Learning SAS: A Guide to Generating Sequential Row Numbers Using the MONOTONIC Function

Authored by

Mohammed looti

November 14, 2025

The **SAS** programming environment is renowned for its powerful capabilities in statistical analysis and data manipulation. A fundamental requirement for effective data preparation and auditing is the ability to accurately track and manage the ordinal position of observations. While often overlooked, the zero-argument **MONOTONIC()** function serves as a crucial, specialized tool within this ecosystem. This function is expertly designed to dynamically generate unique, sequential **row numbers** within any resultant **dataset**, providing a simple yet powerful mechanism for indexing records.

For analysts and programmers, understanding how to effectively deploy **MONOTONIC()** is paramount, especially when the goal is to assign persistent identifiers or select data subsets based purely on their position rather than their inherent data values. This positional indexing is invaluable in many operational contexts, including advanced sampling techniques, creating data pagination schemas, or guaranteeing the reproducible ordering of analytical results. By adopting this function, practitioners can dramatically simplify data preparation tasks that traditionally require cumbersome management of manual counter variables or complex iterative code structures.

This guide offers a dedicated exploration of the practical applications of the **MONOTONIC()** function, concentrating specifically on its deployment within the **PROC SQL** procedure. We will meticulously examine two primary use cases: first, incorporating the function into the query output to create a dedicated column for sequential identifiers; and second, leveraging it directly within the **WHERE clause** to filter observations based on their exact positional index. The goal is to provide readers with detailed, reproducible examples necessary to seamlessly integrate **MONOTONIC()** into their professional SAS workflows.

Understanding the MONOTONIC() Function in PROC SQL

It is important to recognize that the **MONOTONIC()** function is not a universally available function across all SAS processes, such as the standard **DATA step**. Instead, it is a specialized feature intrinsically linked to the processing logic of **PROC SQL**. Its core mission is straightforward: to assign a unique, positive integer identifier to every single row that the procedure reads and processes during the execution of a query. This integer sequence strictly commences at 1 for the first observation retrieved from the source table and increments reliably by one for each subsequent observation, continuing throughout the entire query execution cycle.

What differentiates **MONOTONIC()** from manually constructed sequence numbering is that its management is handled entirely by the SQL processing engine itself. This characteristic makes it an inherently efficient and reliable mechanism for assigning a persistent, ordered identifier to observations as they are streamed from the underlying data source, whether it is a physical table or a view. This automated efficiency often makes it a superior choice compared to attempting to implement traditional SQL window functions or complex iterative logic, especially when the

fundamental requirement is merely a sequential count of records based on their current retrieval order.

The principal benefit of leveraging this function is the inherent simplicity it introduces to otherwise complex data management tasks. By abstracting the counting mechanism, programmers are liberated from the need to manually manage the initialization, incrementing, and resetting of counter variables--steps that are frequently required in traditional **SAS** programming structures. This efficiency is particularly valuable when analysts are dealing with massive volumes of data where minimizing processing overhead is critical. Therefore, **MONOTONIC()** becomes the default choice when the ordinal position of the data record--that is, its sequence in the final result set--is the key metric for subsequent analysis or operations.

Preparing the Demonstration Dataset

To ensure a crystal-clear, practical demonstration of how the **MONOTONIC()** function operates in a real-world context, we must first establish a reproducible sample **dataset**. For the purposes of this tutorial, we will construct a small table named `my_data`, designed to simulate typical performance statistics for several fictional basketball teams. This structure, which includes team names, points scored, and assists recorded, provides a realistic and relatable context for applying and visualizing the effects of row sequencing.

The construction of this sample data is performed using a standard **DATA step**, which ensures that all readers can easily replicate the environment and follow along precisely with the provided examples. Immediately following the data creation, we employ the `PROC PRINT` procedure to display the initial contents of the table. This step is vital as it allows us to confirm the data structure and, crucially, the default order of the nine observations before we apply any SQL transformation or sequencing logic.

```
/*create dataset*/
data my_data;
input team $ points assists;
datalines;
Cavs 12 5
Cavs 14 7
Warriors 15 9
Hawks 18 9
Mavs 31 7
Mavs 32 5
Mavs 35 3
Celtics 36 9
```

```
Celtics 40 7
;
run;

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

Once the preceding code snippet is executed, the `my_data` table is successfully instantiated, containing nine distinct records. The subsequent execution of `PROC PRINT` visually confirms the initial state of our data, as demonstrated in the output image below. It is important to remember that the order shown here represents the implicit input order, which the **MONOTONIC()** function will use as its foundational reference sequence in all subsequent steps unless explicit sorting is applied.

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

As clearly illustrated, the sample **dataset** includes three key variables: `team` (a character variable), and `points` and `assists` (both numeric variables). This well-defined foundational structure provides the ideal environment for showcasing precisely how **MONOTONIC()** can be applied to append or reference sequential identifiers without making any permanent alterations to the original underlying data values.

Method 1: Generating a Column of Row Numbers with MONOTONIC()

The most conventional and widely used application of the **MONOTONIC()** function is the explicit incorporation of the sequential count directly into the structure of the resulting table. This technique is absolutely essential when an analyst requires a unique, ordered ID for every single record, serving purposes such as facilitating subsequent data merging, meeting internal reporting

requirements, or simply ensuring the observation sequence is preserved upon data export or long-term storage. By generating this dedicated column, the row position is converted into a persistent, addressable, and queryable variable.

To successfully execute this operation, we must embed the `monotonic()` function call directly within the **SELECT statement** of our **PROC SQL** query. It is critical to assign an alias to the function call using the `AS` keyword, as this alias dictates the name of the newly created column that will house the sequential **row number**. In the demonstration below, we select the alias `row_ID` to clearly communicate the function's intended purpose to anyone reviewing the code.

```
/*create column called row_ID that contains row numbers*/
proc sql;
select team, monotonic() as row_ID
from my_data;
quit;
```

Upon execution of this query, **PROC SQL** initiates the process of streaming the `my_data` table row by row. For every row retrieved via the **FROM clause**, the **MONOTONIC()** function is evaluated, incrementing its internal counter and assigning that current value to the `row_ID` variable in the output. The resulting table elegantly contains the selected original variables displayed alongside the newly engineered sequential identifier column, providing a clear visual mapping of each record's positional index.

team	row_ID
Cavs	1
Cavs	2
Warriors	3
Hawks	4
Mavs	5
Mavs	6
Mavs	7
Celtics	8
Celtics	9

The output image above definitively confirms the successful implementation of this technique. The `row_ID` column now sequences precisely from 1 through 9, corresponding exactly to the nine observations present in the original data. This method represents a clean, declarative, and highly

efficient approach within the SQL paradigm for assigning unique, ordered identifiers to every record, thereby significantly enhancing the overall analytical utility of the **dataset**.

Method 2: Filtering Observations by Ordinal Position

The robust utility of the **MONOTONIC()** function extends significantly beyond mere column generation; it also functions dynamically when integrated directly into conditional logic. Specifically, leveraging **MONOTONIC()** within the **WHERE clause** of a **PROC SQL** query unlocks powerful capabilities for positional filtering. This capacity is invaluable when analysis demands the selection of records based solely on their sequential index--for instance, retrieving the first N observations, excluding initial header rows, or implementing fundamental data pagination protocols.

To demonstrate this essential filtering capability, let us address the requirement of isolating only an initial subset of records from our established `my_data` table. Our specific goal is to retrieve only those observations whose sequential position (or row index) is strictly less than 5. This operation is designed to effectively select the first four rows, clearly illustrating the function's ability to serve as a dynamic index for highly precise subsetting operations. The following **SAS** code snippet encapsulates this exact positional filtering logic:

```
/*filter where row number is less than 5*/
proc sql;
select *
from my_data
where monotonic() < 5;
quit;
```

In this query, the **SELECT * statement** is utilized to retrieve all variables available from the source table. However, the operational heart of this query resides within the **WHERE clause** condition: `monotonic() < 5`. As **PROC SQL** iterates through the rows, the **MONOTONIC()** function generates the current row index dynamically. Only records where this index satisfies the inequality are retained and presented in the final result set. This provides a clean, native SQL method for managing positional subsetting, completely bypassing the need for dedicated index variables to be present in the source data.

team	points	assists
Cavs	12	5
Cavs	14	7
Warriors	15	9
Hawks	18	9

As confirmed by the output image above, the resulting dataset now exclusively contains the first four observations from the original `my_data` table. This powerful filtering capability demonstrates how **MONOTONIC()** can be used to exert precise control over which records are included in your analysis based purely on their ordinal position. It offers an efficient and flexible mechanism for subsetting data in a manner that is often significantly more challenging when relying solely on traditional, content-based filtering conditions.

Advanced Considerations: Ordering and Performance

While the **MONOTONIC()** function delivers substantial advantages through its simplicity and processing efficiency, users must maintain an acute awareness of its fundamental reliance on the internal processing order of **PROC SQL**. By default, **MONOTONIC()** assigns numbers based strictly on the physical sequence in which rows are retrieved from the underlying table, which may not necessarily correspond to any desired logical or analytical sort order.

If the analytical requirement dictates that the sequential ID must align with a specific sorting criterion--for instance, ranking basketball teams by their points scored--it is absolutely essential to introduce an explicit sorting mechanism into the query logic. This critical alignment is achieved by deliberately combining **MONOTONIC()** with the **ORDER BY clause**. To guarantee that the function assigns numbers based on the sorted sequence rather than the arbitrary input sequence, the sorting operation must logically occur and be completed **before** the **MONOTONIC()** function is evaluated.

The standard best practice for achieving this involves using a nested query structure, typically a subquery or a temporary view. The inner query first handles the necessary sorting of the data using the **ORDER BY clause**, and the outer query then applies the **MONOTONIC()** function to the resulting, already-sorted set. This rigorous approach ensures that the generated **row number** accurately reflects the desired analytical rank or order. Failure to implement this explicit ordering when needed will result in an arbitrary sequencing dictated by system retrieval methods, which will inevitably lead to misleading or invalid analysis.

Furthermore, from a performance standpoint, **MONOTONIC()** is generally highly efficient,

especially when compared against implementing complex analytical window functions or designing iterative loops using the **SAS** macro language. It minimizes computational overhead by efficiently leveraging the optimized SQL engine for sequential counting. This makes it a robust and scalable solution for managing data integrity and assigning unique identifiers, even when processing significantly large SAS tables where performance is a paramount concern.

Conclusion

The **MONOTONIC()** function stands out as a powerful and highly elegant feature within the **SAS** environment, offering an exceptionally efficient method for managing and referencing sequential **row numbers** when executing queries via **PROC SQL**. As demonstrated through practical examples, its capabilities range from generating persistent, unique identifiers within a new output column to facilitating precise positional filtering via the **WHERE clause**, enabling sophisticated and dynamic data subsetting.

By mastering the seamless integration of **MONOTONIC()**, data professionals can substantially enhance both the clarity and the performance of their SAS code. It provides a reliable, SQL-native alternative to convoluted counter logic, ensuring that fundamental tasks such as extracting the top N records, assigning unique sequence numbers, or organizing data based on ordinal position are executed with maximum efficiency and minimal coding complexity. We highly recommend the consistent application of this specialized function to streamline data preparation stages and ensure the generation of robust and trustworthy analytical results.

Additional Resources for SAS Programming

To further advance your expertise in data manipulation and sophisticated programming techniques within the SAS ecosystem, consider exploring the following authoritative resources:

SAS Documentation: Base SAS Procedures Guide
SAS Tutorials and Learning Resources
Wikipedia: Structured Query Language (SQL)