Calculate the Minimum Value of a Column in PySpark

Authored by

stats writer

November 16, 2025

Calculating statistical measures is a fundamental requirement when performing data analysis, particularly within big data frameworks. When working with large datasets managed by PySpark, determining the minimum value of a column is a common aggregation task. This operation helps in identifying outliers, understanding the distribution's lower bound, or performing data validation checks. This guide provides an in-depth, formal explanation of two primary methods available for computing the minimum value across one or several columns within a DataFrame, focusing on clear syntax and operational efficiency.

Understanding Aggregate Functions in PySpark

PySpark is the Python API for Apache Spark, designed for high-performance cluster computing. Central to its data manipulation capabilities are the built-in functions, often imported from the `pyspark.sql.functions` module. When we talk about finding the minimum value, we are employing an aggregate function--a function that summarizes the data in some way, typically returning a single result for an entire group (in this case, the entire column). Understanding the difference between using the `agg()` method and the `select()` method combined with the `min()` function is key to mastering data summaries in this environment. Both approaches achieve the same objective but differ significantly in syntax structure, result handling, and applicability when dealing with single versus multiple column operations.

The two methods detailed below leverage different functions and DataFrame API calls:

Method 1: Using `.agg(F.min()).` This method is generally preferred for calculating minimums (or other single-value summaries) and often works well when you need to calculate aggregates across multiple columns simultaneously using different functions or when you want the result to be easily extracted as a scalar value.

Method 2: Using `.select(min()).` This method is more versatile for projecting columns and applying transformations, including aggregates, and is highly readable when calculating the minimum for multiple columns and displaying the results as a new DataFrame row.

We will explore practical examples of both techniques using a representative PySpark DataFrame created specifically for demonstrating score aggregation across various teams. This structured approach ensures a comprehensive understanding of how to implement these critical data summary tasks efficiently.

Prerequisites: Setting up the PySpark Environment and Sample DataFrame

Before executing the minimum calculation methods, we must first establish a SparkSession and define a sample DataFrame. The SparkSession is the entry point to programming Spark with the DataFrame API, providing the necessary context for operations. The sample data represents

scores across three different games for several hypothetical teams. Establishing this context is mandatory for running any PySpark code examples and verifying the results accurately.

The following code snippet demonstrates the necessary setup, including the initialization of the Spark session, the definition of the data structure (a list of lists), the specification of column names, and the creation and subsequent display of the resulting DataFrame. Pay close attention to the column names, `game1`, `game2`, and `game3`, as these are the columns on which we will perform the aggregation operations to find the respective minimum values.

```
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, named `df`, provides the numerical columns necessary for our

aggregation tests. Analyzing this table visually confirms the range of values in each game column: `game1` ranges from 10 to 30, `game2` ranges from 8 to 22, and `game3` ranges from 10 to 35. This visual inspection helps validate the results obtained from the PySpark aggregate functions later in the process.

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

The most direct and standard way to calculate a single aggregate statistic across a DataFrame column is by utilizing the `.agg()` method, coupled with the built-in function `F.min()`. The `agg()` function is specifically designed to compute aggregate statistics over all rows or groups of rows (if grouping is applied). When used without a `groupBy()` operation, it calculates the aggregate across the entire column. We import `functions as F` to gain access to `F.min()`, which performs the actual minimum calculation.

For retrieving the minimum value from a single, specific column--in this example, `game1`--the syntax is concise and highly efficient. The `df.agg(F.min('column_name'))` call returns a new, single-row DataFrame containing the calculated minimum. Because the result is still a DataFrame, we must follow up with `.collect()` to extract the scalar minimum value directly into a Python variable, making it immediately usable for further calculations or reporting in the Python environment.

Here is the implementation of Method 1, calculating the minimum score exclusively for the `game1` column:

from pyspark.sql import functions as F

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

Analyzing the Single-Column Minimum Result

Executing the code above yields the scalar value corresponding to the minimum score found in the `game1` column. Understanding the steps involved in extracting this value is crucial for interpreting the result effectively. The `.agg()` operation generates a result DataFrame with one column (typically named `min(game1)`) and one row. The subsequent use of `.collect()` converts this Spark DataFrame into a list of Row objects in the local Python environment.

The indexing is used to navigate this structure: selects the first (and only) Row object from the list returned by `collect()`, and the second selects the value of the first (and only) column within that Row object, which is the calculated minimum value.

Using the sample data established earlier, the execution provides the following result:

```
from pyspark.sql import functions as F

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

```
10
```

The calculated minimum value for the `game1` column is **10**. This result aligns perfectly with a manual check of the data. The scores for `game1` are: 25, 22, 14, 30, 15, and 10. The lowest value in this set is indeed **10**, demonstrating the accuracy of the `.agg(F.min())` approach for single-column aggregation tasks.

Method 2: Calculating Minimum Values Across Multiple Columns

While the `.agg()` method is excellent for single-column retrieval, the `.select()` method combined with the `min()` function from `pyspark.sql.functions` offers a more straightforward and often preferred approach when calculating and displaying the minimum values across several columns simultaneously within a new result DataFrame. This method treats the aggregation calculation as a specific type of column projection.

Instead of calculating a single aggregate and then using `collect()` to extract it, we use `.select()` to create new columns, where each new column contains the minimum value of a corresponding input column. By supplying multiple `min(df.column_name)` arguments to the `.select()` function, we instruct Spark to compute the minimum for each column specified and return all results in a single-row DataFrame. This approach is highly readable and produces a nicely formatted table when using the `.show()` command.

To illustrate this method, we will calculate the minimum scores for `game1`, `game2`, and `game3` in one efficient operation:

```
from pyspark.sql.functions import min

#calculate minimum for game1, game2 and game3 columns
df.select(min(df.game1), min(df.game2), min(df.game3)).show()
```

Interpreting Multi-Column Minimum Output

The execution of the multi-column minimum calculation results in a new, distinct DataFrame containing one row and three columns, one for each aggregate computed. The column names are automatically generated by Spark (e.g., `min(game1)`), reflecting the function applied. This structure

is ideal for quick inspection and comparative analysis of the minimums across different variables.

The output from the `.show()` command clearly displays the minimum score for each of the three game columns:

from pyspark.sql.functions import min

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

```
+----------+----------+----------+
|min(game1)|min(game2)|min(game3)|
+----------+----------+----------+
|        10|         8|        10|
+----------+----------+----------+
```

This result set provides the following summary statistics:

The minimum value observed in the **game1** column is **10**.
The minimum value observed in the **game2** column is **8**.
The minimum value observed in the **game3** column is **10**.

This consolidated output ensures that data scientists can efficiently gather all necessary lower bounds in a single query, significantly streamlining the exploratory data analysis process within PySpark. This method is particularly useful when comparing the central tendencies or dispersion of multiple numerical variables.

Comparison of `agg()` vs. `select()` for Minimum Calculation

While both the `.agg()` and `.select()` methods successfully compute the minimum value, the choice between them often depends on the required output format and the necessity for subsequent operations. The `.agg()` method, especially when using `F.min()`, is structurally designed for general aggregate functions. Its strength lies in handling combinations of different aggregates (e.g., finding the minimum of one column and the average of another simultaneously) and working seamlessly after a `groupBy()` operation. Furthermore, because it often leads to extracting a scalar value using `.collect()`, it's suitable when the result is immediately needed for a Python variable.

In contrast, the `.select(min())` approach is fundamentally a column transformation that results in a new DataFrame. While it is highly intuitive for calculating the same aggregate across several columns at once and visualizing the output using `.show()`, extracting a scalar value requires the

same `.collect()` indexing overhead as the `.agg()` method. However, `.select()` is sometimes considered slightly less flexible than `.agg()` when complex, named aggregates are needed or when mixing aggregate types.

For general best practice: when finding the minimum of a single column for immediate use in Python, `df.agg(F.min(...)).collect()` is precise. When calculating the minimums for multiple columns to view in a structured output, `df.select(min(...)).show()` offers superior readability and efficiency within the Spark environment. Both methods are computationally efficient as they leverage Spark's underlying distributed computing capabilities to perform the necessary computations in parallel across the cluster nodes, ensuring optimal performance even with multi-terabyte datasets.

Conclusion: Summary of PySpark Minimum Calculation Techniques

Mastering PySpark requires proficiency in various aggregate functions, and calculating the minimum value is a foundational skill. We have demonstrated two robust and efficient methods for achieving this task. Whether you need to isolate a single minimum value using the focused power of `.agg(F.min())` or simultaneously compute the lower bounds of multiple variables using the versatile `.select(min())`, PySpark provides clear, performance-optimized tools for data summarization.

The key takeaway is that both methods are valid and efficient, but their contextual use depends on the desired output format--scalar value for Python use versus a multi-column result DataFrame for comparative analysis. Always ensure that the necessary functions are correctly imported from `pyspark.sql.functions`, and remember that when extracting a scalar result from an aggregate operation, the use of `.collect()` is essential to transition the data from the distributed Spark execution context back to the local Python environment for reporting.

By integrating these powerful aggregation techniques into your data processing workflow, you can ensure accurate and timely identification of minimum values across vast DataFrames, facilitating thorough data quality checks and robust statistical reporting.