How to Filter Data with the AND Operator in PySpark

Authored by

stats writer

February 10, 2026

The **AND operator** in **PySpark** serves as a fundamental building block for constructing sophisticated data queries. By allowing developers to logically combine two or more conditions, it facilitates the extraction of specific **data subsets** that satisfy multiple requirements simultaneously. In the realm of **Big Data** processing, the ability to refine search criteria is paramount for performance and accuracy. This operator is most frequently utilized within **filtering** transformations to prune **DataFrame** rows, ensuring that only the most relevant records are passed to downstream analytical processes.

Consider a scenario where an analyst needs to isolate a specific demographic from a massive **dataset**. An example of the **AND operator** in action would be filtering a user registry to include only those entries where the "age is greater than 18" and the "gender is female." Without this logical intersection, the analyst would be forced to perform multiple passes over the data, which is inefficient in a **distributed computing** environment. By using a single compound condition, **Apache Spark** can optimize the execution plan, significantly reducing the computational overhead required to return the desired results.

Furthermore, the **AND operator** is indispensable when building nested conditional logic for complex business rules. For instance, a financial application might require data where the "income is greater than 50,000" and simultaneously either the "education level is 'Bachelor's degree'" or the "occupation is 'Manager'." In this context, the **AND operator** acts as a primary gatekeeper, ensuring the first condition is strictly met before evaluating subsequent **Boolean logic**. Overall, mastering this operator is essential for any data engineer looking to harness the full power of **PySpark** for precision-based data manipulation and analysis.

Understanding Logical Filtering in Distributed Environments

In the ecosystem of **Apache Spark**, data is distributed across a **cluster** of machines. When we apply a filter using the **AND operator**, we are essentially defining a predicate that the **Spark engine** uses to evaluate each row across these partitions. This process is highly optimized through a component known as the **Catalyst Optimizer**. By understanding how the **AND** logic is translated into execution instructions, developers can write more efficient code that takes advantage of **predicate pushdown**, a technique where filtering is performed as close to the data source as possible to minimize **I/O** overhead.

The **AND operator** follows the standard rules of **Boolean algebra**, where a result is only "true" if every individual condition evaluates to true. In **PySpark**, this logic is implemented through the **API** in two primary ways: using SQL-style strings or using bitwise-style operators on Column objects. Both methods aim to achieve the same result but offer different syntax styles to suit the developer's preference or the specific requirements of the codebase. Choosing between them often depends on whether the user prefers the readability of **SQL** or the type-safety and flexibility

of the **Python** programmatic interface.

It is also important to consider how the **AND operator** handles **null values**. In many **relational databases** and **data processing** frameworks, the presence of a null value in a logical comparison can lead to "unknown" results. In **PySpark**, if one side of an **AND** expression is false, the result is false regardless of the other side. however, if one side is null and the other is true, the result remains null. Understanding these nuances is critical for maintaining **data integrity** when working with real-world datasets that are often incomplete or contain missing information.

Method 1: Utilizing String-Based Expressions for SQL Compatibility

The first common method for filtering a **PySpark DataFrame** with multiple conditions involves passing a single **SQL-like string** to the `filter()` or `where()` method. This approach is highly intuitive for users who possess a background in **Structured Query Language (SQL)**. By encapsulating the logic within a string, **PySpark** parses the expression and applies it to the **DataFrame** columns. This method often results in cleaner, more readable code when the conditions are straightforward and do not require complex **Python** variable injections.

When using the string-based **AND operator**, the syntax follows standard **SQL** conventions. You specify the column names, the comparison operators (such as `>`, `<`, or `==`), and the word "and" to join them. This format is particularly beneficial when sharing logic between different parts of a **data pipeline** that might use different languages, as **SQL** is a universal standard. However, developers must be careful with string formatting and escaping, especially when dealing with string literals inside the filter expression itself.

One of the primary advantages of this method is its brevity. As demonstrated in the example below, a complex multi-column filter can be written on a single line without the need for multiple sets of parentheses. This can make the code more accessible for **Data Scientists** who are more focused on exploratory analysis than on strict software engineering patterns. Despite its simplicity, **PySpark** still applies its full range of optimizations to these string expressions, ensuring that there is no performance penalty for choosing this more readable syntax.

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

Method 2: Applying the Ampersand Symbol for Columnar Logic

The second method for implementing **AND** logic in **PySpark** involves the use of the **ampersand (&)** symbol. This is the preferred method for many **Python** developers because it treats **DataFrame** columns as objects. By accessing columns via `df.column_name` or `df`, you can create

Boolean expressions that are evaluated row-wise. This approach is more robust for dynamic code where column names might be stored in variables or passed as **API** parameters.

A critical detail when using the **&** operator is the necessity of parentheses. Due to **operator precedence** in **Python**, the bitwise **&** has a higher priority than comparison operators like `>` or `==`. Without wrapping each condition in its own set of parentheses, the code will likely result in a `TypeError` or incorrect logic evaluation. For example, `(df.points > 5) & (df.conference == "East")` is the correct way to structure the statement to ensure the comparisons happen before the logical **AND** join.

This columnar approach offers greater flexibility for **functional programming** patterns. It allows developers to programmatically build filters by chaining **Python** objects together. It also integrates seamlessly with **PySpark's** built-in functions from the `pyspark.sql.functions` module, enabling advanced logic that might be difficult to represent as a simple string. For large-scale **software development** projects, this method is often favored for its explicitness and better integration with **Integrated Development Environments (IDEs)** that provide autocompletion and linting.

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

Practical Implementation: Setting Up the PySpark Environment

Before demonstrating the **AND operator** in practice, it is necessary to establish a **SparkSession**. This session serves as the entry point to all **Spark** functionality and is responsible for managing the **runtime environment**. In a typical script, we initialize the session using the `builder` pattern. Once the session is active, we can define our data and schema to create the **DataFrame** that will be used for our filtering examples.

In the following code block, we construct a sample **dataset** representing sports team statistics. The data includes categorical information like "team" and "conference," as well as numerical metrics like "points" and "assists." Defining a clear **schema** (the column names) is vital for **data processing** tasks, as it allows **PySpark** to map the raw data to a structured format that supports **relational operations**. This structured approach is what distinguishes **DataFrames** from lower-level **Resilient Distributed Datasets (RDDs)**.

Once the **DataFrame** is created, we use the `show()` method to visualize the initial state of our data. This step is crucial in **data engineering** to verify that the **ingestion** process has occurred correctly before applying transformations. The table produced shows several rows of data, some of which meet our target criteria and some of which do not. This provides a clear baseline for understanding how the **AND operator** will eventually narrow down the results to a specific subset

of the "East" conference teams with high scoring points.

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

Example 1: In-Depth Look at String-Based Filtering

In this first detailed example, we apply the **SQL-style** filter to our dataset. The goal is to isolate rows that belong to the "East" conference and have earned more than 5 points. By using `df.filter('points>5 and conference=="East"')`, we provide **PySpark** with a concise instruction. The engine parses this string, identifies the columns **"points"** and **"conference"**, and applies the logic across the **distributed data**. This method is exceptionally clean and minimizes the syntactic noise often associated with **Python**-heavy implementations.

Upon executing the `show()` command, the output displays a refined **DataFrame**. We can observe that the result set has been reduced from six rows down to three. Each remaining row strictly adheres to the compound condition. Specifically, teams like 'B' are excluded because they belong to the "West" conference, even if their points were high, and team 'C' is excluded because its points value (5) is not strictly greater than 5, illustrating the precision of the **AND operator**.

It is worth noting that you are not limited to just two conditions. You can chain multiple **AND** statements within the same string to create highly specific filters. For instance, you could add a third condition to check if "assists" are greater than 3. This extensibility makes the string-based method a powerful tool for rapid **data exploration** and **ad-hoc querying**, where the speed of writing the query is as important as the execution efficiency itself.

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

Example 2: Deep Dive into the Ampersand Operator Syntax

The second example demonstrates the **columnar approach**, which is the standard for programmatic **PySpark** development. By using `df.filter((df.points > 5) & (df.conference == "East"))`, we leverage the **Python API** to define our logic. This syntax is highly explicit, clearly defining which **DataFrame** and which columns are being targeted. While it requires a bit more typing than the string method, it provides better **compile-time** checks and reduces the risk of **SQL injection** or runtime string parsing errors.

As we examine the results of this operation, we see that they are identical to the results of the string-based method. This highlights an important aspect of **PySpark**: multiple paths often lead to the same underlying **Spark execution plan**. Whether you use the **ampersand** or the "and" string, the **Catalyst Optimizer** will likely generate the same optimized set of instructions for the **JVM** (Java Virtual Machine) that runs **Spark**. This ensures that developers can choose the syntax that best fits their personal style without worrying about performance discrepancies.

A significant benefit of this method is its ability to handle dynamic column names and variables. If

the threshold for points was stored in a variable, say `min_points = 5`, you could easily write `df.points > min_points`. This makes the code more reusable and maintainable in large-scale **data engineering** projects. Furthermore, using the **ampersand operator** is the gateway to more advanced **filtering** techniques, such as using the `isin()` method or complex mathematical transformations within the filter itself.

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

Summarizing Best Practices for Multi-Condition Filtering

When choosing between the string-based "AND" and the bitwise **&** operator, consider the context of your project. If you are working in a notebook environment like **Jupyter** or **Databricks** for quick **data analysis**, the string method is often faster to type and easier to read. However, if you are building a robust **data pipeline** that will be deployed to **production**, the columnar **&** operator is generally preferred due to its compatibility with unit testing, **type hinting**, and overall **Pythonic** structure.

Another important consideration is the use of `filter()` vs `where()`. In **PySpark**, these two methods are actually **aliases** of one another. `where()` is provided primarily for users coming from a **SQL** background, while `filter()` aligns more with **functional programming** paradigms found in **Scala** or **Python**. Regardless of which you choose, the **AND operator** functions exactly the same way in both, allowing for consistent and predictable data transformation logic across your entire **application**.

Finally, always remember to verify your results using actions like `show()`, `count()`, or `collect()`. Because **PySpark** uses **lazy evaluation**, the filtering logic is not actually executed until you call an action. This allows **Spark** to optimize the query plan as a whole. By effectively using the **AND operator**, you ensure that your **Spark jobs** are not only accurate but also performant, by filtering out unnecessary data as early as possible in the execution cycle. For more advanced logical operations, such as alternatives to the **AND** logic, you may explore other operators like **OR** or **NOT** to further refine your data processing capabilities.

PySpark: How to Use "OR" Operator