Learn How to Calculate Date Differences in PySpark: A Step-by-Step Guide

Authored by

Mohammed looti

November 11, 2025

Calculating the difference between two dates is a fundamental operation in PySpark, essential for tasks ranging from calculating customer retention periods to measuring employee tenure in data engineering pipelines. Because PySpark is designed for large-scale data processing, it offers highly optimized functions within the `pyspark.sql.functions` module that allow developers to perform complex date arithmetic efficiently across massive datasets.

In this guide, we explore the precise methods available for determining the time elapsed between a start date and an end date, providing solutions for calculating the difference in days, months, and years. Understanding these core functions is crucial for any developer working with time-series or temporal data within the Apache Spark ecosystem.

Essential PySpark Functions for Date Manipulation

Before diving into specific calculations, it is necessary to understand the core functions provided by PySpark SQL that facilitate date differences. All date calculations must handle inputs in a recognizable date format, which is why the `F.to_date()` function is frequently employed. This function ensures that string representations of dates are correctly cast into the internal DateType format required by Spark for accurate arithmetic.

The two primary functions used for calculating date differences are `F.datediff()` and `F.months_between()`. These functions are highly specific to their purpose and provide the foundational logic for deriving time spans in various units. It is important to note that these functions, like most in the `functions` module, are designed to operate on columns within a PySpark DataFrame, enabling vectorized operations rather than row-by-row processing.

When working with these functions, we utilize `F` as an alias for `pyspark.sql.functions`, which is standard practice for brevity and readability in PySpark scripting. Furthermore, the result of any date calculation is often added back to the original DataFrame as a new column using the `withColumn` transformation, preserving the source data while enriching it with the derived temporal metrics.

Method 1: Calculating Date Differences in Days

The simplest and most direct method for finding the time elapsed between two dates is calculating the difference in days. For this purpose, PySpark provides the highly optimized `datediff` function. This function accepts two DateType columns as arguments: the end date (or later date) and the start date (or earlier date), always returning the difference as an integer representing the total number of days.

It is critical that the arguments passed to `datediff` are valid date types. If the input columns (such as `end_date` and `start_date`) are initially stored as string types (as is common when loading data

from CSV or JSON files), they must first be converted using `F.to_date()` within the expression. If the start date is later than the end date, the result will be a negative integer, indicating the direction of the temporal difference.

Below is the standard syntax for calculating the difference between dates in days, adding the result to a new column named `diff_days`:

from pyspark.sql import functions as F

df.withColumn('diff_days', F.datediff(F.to_date('end_date'), F.to_date('start_date'))).show()

Method 2 & 3: Calculating Differences in Months and Years

While `datediff` is useful for day counts, calculating differences in months or years requires a slightly different approach to account for variable month lengths and leap years. PySpark addresses this need with the `months_between` function. This function calculates the number of months between the two specified dates, returning the result as a precise floating-point number. This precision is vital because it accurately reflects partial months, which is necessary for rigorous analytical reporting.

To calculate the difference in months, we apply `months_between` and then use the `round` function to format the output to a specified number of decimal places, typically two, for cleaner presentation. This ensures that the result is readable while retaining necessary accuracy.

To derive the difference in years, we simply leverage the result of the monthly calculation. Since a year is equivalent to 12 months, dividing the floating-point result of `months_between` by 12 yields the precise duration in years. As with the monthly calculation, the `round` function is applied subsequently to maintain a consistent display format.

Method 2: Calculate Difference Between Dates in Months

This code snippet utilizes `F.months_between` and rounds the resulting floating-point value to two decimal places, providing a precise measure of the duration in months.

from pyspark.sql import functions as F

df.withColumn('diff_months', F.round(F.months_between(F.to_date('end_date'), F.to_date('start_date')),2)).show()

Method 3: Calculate Difference Between Dates in Years

By dividing the monthly difference by 12, we convert the result into years. We again apply `F.round` to ensure the output is concise and easy to interpret, typically rounding to two decimal places.

from pyspark.sql import functions as F

df.withColumn('diff_years', F.round(F.months_between(F.to_date('end_date'), F.to_date('start_date'))/12,2)).show()

Practical Implementation Example

To illustrate these methods in a practical context, let us assume we are working with a PySpark DataFrame containing employee tenure information. This DataFrame includes an `employee` identifier, the `start_date` of their employment, and the `end_date` (which could be the current date or their termination date). Our goal is to calculate the total duration of employment in days, months, and years for each record.

We begin by initializing a Spark session and creating a representative DataFrame. Note that the date columns are initially defined as strings, necessitating the use of `F.to_date()` in the subsequent calculation steps. This setup mimics a common scenario where raw data is ingested into the Spark environment.

The structure of our data is simple, designed specifically to demonstrate the application of date functions across multiple records simultaneously. Observing the output of the `df.show()` command confirms the data structure before we apply the transformations.

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()

+--------+----------+----------+
|employee|start_date| end_date|
+--------+----------+----------+
| A|2020-10-25|2023-01-15|
| B|2013-10-11|2029-01-18|
| C|2015-10-17|2022-04-15|
| D|2022-12-21|2023-04-23|
| E|2021-04-14|2023-07-25|
| F|2021-06-26|2021-07-12|
+--------+----------+----------+
```

To execute all three calculations--days, months, and years--in a single, fluent operation, we chain the `withColumn` calls. This chaining mechanism is a powerful feature of withColumn, allowing us to generate three new derived columns efficiently without creating intermediate DataFrames, adhering to best practices for optimizing Spark transformations.

```
from pyspark.sql import functions as F

#create new DataFrame with date differences columns
df.withColumn('diff_days', F.datediff(F.to_date('end_date'), F.to_date('start_date')))
.withColumn('diff_months', F.round(F.months_between(F.to_date('end_date'), F.to_date('start_date')),2))
.withColumn('diff_years', F.round(F.months_between(F.to_date('end_date'), F.to_date('start_date'))/12,2)).show()

+--------+----------+----------+---------+-----------+----------+
|employee|start_date| end_date|diff_days|diff_months|diff_years|
+--------+----------+----------+---------+-----------+----------+
| A|2020-10-25|2023-01-15| 812| 26.68| 2.22|
| B|2013-10-11|2029-01-18| 5578| 183.23| 15.27|
| C|2015-10-17|2022-04-15| 2372| 77.94| 6.49|
| D|2022-12-21|2023-04-23| 123| 4.06| 0.34|
| E|2021-04-14|2023-07-25| 832| 27.35| 2.28|
| F|2021-06-26|2021-07-12| 16| 0.55| 0.05|
+--------+----------+----------+---------+-----------+----------+
```

Interpreting the Results and Advanced Considerations

The output clearly shows the addition of three new columns--`diff_days`, `diff_months`, and `diff_years`--to the original DataFrame. These columns instantaneously provide the calculated duration for every employee record. This vectorized approach is what makes PySpark so powerful for handling large volumes of temporal data.

For instance, examining the first record (Employee A, start date 2020-10-25, end date 2023-01-15), the results are precisely calculated:

The are **812 days** between 2020-10-25 and 2023-01-15.

The are **26.68 months** between 2020-10-25 and 2023-01-15.

The are **2.22 years** between 2020-10-25 and 2023-01-15.

A crucial consideration when dealing with date arithmetic in Spark is understanding how rounding affects precision. We purposefully used the `F.round` function for the month and year differences to limit the output to two decimal places. While this improves readability for reporting, users should adjust the precision parameter based on their analytical requirements; higher precision may be necessary for financial or scientific applications.

Furthermore, while this guide focused on DateType, PySpark also supports TimestampType for calculations involving time components. If time differences are required (e.g., hours, minutes), different functions like `unix_timestamp` or direct subtraction of timestamps (which yields results in seconds) must be employed, followed by appropriate scaling and conversion to the desired unit.

Additional Resources

For continuing your journey in PySpark data manipulation, the following topics are highly relevant and explain how to perform other common tasks:

Converting column types in PySpark.

Working with Timestamp data and time zones.

Implementing Window functions for time-series analysis.