Learn How to Filter DataFrames by Date Range in PySpark with a Practical Example

Authored by

Mohammed looti

November 11, 2025

Mastering Date Range Filtering in PySpark

Handling temporal data is a fundamental task in data engineering and analysis. When working with large-scale datasets managed by PySpark, efficiently filtering records based on a specific date range is critical for generating meaningful insights. This guide details the most robust and idiomatic way to achieve this using the built-in functions provided by the Apache Spark API.

The core functionality relies on the powerful `between()` method available on DataFrame columns. This method allows developers to specify a starting and ending date, inclusively, ensuring that only records falling within that period are retained. This approach is highly recommended due to its clarity and optimized performance within the Spark ecosystem.

The general syntax for implementing date range filtering within a DataFrame utilizes the `filter()` transformation coupled with the between() function, as shown below. It is important to ensure that the column being filtered is correctly cast as a Date or Timestamp type, although PySpark often handles standard ISO date strings automatically during ingestion.

#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 succinct example specifically filters the DataFrame (aliased as `df`) to include only those rows where the date contained in the `start_date` column falls inclusively between the specified boundaries: **2019-01-01** and **2022-01-01**. The use of the splat operator (`*`) unpacks the tuple `dates` into two positional arguments required by the between() function.

Setting Up the PySpark Environment and Sample Data

Before executing the filtering logic, we must first establish a working PySpark session and create a sample DataFrame. For demonstration purposes, we will model a simple scenario containing employee identification and their respective start dates within a company. This setup ensures that we have realistic temporal data to apply our filtering logic against.

The following code snippet initializes the `SparkSession`, defines the raw data as a list of tuples, specifies the schema column names, and finally constructs the DataFrame. Viewing the initial DataFrame is crucial to understanding the dataset we are about to query. We utilize the `show()` action to display the contents of the newly created distributed dataset.

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
+--------+----------+

The resulting DataFrame, `df`, now contains six records, each associated with a unique employee and a corresponding date in the `start_date` column. These dates span several years, providing a perfect testbed for our date range filtration operation. Note that in real-world scenarios, data may require explicit casting using functions like `to_date()` to ensure proper temporal comparisons, especially if the data was originally loaded as a string type with inconsistent formatting.

Executing the Filter Using the between() Function

Our objective is to isolate only those employees who started during a specific three-year window, defined as starting on **2019-01-01** and ending on **2022-01-01**. This filtering operation is achieved by applying the `filter()` transformation to the DataFrame and passing the condition generated

by the between() function.

The simplicity of the between() function allows for highly readable and maintainable code. It inherently includes both the lower and upper bounds in the resulting dataset, which is the standard expectation when performing range queries. We define the date range using a Python tuple for convenience, which is then dynamically passed into the filter expression.

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

Upon execution, the resulting DataFrame clearly demonstrates that the filter successfully extracted the two records (Employee D and Employee E) whose start dates fall within the specified range. Employee D's start date (2019-12-21) is after the start boundary, and Employee E's start date (2021-04-14) is before the end boundary (2022-01-01). The records dated 2017, 2018, and 2022 are correctly excluded, confirming the precision of the filtering operation.

Alternative Approach: Using Standard Comparison Operators

While the `between()` function is the most concise solution for date range filtering, an alternative method involves using standard comparison operators (greater than or equal to, and less than or equal to) combined with the logical AND operator (`&`). This method offers greater flexibility if one needs to implement exclusive bounds or more complex conditional logic involving multiple columns.

To replicate the exact functionality of the inclusive `between()` filter, we must use `>=` for the start date and `<=` for the end date. The entire conditional logic must be enclosed within parentheses due to operator precedence rules in Python and PySpark expressions.

 # Define the boundaries explicitly
start_date = '2019-01-01'
end_date = '2022-01-01'

```
# Filter using standard comparison operators (AND)
df.filter(
(df.start_date >= start_date) &
(df.start_date <= end_date)
).show()
```

This approach yields the identical result as the previous example. However, the `between()` function remains the cleaner choice for simple inclusive range checks, as it reduces the potential for syntax errors and improves code readability. Understanding both methods, however, is beneficial for advanced data manipulation tasks where boundary conditions might need finer control.

Optimizing for Count: Avoiding Full Data Retrieval

In many production environments, especially when dealing with massive datasets processed by PySpark, the analyst often only requires the total count of records that satisfy a specific criterion, rather than retrieving and displaying the entire filtered DataFrame. Executing an action like `show()` on a large filtered DataFrame can be resource-intensive.

To perform this operation efficiently, we chain the `filter()` transformation directly with the `count()` action. The count() function executes the necessary calculations in parallel across the cluster and returns a single integer representing the total number of matched rows. This avoids the overhead associated with materializing and displaying the potentially large intermediate result set.

```
#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
```

As expected, the output confirms that there are exactly two rows within the original dataset where the date in the `start_date` column falls inclusively between the specified boundaries of **2019-01-01** and **2022-01-01**. Utilizing the count() function is an essential practice for performance optimization in big data analytics.

Conclusion and Additional Resources

Filtering DataFrames by date range is a common requirement easily addressed using the versatile

`filter()` transformation combined with the specific column method `between()`. This technique provides a clear, concise, and highly efficient way to subset data based on temporal criteria in PySpark.

For developers seeking more advanced date and time manipulations, PySpark offers a rich library of functions within `pyspark.sql.functions`, enabling complex operations such as calculating date differences, extracting components (year, month, day), and handling time zone conversions. Mastering these functions is key to becoming proficient in temporal data processing within the Spark ecosystem.

For complete details and further advanced usage of the primary function discussed, consult the official documentation:

PySpark Column.between() Official Documentation

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