Paper 2396 - 2018

Using Arrays to Quickly Perform Fuzzy Merge Look-ups: Case Studies in Efficiency

Arthur L. Carpenter
California Occidental Consultants, Anchorage, AK

ABSTRACT

Merging two data sets when a primary key is not available can be difficult. The MERGE statement cannot be used when BY values do not align, and data set expansion to force BY value alignment can be resource intensive. The use of DATA step arrays, as well as other techniques such as hash tables, can greatly simplify the code, reduce or eliminate the need to sort the data, and significantly improve performance.

This paper walks through two different types of examples where these techniques were successfully employed. The advantages will be discussed as will the syntax and techniques that were applied. The discussion will allow the reader to further extrapolate to other applications.

KEYWORDS

ARRAY statement, DO loop, temporary arrays, MERGE statement, Hash Objects, Big Data, Brute force Techniques, PROC PHREG

INTRODUCTION

A fuzzy merge occurs when there is no clear key for matching observations between two or more data sets. This is an especially common problem when one of the matching variables is a date or datetime value. If the key variable is only measured to say the nearest minute, there is a high likelihood that time values measured to a second will not match exactly. Sometimes key values can be consolidated to say the nearest hour before the merge takes place, and this type of coarsening of the scale is sufficient and acceptable. When it is not acceptable, such as when elapsed times or times between events are needed, alternative approaches are required.

Another type of merge problem occurs where we are attempting to associate various time related events with other time related events that may have occurred in the past, perhaps even days in the past. Here we must 'remember' what has gone before. This type of merge is often further complicated by the fact that it is not unusual that multiple events may match making this a many-to-many merge with a fuzzy key.

A variation of this type of merge can be encountered in some survival analysis situations. In two earlier papers (Ake and Carpenter, 2002 and 2003) the development of an "Endpoints" data set is discussed. This data set contains the start and stop points (dates) of periods of time when a series of conditions are constant. When any of several conditions change a new period is started. The construction of this type data set requires the merging of several variables (and data sets) through the use of fuzzy merging techniques.

The example data sets, which have been simplified for this paper and used in the examples that follow, span a one year period for two patients. In the motivating problems there were tens of thousands of patients and the time period spanned several years. The data contain information on prescription drug usage throughout the study period, and this data needs to be matched with information about adverse events of different types. We may want to ask questions such as what drugs were being taken on the day of each event or during the 5 days preceding the event.

The data set DRUGS, has prescription drug fill information, and contains one observation for each drug fill with drug name and number of days of supplied drug (FILL). For the purpose of this paper we will assume that each patient was 'on drug' for each day of the filled prescription. On any given day the subject may be taking any combination of drugs, however the data itself only reflects the dates that the prescriptions were filled.

subject	date	drug	fill
1	05JUN2014	A	15
1	11JUN2014	B	10
1	17JUN2014	A	25
1	18JUN2014	B	5
1	05JUL2014	C	20
1	22JUL2014	C	25
1	31JUL2014	B	15
1	10SEP2014	A	15
1	11SEP2014	B	10

The EVENTS data will have far fewer observations than the DRUGS data and there is small likelihood that either data set will have dates in common. In the portion of this example data set shown here the subject experienced two events one of each type.

subject	eventtype	dates
1	1	22SEP2014
1	2	23NOV2014

Ultimately the information from these two data sets will need to be merged, however without matching dates and with potentially very large data sets that may not even be sortable, the problem solving approach becomes important.

The examples in this study fall into a range of dates specified by the macro variables &START and &STOP. These variables are established using the macro %LET statement. Each of these macro variables contains a SAS® date (number of days since 01Jan1960) and not a date string. For demonstration purposes and my amusement, two different techniques were used to convert a date constant into a SAS date.

```
* Start and stop dates;
%let start = %sysfunc(putn('15may2014'd,5.));
%let stop  = %sysevalf('14may2015'd);
```

The primary reason that I was motivated to write this paper was because of a similar problem that was faced by one of my clients. They used programming techniques similar the first Brute Force approach shown below. Their process took 24 hours to execute, and they felt that this was slow enough to give me a call. Ultimately a solution that employed the use of arrays was applied, and the same result was achieved with a run time of slightly over a minute. This efficiency gain speaks to the power of these techniques.

BRUTE FORCE APPROACHES

Brute force solutions fill out or 'complete' the DRUGS data set so that the merge will have corresponding dates in both data sets. By expanding the DRUGS data set to one observation per date, we can be sure that the DRUGS data set will have a date that will align with the dates in the EVENTS data set, however we also greatly expand the size of the DRUGS data set as well as expending what can be non-trivial resources. There are several ways to expand the drug prescription data to the daily level, but because the point of this paper is to show you alternatives, only one will be shown.

Expanding the Data Using DO Loops

One of the approaches often taken to expand the data is to use DO loops for each drug individually. In this solution a separate data set is created for each drug. Each of these data sets (HASDRGA, HASDRGB, and HASDRGC) is expanded so that it will contain an observation for each date that the drug was prescribed for that subject. Each individual observation is expanded to one observation for each date that the drug was available to the subject. The observations are generated using a DO loop that ranges from the fill date through the number of fill days.

```
data  hasdrgA(keep= subject dates)
      hasdrgB(keep= subject dates)
      hasdrgC(keep= subject dates);
   set time104.drugs;
   if drug='A' then do dates = date to date+fill-1;
      output hasdrga;
   end;
   else if drug='B' then do dates = date to date+fill-1;
      output hasdrgb;
   end;
   else if drug='C' then do dates = date to date+fill-1;
      output hasdrgc;
   end;
   format dates date9.;
   run;
```

Although the individual dates are sorted within subject, there may be overlap between prescriptions, consequently each data set must be sorted using the NODUPKEY option to establish a unique set of dates. Only one of those SORT steps is shown here.

```
proc sort data=hasdrgc nodupkey;
   by subject dates;
   run;
```

Because these individual data sets may not encompass all the dates of our study, we also need to create a data set with all possible rows (all possible dates for each subject). A list of distinct subjects is created using a SQL step and then this list is expanded to one observation per date per subject using an iterative DO that spans the study period. The data set WORK.ALLDATES is now a template for all possible subject X date combinations.

```
proc sql;
   create table subjects as
      select distinct subject
         from time104.drugs;
   quit;

data alldates;
   set subjects;
   do dates = &start to &stop;
      output alldates;
   end;
run;
```

Combining the ALLDATES with each of the drug data sets gives us a picture of which drugs are available on any given day of the study period. Essentially the data set that is to be merged to the events data set has been 'sparsed' to contain one row for every SUBJCT X DATES combination. We now know that a merge with the events data will be successful because a primary key is guaranteed.

Obs	subject	dates	DrugA	DrugB	DrugC
156	1	17OCT2014		x	x
157	1	18OCT2014		x	x
158	1	19OCT2014		x	x
159	1	20OCT2014		x	x
160	1	21OCT2014		x	x
161	1	22OCT2014		x	x
162	1	23OCT2014	x	x	x
163	1	24OCT2014	x	x	x
164	1	25OCT2014	x	x	x
165	1	26OCT2014	x	x	x
166	1	27OCT2014	x	x	

With this base of all possible dates, we now have all the information needed to perform the merge or join. Since several data sets are to be put together, a DATA step MERGE is used. In the step shown here the individual data sets are merged at the same time as the events data are brought in.

ALLDATES provides the template of dates, and the drug prescription and event information is added on as is appropriate. The resulting data set will have one observation per subject and date. If you only need the information associated with the events (TIME104.EVENTS), you could uncomment the subsetting IF, and the size of the data set would be substantially reduced (but *not* the resources required to produce it!).

```
data Alldrgs;
   merge alldates(in=indates)
         hasdrga(in=indrga)
         hasdrgb(in=indrgb)
         hasdrgc(in=indrgc)
         time104.events(in=inevent);
   by subject dates;
   if indates;
/*   if inevent;*/
   if indrga then DrugA='x';
   if indrgb then DrugB='x';
   if indrgc then DrugC='x';
   format dates date9.;
   run;
```

A FUZZY MERGE OF TWO DATA SETS

Ultimately the brute force technique described above changes the problem of the fuzzy merge into one of a standard merge by 'filling in' the holes in the data, and thus guaranteeing that the key is sufficient to avoid a fuzzy merge situation. In the brute force approach the holes are filled by adding observations or rows into the data. A similar result can be achieved with the expenditure of substantially fewer resources through the use of arrays.

Transposing To Arrays

Rather than adding to the number of observations, we can 'fill in the dates' by loading them into arrays. In this example we have two identifiers that we need to track, SUBJECTs and DATEs. This solution uses a doubly subscripted array to track both the subject number and the date. If we wanted to do so, we could process BY SUBJECT and still use arrays subscripted only by date, however this would require us to sort the data. The solution shown here does not require any sorting, and will perform the merge using two SET statements in the same DATA step. A more complete discussion of the use of two SET statements to perform a merge can be found in Carpenter (2013a and 2013b).

In this example we will use both the subject number and the date to specify which drugs were being taken by each subject on any given date. This is a two dimensional array with the first dimension (SUBJECT) ranging from 1 to 50 (there are fewer than 50 subjects in this study and their subject numbers range from 1 to 50. The second dimension is date, which for this study ranges from 15May2014 to 14May2015. These dates correspond to the values 19,858 and 20,222 respectively, which results in a range of 365 values. The total number of memory positions for this array will be 50*365=18,250. This is *NOT* a large array. In the motivating problem the date dimension was actually measured to the nearest minute and spanned more than a year for each subject. Essentially array size is only limited by the amount of available memory, and even a million elements would take less than a megabyte of memory.

```
218  %put &=start;
START=19858
219  %put &=stop;
STOP=20222
```

There will be one array for each of the three drugs. These arrays will be defined such that the data values (SUBJECT, DATE) can be used directly as indexes to the arrays. This array usage is known as key-indexing.

```
array drga {50,&start:&stop} $1 _temporary_;
           ❶      ❷       ❸    ❹
```

❶ Since the values for SUBJECT range from 1 to at most 50, this dimension can be specified with a single number.

❷ The date values do not start at 1, but rather from 19,858 (May 15, 2014). We indicate that the index values will not start at 1 by specifying the start and stop values separated by a colon. The range of this dimension could be calculated as: `%EVAL(&STOP - &START + 1)`.

❸ Since we only need to indicate the presence or absence of a drug for a given date, we need only store an indicator value, and we can do this in a single byte.

❹ Values in temporary arrays are automatically retained, and variable names need not be specified.

```
data EventDrugs(keep=subject date eventtype
                     drugA drugB drugC);
   array drga {50,&start:&stop} $1 _temporary_;
   array drgb {50,&start:&stop} $1 _temporary_;
   array drgc {50,&start:&stop} $1 _temporary_;
```

The arrays will be loaded and merged with the EVENTS data in a single DATA step. Neither of the incoming data sets need to be sorted. Since there are three drugs, there will be three temporary arrays defined – each coordinated by the same index values.

We load the arrays using a DO UNTIL loop. Notice that the SET statement for the drug prescription information is inside of this loop. When a SET or MERGE is inside of a DO loop, this is known as a DOW loop (DO – Whitlock; named for Ian Whitlock who popularized this technique). The use of the term 'DOW loop' is unofficial and while you can find it commonly used in SAS papers, you will not find it used in SAS documentation. By using these DO UNTIL loops all the incoming data will be read in a single pass of the DATA step.

```
do until(drgDone);
   set time104.drugs end=drgdone;

   if drug='A' then do i = date to date+fill-1;
      if &start le i le &stop then drga{subject,i} = 'X';
   end;
   else if drug='B' then do i = date to date+fill-1;
      if &start le i le &stop then drgb{subject,i} = 'X';
   end;
   else if drug='C' then do i = date to date+fill-1;
      if &start le i le &stop then drgc{subject,i} = 'X';
   end;
end;
```

Placing the SET inside of the DO UNTIL has a number of advantages, here it allows us to read all the observations from the first data set – TIME104.DRUGS, before reading any of the event data.

The variable DATE has the fill date which starts the expansion. FILL is the number of days the drug is available, making the last available date that the drug is available DATE+FILL-1. Each of these dates is assigned an 'X' to indicate drug availability. We are now ready to merge in the event data.

The loop is terminated when the last observation is read (END=DRGDONE). At that time the arrays have been fully populated and the prescription fill information has been expanded to note all the dates that each drug was available.

In the same DATA step the second data set (TIME104.EVENTS) is also read using a SET statement within a second DO UNTIL loop. This data set has the date of the event of interest stored in the variable DATES, which is renamed to DATE for consistency. This date, along with the subject number associated with this event, becomes the index to the three drug arrays that store drug availability. Since these arrays have an entry for every date within the study, we are guaranteed to be able to match the event dates to the drug availability information. The merge has been completed without sorting, while maintaining a fairly small memory footprint.

```
do until(evntdone);
   set time104.events(rename=(dates=date))
       end=evntdone;
   DrugA = drga{subject,date};
   DrugB = drgb{subject,date};
   DrugC = drgc{subject,date};
   output eventdrugs;
end;
stop;
run;
```

For this study we only want the drug availability for each event date, consequently the OUTPUT statement is also inside this DOW loop. All the incoming data has now been read through a single pass of the DATA step. The STOP statement prevents subsequent additional passes of the DATA step.

If we had wanted all the dates in our final data set along with both event and drug information, we could have also stored event data in an array and then subsequently retrieve it by stepping through the arrays by subject and date. This is done in the examples that build the ENDPOINTS data later in this paper.

Transposing Using a Hash Table

The array solution described above requires a three doubly subscripted arrays. For the problem described here this is a doable solution, and the most efficient. However there a number of ways that the array solution could become less practical.

1) Arrays have fixed dimension which is established when the DATA step is compiled. The memory is allocated whether or not it is needed.
2) Since each drug requires its own array, even a small increase in the number of drugs could make the resulting coding more complicated.
3) In our example there were only two index variables resulting in two dimensional arrays. Additional index variables would add additional dimensions to the arrays making them more complex and more difficult to manage.
4) We were also fortunate in this example in that each of the index variables (SUBJECT and DATE) were numeric. Since arrays require numeric index values, we would not have been able to use this technique if any of the index variables had been character (sometimes character values can be converted to numeric, but this also adds complexity).
5) An array can only store one value per array location. Of course this value can contain multiple characters, however it is still only one value.

The use of hash tables can eliminate each of the limitations of arrays. While arrays can be, in some ways, more efficient (hash tables have a bit more overhead), hash tables provide a very practical alternative to arrays. Although hash tables are accessed differently than arrays, they are very much like 'super' arrays, without any of the limitations of standard arrays mentioned above.

In this example the LENGTH statement ❶ is not needed, but is included to document the variables and their lengths.

```
data EventDrugs(keep=subject date eventtype
                     drugA drugB drugC);
   length subject date eventtype 8
          drug hasdrug drugA drugB drugC $1; ❶

   * Declare the has table.;
   declare hash hmerge();❷
      rc1 = hmerge.defineKey('subject', 'date', 'drug'); ❸
      rc2 = hmerge.defineData('hasdrug'); ❹
      rc3 = hmerge.defineDone();❺
```

Just as arrays are declared through the ARRAY statement, the hash table is also declared. With hash tables this is a multi-statement process.

❷ The DECLARE statement is used to name the hash table (HMERGE) and to initiate the process of instantiating the hash object.

❸ The DEFINEKEY method is used to name the index variables. Unlike the array indices, these variables can be either numeric or character. In this solution there are three index variables (SUBJECT, DATE, and DRUG).

❹ This hash table is going to store only one item for recovery later. The variable HASDRUG is an indicator that for a given date the subject had access to the drug noted by the variable DRUG. The DEFINEDATA method is used to list one or more variables to be stored. Unlike an array which can store only one item of a fixed type and length, the hash table can store any number of items at each location, and these can be of varying type and length.

❺ When the hash table has been fully defined, the DEFINEDONE method closes the declaration of the hash object.

During DATA step compilation the hash table is instantiated (declared). Then during DATA step execution, we need to load it with information as to the availability of the drugs. Once again a DO loop is used to expand the dates of the prescription fills. As in the array example the data are read using a DOW loop. Unlike the previous example which used a separate array for each type of drug, in this example the drug type (DRUG) is an index to the hash table, therefore we only need to note that a drug is present (HASDRUG).

```
* Load the Hash Table with the drug information;
do until(drgdone);
   set time104.drugs(rename=(date=dt))
       end=drgdone;
   hasdrug='X'; ❻
   do date = dt to dt+fill-1;
     if &start le date le &stop then rc4 = hmerge.add();❼
   end;
end;
```

❻ The variable HASDRUG will be used to indicate that a drug is available. The type of drug (DRUG) is an index variable.

❼ The DO loop steps through all the dates for this prescription and the ADD method is used to write each DATE for this SUBJECT X DRUG combination to the hash table.

Once all the prescription drug availability information has been loaded, the events data can be read. As with the array example a DOW loop is used.

```
do until(evntdone);
   set time104.events(rename=(dates=date))
       end=evntdone;
   * The variable DRUG is one of the index variables
   * and is needed to access the hash table;
   drug='A'; ❽
      rc5 = hmerge.find();❾
      if not rc5 then drugA = hasdrug; else druga=' '; ❿
   drug='B';
      rc6 = hmerge.find();
      if not rc6 then drugB = hasdrug; else drugb=' ';
   drug='C';
      rc7 = hmerge.find();
      if not rc7 then drugC = hasdrug; else drugc=' ';
   output eventdrugs;
end;
stop;
run;
```

❽ Because DRUG is an index variable for the HASH table we need to extract the fill information for this SUBJECT X DATE combination for each of the three types of drug.

❾ The FIND method is used to recover information from the hash table for this combination of the index variables. The return code (RC5) indicates whether or not this combination was found.

❿ A return code of zero (0) indicates success and the stored value of HASDRUG is returned, and we can assign the appropriate indicator value to DRUGA, DRUGB, or DRUGC before writing out the observation.

We can actually simplify this code a bit more, both in the hash table definition and in the code that accesses the hash table. Unlike arrays, hash tables are dynamically allocated. This means that memory is only used when something is stored in the hash table. In our case if a drug is not available nothing is stored. When using an array the indicated position would already be allocated and would remain blank. This means that simply the presence of a combination of index variables is sufficient to indicate a drug and the HASDRUG variable is unnecessary. The hash object declaration is simplified by removing the unnecessary DEFINEDATA method.

```
declare hash hmerge();
   rc1 = hmerge.defineKey('subject', 'date', 'drug');
   rc3 = hmerge.defineDone();
```

We load the hash table by reading the drug fill information and then by again using the ADD method to note each date for which the drug was available. Although nothing is stored in the data portion of the hash table, the combination of key variables establishes that combination in the hash table. Notice that if a given date has already been assigned the ADD fails, but that is OK we only need to know that the drug was available.

```
do until(drgdone);
   set time104.drugs(rename=(date=dt))
       end=drgdone;
   do date = dt to dt+fill-1;
      if &start le date le &stop then rc4 = hmerge.add();
   end;
end;
```

In the second DOW loop we can make our code more flexible by using arrays to hold the names of the variables and the names of the drugs. This would be very important if there were more than three drugs, but a bit of overkill in this example. Two arrays are defined. DRGLIST is temporary and holds the drug names, while DRGVAL defines the three new variables that are to receive the values based on the information in the hash table. In the iterative DO we step through the three drugs, individually recover whether or not a table entry exists (RC5=0), and then assign values to the new drug indicator variables. This approach would require less coding for a large number of drugs.

```
array drglist {3} $1 _temporary_ ('A', 'B', 'C');
array drgval  {3} $1 drugA drugB drugC;
do until(evntdone);
   set time104.events(rename=(dates=date))
       end=evntdone;
   do i = 1 to dim(drgval);
      drug=drglist{i};
      rc5 = hmerge.find();
      if not rc5 then drgval{i} = 'X'; else drgval{i}=' ';
   end;
   output eventdrugs;
end;
stop;
run;
```

When faced with a fuzzy merge, whether you use arrays or a hash table to perform the merge, you can substantially improve performance over the 'brute force' approach.

THE ENDPOINTS PROBLEM

The endpoints data set was described as a part of a survival analysis in Ake and Carpenter (2002 and 2003). The needed information comes from multiple data sets with the key variable being a date. However the dates do not align and cannot be used to merge the data sets. Although there are a number of variations (see the above references for more information on how to use this data set), primarily each observation of this data set will indicate a range (usually a time value of some kind) from start to end over which a set of conditions are constant. When any one of the conditions changes, a new observation is generated. This means that we need to 'remember' all of the values from the previous observation so that we can compare them to the current observation. The comparison process is further complicated when the various indicators (things like types of events and changes to the availability to drugs) are coming from different data sets.

Obs	subject	entry	exit	eventtype	druga	drugb	drugc
28	1	12NOV2014	14NOV2014	.			X
29	1	14NOV2014	28NOV2014	.			
30	1	28NOV2014	29NOV2014	1			
31	1	29NOV2014	12DEC2014	.			
32	1	12DEC2014	16DEC2014	.		X	
33	1	16DEC2014	03JAN2015	.		X	X
34	1	03JAN2015	10JAN2015	.	X	X	X

For the drug/event data used in the previous examples, the 'endpoints' data set will have the

subject number, a start and end date for the range, indicators for presence or absence of drugs, and an event code value. A portion of the desired 'ENDPOINTS' data set is shown to the right. In the previous examples we only wanted those observations that had an event. This data set will have one observation for each unique set of values.

Extending the Arrays Transpose Solution

Many of the same techniques used with the arrays solution above will be applied again here, however this time we will not limit the new data set to only those observations with events. Notice that the array definitions include one for events as well. There will be an array for each variable that is to be used as an indicator. In this case three drugs and an event code. Each array will then be loaded through the use of a SET statement within a DO UNTIL loop. There can be as many arrays and incoming data sets as needed. Each incoming data set will have its own DOW loop, and none of the incoming data sets need to be sorted.

```
data Endpoints(keep=subject entry exit eventtype drugA drugB drugC);
   array drga {50,&start:&stop} $1 _temporary_;
   array drgb {50,&start:&stop} $1 _temporary_;
   array drgc {50,&start:&stop} $1 _temporary_;
   array evnt {50,&start:&stop}    _temporary_;
   * Load the drug arrays;
   do until(drgDone);
      set time104.drugs end=drgdone;

      if drug='A' then do i = date to date+fill-1;
         if &start le i le &stop then drga{subject,i} = 'X';
      end;
      else if drug='B' then do i = date to date+fill-1;
         if &start le i le &stop then drgb{subject,i} = 'X';
      end;
      else if drug='C' then do i = date to date+fill-1;
         if &start le i le &stop then drgc{subject,i} = 'X';
      end;
   end;
```

Since the event code data is held in a separate data set, the array associated with event codes will be loaded through a separate DO UNTIL.

```
* Read and load the event data;
do until(evntdone);
   set time104.events end=evntdone;
   if &start le dates le &stop then evnt{subject,dates}=eventtype; ❶
end;
```

❶ Each event code associated with an event with an in range date is written to the EVNT array.

Once all the indicator values have been loaded into the arrays, the end points (ENTRY and EXIT date) for each set of indicators can be determined. This is done in another loop, however this loop steps through the arrays that have just been established in subject / time sequence.

```
do subject = 1 to 50; ❷
   * Initialize for this subject;
   entry= &start; ❸
   exit = &start;
   druga=' ';
   drugb=' ';
   drugc=' ';
   eventtype=.;
   do date=&start to &stop; ❹
      * step through the arrays looking for changes;
      if (drga{subject,date} ne druga) or ❺
         (drgb{subject,date} ne drugb) or
         (drgc{subject,date} ne drugc) or
         (evnt{subject,date} ne eventtype) then do;
         * This is the first instance for a new combination;
         if date ne &start then do; ❻
            * write out the previous combination;
            exit=date;
            if entry ne exit then output endpoints;
         end;
         entry=date; ❼
         exit=date;
         druga=drga{subject,date};
         drugb=drgb{subject,date};
         drugc=drgc{subject,date};
         eventtype=evnt{subject,date};
         if date=&stop then output endpoints; ❽
      end;
      else if date=&stop then do; ❾
         exit=&stop;
         output endpoints;
      end;
   end;
end;
stop;
run;
```

❷ Step through the arrays one subject at a time. Although the incoming data sets are not necessarily sorted, once the arrays are loaded, the arrays are effectively sorted.

❸ The indicator variables are initialized for each new subject.

❹ Within a subject, we can step through the arrays in date order.

❺ When any indicator variable for the current date is different from the previous date, a new interval must be formed.

❻ Before saving the current values the values of the interval that has just ended (the previous interval) must be written out to the data set;

❼ Start the preparation of the next interval using the indicators for the current date.

❽At this point, since there has been a change in one or more of the indicator variables, the previous interval has been written out ❻. If this date is also the last date in the study then the new current interval also needs to be written out.

❾ The last interval must be written out even if there has not been a change in any of the indicator variables.

Using a HASH Table to Eliminate the Use of Arrays

In the previous example a series of arrays are used to hold the events through time. We can replace these arrays and the need to use them through the use of a hash table. The logic of the DATA step is very similar. The steps include loading the hash table for all incoming data and then extracting that information in order to build the intervals.

```
data Endpoints(keep=subject entry exit eventtype
                    drugA drugB drugC);

   length subject date entry exit eventtype 8
          drugA drugB drugC $1
          heventtype 8 hdrugA hdrugB hdrugC $1;

   * Declare the has table.;
   declare hash endpt();
      rc1 = endpt.defineKey('subject', 'date');
      rc2 = endpt.defineData('hdruga', 'hdrugb',
                             'hdrugc','heventtype');
      rc3 = endpt.defineDone();
```

The hash table ENDPT is declared with two key variables (SUBJECT and DATE). The table stores the indicator variables as data (HDRUGA, HDRUGB, HDRUGC, and HEVENTTYPE).

Since these variables must also appear on the PDV, their attributes must be established as well, and this is being done in the LENGTH statement. Technically, since SUBJECT and DATE are on incoming data sets, they do not need to be on the LENGTH statement.

The declaration of the hash table takes place during the compilation of the DATA step. Then during the execution of the DATA step the hash table is loaded by reading the incoming data. As before, the incoming data is read using a DO UNTIL loop, the fill dates are expanded and the resulting dates with drug availability are added to the hash table.

When using an array, all of the individual array elements are established during the compilation phase of the DATA step, this is not necessarily true for hash tables. Since hash tables are allocated dynamically, a given element (SUBJECT / DATE combination for the ENDPT hash object) does not exist on the hash table until it has been specifically added, and this is accomplish through the use of the ADD method. When a given hash element already exists, the ADD method will fail and the REPLACE method is used instead. This becomes important, when as in our example, we need to add information to the stored data one data element at a time.

Both the ADD and REPLACE methods gather the data to be stored from the corresponding variables on the PDV (in the ENDPT hash table shown in this example, the variables HDRUGA, HDRUGB, HDRUGC, and HEVENTTYPE). This means that the information must be on the PDV before it can be stored in the hash table. When we recover information from the hash table the FIND method is used. When the FIND is successful (the element has been stored previously), the data elements are recovered from the hash table and added to the PDV. However, and this is very important for our example, if the FIND fails (the current combination of SUBJECT and DATE have yet to be added to the hash table), then the current values of the data variables on the PDV are not changed. Because we are adding the four data elements individually, we will need to manually reset these variables to missing when a FIND would not be successful. Fortunately we can check to see if a specific element (SUBJECT and DATE) combination exists in the hash table by using the CHECK method. We check for an element and use FIND and REPLACE if it already exists. If it does not already exist we use the ADD method to add it to the table. This process of checking, data retrieval, and data adding or replacing is shown in the next code block.

Each of the four data values stored in the hash table ENDPT needs to be added separately. This requires us to recover the data values already stored (to add them to the PDV), change the one variable of interest on the PDV, and then write all four back to the hash table.

```
do until(drgDone);
   set time104.drugs end=drgdone;

   if drug='A' then do date = date to date+fill-1;
      if &start le date le &stop then do; ❶
         rcchk=endpt.check(); ❷
         if rcchk = 0 then rc4 = endpt.find(); ❸
         else call missing(hdruga, hdrugb, hdrugc,
                           heventtype);
         hdruga='X'; ❹
         if rcchk ne 0 then rc5=endpt.add(); ❺
         else rc=endpt.replace();
      end;
   end;
   else if drug='B' then do date = date to date+fill-1;
        . . . . similar DO blocks for each drug type are not shown . . . .
```

❶ If the fill date is within the study date bounds, we need to load the availability of the drug into the hash table.

❷ If this SUBJECT DATE combination has already been added to the hash table we use different methods than if this is its first occurrence. The CHECK method returns a 0 if the element already exists.

❸ We now load the four hash data values onto the DATA step's PDV. When the element is already in the hash table (RCCHK=0), the FIND method is used. Otherwise we need to initialize the four variables to missing (this is necessary to clear any potential retained values from a previous iteration.

❹ The indicator variable for this drug is marked to note that this drug was available for this subject on this date.

❺ The values in the four variables that are common to the hash table and to the PDV (HDRUGA, HDRUGB, HDRUGC, HEVENTTYPE) are copied from the PDV to the hash table. If this is a new element the ADD method is used, otherwise the REPLACE method is used.

This process is repeated for each of the four data variables stored in the hash table.

Much as we did when using arrays, once all the data elements have been added to the hash object, we step through the subjects and dates to build the ENDPOINTS data set. An observation is generated for each interval over which all four data variables are constant, with the interval dates indicated by the variables ENTRY and EXIT. When any one of the four data values changes a new interval is initiated. This means that we must 'remember' the previous as well as current values for each of the four variables, as well as tracking the entry and exit dates.

Although none of the incoming data sets were necessarily sorted, we can pull from hash object by subject and date within subject. This allows us to build the ENDPOINTS data set in sorted order.

```
do subject = 1 to 50;
   * Initialize for this subject; ❻
   entry= &start;
   exit = &start;
   druga=' ';
   drugb=' ';
   drugc=' ';
   eventtype=.;
   do date=&start to &stop;
      * Pull drug and event codes from the hash;
      rcchk=endpt.check(); ❼
      if rcchk =0 then rc6 = endpt.find();
      else call missing(hdruga, hdrugb, hdrugc, heventtype);
      if (hdruga ne druga) or ❽
         (hdrugb ne drugb) or
         (hdrugc ne drugc) or
         (heventtype ne eventtype) then do;
         * This is the first instance for a new combination;
         if date ne &start then do;
            * write out the previous combination;
            exit=date; ❾
            if entry ne exit then output endpoints;
         end;
         * Update the saved interval variables;
         entry=date;
         exit=date;
         druga=hdruga;
         drugb=hdrugb;
         drugc=hdrugc;
         eventtype=heventtype;
         if date=&stop then output endpoints; ❿
      end;
      else if date=&stop then do; ❿
         exit=&stop;
         output endpoints;
      end;
   end;
end;
stop;
run;
```

❻ For each subject we initialize the variables of interest.

❼ For this SUBJECT and DATE, either retrieve the data from hash object using the FIND method, or if this combination does not already exist, there was never an entry for this SUBJECT and DATE, set the variables to missing.

❽ Check for differences between the values on the PDV (from the previous interval) and the values from the most recent date (from the hash table). If there is a difference then this will start a new interval and we need to write out the previous interval and prepare to start a new interval ❾.

❾ The previous interval is complete and is written to the ENDPOINTS data set. The saved interval variables are then updated with the current values which will form the basis for the next interval.

❿ When this is the last date the current interval must be written out, as there is no following observation that can trigger its being written. Here the EXIT date is written to the interval indicating that the data has been truncated. Depending on how you set up your study and your analysis truncation might not be appropriate and you may want the exit date to be set to missing in the last interval.

SUMMARY

It is not uncommon for SAS programmers to labor under the misconception that the merging of SAS data sets must be a time and resource intensive operation. When primary keys are not available (fuzzy merge), it is often thought that only large SQL steps can be used. This paper demonstrates that unsorted data sets, even those without primary keys, can be efficiently merged in the DATA step. When properly used, arrays and hash tables can be extremely efficient tools when processing even very large data tables.

ABOUT THE AUTHOR

Art Carpenter's publications list includes; five books, two chapters in *Reporting from the Field*, and numerous papers and posters presented at SAS Global Forum, SUGI, PharmaSUG, WUSS, and other regional conferences. Art has been using SAS since 1977 and has served in various leadership positions in local, regional, and national user groups.

Art is a SAS Certified Advanced Professional Programmer, and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

Recent publications are listed on my sasCommunity.org Presentation Index page. SAS programs associated with this paper can be found at:
http://www.sascommunity.org/wiki/Using_Arrays_to_Quickly_Perform_Fuzzy_Merge_Look-ups_Case_Studies_in_Efficiency

AUTHOR CONTACT

Arthur L. Carpenter
California Occidental Consultants
10606 Ketch Circle
Anchorage, AK 99515

(907) 865-9167
art@caloxy.com
www.caloxy.com

Certified Clinical Trials Programmer Using SAS®9

Certified Advanced Programmer for SAS®9

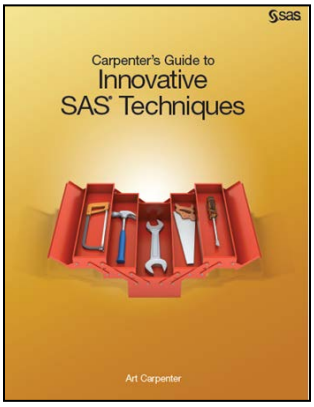
REFERENCES

Papers that describe the ENPOINTS data set and its usage in more detail:
Ake, Christopher F. and Arthur L. Carpenter, 2002, "Survival Analysis with PHREG: Using MI and MIANALYZE to Accommodate Missing Data", Proceedings of the 10th Annual Western Users of SAS Software, Inc. Users Group Conference, Cary, NC: SAS Institute Inc. Also published in the proceedings of WUSS 2012.

Ake, Christopher F. and Arthur L. Carpenter, 2003, "Extending the Use of PROC PHREG in Survival Analysis", Proceedings of the 11th Annual Western Users of SAS Software, Inc. Users Group Conference, Cary, NC: SAS Institute Inc.

Additional information on merging techniques through the use of arrays and hash objects (additional citations can be found in each of these references)**:**

Carpenter, Art, 2012, *Carpenter's Guide to Innovative SAS® Techniques* (SAS Press; Cary, NC: SAS Institute Inc. 2012).



Carpenter, Arthur L., 2013a, "Table Lookup Techniques: From the Basics to the Innovative", Proceedings of the 2013 Wisconsin – Illinois SAS Users Group. Also in the proceedings of the 2014 PharmaSUG Conference, Cary, NC: SAS Institute Inc., Paper FP_15 - 2014.

Carpenter, Arthur L., 2013b, "DATA Step Merging Techniques: From Basic to Innovative", Proceedings of the 2013 Wisconsin – Illinois SAS Users Group. Also in the proceedings of the 2014 SAS Global Forum Conference, Cary, NC: SAS Institute Inc., Paper 1271-2014.

TRADEMARK INFORMATION