Paper 155-27

Can't Relate? A Primer on Using SAS® With Your Relational Database

Garth W. Helf, IBM Corporation, San Jose, CA

ABSTRACT

SAS Software excels at analyzing huge amounts of data. Relational database software (commonly called a Database Management System, or DBMS), such as DB2, Oracle, or Teradata, excels at storing and managing huge amounts of data. This paper, intended for beginning to intermediate SAS users, describes several tools available in the SAS/ACCESS® Software for Relational Databases product for moving data between SAS and your DBMS. First, we explore some ways to submit queries to your DBMS and return the result to a SAS data set, including the SQL Procedure Pass-Through Facility and the new SAS/ACCESS LIBNAME Statement that lets you associate DBMS tables to SAS libraries, then retrieve data from your DBMS transparently from any DATA step or procedure. Second, we look at two efficient ways to join a SAS data set with tables in a DBMS: the data set options DBINDEX= and DBKEY= in the SAS/ACCESS LIBNAME Statement, and the %DBMSlist macro supplied here, which passes SAS data set values in groups to the SQL Procedure Pass-Through Facility. Finally, we look at some ways to create DBMS tables and insert data into them from SAS data sets, including the SQL Procedure Pass-Thru Facility, DATA step, APPEND procedure, and DBLOAD procedure.

INTRODUCTION

Relational database software, such as DB2, Oracle, and Teradata, is commonly used to store huge amounts of data in organizations like businesses, universities, and government agencies. The SAS/ACCESS Software for Relational Databsaes product provides several tools to let SAS software interact with data in DBMS databases. This paper covers three important tasks: getting DBMS data into SAS, joining SAS and DBMS tables efficiently, and getting data from SAS into DBMS tables.

GETTING DBMS DATA INTO SAS

SQL PROCEDURE PASS-THROUGH FACILITY

The SQL Procedure Pass-Through Facility is a special form of the normal SQL Procedure. The SQL Procedure was originally created to let you manipulate SAS data sets with standard SQL syntax rather than the SAS Data Step and SAS Procedures. For example, if you want to subset a large SAS data set based on a few serial numbers, you could submit this SQL Procedure step:

```
proc sql;
 create table History as
 select * from MfgHist where sn in
   ('CGN213','SDA980','WEF765');
 quit;
```

The SQL Procedure Pass-Through Facility was an enhancement to the SQL procedure starting in Version 6 that lets you send DBMS-specific SQL statements to a DBMS server and retrieve the data directly to SAS. Here's how the previous example would look if the data is in a DB2 table instead of a SAS data set:

```
proc sql;
 connect to db2 (dsn=tvdb uid=helf pwd=mypw);
 create table History as select * from
     connection to db2 (
   select * from MfgHist where sn in
     ('CGN213','SDA980','WEF765')
   for fetch only);
 disconnect from db2;
 quit;
```

There are some additional statements in the SQL Procedure Pass-Through query compared to standard PROC SQL step:

- A *Connect* statement that tells the DBMS which database to connect to and what user ID and password to use. You must also specify the SAS/ACCESS Engine name SAS uses to identify your DBMS. In this case it is "DB2", for Oracle it "Oracle". For other DBMS vendors, you can find the correct name in the SAS/ACCESS documentation, listed under "References" at the end of the paper.
- A *Disconnect* statement that terminates the connection to the DBMS.
- A *Connection to* component of the *Select* statement that tells SAS that you want to pass the query to a DBMS rather than process it locally in SAS.

This is called a Pass-Through query because the entire query within the outermost parentheses in the *Create Table* statement is passed through to the DBMS server unaltered, except for resolution of any macro variables that might be present. This means that the syntax of the SQL statement must conform to the requirements of the DBMS software, not SAS Proc SQL. For example, a date constant must be specified as '2002-01-09' (since this is DB2), while in a regular Proc SQL query used with a SAS dataset you would specify a date constant as '9jan2002'd.

Pass-through queries can take advantage of indexes on DBMS columns to process a query quickly and efficiently. In our example, If the column SN in DB2 table MfgHist has an index, this step should run in a few seconds or less, even though there might be 5 million rows in the table.

Another nice feature of the Pass-Through Facility is that it is easy to run an SQL statement for your DBMS that was written in another application besides SAS. Since SQL statements are passed through unaltered to the DBMS server, you simply paste the query from another source into a Pass-Through Facility step like this and you can run the query quickly.

SAS/ACCESS LIBNAME STATEMENT

Starting with Version 7, SAS provides a new way to access DBMS tables through a feature called the SAS/ACCESS LIBNAME statement. This extension to the LIBNAME statement allows you to assign a SAS libref directly to a DBMS database. For example, if we have a DB2 database called TVDB, a SAS/ACCESS LIBNAME statement would look like:

```
libname mydb db2 dsn=tvdb uid=helf pwd=mypw;
```

After this LIBNAME statement is issued, you can access DB2 tables in PROC and DATA steps transparently by refering to data set mydb.*DB2_table_name*. Several examples of this are given later in this paper.

The SAS/ACCESS LIBNAME statement has several options that affect how SAS interacts with the DBMS. Some of the options are described in this paper. For information about all of the options available, refer to the SAS/ACCESS documentation in the References section at the end of this paper.

SASTRACE: A GREAT OPTION FOR DEBUGGING SAS/ACCESS LIBNAME STATEMENTS

There is a great system option called SASTRACE that displays detailed information about the commands that SAS/ACCESS sends to your DBMS. This option is incredibly useful when you are debugging a program that is using a SAS/ACCESS Libname statement. The syntax for this option is:

```
options sastrace=',,,d' sastraceloc=saslog;
```

The SASTRACE option turns on detailed DBMS messages, and the SASTRACELOC option tells SAS where to write the messages, in this case the SAS log. For example, when you submit a LIBNAME statement to assign a libref to a DB2 database, you will see the following messages in the SAS log:

```
TRACE: Successful connection made, connection id
     0 0 1296057922 no_name 0 Submit
TRACE: Database/data source: tfdiskdb 1
     1296057922 no_name 0 Submit
TRACE: USER=HELF, PASS=XXXXXXX 2 1296057922
     no_name 0 Submit
TRACE: AUTOCOMMIT is NO for connection 0 3
     1296057922 no_name 0 Submit
NOTE: Libref DB2SYS was successfully assigned as
     follows:
       Engine:        DB2
       Physical Name: tfdiskdb
```

Each line in the log that starts with TRACE: is a message about how SAS interacts with the DBMS. Further examples of trace output is shown in later sections. This trace information in the log is very useful when you need to talk with SAS Technical Support, and when you want to talk to your database administrator. The administrator of your Oracle or DB2 database probably knows nothing about SAS, but the SASTRACE information in the log will be quite helpful for him or her to debug a database problem.

Hint: I could not find this anywhere in the documentation, but to turn off tracing, submit the following statement:

```
options sastrace=',,,';
```

TRANSPARENT DBMS ACCESS WITH THE SAS/ACCESS LIBNAME STATEMENT

After a SAS/ACCESS LIBNAME statement is issued, you can access DBMS tables in PROC and DATA steps transparently by refering to data set *libref.DBMS_table_name*. Here are several examples, along with the SQL statement created by SAS.

Example 1 - DATA Step With Subsetting WHERE Statement: A DBMS table is specified in a SET statement. The KEEP= data set option identifies the columns the DBMS should return to SAS. Note that a KEEP statement would work differently - since it applies to the output data set, SAS would bring data for all columns from the DBMS into SAS, then save only the columns named in the KEEP statement in the output data set. If your DBMS table has 200 columns and you only want 5 in the output data set, a DATA step with the KEEP= data set option should run faster than one with a KEEP statement. Note that the subsetting WHERE statement is passed on to the DBMS, so only the rows in the DBMS table that match our specification for column LOT will be returned to SAS. ThIs DATA step creates a SAS data set called MfgData from the data returned from the DBMS.

```
data MfgData;
 set db2sys.scor_stat_history
  (keep=lot oper trdate trtime qty);
 where lot like 'DA0025104%';
 run;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  "TRDATE",
     "TRTIME", "LOT", "OPER", "QTY"  FROM
     db2sys.SCOR_STAT_HISTORY  WHERE  ( "LOT"
     LIKE  'DA0025104%' ) FOR READ ONLY
```

Example 2 - DATA Step With Subsetting IF Statement: This example is similar to Example 1, except that a Subsetting IF statement is used instead of a subsetting WHERE statement. Note that no WHERE clause is generated in the SQL statement sent to the DBMS. Instead, SAS has asked for all rows in the DBMS table to be returned to the DATA step, at which point the subsetting IF statement will be applied. This is no problem if the DBMS table is small, but if the DBMS table contains 10 million rows, this DATA step will take much longer to run than the DATA step in Example 1.

```
data MfgData;
 set db2sys.scor_stat_history
  (keep=lot oper trdate trtime qty);
 if lot in ('DA00251044','DA00251045');
 run;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  "TRDATE",
     "TRTIME", "LOT", "OPER", "QTY"  FROM
     db2sys.SCOR_STAT_HISTORY  FOR READ ONLY
```

Example 3 - SQL Procedure With Simple Query: We saw in a previous section that you can get DBMS data into SAS with the SQL Procedure Pass-Through Facility. With the SAS/ACCESS LIBNAME statement, you can also get DBMS data into SAS with standard SQL Procedure syntax. This example is similar to Example 1. We are creating a SAS data set called MfgHist from a DBMS table. The DBMS table is named in the SQL FROM clause, the columns are specified in the SELECT clause, and the row subsetting conditions are identified in the WHERE clause.

```
proc sql;
 create table MfgData as
 select lot, oper, trdate, trtime, qty
 from db2sys.scor_stat_history
 where lot in ('DA00251043','DA00251044');
 quit;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  "LOT", "OPER",
     "TRDATE", "TRTIME", "QTY"  FROM
     db2sys.SCOR_STAT_HISTORY  WHERE  ( ( "LOT"
     IN  ( 'DA00251043' , 'DA00251044' ) ) ) FOR
     READ ONLY
```

Example 4 - SQL Procedure With Aggregate Query: You can write much more complicated queries in SQL than the simple query in Example 3. Here is an example of an aggregate query - one that summarizes one or more variables for distinct groups of other variables. Note that the SAS/ACCESS engine is able to translate this query into an aggregate query that is passed on to the DBMS server. If the DBMS table contains 1 million rows, but only 100 distinct combinations of TRDATE and PRODUCT, the DBMS server will return only the 100 summarized rows to SAS, not all 1 million rows of actual data in the table. Essentially, we are moving the task of aggregration to the DBMS server, which results in less work for SAS and the network, if the DBMS server is connected remotely. This is an example of what SAS/ACCESS calls “SQL Implict Passthru” (not to be confused with the SQL Procedure Pass-Through Facility described earlier), and is discussed more in the next section.

```
proc sql;
 create table MfgData as
 select trdate, product, count(*) as num_lots
 from db2sys.scor_stat_history
 group by trdate, product;
 quit;
```

```
TRACE: SQL stmt prepared on statement 1,
     connection 0 is:   select
     db2sys.scor_stat_history."TRDATE",
     db2sys.scor_stat_history."PRODUCT", COUNT(*)
     as num_lots from db2sys.SCOR_STAT_HISTORY
     group by db2sys.SCOR_STAT_HISTORY."TRDATE",
     db2sys.SCOR_STAT_HISTORY."PRODUCT" FOR READ
     ONLY
TRACE: DESCRIBE on statement 1, connection 0. 10
     1326653673 no_name 0 SQL
SQL Implicit Passthru stmt prepared is:  11
     1326653673 no_name 0 SQL
  select db2sys.scor_stat_history."TRDATE",
     db2sys.scor_stat_history."PRODUCT", COUNT(*)
     as num_lots from db2sys.SCOR_STAT_HISTORY
     group by db2sys.SCOR_STAT_HISTORY."TRDATE",
     db2sys.SCOR_STAT_HISTORY."PRODUCT"
```

Example 5 - Other Procedures: Since SAS/ACCESS LIBNAME technology allows transparent access to DBMS data, it's quite simple to use DBMS tables wherever input or output data sets are required in SAS procedures. Here is an example that uses the PRINT procedure on a DBMS table. Note that the column selections in the VAR statement and the row subsetting conditions in the WHERE statement are passed to the DBMS.

```
proc print data=db2sys.disc_stat_history
noobs;
 title 'PROC PRINT With DBMS Data';
 var lot oper trdate trtime tcmd qty;
 where lot in ('DA00251043','DA00251044');
 run;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is: SELECT  "LOT", "OPER",
     "TRDATE", "TRTIME", "TCMD", "QTY"  FROM
     db2sys.DISC_STAT_HISTORY  WHERE  ( ( "LOT"
     IN  ( 'DA00251043' ,'DA00251044' ) ) ) FOR
     READ ONLY
```

Here is the output from the procedure:

```
PROC PRINT With DBMS Data

LOT          OPER   TRDATE      TRTIME    TCMD   QTY

DA00251043   2000   08JAN2002   12:19:03  STRT    25
DA00251043   2400   08JAN2002   12:19:05  MOVE    25
DA00251044   2000   08JAN2002   12:19:27  STRT    25
DA00251044   2000   08JAN2002   12:19:29  LCOM    25
DA00251044   2400   08JAN2002   12:19:29  MOVE    25
```

Two other handy procedures are DATASETS, to find out what tables are in a DBMS database, and CONTENTS, to see the column definitions for a DBMS table.

PERFORMANCE CONSIDERATIONS FOR SAS/ACCESS LIBNAME QUERIES

DBMS queries are sometimes the most frustrating part of a SAS program. You might think that a simple query should run fast, only to find out that it takes many minutes. There are several factors that affect the performance of SAS/ACCESS LIBNAME queries.

First, transfer only data you need from the DBMS to SAS. DBMS tables can often have tens of millions of rows and hundreds of columns. Select only the rows you need with a WHERE clause in the SQL procedure or a WHERE statement in procedures or DATA steps. Select only the columns you need with a SELECT clause in the SQL procedure, a KEEP= data set option in the DATA step or procedures, or a VAR statement in some procedures.

Second, be aware of the SQL request SAS sends to the DBMS. You want to make sure that SAS passes as much SQL processing as possible to the DBMS, rather than returning more rows than needed for processing in SAS. SAS/ACCESS provides two SQL query optimization features to do this: the WHERE Clause Optimizer and SQL Implicit Passthru. Levine [2001] presented a paper at SUGI 26 that covers both of these in detail.

The Where Clause optimizer looks at WHERE statements in SAS procedures and DATA steps and generates an SQL statement appropriate for the target DBMS. Examples 1, 3, and 5 in the last section show how the Where Clause Optimizer created WHERE clauses in the DBMS queries based on WHERE statements and WHERE clauses in the SAS code.

The SQL Implicit Passthru feature transparently converts certain SQL procedure queries into DBMS-specific statements, then passes the converted query to the DBMS for processing. In SAS 8.2, a query can be converted if it contains one or more of the following: the DISTINCT keyword, inner or outer joins, aggregate functions, or unions. Example 4 in the last section showed how SAS passed the COUNT aggregation function to the DBMS. You can see that Implicit Passthru was used by the note put into the log by SASTRACE.

Finally, have some understanding of how the SQL optimizer for your DBMS works. This applies to queries written with the SQL procedure Pass-Thru Facility as well as SAS/ACCESS LIBNAME queries. Each DBMS vendor has an SQL optimizer that looks at an SQL statement and chooses the best way to process it. Sometimes a small change in an SQL query can have a huge improvement in query performance. For example, these two WHERE clauses behave much differently in the DB2 SQL optimizer:

```
where lot like 'DA0025104%';       ** Good **;
where substr(lot,1,9)='DA0025104'; ** Bad **;
```

A query with the first WHERE clause will run very fast if there is an index on column LOT because the SQL optimizer selects an execution path that uses an index scan. A query with the second WHERE clause will always run slow becuase the SQL optimizer chooses a table scan when the substring function is used. The SQL optimizer is typically very comlex, so it's best to take your SQL query to the DBMS administrator and ask them if it can be improved. Many DBMS vendors supply a tool, like Visual Explain that comes with DB2, that analyzes a query and looks for areas of poor performance.

PUT DBMS CONNECTION INFORMATION IN AUTOEXEC FILE

In the previous examples, the DBMS connection information was specified explicitly in the SQL Procedure Pass-Through Facility Connect statement or in the SAS/ACCESS LIBNAME statement. In real practice, you probably don't want to specify your user ID and password in your SAS programs, for maintenance and security reasons. It is a maintenance headache if your organization requires you to change passwords regularly and you have to go find and modify all of your SAS programs that access a DBMS you just changed the password to. It can be a security concern if you share your programs that contain your user ID and password with other people. You might have higher privileges on the DBMS than they do. Or, if like me you have user IDs on dozens of servers and web sites and you try to keep your password the same on many of them, you probably don't want other people to see the user ID and password for your DBMS.

An easy way to eliminate these maintenance and security concerns is to put your DBMS connection information only in your SAS autoexec file. The SAS autoexec file contains SAS statements that are executed immediately after the SAS System initializes and before any user input is accepted. The SAS autoexec file is a convenient way to automatically execute a set of standard SAS statements like OPTIONS, FILENAME, and macro variable assignments each time you start the SAS system. In some operating systems, like Windows, SAS looks for a file called autoexec.sas in certain directories. In all operating systems, you can specify the name and location of the file with the AUTOEXEC system option when you invoke SAS, in which case the file does not need to be named autoexec.sas. For more information about how the SAS autoexec file works, refer to the *SAS Companion* manual for your operating system.

In my SAS autoexec file, I assign the connection information for each DBMS database I use to a SAS macro variable. You can have as many DBMS connection macro variables as you want. This SAS autoexec file defines macro variables for two DBMS connections, in addition to an OPTIONS statement:

```
/*** My SAS autoexec file ***/
options sasautos=(sasautos 'c:\MySAS\Macros')
    ps=46 ls=80 nodate nocenter;
%let MfgDB=%str(dsn=USAmfg uid=helf pwd=eer23r);
%let tvdb=%str(dsn=tvdb uid=helf pwd=mypw);
```

To use this DBMS connection information in your SAS programs, simply refer to the macro variable in the SAS/ACCESS LIBNAME statement or SQL PROCEDURE Pass-Through Facility *Connect* statement:

```
libname dbms db2 &tvdb;

proc sql;
 connect to db2 (&MfgDB);
 create table ... (Pass-Through query here)
 disconnect from db2;
 quit;
```

As you can see, there are no maintenance or security concerns when you specify your DBMS connection information in the SAS autoexec file. The program requires no maintenance when you change your DBMS password because the password is not stored in your program. When you change a DBMS password, you only need to change the macro variable assignment statement in your SAS autoexec file, and then all of your SAS programs that use that macro variable to connect to the DBMS will run fine without any changes. There are no security concerns with this method because your user ID and password are not stored in the program. In fact, if other SAS users use the same macro variable name in their SAS autoexec file to refer to the same DBMS database, you can send a program to them and they will be able to run it easily.

JOINING SAS AND DBMS TABLES EFFICIENTLY

There are many times when you want to write a query to a DBMS table and return data only for the values of a variable in a SAS data set. For example, you have a flat file of serial numbers from which you create a SAS data set with INFILE and INPUT statements in a DATA step. Or maybe you created a SAS data set from a query against a DBMS table at another location in your company, and now you want to query a DBMS table at your location only for the key values in your SAS data set. SAS provides tools for joining a SAS data set with a DBMS table.

PROC SQL CAN JOIN SAS AND DBMS TABLES, BUT BE CAREFUL!

SAS versions 6.12, 7, and 8 can all join a SAS table to a DBMS table in a PROC SQL step. For example, suppose we have a SAS data set called Warranty that contains the serial numbers (variable sn) for 100 televisions that were returned for warranty service, and we want to join it with a DBMS table called MfgHist which contains the manufacturing history for all one million TV sets made so far. The following PROC SQL step would work:

```
Proc SQL;
 create table History as
 select * from Warranty a, dbms.MfgHist b
 where a.sn=b.sn;
 quit;
```

In SAS Version 6.12, dbms.MfgHist refers to a view descriptor, which is associated with an access descriptor. A discussion of access and view descriptors is beyond the scope of this paper. In Version 7 and 8, dbms.MfgHist refers to a SAS/ACCESS LIBNAME data set association, as described in an earlier section.

What's wrong with this PROC SQL approach to join a SAS data set with a DBMS table? Nothing, if your DBMS table is fairly small. However, when the DBMS table is large, this PROC SQL step is very inefficient because every row in the DBMS table is returned to your SAS session and joined with your SAS data set by PROC SQL. In our TV warranty example, all one million rows from the MfgHist DB2 table will be returned to your SAS session, and all but 100 rows will be discarded because they don't match the serial numbers in our SAS data set.

There are two good ways to solve this problem. One solution for versions 7 and 8 only is to use the DBKEY= and DBINDEX= data set options with the SAS/ACCESS LIBNAME statement. Another is to write a macro that passes the values in your SAS data set in groups to an SQL Procedure Pass-Through Facility query. Such a macro can be used with version 6.12 as well as versions 7 and 8. This paper includes one such macro called %DBMSlist that I use extensively.

SAS/ACCESS LIBNAME STATEMENT

You can also join a SAS data set with a DBMS table using SAS/ACCESS LIBNAME techniques. There are system options, SAS/ACCESS LIBNAME statement options, and data set options that monitor and affect the efficiency of SAS steps that join SAS and DBMS tables. This section describes the SASTRACE system option, and the DBKEY= and DBINDEX= data set options.

DBKEY= DATA SET OPTION

The DBKEY= option lets you specify the column(s) in the DBMS table to use as a serach key. An actual index on this column in the DBMS table is not required, and SAS does not attempt to determine if one exists. It is used like this:

```
libname dbms db2 dsn=tvdb uid=helf pwd=mypw;
proc SQL;
 create table History as
 select * from Warranty a,
   dbms.MfgHist (dbkey=sn) b
 where a.sn=b.sn;
 quit;
libname dbms clear;
```

In this example, the DBKEY= option instructs the SQL procedure to pass the WHERE clause to the SAS/ACCESS engine in a form similar to WHERE SN=*host-variable*. The engine then passes this optimized query to the DBMS server. The host-variable is substituted, one at a time, with SN values from the observations in the SAS data set Warranty. As a result, only rows that match the WHERE clause are retrieved from the DBMS. Without this option, PROC SQL retrieves all the rows from the MfgHist table.

Here are some examples of the DBKEY= option. In these exmaples, data set LOTS contains a variable called LOT whose values will be used as a key to retrieve data from DB2 table db2sys.disc_stat_history. I know that column LOT in this DBMS table has an index.

Example 6 - Single Key Variable: This example shows a simple query with one variable as the search key:

```
proc sql;
 create table lot_data3 as
 select a.lot, a.trdate, a.trtime, a.resource
 from db2sys.disc_stat_history (dbkey=lot) a,
    lots b
 where a.lot=b.lot;
 quit;

TRACE: Using FETCH for file DISC_STAT_HISTORY on
     connection 0 6 1296151584 no_name 0 SQL
TRACE: Change AUTOCOMMIT to YES for connection
     id 0 7 1296151584 no_name 0 SQL
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT * FROM
     db2sys.DISC_STAT_HISTORY 8 1296151584
     no_name 0 SQL
TRACE: DESCRIBE on statement 0, connection 0. 9
     1296151584 no_name 0 SQL
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  LOT, TRDATE,
     TRTIME, RESOURCE  FROM
     db2sys.DISC_STAT_HISTORY  WHERE (((lot= ? )
     OR ((lot IS NULL ) AND ( CAST(? AS LONG
     VARCHAR) IS NULL )))) FOR READ ONLY  10
     1296151584 no_name 0 SQL
TRACE: Open Cursor with new index value 12
     1296151584 no_name 0 SQL
TRACE: Close cursor from statement 0 on
     connection 0 13 1296151584 no_name 0 SQL
```

Note that SAS first prepares a SELECT * statement and then does a DESCRIBE to find the names and attributes of the DBMS columns named in the query. Then, SAS prepares a statement to actually retrieve data and constructs a WHERE condition of

LOT=?. SAS then uses OPEN CURSOR and CLOSE CURSOR statements to pass each value of variable LOT in the SAS data set to the DBMS and return the results to SAS.

The performance of queries using the DBKEY= options is usually quite good. In this example, my LOTS dataset had 200 rows and it took 3 seconds to return the data from the DBMS table that had 120K rows. Without the DBKEY= option, SAS retrieves all of the rows in the DBMS table and applies the WHERE clause in SAS. When I removed the DBKEY= option from this query, it took 41 seconds to complete. I also ran a similar query with the DBKEY= option against a much larger table, about 5M rows, and this query took only 4 seconds.

However, I have seen cases where a PROC SQL query with the DBKEY= option takes 10 times longer than using the macro %DBMSlist described in the next section. This generally happens when the DBMS server is at a remote location with a low bandwidth network connection. The message here is that you need to carefully evaluate different methods for running queries that join SAS and DBMS tables.

Example 7 - Multiple Key Variables: This example shows a query where multiple variables in the SAS data set are used to form the search key. Note that you must enclose two or more column names in parentheses after the DBKEY= token. To save space, I will list only the SASTRACE statement that shows the statement SAS sends to the DBMS.

```
proc sql;
 create table lot_data as
 select a.lot, a.trdate, a.trtime, a.resource
 from db2sys.disc_stat_history (dbkey=(lot
    trdate)) a, lots b
 where a.lot=b.lot and a.trdate=b.trdate;
 quit;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  LOT, TRDATE,
     TRTIME, RESOURCE  FROM
     db2sys.DISC_STAT_HISTORY  WHERE (((lot= ? )
     OR ((lot IS NULL ) AND ( CAST(? AS LONG
     VARCHAR) IS NULL ))) AND ((trdate= ? ) OR
     ((trdate IS NULL ) AND (CAST(? AS
     VARCHAR(50))  IS NULL )))) FOR READ ONLY
```

It appears that PROC SQL with the DBKEY= option will only create WHERE clauses when the join condition is equality, for example A.LOT=B.LOT. If you want to use other SQL clauses like BETWEEN or greater than, use SASTRACE and make sure SAS is building an SQL statement that is efficient for your DBMS.

Example 8 - DBKEY= Option in a DATA step: You can also use the DBKEY= data set option in a DATA step by using the KEY=DBKEY option of the SET statement. This example uses a data step to join the SAS data set with a DBMS table:

```
data lot_data;
 set lots;
 set db2sys.disc_stat_history (dbkey=lot
   keep=lot trdate trtime resource)
   key=dbkey;
 run;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  TRDATE, TRTIME,
     LOT, RESOURCE  FROM db2sys.DISC_STAT_HISTORY
     WHERE (((lot= ? ) OR ((lot IS NULL ) AND (
     CAST(? AS LONG VARCHAR) IS NULL )))) FOR
     READ ONLY
```

WARNING! Be very careful with this DATA step method when the DBMS table may contain multiple rows for each value of your key variable. This DATA step method returns only one row for each value of the key variable, and the row returned may not even be the same if you rerun the query! For example, when I had 10 values of key variable LOT in the LOTS data set, the PROC SQL method in Example 6 returned 202 rows from the DB2 table (because many transactions are performed on each lot during our manufacturing process), but the DATA step method above returned only 10 rows. Also, queries to a DBMS table do not return data in any particular order unless you use the ORDER BY clause. Therefore, I saw cases where several rows in the result set were different when I reran the DATA step compared to the first time. The message here is that even though this DATA step method is described in the SAS documentation and supported by SAS Institute, it may be safer to always use the PROC SQL method instead.

Example 9 - DBNULLKEYS= Data Set Option: In Example 6 we saw that SAS builds a WHERE expression like this when you use the DBKEY= option:

```
WHERE (((lot= ? ) OR ((lot IS NULL ) AND (
     CAST(? AS LONG VARCHAR) IS NULL ))))
```

You can see that the WHERE clause is actually two clauses separated by the OR operator. The first clause is for non-null values of your key variable. The second clause is for null values of the key variable. A compound clause like this is less efficient for your DBMS to process than a simple clause, so SAS provides (starting in Release 8.2) another data set option called DBNULLKEYS= that tells PROC SQL to create the non-null WHERE clause only. If you know that your key variable does not contain null values, you should use this option. The syntax for this option and the WHERE clause created look like this:

```
proc sql;
 create table lot_data as
 select a.lot, a.trdate, a.trtime, a.resource
from db2sys.disc_stat_history (dbkey=lot
 dbnullkeys=no) a, lots b
 where a.lot=b.lot;
 quit;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  "LOT", "TRDATE",
     "TRTIME", "RESOURCE"  FROM
     db2sys.DISC_STAT_HISTORY  WHERE ((("LOT"= ?
     )))) FOR READ ONLY
```

In Release 8.2, the SAS/ACCESS engines checks to see if a DBKEY column is non-nullable, and if so, automatically generates the simpler form of the query. If the DBKEY column was created as nullable, and you know there is no null data in your SAS data set, you can use the the DBNULLKEYS= data set option to send the simpler form of the query to the DBMS.

DBINDEX= DATA SET OPTION

There is another data set option described in the SAS documentation called DBINDEX= that can be used to improve efficiency of PROC SQL queries that join SAS and DBMS tables. Unfortunately, my experience with this option does not match the documentation. The documentation for DBINDEX= shows that you use this feature to tell SAS to query the DBMS to find indexes on the DBMS table. SAS then attempts to use the indexes on the DBMS table to improve performance. However, my experience is that SAS constructs the same query no matter what value you use with the DBINDEX= option, and that the query SAS constructs depends only on the number of observations in your key values SAS data set.

Example 10 - Small Key Values Data Set: When my SAS data set contains between 1 and 200 observations, SAS constructs a WHERE statement that has an IN list of the key values. In this example, my key values data set has 10 observations:

```
proc sql;
 create table lot_data as
 select a.lot, a.trdate, a.trtime, a.resource
 from db2sys.disc_stat_history (dbindex=yes)
   a, lots b
 where a.lot=b.lot;
 quit;
```

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  LOT, TRDATE,
```

```
TRTIME, RESOURCE  FROM
db2sys.DISC_STAT_HISTORY  WHERE  ( ( LOT IN
( 'DI00036136' , 'DI00037986' , 'DI00038082'
, 'DI00038359' , 'DI00038836' , 'DI00038915'
, 'DI00038929' , 'DI00039471' , 'DI00039476'
, 'DI00039480' ) ) ) FOR READ ONLY
```

Example 11 - Large Key Values Data Set: When my SAS data set contains 201 or more observations, SAS creates an SQL statement to return all rows from the DBMS table and processes the join in SAS! This is not what you want SAS to do if your DBMS table is large. Here is the TRACE output for the PROC SQL step from Example 10, but with 201 observations in my key values data set:

```
TRACE: SQL stmt prepared on statement 0,
     connection 0 is:  SELECT  LOT, TRDATE,
     TRTIME, RESOURCE  FROM
     db2sys.DISC_STAT_HISTORY  FOR READ ONLY  617
     1296430260 no_name 0 SQL
```

This query took 55 seconds to run because it brought all 200K rows from the DBMS table into SAS. With 200 observations in my key values dataset, the query ran in 8 seconds because it submitted a WHERE statement to the DBMS. The message here is be sure to use SASTRACE to test your query to see if the DBINDEX= option does what you want. It may work differently with different DBMS systems.

MACRO %DBMSLIST GENERATES PROC SQL PASS-THROUGH QUERIES

Most of the function of macros like %DBMSlist has been replaced by the more elegant techniques just described in the SAS/ACCESS LIBNAME statement in versions 7 and 8.

The trick to writing a macro that generates SQL Pass-Through queries is to format the key values in your SAS data set into an appropriate WHERE clause for your DBMS. In our TV example, the TV serial numbers are defined as character strings in the DB2 table, so the values of variable SN in the SAS data set must be enclosed in single quotes and separated by commas. You might also want your macro to handle key values that are defined as other data types in the DBMS table, such as numeric, date, time, or timestamp. A more complex macro would handle requests based on multiple key columns, for example a serial number and a manufacturing operation number. The %DBMSlist macro discussed next has all of these features.

Macro %DBMSlist has several arguments: the name of the SAS data set your key values are in, the name(s) of the variable that contain the values you want to pass to the DBMS, the type of variables they are, the name of the SAS data set you want to create, and the query you want to run. The macro builds your key values into a list and assigns it to a macro variable, then runs your query and puts the data from the DBMS into the SAS data set you specified. The syntax of macro %DBMSlist is:

```
%DBMSlist(dsn, column, vtype, newdsn, dbname,
  query, bitesize=200, test=no, dlm=%str(#));
```

These arguments to macro %DBMSlist are described in detail in Table 1.

Table 1: Syntax of the %DBMSlist Macro

Parameter	Meaning
dsn	Name of the existing SAS data set that contains the values you want to include in a DBMS query. May be a temporary data set (with a one-part name) or a permanent data set (with a two-part name).
column	Name of the variable (or variables) in the data set named in **dsn** that contains the values you want to submit to the DBMS. If more than one variable name, **column** must be in the form of an SQL template enclosed in the **%str** function, with variable names delimited by a special character (the # symbol by default). Refer to Table 2 for more information.
vtype	Type of variable(s) named in **column**, must have one value for each variable named in **column**. Not case sensitive. Currently, 5 types are supported: • **c** - for character data, values are enclosed in single quotes • **n** - for numeric data • **d** - for date data, values are put in DB2 query format, like '1998-11-22'. • **t** - for time data, values are put in DB2 query format, like '10.32.45'. • **dt** - for timestamp data, values are put in DB2 query format, like '1998-10-11-21.34.54.234242'
newdsn	Name of the new SAS data set you want the macro to create. It may be either temporary or permanent.
dbname	Connection information appropriate for your DBMS. For DB2, it can be: 1) The word **prompt**, in which you will be prompted for userid and password 2) Explicit connection information, like **%str(dsn=engdb uid=helf pwd=mypw)** 3) A macro variable that contains your DBMS connection information, like **&engdb**. These macro variables should be placed in your autoexec.sas file for security and maintenance reasons.
query	The query you want the DBMS to process. Put the macro variable **&mylist** in your WHERE statement to specify your list of key values. The query must be enclosed in the macro function **%nrstr**.
bitesize	Optional parameter, default value is **200**. Only this many serial numbers are passed to the DBMS at a time. Make this value smaller if the query takes too long or you get any errors from the DBMS. You can make it larger if you have a large list and the query runs OK.
test	Optional parameter, default value is **no**. If it is anything else, like **yes** or **Yes** or **YES**, the macro passes the first **bitesize** values to the DBMS and does **not** create the output data set. I use this for testing to see how long the query will take with different values of **bitesize**.
dlm	Optional parameter, default value is the pound symbol (#). Defines the delimiter to be used for marking variables in the SQL template. To enter a different character, use the %str function, for example **%str(@)** to use the @ symbol as a delimiter.

Argument COLUMN of Macro %DBMSlist takes different forms depending on whether one variable or more than one variable in the SAS data set forms the key for the DBMS table query. This argument is described in Table 2.

Table 2: Argument COLUMN for Macro %DBMSlist:

Number of key variables	Value of &COLUMN	Contents of &mylist and Where statement syntax

1 variable name	Single name of the variable that contains the value to be passed to the DBMS, for example:
	
	`file`
		&mylist contains values of the variable you specified, separated by commas, like:
		`'13001111HK',`
		`'13002222HK',`
		`'13003333HK'`
		
		Use a Where statement like:
		`where file in (&mylist)`

More than one variable name	An SQL template enclosed in the %str function that	&mylist contains the SQL template you specified, with variable names

	consists of static text and variable names. Each variable name must be delimited at start and end by a special character (# by default). Example: `%str(wafer=#my waf# and row=#myrow#)`	substituted with the values of those variables, enclosed in parentheses and separated by the word OR, like: `(wafer=258053 and row=33) or (wafer=780784 and row=12)` Use a Where statement like: `where (&mylist)`

Note about Macros: There are several ways to make these macros visible to your SAS program. The easiest way is to set the SASAUTOS option to point to the directory where you put them, like this:

```
options sasautos=(sasautos,'C:\MYSAS\MACRO');
```

You need to run this only once in each session. Better yet, add it to your AUTOEXEC.SAS file so it gets run every time you start SAS.

You might see a warning message about string length. I've noticed a warning message in the SAS 6.12 log about concatenated string length longer than 200 characters, but the macro still runs fine. I don't get this message in SAS 7 or 8.

Example 12 - Single Character Key Variables: This macro call takes a list of serial numbers in variable SHIPCASS in SAS data set PACKPREP and creates a new SAS data set called DISKSHIP that contains the data returned from the ENGDB database. The DBMS connection information was previously assigned to macro variable &ENGDB.

```
%DBMSlist(packprep, shipcass, c, diskship,
&engdb, %nrstr(
 select cassette, trdate, trtime, pwrpak
 from d.current_status
 where cassette in (&mylist) ));
```

Example 13 - Multiple Numeric Key Variables: This macro call takes a list of serial numbers in variables WAFER and ROW in SAS data set MYDATA.SLIDER and creates a new SAS data set called RAWQ that contains data returned from the FABDB2 database. The DBMS connection information was previously assigned to macro variable &FABDB2. The BITESIZE parameter is set to 40 input rows at a time becuase I got an application heap storage size error from the DB2 server when I tried using more.

```
%DBMSlist(mydata.slider, %str(wafer=#wafer#
and row=#row#), n n, rawq, &fabdb2, %nrstr(
 select wafer, row, columnn, wafersize as
size, date, time, stationid as station,
p145201 as HT, p145202 as EC,P145203,p145205
from hrs.op1452 where (&mylist) ),
bitesize=40);
```

SOURCE CODE FOR MACROS

The source code for the three macros %DBMSlist, %MakeList, and %RunQuery are included below. Copy and paste each one to a file by the same name in your autocall macro library to make them visible to your SAS programs. The macro you use is %DBMSlist, the other two are called by this macro. You can copy this code from the CD copy of the Proceedings you receive at the conference, or from the SUGI web site after the conference: http://www.sas.com/usergroups/sugi/proceedings/index.html.

SOURCE CODE FOR MACRO %DBMSLIST

```
%Macro DBMSlist(dsn, column, vtype, newdsn,
    dbname, query, bitesize=200, test=no,
    dlm=%str(#) );

proc sql noprint;
 select count(*) into :gwhxxxx1
 from &dsn;
 quit;

%if &gwhxxxx1=0 %then %do;
 %put ====== WARNING: Input data set &dsn is
    empty, macro ends =======;
 %goto exit;
 %end;

%let totpass=%sysevalf(&gwhxxxx1/&bitesize,
    ceil);

%if &test=no %then %do j=1 %to &gwhxxxx1 %by
    &bitesize;
%let p=%sysevalf(&j/&bitesize, ceil);
%put ================= Starting pass &p of
    &totpass ==================;
data gwhxxxx2;
 set &dsn (firstobs=&j
    obs=%eval(&j+&bitesize-1));
 run;

%MakeList(mylist, gwhxxxx2, &column, &vtype);
%RunQuery(&dbname, gwhxxxx3, &query);

%if &j=1 %then %do;
data &newdsn;
 set gwhxxxx3;
 run;
%end;
%else %do;
Proc append base=&newdsn data=gwhxxxx3;
 run;
%end;
%end;

%else %do; %* Test=yes: do one query for timing;
data gwhxxxx2;
 set &dsn (firstobs=1 obs=&bitesize);
 Run;

%MakeList(mylist, gwhxxxx2, &column, &vtype);
%RunQuery(&dbname, gwhxxxx3, &query);

%end;
%exit: %mend DBMSlist;
```

SOURCE CODE FOR MACRO %MAKELIST

You might need to adjust the formatting for the date, time, and datetime SAS variable types for DBMS systems other than DB2.

```
%macro MakeList(globname, dsn, varinfo,
    vartype);

%local i j;
%global &globname;
%let &globname=; /* return null if macro fails*/

%let numvars=1; %* find number of variables
    specified;
%do %while (%scan(&vartype,%eval(&numvars+1))
    ne);
 %let numvars=%eval(&numvars+1);
 %end;

/*******************************************
 Single variable entered
 *******************************************/
%if &numvars=1 %then %do;

/*************** Character ********/
```

```
%if %upcase(&vartype)=C %then %do;
proc sql noprint;
 Select
    Distinct translate(quote(&varinfo),"'",'"')
 into :&globname separated by ','
 from &dsn;
 quit;
%end;

/*************** Numeric ********/
%else %if %upcase(&vartype)=N %then %do;
proc sql noprint;
 select distinct &varinfo
 into :&globname separated by ','
 from &dsn;
 quit;
%end;

/*************** Date ********/
%else %if %upcase(&vartype)=D %then %do;
proc sql noprint;
 select distinct "'"||
    put(&varinfo, yymmdd10.)||"'"
 into :&globname separated by ','
 from &dsn;
 quit;
%end;

/*************** Time ********/
%else %if %upcase(&vartype)=T %then %do;
proc sql noprint;
 select distinct "'"||
    translate(put(&varinfo, time.),
    '.',':','0',' ')||"'"
 into :&globname separated by ','
 from &dsn;
 quit;
%end;

/*************** Datetime ********/
%else %if %upcase(&vartype)=DT %then %do;
proc sql noprint;
 select distinct "'"||put(datepart(&varinfo),
    yymmdd10.)||"-"||
    translate(put(timepart(&varinfo),
    time15.6),'.',':','0',' ')||"'"

 into :&globname separated by ','
 from &dsn;
 quit;
%end;

%else %put ******* Invalid variable type:
    &vartype *************;

%end;  /* %if &numvars=1  */

/*******************************************
 Multiple variables entered
 *******************************************/
%else %do;

/***** Parse SQL template ******************/
%let j=1;
%do %while (%index(%quote(&varinfo), &dlm)>0);
 %let markloc=%index(%quote(&varinfo), &dlm);
 %let text&j=%substr(%quote(&varinfo), 1,
    %eval(&markloc-1));
 %let varinfo=%substr(%quote(&varinfo),
    %eval(&markloc+1),
    %eval(%length(&varinfo)-&markloc) );
 %let markloc=%index(%quote(&varinfo), &dlm);
 %let var&j=%substr(%quote(&varinfo), 1,
    %eval(&markloc-1));
 %if %length(&varinfo)>&markloc %then
    %let varinfo=%substr(%quote(&varinfo),
    %eval(&markloc+1),
    %eval(%length(&varinfo)-&markloc) );
 %else %let varinfo=;
 %let j=%eval(&j+1);
 %end;

/*** Build macro variable with Proc SQL ******/
proc sql noprint;
 select distinct '(' ||

%do i=1 %to &j-1;

/***** Character variable ******************/
%if %upcase(%scan(&vartype, &i))=C %then
    " &&text&i " ||
    translate(quote(&&var&i),"'",'"') ||;

/***** Numeric variable ********************/
%else %if %upcase(%scan(&vartype, &i))=N %then
    " &&text&i " || compress(put(&&var&i,
    best20.)) ||;

/***** Date variable ***********************/
%else %if %upcase(%scan(&vartype, &i))=D %then
    " &&text&i '" || put(&&var&i,
    yymmdd10.)||"'" ||;

/***** Time variable ***********************/
%else %if %upcase(%scan(&vartype, &i))=T %then
    " &&text&i '" ||
    translate(put(&&var&i, time.),'.',':','0',
    ' ')||"'"||;

/***** Datetime variable *******************/
%else %if %upcase(%scan(&vartype, &i))=DT %then
    " &&text&i '" || put(datepart(&&var&i),
    yymmdd10.)||
    "-" ||translate(put(timepart(&&var&i),
    time15.6), '.',':','0',' ')||"'"||;

%else %put ********** Invalid variable type:
    %scan(&vartype,&i) *************;

%end;  /*  %do i=1 %to &j-1  */

   " &varinfo)" into :&globname separated by
    ' or ' from &dsn;
 quit;
%end;  /* %else for %if &numvars=1  */
%mend MakeList;
```

SOURCE CODE FOR MACRO %RUNQUERY

Change the value for macro variable &DBMStype to the correct value for your DBMS - DB2, Oracle, Teradata, etc.

```
%macro RunQuery(dbinfo, dsname, query);
%let DBMStype=DB2;
proc sql;
 connect to &DBMStype (&dbinfo);
 create table &dsname as select * from
    connection to &DBMStype (
    %unquote(&query) for fetch only);
 %put &sqlxmsg;
 disconnect from &DBMStype;
quit;
%mend RunQuery;
```

GETTING DATA FROM SAS INTO DBMS TABLES

So far we discussed how to get DBMS data into SAS. Now let's see how to get data from SAS data sets into DBMS tables.

SAS/ACCESS LIBNAME STATEMENT

It's very easy to create DBMS tables and insert data into those tables using the SAS/ACCESS LIBNAME statement, assuming your user ID has the privilege to perform those operations in the DBMS. To create a DBMS table and insert data from a SAS data set, just name a SAS/ACCESS LIBNAME data set anywhere in a procedure or DATA step where an output SAS data set is usually named. For example, this DATA step creates a DBMS table called product_sales from the SAS data set prdsale in the SASHELP library:

```
libname sampdb db2 dsn=sample uid=helf
pwd=rete45tr;
data sampdb.product_sales (dbcommit=100);
 set sashelp.prdsale;
 run;
```

The DBCOMMIT= data set option tells SAS to issue a DBMS Commit command after the number of rows specified have been processed. This may improve DBMS performance if you have a large SAS data set. There are many other data set options that affect how SAS/ACCESS interacts with the DBMS. These are described in the SAS/ACCESS Software for Relational Databases reference manual.

When you create a DBMS implicity like this, the SAS/ACCESS engine chooses the DBMS data type for each column based on the format assigned to that variable in the SAS data set. The mappings between SAS variable format and DBMS data type are described in the SAS/ACCESS Reference manual supplemental chapter for each DBMS. If you don't like the DBMS types used for some variables in the SAS data set, you can explicitly create the DBMS table with an SQL Procedure Pass-Thru Facility statement, as described in the next section.

If the DBMS table already exists, and you want to add data from a SAS data set to it, the APPEND procedure works well:

```
proc append base=sampdb.product_sales
(dbcommit=100) data=NewSales;
 run;
```

SQL PROCEDURE PASS-THRU FACILITY

You can create a DBMS table explicitly in the SQL Procedure Pass-Thru Facility by using the EXECUTE statement. This statement sends non-SELECT SQL statements to the DBMS. For example, you can create a DBMS table with this step:

```
proc sql;
connect to db2 (&sample);
execute (create table employee (ID smallint,
 name varchar(9), dept smallint,
 job char(5), salary decimal(7,2)) ) by db2;
disconnect from db2;
quit;
```

Because statements in the Pass-Thru Facility are sent unaltered to the DBMS server, you must make sure the SQL statement conforms to the type and version of DBMS server you are using. Other useful SQL statements you can send to the DBMS include DELETE, DROP, GRANT, INSERT, and UPDATE. You need to look at the SQL documentation for your DBMS to find the correct syntax for these statements.

BULK LOADING DATA INTO DBMS TABLES

Many DBMS vendors provide a bulk-loading utility that loads large amounts of data into a DBMS table much faster thant the SQL INSERT-based methods used by the Pass-Thru Facility and SAS/ACCESS LIBNAME statement. There are many DBMS-specific features to bulk loading, so refer to the SAS/ACCESS Reference manual supplemental chapter for your DBMS. Some DBMS vendors require a higher level of privileges to use the bulk loader compared to other SQL methods described in this paper. To use the bulk loader, you use the data set option BULKLOAD=yes and must usually specify other data set options as well.

DBLOAD PROCEDURE

The DBLOAD procedure is an older tool for creating DBMS tables and loading data into them. SAS Institute recommends that you use the SAS/ACCESS LIBNAME techniques instead of the DBLOAD procedure, but if you are still using SAS Version 6, you may need to know about DBLOAD.

Here is an example of the DBLOAD procedure. There are other options available - refer to the SAS/ACCESS documentation for details. In this example, a DB2 table called PRODSALE is created from SAS data set SASHELP.PRDSALE:

```
proc dbload dbms=db2 data=sashelp.prdsale;
 in='sample'; user='helf'; password='wr5g0p';
 table=helf.prodsale;
 load;
 Run;
```

There is also an APPEND keyword that lets you append a SAS data set to an existing DBMS table:

```
proc dbload dbms=db2 data=newsales append;
 in='sample'; user='helf'; password='lapt0p';
 table=helf.prodsale;
 load;
 run;
```

CONCLUSION

SAS/ACCESS software provides powerful tools for retrieving data from many heterogenous sources, including most relational database software in use today. Using the techniques described in this paper, you can easily get DBMS data into SAS, join SAS and DBMS tables efficiently, and create and load DBMS tables from SAS data sets.

REFERENCES

These manuals are available in hard copy format from SAS Institute Publishing, in HTML format on the *SAS OnlineDoc CD-ROM* that ships free with Version 8 of SAS software, in PDF format on the *SAS OnlineDoc CD-ROM with PDF Files*, and in PDF format you can download from e-Publishing at the SAS web site. You need the *Reference* manual plus the supplemental chapters for the DBMS system(s) you use. (The supplemental chapters have only one chapter each; since you can view the first chapter of each SAS book in PDF format on the SAS web site for free, just print the preview file and save $10!)

SAS Institute Inc. (1999), *SAS/ACCESS Software for Relational Databases: Reference, Version 8,* Cary, NC: SAS Institute Inc. Hard copy: PubCode 57204, $46; e-Publishing: 57954, $32.

SAS Institute Inc. (1999), *SAS/ACCESS Software for Relational Databases: Reference, Version 8 (DB2 under UNIX and PC Hosts Chapter)*, Cary, NC: SAS Institute Inc. Hard copy: PubCode 57207, $10; e-Publishing: 57955, $7.

SAS Institute Inc. (1999), *SAS/ACCESS Software for Relational Databases: Reference, Version 8 (Oracle Chapter)*, Cary, NC: SAS Institute Inc. Hard copy: PubCode 57211, $10; e-Publishing: 57959, $7.

SAS Institute Inc. (1999), *SAS/ACCESS Software for Relational Databases: Reference, Version 8 (Teradata Chapter)*, Cary, NC: SAS Institute Inc. Hard copy: PubCode 57203, $10; e-Publishing: 57964, $7.

Note: there are supplemental chapters for many other DBMS versions - look on the SAS web site under "Publishing".

Levine, Fred (2001) "Using the SAS/ACCESS Libname Technology to Get Improvements in Performance and Optimizations in SAS/SQL Queries," *Proceedings of the Twentysixth Annual SAS Users Group International Conference*, 26, Paper 110-26.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Garth W. Helf
IBM Corporation
5600 Cottle Road
San Jose, CA 95193
Work Phone: 408-256-7514
Fax: 408-256-2410
Email: helf@us.ibm.com

TRADEMARK CITATION