Learning How to Drop Rows with Specific Values in PySpark DataFrames

Authored by

Mohammed looti

November 11, 2025

Handling and cleaning large datasets is a fundamental task in modern data engineering. When working with PySpark, one of the most common requirements is the ability to remove rows that fail to meet specific criteria, often involving excluding known unwanted or outlier values. This article provides a detailed guide on how to efficiently drop rows in a PySpark DataFrame that contain one or more specific values using the powerful **filter()** transformation.

Understanding Data Filtering in PySpark

The core concept behind removing rows in PySpark is not "dropping" them directly, but rather creating a new DataFrame that contains only the rows satisfying a specified condition--a process known as filtering. The **filter()** transformation, an alias for the **where()** function, is a crucial part of the DataFrame API. It operates by evaluating a Boolean expression (a predicate) against every row, retaining only those where the expression evaluates to `True`.

When our goal is to eliminate rows containing a specific value, we must construct a condition that returns `True` for all rows **except** those we intend to remove. This inverse logic is essential for achieving the desired data exclusion. We will explore two primary methods for accomplishing this, ranging from simple inequality checks for single values to complex negation logic for multiple values.

The following methods allow you to drop rows in a PySpark DataFrame that contain a specific value or set of values:

Method 1: Drop Rows with Specific Value: Utilizing the standard inequality operator (**!=**) directly within the **filter()** function to exclude rows matching a single string or numeric value in a designated column.

Method 2: Drop Rows with One of Several Specific Values: Employing the **isin()** function in conjunction with the logical negation operator (**~**) to exclude rows that match any value within a predefined list of undesirable entries.

Method 1: Drop Rows with Specific Value Syntax

```
#drop rows where value in 'conference' column is equal to 'West'
df_new = df.filter(df.conference != 'West')
```

Method 2: Drop Rows with One of Several Specific Values Syntax

```
from pyspark.sql.functions import col

#drop rows where value in 'team' column is equal to 'A' or 'D'
df_new = df.filter(~col('team').isin())
```

The following examples show how to use each method in practice with the following PySpark DataFrame that contains information about various basketball players:

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|
+----+----------+------+
A	East	11
A	East	8
A	East	10
B	West	6
B	West	6
C	East	5
C	East	15
C	West	31
D	West	24
+----+----------+------+

Example 1: Excluding a Single Specific Value in PySpark

Often, data cleaning requires the removal of rows based on a single, known categorical value. In this example, we aim to process only the data related to the 'East' conference and exclude all records associated with the 'West' conference. This is achieved by using the inequality operator (**!=**) directly on the column reference within the **filter()** function.

The syntax `df.conference != 'West'` acts as the predicate. The DataFrame retains only those rows where the value in the `conference` column is not equal to 'West'. This method is highly performant and readable for excluding a singular criterion.

We can use the following syntax to drop rows that contain the value 'West' in the **conference** column of the DataFrame:

#drop rows where value in 'conference' column is equal to 'West'
df_new = df.filter(df.conference != 'West')

#view new DataFrame
df_new.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
A	East	11
A	East	8
A	East	10
C	East	5
C	East	15
+----+----------+------+

Notice that all rows in the DataFrame that contained the value 'West' in the **conference** column have been dropped, successfully filtering the data down to only the 'East' conference records.

Example 2: Excluding Multiple Specific Values in PySpark

When the requirement is to exclude data associated with several distinct values within the same column--such as dropping all data related to teams 'A' and 'D'--the inequality operator becomes cumbersome. A more elegant and efficient solution involves combining the **isin()** function with the logical negation operator (**~**).

The **isin()** function checks if a column value exists within a provided list of values. To achieve the

exclusion of these rows, we prepend the negation operator (**~**), effectively inverting the Boolean result. This requires importing the `col` function from pyspark.sql.functions for proper column reference.

We can use the following syntax to drop rows that contain the value 'A' or 'D' in the **team** column of the DataFrame:

from pyspark.sql.functions import col

#drop rows where value in 'team' column is equal to 'A' or 'D'
df_new = df.filter(~col('team').isin())

#view new DataFrame
df_new.show()

+----+----------+------+
|team|conference|points|
+----+----------+------+
B	West	6
B	West	6
C	East	5
C	East	15
C	West	31
+----+----------+------+

Notice that all rows in the DataFrame that contained the value 'A' or 'D' in the **team** column have been dropped. This technique is highly scalable when excluding large lists of unwanted category identifiers.

Summary of Techniques and Additional Resources

Mastering the **filter()** transformation is essential for effective data manipulation in PySpark. We have demonstrated two robust and efficient techniques for excluding unwanted rows: using the inequality operator (**!=**) for single-value exclusion and leveraging the combination of **isin()** and negation (**~**) for excluding multiple values simultaneously.

These methods ensure that the resulting DataFrame remains clean, focused, and ready for subsequent analytical tasks or storage. Remember that the choice between the two methods depends entirely on the cardinality of the values you intend to exclude.

Note: You can find the complete documentation for the PySpark **filter** function here.

Additional Resources

For those looking to deepen their understanding of Spark DataFrames and distributed data manipulation, we recommend exploring the following topics:

Understanding the concept of **Predicate Pushdown** in Spark for performance optimization.

Using complex boolean logic (`&` and `|`) within the **filter()** transformation for multi-column conditions.

Alternative methods for handling null values, such as using the **dropna()** function.