Learning PySpark: Applying OR Conditions with the WHEN Function for Data Transformation

Authored by

Mohammed looti

November 11, 2025

The foundation of effective data manipulation in a distributed environment like Apache Spark relies heavily on the ability to apply sophisticated, row-wise conditional logic. When processing massive volumes of data using PySpark, data engineers frequently encounter scenarios requiring the creation of new feature columns based on multiple potential criteria. This necessity makes the combination of the robust PySpark when function and the logical **OR condition** an essential tool. This comprehensive guide offers a detailed, practical demonstration of how to implement multi-criteria conditional transformations using `F.when()` and the bitwise **OR** operator within a DataFrame context.

Mastering Conditional Logic: The `F.when` Function and the OR Operator

The core mechanism for implementing conditional logic in PySpark is the `F.when()` function, which mirrors the standard `IF-THEN-ELSE` control flow found in SQL or traditional programming languages. This function evaluates a specific condition and, if that condition yields **True**, assigns a predetermined value to the newly defined column. The real power emerges when we need to evaluate several conditions simultaneously, particularly when the desired outcome is met if **at least one** condition is satisfied--this is the fundamental definition of the **OR condition**.

To link multiple boolean expressions using **OR** logic within DataFrame column operations, we must utilize the bitwise **OR** operator, which is represented by the pipe symbol (`|`). A critical requirement when using this operator is the strict enclosure of all individual conditions within parentheses. This structural necessity prevents common errors related to operator precedence, ensuring that the boolean comparison is executed correctly before the bitwise operation takes place.

Once the complex conditional expression is constructed, it is passed to the withColumn method. This method is the primary utility for either creating a new column or replacing an existing one based on the result of the `F.when()` evaluation. The following standard syntax illustrates how to create a new column, **B10**, which flags rows where the `team` column is 'B' **OR** the `points` column exceeds 10.

import pyspark.sql.functions as F

df_new = df.withColumn('B10', F.when((df.team=='B') | (df.points>10), 1).otherwise(0))

In this sample code, the **OR** operator (`|`) successfully merges two distinct boolean criteria: `(df.team == 'B')` and `(df.points > 10)`. If the evaluation of either expression results in `True` for a given row, the new column **B10** is assigned the value `1`, as specified in the `when` clause. Conversely, if both conditions are simultaneously `False`, the `otherwise` clause triggers, assigning `0`. This mechanism is vital for complex data filtering and categorization tasks in large-scale data processing.

A value of **1** is returned if the **team** column is 'B' *or* the **points** column is greater than 10.

A value of **0** is returned otherwise, meaning both conditions were `False`.

It is important to reiterate that the **`|`** symbol is the designated bitwise **OR** operator used specifically for combining boolean conditions within PySpark column expressions, differentiating it from the standard Python logical `or` keyword.

Setting Up the Environment and Sample PySpark DataFrame

To ground this conditional logic in a tangible example, we must first prepare a sample DataFrame. Our illustrative dataset will represent basketball player statistics, including essential metrics such as team affiliation, player position, and points scored. The preparation involves a standard sequence of steps: initializing a necessary **SparkSession**, defining the raw data structure (a list of lists), specifying the schema (column names), and finally invoking the `createDataFrame` method to construct the immutable data structure.

The resulting DataFrame serves as a clean, structured foundation upon which we can apply and test our sophisticated conditional transformations. Specifically, we will focus on the interaction between the string column `team` and the numerical column `points` to thoroughly demonstrate the efficacy of the **OR condition**.

The following code block provides the complete setup required to instantiate the sample data environment, making the data immediately available for manipulation using PySpark APIs:

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|position|points|
+----+--------+------+
| A| Guard| 11|
| A| Guard| 8|
| A| Forward| 22|
| A| Forward| 22|
| B| Guard| 14|
| B| Guard| 14|
| B| Forward| 13|
| B| Forward| 7|
+----+--------+------+
```

Executing Multi-Criteria Transformations using `F.when` and OR

With the sample DataFrame, `df`, initialized, we are ready to apply our central conditional logic. The objective remains clear: generate a new indicator column, **B10**, that flags players who satisfy either of the following criteria: membership in Team 'B' **OR** scoring a total of points greater than 10. A result of **1** signifies that the condition is met, while **0** indicates that neither criterion was satisfied.

This transformation requires importing the `functions` module from `pyspark.sql`, conventionally aliased as `F`. We then employ the crucial withColumn method, which handles the creation and population of the new column by executing the specified conditional expression row by row across the distributed dataset. The explicit use of parentheses around each boolean expression is non-negotiable for the successful execution of the bitwise **OR** operator (`|`).

The following comprehensive code block demonstrates the practical application of this logic, creating the **B10** column and immediately displaying the resulting DataFrame to verify the output:

```
import pyspark.sql.functions as F

#create new DataFrame
df_new = df.withColumn('B10', F.when((df.team=='B') | (df.points>10), 1).otherwise(0))

#view new DataFrame
df_new.show()
```

```
+----+--------+------+---+
|team|position|points|B10|
+----+--------+------+---+
| A| Guard| 11| 1|
| A| Guard| 8| 0|
| A| Forward| 22| 1|
| A| Forward| 22| 1|
| B| Guard| 14| 1|
| B| Guard| 14| 1|
| B| Forward| 13| 1|
| B| Forward| 7| 1|
+----+--------+------+---+
```

Interpreting Results and Validating the Logical OR Condition

A careful examination of the output from `df_new.show()` confirms the successful application of the multi-criteria **OR condition**. Any row in the resulting DataFrame where the `B10` column contains the value **1** has satisfied the defined logic: the player either belongs to Team 'B' **OR** accumulated more than 10 points. This validation process is crucial for ensuring data quality during complex feature engineering workflows.

We can observe specific examples that highlight the nature of the logical OR condition. Consider the first row: Team 'A' with 11 points. Although the team criterion (Team 'B') is false, the points criterion (points > 10) is true, resulting in a **1**. Conversely, look at the final row: Team 'B' with 7 points. Here, the team criterion is true, leading to a **1**, even though the points criterion is false. In both cases, because at least one condition was met, the logical OR evaluated to true.

The sole row that receives a value of **0** is the player from Team 'A' who scored 8 points. This outcome is significant because it is the only instance where both conditions--Team 'B' membership AND points greater than 10--are simultaneously false. This clear demarcation demonstrates the precise and accurate functionality of the `F.when()` transformation when correctly combined with the bitwise **OR** operator.

Flexibility in Return Values: Handling Non-Numerical Outputs

While using numerical flags (**1** for True, **0** for False) is common practice in quantitative data analysis, the PySpark when function offers complete flexibility regarding the return data type. For enhanced readability in reporting or to meet specific data modeling requirements, it is straightforward to return descriptive categorical string values, such as 'Yes' or 'No', instead of integers.

To modify the output to strings, the developer simply needs to enclose the desired return values within quotation marks in both the `F.when()` and the `.otherwise()` parameters. The underlying conditional logic remains completely unchanged, but the resulting column's data type will be automatically inferred as a string (or cast if necessary).

The revised code below implements this change, demonstrating how to achieve string-based flagging while preserving the identical **OR condition** logic:

import pyspark.sql.functions as F

```
#create new DataFrame
df_new = df.withColumn('B10', F.when((df.team=='B') | (df.points>10), 'Yes').otherwise('No'))

#view new DataFrame
df_new.show()
```

```
+----+--------+------+---+
|team|position|points|B10|
+----+--------+------+---+
| A| Guard| 11|Yes|
| A| Guard| 8| No|
| A| Forward| 22|Yes|
| A| Forward| 22|Yes|
| B| Guard| 14|Yes|
| B| Guard| 14|Yes|
| B| Forward| 13|Yes|
| B| Forward| 7|Yes|
+----+--------+------+---+
```

As evidenced by the output, the **B10** column now successfully returns 'Yes' or 'No'. This demonstrates the high versatility of the when function in adapting to various desired output data types. Developers must always ensure strict consistency between the return types supplied in both the `when` and `otherwise` clauses to maintain a unified data type for the resulting column.

Conclusion: Leveraging Advanced Conditional Transformations

The synergy between the PySpark when function and the bitwise **OR** operator (`|`) furnishes data professionals with an exceptionally powerful, scalable, and highly readable mechanism for defining intricate conditional logic across massive datasets. The crucial takeaways involve correctly utilizing parentheses to group individual boolean expressions, thereby ensuring proper operator precedence and reliable execution of the transformation. This capability is paramount for essential

tasks such as feature engineering, data labeling, and robust data preparation within the PySpark ecosystem.

Achieving proficiency in complex conditional logic is a foundational requirement for optimizing data manipulation workflows in any distributed computing environment. Developers are encouraged to experiment further by customizing return values--whether they are simple numerical indicators or more verbose descriptive strings--by precisely defining them within the `when` and `otherwise` components. Such flexibility ensures that the data output aligns perfectly with downstream reporting or modeling needs.

Additional Resources for PySpark Conditional Logic

The following resources offer further guidance on mastering conditional operations:

PySpark: How to Use When with AND Condition