PharmaSUG 2014 – Paper TT05

Principles of Writing Readable SQL

Kenneth W. Borowiak, PPD, Morrisville, NC

ABSTRACT

PROC SQL is a commonly used data manipulation utility by SAS® users. Despite its widespread use and logical components, there is no accepted standard for writing PROC SQL statements. The benefits of well-written code include ease of comprehension and code maintenance for both the original author and those who inherit it. This paper puts forth some principles of writing readable SQL in an objective manner. Each principle is followed by a suggested coding style. The ultimate goal is to elucidate the attributes of well-written PROC SQL code that users will implement to maximize comprehension by all.

INTRODUCTION

PROC SQL is a commonly used data manipulation utility by SAS users. Despite its widespread use and logical components, there is no accepted standard for writing PROC SQL statements. The benefits of well-written code include ease of comprehension and code maintenance for both the original author and those who inherit it[1]. Conversely, a functional but poorly written statement can add a significant amount of time to debug.

This paper puts forth some principles of writing readable SQL in an objective manner. In each of the following sections of the paper a suboptimally written SQL statement is provided. A principle is then presented, followed by a suggested coding style for it. The ultimate goal is to elucidate the attributes of well-written PROC SQL code that users will implement to maximize comprehension by all.

Before delving into the principles of writing readable SQL statement, below are some general coding standards that the paper will use as a basis.

- **Code should appear within the first 90 characters of a line.** Regardless of the editor used (e.g. SAS Enhanced Editor, TextPad, UltraEdit, etc.), the code should appear on the page without having to scroll horizontally. This also enables printing of code that does not wrap to the next line.
- **NO YELLING!** A de facto standard of writing SAS code is that it should not be all capital letters.
- **Good commenting** - A good comment should not restate the obvious Instead it should tell why the statements exist. For example, /* Sort data by USUBJID */ before a PROC SORT step is not particularly insightful. Conversely, /* Sort data by date to identify baseline records*/ indicates why the procedure will be called.
- **Consistency** - A good style > A consistent but bad style > An inconsistent style
- **Liberal use of whitespace.** Indentation and skipping lines between blocks of code makes it more readable.

PRINCIPLE #1 – ONE KEYWORD PER LINE

Principle #1 - There should be only one statement/clause keyword on each line of a query.

Like many programming languages, the execution of a SAS program involves two stages: compile and

[1] Fehd (2003) makes many good points on the importance of the readability of code.

run. Amongst the activities that happen at the compile-time stage, statements are scanned to see if they adhere to rules of the language. For example, the compile-time step verifies that a procedure-step begins with the keyword PROC or that a DATA step statement ends in a semi-colon. The compiler does not care about the style of the code, only if is valid. If a set of statements is successful in the compile-time stage then it will attempt to execute them.

Programmers are human, so the written style of the code has an effect on readability and comprehension. The SQL step in Figure 1 is syntactically correct and will execute successfully, but has some noticeable short-comings.

```
proc sql; create table class1 as select a.name,a.age,a.weight, a.height,
a.sex,b.education from sashelp.class as a left join work.education as b
on a.name=b.name where a.name like ‘A%’ order by a.sex;quit;
```

Figure 1 – The suboptimal ‘paragraph’ style

SAS code is not intended to be written in paragraphs like a book. SAS has many types of steps code with logical components. DATA steps typically have a number of statements each ending with a semi-colon. The Data Manipulation Language (DML) category of statements of SQL are usually longer and end with a single semi-colon. They have logical components, such as CREATE TABLE/VIEW, SELECT, FROM, etc. and those logical components should be displayed prominently. Consider the counting query in Figure 2, which highlights the components of the statement by using all caps for keywords, in addition to any color-coding that may be applied by the editor.

```
proc sql;
 CREATE TABLE summary AS
 SELECT  member, period, count(*) as cnt
 FROM  membership WHERE member=1267
 GROUP BY member, period ORDER BY member, period desc;
 quit ;
```

Figure 2 – Upper case keywords can be helpful, but not they are not enough for readability

While this appears to be an improvement over the style in Figure 1 there is still room for improvement.

In practice, Principle #1 implies that the statements/clauses CREATE TABLE/VIEW, SELECT, FROM, WHERE, GROUP BY, ORDER BY, HAVING, UPDATE, SET, INSERT should be found at the beginning of each line in an outer query or directly following an open parenthesis for a subquery. The query in Figure 3, which creates a macro variable, adheres to this principle.

```
proc sql ;
  select   max( aestdtn )
  into     :max_date
  separated by ''
  from     adb.adae
  where    usubjid='10019982'
  ;
  quit ;
```

Figure 3 – Create a macro variable without any leading or trailing spaces

While it is tempting to add the INTO and SEPARATED clauses on the same line as the SELECT, the line could be considerably longer if creating a series of macro variables. As we strive for a consistent style, INTO and SEPARATED should be put on their own line, regardless of the amount of macro variables created. Also note that the keywords are no longer in upper case, as their prominence should be apparent by their position, but they certainly could be. Whatever convention you choose or inherit for the case of keywords, whether proper, lower or upper case, stay consistent throughout the program.

PRINCIPLE #2 – ONE COMMA PER LINE

Principle #2 - No more than one comma per line, except when needed for function calls.

This principle related to Principle #1 suggests the code be written in a more vertical fashion by forcing a new line for each comma. Consider Figure 4 below, which is a re-written version of the query in Figure 2.

```
proc sql ;
 create table summary as
 select   member
        , period
        , count(*) as cnt
 from     membership
 where    member=1267
 group by  member
         , period
 order by  member
         , period desc
 ;
 quit ;
```

Figure 4 - One comma per line

In Figure 4, there is a separate line for each of the variables in the SELECT, GROUP BY and ORDER BY statements. One style for applying this principle is to have the comma appear at the beginning of a line rather than at the end of the previous line. This has the benefit of having the comma in a consistent location. This style can be applied to macro to calls to enhance readability, as demonstrated in Figure 5.

```
%align
 (  in_data  = results1
  , out_data = results2
  , symbol   = .
  , where    =  trtpn is not missing )
```

Figure 5 – One comma per line for macro calls

PRINCIPLE #3 – ONE GROUPED CONDITION PER LINE

Principle #3 - Only one grouped condition per line.

This principle applies to CASE statements, join conditions and subsetting conditions. The principle continues to enforce a top-down writing structure. Consider the query below in Figure 6. It does not consistently adhere to any of the three principles, and uses a ‘diagonal’ style.

```
proc sql ;
create table class1 as
   select a.name, a.age, a.weight, a.height, a.sex,
          case when b.education>30 then 30 else b.education end as education
         from    sashelp.class as a
                         inner join
                          work.education as b
                          on a.name=b.name and a.age=b.age
                                where a.name like 'A%'  and a.sex='F'
                                       order by a.sex ;
                                            quit ;
```

Figure 6 –The dreaded ‘diagonal’ style

The ‘diagonal’ style puts the code at risk of not fitting on the first 90 characters of a line, particularly if a subquery is involved. But it also makes for an indecipherable block of code. In Figure 7 below the query is re-written using Principles 1, 2 and 3. Notice how easy it is to see exactly what is being done: CREATE, SELECT, FROM, WHERE and ORDER practically pop off the page. And the elements within each section are clearly defined in a grocery list-like fashion.

```
proc sql ;
  create table class1 as
  select   a.name
         , a.age
         , a.weight
         , a.height
         , a.sex
         , case when b.education>30 then 30
                else b.education
              end as education
   from    sashelp.class as a
           inner join
           work.education as b
           on   a.name=b.name
              & a.age=b.age
   where   a.name like ‘A%’
         & a.sex=’F’
   order by  a.sex
   ;
  quit ;
```

Figure 7 – One condition per line

The query in Figure 7 contains only simple expressions, which *can* be enclosed in parenthesis to enhance readability. For example, the join condition involving the key variable NAME could be written as `( a.name=b.name )`. If a grouped condition is a compound expression then it *should* be enclosed within parentheses and can be put on the same line, as long as it stays within the 90 character limit. If the compound condition is too long to fit on one line then put each sub-expression on its own line. Figure 8 shows to how to write a WHERE clause with an OR condition on a pair of compound expressions.

```
where   ( a.name like ‘A%’ & a.sex=’M’ )
      | ( a.sex=’F’ and a.age > 20 )
….
;
quit ;
```

Figure 8 – One grouped condition per line

PRINCIPLE #4 – INNER AND OUTER QUERY CONSISTENCY

Principle #4 – Principles #1, #2 and #3 (Keywords, Commas and Grouped Conditions) should be applied to both inner and outer queries.

There are different types of subqueries (a.k.a. inner queries), such as in-line views and correlated and uncorrelated subqueries. For consistency, adhere to the principles already presented in the previous sections for inner queries. Even though many subqueries are likely to begin between characters 10 through 20 on a line, using a vertical writing style will enable you to stay within the first 90 characters of the screen. The query below in Figure 9 contains an in-line view and uncorrelated subquery.

```
proc sql;
   create table class2 as
   select    T1.*
           , V2.education
   from      sashelp.class as T1
           , ( select   *
               from     education
               where    gender is not missing ) as V2 /* In-line view */
  where      T1.name=V2.name
           & T1.name in ( select   name
                          from     graduates ) /* Uncorrelated subquery */
  order by   T1.name
  ;
  quit ;
```

Figure 9 – Inner queries should be written in the same manner as outer queries

PRINCIPLE #5 – SUBQUERIES ENCLOSED IN PARENTHESIS

Principle #5 - Each non-trivial subquery should be contained within parentheses.

A common misconception is that a subquery is defined as a query within a set of parentheses. The uncorrelated subquery above in Figure 9 *did not* need to be enclosed within a pair of parentheses. However, enclosing does help distinguish it as a subquery and Principle #5 reinforces this commonly used practice. Note that in-line views in the FROM clause, such as the one in Figure 9, are required to have the parentheses for valid syntax.

Another type of query where this comes into play is when UNION operations are involved. Consider the suboptimally written query below in Figure 10.

```
proc sql ;
 create table class as
 select  * from sashelp.class
 where   sex=’M’
 union all corr SELECT *
 FROM SASHELP.CLASS WHERE SEX=’F’
 ORDERY BY SEX;
 quit ;
```

Figure 10 – A poorly written query involving a UNION operation

The mix of lower case coding for the top part of the query and upper case for the bottom is both inconsistent and misleading about how the query operates. The created table CLASS is the result of two subqueries stacked upon each other, which is then sorted. Following Principle #5, the query should be re-written as in Figure 11 below, with each subquery contained in parentheses.

```
proc sql feedback ;
 create table class as
 ( select  *
   from    sashelp.class
   where   sex='M' )
 union all corr
 ( select   *
   from     sashelp.class
   where    sex='F' )
 order by sex
 ;
 quit ;
```

Figure 11 – A query with parenthesis around the subqueries in a UNION operation

Another type of query worthy of mention in this section are those explicitly using the PROC SQL Pass-Through facility to access non-SAS Relational DataBase Management Systems (RDBMS), such as Oracle, DB2 and Teradata. The explicit Pass-Through query in Figure 12 uses an in-line view with the alias V1 in the FROM clause to execute statements in the DBMS space and return the records to the SAS environment for further processing within the PROC SQL step. Principle 5 is enforced because the parentheses around the in-line view are required for valid syntax. The in-line view V1 is written according to Principle 4 in a top-down style.

```
proc sql  noprint feedback;
  connect to oracle( path=&ocserver.  ) ;

  select  V1.clinical_study_id
  into    :csi
  separated by ''
  from    connection to oracle
          (  select   *
             from     rxa_des.clinical_studies
             where    study = %unquote(%bquote('&nickname.'))  ) as V1
  ;
  disconnect from oracle ;
  %put %str(N)OTE- Macro variable CSI resolves to: &csi. ;
  quit ;
```

Figure 12 – Explicit Pass-Through facility queries are subqueries

MORE DATA MANIPULATION LANGUAGE STATEMENTS

All of the examples presented so far have created tables with a SELECT statement, which falls into the DML category of SQL statements. The principles presented in the paper apply to other DML statements, as well. Consider the query in Figure 13, which includes a DELETE statement involving a correlated subquery with an EXIST operation. It adheres to all relevant principles, namely Principles 1, 3, 4 and 5.

```
proc sql ;
 delete
 from     class as T1
 where    exists( select  1
                  from    education as V2
                  where   T1.name=V2.name )
 ;
 quit ;
```

Figure 13 – A DELETE operation involving a correlated subquery

Another DML statement is INSERT, which has a few different ways to be called and can insert a single row or multiple rows. Consider the query below in Figure 14, which inserts a new record of values into an existing table.

```
proc sql feedback ;
  insert into class
  set   name='Ken'
      , sex='M'
      , height=60
      , weight=.B
      , age=.A
   ;
   quit ;
```

Figure 14 – An INSERT operation of a new record into an existing table

DATA DEFINITION LANGUAGE STATEMENTS

The principles put forth were directed at DML queries. The Data Definition Language (DDL) category of SQL statements, such as CREATE, ALTER and DROP entities such as tables, views and indexes are more limited in the number of clauses that appear. The principles of one comma and one keyword per line can be applied to some types of statements, such as the CREATE TABLE definition of the CLASS1 data set and DROP TABLE statements of the CLASS1 and CLASS2 data sets shown in Figure 15. However, the CREATE INDEX and CREATE TABLE LIKE statements are terse, so writing those on a single line may make more sense.

```
proc sql ;
 create table class1
  (   Name char(8)
    , Sex char(1)
    , Age num
    , Height num
    , Weight num )
 ;

 create index n_a on class1( name, age )
 ;

 create table class2 like class1
 ;

 describe table class2
 ;

 drop table  class1
           , class2
 ;
 quit ;
```

Figure 15 - Data Definition Language statements

DATA SET OPTIONS

A consequence of Principle 1 is that the WHERE data set option should be avoided. The WHERE clause is an important and expected part of a SQL query. Having the `(WHERE=( <condition> ))` directly following the FROM clause is a violation. It also makes sense to use a SQL clause in a SQL step rather than an analogous SAS data set option. Also, a WHERE data set option has the limitation that it only pertains to subsetting, as it will not facilitate join relations between two tables.

While the WHERE data set option has been denounced, other data set options can be advantageous if used appropriately. The KEEP and DROP[2] data set options can be used in the CREATE TABLE statement and FROM clause to process a short-list of variables in lieu of a long explicit list of variables in a SELECT statement. However, if approximately the same number of variables need to be specified in the SELECT statement or a KEEP/DROP data set option in a SQL statement then using the SELECT

[2] See Borowiak (2010) for a detailed discussion on using the KEEP and DROP data set options in PROC SQL.

statement is preferred. The IDXWHERE and IDXNAME[3] data set options can be useful for helping the SQL Query optimizer decide if and which indexes to use, respectively.

CONCLUSION

In general, the principles put forth for writing SQL statements are intended to increase readability and comprehension of the code by both the original author and those who inherit and maintain it. The style in which the principles can be applied can vary in some instances. Regardless of the style, writers should follow the principles and consistently follow their style, whether it is for formal production code or a quick ad hoc query. There may be times where an author purposely abandons the principles, but hopefully it would be for a specific atypical situation or where it serves to enhance the readability of a section of code.

REFERENCES

Borowiak, Kenneth W. (2009), “A Closer Look at PROC SQL’s FEEDBACK Option”, *Proceedings of the NorthEast SAS Users Group Conference*
http://www.lexjansen.com/nesug/nesug09/cc/CC18.pdf

Borowiak, Kenneth W. (2010), “Variable List Short-Cuts in PROC SQL”, *Proceedings of the NorthEast SAS Users Group Conference*
http://www.nesug.org/Proceedings/nesug10/cc/cc03.pdf

Fehd, Ronald J. (2003), *“A Journeyman’s Reference: The Writing for Reading SAS Style Sheet”, Proceedings of the SouthEast SAS Users Group Conference*
http://www.analytics.ncsu.edu/sesug/2003/AD08-**Fehd**.pdf

Raithel, Michael, (2006) – *The Complete Guide to SAS® Indexes*, SAS Institute, Inc., Cary, NC

SAS OnLineDoc®

ACKNOWLEDGEMENTS

The author would like to thank Karen Easterling, Phyllis Wolf, MaryRuth Roth, Paul Lyon, Toby Dunn, Ted Temple and John King for their insightful comments in reviewing this paper.

DISCLAIMER

CONTACT INFORMATION

Your comments and questions are valued and encouraged.

Ken Borowiak
PPD
3900 Paramount Parkway
Morrisville NC 27560

Ken.Borowiak@ppdi.com
Ken.Borowiak@gmail.com

[3] See Raithel (2006) for an examination of SAS indexes and the use of the IDXWHERE and IDXNAME data set options.