Paper 4109-2020

Translating SQL to SAS® and Back:
Performing Basic Functions Using DATA Steps vs. Proc SQL

Katie Haring, Highmark Health

ABSTRACT

Many beginning SAS® software users know how to write SQL code or SAS code, but not both. This quick tip is designed to teach users how to complete basic data manipulation and visualization processes using both DATA steps and PROC SQL.

INTRODUCTION

Having a working knowledge of both SAS® and SQL coding allows users to code more easily and efficiently in SAS software. However, many newer users only have coding experience in one of these languages. Sometimes, even coders who have worked with both languages find that they have knowledge gaps due to learning on the job with no formal instruction. This short paper aims to fill in those gaps by showing readers how to perform the same basic functions in both languages. The target audience of this paper will have limited experience in one or both languages. Readers with no prior coding experience may struggle to understand some basic premises that are not discussed in detail here.

PROC STEPS AND DATA STEPS

Code in SAS software can be run in procedure steps (PROC steps) or DATA steps. PROC steps are chunks of code that perform a procedure of some sort. They can be used to run SQL code, create a frequency table, print results, and more. PROC steps start with the word PROC and are executed with RUN or QUIT depending on the procedure. PROC SQL is used to run SQL code in SAS software. The following code shows how to create a table (Output_table) containing all of the content from another table (Input_table) using PROC SQL:

```
PROC SQL;
   CREATE TABLE Output_table
      AS SELECT *
   FROM Input_table;
QUIT;
```

DATA steps are chunks of code that create or modify data. They begin with the word DATA and are executed using RUN. We can replicate the results of the SQL code above with the following DATA step:

```
DATA Output_table;
   SET Input_table;
RUN;
```

FILTERING

Oftentimes, the user will want to look at only a subset of the dataset. This can be done by filtering on rows or filtering on columns. Filtering on rows means limiting the dataset to only

rows that meet (or do not meet) certain criteria. In both languages, this can be achieved by using a WHERE clause. In SQL, we can filter our original data down to just Females like so:

```
PROC SQL;
   CREATE TABLE Output_table
      AS SELECT *
         FROM Input_table
   WHERE Gender = 'F';
QUIT;
```

Using a DATA step, the same filtering can be done with:

```
DATA Output_table;
   SET Input_table;
   WHERE Gender = 'F';
RUN;
```

Note that the actual WHERE clause (WHERE Gender = 'F';) is identical between the two languages. This will often, but not always, be the case with WHERE clauses. Filtering on columns means limiting the dataset to only a select group of columns. In SQL, this is commonly done by specifying the desired columns instead of using *, which selects all:

```
PROC SQL;
   CREATE TABLE Output_table
      AS SELECT Name,
                Gender
   FROM Input_table;
QUIT;
```

In a DATA step, this can be done by dropping the columns the user isn't interested in:

```
DATA Output_table;
   SET Input_table;
   DROP Birth;
RUN;
```

Or by keeping the columns the user is interested in, with a space between each column name:

```
DATA Output_table;
   SET Input_table;
   KEEP Name Gender;
RUN;
```

FORMATTING

The format of a column of data determines how it is displayed to the user. Often, format will be modified to change the number of decimal places in a numeric field or the display of a date field. The full list of possible formats can be found on support.sas.com. In SQL, formatting can be done in the SELECT statement. The following code formats the Birth column to be displayed as DDMMMYY:

```
PROC SQL;
   CREATE TABLE Output_table
      AS SELECT Name,
                Gender,
                Birth format=DATE7.
   FROM Input_table;
QUIT;
```

In a DATA step, a separate formatting statement can be used:

```
DATA Output_table;
   SET Input_table;
      FORMAT Birth DATE7.;
RUN;
```

APPENDING DATA

Stacking two or more data tables on top of each other to create one longer table is called appending. Appending all data from both tables requires less code in a DATA step than in a PROC SQL statement. Here is an example, appending the Input_table to itself, in PROC SQL:

```
PROC SQL;
   CREATE TABLE Output_table
      AS SELECT * FROM Input_table
         UNION ALL
      SELECT * FROM Input_table;
QUIT;
```

Here is the same example using a DATA step:

```
DATA Output_table;
   SET Input_table
      Input_table;
RUN;
```

MERGING DATA

Merging data combines two or more data tables on their common column(s). Using PROC SQL, the LEFT JOIN, RIGHT JOIN, INNER JOIN, or OUTER JOIN statements can be used to merge. LEFT JOIN will keep all of the data from the first input table and only the matching data from the second input table. RIGHT JOIN will keep all of the data from the second input table and only the matching data from the first input table. INNER JOIN will only keep rows where the common variable is present in both tables. OUTER JOIN will keep all rows from both tables. In a case where both data tables have the exact same common column(s) (for example, the following code is for two tables that both contain data for the same 4 individuals), any JOIN will produce the same results:

```
PROC SQL;
   CREATE TABLE Output_table
      AS SELECT a.Name,
                a.Gender,
                a.Birth,
                b.Pet
      FROM Input_table AS a
         OUTER JOIN Input2 AS b
         ON a.Name = b.Name;
QUIT;
```

If the user wants to use a DATA step to merge two tables, both tables need to be sorted by the field you will be merging on. The MERGE function in a data step is equivalent to an OUTER JOIN in PROC SQL. The following code uses a DATA step to produce the same table that the PROC SQL code above produced:

```
PROC SORT DATA=Input_table;
   BY Name;
RUN;
```

```
PROC SORT DATA=Input2;
   BY Name;
RUN;


DATA Output_table;
   MERGE Input_table Input2;
      BY Name;
RUN;
```

CREATING TABLES

To create a new table from scratch using PROC SQL, users will need to enter the desired variable names, variable types, and data values. The following code creates a table with three columns (Name, Age, and Favorite_Food) and four rows.

```
PROC SQL;
   CREATE TABLE Output3
      (Name CHAR(4),
      Age FLOAT(2),
      Favorite_Food CHAR(7));


      INSERT INTO Output3
         VALUES('Jack',65,'Tacos')
         VALUES('Jane',56,'Samosas')
         VALUES('Jill',52,'Bread')
         VALUES('John',49,'Sushi');
QUIT;
```

The equivalent DATA step does not require the user to specify the variable types, but it is necessary to indicate character variables by placing the $ symbol after the variable name:

```
DATA Output3;
   FORMAT Age 2.;
   INPUT Name $ Age Favorite_Food $;
   DATALINES;
      Jack 65 Tacos
      Jane 56 Samosas
      Jill 52 Bread
      John 49 Sushi;
RUN;
```

PERFORMING CALCULATIONS

To create a new variable that is a calculation of other variables using PROC SQL, the user will SELECT the formula followed by AS and the new variable name. The following code creates as Age variable by subtracting the Birth variable from today's date and dividing by 365:

```
PROC SQL;
   CREATE TABLE Output_table AS
      SELECT a.*,
         (TODAY()- a.Birth)/365 AS Age
      FROM Input_table AS a;
QUIT;
```

In a DATA step, the new variable name comes first, followed by an equals sign and the formula:

```
DATA Output_table;
   SET Input_table;
      Age = (TODAY()- Birth)/365;
RUN;
```

BUT NOT EVERYTHING TRANSLATES...

Some actions are very difficult or impossible to perform using PROC SQL. This section includes some examples of these actions and how to perform them using native SAS coding.

IMPORTING DATA

Data can be imported into a project from various file types, including CSV, XLSX, and JMP, using PROC IMPORT. The user will need to specify the file path of the data to be imported, the file extension type, and the name of the resulting data table. REPLACE can be used to tell SAS to overwrite any other table with the same name:

```
PROC IMPORT FILE='/datapath/Data'
   DBMS=xlsx
   OUT=work.Input_table
   REPLACE;
RUN;
```

PLOTTING

PROC SGPLOT makes it simple to produce plots of all types. The following sample code creates a bar chart of Age by Name from the data in the Output3 table:

```
PROC SGPLOT DATA=Output3;
   VBAR Name/ RESPONSE=Age
   DATASKIN=gloss;
RUN;
```

The resulting plot is shown in Figure 1.

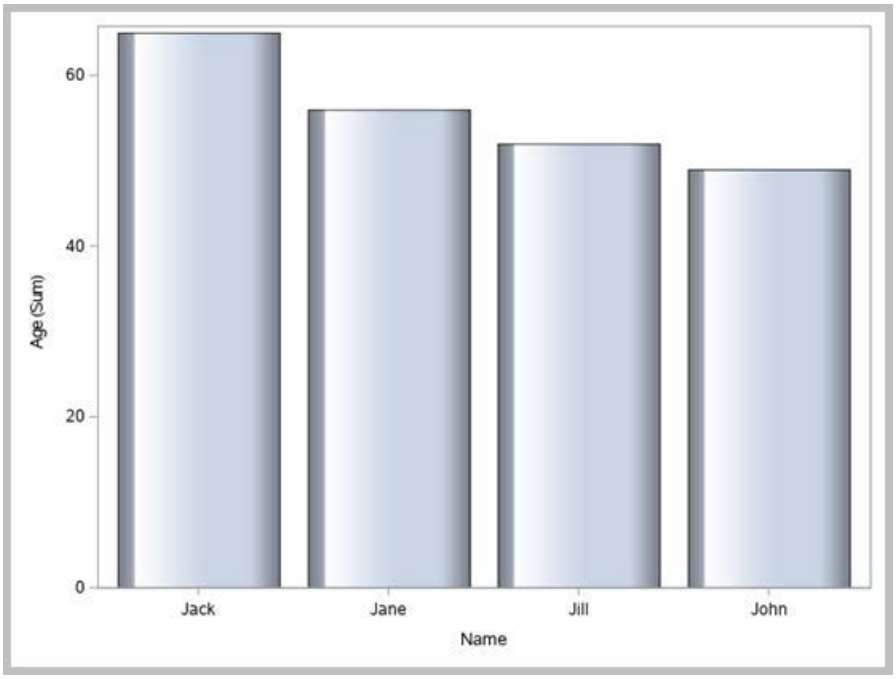

Figure 1. Plot of Age by Name.

TRANSPOSING A TABLE

PROC TRANSPOSE allows the user to take a long-format table and turn it into a wide-format table, or vice versa:

```
PROC TRANSPOSE DATA=Input_table
   NAME=Descriptor
   OUT=Output_table;
      ID Name;
         VAR Birth Gender Pet;
RUN;
```

CREATING AN ACCUMULATING COLUMN

An accumulating column shows a running total of another column. In the following example, a new column (Cumul_Age) shows the running total of the ages in the Age column:

```
DATA Output4;
   SET Output3;
      RETAIN Cumul_Age 0;
         Cumul_Age = Cumul_Age + Age;
RUN;
```

CONCLUSION

Having a firm grasp of basic DATA steps and PROC SQL procedures will allow the reader to more discerningly pick the method that best suits their needs. This paper reviewed methods for filtering, formatting, appending, merging, creating, and calculating using either approach. However, it will sometimes be much simpler to perform certain actions using native SAS code; specific examples reviewed here included importing, plotting, transposing, and accumulating.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Katie Haring
Highmark Health
Katherine.Haring@highmarkhealth.org