracking various system

activities, where we need to measure the duration of each event. Suppose we have a DataFrame containing string columns representing the exact start and end times of these activities. Before performing any calculation, the first critical step is to initialize the Spark Session and convert these string representations into the proper PySpark Timestamp type, as numerical casting only works correctly on time-based column types.

The setup involves defining the sample data, creating the base DataFrame, and then using the `F.to_timestamp()` function to ensure that Spark interprets the date strings correctly according to a specified format (`'yyyy-MM-dd HH:mm:ss'`). This data preparation phase is non-negotiable for accurate temporal analysis. Incorrect data types will either lead to errors during the `.cast('long')` operation or produce incorrect duration values.

The following Python code initializes the Spark session, defines the input data containing start and end times, and transforms the string columns into the required Timestamp format, making the DataFrame ready for the duration calculations in the next step:

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

Upon execution, the `df.show()` command confirms that the DataFrame columns `start_time` and `end_time` are now correctly registered as PySpark Timestamp types, marked by the consistent formatting and precise handling of dates and times. This successful conversion ensures that the subsequent calculation step will utilize the correct underlying numerical structure for accurate subtraction.

Generating Difference Columns Using Chained Transformations

Once the base DataFrame is correctly typed, we proceed to apply the time difference calculation logic. As demonstrated previously, we use the chained withColumn method to sequentially generate the desired duration columns. This chaining is efficient and readable, allowing a single operation block to define multiple calculated fields. The critical operation within each withColumn call is the subtraction of the casted columns: `col('end_time').cast('long') - col('start_time').cast('long')`.

For the `seconds_diff` column, the result of this subtraction is taken directly, as the casted difference is inherently in seconds. For `minutes_diff`, the total seconds are divided by 60, resulting in a floating-point number that accurately represents the duration in minutes. Similarly, for `hours_diff`, the division by 3600 converts the total seconds into hours. It is important to remember that these calculations result in Double type values, which provide the highest precision necessary for granular timing analysis, even if the result is not a whole number.

The code block below applies these calculations to the pre-processed DataFrame `df`, creating `df_new` with the three added duration columns. Observing the output confirms that the duration measurements are calculated correctly across differing time scales, ranging from a few seconds to multiple days, validating the efficacy of the long-casting approach:

```
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|0.3833333333333336|0.006388888888888889|
|2023-10-30 22:20:05|2023-11-02 07:55:00| 207295| 3454.9166666666665| 57.581944444444446|
+-------------------+-------------------+-----------+------------------+------------------+
```

Analyzing the Resulting Duration Columns

The resulting DataFrame `df_new` now holds the original time columns alongside the newly computed duration metrics. Each row successfully measures the elapsed time for that specific activity. This structure is highly beneficial for subsequent analytical steps, such as filtering sessions based on minimum duration, calculating averages, or joining this data with other metrics based on event length.

The three derived columns offer different levels of granularity for analysis:

seconds_diff: This column represents the precise, base-level duration measurement. It is the direct result of the long subtraction and is crucial for high-resolution timing requirements, such as measuring server latency or small differences in log events.
minutes_diff: By dividing the base seconds by 60, this column provides a more human-readable metric for medium-length activities, like average user session length or process run times that last less than an hour.
hours_diff: Dividing by 3600 scales the duration up to hours, making it ideal for tracking long-running tasks, shift durations, or multi-day intervals, providing a concise summary of long temporal spans.

It is important to emphasize the utility of the withColumn function here. By leveraging this function repeatedly, we ensure that the entire operation is executed efficiently across the distributed cluster. Spark handles the distribution and calculation of these new columns in parallel, allowing for rapid processing even when the underlying DataFrame contains billions of records. This technique is standard practice when enriching datasets with derived metrics in a scalable manner.

Alternative Approach: Utilizing PySpark Built-in Functions

While the long-casting subtraction method offers maximum control and precision down to the second, PySpark also provides built-in functions for common temporal difference calculations, simplifying the syntax for specific needs. The most notable is `F.datediff()`, which calculates the difference between two date or timestamp columns exclusively in terms of whole days.

The `F.datediff(end_time, start_time)` function is particularly useful when the required granularity is low (i.e., days) and you are not concerned with the hours, minutes, or seconds within those days. This function handles the underlying date arithmetic, including month lengths and leap years, making the code cleaner for date-only comparisons. However, it is essential to recognize its limitation: it returns an integer representing the count of full days, meaning it ignores sub-day durations.

For scenarios requiring the result to be returned as a specific PySpark interval type, one might also consider functions like `F.timestamp_seconds()` or `F.unix_timestamp()` combined with subsequent interval subtraction, although the long-casting method remains the most direct way to get the numerical second difference. Choosing the right method--direct numerical subtraction for precision or built-in functions for simplicity--depends entirely on the analytical requirement and the required level of temporal detail.

Advanced Considerations: Time Zones and Granularity

When dealing with timestamps, time zone awareness is a critical factor, especially in distributed systems like Spark. By default, Spark timestamps are usually stored relative to UTC internally, but how they are displayed and how arithmetic operations interpret them can be affected by the session's time zone configuration. When using the `.cast('long')` method, the conversion fundamentally relies on the time zone definition of the source data column. If your start and end times belong to different time zones, ensure that they are correctly normalized to a single, consistent time zone (usually UTC) before performing the cast and subtraction to avoid skewed results.

Furthermore, while the long-casting method provides second-level precision, modern data logging often involves milliseconds or microseconds. If higher precision is required, you must adjust the casting and scaling factors accordingly. For instance, to obtain the difference in milliseconds, you would typically need to cast to a double or similar type representing milliseconds since the epoch, and then ensure that the resulting difference is scaled appropriately (e.g., dividing by 1000 to return seconds from milliseconds). Always verify the native storage unit of your timestamp columns to ensure the correct conversion factor is applied during the calculation.

Mastering these temporal manipulations allows for powerful analytical pipelines. For example,

calculating the time difference enables crucial data quality checks (e.g., verifying that `end_time` is always later than `start_time`) and is foundational for building features used in machine learning models, such as dwell time or frequency metrics. The reliability and speed of the PySpark subtraction approach make it a cornerstone of big data time-series analysis.

Further Exploration in PySpark Temporal Functions

Beyond basic subtraction, PySpark offers a rich set of functions for temporal data manipulation. These functions are often leveraged in conjunction with the methods discussed here to perform more sophisticated analyses, such as calculating the time taken between specific business milestones, or aggregating durations across various categories.

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

How to handle date truncation in PySpark for aggregation.
Methods for adding or subtracting specific intervals (months, days) using functions like F.add_months().
Techniques for extracting specific temporal components, such as the year, month, or hour, using F.extract().