Use “AND” Operator in PySpark (With Examples)

Authored by

stats writer

November 16, 2025

Data manipulation is the core of any analytical project, and the ability to selectively retrieve data based on multiple criteria is essential. In the realm of big data processing using Apache Spark, the Python API, known as PySpark, provides robust mechanisms for conditional filtering. When faced with the requirement to satisfy two or more conditions simultaneously, we rely on the logical conjunction, commonly known as the **AND operator**. Understanding how to correctly implement this operator within PySpark is crucial for efficient data processing and transformation. This guide explores the two standard and highly effective methods available for applying the **AND operator** to filter a PySpark DataFrame.

Understanding Conjunctions in PySpark

In PySpark, filtering operations allow users to select a subset of rows that satisfy specific criteria. When dealing with complex business logic, these criteria often require multiple conditions to be true simultaneously--a scenario perfectly addressed by the **AND operator** (or conjunction). PySpark offers two distinct syntaxes for implementing this logic, catering to different coding preferences and performance requirements. The first approach utilizes a simple SQL-like string expression, which is often favored for its readability and familiarity among SQL users. The second, more programmatic approach, leverages the powerful column expression syntax combined with the bitwise `&` symbol.

Both methods achieve the same end goal: filtering a PySpark DataFrame by ensuring every row returned meets Condition A **AND** Condition B. However, the internal mechanics and the resulting DataFrame structure remain consistent across both techniques. Choosing between the SQL string method and the column expression method often comes down to the complexity of the filter conditions, the need for programmatic variable insertion, and adherence to specific organizational coding standards. We will examine both techniques in detail, demonstrating their implementation using the built-in filter function available on all DataFrames.

Setting Up the PySpark Environment and Sample Data

Before diving into the filtering examples, we must initialize a SparkSession and create a sample PySpark DataFrame. The SparkSession is the entry point to all Spark functionality and is necessary to execute any PySpark operations. Our sample data simulates team performance metrics, including the team name, their conference affiliation, points scored, and assists recorded. This dataset will serve as the basis for all subsequent filtering demonstrations, allowing us to clearly observe the results of applying the **AND operator**.

The following code block demonstrates the standard procedure for setting up the environment, defining the schema (column names), loading the raw data into a DataFrame, and finally displaying the initial structure of our dataset using the `.show()` action. This preparation step ensures that the

subsequent examples are run against a consistent and observable data source.

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|conference|points|assists|
+----+----------+------+-------+
A	East	11	4
A	East	8	9
A	East	10	3
B	West	6	12
B	West	6	4
C	East	5	2
+----+----------+------+-------+

Method 1: Using the SQL-Style 'AND' Operator (String Expression)

The first and often simplest method for applying the logical conjunction is by passing a single SQL-like string expression directly into the DataFrame's filter function. This method is particularly accessible for developers familiar with standard SQL syntax, as it allows combining conditions using the keyword `and` (or its equivalent function, `where`). When using this technique, the entire filtering logic is encapsulated within a single string argument, which Spark then parses and optimizes internally. This approach simplifies the code structure, especially when dealing with

numerous conditions, as it avoids complex Python operators and nested parentheses typical of the second method.

To perform a conjunctive filter using this method, the string expression must correctly reference the column names and enclose string literals (like 'East') in quotes, while numerical comparisons (like `points>5`) are written naturally. The key component is the explicit use of the `AND` keyword to join the individual conditions. For instance, if we want to retrieve rows where the `points` column is greater than 5 and the `conference` column is equal to "East," the entire expression becomes `'points>5 and conference=="East"'`. This string is then passed directly to `df.filter()`.

Practical Application of Method 1

We will now apply the SQL-style filtering method to our sample DataFrame. Our goal is to isolate records that satisfy two specific conditions: first, that the value in the `points` column strictly exceeds 5, and second, that the team belongs to the "East" conference. Both criteria must be met for a row to be included in the resulting DataFrame, which is the definition of the **AND operator**.

Observe the output below. Only three rows meet both criteria: Team A's three entries (11, 8, and 10 points) are all greater than 5, and they are all in the "East" conference. The entry for Team C (5 points) is excluded because it fails the first condition (5 is not greater than 5), even though it satisfies the second condition. Similarly, Team B's entries satisfy the points condition (6 points) but fail the conference condition.

#filter DataFrame where points is greater than 5 and conference equals "East"
df.filter('points>5 and conference=="East"').show()

```
+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
+----+----------+------+-------+
```

The resulting DataFrame perfectly illustrates the logical operation performed. Every row satisfies both of the predefined constraints. A significant advantage of using the SQL-style string is the ease with which multiple conjunctions can be chained. If we had a third condition, such as `assists>3`, we would simply append `and assists>3` to the existing filter string, demonstrating the scalability and clarity of this single-string approach for expressing complex logical requirements.

The value in the `points` column is greater than 5.

The value in the `conference` column is equal to "East".

Method 2: Utilizing the Bitwise `&` Symbol (Column Expression)

The second powerful method for applying conjunctive logic in PySpark involves using the bitwise `&` symbol. This approach is preferred by many data engineers because it utilizes PySpark Column objects directly, providing stronger type checking and often resulting in more efficient execution plans, as it is a pure PySpark expression rather than a string interpretation. Unlike the SQL-style method, which uses the keyword `AND`, the column expression method requires the use of the Python bitwise **AND operator**, `&`.

A critical difference when using the `&` symbol is the necessity of enclosing each individual condition within parentheses. In Python, the standard logical operators (`and`, `or`, `not`) cannot be overloaded for use with Spark Column objects. Therefore, the bitwise operators (`&`, `|`, `~`) are utilized instead. Due to operator precedence in Python, without parentheses surrounding each condition (e.g., `(df.points>5)`), Python would attempt to evaluate the comparison operators (`>`, `==`) and the bitwise operator (`&`) in an order that leads to errors, as the comparison results (boolean values) would be incorrectly interpreted. The parentheses enforce the correct evaluation order, ensuring that the boolean result of each condition is generated before the `&` operator combines them.

This methodology is more idiomatic to Python programming and allows for seamless integration of external Python variables or functions into the filtering criteria, making it highly flexible for dynamic data pipelines where filter values change based on run-time parameters.

Practical Application of Method 2

We now demonstrate how to achieve the same filtering result using the column expression syntax combined with the `&` symbol. We again seek rows where the `points` column is greater than 5 **AND** the `conference` column is equal to "East". Note the explicit use of parentheses around both the points condition and the conference condition.

The resulting DataFrame, as anticipated, is identical to the output obtained using Method 1, confirming that both techniques are valid for constructing logical conjunctions in PySpark. This consistency confirms that the choice between the two is often a matter of context and preference, not necessity for functional correctness.

#filter DataFrame where points is greater than 5 and conference equals "East"
df.filter((df.points>5) & (df.conference=="East")).show()

+----+----------+------+-------+
|team|conference|points|assists|

```
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
+----+----------+------+-------+
```

The rows selected again strictly adhere to the combined criteria. This method, while requiring careful attention to parentheses, offers significant advantages in complex data pipelines. When conditions are generated dynamically, constructing the filter using Python variables and the `&` operator is generally cleaner and less susceptible to SQL injection vulnerabilities than dynamically building a SQL-like string expression.

The value in the `points` column is greater than 5.
The value in the `conference` column is equal to "East".

Comparing PySpark Filtering Methods

Both the SQL-style string expression (using `AND`) and the column expression (using `&`) are fully supported ways to apply conjunctive filtering in PySpark. However, understanding their respective trade-offs is crucial for writing maintainable and high-performing Spark applications. The SQL string method is often easier to write quickly and visually inspect, especially for individuals transitioning from relational database environments. It requires less setup in terms of imports (no need for `pyspark.sql.functions` unless using complex functions) and avoids the Python operator precedence issues that necessitate parentheses in Method 2.

Conversely, the column expression method (Method 2) is generally considered the best practice for production PySpark code. Because it operates directly on Spark Column objects, it integrates natively with the DataFrame API, enabling Spark's catalyst optimizer to better understand and optimize the query plan. Furthermore, this method prevents the common pitfalls associated with embedding variables into string literals, such as SQL injection risks or simple syntax errors that might arise during string concatenation. For complex data pipelines, the programmatic control offered by using Column expressions and the bitwise `&` symbol makes filtering logic more robust and easier to debug.

Conclusion

Filtering a PySpark DataFrame using the **AND operator** is a foundational skill in large-scale data analysis. Whether you prefer the simplicity of the SQL-style string expression passed to the filter function using the `AND` keyword, or the robust, programmatic control offered by combining multiple Column conditions with the bitwise `&` symbol, PySpark provides effective tools to handle multi-

conditional constraints. For maximizing performance and ensuring long-term code stability, especially in environments where filtering parameters change dynamically, utilizing the column expression method (Method 2) is highly recommended. Mastering these techniques ensures that you can precisely extract the necessary information from massive datasets efficiently and accurately.

For further exploration into PySpark filtering, you might consider how to handle scenarios where only one of several conditions must be met, which utilizes the **OR operator**.

PySpark: How to Use "OR" Operator