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

Authored by

**stats writer**

February 7, 2026

Calculating the temporal difference between two dates is a fundamental requirement in data analysis, particularly when working with large-scale datasets managed by PySpark. This process leverages specialized, built-in functions provided by the Apache Spark SQL module, ensuring high efficiency and accuracy across distributed environments. Mastering these functions is essential for tasks ranging from calculating customer retention periods to determining employee tenure or lifecycle durations.

The initial and most critical step in performing date arithmetic is ensuring that the date fields within your DataFrame are correctly interpreted as proper date objects, rather than plain strings. PySpark offers the highly useful to_date function for this transformation. Once the dates are correctly formatted, you can employ functions like datediff for day counts or months_between for monthly intervals. These functions simplify complex chronological calculations into straightforward, optimized PySpark expressions.

The following comprehensive guide outlines the specific methods and corresponding syntax necessary to calculate date differences across various granularities--days, months, and years--providing a clear path to implementing these solutions within your data processing pipelines. We will ensure that all examples utilize robust methods applicable to production-level data engineering tasks within the PySpark ecosystem.



# PySpark: Calculate Date Differences

To perform accurate temporal calculations in PySpark, you must first load the necessary functions from the SQL module. Below, we detail three distinct methods for calculating date differences, depending on the required unit of measurement (days, months, or years).

## Prerequisites: Ensuring Proper Date Formatting

Before any calculation can occur, the date columns must be explicitly converted from their input format (usually string type) into PySpark's internal DateType. The to_date function handles this transformation, which is critical because mathematical operations are only valid on typed date objects. If the input string format matches the default 'yyyy-MM-dd' pattern, the function requires only the column name; otherwise, a format pattern must be supplied as a second argument.

It is standard practice in PySpark to import the functions module with an alias, typically F, to keep the code concise and readable. This alias allows easy access to all the utility functions needed for data manipulation, including date conversions and calculations. Failure to correctly cast the column types often leads to null results or incorrect calculations, highlighting the importance of the initial conversion step using to_date.

## Method 1: Calculating Date Difference in Days using datediff()

The most straightforward way to find the difference between two dates is by calculating the total number of days elapsed. datediff is the specific function designed for this purpose. It accepts two date expressions: the end date followed by the start date. The resulting output is an integer representing the count of full days that have passed between the two chronological points.

When applying this method, we use withColumn to introduce a new column into the existing DataFrame. This approach ensures that the original data remains unmodified while providing a new column containing the calculated difference. The resulting column will be highly valuable for analyses requiring precise daily granularity, such as calculating service uptime or project duration.

**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: Calculating Date Difference in Months using months_between()

If your analysis requires the difference measured in months, months_between is the appropriate function. Unlike datediff, this function returns a decimal value (DoubleType), allowing for partial months to be accurately represented. This is particularly useful in financial or reporting contexts where precise fraction of months must be considered.

The syntax mirrors the day calculation, but we frequently incorporate the round function to manage the precision of the output. Rounding is essential for presentation and ensuring results align with reporting standards, typically rounding the result to two decimal places for monetary or tenure calculations.

**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: Deriving Date Difference in Years

While PySpark does not have a dedicated built-in function named `years_between`, calculating the difference in years is trivial once the difference in months is known. Since there are 12 months in a year, we simply take the output of the months_between function and divide it by 12. This mathematical derivation provides an accurate, fractional representation of the time difference in years.

This approach leverages the existing powerful functions and minimizes reliance on custom User Defined Functions (UDFs), maximizing performance and compatibility within the distributed Spark environment. As with the monthly calculation, the use of round is highly recommended to maintain readability and consistency in the resulting data column, ensuring that complex fractional years are presented cleanly.

**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()

Each of these examples relies on accessing the end date from the `end_date` column and the starting date from the `start_date` column within the DataFrame. The relative ordering of the dates within the function call (end date first, then start date) is essential for obtaining a positive result representing the elapsed time.

## Practical Implementation: Setting Up the Sample DataFrame

To demonstrate these methodologies in a real-world scenario, we will first create a sample DataFrame containing employee records, including their start and end dates of employment. This setup requires initializing a PySpark session and defining the schema for our data. This preparatory step is fundamental to any Spark operation, establishing the environment necessary for distributed processing.

The following code snippet initializes a Spark session, defines the input data (a list of employee IDs, start dates, and end dates), specifies the column names, and finally constructs the DataFrame. Viewing the DataFrame immediately after creation confirms that the data structure is ready for the date difference calculations outlined in the subsequent sections.

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

## Combining Calculations for Comprehensive Date Analysis

A significant advantage of using the functional approach in PySpark is the ability to chain operations seamlessly. We can calculate the differences in days, months, and years within a single operation by chaining multiple withColumn calls. This method is highly efficient as it minimizes the number of passes over the dataset, adhering to performance best practices in distributed computing.

The combined syntax below demonstrates the simultaneous application of datediff and months_between, along with the derived years calculation, to produce a robust output table. Each chained withColumn statement builds upon the result of the previous operation, culminating in a single, enriched DataFrame.

**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 Results and Precision Management

The output clearly shows three new columns--`diff_days`, `diff_months`, and `diff_years`--which quantify the time elapsed between the respective start and end dates for each employee record. For instance, analyzing employee 'A' reveals a tenure of 812 days, which translates to 26.68 months or 2.22 years. This detailed breakdown offers immediate insights into the temporal characteristics of the dataset.

It is important to note the role of the `F.round()` function in managing output precision. For monthly and yearly differences, which are fractional by nature due to the non-uniform length of months, rounding is crucial. In our example, we chose to round to two decimal places, a common practice for maintaining data integrity while ensuring the results are easily interpretable for human readers. Analysts should adjust the rounding precision based on the specific requirements of their downstream reporting or modeling tools.

Furthermore, the power of withColumn cannot be overstated. By utilizing this function repeatedly, we avoid materializing intermediate DataFrames, optimizing memory usage and execution speed. This function ensures that the original DataFrame is immutable, and a new DataFrame is returned containing the added column(s), providing a clean and functional programming approach to data transformations in PySpark.

## Summary of Date Functions Used

The successful calculation of date differences hinges upon the correct application of three core functions from the `pyspark.sql.functions` library:

to_date: Essential for converting string representations of dates into a DateType object that PySpark can perform calculations on.

datediff: Provides the precise count of days between two dates, returning an integer value.

months_between: Returns the decimal difference in months, forming the basis for both monthly and yearly tenure calculations.

By combining these functions, we achieve a flexible and efficient mechanism for temporal analysis, suitable for large datasets processed by PySpark. The methods demonstrated here are foundational for advanced time-series analysis and feature engineering based on time elapsed.

## Further Learning and Resources

Mastering date manipulation is just one aspect of working with big data in PySpark. For those looking to expand their knowledge beyond date differences, the following tutorials explain how to perform other common tasks crucial for data preparation and analysis in the Spark ecosystem:

Tutorials focusing on filtering DataFrames based on date ranges.
Guides detailing complex timestamp manipulation and time zone conversions.
Explorations into window functions for time-series aggregation.