How to Filter PySpark Rows Using a List of Values

Authored by

stats writer

January 20, 2026

The fundamental necessity of selecting specific rows is paramount in any data processing workflow. PySpark, the robust Python API for Apache Spark, provides highly optimized and declarative methods for achieving this goal. A common requirement is filtering a DataFrame based on whether the values in a particular column match any element within a predefined Python list. This operation is crucial for targeted analysis, validation, or generating specific subsets of data for subsequent tasks.

In PySpark, selecting rows based on membership within a list is efficiently handled by the built-in **isin** function. This function is specifically designed to check the membership of column values against a collection of specified items. When applied to a DataFrame column, **isin** evaluates each row's value against the provided list, resulting in a conditional boolean column. This elegant approach eliminates the need for complex looping or combining numerous OR conditions, guaranteeing both code clarity and performance optimization, which is essential when dealing with large-scale distributed datasets.

Understanding the isin function and its Role

The **isin** function is a powerful method readily available on all **PySpark Column** objects, allowing users to verify if the content of a column belongs to a specified collection provided as an argument. The function is versatile, accepting standard Python containers such as lists, tuples, or sets of values. Conceptually, its operation mirrors the familiar SQL `IN` clause, but it is implemented natively within the Spark framework to harness the power of parallel processing and distributed computing capabilities. The function's output--a boolean column--is the perfect prerequisite for the subsequent application of the **filter** transformation.

When implementing the isin function, it is critically important that the data type of the values contained within the Python list is compatible with the data type of the column being filtered. For example, if the target column holds string values, the list must strictly contain strings; similarly, numerical columns require numerical lists. Mismatched data types will often result in zero matches being found, as Spark's comparison logic is robustly strict. This adherence to type consistency ensures the reliability and accuracy of data manipulation within the PySpark environment.

Essential Syntax for Filtering a PySpark DataFrame

To successfully execute the filtering operation, we must couple the DataFrame's primary **filter** method with the conditional result generated by the **isin** function applied to the designated column. The filter method requires an input column composed entirely of boolean values--where a `True` value signifies that the row should be retained, and `False` dictates its removal. The isin function naturally produces this exact required boolean output, making the integration seamless and highly readable for data engineers.

The foundational syntax involves three distinct, sequential operations: first, defining the Python list containing the target values; second, accessing the specific column on the DataFrame to be queried; and third, embedding the isin function call within the **filter** method. This declarative programming paradigm is a hallmark of Spark, enabling developers to clearly state the desired transformation without delving into the complexities of distributed execution planning. The example below illustrates this primary concept, showing how to filter based on a list of team names.

The efficiency of this filter operation is a major advantage. Spark's core optimization engine intelligently handles the distribution of the filter list and the comparison operations across all nodes in the cluster. This consistency ensures exceptional performance, regardless of whether the filter list contains ten items or ten thousand, and irrespective of the gigabyte or terabyte size of the source DataFrame.

You can use the following syntax to filter a PySpark DataFrame for rows that contain a value from a specific list:

```
#specify values to filter for
my_list =

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

This particular example strictly instructs Spark to retain only those rows where the value in the **team** column is present among the elements defined in `my_list`. The expression `df.team.isin(my_list)` is pivotal, as it correctly generates the necessary conditional boolean column that dictates the entire data transformation.

Setting Up the Demonstration Dataset

To provide a practical demonstration of the isin function, we must first initialize a suitable dataset. We will simulate a common scenario involving basketball team statistics, recording points scored by various teams. This synthetic dataset will be constructed by defining raw data rows and column names, followed by the mandatory initialization of a Spark Session, which is the gateway for all data operations within the distributed computing environment.

The setup process commences by importing the `SparkSession` class from the `pyspark.sql` module and instantiating an active Spark application. Subsequently, the source data is meticulously defined as a list of lists, where each internal list represents a single row, composed of the team name (a string) and the points scored (an integer). Explicitly defining the column names--`team` and `points`--ensures the resulting data frame possesses a clear, structured schema that is easy to query and manipulate.

The following comprehensive code block illustrates this complete setup sequence. This sequence culminates in the display of the initial, unfiltered dataset. This initial view is essential for establishing the baseline state of the data before we apply any transformation or filtering logic. Pay close attention to the variety of teams represented, as this represents the entire population from which we will select our specific subsets of data.

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|
+-------+------+

Executing the List-Based Row Filter

With the sample dataset successfully instantiated, we can now proceed to execute the filtering logic utilizing the powerful **isin** function. Our specific objective in this demonstration is to isolate all records that correspond exclusively to the "Mavs", "Kings", and "Spurs" teams. This critical selection criteria is formalized within a standard Python list, which acts as the precise parameter input for our column membership comparison operation.

The implementation of the filter is both concise and highly expressive. We first define the list, `my_list`, explicitly containing our target team names. Subsequently, we chain the **filter** method onto our data frame `df`, providing the condition `df.team.isin(my_list)` as the predicate. This command clearly instructs Spark to compare every entry in the `team` column against the set of elements in `my_list`, retaining only those rows that contain a match. This transformative step yields a new data frame containing only the desired rows.

The following code block executes this final transformation and displays the resulting, filtered output. It is remarkable how the underlying complexity of distributed computation is entirely managed by Spark, allowing the user to concentrate solely on the declarative logic required for this specific data manipulation task.

We can use the following syntax to filter the DataFrame for rows where the **team** column is equal to a team name in a specific list:

#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|

```
+-----+------+
```

Analyzing the Filtered Output for Accuracy

Upon careful review of the output generated by the transformation, we can definitively confirm that the **isin** function successfully executed the targeted selection operation. The resultant data frame contains exactly five rows, and, crucially, every value in the **team** column belongs exclusively to the predefined set of target teams: {'Mavs', 'Kings', 'Spurs'}. All rows corresponding to teams not specified in our list--such as 'Nets', 'Lakers', 'Wizards', and 'Magic'--have been correctly and completely excluded from the final result set.

This validation step highlights the effectiveness and precision achieved by using **isin** for list-based filtering. Any incorrect implementation of the filter might have resulted in the inclusion of irrelevant teams, or perhaps the failure to capture all instances of the specified teams. The clean, verifiable output confirms that the underlying distributed comparison logic performed an accurate and exact membership check across the entire dataset structure.

Important Considerations: Case Sensitivity and Data Type

A highly important operational detail when employing the **isin** function, particularly when dealing with string-based columns, is its default behavior of **case sensitivity**. By default, the comparison executed by Spark between the column values and the list elements is exact: it treats 'MAVS', 'Mavs', and 'mavs' as three entirely distinct values. This means that if our source column contained 'mavs' (lowercase) but our filter list contained only 'Mavs' (capitalized), the row containing the lowercase entry would be strictly excluded from the filtered results.

Managing this case sensitivity is essential during the data preprocessing phase. If the analytical objective requires a case-insensitive match, the user must explicitly standardize the column values prior to comparison. A common method involves converting the target column values to a consistent case (e.g., lowercase using `df.column.lower()`) before applying the **isin** filter. Standardization is generally the preferred best practice for maintaining data quality and simplifying the overall filtering logic, ensuring robustness across varied data inputs.

Note #1: The **isin** function is case-sensitive, requiring exact matches between list elements and column values.

Alternative Methods and Documentation Resources

While **isin** stands out as the most recommended and optimized method for filtering based on list membership, PySpark does offer alternative avenues, though they are generally less efficient for

this specific application. One could theoretically construct a lengthy series of `OR` conditions combined using the bitwise OR operator (`|`). However, this technique rapidly becomes complicated, difficult to maintain, and significantly less performant as the number of elements in the filter list increases, due to the added complexity in the logical expression that Spark must parse and optimize across the cluster.

For scenarios involving extremely large filter lists (e.g., hundreds of thousands or millions of unique elements), advanced users may consider an alternative approach: broadcasting the filter list as a small lookup DataFrame and then performing a **semi-join** (an inner join that only retains the rows from the left side if a match is found on the right). However, for lists of typical size (ranging from tens to thousands of elements), the inherent simplicity, readability, and built-in optimization of the isin function establish it as the optimal and preferred choice for list-based filtering.

For those seeking deeper technical insight into the functionality and various parameters of this essential method, consulting the official Spark documentation is highly recommended. A thorough understanding of how Spark internally handles these list comparisons can significantly aid in writing highly scalable, robust, and efficient data processing code.

Note #2: You can find the complete documentation for the PySpark isin function here.

Conclusion and Further Exploration

The **isin** function provides a clean, Pythonic, and highly performant way to implement complex membership checks against columns in a distributed PySpark environment. By transforming a simple list into an optimized boolean filtering condition, it allows data scientists and engineers to easily extract specific subsets of data without resorting to verbose or inefficient conditional logic. Mastering this function is foundational to effective data manipulation within Apache Spark.

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