Learning PySpark: Implementing SQL GROUP BY with HAVING Functionality

Authored by

Mohammed looti

November 11, 2025

Emulating the SQL HAVING Clause in PySpark

The ability to conditionally filter results following an aggregation is a fundamental requirement in advanced data manipulation, a feature traditionally handled by the **HAVING** clause in Structured Query Language (SQL). This powerful clause allows analysts to narrow down groups based on the values calculated during the aggregation step (such as sums, averages, or counts). However, data engineers transitioning from standard SQL environments to the PySpark API quickly discover that it lacks a direct `.having()` method. This absence necessitates a different, yet equally powerful, approach within the **Apache Spark ecosystem**.

Instead of a single dedicated function, PySpark mandates that the equivalent operation be constructed by chaining three core transformations from the **PySpark DataFrame** API. This technique involves sequentially performing the grouping, executing the aggregation, and then applying a filtering step against the newly calculated aggregate column. Understanding and mastering this specific three-step pattern--`groupBy()`, `agg()`, and `filter()`--is absolutely critical for anyone performing complex analytical tasks at scale within the distributed computing environment of Spark, ensuring both correctness and efficiency.

The critical distinction here lies in the timing of the filtering operation. In SQL, the **WHERE** clause filters individual rows *before* grouping, while **HAVING** filters groups *after* aggregation. To mimic this behavior precisely in PySpark, the filtering function, `.filter()`, must operate on the intermediate DataFrame created by the aggregation step. This ensures that the criteria are applied only to the metrics derived from the groups, successfully replicating the logic of the **GROUP BY HAVING** statement. The following sections will provide a detailed breakdown of the required syntax and illustrate its application through practical, real-world examples.

The PySpark Syntax Chain for Group By Having

To successfully execute the equivalent of a **SQL 'GROUP BY HAVING'** statement in PySpark, data engineers must utilize a precise, standardized three-step syntax chain. This sequence is designed to uphold the operational definition of the **HAVING** clause: completing all calculations for the defined groups before applying any filtering conditions based on those results. Any deviation from this ordering will lead to incorrect or inefficient execution, highlighting the importance of adhering to this specific formula.

The core components of this formula function as follows: First, the `.groupBy()` transformation segments the input data based on categorical columns, defining the boundaries for aggregation. Second, the `.agg()` method calculates the desired aggregate metrics (such as sums, counts, or averages) for each segment, simultaneously creating a new column to store this result, often aliased for clarity. Finally, the `.filter()` function evaluates a condition against the values in this

newly created aggregate column, retaining only those groups that satisfy the predefined criteria.

Consider a practical scenario: identifying data groups (e.g., specific departments or teams) that contribute more than a threshold number of records to the dataset. The PySpark implementation requires importing necessary functions, particularly column manipulation functions like `col` and aggregation functions like count. The subsequent code block demonstrates the general structure required to group the data by a column (`team`), calculate the count of records within that group (aliased as `n`), and then apply a conditional filter based on the calculated value of `n`.

The syntax showcased below illustrates how to construct a new DataFrame that selectively retains only those groups where the total count of the grouped column is greater than two. This pattern is exceptionally reusable and adaptable: regardless of the complexity of the aggregation--whether calculating minimums, maximums, or averages--the only necessary modification is substituting the specific aggregation function (like `count`) within the `.agg()` step. The subsequent `.filter()` step remains the mechanism for applying the **HAVING** logic.

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 code snippet executes a precise sequential flow of logic. Initially, it calculates the frequency of each unique value present in the **team** column. Following this, it assigns the alias `n` to this newly calculated frequency column using `.alias('n')`. The most crucial step is the third one, where it filters the resulting aggregated DataFrame, keeping only those rows where the count column, `n`, strictly exceeds the value of 2. This robust, multi-step process is the prescribed and most efficient method for achieving the advanced functionality of GROUP BY HAVING within the PySpark computing environment.

Setting Up the Environment and Sample Data

To effectively illustrate the application of this essential PySpark formula, we will utilize a sample dataset. Our hypothetical scenario involves a PySpark DataFrame that meticulously tracks the performance data--specifically, points scored--across various basketball teams (A, B, C, and D). Our primary objective in this demonstration is to identify which teams meet a specific minimum participation or size requirement, which we will define as having a count of entries greater than two.

The initial, foundational step requires the initialization of the Spark session. This provides the

necessary environment for distributed computation. Once the session is active, we proceed to construct the example DataFrame, ensuring we have the requisite structure and data context to perform our planned aggregation and subsequent filtering operations. The data array below contains nine entries, representing different games or players for Teams A, B, C, and D, providing a perfect microcosm for demonstrating the **GROUP BY HAVING** equivalent in a practical setting.

This preparation step is vital for ensuring reproducibility and clarity. By clearly defining the data and column names (`team` and `points`) before applying the transformations, we establish a clean baseline. The `df.show()` command, as shown in the output below, confirms the successful creation and initial state of our source DataFrame, ready for the three-step PySpark analysis chain.

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
+----+------+

Practical Example 1: Filtering Based on Group Count

With the DataFrame successfully initialized, we now apply the crucial three-step syntax to execute the conditional filtering. This operation begins by grouping all rows based on the unique values found in the **team** column. This segmentation is the preparatory phase for the aggregation. The second step utilizes the `.agg()` method, incorporating the count function to precisely determine the total number of entries recorded for each specific team. This count is then aliased as `n`, making it accessible for the final stage.

The final and most defining step involves the `.filter()` function. This function operates on the newly created column `n`, applying the condition that the count must be strictly greater than 2 (`n > 2`). By implementing this filter immediately after the aggregation, we successfully isolate only those teams that have met our predefined minimum participation threshold. Teams with two or fewer entries are automatically discarded from the resulting dataset, fully mimicking the behavior of the **HAVING** clause in standard SQL queries.

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|
+----+---+

The resulting DataFrame, named `df_new`, clearly confirms the success of the operation. Only Team A and Team C are retained in the final output, as they are the sole teams with a recorded count of exactly 3 entries. Teams B (count 2) and D (count 1) are excluded because they failed to satisfy the condition specified in the `.filter()` step. This exercise successfully validates the implementation of the **GROUP BY HAVING** equivalent in PySpark by filtering based on the results of a calculated aggregate function.

Practical Example 2: Filtering Based on Aggregated Metrics (Average)

The true utility of the GROUP BY HAVING pattern within PySpark is its versatility, extending significantly beyond simple counts. This robust methodology can be applied to nearly any mathematical aggregation metric, including sums, minimums, maximums, and, as demonstrated here, averages. Instead of focusing on the sheer volume of records per team, we now shift our analytical objective to filtering teams based on their average performance score.

In this alternative scenario, our goal is to precisely identify all teams whose calculated average **points** exceed a predefined performance threshold of 10. We maintain the same core three-step structure: first, we use `.groupBy()` on the team column; second, we deploy `.agg()` to calculate the average points using the dedicated `avg` function; and third, we apply `.filter()` using the `col` function to evaluate the condition against the new average column. This process requires ensuring the `avg` function is correctly imported, although using `*` as shown in the code snippet is a common practice for convenience.

The code below systematically calculates the average **points** for every unique **team**, assigning the clear alias `avg` to the resulting column. Following this calculation, the chained `.filter()` operation evaluates this `avg` column, ensuring that only those teams whose computed average score is strictly greater than 10 are retained for the final DataFrame. This demonstrates the seamless operational flexibility inherent in this PySpark approach to complex, aggregated data analysis, proving that the pattern is metric-agnostic.

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

Following execution, the resulting DataFrame, `df_new`, contains only Team C (which had an average of 17.0) and Team D (average 24.0). Significantly, Team A (average 9.67) and Team B (average 6.0) are excluded from this result because their aggregated metrics failed to meet the specified filtering criterion. This outcome decisively confirms that the chained `groupBy`, `agg`, and `filter` functions successfully and precisely replicate the analytical constraints of the **HAVING** clause in SQL, regardless of the specific mathematical aggregation function employed.

Mastering the Three-Step PySpark Pattern

In conclusion, while the PySpark API does not offer a dedicated, high-level `.having()` method, this limitation is effectively bypassed by leveraging the combination of `.groupBy()`, `.agg()`, and `.filter()` functions. This three-step formula constitutes the robust, scalable, and authoritative mechanism for filtering aggregated data results within the PySpark environment. It is the essential pattern that must be mastered for any analytical task requiring conditional selection based on derived group metrics, cementing its importance in the **PySpark DataFrame** API toolkit.

A key conceptual point to reinforce is the strict order of operations: grouping must precede aggregation, and filtering must follow aggregation. The `.filter()` function in this context is uniquely positioned to operate on column values that are generated in the preceding `.agg()` step. Unlike filtering rows before grouping (which is typically done with `.where()` or `.filter()` before `.groupBy()`), this specific pattern targets the group metrics themselves. For comprehensive details regarding the precise usage and parameter specifications of the **filter** function and associated aggregation utilities within PySpark, data practitioners are strongly advised to consult the official Apache Spark documentation to ensure adherence to best practices and optimal performance tuning.

Additional Resources

To further deepen your expertise in PySpark and related distributed data processing tasks, the following resources are recommended for exploring other common data manipulation techniques and functions:

How to perform joins between two DataFrames in PySpark.
Techniques for handling missing values (NaN, null) within a PySpark DataFrame.

Using window functions for complex calculations over defined partitions.