How to Add a Constant Column to a PySpark DataFrame

Authored by

stats writer

February 9, 2026

PySpark: Add New Column with Constant Value

Data manipulation is a fundamental task in large-scale data processing, and when working with big data, PySpark provides robust and scalable tools for transforming datasets. A common requirement in data preparation is adding a new column to a DataFrame where every row contains an identical, fixed value. This constant value might represent a category identifier, a creation timestamp for batch processing, or a default status flag. The most efficient and idiomatic way to achieve this in PySpark involves the synergistic use of the `withColumn()` transformation combined with the `lit()` function.

The `withColumn()` method is central to PySpark's data modification capabilities. It enables users to return a new DataFrame by either adding a new column or replacing an existing one, based on the specified column expression. Since we need the column value to be static across all partitions and rows, we employ the `lit()` function, short for literal, which ensures the input value is treated as a constant expression, guaranteeing uniform assignment across the entire distributed dataset.

This article provides an in-depth guide on utilizing these functions to append columns containing both numeric and string constants, illustrating the methods with practical examples and clear code demonstrations. Understanding this technique is essential for effective data standardization and feature engineering within the **Apache Spark** ecosystem.

The Core Mechanism: Understanding `withColumn()` and `lit()`

To successfully implement the addition of a constant column, it is vital to grasp the roles of the two primary functions involved. The `withColumn()` method, defined on the DataFrame object, accepts two mandatory arguments: the name of the new column (as a string) and the column expression that defines its values. Since PySpark operations are based on expressions and transformations, simply passing a Python constant (like `100` or `'NBA'`) directly into `withColumn()` is insufficient; Spark expects a **Column object** or expression.

This is where the `lit()` function, imported from `pyspark.sql.functions`, becomes indispensable. The purpose of `lit()` function is to take a standard Python value (be it an integer, float, string, or boolean) and wrap it into a Spark Column expression. This conversion is necessary because Spark's engine operates on symbolic execution graphs, not direct Python values, ensuring that the constant value can be efficiently broadcasted and applied across all worker nodes in the cluster without requiring complex User Defined Functions (UDFs).

Therefore, the syntax always follows the pattern: `df.withColumn(column_name, lit(constant_value))`. This combination creates a new column named `column_name`, where every single record inherits the exact value specified inside the `lit()` function. This ensures data

consistency and optimal performance, which are hallmarks of high-quality PySpark code.

Method 1: Adding Constant Numeric Values

Adding a constant numeric value, such as an integer or a floating-point number, is often required when standardizing metric baselines or assigning default numerical identifiers. The process is straightforward: import `lit`, specify the new column name, and pass the desired number to the `lit()` function within the `withColumn()` call.

When adding numeric constants, PySpark automatically infers the data type of the new column based on the input provided to `lit()`. If you pass an integer like `100`, the resulting column schema will likely be `IntegerType`. If you pass a decimal number like `100.50`, it will be inferred as `DoubleType` or `FloatType`, depending on precision requirements. This automatic type inference simplifies the coding process significantly, preventing manual schema definition in most common scenarios.

For instance, if we intend to assign a default salary value of **100** to all personnel records currently stored in our DataFrame, the implementation is concise and immediately expressive of the desired transformation. This is particularly useful in **ETL pipelines** where initial records must be tagged with a placeholder metric before actual calculations or joins occur.

The core syntax for this method is demonstrated below, focusing on assigning an integer constant:

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: Adding Constant String Values

In many analytical scenarios, the constant value required is categorical or descriptive text, such as labeling a batch source, defining a geographical region, or setting a default status. String constants require the same approach as numeric constants, ensuring that the string literal is properly enclosed in quotation marks when passed to the `lit()` function.

When dealing with string constants, the resulting column data type will be `StringType`. It is important to remember that PySpark handles strings efficiently, and using `lit()` ensures that the string content is not serialized multiple times but rather optimized across the cluster operations. This is far more performant than attempting to use complex lambda functions or UDFs for simple constant assignment, maintaining high distributed processing speeds.

Consider a scenario where all rows in a dataset originate from a specific sporting league, such as **'NBA'**. By adding this constant string value, downstream operations can easily filter or group records based on this fixed attribute, enhancing data organization and clarity. This standardization step is foundational for subsequent analysis and reporting tasks, providing a consistent categorical variable across the entire dataset.

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 demonstrating the practical application of these methods, we must establish a working PySpark DataFrame and initialize the necessary environment components. The foundation of any Spark application is the SparkSession, which serves as the entry point for interacting with underlying Spark functionality. We use the standard builder pattern (`SparkSession.builder.getOrCreate()`) to create or retrieve an existing session, ensuring that our environment is correctly configured for distributed processing.

For our demonstration, we define a small, representative dataset containing basic sports statistics. This dataset consists of team identifiers, conference affiliations, points, and assists. Defining the data structure clearly, along with the corresponding column names, allows for a robust and reproducible testing environment. The data is provided as a list of lists, which is then passed to the `createDataFrame` method of the SparkSession object.

Viewing the initial DataFrame using the `.show()` action confirms the successful setup and provides a baseline against which we can compare the results of the transformation operations. This baseline is essential for verifying that the new constant columns are correctly appended without altering the original data integrity or structure.

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: Practical Implementation of Numeric Constants

In our first practical example, we implement Method 1 by adding a constant numerical column. We aim to introduce a column named `salary`, assigning a fixed integer value of **100** to every record in the previously defined DataFrame. This operation utilizes the imported `lit` function to convert the integer 100 into a Column expression, which is then handled by `withColumn()`.

The transformation itself is **non-mutating**; `withColumn()` returns a completely new DataFrame instance, incorporating the new column while leaving the original `df` untouched. This immutability is a core concept in Spark, crucial for fault tolerance and chaining complex transformations reliably. The resulting DataFrame maintains all original data fields (`team`, `conference`, `points`, `assists`) and appends the `salary` column at the end of the schema.

Upon reviewing the output, we can visually confirm that every row--regardless of its original data points--now holds the precise constant value of **100** in the new column. This consistency verifies that the `lit()` function successfully broadcasted the constant value across all distributed data partitions, demonstrating the power and simplicity of this transformation for large-scale data standardization.

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

As observed in the result, the new column `salary` has been successfully appended to the dataset. Every record now uniformly displays the value **100**, confirming the intended behavior of using `lit(100)` within the `withColumn()` function.

Example 2: Practical Implementation of String Constants

The second example focuses on adding a constant string value, which is crucial for assigning labels or category flags to our data. Here, we add a column named `league` and assign the constant string value **'NBA'** to all rows. Just as before, the string `'NBA'` must be wrapped by the `lit()` function to be correctly interpreted as a literal Column expression by the Spark engine.

This transformation demonstrates the versatility of the `lit()` function, confirming its ability to handle different data types seamlessly, whether they are numeric or textual. Using a constant string is exceptionally useful for tasks such as data lineage tracking, where every record processed in a specific batch must carry a marker indicating its source or processing date, represented as a string. It provides immediate, human-readable context to the data.

By executing the code snippet below, we can see the immediate effect of appending the string constant column. This addition often streamlines downstream data filtering, aggregation, and joining operations, as the consistent label provides a reliable key for grouping data segments related to the **'NBA'** context, simplifying complex analytical queries.

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

The resulting DataFrame now includes the `league` column, where every entry is uniformly set to **NBA**, exactly matching the specification provided to the `lit()` function. This confirms that the approach is equally effective for string literals.

Best Practices and Considerations for PySpark Data Transformation

When incorporating constant columns into your large-scale PySpark workflows, adhering to best practices ensures optimal performance and maintainability. Always prioritize using built-in functions like `lit()` over custom **User Defined Functions (UDFs)** for constant assignments. UDFs introduce serialization overhead and inhibit Spark's Catalyst optimizer, leading to slower execution times, whereas `lit()` function is highly optimized and integrated directly into the Spark SQL engine.

The nature of the `withColumn()` operation guarantees immutability, meaning it always produces a new DataFrame. While this is essential for reliability, developers must manage memory carefully when chaining many transformations, as each step generates a new object. It is often advisable to assign the result of the transformation back to the original variable name (e.g., `df = df.withColumn(...)`) or persist the intermediate result if it is needed multiple times downstream to prevent recomputation.

Finally, be mindful of data types. Although `lit()` performs automatic type inference, explicitly casting complex numerical constants or dates might sometimes be necessary to prevent unexpected schema changes. For instance, if a specific column must be `DecimalType(10, 2)`, you may need to use `.withColumn('new_col', lit(100.00).cast('decimal(10, 2)'))` to enforce the precise scale and precision required for financial or scientific applications, ensuring strict type compliance across the cluster.

Note #1: Non-Mutating Operation: The `withColumn` function returns a new DataFrame with the specified column added or modified, ensuring that the original DataFrame object remains

unchanged, adhering to Spark's immutable architecture.

Note #2: Literal Value Creation: The `lit()` function is specifically designed to create a Column expression containing a literal, constant value. This is the correct method for inserting fixed values into a distributed DataFrame efficiently.

The ability to quickly and efficiently add columns with constant values is a cornerstone of effective data preparation in PySpark. By mastering the combination of `withColumn()` and `lit()`, data engineers can ensure that their distributed datasets are consistently structured and ready for advanced analytical processing.

For further learning, the following tutorials explain how to perform other common tasks in PySpark: