How to Implement a “Group By Having” Clause in PySpark for Data Analysis

Authored by

stats writer

February 3, 2026

PySpark stands as an immensely powerful engine for large-scale data processing and analysis, providing users with the capability to execute complex data manipulation tasks, often mirroring the familiar operations found in standard **SQL**. One of the most essential functionalities required for detailed analytical workflows is the ability to aggregate data and subsequently filter those aggregated groups based on specific criteria. In traditional **SQL**, this mechanism is encapsulated by the **Group By Having** clause, a cornerstone technique for refined data segmentation and filtering.

Implementing the **Group By Having** logic within the PySpark environment is crucial for data scientists and analysts dealing with distributed datasets. While **SQL** provides a concise syntax for this operation, PySpark achieves the same result through a chain of dedicated functions applied to a **DataFrame**. This methodology involves using the native `groupBy` function to establish the groups, followed by aggregation functions (like `count`, `sum`, or `avg`), and finally employing the `filter` function to impose the conditional restrictions on the resulting groups.

This approach is significantly more efficient than attempting manual filtering post-aggregation, especially when handling vast quantities of data distributed across a cluster. By leveraging the optimized execution plan of the Spark engine, analysts can quickly identify patterns, outliers, or specific cohorts within their data. Mastering the combination of `groupBy`, aggregation, and `filter` ensures a comprehensive and performant method for sophisticated data analysis, leading directly to more robust, data-driven insights and decision-making processes.

Understanding the Core SQL Group By Having Paradigm

Before diving into the PySpark implementation, it is vital to clearly define the purpose of the **Group By Having** concept as it is utilized in relational databases. The `GROUP BY` clause is used to group rows that have the same values in specified columns into summary rows. This is always paired with aggregate functions that calculate a single summary value for each group. For instance, you might group sales data by product category and calculate the total sales for each category using a function like `SUM()`.

Crucially, the `HAVING` clause steps in where the standard `WHERE` clause cannot function. The `WHERE` clause filters individual rows before they are grouped. Conversely, the `HAVING` clause filters the resulting groups themselves, applying a condition to the results of the aggregate functions. If you wanted to filter out individual sales records based on a price threshold, you would use `WHERE`. However, if you want to only see product categories where the calculated total sales (the aggregate result) exceeds one million dollars, you must use `HAVING`.

In essence, the sequence of operations defined by the **SQL** standard is: 1) Data selection, 2) Filtering of individual rows (WHERE), 3) Grouping of remaining rows (GROUP BY), 4) Calculation

of aggregate statistics, and 5) Filtering of those resulting groups (HAVING). PySpark translates this five-step process into a functional programming pipeline, ensuring the operation occurs in the correct logical sequence, which is paramount for correctness in complex data transformations.

The PySpark Approach: Combining groupBy, agg, and filter

While traditional **SQL** uses the dedicated `HAVING` keyword, PySpark achieves the **Group By Having** functionality by combining three distinct methods applied sequentially to the **DataFrame**. These methods are `groupBy()`, `agg()` (or specific functions like `count()` or `avg()`), and `filter()`. This chain of commands ensures that the filtering condition is applied only after the grouping and aggregation steps have been successfully computed across the distributed dataset, respecting the logical flow of the original SQL paradigm.

The first step, invoking `df.groupBy('column_name')`, is equivalent to the SQL `GROUP BY` clause. It sets the stage for aggregation by logically partitioning the **DataFrame** rows based on the unique values in the specified column. Crucially, this operation does not immediately produce a result set; instead, it returns a `GroupedData` object, which expects an aggregate function to be called next to summarize the partitions created.

The second step is the aggregation, typically using the `agg()` function. This function allows for the calculation of one or more aggregate statistics (like count, mean, max, or min) on the grouped data. It is within this step that the new aggregated column, which will be the basis for the filtering condition, is created. Finally, the third step utilizes the `filter()` function. Unlike using `filter()` immediately on a standard DataFrame (which acts like a SQL `WHERE` clause), when applied directly after aggregation, `filter()` acts exactly like the SQL `HAVING` clause, allowing conditions to be placed upon the newly created aggregate column, thus completing the desired filtering logic.

Syntax for Implementing Grouped Filtering in PySpark

The core syntax for performing the equivalent of a **Group By Having** statement in PySpark involves importing the necessary functions from `pyspark.sql.functions` and then chaining the three primary operations: grouping, aggregating, and filtering. It is a streamlined, declarative pipeline that maximizes performance by letting Spark manage the underlying execution details through its Catalyst optimizer.

The structure begins by defining the initial **DataFrame**, followed immediately by the `groupBy()` operation specifying the column to group by. The subsequent `agg()` function calculates the aggregate metric (e.g., counting the rows in each group) and aliases the new column for clarity and ease of reference. Finally, the `filter()` function applies the conditional logic using the `col()` function to reference the newly aliased aggregate column, which is essential for applying criteria to

the aggregate results.

Consider the scenario where we want to isolate groups that have more than two occurrences in a specific category column, such as **team**. The implementation is precise and readable, clearly defining the steps required to transition from the raw data to the filtered group results. The resulting DataFrame will only contain the group keys and their corresponding aggregate counts that satisfy the provided condition.

You can use the following syntax to perform the equivalent of a **SQL** 'GROUP BY HAVING' statement in PySpark:

from pyspark.sql.functions import *

```
# create new DataFrame that only contains rows where team count>2
df_new = df.groupBy('team')
.agg(count('team').alias('n'))
.filter(col('n')>2)
```

This particular example finds the count of each unique value in the **team** column and then filters the **DataFrame** to only contain rows where the calculated count, aliased as **n**, is greater than 2. This pattern is foundational for filtering based on group frequency.

Practical Scenario: Setting up the Sample Data

To fully illustrate the functionality and syntax of the PySpark Group By Having equivalent, we will use a practical example based on hypothetical sports statistics. Suppose we are tracking points scored by various basketball players assigned to different teams. Our goal is to analyze team performance, but first, we need a clean dataset to apply our grouping logic.

The initial step involves setting up a **SparkSession**, which is the necessary entry point to using Spark functionality, and then defining the raw data. This raw data is structured as a list of lists, where each inner list represents a record containing the team identifier and the points scored by a player on that team. Defining clear column names (**team** and **points**) is necessary before creating the **DataFrame**, ensuring that the schema is explicit and usable for subsequent analytical operations.

By viewing the resulting **DataFrame** using the `df.show()` command, we can confirm the structure and contents of our dataset, which includes teams A, B, C, and D, each with varying numbers of entries and points accumulated. This preparatory step ensures that all subsequent analytical operations are performed on a known, standardized structure.

Suppose we have the following PySpark DataFrame that contains information about points scored by various basketball players:

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|points|
+----+------+
A	11
A	8
A	10
B	6
B	6
C	5
C	15
C	31
D	24
+----+------+

Case Study 1: Filtering Groups Based on Record Count

Our first case study focuses on filtering groups based on the frequency of their records. In a real-world scenario, we might want to discard teams (groups) that have fewer than three recorded player entries, as a small sample size could lead to unreliable statistical conclusions. This requirement precisely mirrors the need for a **Group By Having** operation, as the condition applies to the aggregate count of the group, not the individual records.

To execute this, we start by applying `groupBy('team')`, which logically segregates the data by the unique team identifiers. We then use `agg(count('team').alias('n'))` to calculate how many players belong to each team and store this count in a new column named **n**. This new aggregated DataFrame now contains only two columns: **team** (the group key) and **n** (the count).

The final and critical step is applying `filter(col('n') > 2)`. This instructs Spark to discard any rows (which represent the aggregated teams) where the calculated count **n** is not greater than 2. This chain effectively implements the conditional filtering on the aggregate result, demonstrating the power of the sequential PySpark operations.

We can use the following syntax to filter the DataFrame for rows where the count of values in the **team** column is greater than 2:

from pyspark.sql.functions import *

```
#create new DataFrame that only contains rows where team count>2
df_new = df.groupBy('team')
.agg(count('team').alias('n'))
.filter(col('n')>2)

#view new DataFrame
df_new.show()
```

```
+----+---+
|team| n|
+----+---+
| A| 3|
| C| 3|
+----+---+
```

Notice that each of the rows in the filtered DataFrame have a **team** count greater than 2. Teams B (count 2) and D (count 1) were correctly excluded because their aggregated counts did not meet the condition specified in the `filter()` function.

Case Study 2: Filtering Groups Based on Aggregated Metrics (Average Points)

The flexibility of the PySpark approach extends beyond simple counts. We can apply the same three-step methodology--group, aggregate, filter--to any numerical aggregate metric, such as sums, means, maximums, or standard deviations. For this second case study, let us analyze team performance by focusing on the average points scored per team and filtering based on that average.

The objective here is to identify and retain only those teams whose average points per player exceed a specific benchmark, set at 10 points. This requires calculating the mean points value for each team and then filtering the aggregated results. The implementation syntax remains structurally identical to the previous example, but the aggregate function is swapped from `count` to `avg` to calculate the mean value of the **points** column.

We initialize the operation with `df.groupBy('team')`, ensuring points are aggregated per team. We then use `agg(avg('points').alias('avg'))` to compute the arithmetic mean of the **points** column for every group, aliasing the new column as **avg**. Finally, we apply the conditional filter: `filter(col('avg') > 10)`. This successfully eliminates teams whose average scores fall at or below the threshold, providing a focused view on high-performing groups.

For example, we can use the following syntax to calculate the average **points** value for each team and then filter the DataFrame to only contain rows where the average points value is greater than 10:

from pyspark.sql.functions import *

```
#create new DataFrame that only contains rows where points avg>10
df_new = df.groupBy('team')
.agg(avg('points').alias('avg'))
.filter(col('avg')>10)

#view new DataFrame
df_new.show()

+----+----+
|team| avg|
+----+----+
| C|17.0|
| D|24.0|
+----+----+
```

Notice that the resulting DataFrame only contains rows for the teams with an average points value greater than 10. Team A (average 9.67) and Team B (average 6.0) are correctly excluded, highlighting the accuracy and utility of filtering based on calculated group metrics.

Conclusion and Resources for Further Learning

The implementation of **Group By Having** logic in PySpark, achieved through the sequential use of `groupBy()`, `agg()`, and `filter()`, provides analysts with a highly efficient and flexible mechanism for filtering grouped data. This pattern is fundamental to complex data warehousing and analytical pipelines, allowing for the rapid transformation and reduction of massive distributed datasets based on derived statistical metrics.

It is important to remember the critical distinction between `filter()` used on a standard DataFrame (acting as `WHERE` on individual rows) and `filter()` used immediately after an aggregation operation (acting as `HAVING` on grouped results). Understanding this subtle but critical difference ensures that performance is optimized and analytical results are logically sound, adhering to the core principles of data analysis.

For those seeking to expand their knowledge of PySpark's rich set of capabilities, exploring the full documentation for the various functions available, particularly those within `pyspark.sql.functions`, is highly recommended. These tools enable a vast array of transformations, ranging from simple column manipulations to complex window functions and machine learning preparations, offering boundless opportunities for advanced data manipulation.

Note: You can find the complete documentation for the PySpark **filter** function here.

Related PySpark Tutorials

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

How to perform various Join operations in PySpark using different join types (e.g., inner, left, right).
Methods for optimizing large-scale data processing workflows using caching and persistence strategies.
Applying advanced statistical functions and Window functions to PySpark DataFrames for complex cumulative calculations.