# How to Convert a Boolean Column to Integer in PySpark

Authored by

**stats writer**

February 4, 2026

Data type conversion is a fundamental requirement when preparing data for analysis, especially in large-scale environments like PySpark. When dealing with logical states, it is often necessary to transform a Boolean column--which represents conditions as either true or false--into a numerical format, typically an integer. This conversion allows for direct computation, aggregation, and integration into machine learning models which often require numerical inputs.

In PySpark, this conversion can be achieved primarily through two robust methods: utilizing conditional logic via the powerful when() function, or leveraging the built-in type coercion capabilities of the cast() function. While both methods yield the desired result--mapping Boolean true/false values to 1/0, respectively--they offer different levels of control and readability depending on the complexity of the data transformation required.

Understanding these techniques is crucial for efficient data manipulation within a DataFrame. The preferred method often depends on context: the when() function provides explicit control over the mapping, while the cast() function offers a cleaner, more concise syntax for direct type conversion, assuming standard 1/0 mapping is acceptable.

# PySpark: Convert Column from Boolean to Integer

## The Primary Method: Using Conditional Logic with `when()`

The most commonly used and explicitly readable approach for converting a Boolean column to an integer in PySpark involves the use of conditional expressions. Specifically, we utilize the when() function combined with the `otherwise()` function, providing robust control over how Boolean values map to numerical results.

This approach is highly recommended when dealing with transformations where you want to clearly define that the value `True` corresponds exactly to the integer 1, and `False` corresponds to 0. By using the `withColumn` method on the DataFrame, we can either overwrite the existing Boolean column or, more commonly, create a brand new integer column that holds the transformed numerical data.

The following syntax demonstrates how to apply this conditional logic to map the Boolean states accurately:

**from pyspark.sql.functions import when**

#convert Boolean column to integer column
df_new = df.withColumn('int_column', when(df.bool_column==True, 1).otherwise(0))

## Understanding the `when()` Function Syntax

Analyzing the syntax above provides clarity on the transformation process. The `when(condition, value)` component checks if the specified condition is met; in this case, whether the value in the `bool_column` is `True`. If the condition is met, the new column, `int_column`, is assigned the value `1`.

If the condition (`df.bool_column == True`) is not met--meaning the value is `False` or possibly `null`, although Boolean columns typically contain only true/false--the subsequent `otherwise(value)` clause handles the assignment. In this standard conversion, anything that is not `True` (i.e., `False`) is explicitly mapped to the integer 0.

This particular example successfully transforms a Boolean column, here named `bool_column`, into an integer column named `int_column`. Every instance of `True` in the source column is represented as `1` in the destination column, and every instance of `False` is represented as `0`, establishing a clear numerical proxy for the logical data.

## Practical Example Setup: Creating the PySpark DataFrame

To illustrate the conditional conversion technique, let us establish a sample DataFrame. This DataFrame contains information about several basketball teams, including their name, score (`points`), and a Boolean column indicating whether the team made the playoffs (`playoffs`).

We begin by initializing the PySpark session and defining both the sample data and the corresponding column schema. This preparatory step is essential for working with structured data in a distributed environment.

**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|points|playoffs|
+-------+------+--------+
| Mavs| 18| true|
| Nets| 33| true|
| Lakers| 12| false|
| Kings| 15| true|
| Hawks| 19| false|
|Wizards| 24| false|
| Magic| 28| true|
| Jazz| 40| false|
|Thunder| 24| false|
| Spurs| 13| true|
+-------+------+--------+
```

In this newly created DataFrame, the `playoffs` column is clearly defined as a Boolean type, containing only the values `true` and `false`. Our objective is now to transform this column into a numerical representation suitable for quantitative analysis or modeling tasks, using 1 for successful playoff qualification and 0 otherwise.

## Implementing the Boolean to Integer Conversion

Using the principles of the when() function discussed earlier, we proceed to create a new column, `playoffs_int`, derived directly from the existing `playoffs` column. This transformation is achieved using the `withColumn` operation, which is standard practice in PySpark for adding or modifying columns.

The conditional expression explicitly states that if the `playoffs` column contains `True`, the new column will contain the integer value 1. For all other cases (i.e., `False`), the `otherwise(0)` statement ensures the new column is populated with 0. This guarantees a clean, unambiguous numerical representation of the binary logical state.

```
from pyspark.sql.functions import when

#convert Boolean column to integer column
df_new = df.withColumn('playoffs_int', when(df.playoffs==True, 1).otherwise(0))

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

```
+-------+------+--------+------------+
| team|points|playoffs|playoffs_int|
+-------+------+--------+------------+
| Mavs| 18| true| 1|
| Nets| 33| true| 1|
| Lakers| 12| false| 0|
| Kings| 15| true| 1|
| Hawks| 19| false| 0|
|Wizards| 24| false| 0|
| Magic| 28| true| 1|
| Jazz| 40| false| 0|
|Thunder| 24| false| 0|
| Spurs| 13| true| 1|
+-------+------+--------+------------+
```

As demonstrated in the resulting DataFrame, `df_new`, the new `playoffs_int` column now accurately displays the converted values. Teams that made the playoffs (original value `true`) are marked with `1`, and those that did not (original value `false`) are marked with `0`. This numerical conversion is often the required format for subsequent steps in data processing pipelines.

## Verification and Data Type Inspection

After performing any data transformation in PySpark, it is critical to verify the schema of the resulting DataFrame to ensure the new column has been correctly assigned the intended data type. If the conversion to an integer was successful, the schema should reflect `int` or `bigint` for the newly created column.

We can use the built-in `dtypes` attribute of the DataFrame to inspect the current data types of all columns. This provides an immediate and definitive confirmation of the schema integrity post-transformation, which is crucial for preventing unexpected errors in later stages of data processing.

```
#display data type of each column
df_new.dtypes
```

The output confirms that the new column, `playoffs_int`, is indeed registered as an `int` type, fulfilling the objective of converting the original Boolean data into a suitable numerical format using conditional logic.

## Alternative Approach: Leveraging the `cast()` Function

While the when() function provides explicit control, a simpler, more direct method exists for type conversion: the cast() function. The cast() function is designed to coerce one data type into another, provided the transformation is logically supported by PySpark's internal rules.

When applying cast() to convert a Boolean column to an integer type (`IntegerType()`), PySpark automatically handles the standard numerical mapping: `True` is mapped to 1, and `False` is mapped to 0. This method is concise and requires less code, making it an excellent choice when the default 1/0 mapping is desired.

The syntax for using cast() is straightforward, requiring only the specification of the target data type within the function call:

**from pyspark.sql.types import IntegerType**

df_cast = df.withColumn('playoffs_cast', df.playoffs.cast(IntegerType()))

df_cast.show()

```
+-------+------+--------+-------------+
| team|points|playoffs|playoffs_cast|
+-------+------+--------+-------------+
| Mavs| 18| true| 1|
| Nets| 33| true| 1|
| Lakers| 12| false| 0|
| Kings| 15| true| 1|
| Hawks| 19| false| 0|
|Wizards| 24| false| 0|
| Magic| 28| true| 1|
| Jazz| 40| false| 0|
|Thunder| 24| false| 0|
| Spurs| 13| true| 1|
+-------+------+--------+-------------+
```

## Why Integer Mapping is Crucial for Data Analysis

The necessity of converting Boolean columns to numerical representations like the integer 1 and 0 stems from the requirements of computational data analysis. While DataFrame operations can handle Boolean logic, many statistical functions, aggregation methods (such as calculating the mean or sum), and machine learning algorithms are optimized for numerical data.

For instance, if one wanted to calculate the percentage of teams that made the playoffs, summing the `playoffs_int` column and dividing by the total count immediately yields the desired ratio, a calculation impossible with the original Boolean type. Furthermore, in fields like predictive modeling, a feature must be numerical to be ingested by algorithms such as linear regression or neural networks.

Thus, the conversion of a binary logical state into its numerical proxy (1/0) serves as a critical step in feature engineering, bridging the gap between raw data interpretation and advanced analytical processing within the PySpark ecosystem. Both the when() and cast() methods provide reliable, efficient mechanisms for executing this essential transformation.