Learning PySpark: Mastering Conditional Logic with the ‘when’ Function and AND Operators

Authored by

Mohammed looti

November 11, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learning PySpark: Mastering Conditional Logic with the ‘when’ Function and AND Operators*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=16747

The Necessity of Conditional Logic in PySpark Data Engineering

In the complex landscape of big data processing, the ability to apply **conditional logic** is not merely a feature--it is fundamental to effective data transformation. Data engineers routinely need to create new fields or derive metrics based on specific, often intricate, criteria applied across existing columns. For environments relying on PySpark, this requirement is met efficiently through the powerful **when function**. This function, typically accessed by importing `pyspark.sql.functions` as `F`, provides a clean, expressive method for implementing SQL-like `CASE` statements directly onto a distributed DataFrame. It is the primary tool for critical tasks such as data classification, calculating complex business rules, and generating binary flags for subsequent analysis.

As requirements grow more sophisticated, simple single-condition checks become insufficient. Data transformations frequently require the combination of multiple logical checks, meaning that several conditions must evaluate to true simultaneously before an action is taken. This necessitates the use of the AND condition. A common stumbling block for newcomers to PySpark is the difference between standard Python logic and DataFrame logic: to combine conditional expressions across columns in a Spark DataFrame, one must utilize the bitwise **AND operator** (`&`), rather than the keyword `and`. This distinction is vital for distributed execution.

The correct implementation of combined conditional expressions hinges entirely on proper structure and adherence to **operator precedence**. Every individual condition within the combined expression must be meticulously enclosed within parentheses. Forgetting this step is the most common source of runtime errors and unexpected results in PySpark conditional statements. The parentheses ensure that each comparison (e.g., `df.points > 10`) is evaluated completely as a boolean expression before the bitwise `&` operator attempts to combine the results across the massive dataset partitions managed by Spark's engine.

Mastering the `F.when()` Syntax with the Bitwise AND Operator

To successfully generate a conditional column based on a joint AND condition, data practitioners must integrate the **when function** with the bitwise `&` operator. The standard methodology involves using the withColumn method, which is employed to append a new column to the existing DataFrame while defining its contents based on the conditional logic established by `F.when()`. This sequence must always be terminated by the essential `.otherwise()` clause, which dictates the default value returned if the specified criteria are not met. Without `.otherwise()`, any rows failing the condition would result in a null value, which is often undesirable for analysis.

The primary structural difference between conditional logic in standard Python and PySpark DataFrames is the requirement for the bitwise **AND operator** (`&`). While Python uses the keyword

`and` for boolean operations on scalar values, PySpark conditions operate on entire columns (Series of boolean values), necessitating the bitwise operator for vectorized operations. Furthermore, the strict application of parentheses `()` around each individual comparison is non-negotiable. This ensures that the comparison operations are executed before the logical combination, preventing common operator precedence errors that could otherwise lead to incorrect filtering or schema misalignment.

The following syntax block provides a clear blueprint for implementing the **when function** when multiple criteria are linked by the logical **AND operator**. Notice the strategic placement of parentheses, which rigorously isolates each condition before the `&` operator combines them across the distributed data partitions:

import pyspark.sql.functions as F

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

In this specific example, a new column named **B10** is instantiated. Its value is contingent upon the simultaneous satisfaction of two distinct conditions: the `team` column must equal 'B', AND the `points` column must be greater than 10. If, and only if, both expressions evaluate to true, the output is 1. This methodology is incredibly efficient for processing large-scale datasets because it harnesses Spark's optimized execution engine, allowing the conditional logic to be applied in parallel across numerous data shards, providing massive performance benefits over row-by-row processing.

The resulting values in the generated **B10** column strictly adhere to a binary outcome based on the dual criteria:

A value of **1** is returned only when the value in the **team** column is 'B' and the corresponding value in the **points** column exceeds 10.
A value of **0** is returned in all other scenarios, fulfilling the requirement for a default output defined by the critical `.otherwise()` clause, thereby ensuring the column remains fully populated and consistent.

Practical Example Setup: Defining the PySpark DataFrame

To provide a clear, demonstrable understanding of how combined conditional statements operate, we must first establish a representative sample DataFrame. This dataset simulates common structured data, specifically basketball player statistics, encompassing team affiliation, position, and total points scored. This setup is highly analogous to real-world analytical tasks where business rules dictate categorization or flagging based on multiple performance metrics. The objective here is to isolate players who meet specific, high-performance criteria tied to a particular

organizational unit (Team B).

The initial step involves initializing the PySpark environment by creating a **Spark Session**, which acts as the entry point for all Spark functionality. Following the session creation, we define the raw data structure and the corresponding column names (schema). This meticulous definition ensures that when the DataFrame is instantiated, Spark correctly infers the data types and structure necessary for optimized execution. The resulting data structure is typically viewed using the `.show()` method to confirm the successful creation and structure of the distributed dataset before transformation begins.

The code block below details the necessary imports, the definition of the sample data (`data`), the definition of the schema (`columns`), and the final creation and display of the `df` DataFrame, ready for the application of conditional logic:

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
+----+--------+------+

The resulting DataFrame, `df`, now contains eight distinct records. Our precise analytical goal is to identify and flag every player who meets the dual criteria: they must be affiliated with **Team B** AND they must have scored **more than 10 points**. The successful execution of this dual requirement mandates the correct use of the bitwise **AND operator** (`&`) within the context of the when function, ensuring that the two conditions are evaluated as a single, combined boolean expression across all rows.

Implementing Numerical Flagging: The Binary Output Case

The transformation process begins by applying the conditional logic using the withColumn method to add our derived field. The core of the operation involves structuring the condition checks precisely: we require that both `df.team == 'B'` and `df.points > 10` evaluate to true for any given row. If this demanding standard is met, the new column, **B10**, is assigned the numerical flag value of **1**; otherwise, the value defaults to **0**. This use of binary output (1/0) is extremely valuable, particularly when the resulting column is destined for subsequent mathematical operations, such as aggregation (counting successful instances), statistical modeling, or serving as an engineered feature in a machine learning pipeline.

The use of the bitwise **AND operator** (`&`) enforces a strict intersection of the two criteria. It guarantees that a row will only receive the positive flag (1) if it satisfies both the categorical requirement (Team B) and the quantitative requirement (Points > 10). If a player belongs to Team B but scored 10 points or less, or if a player scored more than 10 points but belongs to Team A, the condition fails, and the default value (0) is applied. This rigorous filtering process showcases the power and efficiency of vectorized conditional logic in Spark.

Executing the following code allows us to observe firsthand how the combined conditions filter the DataFrame and assign the resulting numerical flag to the new column:

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| 0|
| A| Guard| 8| 0|
| A| Forward| 22| 0|
| A| Forward| 22| 0|
| B| Guard| 14| 1|
| B| Guard| 14| 1|
| B| Forward| 13| 1|
| B| Forward| 7| 0|
+----+--------+------+---+
```

A thorough review of the output confirms the effectiveness of the AND condition. Exactly three rows successfully met both criteria: they are associated with 'B', and their point totals (14, 14, and 13) are greater than 10. The final row for Team B, despite meeting the team criteria, failed the points threshold (7 points) and thus received a **0**. Crucially, every row associated with Team A failed the team criteria, guaranteeing they also receive a **0**. This outcome vividly illustrates the strict requirement imposed by the logical conjunction: all component conditions must be true for the overall expression to be true.

Flexibility in Output: Returning Categorical String Labels

While numerical flags (**1** and **0**) are ideal for mathematical analysis and machine learning workflows, there are many scenarios where the end product is intended for human consumption, such as analytical reports, operational dashboards, or business intelligence tools. In these cases, the semantic clarity offered by categorical string labels is often preferred over numerical representations. The **when function** is highly flexible and capable of returning any data type, including strings, provided that the return values specified in both the `when` and otherwise clauses are consistent, maintaining the integrity of the DataFrame schema.

We can effortlessly adapt the preceding example to generate string labels, returning 'Yes' when the combined criteria are satisfied and 'No' when they are not. This modification solely impacts the output format of the new column; the underlying, highly efficient conditional logic remains identical. Changing the output to strings enhances the immediate interpretability of the data, making it instantaneously clear which players successfully met the dual performance threshold without requiring external mapping.

```
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| No|
| A| Guard| 8| No|
| A| Forward| 22| No|
| A| Forward| 22| No|
| B| Guard| 14|Yes|
| B| Guard| 14|Yes|
| B| Forward| 13|Yes|
| B| Forward| 7| No|
+----+--------+------+---+
```

The resulting **B10** column now elegantly displays 'Yes' or 'No', providing immediate confirmation of the conditional outcome for each record based on the combined team and points criteria. This adaptability underscores the power of the **when function** in handling diverse data transformation needs. The ultimate decision between numerical and string output should always be driven by the specific downstream use case and the audience for the final DataFrame.

Advanced Considerations and Best Practices for Complex Conditionals

Effective utilization of the when function, especially when dealing with multiple logical operations, is a hallmark of skilled PySpark data engineering. The most critical lesson to internalize is the absolute necessity of using the bitwise **AND operator** (`&`) in place of the standard Python keyword `and`. This choice is mandatory because the operations are performed on Spark Column objects, not standard Python boolean scalars. Equally vital is the rigorous application of parentheses to properly isolate each conditional statement, which controls **operator precedence** and prevents ambiguous parsing by the Spark execution engine.

Beyond simple conjunctions, these structural principles extend seamlessly to other complex logical arrangements. When working with the logical **OR operator** (`|`), the same rules regarding bitwise operators and strict parentheses apply. Furthermore, PySpark allows for the powerful technique of

chaining multiple `F.when()` statements together. This chaining capability is used to sequentially test multiple conditions, effectively replicating the functionality of a comprehensive SQL `CASE WHEN... THEN... WHEN... THEN... ELSE... END` structure. When chaining, the first condition that evaluates to true dictates the result, and the process stops there.

Finally, a core best practice is maintaining data type consistency. Regardless of whether you are returning numbers, strings, or dates, always ensure that the data type returned by the initial `when` function call aligns perfectly with the data type returned by the subsequent chained `when` statements and, most importantly, with the final otherwise clause. Inconsistent return types will lead to schema errors or unexpected type casting, compromising the reliability and performance of your DataFrame transformations.

Additional Resources

The following tutorials explain how to perform other common and advanced data manipulation tasks in PySpark: