Use “IS NOT IN” in PySpark (With Example)

Authored by

stats writer

November 16, 2025

Introduction to Exclusion Filtering in DataFrames

One of the most frequent requirements in data manipulation and preparation is the ability to selectively exclude records that match a specific set of criteria. In the context of large-scale data processing using PySpark, efficiently performing exclusion filtering is essential for performance and accuracy. This capability mirrors the functionality of the SQL `NOT IN` clause, allowing developers to define a list of values and remove any row where a target column contains one of those values.

The standard approach for filtering in DataFrame operations involves applying a Boolean expression to the `filter()` or `where()` method. When dealing with multiple exclusion criteria, defining numerous `!=` (not equal to) conditions becomes cumbersome and highly inefficient. Therefore, PySpark provides a concise and optimized mechanism utilizing the built-in `isin() function` in combination with a powerful logical negation operator.

This article provides a detailed guide on generating clean, effective exclusion filters using the 'IS NOT IN' logic within PySpark. We will explore the necessary syntax, understand the role of the negation operator, and walk through a comprehensive example demonstrating how to filter DataFrame rows where a value in a specific column is explicitly **not** present in a predefined list or array of values.

The PySpark Syntax for Exclusion

To perform an 'IS NOT IN' operation in PySpark, we leverage the existing `isin() function` and precede the entire Boolean condition with the tilde (~) operator. The `isin()` function checks if a column value is present **in** the provided array. By applying the tilde operator, we logically negate this result, effectively checking if the column value is **not in** the array.

The following fundamental structure outlines how to define the array of exclusion values and apply the negation logic within the DataFrame `filter()` method. This streamlined approach ensures that your code remains readable and performs optimally, even when dealing with large datasets processed by PySpark clusters.

#define array of values to exclude
my_array =

#filter DataFrame to only contain rows where 'team' is not in my_array
df.filter(~df.team.isin(my_array)).show()

In this specific example, the filter criteria instructs PySpark to only retain rows where the value found in the **team** column does not match any element within the defined `my_array`. Therefore, any record associated with the values **A**, **D**, or **E** will be systematically excluded from the resulting

DataFrame. This method is far superior to writing compound conditions like `df.team != 'A' & df.team != 'D'`, especially when the exclusion list grows significantly large.

The Role of the Tilde (~) Operator in PySpark

The key component that transforms the inclusion check (`isin()`) into an exclusion check is the tilde (~) operator. In Python, and consequently in PySpark's column expressions, the tilde symbol serves as a unary operator representing logical negation. When applied to a Boolean expression--which is what `df.team.isin(my_array)` returns--it flips the truth value of every element.

If `df.team.isin(my_array)` evaluates to **True** for a given row (meaning the team is 'A', 'D', or 'E'), applying the tilde operator makes the entire expression **False** for that row. Since the `filter()` function only keeps rows where the condition is **True**, these matching rows are dropped. Conversely, if `isin()` returns **False** (the team is something else, like 'B' or 'C'), the tilde flips it to **True**, and the row is retained.

Understanding this relationship between the `isin() function` and the tilde (~) operator is fundamental to mastering complex conditional filtering in PySpark. This combination allows for a clean, declarative approach to defining exclusion lists, ensuring robust and easily maintainable data processing scripts, crucial for production-level PySpark environments.

Example: How to Use "IS NOT IN" in PySpark

To illustrate this powerful filtering technique, let us construct a practical scenario involving a sample dataset containing basketball player statistics. This example will demonstrate the complete workflow, from setting up the initial DataFrame to applying the exclusion filter and reviewing the resulting output. We will focus on filtering based on the **team** column.

Suppose we have the following PySpark DataFrame, which holds records detailing the performance (points and assists) of players across various teams ('A' through 'E') and conferences ('East' or 'West'). Before analysis, we might need to remove data belonging to certain teams deemed irrelevant for a specific report.

The following code block initializes the Spark session, defines the schema, loads the data, and displays the original DataFrame content, setting the stage for our exclusion operation. Note how the creation process involves defining both the data structure (rows) and the column names (schema).

```
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|assists|
+----+----------+------+-------+
| A| East| 11| 4|
| A| East| 8| 9|
| A| East| 10| 3|
| B| West| 6| 12|
| B| West| 6| 4|
| C| East| 5| 2|
| D| East| 14| 2|
| E| West| 25| 2|
+----+----------+------+-------+
```


Applying the Exclusion Filter for Specific Teams

Our objective is to perform filtering such that the resulting DataFrame excludes all rows corresponding to Team A, Team D, and Team E. This effectively isolates the performance data for Team B and Team C. We will define an array containing these excluded identifiers and then apply the negation logic to the `isin()` check.

This technique is critical in scenarios where data quality requires removing records marked with

specific error codes, or when analysis must focus solely on a subset of categories while discarding all others listed in a reference table. The ability to handle this exclusion using a simple array lookup vastly simplifies the code structure compared to complex joining or sequential filtering operations.

The following PySpark code snippet defines the array, applies the filter using the tilde (~) operator, and outputs the resulting filtered data, clearly demonstrating the removal of the targeted teams.

#define array of values
my_array =

#filter DataFrame to only contain rows where 'team' is not in my_array
df.filter(~df.team.isin(my_array)).show()

+----+----------+------+-------+
|team|conference|points|assists|
+----+----------+------+-------+
B	West	6	12
B	West	6	4
C	East	5	2
+----+----------+------+-------+

Analyzing the Filtered Output

Upon execution, the resulting DataFrame clearly shows only the records where the value in the **team** column is **not** equal to A, D, or E. Specifically, we observe three rows remaining: two rows belonging to Team B and one row belonging to Team C. This validates the effectiveness of the combined `isin() function` and tilde (~) operator for implementing SQL's `NOT IN` logic.

It is important to reiterate that the core mechanism relies on the tilde (~) operator being explicitly placed immediately before the conditional expression. If the tilde is omitted, the operation reverts to inclusion filtering, retaining only teams A, D, and E. Careful placement ensures the correct logical negation is applied across the entire column vector.

This method scales exceptionally well, as PySpark efficiently distributes the array lookup across the cluster nodes, minimizing shuffle operations often associated with more complex joins or external lookups. This efficiency is critical when dealing with truly massive datasets common in big data analytics.

Handling Data Types and Null Values

When using array-based exclusion filtering, it is crucial to ensure that the data types of the

elements in the exclusion list (`my_array`) match the data type of the target column (e.g., `df.team`). Mixing strings and integers without explicit casting can lead to logical errors or unexpected results, as Spark performs strict type comparisons.

Furthermore, the behavior of `isin()` with respect to **null** values must be noted. If a cell in the target column contains a null value, the `isin()` condition will typically evaluate to **False** (as a null cannot be definitively shown to be included in the list). When the tilde (~) operator is applied, this **False** result is negated to **True**, meaning rows containing null values in the filtered column will often be retained by the 'IS NOT IN' logic.

If you need to explicitly exclude or include null values, it is best practice to chain an additional condition using the `isNull()` or `isNotNull()` functions. For instance, to ensure null values are also removed alongside the excluded list, you would chain the condition `& df.team.isNotNull()` onto the existing filter expression.

Advanced Considerations and Performance

For exclusion lists that contain thousands or millions of elements, managing the size and transmission of the array becomes a performance factor in distributed computing frameworks like PySpark. When the list is very large, Spark must serialize and broadcast this array to every executor node. If the array is too large, broadcasting may become a bottleneck.

In such advanced scenarios, where the exclusion list is sourced from another large DataFrame (DF2), a more scalable and generally preferred approach is to use a **Left Anti Join**. A Left Anti Join effectively returns all rows from the primary DataFrame (DF1) that have no matching keys in the secondary DataFrame (DF2, the exclusion list). This prevents the need to materialize a massive Python array and relies on Spark's optimized join algorithms.

However, for moderately sized exclusion lists (hundreds or thousands of items, depending on cluster memory), the `~df.col.isin(array)` method remains the most straightforward and syntactically clean solution for exclusion filtering. It provides simplicity without sacrificing significant performance for typical operational data tasks.

Conclusion: Mastering PySpark Filtering Logic

The ability to perform robust exclusion filtering is a fundamental skill for any data engineer working with PySpark. By combining the `isin() function` with the logical negation provided by the tilde (~) operator, developers can implement the SQL `NOT IN` clause cleanly and efficiently on any DataFrame column.

Remember that the tilde (**~**) operator is the core element used in PySpark to represent **NOT** when

dealing with boolean column expressions. This convention extends beyond `isin()`, applying to other negation requirements such as `~df.col.isNull()` or `~(df.col > 10)`. Mastering this single operator unlocks complex conditional processing essential for advanced data preparation tasks.

By adhering to this simple but powerful syntax--`df.filter(~df.column.isin(exclusion_list))`--you ensure that your data processing pipelines are both highly performant and easy to maintain, directly contributing to more accurate and reliable data analysis outcomes across your distributed computing environment.