Use a Case Statement in PySpark (With Example)

Authored by

stats writer

November 16, 2025

In data processing and analysis, the ability to apply conditional logic--where different outputs are generated based on specific data criteria--is fundamentally important. The Case Statement is a standard construct borrowed from SQL that enables this powerful conditional transformation.

In the context of distributed computing using PySpark, we accomplish the functionality of a **Case Statement** not through a single dedicated keyword, but through the highly flexible combination of the `when()` and `otherwise()` functions. This article provides a comprehensive guide on how to implement robust conditional logic to transform and categorize data within a **PySpark DataFrame**.

Understanding how to correctly chain these functions is crucial for any data engineer or analyst working with large datasets in the Apache Spark environment. This technique allows for the creation of new columns based on complex, multi-tiered rules applied to existing data fields.

Understanding Conditional Logic in PySpark

A **Case Statement** operates by sequentially checking a series of specified conditions. When the first condition evaluates to **true**, the corresponding value is returned, and the remaining conditions are ignored. If none of the defined conditions are met, a default value--if provided--is returned. This structure ensures exhaustive and orderly evaluation of business rules.

In PySpark, this mechanism is implemented using the `when()` function, which is imported from the `pyspark.sql.functions` module. The `when()` function takes two primary arguments: the condition to test, and the value to assign if that condition is met. For more complex logic, multiple `when()` clauses are simply chained together.

The final, indispensable component of this structure is the `otherwise()` method. If all preceding `when()` conditions evaluate to **false**, the value provided to `otherwise()` is assigned. Failing to include `otherwise()` may result in **null** values for records that do not satisfy any specified criteria, which is usually undesirable in production pipelines unless **null** is the intended default outcome.

The PySpark Mechanism: Utilizing when() and otherwise()

The standard practice for applying conditional logic in PySpark DataFrames involves using the withColumn method, which allows us to add a new column or overwrite an existing one based on a defined expression. This expression is where the chained `when()` statements reside.

The method is highly readable and mimics the structure of a traditional **Case Statement**. You begin the chain with the first `when()` call, which encapsulates the highest priority condition. Subsequent conditions are added using additional `.when()` calls, and the entire chain concludes with the mandatory `.otherwise()` call to handle all residual cases.

It is paramount to remember the order of evaluation: the conditions are checked sequentially. If you define a broad condition first (e.g., score < 100) and follow it with a narrower condition (e.g., score < 10), the narrower condition will never be reached for scores less than 10, as they will already satisfy the first condition. Therefore, structuring the conditions from the most restrictive (or specific boundary) to the least restrictive is often the safest approach, though the original example uses increasing upper boundaries, which is also a common pattern when defining exclusive ranges.

Basic Syntax for Implementing a Case Statement

The most straightforward implementation of a conditional categorization in PySpark requires importing the `when` function and then utilizing the DataFrame's withColumn transformation. Here is the canonical syntax used for adding a new column named `class` based on thresholds in the `points` column:

from pyspark.sql.functions import when

df.withColumn('class',when(df.points<9, 'Bad').when(df.points<12, 'OK').when(df.points<15, 'Good').otherwise('Great')).show()

This powerful single line of code performs a complete transformation, generating the `class` column based on the values in the `points` column. It sequentially tests the data against the defined thresholds. The resulting categories for the new column are defined as follows:

Bad: Assigned if the value in the `points` column is strictly less than 9.
OK: Assigned if the value in `points` is 9 or greater, but strictly less than 12 (i.e., falls within the range ,
,
,
,
,
,
,
,
,
]

#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

```
#view dataframe
df.show()

+------+------+
|player|points|
+------+------+
| A| 6|
| B| 8|
| C| 9|
| D| 9|
| E| 12|
| F| 14|
| G| 15|
| H| 17|
| I| 19|
| J| 22|
+------+------+
```

This resulting DataFrame, `df`, provides a perfect scenario for applying our conditional grading. We now have ten records where the `points` column contains integer values ranging from 6 to 22. Our objective is to use a **Case Statement** to assign a performance `class` to each player based on these scores.

Example Implementation: Categorizing Data Points

With the sample data prepared, we can now execute the core logic of the **Case Statement**. This involves using the withColumn function in conjunction with the chained when statements to define the classification rules. The goal is to create a new column named `class`, which will hold the categorized result.

We are applying four distinct categories based on the score thresholds: **Bad** (scores below 9), **OK** (scores 9 up to 12), **Good** (scores 12 up to 15), and **Great** (scores 15 and above). The structure of the conditional chain explicitly defines these boundaries and ensures that the evaluation is performed correctly from the lowest score threshold upwards.

Observe the execution below, which includes the necessary import of the `when` function and the subsequent DataFrame transformation, resulting in a DataFrame with the new classification column appended:

from pyspark.sql.functions import when

df.withColumn('class',when(df.points<9, 'Bad').when(df.points<12, 'OK').when(df.points<15, 'Good').otherwise('Great')).show()

```
+------+------+-----+
|player|points|class|
+------+------+-----+
| A| 6| Bad|
| B| 8| Bad|
| C| 9| OK|
| D| 9| OK|
| E| 12| Good|
| F| 14| Good|
| G| 15|Great|
| H| 17|Great|
| I| 19|Great|
| J| 22|Great|
+------+------+-----+
```

The resulting output clearly demonstrates how the **Case Statement** successfully transformed the numerical score data into categorical labels. For instance, Player A (6 points) falls into the first category, **Bad**, because 6 < 9 is the first condition to evaluate as true. Player E (12 points) is categorized as **Good** because the first two conditions (points < 9 and points < 12) are false, but 12 < 15 is true. Player G (15 points) fails the first three conditions, thus falling into the final `otherwise()` category, **Great**.

Analyzing the Resulting DataFrame

The execution of the conditional logic provides a powerful, categorized view of the underlying data. Let's meticulously examine how the logic flowed for different points values to ensure a deep understanding of the sequential evaluation inherent in the `when()` chain. This sequential checking is the defining characteristic of the **Case Statement**.

For any given row, the when function performed the following prioritized classification:

If `df.points < 9`: The result is **Bad**. (Applies to players A and B).
If the previous condition was false, check `df.points < 12`: The result is **OK**. This captures scores 9, 10, and 11. (Applies to players C and D).
If the previous two conditions were false, check `df.points < 15`: The result is **Good**. This captures scores 12, 13, and 14. (Applies to players E and F).
If all preceding conditions were false, apply the `otherwise()` clause: The result is **Great**. This

applies to all scores 15 and above. (Applies to players G, H, I, and J).

This example highlights the efficiency of using PySpark's SQL functions for conditional transformations. Since these operations are executed within the Spark engine, they are optimized for parallel processing across the cluster, making them extremely performant even when dealing with petabytes of data.

Advanced Considerations and Best Practices

While the basic example uses a single column for condition checks, real-world data science often requires more complex logic. PySpark's `when()` function is flexible enough to handle Boolean combinations of conditions, allowing you to use logical operators such as `&` (AND), `|` (OR), and `~` (NOT) within the condition argument.

For instance, to check if a player scores between 9 and 12 AND is named 'C', the condition would look like: `(df.points >= 9) & (df.points < 12) & (df.player == 'C')`. When combining multiple conditions, it is critical to wrap each individual condition in parentheses to ensure correct operator precedence.

A crucial best practice when using conditional logic on data sourced from various systems is handling missing data. If the `points` column contains **null** values, the comparison operations (e.g., `df.points < 9`) will often return **null**, which is treated as false in conditional logic checks. If you need to explicitly categorize **nulls**, you should add a specific check using `df.points.isNull()` as one of the first when conditions to categorize them before moving on to numerical comparisons.

Conclusion: Summarizing Conditional Transformations

The implementation of a Case Statement in PySpark through the chained use of `when()` and `otherwise()` functions is a cornerstone of effective data manipulation within the Spark ecosystem. This pattern provides a clean, highly scalable, and expressive way to perform complex conditional transformations on large-scale DataFrames.

By mastering the `when().otherwise()` chain, developers and analysts can categorize data, assign derived values, and implement intricate business logic directly within their PySpark jobs, ensuring that data cleaning and feature engineering steps are handled efficiently and reliably across distributed clusters. Remember that the ordering of the `when()` clauses dictates the priority of evaluation, a concept essential for designing accurate conditional logic.

We chose to use three conditions in this particular example, but you can chain together as many when statements as necessary to include even more fine-grained conditions in your own data transformation logic, always concluding with the default `otherwise()` condition to ensure

completeness.