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

Authored by

stats writer

January 2, 2026

ARABPSYCHOLOGY.COM

RECOMMENDED CITATION

stats writer (2026). *How to Multiply Columns in PySpark DataFrames: A Step-by-Step Guide*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=110479

The ability to manipulate and calculate values across different columns is fundamental in large-scale data processing. Using the powerful DataFrame API in PySpark, data engineers and analysts can easily perform mathematical operations, such as multiplication, to derive new insights or features. This operation is crucial for tasks like calculating total sales figures (price multiplied by quantity) or normalizing data.

In PySpark, column multiplication is typically achieved using standard Python arithmetic operators combined with the withColumn() method. The withColumn() function is essential here, as it allows us to introduce a new column into the existing DataFrame, holding the resulting product of the two source columns. This approach maintains the integrity of the original data while efficiently generating the required derived field, which is often termed a **cross product** in mathematical contexts.

Core Methods for PySpark Column Multiplication

When working with DataFrames in PySpark, there are two primary approaches for multiplying columns, depending on whether the calculation needs to be straightforward or conditional. Understanding these methods is key to performing robust data transformations and feature engineering.

Method 1: Simple Column Multiplication

The most direct way to multiply two columns is using the standard arithmetic operator (*) directly within the withColumn() function. This method is suitable when the multiplication should apply uniformly across all rows without requiring any specific logical checks. We define the new column name and provide the expression which references the existing columns using dot notation (e.g., `df.price * df.amount`).

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

This code snippet generates a new column named **revenue**. The values within this column are calculated by multiplying the corresponding row values from the **price** column by the values in the **amount** column. This simple, declarative syntax is efficient for batch processing in distributed environments.

Method 2: Conditional Column Multiplication Using `when()`

Often, real-world data requires conditional logic before calculation. For example, if a transaction is a refund, the calculated revenue should be zero regardless of the original price and amount. To handle such scenarios, PySpark provides the `when()` function, imported from

ARABPSYCHOLOGY.COM

`pyspark.sql.functions`. The `when()` function allows you to apply different calculations or values based on a specified boolean condition within another column, providing powerful flow control.

from pyspark.sql.functions import when

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

In this more complex example, the new **revenue** column is derived conditionally. If the value in the **type** column equals 'refund', the revenue is explicitly set to **0**. Otherwise, for all other transaction types, the standard multiplication (price times amount) is executed. This conditional approach is vital for ensuring accurate financial or operational metrics and showcases the flexibility of the PySpark API.

Prerequisite: Setting Up the PySpark Environment

Before executing any multiplication operations, we must ensure a DataFrame is properly initialized within a live SparkSession. The SparkSession acts as the entry point to programming Spark with the DataFrame API. For illustrative purposes, we will define sample datasets that simulate real-world transactional data for both simple and conditional calculations.

Initializing the environment involves importing the necessary modules, specifically `SparkSession` from `pyspark.sql`, and then creating or retrieving an active Spark session using `SparkSession.builder.getOrCreate()`. This step is mandatory for any PySpark operation and guarantees that the distributed computing context is correctly established for processing large data volumes efficiently.

The following examples show how these methods are used in practice, starting with the creation of the source DataFrame followed by the specific multiplication logic tailored to different business needs.

Example 1: Simple Multiplication of Two Columns

Creating the Base DataFrame for Revenue Calculation

For our first demonstration, we will create a simple dataset representing various retail transactions. This dataset contains two numerical columns: **price**, representing the cost of a single item, and **amount**, representing the quantity sold. Our goal is straightforward: calculate the total **revenue** generated for each line item by multiplying these two source columns.

The following code block defines the initial data as a list of lists, specifies the column schema, and

ARABPSYCHOLOGY.COM

creates the base DataFrame using `spark.createDataFrame()`. This setup ensures we have a clean dataset ready for 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 structure and content
df.show()

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

Executing the Basic Column Multiplication

To perform the required multiplication, we apply the withColumn() transformation. This powerful

ARABPSYCHOLOGY.COM

function is the primary mechanism for adding or replacing columns in a DataFrame. It takes two key arguments: the name of the new column ('revenue') and a column expression that defines the calculation (`df.price * df.amount`).

It is important to note that this operation utilizes Spark's highly optimized Catalyst Optimizer. The use of native column expressions, rather than Python loops or UDFs, ensures that the computation is pushed down to the execution engine, guaranteeing efficiency and scalability across the cluster, which is essential when dealing with petabytes of data.

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

ARABPSYCHOLOGY.COM

Verifying the Calculated Output

Upon inspecting the resulting DataFrame, `df_new`, we can clearly observe the addition of the **revenue** column. Each entry in this new column confirms that it is the precise product of the corresponding values in the **price** and **amount** columns. For instance, the first record shows 14 multiplied by 2 yields 28. This successful verification confirms that the simple multiplication logic was correctly applied across all records.

Example 2: Conditional Multiplication Using `when()`

Introduction to Conditional Logic in DataFrames

Data operations frequently require decision-making logic based on attributes within the data itself. For example, calculating true revenue often necessitates accounting for exceptions, such as 'refunds' or 'adjustments,' which should typically result in zero revenue contribution for that specific transaction. We utilize the `when()` function to implement this control flow directly within the DataFrame transformation pipeline.

For this example, we extend the retail dataset to include a categorical column named **type**, indicating whether the transaction was a standard 'sale' or a 'refund'. Our objective is to calculate revenue normally for sales, but force the revenue to zero for any row marked as a refund, simulating robust financial logic.

Setting Up the DataFrame with Categorical Data

The setup involves initializing the DataFrame using the same process as before, but incorporating the additional **type** column into both the data structure and the column definitions. We must ensure the SparkSession is active to process this data.

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()

+-----+------+------+

ARABPSYCHOLOGY.COM

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

Applying Conditional Multiplication Logic

To implement the conditional multiplication, we first import the `when()` function from `pyspark.sql.functions`. We then use withColumn(), passing the `when()` clause as the calculation expression. The structure is hierarchical: it checks the condition (`df.type == 'refund'`); if true, it applies the result (`0`); and if false, it executes the subsequent logic using `.otherwise()`, which contains our standard multiplication (`df.price * df.amount`).

This pattern is extremely powerful as it allows complex multi-layered conditional logic to be executed efficiently in parallel, far superior to traditional row-by-row iteration methods.

```
from pyspark.sql.functions import when

#create new column called 'revenue' with conditional logic
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|
```

ARABPSYCHOLOGY.COM

12	5	refund	0
10	2	sale	20
10	3	sale	30
+-----+------+------+-------+

Analyzing the Conditional Output

Reviewing the final output, we successfully confirm that the conditional logic was applied. Rows where the **type** column is 'sale' (e.g., Row 1: 14 * 2 = 28) correctly show the product of **price** and **amount**. Crucially, rows where the **type** is 'refund' (e.g., Row 3: 20 * 4 = 80, but result is 0) show in the new **revenue** column, overriding the arithmetic calculation as dictated by the business rule.

This outcome demonstrates the robust capability of using `when()` for complex column derivations, allowing data engineers to build highly sophisticated feature engineering pipelines using native PySpark functions.

Best Practices for PySpark Column Operations

While both methods successfully achieve column multiplication, adhering to certain best practices ensures performance, maintainability, and scalability in large-scale production environments.

Avoid UDFs for Simple Arithmetic: Whenever possible, use native DataFrame operations (like `*`, `+`, or built-in functions like `when()`). User-Defined Functions (UDFs) written in Python incur significant serialization and deserialization overhead, severely degrading performance compared to native Spark expressions optimized by the Catalyst engine.

Use Explicit Column References: It is best practice to reference columns explicitly (e.g., `df.column_name` or `col("column_name")`) rather than relying on positional indexing. Explicit references make the code more readable and robust against schema changes in the source data.

Understand Immutability: Remember that DataFrames in PySpark are immutable. The withColumn() method does not modify the original DataFrame (`df`); instead, it returns an entirely new DataFrame (`df_new`) containing the transformation. Chaining multiple transformations is therefore a clean and efficient process.

Conclusion and Further Reading

Multiplying columns in PySpark is a fundamental operation easily managed through the withColumn() method. Whether you require a simple arithmetic calculation or complex conditional logic using `when()`, PySpark offers efficient and scalable solutions for large datasets. Mastery of

ARABPSYCHOLOGY.COM

these core DataFrame transformation techniques is essential for effective data analysis and preparation in a distributed computing environment.

If you are looking to expand your knowledge of PySpark and DataFrame manipulation, the following tutorials explain how to perform other common tasks:

Joining multiple DataFrames
Aggregating data using GroupBy
Casting column data types

ARABPSYCHOLOGY.COM