

SAS/ACCESS® to External Databases: Wisdom for the Warehouse User

Judy Loren, Health Dialog Data Service, Inc., Portland, ME

ABSTRACT

With SAS/Access, SAS users can read from and write to almost any database product: DB2, Oracle, Informix, Sybase, MS SQL Server, or Teradata just to name a few. ODBC opens up even more warehouse doors.

SAS/Access offers several ways to connect: procs, such as Import and Export; the libname statement option that treats database tables like SAS datasets; and pass-through, which passes user-written SQL directly to the external product to execute and return the results to SAS. The fun starts when the warehouse tables are too large or too complex to allow the easy approach. This tutorial reviews all the techniques briefly, then focuses on the situations that call for advanced expertise. Examples demonstrate using SAS with DB2, Oracle and Microsoft Access. Important details like how to refer to missing values in various databases, and how to use macros in pass-through SQL, make the tutorial concrete and useful. Ever had to select records from a huge external table based on a set of key values in a SAS dataset? In case you missed it, here's some shortcut code.

This tutorial is for SAS users who need access to a non-SAS data warehouse, particularly if that warehouse is really large or complex. Knowledge of SQL is not necessary, but it will help as you follow the examples.

INTRODUCTION

The techniques you can use to read and/or write tables that reside in external databases include:

- 1) PROC IMPORT / PROC EXPORT
- 2) LIBNAME statement option
- 3) Pass-through SQL

This paper shows how to use each technique, and offers examples based on real world experience.

In all of the following examples, it is assumed that:

- 1) You have licensed and installed the SAS/Access software for the DBMS product you need to use. SAS/Access to DB2 is a different product from SAS/Access to Oracle, for example, and must be licensed separately.
- 2) SAS/Access is installed on the same platform as the external database or a client for that database. You can use SAS/Connect to submit your SAS code from one platform to another (let's say from your PC to a central Unix system on which DB2 resides) but you must have SAS on both platforms for that to work. You cannot use SAS on your PC to gain access to DB2 on another platform with having one of the following: SAS on the DB2 platform, or a DB2 client on your PC.
- 3) You have made friends with at least one DBA for the DBMS product at your site. S/he is an invaluable ally and a formidable foe. Get to know her. Bring him donuts, chocolate, flowers, coffee. Make her interests your interests, in other words, be sensitive to how your use of the database affects the DBAs and the other users. Always cooperate fully, and never, never, never do something s/he has asked you not to do.

PROC IMPORT/ PROC EXPORT

These PROCs are available in the base SAS product, but use of them is restricted to .csv, .txt. and delimited files. If you license the product SAS/Access Software for PC File Formats, these procs work on Microsoft Access files (.mdb) and Excel files (.xls), as well as dBase(.dbf), Lotus (.wkn) and generic database format (.dif) files.

IMPORT and EXPORT are very useful for communicating with non-SAS users. Here is one example, where I had to produce a list of member IDs and their associated model scores from a SAS dataset for a Microsoft Access user:

```
libname in 'f:\dsdw\PRA_Survey_Scores\SAS Data';
data prep;
    set in.score_pramaster(keep=mid_id
        mem_date_of_birth msr_date_completed
        pred_hicost pred_pra_cost cost_group);
```

```
PROC EXPORT DATA= prep
    OUTTABLE= "PRA Members"
    DBMS=ACCESS
    REPLACE;
    DATABASE="f:\Judy\PRA List\pra.mdb";
run;
```

Notes:

- 1) Using the KEEP= option on the DATA= dataset does not work, so I had to create a temporary dataset with just the desired variables to use as input to the EXPORT.
- 2) The REPLACE option must be included to overwrite an existing table. It is not necessary if you intend to create a new table in an existing database.
- 3) You can write old versions of Access. For example, you can specify DBMS=ACCESS97.
- 4) To control the type of field that MS-Access creates from each variable in the SAS dataset, associate formats with the variables. For example, a field formatted with a date format such as yymmdd10. will be converted to a MS-Access date-type field. Variables with a format of 4. will become integer-type MS-Access fields.

IMPORT works similarly, but in reverse.

```
PROC IMPORT OUT= second_level_facts
    DATATABLE= "Second Level Facts"
    DBMS=ACCESS2000 REPLACE;
    DATABASE="e:\Facts\Person_level.mdb";
run;
```

LIBNAME STATEMENT OPTION

This technique is easy to use. You simply put the option for the desired database product after the libref, then supply the information necessary to make the connection to the appropriate database within that product. Because these details vary so much from site to site, it is difficult to say what yours will look like, but here are some examples:

```
LIBNAME warehous DB2 SSID=query AUTHID=pref;
```

LIBNAME is the keyword introducing the statement. I've used an example libref suggesting that the user thinks of the database product as the warehouse. The shaded piece is the option that tells SAS this is not a conventional libref but rather a connection

to an external database product. Following this option come the customized pieces for your site.

In this example, the SSID refers to a DB2 subsystem that exists at the company. The AUTHID is the prefix you want the system to use with all the DBMS table names you subsequently reference. You might follow this example LIBNAME statement with a statement such as

```
Proc print data=warehouse.customer_list;
Where zip = '04021';
```

Because the AUTHID on the LIBNAME statement is PREF, SAS will tell DB2 it wants the rows from the table PREF.CUSTOMER_LIST in the subsystem QUERY.

In this first example, the security access has been set up in the background so that it is not necessary to specify the user and password in every access. The system matches the user who submits the code to a list of authorized users. If a user without the requisite security clearance attempts to execute this code, it will fail.

```
LIBNAME wh ORACLE user=jloren password=sugi
path=SOMEPATH.DIRECTORY.COM;
```

In the second example, the code contains the security information. This is convenient, but not particularly secure from the company's point of view. Anyone who can open the code document can execute it, and the security information is open to anyone who can read.

```
LIBNAME tdlb TERADATA USER=&tduserid
PASSWD=&tdpasswd SCHEMA='MODQIA';
```

In this third example, the userid and password are macro variables that are not stored in the code. This is a little more secure than the Oracle example. In this example, the log printed out the macro variable names, not the contents, which helped keep the information secure.

The LIBNAME reference to an external database allows the user to treat a table or a view in an external database as if it were a SAS dataset. You can do almost anything to it that you can do to a SAS dataset, with a couple of important exceptions:

- 1) If you want to SORT the table, you must use the OUT= option to name a SAS dataset where you want the sorted copy to reside. You cannot sort a DBMS table in place like you can a SAS dataset.
- 2) If you let the DBMS do the ordering, which happens when you use an ORDER BY clause in PROC SQL or a BY statement in a DATA step, it is important to understand how the DBMS treats missing values. In SAS, of course, missing values are low, but some DBMS's put missing values at the end of a sorted table.

The LIBNAME technique is very convenient. You can use it for writing to a database, as well as reading from it. If you want to load a table, you can just make it the object of a DATA statement or the OUT= option on a PROC such as SORT. Here is a simple example of creating a table in an external database from a SAS file residing on your disk:

```
LIBNAME warehouse DB2 SSID=query AUTHID=pref;
LIBNAME in 'c:\temp';
```

```
data warehouse.customer_local;
Set in.local_custs;
run;
```

In this case you are letting SAS and DB2 make all the decisions about what types of fields to create. The DB2 field names will be exactly the same as the SAS variable names. The field types will be based on the type (numeric or character) of SAS variable as well as some formatting information (the connection is smart enough to interpret SAS numeric fields with a date type format as SAS date variables and convert them to DB2 date fields).

This arrangement, where the DBMS takes the field names directly from the SAS dataset, maintains excellent control of your data. Situations in which you write your data to a flat file, which is in turn read by a program that loads the DBMS, are fraught with danger in that there is a point in the process where the connection between the data value and the field name it belongs to is broken.

When your DBMS is Teradata, there is an option called FASTLOAD= which can be set to YES. The DBA suggested we use it, and it did make a difference in the load time:

```
libname td teradata user=&tduserid"
password=&tdpasswd"
schema=&tduserid.";
data td.testone(fastload=yes);
set one;
run;
```

If you are planning to join your newly created table with an existing DBMS table, it may enhance performance to create a primary index on the join field(s).

```
libname td teradata user=&tduserid"
password=&tdpasswd"
schema=&tduserid.";
data td.testone(fastload=yes dbcreate_table_opts
= 'primary index(FIELD NAME OR NAMES HERE)');
set one;
run;
```

If this arrangement doesn't suit you or your DBA (and you should always cooperate fully with your DBA), you can take the next step of creating a table shell in the native DBMS language, then using PROC APPEND to insert the data. This technique retains the advantage of matching SAS variable names to DB2 field names, ensuring that you do not inadvertently load values to the wrong field as may happen if you write and then read a flat file.

You may have occasion to load the same table shell repeatedly with different sets of values. It is easy to empty a table before reloading. One way is to use PROC SQL:

```
LIBNAME warehouse DB2 SSID=query AUTHID=pref;

proc sql;
Delete from warehouse.customer_local;
run;
```

Most databases have table-level security that will not allow you to do this to tables you don't own.

A new feature of the LIBNAME access to an external DBMS is documented clearly in the SAS Online documentation (see Reference section for complete citation) SAS/ACCESS Software for Relational Databases: Reference, Part 2, in the chapter entitled SQL Procedure's Interaction with SAS/ACCESS Software, under Passing Joins to the DBMS:

Prior to Version 7 of the SAS System, an SQL query involving one or more DBMS tables or view descriptors was

processed by the SQL procedure as if the DBMS tables were individual SAS files. For view descriptors, the SQL procedure fetched all the rows from each DBMS table and then performed the join processing within SAS.

Although the SQL Procedure Pass-Through Facility has always passed joins to the DBMS, it is now possible to pass joins to the DBMS without using Pass-Through. Beginning in Version 7, the LIBNAME engine allows you to pass joins to the DBMS without using Pass-Through but with the same performance benefits. The DBMS server will perform the join and return only the results of the join to the SAS software. This will provide a major performance enhancement for many of your programs that perform joins across tables in a single DBMS. Both inner and outer joins are supported in this new enhancement.

In this example, two large DBMS tables, TABLE1 and TABLE2, have a column named DEPTNO. An inner join of these tables is performed where the DEPTNO value in TABLE1 is equal to the DEPTNO value in TABLE2. This join will be detected by the SQL Procedure and passed by the SAS/ACCESS engine directly to the DBMS server. The resulting rows will be passed back to the SAS System.

```
proc sql;
select tab1.deptno, dname from
    mydblib.table1 tab1,
    mydblib.table2 tab2
where tab1.deptno=tab2.deptno
    using libname mydblib oracle user=testuser
    password=testpass path=myserver;
```

If you want to perform a join between a large DBMS table and a relatively small SAS data file, you may want to specify the DBKEY= data set option. The DBKEY= data set option causes the SQL Procedure to pass a WHERE clause to the DBMS so that only the rows that match the WHERE condition are retrieved from the DBMS table. Also, if DEPTNO has an ORACLE index defined on it, using DBKEY= will greatly enhance the join's performance. In this example, the DBKEY= option causes only the rows that match DEPTNO to be retrieved. Without this option, the SQL Procedure would retrieve all the rows from TABLE1.

```
libname mydblib oracle user=testuser
password=testpass;
proc sql;
select tab1.deptno, loc from
    mydblib.table1 (dbkey=deptno) tab1,
    sasuser.sasds tab2
where tab1.deptno=tab2.deptno;
```

For more information on this data set option, see SAS/ACCESS Data Set Options.

In that portion of the documentation, we find:

The SQL statement that is created by the SAS/ACCESS engine and passed to the DBMS is similar to the following:

```
select deptno, loc
from bigtab.deptno
where deptno=:hostvariable;
```

The host-variable is substituted, one at a time, with DEPTNO values from the observations in the SAS data file KEYVALUES. The number of SELECT statements issued is equal to the number of rows in the data file. Therefore, for improved performance, the SAS data file should contain relatively fewer rows than the DBMS table to which it is being joined.

The advantage to this enhancement is greatly improved performance of queries written with strictly SAS syntax. It takes

advantage of DBMS indexes and allocates the work appropriately between the DBMS and SAS, keeping the data transfer between the two down to the minimum required for the job.

But it is not a panacea. Functions that do not translate well between SAS and the DBMS cannot be passed. For SAS datasets with large numbers of key values, this technique can wear out its welcome. In some cases, you need to talk directly to the DBMS, and for those there is the PROC SQL Pass-Through feature.

PROC SQL PASS-THROUGH ACCESS

The most powerful, and therefore most complex, method of accessing external databases is to specify exactly what you want the external database to do, in its own language. You have to learn the SQL features of that database, which are often different from SAS's implementation of SQL. If the tables are large, it is often necessary to learn about table join strategies as well. The return on this investment is vastly increased throughput, and the appreciation of all the other warehouse users.

SYNTAX

The components of the pass-through access are:

1) `proc sql;` All pass-through is done through `proc sql`.

2) The `CONNECT` statement, e.g.

```
connect to oracle (user=jloren password=sugi
path=somepath.directory.com);
```

3) A query to the automatic macro variable that contains any error messages from the target database:

```
%put &sqlxmsg;
```

This echoes to the log any error messages that might result from the attempt to connect.

4) The `SAS SELECT ...FROM` statement

```
create table matchmembers as
select datepart(birth_dt) as patdob
    format=yymmdd10.
    , first_name, last_name
from connection to oracle
```

The `FROM` clause always specifies `CONNECTION TO` the database. There is no semicolon yet.

5) The database SQL, enclosed in parentheses.

```
(select B.birth_dt
    , B.first_name
    , B.last_name
from dtmttbo.member B
    , dtemp.july_members A
where A.patdob=B.birth_dt
    AND A.first_name=B.first_name
    AND A.last_name=B.last_name
    AND birth_dt is not null
order by birth_dt, B.first_name, B.last_name
)
```

Note that this bit of code is passed to the database for execution. The only thing SAS does with it is macro substitution. The table names, field names, and SQL syntax must all be compatible with the external database, not with SAS.

6) Additional SAS SQL clauses that would normally follow the `FROM` clause, such as `ORDER BY` or `GROUP BY`. If you

put the ORDER BY inside the parentheses, the external database product does the sort before delivering the rows to SAS. If you put the ORDER BY outside the parentheses, SAS will do the sort after receiving all the rows. This difference is important to note for two reasons: 1) Performance – which place can the sort be accomplished most efficiently? 2) Platform coding systems. If your database resides on an EBCDIC platform and you are submitting the SAS statements from an ASCII platform (or vice versa), the ORDER BY will actually produce different results depending on where it is done.

- 7) The semicolon for the SAS SELECT statement.
- 8) Another query to the automatic macro variable that contains any error messages from the target database:


```
%put &sqlxmsg;
```
- 9) The DISCONNECT statement:


```
disconnect from oracle;
```
- 10) Another query to the automatic macro variable that contains any error messages from the target database:


```
%put &sqlxmsg;
```

Putting these pieces together, the code might look like this:

```
proc sql;
  connect to oracle (user=jloren
                    password=sugi
                    path=somepath.directory.com);
  %put &sqlxmsg;
  create table matchmembers as
  select  datepart(birth_dt) as patdob
          format=yymmdd10.
          , first_name
          , last_name
  from connection to oracle
  (select B.birth_dt
    , B.first_name
    , B.last_name
  from dtmttbo.member B,
    dtemp.July_members A
  where A.patdob=B.birth_dt
    AND A.first_name=B.first_name
    AND A.last_name=B.last_name
    AND birth_dt is not null
  order by birth_dt
    , B.first_name
    , B.last_name );
  %put &sqlxmsg;
  disconnect from oracle;
  %put &sqlxmsg;
quit;
```

If this code executes correctly, it will create a temporary SAS dataset called MATCHMEMBERS in the WORK directory containing 3 variables: PATDOB, FIRST_NAME and LAST_NAME. It matches two tables in Oracle (dtmttbo.member and dtemp.July_members) on 3 variables, and selects only those where the birth_dt is not missing. Note once again that the shaded part is passed to the database for execution. The unshaded part is executed by SAS.

VARIATIONS ON A THEME

To illustrate some of the features of the pass-through syntax, let's make a few changes to the code above and see what happens.

- 1) **SELECT ***
This is often used for convenience to select all the fields from a given table or join. If it is used in the SAS portion of the SQL above, as

```
create table matchmembers as
select  *
from connection to oracle
(select B.birth_dt
    , B.first_name
    , B.last_name .....
```

then all the fields returned by Oracle (in this case birth_dt, first_name and last_name) will be created as variables in the SAS dataset. Since Oracle date fields are full timestamps, the difference will be that birth_dt will not be renamed to patdob, and it will be full timestamp rather than the datepart we were asking for. Not a huge difference.

But if the SELECT * is used in the Oracle portion, as

```
create table matchmembers as
select  datepart(birth_dt) as patdob
          format=yymmdd10.
          , first_name
          , last_name
  from connection to oracle
  (select *
    from dtmttbo.member B,
    dtemp.July_members A
  where A.patdob=B.birth_dt
    AND .....
```

then you may see a degradation in the performance of the query, depending on how many useless variables result from the Oracle SQL. With this syntax, Oracle sends all the fields from both tables in the FROM clause to SAS. The translation into SAS is accomplished, then SAS selects only the three variables named in the SAS SELECT clause to write to the output table. Translating from an external data source into SAS is a relatively expensive undertaking; useless translation should be avoided whenever possible.

- 2) The SAS **datepart** function.
Oracle does not have a function named datepart. It would not work to re-write this query as

```
create table matchmembers as
select  *
from connection to oracle
(select datepart(B.birth_dt) as patdob
    , B.first_name
    , B.last_name
  from dtmttbo.member B,
    dtemp.July_members A .....
```

There is a way to accomplish the same thing, but it looks like

```
create table matchmembers as
select  *
from connection to oracle
(select trunc(B.birth_dt) as patdob
    , B.first_name
    , B.last_name
  from dtmttbo.member B,
    dtemp.July_members A .....
```

Note that this version does not associate the yymmdd10. format with the patdob variable, as is accomplished in the original code.

3) Macro substitution

SAS will resolve macro variables in the portion of the code that is passed through to the DBMS before the code is passed. Provided the value you want to use does not involve quotes, it is pretty straightforward.

```
%let threshold=100;
create table matchmembers as
  select *
    from connection to oracle
      (select trunc(B.birth_dt) as patdob
        , B.first_name
        , B.last_name
      from dtemp.July_members A
      where value > &threshold)
  ;
```

SAS would replace the &threshold with 100 and send the code to the DBMS to execute. Things get more complicated when the values you want to substitute are being compared with character variables or date fields. In many databases, including DB2 and Oracle, single quotes and double quotes are not interchangeable. Dates and character values must be enclosed in single quotes, and as you know, SAS does not resolve macro variables within single quotes. There are many potential solutions to this problem once you are aware of it. For example, you could include the single quotes as part of the macro variable value:

```
%let early_dt='2001-01-31';
```

Or you could specify a macro variable with the value of a single quote:

```
%let q='';
%let early_dt=2001-01-31;

create table matchmembers as
  select *
    from connection to oracle
      (select trunc(B.birth_dt) as patdob
        , B.first_name
        , B.last_name
      from dtemp.July_members A
      where patdob > &q&early_dt&q)
  ;
```

The &q&early_dt&q is replaced with '2001-01-31' before the SQL is passed to the DBMS to execute.

4) Macro variable creation

The SQL Procedure has a great feature that allows you to create a macro variable and store a value in it, much like CALL SYMPUT in the DATA step. It can be used fruitfully with an external DBMS. In the following case, I wanted to write a new row to an existing DBMS table, with a value for UDD that was 1 greater than the maximum existing value.

```
proc sql;
connect to db2 (ssid=query);
select maxudd into :maxudd
from connection to DB2
(select max(udd) as maxudd
 from dss.udd_table);
```

At this point I have a macro variable called MAXUDD which I use in a DATA step to create single observation that can be PROC APPENDED to the existing DBMS table UDD_TABLE.

Note that if I had used the LIBNAME method of accessing the database, I could have run the following code and gotten similar performance and identical results:

```
libname udd db2 ssid=query authid=dss;
proc sql;
select max(udd) into :maxudd
from dss.udd_table;
```

SAS does pass standard functions such as MAX through to the DBMS even when using the LIBNAME reference. You can see this by invoking the following system options:

```
options SASTRACE= ',,,d' SASTRACELOC=SASLOG;
```

This option causes information about the query as passed to the DBMS to be printed to the SAS log, so you can monitor where the work is being done.

5) Missing (NULL) values

SAS uses a period to represent missing numeric values, and a blank for missing character values. So, for example, to find records where the first name is missing you can code

```
where first_name = ' '
```

Further, missing values are the lowest possible SAS values, so you can code

```
where sasvar > .
```

to select all non-missing numeric values. Other DBMS products do not share these conventions. When you want to refer to missing values in passed-through SQL, always use IS NULL or IS NOT NULL, e.g.

```
where first_name IS NULL
```

```
where dbvar IS NOT NULL
```

THE DBPASS MACRO

This macro, originally written by Tom Finn, was created before the DBKEY option existed for linking SAS datasets with DBMS tables (see above for more detail on the DBKEY option). It can still come in handy for the times when you have a large number of values in a SAS dataset and you want to access records in a DBMS table that match those values. It is different from the DBKEY= option as described above and in the SAS documentation in that:

1) It passes a large number of values at a time in a WHERE clause such as:

```
select *
from DBMS table
where dbms.keyfield in (value1, value2,...)
```

2) It can only join on one field.

3) It makes use of PROC SQL pass-through access.

4) The only variable from the SAS dataset containing the key values that can be saved on the output dataset is the key variable itself. Others must be joined back on in a later merge.

If the field you are joining on contains an index in the database, the DBMS will use the index to improve performance.

The DBPASS macro described in this paper allows you to break

up your list of keys into sets of any size. It then executes multiple queries, putting the number of key values you specified into each WHERE clause and looping until all the observations in the SAS dataset have been passed. This is where you would get performance improvement over the DBKEY= method, which creates a separate query for each individual key value you need to match.

The actual macro code for DBPASS is shown in the Appendix to this paper. Before use, you should edit it to connect to the appropriate DBMS for your site. You may want to change some of the parameters, depending on the connection values needed for your site. These are the parameters the macro expects as currently written:

INCOMING DATASET (IDS=)

You can use a two-level name to point to a stored dataset, or you can read in a WORK dataset coming from a previous DATA or PROC step.

OUTPUT DATASET (ODS=)

Specify where you want the output from the join to be stored.

HOW MANY KEYS IN ONE WHERE CLAUSE (N=)

You can experiment with the number to get best results. The limit depends on the number of bytes in your key values and whether they are character (requiring a set of quotes per value) or numeric.

THE NAME OF THE SAS KEY VARIABLE (SASVAR=)

DBPASS will take the values of this variable and put them into the WHERE clause. No other variables from the incoming dataset can be carried through to the output dataset.

THE TYPE OF THE SAS KEY VARIABLE (TYPE=)

CHAR(acter) or NUM(eric). (Actually the macro just looks for CHAR; if the value of TYPE is not CHAR, it assumes numeric.)

THE FORMAT OF THE SAS KEY VARIABLE (VARFMT=)

The format only matters if the key value is numeric. It uses the format to put the numeric values into the WHERE clause. Character values are put in default format with single quotes around them.

THE NAME OF THE DBMS KEY FIELD (DBVAR=)

This is the field DBPASS uses in the WHERE clause to compare to the SAS key values. Note that if this field name appears in more than one table in your FROM list, you must specify a prefix (table name or alias) to decide which table's field will be used in the WHERE clause. You can join multiple DBMS tables in the WHERE parameter of the macro (described below).

THE DB2 SUBSYSTEM YOUR TABLE RESIDES IN (SSID=)

This parameter is included for those who are using DB2. This value is supplied in the CONNECT TO DB2 statement. You can alter the macro to accept and use the appropriate connection parameters for your site.

YOUR SELECT STATEMENT (SELECT=%STR())

Note that the list of fields you want to extract from the DBMS should be enclosed in a %STR() function. Note also that this is the DBMS select, so should contain field names as they appear in the DBMS.

YOUR FROM STATEMENT (FROM=%STR())

Here's where you identify the fully qualified DBMS table name(s) containing the fields you want to extract. When you have more than one table in this list, aliases help in identifying the field sources.

RENAMING THE DBMS FIELDS FOR SAS (AS=%STR())

This parameter allows you to rename the DBMS fields as they come into the resulting SAS dataset (ODS, above). It operates positionally with the SELECT parameter; the first field in the SELECT parameter receives the first variable name in the AS list,

the second field receives the second variable name, etc. It is not necessary to rename fields if you don't want to.

ADDITIONAL WHERE RESTRICTIONS (WHERE=%STR())

The match of the SAS key variable with one DBMS field is taken care of for you by the macro. The WHERE parameter is for specifying additional restrictions on one table, or join criteria for other tables included in the FROM parameter.

RESTARTING (FO=)

If the macro fails in the middle, the results of completed queries remain saved to the output dataset. You can use FO= (stands for FIRSTOBS=) to tell DBPASS where to start (which observation to start with) in the incoming dataset.

APPENDING RESULTS TO AN EXISTING DATASET (FIRSTACT=)

Independently of restarting, you decide whether you want to create a new dataset with this execution of DBPASS or append to an existing dataset. FIRSTACT=CREATE will cause the FIRST query to use CREATE TABLE ... AS; all subsequent queries generated by that execution of DBPASS will use INSERT INTO. If you specify FIRSTACT=INSERT, even the first query will use INSERT INTO.

CONTROLLING THE NUMBER OF LOOPS (NLOOPS=)

You can limit the number of queries DBPASS initiates by coding a number in this parameter. This would be used in much the same circumstances as an OPTIONS OBS=. You would probably then be interested in the TRACK parameter below.

KEEPING TRACK OF WHERE YOU ARE (TRACK=)

Here you specify the name of a dataset that will store the observation number you should start with if you want to complete the list of keys after either a failure part way through the list or a stop caused by hitting the NLOOPS limit you specified.

EXAMPLE OF THE USE OF DBPASS

To illustrate the use of the parameters, suppose we have a list of customer IDs in a SAS dataset and wish to retrieve name, address and phone numbers for each customer from a DBMS. Our list of desired ID numbers resides in a SAS dataset called WANTED with a variable name of CID.

Using DBPASS, we would code:

```
libname mac 'c:\mydir\mymacros';
options sasautos=(mac) nomprint;
%DBPASS(IDS=wanted
,ODS=addr
,N=1000
,SASVAR=cid
,DBVAR=a.cust_id
,SELECT=%str(a.cust_id, city, state,
addr_line1, addr_line2, zip, name, phone)
,AS=%str(cid, city, state, addr1,
addr2, zip, name, phone)
,FROM=%str(pref.cust_addr A
,pref.cust_phone B
,pref.cust_name C)
,WHERE=%str(a.cust_id=b.cust_id and
a.cust_id=c.cust_id)
,SSID=query
);
```

Note that some parameters are not required at all unless you want to take advantage of a particular feature. Other parameters can be allowed to default, such as the TYPE=CHAR, if the default suits your application.

DEFAULTS

You can establish the DBPASS macro at your site with whatever defaults you find most useful. Although it is not shown this way in the Appendix, you can even use default values for DBVAR,

FROM, SASVAR, ODS, etc.

SUGGESTIONS FOR USE

Because of the volume of log generated, it is best to specify the NOMPRINT option. To further limit the lines of feedback, also use NOFULLSTATS and NONOTES.

The values of the FROM, WHERE, SELECT, and AS parameters should be enclosed in the %STR() function. For more complex strings, subqueries for example, you may have to resort to %NRBQUOTE() (no rescan blind quote) or another macro.

HOW TO CHOOSE A METHOD

The main thing to keep in mind when using an external database such as DB2 or Oracle is that you should exercise control over where and how the processing is being done. Your DBA and the other warehouse users will appreciate your thoughtfulness.

One way to do this is to make use of the option described above that prints to the log the query as SAS sent it to the DBMS.

```
options SASTRACE= ' , , , d ' SASTRACELOC=SASLOG;
```

Study this log, and share it with your DBA as necessary when your queries are taking significant time to execute. Often there are better ways to approach joins of multiple large tables to reduce both your wait time and the consumption of system resources.

CONCLUSION

You have several choices when it comes to reading and/or writing tables in external databases from SAS. To make good decisions, understand the structure and volume of the database and of your target tables. Confer with your local DBA to use methods that not only enable you to get your work done, but also interfere minimally with others using the database.

REFERENCES

SAS OnlineDoc®, Version 8, February 2000, Copyright ©2000, SAS Institute Inc..

Loren and Shoemaker, "Retrieving DATA from Large DB2 Tables Based on Keys in SAS: The Sequel", NESUG 1995 Proceedings.

ACKNOWLEDGMENTS

Tom Finn was the original creator and author of the DBPASS macro.

I am indebted to many DBAs and SAS users, some of whom are SAS-L contributors, for insight, hints and corroborating information, including but not limited to Sigurd Hermansen, Don Stanley, Charles Wentzel, Craig Dickstein, Burak Sezen, Gokhan Cakmakci, Ron Goodling, and Rod Deane.

CONTACT INFORMATION

If you have comments or questions about this paper, please get in touch with:

Judy Loren
Health Dialog Data Service, Inc.
39 Forest Avenue
Portland, ME 04101
(207) 822-3708 (W)
JLoren@HealthDialog.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.

SEE FOLLOWING APPENDIX FOR DBPASS
MACRO CODE. THIS CODE IS MADE
AVAILABLE AS IS, WITH NO WARRANTIES
EXPRESSED OR IMPLIED. USE IT AS YOU LIKE
AT YOUR OWN RISK.

Appendix A
DBPASS code

```
%MACRO DBPASS(IDS=WANTED
    ,N=400
    ,SASVAR=
    ,TYPE=CHAR
    ,VARFMT=BEST.
    ,ODS=
    ,SSID=QUERY
    ,SELECT=%STR( )
    ,FROM=
    ,AS=%STR( )
    ,WHERE=
    ,WHEREMAC=
    ,TRACK=
    ,FO=1
    ,FIRSTACT=CREATE
    ,NLOOPS=
    ,INOBS=
    ,DBVAR=);
%* 9409 TCF WHEREMAC AND INOBS
%* 9408 TCF ADDED TRACK,FO,FIRSTACT AS MACRO PARMS;
%* 9408 TCF FIXED ABORT;
%* 9709 CDJ - ADDED FOR FETCH ONLY
%*          ADDED DISCONNECT FROM DB2
%*          DUE TO CONTENTION PROBLEMS
%LET TYPE=%UPCASE(&TYPE);
%LOCAL NOBS;
DATA _NULL_;
    IF _N_=0 THEN SET &IDS NOBS=NOBS;
    PUT NOBS=;
    CALL SYMPUT('NOBS',LEFT(PUT(NOBS,BEST.)));
    STOP;RUN;
%LOCAL I N STOPRC;
%LET STOPRC=0;
%IF &FO NE 1 %THEN %PUT ***** RESTARTING *****;
%IF &NOBS EQ 0 %THEN %DO;
    %PUT FO=&FO NOBS=&NOBS;
    PROC SQL
    %IF %LENGTH(&INOBS) GT 0 %THEN INOBS=&INOBS;
    ;
    CONNECT TO DB2(SSID=&SSID);
    CREATE TABLE &ODS AS
    SELECT * FROM CONNECTION TO DB2(
    SELECT &SELECT
    FROM &FROM
    %IF &TYPE EQ CHAR %THEN WHERE &DBVAR = ' ';
    %ELSE WHERE &DBVAR = . ;
    %IF %LENGTH(&WHERE) GT 0 %THEN AND &WHERE ;
    %IF %LENGTH(&WHEREMAC) GT 0 %THEN AND %&WHEREMAC ;
    FOR FETCH ONLY
    ) AS A(&AS);
    %PUT &SQLXRC &SQLXMSG;
    DISCONNECT FROM DB2;
    DATA &ODS; SET &ODS; STOP;
%END;
%ELSE %DO %WHILE(&FO LE &NOBS AND &STOPRC EQ 0 AND X&NLOOPS NE X0 );
    %PUT FO=&FO NOBS=&NOBS;
    %IF %LENGTH(&NLOOPS) GT 0 %THEN %LET NLOOPS=%EVAL(&NLOOPS-1);
    %IF %LENGTH(&TRACK) GT 0 %THEN %DO;
        DATA &TRACK(KEEP=FO); FO=&FO; IF _N_=1 THEN OUTPUT;
```

```

%END;
%ELSE %DO;
  DATA _NULL_;
%END;
SET &IDS(KEEP=&SASVAR FIRSTOBS=&FO) END=END;
LENGTH NN $7.; NN=LEFT(PUT(_N_, BEST.));
%IF &TYPE EQ CHAR %THEN %DO;
  CALL SYMPUT('V' || NN, '"' || &SASVAR || '"');
%END;
%ELSE %DO;
  CALL SYMPUT('V' || NN, LEFT(PUT(&SASVAR, &VARFMT)));
%END;
IF END OR _N_ GE &N THEN DO;
  CALL SYMPUT('N', NN);
  STOP;
  END;
RUN;
PROC SQL
%IF %LENGTH(&INOBS) GT 0 %THEN INOBS=&INOBS;
;
CONNECT TO DB2(SSID=&SSID);
%IF &FIRSTACT EQ CREATE %THEN CREATE TABLE &ODS AS ;
%ELSE INSERT INTO &ODS ;
%LET FIRSTACT=INSERT;
SELECT * FROM CONNECTION TO DB2(
  SELECT &SELECT
    FROM &FROM
    WHERE &DBVAR IN (&V1 %DO I = 2 %TO &N; , &&V&I %END; )
%IF %LENGTH(&WHERE) GT 0 %THEN AND &WHERE ;
%IF %LENGTH(&WHEREMAC) GT 0 %THEN AND %&WHEREMAC ;
FOR FETCH ONLY
) AS A(&AS);
%PUT &SQLXRC &SQLXMSG;
%IF &SQLXRC NE 0 %THEN %LET STOPRC=&SQLXRC;
DISCONNECT FROM DB2;
%LET FO=%EVAL(&FO+&N);
%END;
%IF X&NLOOPS EQ X0 AND %LENGTH(&TRACK) GT 0 %THEN %DO;
  DATA &TRACK(KEEP=FO); FO=&FO; OUTPUT;STOP;
%END;
%IF &STOPRC NE 0 %THEN %DO;
  DATA _NULL_;
  %IF &STOPRC GT 0 %THEN ABORT &STOPRC ;
  %ELSE ABORT %EVAL(-1*&STOPRC);
%END;
%MEND;

```