MODIFY®
The Most Under-Appreciated of the Data Step File Handling Statements

Abstract

The MODIFY statement has been part of the SAS® set of tools for years, yet it is rarely used. Good programmers have written very complicated code to accomplish that what be done very readily using this statement. It allows changing of a SAS dataset in place using a DATA Step, without creating a new or replacement dataset. If used correctly, it is faster and more straight-forward than alternatives techniques using SQL. As a result it is indispensable when working in a relational database, be it implemented in SAS or another DBMS. This paper will discuss some of its syntax and uses, focusing on its use as a batch transaction processing tool. It will also detail some of the SAS Access components and options that can enhance its functionality against a non-SAS DBMS.

An Introduction to MODIFY

MODIFY is a file-handling statement like SET, MERGE, and UPDATE. These statements read input datasets into the DATA step data vector. Datasets are any data type members of a SAS library. They can be SAS datasets, tables accessed using ODBC, or those connected using the other SAS/ACCESS engines. However, while this is true in theory, some of the Access Engines support only parts of functionally required by MODIFY type processing.

What is unique about MODIFY is that it causes the DATA step to not replace, recreate, or delete the original dataset! In all other uses of the DATA step, the datasets listed in the DATA statement are replaced. If one attempts to change an existing dataset using the DATA step as in the following example:

```
data MyTable;
  set MyTable;
  CurrentDate = date();
run;
```

SAS will read each dataset record into the data vector, make the coded changes, and write each resulting record to a temporary dataset. Upon successfully reading, modifying, and writing all of the records, the original dataset is replaced with the temporary dataset. In many cases, this is the most efficient way this can be done. It however is a severe limitation to those who are processing against datasets that can not be over-written. The main causes of this limitation are relational integrity constraints and the lack of the privileges needed to create a table in the RDBMS in which the dataset is stored.

The following DATA step using MODIFY on the other hand does not replace the original dataset.

```
data MyTable;
  modify MyTable;
  CurrentDate = date();
run;
```

In this example each record from MyTable will be read into the DATA step vector, the coded changes to the record will be made, but then rather than write the results to a

temporary dataset, SAS modifies the record directly in the MyTable dataset. If MyTable is a SAS dataset, these changes are immediate. If MyTable is not a SAS dataset, SAS must do different things based on the nature of the SAS ACCESS engine being used. This can mean calls to ODBC, OLE DB, or the native database API. The net result is the functional equivalent to a SQL update to each individual row. When writing to an RDBMS such as Oracle®, MySQL®, Access®, or Microsoft® SQL Server®, the changes may not be instantaneous. Once again, depending on the engine, there may be Rollback maintained so that all changes will be undone if an error occurs. As you will see later, this not only applies to updated records, but to added and deleted records as well.

Since MODIFY does not replace the dataset, it can be used to update datasets or tables with primary, unique, and foreign keys. It can also make updates to tables in which the user has only row update privileges. Often when a programmer is presented with this problem they choose to use PROC SQL. This is of course a very viable solution if the updates are simple enough that they can be coded in a SQL UPDATE statement. There are however many situations which can benefit greatly from the added power of DATA step process. This power can be used to handle errors encountered during processing, process based on record order, retain temporary values between records, make several changes to a record in a single pass through the data, output to separate datasets, and make use of the powerful functions available within SAS.

There are four documented forms for using the MODIFY Statement. They will each be discussed individually. They are:

MODIFY alone:
A sequential pass through a dataset
MODIFY with a BY Statement:
An ordered match merge
MODIFY with a KEY= Option and a SET Statement:
An individual index based record lookup
MODIFY with a POINT= Statement:
A record number based lookup

In exploring the use of these syntax forms, we will be using the simple sample dataset of clients and their addresses.

Client Number	Address	City	State	ZIP	AddressStartDate	Address EndDate	Current Address	latitude	longitude
716	6009 Silver Oaks Ct SE	Tumwater	WA	98501	01JUN2004:00:00:00	.		.	.
716	2991 Lookout Drive NW	Olympia	WA	98502	01DEC1995:00:00:00	.		.	.
716	144411 Martinson Rd SE	Yelm	WA	98597	01DEC1990:00:00:00	.		.	.
121	215 Legion SW	Olympia	WA	98502	01JUN2004:00:00:00	.		.	.
1221	101 Stewart St	Seattle	WA	98101	01JUN2004:00:00:00	.		.	.
721	1320 Broadway Plaza	Tacoma	WA	98402	01JUN2004:00:00:00	.		.	.
251	585 Liberty Avenue	Salem	OR	98501	01JUN2004:00:00:00	.		.	.
1222	900 SW Fifth Street	Portland	OR	98501	01JUN2004:00:00:00	.		.	.
1	11th Ave. and Columbia	Olympia	WA	98504	01JUN2004:00:00:00	.		.	.
35	1730 Minor Ave.	Seattle	WA	98501	01JUN2004:00:00:00	.		.	.
225	100 SAS Campus Drive	Cary	NC	27513	01JUN2004:00:00:00	.		.	.
104	One Microsoft Way	Redmond	WA	98052	01JUN2004:00:00:00	.		.	.
191	380 New York Street	Redlands	CA	92373	01JUN2004:00:00:00	.		.	.
212	500 Summer St. NE	Salem	OR	97301	01JUN2004:00:00:00	.		.	.
313	521 Wall St.	Seattle	WA	98121	01JUN2004:00:00:00	.		.	.
414	909 A Street	Tacoma	WA	98402	01JUN2004:00:00:00	.		.	.
515	1100 Fairview Avenue North	Seattle	WA	98109	01JUN2004:00:00:00	.		.	.

This is a hypothetical list of addresses for a database of clients. The primary keys of this dataset are ClientNumber and AddressStartDate. In this example all of our clients have a single address except client number 716 which has three, each of which start on different dates.

MODIFY Alone

This syntax form allows us to make a single pass through the records in a dataset, while making changes to those records. The subset of records that are read can be changed using a WHERE clause in the Modify statement. The order the records are read will be the sorted order if reading a SAS dataset, or the order of the datasets primary key if reading from a non-SAS dataset. If no primary key is present, no order may be assumed. Since there is no BY statement, BY group ("FIRST." and "LAST.") type processing is not allowed. Unfortunately, you cannot use a BY statement with a MODIFY statement that has a single dataset. Attempting to do so will return an error to that effect. I will show a trick to get around this later in the paper.

The following example will update all of the records in our address dataset that have a start date of June 1st 2004.

```
data Clients.Addresses;
  modify Clients.Addresses(where =
    (AddressStartDate >= DHMS('01-JUN-2004'd,0,0,0)));
  CurrentAddress='T';
  NewVar1 = 'This wont be kept';
run;
```

In this example the dataset is stored in a Microsoft Access database, so the date values needed to be converted to a date/time type before the comparison could be made. This explains the use of the DHMS function. The `CurrentAddress='T';` line refers to a variable that already exists in the master dataset. This code will change the value of that variable in the master table to the value "T". On the other hand the NewVar1 variable does not exist in the master table. The line changing its value is in effect creating a temporary variable. This variable will not be created in the master table. Any changes made to it will be lost.

This same result can be accomplished just as easily in PROC SQL using an UPDATE query of the following form. Most uses of the MODIFY statement alone can be just as easily be done using SQL, so the value of this syntax form is marginal.

```
proc sql;
  update Clients.Addresses;
  set CurrentAddress='T'
  where AddressStartDate >= DHMS('01-JUN-2004'd,0,0,0);
run;
```

Speed Testing

The earlier example begs the question "Which is faster MODIFY or UPDATE." It turns out much of that depends on the nature of the query and the SAS ACCESS engine being used. I ran the following functionally equivalent step to test this.

```
data TestLib.MyEmptyTable;
  modify TestLib.MyEmptyTable;
  Var1 = ranuni(Var4);
run;
```

And...

```
proc sql;
  update TestLib.MyEmptyTable
  set Var1 = ranuni(Var4);
quit;
```

First I tested them against 100,000 records in a Microsoft Access database using a SAS Library connected using ODBC. I then tested them against 1,000,000 records in standard SAS datasets. Between runs I completely deleted and recreated the database and/or tables to be sure that all runs had the same environment. The use of a SAS only function such as RANUNI with a parameter from within the dataset forces SAS to process the query rather than just passing the SQL straight to Access. The results were interesting.

Dataset	MODIFY	SQL
100,000 Rows in Access Tables	11.5 Seconds	8.5 Seconds
1,000,000 Rows in SAS Datasets	4.9 Seconds	9.0 Seconds

It appears that when processing SAS datasets MODIFY is almost twice as fast as SQL. However when used against a remote RDBMS MODIFY is significantly slower. In situations where processing time is critical, both of these methods should be tested to see which performs the best. When using SAS ACCESS engines, a helpful tool is the following SAS option.

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

This option writes all activity passing through the SAS/ACCESS engines to the log. This activity may be SQL queries, OLE DB calls, or any other communication that SAS is using to perform the tasks required. There are different values allowed for the value of SASTRACE, so see the documentation for the settings appropriate to your needs.

MODIFY with a BY Statement

When you start using MODIFY with a BY statement its real power is revealed. This works a little like a MERGE statement in that is combines two datasets doing ordered match merge processing. Typically you have an existing master dataset and a transaction dataset containing new and/or updated records for that dataset. This syntax form allows BY group processing. It also allows updating existing records, deleting unwanted records, and creating new ones all in the same pass through the data depending. The action taken can be dependent on whether or not you get a match. You can even send any records desired to another dataset.

There are three actions statements that can be used on each record. They are:

OUTPUT – Creates a new record in the master dataset, or an alternate output dataset.
REPLACE – Updates the currently matched record in the master dataset
REMOVE – Deletes the currently matched record in the master dataset

The REPLACE and REMOVE actions are only make sense if the incoming transaction record matches a record in the master dataset. For this reason they can only be used to write to the master dataset, not an alternate output dataset. The OUTPUT action is typically only used to write a record that did not match to the master dataset but it can be used to create duplicate records as well. It is also used to write records to alternate output datasets, as we will see later. If no Action is specified in a DATA step, REPLACE is assumed for matched records. An ERROR condition is returned for unmatched records.

Here is an example we could use to update our Addresses dataset with a list of new address. In this example the new address are stored in the temporary dataset NewClientAddresses.

```
data Clients.Addresses;
  modify Clients.Addresses NewClientAddresses;
  by ClientNumber AddressStartDate;
  select (_IORC_);
    when (%SYSRC(_SOK)) do;
      replace;
    end;
    when (%SYSRC(_DSENMR)) do;
      output;
      _error_=0;
    end;
    when (%SYSRC(_DSEMTR)) do;
      put 'ERR' 'OR: Duplicate Values on transaction dataset';
      stop;
    end;
    otherwise do;
      put 'ERR' 'OR: Unknown IO ';
      stop;
    end;
  end;
run;
```

Note that the master dataset is listed in the DATA statement and both the master and transaction dataset are listed in the MODIFY statement. The BY statement lists the variable on which the matching will occur. This is similar a MERGE statement but don't rely on this similarity. Both the master and transaction dataset must be either sorted by the BY variables, or they must by indexed by those variables. Unlike MERGE, only the rows from the transaction dataset are processed. Rows from the master dataset that are not in the transaction dataset are not loaded into the data vector.

First I must give you a word of warning if you will be attempting this against a non-SAS dataset. The standard LIBNAME statements for pointing at ODBC, OLE DB, or other SAS ACCESS engines, do not have the correct defaults for this type of processing. The following is a correctly formated library for a Microsoft Access database.

```
libname Clients odbc
        noprompt="driver={Microsoft Access Driver (*.mdb)};
        DBQ=c:\temp\ModifyTemp\Clients.mdb"
        reread_exposure=yes DBINDEX=YES UPDATE_LOCK_TYPE= NOLOCK;
```

Those options in the last line are needed to give SAS the access levels it needs. The READ_EXPOSURE option gives SAS the ability to do individual record reads and writes. The DBINDEX option allows SAS to directly reference the indexes in the remote DBMS. Honestly, I am not sure why the UPDATE_LOCK_TYPE option is needed to disable record level locking, but I could not get this to work without that option as well.

It most cases, the BY variables are the primary or a unique key of the master dataset. While that is not required, it is highly recommended. If records with duplicate values for the BY variables are found in the master dataset, only the first record is matched. None of the other records will be matched regardless of how many records with those values are found in the transaction dataset. If transaction dataset has records with duplicates of the BY variables, the last record read will over-write the earlier values.

As you can see in the example, this form of MODIFY requires the use of the _IORC_ or "Input/Output Return Code" automatic variable. This variable actually exists in any DATA step but is rarely used outside MODIFY processing. It contains a numeric value which indicates the status of the last input/output action performed. While this value is numeric, SAS provides the macro function %SYSRC that resolves a set of mnemonic codes to their equivalent numeric values. SAS recommends that these values be tested using %SYSRC in case the numeric return codes change in the future (However, SAS breaks this rule in all of its examples and tests for the value 0 instead of using _SOK). In this example we are testing for the codes returned by the matching of the master and transaction dataset. There are three possible values at this point in the process:

_SOK	The source record matched a record in the master dataset **(0)**
_DSENMR	The source record did not match a record in the master dataset in BY processing. **(1230013)**
_DSEMTR	The source record did not match a record in the master dataset and it is not the first record with that value of the by variables in BY processing. **(1230014)**

In the previous example we first test the _IORC_ value to see if equals the value of _SOK. This indicates that a match was found. If so, the values in the data vector that were loaded from the master dataset will have already been over-written by the values from the transaction dataset. We simply call the REPLACE action to write these updates to the master dataset. If _IORC_ instead equals the value of _DSENMR, we know that the record did not match and use OUTPUT to create a new record in the master dataset. Finally if equal the value of _DSEMTR we known the transaction dataset has duplicates on the BY variable. In this example this is treated as an error, but other actions could also be taken. This example returns an error because the BY variables are a unique key of the maser dataset. If we tried to write the second record after the first has already been written, the unique key of the table would be violated. The record would then be rejected with and uncontrolled error.

As mentioned earlier, if the transaction record does not match the master dataset and we get a _IORC_ value other than 0, this is technically an error condition. These errors will be written to the log, but they will not stop processing. If many transactions are being added, this can add unwanted clutter to the log. For this reason the line `_error_=0` is added to the code. This clears the error before the end of the data vector loop, and SAS does not write it to the log.

MODIFY with a KEY = Option and SET Statement

Functionally this form of MODIFY is virtually the same as using a BY statement. What differs is how the matching to the master dataset is done. Rather than doing a sorted match merge, this form does individual lookups for each transaction record. The lookup is accomplished using an index on the master dataset. For this reason MODIFY does not require that the transaction table be sorted on the BY variables. It is also very efficient for transactions dataset that are small in relation to the master dataset as only the needed records are read, not the entire master table. The master dataset must have an index on the BY variables. Sorting alone is not sufficient. This form also tends to require a more advanced SAS/ACCESS engine. Here is an example that is functionally the same as the previous example.

```
data Clients.Addresses;
  set NewClientAddresses (rename = (address = address_in
                                    city = city_in
                                   state = state_in
                                     zip = zip_in));
  modify Clients.Addresses key=AddressesPK;

  address = address_in;
  city = city_in;
  state = state_in;
  zip = zip_in;
  select (_IORC_);
    when (%SYSRC(_SOK)) do;
      replace;
    end;
    when (%SYSRC(_DSENOM)) do;
      output;
      _error_=0;
    end;
    otherwise do;
      put 'ERR' 'OR: Unknown IO ';
      stop;
    end;
  end;
run;
```

There are a few things that you should notice. First is that the SET statement must occur before the MODIFY statement. This is because the new record must be read into the data vector before the matching key values can be looked up in the master dataset. As a result, if a matching record is found and read into the vector, its values will over-write any existing values loaded from the transaction table. For this reason all of the variables in the transaction dataset are renamed. Then all of the values are written back to their original variables after the MODIFY statement has executed. Some efficiency can be gained here by checking to see if there are any actual changes. Code can be written to test this and only perform a REPLACE if any are found.

Second, notice that the key variables are not mentioned by name at all. They are determined by the variables that make up the index to which the KEY= option points. This requires that you known the name of the index needed. This can also be accomplished by using the DBKEY option to declare the key variable of the master table. The following modify statement is equivalent that of the previous example.

```
modify Clients.Addresses(dbkey = (ClientNumber AddressStartDate)) key=dbkey;
```

This form is useful if you don't know the name of the index you need. It looks up the correct index based on the passed variables names. This is necessary in some ACCESS engines as the name of the index does not appear to be passed correctly if it listed explicitly. This form has worked for me in this situation.

Third, notice that we have introduced a new _IORC_ value.

_ **DSENOM**	The transaction record did not matched a record in the master dataset in KEY= and POINT= processing. **(1230015)**

This acts much the same as _DSENMR, only it applies to KEY= and POINT= processing only. There is no equivalent to _DSEMTR in KEY= processing so you must code your own ways of catching these situation if needed.

In there are multiple records in the transaction dataset with the same key values, special effort must be taken. This is especially true if a matching record is not found in the master dataset. To handle this, the UNIQUE option must be included on the MODIFY statement. By default the MODIFY statement will not re-query if consecutive records with the same key values are found. This can be a problem if the first record is new. In this case the first record will come up as not matched so the code will write out a new record. The subsequent record will still come up as not being matched. When the code tries to write this record, the unique key of the table is likely to be violated. The UNIQUE option forces the match to be re-queried on every incoming record.

MODIFY with a POINT = Option and SET Statement

This form of MODIFY is similar to KEY= processing in most ways, except instead of looking up rows using an index, the records are looked up by the row number of the master table. The row number must have been included in the transaction dataset as an extra variable. This make for very fast processing. The following example updates the latitude and longitude of our address table.

```
data Clients.Addresses;
  set AddressesGeocoded(rename = (latitude  = lat_in
                                  longitude =long_in));
  modify Clients.Addresses point=RecordNumber;
  if _iorc_=%sysrc(_SOK) then do;
    latitude  = lat_in;
    longitude = long_in;
    replace;
  end;
  else do;
    put "ERROR: Invalid record number: " RecordNumber;
    stop;
  end;
run;
```

The transaction dataset used in this example was extracted from the master table and the value of the _N_ automatic variable was stored in a variable named RecordNumber. The table has no other identifying columns. Notice that just like the KEY= processing, the incoming variables must be renamed and the master tables values must be updated with code. It makes no sense to do this type of processing were there are records that don't match. For this reason anything except a match is reported as an error and processing is stopped. The SAS documentation strongly recommends including a stop statement for un-matched records, as not doing so can result in infinite loops.

Despite the potential speed improvements this form may offer, I find it to be very dangerous and personally never use it. The same thing can be accomplished using key values, without the risk. If any records are added or deleted between the extraction of your dataset and the MODIFY step, the wrong records will be updated. Also, most RDBMS do not have a permanent record number concept that is accessible through SAS, so it is of limited value. If you do choose to use this form, I recommend leaving the unique key values in the transaction dataset. This way you can write code to double check the keys of the transaction to those of the matched row. This should alleviate some of the risk.

MODIFY with a KEY = Option and INFILE Statement

This can be considered an undocumented fifth form of the MODIFY statement. It work much the same as MODIFY with the KEY= option and a SET statement, only now the INFILE statement supplies the incoming records instead of the SET statement. This allows one to load data into a dataset directly from a text file. Using MODIFY in a DATA step however, you can accomplish the entire extract, transform, and load process in a single step. Using INFILE you can read the data in to the data vector. Once in, you can do all kinds of transformations and cleaning of the data. When the data is ready to be loaded, you can capture errors in the loading process and handle them appropriately. The next section of this paper will describe this type of error handling in more detail. For now, here is an example of this syntax form that loads data from a text file directly into an empty table.

```
data Clients.Addresses;
  modify Clients.Addresses key=AddressesPK;
  infile "E:\PNWSUG\Papers\Modify\ClientAddresses.txt"
         dsd dlm=",";
  input ClientNumber  Address City State ZIP
        AddressDateIn date11.;
  AddressStartDate = DHMS(AddressDateIn,0,0,0);
  output;
  _error_=0;
run;
```

This is actually the code I used to initially load the Addresses dataset we have been using. As mentioned earlier, the INFILE statement is use in place of the SET statement. In this example we are assuming an empty destination table. This is a limitation of this form. Note that we are able to transform the date value into a date time value before it is output to the master table. This is just an example of what could be done using formats, SAS functions, SAS hash lookups, or even inline queries using the OPEN and FETCH functions. The possibilities are great and in my opinion surpass the capabilities of most native database load facilities.

Capturing and Handling Constraint Violations

This is where MODIFY really stand apart from other options for loading data into relational tables. Constraints are used to enforce many of the business rules of a database. They come in many forms from basic primary, unique, or foreign keys, to the advanced trigger logic available in many RDBMS. When a record is added, or an existing record is updated in such a way that it violates one of these keys, the DBMS returns an error condition and rejects the transaction. If you are attempting to load data using traditional SQL, these errors force a rollback of all of the records already posted. If you need more control than this, the data is often loaded into a temporary table, and DBMS native cursor logic is used to work around the problem. This code can be quite complex. MODIFY provides a very good alternative to this. The following is a simple example of how errors can be captured and the offending records written out to a dataset of rejected records.

```
data ClientsS.Addresses
     RejectedRecords;
  modify ClientsS.Addresses NewClientAddresses;
  by ClientNumber AddressStartDate;
  select (_IORC_);
    when (%SYSRC(_SOK)) do;
      replace ClientsS.Addresses;
      if _IORC_ NE %SYSRC(_SOK) then do;
         ReturnCode = _IORC_;
         ErrorMessage = IORCMSG();
        output RejectedRecords;
         put 'WARNING: Rejected Record On Replace! ' ErrorMessage;
        _error_=0;
      end;
    end;
    when (%SYSRC(_DSENMR)) do;
      output ClientsS.Addresses;
      if _IORC_ NE %SYSRC(_SOK) then do;
        ReturnCode = _IORC_;
        ErrorMessage = IORCMSG();
        output RejectedRecords;
        put 'WARNING: Rejected Record On Output! ' ErrorMessage;
      end;
     _error_=0;
    end;
    when (%SYSRC(_DSEMTR)) do;
      put 'ERR' 'OR: Duplicate Values on the transaction table';
      stop;
    end;
    otherwise do;
      put 'ERR' 'OR: Unknown IO ';
      stop;
    end;
  end;
run;
```

This DATA step is nearly identical to the earlier example of MODIFY with a BY statement. The only changes are the addition of an alternate output dataset named RejectedRecords and an IF block that appears after the OUTPUT and REPLACE statements. Either of these two actions could result in an attempt to apply a change to the dataset that would violate its constraints. As mentioned earlier, the _IORC_ variable represents the result of the last output action as well as the input actions. By testing its value after the attempted output action, we can capture any error conditions. In this example, any result other than success causes the record to be written to the RejectedRecords table and a warning to be written to the log. The value of the _IORC_ variable is captured as well as the value returned by the IORCMSG() function. This function returns a human readable version of the error condition. By writing the rejected record to a dataset along with the error code and message, it can later be reviewed and the errors corrected. An attempt could then be made to load it into the database at a later time. In the mean time, a rollback is not triggered and the DATA step continues processing the rest of the records.

There are other actions that could be taken based on specific values of the _IORC_ variable. Here are some values that this variable can take as the result of an output action.

_SENOCHN The OUTPUT or REPLACE action violated a Unique Key. **(630058)**

_SEICAU	Add or Update failed for data set because data value(s) do not comply with integrity constraints. **(660130)**
_SENMCH	Observation was not added or updated because no match was found for the foreign key value. **(630188)**
_SEINTG	Add or Update failed because data value(s) do not comply with an integrity constraint that utilizes index for corresponding file. **(630025)**

The first of these values can be found in the documentation. It turns out that the code for the SYSRC macro function can be found in C:\Program Files\SAS\SAS 9.1\core\sasmacro\sysrc.sas. This is where I found the other three values listed. There are actually hundreds of these, but I have only found uses for these. For example, you could use the knowledge that a foreign key was broken obtained from the return code _SENMCH to write code that inserted the missing value into the foreign key table, and attempted to re-output the record. Many other possibilities exist as well, making this a very powerful tool.

I must give you another word of warning here. This example will not work against a Microsoft Access database. I have had very good success using it with SAS datasets and Oracle database tables, and I expect that it will work in others. This code is supposed to capture the errors and work around them, and this usually works fine. The MS Access errors however, cause the SAS DATA step loop to stop. This prevents the processing of the remainder of the records. I suspect that this has something to do with Microsoft using OS level errors in Access and that those errors are picked up by SAS as OS errors, not DBMS errors. That is pure speculation however.

Merging a Master Dataset on itself to get BY processing

This is a bit of a non-standard trick I have developed to enable BY group processing to a pass trough a master table. This type of processing is very difficult thing to do using SQL but can be done very quickly with this trick.

```
data Clients.Addresses;
  modify Clients.Addresses Clients.Addresses ;
  by ClientNumber descending AddressStartDate;
  retain LastDate;
  if first.ClientNumber then CurrentAddress = 'T';
  else do;
    AddressEndDate = LastDate;
    CurrentAddress = 'F';
  end;
  replace;
  LastDate = AddressStartDate;
run;
```

In order to get all of the records from the master table to pass through the data vector, and have a BY statement to establish BY group processing, I used the master dataset as both the master and transaction dataset and the BY statement joins them on the values by which I want the records sorted. In this case the desired order is ClientNumber and descending AddressStartDate. This passes through all the address for each client in order and returns the most recent address first followed by any earlier addresses. This way I can mark the first address found for a client as the most recent. I can also create a temporary retained variable which contains the start date of the previous record. In this

way, I can set the end date of an address to equal the start date of the next newer address. This turns the records into address spans.

If it's so good, why isn't everybody using it?

As you can see the concepts of the MODIFY statement are quite a bit different than many of the other SAS concepts. For this reason and others it can be hard to understand. This I think makes it a bit daunting to many. The error messages that occur can also be just as much of a barrier to its use. Working around these can take a fair amount of knowledge or perseverance. Particularly difficult is getting the correct options set up on the SAS ACCESS statements. The errors resulting form getting these incorrect erroneously appear to preclude the use of MODIFY against that engine entirely. This would turn away many programmers, as it did me the first few times I encountered them. Another limitation is that, as I mentioned earlier, not all of the features work with all of the SAS/ACCESS engines. I have found however that using the engine specific to the DBMS as opposed to using ODBC or OLE/DB can give better results. There are of course many programmers that never work with remote and/or relational tables, and therefore have little use for this tool. I suspect that there are also situations where this could be helpful in modifying large dataset with limited resource. And finally, there are situations were MODIFY can be slower than SQL alternatives. This should be tested if performance is critical.

Conclusion

If you are using SAS to maintain relational tables, you should be using MODIFY. If you are using tools other than SAS to do this type of processing, you should consider SAS, as MODIFY makes it a very powerful tool for doing this. With a little trial and error, and the correct SAS/ACCESS modules, I know of no better tool for doing this type of processing with a minimal investment in programming.

Biography

Curtis currently works for Looking Glass Analytics as a SAS consultant and GIS service manager. Prior to that, he worked for the Washington State Department of Social & Health Services Division of Research and Data Analysis, and the US Census Bureau. He has worked extensively with SAS for fifteen years. He has expertise in Geographic Information Systems, database design/programming particularly using Oracle and PL/SQL, and in demographic analysis. He holds a bachelors degree in Geography from the University of Washington.

Curtis Mack
Looking Glass Analytics
Curtis.Mack@lgan.com
www.LGAN.com

Acknowledgements