How to Filter Grouped Data in PySpark Using `groupBy`, `agg`, and `filter`

Authored by

stats writer

January 18, 2026

The Need for Conditional Aggregation in PySpark

When working with large datasets, effective data aggregation is essential for deriving meaningful insights. In traditional SQL environments, the `GROUP BY` clause is used to collect identical data into groups, and the `HAVING` clause is then applied to filter those resulting groups based on aggregate functions (like sums, counts, or averages). Unlike the `WHERE` clause, which filters individual rows before aggregation, `HAVING` acts on the calculated group statistics.

Translating this sophisticated workflow into the functional paradigm of PySpark requires chaining together several powerful DataFrame transformation methods. Since PySpark DataFrames are immutable collections distributed across a cluster, operations must be performed sequentially, often resulting in a more explicit, step-by-step definition of the desired outcome compared to a single declarative SQL statement.

For data engineers accustomed to SQL syntax, finding the exact equivalent of `GROUP BY HAVING` in the PySpark API can sometimes be challenging. The core concept remains the same: first aggregate the data, then filter the aggregated results. PySpark achieves this using a combination of the `groupBy()`, `agg()`, and `filter()` methods, chained together seamlessly to execute the conditional group filtering operation efficiently.

Understanding the PySpark Equivalent of SQL's HAVING Clause

In PySpark, there is no distinct function named `having()`. Instead, the functionality is achieved by utilizing the `filter()` method immediately after the aggregation step. This method allows you to apply a condition based on the newly created aggregate column, effectively replicating the behavior of `HAVING`.

To successfully implement this technique, the aggregation step must be performed first. Once the data is grouped using `groupBy()`, the `agg()` function is used to calculate the necessary metrics--such as the count, average, or maximum value--and assign these metrics to a new column using the `alias()` method. This new column, containing the aggregate metric, then becomes the target for the subsequent filtering operation.

This three-step process--grouping, aggregating, and filtering--ensures that the conditions are applied only to the summary statistics derived from the groups, not to the original individual records. This structure is a cornerstone of advanced data manipulation in distributed computing frameworks like Apache Spark, prioritizing clear transformations and optimized execution plans.

The Core Syntax: Implementing GROUP BY and HAVING

The following syntax provides the blueprint for executing the equivalent of a SQL `GROUP BY`

`HAVING` statement in PySpark. This formula is versatile and can be adapted to filter based on any aggregate function, such as sums, standard deviations, or minimums.

The key components involved are importing necessary functions from `pyspark.sql.functions`, applying the grouping, calculating the aggregate, and finally applying the condition using `.filter()`. The example below demonstrates finding groups where the count of occurrences exceeds a threshold of 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)
```

In this specific implementation, we first group the original DataFrame (`df`) by the `team` column. We then calculate the `count` of items within each team group, labeling this new aggregation column as `n` using `alias('n')`. The subsequent `filter()` operation evaluates the value of `n` for each resulting group and retains only those groups where the count is strictly greater than 2. This sequence is powerful because it allows conditional selection based on derived characteristics of the data groups.

Practical Example: Setting up the PySpark Environment and Sample Data

To illustrate the `GROUP BY HAVING` equivalent in a practical context, let us consider a sample dataset containing basketball statistics. This dataset tracks the points scored by various players assigned to different teams (A, B, C, D). Our objective will be to determine which teams satisfy certain group-level criteria.

Before executing any transformations, we must initialize a PySpark `SparkSession`, which serves as the entry point for utilizing Spark functionality. Following the session creation, we define the raw data list and the corresponding column names, which are then used to construct our initial PySpark DataFrame.

The following code snippet demonstrates the setup, resulting in a DataFrame that clearly shows the distribution of points across the four teams defined in the data:

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


As visible in the output, Team A has three entries, Team B has two, Team C has three, and Team D has one. This initial distribution is critical for the next example, where we will filter the teams based on the frequency of their representation in the dataset.

Filtering Groups Based on Count Thresholds

A common business requirement is to only analyze groups (or teams) that have a sufficient

number of observations. For instance, we might only trust statistical summaries for teams with three or more players. Using the `groupBy()` and `count()` functions, we can calculate the size of each group and then apply a filter based on that size.

The following code applies the standard `GROUP BY HAVING` formula to our sample DataFrame, filtering for teams that appear more than twice (i.e., team count greater than 2). This demonstrates the precise mapping of conditional aggregation logic from SQL environments into PySpark's functional syntax.

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 `df_new` contains only Team A and Team C, both of which have three entries, satisfying the condition `n > 2`. Teams B and D, which had counts of 2 and 1 respectively, are excluded. This process confirms that the filter is successfully applied to the aggregate result, rather than the initial rows, achieving the exact behavior of the SQL `HAVING` clause.

Applying HAVING Logic Based on Calculated Averages

The power of this PySpark pattern extends far beyond simple counts. Any aggregate function can be used to generate the metric upon which the final groups are filtered. A slightly more complex scenario involves calculating a mean or average score for each team and then filtering out teams whose average performance falls below a certain standard.

Suppose we wish to identify only the high-performing teams--those where the average `points` scored is greater than 10. We replace the `count()` function in the previous example with the `avg()` function, then apply the corresponding condition to the resulting average column.

The code below executes this calculation, aliasing the average points as `avg` and then filtering the grouped results based on that new metric. This illustrates the flexibility and modularity inherent in chaining PySpark transformations:

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

Upon examining the original data and the resulting DataFrame, we can verify the outcome. Team C (average of (5+15+31)/3 = 17.0) and Team D (average of 24/1 = 24.0) both meet the criterion of having an average greater than 10. Team A (average of (11+8+10)/3 = 9.67) and Team B (average of (6+6)/2 = 6.0) are correctly excluded from the final filtered dataset. This exemplifies how conditional filtering based on complex metrics can be executed efficiently in Spark.

Detailed Breakdown of the PySpark Functions Used

A deep understanding of the functions utilized in the `GROUP BY HAVING` emulation is essential for mastering PySpark. This process relies on chaining three fundamental DataFrame methods in succession, each serving a distinct, critical purpose in the data pipeline.

The first step, `df.groupBy('column')`, prepares the DataFrame for aggregation. It returns a `GroupedData` object, which is not a DataFrame itself but rather an intermediary structure awaiting an aggregation function. This operation defines the partition key for the data, grouping rows with identical values in the specified column.

Next, `.agg(function('column').alias('new_name'))` performs the actual calculation. The `agg()` method is highly flexible, accepting multiple aggregate expressions. Using `.alias('new_name')` is crucial here, as it labels the newly calculated aggregate column (e.g., `n` or

`avg`), making it accessible for the final filtering step. Failure to alias the column would make it difficult to reference in the subsequent condition.

Finally, the `.filter(col('new_name') > condition)` method applies the condition to the resulting DataFrame. Unlike the SQL `WHERE` clause, which would be applied before aggregation, the `filter()` method here is applied after `agg()`, allowing it to evaluate conditions based on the group-level metrics. It requires the use of the `col()` function, imported from `pyspark.sql.functions`, to explicitly reference the newly created column by its alias.

Conclusion: Leveraging Chained Transformations for Advanced Filtering

The PySpark implementation of SQL's `GROUP BY HAVING` clause showcases the power and expressiveness of the DataFrame API. By chaining the `groupBy()`, `agg()`, and `filter()` methods, developers can execute complex conditional aggregation logic efficiently on distributed data. This technique is indispensable for scenarios requiring statistical validation or the identification of groups that meet specific performance criteria.

Mastering this sequence--grouping the data, generating an aggregate metric, and then filtering based on that metric--is a fundamental skill for any data professional working with Apache Spark. It provides a robust, scalable, and clear method for deriving actionable insights from voluminous datasets, ensuring that analysis focuses only on groups that satisfy predefined criteria.

For comprehensive details on the specific functions used in these transformations, users are encouraged to consult the official documentation. You can find the complete documentation for the PySpark filter function, which is central to this technique, and explore other advanced DataFrame operations to enhance your data processing capabilities.