Paper 131-31

Using Data Set Options in PROC SQL

Kenneth W. Borowiak
Howard M. Proskin & Associates, Inc., Rochester, NY

ABSTRACT

Data set options are an oft over-looked feature when querying and manipulating SAS® data sets with PROC SQL. This paper explores using the data set options DROP, KEEP, LABEL, COMPRESS, SORTEDBY, WHERE, and RENAME in the CREATE TABLE statement and FROM clause of PROC SQL. These options can help facilitate more succinct and efficient code and create parsimonious and well-labeled data sets.

INTRODUCTION

Data set options can be tautologically described as options called upon to control the use of a data set and its variables and indices. Data set options such as KEEP, RENAME and LABEL are found within parentheses directly following a data set reference and pertain to *only* that data set[1]. Data set options differ from *system options* (which influence an entire SAS session) and *statement options* (which provide instructions in a PROC or DATA step).

Figure 1 - Examples of SAS Options

```
option compress=yes;

proc summary data=Sashelp.Prdsal2 nway;
  class country year quarter;
  var actual predict;
  output out=Save.sales_means(label='Means of Actual & Predicted Sales')
         mean= / autoname ;
  run;
```

System option

Statement option

Data set option

Data set options are an invaluable tool for manipulating, modifying, and adding descriptive properties to data sets. However, applications of data set options are seldom found in the context of PROC SQL. The purpose of this paper is to expose the usefulness of data set options in PROC SQL to help facilitate more succinct and efficient code and to create parsimonious and well-labeled data sets. The specific points covered should be of interest to beginning and intermediate users of PROC SQL.

PRELIMINARIES

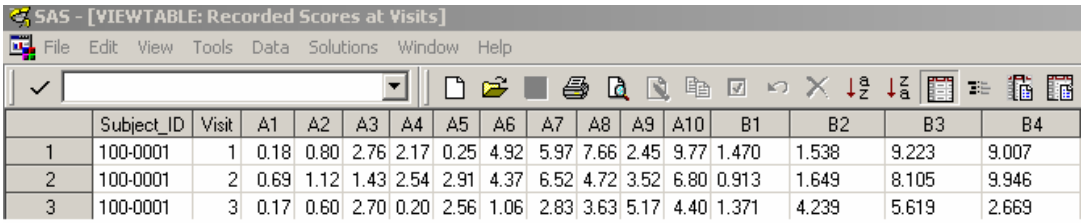
Uses of various data set options in PROC SQL will be presented through a series of examples using three data sets from a fictitious clinical trial. The code that generated the TX, Scores, and Surgery data sets can be found in Appendix 1. The Tx data set contains information on the subject (numeric field with values between 1 and 3000), study center, randomization date, and code of received treatment (numeric field with values of 1 or 2). The Scores data set contains information on the subject (character field concatenation of the subject number and study center), visit, and scores for parameters A1-A10 (numeric) and B1-B10 (character). The Surgery data set contains information on the subject, the visit when a surgical procedure was performed, and a case record number.

DROP and KEEP

PROC SQL queries require the SELECT statement to specify the variables to be included in the output destination. You have the option to explicitly state which variables to keep or use the * wildcard as a short-cut to select all fields from some or all tables involved in the query. Have you ever encountered the problem of having to choose between typing in a long list of variables to keep or use the * to select more variables than are actually needed? The dilemma can be alleviated by using the KEEP and DROP data set options in the FROM clause. These data set options allow you to specify which of the variables should be processed and those that should not, respectively. Suppose that all variables *except* A1 and A3 are needed from the Scores the data set. The query could be written succinctly by using the * wildcard and the DROP data set option to remove the fields as the table is introduced into the query, as demonstrated in Figure 2 on the next page.

[1] See the SAS On-line documentation for a comprehensive list of data set options.

Figure 2 – DROP Data Set Option in the FROM Clause
The Case of ‘ select * except’



	Subject_ID	Visit	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	B1	B2	B3	B4
1	100-0001	1	0.18	0.80	2.76	2.17	0.25	4.92	5.97	7.66	2.45	9.77	1.470	1.538	9.223	9.007
2	100-0001	2	0.69	1.12	1.43	2.54	2.91	4.37	6.52	4.72	3.52	6.80	0.913	1.649	8.105	9.946
3	100-0001	3	0.17	0.60	2.70	0.20	2.56	1.06	2.83	3.63	5.17	4.40	1.371	4.239	5.619	2.669

```
%let label=Scores Data Set without A1 and A3;

proc sql;
  create table Scores1(label="&label") as
  select  *
  from    Scores(drop=A1 A3);
  quit;
```

Partial display of the SCORES data set

DROP data set option

One of the useful aspects of the implementation of SQL by SAS is the availability of the short-cut notations in conjunction with data set options. Suppose that all the fields from the Scores data set were required *except* for the B parameters. The query could be written succinctly making use of any of the four short-cut notations shown below in Figure 3.

Figure 3 – Variable Lists with the DROP and KEEP Data Set Options in the FROM Clause

```
proc sql;
 create table only_As_1(label='Scores for A Parameters Only') as
 select  *
 from    Scores(keep=Subject_id Visit A1-A10);

  /* or */

 create table only_As_2(label='Scores for A Parameters Only') as
 select  *
 from    Scores(drop=B1--B10);

  /* or */

 create table only_As_3(label='Scores for A Parameters Only') as
 select  *
 from    Scores(drop= B:);

  /* or */

 create table only_As_4(label='Scores for A Parameters Only') as
 select  *
 from    Scores(keep=Subject_ID _numeric_);

 quit;
```

Note the use of the numbered range list

Note the use of the name range list

Note the use of the name prefix list

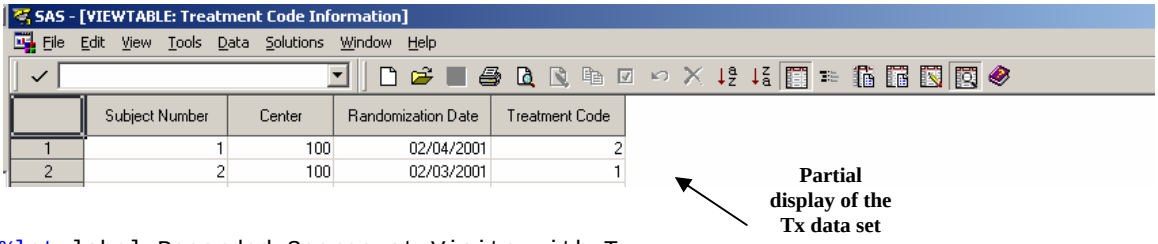
Note the use of the variable class list

Note the use of the *numbered range list* (-) with the KEEP data set option reference in the first query. This short-cut refers to a list of variables with a common prefix with an indexed number suffix. Specifying A1-A10 in the SELECT statement would have created a new variable that is the result of arithmetic subtraction of the two fields. The second query makes use of the *name range list* (--) with the DROP data set option to refer to all variables in the data set located between B1 and B10. Note the use of the *name prefix list*

courtesy of the colon in the third query to drop all variables beginning with the letter B. Note the use of the _numeric_ *variable class list* keyword to reference all of the numeric fields in the fourth query. It is left to you to infer as to how the query could have been written using the _character_ keyword. All of the aforementioned coding short-cuts are not valid in the SELECT statement but they are available when using the DROP and KEEP data set options in the FROM clause.

The KEEP and DROP data set options are also available in the CREATE TABLE statement of PROC SQL. Suppose that the treatment code and randomization date from the Tx data set need to be joined with the information from the Scores data set. It would be redundant to keep the `subject_no` and `center` fields from the Tx data set because the information is captured in the `subject_id` field from the Scores data set. One way the query could be written is displayed below in Figure 4.

Figure 4 – DROP Data Set Option in the CEATE TABLE statement




```
%let label=Recorded Scores at Visits with Tx;
```

Can't drop SUBJECT_NO and CENTER in FROM clause because they are needed to join the tables

These fields are not needed in the output data set because the information is contained in the SUBJECT_ID field

```
proc sql;
 create table scores_tx(label="&label" drop=subject_no center) as
 select  *
 from    Tx T1,
         Scores T2
 where   T1.subject_no=input(substr(T2.subject_id,5),8.) and
        T1.center=input(substr(T2.subject_id,1,3),8.);
 quit;
```

As was the case with the queries in Figure 2, the unwanted variables are specified with the DROP data set option in conjunction with the * wildcard rather than explicitly stating the desired ones. However, the unwanted variables omitted from the resulting data set are needed to join the tables. The WHERE clause is evaluated *after* the FROM clause but *before* the CREATE TABLE statement. Therefore, dropping the variables via a data set option needs to be delayed until the data is written out to the resulting data set.

The three examples in this section have demonstrated how the DROP and KEEP data set options can be used to conveniently and succinctly include fields in a resulting table and reduce the carrying around of unneeded fields.

LABEL

All of the queries in Figures 2 through 4 employ the LABEL data set option to provide a descriptive title of the data set. Specifying a data set label is a good programming practice, especially for permanent SAS data sets that can be accessed by others. The LABEL data set option should follow the data set reference in the CREATE TABLE or CREATE VIEW statements in PROC SQL. This option has no effect on existing data sets referenced in the FROM clause.

COMPRESS

Compression is a method to reduce the size of storing data sets. Data sets that have many character fields with an overabundance of unneeded bytes are good candidates to be compressed, where the size reduction can be substantial. This comes at the expense of increased CPU needed to uncompress the data sets before operating on them. For a more detailed exploration of data set compression, the reader should consult the paper by Karp and Shamlin [2001]. Two ways data sets can be compressed using the SAS system are the COMPRESS system and data set options. When the former is enabled, all datasets created afterwards will be compressed. Using the COMPRESS data set option overrides the system option and applies to only that data set. To compress a data set using PROC SQL, the COMPRESS data set option should be stated after the table reference in the CREATE TABLE statement, as demonstrated in Figure 5 on the next page.

Figure 5 – COMPRESS Data Set Option in the CREATE TABLE Statement

```
proc sql;
 create table only_Bs(compress=yes) as
 select  *
 from    Scores(drop=A1-A10);
 quit;

NOTE: Compressing data set WORK.ONLY_BS decreased size by 57.09 percent.
      Compressed is 460 pages; un-compressed would require 1072 pages.
NOTE: Table WORK.ONLY_BS created, with 60000 rows and 12 columns.
```

COMPRESS data set option overrides the system option

In many circumstances, the COMPRESS data set option allows for more judicious and pertinent use of compression than its system option counterpart does. Case in point, the code in Figure 1 would have resulted in a data set that is 50% larger compressed than uncompressed.

RENAME

The RENAME data set option allows you to change the name of variables in a data set. Consider the queries below in Figure 6, where the scores for the A parameters captured at the baseline visit (i.e. Visit 1) are renamed with the prefix 'Base_'. You could explicitly create the new variables in the SELECT statement with a column alias as in the first query, but at the expense of having to type a long list of fields. Alternatively, making use of the numbered range list in the FROM clause with the RENAME data set option would be particularly helpful. In the second query, the fields A1 through A10 are renamed to an array of fields with a new common prefix and indexed suffix (e.g. Base_A1 through Base_A10).

Figure 6 – RENAME Data Set Option in the FROM Clause

```
proc sql;
 create table Baseline_A_1 as
 select  subject_id,
         a1 as base_a1,
         /* .. etc. .. , */
         a10 as base_a10
 from    Scores
 where   visit=1;

 create table baseline_A_2 as
 select  *
 from    Scores(rename=(a1-a10=base_a1-base_a10) drop=b:)
 where   visit=1;

 quit;
```

Rather than explicitly creating the new variables in the SELECT statement ……

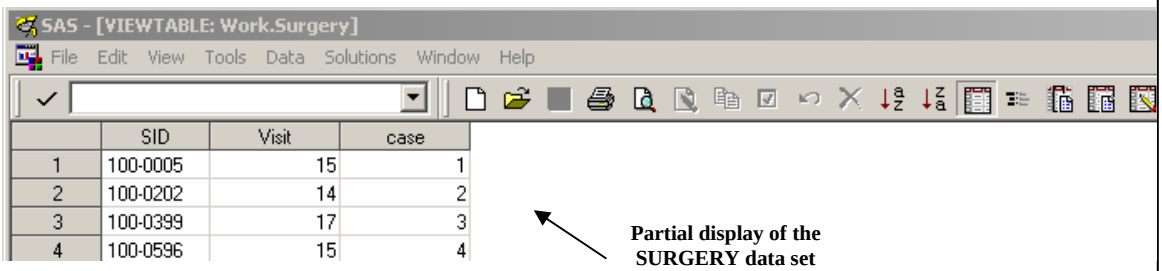
Do the fields that start with 'B' ever make it to the output destination?

… you can create them succinctly with the numbered range list

You may be wondering, "If you use the DROP data set option to remove all the fields that start with ''b', do the newly renamed 'base_a' fields make it to the BASELINE_A_2 data set?". Well, the answer is 'Yes' because even though the DROP data set option *appears after* the RENAME data set option, the DROP and KEEP data set options are *executed before* the RENAME data set option.

The previous example used the RENAME data set option in the FROM clause for brevity in the SELECT statement. The RENAME data set option can also be helpful when joining tables. Explicit reference to the join conditions is made in the WHERE or ON clauses of PROC SQL when joining tables. However, *natural joins* is a coding short-cut where table joins are implicit. These kinds of joins use fields with a common name and field type to perform an equijoin between two tables. Consider the example in Figure 7 on the next page where an inner join between the SCORES and SURGERY data sets is sought. The numeric field VISIT is common to both data sets and would be included as part of a natural join between the two tables. Ideally, the other key fields are SID and SUBJECT_ID from the SURGERY and SUBJECT_ID data sets, respectively. However, they would be excluded from the join of the tables since they do not share the same name. Renaming either subject-identifying field to match that of the other data set would enable you to make use of the natural join. Since the statement option FEEDBACK is specified, a rewritten version of the query will be displayed in the SAS log where it explicitly shows the conditions of any natural joins, resolution of any macro variables and other implicit statements.

Figure 7 – RENAME Data Set Option in the FROM Statement
The Case of the Natural Join



	SID	Visit	case
1	100-0005	15	1
2	100-0202	14	2
3	100-0399	17	3
4	100-0596	15	4

Partial display of the SURGERY data set

```
proc sql feedback _method;
 create table Scores_Surg1 as
 select  *
 from    Scores T1, Surgery T2
 where   T1.subject_id=T2.sid and
         T1.visit=T2.visit;

 create table Scores_Surg2 as
 select  *
 from    Scores T1
          natural inner join
          Surgery(rename=(sid=subject_id)) T2;
 quit;
```

Standard inner join with join conditions in the WHERE clasue

Inner join sans join conditions courtesy of the natural join and RENAME data set option

```
NOTE: Statement transforms to:
   select COALESCE(T2.subject_id, T1.Subject_ID) as subject_id,
      COALESCE(T2.Visit, T1.Visit) as Visit, T2.case, T1.A1, T1.A2, T1.A3, T1.A4,
      T1.A5, T1.A6, T1.A7, T1.A8, T1.A9, T1.A10, T1.B1, T1.B2, T1.B3, T1.B4,
      T1.B5,T1.B6, T1.B7, T1.B8, T1.B9, T1.B10
    from WORK.SCORES T1 inner join WORK.SURGERY T2(rename=(sid=subject_id))
      on (T2.subject_id=T1.Subject_ID) and
          (T2.Visit=T1.Visit);
```

With the FEEDBACK option turned on, the following message is generated in the SAS log for the second query

Lund [2005] points out another example of when using the RENAME data set option is useful. Consider the query below where the Surgery data set is filtered on the field `case`. Since CASE is a statement keyword that initiates the syntax for a created field , the first query in Figure 8 will result in error due to referencing the variable with the same name in the WHERE clause. The second query uses the RENAME data set option to change the name of `case` to `case_id` as the table is brought into the query. Now referencing the field by its new name is no longer problematic. Other names of fields that can cause similar problems using SAS V8 and V9 are CALCULATED, EXISTS, and TRANSLATE, as well as SUBSTRING in V8.

Figure 8 - RENAME Data Set Option in the FROM Clause

```
proc sql _method;
 create table caseLT4_1 as
 select *
 from   Surgery
 where  case lt 4;


 create table caseLT4_2 as
 select *
 from   Surgery(rename=(case=case_id))
 where  case_id lt 4;
 quit;
```

This query is invalid because CASE initiates the syntax for a created field

The field `case` is renamed so it can be referenced in the WHERE clause

SORTEDBY

One of the appealing features of PROC SQL is that it does not require you to pre-sort the data sets before joining them, especially if there are many data sets involved or some of them are large. So what techniques does SQL use to join tables? There are three types of techniques that SQL considers when it needs to perform an inner join and match keys from the different tables exactly (aka *equijoin*). If the key variable(s) in at least one data set is indexed then an *index join* is considered first. If the indexed key variable(s) does not succeed in returning a discriminating amount of results (~ <20%) then a *merge join* is considered. A merge join is similar to a data step merge and may be employed at this point if the SQL optimizer recognizes that the larger of the two data sets being joined is sorted. If the previous condition fails then a *hash join* is considered. The concept of hashing can be found in recent papers by Dorfman *et al.* [2001, 20032. Simplistically put, hashing involves loading the smaller of two tables into memory and uses a fast hybrid direct address-searching algorithm on the keys. A hash join is employed at this stage if there is sufficient amount of memory to accept the smaller table. If a hash join cannot be performed, the data sets will be sorted (even if you did not ask them to be) and a merge join is employed.

A key point is that when PROC SQL develops an efficient strategy to join tables it wants to avoid doing any sorting. The SQL optimizer consults the metadata to determine whether a data set is sorted, and if so, by which fields. However, data sets are sometimes 'in order' without being sorted. For example, the resulting data sets from a PROC SUMMARY statement and DATA step merge (sometimes) are 'in order' by the CLASS and BY variables, respectively, but this information is not captured by the metadata. If these resulting data sets become part of a subsequent multi-table PROC SQL query, they may be unnecessarily sorted if they are too large to be part of a hash join. Superfluous sorting can be avoided by using the SORTEDBY data set option in the FROM clause in PROC SQL. The SORTEDBY data set option provides information regarding by which variables the data set is sorted.

The PROC SQL statement option *_method* should be used to ascertain the join strategy the optimizer decides upon to execute a submitted query. This undocumented option by SAS has been examined in an article by Lavery [2005]. The _method option will provide a high-level summary of the methods used to execute the query. A sample of possible codes produced by _method is provided below.

Table 1 - Message Codes from _method

Code	Description
sqxjm	Merge Join
sqxjhsh	Hash Join
sqxjndx	Index Join
sqxsort	Sort

Let us revisit the second query in Figure 7 where an inner join is used to coalesce the information from the Surgery and Scores data sets. Figure 9A below shows part of the SAS log when the _method option is enabled.

Figure 9A – SQL Execution Methods Without the SORTEDBY Data Set Option

```
NOTE: SQL execution methods chosen are:
      sqxcrta
          sqxjhsh
              sqxsrc( WORK.SCORES(alias = T1) )
              sqxsrc( WORK.SURGERY(alias = T2) )
```

The optimizer does not recognize that the tables are 'in order' by SUBJECT_ID and VISIT and decides to perform a hash join

The message in the log indicates that hashing was used to join the two tables, which by no means is a poor way to combine information from two tables. However, both of the data sets involved in the query are 'in order' by Subject_id and Visit because they were generated that way (see Appendix 1). The query could be improved by specifying the SORTEDBY data set option in the FROM clause for both data sets. Now the optimizer decides to employ a preferred join strategy (i.e. a merge join) under the condition that sorting is not needed. This is confirmed by examining the execution methods displayed in Figure 9B below.

Figure 9B – SQL Execution Methods With the SORTEDBY Data Set Option

```
…..
 from   Scores(sortedby=subject_id visit) T1
        natural inner join
        Surgery(sortedby=subject_id visit rename=(sid=subject_id)) T2;
…..
NOTE: SQL execution methods chosen are:
      sqxcrta
          sqxjm
              sqxsrc( WORK.SURGERY(alias = T2) )
              sqxsrc( WORK.SCORES(alias = T1) )
```

A hint was provided to the optimizer that the tables were 'in order', so it decided to perform a merge join

WHERE

The WHERE clause in PROC SQL is used to subset data and specify join conditions. The WHERE data set option can be applied to tables in the FROM clause to subset the table as it is brought into the query. Consider the query below in Figure 10A where the Scores data set is augmented with an indicator if a surgical procedure was performed at a visit prior to the sixteenth visit.

Figure 10A - Query Without WHERE Data Set Option in the FROM Clause

```
proc sql;
 create table scores_surglt16 as
 select T1.*,
        case when ILV2.Visit gt 0 then 'Y' end as Surgery_Flag
 from   Scores as T1
         left join
        ( select  *
          from    Surgery
          where   Visit lt 16) as ILV2
        on T1.visit=ILV2.visit and
           T1.subject_id=ILV2.sid;
quit;
```

In-line view created in order to subset SURGERY before executing the left-join

The query above creates an in-line view from the Surgery data set that excludes procedures occurring at or after visit 16, which is then left-joined to the Scores data set. As an alternative to creating an in-line view, filtering of the Surgery data set could be done with the WHERE data set option as the table was introduced into the query, as demonstrated below in Figure 10B. Though the two queries are functionally equivalent, it is left to you to determine which is easier to follow.

Figure 10B - Query With WHERE Data Set Option in the FROM Clause

```
proc sql;
 create table scores_surglt16_2 as
 select T1.*,
        case when T2.Visit gt 0 then 'Y' end as Surgery_Flag
 from   Scores T1
        natural left join
        Surgery(where=(visit lt 16)
                rename=(sid=subject_id)) T2;
quit;
```

Subsetting the Surgery data set before it is brought into the query

Another useful aspect of the WHERE data set option is the availability of the colon in conjunction with the IN comparison operator to search for varying length text strings in the beginning of a character field. PROC SQL has available in the WHERE clause string comparison operators such as EQT and LET to mimic the operators =: and <=:, respectively, that are available in the DATA and other PROC steps. However, there is no counterpart to in: available in the WHERE clause of PROC SQL. Note how multiple comparisons using the LIKE operand in the first query can be replaced by making use of the in: comparison operator in the second query.

Figure 11 - WHERE Data Set Option with Colon Modifier

```
proc sql _method;
 create table ex11a as
 select *
 from   scores
 where  subject_id like '100-%' or
         subject_id like '20%';

 create table ex11b as
 select  *
 from   scores(where=(subject_id in: ('100-','20')));
 quit;
```

Multiple varying length text string searches can be replaced with a single condition courtesy of the WHERE data set option and the colon

LIMITATIONS

There are limitations to using data set options with PROC SQL, which include the following:

- With the exception of READ, WRITE, ALTER and PW, data set options are not valid when specified for SQL created views in the FROM clause of queries. You will receive an error message in the log if you try to employ them. However, data set options are valid in the FROM clause for views created with a DATA step.
- Using data set options in the CREATE VIEW statement of PROC SQL will generate a warning indicating that they will be ignored, except for READ, WRITE, ALTER, PW and LABEL.
- Data set options are not valid syntax when querying a non-SAS relational database such as ORACLE and DB2 with the PROC SQL Pass-Through facility. They are valid when querying non-SAS relational databases when the connection to the tables is established with a LIBNAME.
- The data set options POINT and IN are not valid in the FROM clause.
- The data set options COMPRESS, INDEX and LABEL are ignored in the FROM clause, just as they are in the SET and MERGE statements of the DATA step.
- Data set options are not valid in the FROM clause of queries against the SAS system (metadata) DICTIONARY tables.

CONCLUSION

This paper has demonstrated how the following data set options can be effectively employed in PROC SQL:

- DROP and KEEP – Used in the FROM clause and CREATE TABLE statement in conjunction with the * wildcard to selectively include/exclude fields in the output destination while noting the availability of the number range, name range, and name prefix lists to aid in this process.
- LABEL – Adding a descriptive label to data sets in the CREATE TABLE and CREATE VIEW statements is a good programming practice.
- COMPRESS - Data set compression is an excellent way to manage storage space. However, not every data set is good candidate for compression and additional CPU is needed to operate on compressed data sets.
- RENAME – Changing the name of variables in the FROM clause may be necessary when fields have an alternative meaning, such as CASE and CALCULATED. It can be used to change the names of key variables in data sets in order to make use of a natural join to combine tables. The numbered range list is also available to you to rename an array of fields.
- SORTEDBY – Indicating to the SQL query optimizer that data sets are ‘in order’ when this information is not captured in the metadata can direct it to using a more efficient join strategy and/or avoid superfluous sorting.
- WHERE – Can be used as an alternative to the more verbose in-line view and allows for in: as a valid logical operator.

There are other useful data set options that can be used in PROC SQL that have not covered in this paper. The IDXNAME data set option can used in the FROM clause to hint to the query optimizer which index to use. The Scalability Performance Data (SPD) Engine has data set options that are not pertinent to Base SAS engine, such as PARTSIZE and IOBLOCKSIZE.

Balancing clarity, brevity, descriptiveness and efficiency in generating code is a key goal for which all SAS programmers should aim. For those who use PROC SQL, as well as the DATA and other PROC steps, making good use of data set options can help achieve this goal.

References

Delwiche, L.D. and S.J. Slaughter, The Little SAS Book: A Primer 2nd Edition, SAS Institute Inc., Cary, NC 1997

Dorfman, Paul M. (2001), "Table Look-up by Direct Addressing: Key-Indexing – Bitmapping – Hashing", *Proceedings of the 26th Annual SAS Users Group International*

--- and Gregory Snell (2002), "Hashing Rehashed", *Proceedings of the 27th Annual SAS Users Group International*

Johnson, Jim (2005), "The Use and Abuse of the Program Data Vector (The Procs)", *PharmaSUG'05*

Karp, Andrew and David Shamlin (2003), "Indexing and Compressing SAS Data Sets: How, Why and Why Not", *Proceedings of the 28th Annual SAS Users Group International*

Kent, Paul, "SQL Joins – The Long and Short of It", TS-553, SAS Institute, Inc., Cary, NC

Lavery, Russell (2005), "The SQL Optimizer Project: _Method and _Tree in V9.1", *Proceedings of the 30th Annual SAS Users Group International*

Lund, Pete (2005), "An Introduction to SQL in SAS", *Proceedings of the 30th Annual SAS Users Group International*

Murphy, William C. (2004), "Colonoscopy for the SAS Programmer", *Proceedings of the 29th Annual SAS Users Group International*

SAS V9 On-Line Document

Acknowledgements

The author would like to thank Jenni Borowiak and the entire SAS programming department at Howard M. Proskin & Associates, Inc., for their insightful comments in reviewing this paper and discussions surrounding this topic. The author would also like to thank Howard Proskin for his encouragement in being an active member of the SAS community and allowing time at work to write this paper.

Contact Information

Your comments and questions are valued and encouraged.

Contact the author at:
Kenneth W. Borowiak
Howard M. Proskin & Associates, Inc.
300 Red Creek Drive, Suite 220
Rochester, NY 14623

E-mail: kborowiak@hmproskin.com
Jsquare94@aol.com

Appendix 1 – Code to generate sample data sets

```
data Tx(label='Treatment Code Information' drop=a);
 do a=1 to 3000;
       Subject_No=a;
       if a le 1500 then Center=100; else Center=200;
       random_date=int(15000+20*ranuni(315975)); format random_date mmddyy10.;
       if ranuni(97511) lt .5 then tx=1; else tx=2;
   output;
  end;
  label tx='Treatment Code'
        random_date='Randomization Date'
        subject_no="Subject Number";
run;
```

```
NOTE: The data set WORK.TX has 3000 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

```
data Scores(label='Recorded Scores at Visits' drop=j c);
length Subject_ID $8 Visit 8 A1-A10 8 B1-B10 $20; /* B fields made intentionally   */
 array A[10]; array B[10];                        /* longer than they needed to be */
 do c=1 to 3000;
       if c le 1500 then subject_id=compress('100-'||put(c,z4.));
          else subject_id=compress('200-'||put(c, z4.));
    do Visit=1 to 20;
        do j=1 to 10;
           a[j]=j*ranuni(j);
           b[j]=left(put(.5+j**2*ranuni(j),7.3));
         end;
        output;
     end;
  end;
format a: 5.2;
run;
```

```
NOTE: The data set WORK.SCORES has 60000 observations and 22 variables.
NOTE: DATA statement used (Total process time):
      real time           1.11 seconds
      cpu time            1.10 seconds
```

```
data Surgery;
 length SID $8 Visit 8 case 8;
  do k=5 to 3000 by 197;
    if k lt 1500 then SID=compress('100-'||put(k,z4.));
         else SID=compress('200-'||put(k, z4.));

        Visit=10+int(10*ranuni(971156)); /* only one visit per subject */
        case+1;
        drop k;
       output;
  end;
run;
```

```
NOTE: The data set WORK.SURGERY has 16 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```