# Learn How to Add a Column with a Constant Value in PySpark DataFrames

Authored by

**Mohammed looti**

November 11, 2025

## Introduction to Adding Constant Columns in PySpark

When executing large-scale data transformation and enrichment tasks using PySpark, data engineers frequently encounter the requirement to inject a new column into an existing PySpark DataFrame where every single row must hold an identical, predefined value. This constant insertion is crucial for several standard data processing needs, such as providing standardized labeling, setting up global parameters for downstream processes, or flagging datasets for specific analytical pipelines. This comprehensive guide will walk through the most efficient and idiomatic methods available for achieving this objective, leveraging the specialized functions provided within the `pyspark.sql.functions` module.

The mastery of flexible DataFrame manipulation is a foundational skill for effective data engineering within the Spark ecosystem. Our specific focus here is on how to efficiently broadcast a fixed, literal constant value across all records of a newly created column. This technique is remarkably versatile and applies universally, regardless of the constant's underlying data type--whether you are dealing with numerical identifiers, dates, or complex string values. Understanding this mechanism ensures that your data pipelines are robust and scalable.

The core process hinges on the strategic combination of two primary DataFrame transformations. The subsequent sections will systematically introduce these essential functions, guide you through setting up a practical working environment, and provide step-by-step examples demonstrating how to seamlessly insert both numeric and string constant literals into a target DataFrame.

## Essential Functions: `withColumn` and `lit`

The implementation of constant column addition in PySpark is highly streamlined through the combined use of two specific utility functions. To ensure accurate and high-performance data processing in a distributed environment, it is imperative to fully grasp the distinct and complementary role each of these functions plays within the transformation pipeline.

The primary mechanism for altering the schema or content of a DataFrame is the powerful `withColumn` transformation. This function is invoked directly on an existing PySpark DataFrame instance and serves two main purposes: defining a brand new column based on an expression, or updating the contents of an already existing column. It is vital to remember the principle of Spark DataFrames being **immutable**; consequently, `withColumn` always returns a completely new DataFrame containing the desired structural changes, thereby maintaining the integrity and safety of the original data structure.

The second, equally indispensable function is `lit`, a shorthand for "literal." This function must be explicitly imported from `pyspark.sql.functions`. The role of `lit` is to wrap a standard Python scalar value--such as the integer `100` or the string `'NBA'`--and convert it into a **column expression**

that the Spark execution engine can efficiently interpret and apply consistently across every single row and partition. Without this crucial `lit` wrapper, the `withColumn` function would fail to recognize the input as a columnar entity suitable for distributed processing, often resulting in runtime errors or unpredictable results. Therefore, the pattern of combining `withColumn` (to define the new column name) and `lit` (to supply the constant expression) constitutes the canonical and recommended method for this specific data task.

**Method 1: Adding a Constant Numeric Value**

When the objective is to introduce a column populated entirely by a fixed numerical value, such as an integer or a decimal, the syntax is straightforward. We must first import the `lit` function and then pass the desired numeric constant directly to it. The following code snippet demonstrates the necessary import and the structure for adding a column named `salary` with a constant value of 100:

**from pyspark.sql.functions import lit**

# Add new column called 'salary' with constant value of 100 for each row
df.withColumn('salary', lit(100)).show()

**Method 2: Adding a Constant String Value**

Similarly, when working with constant textual data--such as category labels, identifiers, or status codes--the same methodology applies. The critical distinction is ensuring that the constant value passed to the `lit` function is enclosed in quotes, signifying that it is a string literal. This is useful for standardizing labels across a dataset that may be derived from multiple sources.

The syntax below illustrates the process of adding a new column named `league`, populated uniformly with the string value `'NBA'` for all records:

**from pyspark.sql.functions import lit**

# Add new column called 'league' with constant value of 'NBA' for each row
df.withColumn('league', lit('NBA')).show()

## Setting Up the Demonstration PySpark DataFrame

To fully visualize and confirm the practical execution of these transformation techniques, we must first establish a small, foundational PySpark DataFrame. This initial dataset will serve as the necessary baseline target for all subsequent constant column insertions. Our initial setup requires us to initialize the Spark environment, typically accomplished using

`SparkSession.builder.getOrCreate()`. We then define our raw data as a Python list of rows, alongside a corresponding list defining the column schema. This procedure represents the standard, robust practice for creating small, in-memory DataFrames essential for demonstration, unit testing, or rapid prototyping purposes.

The sample data we utilize tracks basic performance statistics for several fictional teams, including their designated conference, total points scored, and assists recorded. This structure provides clear, verifiable existing fields against which we can easily assess the outcome of our constant value transformations. Once the DataFrame, which we name `df`, is successfully created, we immediately employ `df.show()`. This command is critical as it triggers the execution of the Spark plan and displays the DataFrame's contents, allowing us to verify its initial structure and data integrity before any modifications are applied.

The following code block executes the entire environment setup, defines the data and schema, creates the DataFrame, and displays its initial state:

**from pyspark.sql import SparkSession**
**spark = SparkSession.builder.getOrCreate()**

# Define the raw input data rows
data = ,
,
,
,
,
]

# Define the corresponding column names (schema)
columns =

# Create the PySpark DataFrame from the defined data and columns
df = spark.createDataFrame(data, columns)

# Display the initial DataFrame structure
df.show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|

| B| West| 6| 12|
| B| West| 6| 4|
| C| East| 5| 2|
+----+----------+------+-------+

## Practical Application 1: Inserting a Constant Numeric Value

Utilizing the operational DataFrame `df` established in the preceding section, we can now proceed to demonstrate the straightforward addition of a constant numeric column. For this illustration, we select the column name `salary` and uniformly assign it the integer value **100** across every existing row. This critical action is executed by seamlessly chaining the `withColumn` transformation with the expression generated by `lit(100)`. The superior efficiency of this method stems directly from Spark's highly optimized ability to broadcast and insert the literal value across all nodes and partitions within the cluster, effectively sidestepping the need for inefficient, row-by-row iteration typical of non-distributed processing frameworks.

The resulting code snippet below includes the necessary function import and the precise transformation action. Note that the invocation of `.show()` is what forces the immediate, lazy execution of the underlying Spark plan, resulting in the display of the new DataFrame structure with the appended column.

**from pyspark.sql.functions import lit**

# Add new column called 'salary' with constant value of 100 for each row
df.withColumn('salary', lit(100)).show()

+----+----------+------+-------+------+
|team|conference|points|assists|salary|
+----+----------+------+-------+------+
| A| East| 11| 4| 100|
| A| East| 8| 9| 100|
| A| East| 10| 3| 100|
| B| West| 6| 12| 100|
| B| West| 6| 4| 100|
| C| East| 5| 2| 100|
+----+----------+------+-------+------+

A careful review of the output clearly confirms that the new column named `salary` has been successfully appended to the DataFrame schema. Critically, every single row within this new column contains the specified constant value of **100**. This result validates that the essential

combination of `withColumn` and `lit` functions operates precisely as intended for the insertion of constant numeric data types.

## Practical Application 2: Inserting a Constant String Value

Moving beyond numerical data, our second practical example focuses on the addition of a constant string value. We will introduce a column named `league` and populate it uniformly across all rows with the string literal `'NBA'`. This specific technique is exceptionally valuable for crucial tasks such as metadata tagging, standardizing category labels, or providing essential context when consolidating and merging multiple distinct datasets that share a common origin or classification system.

The methodology remains identical to the numeric example: we must import `lit` and then pass the desired string constant, ensuring it is correctly enclosed in quotation marks, directly to the function. This expression then forms the second, critical argument within the `withColumn` call. The resulting code block below succinctly demonstrates this string manipulation process and the subsequent output:

**from pyspark.sql.functions import lit**

# Add new column called 'league' with constant value of 'NBA' for each row
df.withColumn('league', lit('NBA')).show()

+----+----------+------+-------+------+
|team|conference|points|assists|league|
+----+----------+------+-------+------+
| A| East| 11| 4| NBA|
| A| East| 8| 9| NBA|
| A| East| 10| 3| NBA|
| B| West| 6| 12| NBA|
| B| West| 6| 4| NBA|
| C| East| 5| 2| NBA|
+----+----------+------+-------+------+

Following the transformation, the `league` column is successfully added to the DataFrame. Every row now correctly and uniformly displays the string value **NBA**, confirming the robust implementation for handling string constants. This clearly demonstrates the seamless capability of PySpark to consistently manage different underlying data types through the powerful use of the `lit` function.

## Summary of Best Practices and Key Concepts

The accepted standard practice for efficiently creating constant columns within PySpark revolves entirely around the foundational functions detailed throughout this guide: `withColumn` and `lit`. Adopting and consistently applying this specific pattern ensures that your data processing code remains highly readable, easily maintainable, and, most importantly, maximally optimized by Spark's sophisticated underlying execution engine.

It is absolutely paramount for developers to internalize the core concept of Spark DataFrame **immutability**. The `withColumn` function does not, and cannot, modify the DataFrame in place. Instead, its behavior is to return an entirely new instance of the PySpark DataFrame with the desired column modification successfully applied. Consequently, if a developer fails to capture the output of `withColumn` by assigning it back to a variable (e.g., `df = df.withColumn(...)`), the costly transformation operation will still be executed lazily by Spark, but the valuable results will be immediately discarded, leading to unexpected data losses or incorrect downstream processing.

Furthermore, the `lit` function performs a vital and specialized type-casting role. Its primary purpose is to elevate a native Python scalar value into a column object, a conversion that is critically necessary because `withColumn` strictly requires a column expression as its second argument. This essential mechanism ensures that the constant value is correctly interpreted, serialized, and efficiently distributed across all partitions of the cluster. Mastering this fundamental transformation technique is truly indispensable for tackling more sophisticated data manipulation challenges, such as implementing complex conditional logic or effectively utilizing advanced window functions.

## Expanding Your PySpark Data Engineering Toolkit

Building upon the foundational skills of adding simple constant columns is the logical and necessary next step for advancing your data processing capabilities within the PySpark environment. These straightforward transformations frequently serve as the initial building blocks for constructing much more sophisticated SQL-like data operations.

To further expand your knowledge and operational scope regarding DataFrame manipulation, consider exploring the following key areas, which naturally follow the concepts introduced here:

Deeper exploration of methods for creating columns conditionally using the powerful `pyspark.sql.functions.when` function, often coupled with the `otherwise` clause for defining fallback values.

Gaining a comprehensive understanding of core Spark DataFrame operations specifically designed for data reduction and selection, such as filtering data rows using `.filter()` and

selecting or renaming specific columns using `.select()`.

Dedicated study aimed at deepening knowledge of schema definition, ensuring correct data types upon ingestion, and executing necessary data type conversions using functions like `.cast()`.