Learning PySpark: How to Filter DataFrame Rows Using a List of Values

Authored by

Mohammed looti

November 11, 2025

One of the most common and fundamental operations in big data processing is filtering records based on specific criteria. When utilizing PySpark, the Python API for Apache Spark, efficient filtering is crucial for managing massive datasets. This guide details the essential syntax required to filter a DataFrame for rows that contain a value belonging to a predefined Python list, a technique that significantly enhances code readability and execution performance compared to chaining multiple conditional statements.

Introduction to Filtering in PySpark

Data filtering is the process of selecting a subset of rows that satisfy one or more conditions. In the context of large-scale data analysis, the efficiency of these filtering operations directly impacts the overall performance of the ETL (Extract, Transform, Load) pipeline. When dealing with categorical columns, such as team names, product identifiers, or regional codes, analysts often need to isolate records that match any of a handful of specific acceptable values. While one might attempt to use complex logical operators like `OR`, this approach becomes cumbersome and error-prone as the number of desired values increases. PySpark provides highly optimized methods, leveraging the underlying distributed nature of Spark, to handle these set-based comparisons elegantly, primarily through the use of the `isin()` function.

The core challenge addressed by this method is membership testing. Instead of writing verbose code like `df.filter((df.team == 'A') | (df.team == 'B') | (df.team == 'C'))`, which is difficult to maintain and dynamically generate, we can define our target values in a simple Python list. The framework then efficiently broadcasts this list and applies the membership check across all partitions of the distributed DataFrame. This operational paradigm ensures that filtering remains fast and scalable, regardless of the size of the dataset or the complexity of the criteria, provided the input list is manageable.

The Power of the `isin()` Function

The `isin()` function is a method available on PySpark **Column** objects, designed specifically for checking whether the value in that column is contained within a specified collection of values. This function accepts an array, tuple, or list of values and returns a boolean result for each row: `True` if the column value is present in the collection, and `False` otherwise. This boolean result is precisely what the `filter()` transformation requires to determine which rows should be included in the resulting DataFrame.

Understanding the mechanism behind isin is important for performance tuning. When PySpark executes this filter, it attempts to optimize the execution plan. If the list of values is small, it can often be handled very efficiently. This function translates directly into an optimized SQL `IN` clause when Spark converts the DataFrame operations into a physical query plan, which is generally one

of the fastest ways to perform multi-value lookups in a relational context. Therefore, employing `isin()` is the recommended best practice for filtering based on discrete membership criteria.

Practical Implementation: Defining the List and Filter Logic

To implement this filtering mechanism, the process involves two primary steps: first, defining the collection of target values, usually as a standard Python **list**; and second, applying the `filter()` transformation coupled with the isin condition to the DataFrame. The simplicity of this approach allows for dynamic filtering, where the target list can be generated programmatically based on user input or previous data analysis steps.

The following syntax demonstrates how to specify a list of teams--**'Mavs'**, **'Kings'**, and **'Spurs'**--and subsequently apply the filter. This pattern is highly reusable across different datasets and column types, provided the data types in the list match the data type of the column being filtered.

#specify values to filter for
my_list =

#filter for rows where team is in list
df.filter(df.team.isin(my_list)).show()

In this snippet, `df.team` selects the specific column we wish to evaluate. We then chain the isin method, passing our `my_list` as the argument. The entire expression, `df.filter(df.team.isin(my_list))`, constructs a new DataFrame containing only those rows where the value in the **team** column is a member of the specified list. This powerful yet concise approach is fundamental to efficient data manipulation in PySpark.

Constructing the Example DataFrame

To illustrate this functionality clearly, let us establish a sample DataFrame containing basketball statistics. This dataset includes two columns: **team** (categorical string data) and **points** (numeric data). We begin by importing the necessary components from pyspark.sql and creating a `SparkSession`, which is the entry point for all Spark functionality.

The following code block defines the raw data structure and uses the `spark.createDataFrame()` method to materialize the distributed table. This preparation step is vital for ensuring that the subsequent filtering operations have a concrete dataset to work upon, allowing us to observe the results accurately.

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|points|
+-------+------+
| Mavs| 18|
| Nets| 33|
| Lakers| 12|
| Mavs| 15|
| Kings| 19|
|Wizards| 24|
| Magic| 28|
| Nets| 40|
| Mavs| 24|
| Spurs| 13|
+-------+------+
```

As shown in the output above, the initial DataFrame, `df`, contains ten rows representing different teams and their corresponding point totals. Our objective is now to narrow this dataset down, retaining only the rows associated with the Dallas Mavericks (**'Mavs'**), Sacramento Kings (**'Kings'**), and San Antonio Spurs (**'Spurs'**), demonstrating the precision and effectiveness of the isin

method.

Executing the Filter Operation and Analyzing Results

Having established the source data, we can now execute the filtering logic discussed earlier. We specify our target list and apply the `df.filter(df.team.isin(my_list))` expression. This operation instructs Spark to scan the **team** column across all partitions and only keep records where the team name matches one of the three specified strings.

The following code block repeats the syntax and provides the resulting filtered DataFrame. Observe how the output is significantly reduced, containing only the relevant subset of the original data.

#specify values to filter for
my_list =

#filter for rows where team is in list
df.filter(df.team.isin(my_list)).show()

+-----+------+
| team|points|
+-----+------+
| Mavs| 18|
| Mavs| 15|
|Kings| 19|
| Mavs| 24|
|Spurs| 13|
+-----+------+

Upon examining the filtered output, we can confirm the precision of the `isin()` function. The resulting DataFrame contains exactly five rows, and crucially, every row's **team** value is restricted to either **Mavs**, **Kings**, or **Spurs**. All other team entries (Nets, Lakers, Wizards, Magic) have been successfully excluded. This demonstrates the operational efficiency and accuracy of using the isin method for complex inclusion filtering in PySpark environments.

Important Considerations for Using `isin()`

While the isin function is powerful, users must be aware of certain operational nuances to ensure correct and efficient results. The most critical aspect is **case sensitivity**. Like most string operations in PySpark, the comparison performed by `isin()` is case-sensitive by default. If your list contains 'mavs' (lowercase) but the DataFrame column contains 'Mavs' (capitalized), the filter

will fail to match those rows. If case-insensitivity is required, a preliminary transformation step, such as converting the column values to a uniform case (e.g., using `df.col.lower()`), must be applied before the `isin()` check.

Another consideration involves handling **Null values**. If the column being filtered contains `null`, the `isin()` function will treat the comparison logically. If the list of values itself contains `None` or `null`, then rows where the column is null will be included in the results. Conversely, if the list does not contain `None`, null values in the column will automatically be excluded from the filtered output, adhering to standard SQL three-valued logic where comparison against null often yields unknown or false unless specifically handled.

For users needing advanced details regarding function behavior, performance characteristics, and interactions with different data types, the official documentation serves as the ultimate authoritative source. We encourage consulting the complete documentation for the isin function provided by the pyspark.sql library.

Conclusion and Next Steps

The ability to filter a DataFrame based on membership in a Python list, facilitated by the `isin()` function, is a cornerstone of efficient data processing in PySpark. This method not only simplifies the code required for multi-value filtering but also harnesses Spark's optimization capabilities to ensure rapid execution even across massive datasets. Mastering this technique allows data engineers and analysts to quickly isolate relevant subsets of data for further transformation or analysis.

Beyond simple inclusion, PySpark offers a rich suite of filtering and transformation tools. Users should explore related operations, such as filtering for values **not** in a list (achieved by negating the `isin()` condition using the `~` operator), filtering based on complex string patterns (using `like()` or regular expressions), or applying user-defined functions (UDFs) for highly customized row selection logic.

To continue building expertise in PySpark, consider exploring tutorials on how to perform other common but crucial tasks:

Joining two DataFrames based on a shared key.
Grouping and aggregating data (e.g., calculating mean points per team).
Handling missing values (nulls) using functions like `fillna()` or `dropna()`.

These skills, combined with effective filtering, form the foundation for robust big data analytics pipelines.