How to Filter a PySpark DataFrame by Date Range with a Practical Example

Authored by

stats writer

January 20, 2026

Analyzing time-series data or subsets of data based on specific historical windows is a fundamental requirement in modern data processing. When working with large-scale datasets managed by PySpark, efficient date filtering is paramount for performance and accuracy. Fortunately, PySpark provides highly optimized tools for this purpose, eliminating the need for complex custom logic.

The most straightforward and recommended method for isolating records within a defined temporal boundary in a DataFrame utilizes the powerful between function. This function, which is a key component of the pyspark.sql.functions module, is designed specifically to check if a column's value falls inclusively between two provided boundary values. This approach guarantees that analysis can be easily constrained to specific periods, such as filtering for all transactions that occurred during the third quarter of a fiscal year or, as demonstrated later, filtering all employee starts within a specific three-year period.

Using the between function simplifies complex filtering tasks that might otherwise require verbose combinations of greater-than-or-equal-to and less-than-or-equal-to logical operators. By accepting two arguments--a start date and an end date--the function elegantly handles the inclusion criteria, ensuring that any record where the designated date column value is chronologically equal to or after the start date, and equal to or before the end date, is retained for subsequent analysis or manipulation. This efficiency is critical when dealing with the distributed nature of DataFrame operations.

Understanding the PySpark between Function

The core mechanism for date range filtering in PySpark revolves around the column method named `between`. This is not a standalone function imported directly from `pyspark.sql.functions` but rather a method called directly on a specific column object within the DataFrame. It is important to distinguish this usage, as it contrasts slightly with other common filtering functions like `col()` or `when()`. The method inherently performs a logical comparison test, returning a boolean result for every row, which is then utilized by the parent `filter()` transformation.

When implementing the between function for dates, the underlying data type of the column being filtered must be compatible. Although PySpark is flexible enough to handle date strings (e.g., 'YYYY-MM-DD' format) when compared against other strings in the filter criteria, best practice dictates converting date columns to the native Spark `DateType` or `TimestampType` beforehand. Using native types ensures accurate chronological comparison, especially when dealing with time components or differing date format representations across the dataset.

The inclusivity of the `between` operation is a crucial detail. When filtering between Date A and Date B, PySpark includes both Date A (the lower bound) and Date B (the upper bound) in the resulting subset. This characteristic is typically desired in range queries, guaranteeing that the specified

start and end points of the analysis window are fully incorporated. If strict exclusivity is required (i.e., strictly greater than the start and strictly less than the end), one would need to revert to using explicit logical operators such as `>` and `<` rather than `between`.

Essential Syntax for Date Range Filtering

To execute a date range filter, the general approach involves selecting the target DataFrame, applying the `.filter()` transformation, and passing the column selection combined with the `.between()` method as the condition. This structure is concise yet powerful, leveraging the functional programming paradigm inherent to Spark operations. The critical parameters passed to `.between()` are the start date and the end date, which can be provided as string literals (if the column is also a string type) or as proper Date objects.

It is often beneficial to define the boundary dates externally, typically as a tuple or a list, especially if these dates will be reused across multiple filtering operations or functions. This promotes code cleanliness and simplifies maintenance. The following syntax snippet illustrates the clean and effective way to define the date parameters and apply them directly within the filter transformation using the Python tuple unpacking operator (`*`).

You can use the following syntax to filter rows in a PySpark DataFrame based on a date range:

#specify start and end dates
dates = ('2019-01-01', '2022-01-01')

#filter DataFrame to only show rows between start and end dates
df.filter(df.start_date.between(*dates)).show()

This particular example filters the DataFrame to only contain rows where the date in the **start_date** column of the DataFrame is between **2019-01-01** and **2022-01-01**. This demonstrates the fundamental logic: selecting the column and applying the inclusive temporal boundary check.

To further illustrate the practical implementation of this technique, the following example shows how to use this syntax in practice, beginning with the creation of the necessary DataFrame.

Setting up the Example PySpark DataFrame

To fully grasp the practical application of date range filtering, let us establish a concrete scenario. We will create a small, representative PySpark DataFrame modeling employee hiring records. This dataset includes two columns: an `employee` identifier and their corresponding `start_date`. This structure is common in operational data systems where temporal auditing is mandatory.

The first step in any PySpark operation is initializing the necessary environment, specifically acquiring or creating a SparkSession. The SparkSession acts as the entry point to programming Spark with the Dataset and DataFrame API. Once the session is active, we define the raw data and column structure before instantiating the distributed DataFrame.

Suppose we have the following PySpark DataFrame that contains information about the start date for various employees at a company. Observe the varying dates spanning several years, which provides a good test case for range filtering:

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|
+--------+----------+
A	2017-10-25
B	2018-10-11
C	2018-10-17
D	2019-12-21
E	2021-04-14
F	2022-06-26
+--------+----------+

Practical Application: Filtering the DataFrame

With the sample DataFrame now instantiated and visible, we proceed to apply the filtering logic. Our objective is to isolate only those employees who were hired between the start of 2019 and the end of 2021. Specifically, we define the range as **2019-01-01** to **2022-01-01**. Since the end date is set to the first day of 2022, it effectively captures all dates up to and including December 31, 2021, given the date-only nature of the column.

We combine the `.filter()` transformation with the column-specific `.between()` method. This operation is lazy, meaning the actual computation only occurs when an action, such as `.show()`, is called. This reliance on lazy evaluation is a fundamental concept in Spark, ensuring efficient distributed computation by optimizing the execution plan before triggering any data movement.

We use the following syntax to filter the DataFrame to only contain rows where the date in the **start_date** column falls within our desired range:

#specify start and end dates
dates = ('2019-01-01', '2022-01-01')

#filter DataFrame to only show rows between start and end dates
df.filter(df.start_date.between(*dates)).show()

```
+--------+----------+
|employee|start_date|
+--------+----------+
| D|2019-12-21|
| E|2021-04-14|
+--------+----------+
```

Analyzing the Filtered Results

Upon reviewing the output generated by the `.show()` action, it is clear that the filter has successfully isolated only the relevant records based on the inclusion criteria defined by the between function. Employee D (hired 2019-12-21) and Employee E (hired 2021-04-14) are retained, as both dates fall inclusively within the **2019-01-01** and **2022-01-01** range.

Conversely, employees A, B, and C, hired in 2017 and 2018, were correctly excluded because their `start_date` precedes the lower bound of **2019-01-01**. Furthermore, employee F, hired on 2022-06-26, was also excluded because this date is subsequent to the upper boundary of **2022-01-01**. This demonstrates the precision of the range filtering mechanism in handling both ends of the time series distribution.

Notice that the DataFrame has been filtered to only show the rows with the two dates in the **start_date** column that fall between **2019-01-01** and **2022-01-01**. The resulting filtered subset is itself a new PySpark DataFrame, which allows users to immediately chain further transformations or actions onto this result, such as aggregating statistics, joining with other datasets, or writing the filtered data to storage. This sequential, transformation-based processing is what makes the Spark architecture highly suitable for complex ETL pipelines.

Calculating Row Counts Within a Specific Range

In many analytical use cases, the immediate requirement is not the filtered data itself, but rather a simple summary statistic, such as the total count of records that satisfy the date criteria. Instead of executing `.show()` on the filtered DataFrame, we can chain the `.count()` action directly after the `.filter()` transformation. This optimization prevents the display or materialization of large intermediate results and immediately returns the numerical count.

The `.count()` action is one of the terminal operations in Spark, triggering the execution of the entire preceding computation graph across the distributed cluster. When used in conjunction with the date filter, it efficiently calculates exactly how many records across all partitions satisfy the date inclusion criteria defined by the `between` operation, providing instant metrics on data volume within the specified period.

Note that if you only want to know how many rows have a date within a specific date range, then you can use the **count** function as follows:

#specify start and end dates
dates = ('2019-01-01', '2022-01-01')

#count number of rows in DataFrame that fall between start and end dates
df.filter(df.start_date.between(*dates)).count()

2

This tells us that there are two rows in the DataFrame where the date in the **start_date** column falls between **2019-01-01** and **2022-01-01**. This counting capability is often used for validation checks, monitoring data quality, or feeding metrics into dashboarding systems.

Advanced Considerations for Date Handling

While string-based date filtering works well for simple 'YYYY-MM-DD' formats, production environments often demand more robust date handling. It is highly recommended to explicitly cast string date columns to the appropriate native Spark types, such as `DateType` or `TimestampType`,

using functions available in pyspark.sql.functions, especially if the data includes timezone information or requires date arithmetic.

If the column being filtered is of `TimestampType`, the `between` operation will include time components, comparing not just the day but the exact moment specified. For instance, filtering between '2023-01-01 10:00:00' and '2023-01-05 14:00:00' will exclude any records exactly at 14:00:01 on January 5th. Users must be extremely mindful of whether their boundary definitions should include time components (e.g., setting the upper bound time to 23:59:59.999...) or if they intend to ignore time by first casting the column to `DateType`.

Furthermore, when dealing with distributed systems like Spark, timezone management can introduce subtle errors. If your data sources span multiple geographies, ensure that all timestamps are standardized to a common timezone (e.g., UTC) before applying range filters. The SparkSession configuration parameters and functions like `from_utc_timestamp` are crucial tools for maintaining data integrity during these casting and standardization processes, ensuring that filters based on wall-clock time are accurate globally.

Conclusion and Further Learning

Filtering PySpark DataFrames by date range using the between function is an efficient and clean method for temporal data subsetting. By utilizing built-in column methods, developers can write highly readable and optimized code suitable for large-scale data processing tasks. Mastering this technique is fundamental for any data engineer or data scientist working extensively with Spark.

Note: You can find the complete documentation for the PySpark **between** function here.

Related PySpark Tutorials

The following tutorials explain how to perform other common tasks in PySpark, further expanding the skills necessary for complex data manipulation and analysis:

Understanding PySpark Joins and Performance Optimization
Working with Window Functions for Advanced Analytics
Casting Data Types for Schema Enforcement and Consistency