PROC SQL: Tips and Translations for Data Step Users

Susan P Marcella, ExxonMobil Biomedical Sciences, Inc.
Gail Jorgensen, Palisades Research, Inc.

ABSTRACT

SAS® has always been an extremely powerful data manipulation language. The inclusion of PROC SQL in the SAS package made a very powerful addition to the SAS programmer's repertoire of tools. However, for those of us who learned SAS before the addition of PROC SQL, the terminology and very different syntax of this procedure may be enough to prevent us from taking advantage of its flexibility and usefulness. There are several books that teach the concepts of queries, joins, and relational databases to instruct the novice user. This paper, however, is aimed at providing a clear explanation of the PROC SQL query by comparing it to the already-familiar Data step, and will offer some tips and techniques for the types of situations when PROC SQL can be particularly effective and useful.

SQL BASICS

Before going into the details of how to use PROC SQL, we will provide a quick overview of some of the fundamentals of SQL.

TERMINOLOGY

A quick primer on terminology used in PROC SQL and this paper:

PROC SQL thinks in terms of ***tables***, rather than datasets. In keeping with this concept, observations are called ***rows***, and variables are called ***columns***. In this paper, these terms are used interchangeably.

SYNTAX

In order to use PROC SQL, there are a few basics that you need to understand. Generally, a query is structured as follows:

```
Proc SQL;
  create table/view newdsname as
  select var1, var2, … varN
  from dsname
  where   condition     ;
Quit;
```

An invocation of PROC SQL starts with the **PROC SQL** statement. The SQL procedure, like the DATASETS procedure, invokes an environment that stays in effect until ended with a **QUIT** statement. This allows you to process several queries without having to keep reissuing the PROC SQL statement. (While the QUIT statement is officially required to exit the SQL environment, SAS is smart enough to exit the environment automatically if another **PROC** or **DATA** statement is encountered.)

Queries start with a **CREATE TABLE** (or **CREATE VIEW**) statement or a **SELECT** statement. The CREATE TABLE statement (we will discuss Views later in this paper), is the equivalent of the DATA statement – it identifies the table (dataset) to be created. If a CREATE TABLE statement is not included in a query, then the results of the query are sent to the OUTPUT window, making the query essentially equivalent to a DATA Step followed by a PROC PRINT.

The **SELECT** statement is the heart of an SQL query. The SELECT statement identifies the variables to be created or selected from the incoming dataset. Unlike 'regular' SAS procedures and Data steps, SQL likes commas. Variable names in a query are separated by commas, except for the last variable listed before the FROM clause. You can select existing variables in a SELECT statement, or create new ones; you can assign literals as variable values, or assign values conditionally. We will explore all these options.

The **FROM** clause is the equivalent of the SET or MERGE statement; it identifies the incoming dataset(s).

The **WHERE** clause performs the same function as the WHERE or subsetting IF statements in a Data Step, allowing conditional selection of rows.

There are several other optional clauses in an SQL procedure, but these few constitute the basics of a query. We will be examining several of the additional clauses later in this paper.

A query (defined by a SELECT statement and one or more of the clauses described above), is ended by a semicolon. Unlike in a 'regular' SAS procedure, each subclause of the query does not end with a semicolon. The semicolon signals the end of the entire query. Multiple queries, each ending with a semicolon, may be run under a single PROC SQL statement.

JOINS DEMYSTIFIED

In speaking with many experienced SAS programmers, I find that one of the things they find most confusing in SQL is the concept of different types of *joins*. Joins are simply SQL terminology for merging datasets on a common variable or variables. There are two basic categories of joins in SQL: ***inner joins***, which select *only matching* records from the incoming datasets, and ***outer joins***, which select *all the records* (even non-matching ones) from one or more of the incoming datasets.

The best method of showing how the SQL query and DATA step compare is to show examples. Using the following datasets (Drinkers and Smokers), side-by-side examples of SQL queries and comparable DATA steps will be shown, with explanations and notes on differences and issues of note.

For this paper, the following data files will be used (this is a subset of the entire datafile for reference):

Drinkers:

Obs	SubjID	Height	Weight	EverAlc	CurrentAlc	EverBeer	CurrentBeer	EverGW	CurrentGW	EverRW
1	700121	160	55	Yes	Yes	Yes	Yes	No	No	Yes
2	700123	165	54	Yes	Yes	Yes	Yes	No	No	Yes
3	700129	170	75	Yes	Yes	Yes	Yes	No	No	No
4	700130	163	82	Yes	Yes	No	No	No	No	Yes
5	700136	167	60	Yes	Yes	Yes	Yes	No	No	No
6	700146	156	60	Yes	Yes	Yes	Yes	No	No	No
7	700147	168	60	Yes	Yes	Yes	Yes	No	No	No
8	700148	158	70	Yes	Yes	Yes	Yes	No	No	No
9	700150	174	63	Yes	Yes	Yes	Yes	No	No	No
10	700153	170	56	Yes	Yes	Yes	Yes	No	No	No

Smokers:

Obs	ID	Height	Weight	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow
1	700123	165	54	Yes	Yes	Yes	Daily
2	700126	175	73	Yes	Yes	No	Occasionally
3	700129	170	75	Yes	Yes	Yes	Daily
4	700134	171	55	Yes	No	No	Occasionally
5	700152	168	68	Yes	Yes	Yes	Daily
6	700156	176	55	Yes	No	No	Occasionally
7	700161	167	75	Yes	Yes	Yes	Daily
8	700166	168	79	Yes	Yes	Yes	Daily
9	700167	170	75	Yes	Yes	Yes	Daily
10	700168	165	60	Yes	No	No	Occasionally

INNER JOINS

In DATA step terms, an *inner join* on two incoming datasets is equivalent to using a **MERGE** statement with an ***IF ina and inb*** statement. Only records found in both datasets will be in the output dataset. Using our example files, if we want to pull the records for everyone who both drinks and smokes, we would use the following Data Step or PROC SQL code:

```
proc sort data=L.drinkers; by subjid; run;
proc sort data=L.smokers; by id; run;

data L.IJSmokeDrinkDATA;
 merge L.smokers(in=smoke)
          L.drinkers(in=drink re-
name=(subjid=id));
 by id;
 if smoke and drink;
run;
```

```
proc sql;
 create table L.IJSmokeDrinkSQL as
 select smokers.*, drinkers.*
 from L.smokers, L.drinkers
 where smokers.id=drinkers.subjid;
quit;
```

The code above introduces a few concepts that need to be noted:

- **Selecting variables in a dataset**. In a DATA step, unless otherwise specified in a KEEP or DROP statement, all variables in all input datasets will occur in the output dataset. In PROC SQL, you must name each variable you want in the output dataset. You can use the * (asterisk) character to select all variables from a dataset. When there are multiple input datasets, it is necessary to indicate the input dataset from which each variable is taken by preceding the variable name with the dataset name (e.g., Smokers.id indicates the id variable from the Smokers dataset, while Drinkers.* indicates all variables in the Drinkers dataset).
- **Merging on differing field names**. Since the subject id field in the two datasets does not have the same name (SubjID in the Drinkers dataset, ID in the Smokers dataset), the Data Step method of merging requires that you rename one of the fields. In PROC SQL, it is not necessary to do this. However, note that, since we are selecting all fields from both input datasets, both id columns appear in the SQL output dataset.
- **Conditioning joins with a WHERE statement**. The WHERE statement can contain any condition on which to match the datasets.
- **No sorting needed**. Notice that, in the PROC SQL code, there is no preceding sort. One of the advantages of using PROC SQL is that it will merge the incoming datasets appropriately ***without*** re-sorting them. This can be very useful in instances where you wish to retain the original order of your input data.

The results of both the DATA Step and the PROC SQL methods are essentially the same, with the exception of the appearance of both id variables (id and subjid) in the PROC SQL output.

Results from Data Step:

Obs	ID	Height	Weight	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow	EverAlc	CurrentAlc	EverBeer
1	700123	165	54	Yes	Yes	Yes	Daily	Yes	Yes	Yes
2	700129	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
3	700161	167	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
4	700167	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
5	700168	165	60	Yes	No	No	Occasionally	Yes	Yes	Yes
6	700558	179	73	Yes	Yes	Yes	Daily	Yes	Yes	No
7	700559	175	60	Yes	Yes	Yes	Daily	Yes	Yes	Yes
8	700560	183	91	Yes	Yes	Yes	Daily	Yes	Yes	No
9	700561	174	85	Yes	Yes	Yes	Daily	Yes	Yes	Yes
10	700564	178	70	Yes	Yes	Yes	Daily	Yes	Yes	No

Results from SQL Step:

Obs	ID	Height	Weight	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow	SubjID	EverAlc	CurrentAlc	EverBeer
1	700123	165	54	Yes	Yes	Yes	Daily	700123	Yes	Yes	Yes
2	700129	170	75	Yes	Yes	Yes	Daily	700129	Yes	Yes	Yes
3	700161	167	75	Yes	Yes	Yes	Daily	700161	Yes	Yes	Yes
4	700167	170	75	Yes	Yes	Yes	Daily	700167	Yes	Yes	Yes
5	700168	165	60	Yes	No	No	Occasionally	700168	Yes	Yes	Yes
6	700558	179	73	Yes	Yes	Yes	Daily	700558	Yes	Yes	No
7	700559	175	60	Yes	Yes	Yes	Daily	700559	Yes	Yes	Yes
8	700560	183	91	Yes	Yes	Yes	Daily	700560	Yes	Yes	No
9	700561	174	85	Yes	Yes	Yes	Daily	700561	Yes	Yes	Yes
10	700564	178	70	Yes	Yes	Yes	Daily	700564	Yes	Yes	No

OUTER JOINS

There are three types of outer joins:

- LEFT JOIN: A *left join* is the equivalent of using the ***IF ina*** DATA Step statement; it selects all records from table A and only matching records from table B
- RIGHT JOIN: A *right join* is the equivalent of using the ***IF inb*** DATA Step statement; it selects matching records from table A and all records from table B
- FULL JOIN: A *full join* is the equivalent of a DATA Step with **no** subsetting IF statement; it selects all records from both incoming datasets.

LEFT JOINS:

A **left join** takes all the records from the table on the left and merges them with matching records from the table on the right. The *left* and *right* designation refers to the position of the dataset names in the FROM statement: the first table named in the FROM statement is the **left** dataset and the last table named in the FROM statement is the **right** dataset (one limitation of SQL joins is that you can only perform an outer join on two tables at a time; inner joins can be performed on multiple datasets).

Using our sample files, if we want to select all drinkers and add their smoking information, we would use the following code. (Note that, in this example, we are going to assume that both input datasets have the key field named ID.)

```
proc sort data=L.drinkers; by id; run;
proc sort data=L.smokers; by id; run;

data L.LJSmokeDrinkdata;
 merge L.smokers(in=smoke) L.drinkers(in=drink);
 by id;
 if smoke;
run;
```

```
proc sql;
 create table L.LJSmokeDrinkSQL as
 select s.*, d.*
 from L.smokers as s left join L.drinkers as d
 on s.id=d.id;
quit;
```

A few notes about the statements in the above PROC SQL code:

- The keywords **LEFT JOIN** replace the comma between the datasets in the FROM statement
- When using an outer join, the keyword **WHERE** is replaced by the keyword **ON**
- PROC SQL allows the use of an alias to replace dataset names when identifying variables. An **alias** is a shortened ‘nickname’ for a dataset that can be used in SELECT statements to identify the dataset in

which a variable is found. Aliases are assigned in the FROM statement, after the keyword AS. In the example above, the statement FROM L.Drinkers AS d LEFT JOIN d L.Smokers AS s
assigns the alias **d** to the Drinkers dataset and the alias **s** to the Smokers dataset, allowing us to use the notation d.* and s.* instead of drinkers.* and smokers.* in our SELECT statment. Aliases can be any length, but since the whole point of using them is to avoid having to type long dataset names, it makes sense to keep them short.

When comparing the output datasets created by both the merge and join, we find that they are identical.

Results from Data Step:

Obs	ID	Height	Weight	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow	EverAlc	CurrentAlc	EverBeer
1	700123	165	54	Yes	Yes	Yes	Daily	Yes	Yes	Yes
2	700126	175	73	Yes	Yes	No	Occasionally			
3	700129	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
4	700134	171	55	Yes	No	No	Occasionally			
5	700152	168	68	Yes	Yes	Yes	Daily			
6	700156	176	55	Yes	No	No	Occasionally			
7	700161	167	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
8	700166	168	79	Yes	Yes	Yes	Daily			
9	700167	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
10	700168	165	60	Yes	No	No	Occasionally	Yes	Yes	Yes

Results from SQL:

Obs	ID	Height	Weight	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow	EverAlc	CurrentAlc	EverBeer
1	700123	165	54	Yes	Yes	Yes	Daily	Yes	Yes	Yes
2	700126	175	73	Yes	Yes	No	Occasionally			
3	700129	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
4	700134	171	55	Yes	No	No	Occasionally			
5	700152	168	68	Yes	Yes	Yes	Daily			
6	700156	176	55	Yes	No	No	Occasionally			
7	700161	167	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
8	700166	168	79	Yes	Yes	Yes	Daily			
9	700167	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
10	700168	165	60	Yes	No	No	Occasionally	Yes	Yes	Yes

RIGHT JOINS:

A **right join** selects all the records in the last-named dataset (in the FROM statement) and only the matching records from the first-named dataset. Continuing with our sample files, if we want all smokers with their drinking information, we would use the following code:

```
proc sort data=L.drinkers; by id; run;
proc sort data=L.smokers; by id; run;

data L.RJSmokeDrinkData;
merge L.smokers(in=smoke)
        L.drinkers(in=drink);
by id;
if drink;
run;
```

```
proc sql;
create table L.RJSmokeDrinkSQL as
select s.*, d.*
from L.smokers as s right join L.drinkers as d
on s.id=d.id;
quit;
```

Notice, however, that our output this time is NOT the same; for those smokers who have no drinking information, the ID, Height, and Weight fields are blank. This is due to a very important difference in the way SQL handles values of variables found in both incoming datasets. Unlike the DATA Step, when SQL encounters variables of the same name in both datasets, it keeps the value of the **first-seen** dataset. In our example, since the Smokers dataset is named first in the SELECT statement, the value of any variable that exists in both datasets will come from the Smokers dataset. For those records where there is no matching Smokers record, the value of these fields will be blank in our output dataset.

Data Step Results:

Obs	ID	Height	Weight	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow	EverAlc	CurrentAlc	EverBeer
1	700121	160	55				No	Yes	Yes	Yes
2	700123	165	54	Yes	Yes	Yes	Daily	Yes	Yes	Yes
3	700129	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
4	700130	163	82				No	Yes	Yes	No
5	700136	167	60				No	Yes	Yes	Yes
6	700146	156	60				No	Yes	Yes	Yes
7	700147	168	60				No	Yes	Yes	Yes
8	700148	158	70				No	Yes	Yes	Yes
9	700150	174	63				No	Yes	Yes	Yes
10	700153	170	56				No	Yes	Yes	Yes

PROC SQL Results:

Obs	ID	Height	Weight	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow	EverAlc	CurrentAlc	EverBeer
1	.	.	.					Yes	Yes	Yes
2	700123	165	54	Yes	Yes	Yes	Daily	Yes	Yes	Yes
3	700129	170	75	Yes	Yes	Yes	Daily	Yes	Yes	Yes
4	.	.	.					Yes	Yes	No
5	.	.	.					Yes	Yes	Yes
6	.	.	.					Yes	Yes	Yes
7	.	.	.					Yes	Yes	Yes
8	.	.	.					Yes	Yes	Yes
9	.	.	.					Yes	Yes	Yes
10	.	.	.					Yes	Yes	Yes

In order to prevent this type of occurrence, when using a right or left join, make sure that the dataset from which you want to keep all records is in the appropriate position on the SELECT statement (i.e., in a right join, specify the ‘right’ dataset first in the SELECT statement; in a left join, specify the ‘left’ dataset first).

```
proc sql;
create table L.RJSmokeDrinkSQLb as
select d.*, s.*
from L.smokers as s right join L.drinkers as d
on s.id=d.id;
quit;
```

Our output from this code now matches that from the Data Step, except that the columns from the Drinkers data-set now appear before the columns from the Smokers dataset (since we have now listed the Drinkers dataset first in our SELECT statement).

Obs	ID	Height	Weight	EverAlc	CurrentAlc	EverBeer	EverSmoked	Ever100Cigs	EverSmokeDaily	SmokeNow
1	700121	160	55	Yes	Yes	Yes				No
2	700123	165	54	Yes	Yes	Yes	Yes	Yes	Yes	Daily
3	700129	170	75	Yes	Yes	Yes	Yes	Yes	Yes	Daily
4	700130	163	82	Yes	Yes	No				No
5	700136	167	60	Yes	Yes	Yes				No
6	700146	156	60	Yes	Yes	Yes				No
7	700147	168	60	Yes	Yes	Yes				No
8	700148	158	70	Yes	Yes	Yes				No
9	700150	174	63	Yes	Yes	Yes				No
10	700153	170	56	Yes	Yes	Yes				No

FULL JOINS:

In a **full join**, we pull all records from both input datasets, merging those that match, but including all records -- even those that don't match. For this example, we are going to use a different pair of datasets to enable us to highlight some other features of PROC SQL.

Our new sample datasets contain information for subjects in two different smoking studies. There is some over-lap of subjects between the studies – a subject could be in Study A or Study B or both. Both datasets contain identically named variables.

StudyA Data:

Obs	id	Weight	IntDate	EverSmoked	SmokeNow
1	448	60	13JUL2006	Yes	Daily
2	449	64	13JUL2006		
3	450	70	14JUL2006	Yes	Daily
4	451	70	13JUL2006	Yes	Daily
5	452	60	14JUL2006		
6	453	82	13JUL2006		
7	454	70	13JUL2006		
8	455	80	13JUL2006		
9	7001	53	22OCT2003	No	No
10	7002	55	22OCT2003	No	

StudyB Data:

Obs	id	Weight	IntDate	EverSmoked	SmokeNow
1	457	49	13JUL2006	Yes	Daily
2	458	74	13JUL2006	Yes	Daily
3	459	65	13JUL2006	No	
4	460	45	13JUL2006		
5	461	60	13JUL2006		
6	462	70	13JUL2006	No	
7	463	65	13JUL2006	Yes	Daily
8	464	65	13JUL2006	No	
9	465	70	13JUL2006		
10	466	60	13JUL2006		

In this example, we want to pull the data from both studies for each subject, keeping all subjects from both studies. Since some of the variables occur with the same name in both input datasets, and we want to keep the values from both datasets, we will need to rename them for the output dataset so that we can have both sets of variables in each record. One of the advantages of using PROC SQL is that we can easily rearrange the order in which the variables appear on the output dataset by simply specifying the desired variables in the order in which we want them to appear. This allows us to juxtapose similar variables for easier comparison.

PROC SQL also allows you to rename variables, or even create new variables, as you specify them in the SELECT statement, by using the AS keyword followed by the new variable name.

```
proc sort data=L.studya; by id; run;
proc sort data=L.studyb; by id; run;

data L.FJStudyData;
merge L.studya (rename=(weight=WeightA
            intdate=IntDateA
            eversmoked=EverSmokedA
            smokenow=SmokeNowA))
     L.studyb (rename=(weight=WeightB
                        intdate=IntDateB
                        eversmoked=EverSmokedB
                        smokenow=SmokeNowB));
by id;
run;
```

```
proc sql;
create table L.FJStudySQLx  as
select a.id, a.weight as WeightA,  b.weight as WeightB,
         a.intdate as IntDateA,  b.intdate as IntDateB,
         a.eversmoked as EverSmokeA,
         b.eversmoked as EverSmokeB,
         a.smokenow as SmokeNowA, b.smokenow as SmokeNowB
from L.studya as a full join L.studyb as b
on a.id=b.id;
quit;
```

Data Step Results:

Obs	id	WeightA	IntDateA	EverSmokedA	SmokeNowA	WeightB	IntDateB	EverSmokedB	SmokeNowB
1	448	60	13JUL2006	Yes	Daily	.	.		
2	449	64	13JUL2006			.	.		
3	450	70	14JUL2006	Yes	Daily	.	.		
4	457	.	.			49	13JUL2006	Yes	Daily
5	458	.	.			74	13JUL2006	Yes	Daily
6	459	.	.			65	13JUL2006	No	
7	7001	53	22OCT2003	No	No	55	13JUL2006	No	
8	7002	55	22OCT2003	No		60	13JUL2006	No	
9	7003	45	22OCT2003	No		47	13JUL2006	No	

Obs	id	WeightA	IntDateA	EverSmokedA	SmokeNowA	WeightB	IntDateB	EverSmokedB	SmokeNowB
10	7004	80	22OCT2003	Yes	Daily	76	14JUL2006	Yes	Daily

PROC SQL Results:

Obs	id	WeightA	WeightB	IntDateA	IntDateB	EverSmokeA	EverSmokeB	SmokeNowA	SmokeNowB
1	448	60	.	13JUL2006	.	Yes		Daily	
2	449	64	.	13JUL2006	.				
3	450	70	.	14JUL2006	.	Yes		Daily	
4	.	.	49	.	13JUL2006		Yes		Daily
5	.	.	74	.	13JUL2006		Yes		Daily
6	.	.	65	.	13JUL2006		No		
7	7001	53	55	22OCT2003	13JUL2006	No	No	No	
8	7002	55	60	22OCT2003	13JUL2006	No	No		
9	7003	45	47	22OCT2003	13JUL2006	No	No		
10	7004	80	76	22OCT2003	14JUL2006	Yes	Yes	Daily	Daily

Notice that, in the case of the full join, we encounter the same problem that we found with the right join – we are missing the subject id for some of our records. The problem here is that, no matter which dataset we specify first on our SELECT statement, records from the other dataset will fail to have a value shown in the output. This leads us to a discussion of the ways in which to deal with duplicate variable names in PROC SQL.

DEALING WITH DUPLICATE VARIABLE NAMES:

There are three basic approaches to handling the issue of incoming datasets having variables with the same name:

- Always select the value from one dataset (this is actually the default, since SQL will take the value from the first-named dataset)
- Keep the variables from both datasets as separate variables in the output dataset
- Assign the value of the output variable based on the values of the incoming variables

The method we select depends on several factors, including whether or not the variable in question is a key field (a unique field used to identify which records match for merging).

Selecting the value from one dataset:

- Specify the table from which you want to get the variable value (e.g., always take the keyfield value from table A)

  ```
  SELECT a.keyfield, a. var1, a.var2, b.var3, b.var4
  FROM a left join b
  ON a.keyfield=b.keyfield;
  ```

  This method works for key fields ***only*** if you're using a one-sided merge (as discussed in the section on right joins) or if both datasets have all the same keyfield values (as with an inner join); otherwise, you end up with missing values, as we saw in the right join and full join examples

- Drop one of the variables (PROC SQL allows the use of all the same dataset options that you can use in a DATA step: DROP, KEEP, RENAME, etc).

  ```
  SELECT a.*, b.*
  FROM a left join b (drop=weight)
  ON a.idfld=b.idfield;
  ```

This method will not work for key fields, since dropping the variable in the FROM statement results in it not being read into the program vector; it would therefore be unavailable for matching.

Keeping the variable values from both datasets:

In PROC SQL, you can keep the incoming variables from both datasets the same way you would in a Data Step – by renaming the variable from one of the datasets.

```
SELECT a.keyfield, a.var1, a.var2, b.keyfield AS idfld, b.var3, b.var4
FROM a full join b
ON a.keyfield=b.keyfield;
```

This approach will work with key field variables, since values from both tables will be read in and will appear in the output dataset; however, the output dataset will have two separate columns with key field values in them – one named keyfield and one named idfld -- hardly an ideal outcome.
(Note also that the FROM statement referenced the key field on dataset B by its *original* name (keyfield) – this is because the new name (idfld) is only assigned to ***output*** dataset.

Selecting the value for the output dataset based on the values of the incoming variables:

This is usually the preferred method for handling duplicate key field variables, since it allows us to pick whichever incoming variable is non-missing or does not have an invalid value. There are two ways to accomplish this:

- The **CASE** statement -- PROC SQL's method of allowing conditional assignment of values
- The **COALESCE** function – a function that allows the selection of the first ***non-missing*** value in a list of variables.

The simplest way of handling duplicate key field variables is to use the COALESCE function. In our *full join* example above, we would use the following code:

```
proc sql;
     select coalesce(b.subjid, a.id) as ID,  a.weight as WeightA,  b.weight as WeightB,
                 a.height as HeightA, b.height as HeightB,
                 a.eversmoked,  a.smokenow, b.everalc, b.currentalc
    from L.studya as a full join L.studyb as b
          on a.id=b.subjid;
quit;
```

This code causes SQL to select the first non-missing value of either ID field and assign it to the output variable specified after the AS keyword. The output dataset key field variable can have the same name as the incoming variables (as in our example), or you could assign a new variable name after the AS keyword
(e.g., coalesce(b.subjid, a.id) as KEY). The resulting dataset shows that we have solved the problem of missing values in the key field variable of our output dataset.

Obs	ID	WeightA	WeightB	HeightA	HeightB	EverSmoked	SmokeNow	EverAlc	CurrentAlc
1	700121	.	55	.	160			Yes	Yes
2	700123	54	54	165	165	Yes	Daily	Yes	Yes
3	700126	73	.	175	.	Yes	Occasionally		
4	700129	75	75	170	170	Yes	Daily	Yes	Yes
5	700130	.	82	.	163			Yes	Yes
6	700134	55	.	171	.	Yes	Occasionally		
7	700136	.	60	.	167			Yes	Yes
8	700146	.	60	.	156			Yes	Yes
9	700147	.	60	.	168			Yes	Yes
10	700148	.	70	.	158			Yes	Yes

CONDITIONAL VARIABLE ASSIGNMENT IN PROC SQL

Many times, we wish to assign a value to a variable based on the results of a calculation or comparison or the value of another variable. In a DATA Step, this is a simple matter of using an IF/THEN statement. PROC SQL syntax does not allow for the use of the IF statement. However, conditional assignment of a value to a variable is possible using the **CASE** clause of the SELECT statement.

The syntax of the CASE clause can take two forms:

```
CASE
  WHEN (condition) THEN value
  WHEN (condition) THEN value
                   ELSE value
END AS varname
```

```
CASE(varname)
  WHEN (value) THEN statement
  WHEN (value) THEN statement
               ELSE statement
END AS varname
```

Note that the CASE clause can have any number of WHEN statements, but there should be an ELSE statement (much like the SELECT/WHERE construct in the DATA step). If there is no ELSE statement, a warning message will occur in the log, and a missing value will be assigned to the variable being created for any situation not covered by one of the WHEN statements. The CASE statement also requires an **END AS** statement; this is where you specify the name of the variable being created.

In the event of duplicate incoming variable names, a CASE clause could be used to select the value to assign to the output dataset variable as follows:

```
SELECT
 CASE
  WHEN (missing(a.id)) THEN b.id
                       ELSE a.id
 END AS ID,
 a.weight as WEIGHTA, b.weight as WEIGHTB, …
```

In the code above, if the ID variable in dataset A has a missing value (i.e., there is no matching record in dataset A for the ID in dataset B), the value of the ID field in dataset B will be used as the ID field in the output dataset; in all other situations, the value of the ID field from dataset A will be the value of the ID field in the output dataset. (Note that the entire CASE clause is treated like a single variable name, and should be followed by a comma before listing the rest of the variables to select from the incoming datasets.)

The CASE clause can be used with calculations and/or comparisons, as shown in the example below:

```
proc sql;
create table CaseExample as
select *,
   case
         when (weighta > weightb) and not missing(weighta) and not missing(weightb)  then 'Decrease'
         when (weighta < weightb) and not missing(weighta) and not missing(weightb)  then 'Increase'
         when (weighta = weightb) and not missing(weighta) and not missing(weightb)  then 'Stable'
         else ' '
   end as WeightChg,
   case (calculated weightchg)
         when ('Increase') then weightb-weighta
         when ('Decrease') then weighta-weightb
         else .
   end as WeigthDiff
from s.combinedstudiessql2;
```

A few things to note about this example:

- You can have multiple CASE clauses in a query
- To avoid getting a warning message in the log, we used an ELSE statement to assign a null value to any records that don't match one of the WHEN statements
- A CASE clause can only assign values to one variable; to assign values to multiple variables, you must use multiple CASE clauses
- To reference a variable that is created in the query (as opposed to being read in from an incoming data-set), you must preface the variable name with the keyword CALCULATED.

As you may have gathered from these examples, performing conditional variable assignments is one of the functions that is much easier and neater in a Data Step than in a PROC SQL query. However, the CASE clause will accomplish the task if you are using PROC SQL to take advantage of its many other strengths.

ADVANTAGES OF PROC SQL

So, now that we've compared SQL joins to Data Step merges, and seen that they can accomplish many of the same tasks, why would we want to use PROC SQL rather than the Data Step many of us are so much more used to?

There are, in fact, several capabilities of PROC SQL, beyond the useful but not terribly exciting ones that we have seen so far, that render it extremely useful, including some tasks that only it can do. We will explore some of them now.

DOWN CALCULATIONS:

In a Data Step, calculations are generally performed on variables *across* a single observation (with some limited access to previous observations available through the LAG function). One of the unique characteristics of PROC SQL, however, is that it permits the results of *downward* calculations to be added to observations. For instance, if we wish to calculate, for each subject in Study A, the difference between the individual's weight and the average weight of all subjects, we would need to use multiple Data Steps to first total all the weights, then calculate and save the average and use it to calculate the individual differences.

```
proc means data=L.StudyA noprint;
  var weight;
  output out=aw mean=AvgWeight;
run;

data fmaw; set aw; keep  dummy AvgWeight;
  dummy=1;  label AvgWeight='Average Weight';
run;

data fmStudyA; set L.StudyA; dummy=1; run;

data L.DownCalcData;
   merge fmStudyA fmaw;
   by dummy;
   DiffFromAvg=Weight-AvgWeight;
   drop dummy;
   format DiffFromAvg 8.2;
run;
```

Using this approach, we first calculate the average weight of all subjects, using PROC MEANS. We then add a 'dummy' variable to the dataset created by the MEANS procedure and our StudyA dataset, in order to provide a 'by' variable on which to merge the average with our study data. Then, we can finally perform the actual merge.

In PROC SQL, however, this can all be done in one short and simple step:

```
proc sql;
create table DownCalc1 as
select id, height, weight, everalc, currentalc,
        eversmoked, smokenow, weight - AVG(weight) as DiffFromAvg
from s.demogshort;
```

The **AVG** function in this example performs the calculation *down* the weight column, and makes the result available to each incoming record. Our output dataset would look like this:

Obs	ID	Height	Weight	DiffFromAvg
1	700118	160	55	-5.9696
2	700119	158	59	-1.9696
3	700120	140	42	-18.9696
4	700121	160	55	-5.9696
5	700122	159	58	-2.9696
6	700123	165	54	-6.9696
7	700124	167	75	14.0304
8	700125	163	55	-5.9696
9	700126	175	73	12.0304
10	700127	154	54	-6.9696

CONTROL BREAKS:

We can render this ability even more useful by including control breaks. For instance, we might want to group our subjects by smoking frequency, calculating the difference of each subject’s weight from the average weight of subjects in their smoking category, rather than from the average of the total population. To create control breaks in an SQL query, we add the **GROUP BY** clause, as follows:

```
proc sql;
create table DownCalc2 as
select id, weight, eversmoked, smokenow,
          avg(weight) as AvgWeight format 8.2,
          weight - AVG(weight) as DiffFromAvg format 8.2
from L.StudyA
group by smokenow;
quit;
```

Notice that, in our query, we also applied a format to the newly created variables AvgWeight and DiffFromAvg. Formats and labels can be applied to variables in an SQL query by following the name of the variable with the keyword FORMAT (or LABEL), followed by the desired format or text, without an equal sign (=). If the format and/or label applies to a variable that is not the last-named variable, it must appear ***before*** the comma that separates the variable from the next variable.

The results of this query show that the subjects are grouped by smoking status, and the results of the calculation are shown with only two decimal places, as specified by our format. We included the AvgWeight variable so you can see that each group has its own average. The calculation could just as well have been done without including the AvgWeight variable in our output dataset, however.

Obs	id	Weight	EverSmoked	SmokeNow	AvgWeight	DiffFromAvg
1	7090	60	No		63.59	-3.59
2	7012	80	No		63.59	16.41
3	8119	50	Yes	Daily	69.05	-19.05
4	8116	62	Yes	Daily	69.05	-7.05
5	8114	70	Yes	Daily	69.05	0.95
6	7007	65	Yes	Daily	69.05	-4.05
7	8110	59	Yes	No	62.75	-3.75
8	8127	74	Yes	No	62.75	11.25
9	7015	90	Yes	Occasionally	85.00	5.00
10	7025	80	Yes	Occasionally	85.00	-5.00

COUNTS AND SUMMARY FUNCTIONS:

The downward summary functions (COUNT, SUM, MIN, MAX, AVG) in PROC SQL can be extremely useful as tools for gathering quick information. For instance, if we are planning on creating a ‘Do loop’ or array to process multiple records for each subject, it would help to know the number of observations per subject that occur in our data. With PROC SQL, we can quickly determine the range of observations.

In our smoking study, we have multiple samples for each subject. Our Sample table looks like this:

Obs	ID	Height	Weight	SmokeNow	samplecode	timefrom	timeto	Results	SampDate
1	700559	175	60	Daily	AH1873	7:15:00	11:30:00	26.91	21OCT2003
2	700559	175	60	Daily	AH1574	8:30:00	11:30:00	245.50	20OCT2003
3	700559	175	60	Daily	AH3049	8:00:00	11:00:00	0.01	23OCT2003
4	700559	175	60	Daily	QW9775	8:10:00	9:50:00	59.31	27OCT2003
5	700559	175	60	Daily	AH1628	8:40:00	12:30:00	55.36	17OCT2003
6	700559	175	60	Daily	QF0248	8:15:00	12:40:00	29.40	13OCT2003
7	700559	175	60	Daily	QF0974	8:00:00	12:00:00	3149.48	14OCT2003
8	700559	175	60	Daily	QF0500	8:00:00	10:00:00	58.63	24OCT2003
9	700559	175	60	Daily	QW9996	9:00:00	13:10:00	44.42	16OCT2003
10	700559	175	60	Daily	AW3292	9:10:00	13:10:00	32.46	16OCT2003
11	700560	183	91	Daily	QF1271	8:20:00	12:50:00	0.21	16OCT2003
12	700560	183	91	Daily	AH1371	8:15:00	13:10:00	0.01	21OCT2003
13	700560	183	91	Daily	AH1093	8:15:00	13:05:00	0.60	17OCT2003
14	700560	183	91	Daily	QW9922	8:15:00	12:30:00	0.01	15OCT2003
15	700560	183	91	Daily	QW9891	8:10:00	13:50:00	0.01	14OCT2003
16	700560	183	91	Daily	QF0702	8:30:00	12:20:00	0.61	20OCT2003
17	700560	183	91	Daily	AH2140	8:20:00	12:30:00	0.01	27OCT2003
18	700560	183	91	Daily	AH2131	8:15:00	13:50:00	0.01	24OCT2003
19	700560	183	91	Daily	QF0384	8:25:00	13:45:00	0.01	13OCT2003
20	700560	183	91	Daily	AH1532	8:15:00	13:00:00	0.01	23OCT2003

We need to know the minimum and maximum number of samples we have for any subject so we can set up the parameters of our ‘do loop’ or array or whatever other processing we need to do. With the following query, we can easily count the number of samples per subject, saving the results into a new table, and then find the minimum and maximum counts:

```
proc sql;
  create table SampleCounts as
  select distinct id, count(*) as SampleCnt
  from s.smokesamples
  group by id;

 select min(samplecnt) as MinCnt, max(samplecnt) as MaxCnt
 from samplecounts;
quit;
```

The result of the first query is to produce a table with one row per subject id, containing the number of samples for that subject:

SampleCounts table:

ID	SampleCnt
700559	10
700560	11
700561	10
700564	7
700567	5
700569	10
700570	10
700572	5

The second part of the query calculates the minimum and maximum sample counts in the dataset:

Output:

MinCnt	MaxCnt
1	13

With this information, we can decide how to handle any iterative processing we might need to do.

OTHER SAS FUNCTIONS:

It should be noted here that, while the summary functions of COUNT, MIN, MAX, and AVG perform their calculations down rows in PROC SQL, other SAS functions can be used in an SQL query the same way as in a Data step. For instance, in our CASE statement example (on page 11), we could have performed the task of the second CASE statement by using the ABS function:

```
proc sql;
create table CaseExample as
select *,
  case
         when (weighta > weightb) and not missing(weighta) and not missing(weightb)  then 'Decrease'
         when (weighta < weightb) and not missing(weighta) and not missing(weightb)  then 'Increase'
         when (weighta = weightb) and not missing(weighta) and not missing(weightb)  then 'Stable'
         else ' '
  end as WeightChg,
 abs(weighta - weightb) as WeightDiff
from s.combinedstudiessql2;
```

NESTING QUERIES:

With PROC SQL, you can perform queries, including ones with downward calculations, on data from multiple tables. Suppose, in our previous example, we have the same type of sample data for our drinking subjects as for our smoking subjects. We want to calculate the average result for our smoking subjects vs our drinking subjects. We can accomplish this in one query:

```
proc sql outobs=1;
select
        (select avg(results) from s.smokesamples) as AvgSmokeResult,
        (select avg(results) from s.drinksamples) as AvgDrinkResult
from s.smokesamples;
Quit;
```

There are a few things you should note about this query:

- We start the query with a SELECT statement as usual
- We have used the option OUTOBS=1 in our PROC SQL statement. This is because, when you are nesting queries from multiple tables, SQL will automatically produce one line of output for each input record, even if you are only performing summary functions. Without the OUTOBS=1 option, we would have the average results printed as many times as there are records in the two input datasets.
- Each subquery, which contains its own SELECT statement with FROM clause, returns a single value which acts as a variable in the main SELECT statement.
- Each subquery is enclosed within parentheses, with the name of the variable being produced by that subquery **outside** the parentheses.
- The main query must contain a FROM clause, even though all the variables in the query are created by subqueries with their own FROM clauses. If there are other variables in the query that do not come from a subquery, the dataset named in the main FROM clause would be the dataset that contains those variables. In our example, where there are no variables not created in a subquery, you could use the name of any existing, non-empty dataset in the main FROM clause. (If you specify an empty dataset in the main FROM clause, no output will be produced, even if there are observations in the subquery datasets.)

The results of the above query are:

AvgSmokeResult	AvgDrinkResult
30.92641	39.83145

Another situation in which nesting queries can be very useful is when you want to select rows from one table based on values in another. For instance, perhaps I wish to look at the samples for only those subjects in my drinking study who are not also smokers. I do not have any smoking information in my sample table, and, since I only want non-smokers, I do not want to merge with my smoking table. The easiest way to accomplish this is to select the sample data for those subjects who do not appear in the smoking table:

```
proc sql;
create table NonSmokerSamples as
select * from L.Drinksamples
where id not in (select id from L.smokesamples);
quit;
```

The output would be as follows:

Obs	ID	Height	Weight	EverAlc	CurrentAlc	samplecode	timefrom	timeto	Results	SampDate
1	700565	164	60	Yes	Yes	QF1497	8:40:00	11:40:00	9.69	13OCT2003
2	700565	164	60	Yes	Yes	QF0705	7:30:00	14:00:00	3.73	15OCT2003
3	700565	164	60	Yes	Yes	AW3294	8:20:00	14:00:00	2.57	16OCT2003
4	700565	164	60	Yes	Yes	QF0410	7:40:00	14:00:00	3.05	16OCT2003
5	700565	164	60	Yes	Yes	QF0319	7:40:00	13:20:00	7.27	14OCT2003
6	700565	164	60	Yes	Yes	QF1418	7:20:00	11:20:00	15.14	17OCT2003
7	700618	160	65	Yes	Yes	QW9966	7:40:00	14:00:00	0.55	16OCT2003
8	700618	160	65	Yes	Yes	QF1235	7:30:00	14:00:00	1.88	15OCT2003
9	700618	160	65	Yes	Yes	QF0765	8:40:00	11:40:00	5.23	13OCT2003
10	700618	160	65	Yes	Yes	QW9915	7:40:00	13:20:00	2.53	14OCT2003
11	700618	160	65	Yes	Yes	QF1267	7:20:00	11:20:00	9.01	17OCT2003
12	702927	165	60	Yes	Yes	AH1344	12:30:00	19:25:00	71.90	08OCT2004
13	702927	165	60	Yes	Yes	AH0166	12:34:00	19:20:00	166.10	07OCT2004
14	702928	160	60	Yes	Yes	AH2138	12:30:00	19:20:00	88.80	07OCT2004
15	702928	160	60	Yes	Yes	AH1192	13:31:00	19:45:00	93.40	06OCT2004
16	702930	165	56	Yes	Yes	AH2062	4:10:00	11:35:00	431.90	06OCT2004
17	702930	165	56	Yes	Yes	AH2018	4:39:00	9:00:00	212.40	13SEP2004
18	702930	165	56	Yes	Yes	AH1759	4:47:00	11:40:00	185.00	04OCT2004
19	702930	165	56	Yes	Yes	AH1411	4:05:00	11:35:00	35.20	09OCT2004
20	702930	165	56	Yes	Yes	AH1007	3:50:00	11:20:00	45.10	08OCT2004
21	702930	165	56	Yes	Yes	AH0059	4:18:00	11:20:00	29.70	07OCT2004
22	702930	165	56	Yes	Yes	AH1202	4:25:00	11:30:00	227.20	10OCT2004
23	702955	175	70	Yes	Yes	AH2134	12:35:00	19:20:00	249.10	07OCT2004
24	702955	175	70	Yes	Yes	AH1742	12:50:00	19:20:00	35.10	10OCT2004
25	702955	175	70	Yes	Yes	AH1827	12:45:00	19:30:00	44.90	09OCT2004
26	702955	175	70	Yes	Yes	AH1088	4:32:00	11:40:00	192.30	04OCT2004

ACCESSING SAS DICTIONARY TABLES:

One of the things that can only be done with PROC SQL is accessing the SAS dictionary tables. While an in-depth discussion of these tables is outside the scope of this paper, it helps to know a few facts about the type of data that they contain, in order to understand why using PROC SQL to access them can be so helpful.

The following dictionary tables are frequently of use:

DICTIONARY.COLUMNS – Contains information about the columns in your datasets:

Column Name	Column Label
libname	Library Name
memname	Member Name
memtype	Member Type
name	Column Name
type	Column Type

Column Name	Column Label
length	Column Length
npos	Column Position
varnum	Column Number in Table
label	Column Label
format	Column Format
informat	Column Informat
idxusage	Column Index Type
sortedby	Order in Key Sequence
xtype	Extended Type
notnull	Not NULL?
precision	Precision
scale	Scale
transcode	Transcoded?

DICTIONARY.TABLES – Contains the following information about the your datasets:

Column Name	Column Label
libname	Library Name
memname	Member Name
memtype	Member Type
dbms_memtype	DBMS Member Type
memlabel	Data Set Label
typemem	Data Set Type
crdate	Date Created
modate	Date Modified
nobs	Number of Physical Observations
obslen	Observation Length
nvar	Number of Variables
protect	Type of Password Protection
compress	Compression Routine
encrypt	Encryption
npage	Number of Pages
filesize	Size of File
pcompress	Percent Compression
reuse	Reuse Space
bufsize	Bufsize
delobs	Number of Deleted Observations
nlobs	Number of Logical Observations
maxvar	Longest variable name
maxlabel	Longest label
maxgen	Maximum number of generations
gen	Generation number
attr	Data Set Attributes

Column Name	Column Label
indxtype	Type of Indexes
datarep	Data Representation
sortname	Name of Collating Sequence
sorttype	Sorting Type
sortchar	Charset Sorted By
reqvector	Requirements Vector
datarepname	Data Representation Name
encoding	Data Encoding
audit	Audit Trail Active?
audit_before	Audit Before Image?
audit_admin	Audit Admin Image?
audit_error	Audit Error Image?
audit_data	Audit Data Image?
num_character	Number of Character Variables
num_numeric	Number of Numeric Variables

DICTIONARY.MEMBERS – Contains the following external data about your datasets:

Column Name	Column Label
libname	Library Name
memname	Member Name
memtype	Member Type
dbms_memtype	DBMS Member Type
engine	Engine Name
index	Indexes
path	Pathname

With this information available to our queries, we can use PROC SQL to take some shortcuts. For instance, in one of our datasets, we have several variables that are flags for particular circumstances. There are many of these variables, and they all have names that contain the word ‘flag’. Using an SQL query, we can pull all these variables names, save them into a macro variable, and use that to allow us to print or process them without typing out all the names:

```
proc sql;
  select name into :flags separated by ' '
  from dictionary.columns
  where memname='HRDATA' and lowcase(name) contains 'flag';
quit;
proc freq data=L.hrdata;
tables &flags;
run;
```

Sample Output from the above code:

Flag_Absence_begin_date1	Frequency	Percent	Cumulative Frequency	Cumulative Percent
MMDDYY8 FORMAT	31210	100.00	31210	100.00

Frequency Missing = 5691

Flag_Absence_begin_date2	Frequency	Percent	Cumulative Frequency	Cumulative Percent
MMDDYY8 FORMAT	30544	100.00	30544	100.00

Frequency Missing = 6357

Flag_Absence_end_date1	Frequency	Percent	Cumulative Frequency	Cumulative Percent
MMDDYY8 FORMAT	30544	100.00	30544	100.00

Frequency Missing = 6357

Flag_Absence_end_date2	Frequency	Percent	Cumulative Frequency	Cumulative Percent
INVALID FORMAT	1	0.00	1	0.00
MMDDYY8 FORMAT	29985	100.00	29986	100.00

Frequency Missing = 6915

Notes about this query:

CREATING THE MACRO VARIABLE

- To create a macro variable, use the **INTO** keyword, and precede the name of the macro variable with a colon (**:**)
- To store multiple values in a single macro variable, follow the name of the variable with the keywords **SEPARATED BY** and the text (enclosed in quotes) that should separate each value. In our example, since we are using the macro variable *flags* in a PROC FREQ, we use a space (' ') as our separator. If we were creating the macro variable to be used in a PROC SQL query, then we would use a comma and a space (', ') as our separator.
- If we had wanted to store the column names into separate, numbered macro variables, we could have used the syntax SELECT name INTO :flag1 - :flag***n*** (where ***n*** is the number of flag variables we have). This assumes that we know ahead of time how many column names we have.
- Once created, use the macro variable as usual, by prefacing it with an ampersand (&)

SELECTING THE DESIRED COLUMNS

- Table names (memname) and library names (libname) are always stored in the dictionaries as all upper case.
- Column names (name) are stored in the dictionary the way they were created. Because of this, and since the selection criteria is case-sensitive, it is safest to use the UPCASE or LOWCASE functions to ensure that you match all the variables you want (e.g., *lowcase(name) contains 'flag'*).

- If you only want the variables from a particular dataset, it is important to restrict your selection to that particular dataset, and even, possibly, the particular library it is in, to avoid pulling similarly named variables from other datasets in your library.

When creating a single macro variable containing multiple variable names, it is important to know how you will use that macro variable. In addition to needing to use the appropriate separator between the variable names, you may need to append additional characters to each variable name as well. For instance, if we want to use our macro variable in a subsequent join query, we may need to have the variable names refaced with the name or alias of the dataset in which they occur (e.g., ***a.****flag_absence*). This can easily be accomplished by concatenating the name or alias of the dataset to the variable name, as shown in the following code:

```
proc sql;
  select 'h.'|| name into :flags separated by ', '
  from dictionary.columns
  where memname='HRDATA' and lowcase(name) contains 'flag';

  create table flagvariables as
  select h.id, &flags, m.*
  from L.hrdata as h left join moredata as m
  on h.id=m.id;
quit;
```

Note that, in this example, we separated the variable names with a comma and a blank (', ') since we were planning on using the variable names in an SQL query.

USING PROC SQL TO MAKE MASS CHANGES TO DATASETS:

We can use this ability to access the dictionary information in a variety of ways. In our study, we may find, after we have created our datasets, that one of our variables has an incorrect value. Since several of our tables may contain that particular variable, it is important to ensure that we fix the value in all tables that contain the relevant variable. If we do this manually, not only can it be an onerous task, but we run the risk of missing one or more instances. We can, however, use PROC SQL and the dictionary tables to identify all datasets that have the relevant variable and then use a macro to loop through and apply the change to each of them.

In the example below, we have discovered, after creating our datasets, that one of our subjects has an incorrect value in his/her SmokeNow variable. Since the SmokeNow variable occurs in several of our datasets, we want to make sure that we change the value for that subject in all of them. The code below accomplishes this:

```
%macro FixSmokeNow;
proc sql;

/* count number of datasets containing smokenow variable */                    A
select count(memname) into :dscnt
from dictionary.columns
where libname='L' and name='SmokeNow';


%let dscnt=&dscnt; /* removes trailing blanks from &dscnt */                   B


/* get names of all datasets containing the smokenow variable */                      C
select memname into :ds1 - :ds&dscnt
from dictionary.columns
where libname='L' and lowcase(name)='smokenow';
quit;

/* for each dataset, reassign value of smokenow for subject 712271 */          D
%do i=1 %to &dscnt;
       data L.&&ds&i;
       set L.&&ds&i;
       if jcml_id=712271 then smokenow='Daily';
       run;
```

```
        proc print data=L.&&ds&i noobs;                              E
        where jcml_id=712271;
        var jcml_id smokenow;
        title "Corrected Record in Dataset &&ds&i";
        run;
%end;
%mend fixsmokenow;

%fixsmokenow;
```

Let's look at this code step by step:

- Step A: We use the **COUNT** function to store the number of datasets, in the library named L, that contain the *SmokeNow* variable.
- Step B: Since all macro variables are created as text, but we are storing a number (which is right justified), the *dscnt* macro variable contains some leading blanks that interfere with concatenating it with the dataset name in the Do Loop parameters. The **%let dscnt=&dscnt** statement removes those blanks, enabling us to concatenate the count to our last macro variable name.
- Step C: Once we know how many datasets contain the *SmokeNow* variable, we can store the names of those datasets in a series of macro variables named **DS1** through **DS**&dscnt.
- Step D: Now that we have our dataset names stored in macro variables, we can set up a loop to read in each dataset, recreate it with the same name, and update the desired record.
- Step E: As part of our loop, we wish to print out the corrected observation for each updated dataset to create a record of which ones were updated.

Note that our macro contains a mix of PROC SQL queries and Data Steps. When writing any program, we frequently find that we switch between the two tools based on whatever seems simplest or most useful at the time. We could easily accomplish this task using only SQL queries by replacing steps D and E with the following code:

```
%do i=1 %to &sdscnt;
  update L.&&sds&i
         set smokenow='Daily'
         where jcml_id in (select subject_id from smokefixes);

  title "Reassigned SmokeNow Values In Dataset &&sds&i";
  select jcml_id, smokenow
  from L.&&sds&i
  where jcml_id in (select subject_id from smokefixes);
%end;
```

In this version, we use the **UPDATE** statement, which allows us to specify only the variable(s) to be changed in the query, without having to name all the other variables. This saves the trouble of having to type out the names of all the variables in the table.

VIEWS

An SQL ***View*** is a virtual table. Essentially, a view is a predefined window into one or more existing tables; unlike a physical table, it exists only as a definition and does not contain any actual data. A view will appear to a user to be just like a regular table, but no data is loaded into the view until it is accessed.

The ability to create views provides many advantages. In many databases, not all users are allowed to access all the data. In an HR database, for example, managers may be allowed to access the personnel data of employees in their department, but not data pertaining to employees in other departments. Different levels of management may have access to data across departments, but perhaps not to all details. We can create views of our data to restrict the fields and rows that each user is allowed to access. Since the views are essentially predefined queries against a set of tables, this method avoids actual duplication of data, facilitating database updates and preserving data integrity. It also provides the advantage to the user of always giving access to the most current data, since it is not necessary to refresh local or personalized versions of the tables.

Views are created in the same manner as tables, with a **CREATE VIEW** statement. Harking back to our combined drinkers and smokers table, which we created during our discussion of inner joins, we could have used the approach of creating a view, rather than a new physical table. In this manner, when the individual Drinkers and Smokers tables were updated, it would not be necessary to recreate the combined table as well, since the SmokeAndDrinkView would, whenever it was used, automatically recreate itself using the current data in the Smokers and Drinkers datasets.

```
proc sql;
create view s.SmokeAndDrinkView as
select Smokers.*, Drinkers.*
from L.Smokers, L.Drinkers
where drinkers.subjid=smokers.id;
quit;
```

With the exception of replacing the word **TABLE** with the word **VIEW**, this is the exact same query that we ran earlier. At its conclusion, there will be what looks like a table named *SmokeAndDrinkView* in our study library. It will appear and act just like a regular table: when we click on it, it will open and display the data; we can use it in other queries. To the end user, a view will behave exactly as does any other table (albeit a bit slower). However, from a database standpoint, we have only one copy of the data to maintain (helping to avoid our earlier scenario of having to change the value of a variable that occurs in multiple datasets). In addition, we can apply passwords and security restrictions to views just as we do to any SAS datasets. By constructing views for our users, we can easily restrict access to whichever rows and columns and combinations of datasets are necessary for each individual user without having multiple copies of our data to maintain.

CONCLUSIONS

The examples in this paper illustrate just a few of the advantages provided by using PROC SQL as part of your programming tool kit. We have found that using both SQL and the Data Step greatly increases the power and flexibility of the SAS language. Each of these tools has certain tasks that it accomplishes more easily or efficiently; our purpose in this paper is not to advocate one over the other, but to make PROC SQL more familiar and to illustrate the type of situations in which it can greatly enhance your code.

ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

We would like to thank Rob Schnatter and Wendy Huebner for allowing us to ‘borrow’ some of their data and for their support.

CONTACT INFORMATION

Contact the authors at:

Author Name:	Susan P Marcella	Gail Jorgensen
Company:	ExxonMobil Biomedical Sciences, Inc	Palisades Research
Address:	1545 Rt 22 East	154 Southside Ave
City, State ZIP	Annandale, NJ 08801	Bridgewater, NJ 08807
Work Phone:	(908) 730-1063	(908) 722-3604
Fax:	(908) 730-1192	(866) 350-6344
Email:	susan.p.marcella@exxonmobil.com	gail.jorgensen@verizon.net

* *