How to Find the Maximum Value Across Multiple Columns in a PySpark DataFrame

Authored by

stats writer

February 8, 2026

PySpark is recognized globally as an exceptionally powerful interface for harnessing the capabilities of Apache Spark, enabling users to perform sophisticated data analysis and manipulation across massive, distributed datasets. When dealing with complex analytical scenarios, a common requirement is calculating summary statistics, such as the maximum value, not just across rows or columns traditionally, but specifically across a defined subset of columns for each individual record. This functionality is crucial in diverse fields, ranging from financial risk assessment to sports analytics, where identifying the peak performance or highest measurement point is paramount.

The ability of PySpark to efficiently calculate the maximum value across multiple columns in a DataFrame significantly streamlines the data processing workflow. This means that users can precisely and rapidly determine the highest recorded value within a particular observational unit (a row), even when that value is spread across dozens or hundreds of features or metrics represented as columns. Leveraging optimized native Spark SQL functions, PySpark ensures that this task is executed efficiently, regardless of the dataset scale. This efficiency makes it an indispensable tool for data engineers and analysts requiring robust, scalable solutions for critical decision-making processes.

Our focus in this comprehensive guide is to demonstrate the standardized, most efficient method for achieving this cross-column maximum calculation using built-in PySpark functions. We will explore the specific function designed for this purpose, illustrate its proper implementation through practical code examples, and analyze the resulting output to ensure complete clarity. Mastery of this technique is fundamental for anyone working with large-scale tabular data within the DataFrame abstraction.

PySpark: Calculate Max Value Across Columns

Introducing the **greatest** Function for Cross-Column Analysis

The core difficulty in calculating a row-wise maximum across columns in a distributed computing environment like Apache Spark lies in ensuring that the operation remains highly performant and avoids unnecessary data shuffling or serialization. Fortunately, Spark SQL provides a highly optimized function specifically designed for this purpose: the `greatest` function. Unlike the standard `max()` aggregate function, which finds the maximum value within a single column across all rows, `greatest` performs a row-wise comparison across all specified columns, returning the largest value found for that particular row.

The `greatest` function is defined within the `pyspark.sql.functions` module, which is the repository for most of the high-level, column-oriented transformations available in PySpark. When invoked, it requires two or more column names as arguments. It evaluates these columns

simultaneously for every record in the DataFrame. This approach is significantly more efficient than attempting to write complex user-defined functions (UDFs) in Python, as `greatest` is executed using highly optimized Scala/Java code within the Spark Virtual Machine.

To effectively utilize this capability, one typically combines the `greatest` function with the withColumn transformation. The withColumn method is essential for adding a new column to the existing DataFrame, where the values of this new column are derived from the calculation performed by `greatest`. This separation of concerns--calculating the maximum and appending the result--ensures clean, readable, and highly scalable code, adhering to functional programming paradigms common in Spark.

Core Syntax for Calculating the Maximum Value

The fundamental step in performing this cross-column analysis is importing the necessary function and applying it within a transformation call. The syntax remains consistent whether you are analyzing three columns or thirty, provided all specified columns contain comparable numeric data types (integers, floats, or decimals). If any of the input columns contain null values, the behavior of `greatest` is simple: if all input columns are null for a given row, the result will be null; otherwise, it returns the maximum non-null value among the inputs.

You can use the following standard PySpark syntax to calculate the max value across a defined subset of columns in a DataFrame:

from pyspark.sql.functions import greatest

#find max value across columns 'game1', 'game2', and 'game3'
df_new = df.withColumn('max', greatest('game1', 'game2', 'game3'))

In this crucial initial snippet, we first import the `greatest` function from the `pyspark.sql.functions` library. Following this, we invoke the withColumn method on our existing DataFrame, named `df`. The first argument to withColumn, `'max'`, defines the name of the new column we are creating. The second argument applies the `greatest` function, passing the column identifiers `'game1'`, `'game2'`, and `'game3'` as inputs. This process yields a new DataFrame, `df_new`, which now includes the calculated row-wise maximum.

This particular example demonstrates the creation of a new column explicitly named **`max`**. This column will hold the derived maximum value for each row, calculated by comparing the individual values present in the **`game1`**, **`game2`**, and **`game3`** columns within the source DataFrame. It is important to remember that withColumn returns an entirely new DataFrame; it does not modify the original `df` in place, aligning with the immutable nature of Spark transformations.

Detailed Example: Initializing the PySpark Environment and Data

To demonstrate this functionality practically, we will construct a scenario involving sports performance data. Suppose we are analyzing points scored by various basketball teams across three distinct games. Our goal is to determine the highest individual game score achieved by each team in the dataset. This requires setting up the DataFrame structure and initializing a PySpark `SparkSession`, which serves as the entry point for all Spark functionality.

The following detailed example shows how to initialize the session, define the data structure, and create the input DataFrame. We use hardcoded data for simplicity, but this process mirrors reading data from distributed sources like HDFS or S3. Note the explicit definition of the schema via the `columns` list, which ensures clarity and structure in the resulting data object.

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

This resulting DataFrame, `df`, now holds six records, each corresponding to a different basketball team. The crucial numerical columns are `game1`, `game2`, and `game3`, representing the points scored in those respective games. We are now prepared to apply the transformation to identify the peak performance score for each team across these three metrics. The immediate goal is to add a new column, which we will call **`max`**, containing the maximum points scored by each player across all three games recorded.

Implementing the Maximum Calculation using withColumn and greatest

The implementation step is highly straightforward, leveraging the power of the `pyspark.sql.functions.greatest` utility. We apply the logic discussed earlier: import the function, define the new column name, and pass the names of the source columns we wish to compare row-wise. This methodology ensures that the calculation is performed in parallel across the distributed partitions of the DataFrame, maximizing computational efficiency.

We can use the following syntax to execute the calculation and append the result to a new DataFrame:

from pyspark.sql.functions import greatest

#find max value across columns 'game1', 'game2', and 'game3'
df_new = df.withColumn('max', greatest('game1', 'game2', 'game3'))

#view new DataFrame
df_new.show()

+-------+-----+-----+-----+---+
| team|game1|game2|game3|max|
+-------+-----+-----+-----+---+
Mavs	25	11	10	25
Nets	22	8	14	22
Hawks	14	22	10	22
Kings	30	22	35	35
Bulls	15	14	12	15
Blazers	10	14	18	18
+-------+-----+-----+-----+---+

Executing this code sequence generates `df_new`. This new object is identical to the original `df`, but it includes the new **`max`** column positioned as the final column. This column contains the result of the row-wise comparison performed by `greatest` on the `game1`, `game2`, and `game3` columns. Crucially, the use of withColumn here ensures that the DataFrame remains manageable and that the computation engine automatically optimizes the lineage of the transformation.

Analyzing the Result and Verification

Upon examining the output of `df_new.show()`, it is immediately evident that the new **`max`** column accurately reflects the highest value found among the three game columns for every corresponding team. This verification step is vital to confirm the correct application of the `greatest` function and the successful integration via withColumn.

The new **`max`** column consistently holds the maximum value across the **`game1`**, **`game2`**, and **`game3`** columns. We can verify this through a few specific rows of the data:

The max of points for the **`Mavs`** player is **`25`** (from 25, 11, 10).

The max of points for the **`Nets`** player is **`22`** (from 22, 8, 14).

The max of points for the **`Hawks`** player is **`22`** (from 14, 22, 10).

The max of points for the **`Kings`** player is **`35`** (from 30, 22, 35).

And so on. This row-by-row confirmation demonstrates that the `greatest` function performs its duty precisely as intended, providing immediate insight into the peak performance metric for each observational unit. This simple yet powerful transformation saves significant time and complexity compared to achieving the same result using traditional iterative loops or less optimized methods in large-scale data systems.

Understanding the withColumn Transformation

A key component of this process is the withColumn function. It is one of the most frequently used transformation methods in PySpark, allowing users to add a new column or replace an existing one. It adheres to the fundamental principle of immutable DataFrame operations: rather than modifying the original object, it returns a new DataFrame with the requested changes applied.

In the context of calculating the maximum, withColumn takes two main parameters: the name of the new column (e.g., `'max'`) and a Spark SQL expression (in this case, `greatest('game1', 'game2', 'game3')`) that dictates how the values of the new column should be calculated. This clear separation makes code maintainable and allows the Spark Catalyst Optimizer to efficiently determine the most resource-effective way to execute the underlying computation, often pushing

the comparison logic down into the cluster nodes.

Had we named the new column `'game1'` instead of `'max'`, the withColumn function would have effectively replaced the original `game1` column with the new calculated maximum values. This versatility makes withColumn an indispensable tool for data preparation and feature engineering, ensuring that calculated fields, such as row-wise maximums, are seamlessly integrated into the distributed dataset.

Handling Dynamic Column Lists and Scaling

While the example above used a static list of three column names (`'game1'`, `'game2'`, `'game3'`), real-world data often involves dozens or even hundreds of columns that need comparison. Manually listing these columns within the `greatest` function call becomes impractical and error-prone. PySpark handles this scenario elegantly by allowing the column list to be generated dynamically using Python techniques.

If you had 50 columns named `'metric_1'` through `'metric_50'`, you would first generate a list of these column strings. Then, you would use the Python splat operator (`*`) to unpack this list directly into the `greatest` function. This ensures that the greatest function receives the individual column arguments it expects, even though the list was programmatically created. This adaptability is essential for building robust, scalable data pipelines that must handle schema drift or changing input formats.

Furthermore, for extreme scalability concerns, the use of greatest is highly recommended over alternatives like iterating through the columns using Resilient Distributed Dataset (RDD) operations or applying generic Python UDFs. UDFs, particularly those written purely in Python, introduce significant serialization and deserialization overhead, forcing data to move back and forth between the JVM and the Python interpreter. In contrast, `greatest` is a native Spark SQL function that operates directly on the internal optimized data structures, leading to substantial performance gains, especially as the size of the DataFrame grows into the terabyte range.

Summary of Best Practices

To ensure maximum efficiency and code readability when calculating row-wise maximums in PySpark, adhere to these key best practices. Always prioritize built-in Spark SQL functions like `greatest` over custom Python UDFs for simple, vectorizable operations such as finding maximums, minimums, or performing arithmetic. This choice minimizes computational overhead and allows Spark's optimizer to perform its best work.

Utilize the withColumn transformation method to introduce the new calculated column. This maintains the immutability of the DataFrame and promotes a clear, declarative coding style.

Ensure that all input columns passed to `greatest` are of compatible numeric data types; comparing strings or mixing types requires explicit casting beforehand to guarantee predictable and accurate results.

Finally, always import the required functions explicitly from `pyspark.sql.functions` (e.g., `from pyspark.sql.functions import greatest`). This practice enhances code clarity, preventing namespace collisions and making the code easier to debug and maintain. The complete documentation for the PySpark withColumn function is available online, providing further details on its advanced usage and parameters.