How to Multiply Two Columns in PySpark: A Step-by-Step Guide

Authored by

stats writer

February 6, 2026

In the realm of large-scale data processing, efficiently manipulating data structures is paramount. When working with the PySpark framework--the Python API for Apache Spark--a common requirement is performing element-wise arithmetic operations, such as multiplying the values across two columns within a PySpark DataFrame. This crucial task is primarily handled by the dedicated withColumn function, which facilitates data transformation while preserving data integrity.

The withColumn function is designed to add a new column to a DataFrame or replace an existing one. It accepts two primary arguments: the name of the new or targeted column, and a PySpark Column expression defining the calculation. For multiplication, the standard Python arithmetic operator, the asterisk (**), is employed directly between the two target columns. Practical applications for this operation include calculating the total cost (multiplying "price" by "quantity") or determining total profit (multiplying "sales" by "profit margin"). The following detailed guide provides comprehensive methods and practical examples for implementing column multiplication effectively.

Multiply Two Columns in PySpark (With Examples)

Core Methods for PySpark Column Multiplication

In working with PySpark DataFrames, two primary methodologies exist for performing column multiplication. The choice between these methods depends heavily on whether the multiplication is straightforward and unconditional, or if it requires specific criteria to be met before the calculation is executed. Both methods rely on the robust withColumn transformation, which is fundamental to modifying DataFrame structure.

Understanding the core syntax of withColumn is essential. It allows data engineers to derive new features based on existing data fields. When multiplying columns, the operation treats each row independently, ensuring that the resulting new column contains the product of the corresponding values from the two input columns. This approach maintains the parallel processing efficiency that Apache Spark is known for.

Method 1: Direct, Unconditional Column Multiplication

The simplest and most direct approach involves multiplying two columns without any intervening conditional logic. This method is utilized when every row in the resulting column should represent the product of the two source columns. This is often used for straightforward financial calculations, such as calculating total sales value from unit price and quantity sold across an entire dataset.

To execute this, the withColumn function is called on the existing DataFrame. The second argument defines the arithmetic operation using dot notation to reference the column objects and

the standard ***** operator for multiplication.

df_new = df.withColumn('revenue', df.price * df.amount)

This specific code snippet generates a new column named **revenue**. The values in this **revenue** column are calculated by multiplying the numerical entries found in the **price** column by those in the **amount** column on a row-by-row basis. This transformation results in a new PySpark DataFrame, **df_new**, which includes the newly derived field.

Method 2: Conditional Column Multiplication using `when`

In many real-world scenarios, arithmetic operations must be conditional. For instance, if a transaction is flagged as a refund, the revenue generated from that transaction must be recorded as zero, regardless of the original price and amount. To handle such complex requirements, PySpark provides the powerful when function, which is imported from the `pyspark.sql.functions` module.

The when function allows developers to specify a condition, and a result if that condition is true (the 'then' component). If the condition is false, the `.otherwise()` method provides the default result or calculation to be applied. This structure is highly analogous to SQL's CASE statement or if-then-else logic found in general programming languages.

from pyspark.sql.functions import when

df_new = df.withColumn('revenue', when(df.type=='refund', 0)
.otherwise(df.price * df.amount))

As demonstrated above, this conditional operation creates the new column **revenue**. It first evaluates the **type** column. If the value in **type** is 'refund', the **revenue** is immediately set to **0**. Only if the transaction is not a 'refund' will the standard multiplication of **price** and **amount** be performed. This ensures data accuracy in datasets where different transaction types necessitate different calculation rules.

Example 1: Implementing Direct Column Multiplication

To illustrate the application of Method 1, we begin by constructing a sample PySpark DataFrame. This DataFrame simulates transactional data, containing simple information about the unit **price** of items and the **amount** sold. This setup is crucial for demonstrating how the multiplication operator works directly on the column objects.

We initialize a SparkSession, define the sample data as a list of lists, and specify the column

schema. The resulting DataFrame, **df**, provides the foundation for our calculation, ensuring we have clear input values before performing the transformation.

```
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()

+-----+------+
|price|amount|
+-----+------+
| 14| 2|
| 10| 3|
| 20| 4|
| 12| 3|
| 7| 3|
| 12| 5|
| 10| 2|
| 10| 3|
+-----+------+
```

Once the DataFrame is initialized and displayed, the next step is to apply the multiplication transformation using the withColumn function. We aim to create the **revenue** column by simply multiplying **df.price** by **df.amount**.

#create new column called 'revenue' that multiplies price by amount
df_new = df.withColumn('revenue', df.price * df.amount)

#view new DataFrame
df_new.show()

```
+-----+------+-------+
|price|amount|revenue|
+-----+------+-------+
| 14| 2| 28|
| 10| 3| 30|
| 20| 4| 80|
| 12| 3| 36|
| 7| 3| 21|
| 12| 5| 60|
| 10| 2| 20|
| 10| 3| 30|
+-----+------+-------+
```

The output clearly confirms that the values in the newly added **revenue** column are the direct product of the corresponding entries in the **price** and **amount** columns across every row. This successfully demonstrates the mechanism for unconditional column multiplication in PySpark.

Example 2: Implementing Conditional Multiplication Logic

Building upon the previous example, we now introduce a conditional element. This scenario involves accounting for transaction types, specifically differentiating between standard 'sale' transactions and 'refund' transactions. Since refunds should result in zero revenue, we must integrate conditional logic before performing the multiplication. This requires using the when function, as detailed in Method 2.

We begin by defining an updated dataset that includes a third column, **type**, indicating whether the row represents a 'sale' or a 'refund'. This step is essential to test the conditional application of the multiplication operation accurately.

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()

```
+-----+------+------+
|price|amount| type|
+-----+------+------+
| 14| 2| sale|
| 10| 3| sale|
| 20| 4|refund|
| 12| 3| sale|
| 7| 3|refund|
| 12| 5|refund|
| 10| 2| sale|
| 10| 3| sale|
+-----+------+------+
```

With the DataFrame **df** prepared, we now apply the conditional logic. We import the necessary **when** function and chain it with the withColumn transformation. The condition checks if the **type** column equals 'refund'. If true, **revenue** is 0; otherwise, the standard multiplication proceeds.

from pyspark.sql.functions import when

#create new column called 'revenue'
df_new = df.withColumn('revenue', when(df.type=='refund', 0)
.otherwise(df.price * df.amount))

#view new DataFrame
df_new.show()

```
+-----+------+------+-------+
|price|amount| type|revenue|
+-----+------+------+-------+
| 14| 2| sale| 28|
| 10| 3| sale| 30|
| 20| 4|refund| 0|
| 12| 3| sale| 36|
| 7| 3|refund| 0|
| 12| 5|refund| 0|
| 10| 2| sale| 20|
| 10| 3| sale| 30|
+-----+------+------+-------+
```

Upon reviewing the final DataFrame, it is clear that for rows where the **type** is 'refund', the **revenue** field has been correctly set to **0**, overriding the potential product calculation (e.g., 20 * 4 = 80, but the result is 0). Conversely, transactions marked 'sale' successfully underwent the multiplication operation. This confirms the efficacy of using the when function for implementing conditional arithmetic in PySpark transformations.

Conclusion and Further PySpark Operations

Mastering column multiplication in PySpark is a fundamental skill for data manipulation and feature engineering. Whether performing a simple, direct calculation or incorporating sophisticated conditional logic using the when function, the core tool remains the highly adaptable **withColumn** method. These techniques ensure that large-scale data calculations are performed accurately and efficiently within the distributed computing environment of Apache Spark.

For data professionals seeking to expand their proficiency, PySpark offers a wide range of analytical and transformation tools. Similar methods can be applied to perform other common tasks, such as column division, addition, or complex aggregations necessary for comprehensive data analysis.

The following resources detail how to perform other common tasks in PySpark: