

SELECT ITEMS FROM PROC.SQL WHERE ITEMS > BASICS

Alan Dickson & Ray Pass
ASG, Inc.

For those of you who have become extremely comfortable and competent in the DATA step, getting into PROC SQL in a big way may not initially seem worth it. Your first exposure may have been through a training class, a Tutorial at a conference, or a real-world "had-to" situation, such as needing to read corporate data stored in DB2, or another relational database. So, perhaps it is just another tool in your PROC-kit for working with SAS datasets, or maybe it's your primary interface with another environment.

Whatever the case, your first experiences usually involve only the basic CREATE TABLE, SELECT, FROM, WHERE, GROUP BY, ORDER BY options. Thus, the tendency is frequently to treat it as no more than a data extract tool. You then go on to do the "real" work in SAS DATA step post-processing.

While that approach may have the advantage of getting the job done faster (always a consideration these days), there are a number of reasons why it might be worth looking into doing more in the realm of SQL before cranking up that good old DATA step. In fact, maybe you can avoid it altogether!

1) SQL just works differently. Sometimes that's better, sometimes worse. Although it can be a bit of a mental wrench to think SQL rather than DATA step, there are situations where the SQL approach is more appropriate. An actuary once asked one of the authors (AD) how he could force SAS to do an all-to-all merge of two datasets (both with non-unique keys) which he was planning to extract separately from an SQL/DS database. To his credit, by the time we'd talked through his reasons for doing this, he had the "lightbulb experience" - SQL does it that way all the time, so why bypass that power? Why not just "join" them with SQL.

2) Resource constraints. At the risk of being branded heretics for suggesting such "cost-shifting", it may prove easier/faster/cheaper to take advantage of system-owned resources to get some of your processing done. This may be particularly true when using PROC SQL to access corporate relational databases, since they tend to have large sort-space and work-space pools associated with them. Even if your benchmarks show that PROC SORT is more efficient than using an ORDER BY statement, perhaps the three hours run-time is more effective than the three days of politics trying

to get more SAS sort-space. Also, it may be possible to limit the total size of the resulting extract set by doing as much of the joining/merging as close to the raw data as possible.

3) Communications. SQL is becoming a lingua franca of data processing, and this trend is likely to continue for some time. You may find it easier to deal with your systems department, vendors etc., who may not know SAS at all, if you can explain yourself in SQL terms.

4) Career growth. Not unrelated to the above - there's a demand for people who can really understand SQL. Its apparent simplicity is deceiving. On the one hand, you can do an incredible amount with very little code - on the other, you can readily generate plausible, but inaccurate, results with code that looks to be bullet-proof at first glance.

So, our aim is not to proselytize. SQL may not be the answer to all (or even any) of your problems. Nonetheless, if you've only been a surface-scratcher until now, come along and take a look at some examples that go just a little bit deeper. We hope they whet your appetite to do some more exploring of your own.

At this point we would like to emphasize that, in this paper, we are only showing what *can* be done - we're not making any efficiency statements. As always, the results you obtain depend on your environment, your data, and the choice of technique. That last point is particularly important since, as you progress through the examples, you will see that there is frequently more than a single technique that can be applied to solve a specific problem. Even if you're already familiar with one, we hope that we'll expose you to some others that may be more appropriate or efficient. We suggest that you conduct your own benchmarks before committing yourself to any one approach, especially if it will become a "production" application.

It's also important to realize that this paper focuses on the SAS implementation of SQL. That is the background of one of the authors (RP), while the other is more familiar with the DB2 implementation (AD). There are many differences between the two - we'll try to point out any major discrepancies as we go, but be warned that some of this code will not work in non-SAS SQL environments.

SECTION 1 - First Things First

First, let's discuss the title of this paper - we wanted it not only to describe the paper's contents, but also to be syntactically valid SQL code. Here is an example including the paper title in the PROC SQL statement, along with data that would support the code.

```

*-----;
TITLE 'EX 0 - PAPER TITLE';
RUN;

DATA PROC.SQL;
  INPUT @01 KEY      $4.
        @06 ITEMS    2.
        @09 BASICS   2.;

CARDS;
KEY1 34 17
KEY2 22 44
KEY3 18 09
KEY4 18 36
KEY5 52 26
KEY6 12 24
KEY7 40 20 ;
*-----;

PROC SQL;
  SELECT ITEMS
  FROM   PROC.SQL
  WHERE  ITEMS > BASICS;
*-----;

```

We are reading the column ITEMS from the permanently stored (two-level name) SAS dataset PROC.SQL. The resulting output is as follows.

EX 0 - PAPER TITLE	
	ITEMS

	34
	18
	52
	40

Unfortunately, the paper title was frequently typoed in pre-publication advertising - usually by missing the period in PROC.SQL (and thereby missing the point!) We can now demonstrate that even this rendering is syntactically valid, because of the "alias" feature of PROC SQL. The following code produces the exact same output as the originally intended syntax.

```

*-----;
TITLE 'EX 0 - PAPER TITLE';
RUN;

DATA PROC;
  SET PROC.SQL;

PROC SQL;
  SELECT ITEMS
  FROM   PROC SQL
  WHERE  ITEMS > BASICS;
*-----;

```

Here, we are reading the temporary dataset WORK.PROC using the table alias SQL, with the same results as the previous intended code. Let this serve as an early lesson. It's quite possible to produce syntactically correct code which could yield perfectly meaningless results, without much effort at all. Be careful.

For the remainder of this paper, we will be using a fictitious set of medical practice data which is contained in two SAS datasets: dataset PTDEMOG contains variables PATIENT, PRIMDOC (patient's primary doctor), DOB (patient's date of birth) and SEX (patient's sex); dataset PTVISITS contains information about ten visits made by these patients to the doctor's office in variables PATIENT, VISDATE, VISDOC (doctor seen), HX (salient patient history item for visit), DX (diagnosis at time of visit), RX (prescription written -if any- at time of visit) and CHARGES (for the visit.) Although these data are obviously of a simple sample nature, the techniques we employ are totally expandable to all real-life data situations. The source code to create the datasets is as follows.

```

*-----;
DATA PTDEMOG;
  INPUT @001 PATIENT $2.
        @004 PRIMDOC $5.
        @010 DOB      YYMMDD6.
        @017 SEX      $1.;
  FORMAT DOB MMDDYY8.;

CARDS;
P1 SMITH 540221 M
P2 BROWN 431111 M
P3 BROWN 380818 F
P4 SMITH 610704 F
P5 JONES 331219 M
P6 BROWN 420525 F
P7 JONES 660606 F ;
*-----;

DATA PTVISITS;
  INPUT @001 PATIENT $2.
        @004 VISDATE YYMMDD6.
        @011 VISDOC  $5.
        @017 HX      $3.
        @021 DX      $3.
        @025 RX      $4.
        @030 CHARGES 6.2;
  FORMAT VISDATE MMDDYY8.
         CHARGES 6.2;

CARDS;
P1 910511 SMITH 243 087 1831 074.50
P1 921104 SMITH 465 131 NONE 128.00
P1 930122 SMITH 164 676 NONE 085.00
P2 911021 BROWN 227 257 1798 039.00
P3 920907 BROWN 140 673 NONE 110.75
P3 930909 JONES 589 324 2978 068.50
P4 930515 SMITH 995 270 NONE 032.00
P5 920717 JONES 658 195 NONE 106.85
P6 910521 BROWN 754 042 NONE 098.00
P7 940111 JONES 576 357 1130 135.00 ;
RUN;
*-----;

```

It is important to keep in mind the many functional equalities between the SQL procedure and the DATA step, as well as between PROC SQL and some of the other SAS basic building block procedures, e.g. PROC SORT and PROC SUMMARY (or PROC MEANS). SAS *datasets* are composed of *observations* and *variables*. This statement is directly translatable to: SQL *tables* are composed of *rows* and *columns*. In fact, for all intents and purposes, SAS datasets and SQL tables are totally interchangeable in the course of a SAS session.

We typically use the DATA step to read in raw data and to create new computed variables based on manipulations of raw data. New computed columns can also be created in PROC SQL with all the same computational power and features of DATA step assignment statements. This is accomplished quite simply in PROC SQL by including the computational code in the SELECT statement as a new "column", and optionally giving the new column a name (column alias) by using the AS keyword. If the newly created column is not named in this manner, it will still be created, and listed in the SELECT columnar output display, but there will be no column header, and reference to the new "column" will have to be made by repeating the computational code. In the following example, we list the VISDOC, PATIENT and a newly created variable, AGE. A few other features are employed which will be discussed following the output.

```

*-----;
TITLE "EX 1 - 'AS'";
RUN;

PROC SQL NUMBER;

    SELECT  PV.VISDOC,
            PV.PATIENT,
            INT((VISDATE-DOB)/365.25) AS AGE
    FROM    PTDEMOG AS PD,
            PTVISITS AS PV
    WHERE   PD.PATIENT = PV.PATIENT
    ORDER BY PV.VISDOC ;

QUIT;
*-----;

```

This example shows two different uses of AS. It is first used, as noted above, in the SELECT statement to give the alias AGE to the newly computed column. In cases like this, when creating column aliases, AS is required. It is then used in the FROM statement to give aliases to the tables PTDEMOG and PTVISITS. These explicit uses of AS are optional; the FROM statement would have been just fine without them, but good coding practice suggests their use, or even over-use. The use of table aliases here are themselves optional. The WHERE statement could have been written as WHERE PTDEMOG.PATIENT = PTVISITS.PATIENT, but the aliases make the coding a little briefer. The one instance

when AS is required in conjunction with a table name is when the table is being CREATED from an SQL query expression. We'll see an example of this towards the end of the paper.

DB2 SQL allows both column and table aliasing like PROC SQL, but unlike the SAS implementation, does not support the AS keyword in these processes. Only the implicit form of aliasing as mentioned earlier (without the AS) is available. However, similar to the last point made above, AS is required in DB2 when a view is created from a SELECT expression (unlike PROC SQL, DB2 does not currently support creating tables from SELECT.)

The above code yields the following output.

EX 1 - 'AS'			
Row	VISDOC	PATIENT	AGE
1	BROWN	P2	47
2	BROWN	P6	48
3	BROWN	P3	54
4	JONES	P3	55
5	JONES	P7	27
6	JONES	P5	58
7	SMITH	P4	31
8	SMITH	P1	38
9	SMITH	P1	38
10	SMITH	P1	37

NOTE: This output, as well as all other examples in this paper are not actual output, but are rather typed in here to accommodate the columnar formatting requirements of the Proceedings. Many code samples are likewise rearranged to fit into the columns, and are not optimally indented or line-spaced.

Unlike PROC PRINT, the output from a PROC SQL SELECT statement will not automatically include row (observation) numbers. This is easily included in the display by including the NUMBER statement on the PROC SQL statement. Also, the ORDER BY statement will cause the data to be displayed in the desired order (ASC is the default; DESC can also be specified.) It is important to realize that the ORDER BY statement refers to the *display order only*. There is no internal ordering of data in a relational SQL table.

As mentioned above, the age data is computed by the code INT((VISDATE-DOB)/365.25), and is named by the AS AGE column alias. If this newly created column AGE is to be referred to in subsequent code in the PROC SQL, it can be done so by using the CALCULATED

keyword as is shown in the next example. Prior to version 6.08, you would have had to use the calculational code every time you wanted to refer to the age column.

Suppose we now want to group our data into subgroups based on age categories by decade: 20's, 30's, etc. This is easily accomplished in a DATA step by a series of IF-THEN-ELSE statements, or a SELECT statement. Although there is no equivalent to IF-THEN-ELSE in PROC SQL, there is a very powerful alternate to the DATA step SELECT statement, namely CASE. Keep in mind the possibly confusing terminology here. The PROC SQL "CASE" statement is functionally similar to the DATA step "SELECT" statement, not the PROC SQL "SELECT" statement. In our example, we will create a new column AGEGROUP by using CASE as follows.

```

*-----;
TITLE 'EX 2 - 'CASE & CALCULATED'';
RUN;

PROC SQL;

    SELECT PV.VISDOC,
           PV.PATIENT,
           PV.VISDATE,
           INT((PV.VISDATE-PD.DOB)/365.25)
           AS AGE,
           CASE
             WHEN (00 LE CALCULATED AGE LE 19)
               THEN "PDS"
             WHEN (20 LE CALCULATED AGE LE 29)
               THEN "20'S"
             WHEN (30 LE CALCULATED AGE LE 39)
               THEN "30'S"
             WHEN (40 LE CALCULATED AGE LE 49)
               THEN "40'S"
             WHEN (50 LE CALCULATED AGE LE 59)
               THEN "50'S"
             WHEN (60 LE CALCULATED AGE LE 69)
               THEN "60'S"
             WHEN (70 LE CALCULATED AGE LE 79)
               THEN "GER"
             ELSE
               "???"
           END
           AS AGEGROUP
    FROM PTDEMOG AS PD,
         PTVISITS AS PV
   WHERE PD.PATIENT = PV.PATIENT
   ORDER BY PV.VISDOC;

QUIT;
*-----;

```

The PROC SQL CASE statement always yields a single value. In this example, the value of the calculated column AGE for each row is evaluated over the entire CASE statement, and a single value is created AS AGEGROUP, for each row. The CASE statement is extremely powerful and can be used anywhere in a PROC SQL run where a single column value is expected. There are other forms of the CASE

statement as well, similar to the forms of the DATA step SELECT statement. In fact it's best to conceive of the CASE statement as an SQL version of the DATA step SELECT statement. Note that we've included one more column in the current example, VISDATE, and that PROC SQL displays it with its defined display format, MMDDYY8. Unfortunately, neither CALCULATED nor CASE are currently supported in DB2 SQL.

Here is the output from the above code.

EX 2 - 'CASE & CALCULATED'				
VISDOC	PATIENT	VISDATE	AGE	AGEGROUP
BROWN	P2	10/21/91	47	40'S
BROWN	P6	05/21/91	48	40'S
BROWN	P3	09/07/92	54	50'S
JONES	P3	09/09/93	55	50'S
JONES	P7	01/11/94	27	20'S
JONES	P5	07/17/92	58	50'S
SMITH	P4	05/15/93	31	30'S
SMITH	P1	11/04/92	38	30'S
SMITH	P1	01/22/93	38	30'S
SMITH	P1	05/11/91	37	30'S

SECTION II - A PROC of DISTINCTION

A quick examination of the data will reveal that some patients have had more than one visit, and in one case, to more than one doctor. Obviously, in a real-world example, this would be the rule rather than the exception. If we want to remove the duplicate visit situation from our output to see just who visited who, we can use the DISTINCT statement to return only unique combinations of patients and doctors seen. Notice that DISTINCT applies to the *group* of columns which follow it, not just the immediately following column. This is in contrast to other situations where a qualifier typically refers to the immediately preceding (or following) columns only, such as the DESCENDING option in the BY statement within PROC SORT.

```

*-----;
TITLE 'EX 3 - DISTINCT';
RUN;

PROC SQL;
    SELECT DISTINCT PATIENT,
                   VISDOC
    FROM PTVISITS;

QUIT;
*-----;

```

Here is the output from the above code.

EX 3 - DISTINCT	
PATIENT	VISDOC
P1	SMITH
P2	BROWN
P3	BROWN
P3	JONES
P4	SMITH
P5	JONES
P6	BROWN
P7	JONES

Patient P3 was seen by someone other than his primary doctor on one visit, and therefore shows up in the output twice since PROC SQL is evaluating the PATIENT/VISDOC combination for uniqueness.

At this point, let's add the COUNT function to the DISTINCT picture to produce a total number of visits for each patient. To those new to SQL, the following code would probably look sufficient.

```

*-----;
TITLE 'EX 4 - COUNT (BAD)';
RUN;

PROC SQL;
  SELECT  DISTINCT PATIENT,
           COUNT(*)
  FROM    PTVISITS;

QUIT;
*-----;

```

As seen in the resulting output, it is not.

EX 4 - COUNT (BAD)	
PATIENT	
P1	10
P2	10
P3	10
P4	10
P5	10
P6	10
P7	10

Not only does this not give us the desired result, it also generates a warning note as follows:

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

This is because PROC SQL resolves the function across the entire table unless we give it more explicit

instructions. We'll see this problem crop up again later. For now we will solve the problem by using a GROUP BY statement as follows (and we'll also give the COUNT(*) column an alias.)

```

*-----;
TITLE "EX 5 - COUNT (BETTER)";
RUN;

PROC SQL;
  SELECT  DISTINCT PATIENT,
           COUNT(*) AS NUMVIS
  FROM    PTVISITS
  GROUP BY PATIENT;

QUIT;
*-----;

```

Here is the output.

EX 5 - COUNT (BETTER)	
PATIENT	NUMVIS
P1	3
P2	1
P3	2
P4	1
P5	1
P6	1
P7	1

Now that we've seen the inter-relationship between COUNT(*) and GROUP BY, we can try a somewhat more realistic example, including the doctor who attended the patient.

```

*-----;
TITLE "EX 6 - COUNT (BEST)";
RUN;

PROC SQL;
  SELECT  DISTINCT PATIENT,
                   VISDOC,
                   COUNT(*) AS NUMVIS
  FROM    PTVISITS
  GROUP BY PATIENT,
           VISDOC;

QUIT;
*-----;

```

Here is the output.

EX 6 - COUNT (BEST)		
PATIENT	VISDOC	NUMVIS
P1	SMITH	3
P2	BROWN	1
P3	BROWN	1
P3	JONES	1
P4	SMITH	1
P5	JONES	1
P6	BROWN	1
P7	JONES	1

Again, we note the P3/JONES row impacting results, splitting her two visits into one for each doctor.

SECTION III - Let's Get Together

The one feature of PROC SQL that is perhaps touted most often as the real reason for its existence is its ability to easily and powerfully merge multiple tables together with many options either unavailable, or quite difficult to perform using DATA steps alone. The following example demonstrates this basic feature while also using a WHERE statement to limit the processing of rows selected for display based on values of one column in one of the merging tables. Here we show visit information for female patients only.

```

*-----;
TITLE "EX 7 - SIMPLE JOIN-FEMALES ONLY";
RUN;

PROC SQL;

  SELECT  PV.PATIENT,
          PV.VISDOC,
          PV.VISDATE
    FROM  PTDEMOG AS PD,
          PTVISITS AS PV
   WHERE  PD.PATIENT = PV.PATIENT
      AND  PD.SEX = 'F'
   ORDER BY  PV.VISDOC;

QUIT;
*-----;

```

Here is the output for the above code.

EX 7 - SIMPLE JOIN-FEMALES ONLY			
PATIENT	VISDOC	VISDATE	
P3	BROWN	09/07/92	
P6	BROWN	05/21/91	
P3	JONES	09/09/93	
P7	JONES	01/11/94	
P4	SMITH	05/15/93	

While the above demonstrated merging ability is an extremely important aspect of PROC SQL, there are additional features inherent in the procedure for working with multiple tables simultaneously which add to the SAS System user's cache of data manipulation tools. Notable among these are various types of "subqueries." A basic subquery can be used to form the business end of a WHERE statement. It allows you to selectively choose rows for display from one table based on selection criteria derived from another table. The above example could have been coded as follows to make use of a subquery. Don't be fooled by the simplicity of our examples. These features are extremely powerful!

```

*-----;
TITLE 'EX 8 - SUBQUERY-FEMALES ONLY';
RUN;

PROC SQL;

  SELECT  PV.PATIENT,
          PV.VISDOC,
          PV.VISDATE
    FROM  PTVISITS AS PV

   WHERE  PV.PATIENT IN
          (SELECT  PD.PATIENT
            FROM  PTDEMOG AS PD
           WHERE  PD.SEX='F')

   ORDER BY  PV.VISDOC;

QUIT;
*-----;

```

The subquery is the code in parentheses in the WHERE statement (called the inner query) which is evaluated before the outer query. In this example, it yields a series of values for PV.PATIENT which become the IN portion of the WHERE statement. After evaluation of the subquery, the WHERE statement is read internally as WHERE PV.PATIENT IN ('P3', 'P4', 'P6', 'P7'). Here is the output of the above code. Except for the title, it is identical to the previous output.

EX 8 - SUBQUERY-FEMALES ONLY			
PATIENT	VISDOC	VISDATE	
P3	BROWN	09/07/92	
P6	BROWN	05/21/91	
P3	JONES	09/09/93	
P7	JONES	01/11/94	
P4	SMITH	05/15/93	

Subqueries are a very powerful technique in PROC SQL. Suppose we had need to isolate all visits (and charges) by patients to physicians other than their primary doctor. This is easily handled by a subquery as follows.

```

*-----;
TITLE 'EX 9 - SUBQUERY-NONPRIMARY DOCS';
RUN;

PROC SQL;

  SELECT  PV.PATIENT,
          PV.VISDOC,
          PV.VISDATE,
          PV.CHARGES
    FROM  PTVISITS AS PV

   WHERE  PV.PATIENT || PV.VISDOC NOT IN
          (SELECT  PD.PATIENT || PD.PRIMDOC
            FROM  PTDEMOG AS PD);

QUIT;
*-----;

```

Note once again that almost all DATA step operations (in this case, character value concatenation) are available for use in PROC SQL. In this case, the only patient-doctor combination from the visits file that does NOT occur in the demographic file is P3-JONES. Here is the output.

EX 9 - SUBQUERY-NONPRIMARY DOCS			
PATIENT	VISDOC	VISDATE	CHARGES
P3	JONES	09/09/93	68.50

We can also use DISTINCT to isolate the non-primary doctor situation, in conjunction with the alias/qualification of data columns technique covered earlier. This is illustrated as follows.

```

*-----;
TITLE 'EX 10 - DISTINCT-NONPRIMARY DOC';
RUN;

PROC SQL;
  SELECT  DISTINCT FV.PATIENT,
           FV.VISDOC
  FROM    PTVISITS AS FV,
          PTDEMOG AS PD
  WHERE   FV.PATIENT = PD.PATIENT
         AND FV.VISDOC <> PD.PRIMDOC;

QUIT;
*-----;

```

Here is the resulting output.

EX 10 - DISTINCT-NONPRIMARY DOC	
PATIENT	VISDOC
P3	JONES

A word about valid "NOT-EQUALS" syntax may be in order here. SAS accepts NE as usual - other languages may prefer the "<>" as used above. SAS also accepts "<=", but puts out a note:

NOTE: The "<>" operator is interpreted as "not equals".

Yet another approach (do we really need another) to identifying this situation involves the EXISTS statement. EXISTS basically allows you to answer the question - "Is there a row in the subquery that fits the bill, yes or no?". In that regard, it is closely related to ANY (see later). The primary distinction between the two is that EXISTS focuses on a row satisfying certain conditions (usually equalities), whereas ANY focuses on a value within that row satisfying certain conditions (usually comparisons). Here we isolate the

P3/JONES combination using EXISTS.

```

*-----;
TITLE 'EX 11 - EXISTS-NONPRIMARY DOC';
RUN;

PROC SQL;
  SELECT  PATIENT,
          VISDOC
  FROM    PTVISITS AS FV
  WHERE   NOT EXISTS
          (SELECT  PATIENT,
                   PRIMDOC
          FROM    PTDEMOG AS PD
          WHERE   PD.PATIENT=FV.PATIENT
                 AND PD.PRIMDOC=FV.VISDOC);

QUIT;
*-----;

```

Here is the output from the above code.

EX 11 - EXISTS-NONPRIMARY DOC	
PATIENT	VISDOC
P3	JONES

SECTION IV - I'd Rather Do It Myself

Many SAS programmers may be unaware of the fact that it is perfectly legal to issue two discrete SET statements for the same dataset in a single DATA step. While this may at first seem to fall into the realm of "Stupid SAS Tricks", anyone who has spent time with Bob Virgile and his "Puzzle Corner" will have learned that it can be an incredibly powerful technique.

Up until now, we have only looked at subqueries on a second table; we now demonstrate joining a table with itself. The set-logic orientation of SQL makes this concept of working with more than one "image" of a table at one time much more common than it is in the SAS DATA step. In SQL terms, it is known as a "reflexive join". It becomes particularly necessary when trying to answer queries of the form, "Find all rows where X is true, and all other associated rows", or, "Find all rows where X is true, but with no other associated rows having Y being true".

In the next example, we will look for all *other* diagnoses ever given for patients with a specified "trigger" diagnosis (e.g. '131'). To streamline the output results, it may also be appropriate to remove the trigger diagnosis from the display. Obviously, this kind of query could be enhanced to identify associated diagnoses which occurred before or after the trigger, perhaps to aid in early identification of risk or to better plan follow-up treatment.

The code is as follows.

```

*-----;
TITLE 'EX 12 - REFLEXIVE JOIN';
RUN;

PROC SQL;
  SELECT PV2.PATIENT,
         PV2.DX
    FROM PTVISITS AS PV1,
         PTVISITS AS PV2
   WHERE PV2.DX    <> '131'
        AND PV2.PATIENT = PV1.PATIENT
        AND PV1.DX    = '131';
QUIT;
*-----;

```

We are accessing PTVISITS twice, with each access independent of the other. We use the two aliases, PV1 and PV2, to create the two separate "views" of the same table. Here is the resulting output.

EX 12 - REFLEXIVE JOIN	
PATIENT	DX
P1	087
P1	676

Two additional PROC SQL tools, ANY and ALL, are easier to comprehend from seeing them in action than by explanation, so let's dive right in. In this example, the subquery is also a reflexive join, but it does not have to be the same table at all.

```

*-----;
TITLE 'EX 13 - ALL';
RUN;

PROC SQL;
  SELECT PATIENT,
         CHARGES
    FROM PTVISITS
   WHERE VISDOC='SMITH'
        AND CHARGES > ALL
              (SELECT CHARGES
               FROM PTVISITS
               WHERE VISDOC='BROWN');
QUIT;
*-----;

```

The above code produces the following output.

EX 13 - ALL	
PATIENT	CHARGES
P1	128.00

This shows the one and only row where SMITH's charges exceed *all* of BROWN's. Note that the result could easily have been multiple rows.

Now observe the result when we use the same code, but substitute ANY for ALL.

```

*-----;
TITLE 'EX 14 - ANY';
RUN;

PROC SQL;
  SELECT PATIENT,
         CHARGES
    FROM PTVISITS
   WHERE VISDOC = 'SMITH'
        AND CHARGES > ANY
              (SELECT CHARGES
               FROM PTVISITS
               WHERE VISDOC='BROWN');
QUIT;
*-----;

```

Here is the output.

EX 14 - ANY	
PATIENT	
P1	74.50
P1	128.00
P1	85.00

This shows *all* of SMITH's charges *except* the one (P4/\$32) which is lower than the lowest of *any* of Brown's charges (P2/\$39). Note that the result could easily have been a single row.

Once again, let's throw a function (MAX this time) into the mix. In this example, we're trying to find the highest charge each patient has ever incurred in a single visit. From our experience earlier, many of you will probably suspect that the following obvious code won't work.

```

*-----;
TITLE 'EX 15 - HIGHEST (BAD)';
RUN;

PROC SQL;
  SELECT DISTINCT PATIENT,
                  MAX(CHARGES)
    FROM PTVISITS;
QUIT;
*-----;

```

As the following output shows, you were right.

EX 15 - HIGHEST (BAD)	
PATIENT	
P1	135.00
P2	135.00
P3	135.00
P4	135.00
P5	135.00
P6	135.00
P7	135.00

Once again, SAS produces the "remerging" note, and resolves the function across the entire dataset.

Will a GROUP BY solve the problem, as it did earlier? Yes - it will. However, because of volume (i.e. discrete number of GROUP values), that approach may only be suitable for situations of a similar size to this example data. In the real world, GROUP BY is appropriate for studies by SEX, HOSPITAL, perhaps DOCTOR, etc. A better approach to low-level summaries (i.e. specific to one key, but where there are many discrete keys) may be a "correlated subquery". This technique is used in the following example to select the highest charge paid by each patient on any visit.

```

*-----;
TITLE 'EX 16 - CORRELATED SUBQUERY';
RUN;

PROC SQL;
  SELECT DISTINCT PVL.PATIENT,
                 PVL.CHARGES
  FROM PTVISITS AS PVL
 WHERE PVL.CHARGES =
       (SELECT MAX(CHARGES)
        FROM PTVISITS AS PV2
         WHERE PVL.PATIENT=PV2.PATIENT);

QUIT;
*-----;

```

Output from the above code is as follows, with discussion to follow.

EX 16 - CORRELATED SUBQUERY	
PATIENT	CHARGES
P1	128.00
P2	39.00
P3	110.75
P4	32.00
P5	106.85
P6	98.00
P7	135.00

In a "correlated subquery", we are asking PROC SQL to identify, in a subquery, the one row (possibly rows), which satisfies some condition, from a group of "candidate" rows identified in the outer query. This is a complex concept, so let's look at it another way. We have already seen that functions operate on the "universe" of rows available to them. In the examples earlier, that universe was the entire table. A correlated subquery constrains the function to operate on small subsets of the table (the candidate rows from the outer query) one at a time. To achieve this, the key variables in the subquery have to be exactly correlated to those in the outer query.

In our example, only PATIENT needs to be correlated - however, be warned that the real world is seldom this simple. For instance, if we were evaluating charges by doctor/patient combination, both patient equality and doctor equality would have to be established in the subquery.

Another side-effect of the operation of correlated subqueries is that many "universes" (albeit smaller ones) have to be evaluated. To accomplish this, SQL has to establish, retain, and discard working set after working set. This technique is widely regarded as a "hog", and for good reason. It may be worthwhile to look into rewriting such queries as "joins." Theoretically, it should be possible, but the development/maintenance of such code may make it less desirable than correlated subqueries, even with their intrinsic inefficiencies.

SECTION V - You Could Look It Up

How often have you needed to merely display the descriptive information about a SAS dataset, or better yet, collect all the data and use it as a dataset itself? Of course, the place to turn for this type of information is PROC CONTENTS. This procedure does an admirable job (despite the mysterious disappearance of the HISTORY option), but is somewhat inflexible in terms of default output. Once again, PROC SQL provides an alternative which does an excellent job if you are in need of this type of supportive data.

In version 6.08, a few enhancements were added to PROC SQL in addition to the above mentioned CALCULATED keyword. Notable among them is the DICTIONARY component which comes in eight garden varieties. You can build tables (datasets) containing information about CATALOGS (SAS libraries, catalogs, entries), COLUMNS (SAS dataset variables), EXTFILES (external files defined to your current session by filerefs), INDEXES (of SAS datasets), MEMBERS (SAS files and data libraries), OPTIONS (current settings), TABLES

(SAS data files) and VIEWS (SAS data views.) There is some overlap in the choices, but its all there. In addition, there are numerous preconstructed VIEWS of these types of data available in the SASHELP data library which provide selected subgroups of information based on the DICTIONARY component.

As a final example in this loooooonnnng paper, lets simply create a new table containing some descriptive data about the variables in our two working datasets. The following code will accomplish this.

```
*-----;
TITLE 'EX 17 - DICTIONARY';
RUN;

PROC SQL;

  CREATE TABLE EX17 AS

  SELECT  MEMNAME,
          NAME,
          TYPE,
          LENGTH,
          FORMAT

  FROM    DICTIONARY.COLUMNS

  WHERE  LIENAME='WORK'
        AND MEMNAME IN('PTDEMOG','PTVISITS');

QUIT;

PROC PRINT DATA=EX17;
RUN;
*-----;
```

The output is as follows.

EX 17 - DICTIONARY					
OBS	MEMNAME	NAME	TYPE	LENGTH	FORMAT
1	PTDEMOG	PATIENT	char	2	
2	PTDEMOG	PRIMDOC	char	5	
3	PTDEMOG	DOB	num	8	MMDDYY8.
4	PTDEMOG	SEX	char	1	
5	PTVISITS	PATIENT	char	2	
6	PTVISITS	VISDATE	num	8	MMDDYY8.
7	PTVISITS	VISDOC	char	5	
8	PTVISITS	HX	char	3	
9	PTVISITS	DX	char	3	
10	PTVISITS	RX	char	4	
11	PTVISITS	CHARGES	num	8	6.2

We could have simply listed the data by merely SELECTing it and not CREATEing TABLE EX17. We would then not have to PROC PRINT it - it would have

appeared as the PROC SQL default output. Making the data available to other reporting procedures via a dataset however, provides a very powerful tool for developing documentation on all of our data entities.

SECTION VI - Now RUN; with it (even though you don't have to!)

This last section title is our clever way to get you to play with PROC SQL. Throw a RUN; statement after your next PROC SQL and see what happens.

OK, we've now dug into the PROC SQL bag of tricks to come up with a few techniques which only begin to show the power available when you scratch a little harder and deeper than merely at the surface. This exposition is by no means extensive. Our aim here is to show that PROC SQL is a SAS powerhouse, once you take your first steps beyond the basics. We only meant to point you in the right direction and give you a peek over the horizon. We hope we have whetted your appetite to delve further into this new frontier (do you think we used enough cliches there?) Happy coding.

SAS is a registered trademark of SAS Institute, Cary, NC.

DB2 is a registered trademark of International Business Machines, Inc., Armonk, NY.

The authors can currently be contacted as follows:

Alan Dickson
40 Riverbend Drive
Yarmouth, ME 04096

(207) 846-8957
71311.3341@compuserve.com

Ray Pass
ASG, Inc.
1100 Summer Street
Stamford, CT 06905

voice: (203) 356-9540
fax: (203) 967-8644

e-mail: 0005962116@mcimail.com