How to Filter a PySpark DataFrame by Date Range

Authored by

stats writer

February 6, 2026

Filtering data efficiently is a fundamental task in large-scale data processing. When working with PySpark DataFrames, defining filters based on temporal boundaries--specifically, date ranges--is often required for analysis or reporting. PySpark provides a clean and highly optimized syntax to achieve this using the built-in functions.

The primary mechanism involves leveraging the DataFrame's filter function in conjunction with the column expression method, specifically the between condition. This combination allows you to swiftly isolate records where a specified date column falls inclusively within two given endpoints. For instance, if you have a DataFrame named `df` with a date column, the syntax below demonstrates how to retrieve all records spanning the first quarter of 2021:

```
df.filter(df.date.between('2021-01-01', '2021-03-31'))
```

Executing this command yields a new DataFrame containing only those rows where the value in the `date` column is chronologically bounded by **January 1, 2021** and **March 31, 2021**, inclusive of both boundary dates. This technique is essential for focused data extraction and ensuring analytical accuracy in large datasets.

Filter by Date Range in PySpark (With Example)

Introduction to Date Range Filtering in PySpark

Filtering records based on temporal criteria, such as a defined date range, is a cornerstone of data manipulation in Apache Spark. When utilizing PySpark DataFrames, the combination of the built-in `filter` method and the `between` expression provides an exceptionally concise and performant way to accomplish this task. This method is highly recommended as it leverages Spark's internal optimizations for filtering column values, enabling efficient processing of massive distributed datasets.

The standard filtering operation requires passing a Column expression to the `filter` transformation. For date ranges, constructing this expression involves selecting the target date column and applying the between condition, which takes two arguments: the inclusive start date and the inclusive end date. This simple declarative syntax eliminates the need for complex SQL expressions or chaining multiple greater-than/less-than operators, leading to cleaner and more readable code.

It is critical that the column being filtered is of a recognized date or timestamp type, or that the string format matches the expected **YYYY-MM-DD** standard used by Spark for lexicographical comparison. If the dates are not standardized, Spark may produce inaccurate filtering results. The following example demonstrates the fundamental pattern used to select rows in a PySpark

DataFrame based on a temporal boundary.

The Core Mechanism: Using the filter and between Functions

The following syntax illustrates the fundamental pattern used to select rows in a PySpark DataFrame where the values in a specific column fall inclusively between a start date and an end date. We define the start and end points as elements in a Python tuple, which allows for clean unpacking directly into the `between` function call.

 # Specify start and end dates as a tuple for efficient unpacking
dates = ('2019-01-01', '2022-01-01')

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

This specific implementation demonstrates filtering a DataFrame, conventionally referred to as `df`, to retain only those rows where the date stored in the `start_date` column lies chronologically between **January 1, 2019** and **January 1, 2022**. The asterisk (`*`) before the `dates` tuple is a Python feature that unpacks the tuple's elements into separate arguments required by the between condition, ensuring a concise and functional code structure. Understanding this foundational syntax is the key to mastering date range selection in PySpark.

The filter function (or `where` alias) is a high-level transformation, meaning it does not immediately execute the computation but adds a step to the execution plan. Spark optimizes this plan, often pushing the filter operation down to the data source (predicate pushdown), which drastically improves performance, especially when filtering on indexed or partitioned columns.

Example: Implementing Date Range Filtering in PySpark

To fully grasp the mechanism, we will walk through a comprehensive, practical example. Imagine a scenario where a human resources department maintains a large dataset detailing employee onboarding information, requiring filtering based on tenure or hiring campaigns. We start by initializing the Spark environment and then creating a sample dataset that mimics real-world employment records, ensuring the date column is correctly structured for subsequent operations.

The SparkSession is the entry point for all PySpark functionality, enabling the creation and manipulation of DataFrames. Defining the schema implicitly through the data structure is convenient here, assuming the date format is consistent, which allows Spark to correctly infer the `StringType`. While `StringType` works for filtering dates in **YYYY-MM-DD** format using comparison operators, explicit casting to `DateType` is often preferred in production environments for robustness.

The following block demonstrates the entire setup process, including defining the data structure, naming the columns, and displaying the resulting DataFrame. Notice how the dates span across several years, providing a robust test case for our range filter, designed to identify employees hired during a specific three-year window.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

# Define raw employee data, including the start date
data = ,
,
,
,
,
]

Define column names for clarity
columns =

# Create the DataFrame using the defined data and column names
df = spark.createDataFrame(data, columns)

# View the initial DataFrame to verify structure
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
+--------+----------+

Applying the Date Range Filter Syntax

With our sample DataFrame successfully initialized, the next step is applying the targeted filter to extract employees who started within a specific operational window. For this demonstration, we are interested in employees who joined the company starting from **January 1, 2019** up to, but not

including, **January 1, 2022**. This range is particularly useful for analyzing staff turnover or performance during recent financial cycles.

We utilize the established pattern: calling the filter function on the DataFrame `df` and passing the column expression `df.start_date.between(start_date, end_date)`. It is crucial to remember that the between condition performs an inclusive comparison, meaning both the start and end dates specified in the arguments are considered valid matches if they exist in the dataset. This contrasts with some SQL implementations which may handle boundaries differently, making PySpark's clarity advantageous.

The code below executes the filter using the defined boundaries and immediately displays the resulting subset of the data, demonstrating the efficiency of PySpark's declarative API for complex data selection. Notice the precise specification of the dates ensures only the relevant records are processed, minimizing the data volume that needs to be shuffled or collected, thereby significantly improving job latency and resource consumption.

 # Specify start and end dates for the filtering operation
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, it is evident that only employees 'D' (hired 2019-12-21) and 'E' (hired 2021-04-14) remain. These two records satisfy the condition that their `start_date` must be greater than or equal to 2019-01-01 AND less than or equal to 2022-01-01. The employees 'A', 'B', and 'C' were excluded because their start dates precede the lower bound of 2019-01-01, falling outside the desired analysis window.

Furthermore, employee 'F', with a start date of 2022-06-26, was successfully excluded because this date falls chronologically after the upper bound specified (2022-01-01). This confirmation underscores the correct and precise execution of the inclusive date range filtering logic implemented by the between condition, ensuring data integrity for subsequent analytical tasks.

This method is far superior to manually iterating through rows or using less optimized filtering techniques, especially when handling datasets containing billions of records. The use of the filter function ensures that the data processing remains within the Spark execution engine, benefiting from distributed computing capabilities and memory management.

Alternative Approach: Calculating Row Counts within a Range

In many analytical scenarios, the objective is not necessarily to retrieve the full subset of data, but rather to quickly determine the volume or count of records that satisfy a specific temporal criterion. PySpark facilitates this by efficiently chaining the filter function directly with the count function.

This chained approach is highly computationally efficient, as Spark can often optimize the query execution plan to calculate the row count without materializing the entire filtered intermediate DataFrame. This is particularly valuable when dealing with massive datasets where collecting or displaying the filtered rows would be resource-intensive or unnecessary for the current analysis goal, such as calculating KPIs or checking data quality.

Using the same date range criteria (between 2019-01-01 and 2022-01-01), the following code snippet demonstrates how to obtain the exact number of matching rows in a single, atomic operation, which is critical for real-time reporting or monitoring applications:

 # 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

The resulting output, `2`, confirms that there are precisely two records in the DataFrame where the corresponding date in the `start_date` column adheres to the specified temporal boundaries. Employing the count function immediately after filtering is a highly optimized pattern for data validation and initial exploratory data analysis.

Data Type Considerations for Accurate Date Filtering

While the examples above successfully use string comparisons (assuming the date column is implicitly or explicitly a `StringType`), the most robust and highly recommended practice in PySpark is to ensure the column being filtered is explicitly cast as a native `DateType` or `TimestampType`. Filtering strings relies on lexicographical comparison, which works perfectly for the standard **YYYY-MM-DD** format, but can produce errors if date formats are inconsistent (e.g., mixing US and

European formats).

When dealing with large-scale production data, converting the column to the appropriate date type eliminates ambiguity and allows Spark to perform optimized, native date comparisons. If your data column `start_date` was originally a string, you should use the `F.to_date` function (imported from `pyspark.sql.functions`) before applying the filter. This ensures correctness and predictability, especially when dealing with complex time zones or daylight savings considerations which string comparisons cannot handle.

Furthermore, if you need to filter on a column that includes time components (i.e., a `TimestampType`), the between condition will operate on the full timestamp. If you pass only date strings (e.g., '2022-01-01') against a timestamp column, Spark interprets the start date as midnight (00:00:00) on the start day and the end date as midnight on the end day. If you wish to include all records from the end date up to the last millisecond of that day, you must either cast the column to `DateType` first or modify the end boundary to the start of the next day using a less-than (<) operator instead of the inclusive `between`.

Summary and Further Resources

The efficiency and simplicity of the date filtering techniques demonstrated rely heavily on PySpark's optimized API. Mastering the filter function is foundational for nearly all data cleaning and transformation tasks. It is highly recommended to consult the official Apache Spark documentation for in-depth understanding of function behavior and potential performance characteristics, especially concerning catalyst optimization and predicate pushdown.

Specifically, detailed information regarding the column expression methods, including the `between` function, is available directly from the Apache Spark documentation. This documentation covers critical aspects such as null handling, type coercion, and performance notes related to how Spark optimizes filtering operations on distributed clusters. For example, understanding how SparkSession configuration impacts date handling can be vital in production environments.

For continued learning and to expand your PySpark toolkit, the following resources and tutorials explain how to perform other common and related tasks essential for robust data processing within the PySpark environment:

Handling different date formats using `to_date` and `date_format` functions.
Performing window functions partitioned by date or time periods for sequential analysis.
Aggregating data based on specific calendar intervals (e.g., monthly or quarterly counts).
Comparing columns of different date/timestamp types accurately using built-in date functions.

Note: The complete documentation for the PySpark **between** function is available in the official

API reference, providing specific usage guidelines for different data types.