How to Remove Rows with Specific Values in PySpark

Authored by

stats writer

February 7, 2026

PySpark is recognized globally as a foundational library for executing large-scale parallel processing tasks across distributed computing clusters. When dealing with massive datasets--a frequent scenario in modern data analysis--the ability to quickly and efficiently refine the data structure is paramount. One of the most common and necessary operations in this phase is data cleansing, which often involves the precise removal of rows that meet specific, undesirable criteria.

This article focuses specifically on how to effectively drop rows within a DataFrame that contain one or more specified values. In PySpark, this powerful capability is primarily facilitated through the use of the `filter` transformation. Understanding how to leverage the filter function is essential for any data scientist working with Apache Spark, as it provides a clean, declarative, and highly optimized method for conditioning your data.

The core principle is simple: instead of explicitly dropping rows, we define a condition that selects only the rows we wish to retain. By structuring the condition to exclude (or negate) the values we want to discard, we effectively "drop" them from the resulting DataFrame. This approach ensures that the original data structure remains immutable, adhering to the principles of functional programming inherent in Spark, and returns a new DataFrame containing only the validated, cleaned data required for subsequent analytical steps.

PySpark: Drop Rows that Contain a Specific Value

Core Methods for Value Exclusion

There are several idiomatic ways to exclude rows based on column values within a PySpark DataFrame. The choice of method often depends on whether you are filtering out a single value or a list of multiple values. Both approaches utilize the fundamental `filter` function, but the syntax for defining the exclusion criteria differs slightly, particularly when importing specific functionalities from `pyspark.sql.functions`. These methods are highly efficient because the Spark engine optimizes these filtering operations before execution, pushing down the predicates to the source data whenever possible.

Below, we outline the two primary methods used in PySpark for achieving precise row deletion based on content criteria. These techniques are cornerstones of effective data wrangling in distributed environments, providing both flexibility and performance necessary for enterprise-scale data analysis.

Method 1: Single Value Exclusion: This method uses the simple inequality operator (`!=`) directly on the DataFrame column reference. This is the fastest and most readable method for removing rows matching a solitary criterion.

Method 2: Multiple Value Exclusion: This technique requires the use of the `isin()` function combined with the negation operator (`~`) to efficiently check if a column value belongs to a list of excluded items. This is crucial for filtering out categorical data based on a defined set of unacceptable entries.

Setting Up the Demonstration DataFrame

To solidify the understanding of these two filtering methods, we will apply them to a simple, concrete example. We begin by initializing a Spark session and creating a small DataFrame that simulates player statistics, including team, conference affiliation, and points scored. This setup mirrors the necessary environment for practical PySpark development and allows us to clearly observe the effects of the filtering transformations.

The dataset we are using contains nine records, providing sufficient variety across the categorical columns (`team` and `conference`) to test both single and multiple value exclusion filters effectively. Observe the initial state of the data carefully, as these values will be the targets for our subsequent row-dropping operations.

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

This structured data provides the perfect foundation for demonstrating how targeted data cleansing can be executed efficiently, a core requirement in advanced data preprocessing workflows.

Example 1: Dropping Rows with a Single Specific Value

Our first filtering task utilizes Method 1 to remove all rows associated with the 'West' conference. This is a common requirement when isolating a specific categorical subset of the data for deeper investigation, such as focusing an entire analysis exclusively on the 'East' conference. We use the simple inequality check against the `conference` column directly within the filter function.

The logic is expressed using the syntax `df.column != 'value'`. This expression evaluates to `True` for all rows that should be kept (i.e., those not equal to 'West') and `False` for all rows that should be excluded. The resulting DataFrame, `df_new`, will only contain records where the entry in the `conference` column is not 'West'.

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

A careful inspection of the output confirms that all five initial rows belonging to the 'West' conference (teams B, C, and D) have been successfully excluded. This transformation is fundamental for isolating specific subsets of data without modifying the source DataFrame, which preserves the integrity of the raw data while facilitating focused analysis.

Example 2: Dropping Rows with One of Several Specific Values

In the second scenario, we aim to exclude rows corresponding to specific teams--in this case, teams 'A' and 'D'. Since we are dealing with multiple exclusion criteria, we must employ Method 2, leveraging the `isin()` function and logical negation. This method is highly scalable, allowing you to easily add dozens or hundreds of values to the exclusion list without changing the overall structure of the filter expression. It is far superior to manually chaining multiple `!=` conditions together.

We first import the `col` function to allow for flexible column referencing and then apply the `isin()` check. The preceding tilde (`~`) is critical; it ensures that only rows where the `team` column value is **not found** in the specified exclusion list are preserved in the resultant DataFrame. The resultant expression `~col('team').isin()` means "Keep rows that are not team A AND not team D."

from pyspark.sql.functions importcol#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
+----+----------+------+

As expected, all records associated with teams 'A' (three records) and 'D' (one record) have been successfully dropped. The resulting DataFrame contains only records for teams 'B' and 'C', demonstrating the efficiency and flexibility of using negated `isin()` for complex exclusion tasks in PySpark. This method is the professional standard for filtering against lists of discrete values.

Alternative Syntax: Using the where() Function

While the `filter()` function is the canonical method provided by the DataFrame API, PySpark also provides the `where()` function. It is important to understand that `where()` is merely an alias for `filter()`. They perform the exact same transformation, and the Spark Catalyst Optimizer treats them identically.

Developers often prefer `where()` because it closely mirrors the structure of standard SQL queries (e.g., `SELECT * FROM table WHERE condition`). Using `where()` can sometimes enhance code readability, particularly for teams transitioning from traditional relational database environments. For example, the previous single exclusion example could be rewritten as: `df_new = df.where(df.conference != 'West')`.

Choosing between `filter()` and `where()` is purely a matter of coding style and team preference; neither offers a performance advantage over the other. The key takeaway is consistency: regardless of the function name chosen, the underlying mechanism for defining the exclusion condition (i.e., using `!=` or negated `isin()`) remains the same.

Performance Optimization via Predicate Pushdown

When working with distributed systems like Apache Spark, the performance of filtering operations depends heavily on internal optimizations. The most critical optimization related to filtering is **Predicate Pushdown**. Predicate Pushdown is the process where the Spark Catalyst Optimizer attempts to send the filter conditions down to the data source layer itself.

If the data is stored in partitioned formats (like Parquet or ORC) or sourced from an optimized database, applying filters early in the pipeline allows the system to skip reading large chunks of irrelevant data entirely. This dramatically reduces I/O operations, network transfer, and memory consumption. Therefore, when designing a PySpark job, it is best practice to apply filtering operations--especially those designed to drop specific values--as close to the beginning of the data flow as possible.

Furthermore, utilizing vectorized functions like `isin()` for multiple value checks is generally more efficient than complex manual boolean combinations. These optimized functions allow Spark to process the conditions in a highly efficient manner across the distributed clusters, ensuring that data cleansing operations are both effective and fast, which is crucial for high-throughput data

analysis.

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