PROC SQL for reporting and data management

- Introduction
- Sample data
- Syntax
- Examples

Introduction

SQL stands for **S**tructured **Q**uery **L**anguage and is pronounced either S-Q-L or *sequel*.

For many decades, SQL has been used as a language for querying relational databases constructed with software such as Oracle and SQL Server.

A SAS procedure for using SQL within SAS was introduced in the 1980's. This allows people who know SQL to apply their knowledge within SAS.

What is a relational database? [per Yindra] A relational database is a set of tables (think of them like data sets) that are conceptually related and are logically related by a key column or columns. Data within a relational database are typically *normalized* – that is, each piece of data exists in only one place. This makes them efficient for storage but not great for analysis.

What is a query? [per Yindra] A query is a request for information from a table or tables.

When we use PROC SQL within SAS, we can use it either to query relational databases (such as Oracle databases) to make their data available to SAS, or we can use it with SAS data sets, treating them as if they were tables in the SQL sense. In this class, we will use PROC SQL with SAS data sets. You will see that it is an alternative to other data management and reporting tools in SAS, with strengths and weaknesses relative to those other tools.

Terminology:	Data processing	SAS	SQL
	File	SAS data set	Table
	Record	Observation	Row
	Field	Variable	Column

In the remainder of this chapter, column and variable will be used interchangeably, as will record, observation, and row.

Sample Data

The collection of data sets that we will treat as related tables in this chapter are simulated data on patients, their hospital visits, the hospitals, and the doctors (with thanks to Christianna Williams for the basic pattern of most of these data sets). Data set listings:

PATIENTS data set

Obs	ID	Sex	PrimMD	BirthDate	LastName	FirstName
1	001	1	1972	10AUG1941	McGiver	Perry
2	002	2	1972	17MAR1939	Kost	Mary
3	003	1	.	02JUL1929	Thompkins	Felicio
4	004	1	4003	25MAY1926	Collins	Jeffrey
5	005	2	1972	31AUG1941	Ray	Linda
6	006	1	2322	12APR1909	Tolliver	Sidney
7	007	1	3274	07FEB1910	Coppersmith	Eugenio
8	008	2	4003	09NOV1945	Hallman	Marybeth
9	009	2	3274	15SEP1919	Kramer	Carmella
10	010	2	2322	15OCT1949	Kohl	Melissa
11	011	2	1972	04NOV1927	Ou	Ping
12	012	1	7803	16JUN1936	Dahlgren	Elisabeth
13	013	1	4003	03AUG1947	Whybel	Sander
14	014	2	8034	14DEC1942	Blue	Sally
15	015	2	3274	.	Hostetler	Lisa
16	016	1	1971	17JUN1914	DeMarcus	Frederick
17	017	1	2322	17APR1932	Hollings	David
18	018	1	1972	13NOV1948	Pelletier	Crosby
19	019	2	4003	01FEB1934	Fung	June
20	020	2	7803	07AUG1916	Newman	Marjolane

ADMITS data set (hospital admissions)

Obs	PT_ID	AdmDate	DisDate	MD	Hosp	Dest	BP_Sys	BP_Dia	PrimDX	HospCharge
1	001	07FEB2007	08FEB2007	3274	1	1	188	85	410.0	$1,003.22
2	001	12APR2007	25APR2007	1972	1	1	230	101	428.2	$12,017.83
3	001	10SEP2007	19SEP2007	3274	2	2	170	78	813.90	$8,858.21
4	001	06JUN2008	12JUN2008	3274	2	9	185	94	428.4	$6,057.30
5	003	15MAR2007	15MAR2007	2322	3	9	74	40	431	$502.19
6	004	18JUN2007	24JUN2007	7803	2	2	140	78	434.1	$6,901.96
7	005	19JAN2007	22JAN2007	1972	1	1	148	84	411.81	$4,066.27
8	005	10MAR2007	18MAR2007	1972	1	1	160	90	410.9	$9,231.50
9	005	10APR2007	14APR2007	1972	2	1	150	89	411.0	$5,539.29
10	007	28JUL2007	10AUG2007	3274	2	2	136	72	155.0	$13,902.13
11	007	08SEP2007	15SEP2007	3274	2	2	138	71	155.0	$8,329.18
12	008	01OCT2007	15OCT2007	3274	3	1	145	74	820.8	$20,709.55
13	008	26NOV2007	28NOV2007	2322	3	2	135	76	V54.8	$1,904.47
14	008	10DEC2007	12DEC2007	2322	9	2	132	78	V54.8	$2,322.51
15	009	15DEC2007	04JAN2008	1972	2	9	228	92	410.1	$45,338.26
16	010	30NOV2008	01DEC2008	2322	9	2	147	84	E886.3	$3,101.45
17	012	12AUG2007	16AUG2007	4003	5	1	187	106	410.52	$5,532.95
18	014	17JAN2008	20JAN2008	7803	3	1	162	93	414.10	$6,022.64
19	015	25MAY2008	06JUN2008	4003	5	2	142	81	820.8	$10,993.20
20	015	17AUG2008	24AUG2008	4003	5	2	138	79	038.2	$8,834.01
21	016	25JUL2008	30JUL2008	7803	2	1	189	101	412.1	$6,609.49
22	018	01NOV2007	15NOV2007	1972	3	2	170	88	428.1	$15,504.32
23	018	26DEC2007	08JAN2008	1972	3	2	199	93	428.1	$14,227.90
24	020	04JUL2008	08JUL2008	2998	4	1	118	75	414.0	$5,558.61
25	020	08OCT2008	01NOV2008	2322	1	2	162	99	434.0	$30,034.07

HOSPITALS data set

Obs	Hosp_ID	HospName	Town	NBeds	Type
1	1	University Hospital	Granville	841	1
2	2	Our Lady of Peace	East Granville	645	2
3	3	Veteran's Administration	Midtown	1176	3
4	4	Community Hospital	West Granville	448	4
5	5	City Hospital	Midtown	1025	1
6	6	Children's Hospital	Granville	239	2

DOCTORS data set

Obs	MD_ID	LastName	HospAdm
1	1972	McPhee	1
2	1972	McPhee	2
3	2322	Coresh	1
4	2322	Coresh	3
5	2998	Peterson	4
6	3274	Lieberman	1
7	3274	Lieberman	2
8	3274	Lieberman	3
9	4003	Zhao	5
10	7803	Prince	2
11	7803	Prince	3

Because this course focuses on preparing data for analysis, we will mostly use SQL for data management, but it can also be used for reporting. A simple example shows the difference.

```
PROC SQL;
     SELECT *
          FROM ex.admits
          WHERE YEAR(AdmDate)=2008;
```

This code produces a PROC PRINT-like listing of all hospital admissions in 2008 for our group of 20 patients. * after SELECT means that we want to see all variables.

Patient ID	**Admission date**	**Discharge date**	**Physician code**	**Hospital code**	**Discharge destination**	**Systolic blood pressure (mmHg)**	**Diastolic blood pressure (mmHg)**	**Primary diagnosis (ICD9)**	**Cost of hospital stay**
001	06JUN2008	12JUN2008	3274	2	9	185	94	428.4	$6,057.30
010	30NOV2008	01DEC2008	2322	9	2	147	84	E886.3	$3,101.45
014	17JAN2008	20JAN2008	7803	3	1	162	93	414.10	$6,022.64
015	25MAY2008	06JUN2008	4003	5	2	142	81	820.8	$10,993.20
015	17AUG2008	24AUG2008	4003	5	2	138	79	038.2	$8,834.01
016	25JUL2008	30JUL2008	7803	2	1	189	101	412.1	$6,609.49
020	04JUL2008	08JUL2008	2998	4	1	118	75	414.0	$5,558.61
020	08OCT2008	01NOV2008	2322	1	2	162	99	434.0	$30,034.07

What if instead of a report, we wanted to make a new data set containing this set of observations and variables? We can simply add a CREATE TABLE clause before the SELECT keyword.

```
     CREATE TABLE admit2008 AS
          SELECT *
               FROM ex.admits
               WHERE YEAR(AdmDate)=2008;
```

Now, instead of printing a list to the default output location, a new SAS data set is created, and we see this message in the SAS log:

```
NOTE: Table WORK.ADMIT2008 created, with 8 rows and 9 columns.
```

What can we notice so far about PROC SQL code?

- No RUN; statement is required. A statement executes as soon as you submit it.
- There are fewer semi-colons (;) than in regular SAS code – only at the end of the PROC SQL statement and at the end of the SELECT statement (or CREATE / SELECT statement, if you are creating a table).
- You don't need to keep submitting the PROC SQL; statement. PROC SQL will keep processing your statements until you end the procedure with a QUIT; statement.
- You can use SAS DATA step functions such as YEAR with PROC SQL. This includes summary statistics functions such as SUM and MEAN (see examples 5 and 6).

Syntax:

```
PROC SQL <options>;
        <CREATE TABLE table-name AS>              [7]
        SELECT column(s)                          [5]
                FROM table-name                   [1]
                        <WHERE expression>        [2]
                        <GROUP BY column(s)>      [3]
                        <HAVING expression>       [4]
                        <ORDER BY column(s)> ;    [6]
QUIT;
```

Basically what you have is a SELECT statement (possibly CREATE/SELECT) with parts called clauses, where each clause starts with a keyword (FROM, WHERE, etc.) and the FROM clause is required.

Everything in angle brackets <> is optional.

Note that this order of specifying the clauses is REQUIRED. Here's a sentence that might help you remember the order: **S**ome **F**rench **w**orkers **g**lue **h**ardwood **o**ften. In case you are interested, the [numbers] show the actual order of processing of the clauses.

Summary of keyword meanings:

WHERE	Subsetting conditions for the query. Uses the same syntax as the DATA step's WHERE statement.
GROUP BY	Columns for grouping or categorizing as the query is performed.
HAVING	Subsetting conditions for groups if using GROUP BY or to subset on a variable created in the query.
ORDER BY	Columns to use to sort the results of the query. Use ASC or DESC after a column name to control whether ascending or descending order is used for that column. ASC is the default so can be omitted.

Useful PROC SQL statement options (default underlined):

PRINT | NOPRINT

> Specifies whether the output from a SELECT statement is printed. NOPRINT can save time if your goal is simply to write values to macro variables as in Examples 12 and 13.

NUMBER | NONUMBER

> Use NUMBER to produce a Row column in printed output that is similar to PROC PRINT's Obs column.

FEEDBACK | NOFEEDBACK

> Use FEEDBACK to see some details in the log about how the SELECT statement is evaluated – parentheses are put around expressions to show their order of evaluation, an * is expanded into an explicit list of columns, macro variables are resolved, the libref is added to every column name, etc. See Example 14 for a situation where the FEEDBACK option is particularly useful.

If you want to change option settings in the middle of a PROC SQL step, you can use a RESET statement to do so. For example, you might start off with PROC SQL NOPRINT; but later submit RESET PRINT;.

Let's look at some simple reports to see what the different keywords mean, along with some other features of PROC SQL (such as aliases). Remember, you could always precede SELECT with CREATE table-name AS to create a new data set rather than a listing, but listings are more useful for purposes of illustration.

Example 1: List all rows from the HOSPITALS data set, but only certain variables; ordering rows

```
PROC SQL;
     SELECT Town, HospName, NBeds
          FROM ex.hospitals;
```

Hospital location	**Hospital name**	**Number of beds**
Granville	University Hospital	841
East Granville	Our Lady of Peace	645
Midtown	Veteran's Administration	1176
West Granville	Community Hospital	448
Midtown	City Hospital	1025
Granville	Children's Hospital	239

- As opposed to the introductory example where we used SELECT * to select all variables, here we listed the specific variables we wanted to see. Note that the variables names are separated by commas rather than spaces (a difference from variable lists in regular SAS).

- The order of variables in the SELECT statement determines the order of variables in the listing. What do you think would happen if we preceded SELECT with CREATE <table-name> AS? That's right, our new data set would contain these 6 rows and these 3 variables, with the variables in this order rather than the variable order in the original data set.

If we want to control the order of the rows printed (or returned in a new data set, if we start with CREATE <table-name> AS), we simply need to add an ORDER BY clause.

```
SELECT Town, HospName, NBeds
     FROM ex.hospitals
     ORDER BY HospName;
```

Hospital location	Hospital name	Number of beds
Granville	Children's Hospital	239
Midtown	City Hospital	1025
West Granville	Community Hospital	448
East Granville	Our Lady of Peace	645
Granville	University Hospital	841
Midtown	Veteran's Administration	1176

Example 2: Creating a new variable with an expression or formula

```
PROC SQL;
     SELECT PT_ID, PrimDX, AdmDate, DisDate,
            (DisDate-AdmDate+1) AS LOS LABEL='Length of stay',
            HospCharge
          FROM ex.admits
          WHERE Hosp=3;
```

Patient ID	Primary diagnosis (ICD9)	Admission date	Discharge date	Length of stay	Cost of hospital stay
003	431	15MAR2007	15MAR2007	1	$502.19
008	820.8	01OCT2007	15OCT2007	15	$20,709.55
008	V54.8	26NOV2007	28NOV2007	3	$1,904.47
014	414.10	17JAN2008	20JAN2008	4	$6,022.64
018	428.1	01NOV2007	15NOV2007	15	$15,504.32
018	428.1	26DEC2007	08JAN2008	14	$14,227.90

If we want to use our new column LOS in a subsequent expression, we need to precede it with the keyword CALCULATED so that the SQL compiler can find it.

```
     SELECT PT_ID, PrimDX, AdmDate, DisDate,
            (DisDate-AdmDate+1) AS LOS LABEL='Length of stay',
            HospCharge,
            (HospCharge/CALCULATED LOS) AS CostPerDay FORMAT=dollar9.2
          FROM ex.admits
          WHERE Hosp=3;
```

Patient ID	Primary diagnosis (ICD9)	Admission date	Discharge date	Length of stay	Cost of hospital stay	CostPerDay
003	431	15MAR2007	15MAR2007	1	$502.19	$502.19
008	820.8	01OCT2007	15OCT2007	15	$20,709.55	$1,380.64

Patient ID	Primary diagnosis (ICD9)	Admission date	Discharge date	Length of stay	Cost of hospital stay	CostPerDay
008	V54.8	26NOV2007	28NOV2007	3	$1,904.47	$634.82
014	414.10	17JAN2008	20JAN2008	4	$6,022.64	$1,505.66
018	428.1	01NOV2007	15NOV2007	15	$15,504.32	$1,033.62
018	428.1	26DEC2007	08JAN2008	14	$14,227.90	$1,016.28

- We can attach labels and formats to calculated columns by specifying them directly after the column creation expression. We can also specify labels and formats for existing columns.

As a reminder, we can create a new data set based on any SELECT statement (or query) by inserting CREATE TABLE before SELECT. Nothing is printed in this case.

```
PROC SQL;
     CREATE TABLE LOSandDailyCost AS
     SELECT PT_ID, PrimDX, AdmDate, DisDate,
            (DisDate-AdmDate+1) AS LOS LABEL='Length of stay',
             HospCharge,
             (HospCharge/CALCULATED LOS) AS CostPerDay FORMAT=dollar9.2
          FROM ex.admits
          WHERE Hosp=3;
```

```
NOTE: Table WORK.LOSANDDAILYCOST created, with 6 rows and 7 columns.
```

PROC CONTENTS provides this information about the new data set. Note the order of variables and the variable labels and formats.

Variables in Creation Order

#	Variable	Type	Len	Format	Label
1	PT_ID	Char	3		Patient ID
2	PrimDX	Char	8		Primary diagnosis (ICD9)
3	AdmDate	Num	8	DATE9.	Admission date
4	DisDate	Num	8	DATE9.	Discharge date

Variables in Creation Order

#	Variable	Type	Len	Format	Label
5	LOS	Num	8		Length of stay
6	HospCharge	Num	8	DOLLAR12.2	Cost of hospital stay
7	CostPerDay	Num	8	DOLLAR9.2	

Example 3: COUNT function and DISTINCT keyword

The COUNT function in SQL has no exact parallel in the DATA step. If called with the name of a particular variable as an argument, it returns the number of non-missing values of that variable. If called with an asterisk (*) as the argument, it returns the number of rows in the table [note that no other SQL functions accept * as an argument]. Compare the two values returned in this code:

```
* COUNT function;
DATA one;
      DO x=1 TO 10;
            OUTPUT;
      END;
      x=.;  OUTPUT;
RUN;
PROC SQL;
      TITLE 'COUNT(*)'; SELECT COUNT(*) FROM one;
      TITLE 'COUNT(x)'; SELECT COUNT(x) FROM one;
QUIT;
```

```
COUNT(*)

   11

COUNT(x)

   10
```

The DISTINCT keyword appears after SELECT and is followed by an expression, typically a variable or list of variables. It returns the unique set of values of that variable or combination of variables – that is, duplicates are removed.

```
DATA two;
      DO x=1 TO 5;
            OUTPUT; OUTPUT;
      END;
RUN;
PROC PRINT DATA=two; RUN;
PROC SQL;
      SELECT DISTINCT x FROM two;
      SELECT COUNT(DISTINCT x) FROM two;
QUIT;
```

In data set TWO, two records appear with x at each value between 1 and 5.

Obs	x
1	1
2	1
3	2
4	2
5	3
6	3
7	4
8	4
9	5
10	5

DISTINCT x returns this list:

x
1
2
3
4
5

COUNT(DISTINCT x) returns this value:

```
5
```

The combination of COUNT and DISTINCT shown above, as in COUNT(DISTINCT variable-name), is a very powerful one and is commonly used by SAS programmers who use PROC SQL for few other purposes: it returns a count of the number of unique values of a variable. PROC FREQ was recently enhanced with an option to return this information (the NLEVELS option on the PROC FREQ statement, along with an associated ODS output data set), but the SQL method might fit more smoothly into your application.

For an example using our patient data, say we want to count the number of different primary physicians we have among our patients.

```
PROC SQL;
     CREATE TABLE MDCount_SQL AS
          SELECT COUNT(DISTINCT PrimMD) AS MDCount
               FROM ex.patients;
QUIT;

ODS OUTPUT NLEVELS=MDCount_FREQ;
PROC FREQ DATA=ex.patients NLEVELS;
     TABLES PrimMD;
RUN;

PROC PRINT DATA=MDCount_SQL; TITLE 'From SQL'; RUN;
PROC PRINT DATA=MDCount_FREQ; TITLE 'From PROC FREQ'; RUN;
TITLE;
```

```
                                   From SQL

Obs   MDCount
  1         7


                                   From FREQ

Obs   TableVar      TableVarLabel        NLevels   NMissLevels   NNonMissLevels
  1   PrimMD    ID of primary physician        8             1                7
```

As you can see, PROC FREQ provides more information relative to missing values, and it does so even when the MISSING option is not used on the TABLES statement.

Example 3 addendum: For counts by groups, you can use COUNT(*) with GROUP BY. For example, for counts of the number of patients having each physician as their primary:

```
SELECT PrimMD, COUNT(*) AS Count
     FROM ex.patients
     GROUP BY PrimMD;
```

ID of primary physician	**Count**
.	1
1971	1
1972	5
2322	3
3274	3
4003	4
7803	2
8034	1

Example 4: Creating a new variable using CASE

IF/THEN statements are not supported in SQL. Instead, we can use a CASE clause for applying conditional logic of sorts.

```
PROC SQL;
      SELECT PT_ID, PrimDX,
              CASE
                  WHEN SUBSTR(PrimDX,1,3)='155' THEN 'Liver cancer'
                  WHEN SUBSTR(PrimDX,1,3)='410' THEN 'Acute myocardial infarction'
                  WHEN SUBSTR(PrimDX,1,3) IN ('411','414') THEN 'Ischemic heart disease'
                  WHEN SUBSTR(PrimDX,1,3)='428' THEN 'Heart failure'
                  WHEN SUBSTR(PrimDX,1,1)='4' THEN 'Other coronary disease'
                  WHEN SUBSTR(PrimDX,1,1)='E' THEN 'Externally-caused injury'
                  WHEN PrimDX='V54.8' THEN 'Orthopedic aftercare'
                  ELSE 'Other'
                  END AS DXCategory
            FROM ex.admits;
```

Patient ID	Primary diagnosis (ICD9)	DXCategory
001	410.0	Acute myocardial infarction
001	428.2	Heart failure
001	813.90	Other
001	428.4	Heart failure
003	431	Other coronary disease
004	434.1	Other coronary disease
005	411.81	Ischemic heart disease
005	410.9	Acute myocardial infarction
005	411.0	Ischemic heart disease
007	155.0	Liver cancer
007	155.0	Liver cancer
008	820.8	Other
008	V54.8	Orthopedic aftercare
008	V54.8	Orthopedic aftercare

Patient ID	Primary diagnosis (ICD9)	DXCategory
009	410.1	Acute myocardial infarction
010	E886.3	Externally-caused injury
012	410.52	Acute myocardial infarction
014	414.10	Ischemic heart disease
015	820.8	Other
015	038.2	Other
016	412.1	Other coronary disease
018	428.1	Heart failure
018	428.1	Heart failure
020	414.0	Ischemic heart disease
020	434.0	Other coronary disease

- Notice that `WHEN SUBSTR(PrimDX,1,1)='4' THEN 'Other coronary disease'` did not reset DXCategory when DXCategory had already been assigned.

- ELSE is not required, but I recommend including it to show that you are in control. If the ELSE sub-clause is omitted, the new variable is set to missing in cases not covered by WHEN sub-clauses.

- CASE works with numeric values and expressions as well as character ones.

```
SELECT PT_ID, BP_Sys,
          CASE
               WHEN MISSING(BP_Sys) THEN .
               WHEN BP_Sys<=80 THEN 1
               WHEN 80<BP_Sys<=170 THEN 2
               ELSE 3
          END AS BP_Sys_Cat
     FROM ex.admits;
```

Example 5: Doing calculations for groups

This query will show how much each hospital collected from each patient over the period covered by our ADMITS data set.

```
PROC SQL;
     SELECT PT_ID, Hosp,
            SUM(HospCharge) AS TotCharge FORMAT=dollar10.2
          FROM ex.admits
          GROUP BY PT_ID, Hosp;
```

```
 Patient ID   Hospital code   TotCharge
001                       1  $13,021.05
001                       2  $14,915.51
003                       3     $502.19
004                       2   $6,901.96
005                       1  $13,297.77
005                       2   $5,539.29
007                       2  $22,231.31
008                       3  $22,614.02
008                       9   $2,322.51
009                       2  $45,338.26
010                       9   $3,101.45
012                       5   $5,532.95
014                       3   $6,022.64
015                       5  $19,827.21
016                       2   $6,609.49
018                       3  $29,732.22
020                       1  $30,034.07
020                       4   $5,558.61
```

If we want to subset on the variable calculated by groups, we need to use the HAVING clause rather than WHERE. Here we are focusing on patients who ran up charges of more than $10,000 at a hospital.

```
     SELECT PT_ID, Hosp,
            SUM(HospCharge) AS TotCharge FORMAT=dollar10.2
          FROM ex.admits
          GROUP BY PT_ID, Hosp
          HAVING TotCharge>10000;
```

Patient ID	Hospital code	TotCharge
001	1	$13,021.05
001	2	$14,915.51
005	1	$13,297.77
007	2	$22,231.31
008	3	$22,614.02
009	2	$45,338.26
015	5	$19,827.21
018	3	$29,732.22
020	1	$30,034.07

What if we use a summary function but forget to use GROUP BY?

```
     SELECT PT_ID, Hosp,
            SUM(HospCharge) AS TotCharge FORMAT=dollar10.2
          FROM ex.admits;
```

Partial output:

Patient ID	Hospital code	TotCharge
001	1	$253102.51
001	1	$253102.51
001	2	$253102.51

Patient ID	Hospital code	TotCharge
001	2	$253102.51
003	3	$253102.51
004	2	$253102.51
005	1	$253102.51
005	1	$253102.51
005	2	$253102.51
007	2	$253102.51
007	2	$253102.51

The grand total has been computed and put onto every row! We can actually use this feature in computing percentages. Below we are looking at what percent of hospital revenue, as reflected in the ADMITS data set, came from each participant's separate admission. The ORDER BY clause is added here to sort the results in a meaningful way; you might try the example without ORDER BY to see what happens.

```
SELECT PT_ID, Hosp,
       HospCharge/SUM(HospCharge) AS PctCharge FORMAT=percent7.2
     FROM ex.admits
     GROUP BY Hosp
     ORDER BY Hosp, PT_ID;
```

Patient ID	Hospital code	PctCharge
001	1	21.3%
001	1	1.78%
005	1	7.22%
005	1	16.4%
020	1	53.3%
001	2	5.97%
001	2	8.72%
004	2	6.80%
005	2	5.46%

Patient ID	Hospital code	PctCharge
007	2	8.20%
007	2	13.7%
009	2	44.7%
016	2	6.51%
003	3	0.85%
008	3	35.2%
008	3	3.23%
014	3	10.2%
018	3	24.2%
018	3	26.3%
020	4	100%
012	5	21.8%
015	5	43.3%
015	5	34.8%
008	9	42.8%
010	9	57.2%

Notice that the percentages add up to 100 for our GROUP BY variable, Hosp.

On this type of operation you will see a note about "remerging" in the SAS log.

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

When you see this note, check carefully to make sure that the results are what you expect.

Example 6: Which hospital has the highest cost per day?

For our set of hospital admissions, what if we wanted to rank the hospitals from lowest to highest in terms of the average cost per day of a hospital stay?

Here is one way to do this with regular SAS.

```
DATA admits;
     SET ex.admits;
     LOS = DisDate-AdmDate+1;
     CostPerDay = HospCharge/LOS;
RUN;

PROC MEANS DATA=admits NOPRINT NWAY;
     CLASS Hosp;
     VAR CostPerDay;
     OUTPUT OUT=dailycharges N=NumAdmits MEAN=MeanCostPerDay;
RUN;

PROC SORT DATA=dailycharges(DROP=_TYPE_ _FREQ_);
     BY MeanCostPerDay;
RUN;

PROC PRINT DATA=dailycharges NOOBS;
     TITLE 'Hospitals in Rank Order by Average Daily Cost';
     TITLE2 'Computed with Regular SAS';
     FORMAT MeanCostPerDay dollar9.2;
RUN;
```

```
Hospitals in Rank Order by Average Daily Cost
         Computed with Regular SAS

   Hosp  NumAdmits  MeanCostPerDay

      1          5         $920.74

      3          6       $1,012.20

      5          3       $1,018.82
```

Hosp	NumAdmits	MeanCostPerDay
4	1	$1,111.72
2	8	$1,142.46
9	2	$1,162.45

How can we do this with SQL?

```
title 'Hospitals in Rank Order by Average Daily Cost';
title2 'Computed with PROC SQL';
PROC SQL;
     SELECT Hosp,
            COUNT(*) AS NumAdmits,
            MEAN(HospCharge/(DisDate-AdmDate+1)) AS MeanCostPerDay FORMAT=dollar9.2
           FROM ex.admits
           GROUP BY Hosp
           ORDER BY MeanCostPerDay;
```

Hospitals in Rank Order by Average Daily Cost
Computed with PROC SQL

Hospital code	NumAdmits	MeanCostPerDay
1	5	$920.74
3	6	$1,012.20
5	3	$1,018.82
4	1	$1,111.72
2	8	$1,142.46
9	2	$1,162.45

This example uses SQL's powerful COUNT summary function. COUNT(*) says to count the number of rows for each value of Hosp, the GROUP BY variable.

Note that if we had included in the SELECT list any patient-level variables such as HospCharge, the list would have included <u>all</u> ADMITS data set records, not just one row per hospital.

Example 7: Joining tables

A UNION in a SQL query is generally like a DATA step SET and is used for concatenating data sets (see example 10), and the DATA step MERGE equivalent in SQL is referred to as a *join*. Joining data sets using PROC SQL rather than a DATA step MERGE has at least two advantages in terms of convenience. First, there is no need to pre-sort the data sets by the merge variables. Second, the merge variables can have different names in the data sets being combined.

For example, say that we wanted to add hospital names to the physician-level data. Right now, only the hospital ID is on the physician records.

How could we do this with regular SAS, including coding for the fact that the hospital ID variable is named HospAdm on the DOCTORS data set and Hosp_ID on the HOSPITALS data set?

```
PROC SORT DATA=ex.doctors OUT=doctors;
     BY HospAdm;
RUN;

DATA AddHospNamesSAS;
     MERGE doctors(IN=IND) ex.hospitals(KEEP=Hosp_ID HospName
                                  RENAME=(Hosp_ID=HospAdm));
     BY HospAdm;
     IF IND;
RUN;

PROC SORT DATA=AddHospNamesSAS;
     BY MD_ID HospAdm;
RUN;
```

Physician data with hospital names added, done with regular SAS

Obs	MD_ID	LastName	HospAdm	HospName
1	1972	McPhee	1	University Hospital
2	1972	McPhee	2	Our Lady of Peace
3	2322	Coresh	1	University Hospital
4	2322	Coresh	3	Veteran's Administration
5	2998	Peterson	4	Community Hospital
6	3274	Lieberman	1	University Hospital

Obs	MD_ID	LastName	HospAdm	HospName
7	3274	Lieberman	2	Our Lady of Peace
8	3274	Lieberman	3	Veteran's Administration
9	4003	Zhao	5	City Hospital
10	7803	Prince	2	Our Lady of Peace
11	7803	Prince	3	Veteran's Administration

Now, how could we do this with PROC SQL?

```
PROC SQL NOPRINT;
      CREATE TABLE AddHospNamesSQL AS
            SELECT D.MD_ID, D.LastName, D.HospAdm, H.HospName
                  FROM ex.doctors AS D, /* D and H are aliases */
                       ex.hospitals AS H
                  WHERE H.Hosp_ID=D.HospAdm;
```

Physician data with hospital names added, done with PROC SQL

Obs	MD_ID	LastName	HospAdm	HospName
1	1972	McPhee	1	University Hospital
2	1972	McPhee	2	Our Lady of Peace
3	2322	Coresh	1	University Hospital
4	2322	Coresh	3	Veteran's Administration
5	2998	Peterson	4	Community Hospital
6	3274	Lieberman	1	University Hospital
7	3274	Lieberman	2	Our Lady of Peace
8	3274	Lieberman	3	Veteran's Administration
9	4003	Zhao	5	City Hospital
10	7803	Prince	2	Our Lady of Peace

Obs	MD_ID	LastName	HospAdm	HospName
11	7803	Prince	3	Veteran's Administration

Notice the use of aliases D and H for data set names ex.doctors and ex.hospitals, respectively, in this PROC SQL example. Aliases are useful in a PROC SQL statement that uses more than one data set in order to tell the SQL compiler where to find specific columns. You establish an alias with the AS keyword in the FROM clause, and then you can use the alias to precede a variable name in other clauses of the statement.

To reiterate, this SQL query did not require us to sort the data sets before the join, and the join was performed correctly even though the hospital ID variables had different names on the two data sets – the WHERE clause allowed us to check for equality of variables with different names.

Example 8: Find all admissions for females

A common use of SQL is to find all records in one data set that meet some criteria based on values in a different data set. For example, what if we want to find all hospital admissions for females, where gender only appears in the PATIENTS data set?

Here again we are using aliases and joining using variables having different names.

```
PROC SQL;
      SELECT p.id, p.sex, a.admdate, a.primdx
            FROM ex.patients AS P,
                   ex.admits AS A
            WHERE p.sex=2 AND p.id=a.pt_id;
```

Patient ID	Patient gender	Admission date	Primary diagnosis (ICD9)
005	2	19JAN2007	411.81
005	2	10MAR2007	410.9
005	2	10APR2007	411.0
008	2	01OCT2007	820.8
008	2	26NOV2007	V54.8
008	2	10DEC2007	V54.8
009	2	15DEC2007	410.1
010	2	30NOV2008	E886.3
014	2	17JAN2008	414.10
015	2	25MAY2008	820.8
015	2	17AUG2008	038.2
020	2	04JUL2008	414.0
020	2	08OCT2008	434.0

Example 9: A more complicated joining example, and using a nested subquery

Say that we wanted to add the name of each patient's primary physician to his or her patient-level data. Right now, only the physician ID is on the patient records. This is more complicated than the previous example because the DOCTORS data set has multiple records per doctor (one per hospital where the doctor is allowed to admit patients).

How could we do this with regular SAS?

```
PROC SORT DATA=ex.doctors(KEEP=MD_ID LastName)
          OUT=doctors1(RENAME=(MD_ID=PrimMD LastName=DocName)) NODUPKEY;
      BY MD_ID;
RUN;

PROC SORT DATA=ex.patients OUT=patients;
      BY PrimMD;
RUN;

DATA AddDocNamesSAS;
      MERGE patients(IN=INP) doctors1;
      BY PrimMD;
      IF INP;
run;

PROC SORT DATA=AddDocNamesSAS;
      BY ID;
RUN;

PROC PRINT DATA=AddDocNamesSAS; TITLE 'From regular SAS'; RUN; TITLE;
```

From regular SAS

Obs	ID	Sex	PrimMD	BirthDate	LastName	FirstName	DocName
1	001	1	1972	10AUG1941	McGiver	Perry	McPhee
2	002	2	1972	17MAR1939	Kost	Mary	McPhee
3	003	1	.	02JUL1929	Thompkins	Felicio	
4	004	1	4003	25MAY1926	Collins	Jeffrey	Zhao
5	005	2	1972	31AUG1941	Ray	Linda	McPhee
6	006	1	2322	12APR1909	Tolliver	Sidney	Coresh
7	007	1	3274	07FEB1910	Coppersmith	Eugenio	Lieberman
8	008	2	4003	09NOV1945	Hallman	Marybeth	Zhao

Obs	ID	Sex	PrimMD	BirthDate	LastName	FirstName	DocName
9	009	2	3274	15SEP1919	Kramer	Carmella	Lieberman
10	010	2	2322	15OCT1949	Kohl	Melissa	Coresh
11	011	2	1972	04NOV1927	Ou	Ping	McPhee
12	012	1	7803	16JUN1936	Dahlgren	Elisabeth	Prince
13	013	1	4003	03AUG1947	Whybel	Sander	Zhao
14	014	2	8034	14DEC1942	Blue	Sally	
15	015	2	3274	.	Hostetler	Lisa	Lieberman
16	016	1	1971	17JUN1914	DeMarcus	Frederick	
17	017	1	2322	17APR1932	Hollings	David	Coresh
18	018	1	1972	13NOV1948	Pelletier	Crosby	McPhee
19	019	2	4003	01FEB1934	Fung	June	Zhao
20	020	2	7803	07AUG1916	Newman	Marjolane	Prince

Now, how could we do this with PROC SQL?

A first try might look like the following, where the new LEFT JOIN keyword with the associated ON condition says to return a record for every patient, matching on physician ID to add the physician name to the patient records:

```
PROC SQL NOPRINT;
      CREATE TABLE AddDocNamesSQL1 AS
            SELECT P.ID, P.Sex, P.PrimMD, P.BirthDate, P.LastName, P.FirstName,
                   D.LastName AS DocName
                  FROM ex.patients AS P
                       LEFT JOIN
                       ex.doctors AS D
                       ON P.PrimMD=D.MD_ID
                  ORDER BY P.ID;
```

If you try this, you'll see that you get a data set of 36 records rather than the 20 we were expecting (since we have 20 patients).   What's going on? Recall that the physician data set contains multiple records for any doctors who have admitting privileges at more than one hospital.   Well, the default behavior for an SQL join is to return a Cartesian product: that is, every combination of records from the contributing tables. This is actually difficult to do with the SAS DATA step, but with SQL it's what you get unless you carefully ask for something different.   So with this code, for any patient who has a doctor who is allowed to admit to more than one hospital, we end up with more than one record for that patient in the AddDocNamesSQL1 table made above. That's why we end up with 36 records instead of 20.

So how can we get SQL to produce what we want, which is one copy of each patient record with his or her doctor's name added? This code will do it.

```
PROC SQL NOPRINT;
      CREATE TABLE AddDocNamesSQL AS
            SELECT P.ID, P.Sex, P.PrimMD, P.BirthDate, P.LastName, P.FirstName,
                   D.LastName AS DocName
                  FROM ex.patients AS P
                       LEFT JOIN
                       (SELECT DISTINCT LastName, MD_ID
                             FROM ex.doctors) AS D
                       ON P.PrimMD=D.MD_ID
                  ORDER BY P.ID;
```

This code introduces several important SQL features: SELECT DISTINCT, support for subqueries, and LEFT JOIN/ON.

SELECT DISTINCT says to remove duplicate values of the associated variable combination, here LastName and MD_ID, in the query returned from the ex.doctors table.

(SELECT DISTINCT LastName, MD_ID FROM ex.doctors) is an example of a nested query or subquery. In the previous code snippet we saw that joining with the original DOCTORS data set did not work because it contains multiple records for some doctors. In the current code, the alias D now refers to the output from this subquery, which is only one record per physician.   Support for subqueries is a feature that helps make SQL very powerful, though also very complex and sometimes mysterious and ALWAYS REQUIRING CAREFUL CHECKING OF YOUR RESULTS. See Example 9a for an additional nested subquery example, and I also recommend pages 96-103 of the SAS 9.4 SQL Procedure User's Guide http://support.sas.com/documentation/cdl/en/sqlproc/65065/PDF/default/sqlproc.pdf.

LEFT JOIN with the associated ON: Joining in SQL is roughly equivalent to merging in the DATA step (records are combined horizontally). The ON subclause tells how the records are to be matched up (kind of like WHERE). This brief introduction to SQL can barely scratch the surface of joining details, but the table below summarizes how to achieve SQL joins that are the equivalent of DATA step merges using IN flags.

SQL	DATA step

FULL JOIN
(or FULL OUTER JOIN)
	Default MERGE behavior: keep all records in the incoming tables, whether or not they match according to the ON condition; the equivalent of a LEFT JOIN plus a RIGHT JOIN
	
	CREATE TABLE new AS
	SELECT var-list
	FROM table1 AS A
	FULL JOIN
	table2 AS B
	ON A.var1=B.var2;

LEFT JOIN
(or LEFT OUTER JOIN)
	Like having an IN flag on the first table in the FROM clause and keeping IF that IN flag is 1 (keep all records that are in the first table, add information from matching records in the second)
	
	CREATE TABLE new AS
	SELECT var-list
	FROM table1 AS A
	LEFT JOIN
	table2 AS B
	ON A.var1=B.var2;

RIGHT JOIN (or RIGHT OUTER JOIN)	Like having an IN flag on the second table in the FROM clause and keeping IF that IN flag is 1 (keep all records that are in the second table, add information from matching records in the first); code parallel to above

INNER JOIN
(or simply JOIN; can also accomplish with a WHERE clause)
	Like having an IN flag on both tables in the FROM clause and keeping only IF both flags are 1.
	
	CREATE TABLE new AS
	SELECT var-list
	FROM table1 AS A
	INNER JOIN
	table2 AS B
	ON A.var1=B.var2;
	or
	CREATE TABLE new AS
	SELECT var-list

	FROM table1 AS A, table2 as B
	WHERE A.var1=B.var2;
	Examples 7 and 8 show inner joins using WHERE.

Two things to remember: (1) SQL's inclination to make a Cartesian product always applies. (2) You can use a WHERE clause to add subsetting conditions in any query, even one using ON.

Also, many joins in real programs will be more complex than those shown in the comparison table above. For example, you can stack joins and have compound ON conditions. With stacked joins, joins are performed sequentially from the top down.

In the skeleton of a join below, first Table1 and Table2 would be left joined using the first ON clause. Then the result of that join would be left joined with Table3 using the second ON clause.

```
SELECT A.var1, A.var2, B.var4, C.var5
        FROM Table1 as A
                LEFT JOIN
                Table2 as B
                ON A.var1 = B.var1 and A.var2 = B.var3
                LEFT JOIN
                Table3 as C
                ON A.var1 = C.var1 and A.var2 = C.var3
                ;
```

Example 9a: Another Nested Subquery Example

Often, a nested subquery is used on the right side of a WHERE clause to help select rows in one table that meet a criterion based on the rows of a different table.

For example, what if we wanted to list some information about all admissions to hospitals in the town of Midtown? The admissions information is in the ADMITS data set, while the location of each hospital is in the HOSPITALS data set. Using a nested subquery is an excellent technique in this case.

```
PROC SQL;
    SELECT pt_id, admdate, primDX, hosp
        FROM ex.admits
        WHERE hosp IN
            (SELECT hosp_ID
                FROM ex.hospitals
                WHERE UPCASE(town)='MIDTOWN');
```

Patient ID	Admission date	Primary diagnosis (ICD9)	Hospital code
003	15MAR2007	431	3
008	01OCT2007	820.8	3
008	26NOV2007	V54.8	3
012	12AUG2007	410.52	5
014	17JAN2008	414.10	3
015	25MAY2008	820.8	5
015	17AUG2008	038.2	5
018	01NOV2007	428.1	3
018	26DEC2007	428.1	3

The hospital codes listed indicate that the nested subquery worked correctly (but you would always want to check carefully to make sure!). Note that we needed to use IN rather than = before the nested subquery in case several rows were returned from HOSPITALS. If you expected the nested subquery to return a number or single value, the appropriate operator might be =, >, <, >=, etc.

Some rules for subqueries

- Subqueries must be enclosed in parentheses.
- In a usage such as Example 9a, a subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare with its selected columns.
- An ORDER BY cannot be used in a subquery, although the main query can use an ORDER BY. GROUP BY can be used in a subquery to perform the same function.
- Subqueries that return more than one row can only be used with multiple value operators such as IN.

Example 10: Using UNION to concatenate data sets

Like the DATA step's SET statement, PROC SQL's UNION operator can be used to concatenate data sets.

The power of UNION,which doesn't really show up in this example, comes from concatenating the results of SELECT clauses, with all of the flexibility that implies.

The example starts with the Q1 and Q2 data sets of quarterly sales by region that you might have used in BIOS 511. Our goal is to combine these two data sets to form a data set of sales for the first half of the year, with each row representing a region and month. Since in Q1 and Q2 the months are in columns, we need to begin by transposing the data sets (see the reference notes and video on PROC TRANSPOSE to understand the transposing code).

Q1

Obs	Region	January	February	March
1	S	147	202	319
2	W	140	180	177

Q2

Obs	Region	April	May	June
1	S	388	307	201
2	W	210	202	207

```
*** transpose the data sets;
PROC TRANSPOSE DATA=bios511.q1 OUT=q1t(RENAME=(col1=NumSold _name_=Month));
      BY region;
      VAR January February March;
RUN;
PROC TRANSPOSE DATA=bios511.q2 OUT=q2t(RENAME=(col1=NumSold _name_=Month));
      BY region;
      VAR April May June;
RUN;

*** concatenate using SET, keeping regional records together;
DATA sixD;
      SET q1t q2t;
      BY region;
      LABEL month='Month';
RUN;
```

Result of SET operation

Obs	Region	Month	NumSold
1	S	January	147
2	S	February	202
3	S	March	319
4	S	April	388
5	S	May	307
6	S	June	201
7	W	January	140
8	W	February	180
9	W	March	177
10	W	April	210
11	W	May	202
12	W	June	207

```
*** first try at concatenating with SQL's UNION;
PROC SQL;
      CREATE table sixU AS
            SELECT * FROM q1t
            UNION
            SELECT * FROM q2t;
QUIT;
```

Result of first try at using UNION

Obs	Region	Month	NumSold
1	S	April	388
2	S	February	202
3	S	January	147
4	S	June	201
5	S	March	319
6	S	May	307
7	W	April	210
8	W	February	180
9	W	January	140

Obs	Region	Month	NumSold
10	W	June	207
11	W	March	177
12	W	May	202

Note that without our requesting it, SQL has sorted our rows alphabetically by month name. On the other hand, the SET statement maintained the order of rows in the initial data sets as it concatenated. This difference illustrates that SQL is often a black box, not meeting the expectations of people experienced with the DATA step, which is another reason to always examine your SQL results very carefully.

So how can we correct the order? Well, one way is to make a numeric month indicator that can be used for ordering. I don't know of a way to do this except with a brute force approach.

```
*** before second try, make a numeric month indicator;
DATA q1tm;
      SET q1t;
      IF UPCASE(month)='JANUARY'  THEN MonthNum=1;
      IF UPCASE(month)='FEBRUARY' THEN MonthNum=2;
      IF UPCASE(month)='MARCH'    THEN MonthNum=3;
      LABEL month='Month';
RUN;
DATA q2tm;
      SET q2t;
      IF UPCASE(month)='APRIL'  THEN MonthNum=4;
      IF UPCASE(month)='MAY'    THEN MonthNum=5;
      IF UPCASE(month)='JUNE'   THEN MonthNum=6;
      LABEL month='Month';
RUN;
PROC SQL;
      CREATE table sixU2 AS
            SELECT * FROM q1tm
            UNION
            SELECT * FROM q2tm
        ORDER BY region, monthnum;
QUIT;
```

Obs	Region	Month	NumSold	MonthNum
1	S	January	147	1
2	S	February	202	2
3	S	March	319	3
4	S	April	388	4
5	S	May	307	5

```
Obs  Region  Month      NumSold  MonthNum
  6  S       June           201         6
  7  W       January        140         1
  8  W       February       180         2
  9  W       March          177         3
 10  W       April          210         4
 11  W       May            202         5
 12  W       June           207         6
```

Example 11: Taking advantage of a Cartesian product for fuzzy matching

Fuzzy matching means matching records on inexact matches. When using a BY statement in a SAS MERGE step, only records having an exact match on a BY value are combined. What if we want to combine records with a criterion that doesn't involve an exact match? While a SAS MERGE doesn't offer this possibility, often we can carry out such combinations using SQL.

Conceptually, at the heart of a SQL fuzzy match is the default Cartesian product of matching every record in one data set with every record in another. However, we don't really want that complete operation to occur, so we limit the product using some combination of ON conditions and WHERE clauses. Because of the flexibility of a WHERE clause, an exact match is not required. ON conditions can also be written flexibly enough to support fuzzy matching.

For example, say that we want to look at patient re-admissions within 5 weeks (35 days) in our ADMITS data set, no matter what the patient's hospital. That is, if a patient had an additional hospital admission within 35 days of a previous admission, we want to see the records involved.

You could probably program this in a DATA step with some involved and tricky code involving the LAG function, but it's actually fairly easy to program with PROC SQL. The key is to join the admissions data set with itself, which if unconstrained would result in every admissions record being matched to every other admissions record, giving us N^2 records (here, 625 records if omitting the WHERE clause below). We avoid this undesired result by requesting the subset that meet the conditions described above (for the same patient, re-admission within 35 days).

```
TITLE 'Example of a fuzzy match';
PROC SQL;
      SELECT a.pt_id,
             a.admdate LABEL="Admission",
             b.admdate LABEL="Re-admission within 35 days"

            FROM ex.admits AS a, ex.admits AS b

            WHERE a.pt_id = b.pt_id AND          /* match on patient ID */
                  a.admdate < b.admdate AND      /* only if admit before re-admit */
                  b.admdate <= (a.admdate+35);   /* only if re-admit w/in 35 days */
```

Example of a fuzzy match

```
Patient ID      Admission   Re-admission within
                                        35 days

005             10MAR2007            10APR2007

008             26NOV2007            10DEC2007
```

Note that you can also program this using JOIN and ON in place of WHERE. This code will produce the same results as above.

```
TITLE 'Example of a fuzzy match using JOIN/ON';
PROC SQL;
    SELECT a.pt_id,
           a.admdate LABEL="Admission",
           b.admdate LABEL="Re-admission within 35 days"
        FROM ex.admits AS a
                JOIN
             ex.admits AS b
        ON a.pt_id = b.pt_id AND         /* match on patient ID */
           a.admdate < b.admdate AND     /* only keep if admit before re-admit */
           b.admdate <= (a.admdate+35);  /* only if re-admit within 35 days */
QUIT;
TITLE;
```

Example 12: Using INTO to create macro variables

An interesting use of PROC SQL is to write computed values directly to macro variables using the INTO keyword. If you need counts to present as part of text strings for use as column headings, for example, this could be an alternative to running PROC MEANS or PROC FREQ to get counts and then using CALL SYMPUT to write those counts to macro variables.

This bit of SQL code counts the number of times each doctor appears in the DOCTORS data set, which because of the data set structure will be the number of hospitals to which the doctor has admitting privileges. Those counts are printed to the current output destination as well as written to macro variables &HADM1, & HADM2, & HADM3, & HADM4, & HADM5, and &HADM6. Writing to macro variables is what the INTO keyword of the SELECT clause requests. The names of the macro variables are preceded by : , with the dash (-) requesting a range of variable names.

```
PROC SQL;
    SELECT N(md_id) AS NumHospAdms INTO :HAdm1 - :HAdm6
        FROM ex.doctors
        GROUP BY md_id;
%PUT HAdm1=&HAdm1 HAdm2=&HAdm2 HAdm3=&HAdm3
     HAdm4=&HAdm4 HAdm5=&HAdm5 HAdm6=&HAdm6;
QUIT;
```

```
NumHospAdms
          2
          2
          1
          3
          1
          2
```

Printed to the SAS log:

```
78   %PUT HAdm1=&HAdm1 HAdm2=&HAdm2 HAdm3=&HAdm3
79        HAdm4=&HAdm4 HAdm5=&HAdm5 HAdm6=&HAdm6;
HAdm1=2 HAdm2=2 HAdm3=1       HAdm4=3 HAdm5=1 HAdm6=2
```

Note that this technique as shown above has some perils. For example, if you don't know in advance how many doctors are included in the data set, you don't know how many macro variables to request. Also, the macro variable names in this code don't incorporate the associated physician code, so it could be easy to misuse these values.

A safer application of this general technique is computing and saving one N at a time in a query that includes a WHERE clause to subset appropriately. When computing one N at a time, you can assign a descriptive macro variable name and thus avoid potential problems. We do this all the time at the CSCC. You will see complete examples of this in the PROC REPORT chapter, but for now I will show you some partial code.

Say that we wanted to use PROC REPORT to display some pre-computed statistics for our PATIENTS data, with three columns in our table: one column for all patients, one column for females, and one column for males. The column headings should include counts. We could create well-named macro variables containing the appropriate counts with code like this:

```
PROC SQL NOPRINT;

    SELECT COUNT(id) INTO :n SEPARATED BY ' '
        FROM ex.patients;

    SELECT COUNT(id) INTO :n_females SEPARATED BY ' '
        FROM ex.patients
        WHERE sex=2;

    SELECT COUNT(id) INTO :n_males SEPARATED BY ' '
        FROM ex.patients
        WHERE sex=1;

QUIT;

%PUT all=&n females=&n_females males=&n_males ;
```

Then in our reporting code, when creating a label where the number of females was needed, we would simply insert &n_females. Note that `separated by ' '` in the code above is a trick to get SAS to remove extra blanks from the end of the macro variable value. I learned this trick from a student. Try it, you'll like it.

Example 13: Using the dictionary tables

As described in a later unit of this course, every SAS session has available metadata on all current SAS objects: libraries, data sets, filenames, formats, macro variables, etc. This metadata resides in a set of SQL tables and corresponding views that we can query, usually to write information to macro variables for use in controlling subsequent SAS processing in the current program.

For example, say that we need to sequentially process each numeric variable in our ADMITS data set. In some contexts, the _NUM_ keyword would be adequate, but sometimes there is no good substitute for having a list of all numeric variables in a data set that one can scan and process until all variables have been dealt with. A good way to obtain such a list is with PROC SQL code such as this:

```
PROC SQL NOPRINT;
      SELECT NAME INTO :numlist SEPARATED BY ' '
            FROM dictionary.columns
            WHERE libname='EX' and memname='ADMITS' and upcase(type)='NUM';
%PUT &numlist;
```

The SAS log (from the %PUT) reveals that the value of &numlist is

```
AdmDate DisDate MD Hosp Dest BP_Sys BP_Dia HospCharge
```

If we wanted to get fancier and select only numeric variables that aren't date variables, an additional check on the associated format would be fairly reliable.

```
      SELECT NAME INTO :numlist SEPARATED BY ' '
            FROM dictionary.columns
            WHERE libname='EX' and
              memname='ADMITS' and
              upcase(type)='NUM' and
              ^(index(format,'DATE')>0 or index(format,'MDY')>0);
```

The value of &numlist in this case becomes

```
MD Hosp Dest BP_Sys BP_Dia HospCharge
```

Example 14: Using the FEEDBACK option to get rid of a pesky WARNING message

Have you ever run a query with a join to create a new table and had SQL issue a warning message that a certain variable already exists on the file being created? This often happens when you use * to indicate that you want to keep all variables.

```
proc sql;
    create table test1 as
        select * from mets.dr_669 as dr,
                      mets.dema_669 as dema
        where dr.bid=dema.bid;
```

The resulting table is fine, but the log says this:

```
WARNING: Variable BID already exists on file WORK.TEST1.
NOTE: Table WORK.TEST1 created, with 148 rows and 20 columns.
```

How can you get rid of the warning message without having to type in all of the column names to be selected from the data sets being joined? The FEEDBACK option to the rescue!

```
proc sql feedback;  /* or you could submit RESET FEEDBACK; in the middle of your
PROC SQL sequence */

    create table test1 as
        select * from mets.dr_669 as dr,
                      mets.dema_669 as dema
        where dr.bid=dema.bid;
```

Now the log says this:

```
NOTE: Statement transforms to:

        select DR.BID, DR.RAND, DR.TRT, DR.WTLIAB2, DR.RACE1, DR.PSITE, DEMA.BID,
DEMA.VISIT, DEMA.FSEQNO, DEMA.LINENO, DEMA.DEMA1, DEMA.DEMA2, DEMA.DEMA4, DEMA.DEMA5,
DEMA.DEMA6, DEMA.DEMA7, DEMA.DEMA8, DEMA.DEMA9, DEMA.FORM, DEMA.VERS, DEMA.DEMA0B
          from METS.DR_669 DR, METS.DEMA_669 DEMA
         where DR.BID = DEMA.BID;

WARNING: Variable BID already exists on file WORK.TEST1.
NOTE: Table WORK.TEST1 created, with 148 rows and 20 columns.
```

Because of the FEEDBACK option, SQL has generated the needed variable lists for you! You can copy the generated SELECT statement from the log and paste it into your program, commenting out or deleting the second reference to the variable that SQL warned about.

```
create table test2 as
    select DR.BID, DR.RAND, DR.TRT, DR.WTLIAB2, DR.RACE1, DR.PSITE,
        /* DEMA.BID, */ DEMA.VISIT, DEMA.FSEQNO, DEMA.LINENO, DEMA.DEMA1,
        DEMA.DEMA2, DEMA.DEMA4, DEMA.DEMA5, DEMA.DEMA6, DEMA.DEMA7,
        DEMA.DEMA8, DEMA.DEMA9, DEMA.FORM, DEMA.VERS, DEMA.DEMA0B
      from METS.DR_669 DR, METS.DEMA_669 DEMA
     where DR.BID = DEMA.BID;
```

This query produces the same table as before, but now the log is clean.

SAS's SQL compiler

When SAS encounters the semi-colon that ends a SELECT statement, it sends the entire SELECT statement to its SQL compiler. The compiler parses the statement into its various clauses as determined by the keywords FROM, WHERE, GROUP BY, and so forth. These clauses are checked for syntactical correctness, and then they are executed in the order given earlier in this chapter. After the SELECT statement has been completely executed, PROC SQL waits for another SELECT statement to process or for a QUIT; statement or another SAS step (DATA or PROC), which will end PROC SQL. Thus, one PROC SQL call can create multiple reports and data sets.

SQL advantages and disadvantages compared with regular SAS

Of course, for an SQL programmer who needs to use SAS for some reason, being able to use his or her SQL knowledge within SAS is a huge advantage. On the other hand, a traditional SAS user might find SQL to be a mysterious black box. Beyond these obvious statements, are there some more objective advantages and disadvantages of using SQL rather than regular SAS statements?

Advantages:

- With SQL, there is no need to sort data sets or rename variables in order to perform certain operations.
- The ability to easily do a Cartesian product makes SQL preferable in certain situations (though it can lead to peril if not expected) and facilitates fuzzy matching.
- With SQL, you can calculate values and summarize them all in one step.

Disadvantages:

- With SQL, you can only create one new data set at a time (one CREATE TABLE at a time), while a DATA step can easily create multiple data sets with one logical sequence.
- Weak feedback in SAS log compared with DATA steps, although PROC SQL's FEEDBACK option could add helpful information.

A caveat about this chapter

The material in this chapter is the merest introduction to SQL (and PROC SQL). Many people make careers out of doing SQL coding well, just as some people make careers out of knowing SAS well. I hope it has provided enough information to get you started and provoke your interest in PROC SQL as a useful tool in your SAS arsenal.

References

The many useful papers on PROC SQL include the following:

Rosalind Gusinow - Introduction to PROC SQL
http://www2.sas.com/proceedings/sugi25/25/btu/25p061.pdf

Jane Stroupe and Linda Jolley - Dear Miss SASAnswers: A Guide to Efficient PROC SQL Coding
http://support.sas.com/resources/papers/sgf09/336-2009.pdf

Ian Whitlock - PROC SQL – Is It a Required Tool for Good SAS Programming?
http://www2.sas.com/proceedings/sugi26/p060-26.pdf

Christianna Williams - PROC SQL for DATA Step Die-hards
http://www.ats.ucla.edu/stat/sas/library/nesug99/ad121.pdf

Chris Yindra - An Introduction to the PROC SQL Procedure
http://www.ats.ucla.edu/stat/sas/library/nesug99/bt082.pdf

Also see Michele Burlew's invaluable book, Combining and Modifying SAS Data Sets: Examples, 2nd Edition. The definitive SAS Institute book on PROC SQL is SAS 9.4: SQL Procedure User's Guide
http://support.sas.com/documentation/cdl/en/sqlproc/65065/PDF/default/sqlproc.pdf. See especially Chapter 3, Retrieving Data from Multiple Tables, which covers joining, setting, and using subqueries.