Learning PySpark: Using the “AND” Operator for Conditional Filtering

Authored by

Mohammed looti

November 10, 2025

Introduction to Conditional Filtering in PySpark

In the realm of big data processing, the ability to selectively isolate specific subsets of information is paramount for effective analysis and transformation. When utilizing PySpark, the powerful Python API for Apache Spark, conditional filtering serves as the foundation for tasks ranging from data quality checks to complex feature engineering. Central to this process is the implementation of logical conjunction, universally known as the **AND** logical operator. This operator dictates a strict requirement: a data record, or row, must simultaneously satisfy two or more distinct criteria to be included in the resulting DataFrame. Mastering the various syntaxes for implementing the **AND** condition is non-negotiable for achieving precise and accurate data preparation at scale.

PySpark offers developers significant flexibility by supporting two primary, yet functionally equivalent, methods for applying the **AND** condition within filtering operations. These methods are typically executed using the dedicated filter function (or its alias, the `where()` method), which is native to the DataFrame API. The first method adopts a traditional SQL-style approach, embedding the literal string "AND" within a comprehensive query expression. The second method, preferred by many Python developers for its programmatic advantages, relies on the Python bitwise operator, the ampersand symbol (**&**), which operates directly on PySpark Column objects. Understanding the subtle differences and appropriate use cases for each technique ensures that data pipelines are both efficient and maintainable.

Although both syntaxes ultimately leverage the same underlying execution engine--the highly optimized Apache Spark Catalyst Optimizer--to generate identical results, the choice between them often impacts code readability and dynamic query construction. The SQL-style string expression is favored for its simplicity in static, fixed queries, resembling familiar database operations. Conversely, the Column expression method, utilizing the **&** symbol, provides enhanced robustness when dealing with filters built from external variables or complex, nested conditional logic. We will explore both implementations in detail, demonstrating their practical application and discussing the circumstances under which one method may be superior to the other.

Setting Up the PySpark Environment and Sample Data

To effectively demonstrate the mechanics of the **AND** operator, a robust and reproducible environment is essential. Our initial step involves initializing the PySpark environment and constructing a representative sample DataFrame. This foundational exercise begins with importing the required modules, primarily the `SparkSession`, which serves as the entry point to all Spark functionality. Initialization ensures that the necessary cluster resources, even if running locally, are allocated and ready for distributed computation. Defining the data structure clearly before execution is crucial for Spark to infer or explicitly assign the correct schema, guaranteeing consistent data types across the columns.

For illustrative purposes, our sample dataset models hypothetical statistics for several sports teams. It includes key categorical features like `team` and `conference`, alongside crucial numerical metrics such as `points` scored and `assists` recorded. This blend of data types is deliberately chosen to allow us to construct sophisticated conditional filtering examples that combine both string comparisons and numerical thresholds. This setup mimics real-world scenarios where filtering must often cross column types to achieve the desired data subset. Explicitly defining the column names--`team`, `conference`, `points`, and `assists`--ensures clarity and avoids potential schema inference issues that can arise when dealing with complex, real-world data sources.

The following code block encapsulates the standard boilerplate required for any PySpark script: environment setup, data definition, DataFrame creation, and a final verification step to display the base table. Analyzing this initial output is vital, as it establishes the ground truth against which we will compare the results of our subsequent conditional filtering operations. This step confirms that the data is correctly loaded into the PySpark ecosystem and ready for manipulation using the DataFrame API.

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" Keyword for Filtering

The first and often simplest method for implementing logical conjunction in PySpark is by leveraging the SQL-style syntax. This approach involves passing a single, comprehensive query string directly to the `filter()` or `where()` method of the DataFrame. Within this string, the separate conditions are connected using the literal keyword `AND` (case-insensitivity is generally supported, but `and` is standard). This technique is highly intuitive for developers migrating from traditional relational database environments, as it closely mirrors standard SQL query construction, emphasizing clarity and reducing the Python-specific verbosity inherent in the alternative method.

When employing the SQL-style filter, the Apache Spark execution engine efficiently parses the string expression, converting it into an internal logical plan that can be optimized by the Catalyst Optimizer. This method is particularly well-suited for scenarios involving straightforward, static filtering criteria where the conditions do not rely on complex Python variables or dynamic runtime construction. For instance, filtering records where a numerical column exceeds a fixed value AND a categorical column matches a fixed string is perfectly handled by this concise syntax. Developers must, however, maintain strict attention to quoting conventions. While column names are referenced directly, any string literals (e.g., conference names) within the filter string must be correctly quoted, typically using double quotes when the filter string itself is enclosed in single quotes, to avoid parsing errors.

To illustrate, we will filter our sports statistics DataFrame to retrieve only those records where the team's `points` score is strictly greater than 5, **AND** the team belongs to the "East" `conference`. This requires combining a numerical threshold condition with a categorical equality condition, linked by the `and` keyword within the single string argument passed to the filter function. This example clearly showcases the readability and conciseness offered by the SQL-style approach when dealing with simple, fixed comparisons.

Example 1: Filter DataFrame Using "AND" Keyword

Applying Method 1 to our sample DataFrame `df` demonstrates how efficiently PySpark executes the SQL string expression. The resulting DataFrame will only contain rows that satisfy the conjunction of both filtering rules, confirming that the logical **AND** operation was performed successfully.

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

A meticulous review of the output confirms that all returned rows rigorously adhere to the dual constraints imposed by the **AND** condition. Specifically, every row successfully passes both checks: the `points` column value is indeed greater than 5 (values 11, 8, and 10), and the `conference` column contains the string "East". This method proves highly scalable; should a developer need to introduce a third, fourth, or fifth condition, the process involves merely appending another `and condition` statement to the existing filter string.

Method 2: Employing the Pythonic Bitwise & Operator

The second primary method for applying the **AND** condition in PySpark utilizes the Python bitwise operator, represented by the ampersand symbol (**&**). This technique is characteristic of Pythonic data manipulation and is the standard choice for advanced PySpark users, offering deep integration with the DataFrame's Column objects. Unlike Method 1, which relies on string parsing, Method 2 executes logic directly against the column references, making it significantly more adaptable for dynamic filtering scenarios, integration with UDFs (User Defined Functions), and complex expression construction built from iterating structures.

A critical requirement when using the bitwise **&** operator is the mandatory enclosure of each individual comparison in parentheses. This strict rule is imposed due to Python's operator precedence rules. In Python, the bitwise **&** operator has a higher precedence than the comparison operators (such as `>`, `<`, or `==`). If parentheses are omitted, Python attempts to execute the bitwise operation on the column objects before the logical comparison is completed, invariably leading to incorrect results or runtime errors. By wrapping each condition--for example, `(df.points > 5)`--in parentheses, we ensure that the comparison is evaluated first, producing a boolean Column object, which is then correctly combined with other boolean Column objects using the **&** operator.

The programmatic strength of Method 2 stems from its explicit manipulation of PySpark Column objects. This approach is superior when filter criteria are stored in variables, derived from function results, or generated through iterative loops, as it eliminates the complexities associated with

dynamically formatting and escaping SQL strings required by Method 1. This method promotes cleaner, more robust code when the logic is inherently complex or requires external parameterization.

Example 2: Filter DataFrame Using & Symbol

We now replicate the exact filtering objective from Example 1, but employing the Column-based approach with the bitwise **&** operator. Our aim remains to isolate rows where **points** > 5 **AND conference** == "East". Notice the required use of `df.column_name` to explicitly reference each column and the essential use of parentheses surrounding each logical clause, which correctly structures the logical operator evaluation.

#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 output generated by Method 2 perfectly aligns with the results from Method 1, confirming that the two syntaxes are functionally equivalent for conditional filtering in PySpark. This consistency underscores that the underlying Apache Spark engine processes both filter inputs into the same optimized execution plan. Every row in the resulting DataFrame confirms the dual satisfaction of the criteria: the `points` value exceeds 5, and the `conference` affiliation is "East." This method, despite its slightly increased verbosity due to the explicit column referencing and parentheses, is highly valued for its predictability and reliability when integrating complex transformations or expressions defined using the `pyspark.sql.functions` module.

Comparison: Performance, Readability, and Use Cases

Selecting the appropriate **AND** implementation method--SQL-style string versus Column-based **&** operator--is a pragmatic decision dependent on the specific requirements of the PySpark application, balancing developer experience against programmatic flexibility. For projects emphasizing rapid development and high SQL familiarity, Method 1 offers superior readability. Its concise nature minimizes code clutter and is generally easier to interpret for simple, fixed filtering rules that are hardcoded into the script. This method shines when the filter expression itself is

static and uncomplicated.

Conversely, Method 2, utilizing the **&** operator, is the definitive choice for sophisticated, programmatic data manipulation. If filtering logic must be constructed dynamically based on runtime parameters, external configuration files, or user inputs, the Column-based approach avoids the inherent security risks (though minor in localized scripts) and maintenance nightmares associated with complex string formatting and escaping. Furthermore, when integrating with sophisticated DataFrame transformations--such as those involving chained function calls or conditional logic structures like `pyspark.sql.functions.when()`--the Column object paradigm is the only viable path. The explicit referencing of columns (e.g., `df.column_name`) ensures type safety and clarity in complex environments.

From a performance standpoint, developers often worry about the overhead of one method versus the other. However, due to the efficiency of the Apache Spark Catalyst Optimizer, modern PySpark environments effectively translate both the SQL string and the Column object expressions into an identical, highly optimized internal logical plan. Consequently, performance differences are typically negligible, especially in comparison to the time spent on I/O or shuffling operations. Therefore, the primary factors guiding the choice should be code clarity, maintainability, and the need for dynamic query generation, rather than perceived micro-optimizations in execution speed. Developers should prioritize choosing the method that makes the code base easiest to understand and update.

Handling Complex Logic, Chaining, and Operator Precedence

The true utility of the **AND** logical operator is realized when multiple conditions must be chained together to achieve highly granular data segmentation. Both PySpark filtering methods readily support chaining. When using the SQL-style string method (Method 1), chaining is accomplished by simply inserting additional `and` keywords between each criterion within the single string argument passed to the filter function. For example, filtering for points greater than 5, conference being 'East', AND assists less than 10 would involve concatenating these three clauses with two `and` connectors.

For the Column-based method (Method 2), chaining is achieved by linking each parenthesized condition using the **&** symbol. This requirement for parentheses becomes increasingly critical as the number of chained conditions grows, ensuring that the desired sequence of logical evaluation is strictly maintained. For instance, to implement the three conditions mentioned above, the syntax would be: `df.filter((df.points > 5) & (df.conference == "East") & (df.assists < 10)).show()`. The explicit definition of each condition via parentheses prevents Python's operator precedence rules from misinterpreting the comparison operations before the bitwise combination is performed.

Conditional filtering often necessitates combining the **AND** operator with the **OR** operator

(represented by `or` in the string method or the bitwise **|** symbol in the Column method). When mixing conjunctions (**AND**) and disjunctions (**OR**) within a single filter expression, the precise definition of operator precedence is absolutely mandatory. For example, if the requirement is to satisfy `(Condition A AND Condition B) OR Condition C`, the Column-based approach requires explicit nested parentheses to enforce the grouping: `((df.A) & (df.B)) | (df.C)`. If the parentheses were structured differently, the logical output could be drastically altered. This control over grouping and precedence is the key reason why Method 2 is highly favored in complex analytical scenarios where filter logic is non-trivial and must be guaranteed through explicit code structure.

Initializing the Spark Environment: SparkSession Details

A fundamental component of any PySpark application is the `SparkSession`. This object serves as the unified entry point for reading data, executing SQL queries, and creating DataFrames. Before any filtering operations, including those utilizing the **AND** operator, can be performed, the session must be correctly initialized. The command `SparkSession.builder.getOrCreate()` is the recommended standard practice, as it intelligently reuses an existing session if one is already active, or creates a new one if necessary, ensuring resource efficiency and avoiding potential conflicts.

Beyond simple initialization, the `SparkSession` provides crucial configuration points that can influence how filtering and other operations are executed. For example, developers can configure memory limits, set the application name, or specify the deployment mode (local or cluster). While these configurations do not directly alter the logic of the **AND** operator, they significantly impact the performance and stability of the underlying Apache Spark cluster when processing massive datasets. Understanding the role of the SparkSession is therefore crucial for scaling filtering operations from simple examples to production-grade data pipelines.

The initial setup code snippet demonstrated the essential import statement: `from pyspark.sql import SparkSession`. This import grants access to the necessary API components. Following initialization, the `SparkSession` object is used to invoke the `createDataFrame()` method, which converts the defined Python list structures (data and columns) into the distributed PySpark DataFrame. This transition from local Python objects to distributed Spark objects is the moment the data becomes ready for large-scale, optimized operations, including the conditional filtering discussed throughout this guide.

Additional Resources for PySpark Filtering

Mastering conditional logic in PySpark is foundational for effective data engineering and analytics. The use of the **AND** operator, whether through SQL strings or the bitwise **&**, enables precise

control over data subsets. To further enhance your proficiency, we recommend exploring related concepts, particularly how to combine conjunctions with disjunctions (OR operations) and how to manage null values within complex filter statements.

PySpark: How to Use "OR" Operator to combine conditions (using `or` or **|**).
Official PySpark SparkSession Documentation for configuration details.
Detailed documentation on the filter function and its performance characteristics.