Use “OR” Operator in PySpark (With Examples)

Authored by

stats writer

November 16, 2025

Introduction to Conditional Filtering in PySpark

When working with large datasets, the ability to selectively retrieve records based on complex criteria is fundamental. In PySpark DataFrames, this is achieved using the powerful filter function (or its alias, `.where()`). Often, filtering requires satisfying one of several possible conditions, necessitating the use of the **OR operator**. Understanding how to correctly implement this logical operator is crucial for efficient data manipulation and accurate query formulation.

The **OR operator** is a core concept derived from Boolean algebra, which dictates that a combined condition returns true if at least one of its constituent conditions is true. PySpark offers two distinct, yet functionally equivalent, methods for applying this disjunctive logic to combine multiple filtering requirements within a DataFrame query. Choosing the right method often depends on context, readability preferences, and whether you are working with native SQL syntax or programmatic Python expressions.

Here we explore the two most common and effective ways to filter a PySpark DataFrame using the logical OR constraint. These methods ensure that your data pipelines are both robust and precise, allowing for complex data selection based on flexible criteria.

Method 1: Using SQL String Expressions with "OR"

The first approach leverages standard SQL expressions passed as a single string argument to the `.filter()` transformation. This method is often favored by users familiar with traditional database querying languages, as it offers a syntax that closely mimics SQL's `WHERE` clause. The keyword `or` is used directly within the quoted string to separate the conditions, treating the entire argument as a SQL predicate.

This method simplifies complex logic by condensing it into a single readable string, which can improve code clarity for those deeply rooted in SQL environments. However, developers must exercise caution regarding syntax correctness, as Python does not validate the SQL string until runtime on the Spark cluster. When using this technique, ensure that column names and literal values are properly quoted or handled according to SQL standards to avoid runtime exceptions.

#filter DataFrame where points is greater than 9 *or* team equals "B"
df.filter('points>9 or team=="B"').show()

Method 2: Using the PySpark Column API with the Pipe Symbol (|)

The second, and often more robust, method involves constructing conditions programmatically using the PySpark Column API. Instead of passing a string, we define individual conditions using

native Python syntax combined with the bitwise OR operator, represented by the pipe symbol (`|`). When applied to PySpark Column objects, this symbol is intentionally overloaded to represent the logical OR operation, distinct from its typical use in standard Python for bitwise operations.

A significant advantage of this programmatic approach is that conditions are built using actual PySpark column objects, offering better integration with the rest of the Python ecosystem and providing earlier error detection during development. Crucially, when combining multiple conditions using the `|` symbol, it is mandatory to wrap each individual condition within parentheses (e.g., `(condition1) | (condition2)`). This guarantees that the order of operations is correctly interpreted, preventing unintended precedence issues where comparison operators like `>` or `==` might interact incorrectly with the logical OR operator `|`.

#filter DataFrame where points is greater than 9 *or* team equals "B"
df.filter((df.points>9) | (df.team=="B")).show()

Setting Up the Sample PySpark DataFrame

To effectively demonstrate both filtering techniques, we first need to establish a working PySpark DataFrame. This dataset represents performance metrics for several fictional teams across different conferences. This setup requires initializing a **SparkSession**, defining the raw data structure, specifying column headers, and finally using the `createDataFrame` method to materialize the structure into a distributed dataset.

The sample data contains six rows and four columns: `team` (a categorical identifier), `conference` (East or West), `points` (an integer performance metric), and `assists` (a secondary integer metric). Observing the structure of this base DataFrame is an essential prerequisite before applying the filters, as it allows for immediate visual verification of the resulting output in the subsequent examples.

The following code snippet initializes the Spark environment and generates the DataFrame structure used throughout this tutorial:

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()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|  A|      East|    11|      4|
|  A|      East|     8|      9|
|  A|      East|    10|      3|
|  B|      West|     6|     12|
|  B|      West|     6|      4|
|  C|      East|     5|      2|
+----+----------+------+-------+
```

Example 1: Demonstrating Filter with the SQL Keyword "OR"

In this first practical application, we utilize the string-based filter function combined with the `or` keyword. Our precise objective is to isolate rows where the team scored more than 9 points in the `points` column, **OR** where the team identifier in the `team` column is equal to "B". This query successfully implements the logical union of records satisfying either performance thresholds or specific group membership requirements.

The condition is encapsulated entirely within a single string: `'points>9 or team=="B"'`. PySpark's optimization engine, Catalyst, parses this string as a standard SQL predicate and evaluates it row by row. If the condition (points > 9) evaluates to true, or the condition (team is "B") evaluates to true, the row is included in the resulting DataFrame. This flexibility is characteristic of the **OR operator**.

Reviewing the resulting DataFrame confirms that rows meeting at least one of the two specified criteria are retained. Specifically, rows from Team A with 11 and 10 points satisfy the points threshold, while both rows associated with Team B (regardless of their points) satisfy the team identifier condition.

#filter DataFrame where points is greater than 9 *or* team equals "B"
df.filter('points>9 or team=="B"').show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
|  A|      East|    11|      4|
|  A|      East|    10|      3|
|  B|      West|     6|     12|
|  B|      West|     6|      4|
+----+----------+------+-------+
```

We notice that each of the rows in the resulting DataFrame meets at least one of the following conditions:

The value in the points column is greater than 9
The value in the team column is equal to "B"

It is important to note the robustness of this method; while this example uses only one `or` operator, you can chain multiple **SQL expressions** together to form highly complex filtering logic using any combination of logical operators (AND, OR, NOT) within the string predicate.

Example 2: Demonstrating Filter with the Pipe Symbol (|)

The second example achieves the exact same filtering result but employs the programmatic PySpark Column expression syntax, which leverages the pipe symbol (`|`) for the logical OR operation. This method requires explicitly referencing DataFrame columns (e.g., `df.points`) and, crucially, wrapping each comparison operation in parentheses. This ensures Python correctly handles operator precedence before applying the OR operator logic.

The structure `(df.points > 9) | (df.team == "B")` explicitly defines two distinct boolean conditions linked by the logical OR. The explicit use of parentheses is essential because the relational operators (like `>` and `==`) have a higher precedence than the **bitwise operator** (`|`) in Python. Without parentheses, the expression would be evaluated incorrectly, leading to errors or undesirable results. This disciplined structure contributes to safer and more readable PySpark code.

As expected, running the query using the programmatic `|` symbol yields a DataFrame identical to the one produced in Example 1. This verifies that both methods are valid and produce equivalent results when implemented correctly, providing developers with the flexibility to choose the style that best fits their team's coding standards.

#filter DataFrame where points is greater than 9 *or* team equals "B"
df.filter((df.points>9) | (df.team=="B")).show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 10| 3|
| B| West| 6| 12|
| B| West| 6| 4|
+----+----------+------+-------+
```

Notice that each of the rows in the resulting DataFrame meet at least one of the following conditions, confirming the logical union operation:

The value in the points column is greater than 9
The value in the team column is equal to "B"

Also note that this DataFrame matches the DataFrame from the previous example exactly, reinforcing the equivalence of the two distinct syntaxes for applying logical OR filtering in PySpark.

Comparison and Best Practices for OR Filtering

Understanding the trade-offs between the SQL string method and the Column API method is vital for professional PySpark development. While both methods are functionally sound, they cater to different needs and offer varying degrees of robustness and developer experience.

The string-based approach (Method 1) is quick for prototyping and simpler for those migrating from a pure SQL environment, offering concise syntax. However, it requires careful manual handling of quotes and escapes, and errors related to column spelling or incompatible types are only discovered late in the execution pipeline. This makes debugging more challenging on distributed systems.

The Column API method (Method 2) using the bitwise operator `|` is the preferred standard for complex, production-ready ETL jobs. By building conditions using explicit column references (`df.col`), you benefit from PySpark's internal validation mechanisms and static analysis capabilities. Although the necessity of mandatory parentheses adds verbosity, this structure ensures predictable operator precedence and minimizes the risk of logical errors in complex filtering scenarios. This method integrates seamlessly with functions from `pyspark.sql.functions`, further enhancing its flexibility.

Advanced Use Cases: Chaining Multiple Conditions

The real power of conditional filtering is realized when dealing with three or more independent conditions that must be combined using the OR logic. Both syntaxes support chaining these conditions indefinitely.

If we needed to filter for rows where the team is 'A', OR the points are greater than 10, OR the assists are less than 5, the logic must be applied sequentially. Using the programmatic approach, this involves chaining multiple conditional expressions separated by the pipe symbol (`|`), ensuring each condition remains individually parenthesized:

```
df.filter((df.team == 'A') | (df.points > 10) | (df.assists < 5)).show()
```

This structured method is highly scalable and prevents ambiguity in complex logical operations, which is critical when processing extremely large-scale PySpark DataFrames where execution time is paramount. Maintaining clean, explicit logic allows the Spark optimizer to efficiently plan the execution flow, often leading to better performance outcomes than overly complex or poorly structured string-based SQL expressions.