How to Convert a Boolean Column to Integer in PySpark: A Simple Guide

Authored by

stats writer

January 2, 2026

The Necessity of Type Conversion in PySpark

When working with large datasets and distributed computing environments like PySpark, efficient data handling is paramount. One common requirement in data preprocessing and feature engineering is converting column data types, especially transitioning from boolean representations to their numerical integer equivalents. While PySpark offers inherent support for boolean types, certain downstream analytical tasks, such as model training or specific SQL aggregations, often require numerical input (0s and 1s) rather than logical True/False values.

The conversion process is straightforward in PySpark and ensures that your DataFrame maintains structure while facilitating computational needs. The fundamental principle is mapping True to 1 and False to 0. Understanding the different methods available allows data practitioners to choose the most robust and readable approach for their specific coding environment.

While the simple `.cast()` method provides the quickest route for this transformation, using conditional functions like `when()` offers greater control over how edge cases, such as null values, are handled during the conversion process. This guide will explore both techniques, focusing primarily on the highly explicit and widely used conditional approach, which enhances code clarity and maintainability for complex data pipelines built using PySpark.

Advanced Conversion using when() and Conditional Logic

Although the `.cast('integer')` method is concise, data professionals often prefer the `when()` function, imported from `pyspark.sql.functions`, for explicit Boolean-to-Integer mapping. This technique is particularly valuable when you need precise control over the mapping, or when the column might contain non-standard values or requires specific handling

for `NULL` entries. The `when()` function evaluates a series of conditions and returns a result when a condition is met, otherwise defaulting to the `otherwise()` clause.

To implement this conversion, we utilize the `withColumn()` method on the PySpark DataFrame to create a new column or overwrite an existing one. Inside `withColumn()`, the conditional logic checks if the value in the source boolean column is True. If it is, it assigns the integer value 1; otherwise (if False or Null), it assigns 0. This explicit approach prevents ambiguity and ensures the resulting column is fully compliant with numerical analysis requirements.

The standard syntax for converting a boolean column using this conditional approach in PySpark is demonstrated below.
We are mapping a source column, `bool_column`, to a new target column, `int_column`, ensuring a rigorous transformation process based on SQL-like conditional logic provided by the `when()` function.

The following PySpark syntax illustrates how to use the `when()` function to explicitly convert a boolean column into an integer column:

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))

This specific implementation converts the source boolean column named `bool_column` into a numerical column called `int_column`. Note that the `when()` function is central to defining the transformation rule.

Consequently, every value originally stored as `True` in the source column is accurately mapped and displayed as the numerical `1` in the resulting integer column.

Similarly, all values equal to `False` in the original boolean column are reliably transformed and represented as the integer value `0` in the newly created column.

To further solidify this concept, the following sections provide a complete, executable example demonstrating this syntax in a practical PySpark environment.

Prerequisites: Setting Up the PySpark Environment

Before executing the conversion logic, it is essential to initialize the PySpark environment by creating a `SparkSession`. The `SparkSession` is the entry point to nearly all PySpark functionality, allowing us to define and manipulate DataFrames and execute distributed computing tasks. For this demonstration, we will also need standard imports for creating the initial data structure.

We will simulate a dataset containing information about various basketball teams. This sample DataFrame includes three columns: `team` (string), `points` (numerical), and `playoffs` (a boolean column indicating qualification status). The goal is to convert the `playoffs` column from its logical representation (True/False) to its numerical counterpart (1/0).

The following code block sets up the environment, defines the sample data, specifies the column schemas, and creates the initial PySpark DataFrame. This provides a clear, verifiable starting point for our boolean-to-integer conversion tutorial.

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

Applying the when() Function for Conversion

With the initial DataFrame, `df`, successfully created, we can now proceed with the core transformation. The column targeted for conversion, `playoffs`, is currently of the boolean data type. Our objective is to derive a new column, `playoffs_int`, where successful qualification (`True`) is numerically encoded as `1`, and non-qualification (`False`) is encoded as `0`.

We must first import the `when` function from `pyspark.sql.functions` if it hasn't been imported already. We then chain the `withColumn` method, specifying the name of the new column, and define the conditional logic. The syntax `when(df.playoffs == True, 1)` checks the condition, while `.otherwise(0)` serves as the fallback for all other rows, which in this case includes `False` values and any potential `NULL` values that might be present.

The code below executes this transformation, generating the `playoffs_int` column and displaying the modified DataFrame to visually confirm the success of the conversion. This method ensures that the original data integrity is maintained, as the original `playoffs` column remains unchanged,

allowing for easy comparison between the boolean and integer formats.

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

Verification of Data Types

After performing any critical data transformation, it is standard practice to verify the resulting schema to ensure that the new column has been correctly assigned the desired data type. In this context,
we need to confirm that `playoffs_int` is truly an
integer
column, as intended for subsequent numerical analysis. Failure to verify the data type could lead to runtime errors or incorrect results in downstream processes, such as machine learning training pipelines that strictly require numerical input.

PySpark
provides the `.dtypes` attribute, which returns a list of tuples detailing the name
and data type for every column in the
DataFrame.

By applying this function to our newly created DataFrame, `df_new`, we can inspect the schema and confirm the successful type conversion.

As demonstrated by the output below, the `playoffs_int` column is successfully identified as `'int'` (integer). Furthermore, we can observe that the `playoffs` column retained its original `'boolean'` type, validating that the `withColumn` operation created a distinct, correctly typed feature based on the conditional mapping logic we defined using `when()`.

#display data type of each column
df_new.dtypes

The schema validation confirms that `playoffs_int` is indeed an integer column, allowing it to be used reliably in numerical computations and SQL queries requiring numerical rather than boolean fields.

Alternative: Handling Nulls and Edge Cases

While the `when(condition, 1).otherwise(0)` pattern is highly effective, it inherently treats `NULL` values in the input boolean column as `False` (since the initial condition `df.col == True` fails for NULLs, leading them to the `otherwise(0)` branch). In some analytical contexts, `NULL` values should be explicitly handled--perhaps by assigning a unique indicator (like `-1`) or by preserving the `NULL` value in the new integer column.

If the requirement is to preserve `NULL` values as `NULL` in the resulting integer column, a slightly modified conditional structure must be employed. This involves checking for non-null status before applying the boolean mapping. Alternatively, if we wanted to map `NULL` to a third category, say `2` (representing unknown status), we would chain an additional `when()` clause using `pyspark.sql.functions.isnull`.

For most standard numerical conversion tasks aiming for 0/1 binary encoding, the simplified `when().otherwise()` method is sufficient and elegant. However, when working with dirty or sparse data, being aware of how the conditional statements implicitly handle edge cases like `NULL`s is crucial for robust data engineering pipelines.

Conclusion: Summary and Best Practices

Converting a column from a boolean data type to an integer data type is a frequent operation in data preparation using PySpark. This transformation is vital for ensuring compatibility with numerical models and performing efficient numerical computations and aggregation queries using Spark SQL. We have demonstrated two primary methods: the straightforward `.cast()` approach and the more controlled `when().otherwise()` conditional mapping.

The `when()` function, while slightly more verbose, offers unparalleled clarity and robustness, especially when dealing with complex scenarios or when explicit control over the `True`-to-`1` and `False`-to-`0` mapping is required. Best practice dictates using the most explicit method that clearly communicates the intent of the transformation, minimizing future maintenance overhead.

Always remember to verify the schema using `.dtypes` after any critical transformation to ensure that the intended data type has been correctly assigned. Mastering these fundamental data type conversions is key to successfully manipulating and preparing large datasets within the distributed computing framework of PySpark.