How to Easily Filter SAS PROC SQL Queries with the IN Operator

Authored by

stats writer

November 22, 2025

The IN operator is a fundamental logical construct within Structured Query Language (SQL), and its implementation in SAS via the PROC SQL procedure is essential for efficient data retrieval. This powerful operator allows users to test whether a specified value matches any item present in a predefined list of values. Functionally, it serves as a condensed version of multiple **OR** conditions.

When executed, the **IN** operator evaluates the criteria and returns a Boolean result--either **TRUE** or **FALSE**. If the tested value is found within the supplied list, the result is TRUE, and the corresponding record is included in the output. Conversely, if the value is not found, the result is FALSE, and the record is excluded. Its primary role is to drastically simplify the syntax required for filtering results in the **WHERE** clause of a query, making complex data subsetting operations cleaner and more readable.

Understanding the proper application of the **IN** operator is crucial for any data analyst working within the SAS environment, particularly those utilizing the SQL pass-through or the native **PROC SQL** procedure. This guide delves into the specific implementation, providing clear examples and detailing best practices for both the positive (**IN**) and negative (**NOT IN**) applications of this versatile operator.

When working with large datasets in **SAS**, efficient data subsetting is paramount. The primary utility of the **IN** operator within the **PROC SQL** statement is to selectively return rows only when a target variable's value is present within a specified list of acceptable values. This method significantly enhances query performance and maintainability compared to chaining numerous disjunctive conditions.

The following detailed examples illustrate the practical implementation of the **IN** operator, demonstrating how to construct a query that leverages this powerful filtering mechanism effectively.

Understanding the IN Operator Syntax and Logic

The syntax for the **IN** operator is straightforward, residing primarily within the **WHERE** clause of an SQL statement. The general structure requires specifying the column name to be evaluated, followed by the keyword **IN**, and finally, a parenthetical list containing the values for comparison. These values must be enclosed in single quotes if they are character strings, or left unquoted if they are numeric values.

Logically, the operator performs a sequential comparison: `variable IN (value1, value2, value3)` is functionally equivalent to `variable = value1 OR variable = value2 OR variable = value3`. The use of **IN** drastically improves the readability of the query, especially when the list of values grows large. Furthermore, database optimization engines often process the **IN** clause more efficiently than a lengthy series of **OR** clauses, potentially yielding performance benefits.

It is important to ensure that the data types of the column being filtered and the list of values provided are consistent. Mismatches in data type (e.g., comparing a numeric column against a list of character strings) will typically result in errors or unintended behavior within the **PROC SQL** environment. Proper handling of quotation marks for character variables is a key consideration for successful execution.

Setting up the Example Dataset in SAS

To demonstrate the functionality of the **IN** operator, we first establish a sample dataset in **SAS**. This dataset, named **my_data**, contains fictional information relating to basketball players, specifically tracking their team affiliation and points scored. This simple structure allows for clear visualization of how the filtering process works.

We use the standard SAS **DATA step** combined with the **DATALINES** statement to input the sample records quickly. The variables defined are **team** (a character variable denoted by **$**) and **points** (a numeric variable). Once the dataset is created, we use **PROC PRINT** to verify its structure and content before moving on to the SQL query.

```
/*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;
```

The execution of this DATA step creates the foundation for our analysis, containing twelve observations across five distinct teams (A, B, C, D, E). The following image represents the output generated by the **PROC PRINT** statement, confirming the successful creation of the dataset:

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

Practical Application of the IN Operator

Now that the dataset is ready, we can use the **IN** operator within **PROC SQL** to perform targeted data extraction. Our objective here is to retrieve only the records associated with teams A, B, or E. This is a common filtering task where we need to isolate data points belonging to a specific subset of categories.

The **PROC SQL** statement simplifies this process greatly. By placing the condition **WHERE team IN ('A', 'B', 'E')**, we instruct SAS to check each row: if the value in the **team** column is one of the three specified characters, that row is selected. Note the use of single quotes around the team letters because **team** is a character variable.

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

Executing this query results in a filtered table containing only the rows matching the specified team affiliations. This is a highly efficient way to subset data, avoiding the verbose syntax required by using multiple **OR** conditions.

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

The resulting output clearly demonstrates the filtering effect: only the rows where the **team** variable equals A, B, or E are returned, confirming the successful operation of the **IN** operator.

The Complementary Operator: NOT IN

The logical opposite of the **IN** operator is NOT IN. While **IN** selects rows where a variable matches an item in the list, **NOT IN** selects rows where the variable specifically does **not** contain any value found within the supplied list. This is exceptionally useful for exclusion filtering, allowing analysts to isolate data that falls outside a particular set of criteria.

Using the same dataset, we can employ the **NOT IN** operator to select all rows where the team is **not** equal to A, B, or E. Essentially, we are asking the query to return only the records associated with teams C and D.

```
/*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;
```

This application demonstrates the power of exclusion filtering. Instead of explicitly listing all teams

we want (C, D), we list the teams we wish to exclude (A, B, E), often simplifying the query logic, especially when the excluded list is smaller than the included list.

team	points
C	35
C	36
C	40
D	28

As anticipated, the resulting table contains only the records for teams C and D. This confirms that **NOT IN** correctly filters out any observation whose team value matches one of the values listed in the parentheses.

Advanced Usage: Combining IN with Subqueries

One of the most powerful uses of the **IN** operator is its application alongside a Subquery (or nested query). Instead of hardcoding a static list of values, you can dynamically generate the comparison list from the results of another SQL query executed against the same or a different dataset. This allows for highly flexible and conditional filtering based on real-time data analysis.

For example, imagine you needed to select all players whose team scored an average of more than 30 points. You would first use a subquery to calculate the average points per team and filter for teams meeting the 30-point threshold. The outer query would then use the **IN** operator to select all rows whose team name is present in the list returned by the subquery.

The structure typically looks like this: `WHERE variable IN (SELECT distinct values FROM another_table WHERE condition)`. When employing subqueries, performance becomes a critical factor. It is crucial to ensure that the subquery is optimized, as it must execute fully before the outer query can proceed with the filtering operation. **PROC SQL** handles these nested operations efficiently, but complex, poorly indexed subqueries can significantly slow down processing time.

Performance Considerations and Alternatives

While the **IN** operator is often much faster and cleaner than a long string of **OR** conditions, analysts must be aware of its operational characteristics, especially when dealing with extremely large lists or null values.

When the list inside the **IN** clause is static and small, performance is excellent. However, if the list contains thousands of distinct values (especially when derived from a subquery), the SAS engine must perform extensive lookups, which can impact speed. In some extreme cases, particularly in specific database management systems (though less common in native SAS), joining the main table to the derived list table might be more efficient than using a lengthy **IN** clause.

A crucial detail regarding the **NOT IN** operator is its interaction with null values. If the list provided to **NOT IN** contains even a single null value, the entire condition will evaluate to unknown (missing), meaning no rows will be returned. This is because SQL cannot definitively determine if a value is not equal to an unknown value (null). Analysts must proactively handle nulls--either by excluding them from the comparison list or by using explicit checks like `WHERE variable NOT IN (...) AND variable IS NOT NULL`.

Advantages of Using IN Over OR Conditions

Using the **IN** operator provides several distinct advantages over relying solely on repeated **OR** conditions, contributing to better code quality and processing efficiency in **PROC SQL**.

Improved Readability: A list of 10 or 20 values is significantly easier to read and verify when contained within parentheses using **IN**, rather than spread across multiple lines linked by **OR**.

Reduced Coding Errors: The compact syntax reduces the likelihood of syntax errors, such as misplaced parentheses or missing keywords, which are common when constructing complex chains of **OR** conditions.

Optimized Execution: Most SQL query optimizers, including the one used by **PROC SQL**, are designed to handle the **IN** operator very efficiently, often converting it internally into highly optimized lookup tables or hash joins, which are typically faster than sequential evaluation of multiple **OR** conditions.

Dynamic Filtering Capability: As highlighted earlier, **IN** integrates seamlessly with subqueries, enabling dynamic list generation. This capability is cumbersome or impossible to replicate using simple **OR** logic.

Summary of Best Practices for IN and NOT IN

To ensure robust and efficient data querying using the **IN** and **NOT IN** operators in **SAS PROC SQL**, adherence to specific best practices is recommended. These guidelines help maintain code integrity and optimize processing time.

Match Data Types: Always verify that the data type of the column being filtered matches the data

type of the values in the list. Character values require single quotes; numeric values do not.

Handle Nulls with NOT IN: When using **NOT IN**, explicitly manage null values in the target column by adding an `IS NOT NULL` clause to prevent the query from returning zero records unintentionally.

Keep Lists Manageable: If the list of values becomes excessively large (e.g., thousands of unique IDs), consider whether restructuring the query using a **JOIN** to a temporary table containing the list of IDs might offer better performance.

Optimize Subqueries: Ensure that any subquery feeding into an **IN** clause is highly optimized, particularly by indexing the columns used in the subquery's **WHERE** and **SELECT** clauses.

Mastering the **IN** operator is essential for writing professional and effective SQL code within the SAS environment, enabling precise and flexible data manipulation.