SAS: Use SELECT DISTINCT in PROC SQL what is the procedure to use SELECT DISTINCT in PROC SQL?

Authored by

stats writer

November 19, 2025

The ability to handle and manage data duplicates is a foundational skill for any data analyst working within the SAS environment. When executing complex queries, particularly during data preparation or reporting phases, ensuring the resulting dataset contains only unique observations is often paramount to accurate analysis. This is precisely where the **SELECT DISTINCT** clause, utilized within PROC SQL, proves indispensable. It serves as a powerful filter, allowing users to efficiently prune redundant information from the output of their query.

The fundamental mechanism of SELECT DISTINCT is straightforward yet highly effective: it instructs the SQL processor to examine the entire result set and eliminate any rows that are exact duplicates across all columns specified in the `SELECT` statement. If a query selects all columns (using the asterisk `*`), then two rows are considered duplicates only if the values in every single column are identical. Conversely, if only a subset of columns is selected, the uniqueness check applies exclusively to that specified subset. Understanding this nuance is crucial for data integrity, as utilizing **SELECT DISTINCT** ensures that statistical calculations, aggregations, and visual reports are not biased or inflated by repeated records, ultimately leading to more trustworthy and reliable analytical outcomes within the SAS platform.

Understanding SELECT DISTINCT and PROC SQL

The SQL procedure, commonly referred to as PROC SQL in SAS, provides a fully integrated, industry-standard language interface for data manipulation, retrieval, and structural management. Unlike traditional Base SAS procedures that often require multiple steps (such as sorting followed by data steps) to achieve complex transformations, PROC SQL allows users to perform these actions in a single, consolidated query, significantly streamlining the coding process. The **SELECT DISTINCT** clause is an integral part of this framework, offering a robust method to ensure data quality and uniqueness directly at the query execution level, saving computational time that might otherwise be spent on post-processing cleanup.

In practice, the **SELECT DISTINCT** statement must immediately follow the `SELECT` keyword in the query syntax. Its presence fundamentally alters how the result set is constructed, forcing an exhaustive comparison across all retrieved rows before they are returned to the user or stored in a new table. This comparison is not merely checking for an identical primary key, but rather performing a holistic row-level check based on the selected attributes. For users transitioning from other database environments, the syntax and behavior of **SELECT DISTINCT** in PROC SQL will feel highly familiar, reinforcing SAS's commitment to supporting standard SQL conventions. This feature is particularly valuable when merging disparate data sources or when dealing with raw transactional data where redundant logging is a common occurrence.

It is important to differentiate the function of **SELECT DISTINCT** from simple grouping clauses. While a `GROUP BY` clause achieves uniqueness for the grouped columns, its primary purpose is to

perform aggregate calculations (like sums or averages) for those groups. **SELECT DISTINCT**, conversely, focuses purely on de-duplication, returning the full, unique rows without requiring any aggregation functions. This subtle distinction determines which method is appropriate for a given analytical task: if the goal is strictly to identify and isolate unique records, **SELECT DISTINCT** is the cleaner and more direct approach. If subsequent calculations are required, `GROUP BY` remains the preferred tool.

The Necessity of Unique Data: Why Use DISTINCT?

In many real-world data collection scenarios, particularly those involving automated processes, human data entry, or integrated systems, data duplication is an unavoidable reality. Common reasons for duplicate entries include system errors, repeated imports, minor variations in input leading to partial matches, or simply the nature of transactional logs. Failing to address these redundancies can lead to significant analytical errors. For instance, if a dataset includes two identical entries for a customer's sale, calculating the total revenue without filtering duplicates would artificially inflate the result, leading to inaccurate business intelligence and skewed decision-making. The integrity of any statistical model, forecast, or report hinges directly on the cleanliness and uniqueness of the underlying data.

The use of **SELECT DISTINCT** addresses this data quality challenge by serving as an efficient integrity check during the data retrieval phase. By applying this clause, analysts can quickly determine the true cardinality of their data based on specific variables or the entire row structure. This is especially useful during the exploratory data analysis (EDA) phase, where understanding the distribution and redundancy of key identifiers (like product codes, customer IDs, or geographical markers) is essential. If the count of rows returned by a standard `SELECT *` query is significantly higher than the count returned by `SELECT DISTINCT *`, it immediately alerts the analyst to a serious duplication issue that requires remediation before proceeding to advanced modeling or reporting.

Moreover, the utilization of **SELECT DISTINCT** is often optimized for performance within the SAS execution engine. While alternative methods exist in Base SAS, such as using the `SORT` procedure followed by a `DATA` step with the `NODUPKEY` option, the SQL approach is often favored for its conciseness and conceptual simplicity. By embedding the de-duplication logic directly into the query, developers reduce the need for multiple procedural steps, minimizing I/O operations and intermediary storage requirements. This makes SELECT DISTINCT a pragmatic choice for analysts working with large volumes of data where computational efficiency is a high priority.

Setting Up the Environment: Creating the Sample Dataset

To illustrate the practical application of **SELECT DISTINCT**, we will utilize a small, simulated

dataset containing information about various basketball players. This example is intentionally constructed to include multiple duplicate rows, allowing us to clearly observe how the de-duplication clause functions. The following Base SAS code snippet demonstrates the creation and initial population of our sample data, named `my_data`, which captures key attributes such as team affiliation, player position, and points scored during a recent game.

The `DATA` step is the foundational method in SAS for creating or modifying datasets. Within this step, we define the structure using the `input` statement, where the dollar sign (`$`) denotes a character variable (like `team` and `position`), and the absence of a dollar sign denotes a numeric variable (like `points`). The `datalines` statement signals that the data records immediately follow. Crucially, note the intentional entry of several redundant rows: team A's Guard scoring 14 points is listed twice, team A's Forward scoring 13 points is listed twice, and team B's Guard scoring 22 points is also listed twice. This redundancy sets the stage for our de-duplication exercise.

After defining and populating the data, we use `PROC PRINT` to view the raw, unsorted, and unfiltered data, confirming that the duplicate entries are present before applying any SQL logic. This initial visualization is essential for verifying the starting state of the data before transforming it. The structure of the dataset is clear, showing 10 total observations, many of which are exact row-level copies, making this an ideal candidate for demonstrating the power of **SELECT DISTINCT** within PROC SQL.

```
/*create dataset*/
data my_data;
input team $ position $ points;
datalines;
A Guard 14
A Guard 14
A Guard 24
A Forward 13
A Forward 13
B Guard 22
B Guard 22
B Forward 34
C Forward 15
C Forward 18
;
run;

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

Obs	team	position	points
1	A	Guard	14
2	A	Guard	14
3	A	Guard	24
4	A	Forward	13
5	A	Forward	13
6	B	Guard	22
7	B	Guard	22
8	B	Forward	34
9	C	Forward	15
10	C	Forward	18

Implementation 1: Selecting All Unique Rows (SELECT DISTINCT *)

Our first application of the de-duplication clause will utilize the asterisk (`*`) wildcard immediately following **SELECT DISTINCT**. This syntax instructs PROC SQL to evaluate the uniqueness of every row based on the combined values across *all* columns present in the source table, `my_data`. In this context, a duplicate is defined as a row that is an exact match for another row across the `team`, `position`, and `points` variables simultaneously. This is the most comprehensive form of de-duplication, ensuring that only truly unique observations remain in the final result set.

The code structure is simple: we initiate PROC SQL, specify the selection of distinct rows using the wildcard, identify the source table with the `FROM` clause, and conclude the procedure with `QUIT`. The efficiency of this operation lies in the way the SAS SQL engine handles the comparison; it processes the input records, temporarily stores unique combinations, and discards subsequent records that match those already encountered. For data governance, this query is invaluable because it provides a clean, primary list of operational records without the noise of accidental redundancies that might have crept into the data source.

Upon execution, the system performs a multi-variable comparison across the dataset. For instance, while there are three rows with `team='A'` and `position='Guard'`, only one of these records also has `points=14`. Since the first two records (A, Guard, 14) are identical across all three variables, the second record is dropped. The third record (A, Guard, 24) is retained because the `points` value (24) is unique compared to the existing 'A Guard' combination. This meticulous, row-level scrutiny ensures that the output represents the minimum necessary set of observations to describe all unique permutations present in the original data structure.

/*select all unique rows*/

```
proc sql;
select distinct *
from my_data;
quit;
```

Interpreting the Results

The output generated by executing the `SELECT DISTINCT *` query clearly demonstrates the successful removal of duplicate observations from the original 10-row dataset. As shown in the visualization below, the resulting table now contains only 7 rows. The three rows that were eliminated were those that were exact, whole-row matches to other observations already present in the output. Specifically, the second instance of `(A, Guard, 14)`, the second instance of `(A, Forward, 13)`, and the second instance of `(B, Guard, 22)` were successfully filtered out by the **SELECT DISTINCT** clause.

Analyzing the resulting table, we can confirm that every single remaining row is unique when considering the composite values of `team`, `position`, and `points`. For example, even though Team A appears multiple times, the combination of its attributes is now unique across the list: we have (A, Guard, 14), (A, Guard, 24), and (A, Forward, 13). Had we not used **SELECT DISTINCT**, the analysis of average scores or team statistics would have been inaccurate due to the repeated counting of certain player performances. This filtered result set provides the clean foundation necessary for subsequent data analysis, ensuring that each unique player performance combination is represented exactly once.

This immediate visual confirmation underscores the utility of embedding de-duplication logic directly into the query. By contrasting the initial raw data with this final output, analysts gain confidence that the data processing step has been executed correctly. Furthermore, the ability of PROC SQL to handle this task with minimal code complexity makes it a highly efficient tool for routine data cleaning operations. The output set, while smaller, retains all the unique information from the original source without the burden of redundancy.

team	position	points
A	Forward	13
A	Guard	14
A	Guard	24
B	Forward	34
B	Guard	22
C	Forward	15
C	Forward	18

Implementation 2: Targeting Specific Column Combinations

While using the wildcard (`*`) ensures row-level uniqueness, often analysts are interested in the unique combinations of a subset of variables, rather than the entire row. The power of **SELECT DISTINCT** becomes even more flexible when we explicitly name the columns we wish to include in the uniqueness check. When a list of columns follows **SELECT DISTINCT**, PROC SQL only considers two rows duplicates if their values in the specified columns are identical, regardless of the values held in any unselected columns. This allows for focused de-duplication based on specific business criteria.

In our basketball example, suppose we are interested only in identifying all the unique combinations of `team` and `position` that exist in our dataset, without concern for the points scored. We modify our query to specifically list `team` and `position` after **SELECT DISTINCT**. The SQL engine will then perform the uniqueness check only on these two variables. For example, all rows belonging to Team A's Guards--regardless of whether they scored 14 or 24 points--will be treated as duplicates of the `(A, Guard)` combination, and only one instance of that combination will be retained in the final output.

This targeted approach is exceptionally useful for creating look-up tables, dimension tables, or summary lists where the focus is on categorical variables. It allows analysts to efficiently map out the structural composition of their data. In data warehousing, for instance, generating a list of unique geographical regions and product lines is a prerequisite for building dimension tables, and **SELECT DISTINCT** provides the fastest way to extract this clean list of combinations directly from the raw fact tables. The ability to control the scope of the uniqueness check is what makes **SELECT DISTINCT** such a versatile tool in the PROC SQL arsenal.

/*select all unique combinations of team and position*/
proc sql;

select distinct team, position
from my_data;
quit;

team	position
A	Forward
A	Guard
B	Forward
B	Guard
C	Forward

Advanced Considerations and Best Practices

When working with large datasets in SAS, it is crucial to be aware of the performance implications of using **SELECT DISTINCT**. Implementing this clause often requires the SQL engine to perform an internal sort or hash operation on the data to identify and compare all unique combinations. For tables containing millions of observations, this process can be resource-intensive, particularly if the selected columns are numerous or contain long character strings. Analysts should always strive to apply the **SELECT DISTINCT** operation as early as possible in a query chain and only on the necessary columns to minimize the computational overhead. Furthermore, if the de-duplication is a permanent requirement for a table, creating an indexed, unique version of the data may offer better long-term performance than repeatedly running **SELECT DISTINCT** on the raw source table.

A key aspect of using SELECT DISTINCT in PROC SQL involves the handling of missing values. In SAS, missing numeric values are represented by a period (.), and missing character values are represented by spaces. When **SELECT DISTINCT** performs its comparison, two rows that both contain a missing value in the same specified column are considered identical for the purpose of de-duplication. This behavior is usually desirable, as analysts typically want to collapse all records representing an unknown or missing state into a single unique observation. However, analysts must be mindful of this rule, especially when dealing with data cleaning steps, ensuring that missing values are consistently represented to avoid unintended results where two different representations of 'missing' might be treated as unique records.

Finally, while **SELECT DISTINCT** is the primary SQL method for de-duplication, SAS offers powerful alternatives for specific scenarios. As mentioned previously, using `PROC SORT` with the `NODUPKEY` option followed by a `DATA` step is an excellent Base SAS alternative, especially when the goal is to create a physical, permanent dataset that is sorted and guaranteed to be unique based

on specific key variables. The choice between the SQL method and the Base SAS method often depends on the overall coding environment and the ultimate destination of the data. For quick, one-off analyses and integration into complex SQL queries, **SELECT DISTINCT** offers unparalleled brevity and ease of use, making it an essential skill for efficient data manipulation in the SAS environment.