Paper SAS2975-2019

# Comparing Methods for Dynamically Updating Data: The MODIFY Statement, Temporary Arrays, and Hash Objects

Richard D. Langston

## ABSTRACT

This paper explains the implementation of an account voting mechanism that enables dynamic updating of data. Three implementation methods are compared: using the MODIFY statement, using temporary arrays, and using hash objects. This paper also explains how the three methods impact performance.

## INTRODUCTION

I was contacted by SAS Technical Support to help an inquiring user with this unique problem. It turned out to be a very good example to show various methods to solve the problem, and I had the opportunity to compare the performance of these methods.

The data set consists of pairs of policies and holders. The order is significant. The data set is ordered by policy and within policy by holders. The first holder encountered for a policy that has not voted gets to vote on that policy. Once a policy has been voted on, no one else can vote for that policy. And once a holder has voted, that holder can no longer vote for anything else. It is possible for the same holder to appear multiple times for the same policy, and it is also possible for the same holder to appear for different policies. It is possible that a policy might not receive a vote and that a holder might not get to vote, as shown in the following table:

| Policy | Holder | Action |
|---|---|---|
| A | 1 | Holder 1 votes on policy A. |
| A | 2 | Policy A was already voted on, so no action is taken. |
| B | 2 | Holder 2 votes on policy B. |
| B | 2 | Policy B was already voted on, so no action is taken. |
| C | 1 | Holder 1 already voted, so there are no votes for policy C. |
| D | 7 | Holder 7 votes on policy D. |
| D | 3 | Policy D was already voted on, so no action is taken. |
| E | 1 | Holder 1 already voted, so no action is taken. |
| E | 8 | Holder 8 votes on policy E. |
| F | 2 | Holder 2 already voted, so no action is taken. |
| F | 7 | Holder 7 already voted, so there are no votes for policy F. |

**Table 1. Examples of Policy and Holder Actions**

This data cannot be updated by merging because the status of the holders or policies changes as the voting takes place. So, updating can be performed by one of these methods: using a MODIFY statement or a REPLACE statement with indexing, using an array holding the voting status, or using a hash object.

The user's data has around 4 million observations, so my approach was to test with a small amount of data first and then scale up with random data to ensure that I still had good performance.

## OUR SAMPLE DATA

Here we create a small data set based on the preceding table:

```
data orig;
     length policy holder $32;
     input policy $ holder $;
     datalines;
A 1
A 2
B 2
B 2
C 1
D 7
D 3
E 1
E 8
F 2
F 7
run;
```

## METHOD 1: USING THE MODIFY STATEMENT OR THE REPLACE STATEMENT

Our first approach is to have a separate indexed data set of just the holders. The observations will be updated via the MODIFY statement when the holder gets to vote, and subsequent accessing of the holder's observation will indicate that the holder has already voted.

We first create the holder data set from our input data, using the NODUPKEY option to keep only the unique values. We add a placeholder variable, policy_voted_on, to hold the vote status, and then we create our index based on the holder variable, as shown in the following example:

```
proc sort data=orig(keep=holder)
          out=unique_holder nodupkey;
     by holder;
     run;
data unique_holder; set unique_holder;
     policy_voted_on=' ';
     run;
proc datasets library=work;
     modify unique_holder;
     create index holder;
     quit;
```

Although not necessary for a small data set, my testing showed that the method using MODIFY/REPLACE statements takes too much execution time when run with the large data set. This was because of the swapping of buffers with the random access of data. This issue was resolved by using the SASFILE statement to ensure that the entire data set resides in memory:

```
sasfile unique_holder load;
```

Here is the code for using the MODIFY/REPLACE statements. Note that we do not need the unique_holder data set after this step is complete. However, the DATA step syntax requires that the unique_holder data set appear in a DATA statement if it also appears in a MODIFY statement. Also note that the results, which show the policy and which holder (if any) voted, are stored in a temporary file indicated by the RESULTS1 fileref. The results files for all the methods should match and correspond to Table 1:

```
filename results1 temp;
data unique_holder;
     length voter $32;
     retain voter;
     set orig; by policy notsorted;
     if first.policy then voter='NONE';
     modify unique_holder key=holder / unique;
     if voter='NONE' and policy_voted_on=' ' then do;
        policy_voted_on='Y';
        voter=holder;
        replace;
        end;
     file results1;
     if last.policy then put policy= voter=;
     run;
sasfile unique_holder close;
```

## METHOD 2: USING A _TEMPORARY_ ARRAY

For this method, a _temporary_ array contains the status for each holder. But because the holder numbers are not necessarily 1-based (for example, our test data could have holder numbers of 100, 1000, and 10000, among others), we must create a new variable, holder_order, that is indeed 1-based. And we must determine the number of unique holder numbers so that we can declare our array properly. Here is our code for setting up method 2:

```
%global n_holders;
data temp; set orig;
     seqnum+1;
     run;
proc sort data=temp; by holder; run;
data temp; set temp end=eof; by holder;
     if first.holder then holder_order+1;
     if eof
        then call symputx('n_holders',holder_order);
     run;
proc sort data=temp; by seqnum; run;
```

We must first add seqnum to our data set so that we can sort back to the original order. Then we can temporarily sort by the holder ID, adding the new 1-based holder_order value. Once we arrive at our last observation, the N_HOLDERS macro variable will contain the number of unique holders, so we can declare our _temporary_ array with the right number of elements.

Now we can do the voting with this code:

```
filename results2 temp;
```

```
data _null_; set temp end=eof; by policy notsorted;
     array holder_votes{&n_holders} $1 _temporary_;
     length voter $32;
     retain voter;
     if first.policy then do;
        voter='NONE';
        end;
     if voter='NONE' and holder_votes{holder_order}=' ' then do;
        holder_votes{holder_order}='Y';
        voter=holder;
        end;
     if last.policy;
     file results2;
     put policy= voter=;
     run;
```

We have our _TEMPORARY_ array with the right element count, and we use holder_order as our index into the array. And if the array element is blank, this means that the holder has not yet voted, and the value can be set to Y. And as in method 1, we write out the value of policy and voter to an output file, which in this case is fileref RESULTS2.

## METHOD 3: USING A HASH OBJECT

We start out the same as method 1, creating a data set containing just the unique holders. We need a different name than 'holder’ so that it won't interfere with the original 'holder' variable. And we create the placeholder 'voted' variable because its value in the hash object will be updated. Here is the code for preparing the additional input data set for the hash object:

```
proc sort data=orig(keep=holder)
          out=holders nodupkey; by holder;
data holders; set holders;
     voted=' ';
     rename holder=policy_holder;
     run;
```

Now we can go through the policy data set and use the hash object to hold the voting status for each holder:

```
filename results3 temp;
data _null_; set orig; by policy notsorted;
     if _n_=1 then do;
        declare hash h(dataset: "work.holders");
        h.defineKey('policy_holder');
        h.defineData('voted');
        h.defineDone();
        end;
     length voter $32;
     retain voter;
```

```
        if first.policy then voter='NONE';
        if voter='NONE' then do;
           policy_holder=holder;
           rc = h.find();
           if voted=' ' then do;
              voter=policy_holder;
              voted='Y';
              h.replace();
              end;
           end;
        if last.policy;
        file results3;
        put policy= voter=;
        run;
```

We set up the hash table during the first pass of the data set. The key is the renamed variable policy_holder. That is set from holder and searched for via the find() method. Note that we don't check the return code from the find() method, since the holder will always be found. If the voted variable is blank, then we update it with the replace() method. And as for the other approaches, we record the results for later comparison.

## METHOD 4: USING A HASH OBJECT WITH DYNAMIC ADDS

This variant of method 3 also uses a hash object, without preprocessing the data into a separate data set. The difference is that the add method is called for every input observation, even though the add method will fail if the policy_holder value has already been seen. It is not necessary for the hash object to contain policy_holders not yet seen, so the hash object grows only as new policy_holder values are added. Otherwise, the code looks very similar to method 3 above:

```
   filename results4 temp;
   data _null_;
        length voted $1;
        if _n_=1 then do;
           declare hash h();
           h.defineKey('policy_holder');
           h.defineData('voted');
           h.defineDone();
           end;
        set orig; by policy notsorted;
        voted=' '; rc = h.add();
        length voter $32;
        retain voter;
        if first.policy then voter='NONE';
        if voter='NONE' then do;
           policy_holder=holder;
           rc = h.find();
           if voted=' ' then do;
              voter=policy_holder;
              voted='Y';
```

```
               h.replace();
               end;
            end;
         if last.policy;
         file results4;
         put policy= voter=;
         run;
```

## RESULTS

When run with the test data, all four of the results files showed these values:

```
policy=A voter=1
policy=B voter=2
policy=C voter=NONE
policy=D voter=7
policy=E voter=8
policy=F voter=NONE
```

And this matches with the expected results that were described.

## USING WITH LARGER DATA

The actual data sets that the user needed to process had around 4 million observations. To determine the best performer among the methods using large data sets, we used this code to produce random data:

```
   %let n_policies=4e6;
   %let n_holders=1e6;
   %let max_holders_per_policy=10;
   data policies;
        length holder $32;
        do obsnum=1 to &n_policies;
           holder='holder '||put(obsnum,z10.);
           output;
           end;
        keep obsnum holder;
        run;
   data orig;
        length policy $32;
        do i=1 to &n_holders;
           policy='policy '||put(i,z10.);
           n_holders=ceil(ranuni(13131)*&max_holders_per_policy);
           do j=1 to n_holders;
              obsnum=ceil(ranuni(13131)*nobs);
              seqnum+1;
              output;
              end;
           end;
        stop;
        set policies nobs=nobs;
        keep policy seqnum obsnum;
        run;
   proc sort data=orig; by obsnum; run;
   data orig; merge orig(in=want) policies; by obsnum;
        if want;
        run;
   proc sort data=orig; by seqnum;
```

Then we applied our four methods using this much larger data set, and the execution times were as follows:

```
WALLCLOCK1=0:00:34.516
WALLCLOCK2=0:00:27.660
WALLCLOCK3=0:00:09.197
WALLCLOCK4=0:00:03.647
```

The times were progressively better as we tried the various methods. It is interesting to note that using a hash object with the add() method is much faster than loading the entire data set at once into a new hash object. Because we don't have to preprocess the data, there is more time savings.

## CONCLUSION

Although there were many possible approaches to resolving this problem, including MODIFY/REPLACE statements and _TEMPORARY_ arrays, my results showed that using a hash object with the add(), find(), and replace() methods allowed for the best execution time, without the need for additional preprocessing.

## ACKNOWLEDGMENTS

I would like to thank Robert Putnam of American Family Mutual Insurance Company for the opportunity to present these solutions.

## CONTACT INFORMATION

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

Richard D. Langston
rlangston@nc.rr.com