Learning PySpark: How to Use the OR Operator for Data Filtering with Examples

Authored by

Mohammed looti

November 11, 2025

Understanding Logical OR Operations in PySpark

When working with large-scale data processing using the PySpark library, one of the most fundamental tasks is filtering data based on complex, conditional criteria. Often, these criteria require evaluating multiple conditions simultaneously, where satisfying any single condition is sufficient to retain a record. This necessity highlights the critical role of the **logical OR operation**. Within the context of a PySpark DataFrame, the **OR operator** allows users to select rows where either Condition A is true, Condition B is true, or both are true. This mechanism effectively broadens the selection scope, offering a flexible contrast to the restrictive nature of the **AND operator**.

Implementing the logical **OR** in PySpark can be achieved through two distinct, yet functionally equivalent, methodologies. The choice between these methods often depends on developer preference, the complexity of the filtering logic required, and whether the coding style should emulate traditional database query languages or embrace the native, column-based expressions inherent to the DataFrame API. The first approach utilizes a string expression containing the keyword `or`, which closely mirrors SQL syntax. The second, and generally preferred method for building complex, programmatic operations, employs the **bitwise OR symbol** (`|`) applied directly to column objects, which necessitates careful management of operator precedence using parentheses.

This comprehensive guide explores both methods for applying the **OR operator** within the PySpark `filter()` function, providing detailed code examples and examining the nuances of each implementation. We will first establish a working environment and a sample DataFrame, which will serve as the foundation for demonstrating these powerful data manipulation techniques. Understanding these two approaches is crucial for efficiently selecting and subsetting massive datasets in distributed computing environments, a core task in modern data engineering.

Setting Up the PySpark Environment and Sample Data

Before diving into the practical filtering examples, we must first initialize a **Spark Session**. This session is the essential entry point for utilizing all Spark functionality, acting as the coordinating component for distributed resource management. Once the session is active, we proceed to create a sample DataFrame. This sample data is designed to simulate a small dataset tracking team performance, including columns for team name, conference, points scored, and assists made. This structured setup ensures that the demonstration of the **OR** filtering logic is clear, easily reproducible, and provides a tangible context for the conditional operations we will perform.

The following code snippet demonstrates the necessary steps required for initialization and data creation. We begin by importing the required libraries and then use

`SparkSession.builder.getOrCreate()` to either retrieve an existing session or instantiate a new one, which is standard practice in modern PySpark development. We explicitly define the raw data records and the list of column names, allowing Spark to infer the appropriate data types and structure, ensuring the resulting data structure is correctly typed for subsequent filtering operations.

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

# Define the raw data records
data = ,
,
,
,
,
]

Define the column names
columns =

# Create the DataFrame from the data and columns
df = spark.createDataFrame(data, columns)

# View the resulting DataFrame structure and contents
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
+----+----------+------+-------+

Method 1: Utilizing the String-Based `or` Operator (SQL Style)

The first technique for implementing **OR** logic relies on passing a SQL-like expression string directly into the DataFrame's `filter()` function. This method is particularly accessible to users already familiar with standard SQL syntax, as it allows for concise and highly readable filtering

statements that minimize the Python code required. When using this approach, the entire conditional logic--including column references, comparison operators, and the `or` keyword--is fully encapsulated within a single string argument. PySpark internally parses this string expression and translates it into an optimized execution plan for distribution across the cluster.

For our initial example, we aim to filter the DataFrame (`df`) to include only those rows where the value in the `points` column is significantly high **OR** the record belongs to a specific team designation. Specifically, we are looking for rows where `points` is greater than 9 **OR** the `team` column is exactly "B". This demonstrates a classic application of the **logical operator** of disjunction. The string-based approach requires careful attention to quotation management, ensuring that column names are correctly referenced and any string values (such as "B") are properly enclosed within the inner quotes of the filter string to prevent parsing errors.

 # Filter DataFrame using the SQL-like 'or' keyword
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|
+----+----------+------+-------+
```

Upon reviewing the resulting DataFrame, we can immediately confirm that every selected row satisfies at least one of the conditions specified by the **logical OR** operation. For example, the first two rows satisfy the condition that `points` are greater than 9 (11 and 10), even though their team is "A". Conversely, the last two rows satisfy the condition that `team` equals "B", even though their `points` value (6) is not greater than 9. This method is highly flexible and can easily accommodate multiple `or` operators chained together to handle increasingly complex filtering requirements.

The value in the `points` column is greater than 9.

The value in the `team` column is equal to "B".

Method 2: Employing the Column-Based Bitwise OR Symbol (|)

The second method, which is often considered the more idiomatic Python/PySpark way to implement **OR** logic, involves using the **bitwise OR symbol** (`|`) directly on column expressions.

This approach treats conditions not as components of a single SQL string, but as individual PySpark Column objects that evaluate to Boolean results (True/False). When these column expressions are combined using the `|` operator, the result is a new Boolean column indicating whether either expression evaluated to True for a specific row. This method is generally preferred by Python developers as it integrates seamlessly with native Python syntax and operator overloading.

A critical consideration when using the `|` symbol for **logical operations** is understanding **operator precedence**. In Python, standard comparison operators (e.g., `>`, `==`, `<`) inherently have a higher precedence than the **bitwise operators** (`|` and `&`). Consequently, each individual condition must be explicitly enclosed within parentheses. Failure to include these parentheses will cause Python to attempt to apply the **bitwise OR** operation between the column objects before the comparison operations are completed, invariably leading to errors or incorrect evaluation results in the distributed context.

Using the same filtering criteria as before (`points > 9` **OR** `team == "B"`), the column-based approach looks as follows. Note how each sub-condition is wrapped in parentheses, ensuring that the comparison is evaluated first, producing a Boolean series, before the `|` operator performs the final **logical disjunction** across the rows. This structure is significantly cleaner and more reliable for complex, programmatic generation of filters, especially when conditional logic needs to be constructed dynamically.

 # 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 anticipated, the output DataFrame generated by the **bitwise OR** method is functionally identical to the result obtained using the string-based `or` keyword from Method 1. This confirms the functional equivalence of the two approaches for simple logical filtering. Developers should be aware that while the string method is often simpler for quick, one-off filters, the column-based method using `|` is generally more robust for integrating with other sophisticated PySpark column operations and transformations.

The value in the `points` column is greater than 9.

The value in the `team` column is equal to "B".

Choosing the Right Approach: Best Practices and Performance

When deciding between the string-based `or` (Method 1) and the column-based `|` (Method 2), developers must weigh factors related to code clarity, maintainability, and compatibility with the surrounding codebase. The string method provides a high degree of readability for those accustomed to SQL syntax, making the filtering intent immediately clear. It is also very concise, minimizing the amount of code required for simple, static filters. However, its primary limitation arises when attempting to integrate Python variables or complex expressions into the filter, as this requires messy string formatting and concatenation, which can be prone to runtime errors.

The column-based approach using `|` is the preferred standard for complex and dynamic data manipulation in PySpark. Since it works directly with PySpark Column objects, it allows for seamless integration with variables, user-defined functions, and complex conditional logic built up across multiple lines of code. Although it demands the strict use of parentheses to manage **operator precedence**, this rigorous structure ultimately results in more maintainable and less error-prone code when dealing with numerous combined conditions in a production environment.

It is crucial to note that both methods are optimized internally by the framework. Regardless of whether you use the SQL string expression or the Column object expressions, the **Spark Catalyst Optimizer** translates these operations into the same efficient, optimized execution plan. Therefore, performance differences between the two methods are negligible for standard filtering tasks. The decision should primarily be based on coding style consistency: use the SQL string for quick, static filters, and use the `|` method for dynamic or complex logic that needs to interact heavily with other PySpark column functions.

Summary of Logical OR Filtering and Further Resources

Effectively filtering data is central to data preparation in any **data engineering** pipeline, and the logical **OR operator** is essential for selecting records based on a disjunction of conditions across wide datasets. PySpark offers two robust methods for achieving this--the string-based `or` and the column-based bitwise OR symbol (`|`)--allowing developers flexibility in their implementation. While the performance is identical due to Spark's internal optimization, the column-based approach is generally recommended for its superior ability to handle complex, dynamic, and production-ready filtering logic.

The following tutorials explain how to perform other common data transformation and filtering tasks in PySpark, building upon the foundational knowledge presented here:

Filtering DataFrames using the **AND** Operator (The restrictive counterpart to OR).

Applying Complex Conditional Logic with `when().otherwise()` expressions.

Understanding and using Spark's `withColumn()` transformation for feature engineering.