PhUSE 2006

Paper SS05

An Introduction to PROC SQL

Anthony Cooper, GlaxoSmithKline, Harlow, UK

ABSTRACT

PROC SQL is a powerful tool for manipulating data within SAS which can be used as an alternative to conventional SAS DATA and PROC steps for tasks such as querying and combining SAS data sets and creating reports. The purpose of this paper is to give an introduction to PROC SQL to users with no or limited previous experience of the procedure. It contains details of how to perform basic queries, combine data sets and create new data sets using PROC SQL. Examples of performing these tasks are included, along with comparisons to performing the same tasks using SAS DATA step programming, and some hints and tips.

INTRODUCTION

PROC SQL is the SAS implementation of Structured Query Language (SQL). SQL is a commonly used, standard tool that is used to create, modify, retrieve and manipulate data from tables and views in database management systems (DBMS), e.g. Oracle, Microsoft Access.

Within SAS, PROC SQL can be used not only with DBMS but also SAS data sets and views. PROC SQL can be used to perform the following tasks, which are covered in this paper:

- Querying SAS data sets
- Grouping and summarising data
- Creating reports
- Creating SAS macro variables
- Combining SAS data sets
- Creating SAS data sets
- Modifying SAS data sets

The following tasks are outside the scope of this paper, more information on them can be found in the SAS Guide to the SQL Procedure:

- Creating and Maintaining Views
- Creating and Maintaining Indexes
- Interacting with DBMS

NB the terms table, row and column in SQL programming are analogous to data set, observation and variable in DATA step programming. Both terminologies are used interchangeably throughout this paper.

The data sets used in the examples can be found in Appendix 1.

PERFORMING BASIC QUERIES

INTRODUCTION

The SELECT statement is used to query tables in SQL. To perform a basic query, the following syntax can be used:

```
proc sql options;
  select column(s)
  from table-name
  where condition
  group by column(s)
  order by column(s)
  ;
quit;
```

The SELECT statement is used to specify which columns are retrieved from the table specified in the FROM clause. It can also be used to choose data that meet certain conditions using the WHERE clause, group data together using the GROUP BY clause (e.g. for producing summary statistics) and sort the data using the ORDER BY clause. These ideas and others are explored in the following sections.

SPECIFYING COLUMNS TO DISPLAY

As a minimum, a SELECT statement with a FROM clause is required to perform a basic query. An asterisk can be used to select all the columns from the table. For example, the following code can be used to display all the data in the DEMOG table (SAS output shown underneath):

```
proc sql;
  select *
  from demog
  ;
quit;
```

```
                          Demog
 Patient  Treatment   collection             Date of
      ID  ID                date  Gender       birth
---------------------------------------------------
       1  Active       03SEP2002  M        05JUL1935
       2  Active       03SEP2002  F        16APR1940
       3  Placebo      03SEP2002  M        27JAN1924
       4  Placebo      10SEP2002  F        12JUL1941
       5  Active       04SEP2002  M        26JAN1933
       6  Placebo      05SEP2002  F        03AUG1936
       7  Active       18SEP2002  M        13DEC1936
```

To view the expanded SELECT statement in the log, use the FEEDBACK option on the PROC SQL statement.

Particular columns can be requested by specifying the column names in the SELECT statement, separating them by commas (this is a convention in SQL):

```
proc sql;
  select patient, gender, dob
  from demog
  ;
```

```
 Patient            Date of
      ID  Gender      birth
---------------------------
       1  M       05JUL1935
       2  F       16APR1940
       3  M       27JAN1924
       4  F       12JUL1941
       5  M       26JAN1933
       6  F       03AUG1936
       7  M       13DEC1936
```

It is not necessary to repeat the PROC SQL statement before each query, it only needs to be repeated if a DATA step or another procedure is executed between queries. Each query is processed individually and a RUN statement is not needed. The results of each query are automatically printed (unless the NOPRINT option is in effect). The end of each query is denoted by a semi-colon and the QUIT statement is used to terminate the procedure.

SPECIFYING ROWS TO DISPLAY

The WHERE clause is used to subset the data. Common comparison operators (EQ, LT, GT, IN etc.) and SAS functions can be used in the WHERE clause and multiple expressions can be specified using logical operators (OR, AND, NOT etc.). For example, suppose we wish to display demography information for patients in the Active treatment group:

```
proc sql;
  select patient, gender, dob
  from demog
  where treat='Active'
  ;
```

```
 Patient            Date of
      ID  Gender      birth
---------------------------
       1  M       05JUL1935
       2  F       16APR1940
       5  M       26JAN1933
       7  M       13DEC1936
```

In addition, the following special operators can be used in the WHERE clause:

- CONTAINS or ? selects rows that include the specified substring, e.g. `select prefterm from adverse where prefterm ? 'ache'`
- IS NULL or IS MISSING selects rows where the value of the column is missing, e.g. `select patient, visitid from vs where diastolic is missing`
- BETWEEN-AND selects rows where the value of the column falls within a range of values, e.g. `select * from adverse where aestart between '16SEP2002'd and '23SEP2002'd`
- LIKE selects rows by comparing character values to specified patterns. A percent sign (%) replaces any number of characters and an underscore (_) replaces one character, e.g. `select prefterm from adverse where prefterm like 'A%'`
- =* (the sounds like operator) selects rows that contain a spelling variation of the word specified, e.g. `select visitid from vs where visitid=*'Weak';`

By default, a query returns all rows in the table where the condition is true. So, in the example of the CONTAINS operator above, the following output is produced:

```
AE preferred term
----------------------------------------
Toothache
Headache
Headache
Headache
Headache
```

The DISTINCT keyword is used to eliminate duplicates from the output:

```
proc sql;
  select distinct prefterm
  from adverse
  where prefterm ? 'ache'
  ;
```

```
AE preferred term
----------------------------------------
Headache
Toothache
```

If more than one column is specified in the SELECT statement, the DISTINCT keyword applies to every column and the output contains one row for each combination of values (similar to using PROC SORT with the NODUPKEY option).

CALCULATING A NEW COLUMN

The following code shows how a new column, in this case the adverse event duration, can be calculated from existing columns in a table:

```
proc sql;
  select patient, prefterm, aeend, aestart, (aeend - aestart + 1) as aedur
```

```
  from adverse
  ;
```

```
 Patient                                                AE end  AE start
      ID  AE preferred term                               date      date     aedur
--------------------------------------------------------------------------------
       1  Abdominal pain                                 15605     15602         4
       2  Dizziness                                      15600     15600         1
       2  Fatigue                                        15606     15606         1
       5  Toothache                                      15599     15589        11
       5  Headache                                       15599     15599         1
       6  Insomnia                                       15601     15588        14
       6  Muscle spasms                                  15615     15588        28
       6  Dizziness                                      15619     15599        21
       6  Constipation                                   15619     15603        17
       7  Headache                                       15611     15611         1
       7  Headache                                       15618     15617         2
       7  Anxiety                                        15617     15617         1
       7  Headache                                       15630     15620        11
```

The column aedur is calculated from the adverse event start and end dates in exactly the same way as in DATA step code. The AS keyword is used to name the new column, although it is not required it is good practice to use it. SAS functions can also be used to calculate a new column, e.g. `int((demogdt-dob)/365.25) as age`

Suppose that we wish to find the adverse events that lasted for more than one week. Simply adding the clause `where aedur>7` to the previous code produces the following message in the log:

```
ERROR: The following columns were not found in the contributing tables: aedur.
```

This is because the WHERE clause is evaluated first, i.e. columns in the WHERE clause must exist in the table or be derived from existing columns. One solution is to repeat the calculation in the WHERE clause, however the CALCULATED keyword can be used to reference derived columns:

```
proc sql;
  select patient, prefterm, aestart, aeend, (aeend - aestart + 1) as aedur
  from adverse
  where calculated aedur > 7
  ;
```

```
 Patient                                              AE start    AE end
      ID  AE preferred term                               date      date     aedur
--------------------------------------------------------------------------------
       5  Toothache                                      15589     15599        11
       6  Insomnia                                       15588     15601        14
       6  Muscle spasms                                  15588     15615        28
       6  Dizziness                                      15599     15619        21
       6  Constipation                                   15603     15619        17
       7  Headache                                       15620     15630        11
```

The CASE expression can be used to perform conditional processing. For example, the following code could be used to determine the age of each patient in the DEMOG table and assign them to an age category:

```
proc sql;
  select patient, demogdt, dob,
    int((demogdt-dob)/365.25) as age,
    case
      when calculated age < 65 then '<65'
      when calculated age < 75 then '65-74'
      else '>=75'
    end as agegroup
  from demog
  ;
```

Patient ID	Demog collection date	Date of birth	age	agegroup
1	03SEP2002	05JUL1935	67	65-74
2	03SEP2002	16APR1940	62	<65
3	03SEP2002	27JAN1924	78	>=75
4	10SEP2002	12JUL1941	61	<65
5	04SEP2002	26JAN1933	69	65-74
6	05SEP2002	03AUG1936	66	65-74
7	18SEP2002	13DEC1936	65	65-74

FORMATTING AND ORDERING RESULTS

In the example of calculating adverse event duration, the AE start and end date columns were displayed as numbers rather than as dates and the AE duration column did not have a meaningful label. The appearance of columns can be modified by using the FORMAT= and LABEL= options after the column name (NB PROC SQL automatically uses any existing labels and formats in the table):

```
proc sql;
  select patient, prefterm, aestart format=date9., aeend format=date9.,
         (aeend - aestart + 1) as aedur label='AE duration'
  from adverse
  where calculated aedur > 7
  ;
```

Patient ID	AE preferred term	AE start date	AE end date	AE duration
5	Toothache	06SEP2002	16SEP2002	11
6	Insomnia	05SEP2002	18SEP2002	14
6	Muscle spasms	05SEP2002	02OCT2002	28
6	Dizziness	16SEP2002	06OCT2002	21
6	Constipation	20SEP2002	06OCT2002	17
7	Headache	07OCT2002	17OCT2002	11

Suppose that we now want to order the results of the previous query to show the events with the longest duration for each patient first. This is can be achieved using the ORDER BY clause, columns are sorted in ascending order unless the DESC keyword is used (NB Variables do not have to appear in the SELECT statement to be used in the ORDER BY clause):

```
proc sql;
  select patient, prefterm, aestart format=date9.,  aeend format=date9.,
         (aeend - aestart + 1) as aedur label='AE duration'
  from adverse
  where calculated aedur > 7
  order by patient, aedur desc, prefterm
  ;
```

Patient ID	AE preferred term	AE start date	AE end date	AE duration
5	Toothache	06SEP2002	16SEP2002	11
6	Muscle spasms	05SEP2002	02OCT2002	28
6	Dizziness	16SEP2002	06OCT2002	21
6	Constipation	20SEP2002	06OCT2002	17
6	Insomnia	05SEP2002	18SEP2002	14
7	Headache	07OCT2002	17OCT2002	11

Note that this report has been created in one step using PROC SQL whereas using conventional SAS DATA step programming would have required three DATA/PROC steps:

- A DATA step to calculate the variable aedur
- A PROC SORT to order the data correctly
- A PROC PRINT to display the results

SUMMARISING AND GROUPING DATA

PROC SQL has several functions for calculating summary statistics, e.g. MEAN, COUNT, MIN, MAX, SUM etc. Suppose that we wish to calculate the average systolic blood pressure at each visit:

```
proc sql;
  select visitid, mean(systolic) as sys_mean format=8.1 label='Mean Systolic BP'
  from vs
  ;
```

This query result contains one row for each observation in the VS data set (partial output shown below) along with a message in the log:

```
                            Mean
                        Systolic
Visit ID                      BP
------------------------------
Screening                  127.4
Week 1                     127.4
Week 2                     127.4
Screening                  127.4
Week 1                     127.4
Week 2                     127.4
.
.

NOTE: The query requires remerging summary statistics back with the original data.
```

The MEAN function has calculated statistics based on the whole table, and then remerged the mean value with each individual row in the table. The GROUP BY clause can be used to classify the data into groups, it works in a similar way to a BY statement:

```
proc sql;
  select visitid, mean(systolic) as sys_mean format=8.1 label='Mean Systolic BP'
  from vs
  group by visitid
  ;
```

```
                            Mean
                        Systolic
Visit ID                      BP
------------------------------
Screening                  123.4
Week 1                     125.9
Week 2                     133.7
```

Similar code can be used to determine the number of subjects with each adverse event using the COUNT function:

```
proc sql;
  select prefterm, count (distinct patient) as count label='No. of subjects'
  from adverse
  group by prefterm
  ;
```

```
                                             No. of
AE preferred term                          subjects
---------------------------------------------------
Abdominal pain                                    1
Anxiety                                           1
Constipation                                      1
Dizziness                                         2
Fatigue                                           1
Headache                                          2
Insomnia                                          1
Muscle spasms                                     1
Toothache                                         1
```

The DISTINCT keyword is used so that each patient is only counted once for each preferred term, otherwise patient 7 would be counted 3 times for Headache.

In the same way that the WHERE clause can be used to select individual rows, the HAVING clause can be used to select entire groups of data, e.g. those adverse events experience by more than one patient:

```
proc sql;
  select prefterm, count (distinct patient) as count label='No. of subjects'
  from adverse
  group by prefterm
  having count>1
  ;
```

```
                                             No. of
AE preferred term                          subjects
---------------------------------------------------
Dizziness                                         2
Headache                                          2
```

In both of these examples, we have again seen that PROC SQL can create a report in one step which would have taken several steps using DATA/PROC steps.

SUB-QUERIES

Queries may be nested within other queries. These nested queries are commonly referred to as sub-queries or inner queries. They are normally used in a WHERE or a HAVING clause, are evaluated before the outer query, and return one or more values to be used by the outer query. Suppose we want to select the adverse events reported by male patients:

```
proc sql;
  select patient, prefterm
  from adverse
  where patient in
   (select patient from demog where gender='M')
  order by patient, prefterm
  ;
```

```
 Patient
      ID  AE preferred term
---------------------------------------------------
       1  Abdominal pain
       5  Headache
       5  Toothache
       7  Anxiety
       7  Headache
       7  Headache
       7  Headache
```

In this example, the inner query returns the list of male patients from the DEMOG table. This is then used by the outer query to select the rows from the ADVERSE table for those patients.

There are two keywords that can be used to compare a value with a set of values returned by a sub-query. If the keyword ANY is used, the comparison is true if it is true for any of the values returned by the sub-query. For example, suppose we want to find female patients born before any male patients:

```
proc sql;
  select patient
  from demog
  where gender = 'F' and dob < any
    (select dob from demog where gender='M')
  ;

 Patient
      ID
--------
       6
```

Similarly, if the keyword ALL is used, the comparison is true if it is true for all values returned by the sub-query. Repeating the previous example using the ALL keyword instead of ANY does not return any rows since there are no female patients born before all male patients.

In a correlated sub-query, the WHERE clause in the sub-query refers to values in a table in the outer query. The correlated sub-query is evaluated for each row in the outer query. PROC SQL executes the sub-query and the outer query together. Suppose we want to identify the treatment being taken by patients with an adverse event of Dizziness:

```
proc sql;
  select patient, treat
  from demog
  where 'Dizziness' in
    (select prefterm from adverse where demog.patient=adverse.patient)
  ;

 Patient  Treatment
      ID  ID
--------------------
       2  Active
       6  Placebo
```

The correlated sub-query resolves by substituting each value of PATIENT in DEMOG into the WHERE clause of the sub-query one row at a time. For example, when DEMOG.PATIENT=2 the sub query retrieves the rows from ADVERSE WHERE PATIENT=2 and passes their PREFTERM to the WHERE clause of the outer query (i.e. Dizziness and Fatigue). The outer query then checks to see if Dizziness is in the list. If it is, the values of PATIENT and TREAT from DEMOG are returned. NB since the variable PATIENT is in both tables, the table names must be specified in the WHERE clause in the sub-query.

CREATING MACRO VARIABLES

The INTO clause is used to create SAS macro variables using the SQL procedure (the NOPRINT option is used to stop the result of the query being written to the output window):

```
proc sql noprint;
  select mean(diastolic) into :dia_mean
  from vs
  ;
```

Several macro variables can be created in one query:

```
proc sql noprint;
  select mean(diastolic), std(diastolic) into :dia_mean, :dia_std
  from vs
  ;
```

NB Macro variables can be used within PROC SQL just as within normal SAS code, e.g. to select outliers from the vital signs data set using the macro variables previously created:

```
proc sql;
  select * from vs
  where diastolic > (&dia_mean + 2*&dia_std) or diastolic < (&dia_mean - 2*&dia_std)
  ;
```

COMBINING TABLES

INTRODUCTION

SQL can be used to combine data sets horizontally (referred to as join operations, akin to the MERGE statement in data step code) or vertically (referred to as set operations, akin to the SET statement in data step code).

The following types of joins can be performed:

- Inner joins (return matching rows from two or more tables)
- Outer joins (return matching rows from two tables plus non-matching rows from the left, right or both tables)

Set operators are used to combine two tables, one on top of the other. There are four types of set operators:

- EXCEPT (selects unique rows from the first table that are not found in the second table)
- INTERSECT (selects common unique rows from both tables)
- UNION (selects all unique rows from both tables)
- OUTER UNION (selects all rows from both tables)

These methods for combining tables are discussed further in the following sections.

INNER JOINS

An inner join returns rows with matching key values. It is sometimes referred to as a conventional join. Suppose we wish to merge the DEMOG and PHYSEXAM data sets by PATIENT:

```
proc sql;
  select *
  from demog, physexam
  where demog.patient=physexam.patient
  ;
```

```
                         Demog
 Patient  Treatment   collection              Date of   Patient    Height    Weight
      ID  ID                date  Gender        birth        ID      (cm)      (kg)
-----------------------------------------------------------------------------------
       1  Active       03SEP2002  M         05JUL1935         1       146      67.9
       2  Active       03SEP2002  F         16APR1940         2       168      98.4
       4  Placebo      10SEP2002  F         12JUL1941         4       150      85.3
       5  Active       04SEP2002  M         26JAN1933         5       130      56.9
       7  Active       18SEP2002  M         13DEC1936         7       179      89.4
```

Since the column PATIENT exists in both tables it is displayed twice in the output. To display the column once in the output, we must explicitly refer to the variable PATIENT in one of the tables in the SELECT statement, e.g. `select demog.patient, treat, demogdt, gender, dob, height, weight`. Alternatively, we can select all columns from DEMOG using the asterisk syntax and HEIGHT and WEIGHT from VS:

```
proc sql;
  select demog.*, height, weight
  from demog, physexam
  where demog.patient=physexam.patient
  ;
```

```
                      Demog
 Patient  Treatment   collection            Date of    Height    Weight
      ID  ID                date  Gender      birth      (cm)      (kg)
-----------------------------------------------------------------------
       1  Active       03SEP2002  M       05JUL1935       146      67.9
       2  Active       03SEP2002  F       16APR1940       168      98.4
       4  Placebo      10SEP2002  F       12JUL1941       150      85.3
       5  Active       04SEP2002  M       26JAN1933       130      56.9
       7  Active       18SEP2002  M       13DEC1936       179      89.4
```

To save having to type out the full table name in the SELECT statement and WHERE clause, we can define aliases for the tables on the FROM clause. These aliases can either be given arbitrary names (e.g. A and B) or more meaningful names (e.g. three letter abbreviations of the full table name):

```
proc sql;
  select dem.*, height, weight
  from demog dem, physexam pex
  where dem.patient=pex.patient
  ;
```

To perform the equivalent join or merge using SAS data step programming, the following code would be used:

```
data demog_ij;
  merge demog (in=a) physexam (in=b);
  by patient;
  if a and b;
run;
```

This assumes that the data sets are both sorted by PATIENT. However, PROC SQL does not require tables to be sorted before they are joined. This is another advantage of using PROC SQL instead of DATA step programming. In terms of efficiency, conventional DATA step programming is generally more efficient than using PROC SQL when joining small tables. The opposite is true when joining large tables. Several papers have been written which discuss the relative efficiency of DATA step and PROC SQL methods, see Further Reading for more information.

If the WHERE clause is omitted, all combinations of rows from all tables are matched (partial output shown):

```
proc sql;
  select dem.*, height, weight
  from demog dem, physexam pex
  ;
```

```
                      Demog
 Patient  Treatment   collection            Date of    Height    Weight
      ID  ID                date  Gender      birth      (cm)      (kg)
-----------------------------------------------------------------------
       1  Active       03SEP2002  M       05JUL1935       146      67.9
       2  Active       03SEP2002  F       16APR1940       146      67.9
       3  Placebo      03SEP2002  M       27JAN1924       146      67.9
       4  Placebo      10SEP2002  F       12JUL1941       146      67.9
       5  Active       04SEP2002  M       26JAN1933       146      67.9
       6  Placebo      05SEP2002  F       03AUG1936       146      67.9
       7  Active       18SEP2002  M       13DEC1936       146      67.9
       1  Active       03SEP2002  M       05JUL1935       168      98.4
       2  Active       03SEP2002  F       16APR1940       168      98.4
       .
       .
```

When performing a join using PROC SQL, SAS creates an intermediate table internally which contains all combinations of rows from all tables. This intermediate table is referred to as a Cartesian product. The number of rows it contains is the product of the number of rows in each table in the join. The intermediate table then becomes the input to the rest of the query, if a WHERE clause is present the matching rows are selected from the Cartesian product. To improve query performance, SAS reduces the size of the Cartesian product via an optimising algorithm.

Joins and subqueries can often be used to achieve the same result. For example, we can identify the patients with an adverse event of Dizziness using the following inner join:

```
proc sql;
  select adv.patient, treat
  from demog dem, adverse adv
  where dem.patient=adv.patient and prefterm='Dizziness'
  ;
```

OUTER JOINS

An outer join returns matching rows as well as non-matching rows. The three types of outer join are shown below.

In a left join, all matching rows are returned along with unmatched rows from the left table (the first table in the FROM clause):

```
proc sql;
  select dem.*, height, weight
  from demog dem left join physexam pex
  on dem.patient=pex.patient
  ;
```

```
                         Demog
 Patient  Treatment   collection              Date of    Height    Weight
      ID  ID                date  Gender        birth      (cm)      (kg)
-----------------------------------------------------------------------
       1  Active       03SEP2002  M         05JUL1935       146      67.9
       2  Active       03SEP2002  F         16APR1940       168      98.4
       3  Placebo      03SEP2002  M         27JAN1924         .         .
       4  Placebo      10SEP2002  F         12JUL1941       150      85.3
       5  Active       04SEP2002  M         26JAN1933       130      56.9
       6  Placebo      05SEP2002  F         03AUG1936         .         .
       7  Active       18SEP2002  M         13DEC1936       179      89.4
```

The left join is equivalent to the following in SAS DATA step code:

```
data demog_lj;
  merge demog (in=a) physexam (in=b);
  by patient;
  if a;
run;
```

In a right join, all matching rows are returned along with unmatched rows from the right table (the second table in the FROM clause):

```
proc sql;
  select pex.patient, treat, demogdt, gender, dob, height, weight
  from demog dem right join physexam pex
  on dem.patient=pex.patient
  ;
```

```
                         Demog
 Patient  Treatment   collection              Date of    Height    Weight
      ID  ID                date  Gender        birth      (cm)      (kg)
-----------------------------------------------------------------------
       1  Active       03SEP2002  M         05JUL1935       146      67.9
       2  Active       03SEP2002  F         16APR1940       168      98.4
       4  Placebo      10SEP2002  F         12JUL1941       150      85.3
       5  Active       04SEP2002  M         26JAN1933       130      56.9
       7  Active       18SEP2002  M         13DEC1936       179      89.4
       8                       .                    .       121      56.9
       9                       .                    .       147      67.5
```

The right join is equivalent to the following in SAS DATA step code:

```
data demog_rj;
  merge demog (in=a) physexam (in=b);
  by patient;
  if b;
run;
```

In a full join, all matching rows are returned along with unmatched rows from both tables

```
proc sql;
  select *
  from demog dem full join physexam pex
  on dem.patient=pex.patient
  ;
```

Patient ID	Treatment ID	Demog collection date	Gender	Date of birth	Patient ID	Height (cm)	Weight (kg)
1	Active	03SEP2002	M	05JUL1935	1	146	67.9
2	Active	03SEP2002	F	16APR1940	2	168	98.4
3	Placebo	03SEP2002	M	27JAN1924	.	.	.
4	Placebo	10SEP2002	F	12JUL1941	4	150	85.3
5	Active	04SEP2002	M	26JAN1933	5	130	56.9
6	Placebo	05SEP2002	F	03AUG1936	.	.	.
7	Active	18SEP2002	M	13DEC1936	7	179	89.4
.		.		.	8	121	56.9
.		.		.	9	147	67.5

Since PATIENT is not populated for all rows in both tables, it would be useful to overlay the values from the PATIENT columns. This can be achieved using the COALESCE function:

```
proc sql;
  select coalesce(dem.patient,pex.patient) label='Patient ID', treat, demogdt,
    gender, dob, height, weight
  from demog dem full join physexam pex
  on dem.patient=pex.patient
  ;
```

Patient ID	Treatment ID	Demog collection date	Gender	Date of birth	Height (cm)	Weight (kg)
1	Active	03SEP2002	M	05JUL1935	146	67.9
2	Active	03SEP2002	F	16APR1940	168	98.4
3	Placebo	03SEP2002	M	27JAN1924	.	.
4	Placebo	10SEP2002	F	12JUL1941	150	85.3
5	Active	04SEP2002	M	26JAN1933	130	56.9
6	Placebo	05SEP2002	F	03AUG1936	.	.
7	Active	18SEP2002	M	13DEC1936	179	89.4
8		.		.	121	56.9
9		.		.	147	67.5

The full join is equivalent to the following in SAS DATA step code:

```
data demog_fj;
  merge demog physexam;
  by patient;
run;
```

NB in outer joins the ON clause replaces the WHERE clause. An inner join can also be expressed in this way:

```
proc sql;
  select dem.*, height, weight
  from demog dem inner join physexam pex
  on dem.patient=pex.patient
  ;
```

When using the DATA step MERGE statement to combine data sets there must be matching BY variables on the data sets. However, in PROC SQL, the columns used to perform the join do not need to have the same name and joins can be performed using more complex expressions. For example, the following code determines the dose of study medication taken on the day the adverse event starts:

```
proc sql;
  select patient, prefterm, aestart format=date9., aeend format=date9.,
    dose as aedose label='Dose at AE onset'
  from adverse adv left join dose dos
  on adv.patient=dos.patid and dos.dosstart<=adv.aestart<=dos.dosend
  ;
```

```
 Patient                                               AE start     AE end   Dose at
      ID  AE preferred term                                date       date  AE onset
-------------------------------------------------------------------------------------
       1  Abdominal pain                              19SEP2002  22SEP2002       200
       2  Dizziness                                   17SEP2002  17SEP2002       100
       2  Fatigue                                     23SEP2002  23SEP2002       100
       5  Headache                                    16SEP2002  16SEP2002       100
       5  Toothache                                   06SEP2002  16SEP2002         .
       6  Constipation                                20SEP2002  06OCT2002       200
       6  Dizziness                                   16SEP2002  06OCT2002       100
       6  Muscle spasms                               05SEP2002  02OCT2002         .
       6  Insomnia                                    05SEP2002  18SEP2002         .
       7  Headache                                    07OCT2002  17OCT2002       300
       7  Anxiety                                     04OCT2002  04OCT2002       300
       7  Headache                                    04OCT2002  05OCT2002       300
       7  Headache                                    28SEP2002  28SEP2002       100
```

To program this using conventional DATA step code would require the DOSE data set to be expanded to one observation per patient identifier, dose and date which could then be merged with the ADVERSE data set by patient identifier and date (these variables would have to be renamed to have the same names on both data sets).

SET OPERATORS

The EXCEPT operator selects unique rows from the first table that are not found in the second table. For example, suppose we wish to determine patients who are in the DEMOG data set but not in the PHYSEXAM data set:

```
proc sql;
  select patient
  from demog
  except
  select patient
  from physexam
  ;
```

```
 Patient
      ID
--------
       3
       6
```

The INTERSECT operator selects common unique rows from both tables. Suppose we want to find subjects who are in both the DEMOG and PHYSEXAM data sets:

```
proc sql;
  select patient
  from demog
  intersect
  select patient
  from physexam
  ;


 Patient
      ID
--------
       1
       2
       4
       5
       7
```

The UNION operator selects all unique rows from both tables. Suppose we want to find subjects who are in either the DEMOG or the PHYSEXAM data set:

```
proc sql;
  select patient
  from demog
  union
  select patient
  from physexam
  ;

 Patient
      ID
--------
       1
       2
       3
       4
       5
       6
       7
       8
       9
```

With the EXCEPT, INTERSECT and UNION set operators, the columns in the output are determined by the first table and duplicate rows are automatically deleted from the output. To keep duplicate rows, the ALL keyword can be used. It also improves the performance of the query since one less pass of the data is required (output not shown):

```
proc sql;
  select patient
  from demog
  union all
  select patient
  from physexam
  ;
```

The OUTER UNION operator selects all rows from both tables and is very similar to using a SET statement in SAS data step programming to concatenate data sets.

```
proc sql;
  select *
  from demog
  outer union
  select *
  from physexam
  ;
```

```
                       Demog
 Patient  Treatment   collection              Date of   Patient    Height    Weight
      ID  ID                date  Gender        birth        ID      (cm)      (kg)
-------------------------------------------------------------------------------------
       1  Active       03SEP2002  M       05JUL1935         .         .         .
       2  Active       03SEP2002  F       16APR1940         .         .         .
       3  Placebo      03SEP2002  M       27JAN1924         .         .         .
       4  Placebo      10SEP2002  F       12JUL1941         .         .         .
       5  Active       04SEP2002  M       26JAN1933         .         .         .
       6  Placebo      05SEP2002  F       03AUG1936         .         .         .
       7  Active       18SEP2002  M       13DEC1936         .         .         .
       .                       .                  .         1       146      67.9
       .                       .                  .         2       168      98.4
       .                       .                  .         4       150      85.3
       .                       .                  .         5       130      56.9
       .                       .                  .         7       179      89.4
       .                       .                  .         8       121      56.9
       .                       .                  .         9       147      67.5
```

The ALL keyword is not used with OUTER UNION since the default is to include all rows in the result.

Columns with the same name can be overlaid by using the CORRESPONDING keyword, or CORR for short:

```
proc sql;
  select *
  from demog
  outer union corr
  select *
  from physexam
  ;
```

```
                       Demog
 Patient  Treatment   collection              Date of    Height    Weight
      ID  ID                date  Gender        birth      (cm)      (kg)
---------------------------------------------------------------------------
       1  Active       03SEP2002  M       05JUL1935         .         .
       2  Active       03SEP2002  F       16APR1940         .         .
       3  Placebo      03SEP2002  M       27JAN1924         .         .
       4  Placebo      10SEP2002  F       12JUL1941         .         .
       5  Active       04SEP2002  M       26JAN1933         .         .
       6  Placebo      05SEP2002  F       03AUG1936         .         .
       7  Active       18SEP2002  M       13DEC1936         .         .
       1                       .                  .       146      67.9
       2                       .                  .       168      98.4
       4                       .                  .       150      85.3
       5                       .                  .       130      56.9
       7                       .                  .       179      89.4
       8                       .                  .       121      56.9
       9                       .                  .       147      67.5
```

The CORR keyword causes PROC SQL to match the columns by name rather than by the order in which they appear. When used with the EXCEPT, INTERSECT and UNION set operators, CORR suppresses columns that are not found in both tables.

To perform a simple concatenation of data sets, the DATA step SET statement is more efficient than PROC SQL set operators (PROC APPEND is the fastest method). If further processing is required then either method can be used. As with join operations, it may be possible to perform set operations in fewer steps by using PROC SQL.

CREATING AND MODIFYING TABLES

INTRODUCTION

So far, we have used PROC SQL to perform queries and combine tables. In SAS data step programming, we normally want to create new data sets from existing data sets (e.g. using the MERGE or SET statements) and/or create new variables from existing variables. PROC SQL provides functionality to create and modify tables. These topics are discussed in the following sections.

CREATING TABLES

The CREATE TABLE statement is used to create new tables in PROC SQL. This can be done in one of three ways:

- By defining the table's columns from scratch, then filling the table with data
- By defining the table's columns based on the column definitions of another table, then filling the table with data
- By defining the table's columns and rows based on the result of a query.

The following syntax can be used to create a table from scratch:

```
proc sql;
  create table table-name(
    var-name type(length) label format,
    var-name type(length) label format,
    .
    .
  );
```

The column name and type are required, width, format, informat and label are optional. For example, to create the DEMOG data set used in this paper:

```
proc sql;
  create table demog(
  patient num label='Patient ID',
  treat char(10) label='Treatment ID',
  demogdt num label='Demog collection date' format=date9.,
  gender char(1) label='Gender',
  dob num label='Date of birth' format=date9.)
  ;
```

The following syntax can be used to create a table based upon the column definitions of another table:

```
proc sql;
  create table table-name
  like table-name
  ;
```

SAS data set options (e.g. DROP=, KEEP=, RENAME=) can be used with either table to modify the columns created. Suppose that there is a permanent DEMOG data set that we wish to base our DEMOG data set on:

```
proc sql;
  create table demog (drop=initials)
  like master.demog
  ;
```

Note that in both of these examples, the table created does not yet contain any data. To enter rows of data, the INSERT statement is used. Either a VALUES clause or a SET clause can be used:

```
proc sql;
  insert into demog
    values(1,'Active','03SEP002'd,'M','05JUL1935'd)
    values(2,'Active','03SEP002'd,'F','16APR1940'd)
    ;
  insert into demog
    set patient=3,treat='Placebo',demogdt='03SEP002'd,gender='M',dob='27JAN1924'd
    set patient=4,treat='Placebo',demogdt='10SEP2002'd,gender='F',dob='12JUL1941'd
    ;
```

With the VALUES clause values are matched positionally, with the SET clause values are matched by their name.

To store the results of a query as a new table the AS keyword is used with the CREATE TABLE statement. The following code creates a new table which contains demography and physical examination data for female patients (PROC PRINT of data set created shown beneath):

```
proc sql;
  create table females as
  select dem.*, height, weight
  from demog dem, physexam pex
  where dem.patient=pex.patient and gender='F'
  ;
```

Patient ID	Treatment ID	Demog collection date	Gender	Date of birth	Height (cm)	Weight (kg)
2	Active	03SEP2002	F	16APR1940	168	98.4
4	Placebo	10SEP2002	F	12JUL1941	150	85.3

The order of the columns in the resulting data set is determined by the order they appear in the SELECT statement, therefore PROC SQL provides an alternative method for re-ordering variables in a data set.

MODIFYING TABLES

PROC SQL can be used to modify a table in the following ways:

- Alter the columns in a table
- Update the values in a table
- Delete rows from a table

The ALTER TABLE statement is used to modify the columns in a table. The syntax is as follows:

```
proc sql;
  alter table table-name
  add column(s)
  modify column(s)
  drop column(s)
  ;
```

The ADD clause adds new columns to the table, the MODIFY clause updates the attributes of existing columns in the table, the DROP clause drops columns from the table. For example, suppose we want to add a new column AGE to the FEMALES table created in the previous example, update the label for HEIGHT and drop the column GENDER:

```
proc sql;
  alter table females
  add age num format=8.0 label='Age'
  modify height label='Height (in)' format=8.1
  drop gender
  ;
```

Patient ID	Treatment ID	Demog collection date	Date of birth	Height (in)	Weight (kg)	Age
2	Active	03SEP2002	16APR1940	168.4	98.4	.
4	Placebo	10SEP2002	12JUL1941	149.6	85.3	.

The UPDATE statement with the SET clause can be used to modify the values of columns in existing rows of the table. For example, it can be used to convert the HEIGHT column to inches and to populate the AGE column:

```
proc sql;
  update females
  set height=height/2.54, age=int(demogdt-dob)/365.25
  ;
```

Patient ID	Treatment ID	Demog collection date	Date of birth	Height (in)	Weight (kg)	Age
2	Active	03SEP2002	16APR1940	66.3	98.4	62
4	Placebo	10SEP2002	12JUL1941	58.9	85.3	61

As with the SELECT statement, the WHERE clause can be used to update selected rows in the table and conditional processing can be performed using the CASE expression.

The DELETE statement can be used to remove rows from a table, e.g. to remove Placebo patients from the FEMALES table:

```
proc sql;
  delete
  from females
  where treat='Placebo'
  ;
```

Patient ID	Treatment ID	Demog collection date	Date of birth	Height (in)	Weight (kg)	Age
2	Active	03SEP2002	16APR1940	66.3	98.4	62

CONCLUSION

This paper has shown how the SAS SQL procedure can be used to perform queries, combine data sets, and create and modify tables. Whilst the examples included in this paper have been relatively simple, it should be apparent how they could be expanded to perform more complicated tasks.

In summary, PROC SQL combines the functionality of several procedures (e.g. for sorting and summarising data) and can reduce the amount of code that needs to be written when compared with conventional DATA step programming since several steps can be performed using one statement. When joining tables using PROC SQL, there is no need to sort the tables beforehand and the columns on which the join is performed do not have to have the same name.

FURTHER READING

SAS Institute Inc., SAS® Guide to the SQL Procedure: Usage and Reference, Version 6, First Edition

Gajanan Bhat, Merging Tables in DATA Step vs. PROC SQL: Convenience and Efficiency Issues, Paper 104-26, SUGI 26

Steve Feder, Comparative Efficiency of SQL and Base Code When Reading from Database Tables and Existing Data Sets, Paper 076-28, SUGI 28

CONTACT INFORMATION

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

Anthony Cooper
GlaxoSmithKline
New Frontiers Science Park
Third Avenue
Harlow
CM19 5AW
Work Phone: +44 (0) 1279 646349
Fax: +44 (0) 1279 644430
Email: Anthony.J.Cooper@gsk.com
Web: www.biometrics.com

APPENDIX 1 – DATA SETS USED IN EXAMPLES

ADVERSE

patient	prefterm	aestart	aeend	intens	intensc
1	Abdominal pain	15602	15605	1	Mild
2	Dizziness	15600	15600	1	Mild
2	Fatigue	15606	15606	2	Moderate
5	Toothache	15589	15599	1	Mild
5	Headache	15599	15599	1	Mild
6	Insomnia	15588	15601	1	Mild
6	Muscle spasms	15588	15615	1	Mild
6	Dizziness	15599	15619	1	Mild
6	Constipation	15603	15619	1	Mild
7	Headache	15611	15611	1	Mild
7	Headache	15617	15618	1	Moderate
7	Anxiety	15617	15617	1	Mild
7	Headache	15620	15630	1	Severe

DEMOG

patient	treat	demogdt	gender	dob
1	Active	03SEP2002	M	05JUL1935
2	Active	03SEP2002	F	16APR1940
3	Placebo	03SEP2002	M	27JAN1924
4	Placebo	10SEP2002	F	12JUL1941
5	Active	04SEP2002	M	26JAN1933
6	Placebo	05SEP2002	F	03AUG1936
7	Active	18SEP2002	M	13DEC1936

DOSE

patid	visitid	dosstart	dosend	dose
1	Week 1	10SEP2002	16SEP2002	100
1	Week 2	17SEP2002	23SEP2002	200
2	Week 1	10SEP2002	23SEP2002	100
3	Week 1	10SEP2002	16SEP2002	100
3	Week 2	17SEP2002	23SEP2002	300
4	Week 1	11SEP2002	24SEP2002	100
4	Week 2	25SEP2002	01OCT2002	200
5	Week 1	11SEP2002	17SEP2002	100
5	Week 2	18SEP2002	24SEP2002	200
6	Week 1	11SEP2002	17SEP2002	100
6	Week 2	18SEP2002	24SEP2002	200
7	Week 1	25SEP2002	01OCT2002	100
7	Week 2	02OCT2002	08OCT2002	300

PHYSEXAM

patient	height	weight
1	146	67.9
2	168	98.4
4	150	85.3
5	130	56.9
7	179	89.4
8	121	56.9
9	147	67.5

VS

patient	visitid	visitdt	systolic	diastolic	heartrte
1	Screening	03SEP2002	118	68	58
1	Week 1	17SEP2002	100	62	54
1	Week 2	23SEP2002	108	64	56
2	Screening	03SEP2002	150	88	76
2	Week 1	17SEP2002	183	120	96
2	Week 2	23SEP2002	110	68	58
3	Screening	03SEP2002	110	64	54
3	Week 1	17SEP2002	110	.	68
4	Screening	10SEP2002	106	68	62
4	Week 1	24SEP2002	100	60	56
4	Week 2	01OCT2002	186	54	98
5	Screening	04SEP2002	102	68	56
5	Week 1	17SEP2002	100	60	76
5	Week 2	24SEP2002	98	62	66
6	Screening	05SEP2002	140	82	72
6	Week 1	17SEP2002	150	80	76
6	Week 2	24SEP2002	160	62	66
7	Screening	18SEP2002	138	70	68
7	Week 1	01OCT2002	138	70	68
7	Week 2	08OCT2002	140	72	68