ABSTRACT

The SAS/ACCESS interface software to the relational database management systems DB2®, SQL/DS®, ORACLE®, and Rdb/VMS™ provides SAS users with an easy way to access data stored in relational database management system (DBMS) tables. After mastering the basic tasks such as extracting a specific subset of data from a relational DBMS table or inserting new rows into a relational DBMS table, users may be faced with more difficult tasks such as extracting data from a relational DBMS table using key values from a SAS data set or designing applications that allow interactive access to data in relational DBMS tables. This paper describes ways to accomplish some of the more advanced tasks that users of SAS/ACCESS interface software might face. This paper is intended for users who are already familiar with the basic use of the SAS/ACCESS interface to either DB2, SQL/DS, ORACLE, or Rdb/VMS.

INTRODUCTION

Basic SAS programming tasks that access data stored in relational DBMS tables are easy to accomplish thanks to the user-friendliness of SAS/ACCESS software. Times will come when it will be necessary for a SAS program to use a more advanced approach when accessing data from a database management system. The reason may be that a more sophisticated application is being designed or there is a need for greater processing efficiency than is available with the basic methods. The techniques used for these advanced needs are essentially the same for all the SAS/ACCESS interfaces to relational database management systems. These SAS/ACCESS products include the interfaces to DB2, SQL/DS, ORACLE, and Rdb/VMS.

Some of the more advanced needs addressed by this paper are:
- extracting data from a DBMS table using key values from a SAS data set,
- updating a DBMS table with values from a SAS data set,
- combining data from two DBMS tables,
- using SAS/ACCESS views in Screen Control Language (SCL) applications with SAS/AF® and SAS/FSP® software,
- controlling access to a subset of a DBMS table based on a user.

The techniques explained in this paper are generally applicable to Release 6.06 of the SAS® System. The techniques described are also based on efficient coding techniques for the SAS/ACCESS products. These efficiency guidelines are from papers presented at SUGI 15 (Plemmons 1980, Jacobs 1990). For explanations of these efficiencies, please refer to these papers. The example DBMS tables used in this paper are based on the sample tables distributed with the SAS/ACCESS Interface products. See the "Example Data" appendix in your usage and reference guide for the SAS/ACCESS Interface for more information on these tables.

EXTRACTING DATA FROM DBMS TABLES USING KEY VALUES FROM A SAS DATA SET

One of the more common tasks that is needed is to extract rows from a DBMS table based on key values stored in a SAS data set. A typical example of this would be a SAS data set which contains a particular subset of customer identification numbers that are of interest. You want to go to the CUSTOMERS DBMS table and retrieve the rows which correspond to this set of customer identification numbers.

Suppose that there is a SAS data set named CUSTNUMS in the SAS data library with the libref DATA/B. This data set has one variable called ID whose value is the customer identification number of a customer that is of interest. This data set could have been generated by a user of an interactive application or by another application. An example of this data set is as follows:

<table>
<thead>
<tr>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>14569677</td>
</tr>
<tr>
<td>27654351</td>
</tr>
<tr>
<td>43459747</td>
</tr>
</tbody>
</table>

The information on all the customers has been stored in a DBMS table called CUSTOMERS. A SAS/ACCESS view descriptor has been created for this table by the name of CUSTVIEW and stored in the library referred to by the libref VIEWLIB. A portion of this table is as follows:

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>14569677</td>
<td>Precision Products</td>
<td>190 Fayetteville Rd</td>
</tr>
<tr>
<td>14966629</td>
<td>University Biomedical</td>
<td>1508 Piccard Dr</td>
</tr>
<tr>
<td>27654351</td>
<td>Institut De Recherche</td>
<td>103 Rue D'Egmont</td>
</tr>
<tr>
<td>48045514</td>
<td>Gulf Scientific</td>
<td>P.O. Box 6032</td>
</tr>
</tbody>
</table>

The resulting data set that is desired is a data set that has the name and address for each customer identification number in the CUSTNUMS data set. For the CUSTNUMS data set shown above, this data set would be as follows:

<table>
<thead>
<tr>
<th>CUSTOMER</th>
<th>NAME</th>
<th>ADDRESS</th>
</tr>
</thead>
<tbody>
<tr>
<td>14569677</td>
<td>Precision Products</td>
<td>190 Fayetteville Rd</td>
</tr>
<tr>
<td>27654351</td>
<td>Institut De Recherche</td>
<td>1508 Piccard Dr</td>
</tr>
<tr>
<td>43459747</td>
<td>Research Outfitters</td>
<td>191 Lower Plenty Rd</td>
</tr>
</tbody>
</table>

Once the problem has been defined as above, it is recognized as a join or a merge problem. Possible solutions to this problem are as follows:

- Use the SQL procedure to join the SAS data set and DBMS table.
- Use a DATA step to merge the SAS data set and DBMS table.
• Use the macro language to generate a WHERE statement from the SAS data set to retrieve rows from the DBMS table.

The first solution is to use PROC SQL to join the SAS data set and the SAS/ACCESS view descriptor. This solution is the easiest to code, but it may become less efficient as the table becomes large. This is because all the join processing is done in the SAS System. An example of how the PROC SQL code would look to join these two files is as follows:

```sql
proc sql;
    select custnums.id, custview.name, custview.address
    from datalib.custnums, viewlib.custview
    where custview.customer = custnums.id;
run;
```

In this example the first file listed in the FROM clause is the SAS data set which contains the customer identification numbers for the desired customer names and addresses. The second file listed in the FROM clause is a SAS/ACCESS view descriptor which retrieves data from the DBMS table. The WHERE clause identifies the join condition used to specify which rows you want in the resulting file.

The second solution that you could use for this problem is to merge the SAS data set and the DBMS table. This can be coded in a DATA step using the MERGE statement. One requirement of the MERGE statement is that both input data sets must be presented in sorted order by the key variable. How that sorted order is achieved is different for the SAS data set and the DBMS table. The SAS data set must either be created in sorted order or processed by the SORT procedure prior to the DATA step to achieve the sorted order. As for the DBMS table, one feature of the SAS/ACCESS engines is that if a particular order is requested through a BY statement that request is passed through to the DBMS. A greater efficiency can be achieved because the DBMS can select the best method to sort the requested table. Some example DATA step code which uses this technique is as follows:

```sql
data mergecust;
    merge datalib.custnums (in=in_cust, rename=(id=customer))
    viewlib.custview
    by customer;
    if in_cust then output;
run;
```

The RENAME= data set option is used in the CUSTNUMS data set to rename the ID variable to CUSTOMER. This is because the BY statement requires that the BY variable have the same name in both data sets being merged. The IN= data set option identifies which of the merged rows had values contributed by the CUSTNUMS data set. The subsetting IF statement is required so that only the merged rows that include the key values are written to the output data set.

If the sort of the DBMS table becomes too expensive, this technique can be modified to do a straight sequential pass of the DBMS table. The modification that needs to be made is to use a SET statement to read in the DBMS table and then search for that row in the SAS data set. An example of a DATA step that uses this technique is as follows:

```sql
data combcust;
    set viewlib.custview;
    /* any table look-up */
    /* technique can be used */
    do i = 1 to nobscust while (not found);
        set datalib.custnums point=1 nob=1; //nob=nobscust;
        if id = customer then found = 1;
        endif; //end do loop
    if found then output;
run;
```

The inner DO loop can be replaced with any table look-up technique. See the PROCEEDINGS from SUGI 14 (Johnston and Ray 1989) for more information on table look-up technique.

The previous solution with the SET statement and a table look-up on the SAS data set is somewhat backward from standard programming practices. A more standard approach would be to read through the key values in the SAS data set and search the larger DBMS table for the matching rows. The method used to implement this solution in a DATA step, is to generate a WHERE statement which contains the key values from the SAS data set. This solution is more difficult to code in your application, but it can be much more efficient than previous solutions if the subset to be retrieved is a small percentage of the total DBMS table and the key field in the DBMS table is indexed. To implement this solution, you first need a DATA step to generate macro variables containing the key values. An example is as follows:

```sql
%macro getit;
    data getcusts;
        set viewlib.custview;
        where customer in %do i = 1 %to %eval(wherenum-1);
            "%thei",
        %end;
    %end;
%mend;
getit;
```

After this step executes all, the key values are stored in separate macro variables. The names of the macro variables that contain the key values are WHERE1, WHERE2, WHERE3, etc... A macro variable is also created that contains the total number of key values. This macro variable is named WHERENUM. To extract the rows from the DBMS table, another DATA step is used. This DATA step has a WHERE statement which uses an IN expression to specify the rows desired. This WHERE statement is generated by a macro from the macro variables created above. An example macro and DATA step are as follows:

```sql
%macro getit;
    data getcusts;
        set viewlib.custview;
        where customer in %do i = 1 %to %eval(wherenum-1);
            "$where1",
        %end;
    %end;
%mend;
getit;
```

The %DO loop in the macro has an upper limit set by the number of key values. The %DO loop's function is to retrieve the key values from the WHEREIN macro variables and generate the list of literals for the IN expression. The reason the WHEREIN macro variable
is decremented by one so that the last key value is not followed by a comma. This is proper syntax for the IN expression. If this macro is executed using the key values from the CUSTNUMS data set discussed earlier, it generates the following code:

data getcusts;
set viewlib.custview;
where customer in ("18543489", "19783482", "14324742");
run;

The use of the WHEN statement is preferred over the nesting IF statement because the SAS/ACCESS engines can pass this condition through to the DBMS. This enables the DBMS to use available indexes to satisfy the request. Note that memory space is required for each macro variable. If there is not enough memory available for these macro variables, the key values may be written to an external file and then included into a later DATA step with the %INCLUDE statement.

UPDATING A DBMS TABLE WITH VALUES FROM A SAS DATA SET

A task similar to extracting data from DBMS tables using key values from a SAS data set is to use the values from a SAS data set to update columns in a DBMS table. A typical application where this might occur is an application which updates the street address for an external file for each observation in a temporary data set and generates SQL UPDATE statements that can be submitted to the DBMS. Once the DATA step writes one UPDATE statement to the temporary external file for each observation in the ADDACHING data set, the SET clause of each UPDATE statement contains the value of the new address for that customer and the WHERE clause uses the value of the IDNUM variable to locate the correct customer whose address needs to be updated. After this DATA step executes, the temporary external file contains SQL UPDATE statements that look like this:

<table>
<thead>
<tr>
<th>IDNUM</th>
<th>NEWADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>18543489</td>
<td>1201 Silverado Way</td>
</tr>
<tr>
<td>19783482</td>
<td>458 Patriots Blvd</td>
</tr>
<tr>
<td>14324742</td>
<td>3323 Oceanside Dr</td>
</tr>
</tbody>
</table>

This SAS data set could have been generated from an interactive data entry application or some other application. The method used to make the updates in the DBMS table takes the values from the data set and generates SQL UPDATE statements that can be submitted to the DBMS. You must generate one UPDATE statement for each observation in the SAS data set. The IDNUM variable in the data set locates the correct row for the update and the NEWADDR variable specifies the new address for that customer. A DATA step which generates these UPDATE statements and writes them to a temporary file is as follows:

```
filename tempsql 'tempfile';
data _null_;
file tempsql;
put "SQL UPDATE CUSTOMERS SET ADDRESS = '1201 Silverado Way' WHERE CUSTOMER = '18543489';";
put "SQL UPDATE CUSTOMERS SET ADDRESS = '458 Patriots Blvd' WHERE CUSTOMER = '19783482';";
put "SQL UPDATE CUSTOMERS SET ADDRESS = '3323 Oceanside Dr' WHERE CUSTOMER = '14324742';";
run;
```

The DATA step writes one UPDATE statement to the temporary external file for each observation in the ADDACHING data set. The SET clause of each UPDATE statement contains the value of the new address for that customer and the WHERE clause uses the value of the IDNUM variable to locate the correct customer whose address needs to be updated. After this DATA step executes, the temporary external file contains SQL UPDATE statements that look like this:

```
SQL UPDATE CUSTOMERS
  SET ADDRESS = '1201 Silverado Way'
  WHERE CUSTOMER = '18543489';
SQL UPDATE CUSTOMERS
  SET ADDRESS = '458 Patriots Blvd'
  WHERE CUSTOMER = '19783482';
SQL UPDATE CUSTOMERS
  SET ADDRESS = '3323 Oceanside Dr'
  WHERE CUSTOMER = '14324742';
```

Once these UPDATE statements have been generated and written to the temporary external file, they can be included into a PROC DBLOAD step using the %INCLUDE statement. This PROC DBLOAD step would be as follows:

```
proc dbload dbms=your-dbms;
  include tempsql;
run;
```

The DBLOAD procedure submits these statements to the DBMS and the changes are made in the DBMS table.

COMBINING DATA FROM TWO DBMS TABLES

Another task worth exploring is combining data from two DBMS tables. What makes this a more advanced topic is not the operation itself but doing it in an efficient manner. There are a number of ways to combine two DBMS tables in your SAS application. Ways that you could combine DBMS tables are as follows:

1. Use PROC SQL to join the two DBMS tables.
2. Use a DATA step to merge the DBMS tables.
3. Use the macro language to generate a WHERE statement from the first DBMS table to retrieve rows from the second DBMS table.
4. Create a view in the DBMS which joins the two tables.

The first three methods are described earlier in EXTRACTING DATA FROM DBMS TABLES USING KEY VALUES FROM A SAS DATA SET. The methods are implemented the same way except that the SAS data set is replaced with a second DBMS table. Here
are some guidelines for choosing which method to use:

PROC SQL join: When the tables to be combined are relatively small, this method is efficient and easy to code.

DATA step merge: When both tables are medium-sized, this method is fairly easy to code and allows the most flexibility for combining the tables using DATA step programming.

Macro-generated WHERE statement: This method is best when one table is small compared to the second table and the variable used to combine the tables is indexed in the larger table. This method is more difficult to code than the first two methods.

Create a DBMS view: This method is generally best when both tables are large. This method is not difficult to code but requires that a view be created in the DBMS. A further description of this method follows.

Implementing the DBMS View Method

The DBMS view method enables the programmer to use all the power of the DBMS to combine the DBMS tables. This method does not have the flexibility of DATA step programming. This method does have the ability to handle large tables. The first step in this method is to create a view in the DBMS which defines the way the tables are combined. This is done with the SQL CREATE VIEW statement.

Note that this SQL statement is submitted to the DBMS, not to PROC SQL. An example CREATE VIEW statement is as follows:

```
cREATE VIEW billings AS
    SELECT customers.contact,
           customers.name,
           invoice.amtbilled
    FROM customers, invoice
    WHERE customers.customer = invoice.billedto;
```

Once this view has been created, it is accessed from a SAS application just as if it were a regular DBMS table. Whenever the view is referenced, the DBMS combines the tables and presents the results to the SAS System and to your application.

USING SAS/ACCESS VIEWS IN SCREEN CONTROL LANGUAGE

There are many types of applications possible using Screen Control Language in SAS/AF and SAS/FP software that involve access to DBMS tables. Most of these applications are just the same as normal SAS programming. There are a few areas, though, that deserve special mention.

Looking of DBMS Tables When Editing

One area that SQL programmers must be careful of when working with SAS/ACCESS views is locking done by the DBMS when editing tables. Use of the FSEDIT or FSVIEW procedure causes at least an UPDATE intent lock on the rows being displayed the entire time the procedure is running. This will probably cause concurrency problems if multiple users are trying to access the table. The way to avoid this concurrency problem is to extract the data of interest into an actual SAS data file and do the editing on the copy. After the desired changes have been made, another procedure such as PROC SQL or PROC DBLOAD can be used to make the specified changes in the DBMS table. This minimizes the time that locks are held on the table to only that necessary to make the changes. The table will not be locked while the user types in changes. Additional logic will need to be put into the application to ensure that two people are not updating the same data at the same time. This logic can be accomplished by using some type of checkout system where the application flags other users with a particular row or subset of the DBMS table is currently being edited and is not available. A possibility of such a checkout system is described below. This system requires that a column be added to the DBMS table as flag that a particular row is being edited and the column identifies who is editing the row.

The checkout system described can be implemented as a SAS/AF application. The entire application can be implemented in one program screen that looks like the following:

```
COMPOUND EXEC SQL
SAS/ACCESS Interactive Edit Example
Editing DBMS table ORDERS

ENTER ORDER NUMBER TO BE EDITED: 

The source for the program screen is as follows:

```c
create view billings as
    select customers.contact,
           customers.name,
           invoice.amtbilled
    from customers, invoice
    where customers.customer = invoice.billedto;
```

This checkout system starts off with a PROC DBLOAD step that will use a key value supplied by the user and checks that row is available for editing. If the row is available, it flags it as unavailable to other users. The PROC DBLOAD step is as follows:

```
proc dbload dbname=your-dbms;
sql update orders set editid=userid
   where ordernum=value and editid is NULL;
```

Here the user supplies the order number for a particular order in the ORDERS DBMS table that he or she is interested in changing. The order number value is entered by the user in a field of a SAS/AF program screen. The WHERE clause of the SQL UPDATE statement checks to see if this row is available for searching for the ORDERNUM value and then checking if the EDITID column is NULL. The EDITID column is the column added to the table to flag which rows are available for editing. If the row is located and it is available for editing, then the SET clause sets the EDITID column to the user's
of the user who wished to edit the row. By changing this column to a non-null value, the row is now marked as unavailable to other users. The SQL reserved keyword USER is used to set the EDITID column's value to the user of the row who has the row checked out. This makes it easy to determine who has a particular row checked out, if that is needed.

The next part of the checkout system extracts the desired rows if they were available for editing. It is carried done with a DATA step that reads from a SAS/ACCESS view descriptor for the DBMS table. One such DATA step is as follows:

```sql
data suborder;
set viewlib.ordersvw;
/* VIEW: view descriptor */
/* includes WHERE EDITID=USER */
where ordernum=value;
editid='*';
```

Note that the view descriptor used here has a permanent WHERE clause that was specified when the view was created. The WHERE clause makes sure that the EDITID column is equal to the SQL reserved keyword USER. This ensures that the rows are extracted only if they have been checked out to this user. The WHERE statement in the DATA step specifies the key value for the rows to be extracted. This WHERE expression is passed through to the DBMS by the SAS/ACCESS engine. The DBMS can then use this expression to directly access the desired row if an index is available on the key. The EDITID variable in the extracted data set is set to missing. This is necessary for when the user changes are updated into the DBMS table. This requirement is explained later. If no rows are extracted, the DATA step is a signal that either the key value is not in the table or the rows have been checked out to another user.

Once the data have been extracted, they can be now be edited by the user. In this application the data are presented to the user with a PROC FSEDIT as follows:

```sql
call fsendit('suborder');
```

Note that the user is making changes to a local copy of the data. Because of this there are no locks being held by the DBMS at this point.

After the user has finished making any desired changes to the data, the new values must be updated into the DBMS table. At this point it is also necessary to flag that the rows are now available for other users to check out. In this example this is accomplished using the DBLOAD and APPEND procedures as follows:

```sql
proc dbload obms=your_obms;
sql delete from orders
where ordernum=value and editid=USER;
proc append base=viewlib.ordersvw
data=suborder;
```

Here PROC DBLOAD is first used to delete the rows from the table that the user has checked out. Then PROC APPEND is used to add in the rows from the user's local copy which contains the changes the user has made. Since the EDITID variable in the user's local copy was set to missing when the rows were extracted, these rows are available for other users to update as soon as the rows are added into the DBMS table. Note that using this method of deleting the old rows before adding the updated rows does present a small window of exposure to data loss. This method was chosen here because it is easy to code and does not violate any uniqueness constraints that have been placed on the table's keys. An alternate method for updating the new values into the table without having this window of exposure is described earlier in UPDATING A DBMS TABLE WITH VALUES FROM A SAS DATA SET.

Note that the application described will only guarantee data integrity if all applications that update the table have logic in them to check the EDITID column. Additional logic can be added to this application to remove this restriction. The additional logic would be added when the updates are made in the DBMS table. This logic would verify that the columns values had not changed since the data was extracted.

Editing Two DBMS Tables That Are Joined

Another area that comes up in Screen Control Language applications is the need to edit a PROC SQL view that is a join of two SAS/ACCESS view descriptors. Currently there is a restriction that a view created with PROC SQL cannot be edited. If your PROC SQL view is a join of DBMS tables, then editing the view is a vaguely defined operation anyway. For example, if your view joins the INVOICE table and the CUSTOMERS table as follows:

```sql
select invoice.invocenun,
invoice.amtbilled,
customers.name
from invoice, customers
where invoice.billedto=customers.customer;
```

It produces the following results:

<table>
<thead>
<tr>
<th>INVOICENUM</th>
<th>AMTBILLED</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>11273</td>
<td>252148</td>
<td>21st Century Materials</td>
</tr>
<tr>
<td>11278</td>
<td>1400625</td>
<td>University Biomedical</td>
</tr>
<tr>
<td>1286</td>
<td>12679156</td>
<td>Research Outfitters</td>
</tr>
<tr>
<td>12478</td>
<td>34891210</td>
<td>Instituto De Biologia</td>
</tr>
</tbody>
</table>

If you were able to go in and edit the NAME field, what change should be made in the underlying tables? Should the CUSTOMERS table be changed with the new name or should the INVOICE table be changed so the BILLEDTO field now points to a different customer name?

To solve this problem, it is necessary to further define the problem. In this example, you want to change the INVOICE table and when these changes are made you want to display the corresponding information from the CUSTOMERS table. Now that the problem is a little more defined, it can be solved using SQL in a PROC FSEDIT screen to retrieve the values from the CUSTOMERS table that correspond to the row that is currently being edited in the INVOICE table. PROC FSEDIT would be invoked on the SAS/ACCESS view for the INVOICE table. In the SQL program defined in the PROC FSEDIT screen, the SAS/ACCESS view for the CUSTOMERS table is opened. Computed variables are defined on the screen which displays the values from the CUSTOMERS table. Since these are defined on the screen as computed variables, they cannot be edited. The SQL WHERE function is invoked as each row of the INVOICE table is displayed. The WHERE function locates the correct row from the CUSTOMERS table. The screen for this application would look like this:
and the SCL code would be as follows:

```sas
create view authcust as
select name
from customers, authtab
where
  customers.country = authtab.country
and authtab.userid = USERID;
```

This view enables employees to read from the CUSTOMERS table but only for the users in the countries that they are authorized for. This view makes use of the reserved SQL keyword USER to identify the user making the query. AUTHTAB is a table that contains columns for userid and country. There is a row in the AUTHTAB table for each userid and country authorization. The AUTHTAB table would look like the following:

<table>
<thead>
<tr>
<th>USERID</th>
<th>COUNTRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Richard</td>
<td>USA</td>
</tr>
<tr>
<td>Carol</td>
<td>France</td>
</tr>
<tr>
<td>Leonard</td>
<td>USA</td>
</tr>
<tr>
<td>Leonard</td>
<td>Japan</td>
</tr>
</tbody>
</table>

All that is required to give a user access to the customers in a particular country is to add a row to the AUTHTAB table with that user ID and the country that is being authorized. All maintenance of authorizing users is done from this table.

This view can be defined with PROC SQL and any SAS user that accesses the CUSTOMERS table through this view only has access to the rows they are authorized for. Any users accessing the CUSTOMERS table using a different view would not be checked for authorization. Also users accessing the CUSTOMERS table from software other than the SAS System would not be checked. To be totally foolproof the view must be defined in the DBMS. Users are only given DBMS authorization to access the view, not the CUSTOMERS table directly.

**CONCLUSION**

SAS/ACCESS software has made it very easy for SAS programmers to accomplish basic tasks that access data stored in relational database management system tables. Eventually the SAS programmer might be faced with a task that requires more than the basic programming methods. This paper has attempted to cover the more common tasks that have surfaced since the release of Version 6 of the SAS System. Even if these techniques cannot be applied directly, it is hoped that they may give guidance or ideas to the SAS programmer to tackle the task at hand.

**REFERENCES**

Jake Jacobs' "Writing Efficient SAS/DB2® Applications in Version 6 of the SAS System" in Proceeding of the Fourteenth Annual SAS Users Group International Conference

Howard Priestman's "Performance Considerations for the Database Engine" in Proceeding of the Fourteenth Annual SAS Users Group International Conference

James E. Johnstone and Craig Ray's "A Survey of Table Lookup Techniques for the SAS System" in Proceeding of the Fourteenth Annual SAS Users Group International Conference