Paper 3237-2019

Sometimes SQL Really Is Better: A Beginner's Guide to SQL Coding for DATA Step Users

Brett Jepson, Rho, Inc.

ABSTRACT

Structured Query Language (SQL) in SAS® provides not only a powerful way to manipulate your data, it enables users to perform programming tasks in a clean and concise way that would otherwise require multiple DATA steps, SORT procedures, and other summary statistical procedures. Often, SAS users use SQL for only specific tasks with which they are comfortable. They do not explore its full capabilities due to their unfamiliarity with SQL. This presentation introduces SQL to the SQL novice in a way that attempts to overcome this barrier by comparing SQL with more familiar DATA step and PROC SORT methods, including a discussion of tasks that are done more efficiently and accurately using SQL and tasks that are best left to DATA steps.

INTRODUCTION

As SAS programmers, we often learn one way of reaching a result in our programming, and only explore another method after seeing either a considerable benefit or increase in efficiency. SQL programming often plays the part of this 'other' method that is largely unfamiliar and seems impossible to learn, especially compared to our more familiar methods of SAS programming (DATA steps, SORT procedures, MEANS and FREQ procedures, etc.).

This paper explains some of the basics of SQL coding using comparisons to familiar methods to help make this cloudy window of SQL to appear more clear to the SQL novice. As such, at least a basic understanding of DATA steps and SORT procedures is implied. The intent is to show and explain the code needed to perform each task. As output is not shown, the intent is for you to copy and explore the code for yourself.

There are many ways to perform similar tasks in SQL. This paper takes some of the most straightforward methods to help establish a solid foundation in SQL. You are encouraged to explore more complex and creative SQL methods once this foundation is set.

WHY LEARN SQL CODING?

While the majority of coding tasks can be performed in some way or another using DATA steps and a combination of familiar procedures (e.g. SORT, FREQ, SUMMARY, MEANS, etc.), the tasks can often be done more efficiently, and in some cases less prone to error, in SQL. What I mean is that these tasks can all be done in a single (albeit, sometimes complicated) query, creating only one dataset. Creating intermediate datasets is unnecessary, so you are less likely to inadvertently err by reading in an incorrect intermediate dataset- that dataset never existed independently in the first place. More discussion will be given to specific advantages of SQL throughout the paper.

Often, with a comparison of SQL and the SORT procedure, there is a discussion about the amount of computing power and time required to perform a task. As this is only evident using large datasets from programs that take a long amount of time to run, this paper only focuses on the coding aspects and relative efficiencies. However, if you do have a program that takes hours to run due to using the SORT procedure on one or more large datasets, SQL may be your solution.

SQL CODING STRUCTURE

In order to use SQL coding, you first have to begin with opening SQL, which you can do by using the following code structure:

```
PROC SQL;
<INSERT QUERIES>
QUIT;
```

Once SQL is open, you can insert as many queries prior to the QUIT command as you would like or need. In other words, you can create multiple datasets within one SQL session.

The following clauses consist of the major components that you use in creating a dataset in SQL (though not all are required):

1. **CREATE TABLE:** specifies that you want to create a dataset, the name of which will follow
2. **SELECT:** specifies and/or derives variables to include in your dataset (required)
3. **FROM:** specifies from where you are pulling your data (required)
4. **WHERE:** specifies subsets applied to your input dataset defined in the FROM statement
5. **GROUP BY:** specifies variables by which you are summarizing; GROUP BY is only used when including a summary function within your query
6. **HAVING:** specifies a subset to apply upon outputting a dataset once all other derivations have taken place
7. **ORDER BY:** specifies a sort order to apply upon outputting a dataset

Note that the statements, though not all required, must be in this order if used. The mnemonic device I learned to remember the order was **S**o **F**ew **W**orkers **G**o **H**ome **O**n time, to correspond to SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. Technically, in order to create a query that goes to your output window, those clauses are sufficient; however, to create a dataset, CREATE TABLE is required, which is the primary focus of this paper.

While DATA step coding and SQL coding are different in structure, the individual pieces of code you generate to perform the same task can be mapped to each other. Some examples will give more context and allow you to see side-by-side how each component relates to its counterpart. For all examples, the SASHELP dataset library is used, allowing you to copy and submit the code for yourself.

SQL AND DATA STEP COMPARISONS

CREATING A SIMPLE DATASET

Table 1 shows DATA step and SQL coding side-by-side to create a simple output dataset named DATA1, derived from the CARS input dataset. Colored code in the SQL coding example is mapped to equivalent code of the same color in the DATA step coding example.

SQL Coding	DATA Step Coding

`proc sql;`
`  create table data1 as`
`  select *`
`  from sashelp.cars;`
`quit;`
	`data data1;`
	`  set sashelp.cars;`
	`run;`

Table 1. Creating a simple dataset

This simple example reveals some initial differences between the coding structure:

- Clauses in SQL are not separated by semicolons like in DATA step coding. In fact, there are no separators between clauses except the beginning of the next clause. Only a final semicolon is required.
- SQL requires you to specify after the SELECT clause the variables you are keeping. An asterisk (*) represents all variables from the CARS dataset, indicating that all variables are kept in the output dataset.

KEEPING A SUBSET OF VARIABLES/RENAMING VARIABLES

Table 2 coding expands on our simple dataset coding to keep only a subset of variables.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, model, msrp as price
  from sashelp.cars;
quit;
	data data1;
	  set sashelp.cars;
	  rename msrp = price;
	  keep make model price;
	run;

Table 2. Specifying variables to keep in the output dataset

Here are some observations that stem from this example:

- Only variables in the SELECT clause are kept in our output dataset.
- Within an individual clause (in this case, the SELECT clause), if more than one variable is specified or derived, a comma must separate each variable. The traditional DATA step requires only a space between variables. This convention applies to all other clauses except CREATE TABLE.
- Renaming variables in SQL is done by simply by including the previous variable name, an AS clause, and the new variable name. With this simple coding, the final output dataset preserves all attributes (length, type, format and label) of the renamed variable from the input variable. If any other derivation is applied to a variable instead of simply renaming it, attributes may be lost. Note that renaming a variable does not limit your ability to also keep a duplicate variable of the original name in the dataset in SQL coding (e.g. MSRP and the newly renamed variable PRICE can both exist within the same dataset by adding ", msrp" within the SELECT clause).

CREATING NEW VARIABLES AND SPECIFYING VARIABLE ATTRIBUTES

Table 3 shows an extension of our previous example by adding coding to create a new variable named MPD_CITY (miles per dollar of gas for city driving, assuming gas costs $2 per gallon), while also specifying the length and format of the new variable. The length of an existing variable MODEL is also set.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, model length=50
  , type, msrp
  , mpg_city/2 as mpd_city
  label='City Miles/$'
  format=8.1
  from sashelp.cars;
quit;
	data data1;
	  length model $50;
	  set sashelp.cars;
	  label mpd_city = 'City Miles/$';
	  format mpd_city 8.1;
	  mpd_city = mpg_city/2;
	  keep make model type msrp
	  mpd_city;
	run;

Table 3. Creating new variables and specifying variable attributes

This example offers the following new observations:

- Most derivations not involving IF/THEN/ELSE logic have similar coding between DATA step and SQL

coding, including the use of most functions.

- Compared to DATA step coding, deriving variables in SQL requires the derivation to occur first and the name of the variable to occur after an AS clause, which takes the place of an equal sign.
- Attributes of length, format and label are set within the specifications for the variable of interest in the SELECT clause, instead of being set in separate lines of code, like in DATA step coding. You can place these attribute modifiers anywhere after the variable derivation, even before the AS clause.
- Specifying the length of a character variable does not require a dollar sign to proceed the length.
- In SQL coding, a newly created variable will automatically appear in the output dataset. In DATA step coding using a KEEP clause, you must include newly created variables in the list of kept variables.

CREATING NEW VARIABLES WITH CASE WHEN LOGIC

Table 4 shows CASE WHEN coding to create a new variable named PRICECAT, which is a categorical classification based on the MSRP price. CASE WHEN coding is similar to DATA step IF/THEN/ELSE coding in this example.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select *
  , case
  when msrp < 20000
  then '<$20K'
  when msrp <= 35000
  then '$20-35K'
  else '>$35K'
  end label='Price' as pricecat
  from sashelp.cars;
quit;
	data data1;
	  set sashelp.cars;
	  label pricecat = 'Price';
	  if msrp < 20000
	  then pricecat = '<$20K ';
	  else if msrp < 35000
	  then pricecat = '$20-35K';
	  else pricecat = '>$35K';
	run;

Table 4. Creating new variables using CASE WHEN logic

Here are some new observations relating to CASE WHEN logic:

- In SQL, you close a CASE with an END command, with WHEN/ELSE logic occurring between CASE and END.
- The name of the variable is only set once after the END command, as opposed to DATA step coding, which requires you to specify the variable name on each line where the variable definition is modified.
- WHEN clauses have to contain logic, as opposed to the ELSE clause, which acts as a catch-all as the last line of the block, setting all records not meeting any of the previous criteria to the value you specify.
- After the first WHEN clause, all subsequent statements operate like an ELSE IF statement in DATA step coding. Only if a record does not fit any of the previous WHEN clause will it be considered for the current WHEN clause logic.
- Once a variable is created, it cannot be modified within the same query. As such, circular logic is not possible.
- SQL automatically scours the possible values applied to a new character variable to ensure the length is appropriate for the longest value. With DATA step coding, the first value specified (in this case '<$20K') determines the length of the variable, which could lead to truncation of subsequent longer values. In the DATA step example, in order to avoid a truncation problem, you must either pre-specify the length of the variable to at least the greatest length observed, or add spaces as I did at the end of the first value. This correction is not required to be made in SQL.

- Attributes of a variable must be set after the END command when using CASE logic.
- If you want to preserve all variables from the input data set and include new variables, simply add more variables to the SELECT clause after the asterisk.

Nested CASE WHEN logic can appear within a CASE WHEN block. This logic is similar to the IF/THEN DO logic used in DATA step coding. Table 5 shows an example of nested CASE WHEN logic in creating a new variable PRICECAT, which further distinguishes MSRP levels by adding an indicator of being either an SUV or a Non-SUV.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select *
  , case
  when type in ('SUV')
  then case
  when msrp < 20000
  then 'SUV <$20K'
  when msrp <= 35000
  then 'SUV $20-35K'
  else 'SUV >$35K'
  end
  else case
  when msrp < 20000
  then 'Non-SUV <$20K'
  when msrp <= 35000
  then 'Non-SUV $20-35K'
  else 'Non-SUV >$35K'
  end
  end as pricecat
  from sashelp.cars;
quit;
	data data1;
	  set sashelp.cars;
	  length pricecat $15;
	  if type in ('SUV') then do;
	  if msrp < 20000
	  then pricecat = 'SUV <$20K';
	  else if msrp < 35000
	  then pricecat = 'SUV $20-35K';
	  else pricecat = 'SUV >$35K';
	  end;
	  else do;
	  if msrp < 20000
	  then pricecat =
	  'Non-SUV <$20K';
	  else if msrp < 35000
	  then pricecat =
	  'Non-SUV $20-35K';
	  else pricecat = 'Non-SUV >$35K';
	  end;
	run;

Table 5. Creating new variables using nested CASE WHEN logic

The entire nested CASE WHEN block appears in place of a set variable value that would normally follow the THEN clause, and the nested block has its own END command. Determining the block to which each clause belongs can be confusing, but a recommendation is to add parentheses around any entire nested CASE WHEN block for a clear divider.

SORTING DATASETS

Table 6 shows how to sort datasets using both methods of coding. Unlike DATA step coding, SQL coding allows you to sort when outputting the dataset, instead of requiring a separate procedure. The ORDER BY clause can contain input variables or existing variables, regardless of whether these variables are kept in the final output dataset (though you will get a note in the log stating "the query as specified involves ordering by an item that doesn't appear in its SELECT clause").

In addition, variable derivations which are not derived in the SELECT clause can be included in the ORDER BY clause, including functions and CASE WHEN blocks. Since these variables are not included in the SELECT clause, they will not appear in the output dataset.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, model, type
  , mpg_city/2 as mpd_city
  from sashelp.cars
  order by make, type, calculated
  mpd_city descending;
quit;
	data data1;
	  set sashelp.cars;
	  mpd_city = mpg_city/2;
	  keep make model type mpd_city;
	run;
	
	proc sort data = data1;
	** by make type descending mpd_city;**
	run;

Table 6. Sorting datasets

This example offers the following new observations:

- If you refer to previously derived variables, whether in subsequent variable creation in a SELECT clause, or a GROUP BY, HAVING or ORDER BY clause, you must precede the variable name with the word CALCULATED. This does not apply to variables that were simply renamed with no change of attributes (as in Table 2). Since the WHERE clause applies to the input dataset, CALCULATED is not applicable in a WHERE clause.
- The DESCENDING modifier appears after the variable name in SQL coding, instead of before the variable name, as required in PROC SORT.

CREATING SUBSETS WITH WHERE AND HAVING

Table 7 demonstrates creating subsets based on both input data and derived variables. The WHERE clause in SQL applies a subset upon the input dataset, similar to the WHERE clause in DATA step coding. In both examples, we are only keeping non-trucks.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, model, msrp
  , mpg_city/2 as mpd_city
  from sashelp.cars
  where type ^= 'Truck'
  having calculated mpd_city >= 13;
quit;
	data data1;
	  set sashelp.cars;
	  where type ^= 'Truck';
	  mpd_city = mpg_city/2;
	  if mpd_city >= 13;
	  keep make model msrp mpd_city;
	run;

Table 7. Creating subsets while inputting and outputting datasets

The HAVING clause applies a subset upon outputting the dataset, which allows you to create subsets based on derived variables within the query, as well as previously existing variables, which is similar to the IF clause used in the DATA step coding example. In this example, we only want to keep the records of cars that have a city-miles per dollar value of at least 13. Since this variable was derived within this query, it is necessary to include in the HAVING clause instead of the WHERE clause.

The HAVING clause can also create subsets based on summary variables, which will be addressed later.

CREATING DATASETS WITH SUMMARY VARIABLES

Table 8 demonstrates how to find summary statistics of a variable using both methods of coding. In this example, we are finding the count, mean, minimum and maximum of MSRP separately within each value of MAKE. While DATA step coding produces a dataset with a couple more variables, and an extra record for the overall summary statistics, the output produced looks very similar between methods.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, count(msrp) as n
  , mean(msrp) as mean
  , min(msrp) as min
  , max(msrp) as max
  from sashelp.cars
  group by make;
quit;
	proc means data = sashelp.cars mean;
	  class make;
	  var msrp;
	  output out = data1 n=n mean=mean
	  min=min max=max;
	run;

Table 8. Creating a dataset of summary statistics of MSRP by make of car

In the SQL example, summary functions are used within a query. These summary stats can be mixed with other variables within the SELECT clause (say, for example a user wants to include the minimum MSRP on each record). SQL coding can also add variables for summary statistics on more than one variable (e.g. the mean MPG_CITY can be added as a variable). There are many summary functions that can be used in SQL coding; please refer to Pete Lund's paper for a list and description of available summary functions (see recommended readings).

The GROUP BY clause indicates that the summary functions will calculate separately within each level of each included variable, which is MAKE in this example. As with other clauses, multiple variables can be used in this clause.

CREATING SUBSETS BASED ON SUMMARY VARIABLES

Table 9 demonstrates creating subsets based on summary statistics. Since SQL does this all within a query, the relative efficiency of using SQL becomes more evident. Each coding example in the table produces a subset of the CARS dataset that includes all cars with an MSRP that is greater than the average MSRP for its MAKE and TYPE.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, model, type, msrp
  from sashelp.cars
  group by make, type
  having msrp > mean(msrp)
  order by type, make, model;
quit;M
	proc means data = sashelp.cars mean;
	  class make type;
	  var msrp;
	  output out = means mean=meanmsrp;
	run;
	
	proc sort data = means;
	  by make type;
	  where cmiss(make,type) = 0;
	run;
	
	proc sort data = sashelp.cars
	  out = data1sort;
	  by make type;
	run;
	
	data data1;
	  merge data1sort means;
	  by make type;
	  if msrp > meanmsrp;
	  keep make model type msrp;
	run;
	
	proc sort data = data1;
	  by type make model;
	run;

Table 9. Creating a subsets of cars with an MSRP greater than the average per make and type

In the SQL example, the HAVING clause contains a summary function (MEAN) of MSRP. Instead of including the mean MSRP by MAKE and TYPE in the output dataset (which we are not interested in keeping in our output dataset), it is being used solely in creation of the subset, though the mean MSRP could be included in the output dataset if wanted. By including the GROUP BY clause, the mean summary function calculates separately within each level of each included variable. This all occurs within one query.

In contrast, in order to properly create this subset, DATA step coding requires you to calculate the means separately within another procedure (in this example, MEANS). You are then required to sort both the original dataset and the means dataset in order to merge them together. This process is further complicated if a subset is needed, which must either be created in a dataset to be used in both the MEANS procedure and the DATA step, or must be applied separately in all applicable datasets prior to merging or calculating summary statistics.

USING SUBQUERIES

SQL coding allows for subqueries to be written inside of queries, eliminating the need for creating datasets outside of the query if not necessary. Subqueries can be included in all

clauses except for CREATE TABLE. Subqueries have the same structure and rules as a query (e.g. the ordering of clauses must follow the same rules, there must be a SELECT clause and a FROM clause, etc.). However, CREATE TABLE and ORDER BY are not included in subqueries, as there is not a physical output for subqueries- they just live within the query and any result of the subquery is manifest in the final output of the query.

This section addresses subqueries on the SELECT, WHERE and HAVING clauses. For an example using a subquery on the FROM clause, see the section Joins Involving More than Two Datasets. However, there are more ways to use subqueries than those addressed in this paper. You are invited to explore other ways to use subqueries in your work.

Subqueries Within a SELECT Clause

Table 10 shows a subquery within a variable definition on the SELECT clause. The derived variable RATIO represents the ratio of the MSRP variable to the overall mean MSRP in the CARS dataset.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, model, type, msrp
  , msrp/(select mean(msrp) from
  sashelp.cars) as ratio
  format = 8.2
  from sashelp.cars;
quit;
	proc means data = sashelp.cars mean;
	  var msrp;
	  output out = means mean=meanmsrp;
	run;
	
	data _null_;
	  set means;
	  call symputx ('mean',meanmsrp);
	run;
	
	data data1;
	  merge sashelp.cars;
	  format ratio 8.2;
	  ratio = msrp/&mean.;
	  keep make model type msrp ratio;
	run;

Table 10. Using a subquery within the SELECT clause to derive a variable

The subquery calculates the mean MSRP from the CARS dataset and inserts it into the calculation of the ratio, instead of calculating it externally and pulling it in from an outside source, as is done in the DATA step equivalent. Though there are several ways to pull in the mean, this method creates a macro variable that will be used for this purpose only.

Notice that the subquery returned only one result, which is appropriate for a SELECT clause.

Subqueries Within a WHERE or HAVING Clause

Table 11 shows a subquery within a WHERE clause. This example creates an output dataset from input CARS dataset that only includes car types (TYPE) that have at least 50 models.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select make, model, type, msrp
  from sashelp.cars
  where type in
  (select type
  from sashelp.cars
  group by type
  having count(*) >= 50);
quit;
	proc freq data = sashelp.cars;
	  table type/out = counts;
	run;
	
	proc sort data = sashelp.cars
	  out = data1sort;
	  by type;
	run;
	
	data data1;
	  merge counts data1sort;
	  by type;
	  if count >= 50;
	  keep make model type msrp;
	run;

Table 11. Using a subquery within the WHERE clause to determine a subset

The subquery creates one record per value of TYPE, keeping only the values of TYPE that have a count of at least 50 models. The WHERE clause takes the result of this subquery and keeps only the records that have a value of TYPE matching a record in the subquery. The subset is applied upon reading in the CARS dataset. Note that the number of records resulting from the subquery is allowed to be more than one (two in this case- 'SUV' and 'Sedan'), but must result in only one variable.

The DATA step coding performs the same task, but requires an outside count per TYPE to be merged with the input dataset, and applies the subset on all records of the CARS dataset upon output.

In this example, if we had used the subquery on a HAVING clause instead of on the WHERE clause, the result would be the same. This will not always be the case, but this example provides just one way to use a subquery on a WHERE or a HAVING clause.

MERGING DATASETS USING JOINS

In terms of relative efficiencies, SQL joins provide some of the greatest advantages over the SORT procedure/DATA set counterparts. Join clauses (e.g. FULL JOIN, LEFT JOIN, RIGHT JOIN and INNER JOIN), while similar to MERGE statements in DATA step coding, have some obvious differences, which are briefly mentioned here:

1. Joins do not require prior sorting as is required by merges.
2. Joins can be performed on variables that do not have the same name or length. In fact, joins can be performed on variables derived during the joining process. There are a plethora of other joining techniques to result in the desired output dataset that can't all be described in this paper.
3. Joins can involve more than 2 input datasets, and all datasets don't need to be joined by the same variables.
4. Joins eliminate the risk of accidentally rewriting over common variables between 2 or more datasets, as an attempt to join datasets with variables of the same name produces a warning for each duplicate variable. You are forced to decide how to deal with those variables in the SELECT clause.

There are 4 types of joins that are discussed briefly in this paper.

LEFT JOIN

Table 12 demonstrates performing a left join using the GCSTATE and US_DATA, which are joined on the variables MAPIDNAME (in the GCSTATE dataset) and STATENAME (in the US_DATA dataset). In this example, we are including all variables from both datasets, but we are only including records that appear in the GCSTATE dataset.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select A.*, B.*
  from sashelp.gcstate A
  left join sashelp.us_data B
  on A.mapidname = B.statename
  order by A.mapidname
  ;
quit;
	proc sort data = sashelp.gcstate
	  out = gcstate;
	  by mapidname;
	run;
	
	proc sort data = sashelp.us_data
	  out=us_data;
	  by statename;
	run;
	
	data data1;
	  merge gcstate (in=a) us_data (in=b
	  rename=(statename=mapidname));
	  by mapidname;
	  if a;
	run;

Table 12. Performing a left join

To perform joins, we include a FROM clause, which includes an initial dataset (in this case, GCSTATE). Since variables from multiple datasets are included in our output dataset, the dataset is nicknamed 'A', and all references to variables from this dataset will be preceded by 'A.'. Note that using a nickname for an input dataset (and preceding all variables referenced from that dataset) is not always required; in this example, we would get the same result without setting a nickname since no variable name appears in both datasets. However, it is good practice to include a nickname and precede each input variable with a nickname to ensure the correct variable is being referenced.

The next step is to join another dataset (US_DATA) onto our first dataset using a JOIN clause. This dataset has the nickname 'B'. JOIN clauses use an ON clause similarly to a BY statement in a DATA step merge. In a merge, records are mapped to each other if they share the same BY variables. However, joins do not require variables to have the same name or length. In this example, MAPIDNAME and STATENAME are similar in structure and content, so we can easily join the 2 datasets on these variables.

The basic format of the ON clause is: *<variable from one dataset>* = *<variable from another dataset>*. With the use of AND and OR clauses, derivations of variables, sub-queries, and more, ON can unlock the power of joins to perform almost any join imaginable. More examples later in the paper will show some of the potential of joins.

Since a left join is used and GCSTATE is the left dataset, all provinces of Canada are included in the output dataset, as they are from GCSTATE. Notice that all variables from US_DATA are empty for the Canada provinces, as US_DATA does not include Canadian provinces. Notice also that the record in US_DATA for Puerto Rico is not in the output dataset. This left join performs the same subsetting as the combination of IN= and IF statements in the DATA step example.

The SELECT clause uses the asterisks to indicate that all variables from GCSTATE (A) and US_DATA (B) are kept in the final dataset. If there are variables of the same name in both datasets (as there are not in this example), we would have to decide to keep one only, rename one, or combine them using a COALESCE function, which will be discussed later.

There are 2 important differences between the methods to note at this point:

1. The SQL example doesn't require data being pre-sorted. The DATA step example could have bypassed sorting as well since the input datasets are already properly sorted, but as a general practice, sorting of all input datasets does not need to be considered in SQL coding.
2. The SQL example now has both MAPIDNAME and STATENAME in the final output. The DATA step coding required a rename of STATENAME to match MAPIDNAME, and therefore we lost the STATENAME variable prior to the merge.

RIGHT JOIN

Table 13 demonstrates performing a right join using the GCSTATE (left dataset) and US_DATA (right dataset), which are joined on the variables MAPIDNAME (in the GCSTATE dataset) and STATENAME (in the US_DATA dataset). This example is similar to the left join, except in this case, we only include records that appear in the US_DATA dataset.

SQL Coding	DATA Step Coding

`proc sql;`
`  create table data1 as`
`  select A.*, B.*`
`  from sashelp.gcstate A`
`  right join sashelp.us_data B`
`  on A.mapidname = B.statename`
`  order by B.statename`
`  ;`
`quit;`
	`<pre-sorting of input datasets>`
	
	`data data1;`
	`  merge gcstate (in=a) us_data (in=b`
	`  rename=(statename=mapidname));`
	`  by mapidname;`
	`  if b;`
	`run;`

Table 13. Performing a right join

The output dataset now includes the Puerto Rico record from the US_DATA dataset, while excluding all the Canadian provinces from the GCSTATE dataset. The Puerto Rico record is missing all variables from the GCSTATE dataset. Note that the ORDER BY clause is now ordering by a variable in the right dataset (STATENAME), since MAPIDNAME is missing for the Puerto Rico record, as it is not in the GCSTATE dataset.

INNER JOIN

Table 14 demonstrates performing an inner join using the GCSTATE and US_DATA datasets, which are joined on the variables MAPIDNAME (in the GCSTATE dataset) and STATENAME (in the US_DATA dataset). This example is similar to both the left and right joins, except in this case, we only include records that appear in both input datasets.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select A.*, B.*
  from sashelp.gcstate A
  inner join sashelp.us_data B
  on A.mapidname = B.statename
  order by A.mapidname
  ;
quit;
	<pre-sorting of input datasets>
	
	data data1;
	  merge gcstate (in=a) us_data (in=b
	  rename=(statename=mapidname));
	  by mapidname;
	  if a and b;
	run;

Table 14. Performing an inner join

The output dataset now includes only the 50 United States and the District of Columbia, since these records are in both datasets.

FULL JOIN

Table 15 demonstrates performing a full join using the GCSTATE and US_DATA datasets, which are joined on the variables MAPIDNAME (in the GCSTATE dataset) and STATENAME (in the US_DATA dataset). This example is similar all previous joins, except in this case, we include all records that appear in either of the input datasets.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select A.*, B.*,
  coalesce(A.mapidname,B.statename)
  as combinedname,
  from sashelp.gcstate A
  full join sashelp.us_data B
  on A.mapidname = B.statename
  order by calculated combinedname
  ;
quit;
	<pre-sorting of input datasets>
	
	data data1;
	  merge gcstate (in=a) us_data (in=b
	  rename=(statename=mapidname));
	  by mapidname;
	  if a or b;
	run;

Table 15. Performing a full join

The output dataset now includes the 50 United States, the District of Columbia, Puerto Rico and all Canadian provinces- all records from either of the input datasets. This is similar to a standard merge in DATA step coding (where the extra IF statement shown is not necessary).

The output dataset alone is not very useful, as there is no variable to show the state/province name- these names are spread between MAPIDNAME and STATENAME. For that reason, we create an extra variable COMBINEDNAME, which takes the first non-missing value between MAPIDNAME and STATENAME. This process is done within the COALESCE function, which is your friend when performing full joins. COALESCE is especially useful when both input datasets have the same variable name and are in need of combining while performing a full join. For example, if MAPIDNAME appeared in both datasets, joining on A.MAPIDNAME=B.MAPIDNAME would then require a 'COALESCE(A.MAPIDNAME,B.MAPIDNAME) as MAPIDNAME' in the SELECT clause to properly combine both variables without generating a warning.

Less than Straight Forward Joins

As opposed to DATA step merges, SQL joins allow you to join datasets based on derivations of existing variables. You can also perform joins based on different criteria for joining per input dataset, among other unique situations. These joins are done within SQL itself and do not require prior preparation. There are many examples of how this can be effective, but we will only discuss a few in this paper.

Format Discrepancies Between Joining Variables

With real data, we often don't have easily mapped variables on which to merge. Table 16 shows as example of how this can be the case. In this example, we need to include the CHANGE_2010 variable from US_DATA on the proper MAPIDNAME record in GCSTATE, and will use MAPIDNAME2 instead of MAPIDNAME to demonstrate how we can get around the issue of differently formatted merging variables.

SQL Coding	DATA Step Coding

proc sql;
  create table data1 as
  select A.*, B.change_2010
  from sashelp.gcstate A
  left join sashelp.us_data B
  on A.mapidname2 =
  upcase(compress(B.statename))
  order by A.mapidname2;
quit;
	proc sort data = sashelp.gcstate
	  out = gcstate;
	  by mapidname2;
	run;
	
	data us_data;
	  set sashelp.us_data;
	  mapidname2 =
	  upcase(compress(statename));
	  keep change_2010 mapidname2;
	run;
	
	proc sort data = us_data;
	  by mapidname2;
	run;
	
	data data1;
	  merge gcstate (in=a) us_data;
	  by mapidname2;
	  if a;
	run;

Table 16. Performing a left join using differently formatted ON variables

MAPIDNAME2 is all caps and is compressed to have no spaces between 2-word state names (i.e. 'NEWYORK'). STATENAME has the proper case and spacing (i.e. 'New York'). In the DATA step merge, these variables need to be exactly the same in both datasets to properly merge, and differences in length can lead to truncation and potentially improper merging. SQL joins allow you to apply functions (in this case, UPCASE and COMPRESS) to the STATENAME variable to make it conform to MAPIDNAME2 in the ON clause itself, and variable lengths accompanied by truncation warnings are not an issue.

Many-to-Many Joins

SQL flexes its muscles with many-to-many joins, as this process becomes very complex and tedious in DATA step coding (if even possible). What is meant by a many-to-many join is the joining of 2 or more datasets where there is more than one record per joining variable

combination. All possible combinations of all records between 2 input datasets are in the output dataset. By default, SQL performs many-to-many joins where the datasets warrant a many-to-many join (i.e. both input datasets have at least one case where more than one record per combination of joining variables).

The following is an example of a many-to-many join using the CARS dataset. The desired output dataset matches all records in the CARS dataset to every other record in the dataset with the same TYPE. The variables we are keeping are TYPE, MODEL, ORIGIN and MSRP for both cars being matched together, as we do in the following code:

```
proc sql;
  create table data1 as
  select A.type, A.model, A.origin, A.msrp
    , B.model as model2, B.origin as origin2, B.msrp as msrp2
  from sashelp.cars A
  left join sashelp.cars B on A.model ^= B.model and A.type = B.type
  order by A.type, A.model, B.model;
quit;
```

Since SQL does this type of task with relative ease, DATA step coding is not included for this example.

Notice how each record has now multiplied to the number of records within the same value of TYPE (e.g. each truck now has a record matching it with every other truck), less one since we excluded the records where it was matched to itself (see the ON clause). This is most evident in the hybrid cars, as there are 3 total records in the input datasets, but now there are 6 records in the output dataset- one for each pairwise matching.

Conditional Joins

Conditional joins apply different criteria to the join depending on the values of the records. To demonstrate, we build on the previous example of a many-to-many join to further narrow down the list of matched cars.

In the following example, we now match each record by TYPE and ORIGIN. However, we still match all sports cars to every other sports car, regardless of ORIGIN. This is done using the following code:

```
proc sql;
  create table data1 as
  select A.type, A.model, A.origin, A.msrp
    , B.model as model2, B.origin as origin2, B.msrp as msrp2
  from sashelp.cars A
  left join sashelp.cars B on A.model ^= B.model and A.type=B.type
    and (A.type = 'Sports' or (A.type ^= 'Sports' and A.origin = B.origin))
  order by A.type, A.model, B.model;
quit;
```

We take this example one step further by now searching for other cars within the same TYPE (and in the case of all non-sports cars, the same ORIGIN), that are priced within $1000 of the car in the applicable record, as seen here:

```
proc sql;
  create table data1 as
  select A.type, A.model, A.origin, A.msrp
    , B.model as model2, B.origin as origin2, B.msrp as msrp2
  from sashelp.cars A
```

```
  left join sashelp.cars B on A.model ^= B.model and A.type=B.type
    and (A.type = 'Sports' or (A.type ^= 'Sports' and A.origin = B.origin))
    and B.msrp-1000 <= A.msrp <= B.msrp+1000
  having model2 is not null
  order by A.type, A.model, B.model;
quit;
```

Joins Involving More than Two Datasets

Joining more than 2 input datasets at a time by different variables is done in one query in SQL, instead of multiple merges by different sets of variables. The following example shows how this process is done. We are using a subquery to create an input dataset with the TYPE and mean of MSRP by TYPE. In addition, we are using another subquery to create an input dataset with the ORIGIN and mean of MSRP by ORIGIN.

```
proc sql noprint;
  create table data1 as
  select A.make, A.type, A.model, A.origin, A.msrp
    , B.mean as meantype format=dollar8.
    , C.mean as meanorigin format=dollar8.
  from sashelp.cars A
     left join (select type, mean(msrp) as mean from sashelp.cars
       group by type) B on A.type=B.type
     left join (select origin, mean(msrp) as mean from sashelp.cars
       group by origin) C on A.origin=C.origin
     order by A.make, A.type, A.model;
quit;
```

This process would require more than one merge in a DATA step, along with all of the SORT procedures necessary to perform each merge. In SQL, both of these merges are done in the same query.

SITUATIONS TO NOT USE SQL

This paper focuses on the capabilities of SQL, and touches on some situations that are much easier done and less error-prone using SQL. Despite the powers of SQL, there are several situations where you may determine that SQL coding is not as efficient or necessary. Often, a mixture of SQL and DATA steps can provide what you need, and it is best to know those before chasing a solution that may not exist. Here are a few of those reasons you may want to use a DATA step instead of SQL:

1. The programming is relatively simple and does not involve merges/joins or summary variables. For easy datasets, the amount of code to create those datasets in SQL may not be worth the time. If, however, a user knows that subsequent datasets will need to be created and merged within a program, it may be best to start with SQL and build upon the code as the program requires more derivations, as they inevitably do.
2. Stacking of datasets is required. SQL uses the UNION clause to stack datasets, which is not covered in this paper. It is my opinion that given the nuances of unions in SQL, stacking is much easier using a SET statement in a DATA step.
3. Multiple records will be created from one record. The OUTPUT command in the DATA step is extremely powerful in creating multiple records, something PROC SQL does not do easily, if at all.
4. The output dataset has no input dataset, but needs to be created from scratch. Through the use of DATALINES/CARDS and OUTPUT commands, datasets created from scratch (e.g. a shell dataset to be merged with other data) are best left to a DATA step.

5. There are multiple variables created by a single IF/THEN DO block. SQL requires that each variable is created independently, so if the user intends to create multiple variables within the same logical block, it may be best in a DATA step.
6. The same attributes of length and format need to be applied to many variables. As variables are created independently, along with attributes being set independently, if a lot of the attributes are the same between variables, this process can be tedious. It may be best to create the variables in SQL, then update the attributes with LENGTH and FORMAT statements afterwards in a DATA step.
7. Other programmers will need to use or modify your code. While this is said somewhat in jest, a small percentage of programmers seem to have the ability to read and create SQL code, especially when it becomes more complicated with several joins and/or subqueries. If your code is intended to be shared with others with little knowledge of SQL coding, there may be a learning curve to help others understand your code. That being said, programming in SQL could offer invaluable job security (again, said in jest).

CONCLUSION

SQL provides many advantages to the traditional methods that, in general, we are initially taught. Using SQL coding, either in tandem with or in lieu of DATA step coding, several tasks we do all the time can be done either more efficiently and/or with less room for error. I encourage you to use and build upon the examples described in this paper to become more proficient in SQL and add to your SAS toolbox.

RECOMMENDED READING

- Lund, Pete. 2005. “An Introduction to SQL in SAS.” *SUGI 30 Proceedings*. Available at https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/257-30.pdf.

CONTACT INFORMATION

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

Brett Jepson
Rho, Inc.
801-390-8501
Brett.Jepson@gmail.com