How to Use the OR Operator in PySpark to Filter Data

Authored by

stats writer

February 10, 2026

An Introduction to Data Filtering with the PySpark OR Operator

In the expansive ecosystem of big data processing, **PySpark** stands out as a premier tool for managing large-scale datasets with efficiency and speed. One of the most fundamental operations performed during data transformation is filtering, which allows analysts to isolate specific subsets of information based on defined criteria. The **OR** operator serves as a critical **logical operator** within this environment, enabling the evaluation of multiple conditions simultaneously. By returning a **Boolean** value of "True" if at least one condition in a set is met, it provides the flexibility required to handle complex real-world data scenarios where a single criterion is often insufficient.

The ability to construct intricate logical expressions is essential for building robust data pipelines. Whether you are dealing with financial records, user logs, or sensor data, you will frequently encounter situations where you need to extract records that satisfy any one of several possible requirements. For instance, a marketing analyst might need to identify customers who have either made a purchase in the last thirty days or have signed up for a premium newsletter. In such cases, the **OR** operator acts as the primary mechanism for broadening the scope of a search without losing the precision provided by the individual conditions themselves.

Beyond simple selection, the **OR** operator is deeply integrated into the **Apache Spark** engine's optimization framework. When a developer applies a filter using this operator, the underlying **API** translates the high-level code into optimized logical plans. This ensures that the computational cost of evaluating multiple conditions remains manageable, even when the **DataFrame** contains billions of rows. Understanding how to leverage this operator effectively is a hallmark of an expert data engineer, as it directly impacts both the readability of the code and the performance of the data processing job.

Core Logic and the Mechanics of Disjunction

At its heart, the **OR** operator implements a concept known as logical disjunction. In a **Boolean** context, a disjunction is true if any of its operands are true. If all operands are false, the entire expression evaluates to false. This basic principle of formal logic is what allows **PySpark** to perform complex row-wise evaluations. For example, if we are filtering a dataset where column A must be true or column B must be true, the filter identifies every row that satisfies either condition A, condition B, or both simultaneously.

In the **Python** language, the standard **OR** keyword is used for scalar values and control flow statements like if-else blocks. However, when working with distributed datasets in **Apache Spark**, the syntax shifts to accommodate the needs of a distributed environment. Developers can choose between using a **SQL**-like string syntax or a more programmatic bitwise operator syntax. Each approach has its merits, but they both ultimately rely on the same fundamental logical rules to

produce a subset of the original data.

To illustrate the basic logic, consider a **DataFrame** filter designed to select rows where a specific condition is met. The code `df.filter((df == True) | (df == True))` explicitly tells the engine to scan each record and check if either of the specified columns contains a truthy value. This capability is not just limited to simple equality checks; it can be combined with inequalities, null checks, and pattern matching to create highly sophisticated data selection logic that is both expressive and powerful.

Establishing the Environment: The SparkSession and DataFrame

Before we can explore the practical implementation of the **OR** operator, it is necessary to establish a functional **PySpark** environment. This process begins with the initialization of a **SparkSession**, which serves as the entry point for all Spark functionality. The session manages the connection to the cluster and provides the necessary tools to create and manipulate data structures. In a typical development workflow, you would define your data, specify the column headers, and then transform that raw information into a distributed **DataFrame**.

The following code snippet demonstrates the standard procedure for creating a mock dataset that we will use for our examples. This dataset includes information about sports teams, their conferences, and their performance metrics such as points and assists. By visualizing the data before applying filters, we can better understand how the **OR** logic transforms the input into the desired output.

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

As shown in the output above, our initial **DataFrame** contains six distinct rows. Each row represents a specific observation with varying values across four different features. This structure is ideal for testing logical filters because it contains diverse values that allow us to see how different conditions interact. With this foundation in place, we can now proceed to explore the two primary methods for applying the **OR** operator.

Method 1: Implementing SQL-Style String Expressions

The first method for applying an **OR** filter in **PySpark** involves using a **SQL**-style string. This approach is often favored by developers who have a background in traditional relational databases, as the syntax is nearly identical to a standard WHERE clause. By passing a string directly into the `filter()` or `where()` method, you can define your logic using natural language keywords. In this case, the literal word "or" is used to separate the conditions you wish to evaluate.

One significant advantage of the string-based method is its readability. It allows for a clean and concise expression that is easy for other developers to interpret. However, because the logic is encapsulated within a string, it lacks the compile-time safety checks that come with the programmatic approach. Despite this, it remains a highly effective way to perform rapid data exploration and is frequently used in production environments where **SQL** compatibility is a priority. The syntax is straightforward: wrap the entire condition in quotes, using the standard comparison operators like greater than (`>`) or equality (`==`).

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

When this code is executed, the **Apache Spark** engine parses the string, identifies the columns involved, and applies the logic to every partition of the data. This demonstrates the seamless

integration between **SQL** semantics and the **DataFrame API**, providing a bridge for those transitioning from database management to big data engineering.

Deep Dive: Filtering Data Using the String-Based OR Approach

To see the string-based **OR** operator in action, we can apply it to our sports dataset. Let us consider a scenario where we want to find all rows where the "points" metric exceeds 9 or the "team" name is exactly "B". This request involves two different columns and two different types of comparisons (numerical and categorical). By using the string syntax, we can combine these disparate requirements into a single, cohesive command.

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

The resulting **DataFrame** now only contains four rows. If we analyze the results, we can see that rows 1 and 3 were included because their "points" value (11 and 10, respectively) satisfied the first condition. Rows 4 and 5 were included because their "team" value ("B") satisfied the second condition. Row 2 and Row 6 were excluded because they failed both tests. This clearly illustrates how the **OR** operator expands the selection criteria to capture any record that meets at least one part of the query.

It is important to note that the string-based filter can be extended indefinitely. You are not limited to just two conditions; you can string together multiple "or" keywords to create a wide net for your data. For example, you could filter for `points > 9 or team == "B" or assists < 5`. This flexibility makes it an excellent choice for complex filtering tasks where multiple potential indicators of interest exist within the dataset.

Method 2: Using the Bitwise Pipe Symbol for Logical Disjunction

The second and perhaps more common method in modern **PySpark** development is the use of the pipe symbol (`|`). This is a bitwise **logical operator** that **Apache Spark** redefines for use with

Column objects. Unlike the string method, this approach is fully programmatic and allows for the use of Python variables and dynamic column references. When using this method, it is crucial to wrap each individual condition in parentheses to ensure the correct order of operations, as the bitwise operator has higher precedence than comparison operators in **Python**.

The pipe symbol approach is often preferred in large-scale software engineering projects because it integrates better with IDE features like autocomplete and syntax highlighting. It also allows for more complex logic that can be built dynamically at runtime. For instance, you could programmatically generate a list of conditions and join them using the `|` operator. This level of control is vital for building generalized data processing frameworks that need to adapt to different schemas and requirements without manual code changes.

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

By using the `df.column_name` or `df` syntax, you are directly interacting with the **DataFrame** schema. This method provides a layer of safety, as any typos in column names will result in an immediate error, helping to catch bugs early in the development cycle. The pipe symbol serves as a powerful shorthand for "either this or that," making it a staple in the toolkit of any serious Spark developer.

Deep Dive: Filtering Data Using the Column-Based Pipe Operator

Applying the pipe operator to our dataset yields the same logical results as the string-based method, but with a different syntax. This consistency is important, as it ensures that the choice of method is a matter of style and context rather than a functional difference. Let us re-run our previous filter--selecting rows where points are greater than 9 or the team is "B"--using the bitwise syntax to confirm the behavior.

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

As expected, the output is identical to the previous example. The rows for Team A with 11 and 10 points are present, as are both entries for Team B. The logic remains robust: the **OR** operation ensures that a row is kept if it satisfies the first condition, the second condition, or both. This redundancy check is a key feature of the operator, ensuring no relevant data is dropped during the **filtering** process.

When working with the pipe operator, developers often find it helpful to break down very long logical expressions into multiple lines. Because **Apache Spark** uses **lazy evaluation**, these filters are not executed until an action like `.show()` or `.collect()` is called. This allows the Spark Catalyst Optimizer to look at the entire chain of filters and combine them into the most efficient physical execution plan possible, minimizing the amount of data that needs to be read from storage.

Advanced Considerations: Performance and Complex Logic

While the **OR** operator is straightforward to use, its performance implications in large datasets are worth considering. In many cases, using an **OR** condition can prevent the Spark optimizer from using certain "predicate pushdown" optimizations that are more easily applied to **AND** conditions. When you use **AND**, Spark can often discard large chunks of data early in the process. With **OR**, the engine must ensure that it doesn't discard a row that might satisfy the *second* condition even if it fails the first. Consequently, extensive use of **OR** logic might lead to slightly longer execution times compared to more restrictive filters.

To mitigate performance issues, it is often beneficial to ensure that the most "selective" condition (the one that filters out the most data) is evaluated efficiently. Additionally, when dealing with a large number of **OR** conditions on the same column--for example, `team == 'A' or team == 'B' or team == 'C'`--it is much more efficient to use the `isin()` method. The `isin()` function is optimized for these specific scenarios and provides a much cleaner syntax than chaining multiple pipe operators together.

Finally, mastering the **OR** operator involves understanding how it interacts with other **logical operators** like **AND** (`&`) and **NOT** (`~`). By combining these, you can create complex nested logic, such as `(condition_1 | condition_2) & condition_3`. Always remember the importance of parentheses in these expressions; they are the only way to guarantee that the **PySpark** engine interprets your intended logic correctly. With these tools and techniques, you can handle virtually any data selection challenge that comes your way.