# How can I use the where() and filter() functions in PySpark to efficiently filter data?

Authored by

**stats writer**

June 24, 2024

The where() and filter() functions in PySpark are powerful tools that enable users to efficiently filter data based on specific criteria. These functions allow users to specify conditions for selecting and removing data, making it easier to extract relevant information from large datasets. By using these functions, users can significantly reduce the amount of time and resources needed to process and analyze data. Furthermore, the flexibility and simplicity of these functions make them ideal for handling complex data filtering tasks. Overall, the where() and filter() functions in PySpark are essential for efficient data manipulation and analysis, making them valuable tools for any data scientist or analyst.

In this PySpark article, you will learn how to apply a filter on DataFrame columns of string, arrays, and struct types by using single and multiple conditions and also using `isin()` with PySpark (Python Spark) examples.

## 1. Introduction to PySpark DataFrame Filtering

PySpark `filter()` function is used to create a new DataFrame by filtering the elements from an existing DataFrame based on the given condition or SQL expression. It is similar to Python's filter() function but operates on distributed datasets. It is analogous to the SQL `WHERE` clause and allows you to apply filtering criteria to DataFrame rows.

Alternatively, if you have a background in SQL, you can opt to use the `where()` function instead of `filter()`. Both functions work identically. They generate a new DataFrame containing only the rows that satisfy the specified condition.

**Related Article:**

**Note:**PySpark Column Functions provides several options that can be used with this function.

### filter() Syntax

Following is the syntax.

```
# Syntax
filter(condition)
```

Here,

`condition:` It is the filtering condition or expression. It can be specified using various constructs such as SQL expressions, DataFrame API functions, or user-defined functions (UDFs). The condition evaluates to `True` for rows that should be retained and `False` for rows that should be

discarded.

For example, let's say you have the following DataFrame. Here, I am using a DataFrame with StructType and ArrayType columns, as I will also cover examples with struct and array types.

```
# Imports
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType,StructField
from pyspark.sql.types import StringType, IntegerType, ArrayType

# Create SparkSession object
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()

# Create data
data = ,"OH","M"),
(("Anna","Rose",""),,"NY","F"),
(("Julia","","Williams"),,"OH","F"),
(("Maria","Anne","Jones"),,"NY","M"),
(("Jen","Mary","Brown"),,"NY","M"),
(("Mike","Mary","Williams"),,"OH","M")
]

# Create schema
schema = StructType()),
StructField('languages', ArrayType(StringType()), True),
StructField('state', StringType(), True),
StructField('gender', StringType(), True)
])

# Create dataframe
df = spark.createDataFrame(data = data, schema = schema)
df.printSchema()
df.show(truncate=False)
```

This yields below schema and DataFrame results.

```
root
|-- name: struct (nullable = true)
| |-- firstname: string (nullable = true)
| |-- middlename: string (nullable = true)
```

```
| |-- lastname: string (nullable = true)
|-- languages: array (nullable = true)
| |-- element: string (containsNull = true)
|-- state: string (nullable = true)
|-- gender: string (nullable = true)
```

```
+----------------------+-----------------+-----+------+
|name |languages |state|gender|
+----------------------+-----------------+-----+------+
| ||OH |M |
| ||NY |F |
| | |OH |F |
| | |NY |M |
| | |NY |M |
|| |OH |M |
+----------------------+-----------------+-----+------+
```

## 2. DataFrame filter() with Column Condition

When using `filter()` with column conditions, you typically specify the condition using column expressions. These expressions can involve comparisons, logical operations, or even functions applied to DataFrame columns. In the below example, I am using `dfObject.colname` to refer to column names.

```
# Using equal condition
df.filter(df.state == "OH").show(truncate=False)
```

```
# Output
#+----------------------+-----------------+-----+------+
#|name |languages |state|gender|
#+----------------------+-----------------+-----+------+
#| ||OH |M |
#| | |OH |F |
#|| |OH |M |
#+----------------------+-----------------+-----+------+
```

### Using not equal filter condition

To retain rows where the value in the "state" column is not equal to "OH" (Ohio), use the below

syntaxes. However, they use slightly different approaches to express the filtering condition.

```
# Not equals condition
df.filter(df.state != "OH")
.show(truncate=False)
```

```
# Another expression
df.filter(~(df.state == "OH"))
.show(truncate=False)
```

**Using `!=` operator**:

**Using `~` (Negation) operator**:

### Using col() Function

You can also use the col() function to refer to the column name. In order to use this first, you need to import `from pyspark.sql.functions import col`

```
# Using SQL col() function
from pyspark.sql.functions import col
df.filter(col("state") == "OH")
.show(truncate=False)
```

## 3. Filtering with SQL Expression

If you have an SQL background, you can use that knowledge in PySpark to filter DataFrame rows with SQL expressions.

```
# Using SQL Expression
df.filter("gender == 'M'").show()
```

```
# For not equal
df.filter("gender != 'M'").show()
df.filter("gender <> 'M'").show()
```

## 4. PySpark Filter with Multiple Conditions

In PySpark, you can apply multiple conditions when filtering DataFrames to select rows that meet specific criteria. This can be achieved by combining individual conditions using logical operators like `&` (AND), `|` (OR), and `~` (NOT). Let's explore how to use multiple conditions in PySpark DataFrame filtering:

```
# Filter multiple conditions
df.filter( (df.state == "OH") & (df.gender == "M") )
.show(truncate=False)
```

# Output
#+----------+----------+-----+------+
#|name |languages |state|gender|
#+----------+----------+-----+------+
#| ||OH |M |
#|| |OH |M |
#+----------+----------+-----+------+

The conditions are combined using the `&` operator, indicating that both conditions must be true for a row to be retained.

To use the `OR` operator, replace `&` with `|`.

```
# Filter using OR operator
df.filter( (df.state == "OH") | (df.gender == "M") )
.show(truncate=False)
```

## 5. Filter Based on List Values

The `isin()` function from the Python Column class allows you to filter a DataFrame based on whether the values in a particular column match any of the values in a specified list. And, to check not isin() you have to use the not operator (~)

```
# Filter IS IN List values
li=
df.filter(df.state.isin(li)).show()
```

```
# Output
#+--------------------+-----------------+-----+------+
#| name| languages|state|gender|
#+--------------------+-----------------+-----+------+
#| || OH| M|
#| | | OH| F|
#|| OH| M|
#+--------------------+-----------------+-----+------+

# Filter NOT IS IN List values
# These show all records with NY (NY is not part of the list)
df.filter(~df.state.isin(li)).show()
df.filter(df.state.isin(li)==False).show()
```

## 6. Filter Based on Starts With, Ends With, Contains

Use `startswith()`, `endswith()` and `contains()` methods of Column class to select rows starts with, ends with, and contains a value. For more examples on Column class, refer to PySpark Column Functions.

```
# Using startswith
df.filter(df.state.startswith("N")).show()

# Output
#+--------------------+-----------------+-----+------+
#| name| languages|state|gender|
#+--------------------+-----------------+-----+------+
#| || NY| F|
#|| | NY| M|
#| | | NY| M|
#+--------------------+-----------------+-----+------+

#using endswith
df.filter(df.state.endswith(H")).show()

#contains
df.filter(df.state.contains(H")).show()
```

## 7. Filtering with Regular Expression

If you are coming from SQL background, you must be familiar with `like` and `rlike` (regex like). PySpark also provides similar methods in the Column class to filter similar values using wildcard characters. You can use rlike() for case insensitive.

```
# Prepare Data
data2 =
df2 = spark.createDataFrame(data = data2, schema = )
```

```
# like - SQL LIKE pattern
df2.filter(df2.name.like("%rose%")).show()

# Output
#+---+----------+
#| id| name|
#+---+----------+
#| 5|Rames rose|
#+---+----------+

# rlike - SQL RLIKE pattern (LIKE with Regex)
# This check case insensitive
df2.filter(df2.name.rlike("(?i)^*rose$")).show()

# Output
#+---+------------+
#| id| name|
#+---+------------+
#| 2|Michael Rose|
#| 4| Rames Rose|
#| 5| Rames rose|
```

## 8. Filtering Array column

To filter DataFrame rows based on the presence of a value within an array-type column, you can employ the first syntax. The following example uses array_contains() from PySpark SQL functions. This function examines whether a value is contained within an array. If the value is found, it returns true; otherwise, it returns false.

```
# Using array_contains()
from pyspark.sql.functions import array_contains
df.filter(array_contains(df.languages,"Java"))
.show(truncate=False)
```

```
# Output
#+---------------+-----------------+-----+------+
#|name |languages |state|gender|
#+---------------+-----------------+-----+------+
#|||OH |M |
#| ||NY |F |
#+---------------+-----------------+-----+------+
```

## 9. Filtering on Nested Struct columns

In case your DataFrame consists of nested struct columns, you can use any of the above syntaxes to filter the rows based on the nested column.

```
# Struct condition
df.filter(df.name.lastname == "Williams")
.show(truncate=False)
```

```
# Output
#+---------------------+-----------+-----+------+
#|name |languages |state|gender|
#+---------------------+-----------+-----+------+
#| ||OH |F |
#|||OH |M |
#+---------------------+-----------+-----+------+
```

## 10. FAQs on filter()

**What is the difference between where and filter in PySpark?**

In PySpark, both `filter()` and `where()` functions are used to select out data based on certain conditions. They are used interchangeably, and both of them essentially perform the same operation.

**Is DataFrame filtering in PySpark lazy**

Yes, DataFrame filtering in PySpark follows lazy evaluation, meaning the filtering operation is only executed when an action is performed on the DataFrame

**Can I use SQL expressions for DataFrame filtering in PySpark?**

You can use SQL expressions for filtering in PySpark by using functions like `expr()` or by registering the DataFrame as a temporary view and executing SQL queries on it.

**How can I optimize DataFrame filtering performance in PySpark?**

Optimizing DataFrame filtering performance in PySpark involves strategies such as minimizing data shuffling, repartitioning, and caching intermediate results where appropriate.

**Are there any limitations to DataFrame filtering in PySpark?**

While DataFrame filtering in PySpark is powerful, it may encounter limitations related to complex conditions, performance overhead, and resource management, which require careful consideration and optimization.

## 11. Conclusion

Examples explained here are also available at PySpark examples GitHub project for reference.

Overall, the `filter()` function is a powerful tool for selecting subsets of data from DataFrames based on specific criteria, enabling data manipulation and analysis in PySpark. In this tutorial, you have learned how to filter rows from PySpark DataFrame based on single or multiple conditions and SQL expression, also learned how to filter rows by providing conditions on the array and struct column with Spark with Python examples.

Alternatively, you can also use `where()` function to filter the rows on PySpark DataFrame.

Happy Learning !!

## Related Articles