

USING PROC SQL EFFECTIVELY WITH SAS DATA SETS

JIM DEFOOR
LOCKHEED FORT WORTH COMPANY

INTRODUCTION

This paper is a beginning tutorial on reading and reporting Indexed SAS Data Sets with PROC SQL. Its examples deal with the simple Joins of two SAS Data Sets using variables that are the primary Indexes of the Data Sets. The following topics are covered:

- Reasons for using the SQL Procedure
- Syntax of PROC SQL
- Efficiently Merging Data Sets
- Reporting the Results of PROC SQL
- Creating an Index for a SAS Data Set
- Testing SQL code
- Using an Index with the SAS Data Step
- When to Use an Index
- How PROC SQL Uses Indexes to Join Data Sets

Several terms will be used interchangeably in this paper.

SAS Data Set	<-->	Table
Variables	<-->	Columns
Observations	<-->	Rows

Tables, Columns, and Rows are SQL terms which have a unique meaning when the Table is not within a SAS Library. When a Table is part of a SAS Library, it is equivalent to a SAS Data Set. Thus, its Columns and Rows are the same as a Data Set's Variables and Observations.

Also, the Data Step term 'Match-Merge' will be used synonymously with the SQL term 'Join'. Both refer to merging the observations of two Data Sets based upon the values of one or more variables.

This paper should provide the basic information a SAS Developer needs to create and use Indexed SAS Data Sets with PROC SQL. It should also explain the advantages of using PROC SQL to perform simple inquiries against Indexed SAS Data Sets.

REASONS FOR USING THE SQL PROCEDURE

PROC SQL performs many of the same functions as a SAS program which match-merges two SAS Data Sets and then reports the results with a PROC PRINT.

```
DATA WORK.MIDWEST(KEEP=CITIES POPULATN MILEAGE MURDERS);
  MERGE WORK.LIST      (IN=IN_LIST)
        STATISTC.NATION(IN=IN_NATN);
  BY CITIES;
  IF IN_LIST AND
     IN_NATN THEN
    OUTPUT;
RUN;

PROC PRINT DATA = MIDWEST;
  TITLE1 'POPULATION, MILEAGE, AND MURDERS IN MIDWESTERN CITIES';
  ID CITIES;
  VAR POPULATN MILEAGE MURDERS;
RUN;
```

These are the functions performed by this Data Step and PROC PRINT:

- Kept only certain variables.
- Match-Merged two Data Sets based upon a particular variable.
- Selected particular observations.
- Reported the results of the processing.

PROC SQL would have performed the same functions with less code:

```
PROC SQL; /* generated report automatically */
  TITLE1 'POPULATION, MILEAGE, AND MURDERS IN MIDWESTERN CITIES';
  SELECT NATION.CITIES, NATION.POPULATN, NATION.MILEAGE,
        NATION.MURDERS /* kept these variables */
  FROM WORK.LIST,
        STATISTC.NATION /* joined these data sets */
  WHERE LIST.CITIES=NATION.CITIES; /* selected matching values */
```

The smaller amount of coding is not the reason for using PROC SQL, however. PROC SQL should be used because it can be far more efficient than the SAS Data Step for match-merging observations from two SAS Data Sets - when at least the larger SAS Data Set is indexed.

The Institute added indexing to the SAS Data Libraries on MVS with Version 6.06. It did not, however, modify the SAS Data Step Match-Merge process to use indexes when evaluating matching values of the BY variables. All observations of both Data Sets are still brought into the Data Step's Program Data Vectors before the IN= variables are evaluated.

```
DATA WORK.MIDWEST(KEEP=CITIES POPULATN MILEAGE MURDERS);
  MERGE WORK.LIST      (IN=IN_LIST)
        STATISTC.NATION(IN=IN_NATN);
  BY CITIES;
  IF IN_LIST AND      /* no use of indexes in evaluation */
     IN_NATN THEN
    OUTPUT;
RUN;
```

SYNTAX OF PROC SQL

PROC SQL has a substantively different syntax than most SAS Procedures:

```
PROC SQL;  
  CREATE TABLE WORK.MIDWEST AS  
    SELECT NATION.CITIES, NATION.POPULATN, NATION.MILEAGE,  
          NATION.MURDERS  
    FROM WORK.LIST,  
          STATISTC.NATION  
    WHERE LIST.CITIES=NATION.CITIES;
```

Here are the major differences:

The RUN Statement is never needed.

The output of the procedure is reported automatically if a Data Set is not created.

Variables must usually be referenced in conjunction with the name of the SAS Data Set to which they belong.

Semicolons mark only major statement boundaries. Blanks mark minor statement boundaries. Commas separate variable names and Data Set names.

Input Data Sets are specified as part of the FROM Statement, not as part of the PROCEDURE Statement.

The order of the Data Sets in the FROM Statement is very important in determining the processing of the data by PROC SQL.

PROC SQL has this form when the output of the procedure is reported:

```
PROC SQL;  
  TITLE1 'POPULATION, MILEAGE, AND MURDERS IN MIDWESTERN CITIES';  
  SELECT NATION.CITIES, NATION.POPULATN, NATION.MILEAGE,  
        NATION.MURDERS  
    FROM WORK.LIST,  
        STATISTC.NATION  
    WHERE LIST.CITIES=NATION.CITIES;
```

PROC SQL has this form when the output of the procedure is stored in a Data Set.

```
PROC SQL;
  CREATE TABLE WORK.MIDWEST AS /* output data set */
    SELECT NATION.CITIES, NATION.POPULATN, NATION.MILEAGE,
          NATION.MURDERS
    FROM WORK.LIST,
          STATISTC.NATION
    WHERE LIST.CITIES=NATION.CITIES;
```

Notice that the CREATE statement has replaced the TITLE Statement, but has no semicolon ending it.

PROC SQL has this form when restrictions are added to the WHERE Statement so that the output of the Join is limited to only those observations with a certain value:

```
PROC SQL;
  CREATE WORK.MIDWEST AS
    SELECT NATION.CITIES, NATION.POPULATN, NATION.MILEAGE,
          NATION.MURDERS
    FROM WORK.LIST,
          STATISTC.NATION
    WHERE LIST.CITIES=NATION.CITIES AND
          NATION.POPULATN GE 100000 AND
          LIST.LOCATION = 'MIDWEST'; /* output restrictions */
```

Attaching these restrictions to the WHERE Statement causes PROC SQL to evaluate the matching observations from the two Data Sets and output only those which represent Midwestern Cities with a population over 100,000. It produces the same result as this SAS Data Step:

```
DATA WORK.MIDWEST(KEEP=CITIES POPULATN MILEAGE MURDERS);
  MERGE WORK.LIST      (IN=IN_LIST)
        STATISTC.NATION(IN=IN_NATN);
  BY CITIES;
  IF IN_LIST AND
     IN_NATN THEN
  DO;
    IF POPULATN GE 100000 AND
       LOCATION EQ 'MIDWEST' THEN
      OUTPUT;
  END;
RUN;
```

Another way to limit output is to use the WHERE= Data Set Option to restrict the observations of each Data Set to certain values before the Data Sets are joined:

```
PROC SQL;
  CREATE TABLE WORK.MIDWEST AS
    SELECT NATION.CITIES, NATION.POPULATN, NATION.MILEAGE,
          NATION.MURDERS
          /* input restrictions */
    FROM WORK.LIST(WHERE=(LOCATION = 'MIDWEST')),
          STATISTC.NATION(WHERE=(POPULATN GE 100000))
    WHERE LIST.CITIES=NATION.CITIES;
```

Using a WHERE= Data Set Option limits the data passed to the Join. In this case, it restricts the cities from WORK.LIST to only those with a MIDWEST location. It also limits the cities from STATISTC.NATION to only those with a population equal-to or greater-than 100,000. This Data Step performs the same functions:

```
DATA WORK.MIDWEST(KEEP=CITIES POPULATN MILEAGE MURDERS);
  MERGE WORK.LIST      (IN=IN_LIST WHERE=(LOCATION='MIDWEST'))
    STATISTC.NATION(IN=IN_NATN WHERE=(POPULATN GE 100000));
  BY CITIES;
  IF IN_LIST AND
    IN_NATN THEN
    OUTPUT;
RUN;
```

When the WHERE= Data Set Option is used - in either PROC SQL or the Data Step - it is performed against the read buffer of that Data Set. Only those observations which pass the WHERE= Option are passed to PROC SQL (or the Data Step).

In PROC SQL, the WHERE Statement restrictions are performed after the Join of the two Data Sets. The WHERE= Data Set Options are performed before the Join. When they can be used, the WHERE= Options usually save processing time.

EFFICIENTLY MERGING SAS DATA SETS

As stated in the introduction, PROC SQL's capability of merging two SAS Data Sets more efficiently than the SAS Data Step is the primary reason for using PROC SQL instead of the SAS Data Step. This advantage does not exist if neither of the two Data Sets are indexed by the variables listed in the WHERE Statement or if the Data Sets are not correctly ordered in the FROM Statement.

```
FROM WORK.LIST,
  STATISTC.NATION
  WHERE LIST.CITIES=NATION.CITIES
```

Here are several rules for efficiently coding PROC SQL:

1. Specify the Indexed Data Set last in the FROM Statement if only one Data Set is Indexed. Otherwise, specify the larger Data Set last in the FROM Statement.

```
FROM WORK.LIST,
  STATISTC.NATION /* larger and indexed data set */
  WHERE LIST.CITIES=NATION.CITIES
```

2. Add WHERE= options to the Data Sets in the FROM Statement whenever relevant, especially if they involve an Indexed variable on the larger Data Set or if they substantively reduce the size of the smaller Data Set.

```
FROM WORK.LIST(WHERE=(LOCATION='MIDWEST')), /* not an index */
      STATISTC.NATION(WHERE=(POPULATN GE 1000000)) /* index */
WHERE LIST.CITIES=NATION.CITIES;
```

3. When WHERE= options are used on the Data Sets in the FROM Statement, and both involve Indexed variables, determine which Data Set will be smaller after the WHERE= options are performed. Place the smaller first in the FROM Statement even if it is larger before the WHERE= option is performed. This is a modification to Rule 1.

```
FROM STATISTC.NATION(WHERE=(POPULATN GE 1000000)),
      WORK.LIST
WHERE NATION.CITIES=LIST.CITIES;
```

4. When the WHERE= options involve variables which are not indexed, performance of the SQL procedure should be no worse than having those restrictions as part of the WHERE statement.

```
FROM WORK.LIST,
      STATISTC.NATION
WHERE LIST.CITIES=NATION.CITIES AND
      NATION.POPULATN GE 100000 AND
      LIST.LOCATION = 'MIDWEST';
```

Essentially, these rules boil down to:

1. Use the WHERE= Data Set Options with Indexed variables whenever possible to reduce the size of both Data Sets before they are joined.
2. Use a WHERE= option with a Non-Indexed variable to reduce the size of the smaller Data Set if an Indexed variable can't be used.
3. Specify the larger Data Set - after any WHERE= processing - last in the FROM Statement.
4. Don't worry about replacing the WHERE Statement with the WHERE= Data Set Option when the WHERE= will work.

These rules are neither perfect nor comprehensive. There are always exceptions, but these will usually hold true for simple Joins involving only two Data Sets and at least one primary Index, which should be the situation for most SQL processing. Rules for complex Joins can be found in the REFERENCES.

REPORTING THE RESULTS OF PROC SQL

```
PROC SQL;
  TITLE1 'POPULATION, MILEAGE, AND MURDERS IN MIDWESTERN CITIES';
  SELECT NATION.CITIES,
         NATION.POPULATN,
         NATION.MILEAGE,
         NATION.MURDERS
  FROM WORK.LIST,
       STATISTC.NATION
  WHERE LIST.CITIES=NATION.CITIES;
```

If a Data Set is not created, PROC SQL will automatically create a listing of the output of the procedure. The order of the variables in the report will be the same as specified in the SELECT Statement.

POPULATION, MILEAGE, AND MURDERS IN MIDWESTERN CITIES

CITIES	POPULATN	MILEAGE	MURDERS
CHICAGO	5000000	25000	900
DALLAS	1000000	20000	500
FT. WORTH	700000	10000	250
HOUSTON	1500000	30000	600

Improve the appearance of the listing with a few simple modifications to the SELECT Statement. Just add Label and Format modifiers to each variable.

```
PROC SQL;
  TITLE1 'POPULATION, MILEAGE, AND MURDERS IN MIDWESTERN CITIES';
  SELECT NATION.CITIES LABEL='CITY' FORMAT=$15.0,
         NATION.POPULATN LABEL='POPULATION' FORMAT=COMMA9.0,
         NATION.MILEAGE LABEL='HIGHWAY MILEAGE' FORMAT=COMMA9.0,
         NATION.MURDERS LABEL='MURDERS PER YEAR' FORMAT=COMMA9.0
  FROM WORK.LIST,
       STATISTC.NATION
  WHERE LIST.CITIES=NATION.CITIES AND
        NATION.POPULATN GE 100000 AND
        LIST.LOCATION = 'MIDWEST';
```

PROC SQL will automatically split any column labels that are longer than the length of the variable.

POPULATION, MILEAGE, AND MURDERS IN MIDWESTERN CITIES

CITY	POPULATION	HIGHWAY MILEAGE	MURDERS PER YEAR
CHICAGO	5,000,000	25,000	900
DALLAS	1,000,000	20,000	500
FT. WORTH	700,000	10,000	250
HOUSTON	1,500,000	30,000	600

Of course, if a high-degree of customization is required, the results of the PROC SQL can be stored in a Data Set and passed to a Data _Null_ Step or a PROC REPORT.

CREATING AN INDEX FOR A SAS DATA SET

An Index is another SAS Data Set. Instead of containing information such as the names of cities, their population, and their highway mileage, it contains references to the location of that information for a particular city on the Indexed Data Set. For example, an Index might indicate that the information for DALLAS was on line 200 of the Indexed Data Set.

Indexes should be created if their use will reduce the processing needed to retrieve the observations from the Indexed Data Set. Generally, if only a few observations must be retrieved from Data Set to satisfy most inquiries, that Data Set is a good candidate for an Index. The best candidate for an Index is a Data Set with many variables, but with only two or three that are usually needed for an inquiry. Good examples are Data Sets that have a lot of information on discrete elements, such as people, cities, businesses, households, products available for sale, or parts in inventory.

Creating an Index is easy. PROC DATASETS provides one method:

```
PROC DATASETS LIBRARY=STATISTIC MEMTYPE=DATA;
  MODIFY NATION          /* name of data set */
    INDEX CREATE CITIES /* names of variables to be indexed */
      POPULATN;
RUN;
```

This PROC DATASETS creates two Indexes: one for CITIES and one for POPULATN. These messages will be displayed in the SAS LOG:

```
NOTE: Single Index CITIES defined.
NOTE: Single Index POPULATN defined.
```

PROC CONTENTS will report the names and content of the Indexes which exist for a SAS Data Set, as will PROC DATASETS with a CONTENTS Statement.

-----Alphabetic List of Indexes and Attributes -----

#	Index	Unique Option	Var1	Var2
1	CITIES	YES		
2	POPULATN			

Indexes with only one entry per value can have the Unique option marked with a YES. Simple indexes will not have any entries in the VAR1 and VAR2 fields because they have only one variable involved in the index. If entries are listed in the VAR1 and VAR2 fields, the index is a composite of those two variables, such as would be produced if CITIES and STATES were joined together with the name of C_STATE.

#	Index	Unique Option	Var1	Var2
1	C_STATE	YES	CITIES	STATES

To create a unique composite index, the PROC DATASETS must be coded as follows:

```
PROC DATASETS LIBRARY=STATISTC MEMTYPE=DATA;
  MODIFY NATION
    INDEX CREATE C_STATE=(CITIES STATES) / UNIQUE;
RUN;
```

To use a unique composite index in PROC SQL, just refer to both variables in the WHERE Statement.

```
FROM WORK.LIST,
  STATISTC.NATION
WHERE LIST.CITIES=NATION.CITIES AND
      LIST.STATES=NATION.STATES;
```

TESTING SQL CODE

Testing PROC SQL is only slightly different than testing other SAS code. The most important difference deals with the use of the OBS= and FIRSTOBS= Options. When the SQL procedure is called, OBS must be set to MAX and FIRSTOBS must be set to 1.

```
OPTIONS OBS=MAX FIRSTOBS=1;
```

If OBS= is less than MAX or FIRSTOBS= is greater than 1, PROC SQL will fail with this error message:

```
ERROR: A where clause may not be used with the FIRSTOBS or the OBS option.
ERROR: A where clause may not be used with the FIRSTOBS or the OBS option.
ERROR: A where clause may not be used with the FIRSTOBS or the OBS option.
```

NOTE: The SAS System stopped processing this step because of errors.

To avoid this error and still keep observations to a minimum during testing, start the SAS program with an OBS= option, such as

```
OPTIONS OBS=200;
```

Then, just before the SQL procedure is called, reset the OBS= option to MAX;

```
OPTIONS OBS=MAX;
```

Another difference, but a minor one, deals with the content of the NOTES printed at the end of a successful PROC SQL.

NOTE: Table WORK.FOUND created, with 15 rows and 3 columns.

This NOTE, as indicated in the INTRODUCTION, can be completely translated to the familiar:

NOTE: The data set WORK.FOUND has 15 observations and 3 variables.

If no matches are found, this NOTE will be printed:

NOTE: No rows were selected.

USING AN INDEX WITH THE SAS DATA STEP

SAS partially alleviated the match-merge shortcomings of the Data Step by adding the KEY= option to the SET Statement with 6.07.

```
DATA WORK.MIDWEST(KEEP=CITIES POPULATN MILEAGE MURDERS);  
  SET WORK.LIST;  
  SET STATISTC.NATION KEY=CITY;  
  IF _IORC_ = 0 THEN  
    OUTPUT;  
RUN;
```

With the KEY= option, the SAS Data Step uses the value of the KEY variable received from the first Data Set to evaluate the Index of the KEY variable in the second Data Set. If a match is found, the Data Step retrieves the matching observation from the second Data Set and sets the automatic variable _IORC_ to zero. The SAS developer can then limit the observations to those which match the Indexed Data Set by checking the value of _IORC_ before outputting an observation.

The KEY= approach has a major limitation, however. It retrieves only the first occurrence of the value of the Indexed variable from the Indexed Data Set. Other matches to that value are not retrieved.

PROC SQL does not have this limitation. It retrieves only the observations which match the Indexed variable, and does so without any _IORC_ checking.

WHEN TO USE AN INDEX

Use PROC SQL or the KEY= Option with an Index to retrieve data from an Indexed SAS Data Set when 25% or less of a Data Set will be selected. Don't use PROC SQL or the KEY= Option index when 50% or more of the Data Set would be retrieved. In between 25% and 50%, experiment. See the Armstrong and Beatrous article for details.

Using an Index always incurs the cost of reading the Index, but saves the cost of not reading all of the observations of the Indexed Data Set. Not using an Index avoids the cost of reading the Index but requires the reading of all observations of the Indexed Data Set.

When only a small number of observations are retrieved from the Indexed Data Set, the cost of reading the Index is small compared to the savings of not reading the entire Indexed Data Set. As the number of observations increases, however, eventually the cost of reading the Index exceeds the savings of not reading the entire Indexed data set.

If all or most of the records of a Data Set are to be processed, sort both Data Sets by the variables to be matched and use the SAS Data Step Match-Merge instead of PROC SQL or the KEY= Option to retrieve the desired data.

HOW PROC SQL USES INDEXES TO JOIN DATA SETS

When both of the Data Sets in a Join are Indexed, PROC SQL matches the Indexes and then retrieves only the observations from each Data Set that are tied to the matching Indexes. For example, suppose WORK.LIST has 100 observations and NATION.STATISTIC has 1000 observations and that each observation in WORK.LIST matches one row in NATION.STATISTIC. PROC SQL would create a Joined Index of both tables and then retrieve the 100 matching observations from each Data Set. It would not retrieve the 900 observations in NATION.STATISTIC which did not match WORK.LIST. The Data Step Match-Merge would retrieve all 1000 observations of NATION.STATISTIC.

The larger Indexed Data Set should be listed last in the FROM Statement to reduce the expense of matching the Indexes of both Data Sets. Generally, less reads are required to match few-to-many than many-to-few.

If only one of the Data Sets are Indexed, PROC SQL will still use the Index of that Data Set in the Join process but only if it is listed last in the FROM Statement. Otherwise, it will sort both Data Sets by the variables listed in the WHERE Statement and then match-merge the two Data Sets.

When used with an Indexed variable, the WHERE= Option will, under some circumstances, speed the Join process by reducing the size of the Joined Index and, thus, the number of retrievals necessary to match the observations of each Data Set with the Joined Index. When this occurs, the WHERE= Option is very efficient because it avoids even the cost of checking the read buffer of that Data Set to see whether the observation matches the requirements of the WHERE= option.

CONCLUSION

This paper is meant to provide enough information regarding PROC SQL to convince the reader that it should be added to their toolbox and to enable the reader to build simple SQL procedures. Many of the features of PROC SQL have been ignored. For example, SQL can retrieve data from only one Data Set. Data Sets do not have to be joined by SQL. Also, SQL can perform calculations and then use those calculations in the selection of observations from a Data Set. Finally, SQL can sort the Joined data before reporting it or storing it in a Data Set. These and other features of SQL are described in the articles and publications noted in the REFERENCES.

REFERENCES

Kent, Paul, 'A Talk on SQL', Proceedings of the South Central SAS Users Regional Conference '92, Cary, NC, SAS Institute Inc., pp. 156 - 167.

Kent, Paul, 'Fun with the SQL Procedure - an Advanced Tutorial', Proceedings of the Seventeenth Annual SAS Users Group International Conference, Cary, NC, SAS Institute Inc., 1992, pp. 167 - 175.

Stanley. Don, 'Making Use of the SAS Implementation of SQL', Proceedings of the Seventeenth Annual SAS Users Group International Conference, Cary, NC, SAS Institute Inc., 1992, pp. 303 - 308.

Armstrong, Karen and Steve Beatrous, 'Effective Uses of Indexes in the SAS System, Proceedings of the Sixteenth Annual SAS Users Group International Conference, Cary, NC, SAS Institute Inc., 1991, pp. 605 - 614.

SAS Language: Reference, Version 6, First Edition, Cary NC: SAS Institute Inc., 1990. 1042 pp.

SAS Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition, Cary, NC, SAS Institute Inc., 1989, 210 pp.