Learning to Filter Rows by String Content in SAS Datasets

Authored by

Mohammed looti

November 1, 2025

Introduction to String Filtering in SAS

When performing data analysis using the **SAS system**, one of the most frequent tasks involves subsetting large tables to focus on specific records. Efficiently extracting relevant data rows based on the presence of particular character strings within variables is a fundamental skill for any data professional working with **SAS datasets**. This process, commonly known as **filtering**, allows analysts to refine their data scope, leading to faster processing and more focused results. This comprehensive tutorial introduces two distinct and highly effective methods for performing this crucial string-based filtering operation within the SAS environment, highlighting the difference between searching for partial substrings and matching against exact values in a list.

The core mechanism for implementing these filters is the **WHERE clause**, which is universally applicable across various SAS procedures, including the **DATA step** and **PROC SQL**. Understanding which operator to deploy--whether it is the `CONTAINS` operator for loose matching or the `IN` operator for strict list comparison--is essential for writing clean, performant, and reliable **SAS programming** code.

Method 1: Utilizing the CONTAINS Operator for Substring Matching

The most straightforward and flexible approach for identifying rows that contain a specific partial string, or **substring**, within a character variable is by using the **CONTAINS operator**. This operator is inherently designed for pattern matching, searching the entire content of the specified variable for the presence of the defined string literal. It is crucial to remember that by default, the **CONTAINS operator** is case-sensitive. If your data contains mixed capitalization, you might need to employ functions like `UPCASE()` or `LOWCASE()` on the variable within the **WHERE clause** to ensure comprehensive filtering.

This first example demonstrates the basic syntax for implementing the `CONTAINS` operator within a **DATA step**. We are creating a new dataset, `specific_data`, by reading from `original_data` and applying the filter condition. This technique is highly effective when you do not know the exact formatting of the field but need to confirm the existence of a specific component, such as an abbreviated product code or a common location name, regardless of where it appears within the string.

```
/*filter rows where var1 contains "string1"*/
data specific_data;
set original_data;
where var1 contains 'string1';
run;
```

When utilizing the **CONTAINS operator**, it is important to be mindful of its performance characteristics. While it is extremely versatile, for very large datasets, using indexed variables and simpler comparison operators might sometimes yield faster results. However, for general-purpose substring searching, especially when the search term is consistently short or non-indexed, `CONTAINS` offers an unparalleled balance of simplicity and functionality. For more complex pattern matching involving wildcards, SAS also provides the `LIKE` operator, which complements `CONTAINS` by offering greater control over positional matching.

Practical Demonstration: Setting Up the Sample Dataset

To properly illustrate the functional differences between the `CONTAINS` and `IN` operators, we must first establish a reproducible sample dataset. For this purpose, we will create a small table named `nba_data`, which contains a list of fictional basketball team names and their corresponding scores. This dataset provides a simple yet effective context for demonstrating both partial string searches and exact list matching.

The following **DATA step** utilizes the `DATALINES` statement to quickly populate the dataset with the necessary records. This method is often preferred for rapid development and demonstration purposes. Following the data creation, we use `PROC PRINT` to display the initial structure and contents of the newly created `nba_data`, allowing us to visualize the data before any **filtering** operations are applied.

```
/*create dataset*/
data nba_data;
input team $ points;
datalines;
Mavs 95
Spurs 99
Warriors 104
Rockets 98
Heat 95
Nets 90
Magic 99
Cavs 106
;
run;

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

Obs	team	points
1	Mavs	95
2	Spurs	99
3	Warriors	104
4	Rockets	98
5	Heat	95
6	Nets	90
7	Magic	99
8	Cavs	106

As shown in the output above, the `nba_data` dataset is ready. It consists of eight observations, each with a character variable named `team` and a numeric variable named `points`. The examples that follow will use the `team` variable as the target for our string **filtering** criteria, demonstrating how to isolate specific teams based on the text they contain.

Executing Substring Search with CONTAINS

Now, we apply the **CONTAINS operator** to the `nba_data` table. Our goal is to retrieve all team records that contain the specific, three-letter substring "avs" anywhere within the team name. Because the **CONTAINS operator** searches for the pattern regardless of its position, this single filter condition successfully captures both 'Mavs' (where 'avs' is internal) and 'Cavs' (where 'avs' is internal), assuming case sensitivity is maintained as per the original data structure.

The following code snippet executes the subsetting logic within a **DATA step**. The resulting dataset, `specific_data`, will contain only the records that satisfy the condition defined in the **WHERE clause**, demonstrating the powerful ability of SAS to quickly isolate subsets based on fuzzy or partial text matches. The subsequent `PROC PRINT` step confirms the outcome.

```
/*filter rows where team contains the string 'avs'*/
data specific_data;
set nba_data;
where team contains 'avs';
run;

/*view resulting rows*/
proc print data=specific_data;
```

Obs	team	points
1	Mavs	95
2	Cavs	106

The resulting dataset successfully identifies and isolates the two records where the `team` column contains the partial string 'avs'. This result confirms that the **CONTAINS operator** is the appropriate tool for scenarios requiring partial string matching, contrasting sharply with operators that require strict equality.

Method 2: Precision Filtering with the IN Operator

In contrast to searching for substrings, data filtering often requires checking if a variable's value is an exact match to one of several predefined values. When this goal is to match rows against a finite and specific list of exact character strings, the **IN operator** provides the most concise, readable, and often the most efficient syntax. Instead of writing a cumbersome series of `OR` conditions (e.g., `WHERE var1 = 'A' OR var1 = 'B' OR var1 = 'C'`), the **IN operator** allows the developer to list all acceptable values within a single set of parentheses.

The primary characteristic of the **IN operator** is that it demands an exact, full match for the entire string. If the data value is 'Mavs ' (with a trailing space) and the list includes only 'Mavs' (without a space), the record will not be selected. This strict matching makes it ideally suited for working with categorical data, predefined codes, or look-up lists where precision is paramount. Furthermore, SAS handles the internal optimization of the `IN` operation, which often results in superior performance compared to manually chained `OR` comparisons, especially when the list of values is extensive.

Understanding the distinction between `CONTAINS` and `IN` is critical for effective **SAS programming**. While `CONTAINS` looks for a pattern *within* the string, `IN` checks if the string is *identical* to one of the specified list members. This fundamental difference dictates which operator should be selected based on the specific business requirement for **filtering**.

Detailed Implementation of the IN Operator

We will now implement the **IN operator** using our `nba_data` dataset. In this example, we aim to filter the dataset to include only teams that are exactly 'Mavs', 'Nets', or 'Rockets'. Notice that we must provide the exact capitalization and spelling for each team name within the parentheses, separated by commas.

The following **DATA step** demonstrates the concise syntax of the `IN` operator. This code

generates a new dataset containing only the observations that meet this strict list-based criterion. It is a highly readable method that clearly communicates the intent to select records based on a known set of acceptable values, which significantly enhances the maintainability of the **SAS programming** script.

```
/*filter rows where team contains the string 'Mavs', 'Nets', or 'Rockets'*/
data specific_data;
set nba_data;
where team in ('Mavs', 'Nets', 'Rockets');
run;

/*view resulting rows*/
proc print data=specific_data;
```

Obs	team	points
1	Mavs	95
2	Rockets	98
3	Nets	90

As clearly demonstrated by the output, the dataset has been successfully filtered down to include only the three specified team names. This showcases the effectiveness of the **IN operator** for list-based exact matching, providing a cleaner alternative to multiple equality checks. For those working with numeric data, the `IN` operator functions identically, allowing lists of numbers to be checked efficiently within the **WHERE clause**.

Choosing the Right Operator: CONTAINS vs. IN

Effective data manipulation in SAS hinges on selecting the appropriate tool for the job. Both the **CONTAINS operator** and the **IN operator** offer powerful methods for subsetting **SAS datasets** based on character criteria, but their application contexts are fundamentally different. The primary factor in your decision should be whether you require a partial match (a substring existing anywhere within the field) or an exact match against a predefined list of valid values.

Use the **CONTAINS operator** when your search term is a fragment of the data you expect to find. This is useful for searching memo fields, descriptive text, or when dealing with inconsistent data where full matches cannot be guaranteed. Remember to handle case sensitivity explicitly if necessary, often by wrapping the target variable in a function like `UPCASE()` to standardize the comparison. For instance, `WHERE UPCASE(description) CONTAINS 'ERROR';` ensures that

capitalization differences do not impede your **filtering**.

Conversely, rely on the **IN operator** when you have a definitive, finite list of acceptable values. This method is superior for validating fields against standard codes (e.g., state abbreviations, product types) or when migrating conditional logic from chained `OR` statements. Utilizing `IN` not only improves the readability of your **DATA step** or `PROC SQL` query but also often benefits from internal SAS performance optimizations. Mastering these two operators ensures that your **SAS programming** solutions are both accurate and efficient.