How to Add AND Conditions to PySpark When Functions

Authored by

stats writer

February 3, 2026

The PySpark ecosystem provides powerful tools for large-scale data processing. Central to manipulating data based on specific criteria is the conditional execution provided by the built-in functions. Specifically, the when function allows developers to apply SQL-like conditional logic directly within a DataFrame transformation pipeline.

While simple conditions are straightforward, real-world data tasks often require checking for the simultaneous truth of multiple criteria. To achieve this, the **when** function must be used in conjunction with a logical AND condition. Mastering this technique enables highly specific data filtering, feature engineering, and complex data categorization necessary for advanced analytics. This guide will meticulously detail the syntax and demonstrate practical examples of combining multiple conditions using the **when** function in PySpark.

PySpark: Using the `when` Function with Complex AND Logic

Introduction to Conditional Logic in DataFrames

Data manipulation frequently requires assigning new values or creating flags based on the existing content of rows. In relational databases, this is typically handled using the `CASE WHEN` structure. PySpark mirrors this functionality through the specialized functions available in the pyspark.sql.functions module, allowing for functional programming paradigms to be applied to structured data.

Implementing conditional logic efficiently is crucial when working with massive datasets, as using native Python logic (like standard `if/else` statements) can lead to highly inefficient operations due to serialization and distribution overhead. PySpark's native functions, such as **when** and **otherwise**, are optimized for the distributed nature of the Apache Spark engine, ensuring performance scalability. When creating new columns or updating existing ones based on complex rules, the **when** function is the preferred, idiomatic way to handle conditional assignments within a PySpark DataFrame.

Understanding the PySpark `when` Function

The core purpose of the when function is to evaluate a specified condition against every row in a DataFrame column. If the condition evaluates to true, a corresponding value is returned. If the condition is false, the function moves on to the next chained **when** clause (if any) or defaults to the value provided by the **otherwise** function. This structure allows for multi-layered conditional checks, similar to nested `if-elif-else` statements.

The basic syntax requires two primary arguments: the condition to evaluate and the resulting value if the condition is met. However, for practical use, it is almost always paired with the **otherwise** function to ensure that all rows are assigned a defined value, preventing nulls where logic is incomplete. This guarantees completeness in the resulting transformation.

Implementing Complex Logic: The AND Condition

When the assignment of a value depends on multiple criteria being simultaneously satisfied--for instance, a record must belong to a specific category **AND** exceed a certain threshold--we must employ the logical AND condition. Unlike standard Python where `and` is used for boolean logic, PySpark, operating within the context of column expressions, requires the bitwise AND operator, represented by the ampersand symbol: `&`.

It is imperative to wrap each individual condition within parentheses when using the `&` operator. This ensures that PySpark correctly groups the individual boolean comparisons before applying the logical intersection. Without proper parenthesization (e.g., `(condition1) & (condition2)`), the operator precedence rules might lead to unexpected or incorrect evaluation of the expressions, often resulting in errors or logically flawed output because it might try to apply the bitwise operator before evaluating the comparison operators (like == or >).

Essential Syntax for Combining Conditions

To effectively utilize the **when** function with an AND condition, we must import the necessary functions and apply the logic within a transformation function like withColumn. This method is used specifically to add a new column or replace an existing one based on the result of the conditional logic applied to the input columns.

The standard syntax below demonstrates how to create a new column, here named `B10`, by checking if the `team` column equals 'B' **and** if the `points` column is greater than 10. The output is a binary flag (1 or 0), a common practice in data preparation pipelines.

You can use the following syntax to use the **when** function in PySpark with AND conditions:

import pyspark.sql.functions as F

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

This particular example creates a new column named **B10** that returns the following values based on the complex criteria evaluated across the DataFrame:

1 if the **team** column is B *and* the **points** column is strictly greater than 10. Both conditions must

be true for the output to be 1.

0 otherwise. If one or both conditions are false, the default value of 0 is applied to the row in the new column.

Practical PySpark Example: Initial DataFrame Setup

To demonstrate this concept practically, we will first set up a simple DataFrame containing basketball player data, including team affiliation, position, and accrued points. This requires initializing a PySpark `SparkSession` and defining the schema (column names) for our dataset. The use of a simple, easily traceable dataset ensures that the conditional logic results are transparent and easy to verify against the source data.

The data structure is simple, enabling clear visualization of how the conditional logic flags rows that meet the combined criteria. Observe the steps below for creating and displaying the initial dataset, which serves as the foundation for all subsequent transformations:

Suppose we have the following PySpark DataFrame that contains information about various basketball players. We use the following code block to initialize the session, define the data, and display the starting DataFrame structure:

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

Case Study 1: Applying Numerical Flags (1 or 0)

Our first transformation goal is to identify players who belong to team 'B' **AND** have scored more than 10 points. We will use the **when** function nested within the withColumn method to add the indicator variable `B10` to our DataFrame. This approach is standard practice when feature engineering for machine learning models, where binary (0 or 1) flags are required to indicate the presence or absence of a specific characteristic.

We import `pyspark.sql.functions` as `F` to access the conditional functions efficiently. The combined condition `(df.team=='B') & (df.points>10)` is evaluated row-by-row. If both parts of the condition are logically true, the column `B10` receives a value of 1; otherwise, it receives 0.

We can use the following syntax to create a new column named **B10** that returns **1** if the **team** is B *and* the **points** is greater than 10, or **0** otherwise:

```
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
+----+--------+------+---+

Analyzing the Results of Numerical Flagging

Upon reviewing the resulting `df_new` DataFrame, we can clearly see which rows satisfied the combined condition. The output shows three rows where the **B10** column contains **1**. These three specific rows correspond exactly to players: (1) Team B, 14 points; (2) Team B, 14 points; and (3) Team B, 13 points. Only these rows simultaneously met the criteria of being on Team B and scoring above 10 points.

It is important to notice why the other rows from Team B were excluded. For example, the final row (Team B, 7 points) failed the second condition (points > 10), resulting in a 0. Similarly, all players from Team A failed the first condition (team == 'B'), regardless of their point total, thus also resulting in a 0. This demonstrates the strict requirement of the logical AND condition: all specified criteria must hold true simultaneously for the positive result (1) to be returned.

Case Study 2: Returning String Values (Yes or No)

While numerical flags (1/0) are useful for computational purposes, sometimes a dataset requires more human-readable outputs, such as strings like 'Yes' or 'No', particularly for final reporting or verification steps. The **when** and **otherwise** functions are highly flexible and can return any compatible data type--integers, floats, strings, or even complex nested column expressions.

To modify the previous example to return strings, we simply adjust the values passed to the **when** and **otherwise** functions, ensuring they are properly enclosed in quotes. The conditional logic itself remains identical because the evaluation criteria are based solely on the numerical and team columns, not the output type.

Also note that you could return 'Yes' or 'No' instead of **1** and **0** by using the following syntax:

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

Flexibility in Conditional Output Values

The resulting DataFrame now clearly shows 'Yes' or 'No' in the new **B10** column, confirming that the output type is independent of the input condition logic. This flexibility is a key feature of the when function, allowing data engineers to tailor the output format precisely to downstream requirements, whether those are numerical for modeling or categorical for reporting.

Feel free to return whatever values you'd like by specifying them in the **when** and **otherwise** functions. For instance, you could return a calculated metric, a value derived from another column, or even a null value (by using `lit(None)`) if necessary. The integrity of the conditional structure, particularly the correct use of the `&` operator for combined conditions and proper parenthesization, remains the most critical aspect of implementation.

Conclusion and Further Reading

Leveraging the **when** function with complex AND conditions is fundamental for advanced data manipulation within PySpark. By ensuring that individual conditions are correctly parenthesized and combined using the `&` operator, developers can precisely control data flows and create highly accurate derived fields necessary for business intelligence and data science applications. This technique forms the basis for complex feature engineering and ETL tasks in distributed environments.

The following tutorials explain how to perform other common tasks in PySpark: