PharmaSUG 2014 - Paper BB10

# Using the Power of SAS SQL

Jessica Wang, Regeneron Pharmaceuticals Inc., Basking Ridge, NJ

## ABSTRACT

SAS is a flexible language in that the same task can be accomplished in numerous ways. SAS SQL is a powerful tool for data manipulation and query. SAS SQL can make your programs more efficient, simpler and more readable. Topics in the paper will cover: merging multiple tables by different columns and different rules; SQL in-line view; SQL set operation; using dictionary tables; creating empty tables with pre-defined dataset structure; inserting rows into a dataset; assigning a list of macro variables. Some tricks and tips from the author's personal experience as a SAS user will also be shared: e.g. using COMPRESS to save storage size; using SQL options _METHOD to understand your SQL code better; using NOPRINT to compress unnecessary display in output window. This paper is intended for intermediate to advanced SAS SQL users who already know the basics of SAS/SQL, and want to better exploit the power that SQL offers.

## INTRODUCTION

PROC SQL is SQL (Structured Query Language) built into SAS system, used for data retrieving, manipulation, analysis, and reporting. It also brings SAS language elements, such as format/informat, functions, and data set options into SQL. The syntax for a basic SQL query looks like the following:

```
PROC SQL;
    CREATE TABLE new_table_name AS
           SELECT column_names
           FROM table_name
           WHERE conditions
           GROUP BY column_names
           HAVING having_conditions
           ORDER BY column_name_list
    ;
QUIT;
```

The **PROC SQL** statement, and the **SELECT** and **FROM** clauses are required in a SQL query. Others are optional. The SELECT clause can list out the columns to be selected or use asterisk sign (*) to select all columns; it can also create new columns by using function, calculation, and assign alias names for the new columns. The FROM clause can list the original table name. CREATE TABLE clause save the query result into a new table. The WHERE clause list the query conditions that you want to be added to the query result. The ORDER BY clause is used to sort the result. GROUP BY is usually used together with group functions in the SELECT clause to subset the query result or to summarize the result. The HAVING clause, used together with GROUP BY clause, select only the query result that satisfies the condition after the group function. The order of the SQL clauses should follow the order given in the syntax above.

In SQL language, SAS terminology **dataset**, **observation**, **variable** are also called **table**, **row**, and **column** respectively. To differentiate from other SAS procedures, we use SQL language names in this paper. Now, we are going to exploit the SQL power further.

In this paper, fake clinical trial data in pharmaceutical industry is used for examples; such as Demographics (DM), Exposure (EX), Adverse Events (AE), Laboratory Test Results (LB), IVRS (Interactive Voice Response System), Randomization, etc. Data definition metadata table is listed in Appendix 1.

## 1. MERGING MULTIPLE TABLES

In some cases we need to obtain data from multiple tables and combine the tables horizontally by specified conditions, PROC SQL JOIN can do this type of work efficiently.

Example: Three tables, *IVRS*, *RANDTRT*, and *TRTDEC* (alias names as *A, B, C*), contain the treatment information we need for patients. Table *A* has individual patient information and randomization numbers; *B* has randomization number and corresponding treatment *group. C* has treatment group and treatment description. We need to merge *A*, *B* based on randomization numbers, and merge *B* and *C* by treatment groups in order to get subject's treatment group and treatment description. Note: randomizing number in table *A* and *B* has different names (*rand_number* and

*rand_num* respectively).

Code Example 1-1 (inner join)

```
PROC SQL NOEXEC;
    create table randomiz as
           select a.*, b.trtgrp, c.trt_description
           from ivrs as a, randtrt as b, trtdec as c
           where a.rand_number = b.rand_num
                  and b.trtgrp = c.trtgrp
           order by subjid
    ;
QUIT;
```

Another example is merging tables using inequality conditions. Table *AE* has AE term, serious event flag, and AE start date. In this example, we want to get the incidence treatment information when the serious AE occurred. We merge *AE* with *EX* table by comparing AE start date with EX start and end date. If AE start date is between EX start and end date, we retrieve the treatment information at that period as the query result.

Code Example 1-2 (outer joins – left join)

```
PROC SQL;
    create table saeinfo as
           select aeterm, b.extrt,aestdt, exendt, exstdt
           from ae (where=(aeser='Y')) as a
           left join ex as b
           on a.subjid = b.subjid
                  and b.exstdt <= a.aestdt< b.exendt
                  and not missing(b.exstdt)
           order by a.subjid, a.aestdt
    ;
QUIT;
proc print; run;
```

**Log File** (for example 1-2)

```
NOTE: Table WORK.SAEINFO created, with 29 rows and 8 columns.
```

**Output File** (part of the output file for example 1-2)

```
 ...
52    RG200mg    11JUN2013    03SEP2013    01OCT2012
53    RG300mg    29NOV2012    14AUG2013    10SEP2012
54    Placebo    07MAR2013    23OCT2013    21NOV2012
55    RG300mg    18NOV2013    24DEC2013    29JAN2013
```

There are several advantages of using PROC SQL for merge: 1. no sorting is needed ; 2. variable names to merge by can be different; 3. More sophiscated data manipulation can be accomplished by using not only the equal operator, but also comparison operators >, >=, <, <=, between etc.; 4. multiple tables can be joined using different rules within one single SQL statement.

When you do multiple tables join, especially full join, such as when conducting a query to combine all abnormal LB and VS data into one single table, you would have column *subjid* in both tables. However, if *subjid* value is taken from LB, *subjid* will be missing for patients who only have abnormal VS data. . In this case COALESCE function can help you take the first non-missing value by checking variable values from left to right such as in *COALESCE(a.subjid, b.subjid)*.

Code Example 1-3 (full join with value missing in common column)

```
PROC SQL;
    create table abnormal as
           select COALESCE(a.subjid, b.subjid) as subjid, a.lbtest, a.lbstresn,
           b.vstest, b.vsstresn
           from lb as a
           full join vs as b
```

```
            on lbabnfl= 'Y'
            order by subjid
    ;
QUIT;
```

Notes: the key word NOEXEC in the PROC SQL statement in example 1-1 is used to check the syntax only. All the statements under this PROC SQL statement will not be executed. Another way to do the syntax check is by using the VALIDATE statement. Its scope is within the statement only. The key word VALIDATE has to be used before CREATE or SELECT clause.

Example 1-2 uses SAS dataset option *WHERE*= in the FROM clause to subset the AE table to process only SAEs. SAS function MISSING is used on the join condition. Most of the SAS functions can be used in PROC SQL, but there are some exceptions- SAS ARRAY cannot be used in SQL, or the SAS variable information functions.

Example 1-3 is a full join, which may not be able to fully utilize the optimizer for SQL execution. It might take a long time to execute if the tables to be joined are large in size.

## 2. SQL IN-LINE VIEW

SQL IN-LINE VIEW is an alternative way to join multiple tables together. An in-line view exists in the FROM clause. Example: you would like to calculate how many patients in each treatment group who has AE of hypertension. This query can be achieved in several ways. In example 2-1, we get the patient count in two simple steps. Step 1, get unique patient list that has *aeterm* as hypertension from *AE* table, and save the result in table *HYPERTEN*; step 2, join *EX* table with table *HYPERTEN* by *subjid*, group by *extrt*, and count distinct *subjid* for each treatment group.

Code Example 2-1 (two step query)

```
PROC SQL;
    create table hyperten as
           select distinct subjid
           from ae
           where upcase(aeterm)='HYPERTENSION'
    ;
    create table sumtb1 as
           select extrt, count(distinct a.subjid) as count
           from ex as a
           join hyperten as b
           on a.subjid = b.subjid
           group by extrt
           order by extrt
    ;
QUIT;
```

To create the same query result, you can also use in-line view in one SQL statement. From example 2-2, you can see that an in-line view is created in the FROM clause by putting query in parentheses. The in-line view can be the same or a different table as the main query. In this example, you use in-line view to create a temp query result of *aeterm* = HYPERTENTION by querying *AE* table, and using this result for the outer query processing. The in-line view in this example serves the same function as the *HYPERTEN* table in example 2-1. You can understand the meaning of the name ***in-line*** as it is an intermediate step and temporary view that does not have a name, and can only be referenced within this SQL statement where it is defined; it follows all the rules/restrictions for a SQL view, e.g. it cannot include an ORDER BY clause.

Code Example 2-2 (in-line view)

```
PROC SQL;
    create table sumtb2 as
    select extrt, count(distinct subjid) as count
    from ex
    where subjid in
    (     select distinct subjid
          from ae
          where upcase(aeterm)='HYPERTENTION'
    )
    group by extrt
```

```
    order by extrt
    ;
QUIT;
proc compare data=sumtb1 compare=sumtb2; run;
```

**Output File** (part of the compare output file for example 2-2)

```
 ...
Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 31.

NOTE: No unequal values were found. All values compared are exactly equal.
```

Output file show the query results from the two-step SQL and the in-line view query are identical.

Notes: when making a decision whether to use in-line view or to create multiple tables, one should think about if this data will be used elsewhere and whether it will be used frequently Key word DISTINCT removes duplications, and keeps only unique *subjid*; or in the group functions, such as *count,* it makes sure that the same *subjid gets counted* only once.

## 3. SQL SET OPERATION

Horizontally combining tables is discussed in section 1 and 2 by using SQL JOIN and IN-LINE VIEW, this section will focus on SQL SET operation as a way to combine tables vertically. SQL SET operators include **EXCEPT**, **INTERSECT**, **UNION**, and **OUTER UNION**. The relationship between the different SET operators is illustrated in Figure 1:

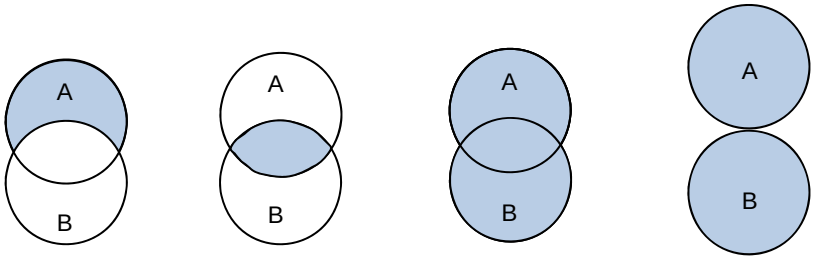


**Figure 1. Relation between different SET operators**

Row level: EXCEPT is in table A BUT NOT in table B. INTERSECT is in BOTH table A and B. UNION is in EITHER table A or table B (or both). OUTER UNION is to concatenate A and B, regardless of the relation between A and B. By default, the EXCEPT, INTERSECT, and UNION operators do not eliminate duplicate rows after the combination. OUTER UNION keeps only unique rows.

Column level: EXCEPT, INTERSECT, and UNION takes the column name from the first table, and overlay the columns by query POSITION, regardless of whether columns with the same name exist or not. OUTER UNION does not overlay columns, keeping columns from both tables.

```
PROC SQL;
    SELECT column_names
    FROM tableA
    set-operator <ALL> <CORR/CORRESPONDING>
    SELECT column_names
    FROM tableB
    ;
QUIT;
```

The set-operator can be chosen from one of the set operations - EXCEPT, INTERSECT, UNION, and OUTER UNION. The <ALL> <CORR/CORRESPONDING> are optional keywords. Two key words CORR and ALL can be used to modify the SET operation default results. Key word ALL will select all rows, regardless of duplication. CORR overlay columns by name instead of by position; in cases where alas names are defined, columns will overlay by

alias names.

In Example 3-1, we do a query to find patients with any AE or abnormal vital sign values by setting query results from table *AE* and table *VS* together vertically with UNION operator.

Code Example3-1 (set operator - UNION)

```
PROC SQL;
    select ae.subjid
    from ae
    UNION
    select vs.subjid
    from vs
    where vsabnfl='Y'
    order by 1
    ;
QUIT;
```

Example 3-2 is used in a scenario where you already have an AE report called *RPTAE1* from last month, and now you need to do another AE report for new AEs only. You can create a preliminary AE report from the *AE* table first, and then use SET operation – EXCEPT to keep only those AEs that are not part of the reporting in *RPTAE1*.

Code Example3-2 (set operator - EXCEPT)

```
PROC SQL;
    select *
    from ae
    EXCEPT
    select *
    from rptae1
    order by subjid
    ;
QUIT;
```

By comparison, SAS DATA step SET statement is more like OUTER UNION in the SQL SET operation. SQL SET operators, provide more flexibility with identical query results. In many case, the SQL query are more straightforward in expressions and in defining the conditions.

## 4. USING DICTIONARY TABLES

SQL dictionary tables are generated at the beginning of the SAS session, and get updated by the system automatically during the session, and are read only. These tables contain real time information on data library, table, column, macro, external files in use, current effective titles and footnotes, and system options, etc. Table 1 lists some of the useful tables.

| Dictionary Table Name | Dictionary Table Contains |
|---|---|
| Dictionaries | Information of the dictionary tables |
| Members | Objects in current data libraries |
| Catalogs | catalog entries, e.g. format catalog |
| Macros | global macro variables defined by system and users |
| Options | current settings of SAS system options |
| Tables | detailed information about data sets |

| | |
|---|---|
| Columns | variables names and attributes in tables |

Table 1 Selected SQL Dictionary Tables

The following is an example of using the dictionary tables. The first SQL statement is an example of checking all the variables in the dictionary catalogs table; it will give you the catalogs table structure, that you can utilize for further query regarding the contents of catalogs table. The second statement in the example is a query on the *AE* table. It will give you all the column names in the *AE* table along with the columns' properties, such as number of records with non-missing value, number of missing records, etc.

Code Example 4-1 (using dictionary tables)

```
PROC SQL;
    describe table dictionary.Catalogs;
    select * from dictionary.columns
          where libname='WORK' and name='AE';
QUIT;
```

**Log File** (for example 4-1)

```
237  PROC SQL ;
238      describe table dictionary.Catalogs;
NOTE: SQL table DICTIONARY.CATALOGS was created like:

create table DICTIONARY.CATALOGS
  (
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   objname char(32) label='Object Name',
   objtype char(8) label='Object Type',
   objdesc char(256) label='Object Description',
   created num format=DATETIME informat=DATETIME label='Date Created',
   modified num format=DATETIME informat=DATETIME label='Date Modified',
   alias char(32) label='Object Alias',
   level num label='Library Concatenation Level'
  );
...
NOTE: PROCEDURE SQL used (Total process time):
      real time           7:20.74
      cpu time            9.22 seconds
```

Note: In comparison with PROC DATASETS procedure, the PROC SQL outputs are easier to manipulate. SASHELP stores PROC SQL LIBRARY views, e.g. VTABLE, VCOLUNM, VTITLE, VMACRO.

## 5. CREATING TABLES WITH PRE-DEFINED TABLE STRUCTURE

PROC SQL provides several ways to create new tables. 1. Using the key word LIKE to create an empty table to have a structure similar to an existing table; 2. Creating an empty table by defining the columns; 3. Creating a new table from a query result. You are going to see examples for each of these three methods.

Example 5-1 is a scenario where you already have a statistical table called *STAT1*, and you need to generate another empty table called *STAT2* with the same structure as stat1. The empty table *STAT2* will be used to hold analysis results in a later stage. The following code will create *STAT2* table.

Code Example 5-1 (creating an empty table using LIKE)

```
PROC SQL;
    create table stat2
```

```
            like work.stat1
    ;
QUIT;
```

**Log File** (for example 5-1)

```
NOTE: Table WORK.STAT2 created, with 0 rows and 6 columns.
```

Example in 5-2 shows how to create a new table by defining columns. First part of the code shows that when a table (*DM*) with a similar structure is available, one can use DESCRIBE TABLE statement to display the table definition code in the log file for this particular table; and then you can copy, paste, and modify this code (e.g. keep the columns you need, add new columns, define index) to create a new table named *NEWDM* as shown in Part two. Note that column *armn* is dropped, and a new column *sexn* is added. The last statement CREATE INDEX creates an index *subjid* on column *subjid.*

Code Example 5-2 (creating an empty table by defining columns)

```
PROC SQL;
    describe table dm ;
QUIT;
```

**Log File** (for example 5-2 part 1)

```
PROC SQL ;
describe table dm ;
NOTE: SQL table WORK.DM was created like:

create table WORK.DM( bufsize=12288 )
  (
   SUBJID char(6) label='Subject Identifier for the Study',
   ARM char(17) label='Description of Planned Arm',
   TRTSDT num format=DATE9. label='Date of First Exposure to Treatment',
   AGE num label='Age',
   SEX char(1) format=$6. informat=$6. label='Sex',
   RACE char(41) label='Race',
   RANDFL char(1) label='Randomization Flag',
   RANDFN num label='Randomization Flag (N)',
   RANDDT num format=DATE9. label='Date of Randomizaiton',
   armn num
  );
```

```
PROC SQL;
    create table WORK.NEWDM
    (
       SUBJID char(6) label='Subject Identifier for the Study',
       ARM char(17) label='Description of Planned Arm',
       TRTSDT num format=DATE9. label='Date of First Exposure to Treatment',
       AGE num label='Age',
       SEX char(1) format=$6. informat=$6. label='Sex',
       RACE char(41) label='Race',
       RANDFL char(1) label='Randomization Flag',
       RANDFN num label='Randomization Flag (N)',
       RANDDT num format=DATE9. label='Date of Randomizaiton',
       SEXN num label='Sex (N)'
    );
    create index SUBJID on NEWDM(SUBJID);
QUIT;
```

**Log File** (for example 5-2 part 2)

```
NOTE: Table WORK.NEWDM created, with 0 rows and 10 columns.
create index SUBJID on newdm(SUBJID);
NOTE: Simple index SUBJID has been defined.
```

Example 5-3 shows how to create a table from a query result. Note that SAS options KEEP or DROP can be used with SQL to specify a subset of columns to be copied/dropped from the existing table. Syntax *arm:* indicates all columns whose names start with arm (e.g. *arm* and *armn*) will be dropped.

Code Example 5-3 (creating table from a query result)

```
PROC SQL;
    create table dmmale as
           select *
           from dm (drop=arm:)
           where sex='M'
    ;
QUIT;
```

Notes: in example 5-1, using the key word LIKE to generate a table with zero rows and keep the same structure of an existing table.

## 6. INSERT ROWS INTO A TABLE

We looked at ways to create new tables (including empty tables), now let's take a look at how to insert rows into an existing table. There are three ways in PROC SQL to insert rows into a table (the table can be empty), and the three ways are SET clause, VALUES clause, and to insert using query results. You are going to see how each one of these three methods can be used in the following examples.

Code Example 6-1 (inserting with SET clause)

```
PROC SQL;
    INSERT INTO stat2
           SET name='Ave', seq='B1', n=5
           SET name='Ave', seq='B2', n=9
    ;
QUIT;
```

Code Example 6-2 (inserting with VALUES clause)

```
PROC SQL;
    INSERT INTO stat2 (name, seq, n)
           VALUES('Ave','B1',5)
           VALUES('Ave','B2',9)
    ;
QUIT;
```

Code Example 6-3 (inserting with query results)

```
PROC SQL;
    INSERT INTO stat2 (name, seq, n)
           SELECT name, seq, n
           FROM stat1
           WHERE n >= 5
    ;
QUIT;
```

Methods in section 5 and 6 can be combined to generate the table you want with the desired structure and contents.

## 7. ASSIGNING MACRO VARIABLES

SAS SQL is one of the most popular ways to assign macro variables. The advantage is that you can not only assign a value of a variable to a macro variable, you can also assign a calculated value to a macro variable; you can not only assign to a single macro variable, but also can assign a set of macro variables in one step; you can even choose to assign a string value to a macro variable.

In example 7-1, it calculates randomized patient in table *DM* and assigns the value to macro variable *allcnt*. In this example, only one macro variable *allcnt* is created.

Code Example 7-1 (assign calculated value to a macro variable)

```
PROC SQL noprint;
    select count(distinct subjid) into :allcnt
    from dm
    where randfl='Y'
    ;
QUIT;
```

Example 7-2 calculates how many patients are in each treatment group and assigns the calculated values to up to nine macro variables, named *TRT1, TRT2…TRT9*. Macro variables are assigned dynamically depending on how many treatment groups are in the table. For example, *EX* table only has three treatment groups, therefore only *TRT1, TRT2*, and *TRT3* are assigned. The %put _user_ shows in the log file that only macro variables *TRT1*, *TRT2*, and *TRT3* are defined and assigned. This method lets you catch up to nine treatment groups information without having to know exactly how many treatment groups are in the study beforehand.

Code Example 7-2 (assign calculated values to a set of macro variables)

```
PROC SQL noprint;
    select count(distinct subjid) into :trt1 - :trt9
    from ex
    group by extrt
    ;
    %put _user_ ;
QUIT;
```

**Log File** (for example 7-2)

```
GLOBAL TRT1 65
GLOBAL TRT2 64
GLOBAL TRT3 66
```

An alternative way of doing the same task is to divide the entire process into two steps. Step 1, you find out how many treatment groups exist in ex and assign that value to macro variable *trtcnt*. Step 2, use *trtcnt* to assign the exact number of treatment groups.

Code Example 7-3 (determine and assign macro variables in two steps)

```
PROC SQL noprint;
    select count(distinct extrt) into :trtcnt
    from ex
    ;
    %put trtcnt : &trtcnt  ;

    select count(distinct subjid) into :trt1 - :trt%eval(&trtcnt)
    from ex
    group by extrt
    ;
    %put treatment group patient count: &trt1 &trt2 &trt3;
QUIT;
```

**Log File** (for example 7-3)

```
%put trtcnt : &trtcnt  ;
trtcnt :        3
...
%put treatment list: &trt1 &trt2 &trt3;
treatment group patient count: 65 64 66
```

Example 7-4 shows how the treatment names are obtained, duplicates are removed, and the treatment names are concatenated with a space in between, and then sorted in alphabet order, and in the end, how the concatenated value is assigned to a macro variable *treatnm*. In clinical trial report, very often you need to list the treatment groups; in this case you can use *treatnm* to print out the information in log.

Code Example 7-4 (Concatenating Values in Macro Variables)

```
PROC SQL noprint;
    select distinct extrt  into :treatnm separated by ' '
    from ex
    order by extrt
    ;
    %put treatment group list: &treatnm ;
QUIT;
```

**Log File** (for example 7-4)

```
%put treatment group list: &treatnm ;
treatment group list: Placebo RG200mg RG300mg
```

Notes: By default, if SQL query starts with select clause it will generate outputs in the output window. That can be inconvenient if the output window pop out every time when you just want to define macro variables. Option NOPRINT in PROC SQL statement compresses unnecessary display of the query result in the output window.

## 8. SQL AUTOMATIC CREATED MACRO VARIABLES

Besides dictionary table, PROC SQL also creates automatic macro variables. SQL creates automatic macro variables after each SQL statement. Table 2 lists a number of SQL automatic macro variables that are very useful.

| SQL automatic macro variable | contents | Usage example |
|---|---|---|
| SQLOBS | Number of rows in the newly created table | Check if a table/view is empty to decide whether to continue on to the next step. |
| SQLOOPS | Number of iterations the SQL inner loop processes | When the query becomes too complex, by define LOOPS=xx to restrict the number of iterations of the SQL processing. |
| SQLRC | Whether the SQL process was done successfully or not.<br>0-complete successfully<br>4-processed with issue<br>8-process stopped with error<br>24-processed with a system error | To determine if the SQL process was done successfully, and to check if any error has occured. |
| SQLXMSG | Message for the DBMS pass-through facility. | To determine if the SQL pass-through facility was done successfully. |

Table 2: SQL automatic macro variables

In the following example, after each SQL statement, the automatic macro variable SQLOBS is called and the value is checked. It indicates table *AE1* has 567 rows and table *AE2* has 0 rows (an empty table). You might need to make a decision about your next step at this point based on the fact that *AE2* is empty.

Code Example 8-1 (using automatic SQL macro variables)

```
PROC SQL;
    create table ae1 as
           select subjid, aeterm, aestdt
           from ae
    ;
    %put rows in created table ae1: &SQLOBS;
    create table ae2 as
           select subjid, aeterm, aestdt
           from ae
           where missing(subjid)
           order by 1, 2, 3
    ;
    %put rows in created table ae2: &SQLOBS;
QUIT;
```

**Log File** (for example 8-1)

```
%put rows in created table ae1: &SQLOBS;
rows in created table ae1: 567
...
%put rows in created table ae2: &SQLOBS;
rows in created table ae2: 0
```

Notes: If used inside a SAS macro, you can add programming checks by using macro variable SQLOBS. If &SQLOBS=0, you can use %GOTO label statement to branch the macro processing to a specified label. As such, SAS can skip some steps in your program that might cause errors due to the existence of an empty dataset.

## 9. OPTION – FEEDBACK _METHOD

Using SQL options _METHOD and FEEDBACK, you can sort of see what is happening behind the scenes, this can help you better understand how SQL processes the data. With the FEEDBACK option turned on, the Statement "transforms to:" in log will show you the decode information of the SQL process. With the _METHOD option turned on, the SQL execution methods and sub-query execution methods chosen by SQL Optimizer are displayed in the log file.

Code Example 9-1 (options – FEEDBACK _METHOD)

```
PROC SQL feedback _method;
    create table newex as
    select *
    from vs
    where subjid in
    (     select distinct subjid
          from ae
          where upcase(aeterm)='HEMORRHAGE'
    )
    order by subjid
    ;
QUIT;
```

**Log File** (for example 9-1)

```
NOTE: Statement transforms to:
        select VS.SUBJID, VS.VISIT, VS.VSTEST, VS.VSSTRESN, VS.VSABNFL
        from WORK.VS
        where VS.SUBJID in
               ( select distinct AE.SUBJID
                   from WORK.AE
                  where UPCASE(AE.AETERM) = 'HEMORRHAGE'
               )
       order by VS.SUBJID asc;
NOTE: SQL execution methods chosen are:
      sqxcrta
          sqxsort
              sqxfil
                  sqxsrc( WORK.VS )
NOTE: SQL subquery execution methods chosen are:
          sqxsubq
              sqxunqs
                  sqxsrc( WORK.AE )
```

## 10. OPTION - COMPRESS

When you work with a large size table, you might want to find ways to save storage space. Using COMPRESS data set option can reduce the storage size dramatically especially when the table has many overabundance text columns. The way to use it is putting COMPRESS=YES in a pair of parentheses after the name of the table to be created. The trade-off here is that the compression process when storing the table, and the un-compress process when retrieving the table will both increase the CPU time. So the decision partly depends on which one is more important to you, the storage size or the CPU time. Example 10-1 compresses the storage size of table *NEWAE* by 46.15 percent.

Code Example 10-1 (option COMPRESS)

```
PROC SQL;
    create table newae (COMPRESS=YES)as
           select *
           from ae
    ;
QUIT;
```

**Log File** (for example 10-1)

```
NOTE: Compressing data set WORK.NEWAE decreased size by 46.15 percent.
      Compressed is 7 pages; un-compressed would require 13 pages.
NOTE: Table WORK.NEWAE created, with 767 rows and 7 columns.
```

## CONCLUSION

PROC SQL is an alternative approach besides SAS data step and other procedures to manipulate, query, analyze data and generate reports. In most of the cases, one SQL statement is a combination of data steps and several SAS procedures, so it makes the code shorter and easy to read if you understand the SQL well. SQL is a very powerful, flexible, and efficient tool if you master it and use it wisely.

## APPENDIX 1:

Data definition metadata table

| DOMAIN | VARIABLE | LABEL | TYPE | COMMENTS |
|---|---|---|---|---|
| COMMON | SUBJID | Subject Identifier for the Study | Char | SUBJID is common variable in each dataset |
| COMMON | VISIT | Visit Name | Char | VISIT is common variable in visit dependent dataset. eg. LB, VS, and EX. |
| AE | AETERM | Reported Term for the Adverse Event | Char | |
| AE | AEBODSYS | Body System or Organ Class of AE | Char | |
| AE | AESTDT | Start Date of Adverse Event | Num | |
| AE | AEENDT | End Date of Adverse Event | Num | |
| AE | AESER | Serious Event Flag | Char | |
| AE | AESEV | AE Severity | Char | |
| DM | ARM | Description of Planned Arm | Char | |
| DM | AGE | Age | Num | |

| | | | | |
|---|---|---|---|---|
| DM | SEX | Sex | Char | |
| DM | RACE | Race | Char | |
| DM | RANDFL | Randomization Flag | Char | |
| EX | EXTRT | Name of Actual Treatment | Char | |
| EX | EXSTDT | Date of First Exposure to Treatment | Num | |
| EX | EXENDT | Date of Last Exposure to Treatment | Num | |
| LB | LBTEST | Lab Test or Examination Name | Char | |
| LB | LBSTRESN | Numeric Result/Finding in Standard Units | Num | |
| LB | LBABNVAL | Lab Abnormal Value Flag | Char | |
| VS | VSTEST | Vital Signs Test Name | Char | |
| VS | VSSTRESN | Numeric Result/Finding in Standard Units | Num | |
| VS | VSABNVAL | Vital Sign Abnormal Value Flag | Char | |
| IVRS | RAND_NUMBER | Randomization Number | Num | |
| RANDTRT | RAND_NUM | Randomization Number | Num | |
| RANDTRT | TRTGRP | Treatment Group | Char | |
| TRTDEC | TRT_DESCRIPTION | Treatment Group Description | Char | |
| TRTDEC | TRTGRP | Treatment Group | Char | |

## REFERENCES

SAS® 9.3 SQL Procedure User's SAS® 9.3 SQL® Procedure User's Guide. Available at URL: https://support.sas.com/documentation/cdl/en/sqlproc/63043/PDF/default/sqlproc.pdf

Yindra, Chris. AN INTRODUCTION TO THE SQL® PROCEDURE. Available at URL: http://www.ats.ucla.edu/stat/sas/library/nesug99/bt082.pdf

Gusinow, Rosalind and Miscisin, Michael. An Introduction to PROC SQL® Hands-on Workshops. Available at URL: http://www2.sas.com/proceedings/sugi23/Handson/p130.pdf

## CONTACT INFORMATION

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

Name: Jessica Jun Wang
Enterprise: Regeneron Pharmaceuticals Inc.
Address: 110 Allen Road
City, State ZIP: Basking Ridge, NJ 07920
Email: Jun.Wang@Regeneron.com
Work Phone: 914-847-7355
Fax: 914-847-7500

## DOCUMENT PROPERTIES

Title: Using the Power of SAS SQL
Author: Jessica Wang, Regeneron Pharmaceuticals Inc
Subject: **SAS SQL is a powerful tool for data manipulation and query. This paper is intended for intermediate to advanced SAS SQL users who already know the basics of SAS/SQL, and want to better exploit the power that SQL offers.**
Keywords: ***SAS, SAS*** SQL, PROC SQL, **SQL IN-LINE VIEW, SET OPERATION, SQL JOIN, DICTIONARY TABLE**