Learning to Filter Data with the LIKE Operator in SAS PROC SQL

Authored by

Mohammed looti

April 28, 2026

In the highly specialized realm of data management and sophisticated statistical analysis using **SAS**, the capacity for effective and nuanced data filtering stands as a paramount skill. While simple equality checks are sufficient for exact matches, real-world data frequently necessitates the identification of patterns, substrings, or variations within text fields. This is precisely where the **LIKE operator**, utilized within **PROC SQL**, proves indispensable. This powerful tool liberates analysts from the constraints of exact string comparison, allowing them to select rows based on whether a specified column's value adheres to a defined string pattern.

This comprehensive guide is designed to serve as an in-depth reference for leveraging the **LIKE operator** within the **SAS PROC SQL** environment. We will meticulously break down the mechanics of pattern construction, focusing on the essential role of **wildcard characters**. Furthermore, we will explore the critical functionality of the **NOT LIKE operator**, which provides the necessary inverse logic for defining exclusion criteria, ensuring that users can achieve precise control over their resulting datasets. Through clear, actionable examples, you will learn how to transition from basic filtering to advanced pattern recognition, drastically enhancing your data manipulation capabilities in **SAS**.

Mastering Wildcard Characters for Precise Pattern Matching

The operational efficiency and flexibility of the **LIKE operator** are entirely dependent upon its strategic integration with **wildcard characters**. These special symbols are not merely placeholders; they are critical components that enable the definition of patterns representing a wide spectrum of possible string values, contrasting sharply with the rigidity of a single, exact string match. Proficiency in using these characters is mandatory for crafting precise and efficient pattern-matching queries in **PROC SQL**.

In the context of **SAS PROC SQL**, two primary **wildcard characters** govern pattern construction:

% (Percent Sign): This is the most frequently used wildcard, representing any sequence of zero or more characters. Its power lies in its ability to match arbitrary lengths of strings. For example, using the pattern 'B%' will select any string beginning with 'B' (e.g., 'Basketball', 'B', 'Bears'). Conversely, '%ALL' will match any string that concludes with 'ALL' (e.g., 'FOOTBALL', 'CALL'). If you require a string that merely contains a specific sequence, such as '%WORD%', the query will successfully identify 'MYWORDISHERE' or 'WORD'.
_ **(Underscore)**: This character is designed for exact positional matching, as it represents precisely one single character. This offers a level of granularity that the percent sign cannot provide. Consider the pattern 'S_L'. This pattern would successfully match 'SAL', 'SCL', or 'S9L', but it would explicitly fail to match 'SL' (zero characters) or 'SOOL' (two characters between S and L). The underscore is invaluable when searching for strings of a known, fixed length that contain minor variations.

Developing a sophisticated understanding of how to combine and deploy these **wildcard characters** is crucial. They are the mechanism by which you can search for partial strings, enforce specific positional requirements, or define complex combinatorial patterns, thus rendering your data selection highly adaptable to diverse analytical requirements.

Establishing the Sample Dataset for Demonstration

To comprehensively demonstrate the functional utility of the **LIKE operator**, we must first establish a robust, yet straightforward, sample dataset within the **SAS** environment. This dataset, which we will name **my_data**, contains essential information pertaining to a fictional roster of basketball players, including their respective team affiliations and the number of points they scored during a specified period. This data will serve as the foundation for all subsequent pattern-matching operations.

The creation of this data structure is achieved through the utilization of the **DATA step** code provided below. Following the data creation, a **PROC PRINT** statement is executed. This step is critical for verifying the integrity of the input data and providing a visual confirmation of the dataset's contents, allowing analysts to clearly see the raw data before filtering begins. The structure includes both short and long team names, providing varied lengths upon which the pattern matching will be tested.

```
/*create dataset*/
data my_data;
input team $ points;
datalines;
Cavs 12
Cavs 14
Warriors 15
Hawks 18
Mavs 31
Mavs 32
Mavs 35
Celtics 36
Celtics 40
;
run;

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

Upon the successful execution of this script, the **PROC PRINT** procedure furnishes the complete, unfiltered dataset output. This initial visualization is indispensable, as it allows us to confirm that the data is correctly structured and readily prepared for the detailed filtering operations that we will apply using the pattern-matching capabilities of **PROC SQL**. The following image represents the structure of the data we will be querying.

Obs	team	points
1	Cavs	12
2	Cavs	14
3	Warriors	15
4	Hawks	18
5	Mavs	31
6	Mavs	32
7	Mavs	35
8	Celtics	36
9	Celtics	40

Implementing the LIKE Operator for Substring Identification

With our sample data established, we can now proceed to implement the **LIKE operator**. Our specific analytical goal is to extract only those records from the **my_data** dataset where the 'team' variable explicitly contains the substring 'avs'. This type of operation is exceedingly common in data cleansing, where variations in naming conventions must be unified, or in targeted data extraction where only records containing a specific keyword are required.

We leverage the powerful querying environment of **PROC SQL** to construct a highly focused query. Central to this query is the pattern '%avs%'. The leading '%' wildcard signifies that any number of characters (including zero) can precede 'avs', and the trailing '%' indicates that any number of characters (including zero) can follow it. This ensures that 'avs' can be matched anywhere within the string, whether it is at the beginning, the end, or in the middle. This flexibility is a core strength of pattern matching over strict textual equality.

```
/*select all rows where team contains 'avs'*/
proc sql;
select *
from my_data
where team like '%avs%';
```

```
quit;
```

The execution of this structured query yields a precise subset of the original data. As expected, only the rows corresponding to 'Cavs' and 'Mavs' are returned, as both team names successfully contain the substring 'avs'. This result clearly demonstrates the ability of the **LIKE operator**, when appropriately combined with **wildcard characters**, to efficiently filter data based on the occurrence of partial strings. This is a crucial technique for analysts dealing with large text fields where complete string matching is impractical.

team	points
Cavs	12
Cavs	14
Mavs	31
Mavs	32
Mavs	35

Utilizing NOT LIKE for Defining Exclusion Criteria

While the **LIKE operator** is designed for pattern inclusion, data analysis often requires the ability to define and enforce exclusion criteria--that is, selecting all data records that explicitly do not match a specified pattern. For these scenarios, **SAS PROC SQL** provides the indispensable **NOT LIKE operator**. Functionally, **NOT LIKE** serves as the logical complement to **LIKE**, returning rows only when the specified variable's value fails to contain the defined string pattern.

To demonstrate this essential inverse functionality, we will employ the **NOT LIKE operator** to select every row from our **my_data** dataset where the 'team' variable explicitly does not contain the 'avs' pattern. This operation is highly valuable when isolating outliers, removing noise, or focusing analysis on groups that fall outside a specific classification or naming convention. The structure of the query is almost identical to the previous example, with the critical substitution of the operator.

```
/*select all rows where team does not contain 'avs'*/
proc sql;
select *
from my_data
where team not like '%avs%';
quit;
```

Upon execution, this query generates a result set that precisely excludes all teams containing 'avs'. This means that 'Cavs' and 'Mavs' are filtered out, leaving only 'Warriors', 'Hawks', and 'Celtics'. The **NOT LIKE operator** is an essential component in defining exclusion boundaries, providing analysts with the ability to define exactly what data should be omitted from their focus, thereby ensuring that the remaining data set is clean and relevant to the investigation.

team	points
Warriors	15
Hawks	18
Celtics	36
Celtics	40

Conclusion and Advanced Best Practices for SAS Pattern Matching

The **LIKE** and **NOT LIKE operators** within **SAS PROC SQL** represent foundational elements for executing sophisticated string pattern matching operations. They offer capabilities that extend significantly beyond simple equality checks, enabling analysts to accurately identify, select, include, or exclude data based on partial string matches and positional requirements. True mastery of these operators hinges on the strategic deployment of **wildcard characters**: the percent sign (**%**) for handling variable-length sequences of characters, and the underscore (_) for matching a single character in a specific position.

When incorporating these pattern-matching techniques into large-scale projects, adherence to certain best practices is highly recommended to ensure accuracy and performance. Firstly, always employ rigorous testing: define your pattern and test it on a small, controlled subset of your data to confirm that the results align with your expectations before deploying the query across an entire production dataset. Secondly, and perhaps most critically in string manipulation, you must remain acutely aware of case sensitivity. By default, string comparisons in **SAS** are case-sensitive; consequently, 'cavs' will not match 'Cavs'. To achieve case-insensitive matching, which is often necessary for robust data cleaning, always preprocess your variables using functions such as `UPCASE()` or `LOWCASE()` on both the variable and the pattern string within your query's `WHERE` clause. Implementing these techniques will significantly enhance the precision, reliability, and overall efficiency of your data manipulation tasks in **SAS**.

Further Exploration in SAS Data Manipulation

To continue deepening your expertise and expanding your operational capabilities within the **SAS**

programming environment, we highly recommend exploring tutorials that address other common and advanced data manipulation tasks. Building upon the solid foundation of pattern matching mastered through the use of **LIKE** and **NOT LIKE**, you can begin to integrate more sophisticated data processing techniques, such as conditional aggregation and complex join operations.

The following areas and resources provide further guidance on various advanced operations within **SAS**, serving as crucial steps toward continuously enhancing your programming skills, optimizing query performance, and achieving maximum efficiency in your statistical analysis projects: