Sum Multiple Columns in PySpark (With Example)

Authored by

Mohammed looti

November 11, 2025

Introduction to Efficient Row-Wise Summation in PySpark

When dealing with massive datasets, the ability to perform efficient row-wise calculations is crucial. **PySpark**, the Python API for **Apache Spark**, offers powerful methods for aggregating values across specific columns within a **DataFrame**. A frequent requirement in data analysis is calculating the total value derived from several numeric columns for every record. This article details the most concise and idiomatic method to sum values across multiple designated columns simultaneously in PySpark, leveraging built-in functions optimized for distributed computing.

The challenge in this operation is efficiently instructing Spark to treat a list of columns as inputs for a single arithmetic operation applied row-by-row. PySpark operations must be expressed using Spark SQL functions or expressions to ensure they are executed efficiently across the cluster. We focus on utilizing the powerful `expr` function, which allows us to pass a SQL-style expression directly to the DataFrame transformation API, significantly simplifying the process and maximizing performance.

Core Syntax for Summing Multiple Columns

To sum the values present across a list of columns in a PySpark DataFrame, we combine the `withColumn` transformation with the `expr` function, which is available via `pyspark.sql.functions`. The core technique involves dynamically generating a string expression that explicitly lists the column identifiers separated by the addition operator (`+`).

The standard syntax pattern for this aggregation is demonstrated below:

from pyspark.sql import functions as F

#define columns to sum
cols_to_sum =

#create new DataFrame that contains sum of specific columns
df_new = df.withColumn('sum', F.expr('+'.join(cols_to_sum)))

This implementation constructs a new column, provisionally named **sum**, which contains the aggregate of values across the specified columns (**game1**, **game2**, and **game3** in this example). The efficacy of this method stems from Python's `join()` method, which dynamically creates the required **Spark SQL** expression string--for instance, `"game1+game2+game3"`. This highly optimized expression is then processed row-wise by `F.expr()`, ensuring quick execution within the distributed framework.

Detailed Example: Calculating Team Total Scores

To provide a clear application of this methodology, we will use a practical scenario involving sports statistics. Suppose we possess a PySpark DataFrame containing information about points scored by various basketball teams during three distinct games. Our precise objective is to compute the total points accumulated by each team across all three games and seamlessly integrate this result into a new column in the DataFrame.

The initial step requires setting up the Spark Session and defining the source data 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|game1|game2|game3|
+-------+-----+-----+-----+
Mavs	25	11	10
Nets	22	8	14
Hawks	14	22	10
Kings	30	22	35
Bulls	15	14	12
Blazers	10	14	18
+-------+-----+-----+-----+

The DataFrame `df` is now ready, featuring the numeric columns `game1`, `game2`, and `game3`. Our immediate objective is to apply the summation logic to create a new column, **sum**, reflecting the combined score for each team across these three dimensions.

Applying the Summation Logic and Verification

We now execute the required transformation by defining the list of columns to be summed and applying the `withColumn` operation using the dynamic expression generated by string joining. It is imperative to ensure that the `functions` module is imported and aliased as `F` for concise access to `F.expr`.

from pyspark.sql import functions as F

#define columns to sum
cols_to_sum =

#create new DataFrame that contains sum of specific columns
df_new = df.withColumn('sum', F.expr('+'.join(cols_to_sum)))

#view new DataFrame
df_new.show()

+-------+-----+-----+-----+---+
| team|game1|game2|game3|sum|
+-------+-----+-----+-----+---+
Mavs	25	11	10	46
Nets	22	8	14	44
Hawks	14	22	10	46
Kings	30	22	35	87
Bulls	15	14	12	41
Blazers	10	14	18	42
+-------+-----+-----+-----+---+

The resulting DataFrame, `df_new`, successfully incorporates the new **sum** column. This column accurately reflects the summation of values across **game1**, **game2**, and **game3** for every record. This method demonstrates exceptional flexibility; should the number of columns requiring summation change, only the list `cols_to_sum` requires adjustment, maintaining the integrity and conciseness of the core transformation logic.

We can verify the correctness of the generated results by confirming the aggregate scores for the initial rows:

The total points for the **Mavs** team is calculated as 25 + 11 + 10, correctly yielding **46**.
The total points for the **Nets** team is calculated as 22 + 8 + 14, resulting in **44**.
The total points for the **Hawks** team is calculated as 14 + 22 + 10, resulting in **46**.

These verifications confirm that the combination of `F.expr` and string manipulation correctly executes the desired row-wise summation, providing an efficient mechanism for complex arithmetic within PySpark DataFrames.

Understanding the `withColumn` Transformation

The success of this operation relies heavily on the use of the **withColumn** function. This function is fundamental in PySpark for DataFrame manipulation, as it returns a new DataFrame resulting from adding a new column or replacing an existing column with a specified expression. In our case, the expression is the dynamically generated summation string passed through `F.expr`.

It is crucial to understand that PySpark DataFrames are immutable. Therefore, `withColumn` does not modify the original DataFrame (`df`); instead, it produces a new DataFrame (`df_new`) containing the results of the transformation. This immutable nature is key to Spark's fault tolerance and distributed processing model.

The `withColumn` function accepts two primary arguments: the name of the new column (`'sum'`) and the column expression that defines its values (`F.expr('+'.join(cols_to_sum))`). This structure makes it highly versatile, allowing for simple column additions or complex transformations using various PySpark functions.

Alternative Method: Utilizing Python's Reduce Function

While the `F.expr` method is often considered the most concise for simple summation, an alternative, more programmatic approach exists using **Python's reduce function** in conjunction with PySpark's native column arithmetic capabilities. This method is preferred when developers wish to avoid generating SQL strings or need greater control over complex column operations.

This alternative method requires converting the list of column names into a list of PySpark Column objects (using `F.col(column_name)`) and then iteratively applying the addition operator across these objects using `reduce` from Python's `functools` library. Since PySpark column objects support operator overloading, they behave like numerical inputs in the context of `reduce`, accumulating the sum:

Convert column names to Column objects: .
Apply `reduce` using the addition operator (`lambda x, y: x + y`) to combine the Column objects.
Pass the resulting aggregated Column expression to `df.withColumn()`.

Although slightly more verbose, this method offers a pure PySpark API solution, avoiding potential edge cases related to SQL string formatting when dealing with highly complex transformations or column names that might require special escaping in SQL.

Summary of Best Practices

The technique employing `F.expr('+'.join(cols_to_sum))` remains a highly recommended practice for summing multiple columns due to its readability and performance. By leveraging native **Spark SQL** execution, it minimizes overhead.

When implementing this column summation, users must adhere to specific best practices:

Numeric Integrity: All columns included in `cols_to_sum` must possess a numeric data type (e.g., Integer, Double). Inclusion of string or complex types will lead to execution errors.
Handling Nulls: If a row contains `NULL` values in any of the summed columns, the resulting sum for that row will also be `NULL` by default in Spark's arithmetic model. To treat nulls as zero, preprocessing steps such as using `F.coalesce(F.col(c), F.lit(0))` must be applied before the summation.
Performance: Utilizing native PySpark functions like `F.expr` and `withColumn` maximizes performance, as these operations are processed directly on the JVM, significantly faster than equivalent logic implemented via less optimized Python UDFs.

Additional Resources

For further exploration of PySpark transformations, data type handling, and advanced DataFrame manipulation techniques, we recommend reviewing the following official documentation pages:

PySpark SQL Functions Documentation: A comprehensive guide to all available PySpark functions for efficient data manipulation.
DataFrame Transformation Guide: Detailed explanations and examples of common DataFrame operations, including adding and manipulating columns.