How to Find the Minimum Value in a PySpark Column

Authored by

stats writer

February 8, 2026

Calculating statistical measures like the minimum value is a fundamental operation when analyzing large datasets using distributed computing frameworks. When working with PySpark, the Python API for Apache Spark, efficient methods are available to determine the lowest observation within a specific column of a DataFrame. These methods leverage Spark's optimized execution engine, ensuring performance even with petabyte-scale data.

This guide explores the two primary and most efficient techniques available in PySpark for achieving this task: using the highly versatile **agg function** in combination with the built-in **min function**, and employing the **select function** for aggregating results across one or more columns simultaneously. Understanding both approaches allows data engineers and analysts to choose the most appropriate syntax based on their specific analytical needs and desired output format.

The standard approach involves using the DataFrame's `agg()` transformation. This method facilitates aggregate calculations over the entire dataset or specific partitions, returning a new DataFrame containing the resulting minimum value. For quick retrieval, especially when dealing with a single minimum value, integrating the `collect()` action is often necessary to pull the result back into the local Python environment. We will detail the implementation of both methods, ensuring clarity on syntax, performance considerations, and output interpretation, all within the context of robust data processing.

Calculating the Minimum Value of a Column within a PySpark DataFrame

Core Functions for Minimum Value Calculation

To effectively retrieve the minimum numerical value from a specified column within a DataFrame, PySpark provides two robust methodologies. Both rely heavily on the functions available within the `pyspark.sql.functions` module, which must typically be imported prior to use. It is crucial to understand that while both methods yield the correct minimum value, they differ in syntax complexity, output format, and suitability for single versus multiple column aggregations.

The first method leverages the `agg()` transformation, which is ideal when you need to perform one or more aggregate operations and immediately retrieve the scalar result using an action like `collect()`. The second method utilizes the `select()` transformation, which is often preferred when performing aggregates on multiple columns simultaneously and presenting the results neatly in a new DataFrame row. We will analyze the mechanics of each method in detail using practical code examples.

Method 1: Calculating Minimum for a Single Specific Column using agg()

The most idiomatic way to calculate a single aggregate statistic like the minimum is through the agg function. This function takes dictionary arguments or column expressions specifying the aggregation to be performed. Since `agg()` returns a new DataFrame, we must subsequently use the `collect()` action to extract the scalar minimum value from the distributed Spark environment back to the local Python application.

We achieve this by importing the necessary functions, often aliased as `F` for convenience, and applying `F.min()` to the target column name. The resulting structure, `df.agg(F.min('column_name')).collect()`, is highly efficient for single value retrieval, minimizing the necessity for complex DataFrame restructuring if only the final number is needed.

The syntax for this approach is demonstrated below:

from pyspark.sql import functions as F

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

Method 2: Calculating Minimum for Multiple Columns using select()

While `agg()` is powerful, calculating minimums across multiple distinct columns simultaneously often becomes cleaner and more manageable using the **select function** combined with the **min function**. Unlike standard `select()` operations that fetch existing columns, when an aggregate function like `min()` is applied within `select()`, it treats the entire column's data as a single group, yielding the aggregated result for that column.

This method is particularly useful when the desired output is a new DataFrame row showing the minimums side-by-side for easy comparison, which can then be visualized using the `show()` action. This avoids the necessity of using `collect()` and manually accessing array indices, simplifying the output presentation for interactive work or for passing the result to another Spark transformation step.

The necessary syntax, which requires importing the `min` function directly, is shown below:

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()

Prerequisites: Defining the PySpark DataFrame

Before executing the calculation methods, we must first define and initialize a sample DataFrame. This process involves creating a SparkSession--the entry point for all Spark functionality--and defining both the data structure (rows) and the schema (column names). Our example utilizes basketball team scores across three hypothetical games to illustrate minimum value identification across multiple numerical fields.

The creation process ensures that the data is correctly distributed across the Spark cluster, making it ready for high-performance aggregate transformations. Note that the numerical columns `game1`, `game2`, and `game3` are the primary targets for our minimum value calculations, while the `team` column serves as the categorical identifier.

The following code block demonstrates the necessary setup steps to define our example DataFrame and display its initial contents, providing the context necessary for the upcoming examples:

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 Example 1: Calculating Minimum for a Single Column using agg()

The `agg()` method is specifically optimized for retrieving a single aggregate result efficiently. In this scenario, we aim to find the absolute minimum score recorded in the `game1` column. We utilize the function `F.min('game1')` within the `agg()` transformation call. This generates an intermediate DataFrame with one row and a computed column representing the minimum value.

To retrieve the actual number into our local Python environment, we must chain the `collect()` action. Since `collect()` returns a list of `Row` objects (which are indexed like tuples), we access the first element of the list (index) and then the first element of that Row object (index) to successfully extract the scalar numerical value. This pattern is standard for single aggregate extraction in PySpark.

Applying this technique to the `game1` column demonstrates the efficient retrieval of the minimum value across all team entries:

from pyspark.sql import functions as F

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

10

The output clearly indicates that the minimum value observed in the `game1` column is **10**. This result is verified by manually inspecting the dataset. The scores for game 1 are 25, 22, 14, 30, 15, and 10.

The usage of `collect()` and subsequent indexing emphasizes the transition from distributed computation (Spark) back to local processing (Python). This step is necessary whenever a final, single numeric result is required outside of the Spark environment itself.

Detailed Example 2: Calculating Minimum for Multiple Columns using select()

When the requirement shifts to simultaneously calculating minimums across several columns--

specifically `game1`, `game2`, and `game3`--the **select function** provides a highly organized and readable output format. By passing multiple `min()` expressions to `select()`, Spark computes each minimum independently across the respective column and returns a consolidated result in a new, single-row DataFrame.

This method is generally preferred by analysts who need a comparative summary of aggregate statistics displayed side-by-side, avoiding the need for manual scalar extraction. The resulting DataFrame clearly labels the minimum output for each column using the default column names assigned by the **min function**, which are prefixed with `min(...)`.

Executing the calculation for all three game scores using this method is demonstrated below:

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

The resulting output DataFrame provides an immediate summary of the lowest recorded score for each category. Based on the output:

The minimum of values in the **game1** column is **10**.
The minimum of values in the **game2** column is **8**.
The minimum of values in the **game3** column is **10**.

Comparative Analysis of Aggregation Techniques in PySpark

Choosing between the `agg()` and `select()` methods depends largely on the intended outcome and subsequent steps in the data pipeline. While both execute the minimum calculation efficiently across the cluster, their utility differs based on whether the goal is scalar extraction or structured DataFrame output.

The `agg()` method, particularly when utilized with `collect()`, is often the most performant and direct approach when only a single scalar result is required for immediate use in local Python code. It is less verbose if the final goal is merely a numeric variable.

Conversely, using aggregate functions within the **select function** excels in generating summary tables containing multiple aggregates. If an analyst requires minimum, maximum, average, and count all at once for several columns, using `select()` provides a clear, tabular output which is easier to inspect interactively or save as a new summary DataFrame. This method keeps the result within the Spark environment, which is beneficial for chaining further Spark operations.

Conclusion

Mastering these two aggregation functions is essential for effective data analysis in the PySpark ecosystem. Both the **agg function** and the **select function**, when combined with the `min()` function, offer flexible and high-performing tools to identify the minimum values within large, distributed datasets managed by Apache Spark. By selecting the method that aligns best with the output requirement--scalar result versus summary table--data practitioners can maximize efficiency and code clarity.

The following tutorials explain how to perform other common tasks in PySpark: