

Less Well Known Features of SASTM PROC SQL

Timothy J. Harrington, Trilogy Consulting Corporation, Kalamazoo, MI.

ABSTRACT

This paper discusses the less well known features and options of PROC SQL, all of which are listed in SAS HELP.

INTRODUCTION

There are several less well known features of PROC SQL which would be useful to know in some common programming situations. This paper explains these features, discusses where such features would be useful, how they compare with other programming methods, and shows examples in each case. The topics are listed alphabetically. In earlier versions of SAS (prior to 6.12) some of these facilities may not have been implemented.

SAMPLE DATA

The examples shown in this paper use the following data sets, STUDYA and STUDYB. These data sets contain the demographic information for 13 and 7 patients respectively in a hypothetical clinical trials study. The columns are:

PATIENT - Patient ID number (numeric)
 DATEVIS - The date of visit when this data was obtained (numeric)
 DATEBRTH - Date of birth (numeric)
 SEX - Gender (numeric 1=Male, 2=Female)
 RACE - Race (numeric 1=White, 2=Black, 3=Hispanic, 4=Asian, 5=Other)
 DOCTOR - The name of the investigating physician (15 byte character)
 PRINCIPAL - The name of the Principal Investigator (15 byte character)

OBS	PATIENT	DATEVIS	DATEBRTH	SEX	RACE	DOCTOR	PRINCIPAL
1	2001	28OCT97	14JUN41	Male	White	Para	Johnson
2	2002	22OCT97	31AUG58	Male	Black	Sargent	Johnson
3	2003	28OCT97	08SEP42	Male	Black	Sargent	Higgins
4	2004	20OCT97	10SEP41	Male	White	Para	Johnson
5	2005	31OCT97	14DEC57	Female	Black	Sargent	Johnson
6	2007	12NOV97	21JUN55	Male	White	Reichman	Johnson
7	2008	23OCT97	06AUG47	Male	White	Reichman	Davies
8	2009	14NOV97	22FEB73	Female	Black	Sargent	Johnson
9	2010	11DEC97	05JUN70	Male	White	Beal	Johnson
10	2011	16DEC97	16NOV68	Male	White	Sweet	Higgins
11	2012	05JAN98	09JUL68	Male	White	Beal	Higgins
12	2014	05JAN98	01AUG61	Male	Hisp	Hewitt	Davies
13	2015	13JAN98	25DEC59	Male	White	Hewitt	Johnson

OBS	PATIENT	DATEVIS	DATEBRTH	SEX	RACE	DOCTOR	PRINCIPAL
1	2084	11FEB98	01JUL73	Male	White	Beal	Saxby
2	2085	19DEC97	19MAY78	Male	White	Sweet	Davies
3	2086	28NOV97	10MAR41	Male	White	Para	Davies
4	2087	24NOV97	25JUL60	Male	White	Sargent	Saxby
5	2088	04NOV97	01JUN52	Male	Black	Sargent	Saxby
6	2089	22OCT97	18DEC59	Male	White	Sargent	Saxby
7	2090	14OCT97	28NOV42	Male	White	Para	Saxby

SQL STATEMENTS

Alter Table

This command is used with the modify clause to change column labels, formats, informats, or the length of a character column. If the new length is longer than the old length the character string is padded with spaces. If the new length is shorter than the old length truncation takes place. The length of a numeric variable cannot be changed. Alter can also be used to drop (delete) one or more columns. Alter and Modify can be used with both tables and views. The example below shows a table STUDY1, which is a copy of STUDYA, and changes the label on DATEVIS from 'Date of Visit' to 'Date of Sign Up', changes the format of DATEBRTH from DATE7 to MMDDYY8., and drops the RACE column.

```
alter table study1
  modify datevis label='Date of Sign Up'
  modify datebrth format=mmddyy8.
  drop race;
```

OBS	PATIENT	DATEVIS	DATEBRTH	SEX	DOCTOR	PRINCIPAL
1	2001	28OCT97	06/14/41	Male	Para	Johnson
2	2002	22OCT97	08/21/58	Male	Sargent	Johnson
3	2003	28OCT97	09/08/42	Male	Sargent	Higgins
4	2004	20OCT97	10/21/41	Male	Para	Johnson
5	2005	31OCT97	12/14/57	Female	Sargent	Johnson
6	2007	12NOV97	06/21/55	Male	Reichman	Johnson
7	2008	23OCT97	07/06/47	Male	Reichman	Davies
8	2009	14NOV97	02/22/73	Female	Sargent	Johnson
9	2010	11DEC97	05/09/70	Male	Beal	Johnson
10	2011	16DEC97	11/16/68	Male	Sweet	Higgins
11	2012	05JAN98	07/09/68	Male	Beal	Higgins
12	2014	05JAN98	08/01/61	Male	Hewitt	Davies
13	2015	13JAN98	12/25/59	Male	Hewitt	Johnson

Alter and **Modify** cannot be used with **RENAME** or **KEEP** or to change the data type.

The Between Condition

Between selects rows where column values are within a range of values in a **WHERE** or **HAVING** clause. The **Between** condition includes the boundary values themselves, though care should be taken regarding the **FUZZ** factor (different host hardware imprecision). The example below creates a table **STUDY2** with those patients with id numbers 2000 - 2005 inclusive and aged between 30 and 50 inclusive.

```
create table study2 as
select *
  from studya
 where patient between 2000 and 2005
        having (datevis-datebrth)/365.25
           between 30 and 50;
```

Table STUDY2 - Patients 2000 - 2005 Aged Between 30 and 50

PATIENT	DATEVIS	DATEBRTH	SEX	RACE	DOCTOR	PRINCIPL
2004	200CT97	10FEB63	Male	White	Para	Johnson

The Calculated Reference.

This refers to a created variable name not in the table, containing the result of a calculation involving one or more columns. Calculated variables can be nested.

In this example the calculated variable **AGE** is created and is used to store the difference between the date of visit and the date of birth. The table, **STUDY3**, is created with those patients aged 50 or over and the column **YGT50** is created containing the number of whole years those patients are aged over 50.

```
create table study3 as
select *,
  (datevis-datebrth)/365.25 as age,
  int(calculated age-50) as ygt50
  from studya
   having calculated age ge 50;
```

Table STUDY3 - Patients Aged 50 or Over

PATIENT	DATEVIS	DATEBRTH	SEX	RACE	DOCTOR	PRINCIPL	AGE	YGT50
2001	18OCT97	14JUN41	Male	White	Para	Johnson	56.33	6
2003	28OCT97	06SEP47	Male	Black	Sargent	Higgins	50.14	0

The Coalesce Function

When performing left, right, or full joins **coalesce** is used to specify how to handle resulting missing values, that is what these missing values are to be substituted with. In the example below the table **STUDY4** is produced by 'left' joining the table **STUDYA** with table **STUDY3** (above) by matching the column **PATIENT**. The observations in **STUDY4** where the patients are common to both contributing data sets are shown in bold.

```
create table study4 as
select a.patient, a.datebrth,
       a.datevis, a.sex, a.race,
       b.doctor as physican, a.principl
  from studya a left join study3 b
    on (a.patient=b.patient)
   order by patient;
```

Table STUDY4 - Missing Values of DOCTOR become PHYSICAN in a Left Join

OBS	PATIENT	DATEVIS	DATEBRTH	SEX	RACE	PHYSICAN	PRINCIPL
1	2001	18OCT97	14JUN41	Male	White	Para	Johnson
2	2002	22OCT97	21APR61	Male	Black		Johnson
3	2003	28OCT97	06SEP47	Male	Black	Sargent	Higgins
4	2004	20OCT97	10FEB63	Male	White		Johnson
5	2005	21OCT97	14DEC67	Female	Black		Johnson
6	2007	12NOV97	21OVR65	Male	White		Johnson
7	2008	23OCT97	06JUN67	Male	White		Davies
8	2009	14NOV97	22FEB73	Female	Black		Johnson
9	2010	11DEC97	09JUN70	Male	White		Johnson
10	2011	16DEC97	16NOV68	Male	White		Higgins
11	2012	05JAN98	05JUL68	Male	White		Higgins
12	2014	05JAN98	01APR61	Male	Hispanic		Davies
13	2015	13JAN98	25DEC59	Male	White		Johnson

All of the columns except **DOCTOR** are taken from the table with the alias **a**, that is **STUDYA**. The values of **PHYSICAN** (Physician) are taken from **DOCTOR** in table **b**, **STUDY3**, hence where a patient is present in **STUDYA** but not in **STUDY3** the value of 'b.DOCTOR' and hence **PHYSICAN** is missing. If the rules of the study state where a patient is not present in **STUDY3** (not over 50 years of age) the Principal Investigator **PRINCIPL** becomes **PHYSICAN** instead of **DOCTOR** the **coalesce**

function is now used to perform this substitution:

```
create table study5 as
  select a.patient, a.datebrth,
         a.datevis, a.sex, a.race,
         coalesce(b.doctor, a.prncipl) as
           physcn length=15
           label='Assigned Investigator'
         from studya a left join study3 b
           on (a.patient=b.patient)
         order by patient;
```

Table STUDY5 - Substitution of Missing Join Values with Another Column

OBS	PATIENT	DATEVIS	DATEBRTH	SEX	RACE	PHYSICN
1	2001	24OCT97	14JUN41	Male	White	Papa
2	2802	22OCT97	31AGC68	Male	Black	Johnson
3	2003	28OCT97	08SEP47	Male	Black	Sargent
4	2004	20OCT97	10FEB63	Male	White	Johnson
5	2005	11OCT97	24DEC67	Female	Black	Johnson
6	2007	12NOV97	21JUN65	Male	White	Johnson
7	2006	29OCT97	06JUL67	Male	White	Davies
8	2003	14NOV97	22FEB73	Female	Black	Johnson
9	2010	11DEC97	05JUN70	Male	White	Johnson
10	2011	15DEC97	15NOV68	Male	White	Higgins
11	2012	05JAN98	03JUL68	Male	White	Higgins
12	2013	05JAN98	01NOV61	Male	Hispanic	Davies
13	2015	13JAN98	25DEC59	Male	White	Johnson

In table STUDY5 where there is no matching value of b.PATIENT b.DOCTOR is missing but the coalesce function substitutes these missing values with the corresponding values of a.PRINCIPL, the Principal Investigator, from STUDYA, and stores the resulting values in the new column PHYSICN.

Coalesce may be used to substitute values in a corresponding column of the same name in the table being joined to (e.g. coalesce(b.doctor, a.doctor)), or another column in the same table (e.g. coalesce(b.doctor, b.prncipl)), or a hard-coded value (e.g. coalesce(b.doctor, "NONE")). The column containing the result of the coalescence may be the same column name (inheriting the same column attributes) as either argument column. However, both arguments of the coalesce function must be compatible, that is of the same data types. (The PUT or INPUT function can be used for coalescing character and numeric types). If both input columns are character but of different lengths truncation may occur so a length should be specified.

The Contains Condition

Contains tests for the presence of a character sub-string in a WHERE clause. It is case sensitive and the sub-string may be blank or contain one or multiple blanks. Contains cannot be used on numeric variables or on formats (The PUT function, as shown in this example, should be used).

Table STUDY6 contains all the patients in STUDYA who have a date of visit in the month of November. Note the use of the PUT function with the DATE7 format because DATEVIS is a numeric SAS internal date.

```
create table study6 as
  select *
    from studya
   where put(datevis,date7.)
         contains 'NOV';
```

Table STUDY6 - Patients with a Date of Visit in November

PATIENT	DATEVIS	DATEBRTH	SEX	RACE	DOCTOR	PRINCIPL
2007	12NOV97	21JUN65	Male	White	Reichman	Johnson
2009	14NOV97	22FEB73	Female	Black	Sargent	Johnson

The Delete Statement

Delete removes one or more rows in a table according to a WHERE condition. Caution: *Delete used without a WHERE clause deletes the entire table contents!* (The data set itself is not deleted, nor are any of the column attributes such as labels and formats). Delete, when used with a view of one database deletes from the source database as well as from the view. Delete cannot be used on a view if there is not authorization to delete from the source database or on views with multiple source databases.

This example removes all the observations for patients with id numbers of 2005 and less. Table STUDY7 is a copy of STUDYA.

```
delete from study7
  where patient le 2005;
```

The Describe Statement

Describe returns information on the structure and contents of a table, or view. Such details are variable names, and variable attributes. The information is written to the log file, therefore describe is a useful debugging aid. If the FEEDBACK option is specified on the preceding PROC SQL more detailed information is shown. The example below is a description of the table STUDYA. Describe table describes a table. Describe view describes a view.

```
Describe table studya;
```

The following is written to the log file:

```
create table WORK.STUDYA( compress=YES
  bufsize=2048 )
  PATIENT num label='Patient',
  DATEVIS num format=DATE7.
informat=DATE. label='Date of Visit',
  DATEBRH num format=DATE7.
informat=DATE. label='Date of Birth',
  SEX num format=SEX. label='Sex',
  RACE num format=RACEF. label='Race',
  DOCTOR char(15) label='Investigator',
  PRINCIP char(15) label='Principal
Investigator'
);
```

The Dictionary Tables

These are tables located in a SAS system library called DICTIONARY and contain information about tables, views, libraries, files, and their contents. A series of built in SASHELP views are provided (with versions 6.08 and upward). All the available SASHELP views and their definitions are listed under the 'The SQL Procedure' of the SAS HELP menu. The example below creates a data set named VARINFO which contains information about the STUDYA data set. *Libname* is the library and *memname* is the data set or view name. The data set DICTIONARY.COLUMNS contains information on all data sets and all views in all the currently defined libraries. *Memtype* is the table type, a DATA set, VIEW, or Mddb (Multidimensional Database).

```
create table varinfo as
select *
  from dictionary.columns
  where libname = 'WORK' and
  memname = 'STUDYA';
```

Table VARINFO - The Contents of DICTIONARY.COLUMNS Pertaining to STUDYA

LIBNAME	MEMNAME	MEMTYPE	MEMNAME	TYPE	LEN	POS	LABEL	FORMAT
WORK	STUDYA	DATA	PATIENT	num	8	0	Patient	
WORK	STUDYA	DATA	DATEVIS	num	8	8	Date of Visit	DATE7.
WORK	STUDYA	DATA	DATEBRH	num	8	16	Date of Birth	DATE7.
WORK	STUDYA	DATA	SEX	num	8	24	Sex	SEX.
WORK	STUDYA	DATA	RACE	num	8	32	Race	RACEF.
WORK	STUDYA	DATA	DOCTOR	char	15	40	Investigator	
WORK	STUDYA	DATA	PRINCIP	char	15	45	Principal	

(Not all the variables in DICTIONARY.COLUMNS are shown here!)

The next example uses the SASHELP view VINDEXT to list the indices on the data sets currently in the work library. The table STUDY8 is a copy of the table STUDYA, but has the indices (created with CREATE INDEX) as listed below:

```
create table indices as
select *
  from sashelp.vindex
  where libname = 'WORK';
```

Table INDICES - The SASHELP view Containing Table Index Information

LIBNAME	MEMNAME	MEMTYPE	MEMNAME	INDEXPAGE	INDEXNAME	INDEXPOS	UNIQUE
WORK	STUDY8	DATA	PATIENT	COMPOSITE	APPOINT	0	
WORK	STUDY8	DATA	DATEVIS	COMPOSITE	APPOINT	8	
WORK	STUDY8	DATA	PATIENT	SIMPLE	PATIENT		yes

The Drop Statement

Drop deletes a table, view, or index (Not to be confused with DELETE, which removes observations, and not be confused with DROP in ALTER table/view, which removes a column). Drop does not delete the database(s) from which a view was created, only the view itself. The following examples delete the table OLDDATA, the views TEMP1, TEMP2, OLDVIEW1 thru OLDVIEW5, and remove the index APPOINT from the table STUDY8.

```
drop table olddata;
drop view temp1, temp2, oldview1-
oldview5;
drop index appoint from Study8;
```

The Exists Condition

Exists tests the result of a sub-query. In the example, if PATIENT has a value of 2009 in the STUDYA data set the contents of STUDY6 are copied to the output data set STUDY9. STUDY9 becomes an exact copy of STUDY6, (whether PATIENT 2009 is in STUDY6 or not). If PATIENT 2009 is not present in STUDYA, STUDY9 would be empty but have the same columns and attributes as STUDY6.

```
create table study9 as
  select *
    from study6
      where exists
        (select patient from studya
         where patient=2009);
```

The In and Is Conditions

In tests a variable or calculated result against a list of values (set membership). Is tests for NULL or MISSING. In this example table STUDY10 will have all the observations where the patient number is in the inclusive range 2000 to 2005 and the date of visit is not missing. NULL and MISSING are used where the data type of the column being tested is uncertain, such as when the value of a macro variable specifies a column name, which could be of numeric or character data type.

```
create table study10 as
  select *
    from studya
      where patient in
        (2000 2001 2002 2003 2004 2005)
        and datevis is not missing;
```

The Index Statement

Creates an index with one or more key columns to improve table access efficiency. An index may be one column or a

combination of two or more columns. If the index is one column the index is assigned the same name as the column to which it refers. If the index is to be made up of multiple columns it must be assigned a name. Indices may be assigned to columns with duplicate values, but to restrict an index to unique values the UNIQUE keyword must be specified. UNIQUE specifies no duplicates or missing values. Once an index has been defined it remains with the table or view, even after QUITting from the current PROC SQL, until it is removed by using the DROP keyword. Benchmark tests show that indices improve access speed by a factor of between 1:3 and 1:10, or even better for very large tables. However benchmark tests have also shown that conflict can occur between indices defined in a DATA step and indices defined in PROC SQL resulting in a decrease in performance. The action of the index is automatic, that is the index name does not have to be referred to when accessing the data set for it be effective. The index name and details are stored in the DICTIONARY tables for reference, as in the example in the DICTIONARY section above. The following example makes a unique (no duplicates) index of the column PATIENT and a compound index named APPOINT (appointment) out of the columns PATIENT and DATEVIS, and removes the index OLDINDX from the column DOCTOR. (The column name does not have to be specified when an index is dropped). STUDY8 is a copy of STUDYA.

```
create unique index
  patient on study8 (patient);

create index
  appoint on study8 (patient,datevis);

drop index oldindx on study8;
```

Insert and Into

Insert adds new rows to an existing table, it does not add new data to an existing row (To do this UPDATE is used). There are three methods for inserting, each of which is shown in the examples below, where the

table STUDY11 is initially a copy of STUDYA. In the first example the SET keyword is used to add a new patient, 2006, who is female and white (SEX=2, RACE=1). The INPUT function is used to store the date columns as SAS internal numeric dates. Also, two observations for PATIENT 2099 are added, duplicates can be inserted unless one or more of the columns are part of a UNIQUE index. In the case of PATIENT 2099 the date of visit and the date of birth are not specified so they are stored as missing values. In the second example new observations are being inserted from another table with the same columns and column data types. The observations for PATIENT 2087 and PATIENT 2090 are being copied from the table STUDYB and are then added to the STUDY11 table. In the third example the VALUES clause is being used to directly insert hard coded values (these could be the contents of macro variables) into the corresponding columns. A new observation for PATIENT 2075 is being created with March 8, 1998 as the date of visit and July 15, 1970 as the date of birth. The values must be of the correct data type. (The INPUT function cannot be used here). The patient is female and black, so SEX and RACE are both 2, the DOCTOR name is 'Jones', and the Principal Investigator Name is 'Davies'. In all three cases the new observations are appended to the table. To reorder the table in PATIENT sequence ORDER BY or a separate PROC SORT must be used.

Adds Observations 14, 15, and 16:

```
insert into study11
  set patient=2006,
    datevis=input('02FEB97',date7.),
    datebrth=input('11APR70',date7.),
    sex=2,
    race=1,
    doctor='Para',
    principl='Saxby'
  set patient=2099,
    sex=1,
    race=1,
    doctor='James',
    principl='Saxby'
  set patient=2099,
    sex=2,
    race=1,
    doctor='Wilson',
    principl='Saxby';
```

Adds Observations 17 and 18:

```
insert into study11
  (patient, datevis, datebrth, sex,
   race, doctor)
  select *
    from studyb
   where patient in (2087 2090);
```

Adds Observation 19:

```
insert into study11
  values (2075 '08MAR98'd '15JUL70'd 2
         2 'Jones' 'Davies');
```

Table STUDY11 - Inserting New Observations

OBS	PATIENT	DATEVIS	DATEBRTH	SEX	RACE	DOCTOR	PRINCIPAL
1	2001	14OCT97	14JUN41	Male	White	Para	Johnson
2	2002	22OCT97	11AUG68	Male	Black	Sargent	Johnson
3	2003	29OCT97	08SEP47	Male	Black	Sargent	Higgins
4	2004	20OCT97	10FEB63	Male	White	Para	Johnson
5	2005	31OCT97	14DEC67	Female	Black	Sargent	Johnson
6	2007	12NOV97	21JUN65	Male	White	Reichman	Johnson
7	2008	23OCT97	06JUL67	Male	White	Reichman	Davies
8	2009	14NOV97	22FEB73	Female	Black	Sargent	Johnson
9	2010	11DEC97	09JUN70	Male	White	Beal	Johnson
10	2011	16DEC97	16NOV68	Male	White	Sweet	Higgins
11	2012	05JAN98	09JUN68	Male	White	Beal	Higgins
12	2014	05JAN98	01AUG61	Male	Hispan	Hewitt	Davies
13	2015	13JAN98	25DEC59	Male	White	Hewitt	Johnson
14	2006	02FEB97	11APR70	Female	White	Para	Saxby
15	2099	.	.	Male	White	James	Saxby
16	2099	.	.	Female	White	Wilson	Saxby
17	2087	24NOV97	25JUL60	Male	White	Sargent	Saxby
18	2090	14OCT97	20NOV42	Male	White	Para	Saxby
19	2075	08MAR98	15JUL70	Female	Black	Jones	Davies

Insert Into and Macro Variables

Insert with Into is used to place run time data into a macro variable, similar to CALL SYMPUT in a DATA step. The following listing shows the execution of a macro SQLDEMO, which contains examples of three uses of macro variables with PROC SQL.

The first example selects the values of PATIENT, DATEVIS, and DOCTOR into the macro variables PAT, VIS, and DOC for PATIENT 2004. The NOPRINT option is to suppress output to the output window.

In the second example all the DOCTOR names are listed into the macro variable MEDPROS, each name separated by a comma (a space, or any other character(s) may be used as separators). The DISTINCT keyword prevents duplicates. The default order in which the DOCTOR names are

placed into the macro variable is the same order as the names appear in the table. However, ORDER BY may be used to sort the macro variable contents. Adding ORDER BY DOCTOR to this example code would store the names in alphabetical order. When a character variable is written to a macro variable trailing spaces are eliminated unless NOTRIM is specified, when the full length of the character variable including all embedded spaces is stored.

In the third example SQL system macro variables which contain information about the created table STUDY12 are being displayed in the log file. SQL system macro variables are always predefined by PROC SQL and are useful for debugging and benchmarking purposes. The following is a list of these macro variables:

SQLQBS is the number of observations processed.

SQLRC is a return code from the most recently executed segment of SQL code. SQLRC returns with one of the following values:

```

0  The segment ran successfully
4  A WARNING occurred
8  An ERROR occurred
16 A specific runtime ERROR occurred
24 A system ERROR occurred
12 or 28 An internal error, such as bug in
    PROC SQL was encountered.

```

SQLQOPS indicates the number of internal loops (amount of internal processing) that have taken place during the procedure.

SQLXRC and SQLXMSG contain DBMS specific return code information when the SQL Pass Through facility is used.

```

276 %macro sqldemo;
277
278 /* Example 1: Select as macro
variables */
279
280 proc sql noprint;
281   select patient, datevis, doctor
282     into :pat, :vis, :doc
283     from studya
284     where patient = 2004;
285 %put PAT= &pat VIS= &vis DOC=
&doc;
286
287 /* Example 2: Delimited macro string
*/
288
289   select distinct doctor
290     into :medpros separated by ','
291     from studya;
292 %put MEDPROS= &medpros;
293
294 /* Example 3: Predefined SQL macros
*/
295
296   create table study12 as
297     select *
298     from studya;
299
300 %put SQLQBS= &sqlqbs SQLRC= &sqlrc
SQLQOPS= &sqlqops;
301 quit;
302
303 %mend sqldemo;
304
305 %sqldemo;

```

PAT= 2004 VIS= 20OCT97 DOC= Para

MEDPROS=

Beal, Hewitt, Para, Reichman, Sargent, Sweet

NOTE: Compressing data set WORK.STUDY12 decreased size by 0.00 percent. Compressed is 1 pages; un-compressed would require 1 pages.

NOTE: Table WORK.STUDY12 created, with 13 rows and 6 columns.

SQLQBS= 13 SQLRC= 0 SQLQOPS= 36

NOTE: PROCEDURE SQL used 702k.

NOTE: 2524K of contiguous memory remained unallocated.

A further point to note is that in the second example, instead of using one macro variable, MEDPROS, to store all the values, a sequence of like macro variables may be used to store successive values. Example 2 could be modified as:

```

287  /* Example 2: Delimited macro
string */
288
289  select distinct doctor
290      into :medpro1 - :medpro99
291      from studya;

```

Now the first value of DOCTOR is stored in MEDPRO1, the second distinct value is stored in MEDPRO2, the third in MEDPRO3 and so on. If there are less than 99 values of DOCTOR only the number of values (in this case six) of MEDPRO are used., ie: MEDPRO1 = Beal, MEDPRO2 = Hewitt, MEDPRO3 = Para, MEDPRO4 = REICHMAN, MEDPRO5 = Sargent, MEDPRO6 = Sweet.

The Like Condition

Like tests for a pattern match with a character string in a WHERE or HAVING clause. The percent sign (%) specifies any number of characters, including a blank (missing value). The underscore (_) is a one character placeholder. The like operator is case sensitive. In this example all the DOCTOR names which contain the letter 'P' followed by two other characters and then an 'a' are selected together with all the names which contain a 'S' followed by a 't'. The 't' can be the next character after the 'S' or there can be any number of characters between the 'S' and the 't'.

```

create table study13 as
select *
from studya
where (doctor like 'P__a') or
      (doctor like 'S&t');

```

Table STUDY13 - Selecting Observations with the LIKE Operator

PATIENT	DATEVIS	DATEBTH	SEX	RACE	DOCTOR	PRINCIPAL
2001	14OCT97	14JUN41	Male	White	Para	Johnson
2002	22OCT97	31AUG68	Male	Black	Sargent	Johnson
2003	28OCT97	08SEP47	Male	Black	Sargent	Higgins
2004	20OCT97	10FEB63	Male	White	Para	Johnson
2005	31OCT97	14DEC67	Female	Black	Sargent	Johnson
2009	14NOV97	22FEB73	Female	Black	Sargent	Johnson
2011	16DEC97	16NOV68	Male	White	Sweet	Higgins

The Sounds Like operator.

SAS provides a 'soundex' key to use in a WHERE or HAVING clause to match against character variables, such as persons' names. To specify a 'Sounds Like' key the 'sounds like' character string is preceded by an asterisk as *(<character string>). The example below selects all the observations with DOCTOR names that 'sound like' "Richman":

```

create table study14 as
select *
from studya
where doctor =*'Richman';

```

Table STUDY14 - Doctors with names which 'sound like' "Richman"

PATIENT	DATEVIS	DATEBTH	SEX	RACE	DOCTOR	PRINCIPAL
2007	02NOV97	21JUN65	Male	White	Reichman	Johnson
2008	23OCT97	06JUL67	Male	White	Reichman	Davies

The Update Table Statement

Update is used to change the contents of existing observations, and is therefore useful for correcting erroneous data. Using the table STUDY11, created with the INSERT examples above, the date of visit of 22OCT97 for PATIENT 2002 is believed to be incorrect, so it is changed with the SET statement to the correct date of 22SEP97. Similarly one of the PATIENT 2099 observations is incorrect, the second PATIENT 2099 should have been input as 3000 (she is a different person from 2099!) so the correct number, 3000, is made to overwrite the incorrect one in the second update. In the WHERE clause a unique value for the observation must be specified where there are duplicate values, SEX=2 (female) is needed to specify which of the two PATIENT 2099 observations should be changed. If this SEX=2 was omitted *both* PATIENT 2099 observations would be changed to 3000. The date of visit and date of birth for PATIENT 3000 are now known, so the missing values of these dates are updated with the actual

values. The changed data is shown in bold type.

```
update study11
  set datevis=input('22SEP97',date7.)
    where patient=2002;
```

```
update study11
  set patient=3000,
    datevis=input('30OCT97',date7.),
    datebrth=input('07JAN62',date7.)
    where patient=2099 and sex=2;
```

Table STUDY11 - Making Corrections with UPDATE

OBS	PATIENT	DATEVIS	DATEBRTH	SEX	RACE	DOCTOR	PRINCIPAL
1	2001	14OCT97	14JUN41	Male	White	Para	Johnson
2	2002	22SEP97	31AUG62	Male	Black	Sargent	Johnson
3	2003	28OCT97	08SEP47	Male	Black	Sargent	Higgins
4	2004	20OCT97	10XEH53	Male	White	Para	Johnson
5	2005	31OCT97	14OEG57	Female	Black	Sargent	Johnson
6	2007	12NOV97	21JUN65	Male	White	Reichman	Johnson
7	2008	23OCT97	06JUL67	Male	White	Reichman	Davies
8	2009	14NOV97	22XES73	Female	Black	Sargent	Johnson
9	2010	11DEC97	05JUN70	Male	White	Beal	Johnson
10	2011	16DEC97	16NOV68	Male	White	Sweet	Higgins
11	2012	05NOV97	09JUL64	Male	White	Beal	Higgins
12	2014	05JAN98	01AUG61	Male	Hispanic	Hewitt	Davies
13	2015	13JAN98	25DEC59	Male	White	Hewitt	Johnson
14	2006	02FEB97	11APR70	Female	White	Para	Saxby
15	2099			Male	White	James	Saxby
16	3000	30OCT98	07JAN62	Female	White	Wilson	Saxby
17	2087	24NOV97	25JUL60	Male	White	Sargent	Saxby
18	2090	14OCT97	20NOV42	Male	White	Para	Saxby
19	2075	02MAR98	15JUL78	Female	Black	Jones	Davies

Options on the PROC SQL Statement

The following options on PROC SQL control execution:

ERRORSTOP (Default) | NOERRORSTOP specifies whether PROC SQL should stop executing when an error is incurred. The syntax of any following statements is still checked.

EXEC (Default) | NONEXEC specifies whether the SQL statements should be executed or just checked for correct syntax. This is like RUN CANCEL in a DATA step. When an error is incurred in a SQL statement NOEXEC is set until a QUIT is encountered.

FEEDBACK | NOFEEDBACK (Default) specifies whether more detailed information is written to the log file or not. Examples of such detailed information are the expansion of a * in a SELECT statement and details of view data transformations.

INOBS=n, OUTOBS=n, LOOPS=n, restricts the number of input observations, output observations, or internal SAS loops executed respectively to the value of n (default is infinity). These options are useful when testing SQL code on large data sets. The PROMPT option instructs PROC SQL to issue a prompt when any of these limits is reached. NOPROMPT (default) turns the prompting off.

PRINT | NOPRINT specifies whether the results of each subsequent SELECT query are written to the output file or not. When an output table or view is specified (e.g. with a CREATE statement) the default is NOPRINT. When no such output table or view is specified the default is PRINT.

STIMER (default) | NOSTIMER instructs PROC SQL to write timing information to the SAS log (useful for benchmarking). NOSTIMER turns off this information.

UNDO_POLICY= NONE (default) | OPTIONAL | REQUIRED specifies how to handle updates to tables when there is an interruption. Rollback is not currently supported.

The RESET statement resets all options back to their defaults, without restarting the procedure.

CONCLUSIONS

Exploiting the features and facilities of PROC SQL discussed in this paper sometimes provides a valuable shortcut around an otherwise difficult programming situation. Such features and facilities are useful for testing and debugging.

REFERENCES/ACKNOWLEDGMENTS

SAS Help Documentation
SAS PROC SQL manuals
The Consultant Resource Center of Trilogy Consulting Corporation
Papers 35 and 131 of the Proceedings of SUGI 23, Nashville TN, March 1998

SAS and SAS PROC SQL are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. TM indicates USA registration.

Timothy J. Harrington
Statistical Programmer
Trilogy Consulting Corporation
5148 Lovers Lane
Kalamazoo Michigan 49001

(616) 344 1996
1-800-323-7528
FAX (616) 344 1887

TJHARRIN@TRILOGYUSA.COM