Paper 244-31

Think FAST! Use Memory Tables (Hashing) for Faster Merging

Gregg P. Snell, Data Savant Consulting, Shawnee, KS

ABSTRACT

The objective of this paper is to present a simple way to merge datasets using memory tables. Well, actually, it will be an associative array (or hash) object. However, by simply thinking of this "object" as an in-memory table, it just may help you to grasp the hashing concepts and feel less intimidated about learning and using (what I believe is) the fastest method available for merging datasets.

INTRODUCTION

Merging datasets is something we all do. Unfortunately, the bigger the files get and the more often you need to do it, the faster you want the process to be. Like many of you, I learned how to use sort and match-merging very early in my SAS career and it became a mainstay for joining tables. The method is simple, strait forward, and it works – FAST! Prudent use of indexes can make match-merging even FASTER by eliminating the need to sort and thus reducing I/O. Indexes also allow you to make use of key-indexing techniques. But now with SAS 9, hashing is (within certain limitations) the FASTEST way to merge.

Unfortunately, many of you still have not taken the time to learn about hashing, in part, because so much of the syntax and terminology seems foreign. This paper is an attempt to remedy that situation by presenting examples of basic hashing concepts in simple terms as they relate to traditional SAS code.

HASHING DEFINED

Hashing is the process of converting a long-range key (numeric or character) to a smaller-range integer number with a mathematical algorithm or function – coupled with key-indexing. Key-indexing is the concept of using the value of a table's key as the index into that table, i.e. using zip code *values* as the index of the key variable:

```
client(66216)="POTENTIAL CLIENT";
```

Hashing (as a hand-coded process) was introduced to the SAS world by Paul Dorfman at SUGI 25 with *"Private Detectives In A Data Warehouse: Key-Indexing, Bitmapping, And Hashing"*. With the release of SAS 9, hashing was incorporated into the DATA step with two predefined component objects: the hash object and the hash iterator object. These objects provide a quick and efficient method to store, search, and retrieve data based on lookup keys. For the purposes of this paper I will only be covering the hash object. An explanation of how to use the iterator object can be found in *"Hashing: Generations"* which I presented at SUGI 28 along with Paul Dorfman. Full details on hashing syntax can be found in the SAS OnlineDoc® 9.1. Look under the contents tab and select: Base SAS **>** SAS Language Reference: Concepts **>** DATA Step Concepts **>** Using DATA Step Component Objects.

PROPAEDEUTICS

Unfortunately, the new hashing syntax uses "component objects" and something called "attributes and methods" as opposed to good old fashioned "statements". But do not worry. You only need to learn a handful of new commands to do basic merging and I will show them alongside traditional merge code. However, to effectively use these new hashing commands, you must have a good understanding of sequential/direct access, implicit/explicit looping and indexes. So before I jump into the hashing code, let me offer a quick review these three DATA step concepts.

SEQUENTIAL/DIRECT ACCESS

Sequential access is the concept of accessing (or reading) records from a table in sequential order, i.e. from the top to bottom, one after another. Here is a very simple example:

```
/* sequential access */
data work.sequential;
   set sashelp.class;
   put _all_;
   output;
run;
```

```
Name=Alfred  Sex=M Age=14 Height=69   Weight=112.5 _ERROR_=0 _N_=1
Name=Alice   Sex=F Age=13 Height=56.5 Weight=84    _ERROR_=0 _N_=2
Name=Barbara Sex=F Age=13 Height=65.3 Weight=98    _ERROR_=0 _N_=3
Name=Carol   Sex=F Age=14 Height=62.8 Weight=102.5 _ERROR_=0 _N_=4
Name=Henry   Sex=M Age=14 Height=63.5 Weight=102.5 _ERROR_=0 _N_=5
Name=James   Sex=M Age=12 Height=57.3 Weight=83    _ERROR_=0 _N_=6
Name=Jane    Sex=F Age=12 Height=59.8 Weight=84.5  _ERROR_=0 _N_=7
Name=Janet   Sex=F Age=15 Height=62.5 Weight=112.5 _ERROR_=0 _N_=8
Name=Jeffrey Sex=M Age=13 Height=62.5 Weight=84    _ERROR_=0 _N_=9
Name=John    Sex=M Age=12 Height=59   Weight=99.5  _ERROR_=0 _N_=10
Name=Joyce   Sex=F Age=11 Height=51.3 Weight=50.5  _ERROR_=0 _N_=11
Name=Judy    Sex=F Age=14 Height=64.3 Weight=90    _ERROR_=0 _N_=12
Name=Louise  Sex=F Age=12 Height=56.3 Weight=77    _ERROR_=0 _N_=13
Name=Mary    Sex=F Age=15 Height=66.5 Weight=112   _ERROR_=0 _N_=14
Name=Philip  Sex=M Age=16 Height=72   Weight=150   _ERROR_=0 _N_=15
Name=Robert  Sex=M Age=12 Height=64.8 Weight=128   _ERROR_=0 _N_=16
Name=Ronald  Sex=M Age=15 Height=67   Weight=133   _ERROR_=0 _N_=17
Name=Thomas  Sex=M Age=11 Height=57.5 Weight=85    _ERROR_=0 _N_=18
Name=William Sex=M Age=15 Height=66.5 Weight=112   _ERROR_=0 _N_=19
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.SEQUENTIAL has 19 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

As you can see by the sequence of **_N_** values, records were read from the table **sashelp.class** sequentially from top to bottom.

Direct access is the concept of accessing (or reading) *specific* records from a table in no particular order. To do this, you must specify which row(s) to read. Here is a very simple example that reads records by the observation or **_N_** value:

```
/* direct access */
data work.direct;
   do i=2, 3, 9;
      set sashelp.class point=i;
      put _all_;
      output;
   end;
   stop;
run;

i=2 Name=Alice Sex=F Age=13 Height=56.5 Weight=84 _ERROR_=0 _N_=1
i=3 Name=Barbara Sex=F Age=13 Height=65.3 Weight=98 _ERROR_=0 _N_=1
i=9 Name=Jeffrey Sex=M Age=13 Height=62.5 Weight=84 _ERROR_=0 _N_=1
NOTE: The data set WORK.DIRECT has 3 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

In this example, the row number value was stored in variable **i**. The **point=** option with the **set** statement read specific rows from the table. If this concept is new to you then I would encourage you to read more about it in the SAS OnlineDoc 9.1 under contents: Base SAS **>** SAS Language Reference: Dictionary **>** Dictionary of Language Elements **>** Statements **>** SET Statement. Another method of direct access will be described with the explanation of indexes.

IMPLICIT/EXPLICIT LOOPING

By default, DATA steps execute under an *implicit* loop – beginning with the first record of the table and looping through all the statements within the step until the last record is accessed. By doing this SAS is smart enough to know when the end-of-file (EOF) has been encountered and exits the implicit loop. However, you do not have to accept the default and can explicitly specify the looping. Here are two examples of sequential code presented side-by-side to further illustrate this concept. Both of these code snippets essentially do the exact same thing:

```
/* implicit looping */
data work.sequential;

   set sashelp.class;
   put _all_;
   output;

run;
```

```
/* explicit looping */
data work.sequential;
   do until (eof);
      set sashelp.class end=eof;
      put _all_;
      output;
   end;
run;
```

Please keep in mind that when *directly* accessing tables, SAS has no way of knowing when you are finished reading records. Consequently, you must utilize *explicit* looping to prevent an endless loop. In the previous direct access example, I used the **stop** statement to terminate looping with only a single pass through the DATA step. Just for grins, rerun that example without the **stop** statement – just be prepared to interrupt the job or it will run forever!

INDEXES

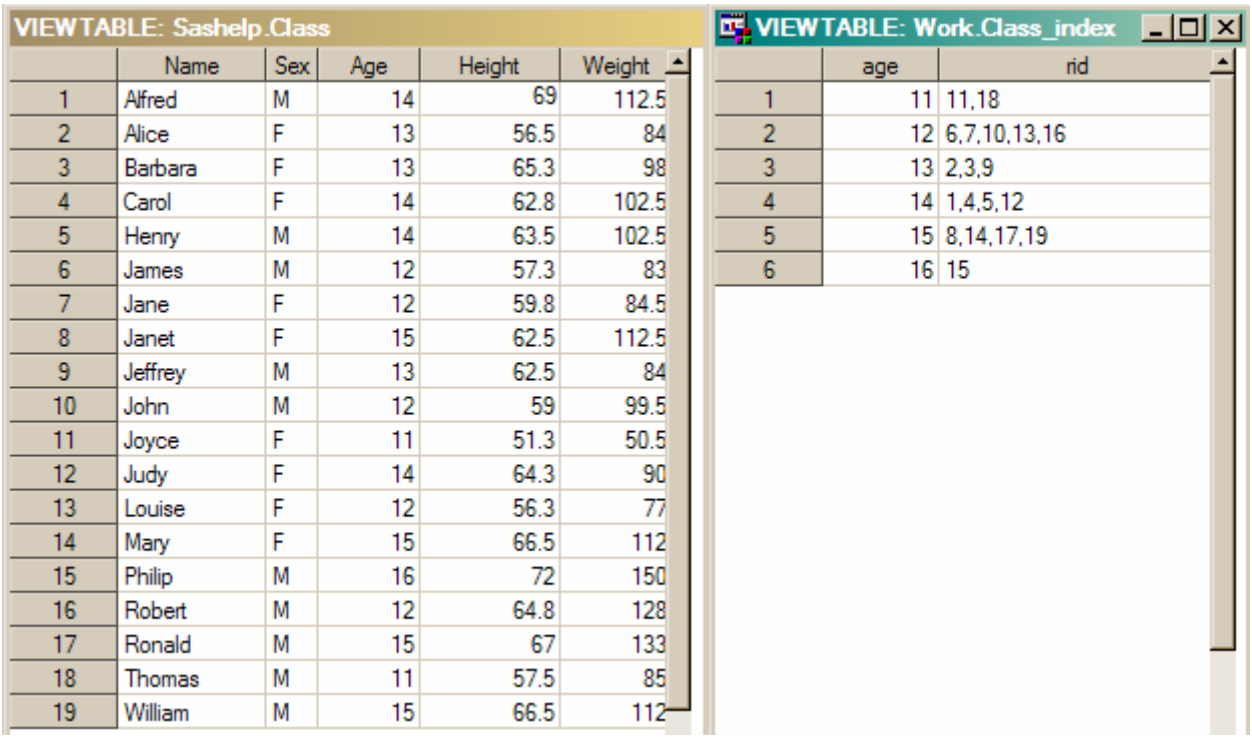
An index is an optional file created for a SAS dataset that provides *direct access* to specific records based on key *values*. The index stores the key values in *ascending order* for the designated variables and includes pointers to the records matching the corresponding values. You can learn greater details about creating and using indexes from the SAS OnlineDoc 9.1 under contents: Base SAS **>** SAS Language Reference: Concepts **>** SAS Files Concepts **>** SAS Data Files **>** Understanding SAS Indexes.

Remember how the previous direct access example pointed to specific table records by row number? If the table had an index, I could have directly accessed specific records based on a *value*, rather than the row number, which makes much more sense programatically. What would an index look like? Well, we can not open or look at SAS indexes, but here is an example you can run to simulate an index:

```
/* simulate an index on variable: age */
data work.class_index;
   set sashelp.class;
   row_id=_n_;
   keep age row_id;
run;

proc sort data=work.class_index;
   by age row_id;
run;

data work.class_index;
   keep age rid;
   retain age rid;
   length rid $20;
   set work.class_index;
   by age;
   if first.age then
      rid = trim(put(row_id,best.-L));
   else
      rid = trim(rid) || ',' ||
            trim(put(row_id,best.-L));
   if last.age then output;
run;
```



VIEWTABLE: Sashelp.Class

	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69	112.5
2	Alice	F	13	56.5	84
3	Barbara	F	13	65.3	98
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83
7	Jane	F	12	59.8	84.5
8	Janet	F	15	62.5	112.5
9	Jeffrey	M	13	62.5	84
10	John	M	12	59	99.5
11	Joyce	F	11	51.3	50.5
12	Judy	F	14	64.3	90
13	Louise	F	12	56.3	77
14	Mary	F	15	66.5	112
15	Philip	M	16	72	150
16	Robert	M	12	64.8	128
17	Ronald	M	15	67	133
18	Thomas	M	11	57.5	85
19	William	M	15	66.5	112

VIEWTABLE: Work.Class_index

	age	rid
1	11	11,18
2	12	6,7,10,13,16
3	13	2,3,9
4	14	1,4,5,12
5	15	8,14,17,19
6	16	15

You will notice that the table **work.class_index** is in ascending order of the index variable **age.** Variable **rid** (which is character for display purposes only) contains all of the row numbers with corresponding values of age. This is what I think a SAS index file looks like. Now, let us build a real index and then retrieve the values in ascending order:

```
options msglevel=i;

/* class with index*/
data work.class (index=(age));
   set sashelp.class;
run;

NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.CLASS has 19 observations and 5 variables.
NOTE: Simple index age has been defined.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds


/* sequential access in ascending order */
data work.sortedclass;
   set work.class;
   by age;
run;

INFO: Index Age selected for BY clause processing.
NOTE: There were 19 observations read from the data set WORK.CLASS.
NOTE: The data set WORK.SORTEDCLASS has 19 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

VIEWTABLE: Work.Class

	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69	112.5
2	Alice	F	13	56.5	84
3	Barbara	F	13	65.3	98
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
6	James	M	12	57.3	83
7	Jane	F	12	59.8	84.5
8	Janet	F	15	62.5	112.5
9	Jeffrey	M	13	62.5	84
10	John	M	12	59	99.5
11	Joyce	F	11	51.3	50.5
12	Judy	F	14	64.3	90
13	Louise	F	12	56.3	77
14	Mary	F	15	66.5	112
15	Philip	M	16	72	150
16	Robert	M	12	64.8	128
17	Ronald	M	15	67	133
18	Thomas	M	11	57.5	85
19	William	M	15	66.5	112

VIEWTABLE: Work.Sortedclass

	Name	Sex	Age	Height	Weight
1	Joyce	F	11	51.3	50.5
2	Thomas	M	11	57.5	85
3	James	M	12	57.3	83
4	Jane	F	12	59.8	84.5
5	John	M	12	59	99.5
6	Louise	F	12	56.3	77
7	Robert	M	12	64.8	128
8	Alice	F	13	56.5	84
9	Barbara	F	13	65.3	98
10	Jeffrey	M	13	62.5	84
11	Alfred	M	14	69	112.5
12	Carol	F	14	62.8	102.5
13	Henry	M	14	63.5	102.5
14	Judy	F	14	64.3	90
15	Janet	F	15	62.5	112.5
16	Mary	F	15	66.5	112
17	Ronald	M	15	67	133
18	William	M	15	66.5	112
19	Philip	M	16	72	150

Here you can see that we were able to access the unsorted table in *ascending order* of age because an index stores values in ascending order. Of course, this does not do you much good if you need a descending sort. Also, not having to sort tables *may* save some processing time but be aware that there are CPU and I/O costs associated with creating and using indexes. To learn more about this, look for "*Deciding Whether to Create an Index*" in the SAS OnlineDoc 9.1 contents: Base SAS **>** SAS Language Reference: Concepts **>** SAS Files Concepts **>** SAS Data Files **>** Understanding SAS Indexes.

But the real benefit of indexes is that they provide *direct access* to specific observations by *value* rather than simply row number. You may not know how many, or which, records will match your criteria, but the index does! So, if I wanted to read only those records with an **age** value of **13** or **14**, I could use the **key=** option:

```
/* direct access */
data work.direct;
   do age=13,14;
      do until (eof);
         set class key=age end=eof;
          if _IORC_=0 then do; /* 0 indicates a match was found */
            put _all_;
            output;
          end;
          else _ERROR_=0; /* if no match, reset the error flag and continue */
     end;
   end;
   stop;
run;
```

```
age=13 eof=0 Name=Alice Sex=F Height=56.5 Weight=84 _ERROR_=0 _IORC_=0 _N_=1
age=13 eof=0 Name=Barbara Sex=F Height=65.3 Weight=98 _ERROR_=0 _IORC_=0 _N_=1
age=13 eof=0 Name=Jeffrey Sex=M Height=62.5 Weight=84 _ERROR_=0 _IORC_=0 _N_=1
age=14 eof=0 Name=Alfred Sex=M Height=69 Weight=112.5 _ERROR_=0 _IORC_=0 _N_=1
age=14 eof=0 Name=Carol Sex=F Height=62.8 Weight=102.5 _ERROR_=0 _IORC_=0 _N_=1
age=14 eof=0 Name=Henry Sex=M Height=63.5 Weight=102.5 _ERROR_=0 _IORC_=0 _N_=1
age=14 eof=0 Name=Judy Sex=F Height=64.3 Weight=90 _ERROR_=0 _IORC_=0 _N_=1
NOTE: The data set WORK.DIRECT has 7 observations and 5 variables.
```

```
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Now this is a very simple example, and you could have simply coded this with a **where** clause (which does not even require an index). *However*, by using the **key=** option, you can drive the direct access with another table:

```
/* direct access */
data work.driver;
   age=13; output;
   age=14; output;
run;

data work.direct;
   set work.driver;  /* <- sequential access & implicit loop */
   do until (eof);   /* <- explicit loop */
      set work.class key=age end=eof;  /* <- direct access */
      if _IORC_=0 then do; /* 0 indicates a match was found */
         put _all_;
         output;
      end;
      else _ERROR_=0; /* if no match, reset the error flag and continue */
   end;
run;

age=13 eof=0 Name=Alice Sex=F Height=56.5 Weight=84 _ERROR_=0 _IORC_=0 _N_=1
age=13 eof=0 Name=Barbara Sex=F Height=65.3 Weight=98 _ERROR_=0 _IORC_=0 _N_=1
age=13 eof=0 Name=Jeffrey Sex=M Height=62.5 Weight=84 _ERROR_=0 _IORC_=0 _N_=1
age=14 eof=0 Name=Alfred Sex=M Height=69 Weight=112.5 _ERROR_=0 _IORC_=0 _N_=2
age=14 eof=0 Name=Carol Sex=F Height=62.8 Weight=102.5 _ERROR_=0 _IORC_=0 _N_=2
age=14 eof=0 Name=Henry Sex=M Height=63.5 Weight=102.5 _ERROR_=0 _IORC_=0 _N_=2
age=14 eof=0 Name=Judy Sex=F Height=64.3 Weight=90 _ERROR_=0 _IORC_=0 _N_=2
NOTE: There were 2 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.DIRECT has 7 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

This is also known as the double set method of joining tables. You can learn more about this method (and what _IORC_ is all about) by reading "*Table Lookups: From IF-THEN to Key-Indexing*" by Art Carpenter. Did you notice that in this example I used both sequential and direct access methods as well as implicit and explicit looping? The table **work.driver** was sequentially accessed within an implicit loop while the table **work.class** was directly accessed within an explicit loop.

HASH OBJECT = MEMORY TABLE

The hash object is a fairly complicated "thing" with plenty of options and lots of, shall I say, *interesting* syntax. But I think the best way to get started with hashing is by thinking of the hash object as a table in memory - a traditional row/column or record/variable table with an index. All you need to do is create it, define it, fill it, and then access it using the DATA step object dot notation commands. Please note that all of these commands are valid *only within* a DATA step boundary. Each of the following code snippets must exist between a **data <somename>;** statement and a **run;** statement.

CREATE IT

Hash objects are dynamic run-time memory tables so they do not exist until you create them. You can also dynamically delete them but, by default, they will go away once the DATA step completes. To create a (hash) memory table, use the **declare** statement which also gives it a single-level name:

```
/* Create it */
declare hash h_small ();
```

This statement created a memory table called **h_small** – not **work.h_small**, just **h_small**. But at this point it has no structure (variables/index) or content (rows).

DEFINE IT

To define the memory table structure, just use some define commands:

```
/* Define it */
length keyvar smallvar1-smallvar4 8;
rc = h_small.DefineKey  ( "keyvar" );
rc = h_small.DefineData ( "smallvar1","smallvar2","smallvar3","smallvar4");
rc = h_small.DefineDone ();
```

These statements created an index variable called **keyvar** and four other variables named **smallvar1** through **smallvar4**. Notice that the length of these variables was declared *before* using them in the define commands. This is very important because the SAS compiler must know their lengths before they are used in the define commands. (Please do not get too hung up about this technicality, just go with it for now.) Also, more than one key could have been specified and they could have been any mix of numeric or character variables. Likewise more variables could have been listed, or eliminated altogether if they were not required (simply leave out the **.DefineData** statement). Once the **.DefineDone** command is issued, no other changes to the key or data variables can be made without first deleting the table and then recreating it. Also, the variable **rc** contains a return code for each execution of the statement, where a value of 0 indicates success.

FILL IT

Our memory table is now created and defined to have a single index with four variables. Now it needs filled with data from some other table:

```
/* Fill it */
do until ( eof_small );
   set work.small (keep=keyvar smallvar1-smallvar4) end = eof_small;
   rc = h_small.add ();
end;
```

This example used explicit looping to fill memory table **h_small** with every record from the SAS dataset **work.small**. Of course, you can use any method available within a DATA step to fill your memory table including hard-coded values, derived values, sub-set, concatenate, etc.

ACCESS IT

Our memory table is now complete except for accessing it. There are multiple ways to access the data within a memory table, but for simplicity, I will only discuss using direct access methods that utilize an index. For example:

```
/* Access it */
do until ( eof_big);
   set work.big end = eof_big;
   smallvar1=.; smallvar2=.; smallvar3=.; smallvar4=.;
   rc = h_small.find ();
   output;
end;
```

Once again, explicit looping was used to read every record of the SAS dataset **work.big** which contains the variable **keyvar** but not the **smallvar** variables. With each iteration of this loop

1) a row of data was retrieved from **work.big,** causing the key variable **keyvar** to have a value
2) variables **smallvar1** through **smallvar4** were initialized to missing since they were not coming in from **work.big** and were therefore *essentially* retained for each iteration of the loop
3) the **.find** command retrieved a row from the memory table **h_small** *only if* there was a corresponding entry that matched the current **keyvar** value.

Let me reiterate how memory table behavior is a bit different from traditional merging. The values of **smallvar1** through **smallvar4** will *only* be assigned new values *if* a match is found in the memory table. If no match is found, they *will not* be overwritten! You must do that yourself. I prefer to initialize them prior to executing the **.find** command but you could also code a conditional **if** following the **.find** as:

```
if rc ne 0 then do;
   smallvar1=.; smallvar2=.; smallvar3=.; smallvar4=.;
end;
```

That is it! No more syntax to learn! It is now time to put memory table merging to the test!

COMPARING MERGE METHODS

I have often said there are always 12 ways to do anything with SAS and merging is no exception. There are also (probably) 12 ways to do hash merging, but for simplicity I have only shown you one of those ways. So let us limit the comparison to two of the basic garden-variety merge techniques: match merging and merging with indexes.

All of the following examples were executed on my laptop -- Pentium 4, 2.4GHz, 1.25GB ram, 50GB 7200rpm disk, XP Pro SP2 with SAS 9.1 (TS1M3). Also, each example was executed from a new session to help eliminate subsequent benefits of disk caching that might skew results. Inserting standard disclaimer here: *Your mileage may vary*.

SAMPLE TABLES

To begin with, we need to create some sample data. By the way, this code was adapted from a posting on SAS-L (one of the best resources for learning SAS):

```
%let large_obs = 500000;

data work.small ( keep = keyvar small: )
     work.large ( keep = keyvar large: )
     ;
   array keys(1:500000) $1 _temporary_;
   length keyvar 8;
   array smallvar [20]; retain smallvar 12;
   array largevar [682]; retain largevar  55;
   do _i_ = 1 to &large_obs ;
      keyvar = ceil (ranuni(1) * &large_obs);
      if keys(keyvar) = ' ' then do;
         output large;
         if ranuni(1) < 1/5 then output small;
          keys(keyvar) = 'X';
    end;
   end;
run;

NOTE: The data set WORK.SMALL has 63406 observations and 21 variables.
NOTE: The data set WORK.LARGE has 315975 observations and 683 variables.
NOTE: DATA statement used (Total process time):
      real time           1:57.29
      cpu time            12.48 seconds
```

VIEWTABLE: Work.Large

	keyvar	largevar1	largevar2	largevar3	largevar4	largevar5	largev
1	92482	55	55	55	55	55	
2	199913	55	55	55	55	55	
3	460802	55	55	55	55	55	
4	271490	55	55	55	55	55	
5	24898	55	55	55	55	55	
6	409660	55	55	55	55	55	
7	426698	55	55	55	55	55	
8	478512	55	55	55	55	55	
9	136306	55	55	55	55	55	
10	488383	55	55	55	55	55	
11	344119	55	55	55	55	55	
12	279278	55	55	55	55	55	
13	237895	55	55	55	55	55	
14	317263	55	55	55	55	55	
15	291291	55	55	55	55	55	

VIEWTABLE: Work.Small

	keyvar	smallvar1	smallvar2	smallvar3	smallvar4	smallvar5	smallva
1	24898	12	12	12	12	12	
2	426698	12	12	12	12	12	
3	339763	12	12	12	12	12	
4	24773	12	12	12	12	12	
5	202410	12	12	12	12	12	
6	226744	12	12	12	12	12	
7	219907	12	12	12	12	12	
8	51637	12	12	12	12	12	
9	210354	12	12	12	12	12	
10	94925	12	12	12	12	12	
11	288002	12	12	12	12	12	
12	60782	12	12	12	12	12	
13	210687	12	12	12	12	12	
14	74215	12	12	12	12	12	
15	290910	12	12	12	12	12	

MATCH MERGING

It does not get anymore basic than this. Match merging requires sorting, but this is probably the easiest method to understand and code. Consequently, I would venture that this method is used in 80% (can you tell I am not a statistician?) of the code currently in use today.

```
/* basic match-merge with sort */

proc sort data=work.small; by keyvar; run;

NOTE: There were 63406 observations read from the data set WORK.SMALL.
NOTE: The data set WORK.SMALL has 63406 observations and 21 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           2.00 seconds
      cpu time            0.23 seconds

proc sort data=work.large; by keyvar; run;

NOTE: There were 315975 observations read from the data set WORK.LARGE.
NOTE: The data set WORK.LARGE has 315975 observations and 683 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           11:59.46
      cpu time            51.75 seconds

data work.match_merge;
   merge work.large (in=a)
         work.small (in=b);
   by keyvar;
   if a;
run;

NOTE: There were 315975 observations read from the data set WORK.LARGE.
NOTE: There were 63406 observations read from the data set WORK.SMALL.
NOTE: The data set WORK.MATCH_MERGE has 315975 observations and 703 variables.
NOTE: DATA statement used (Total process time):
      real time           8:39.31
      cpu time            20.14 seconds
```

As you can see, I/O was the real speed killer with having to sort both tables before merging. Result:

- **12 minute sort + 8:39 merge = 20.5 minutes**

MERGE WITH AN INDEX

As stated earlier, using an index can eliminate the need for sorting and that *usually* speeds things up. Let us find out:

```
options msglevel=i;

/* creating indexes */

proc datasets lib=work nolist;
  modify small; index create keyvar;
  modify large; index create keyvar;
quit;

INFO: Multiple concurrent threads will be used to create the index.
NOTE: Simple index keyvar has been defined.
NOTE: MODIFY was successful for WORK.SMALL.DATA.
INFO: Multiple concurrent threads will be used to create the index.
NOTE: Simple index keyvar has been defined.
NOTE: MODIFY was successful for WORK.LARGE.DATA.
NOTE: PROCEDURE DATASETS used (Total process time):
      real time           59.46 seconds
      cpu time            6.40 seconds

/* merge with indexes (no sorting) */

data work.match_merge_index;
   merge work.large (in=a)
         work.small (in=b);
   by keyvar;
   if a;
run;

INFO: Index keyvar selected for BY clause processing.
INFO: Index keyvar selected for BY clause processing.
NOTE: There were 315975 observations read from the data set WORK.LARGE.
NOTE: There were 63406 observations read from the data set WORK.SMALL.
NOTE: The data set WORK.MATCH_MERGE_INDEX has 315975 observations and 703
variables.
NOTE: DATA statement used (Total process time):
      real time           1:21:18.98
      cpu time            1:03.39
```

Well, uh, that didn't exactly turn out very well. Look at the difference between REAL time and CPU time. It is safe to conclude that, for this case, using an index resulted in disk thrashing. Result:

- **1 minute index + 81 minute merge = 82 minutes**

Of course, it is reasonable to presume that a large table would be persisted in a sorted state. With that in mind, let us compare merging where only **work.small** needed an index:

```
/* creating indexes */

proc datasets lib=work nolist;
  modify small; index create keyvar;
quit;

INFO: Multiple concurrent threads will be used to create the index.
NOTE: Simple index keyvar has been defined.
NOTE: MODIFY was successful for WORK.SMALL.DATA.
NOTE: PROCEDURE DATASETS used (Total process time):
      real time           2.29 seconds
```

```
      cpu time            0.22 seconds

/* merge with index on small (large is already sorted) */

data work.match_merge_index;
   merge work.large (in=a)
         work.small (in=b);
   by keyvar;
   if a;
run;

INFO: Index keyvar selected for BY clause processing.
NOTE: There were 315975 observations read from the data set WORK.LARGE.
NOTE: There were 63406 observations read from the data set WORK.SMALL.
NOTE: The data set WORK.MATCH_MERGE_INDEX has 315975 observations and 703
variables.
NOTE: DATA statement used (Total process time):
      real time           7:46.57
      cpu time            24.20 seconds
```

Simply relying on indexes without sorting was a disaster, but sorting the larger dataset before indexing improved things significantly. Result:

- **2 second index + 7:47 merge = 7.8 minutes (work.large already sorted)**

MEMORY TABLE MERGE

The moment you have all been waiting for…

```
/* merge with memory table (no sorting or indexing required!) */

data work.hash_merge (drop=rc i);

   /* Create it */
   declare hash h_small ();

   /* Define it */
   length keyvar smallvar1-smallvar20 8;
   array smallvar(20);
   rc = h_small.DefineKey  ( "keyvar" );
   rc = h_small.DefineData ( "smallvar1","smallvar2","smallvar3","smallvar4",
                             "smallvar5","smallvar6","smallvar7","smallvar8",
                          "smallvar9","smallvar10","smallvar11","smallvar12",
                         "smallvar13","smallvar14","smallvar15","smallvar16",
                        "smallvar17","smallvar18","smallvar19","smallvar20" );
   rc = h_small.DefineDone ();

   /* Fill it */
   do until ( eof_small );
      set work.small end = eof_small;
      rc = h_small.add ();
   end;

   /* Merge it */
   do until ( eof_large );
      set work.large end = eof_large;
      /* this loop initializes variables before merging from h_small */
      do i=lbound(smallvar) to hbound(smallvar);
         smallvar(i) = .;
      end;
      rc = h_small.find ();
     output;
   end;
run;
```

```
NOTE: There were 63406 observations read from the data set WORK.SMALL.
NOTE: There were 315975 observations read from the data set WORK.LARGE.
NOTE: The data set WORK.HASH_MERGE has 315975 observations and 703 variables.
NOTE: DATA statement used (Total process time):
      real time           7:17.23
      cpu time            16.59 seconds
```

Well, it looks like memory table merging was the fastest! Result:

- **7:17 merge = 7.3 minutes**

STACKING THE ODDS?

I suppose some of you may think that I selected the tables **work.small** and **work.large** simply because they would execute faster with hashing. This is not true. And to prove it, let us rerun the last two merges (those that were fastest) by reversing the merge order.

```
/* merge with index on small (large is already sorted) */

data work.match_merge_index;
   merge work.small (in=a)
         work.large (in=b keep=keyvar largevar1-largevar20);
   by keyvar;
   if a;
run;

INFO: Index keyvar selected for BY clause processing.
NOTE: There were 63406 observations read from the data set WORK.SMALL.
NOTE: There were 315975 observations read from the data set WORK.LARGE.
NOTE: The data set WORK.MATCH_MERGE_INDEX has 63406 observations and 41
variables.
NOTE: DATA statement used (Total process time):
      real time           2:08.84
      cpu time            7.11 seconds
```

Result:

- **2 second index + 2:08 merge = 2.2 minutes (work.large already sorted)**

And now using a memory table:

```
data work.hash_merge (drop=rc i);

   /* Create it */
   declare hash h_large ();

   /* Define it */
   length keyvar largevar1-largevar20 8;
   array largevar(20);
   rc = h_large.DefineKey  ( "keyvar" );
   rc = h_large.DefineData ( "largevar1","largevar2","largevar3","largevar4",
                             "largevar5","largevar6","largevar7","largevar8",
                          "largevar9","largevar10","largevar11","largevar12",
                         "largevar13","largevar14","largevar15","largevar16",
                        "largevar17","largevar18","largevar19","largevar20" );
   rc = h_large.DefineDone ();

   /* Fill it */
   do until ( eof_large );
      set work.large(keep=keyvar largevar1-largevar20) end = eof_large;
      rc = h_large.add ();
   end;

   /* Merge it */
```

```
   do until ( eof_small );
      set work.small end = eof_small;
      do i=lbound(largevar) to hbound(largevar);
         largevar(i) = .;
      end;
      rc = h_large.find ();
      output;
   end;
run;

NOTE: There were 315975 observations read from the data set WORK.LARGE.
NOTE: There were 63406 observations read from the data set WORK.SMALL.
NOTE: The data set WORK.HASH_MERGE has 63406 observations and 41 variables.
NOTE: DATA statement used (Total process time):
      real time           1:19.46
      cpu time            6.43 seconds
```

Result:

- **1:19 merge = 1.3 minutes**

Once again, memory table merging was fastest!

LIMITATIONS AND OVERCOMING THEM

While there are some definite limitations to using memory tables for merging, they certainly are not insurmountable. Following are what I consider to be the most significant limitations and ways around them.

LIMITATION #1

Hash memory tables are not persisted beyond the DATA step in which they are created. Remember when I previously stated that hash memory tables go away once the DATA step completes? That means the overhead incurred to create the memory table within a DATA step must be repeated if you were hoping to do the same thing in another DATA step a bit later in your program.

However, by using explicit looping, you can easily accomplish multiple merges within a single DATA step and eliminate the need to persist the table. And if you have enough memory, more than one memory table can be merged at a time even using completely different keys! And without sorting or indexing! This scenario represents the minimum I/O possible – a single pass through each table.

LIMITATION #2

Hash memory tables are, as you might suspect, limited by available memory. Now, this is not as big of an issue as you might first think. Just do a quick calculation of variables*length*records for an estimate of how many bytes of memory will be consumed. Take the final merge as an example. If you wanted to include all 682 variables from **work.large**, the memory requirement would have been about (682+1)*8*315975 or about 1.7gig. This is just a bit outside of my laptop's current capacity but certainly within the limits of most corporate computers.

Did you run out of memory? Just add an appropriate **-memsize** option when you invoke SAS. Still not enough memory? You could do a divide-and-conquer approach such that you create the memory table and fill it with only half of the records needed. Then merge these, delete the memory table, and recreate with the other half of the records. Unfortunately, this method also requires two passes of the master table which means extra I/O. Another method is to reduce the space taken up by the data variables. Just concatenate them into smaller strings and parse them back out at merge time. This adds extra calculation overhead for every record read, but it does preserve the benefit of not requiring an index or sort. Alas, there are some tables that are simply too large to be held in a memory table either due to row count, variable count, or both.

LIMITATION #3

Another interesting limitation to hash memory tables is that the key values must be distinct, i.e. no duplicate records by key. Here is an example that creates a SAS dataset *similar* to our memory table. Again, please don't confuse the SAS dataset **work.small** with the hash memory table **h_small**:

```
/* simulate hash object h_small */
proc sort data=work.small (keep=keyvar smallvar1-smallvar4)
          out=work.h_small (index=(keyvar))
           nodupkey;
   by keyvar;
```

```
run;


/* corresponding hash commands */
rc = h_small.DefineKey  ( "keyvar" );
rc = h_small.DefineData ( "smallvar1","smallvar2","smallvar3","smallvar4");
```

As a general rule, you will not want to use a hash memory table as the "many" in a many-to-one merge. The easiest way to overcome this limitation is to simply add addition variables to the key until it becomes unique. Another option is to create a sequence variable and add it to the key, thus making it unique. Here is a quick example:

```
rc = h_large.DefineKey  ( "keyvar","keyseq" );
rc = h_large.DefineData ( "largevar1","largevar2","largevar3", … )
rc = h_large.DefineDone ();

/* Fill it */
maxkeyseq=0;
do until ( eof_large );
   set work.large(keep=keyvar) end = eof_large;
   by keyvar;
   if first.keyvar then keyseq=0;
   keyseq+1;
   rc = h_large.add ();
   if last.keyvar then maxkeyseq=max(maxkeyseq,keyseq);
end;

/* Merge it */
do until ( eof_small );
   set work.small end = eof_small;
   do keyseq=1 to maxkeyseq;
      do i=lbound(largevar) to hbound(largevar);
         largevar(i) = .;
      end;
      rc = h_large.find ();
      output;
   end;
   drop maxkeyseq;
end;
```

CONCLUSION

Here is a quick recap of our merge results:

Match merge w/sorting = 20.5 minutes
Index merge w/o sorting = 82 minutes
Index merge w/sorting = 7.8 minutes
Memory table merge = 7.2 minutes (no sorting or indexing)

And when reversing the order of the tables:

Index merge w/sorting = 2.2 minutes
Memory table merge = 1.3 minutes

As stated at the beginning of this paper, I believe merging with hash memory tables is (within certain limitations) the fastest way to merge. I am not claming this to be true for *every* scenario possible, just for *most* of the situations I have encountered over the past several years. If you have a need for speed where you work, then I would definitely encourage you to use the examples from this paper and try it for yourself. I think you will be pleasantly surprised.

REFERENCES

Dorfman, Paul M. "Private Detectives In A Data Warehouse: Key-Indexing, Bitmapping, And Hashing" Proceedings of the Twenty Fifth Annual SAS® Users Group International Conference. April 2000.
<http://www2.sas.com/proceedings/sugi25/25/dw/25p129.pdf>

Dorfman, Paul M. and Snell, Gregg P. "Hashing: Generations" Proceedings of the Twenty Eighth Annual SAS® Users Group International Conference. March 2003.
<http://www2.sas.com/proceedings/sugi28/004-28.pdf>

SAS OnlineDoc® 9.1
<http://support.sas.com/91doc/docMainpage.jsp>

Carpenter, Arthur L. " Table Lookups: From IF-THEN to Key-Indexing" Proceedings of the Twenty Sixth Annual SAS® Users Group International Conference. April 2001.
<http://www2.sas.com/proceedings/sugi26/p158-26.pdf>

SAS-L (an electronic listserv of global SAS professionals) “Our Most Excellent Archives”
<http://www.listserv.uga.edu/archives/sas-l.html>

RECOMMENDED READING

The examples in this paper used but a fraction of the available DATA step object dot notation commands. This was intentional as an attempt to facilitate learning. An excellent reference sheet to accompany the OnlineDoc can be found here: <http://support.sas.com/rnd/base/topics/datastep/dot/hash-tip-sheet.pdf>

ACKNOWLEDGMENTS

This paper and, dare I say “hashing with SAS”, would probably not be around today except for the earlier work and teachings of *the* Hash Man – Paul Dorfman. I would also like to acknowledge Richard DeVenezia as another hashing advocate with numerous posts to SAS-L and some great examples on his own website: www.DeVenezia.com.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. I would also like to strongly encourage you to become part of the SAS-L community where you can learn just about anything you ever wanted to know about SAS. And at least one *frequent* poster will always advocate PERL solutions, but you can just ignore him! (☺)

You may contact me at:

Gregg P. Snell
Data Savant Consulting
5632 Noland Rd
Shawnee, KS 66216
Phone: (913) 638-4640
Fax: (208) 977-1943
E-mail: gsnell@DataSavantConsulting.com
Web: www.DataSavantConsulting.com