Learning PROC SQL: How to Use the IN Operator in SAS

Authored by

Mohammed looti

April 29, 2026

The **SAS** System stands as a leading platform for advanced analytics, offering a comprehensive suite of tools for business intelligence, predictive modeling, and sophisticated **data management**. Central to its power is the **PROC SQL** procedure, which seamlessly integrates the robust capabilities of the **Structured Query Language** (**SQL**) directly within the **SAS** environment. This unique synergy allows users to leverage familiar **query** syntax for complex **data manipulation** and retrieval tasks on **SAS datasets**, effectively combining the best aspects of both systems for highly efficient workflows.

Among the most fundamental and efficient tools available in **SQL**, and thus in **PROC SQL**, is the **IN operator**. This operator is crucial for streamlining the process of **data filtering**, allowing analysts to select rows based on whether a specified column's value is contained within a predefined list of possibilities. Rather than constructing convoluted chains of multiple **OR conditions**, the **IN operator** provides a concise, readable, and highly maintainable alternative, significantly enhancing the clarity of complex queries.

This expert guide will thoroughly examine the practical implementation of the **IN operator** within **PROC SQL**. We will detail its required **syntax**, provide clear, actionable examples, and introduce its powerful logical inverse, the **NOT IN operator**, which is essential for exclusion-based filtering. By the conclusion of this article, you will possess a robust understanding of how to employ these operators for precise and effective **data filtering** operations in your daily **SAS** programming efforts.

Understanding the IN Operator: Syntax and Purpose

The **IN operator** functions as a crucial logical tool within the **WHERE clause**, designed to validate if an expression's value corresponds to any item within a defined set of values. This capability is exceptionally valuable when the goal is to segment or filter a **dataset** based on predetermined criteria, such as selecting records that belong to specific categories, regions, or groups. By consolidating multiple comparisons into a single, straightforward condition, the **IN operator** simplifies the code required for targeted **data selection**, making your queries far easier to read and maintain.

The fundamental **syntax** for applying the **IN operator** within a standard **PROC SQL** query is highly intuitive. It involves specifying the column you wish to check, followed by the `IN` keyword, and then listing the acceptable values, enclosed in parentheses. This structure clearly defines the inclusion criteria for the data being retrieved.

```
PROC SQL;
SELECT column1, column2, ...
FROM your_dataset
WHERE column_name IN (value1, value2, value3, ...);
QUIT;
```

In this structure, `column_name` represents the variable undergoing the filtering process, while the comma-separated list `value1, value2, value3, ...` dictates the specific conditions for inclusion. It is paramount to remember the rules for specifying these values: character strings must always be enclosed in single quotes, whereas numeric values should not have quotes. Crucially, the **IN operator** inherently translates into an implicit logical **OR condition**, meaning it returns records where `column_name` matches any one of the provided values. This mechanism is what allows the operator to replace numerous individual comparisons with a single, streamlined expression.

Practical Application: Filtering Data with the IN Operator

To fully appreciate the efficiency of the **IN operator**, let us apply it to a practical scenario involving a sample **SAS dataset**. We will assume the existence of a dataset named `my_data`, containing essential performance metrics for basketball players, specifically tracking their team assignment and points scored. Our goal will be to selectively retrieve player records based exclusively on their team affiliations, demonstrating precise **data selection** capabilities. Before applying the filter, we must first ensure the sample data is available and review its initial state.

The following **SAS** code block illustrates the creation of the `my_data` dataset using a standard **DATA step**, followed by the use of **PROC PRINT** to visualize its contents. This initial step confirms that our dataset includes various teams (A, B, C, D, and E), setting the stage for targeted filtering.

```
/*create dataset*/
data my_data;
input team $ points;
datalines;
A 12
A 14
A 15
A 18
B 31
B 32
C 35
C 36
C 40
D 28
E 20
E 21
;
run;
```

```
/*view dataset*/
proc print data=my_data;
```

Obs	team	points
1	A	12
2	A	14
3	A	15
4	A	18
5	B	31
6	B	32
7	C	35
8	C	36
9	C	40
10	D	28
11	E	20
12	E	21

Our objective is clear: we want to isolate the records corresponding only to teams A, B, and E. If we were to implement this requirement without the **IN operator**, the necessary **WHERE clause** would be verbose and cumbersome, requiring explicit chaining: `WHERE team = 'A' OR team = 'B' OR team = 'E'`. Utilizing the **IN operator**, however, allows us to express this complex logic in a single, highly readable condition. This not only enhances the code's clarity but also contributes to better performance and reduced chances of logical errors, which is critical for efficient **data analysis**.

The following **PROC SQL** statement demonstrates how efficiently we can filter `my_data` to include only the desired teams by passing the list ('A', 'B', 'E') directly to the `IN` clause. This approach is the cornerstone of effective list-based filtering in **SAS** workflows.

```
/*select all rows where team is A, B, or E*/
proc sql;
select *
from my_data
where team in ('A', 'B', 'E');
quit;
```

team	points
A	12
A	14
A	15
A	18
B	31
B	32
E	20
E	21

The NOT IN Operator: Excluding Specific Values

While the **IN operator** is employed to select records that are present within a designated list, the **NOT IN operator** provides the exact logical inverse, focusing on exclusion. This powerful operator is specifically designed to retrieve all rows where a column's value does *not* match any item contained within the specified list. This capability is indispensable for scenarios requiring the elimination of known exceptions, outliers, or irrelevant data points from an analysis, allowing the user to concentrate on the remaining, residual data segments.

The **syntax** for implementing the **NOT IN operator** mirrors that of the **IN operator**, distinguished only by the inclusion of the `NOT` keyword immediately preceding `IN`. This slight modification completely reverses the filtering logic, transforming the condition from one of inclusion to one of exclusion.

```
PROC SQL;
SELECT column1, column2, ...
FROM your_dataset
WHERE column_name NOT IN (value1, value2, value3, ...);
QUIT;
```

Just like the **IN operator**, the values provided in the list can be either numeric or character strings, maintaining the requirement for single quotes around character values. Logically, the **NOT IN operator** executes a series of implicit `AND NOT` comparisons, ensuring that `column_name` is not equal to `value1` AND not equal to `value2`, and so forth. Mastery of the **NOT IN operator** is a vital skill for robust **data filtering**, enabling precise control over which observations are included in or excluded from your final **SAS datasets**.

Practical Application: Excluding Data with the NOT IN Operator

Let's revisit our `my_data` dataset to demonstrate the practical effect of exclusion filtering. Building on the previous example, suppose our analytical focus shifts from including teams A, B, and E to analyzing all teams *except* those three. This scenario, common in comparative **data analysis**, requires an efficient method to define the unwanted group and discard them from the resulting selection. The **NOT IN operator** is perfectly suited to handle this requirement with minimal code complexity.

We achieve this efficient exclusion by simply modifying our previous **query** structure to include `NOT` before `IN`. This tells **PROC SQL** to identify and return every row where the value in the `team` column does not match any of the entries in the provided list ('A', 'B', 'E'). This method ensures that the remaining teams--C and D, in this case--are isolated for focused examination, providing a clear demonstration of exclusion criteria in action.

```
/*select all rows where team is not A, B, or E*/
proc sql;
select *
from my_data
where team not in ('A', 'B', 'E');
quit;
```

team	points
C	35
C	36
C	40
D	28

As anticipated, the output demonstrates that only the rows where the team is not A, B, or E are returned, confirming the effectiveness of the **NOT IN operator** in filtering out undesired values. This result provides a clean and focused subset of the original **SAS dataset** for further **analysis** or reporting. This precision in exclusion is a cornerstone of robust **data management**.

Key Considerations and Best Practices

While the **IN** and **NOT IN operators** offer significant advantages in terms of query readability and simplicity, leveraging them optimally in **PROC SQL** requires adherence to specific best practices,

particularly regarding data integrity and performance efficiency. Addressing these technical nuances ensures that your queries are not only clean but also robust and fast, even when dealing with large volumes of information.

One critical aspect to manage, especially with the **NOT IN operator**, is the presence of **missing values**. In **SQL** logic, a **NULL** value (representing missing data) cannot be definitively evaluated as being either equal to or not equal to any value in the exclusion list. Consequently, the **NOT IN operator** will automatically exclude rows where the column being filtered contains a **NULL**. If your intention is to include these **missing values** in the final result set, you must explicitly add an `OR column_name IS NULL` condition to the **WHERE clause**.

Performance is another crucial factor. While the **IN operator** is efficient for small to moderately sized lists, handling extremely long lists of values can potentially impact query execution speed. While **SAS** is highly optimized, for scenarios involving thousands of values, analysts should evaluate alternative approaches, such as performing a **JOIN** operation with a pre-filtered temporary table or leveraging other high-performance **SAS procedures** optimized for large-scale filtering, to ensure faster processing times.

Furthermore, the **IN operator** is frequently paired with **subqueries**, a feature that allows the list of values to be dynamically generated by executing another **query**. This dynamic generation capability is invaluable for complex **data manipulation** tasks where the filtering criteria are dependent on the current state of the data. However, careful consideration must be given to the performance of nested **subqueries**, especially when used in conjunction with the **NOT IN operator**, as they can sometimes be less efficient than using `NOT EXISTS` in traditional database environments, although **SAS** generally manages this complexity well.

Handling Missing Values: Always account for **missing values** when using **NOT IN** by explicitly adding `OR column_name IS NULL` if you intend to include records where the column value is unknown.
Performance with Large Lists: For lists exceeding hundreds of values, assess potential performance impacts and consider using a **JOIN** against a small table instead of an inline list within the **IN operator**.
Data Type Consistency: Ensure strict consistency between the **data type** of the column being filtered and the **data type** of the values provided in the list to prevent unexpected logical outcomes or errors.
Subqueries with IN/NOT IN: Leverage **subqueries** for dynamically generated lists, but monitor execution time to ensure efficiency, especially in production environments.

Conclusion and Further Exploration

The **IN operator** and its crucial logical counterpart, the **NOT IN operator**, are fundamental tools

within the **Structured Query Language** environment, and their powerful integration into **PROC SQL** significantly enhances **SAS** programming efficiency. These operators offer a clean, elegant, and highly efficient means of managing list-based comparisons, enabling complex **data filtering** tasks--from targeted **data selection** to comprehensive **data cleaning**--with clarity that traditional OR/AND logic cannot match.

We have provided detailed examples demonstrating the construction and execution of **PROC SQL** queries that successfully include or exclude specific values from a sample **SAS dataset**. We also emphasized crucial operational considerations, such as the proper management of **missing values** and the importance of **data type** consistency, which are vital for generating accurate and reliable analytical results. By mastering these operators, you significantly elevate your proficiency in **SAS programming** and improve overall **data management** practices within the **SAS** framework.

To continue advancing your skills beyond basic filtering, we strongly encourage exploring advanced **query language** concepts, including complex table **joins**, efficient **aggregate functions**, and advanced usage of **subqueries**. The official **SAS documentation** is an invaluable source for detailed technical guidance and insights into maximizing the performance of your **SAS programming** endeavors.

Additional Resources

For those seeking to expand their foundational knowledge of the **SAS** System and the intricacies of **PROC SQL**, the following curated resources offer valuable documentation, tutorials, and community support:

SAS PROC SQL User's Guide: The official comprehensive guide detailing all features and syntax of **PROC SQL**.
SQL - Wikipedia: A foundational reference offering a general overview of the **Structured Query Language** principles.
SAS Free Training: Access valuable complimentary courses and educational materials provided directly by **SAS**.
SAS Support Communities: A platform to collaborate with fellow **SAS** users and engage with expert discussions for troubleshooting and advanced techniques.
SQL IN Operator - W3Schools: A concise, web-based tutorial illustrating the function of the **IN operator** in general **SQL** contexts.