PySpark: Add New Column with Constant Value

Authored by

stats writer

November 16, 2025

Introduction: Mastering Constant Column Addition in PySpark

Welcome to this detailed guide on a fundamental data manipulation technique within the **PySpark** framework: adding a new column populated entirely by a **constant value**. This operation is indispensable in various data engineering workflows, often required for standardization, flagging records, assigning default parameters, or preparing data for subsequent joins and transformations. Efficiently appending a column with a uniform value ensures consistency across the entire **DataFrame**, regardless of the existing data size or structure.

Understanding how to perform this task cleanly and efficiently is essential for anyone working with big data processing using **PySpark**. The primary methodology relies on two key components provided by the `pyspark.sql` module: the `withColumn` transformation and the `lit` (literal) function. These tools work in tandem to guarantee that the operation is executed in a distributed, optimized manner characteristic of the Apache Spark engine.

This article will provide an in-depth exploration of the required syntax, demonstrate practical examples using both numeric and string constants, and offer a thorough explanation of the underlying functions. By the end of this tutorial, you will possess the knowledge to seamlessly integrate constant columns into your **PySpark DataFrame** manipulations, enhancing your data preparation pipeline.

The Crucial Role of `withColumn` and `lit()` Functions

When working with **PySpark**, modifications to a **DataFrame** are typically achieved using transformation functions. To add or modify a column, the standard method is utilizing the **`withColumn`** function. This function is immutable; it does not alter the original DataFrame but instead returns a new DataFrame instance incorporating the specified change. The function requires two main arguments: the name of the new or replacement column, and the expression defining the new column's values.

However, directly passing a simple Python primitive (like `100` or `'NBA'`) as the column expression is generally insufficient in the Spark environment, which relies on optimized Column objects for vectorized operations. This is where the **`lit`** function becomes essential. The **`lit`** function, short for "literal," is designed specifically to convert a standard Python value into a Spark Column object containing that **constant value** repeated for every row in the DataFrame.

The synergy between **`withColumn`** and **`lit`** creates a concise and highly efficient mechanism for injecting a uniform value across the dataset. Whether the constant is a number, a string, a date, or a boolean, the `lit` function handles the conversion seamlessly, ensuring that the transformation is executed correctly within the distributed context of **PySpark**.

Method 1: Implementation Using Constant Numeric Values

When the objective is to assign a fixed numeric identifier, a default score, or a placeholder number (such as 0 or 100) to every record, the combination of `withColumn` and `lit` is employed using an integer or floating-point argument within the `lit` function. This technique is often used in scenarios where data needs normalization or when creating binary flags based on external conditions not yet present in the dataset.

The syntax is straightforward. First, ensure `lit` is imported from `pyspark.sql.functions`. Then, chain the **`withColumn`** method onto your existing DataFrame, specifying the new column name (e.g., 'salary') and wrapping the desired numeric **constant value** (e.g., 100) within the **`lit`** function.

The following code snippet illustrates the basic implementation for adding a numeric constant column:

from pyspark.sql.functions import lit

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

Method 2: Implementation Using Constant String Values

Similar to numeric values, appending a constant string is equally necessary for data preparation. Use cases include appending a standardized source label, a category identifier, or a regional tag (e.g., 'US_Region', 'Q4_2023', or 'Pending') to all rows. When dealing with strings, the **`lit`** function handles the quoting internally, ensuring the output column is of string type (or suitable equivalent).

The structure mirrors the numeric example, but the argument passed to `lit` must be enclosed in quotes, designating it as a string literal. This maintains clarity and ensures the resulting column is correctly typed as `StringType` within the Spark schema. This method is particularly useful when standardizing metadata across disparate datasets consolidated into a single **DataFrame**.

Here is the code demonstrating how to add a constant string value:

from pyspark.sql.functions import lit

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

Setting Up the PySpark Environment and Sample Data

Before executing the examples above, we must establish a working Spark Session and define the base **DataFrame** that we will be manipulating. This standardized preparation ensures that all subsequent code blocks are runnable and verifiable. We utilize the `SparkSession.builder` pattern, which is the foundational entry point for using Spark functionality through **PySpark**.

Our sample data represents simple sports statistics, containing categorical (team, conference) and numeric (points, assists) columns. This data setup is intentionally simple, allowing us to focus solely on the column addition mechanism without being distracted by complex data types or existing derived fields. Once the data and column names are defined, we use `spark.createDataFrame` to materialize the distributed dataset.

The initial structure of the DataFrame, before any modifications, serves as our baseline. Note how the data is organized into rows and columns, which will be extended by the subsequent constant columns.

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|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
+----+----------+------+-------+

Example 1: Adding a Constant Numeric Value Column

In this first comprehensive example, we apply Method 1 to standardize a hypothetical salary value across all teams in our dataset. Perhaps this **constant value** represents a minimum bonus applied universally, or a base salary prior to performance adjustments. We will name this new column `salary` and assign the **constant value** of `100` to every row. This operation confirms the data type conversion handled implicitly by the `lit` function, typically resulting in an integer or long type column depending on the magnitude of the literal input.

We use the **`withColumn`** function, passing `'salary'` as the column name and `lit(100)` as the column expression. The result is a new DataFrame where the existing structure remains intact, but an additional column is appended to the right side of the schema.

Review the syntax and the resulting output below, paying close attention to the inclusion and placement of the new column, `salary`.

from pyspark.sql.functions import lit

#add new column called 'salary' with 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
+----+----------+------+-------+------+

Analyzing the Numeric Output

The output clearly demonstrates the success of the transformation. A new column, `salary`, has been successfully integrated into the **DataFrame** schema. Crucially, every single row, regardless of the unique values in the `team`, `conference`, `points`, or `assists` columns, now possesses the identical value of `100` in the `salary` column. This uniformity highlights the definition of a **constant value** column.

It is important to reiterate that the **`withColumn`** function, as a transformation, generates a new DataFrame. If you were to subsequently use the original variable `df` without assigning the result of the transformation back to it (e.g., `df = df.withColumn(...)`), the original `df` would remain unmodified, adhering to Spark's principle of immutability.

This operation is foundational for subsequent filtering or grouping tasks. For instance, if you later wanted to filter records based on this base salary, the column is ready for use without any further calculations required. This efficiency is a primary benefit of inserting constants directly into the data structure.

Example 2: Adding a Constant String Value Column

Next, we apply Method 2 to add a constant string identifier to our DataFrame. We will assume all these records belong to a major athletic organization, such as the National Basketball Association (NBA). We define a new column called `league` and populate it with the string constant `'NBA'` for every single row. This is typical when combining data from multiple sources where standardization of organizational identifiers is mandatory.

Similar to the numeric example, we invoke **`withColumn`**, specifying `'league'` as the new column name. The string literal `'NBA'` is passed to the **`lit`** function. The `lit` function correctly interprets the quoted value and ensures the resulting column is of the appropriate `StringType`.

Observe the executed code and the resultant DataFrame structure, confirming the addition of the `league` column.

from pyspark.sql.functions import lit

#add new column called 'league' with 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|
+----+----------+------+-------+------+
```

Analyzing the String Output and Key Takeaways

The final output confirms that the new column, `league`, has been successfully added to the DataFrame, positioned as the last column. Every single record in this resulting DataFrame now carries the **constant value** `NBA` in this column. This demonstrates the versatility of the `lit` function, which can handle various data types--including strings, integers, floats, and booleans--as constants.

Understanding the specific roles of the functions used is paramount for advanced **PySpark** development:

Note #1: Function Immutability (`withColumn`): The `withColumn` function is a transformation that always returns a new DataFrame. It does not modify the DataFrame it is called upon. This adherence to immutability is fundamental to how Spark manages distributed computations and lineage tracking. The new DataFrame contains the specified column modified or added, while all other columns remain unchanged from the original.

Note #2: The Literal Column Creator (`lit`): The `lit` function is specifically designed to create a Spark Column object from a standard Python primitive. This step is necessary because Spark operations require Column expressions, not raw Python values, to ensure the transformation can be efficiently distributed and optimized across the cluster nodes. When you use `lit(X)`, Spark broadcasts the value `X` and ensures that it is replicated across all partitions as the value for the new column.

By combining these two powerful functions, data engineers can reliably and efficiently introduce standardized metadata or default values into massive datasets, streamlining complex data integration and processing tasks within the **PySpark** environment.