

Paper 149-2012

Queries, Joins, and WHERE Clauses, Oh My!! Demystifying PROC SQL

Christianna S. Williams, Chapel Hill, NC

ABSTRACT

Subqueries, inner joins, outer joins, HAVING expressions, set operators...just the terminology of PROC SQL might intimidate SAS® programmers accustomed to getting the DATA step to do our bidding for data manipulation. Nonetheless, even DATA step die-hards must grudgingly acknowledge that there are some tasks, such as the many-to-many merge or the "not-quite-equi-join," requiring Herculean effort to achieve with DATA steps, that SQL can accomplish amazingly concisely, even elegantly. Through increasingly complex examples, this workshop illustrates each of PROC SQL's clauses, with particular focus on problems difficult to solve with "traditional" SAS code. After all, PROC SQL is part of Base SAS® so, although you might need to learn a few new keywords to become an SQL wizard, no special license is required!

INTRODUCTION

PROC SQL is an incredibly powerful tool for data manipulation in SAS. However, SQL "thinks" about data a bit differently than 'traditional' SAS, and these difference run deeper than terminology – beyond the question of whether we talk about 'tables or 'datasets', 'columns' or 'variables', 'observations' or 'rows'. Sometimes these differences can perplex accomplished SAS programmers. Nonetheless, because of the ability of SQL to combine data aggregation and linkage, not to mention flexibility in joins far beyond what is feasible in a DATA step MERGE, SAS programmers who need to rearrange their data ignore PROC SQL at their peril. I've been using the DATA Step and other 'traditional' SAS tools like PROC SUMMARY and MEANS for a very long time, but I've come to appreciate how PROC SQL can simplify many data manipulation tasks. I have learned some of this through trial and error and trying many things that I thought would work but didn't and then going on to figure out how to make it work. I'm hoping to share some of what I've learned about SQL in this paper, through lots of examples. In an earlier conference paper that I've presented several times, 'PROC SQL for DATA Step Die-Hards', every example was a comparison of DATA Step and SQL methods. I'm not using that approach in this paper for a couple of reasons. One is that more and more programmers are learning SQL in tandem with the DATA Step, so the 'translation' issue is not as important. Another is that for some of the examples in this paper, it would just be silly to try to do them with the DATA step. And the third is that, without having to go through lots of DATA Step examples, I can present even more SQL!! Still, I can't help but make references throughout to how the processing differs between SQL and the DATA Step; if you are not a DATA Step-per, you can just ignore these interludes. Let's go!

THE DATA

First, a brief introduction to the data sets. Table 1 describes the five logically linked data sets, which concern the hospital admissions and emergency room visits for twenty completely fictitious patients. The variable or variables that uniquely identify an observation within each data set are indicated in bold; the data sets are sorted by these keys. Complete listings are included at the end of the paper. Throughout the paper, it is assumed that these data sets are located in a data library referenced by the libref EX.

| Data set (Table) | Variable (Column) | Description |
|-------------------|--------------------------------|---------------------------------|
| ADMISSIONS | pt_id | patient identifier |
| | admdate | date of admission |
| | disdate | date of discharge |
| | hosp | hospital identifier |
| | bp_sys | systolic blood pressure (mmHg) |
| | bp_dia | diastolic blood pressure (mmHg) |
| | dest | discharge destination |
| | primdx | primary diagnosis (ICD-9) |
| md | admitting physician identifier | |

| Data set (Table) | Variable (Column) | Description |
|------------------|-------------------|---|
| PATIENTS | id | patient identifier |
| | lastname | patient last name |
| | firstname | patient first name |
| | sex | gender (1=M, 2=F) |
| | dob | date of birth |
| | primmd | primary physician identifier |
| | zipcode | patient residence zip code |
| HOSPITALS | hosp_id | hospital identifier |
| | hospname | hospital name |
| | zip | hospital zip code |
| | beds | number of beds |
| | has_er | Y if hospital has ER, N otherwise |
| DOCTORS | md_id | physician identifier |
| | hospadm | hospital at which MD has admitting privileges |
| | lastname | physician last name |

Table 1. Listing of tables and variables used in examples. Records on each table are uniquely identified by the columns that are in bold; tables are also sorted by these variables

EXAMPLE 1: SUBSETTING VARIABLES (COLUMNS) AND OBSERVATIONS (ROWS)

In this first, extremely simple example, we just want to create a subset of the ADMISSIONS data set that contains selected variables (columns) for all the admissions to the Tarheel Hospital (hosp=3). The PROC SQL code shown below for Example 1a demonstrates how to do this when you just want to produce a 'report' or listing of the selected rows and columns – that is, no new data set (table) is produced.

In addition to the PROC SQL statement, which, of course, invokes the procedure, this simple example demonstrates a simple query (which always starts with the keyword SELECT) and two clauses. In the first part of the SELECT statement, we specify the columns that we want in our report. Note that they are separated by commas, which can always be a bit tricky for those of us used to just delimiting lists with spaces in the DATA step. The FROM clause, which is the only required clause in a SELECT statement, specifies the entity or entities (here a single data set) on which the SELECT statement is acting. And the WHERE clause, which is optional, places conditions on the rows that will be selected from the entities in the FROM clause – here specifying that we want the rows that have the HOSP variable with a value of 3. Note that although I've placed them on separate lines for clarity, both the FROM and WHERE clauses are part of the SELECT statement – and therefore, there is no semi-colon until the end of the WHERE clause. As an interactive procedure, the RUN statement has no meaning for SQL. A single PROC SQL statement can have multiple queries (SELECT statements). A step boundary is forced by the QUIT statement.

```
TITLE1 'SGF 2012 - Queries, Joins & Where Clauses - Demystifying SQL';
TITLE3 'Example 1 - Subsetting variables (columns) & observations (rows)';

TITLE4 '1a - Produce a "report" with desired columns and rows';
PROC SQL;
  SELECT pt_id, admdate, disdate
  FROM ex.admissions
  WHERE hosp EQ 3;
QUIT;
```

Assuming that you are sending your printed output to the listing destination, the code above will produce the output shown in Output 1a. Note that, unlike in PROC PRINT, which would be another way to produce a very similar report, PROC SQL will by default but the variable labels at the tops of the columns, rather than the variable names. If a variable has no label, the column header will be the variable name. If you do NOT want the labels at the tops of the

columns, you can use the SAS global option NOLABEL; this will stay in force until you reset with LABEL option. Additionally, SQL does not number the rows in the output by default; if you want row numbers, use the NUMBER option on the PROC SQL statement.

Example 1 - Subsetting variables (columns) & observations (rows)
1a - Produce a "report" with desired columns and rows

| Patient ID | Admit Date | Discharge Date |
|---------------|---------------|-------------------|
| 003 | 17OCT2010 | 21OCT2010 |
| 003 | 15NOV2010 | 15NOV2010 |
| 005 | 11APR2010 | 28APR2010 |
| 008 | 01OCT2010 | 15OCT2010 |
| 008 | 26NOV2010 | 28NOV2010 |
| 014 | 17JAN2011 | 20JAN2011 |
| 018 | 01NOV2010 | 15NOV2010 |
| 018 | 26DEC2010 | 08JAN2011 |

Output 1a. Result of Example 1a, selecting rows and columns

The code above requires only a small tweak if you wish to generate a new SAS data set with the desired rows and columns instead of a listing. As shown below, you simply add the CREATE TABLE clause before the SELECT, specifying the name you wish to give the new table after the TABLE keyword. The AS keyword is also required; in effect it says that the table name provided (here EX1) is an alias for the result of the subsequent query. When you use PROC SQL to generate a new data set in this way, there is no printed output generated. You could get a listing by either executing another simple SELECT query (on the new data set) or by a PROC PRINT step, as shown below.

There are two additional, somewhat subtle features of this simple program that are worth pointing out – ways in which SQL may differ a bit than how the DATA step operates. First, note that even though the selection of rows for the output data set is based on the values of the HOSP variable, this variable is not in the SELECT list and so it is not put on the resulting data set. You could certainly include it in the SELECT list if you wanted it on the new data set, but it is not required by SQL that you SELECT it in order to have the query use it for isolating the desired rows. The second feature is not obvious from the code – the order of the columns listed in the SELECT clause specifies the order that they will be on the output data set EX1 – not the sort order but the actual internal position of the columns on the data set, even if different from on the input data set EX.ADMISSIONS. This position typically doesn't really 'matter' in SAS, but sometimes it is handy to be able to specify the order of the columns on a data set, and SQL provides a straightforward way to do that.

```
TITLE4 '1b - Generate a data set with desired columns and rows';
PROC SQL;
CREATE TABLE ex1 AS
  SELECT pt_id, admdate, disdate
  FROM ex.admissions
  WHERE hosp EQ 3;
QUIT;

PROC PRINT DATA = ex1 N;
ID pt_id;
RUN;
```

STRUCTURE OF A QUERY

Before moving on to the next example, a brief tutorial on the general structure of an SQL query is in order. Additional details about the syntax of each of the clauses will be explained as we proceed through the examples. Figure 1 shows the clauses that can be part of an SQL query. There must be a SELECT, which is really the start of the query and that can be used without the CREATE clause to simply return rows to the designated output destination. The only clause that is required is the FROM clause – you must specify the source(s) FROM which you are SELECTing rows. If the any or all of the other clauses are present (and examples of each will be provided in this paper), they must be in the order shown.

```

PROC SQL;
  CREATE ... AS
  SELECT ...
  FROM ...
  WHERE ...
  GROUP BY...
  HAVING...
  ORDER BY... ;
QUIT;

```

Figure 1. General structure of a PROC SQL query, showing the order of the clauses. Only SELECT and FROM are required.

EXAMPLE 2: CREATING NEW COLUMNS/VARIABLES

In this set of examples we explore ways of creating new variables within PROC SQL and performing selection based on those variables. In Example 2a, we want to calculate a length of stay (LOS) for each hospital admission, and add that to a new data set along with a few other variables from the ADMISSIONS file. The code is shown below.

```

TITLE4 '2a: Calculate length of stay (LOS) for admissions';
PROC SQL;
CREATE TABLE los1 AS
  SELECT pt_id, hosp, admdate, disdate,
         (disdate-admdate) + 1 AS los
  FROM ex.admissions ;
QUIT;

```

The expression '(disdate – admdate) + 1' is evaluated and its result is stored in a column that we give the name LOS. The AS keyword is required and it basically says that LOS is an alias for the result of the expression. Note that while I chose to SELECT both ADMDATE and DISDATE to put on the output data set LOS, there is no requirement for this – PROC SQL can use columns coming from the data set(s) on the FROM clause in calculations whether or not they are put on (i.e. included in the SELECT clause) the report or table being generated. A partial listing of the output is shown in Output 2a.

| Example 2 - Creating new variables | | | | |
|---|----------|------------|----------------|-----|
| 2a: Calculate length of stay (LOS) for admissions | | | | |
| Patient ID | Hospital | Admit Date | Discharge Date | los |
| 001 | 01 | 07FEB2010 | 08FEB2010 | 2 |
| 001 | 01 | 12APR2010 | 25APR2010 | 14 |
| 001 | 02 | 10SEP2010 | 19SEP2010 | 10 |
| 001 | 05 | 19SEP2010 | 22SEP2010 | 4 |
| 003 | 03 | 17OCT2010 | 21OCT2010 | 5 |
| 003 | 03 | 15NOV2010 | 15NOV2010 | 1 |
| 004 | 02 | 18JUN2010 | 24JUN2010 | 7 |
| 005 | 01 | 19JAN2010 | 22JAN2010 | 4 |
| 005 | 01 | 10MAR2010 | 18MAR2010 | 9 |
| 005 | 02 | 10APR2010 | 11APR2010 | 2 |
| 005 | 03 | 11APR2010 | 28APR2010 | 18 |
| 006 | 05 | 11SEP2011 | 13SEP2011 | 3 |

Output 2a. (partial) listing of data set LOS1, created in Example 2a.

Demystifying SQL, *continued*

Example 2b demonstrates that variable attributes, such as LENGTH, LABEL, or FORMAT can be provided for the new variable, by listing them as shown, immediately after the name of the new column. Attributes can be added or modified for existing variables using the same type of syntax. Example 2a also introduces a new clause – ORDER BY, which specifies the sort order in the output data set. Specifically, the data set LOS2 will be sorted by ascending PT_ID and, within groups of rows for the same PT_ID, by the new LOS variable (from shortest to longest stay). Not only does the ORDER BY clause put the rows in the specified order, it also sets the sorted attribute for the resulting data set (just as a PROC SORT would) – which would be shown in PROC CONTENTS output and which can be used internally by SAS to prevent unnecessary sorting.

```
TITLE4 '2b: Calculate length of stay (LOS) for admissions';
TITLE5 '    Add attributes & sort';

PROC SQL;
CREATE TABLE los2 AS
    SELECT pt_id, hosp, admdate, disdate,
           (disdate-admdate) + 1 AS los LENGTH=4 LABEL='Length of Stay'
    FROM ex.admissions
    ORDER BY pt_id, los ;
QUIT;
```

In the next elaboration of this example (Example 2c), we select rows from the admissions table based on the value of the newly created LOS variable – specifically, we wish to output only those rows where the length of stay is at least 14 days. If you want the ORDER BY clause to sort the rows in a descending fashion (e.g. from longest stay to shortest), place the DESCENDING *after* the desired column name (i.e. the clause below would change to ORDER BY pt_id, los DESCENDING). Note that this is the opposite of in a BY statement in the DATA step or other PROCs.

```
TITLE4 '2c: Select admissions that are at least 2 weeks long';
TITLE5 '    Sort by patient ID and descending LOS';

PROC SQL;
CREATE TABLE twowks AS
    SELECT pt_id, hosp, admdate, disdate,
           (disdate-admdate) + 1 AS los LENGTH=4 LABEL='Length of Stay'
    FROM ex.admissions
       WHERE CALCULATED los GE 14
    ORDER BY pt_id, los DESCENDING;

SELECT * FROM twowks;
QUIT;
```

The second query in the Example 2c code (SELECT * FROM twowks;) will simply produce a listing of the entire TWOWKS data set; the '*' syntax is a wild card – short hand for all the columns in the FROM table(s). The complete output is shown in Output 2c. Note that for patient 018, the two records are in descending order of length of stay.

| Example 2 - Creating new variables | | | | | |
|--|----------|------------|----------------|----------------|--|
| 2c: Select admissions that are at least 2 weeks long | | | | | |
| Patient ID | Hospital | Admit Date | Discharge Date | Length of Stay | |
| 001 | 01 | 12APR2010 | 25APR2010 | 14 | |
| 005 | 03 | 11APR2010 | 28APR2010 | 18 | |
| 007 | 02 | 28JUL2010 | 10AUG2010 | 14 | |
| 008 | 03 | 01OCT2010 | 15OCT2010 | 15 | |
| 009 | 02 | 15DEC2010 | 04JAN2011 | 21 | |
| 018 | 03 | 01NOV2010 | 15NOV2010 | 15 | |
| 018 | 03 | 26DEC2010 | 08JAN2011 | 14 | |
| 020 | 01 | 08OCT2011 | 01NOV2011 | 25 | |

Output 2c. (partial) listing of data set TWOWKS1, created in Example 2c. Note that for patient 018, the two records are in descending order of length of stay.

Demystifying SQL, *continued*

Example 2d illustrates another way to create a new variable in PROC SQL – the CASE expression. In this example we want to create a variable called DXGRP that categorizes the primary diagnosis into one of three categories (myocardial infarction [MI], congestive heart failure [CHF] or other), based on the ICD-9 code (PRIMDX). This example also shows the LIKE syntax, which in conjunction with the '%' wild card, will assign any admission with a PRIMDX value beginning with '410' to a value for the DXGRP variable of 'MI' and any PRIMDX value beginning with '428' to DXGRP='CHF'; all other PRIMDX values (including any missing) to DXGRP = 'other'. The 'ELSE' portion of the CASE clause is not required, but is good practice. A partial listing of the resulting data set is shown in Output 2d.

```
TITLE4 '2d: Categorize admissions by diagnosis - CASE clause';

PROC SQL FEEDBACK;
CREATE TABLE grouping AS
  SELECT *,
    CASE
      WHEN primdx LIKE '410%' THEN 'MI'
      WHEN primdx LIKE '428%' THEN 'CHF'
      ELSE 'other'
    END AS dxgrp LABEL='Diagnosis Group'
  FROM ex.admissions;
QUIT;
```

A few other notes on the CASE expression. First, it cannot be used to assign new values to a variable that is also being selected – it won't generate an error but you will get a WARNING that the column already exists. What the warning doesn't tell you is that the values will not be changed – basically, SQL will not let you create a new column with the same name as an existing column. There are certainly ways to get around this limitation, but it is important to keep in mind. A second regards the length of a character variable created by the CASE expression. It will be the length required by the longest assigned value – 5 in this example. This is not dependent on the order of the data on the table being read or the order of the values in the CASE expression. I point this out because it is different from how length is assigned to a character variable getting its value from a series of IF/THEN statements in a DATA step.

Example 2 - Creating new variables
2d: Categorize admissions by diagnosis - CASE clause

| Patient ID | Admit Date | Primary diagnosis | Diagnosis Group |
|------------|------------|-------------------|-----------------|
| 001 | 07FEB2010 | 410 | MI |
| 001 | 12APR2010 | 428.2 | CHF |
| 001 | 10SEP2010 | 813.9 | other |
| 001 | 19SEP2010 | 428.4 | CHF |
| 003 | 17OCT2010 | 410.01 | MI |
| 003 | 15NOV2010 | 431 | other |
| 004 | 18JUN2010 | 434.1 | other |
| 005 | 19JAN2010 | 411.81 | other |
| 005 | 10MAR2010 | 410.9 | MI |
| 005 | 10APR2010 | 411 | other |
| 005 | 11APR2010 | 411 | other |
| 006 | 11SEP2011 | 820.01 | other |

Output 2d. Partial listing of data set GROUPING, produced by Example 2d. Only a subset of the rows and columns is shown.

Another useful option is demonstrated in Example 2d – the FEEDBACK option on the PROC SQL statement. This option results in the expansion of the query in the SAS log and is particularly useful in conjunction with the SELECT * syntax – especially when a TABLE has many columns on it. The log file for this example is shown below (Log 2d). Note how we get a listing of all the columns that are being selected, and the two-level names also show what table they are coming from – you can see how this could be even more helpful when your query is operating on multiple tables, as we'll see in later examples.

Demystifying SQL, *continued*

```

63     TITLE4 '2d: Categorize admissions by diagnosis - CASE clause';
64     PROC SQL FEEDBACK;
65     CREATE TABLE grouping AS
66     SELECT *,
67     CASE
68     WHEN primdx LIKE '410%' THEN 'MI'
69     WHEN primdx LIKE '428%' THEN 'CHF'
70     ELSE 'other'
71     END AS dxgrp LABEL='Diagnosis Group'
72     FROM ex.admissions;
NOTE: Statement transforms to:

        select ADMISSIONS.pt_id, ADMISSIONS.admdate, ADMISSIONS.disdate,
ADMISSIONS.md,
ADMISSIONS.hosp, ADMISSIONS.dest, ADMISSIONS.bp_sys, ADMISSIONS.bp_dia,
ADMISSIONS.primdx, case
                when ADMISSIONS.primdx like '410%' then 'MI'
                when ADMISSIONS.primdx like '428%' then 'CHF'
                else 'other'
                end as dxgrp label='Diagnosis Group'
        from EX.ADMISSIONS;

NOTE: Table WORK.GROUPING created, with 30 rows and 10 columns.

```

Log 2d. Log generated by Example 2d, demonstrating the expansion of the query resulting from the FEEDBACK option.

Finally, in Example 2e, I illustrate what I call a “mixed method” in which we take advantage of features of both PROC SQL and “regular” SAS – i.e. some DATA SET options, which are not ‘standard ANSI SQL’ but work just fine in PROC SQL. I especially like the way the SELECT * syntax, used in conjunction with the ‘DROP=’ option operating on the FROM data set allows you to easily KEEP most of the variables on the input dataset. The colon after ‘bp’ is also a handy trick (which, of course, works anywhere you can use a DATA set option) to specify (here, to DROP) all the variables that start with whatever comes before the colon. In this example, it allows us to drop BP_SYS and BP_DIA.

```

TITLE4 '2e: Categorize admissions by diagnosis - using DATA set options';

PROC SQL FEEDBACK;
CREATE TABLE grouping2a (RENAME = (primdx=ICD9)) AS
  SELECT * ,
  CASE
    WHEN primdx LIKE '410%' THEN 'MI'
    WHEN primdx LIKE '428%' THEN 'CHF'
    ELSE 'other'
  END AS dxgrp
  FROM ex.admissions (DROP = bp:) ;
QUIT;

```

EXAMPLE 3: SUMMARY FUNCTIONS

Summary functions are extremely useful in SQL. Sometimes, tasks that would take a few DATA steps and a PROC SUMMARY or two can be achieved quite concisely with a single SQL query. However, the syntax and way that these sometimes complex queries work, can be a little daunting, especially if your SAS brain is trained to think like the DATA step. We’ll walk through a number of examples, in hopes of gaining some insights into this type of coding.

The first example with summary functions is a very simple one, but it illustrates a handy technique. The following code (Example 3a) will produce a one-row ‘report’ that lists the dates of the earliest and latest admissions and discharges on the admissions table. Because ADMDATE and DISDATE are SAS dates, the MIN and MAX functions will return the earliest and latest dates, respectively. Unlike the MIN and MAX functions in the DATA step, these operate more like the MIN and MAX keywords in procedures like MEANS and SUMMARY; that is, they operate across rows in the table (rather than across columns within a row). Note that although ADMDATE and DISDATE themselves are formatted as DATE9., this formatting is not inherited by the newly created variables, but can be

Demystifying SQL, *continued*

supplied as shown. I didn't bother to provide labels, but you certainly could. The one-line report produced is shown below, in Output 3a.

```
TITLE3 'Example 3 - Summary Functions';
TITLE4 '3a: Earliest and latest admission and discharge dates';
PROC SQL ;
  SELECT MIN(admdate) AS adm_f FORMAT=date9.,
         MAX(admdate) AS adm_l FORMAT=date9.,
         MIN(disdate) AS dis_f FORMAT=date9.,
         MAX(disdate) AS dis_l FORMAT=date9.
  FROM ex.admissions ;
QUIT;
```

Example 3 - Summary Functions
3a: Earliest and latest admission and discharge dates

| adm_f | adm_l | dis_f | dis_l |
|-----------|-----------|-----------|-----------|
| 19JAN2010 | 30NOV2011 | 22JAN2010 | 06DEC2011 |

Output 3a. Result of Example 3a, showing earliest and latest admissions.

The next program, Example 3b, demonstrates a similar technique; however, in this case, we introduce the GROUP BY clause in order to get the birthdates of the youngest and oldest patients, by gender. The date of birth (DOB) is on the PATIENTS data set. Note that the PATIENTS data set is not sorted by SEX; this does not cause a problem for the GROUP BY clause. The FORMATTING of the sex variable is not required but it makes the output (shown in Output 3b) a bit clearer; the output has one row for each unique value of the GROUP BY variable. This example also illustrates the N function, which counts the non-missing values of the variable in its argument (here the patient ID). This function does NOT eliminate duplicates – in other words if a given ID had multiple records in the PATIENTS table, it would be counted twice. If you want to count unique values with the N function, add the DISTINCT keyword before the argument (e.g. N(DISTINCT id)).

```
TITLE4 '3b: Earliest and latest birth dates of patients by gender';
PROC FORMAT;
VALUE sexf
  1 = 'Men'  2 = 'Women';
RUN;
PROC SQL ;
  SELECT sex FORMAT=sexf. ,
         MIN(dob) as first_dob FORMAT=date9.,
         MAX(dob) as last_dob  FORMAT=date9.,
         N(id) as N_patients
  FROM ex.patients
  GROUP BY sex ;
QUIT;
```

Example 3 - Summary Functions
3b: Earliest and latest birth dates of patients by gender

| Gender | first_dob | last_dob | N_patients |
|--------|-----------|-----------|------------|
| Men | 17JUN1923 | 09NOV1950 | 11 |
| Women | 04NOV1926 | 14OCT1948 | 9 |

Output 3b. Result of Example 3b, showing earliest and latest birth dates for men and women in the PATIENTS data set.

Demystifying SQL, *continued*

There are a few other features worth noting about GROUP BY in PROC SQL. As mentioned above, the result includes a row for each value of the GROUP BY variable. If there were rows on the PATIENTS table that had a missing value for SEX, the report would include a row for these cases also (like invoking the MISSING option in PROC SUMMARY/MEANS). If you wished to exclude the values with missing SEX in the PROC SQL code, you could add a WHERE clause (i.e. WHERE sex ne .), recalling that the WHERE clause has to come after the FROM clause and before the GROUP BY.

The next example, Example 3c, illustrates the use of more than one GROUP BY variable. This program creates a new data set called ADMSUM that has some summary information about admissions, grouped by patient and hospital. Here we see that the argument to functions such as MAX and MEAN can be an expression. Note that, as usual for lists in PROC SQL, the GROUP BY variables are separated by commas. The resulting data set, a portion of which is shown in Output 3c, includes a row for all the existing combinations of the GROUP BY columns on the input data set – here, PT_ID and HOSP on the ADMISSIONS table. This example also introduces the COUNT(*) syntax, which will count the number of rows that have been summarized for each grouping of the GROUP BY variables, regardless of missing values.

Another crucial point about this code is that while summary functions operate on the ADMDATE and DISDATE variables, creating average length of stay (AVGLOS) and longest length of stay (MAXLOS) for each patient-hospital combination, the ADMDATE and DISDATE variables themselves are NOT selected. Indeed, if they were, a very different result would be produced. Specifically, note that in this code – and in the previous examples in this section, the ONLY variables that are selected (either for a report or a new data set) are the GROUP BY variable(s) and variables that are the result of summary functions. We will re-visit this in later exercises.

```
TITLE4 '3c: Info on Admissions, by patient and hospital';

PROC SQL;
  CREATE TABLE admsum AS
    SELECT pt_id,
           hosp,
           COUNT(*) AS nstays,
           MAX(disdate - admdate + 1) AS maxlos,
           MEAN(disdate - admdate + 1) AS avglos
    FROM ex.admissions
    GROUP BY pt_id, hosp ;
QUIT;
```

Example 3 - Summary Functions
3c: Info on Admissions, by patient and hospital

| pt_id | hosp | nstays | maxlos | avglos |
|-------|------|--------|--------|--------|
| 001 | 01 | 2 | 14 | 8.0 |
| | 02 | 1 | 10 | 10.0 |
| | 05 | 1 | 4 | 4.0 |
| 003 | 03 | 2 | 5 | 3.0 |
| 004 | 02 | 1 | 7 | 7.0 |
| 005 | 01 | 2 | 9 | 6.5 |
| | 02 | 1 | 2 | 2.0 |
| | 03 | 1 | 18 | 18.0 |
| 006 | 05 | 1 | 3 | 3.0 |

Output 3c. Partial listing of the data set ADMSUM, generated by Example 3c.

Demystifying SQL, *continued*

EXAMPLE 4: MORE ON SUMMARY FUNCTIONS - SELECTION

We set up the next series of examples with one that is not too different from what we've just seen. Example 4a determines the longest LOS for each patient and orders the output data from longest to shortest MAXLOS. The MAXLOS data set has 15 rows (one for each patient that has an admission); a listing of the first 10 rows is shown in Output 4a.

```
TITLE3 'Example 4 - More on summary functions - Selection ';
TITLE4 '4a: Determine longest hospital stay for each patient';

PROC SQL;
  CREATE TABLE maxlos AS
  SELECT pt_id,
         COUNT(*) AS nstays,
         MAX(disdate - admdate + 1) AS maxlos
  FROM ex.admissions
  GROUP BY pt_id
  ORDER BY maxlos DESCENDING;
QUIT;
```

Example 4 - More on summary functions - Selection
4a: Determine longest hospital stay for each patient

| Obs | pt_id | nstays | maxlos |
|-----|-------|--------|--------|
| 1 | 020 | 2 | 25 |
| 2 | 009 | 1 | 21 |
| 3 | 005 | 4 | 18 |
| 4 | 018 | 2 | 15 |
| 5 | 008 | 4 | 15 |
| 6 | 001 | 4 | 14 |
| 7 | 007 | 2 | 14 |
| 8 | 015 | 2 | 13 |
| 9 | 004 | 1 | 7 |
| 10 | 010 | 1 | 7 |

Output 4a. Listing of the first 10 rows of MAXLOS, generated by Example 4a.

The next example, Example 4b, extends this by selecting those patients whose longest admission is at least 14 days long. To do this, we use a HAVING expression for the first time. Note that ALL that has been changed from Example 4a is the addition of the HAVING expression. HAVING operates on the groups defined by the GROUP BY clause. Based on the result of Example 4a, we see that we should identify 7 patients who have an admission that is at least 14 days long. They are listed in Output 4b.

```
TITLE4 '4b: Identify patients with an admission at least 2 weeks long';
PROC SQL;
  CREATE TABLE twoweeks AS
  SELECT pt_id,
         COUNT(*) AS nstays,
         MAX(disdate - admdate + 1) AS maxlos
  FROM ex.admissions
  GROUP BY pt_id
  HAVING maxlos GE 14
  ORDER BY maxlos DESCENDING ;
QUIT;
```

Example 4 - More on summary functions - Selection
 4b: Identify patients with an admission at least 2 weeks long

| pt_id | nstays | maxlos |
|-------|--------|--------|
| 020 | 2 | 25 |
| 009 | 1 | 21 |
| 005 | 4 | 18 |
| 018 | 2 | 15 |
| 008 | 4 | 15 |
| 001 | 4 | 14 |
| 007 | 2 | 14 |
| N = 7 | | |

Output 4b. Listing of patients who have an admission at least 14 days long.

We take this one step further to help illustrate the distinction between WHERE and HAVING. In Example 4c, we wish to identify the patients who have an admission at least 14 days long in 2010. To do this, we simply add a WHERE clause to the code from Example 4b. WHERE acts on individual rows, essentially specifying which rows on the FROM table will be summarized using the GROUP BY columns. HAVING then dictates which of the summarizations (i.e. groups) will be selected for output.

```
TITLE '4c: Identify patients with an admission at least 2 weeks long in 2010';

PROC SQL;
  CREATE TABLE twowks2010 AS
  SELECT pt_id,
         COUNT(*) AS nstays,
         MAX(disdate - admdate + 1) AS maxlos
  FROM ex.admissions
  WHERE year(admdate) = 2010
  GROUP BY pt_id
  HAVING maxlos GE 14
  ORDER BY pt_id ;
QUIT;
```

Example 4 - More on summary functions - Selection
 4c: Identify patients with an admission at least 2 weeks long in 2010

| pt_id | nstays | maxlos |
|-------|--------|--------|
| 001 | 4 | 14 |
| 005 | 4 | 18 |
| 007 | 2 | 14 |
| 008 | 4 | 15 |
| 009 | 1 | 21 |
| 018 | 2 | 15 |
| N = 6 | | |

Output 4c. Listing of patients who have an admission at least 14 days long in 2010.

EXAMPLE 5: STILL MORE ON SUMMARY FUNCTIONS – RE-MERGE

In all of the examples using summary functions so far, the only columns that are selected are either those in the GROUP BY (i.e. variables required to specify the groups being acted upon by summary functions) or the summary variables themselves. Sometimes this is not enough. Recall Example 3b, in which we identified the earliest and latest

Demystifying SQL, *continued*

birthdates for men and women. Suppose that we don't just want to find the first and last birthdates but that we also want to get information about the patients having the first and last birthdates – such as their names, or patient ID's, along with the birthdate,. A first stab at the code to do this might be as shown below, in which we simply add another column (FIRSTNAME) to the SELECT clause from Example 3b.

```
TITLE4 '5b (3B redux): Earliest and latest birth dates of patients by gender';

PROC SQL ;
CREATE TABLE oldyoung AS
SELECT sex FORMAT=sexf. ,
       firstname,
       dob,
       MIN(dob) as first_dob FORMAT=date9.,
       MAX(dob) as last_dob  FORMAT=date9.,
       N(id) as N_patients
FROM ex.patients
GROUP BY sex ;
QUIT;
```

Recall that without the columns FIRSTNAME and DOB in the SELECT clause, this query produced 2 rows, one for women and one for men (see Output 3b). While we might expect (hope) that the Example 5b code would simply augment those two rows with two columns telling us the FIRSTNAME and DOB of those patients, this is not what happens. Instead the table OLDYOUNG includes 20 rows – one for each patient. A portion of the records are shown in Output 4b. Note that we also get the following message in the log, indicating a REMERGE has occurred:

```
NOTE: The query requires remerging summary statistics back with the original
data.
NOTE: Table WORK.OLDYOUNG2 created, with 20 rows and 6 columns.
```

You will get this re-merge note any time you use a GROUP BY clause and select columns other than the GROUP BY columns and columns that are the result of summary functions. Output 4b shows that the values summary variables (FIRST_DOB, LAST_DOB and N_PATIENTS) have been copied across all rows within the GROUP BY group. Clearly we need to do something differently to get just the youngest and oldest patients, with the individual and summary information.

| Example 5 - More on summary functions - RE-MERGE | | | | | |
|--|-----------|-----------|-----------|-----------|------------|
| 5b: First try - Determine FIRST NAME of oldest and youngest patients | | | | | |
| sex | firstname | dob | first_dob | last_dob | N_patients |
| Men | Albert | 16JUN1935 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Adam | 17APR1935 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Anthony | 12APR1929 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Henrik | 09NOV1950 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Shelby | 13FEB1940 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Lars | 07FEB1938 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Hugh | 10AUG1931 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Riley | 03AUG1946 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Michael | 02JUL1927 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Geoffrey | 25MAY1925 | 17JUN1923 | 09NOV1950 | 11 |
| Men | Antonio | 17JUN1923 | 17JUN1923 | 09NOV1950 | 11 |
| Women | Hannah | 06JUN1937 | 04NOV1926 | 14OCT1948 | 9 |
| Women | Karen | 04NOV1926 | 04NOV1926 | 14OCT1948 | 9 |
| Women | Josephine | 14OCT1948 | 04NOV1926 | 14OCT1948 | 9 |

Output 5b. Partial listing of results from Example 5b . The boxes highlight the rows for the men with the earliest and latest birth dates.

Demystifying SQL, *continued*

The code shown in Example 5c improves upon the 5b version by creating boolean indicators of whether the date of birth on an individual row matches the minimum or maximum birthdate for the GROUP BY group (i.e. for men or women) and then using HAVING to select only the rows meeting either the OLDEST or YOUNGEST criteria. We still get the "REMERGE" note in the log, but the resulting table (shown in Output 5c) has just 4 rows as desired.

```
TITLE4 '5c: Better - Determine FIRST NAME of oldest and youngest patients';

PROC SQL ;
CREATE TABLE oldyoung3 AS
SELECT  firstname,
        sex FORMAT=sexf. ,
        dob,
        (dob = MIN(dob)) as oldest ,
        (dob = MAX(dob)) as youngest
FROM ex.patients
GROUP BY sex
HAVING (oldest = 1 or youngest = 1)
ORDER BY sex, dob ;
QUIT;
```

Example 5 - More on summary functions - RE-MERGE
5c: Better - Determine FIRST NAME of oldest and youngest patients

| firstname | sex | dob | oldest | youngest |
|-----------|-------|-----------|--------|----------|
| Antonio | Men | 17JUN1923 | 1 | 0 |
| Henrik | Men | 09NOV1950 | 0 | 1 |
| Karen | Women | 04NOV1926 | 1 | 0 |
| Josephine | Women | 14OCT1948 | 0 | 1 |

Output 5c. Listing of the data set produced by Example 5c, showing the first name and date of birth for the oldest and youngest male and female patients.

A variation on this method is shown in Example 5d, which uses the CASE clause. Output 5d shows that the same rows are selected by 5c and 5d.

```
TITLE4 '5d: Also Better - Determine FIRST NAME of oldest and youngest patients';

PROC SQL ;
CREATE TABLE oldyoung4 AS
SELECT  firstname,
        sex FORMAT=sexf. ,
        dob,
        CASE
            WHEN dob = MIN(dob) THEN 'OLDEST'
            WHEN dob = MAX(dob) THEN 'YOUNGEST'
            ELSE '?'
        END AS old_young LABEL='Oldest or youngest'
FROM ex.patients
GROUP BY sex
HAVING old_young in ('OLDEST','YOUNGEST')
ORDER BY sex, dob ;
QUIT;
```

Example 5 - More on summary functions - RE-MERGE
 5d: Also Better - Determine FIRST NAME of oldest and youngest patients

| Patient First Name | Gender | Patient Date of Birth | Oldest or youngest |
|--------------------------|--------|-----------------------------|--------------------------|
| Antonio | Men | 17JUN1923 | OLDEST |
| Henrik | Men | 09NOV1950 | YOUNGEST |
| Karen | Women | 04NOV1926 | OLDEST |
| Josephine | Women | 14OCT1948 | YOUNGEST |

Output 5d. Listing of the data set produced by Example 5d, showing an alternative method to retrieve the first name and date of birth for the oldest and youngest ages

EXAMPLE 6: INLINE VIEWS AND SUBQUERIES

We introduce the next technique by attempting to expand upon the types of queries we were working with in the last set of examples – that is, combining detail and summary information. Let's say we want to identify the patient with the most admissions. We might try the following code (Example 6a). With the COUNT(*) syntax we are counting the number of rows in each GROUP BY group (i.e. hospital stays per patient), and then, in the HAVING clause, we are *trying* to select the group(s) with the largest number of stays...

```
TITLE4 '6a: Try to Identify patient(s) with most admissions';
TITLE5 'Unfortunately this does not work';

PROC SQL;
  CREATE TABLE mostadmits0 AS
  SELECT pt_id,
         COUNT(*) AS nstays
  FROM ex.admissions
  GROUP BY pt_id
  HAVING nstays = MAX(nstays) ;
QUIT;
```

Unfortunately, this code does not work. Instead it generates an error, telling us

```
ERROR: Summary functions nested in this way are not supported.
```

The level of aggregation needed for the two summary functions is different – COUNT is operating at the level of GROUPS (stays within patients) while we are asking MAX to operate across groups; this is not possible within the same SELECT. So, we need to try another tactic. The message regarding the nesting of summary functions suggests one strategy – break the task into two queries. This is precisely what the code in Example 6b does.

```
TITLE4 '6b: Try to Identify patient(s) with most admissions';
TITLE5 'Two-step process - DOES work';

PROC SQL;
  CREATE TABLE numstays AS
  SELECT pt_id,
         COUNT(*) AS nstays
  FROM ex.admissions
  GROUP BY pt_id;

  CREATE TABLE mostadmits1 AS
  SELECT * FROM numstays
  HAVING nstays = MAX(nstays) ;
QUIT;
```

Demystifying SQL, *continued*

The first query counts the number of stays per patient, while the second (note there is no GROUP BY because the aggregation happened in the first query) selects the patient (or patients) whose number of stays is equal to the largest number of stays. This pair of queries generates the desired output – Output 6b. The first part shows the intermediate step, printing the NUMSTAYS data set; the second part shows the final desired result – a record for each of the patients with the largest number of admissions.

In Example 6b, you might think that you could use WHERE instead of having in the second query, but this would generate an error, telling you that “Summary functions are restricted to the SELECT and HAVING clauses only”. This is a key distinction between HAVING, which works on GROUPS and with summary functions, whereas WHERE works on a row-by-row basis on the FROM table(s).

| 6b: Try to Identify patient(s) with most admissions | | |
|--|-------|--------|
| Two-step process - Step 1 - Count admits (NSTAYS) for each patient | | |
| Obs | pt_id | nstays |
| 1 | 001 | 4 |
| 2 | 003 | 2 |
| 3 | 004 | 1 |
| 4 | 005 | 4 |
| 5 | 006 | 1 |
| 6 | 007 | 2 |
| 7 | 008 | 4 |
| 8 | 009 | 1 |
| 9 | 010 | 1 |
| 10 | 012 | 2 |
| 11 | 014 | 1 |
| 12 | 015 | 2 |
| 13 | 016 | 1 |
| 14 | 018 | 2 |
| 15 | 020 | 2 |

| Two-step process - Step 2 - Select patient(s) with largest number of stays | | |
|--|-------|--------|
| Obs | pt_id | nstays |
| 1 | 001 | 4 |
| 2 | 005 | 4 |
| 3 | 008 | 4 |

Output 6b. Results of each of the two steps used in Example 6b.

Still, can we improve upon this? Our first attempt at doing this in one step did not work, but perhaps there is another way...in fact, there is. Although we cannot nest summary functions as we tried in Example 6a, we can nest queries...so, this is the strategy of Example 6c.

```
TITLE5 '6c: Identify patient(s) with most admissions';
TITLE4 'Inline view';
PROC SQL;
CREATE TABLE mostadmits2 AS
SELECT *
FROM
(SELECT pt_id,
COUNT(*) AS nstays
FROM ex.admissions
GROUP BY pt_id)
HAVING nstays = MAX(nstays) ;
QUIT;
```

Demystifying SQL, *continued*

If you compare the code in Example 6c to that in Example 6b, you will observe that we have basically transplanted the first query (counting the stays for each patient) into the FROM clause of the second query and enclosed it within parenthesis. This is perfectly valid – the argument to the FROM clause can itself be another query. A key difference here is that we are not generating the intermediate table (called NUMSTAYS in Example 6b). The result is shown in Output 6c, which is identical to the final result of 6b. This method is called an “Inline View” – note that we are basically creating a virtual table (a “view”) with the inner query (between the parentheses), that is then treated as the object from which the main or outer query is selecting rows. In effect, the in-line view is pre-processing the ADMISSIONS table before it is acted upon by the main query.

| Example 6 -- Inline views & Subqueries | |
|--|--------|
| 6c: Identify patient(s) with most admissions - Inline view | |
| pt_id | nstays |
| 001 | 4 |
| 005 | 4 |
| 008 | 4 |

Output 6c. Results of Example 6c, using an inline view to identify the patients with the most hospital admissions

A somewhat similar technique to the inline view is the subquery, and we shift gears here slightly to illustrate this. Recall Example 1 in which we were selecting all admissions to the Tarheel Hospital, which is identified on the ADMISSIONS table with a HOSP value of 3. Suppose instead that we didn't know the hospital ID, but still wanted to get the Tarheel admissions; yet, the hospital name is not on the ADMISSIONS table – it is on the HOSPITALS table. To do the selection of rows from the ADMISSIONS table based on information in the HOSPITALS table, we can employ a subquery, as shown in Example 6d below.

```
TITLE4 '6d - Selecting rows based on information from another table (subquery)';
TITLE5 'Admissions to Tarheel hospital';

PROC SQL ;
  CREATE TABLE adm_tarheel AS
    SELECT *
  FROM ex.admissions
    WHERE hosp IN
      (SELECT hosp_id
       FROM ex.hospitals
        WHERE hospname = 'Tarheel')
  ORDER BY pt_id, admdate ;
QUIT;
```

A subquery is a query-expression that is nested within another query-expression. The value of the hospital identifier (HOSP) on the ADMISSIONS data set is compared to the result of a subquery of the HOSPITALS data set. Using IN (rather than EQ or =) in the WHERE clause allows for the possibility that the subquery might return more than a single value (i.e. if more than one HOSP_ID was associate with the name “Tarheel” or the criteria more obviously would select more than one row); the return of multiple rows from the subquery would cause an error if you used EQ or =. Note that no columns are added to the resulting table from the HOSPITALS data set – a JOIN, which we'll get to in the next set of examples would be required if, for example, we wanted to add hospital characteristics to the ADMISSIONS data. Additionally, no explicit sorting is required for the subquery to work (as would be the case in a DATA Step MERGE). The ORDER BY clause dictates the sort order of the output data set. The output, shown in Output 6d, selects the same 8 rows from the admissions table as we did in Example 1A.

```
Example 6 -- Inline views & Subqueries
6d - Selecting rows based on information from another table (subquery)
Admissions to Tarheel hospital
```

| pt_id | admdate | disdate | md | hosp | dest | bp_sys | bp_dia | primdx |
|-------|-----------|-----------|------|------|------|--------|--------|--------|
| 003 | 17OCT2010 | 21OCT2010 | 8081 | 03 | 1 | 155 | 92 | 410.01 |
| 003 | 15NOV2010 | 15NOV2010 | 2322 | 03 | 9 | 74 | 40 | 431 |
| 005 | 11APR2010 | 28APR2010 | 7803 | 03 | 1 | 145 | 91 | 411 |
| 008 | 01OCT2010 | 15OCT2010 | 3274 | 03 | 1 | 145 | 74 | 820.8 |
| 008 | 26NOV2010 | 28NOV2010 | 2322 | 03 | 2 | 135 | 76 | V54.8 |
| 014 | 17JAN2011 | 20JAN2011 | 7803 | 03 | 1 | 162 | 93 | 414.1 |
| 018 | 01NOV2010 | 15NOV2010 | 1972 | 03 | 2 | 170 | 88 | 428.1 |
| 018 | 26DEC2010 | 08JAN2011 | 1972 | 03 | 2 | 199 | 93 | 428.1 |

Output 6d. Listing of results of query in Example 6d, which uses a subquery to identify records in one table based on information in another table.

Compare the general syntax of the code from 6c (inline view) and 6d (subquery). Both have a “query within a query”, but how do they differ? First, an inline view serves as a substitute for a table reference in the FROM clause – remember we used it to replace the actual creation of an intermediate table, while the subquery is in the WHERE clause and is referencing a different table. A second important difference is that we are not using the subquery to add columns to the table we are creating; indeed, a subquery can request only a single column. In contrast, the inline view can retrieve multiple columns.

EXAMPLE 7: INNER JOINS OF TWO TABLES

Now we begin more explicitly to combine information from multiple tables using SQL joins. Let’s say we want to augment the information in the ADMISSIONS table with some additional patient-level information, found in the PATIENTS table. A DATA Step-per would think MERGE; in PROC SQL, think INNER JOIN. Here’s the code.

```
TITLE3 'Example 7 - Inner Joins (two tables) ';
TITLE4 '7a - Combining patient info with hospital admission info';

PROC SQL ;
CREATE TABLE adm_pt1 AS
SELECT a.pt_id,
       a.admdate,
       a.disdate,
       a.hosp,
       a.md,
       b.dob,
       b.sex,
       b.primmd
FROM ex.admissions AS a
     INNER JOIN
     ex.patients AS b
     ON a.pt_id = b.id
ORDER BY a.pt_id, a.admdate ;
QUIT;
```

The INNER JOIN syntax says explicitly that the only records that will be delivered from the query (here – put into the new table ADM_PT1) are those where a record is coming in from both the ADMISSIONS table and the PATIENTS table. In other words, if there admissions for patients not found in the PATIENTS table or patients without hospital stays in the ADMISSIONS table, these records will not contribute any information to the resulting ADM_PT1 table. The join criteria (i.e. which records constitute a ‘match’ in this case) is specified in the ON condition – here, that the PT_ID value on the ADMISSIONS table matches the ID value on the PATIENTS table. In this example, we have a one-to-many join; that is, a given patient (ID) in the PATIENT table can have multiple matches in the ADMISSIONS table (more than one admit per PT_ID); this is no problem – the patient info (sex,dob, and primmd) get added for all the matching admissions. If there were multiple records for a patient in both tables, all possible combinations of

Demystifying SQL, *continued*

these records would be placed in the resulting table – this might or might not be what we want, but it is predictable – and different from what would happen in a DATA Step MERGE in the same situation.

Note that the columns in the SELECT clause are a mix of variables coming from ADMISSIONS and those coming from PATIENTS. The two level column names (e.g. a.admdate, b.dob) are used to specify which table each column is coming from. The letters 'a' and 'b' are associated with EX.ADMISSIONS and EX.PATIENTS in the FROM clause with the 'AS a' and 'AS b' syntax respectively. This aliasing is not required; it just simplifies the code and reduces typing. Instead of referring to 'a.pt_id' and 'b.dob', you could forego the aliasing and refer to *admissions.pt_id* and *patients.dob*; likewise for all the other column names in the code. Further, the only time two level names for columns (aliased or not) are *required* is when there are columns of the same name on two or more tables in the join (whether or not they are being selected or are part of the join criteria) – because in that case SQL needs to know which one you are referring to. So, in this particular example, the two-level names are not needed because the two tables have no variable names in common, BUT I highly recommend the practice as it does make the code clearer.

There are a few other features of this simple join to highlight. First, note that although the ID variable on the patients table is part of the JOIN criteria, we are not SELECTing it onto the query result. You could if you wanted to, but it is not necessary, and here would be redundant with PT_ID from ADMISSIONS. Second, as we've pointed out before in SQL, there is no explicit SORT step required – the two data sets do NOT need to be in order by the variable (or variables) on which you are JOINing; the ORDER BY included here is simply to specify the order we want the rows in our new, composite table. This doesn't mean that SAS is not having to do some sorting in the background to perform the join; you simply don't have to have a separate SORT step before running this join. Finally, I point out that the order in which the columns are listed in the SELECT clause does dictate the order in which they will be placed in the output table; this feature of a join is useful if you have reason to care about the order of columns in the joined table. The output is in Output 7a.

| Example 7 - Inner Joins (two tables) | | | | | | | |
|--|-----------|-----------|------|------|-----------|-----|--------|
| 7a - Combining patient info with hospital admission info | | | | | | | |
| pt_id | admdate | disdate | hosp | md | dob | sex | primmd |
| 001 | 07FEB2010 | 08FEB2010 | 01 | 3274 | 10AUG1931 | 1 | 1972 |
| 001 | 12APR2010 | 25APR2010 | 01 | 1972 | 10AUG1931 | 1 | 1972 |
| 001 | 10SEP2010 | 19SEP2010 | 02 | 3274 | 10AUG1931 | 1 | 1972 |
| 001 | 19SEP2010 | 22SEP2010 | 05 | 3274 | 10AUG1931 | 1 | 1972 |
| 003 | 17OCT2010 | 21OCT2010 | 03 | 8081 | 02JUL1927 | 1 | 8081 |
| 003 | 15NOV2010 | 15NOV2010 | 03 | 2322 | 02JUL1927 | 1 | 8081 |
| 004 | 18JUN2010 | 24JUN2010 | 02 | 7803 | 25MAY1925 | 1 | 4003 |
| 005 | 19JAN2010 | 22JAN2010 | 01 | 1972 | 31AUG1940 | 2 | 1972 |
| 005 | 10MAR2010 | 18MAR2010 | 01 | 1972 | 31AUG1940 | 2 | 1972 |
| 005 | 10APR2010 | 11APR2010 | 02 | 1972 | 31AUG1940 | 2 | 1972 |
| 005 | 11APR2010 | 28APR2010 | 03 | 7803 | 31AUG1940 | 2 | 1972 |
| 006 | 11SEP2011 | 13SEP2011 | 05 | 8081 | 12APR1929 | 1 | 2322 |
| 007 | 28JUL2010 | 10AUG2010 | 02 | 3274 | 07FEB1938 | 1 | 3274 |
| 007 | 08SEP2010 | 15SEP2010 | 02 | 8081 | 07FEB1938 | 1 | 3274 |
| 008 | 13APR2010 | 19APR2010 | 02 | 1972 | 09NOV1942 | 2 | 4003 |

Output 7a. Partial listing of table ADM_PT1, created by an Inner join of ADMISSIONS and PATIENTS

The next example (Example 7b) is presented just to show an alternative form of the INNER JOIN that you may encounter – it doesn't include the keyword JOIN at all, but produces exactly the same result as Example 7a. It employs an implicit join, specifying the matching criteria on the WHERE clause rather than the ON condition of the explicit JOIN in Example 7a. Also, note the comma between the two data sets in the FROM clause – easy to forget!

Demystifying SQL, *continued*

```

TITLE3 'Example 7 - Inner Joins (two tables) ';
TITLE4 '7b - Combining patient and hospital admission info (alternative code)';

PROC SQL ;
CREATE TABLE adm_pt2 AS
  SELECT a.pt_id,
         a.admdate,
         a.disdate,
         a.hosp,
         a.md,
         b.dob,
         b.sex,
         b.primmd
FROM ex.admissions a,
     ex.patients AS b
   WHERE a.pt_id = b.id
ORDER BY a.pt_id, admdate ;
QUIT;

```

The output is not included again. It is identical to Output 7a.

EXAMPLE 8: MORE INNER JOINS – MORE THAN TWO TABLES

Inner joins are not limited to two tables as we'll see in this next set of examples. For Example 8a, our objective is to identify patients who died in the hospital, and determine their age at death and the size of the hospital in which they died. This requires information from three tables – the ADMISSIONS table will tell us which hospitalizations ended in death (DEST = 9) as well as the discharge date, which we need for computing age; the PATIENTS table to obtain date of birth (DOB), also needed for computing age; and the HOSPITALS table, to determine hospital size (BEDS). Here we use the implicit join method, listing the three tables (separated by commas, of course) in the FROM clause and the join criteria in the WHERE clause. Specifically, as it is an inner join, we require that the patient ID on the ADMISSIONS table (PT_ID) matches the patient ID on the PATIENTS table (ID) as well as that the hospital identifier on the ADMISSIONS table (HOSP) matches the hospital identifier on the HOSPITALS table (HOSP_ID). Further, to select only admissions where the patient died, we also specify that DEST=9 in the WHERE clause.

Again, I point out a few other parts of this code. The “a.disdate AS dthdate” syntax essentially renames the discharge date variable to make it clear it is the date of death; using this syntax, the DTHDATE variable inherits the attributes of DISDATE, including its FORMAT (DATE9.) and its label – so we assign a new label. Additionally, in the SELECT clause we can do calculations to create new variables, based on columns from more than one table – computing age at death in this example; note, that I didn't have to include DOB on the resulting table (but you could). Also, as we've seen before, it is not necessary to select the DEST variable, even though it is used in the WHERE clause.

```

TITLE3 'Example 8 - More Inner Joins ';
TITLE4 '8a - Inner Join of 3 tables + row selection';
TITLE5 'Identify Patients who died in hospital, determining age & hospital size';

PROC SQL ;
CREATE TABLE deceased1 AS
SELECT a.pt_id,
       a.hosp,
       a.disdate AS dthdate LABEL='Date of Death',
       INT((a.disdate-b.dob)/365.25) AS agedth LABEL='Age at Death',
       c.beds
FROM ex.admissions a,
     ex.patients b,
     ex.hospitals c
WHERE (a.pt_id = b.id) AND (a.hosp = c.hosp_id) AND a.dest EQ 9
ORDER BY pt_id ;
QUIT;

```

A PRINT of the new table DECEASED1 is shown in Output 8a.

| Example 8 - More Inner Joins | | | | |
|---|----------|---------------|--------------|----------------|
| 8a - Inner Join of 3 tables + row selection | | | | |
| Identifying patients who died in the hospital, determining age at death & hospital size | | | | |
| Patient ID | Hospital | Date of Death | Age at Death | Number of Beds |
| 001 | 05 | 22SEP2010 | 79 | 475 |
| 003 | 03 | 15NOV2010 | 83 | 724 |
| 009 | 02 | 04JAN2011 | 83 | 1176 |
| 012 | 05 | 09JAN2011 | 75 | 475 |

Output 8a. Listing of table DECEASED1, created by an Inner join of three tables in Example 8a.

For completeness and illustration I include Example 8b, which accomplishes the same task as Example 8a, but uses an alternative coding, with explicit INNER JOINS. There is no particular advantage to either method; I show both to demonstrate (as you likely know if you've been using SAS for more than a week ☺), that there are almost always multiple ways to accomplish the same result in SAS, and one is not always superior to another – sometimes it is simply personal preference that dictates the choice. The output is not included again. It is identical to Output 8a.

```
TITLE4 '8b - Inner Join of 3 tables + row selection (alternative code)';
TITLE5 'Identify Patients who died in hospital, determining age & hospital size';

PROC SQL ;
CREATE TABLE deceased2 AS
SELECT a.pt_id,
       a.hosp,
       a.disdate AS dthdate,
       INT((a.disdate-b.dob)/365.25) AS agedth,
       c.beds
FROM (ex.admissions a
      INNER JOIN
      ex.patients b
      ON a.pt_id = b.id)
     INNER JOIN
     ex.hospitals c
     ON a.hosp = c.hosp_id
WHERE a.dest EQ 9
ORDER BY pt_id ;
QUIT;
```

The next example is included to demonstrate that one of the operands in the FROM clause of a join can itself be a query rather than a table. In Example 8c, the objective is to identify patients who were admitted to the hospital by their primary physician and include in the output both the patient and physician names. The code is below.

```
TITLE4 '8c - Inner Join of two tables and a query';
TITLE5 'Identify patients admitted by primary MD and report patient & MD names';

PROC SQL ;
CREATE TABLE primdoc1 AS
SELECT a.pt_id, a.admdate, a.hosp,
       b.lastname AS ptname,
       c.lastname AS mdname,
       c.md_id
FROM ex.admissions a,
     ex.patients b,
     (SELECT DISTINCT md_id, lastname FROM ex.doctors) c
WHERE (a.pt_id EQ b.id) AND (a.md EQ b.primmd) AND (a.md EQ c.md_id)
ORDER BY a.pt_id, admdate ;
QUIT;
```

Demystifying SQL, *continued*

The third item in the FROM clause is itself a query. This syntax, with the DISTINCT keyword, is needed in order to eliminate duplicate rows for physicians in the DOCTORS table – physicians have multiple rows in that table if they can admit patients to more than one hospital. This might be best seen as an inline view as more than one column is selected (unlike a true subquery), although it is pre-processing a different table than either of the others in the FROM clause. The result is shown in Output 8c.

| Example 8 - More Inner Joins | | | | | | |
|---|-----------|------|----------|-----------|-------|--|
| 8c - Inner Join of two tables and a query | | | | | | |
| Identify patients admitted by their primary MD and report patient and MD name | | | | | | |
| pt_id | admdate | hosp | ptname | mdname | md_id | |
| 001 | 12APR2010 | 01 | Williams | Fitzhugh | 1972 | |
| 003 | 17OCT2010 | 03 | Gillette | Premnath | 8081 | |
| 005 | 19JAN2010 | 01 | Abbott | Fitzhugh | 1972 | |
| 005 | 10MAR2010 | 01 | Abbott | Fitzhugh | 1972 | |
| 005 | 10APR2010 | 02 | Abbott | Fitzhugh | 1972 | |
| 007 | 28JUL2010 | 02 | Pedersen | Hanratty | 3274 | |
| 010 | 30NOV2011 | 04 | Alberts | MacArthur | 2322 | |
| 018 | 01NOV2010 | 03 | Baker | Fitzhugh | 1972 | |
| 018 | 26DEC2010 | 03 | Baker | Fitzhugh | 1972 | |
| N = 9 | | | | | | |

Output 8c. Results of Example 8c, illustrating that one of the operands in a join can be a query

The explicit join syntax could also be used for this task. The code is shown below. The output is identical to Output 8b.

```
TITLE4 '8d - Inner Join of two tables and a query (alternative code)';
TITLE5 'Identify patients admitted by primary MD and report patient & MD names';

PROC SQL ;
CREATE TABLE primdoc2 AS
SELECT a.pt_id,
       a.admdate,
       a.hosp,
       b.lastname AS ptname,
       c.lastname AS mdname,
       c.md_id
FROM ex.admissions a
     INNER JOIN
ex.patients b
     ON a.pt_id EQ b.id
     INNER JOIN
(SELECT DISTINCT md_id,
                 lastname
 FROM ex.doctors) c
     ON a.md EQ c.md_id
     AND a.md EQ b.primmd
ORDER BY a.pt_id, admdate ;
QUIT;
```

Demystifying SQL, *continued*

EXAMPLE 9: LEFT JOINS

A left join is a type of full join. You can think of it as an inner join that is augmented by rows in the left table that have no rows in the right table that meet the join conditions. Of course, an example will help illustrate. Let's say we want to determine which hospitals had admissions – that is, we want to add a column to the HOSPITALS table indicating whether or not there are any matching records in the ADMISSIONS table. The code is shown below.

```
TITLE3 'Example 9 - Left Outer Joins ' ;
TITLE4 '9a - Determine which hospitals had admissions';

PROC SQL ;
CREATE TABLE hospinfo AS
SELECT DISTINCT a.*,
       b.hosp IS NOT NULL AS hasadmit
FROM ex.hospitals a
LEFT JOIN
     ex.admissions b
ON a.hosp_id = b.hosp ;
QUIT;
```

The 'a.*' syntax is shorthand for listing all the columns on the 'a' table (here, EX.HOSPITALS). The DISTINCT keyword is required to eliminate duplicates – there is only one row per hospital in the HOSPITALS table, but if there are multiple admissions to the hospital, as there are for several of them, without DISTINCT the resulting join would have a row for each admission, which we don't want. We are not actually selecting any columns from the ADMISSIONS table – doing so would also cause duplication. However, the expression 'b.hosp IS NOT NULL AS hasadmit' creates a column named HASADMIT which will be a 1 if there is a match on the join condition (i.e. there is a record in the admissions table for a given hospital). There are seven hospitals in the HOSPITALS table so we expect seven rows in the new HOSPINFO table. See Output 9a; all but one hospital has at least one matching record in the ADMISSIONS table.

```
Example 9 - Left Outer Joins
9a - Determine which hospitals had admissions
```

| hosp_id | hospname | hasadmit |
|---------|-----------|----------|
| 01 | Deacon | 1 |
| 02 | City | 1 |
| 03 | Tarheel | 1 |
| 04 | Peace | 1 |
| 05 | BlueDevil | 1 |
| 06 | Wolfpack | 1 |
| 07 | FarOut | 0 |

N = 7

Output 9a. Example 9a results, using a left join to create an indicator of which hospitals have admissions.

Let's take this example a little further and say that (for Example 9b) we want to count the number of admissions at each hospital, but we can't just summarize the ADMISSIONS table to count records per hospital because we want to have a 0 for hospitals that have no admissions. Code that will accomplish this is shown below.

Demystifying SQL, *continued*

```

TITLE3 'Example 9 - Left Outer Joins ' ;
TITLE4 '9b - Determine number of admits at each hospital, including zeros';

PROC SQL ;
CREATE TABLE hospinfo2 AS
SELECT a.*,
      CASE
        WHEN b.hosp IS NULL THEN 0
        WHEN b.hosp IS NOT NULL THEN b.n_adm
        ELSE .
      END AS numadmit LABEL='# of admissions'
FROM ex.hospitals a
LEFT JOIN
  (SELECT hosp,
          count(*) AS n_adm
   FROM ex.admissions
   GROUP BY hosp) b
ON a.hosp_id = b.hosp
ORDER by a.hosp_id ;
QUIT;

```

Example 9b is still a LEFT JOIN, but now we are joining the HOSPITALS table to a query. That query is a simple summarization of the ADMISSIONS file that counts the number of records per HOSP and stores it in a column called N_ADM. Note that at this point N_ADM will have a minimum value of 1. The CASE expression in the outer SELECT clause will set the value of a new variable NUMADMIT to 0 if there are no records coming from the inner query and will set it to the count N_ADM that resulted from that inner query if there is a match. There shouldn't be any records caught by the ELSE part of the CASE expression but it is good practice to include this (and you will get a WARNING in the log if you do not). Note that because of the rule that a CASE expression cannot be used to alter the values of existing columns, we have to create the new column NUMADMIT (which we then include on the output data set instead of N_ADM) rather than just assigning 0's to N_ADM counter. You could include N_ADM on the output file if you wanted to also – it would be the same as NUMADMIT for all non-zero values and would be missing when NUMADMIT is 0. See Output 9b.

| Example 9 - Left Outer Joins | | | | | | |
|---|-----------|-------|------|--------|----------|--|
| 9b - Determine number of admits at each hospital, including zeros | | | | | | |
| hosp_id | hospname | zip | beds | has_er | numadmit | |
| 01 | Deacon | 27105 | 202 | Y | 5 | |
| 02 | City | 27607 | 1176 | Y | 8 | |
| 03 | Tarheel | 27514 | 724 | Y | 8 | |
| 04 | Peace | 28585 | 839 | N | 2 | |
| 05 | BlueDevil | 27708 | 475 | Y | 6 | |
| 06 | Wolfpack | 27603 | 650 | N | 1 | |
| 07 | FarOut | 27850 | 68 | Y | 0 | |
| N = 7 | | | | | | |

Output 9b. Result of Example 9b, which uses a left outer join to generate a count of admissions at each hospital.

EXAMPLE 10: MANY-TO-MANY JOINS

We now move on to exploring the many-to-many join – when both tables have multiple records that ‘match’. It is well-known that the DATA Step MERGE does not handle such JOINS very well. They are not without their complexities in SQL either. Both the ADMISSIONS table and the ERVISITS table can have multiple records for a patient. Also, there are some patients with records in ADMISSIONS but no records in ERVISITS and vice versa. The patient identifier has the same name in both (PT_ID). Example 10a is simply a full join of the two tables, matching on patient, which we expect to create all possible combinations of hospital admissions and ER visits. Because we want to get a record for all the admissions and all the ER visits (i.e. whether the PT_ID is represented in ADMISSIONS or

Demystifying SQL, *continued*

ERVISITS or both, we use a FULL JOIN and SELECT the PT_ID column from both (A.PT_ID and B.PT_ID). Think of a FULL JOIN as an inner join that has been augmented with rows in either table that are not in the other table.

```
TITLE3 'Example 10 - Many-to-Many and Full Joins';
TITLE4 '10a- 1st try: All combinations of hospital & ER visits for each patient';

PROC SQL ;
CREATE TABLE adm_er_fulll1 AS
SELECT a.pt_id ,
       a.admdate,
       a.hosp,
       b.pt_id ,
       b.visitdate,
       b.er
FROM   ex.admissions AS a
FULL JOIN
       ex.ervisits AS b
ON     a.pt_id = b.pt_id
ORDER BY a.pt_id, b.pt_id, a.admdate, b.visitdate ;
QUIT;
```

First, observe the warning in the log:

```
WARNING: Variable pt_id already exists on file WORK.ADM_ER_FULLL1.
```

We get this message because we are attempting to select PT_ID from both files, and SQL doesn't really know what to do, since a SAS data set cannot have two variables/columns with the same name, and SQL by default will not blend the information from the two sources. The resulting table has 50 rows; the first 20 are shown in Output 10a..

| Example 10 - Many-to-Many and Full Joins | | | | | | |
|--|-------|-----------|------|-----------|----|--|
| 10a - First try: All combinations of hospital and ER visits for each patient | | | | | | |
| Obs | pt_id | admdate | hosp | visitdate | er | |
| 1 | . | . | . | 18JUL2011 | 07 | |
| 2 | . | . | . | 04JUL2010 | 07 | |
| 3 | . | . | . | 04JUL2011 | 07 | |
| 4 | . | . | . | 31OCT2010 | 03 | |
| 5 | . | . | . | 17MAY2011 | 01 | |
| 6 | 001 | 07FEB2010 | 01 | 12APR2010 | 01 | |
| 7 | 001 | 07FEB2010 | 01 | 09SEP2010 | 02 | |
| 8 | 001 | 12APR2010 | 01 | 12APR2010 | 01 | |
| 9 | 001 | 12APR2010 | 01 | 09SEP2010 | 02 | |
| 10 | 001 | 10SEP2010 | 02 | 12APR2010 | 01 | |
| 11 | 001 | 10SEP2010 | 02 | 09SEP2010 | 02 | |
| 12 | 001 | 19SEP2010 | 05 | 12APR2010 | 01 | |
| 13 | 001 | 19SEP2010 | 05 | 09SEP2010 | 02 | |
| 14 | 003 | 17OCT2010 | 03 | 16OCT2010 | 07 | |
| 15 | 003 | 15NOV2010 | 03 | 16OCT2010 | 07 | |
| 16 | 004 | 18JUN2010 | 02 | . | . | |
| 17 | 005 | 19JAN2010 | 01 | 01JAN2010 | 03 | |
| 18 | 005 | 19JAN2010 | 01 | 10APR2010 | 02 | |
| 19 | 005 | 10MAR2010 | 01 | 01JAN2010 | 03 | |
| 20 | 005 | 10MAR2010 | 01 | 10APR2010 | 02 | |

Output 10a. Partial listing of results of Example 10a, an attempt to generate all possible combinations of hospitalizations and ER visits for each patient.

PT_ID is never missing in either ADMISSIONS or ERVISITS, so why is it missing in the first 5 rows of the joined table? This illustrates a fundamental difference between SQL and the DATA step. It is clear that these records are coming from the ERVISITS table, since they have values for VISITDATE and ER, columns on the ERVISITS table.

Demystifying SQL, *continued*

So, what happened to PT_ID? A clue is the warning we got in the log, telling us that ADM_ER_FULL1 – our output table *already* had the variable PT_ID. This is referring to the first PT_ID listed, which is A.PT_ID – the patient ID on ADMISSIONS. Unlike the DATA Step, SQL will not by default put data from another PT_ID variable into the existing one, and the output table can't have two variables with the same name, so, if there is no value for the first one (coming from ADMISSIONS), it ends up being missing. The rest of the records are pretty much as we would expect. For example PT_ID 001 has 4 records in the ADMISSIONS table and 2 records in the ERVISITS table; thus, in the full join there are 4*2=8 records, corresponding to all possible combinations of the 4 hospital admits and 2 ER visits. PT_ID 003 has 2 admits and 1 ER visit, and so as 2*1 = 2 records in the joined table. And so on. Unlike the 5 records with missing PT_ID, which correspond to ER visits for patients with no hospital admissions, the one record listed that is missing on the ER variables is for PT_ID 004, who has one hospital admission and no ER visit. Because the ADMISSIONS PT_ID is given precedence, this record does not 'lose' its PT_ID like the unmatched ER visits.

So, how can we modify this code so that we don't lose the PT_ID for the records that have an ER visit and no hospitalizations? One obvious strategy is to rename the ID variables so that we don't lose the information, as shown in Example 10b.

```
TITLE4 '10b-2nd try: All combinations of hospital & ER visits for each patient';

PROC SQL ;
CREATE TABLE adm_er_full2 AS
SELECT a.pt_id AS pt_id_h,
       a.admdate,
       a.hosp,
       b.pt_id AS pt_id_e,
       b.visitdate,
       b.er

FROM
ex.admissions AS a
FULL JOIN
ex.ervisits AS b
ON
a.pt_id = b.pt_id
ORDER BY a.pt_id, b.pt_id, a.admdate, b.visitdate ;
QUIT;
```

Again, the full result has 50 records. The first 20 records are shown in Output 10b. This is closer to what we want – we haven't lost any IDs, but ideally we wouldn't have to have two separate ID variables...

| Example 10 - Many-to-Many and Full Joins | | | | | | |
|---|---------|-----------|------|---------|-----------|----|
| 10b - Second try: All combinations of hospital and ER visits for each patient | | | | | | |
| Better but not quite right... | | | | | | |
| Obs | pt_id_h | admdate | hosp | pt_id_e | visitdate | er |
| 1 | . | . | . | 002 | 18JUL2011 | 07 |
| 2 | . | . | . | 011 | 04JUL2010 | 07 |
| 3 | . | . | . | 011 | 04JUL2011 | 07 |
| 4 | . | . | . | 017 | 31OCT2010 | 03 |
| 5 | . | . | . | 019 | 17MAY2011 | 01 |
| 6 | 001 | 07FEB2010 | 01 | 001 | 12APR2010 | 01 |
| 7 | 001 | 07FEB2010 | 01 | 001 | 09SEP2010 | 02 |
| 8 | 001 | 12APR2010 | 01 | 001 | 12APR2010 | 01 |
| 9 | 001 | 12APR2010 | 01 | 001 | 09SEP2010 | 02 |
| 10 | 001 | 10SEP2010 | 02 | 001 | 12APR2010 | 01 |
| 11 | 001 | 10SEP2010 | 02 | 001 | 09SEP2010 | 02 |
| 12 | 001 | 19SEP2010 | 05 | 001 | 12APR2010 | 01 |
| 13 | 001 | 19SEP2010 | 05 | 001 | 09SEP2010 | 02 |
| 14 | 003 | 17OCT2010 | 03 | 003 | 16OCT2010 | 07 |
| 15 | 003 | 15NOV2010 | 03 | 003 | 16OCT2010 | 07 |
| 16 | 004 | 18JUN2010 | 02 | . | . | . |

Output 10b. Partial listing of Example 10b results, another attempt to full join admissions and ER visits.

Demystifying SQL, *continued*

Well, as it turns out, SQL has a function, called COALESCE that is specifically geared towards this purpose – it allows us to tell SQL that these two variables are really the same variable. Example 3c makes use of it. Otherwise the code is identical to Example 10a.

```
TITLE4 '10c - 3rd time is the charm? ';
TITLE5 'All combinations of hospital and ER visits for each patient';

PROC SQL ;
CREATE TABLE adm_er_full3 AS
SELECT COALESCE(a.pt_id, b.pt_id) AS pt_id FORMAT=Z3.,
       a.admdate,
       a.hosp,
       b.visitdate,
       b.er
FROM
  ex.admissions AS a
FULL JOIN
  ex.ervisits AS b
ON
  a.pt_id = b.pt_id
ORDER BY pt_id, a.admdate, b.visitdate ;
QUIT;
```

Output 10c shows the first dozen observations in the resulting table and demonstrates that we finally got the result we wanted.

| Example 10 - Many-to-Many and Full Joins | | | | | | |
|---|-------|-----------|------|-----------|----|--|
| 10c - 3rd time is the charm?: All combinations of hospital and ER visits for each patient | | | | | | |
| Obs | pt_id | admdate | hosp | visitdate | er | |
| 1 | 001 | 07FEB2010 | 01 | 12APR2010 | 01 | |
| 2 | 001 | 07FEB2010 | 01 | 09SEP2010 | 02 | |
| 3 | 001 | 12APR2010 | 01 | 12APR2010 | 01 | |
| 4 | 001 | 12APR2010 | 01 | 09SEP2010 | 02 | |
| 5 | 001 | 10SEP2010 | 02 | 12APR2010 | 01 | |
| 6 | 001 | 10SEP2010 | 02 | 09SEP2010 | 02 | |
| 7 | 001 | 19SEP2010 | 05 | 12APR2010 | 01 | |
| 8 | 001 | 19SEP2010 | 05 | 09SEP2010 | 02 | |
| 9 | 002 | . | . | 18JUL2011 | 07 | |
| 10 | 003 | 17OCT2010 | 03 | 16OCT2010 | 07 | |
| 11 | 003 | 15NOV2010 | 03 | 16OCT2010 | 07 | |
| 12 | 004 | 18JUN2010 | 02 | . | . | |

Output 10c. Partial listing of results of Example 10c, in which the full join of admissions and ER visits is achieved as desired.

Why stop there? One more version (Example 10d) shows an alternative to achieve the same result. The syntax “NATURAL FULL JOIN” tells SQL to coalesce like-named columns. There is an important caveat here, suggested by the fact that there is no ON condition with the NATURAL JOIN – SQL is matching on (and COALESCING) all like-named columns in the two tables. To use this method, the columns on which you want to match must have the same names in the incoming tables and any variables on which you *don't* want to match must have different names. For these two tables, the results of Example 10c and Example 10d are the same; so, the output is not reproduced here.

Demystifying SQL, *continued*

```

TITLE4 '10d-Alternative: All combinations of hospital & ER visits for each patient';

PROC SQL ;
CREATE TABLE adm_er_full4 AS
SELECT pt_id FORMAT=z3.,
       a.admdate,
       a.hosp,
       b.visitdate,
       b.er
FROM
     ex.admissions AS a
NATURAL FULL JOIN
     ex.ervisits AS b
ORDER BY pt_id, a.admdate, b.visitdate ;
QUIT;

```

Often, when one does a many-to-many join, the desired result is not actually all possible combinations of records in the two tables, but some subset of the matches. The remaining two examples in this section illustrate this. Our goal is to calculate the time from ER visit to hospital admission for each ER visit – admission combination for a patient. We go back to an inner join in Example 10e. Note that I specify simply 'JOIN' not 'INNER JOIN'. While it is probably good practice to explicitly state INNER JOIN for clarity, in PROC SQL an INNER JOIN is the default; so 'JOIN' and 'INNER JOIN' are equivalent. Also, the CALCULATED keyword is required on the ORDER BY date in order to have the resulting table sorted on a column that was created in the query.

```

TITLE4 '10e - INNER JOIN of Admissions & ERvisits, calculating days from ER visit
       to admission';

PROC SQL ;
CREATE TABLE adm_er_both1 AS
SELECT a.pt_id,
       a.admdate,
       a.hosp,
       b.visitdate,
       b.er,
       (a.admdate - b.visitdate) AS gap
FROM ex.admissions AS a
JOIN
     ex.ervisits AS b
ON a.pt_id = b.pt_id
ORDER BY pt_id, admdate, CALCULATED gap, visitdate ;
QUIT;

```

Output 10e shows the results for the first three patient IDs who have at least one hospitalization and at least one ER visit. We can tell from this that PT_ID 01 had 4 admissions and 2 ER visits, PT_ID 003 had 2 admissions and 1 ER visit and PT_ID 005 had 4 admissions and 2 ER visits.

Example 10 - Many-to-Many and Full Joins
10e - INNER JOIN of Admissions & ERvisits, calculating days from ER visit to admission

| pt_id | admdate | hosp | visitdate | er | gap |
|-------|-----------|------|-----------|----|------|
| 001 | 07FEB2010 | 01 | 09SEP2010 | 02 | -214 |
| 001 | 07FEB2010 | 01 | 12APR2010 | 01 | -64 |
| 001 | 12APR2010 | 01 | 09SEP2010 | 02 | -150 |
| 001 | 12APR2010 | 01 | 12APR2010 | 01 | 0 |
| 001 | 10SEP2010 | 02 | 09SEP2010 | 02 | 1 |
| 001 | 10SEP2010 | 02 | 12APR2010 | 01 | 151 |
| 001 | 19SEP2010 | 05 | 09SEP2010 | 02 | 10 |
| 001 | 19SEP2010 | 05 | 12APR2010 | 01 | 160 |

Demystifying SQL, *continued*

| | | | | | |
|-----|-----------|----|-----------|----|-----|
| 003 | 17OCT2010 | 03 | 16OCT2010 | 07 | 1 |
| 003 | 15NOV2010 | 03 | 16OCT2010 | 07 | 30 |
| 005 | 19JAN2010 | 01 | 10APR2010 | 02 | -81 |
| 005 | 19JAN2010 | 01 | 01JAN2010 | 03 | 18 |
| 005 | 10MAR2010 | 01 | 10APR2010 | 02 | -31 |
| 005 | 10MAR2010 | 01 | 01JAN2010 | 03 | 68 |
| 005 | 10APR2010 | 02 | 10APR2010 | 02 | 0 |
| 005 | 10APR2010 | 02 | 01JAN2010 | 03 | 99 |
| 005 | 11APR2010 | 03 | 10APR2010 | 02 | 1 |
| 005 | 11APR2010 | 03 | 01JAN2010 | 03 | 100 |

Output 10e. Partial listing of ADM_ER_BOTH1, an INNER JOIN of ADMISSIONS and ERVISITS, along with calculation of days from ER visit date to hospital admit date.

The real goal here is to find the ER visit that is closest to – but before – each hospital admission. The last code and output will allow us to check that we are getting desired record with the code for Example 10f, shown below. All of the code through the JOIN...ON is identical to Example 10e. In order to select only matches where the ER visit is before the admission date, we supply the WHERE clause – remembering to include CALCULATED. To choose the closest match meeting that criteria, we need to aggregate across all the matches with non-negative GAP values for each admission; thus, we GROUP BY patient ID and admission date. Finally, the HAVING condition will select for output the rows where the GAP value is equal to the smallest GAP value for each GROUP BY group. Note that we do get a NOTE in the log that the query requires REMERGING summary statistics with the original data because we are selecting columns other than summary columns and GROUP BY columns, but that is OK in this case. The output is shown in Output 10f. As expected, we have retrieved one ER visit for each hospital admission.

```
TITLE4 '10f - Select only the ER visit closest to and before each admission';
PROC SQL ;
CREATE TABLE adm_er_last AS
SELECT a.pt_id, a.admdate, a.hosp,
       b.visitdate, b.er,
       (a.admdate - b.visitdate) AS gap
FROM ex.admissions AS a
JOIN
     ex.ervisits AS b
ON a.pt_id = b.pt_id
WHERE CALCULATED gap GE 0
GROUP BY a.pt_id, a.admdate
       HAVING gap = MIN(gap)
ORDER BY a.pt_id, a.admdate, b.visitdate ;
QUIT;
```

| Example 10 - Many-to-Many and Full Joins | | | | | | |
|---|-----------|------|-----------|----|-----|--|
| 10f - Select only the ER visit closest to and before each admission | | | | | | |
| pt_id | admdate | hosp | visitdate | er | gap | |
| 001 | 12APR2010 | 01 | 12APR2010 | 01 | 0 | |
| 001 | 10SEP2010 | 02 | 09SEP2010 | 02 | 1 | |
| 001 | 19SEP2010 | 05 | 09SEP2010 | 02 | 10 | |
| 003 | 17OCT2010 | 03 | 16OCT2010 | 07 | 1 | |
| 003 | 15NOV2010 | 03 | 16OCT2010 | 07 | 30 | |
| 005 | 19JAN2010 | 01 | 01JAN2010 | 03 | 18 | |
| 005 | 10MAR2010 | 01 | 01JAN2010 | 03 | 68 | |
| 005 | 10APR2010 | 02 | 10APR2010 | 02 | 0 | |
| 005 | 11APR2010 | 03 | 10APR2010 | 02 | 1 | |

Output 10f. Partial listing of ADM_ER_LAST, which selected the ER visit closest to but prior to each hospital admission.

Demystifying SQL, *continued***EXAMPLE 11: SET OPERATORS – UNION**

In this section, we examine one of the SQL SET operators – UNION – which is used for concatenation of tables, or ‘stacking’ rows from one table on top of another. We’ll see how UNION is similar to – and more importantly, different from the DATA Step SET statement. In order to do this, we are first going to take the ADMISSIONS data set apart, separating the 2010 and 2011 admissions into separate tables. Example 11a shows the PROC SQL code for this – we simply execute two very similar CREATE TABLE queries. As a bit of foreshadowing, I point out that by using the SELECT * syntax we are selecting all columns from the ADMISSIONS table and NOT changing the order of the columns on the ADMIT2010 and ADMIT2011 tables. Output 11a shows just a few rows from each of these tables.

```
TITLE3 'Example 11 - Set Operators (UNION) ';
TITLE4 '11a (preparation) - Creating Two Data Sets from One';
TITLE5 'Generate separate tables for the 2010 and 2011 admissions';

PROC SQL ;
  CREATE TABLE admit2010 AS
    SELECT * FROM ex.admissions
      WHERE YEAR(admdate) = 2010;
  CREATE TABLE admit2011 AS
    SELECT * FROM ex.admissions
      WHERE YEAR(admdate) = 2011;
QUIT;
```

```
Example 11 - Set Operators (UNION)
11a (preparation) - Creating Two Data Sets from One
Generate separate tables for the 2010 and 2011 admissions
```

2010 admissions only

| pt_id | admdate | disdate | md | hosp | dest | bp_sys | bp_dia | primdx |
|-------|-----------|-----------|------|------|------|--------|--------|--------|
| 001 | 07FEB2010 | 08FEB2010 | 3274 | 01 | 1 | 188 | 85 | 410 |
| 001 | 12APR2010 | 25APR2010 | 1972 | 01 | 1 | 230 | 101 | 428.2 |
| 001 | 10SEP2010 | 19SEP2010 | 3274 | 02 | 2 | 170 | 78 | 813.9 |
| 001 | 19SEP2010 | 22SEP2010 | 3274 | 05 | 9 | 185 | 94 | 428.4 |
| 003 | 17OCT2010 | 21OCT2010 | 8081 | 03 | 1 | 155 | 92 | 410.01 |

2011 admissions only

| pt_id | admdate | disdate | md | hosp | dest | bp_sys | bp_dia | primdx |
|-------|-----------|-----------|------|------|------|--------|--------|--------|
| 006 | 11SEP2011 | 13SEP2011 | 8081 | 05 | 2 | 129 | 83 | 820.01 |
| 010 | 30NOV2011 | 06DEC2011 | 2322 | 04 | 1 | 147 | 84 | E886.3 |
| 012 | 04JAN2011 | 09JAN2011 | 4003 | 05 | 9 | 201 | 98 | 433.4 |
| 014 | 17JAN2011 | 20JAN2011 | 7803 | 03 | 1 | 162 | 93 | 414.1 |
| 015 | 25MAY2011 | 06JUN2011 | 4003 | 05 | 2 | 142 | 81 | 820.8 |

Output 11a. Partial listing of ADMIT2010 and ADMIT2011 tables, generated by Example 10a.

Imagining that we started with the separate year files, we want to concatenate them into a single table with both years. The code to do so is Example 11b.

```
TITLE4 '11b - Concatenation (UNION)';

PROC SQL ;
  CREATE TABLE alladmits AS
    SELECT * FROM admit2010
  UNION CORRESPONDING
    SELECT * FROM admit2011
  ORDER BY pt_id, admdate;
QUIT;
```

The UNION operator works on the result of two queries. The resulting table is not printed here; it is identical to the original ADMISSIONS table. Importantly, the CORRESPONDING keyword causes SQL to match the columns by

Demystifying SQL, *continued*

name rather than position. In this case, it does not change the result because the like-named columns are in the same order on the two files, but it can make a BIG difference – one that could catch you by surprise – so, we reinforce the point in the next example.

We first do a preparatory step to create the two tables we want to put together with a UNION – Example 11c. Here we select different columns from the two years. Admittedly this is rather artificial, but it will help illustrate the point.

```
TITLE4 '11c (preparation) - importance of CORRESPONDING in UNION';

PROC SQL ;
CREATE TABLE adm2010r AS
SELECT pt_id, admdate, hosp
FROM ex.admissions
WHERE YEAR(admdate) = 2010 ;

CREATE TABLE adm2011r AS
SELECT pt_id, disdate, md
FROM ex.admissions
WHERE year(admdate) = 2011 ;
QUIT;
```

Again we show just a few rows from each table in Output 11c. The columns are PRINTED in the order they are on the data set.

```
Example 11 - Set Operators (UNION)
11c (preparation) - importance of CORRESPONDING in UNION

2010 admissions only
pt_id    admdate    hosp
001      07FEB2010    01
001      12APR2010    01
001      10SEP2010    02
001      19SEP2010    05
003      17OCT2010    03
003      15NOV2010    03

2011 admissions only
pt_id    disdate    md
006      13SEP2011    8081
010      06DEC2011    2322
012      09JAN2011    4003
014      20JAN2011    7803
015      06JUN2011    4003
015      24AUG2011    4003
```

Output 11c. Partial listing of ADM2010r and ADM2011r, generated by Example 11c. Note selection of different columns in the two tables.

To put these two data sets back together we use code that is *almost* identical to Example 11b. All that is different is the names of the component tables *and* the absence of CORRESPONDING.

```
TITLE3 'Example 11 - Set Operators ';
TITLE4 '11d - Concatenation (UNION) - without CORRESPONDING';

PROC SQL ;
CREATE TABLE alladm_r AS
SELECT * FROM adm2010r
UNION /* <---- note absence of CORRESPONDING */
SELECT * FROM adm2011r
ORDER BY pt_id, admdate;
QUIT;
```

Demystifying SQL, *continued*

Now, if UNION operated like the DATA Step SET statement, the result would simply have the 2010 rows on top of the 2011 rows, and the 2010 rows would all have missing values for DISDATE and MD and the 2011 rows would all have missing values for ADMDATE and HOSP. That is NOT what happens. See Output 11d.

```

Example 11 - Set Operators (UNION)
11d - importance of CORRESPONDING in UNION (leave out CORRESPONDING)

pt_id      admdate      hosp
-----
001        07FEB2010      01
001        12APR2010      01
001        10SEP2010      02
< some rows deleted here >
006        13SEP2011      **
007        28JUL2010      02
007        08SEP2010      02
< some rows deleted here >
009        15DEC2010      02
010        06DEC2011      **
012        12AUG2010      05
012        09JAN2011      **
014        20JAN2011      **
015        06JUN2011      **
015        24AUG2011      **

```

Output 11d. Partial listing of results of Example 11d. Examine what occurred with the DISDATE and MD information for the 2011 records.

What happened?!?!? At first glance it may appear that SQL just dropped the variables that were unique to the 2011 file (DISDATE and HOSP), but when you look at the output more closely you will see that what WERE discharge dates in the 2011 table are now being stored in the ADMDATE variable!...This might even have escaped notice at least initially, since both are SAS date variables, but observe what has happened for the HOSP variable! Note that all the 2011 observations, HOSP (which is a numeric variable with a Z2. format) is being displayed as '**', which SAS uses when it can't display the variable value in an assigned format, which is the case here because the MD variable is a 4-digit code and so can't 'fit' in the Z2. format. If we removed the Z2. format from HOSP, we'd see that the MD values are now being stored there!

Perhaps my level of surprise (and indignation!) the first time I saw this type of result of an SQL UNION is indicative of just how ingrained I am in my DATA Step thinking, but it sure does drive home the point that SQL 'thinks' first about the order of the columns on a table, and has to be 'told' to give precedence to column names! I will also point out that there is a totally "clean log" – nothing to indicate the rather peculiar result – except that the resulting table has 3 columns rather than the expected 5.

So, will just adding CORRESPONDING 'fix' this? We run the code shown below (Example 11e). All that is different from Example 11d is the addition of CORRESPONDING after UNION.

```

TITLE4 '11e - importance of CORRESPONDING in UNION (put CORRESPONDING in)';

PROC SQL ;
CREATE TABLE alladm_r2 AS
SELECT * FROM adm2010r
UNION CORRESPONDING
SELECT * FROM adm2011r
ORDER BY pt_id;
QUIT;

```

Output 11e, which is the complete listing, shows that we get yet another unanticipated result. Not only does the result have only a single column (PT_ID), but we've "lost" rows as well...SQL has eliminated duplicates on PT_ID, so that those who had multiple rows in either or both of the 2010 and 2011 files are now represented with a single row. So, what is SQL "thinking"? Well, CORRESPONDING in this situation tells SQL to line up only like-named columns – and discard the rest. Further, by default SQL will eliminate duplicates, and rows that would NOT have been

Demystifying SQL, *continued*

duplicates were the other columns present, ARE duplicates when only the CORRESPONDING column PT_ID is considered. (Note that 'CORRESPONDING ALL' would have maintained the multiple rows per patient but would not have saved the other columns).

| Example 11 - Set Operators (UNION) | |
|---|--------------|
| 11e - importance of CORRESPONDING in UNION (put CORRESPONDING in) | |
| | <u>pt_id</u> |
| | 001 |
| | 003 |
| | 004 |
| | 005 |
| | 006 |
| | 007 |
| | 008 |
| | 009 |
| | 010 |
| | 012 |
| | 014 |
| | 015 |
| | 016 |
| | 018 |
| | 020 |

Output 11e. Full listing of result of Example 11e.

To get a true concatenation of the two tables, we need to use OUTER UNION CORRESPONDING, as shown in Example 11f (and Output 11f)...success, at last!!

```
TITLE4 '11f - importance of CORRESPONDING in UNION ' ;
TITLE5 ' DATA STEP SET-like behavior requires OUTER & CORRESPONDING' ;
PROC SQL ;
CREATE TABLE alladm_r3 AS
  SELECT * FROM adm2010r
  OUTER UNION CORRESPONDING
  SELECT * FROM adm2011r
ORDER BY pt_id, admdate;
QUIT;
```

Example 11 - Set Operators (UNION) - 11f - importance of CORRESPONDING in UNION
DATA STEP SET-like behavior requires OUTER & CORRESPONDING

| pt_id | admdate | hosp | disdate | md |
|----------------------------|-----------|------|-----------|------|
| 001 | 07FEB2010 | 01 | . | . |
| 001 | 12APR2010 | 01 | . | . |
| 001 | 10SEP2010 | 02 | . | . |
| < some rows deleted here > | | | | |
| 006 | . | . | 13SEP2011 | 8081 |
| 007 | 28JUL2010 | 02 | . | . |
| 007 | 08SEP2010 | 02 | . | . |
| < some rows deleted here > | | | | |
| 009 | 15DEC2010 | 02 | . | . |
| 010 | . | . | 06DEC2011 | 2322 |
| 012 | . | . | 09JAN2011 | 4003 |
| 012 | 12AUG2010 | 05 | . | . |
| 014 | . | . | 20JAN2011 | 7803 |
| 015 | . | . | 24AUG2011 | 4003 |

Output 11f. Partial listing of result of Example 11f.

EXAMPLE 12: SET OPERATORS – EXCEPT

The last set of examples illustrate the use of another SQL set operator – EXCEPT, which is used to produce rows that are in the first query only. We start with the two yearly datasets (ADMIT2010 and ADMIT2011) we generated in Example 11a, which have the same columns but just separate the ADMISSIONS file in to 2010 and 2011 records (refer to Output 11a). If we want to identify the patients (PT_IDs) that have admissions only in 2010, we can use the following code. The resulting table is PRINTed in Output 12a.

```
TITLE3 'Example 12 - Set Operators (EXCEPT) ';
TITLE4 '12a - Selecting IDs Unique to One Table -- IDs w/ admits only in 2010';

PROC SQL ;
  CREATE TABLE only2010 AS
    SELECT pt_id
      FROM admit2010
EXCEPT
  SELECT pt_id
      FROM admit2011;
QUIT;
```

```
Example 12 - Set Operators (EXCEPT)
12a - Selecting IDs Unique to One Table -- IDs with admissions only in 2010
```

| Obs | pt_id |
|-----|-------|
| 1 | 001 |
| 2 | 003 |
| 3 | 004 |
| 4 | 005 |
| 5 | 007 |
| 6 | 008 |
| 7 | 009 |
| 8 | 018 |

Output 12a. Listing of results of Example 12a, selecting PT_ID values unique to 2010 with EXCEPT.

This is pretty straightforward. However, if we actually want information other than just the PT_ID from the 'ONLY2010' admissions, it does NOT work to simply add more columns (e.g. ADMDATE and HOSP) to the first SELECT query. Instead, if we try it (Example 12b), we get the WARNING and NOTE in the log shown in Log12b. The 21 rows produced are ALL the rows for patients who have ANY admissions in 2010. What has happened is that SQL is trying to compare all the selected columns and since there is only 1 selected column in the second query, it adds two more 'null columns' and as there are now rows coming from the first query that match these rows with null columns from the second query, no rows get eliminated.

```
TITLE3 'Example 12 - Set Operators (EXCEPT) ';
TITLE4 '12b - ATTEMPT to SELECT other columns for 2010-only admits';
PROC SQL ;
  CREATE TABLE only2010r AS
    SELECT pt_id, admdate, hosp
      FROM admit2010
EXCEPT
  SELECT pt_id
      FROM admit2011;
QUIT;
```

```
WARNING: A table has been extended with null columns to perform the EXCEPT set
operation.
NOTE: Table WORK.ONLY2010R created, with 21 rows and 3 columns.
```

Log 12b.

Demystifying SQL, *continued*

So, we clearly need another tactic. Code that will work to generate a listing of all the hospital admissions information for patients who have admissions only in 2010 is shown below (Example 12c) and uses some of the techniques we've seen before.

```
TITLE3 'Example 12 - Set Operators (EXCEPT) ';
TITLE4 '12c - Selecting Records for IDs Unique to One Table';
TITLE5 'Information on admissions for IDs with admissions only in 2010';

PROC SQL ;
CREATE TABLE only2010_a AS
SELECT * FROM admit2010 AS a
INNER JOIN
  ( SELECT pt_id
    FROM admit2010
  EXCEPT
    SELECT pt_id
    FROM admit2011) AS b
ON a.pt_id = b.pt_id ;
QUIT;
```

We perform an INNER JOIN between the complete ADMI2010 table (all columns for the 2010 admission) AND the query from Example 12a which isolates the PT_IDs of those with admits in 2010 and not 2011. The match key is the patient ID; the query with the EXCEPT clause will have the ID's we want and the join with the full 2010 table will ensure we have all the desired columns. Nice! The output is shown in Output 12c.

```
Example 12 - Set Operators (EXCEPT)
12c - Selecting Records for IDs Unique to One Table
Information on admissions for IDs with admissions only in 2010
```

| pt_id | admdate | disdate | md | hosp | dest | bp_sys | bp_dia | primdx |
|-------|-----------|-----------|------|------|------|--------|--------|--------|
| 001 | 07FEB2010 | 08FEB2010 | 3274 | 01 | 1 | 188 | 85 | 410 |
| 001 | 12APR2010 | 25APR2010 | 1972 | 01 | 1 | 230 | 101 | 428.2 |
| 001 | 10SEP2010 | 19SEP2010 | 3274 | 02 | 2 | 170 | 78 | 813.9 |
| 001 | 19SEP2010 | 22SEP2010 | 3274 | 05 | 9 | 185 | 94 | 428.4 |
| 003 | 17OCT2010 | 21OCT2010 | 8081 | 03 | 1 | 155 | 92 | 410.01 |
| 003 | 15NOV2010 | 15NOV2010 | 2322 | 03 | 9 | 74 | 40 | 431 |
| 004 | 18JUN2010 | 24JUN2010 | 7803 | 02 | 2 | 140 | 78 | 434.1 |
| 005 | 19JAN2010 | 22JAN2010 | 1972 | 01 | 1 | 148 | 84 | 411.81 |
| 005 | 10MAR2010 | 18MAR2010 | 1972 | 01 | 1 | 160 | 90 | 410.9 |
| 005 | 10APR2010 | 11APR2010 | 1972 | 02 | 2 | 150 | 89 | 411 |
| 005 | 11APR2010 | 28APR2010 | 7803 | 03 | 1 | 145 | 91 | 411 |
| 007 | 28JUL2010 | 10AUG2010 | 3274 | 02 | 2 | 136 | 72 | 155 |
| 007 | 08SEP2010 | 15SEP2010 | 8081 | 02 | 2 | 138 | 71 | 155 |
| 008 | 13APR2010 | 19APR2010 | 1972 | 02 | 1 | 140 | 80 | 428.4 |
| 008 | 01OCT2010 | 15OCT2010 | 3274 | 03 | 1 | 145 | 74 | 820.8 |
| 008 | 26NOV2010 | 28NOV2010 | 2322 | 03 | 2 | 135 | 76 | V54.8 |
| 008 | 10DEC2010 | 12DEC2010 | 8081 | 06 | 2 | 132 | 78 | V54.8 |
| 009 | 15DEC2010 | 04JAN2011 | 1972 | 02 | 9 | 228 | 92 | 410.1 |
| 018 | 01NOV2010 | 15NOV2010 | 1972 | 03 | 2 | 170 | 88 | 428.1 |
| 018 | 26DEC2010 | 08JAN2011 | 1972 | 03 | 2 | 199 | 93 | 428.1 |

Output 12c. Listing of Results of Example 12c, using a nested query to obtain other information on the admissions for patients who had hospitalizations only in 2010.

CONCLUSIONS

I hope that by meandering through these examples with me you have learned something about PROC SQL that will make it a little less daunting the next time you need to understand someone else's code or develop a query of your own. I have gradually become a big fan of SQL's ability to seamlessly combine detail and summary information and

Demystifying SQL, *continued*

perform joins that are complex in their criteria and/or require multiple components that may have different key variables or are at different levels of aggregation. Nonetheless, I continue to find it useful to try new SQL code on small data sets and build complex queries from simpler components – to make sure there aren't any ugly surprises because of misunderstandings about how PROC SQL thinks. There's no need to use only DATA Step methods or become a total convert to PROC SQL; by having an understanding of both, you'll be able to choose the best tool for the tasks at hand! Best of luck.

REFERENCES AND RECOMMENDED READING

SAS Institute Inc. 2009. 'The SQL Procedure', *Base SAS® 9.2 Procedures Guide*. Cary, NC: SAS Institute Inc. Available at: <http://support.sas.com/documentation/cdl/en/proc/61895/PDF/default/proc.pdf>

Williams, Christianna S. 2009 "PROC SQL for DATA Step Die-Hards." Proceedings of NorthEast SAS User Group 2009 Conference. Available at: <http://www.nesug.org/Proceedings/nesug09/hw/hw07.pdf>

Schreier, Howard. 2008. *PROC SQL by Example: Using SQL within SAS®*. Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at Christianna.S.Williams@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX

Full listing of data sets for the Examples

EX.ADMISSIONS (N=30)

| <i>pt_id</i> | <i>admdate</i> | <i>disdate</i> | <i>md</i> | <i>hosp</i> | <i>dest</i> | <i>bp_sys</i> | <i>bp_dia</i> | <i>primdx</i> |
|--------------|----------------|----------------|-----------|-------------|-------------|---------------|---------------|---------------|
| 001 | 07FEB2010 | 08FEB2010 | 3274 | 01 | 1 | 188 | 85 | 410 |
| 001 | 12APR2010 | 25APR2010 | 1972 | 01 | 1 | 230 | 101 | 428.2 |
| 001 | 10SEP2010 | 19SEP2010 | 3274 | 02 | 2 | 170 | 78 | 813.9 |
| 001 | 19SEP2010 | 22SEP2010 | 3274 | 05 | 9 | 185 | 94 | 428.4 |
| 003 | 17OCT2010 | 21OCT2010 | 8081 | 03 | 1 | 155 | 92 | 410.01 |
| 003 | 15NOV2010 | 15NOV2010 | 2322 | 03 | 9 | 74 | 40 | 431 |
| 004 | 18JUN2010 | 24JUN2010 | 7803 | 02 | 2 | 140 | 78 | 434.1 |
| 005 | 19JAN2010 | 22JAN2010 | 1972 | 01 | 1 | 148 | 84 | 411.81 |
| 005 | 10MAR2010 | 18MAR2010 | 1972 | 01 | 1 | 160 | 90 | 410.9 |
| 005 | 10APR2010 | 11APR2010 | 1972 | 02 | 2 | 150 | 89 | 411 |
| 005 | 11APR2010 | 28APR2010 | 7803 | 03 | 1 | 145 | 91 | 411 |
| 006 | 11SEP2011 | 13SEP2011 | 8081 | 05 | 2 | 129 | 83 | 820.01 |
| 007 | 28JUL2010 | 10AUG2010 | 3274 | 02 | 2 | 136 | 72 | 155 |
| 007 | 08SEP2010 | 15SEP2010 | 8081 | 02 | 2 | 138 | 71 | 155 |
| 008 | 13APR2010 | 19APR2010 | 1972 | 02 | 1 | 140 | 80 | 428.4 |
| 008 | 01OCT2010 | 15OCT2010 | 3274 | 03 | 1 | 145 | 74 | 820.8 |
| 008 | 26NOV2010 | 28NOV2010 | 2322 | 03 | 2 | 135 | 76 | V54.8 |
| 008 | 10DEC2010 | 12DEC2010 | 8081 | 06 | 2 | 132 | 78 | V54.8 |
| 009 | 15DEC2010 | 04JAN2011 | 1972 | 02 | 9 | 228 | 92 | 410.1 |
| 010 | 30NOV2011 | 06DEC2011 | 2322 | 04 | 1 | 147 | 84 | E886.3 |
| 012 | 12AUG2010 | 16AUG2010 | 4003 | 05 | 1 | 187 | 106 | 410.52 |
| 012 | 04JAN2011 | 09JAN2011 | 4003 | 05 | 9 | 201 | 98 | 433.4 |
| 014 | 17JAN2011 | 20JAN2011 | 7803 | 03 | 1 | 162 | 93 | 414.1 |
| 015 | 25MAY2011 | 06JUN2011 | 4003 | 05 | 2 | 142 | 81 | 820.8 |
| 015 | 17AUG2011 | 24AUG2011 | 4003 | 05 | 2 | 138 | 79 | 38.2 |
| 016 | 25JUL2011 | 30JUL2011 | 7803 | 02 | 1 | 189 | 101 | 412.1 |
| 018 | 01NOV2010 | 15NOV2010 | 1972 | 03 | 2 | 170 | 88 | 428.1 |
| 018 | 26DEC2010 | 08JAN2011 | 1972 | 03 | 2 | 199 | 93 | 428.1 |
| 020 | 04JUL2011 | 08JUL2011 | 2998 | 04 | 1 | 118 | 75 | 414 |
| 020 | 08OCT2011 | 01NOV2011 | 2322 | 01 | 2 | 162 | 99 | 434 |

EX.PATIENTS (N=20)

| <i>id</i> | <i>sex</i> | <i>primmd</i> | <i>lastname</i> | <i>firstname</i> | <i>dob</i> | <i>zipcode</i> |
|-----------|------------|---------------|-----------------|------------------|------------|----------------|
| 001 | 1 | 1972 | Williams | Hugh | 10AUG1931 | 27516 |
| 002 | 2 | 1972 | Franklin | Susan | 17MAR1938 | 27402 |
| 003 | 1 | 8081 | Gillette | Michael | 02JUL1927 | 29401 |
| 004 | 1 | 4003 | Wallace | Geoffrey | 25MAY1925 | 27699 |
| 005 | 2 | 1972 | Abbott | Celeste | 31AUG1940 | 27114 |
| 006 | 1 | 2322 | Capel | Anthony | 12APR1929 | 27155 |
| 007 | 1 | 3274 | Pedersen | Lars | 07FEB1938 | 27516 |
| 008 | 2 | 4003 | Lieberman | Marianne | 09NOV1942 | 27604 |
| 009 | 2 | 3274 | Jacobson | Frances | 15SEP1927 | 27708 |
| 010 | 2 | 2322 | Alberts | Josephine | 14OCT1948 | 28544 |
| 011 | 2 | 1972 | Erickson | Karen | 04NOV1926 | 29904 |
| 012 | 1 | 7803 | Collins | Albert | 16JUN1935 | 27340 |
| 013 | 1 | 4003 | Greene | Riley | 03AUG1946 | 27615 |
| 014 | 1 | 8034 | Dohlman | Henrik | 09NOV1950 | 27516 |
| 015 | 2 | 3274 | Zakur | Hannah | 06JUN1937 | 28117 |
| 016 | 1 | 1972 | DeLucia | Antonio | 17JUN1923 | 28083 |
| 017 | 1 | 2322 | Cohen | Adam | 17APR1935 | 27511 |
| 018 | 1 | 1972 | Baker | Shelby | 13FEB1940 | 27533 |
| 019 | 2 | 4003 | Wallace | Judith | 01FEB1933 | 28083 |
| 020 | 2 | 7803 | Carrier | Sarah | 07AUG1935 | 28357 |

Demystifying SQL, *continued***EX.HOSPITALS (N=7)**

| <i>hosp_id</i> | <i>hospname</i> | <i>zip</i> | <i>beds</i> | <i>has_er</i> |
|----------------|-----------------|------------|-------------|---------------|
| 01 | Deacon | 27105 | 202 | Y |
| 02 | City | 27607 | 1176 | Y |
| 03 | Tarheel | 27514 | 724 | Y |
| 04 | Peace | 28585 | 839 | N |
| 05 | BlueDevil | 27708 | 475 | Y |
| 06 | Wolfpack | 27603 | 650 | N |
| 07 | FarOut | 27850 | 68 | Y |

EX.DOCTORS (N=14)

| <i>md_id</i> | <i>lastname</i> | <i>zip_office</i> | <i>hospadm</i> |
|--------------|-----------------|-------------------|----------------|
| 1972 | Fitzhugh | 27105 | 01 |
| 1972 | Fitzhugh | 27105 | 02 |
| 2322 | MacArthur | 27514 | 01 |
| 2322 | MacArthur | 27514 | 03 |
| 2998 | Rosenberg | 28585 | 04 |
| 3274 | Hanratty | 27708 | 01 |
| 3274 | Hanratty | 27708 | 02 |
| 3274 | Hanratty | 27708 | 03 |
| 4003 | Colantonio | 28544 | 05 |
| 7803 | Avitable | 27105 | 02 |
| 7803 | Avitable | 27105 | 03 |
| 8081 | Premnath | 27607 | 02 |
| 8081 | Premnath | 27607 | 05 |
| 8081 | Premnath | 27607 | 06 |

EX.ERVISITS (N=20)

| <i>pt_id</i> | <i>visitdate</i> | <i>er</i> | <i>waitmin</i> |
|--------------|------------------|-----------|----------------|
| 001 | 12APR2010 | 01 | 28 |
| 001 | 09SEP2010 | 02 | 22 |
| 002 | 18JUL2011 | 07 | 41 |
| 003 | 16OCT2010 | 07 | 18 |
| 005 | 01JAN2010 | 03 | 15 |
| 005 | 10APR2010 | 02 | 54 |
| 006 | 15MAY2010 | 05 | 58 |
| 006 | 24AUG2010 | 05 | 180 |
| 006 | 15MAR2011 | 05 | 74 |
| 008 | 21APR2010 | 02 | 126 |
| 008 | 01OCT2010 | 03 | 78 |
| 009 | 12DEC2010 | 02 | 90 |
| 011 | 04JUL2010 | 07 | 12 |
| 011 | 04JUL2011 | 07 | 35 |
| 012 | 03JAN2011 | 05 | 175 |
| 014 | 01MAR2010 | 03 | 43 |
| 014 | 31MAY2010 | 02 | 3 |
| 017 | 31OCT2010 | 03 | 29 |
| 018 | 01NOV2010 | 02 | 46 |
| 019 | 17MAY2011 | 01 | 17 |