How to Drop Rows in PySpark Using Multiple Conditions

Authored by

stats writer

February 3, 2026

When working with large-scale data processing using PySpark, data cleaning and preparation are fundamental steps. A common requirement is the need to efficiently exclude or "drop" rows from a DataFrame that satisfy complex criteria. Unlike simple filtering, dropping rows based on multiple rules requires sophisticated application of boolean logic. This process is primarily managed through the built-in **.filter()** function, although the **.where()** function serves as a functional alias for the same operation.

The true power lies in combining conditions using fundamental logical operators--specifically AND (&), OR (|), and NOT (~). By leveraging these operators, developers can construct precise, multi-conditional statements necessary to identify and subsequently remove unwanted records from the dataset. Mastering this technique is crucial for ensuring the integrity and quality of data used in subsequent analysis or machine learning pipelines, offering unparalleled flexibility in big data manipulation.

PySpark: Drop Rows Based on Multiple Conditions

The Necessity of Conditional Row Manipulation in Big Data

In large-scale data environments, especially those handled by Apache Spark, the ability to selectively remove records based on specific criteria is not merely a convenience--it is an operational necessity. Dataframes often contain inconsistent, erroneous, or irrelevant observations that must be purged before statistical modeling or reporting can commence. Relying solely on filtering for a single column is often insufficient when dealing with complex datasets where row validity depends on the interaction between several feature values.

For instance, a transactional record might only be invalid if the **user ID** is null AND the **transaction amount** is zero. Removing rows where either condition is met independently would lead to data loss. Therefore, combining multiple conditions using precise logical constructs (e.g., conjunctions and disjunctions) is essential to isolate the precise subset of rows targeted for exclusion. PySpark's .filter() function provides the framework necessary to execute these complex queries efficiently across distributed clusters.

Understanding how PySpark handles boolean logic and column references (via `pyspark.sql.functions.col`) is the foundation of successful conditional dropping. Since we aim to **drop** rows, we are essentially looking for rows that satisfy the conditions and then applying the logical **NOT** operator (~) to retain all rows that do *not* meet those specific criteria. This inversion is key to transforming a selection process into an exclusion process.

The Core Syntax: Combining Conditions for Exclusion

To execute a multi-conditional drop operation in PySpark, we utilize the `.filter()` method in conjunction with column expressions imported from the `pyspark.sql.functions` module, often aliased as `F`. The syntax involves wrapping the conditional logic in parentheses and prefixing the entire expression with the logical NOT operator (~).

The general structure follows the pattern: `DataFrame.filter(~(Condition 1 & Condition 2 & ...))`. The use of `&` signifies a logical AND operation, meaning a row must satisfy **all** specified conditions within the parentheses to be selected for exclusion. Conversely, if we used `|` (logical OR), any row satisfying at least one condition would be selected for exclusion. The outer `~` ensures that only rows that fail the combined condition are kept in the resulting DataFrame.

Below is the canonical syntax used to drop rows from a PySpark DataFrame based on a conjunctive set of criteria (Condition A AND Condition B). Note the crucial role of the bitwise AND operator (`&`) in connecting the separate column evaluations:

import pyspark.sql.functions as F

#drop rows where team is 'A' and points > 10
df_new = df.filter(~((F.col('team') == 'A') & (F.col('points') >10)))

This specific expression precisely targets records where the value in the **team** column equals 'A' *and* simultaneously the value in the **points** column is strictly greater than 10. The resulting DataFrame, `df_new`, will exclude every row that meets this exact, combined criteria, demonstrating the precision achievable with multi-conditional filtering logic in a distributed computing environment.

Setting Up the PySpark Environment and Sample Data

To illustrate this powerful mechanism, we will first establish a working **SparkSession** and define a sample dataset. This ensures that the code snippets provided are immediately executable and verifiable. Our example data will simulate basketball player statistics, including their assigned team, position, and total points scored. This simple structure allows us to clearly observe which rows are dropped based on our criteria.

Defining the data structure involves creating a list of rows and associating them with a list of column names. Crucially, when working in PySpark, the `SparkSession` instance is required to invoke `createDataFrame`, transforming the raw Python data into the optimized, distributed DataFrame format necessary for efficient processing.

Execute the following Python code to set up the environment, define the sample data, and display the initial state of the DataFrame (`df`):

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|position|points|
+----+--------+------+
A	Guard	11
A	Guard	8
A	Forward	22
A	Forward	22
B	Guard	14
B	Guard	14
B	Forward	13
B	Forward	7
+----+--------+------+

The initial DataFrame contains eight records. We observe multiple records belonging to Team 'A' and Team 'B', with varying positions and points totals. Our objective is to selectively eliminate only those records from Team 'A' that represent exceptionally high performance (i.e., those scoring

more than 10 points), thereby isolating specific outliers or unwanted high-score entries for Team A.

Practical Example: Implementing Multi-Conditional Exclusion

We will now apply the multi-conditional filtering syntax introduced earlier. The specific requirement is to drop any row where **team** is 'A' *AND* **points** are greater than 10. This requires the use of the `&` (AND) operator to link the two conditions, and the `~` (NOT) operator applied externally to ensure we retain only the rows that fail this combined criteria.

When constructing the filter expression, it is vital to remember that comparison operators (`==`, `>`, etc.) applied to PySpark columns return column-level boolean expressions, not standard Python booleans. These expressions must be explicitly combined using PySpark's bitwise logical operators (`&`, `|`) and correctly parenthesized to enforce operator precedence, preventing parsing errors and ensuring the correct logical grouping.

The following code snippet imports the necessary functions, defines the exclusionary filter, applies it to the original DataFrame (`df`), and then displays the resulting DataFrame (`df_new`). Pay close attention to the rows that are successfully removed, confirming the application of the AND logic:

import pyspark.sql.functions as F

#drop rows where team is 'A' and points > 10
df_new = df.filter(~((F.col('team') == 'A') & (F.col('points') >10)))

#view new DataFrame
df_new.show()

+----+--------+------+
|team|position|points|
+----+--------+------+
A	Guard	8
B	Guard	14
B	Guard	14
B	Forward	13
B	Forward	7
+----+--------+------+

Analyzing the Results and Logic

Upon viewing the resulting `df_new`, we can clearly identify the rows that were successfully dropped based on the complex condition. The original DataFrame contained three rows that met the

exclusionary criteria: one row where Team='A' and Points=11, and two rows where Team='A' and Points=22. All three of these records are absent in the final output.

Crucially, records that partially met the condition were retained. For example, the row where Team='A' but Points=8 was kept because, although the first condition (Team='A') was true, the second condition (Points > 10) was false. Since the overall linked condition required *both* statements to be true (due to the `&` operator), this row failed the exclusionary test and was therefore retained by the external `~` (NOT) operator.

This demonstrates the foundational principle of conditional dropping: a row must satisfy **all** combined conditions specified within the internal parentheses to be tagged for exclusion. If a developer intends for a row to be dropped if it meets *either* condition, the logical OR operator (`|`) must be substituted for the logical AND operator (`&`).

Advanced Techniques: Using OR and Complex NOT Operators

While the AND operator (`&`) is useful for precise intersection-based exclusion, the OR operator (`|`) is often necessary when cleaning data based on multiple independent failures. If, for instance, we wanted to drop rows where Team='A' OR Position='Guard', we would use the `|` operator. The syntax remains structurally identical, only the internal operator changes: `df.filter(~((F.col('team') == 'A') | (F.col('position') == 'Guard')))`.

Furthermore, combining AND and OR logic allows for highly nuanced data exclusion. Imagine dropping rows only if they belong to Team 'A' AND (Points > 20 OR Position = 'Forward'). This would require nesting and careful parenthesization: `df.filter(~(F.col('team') == 'A') & ((F.col('points') > 20) | (F.col('position') == 'Forward')))`. Correct use of parentheses is paramount when mixing logical operators to guarantee the intended order of evaluation.

It is also important to remember the functional parity between .filter() and **.where()**. Both methods are optimized for distributing filtering operations across the Spark cluster. Choosing between them is largely a matter of style preference, though `.where()` sometimes offers slightly clearer semantic intent when dealing with SQL-like queries. Regardless of which method is chosen, the underlying mechanism for combining multiple column expressions remains the same, relying heavily on the `pyspark.sql.functions.col` representation.

Performance Considerations When Dropping Rows

When dealing with terabytes or petabytes of data, even simple operations like filtering can impact performance. In Apache Spark, operations like `.filter()` are considered **narrow transformations**, meaning they do not require data shuffling across the network, which is

beneficial for speed. However, highly complex multi-conditional filters can still introduce overhead.

One key optimization strategy is **predicate pushdown**, where Spark attempts to push filtering conditions down to the data source (like Parquet or ORC files) so that less data is read from disk in the first place. When constructing complex filter logic, ensuring the conditions are simple and deterministic helps Spark's Catalyst Optimizer apply these performance enhancements effectively.

For filters involving multiple columns, developers should prioritize applying conditions on indexed or partitioned columns first, if applicable, although Spark's execution plan often handles the optimal order internally. If performance becomes a bottleneck, reviewing the query execution plan (using `df.explain()`) can reveal if the complex logic is causing unforeseen full table scans or suboptimal resource utilization.

Summary of Best Practices for PySpark Filtering

Successfully implementing conditional row dropping in PySpark relies on adherence to several best practices. First and foremost, always ensure that column references are made using `F.col()` or similar functions to correctly interact with the distributed DataFrame structure, rather than attempting standard Python variable comparisons.

Secondly, diligent use of parentheses is non-negotiable, especially when combining the AND (`&`), OR (`|`), and NOT (`~`) logical operators. Misplaced parentheses can entirely change the logical outcome of the exclusion criteria, leading to either unintended data loss or failure to clean the intended subset of records.

Finally, for maximum clarity and maintainability, separate the conditional logic into readable variables before passing them to the `.filter()` function. For example, define `condition_A = (F.col('team') == 'A')` and `condition_B = (F.col('points') > 10)`, and then execute `df_new = df.filter(~(condition_A & condition_B))`. This practice significantly improves code readability and simplifies debugging of complex exclusion rules.

Related PySpark Data Manipulation Tutorials

The following tutorials explore alternative methods and common challenges related to data processing in the PySpark environment:

The following tutorials explain how to perform other common tasks in PySpark: