Calculate the Max Value of a Column in PySpark

Authored by

stats writer

November 16, 2025

Working with large-scale datasets often requires calculating statistical measures, and finding the maximum value within a specific column is a fundamental operation. In the context of big data processing, PySpark provides highly optimized functions to perform these aggregations efficiently across a distributed environment. This guide details two primary, robust methods for calculating the maximum value(s) in a DataFrame column(s).

Understanding these techniques is essential for effective data profiling and analysis. We will explore methods leveraging the power of PySpark's SQL functions module, specifically demonstrating how to handle both single-column and multi-column maximum calculations. Each approach is tailored to different output requirements and operational contexts within your data pipeline.

Method 1: Calculating the Maximum Value for a Single Column using `agg()`

The most idiomatic way to calculate a single aggregated statistic across an entire DataFrame column is by utilizing the `agg()` function in conjunction with the specialized functions provided by the `pyspark.sql.functions` module (imported here as `F`).

The `agg()` transformation is highly efficient as it executes the aggregation logic on the distributed clusters before returning a concise result. When calculating the maximum value for a specific column, such as 'game1', we pass `F.max('column_name')` into the `agg()` function. Because `agg()` returns a new DataFrame containing the aggregated result, we typically use the `collect()` action to retrieve the value back to the driver program as a Python list of `Row` objects. Indexing () is then used to extract the scalar maximum value directly.

from pyspark.sql import functions as F

#calculate max of column named 'game1'
df.agg(F.max('game1')).collect()

Method 2: Calculating Maximum Values for Multiple Columns using `select()`

While the `agg()` method is excellent for retrieving a single result, sometimes you need to calculate the maximum value across several columns simultaneously and retain the output as a new DataFrame for further processing or immediate display. In this scenario, the `select()` transformation, combined with the `max` function, provides a cleaner alternative.

The `select()` function allows us to define new columns based on existing ones or apply aggregate functions directly. When we use `df.select(max(df.col1), max(df.col2), ...)`, PySpark calculates the maximum for each specified column and presents these results horizontally

in a single-row DataFrame. This method is particularly useful when comparing maximums side-by-side or when integrating the results into a broader PySpark workflow.

from pyspark.sql.functions import max

#calculate max for game1, game2 and game3 columns
df.select(max(df.game1), max(df.game2), max(df.game3)).show()

Initial PySpark Setup and DataFrame Definition

Before diving into the aggregation methods, we must establish a working environment and define the source DataFrame. This foundational setup involves initializing the SparkSession and populating the DataFrame with sample data representing team performance metrics across three games. The following code block demonstrates how to create the necessary DataFrame structure that will be used in the subsequent examples.

The data below serves as the basis for all subsequent calculations, allowing us to accurately verify the maximum values derived from the computational methods. Note the structure, featuring a categorical column ('team') and three numerical columns ('game1', 'game2', 'game3') that we will analyze.

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

Detailed Walkthrough: Single Column Max Example (`game1`)

We apply Method 1 to determine the highest score recorded in the **game1** column. This example showcases the streamlined process of using the `agg()` function, which is generally preferred when you need a single, scalar result returned to the driver program, enabling immediate use in subsequent Python logic. The execution of this command involves PySpark scanning the `game1` column across all partitions, identifying the maximum value, and returning that value (30) directly.

```
from pyspark.sql import functions as F

#calculate max of column named 'game1'
df.agg(F.max('game1')).collect()
```

```
30
```

As demonstrated by the output, the maximum value observed in the **game1** column is **30**. We can verify this result by manually inspecting the original DataFrame values for the 'game1' column (10, 14, 15, 22, 25, 30). The efficiency of PySpark ensures that this calculation remains performant even when dealing with gigabytes or terabytes of records.

Detailed Walkthrough: Multiple Column Max Example (`game1`, `game2`, `game3`)

We use Method 2 to simultaneously calculate the maximum values for **game1**, **game2**, and **game3** columns. By using `.show()` on the resulting DataFrame, the output clearly illustrates the maximum score achieved across all teams for each respective game category. This streamlined aggregation is a cornerstone of efficient statistical analysis in distributed computing.

Notice that when using `select()` with aggregation functions, the resulting column names default to the function applied, e.g., `max(game1)`. If custom column names were required, aliases would typically be applied within the `select()` statement. For quick inspection, however, the default

naming convention provides immediate clarity regarding the calculation performed.

from pyspark.sql.functions import max

#calculate max for game1, game2 and game3 columns
df.select(max(df.game1), max(df.game2), max(df.game3)).show()

```
+----------+----------+----------+
|max(game1)|max(game2)|max(game3)|
+----------+----------+----------+
| 30| 22| 35|
+----------+----------+----------+
```

The resulting single-row DataFrame provides the maximum values calculated for the three columns:

The max of values in the **game1** column is **30**.
The max of values in the **game2** column is **22**.
The max of values in the **game3** column is **35**.

Choosing Between `agg()` and `select()` for Maximum Calculation

While both `agg()` and `select()` can calculate maximum values, they serve distinct purposes regarding the output format and operational context. The `agg()` function is designed specifically for reducing a large amount of data into a single summary row, making it the canonical choice for overall dataset statistics. When combined with `collect()`, it provides the quickest way to pull a scalar value back to the driver.

Conversely, `select()` is a versatile transformation used for column manipulation and projection. When applying aggregation functions within `select()`, the resulting DataFrame still represents the aggregated output, but this method is often preferred when performing multiple simple aggregations or when the intent is to chain further DataFrame transformations onto the aggregated result, rather than immediately returning the value to the Python environment.

Performance Considerations in PySpark Aggregation

It is important to remember that both methods rely on distributed processing provided by PySpark. When calculating the maximum, Spark executes a shuffle operation to ensure all data for a specific group (in this case, the entire dataset, treated as one implicit group) is brought together before applying the `max` function. While aggregation is highly optimized, the cost is dependent on the size of the data being processed and the number of partitions.

For massive datasets, developers should minimize the use of `.collect()`, as this action forces all resulting data to be transferred from the distributed executors back to the single driver machine, potentially causing OutOfMemory errors if the resulting data set is large. Since the maximum calculation produces a very small output (a single row), using `collect()` with `agg()` is generally safe and highly convenient for retrieving the final scalar value.