Paper 029-2008

How Do I Love Hash Tables? Let Me Count The Ways!

Judy Loren, Health Dialog Analytic Solutions, Portland, ME

ABSTRACT

I love hash tables. You will, too, after you see what they can do. This paper will start with the classic 2 table join. Nothing new, you say? But this version does it in one step without sorting. And, it's the fastest method around. Next, you will see the use of hash tables to join multiple datasets on different keys in one pass, again with no sorting. How about unlimited recursive chained lookups (like when you need to update an ID through as many incarnations as it may have had)? Hash tables solve that one, too. And before you catch your breath, you'll see how to create n datasets from one (or more), where n is determined (at runtime) by the number of values encountered in a variable. No muss, no fuss. One step (well, OK, this one does require a prior sort) and you're done. We'll wind up with an outer join, just to show that hash tables can do that, too. Take a look at what they can deliver, and you'll be counting the ways you love hash tables before the year is out.

INTRODUCTION

This paper is not a reference document on hash tables, nor an explanation of how they work. Both of those have been done better than I could do them by experts in the field (see References for a paper on the subject by Paul Dorfman). This paper is more of an homage to Paul and to the power of hash tables. It shows a variety of situations, fairly common among data processors, in which hash tables are an excellent, if not the best, solution.

That said, people who learn best by example will find this paper a good training tool. If you can identify with at least one of the specific examples, and begin to apply it in your work, you will have a basis for understanding the theoretical underpinnings of the technique and can then expand your usage to other applications.

WHAT ARE HASH TABLES?

For those who want it straight from the horse's mouth, here is a blurb from the SAS online documentation:

> SAS now provides two pre-defined component objects for use in a DATA step: the hash object and the hash iterator object. These objects enable you to quickly and efficiently store, search, and retrieve data based on lookup keys.
>
> The DATA step component object interface enables you to create and manipulate these component objects by using statements, attributes, and methods. You use the DATA step object dot notation to access the component object's attributes and methods.
>
> The hash and hash iterator objects have one attribute, fourteen methods, and two statements associated with them. See DATA Step Object Attributes and Methods.

Search the online documentation for the phrase "DATA Step Object Attributes and Methods" for the rest of the story.

Briefly, hash tables, much like arrays, are Data step constructs. A hash table is declared and used within one Data step, and it disappears when the Data step completes. Also like arrays, hash tables are accessed via an index. But unlike arrays, the index consists of a lookup key defined by the user rather than a simple sequential variable. Also, one hash table can contain multiple data elements (a bit like a series of arrays). Because the lookup keys are based on the values themselves access is extremely efficient. SAS does the work of building this efficiency so the user does not have to code anything other than identifying the key fields to use.

Hash tables are objects, so you call methods to use them. The syntax of calling a method is to put the name of the hash table, then a dot (.), then the method you want to use. Following the method is a set of parentheses in which you put the specifications for the method. You will see this kind of syntax in the statement below:

```
plan.DefineKey('plan_id');
```

In this case, Plan is the name of the hash table. DefineKey is the method and 'plan_id' is a specification expected by the method DefineKey (analogous to a positional parameter on a macro call).

A method can be called using the following expanded syntax:

```
rc = plan.definekey('plan_id');
```

In this case, the variable rc is specified by the user to contain the feedback from executing the method. If the instruction plan.definekey('plan_id') executes successfully, rc will contain a value of zero. If not, the value of rc will be something other than zero.

A note about the term associativearray: While it can be used to declare a hash table, it will not work with the iterator. Not all hash applications require an iterator, but since the term "hash" is shorter to type and works with the iterator, it seems like a good idea to get in the habit of using it instead.

Now, without further ado, on to the first example.

EXAMPLE 1: JOIN 2 DATASETS (WITHOUT SORTING)

Imagine that you have 2 datasets and you need to join them on a key. Let us consider the following case:

Members	
Member_id	Plan_id
164-234	XYZ
297-123	ABC
344-123	JKL
395-123	XYZ
495-987	ABC
562-987	ABC
697-123	XYZ
833-144	JKL
987-123	ABC

Plans	
Plan_id	Plan_desc
XYZ	HMO Salaried
ABC	PPO Hourly
JKL	HMO Executive

You are faced with the straightforward task of adding the Plan_desc to the Members dataset by matching the values in the column Plan_ID on the Members table to the Plan_ID on the Plans table. Before hash tables, you might have used a **proc sort** and a **data** step. This would involve sorting the Members dataset, which might be very large, on a variable that is unlikely to be the desired final sort key—a waste of time and resources. There are other solutions (see Loren, 2005) but the purpose of this paper is to introduce you to hash tables in a simple setting. This example does not highlight the reason for using hash tables but it does show how to use them.

APPROACH

As shown in the code below, we accomplish the task in one Data step. First we load the data from the Plans dataset into a hash table. We then process the Members data (in whatever order we happen to find it), looking up the value for the Plan_desc variable from the hash table for each record. The .find() method with no modifier uses the current value of the key variable Plan_id in the program data vector to identify the desired row in the hash table. It then puts the values of the field(s) listed in the .DefineData() method (in this case, Plan_desc) from the identified row in the hash table into the variable(s) of the same name (Plan_desc) in the program data vector, making it available for output. When we output the record to the dataset called Both, it will contain all the variables from the Members dataset plus the Plan_desc.

A note on error handling: In the code below, you see that we are initializing Plan_desc to missing immediately before attempting to read the hash table to pull in the Plan_desc for the current value of Plan_id. The purpose of this is to prepare for the case where the value of Plan_id from the members table does not exist in the dataset plans, and therefore will not be found on any row of the hash table. There are other methods of detecting and handling this that will be introduced below. Here we assume that the user wants all records from members in the dataset both, even if the value of Plan_id does not exist in the dataset Plans, but in that case, the user wants the Plan_desc to be missing.

Here is the code:

```
data both(drop=rc);
  declare Hash Plan ();                          /*  declare the name Plan for hash  */

   rc = plan.DefineKey ('Plan_id');              /*  identify fields to use as keys  */
   rc = plan.DefineData ('Plan_desc');           /*  identify fields to use as data  */
   rc = plan.DefineDone ();                      /*  complete hash table definition  */

  do until (eof1) ;                              /*  loop to read records from Plan  */
     set plans end = eof1;
     rc = plan.add ();                           /*  add each record to the hash table */
  end;

  do until (eof2) ;                              /* loop to read records from Members */
     set members end = eof2;
     call missing(Plan_desc);                    /* initialize the variable we intend to fill */
     rc = plan.find ();                          /* lookup each plan_id in hash Plan */
     output;                                     /* write record to Both */
  end;
  stop;
run;
```

The resulting dataset looks like this:

Both			
Obs	Plan_id	Plan_desc	Member_id
1	XYZ	HMO Salaried	164-234
2	ABC	PPO Hourly	297-123
3	JKL	HMO Executive	344-123
4	XYZ	HMO Salaried	395-123
5	ABC	PPO Hourly	495-987
6	ABC	PPO Hourly	562-987
7	XYZ	HMO Salaried	697-123
8	JKL	HMO Executive	833-144
9	ABC	PPO Hourly	987-123

EXAMPLE 2A: JOIN 3 DATASETS (WITHOUT SORTING, ONE STEP)

The second example is an enhancement of the first. It involves adding a second field to the Members dataset based on a different key. In the first example we needed to pick up Plan_desc based on the value of Plan_id. Hash tables allowed us to avoid sorting the large Members dataset. Since we don't have to sort to accomplish a join, why not do more than one at a time? Let's add a lookup on Group_id to Group_name.

Members		
Member_id	Plan_id	Group_id
164-234	XYZ	G123
297-123	ABC	G123
344-123	JKL	G456
395-123	XYZ	G123
495-987	ABC	G456
562-987	ABC	G123
697-123	XYZ	G456

Groups	
Group_id	Group_name
G456	Toy Company
G123	Umbrellas, Inc.

Now our Members table starts with 3 variables, and we need to add 2 more (Plan_desc, as in Example 1, and Group_name).

APPROACH

We will modify the code shown for Example 1 to create a second hash table in the same Data step, and look up 2 values for each record in the Members dataset. In the code below, the old code is shown in plain font; the additions to accomplish the second join are in **bold**.

```
data all (drop=rc);

  declare hash Plan ();                    /*  same as Example 1     */
   rc = plan.DefineKey ('Plan_id');
   rc = plan.DefineData ('Plan_desc');
   rc = plan.DefineDone ();

  declare hash Group ();                   /* similar code, 2nd table */
   rc = group.DefineKey ('Group_id');
   rc = group.DefineData ('Group_name');
   rc = group.DefineDone ();

  do until (eof1) ;                        /*  same as Example 1     */
     set plans end = eof1;
     rc = plan.add ();
  end;

  do until (eof2) ;                        /* similar code, 2nd table */
     set groups end = eof2;
     rc = group.add ();
  end;

  do until (eof3) ;
     set members end = eof3;
     call missing(Plan_desc);              /* initialize both lookups*/
     rc = plan.find ();                    /* same as Example 1      */
     call missing(Group_name);              /* initialize both lookups*/
     rc = group.find ();                   /* similar code, 2nd table */
     output;
  end;
stop; run;
```

The resulting dataset looks like this:

All					
Obs	Member_id	Plan_id	Plan_desc	Group_	Group_name
1	164-234	XYZ	HMO Salaried	G123	Umbrellas, Inc.
2	297-123	ABC	PPO Hourly	G123	Umbrellas, Inc.
3	344-123	JKL	HMO Executive	G456	Toy Company
4	395-123	XYZ	HMO Salaried	G123	Umbrellas, Inc.
5	495-987	ABC	PPO Hourly	G456	Toy Company
6	562-987	ABC	PPO Hourly	G123	Umbrellas, Inc.
7	697-123	XYZ	HMO Salaried	G456	Toy Company

EXAMPLE 2B: JOIN ON 2 LEVELS OF SPECIFICITY

Conditional joins are a special case of the 3 dataset example. In a conditional join you want to merge a record from one dataset with a record from one of two other datasets depending on a condition. The example we will use here is joining demographic summary data to an individual record based on zip code, where the zip code might be 5 digits or 9 digits long. U.S. zip codes at the 9-digit level are small subsets of the 5-digit zip code. In

some towns or cities, 9-digit zip codes contain enough people to provide summary statistics. In others, a single person might have his/her own 9-digit zip code. In the latter case, summary statistics would be available only at the 5-digit level. Given a dataset of members, we want to join the summary statistics at the finest level of detail available (try to join on 9-digit but if that is not available join on the 5-digit).

Members		
Obs	zip	Member_id
1	04021	164-234
2	22003-1234	297-123
3	45459-0306	344-123
4	03755	395-123
5	94305	495-987
6	78277-8310	562-987
7	88044-3760	697-123

Zip		
Obs	zip	income
1	04021	$45,000
2	22003	$56,000
3	03755	$75,000
4	45459	$72,000
5	94305	$96,000
6	78277	$32,000
7	88044	$47,000

Zip_plus_4		
Obs	zip	income
1	04021-0306	$45,999
2	22003-1234	$56,999
3	45459-0306	$72,999
4	03755-1234	$75,999
5	94305-1234	$96,999
6	78277-8310	$32,999
7	88044-3760	$47,999

In the illustration above, some members have a 5-digit zip code and some have a full 9-digit zip code. The reasons for the failure to match at the 9-digit level do not affect our strategy. If a match exists on the 9-digit table, we want the value of the income field from the 9-digit table to go on the Members record. If a match does not exist on the 9-digit table, for any reason, we want to fall back on the income field from the 5-digit table.

APPROACH

As in Example 2B, we will use one Data step and load 2 hash tables (one for the Zip5 level data and one for the Zip9 level data). For each record in the Members dataset we will first try to join to the Zip9 hash table. If this fails, we will join to the Zip5 hash table.

This example enables us to introduce more error handling techniques. The ability to detect whether the lookup has been successful is useful in many scenarios. Here we use it to determine whether we need to attempt the lookup to a second table or not. The variable you name in the assignment statement that executes the method (for example, **`rc = zip5.find ();`**) will contain a zero after a successful execution of the method, or a non-zero value if the execution was not successful. In this case, if a row in the hash table Zip5 exists for the current value of the key variable Zip, rc = 0. If no row matching Zip5 exists, the value of rc will be non-zero immediately after the statement is executed.

```
data member_income(drop=rc);

  length zip $ 9;
  declare hash Zip5 (dataset:  'Zip');          /* uses different way to load data */
   rc = zip5.DefineKey ('Zip');
   rc = zip5.DefineData ('Income');
   rc = zip5.DefineDone ();

  declare hash Zip9 (dataset:  'Zip_plus_4');   /* uses different way to load data */
   rc = Zip9.DefineKey ('Zip');
   rc = Zip9.DefineData ('Income');
   rc = Zip9.DefineDone ();

  do until (eof3) ;
     set members end = eof3;
     income = .;                    /* initialize income in case zip is not found at all */
     rc = zip9.find ();
     if rc ne 0 then                /* if 9 digit match does not exist, rc will not be zero */
     rc = zip5.find ();
     output;
  end;
stop;
run;
```

This code illustrates some new aspects of hash table processing. Instead of loading data into the hash table through a DO loop and SET statement, we take advantage of a feature of the DECLARE statement that allows us

to name a dataset for loading the hash table. This is convenient when no modifications or exceptions are needed and the hash table needs to be loaded only once. The data are loaded when the DefineDone method is called. Only the fields identified in the DefineKey or DefineData method are excerpted from the named dataset.

In the DO UNTIL (eof3) loop, we read each record from the Members dataset. We initialize income to missing before attempting to look up the income for that member's zip code. This is necessary because the income variable will continue to hold the value from the last successful lookup. If neither lookup is successful for a given member, we run the risk of holding the income from the last member.

The conditional lookup occurs next. First we attempt a match to the zip9 hash table (**`rc = zip9.find ();`**) Next we test the rc variable to see if that lookup was successful (**`if rc ne 0 then`**). Only if that lookup was not successful do we attempt a lookup to the zip5 hash table (**`rc = zip5.find ();`**).

The resulting dataset looks like this:

Member_income			
Obs	Member_id	zip	income
1	164-234	04021	$45,000
2	297-123	22003-1234	$56,999
3	344-123	45459-0306	$72,999
4	395-123	03755	$75,000
5	495-987	94305	$96,000
6	562-987	78277-8310	$32,999
7	697-123	88044-3760	$47,999

You can see that the Member records with 9-digit zip codes contain incomes from the 9-digit table (conveniently created ending in 999) and Member records with 5-digit zip codes contain incomes from the 5-digit table. This is the desired result—chalk up another one for hash tables!

ERROR HANDLING

Two techniques to avoid data errors in the case of mismatched data have been shown. It is essential to use one or more of them every time you employ hash tables. The choice depends on what you want your resulting dataset to contain.

In example 1 above, we wanted to keep all the records from the main dataset, whether or not they were found on the hash table, so we took the precaution of initializing the looked up variable values to missing prior to each lookup. If the lookup failed, the variables remain missing.

If you want an inner join, that is, you want the output dataset to contain only those records that are found on the hash table, you can control the output with the assignment variable:

```
If rc = 0 then output;
```

In this situation, it is not as essential that you initialize the variable(s) you are looking up, because in the event of a lookup failure, you are deleting the record.

Now, on to more examples.

EXAMPLE 3: DÉJÀ VU

We now leave the world of joins to show a different application of hash tables. In this case, we need to keep track of what we have encountered in this pass of a dataset and compare subsequent records to every value of a particular variable we have seen before. We will take advantage of one of the huge assets of hash tables: no need to dimension them in advance (we don't need to know how many values we might eventually load).

Specifically, let's conjure up a dataset of health care claims. For simplicity, let's consider just the Member_id, the service date, and the diagnosis associated with the claim (dx).

Claims			
Obs	Member_id	svc_dt	dx
1	164-234	2005/01/01	250
2	297-123	2005/02/03	4952
3	164-234	2005/03/15	78910
4	297-123	2005/04/14	250
5	297-123	2005/08/19	12345
6	164-234	2005/09/13	250
7	297-123	2005/11/01	4952

We will sort them in chronological order, and as we process each one, we want to know if it is the first claim for a given member (or, conversely, if we have seen that member before in this dataset of claims). As a bonus, let's also keep track of the latest (most recent) claim date for each member. And finally, let's output that information (a list of the unique members we have seen and the latest claim date for each) when we reach the end of the claims. This is in addition to the claims level data, which we will also output to a new dataset.

APPROACH

Unlike the previous examples, we do not have any data to load into the hash table when the Data step starts. Instead, we will create the structure only. As we process each claim, we will check the hash table to see whether the member is in there. In the beginning, no members will be in there. When we don't find a member, we will insert a record in the hash table with that member_id. We will also insert the date of the claim. As we process claims, we will eventually get to one for a member we have seen before. At that point, we do not want to insert a new record in the hash table, but we do want to update the field for the latest claim date. Since the claims are in chronological order, each new claim we encounter will have a date at least the same but more likely later than the date already entered for that member. When we get to the end of the claims dataset, we will output the hash table to a SAS dataset to preserve the information for later processing.

```
data Processed_Claims(drop=rc);

   dcl hash members (ordered:  'a');
   rc = members.DefineKey ('Member_id');
   rc = members.DefineData ('Member_id','Latest_dt');
   rc = members.DefineDone ();

   do until (eof) ;
     set claims end = eof;
     latest_dt = .;
     rc = members.find ();
     if rc eq 0 then do;
                  * insert code to run if we have seen this member before *;
         seen_it = 'YES';
     end;
     else do;
                  * insert code if we have not seen this member before*;
         seen_it = 'NO';
     end;
     latest_dt = svc_dt;
     members.REPLACE();
     output;                                    * output processed claim;
  end;
  members.OUTPUT(dataset:  'Member_latest');  * output member summary;
stop;
run;
```

In this code we see another feature of the DECLARE statement: the ability to specify that the data in the hash table be ordered in ascending sequence (by the key—you may not sort a hash table by anything other than the key). This is useful when the hash table will be output as a dataset.

We also see a new wrinkle in the Define section: the key variable (Member_id) is listed both as the key and as an element of Data. This prepares for creating a dataset from the hash table data. The hash table key is used behind the scenes and does not exist as a data element in the hash table unless specified in the DefineData method.

No data are loaded to the hash table initially. The Claims dataset is processed inside a DO loop (meaning that the Data step will execute in its entirety only once). After each record is read, we initialize the variable latest_dt to avoid holding it over from the last record. Then we try to find the Member_id in the hash table. If the rc eq 0 (meaning the member_id is already in the hash table), we can execute whatever other code would be appropriate for members we have seen before. We set a variable called seen_it to the value 'YES' just to show how the processing works. If we do not find the member_id in the hash table (if rc ne 0), we can execute the code appropriate to that circumstance. We set the variable seen_it to 'NO'. Now, whether we have seen the member before or not, we set the latest_dt to the service date of the current claim and execute the REPLACE method. This will do one of two things: if the member_id already exists on the hash table, it will overwrite the record with the new latest_dt. If the member_id does not already exist on the hash table, it will add a record (with the current member_id and latest_dt) to the hash table.

After the DO UNTIL loop completes, we know we have processed all the records in the claims dataset. At that point, we execute the OUTPUT method on the hash table Members to create a SAS dataset called Member_latest.

The resulting datasets look like this:

Processed_claims				
Obs	Member_id	svc_dt	dx	seen_it
1	164-234	2005/01/01	250	NO
2	297-123	2005/02/03	4952	NO
3	164-234	2005/03/15	78910	YES
4	297-123	2005/04/14	250	YES
5	297-123	2005/08/19	12345	YES
6	164-234	2005/09/13	250	YES
7	297-123	2005/11/01	4952	YES

Member_latest		
Obs	Member_id	latest_dt
1	164-234	2005/09/13
2	297-123	2005/11/01

The variable seen_it allows us to note that the code executed correctly in determining whether each claim was that member's first claim in the dataset. The rows in the Member_latest dataset show that we correctly identified all the unique members and the latest claim date for each.

EXAMPLE 4: CHAIN (UNLIMITED RECURSIVE LOOKUPS)

Hash tables are the ideal solution to this problem. Suppose you have a membership table, such as an insurance company might maintain on its members. At various points in a member's history the unique ID might have changed. For example, in the health insurance industry when federal legislation required that the SSN become a private rather than a public ID, health insurance companies were required to assign new ID numbers to the entire membership. When ID's change, you are faced with the problem of historical data: claims or other transactions that occurred prior to the change have the old ID while later transactions have the new ID. Furthermore, other events can cause member ID changes, so that a given member might have 3, 4, or more IDs over her entire history. How do you process a series of transactions with ID's of an unknown generation on them? As long as you have a database of changes, hash tables provide an easy answer.

Consider the following datasets:

Members			
Obs	member_id	plan_id	group_id
1	164-234	XYZ	G123
2	297-123	ABC	G123
3	344-123	JKL	G456
4	395-123	XYZ	G123
5	495-987	ABC	G456
6	562-987	ABC	G123
7	697-123	XYZ	G456

Old_new_xwalk		
Obs	old_id	new_id
1	164-234	N164-234
2	297-123	N297-123
3	344-123	C344-123
4	N164-234	M164-234
5	N297-123	B297-123
6	M164-234	P164-234
7	P164-234	A164-234

You can see that the Old_new_xwalk shows a chain of ID changes for 164-234. It was changed to N164-234 (Obs 1), then N164-234 was changed to M164-234 (Obs 4), then M164-234 was changed to P164-234 (Obs 6), and finally P164-234 was changed to A164-234 (Obs 7). Although most real cases don't involve this many steps, the primary problem is that you don't know how many steps a given ID might have and it is hard to write code to check "until done".

APPROACH

The code below shows how to use a hash table to solve this problem. The first step is to load the Old_new_xwalk into a hash table. Then we process the records from the Members table, one at a time. For each record with a member_id, we check the hash table for rows where that member_id occurs in the "old_id" position. If we find a row, we have the new_id that it was changed to in the variable called New_id. To find out whether that ID was ever changed, we move that value to the Old_ID position and again check the hash table for a row. If we find another row, it means that the new_id from the first row became the old_id in a new pair later on. We continue this process (putting the New_id in the Old_id position and checking the hash table) until we no longer find a row. When this happens, we know that the value currently stored in the Old_id position does not occur on the crosswalk table with any new_id assigned to it, so we conclude that we now have the latest ID for that person.

```
data members_update(drop=rc );

  if 0 then set old_new_xwalk;                /* sets up the variable types */

  dcl hash xwalk (dataset:  'old_new_xwalk');
    xwalk.definekey ('old_id');
    xwalk.definedata ('new_id');
    xwalk.definedone ();

  do until (eof);
     set members end=eof;
     new_id = ' ';
     old_id = member_id;
     do lookups = 1 by 1 until (rc ne 0 or lookups > 1000);
            /* provides a counter called lookups and stops infinite looping.  See text below. */
      rc = xwalk.FIND();
      old_id = new_id;
    end;
    output;
  end;
stop;
run;
```

The integrity of the crosswalk is key to making this (or any solution) work. For example, if a given old_id occurs twice on the crosswalk assigned to two different New_ids, you have a "split" and it requires business input to decide how you want to treat those. The code shown in this example will not accept two rows into the hash table with the same key (old_id) so splits will not be present at processing time. If your crosswalk contains "merges" [cases where two different old_ids are assigned to the same new_id], this solution will process them correctly and assign all claims under both old_ids to the same new_id. The one problem we have to code around is the potential infinite loop that will occur if there are 2 records that assign A to B and then B to A. This is a business

and data problem, but it will become a processing problem if allowed to sneak through. The code that checks for lookups > 1000 is one way to stop infinite looping caused by this data issue. It would be preferable to clean up your crosswalk before entering this process.

The resulting dataset is shown below:

Members_update						
Obs	old_id	new_id	member_id	plan_id	group_id	lookups
1	A164-234	A164-234	164-234	XYZ	G123	5
2	B297-123	B297-123	297-123	ABC	G123	3
3	C344-123	C344-123	344-123	JKL	G456	2
4			395-123	XYZ	G123	1
5			495-987	ABC	G456	1
6			562-987	ABC	G123	1
7			697-123	XYZ	G456	1

The latest ID for each member is shown in both the Old_id and New_id field because the last step before stopping the lookup is to assign the latest value of the New_id variable to the Old_id field for another lookup. If no value occurs in either field it is because the member_id does not exist on the crosswalk in the Old_id field (meaning that ID was never changed).

The counter in the Lookups field shows how many times we iterated through change before finding a value of New_id that had not been changed. Every member_id is looked up at least once, but if it does not exist in the crosswalk at all, no value appears in the New_id field.

EXAMPLE 5: OUTPUT N DATASETS

In this example, the goal is to split a dataset into N pieces, where N is determined by the number of distinct values in a particular field. In the past this might have been accomplished using macro code: preprocess the data to determine the number of datasets needed, then use macros to write a data statement with N datanames. With hash tables, no preprocessing is necessary. And as a bonus, you can give names to the datasets reflecting the value of the distinguishing variable represented in each dataset. Perhaps a concrete example will help clarify.

Once again, consider our Members dataset.

Members			
Obs	member_id	plan_id	group_id
1	164-234	XYZ	G123
2	297-123	ABC	G123
3	344-123	JKL	G456
4	395-123	XYZ	G123
5	495-987	ABC	G456
6	562-987	ABC	G123
7	697-123	XYZ	G456

You can see that members belong to one of two different groups (G123 or G456). If we know this in advance, we can write code such as:

```
Data group_G123
     group_G456;
 set members;
 if group_id = 'G123' then output group_G123;
 else if group_id = 'G456' then output group_G456;
run;
```

But what if you don't know how many groups there are? We are using simple data for brevity of illustration; the code is the same for the larger situation.

APPROACH

To use hash tables in the way shown below, the Members dataset must be sorted by Group_id. You might consider this a small price to pay for the convenience of the solution. Once the data are sorted, we can process each group using the first.var and last.var automatic variables provided in the Data step for variables named in the By statement. Here is the code, followed by an explanation:

```
proc sort data=members;
  by group_id;
run;

data _null_;
  if 0 then set members;                          /* create variable types */
  dcl hash groups ( ordered:  'a');

  groups.definekey ('member_id','plan_id','_n_');
  groups.definedata ('member_id','plan_id'     );
  groups.definedone ();

  do _n_ = 1 by 1 until (last.group_id);
    set members;
      by group_id;
      groups.add();
  end;
  groups.output (dataset:  compress('GROUP_'||group_id));
run;
```

After the sort, we start a Data step and create a hash table structure. Unlike our previous examples in which each hash table was filled only once, this example shows that you can empty and fill a hash table multiple times during the execution of the Data step. We will read successive records from Members and fill a hash table with all the records for a particular Group_ID. When we encounter the last record for that Group_id, we will output the contents of the hash table to a SAS dataset named using the value in the Group_id field at the time. We then allow the Data step to loop as it normally does, emptying and re-creating the hash table structure before starting to read the records associated with the next value of Group_id.

Once again, this example shows that we have to put all the variables that we want to see in the output dataset in the DefineData method because the key variables are used only to construct the index. They do not exist on the hash table as data fields. In this example, the key actually contains more fields than the data. The reason for this is to allow multiple rows with the same member_id and plan_id. Remember that hash tables cannot have more than one row with the same key value. Unique keys are essential to the construction that delivers the amazing efficiency. Adding the _n_ variable to the key enforces uniqueness of key while allowing multiple rows with the same member_id and plan_id in the incoming (and outgoing) dataset.

This would be a good time to discuss the heavy use of quoting in specifying parameters to the methods. The benefits of this are shown in the **`groups.output`** statement. All the methods are set up to accept variables as well as hard-coded values as parameters, allowing for data-driven code. In the case of the **`groups.output`** we can specify that the name of the dataset to be created is GROUP_ plus whatever value is in the group_id field at the time the statement is executed. This flexibility of allowing variables to be parameters requires that you quote anything you are hardcoding, to distinguish those from variable names (the same reason you have to code A="YES" instead of A=YES to distinguish the value YES from the variable name YES).

The resulting datasets are shown below:

GROUP_G123		
Obs	member_id	plan_id
1	164-234	XYZ
2	297-123	ABC
3	562-987	ABC
4	395-123	XYZ

GROUP_G456		
Obs	member_id	plan_id
1	344-123	JKL
2	495-987	ABC
3	697-123	XYZ

You can see in the code that we specified only Member_id and Plan_id as fields on the hash tables; therefore, those are the only fields that show up in the output datasets. We could have kept Group_id to show that only Group_id = "G123" show up in the dataset GROUP_G123. All we would have to do is add Group_id to the DefineData method statement.

EXAMPLE 6: OUTER JOIN

The last example is included not so much because hash tables are always the right tool to use in this situation, but to respond to questions about whether a hash table can be used when an outer join is needed. Recall that an outer join of 2 datasets contains the union of all records in the datasets. It matches the records where the key value exists in both datasets, and it also includes records from each dataset where the key values do not appear in the other dataset. This example also introduces the only other object currently available in the Data Step Component Object Interface: the hash iterator object, or hiter.

Consider the following datasets. One contains a list of members along with their Plan_id. The other contains members and their Group_ids. We want to build one dataset with all the members that exist in either table, along with their Plan_ids and Group_ids.

Mbr_plan		
Obs	Member_id	Plan_id
1	164-111	XYZ
2	297-111	ABC
3	344-111	JKL
4	395-111	XYZ
5	495-111	ABC
6	562-111	ABC
7	697-111	XYZ

Mbr_group		
Obs	Member_id	Group_id
1	164-111	G123
2	297-888	G123
3	344-111	G456
4	395-111	G123
5	495-111	G456
6	562-888	G123
7	697-111	G456
8	833-888	G456

Can we do an outer join on Member_id using hash tables? But of course!

APPROACH

We will start by loading one dataset (Mbr_group) into a hash table, adding a column for flagging whether that row in the hash table matches a row from the Mbr_plan dataset. We then process the Mbr_plan dataset one row at a time, looking it up in the hash table and flagging the rows we find. When we are completely finished with Mbr_plan, we use the hash iterator object to process the rows in the hash table (Mbr_group), and write to the output dataset the rows that were not flagged as previously found.

Here is the code:

```
data all ;* (drop=_:);                  /* use this form of drop= if desired */
  if 0 then set mbr_group;              /*  set up variables */

  dcl hash hh   (ordered: 'a');
  dcl hiter hi  ('hh');                 /*  hash iterator object  */
  hh.definekey  ('member_id');
  hh.definedata ('member_id','group_id','_f');
  hh.definedone ();

  do until (eof2);
    set mbr_group end = eof2;
```

```
      _f = .;                          /*  initialize flag _f in hash table */
      hh.add();
  end;
* Output all rows in mbr_plan *;
  do until(eof);
    set mbr_plan end = eof;
    call missing (group_id);           /*  initialize group_id for each new record in Mbr_plan*/
    if hh.find() = 0 then do;
      _f = 1;
      hh.replace();                    /*  overwrite record in hash table with new value of _f */
      output;
      _f = .;                          /*  initialize _f back to missing to prepare for next */
    end;
    else output;
  end;
                                       /*  at this point all Mbr_plan records have been read */
* Output remaining rows *;
  do _rc = hi.first() by 0 while (_rc = 0);
    if _f ne 1 then do;
      call missing (plan_id);          /*rows in the hash table will not have values for plan_id
*/
      output;
    end;
    _rc = hi.next();                   /*  move to next row in hash table */
  end;
  stop;
run;
```

We set up a Data step to write a dataset called ALL. The drop= option is commented out but can be used as a convenience to drop all variables that start with an underscore. This will drop the variable _f from the dataset ALL. If you get in the habit of naming all your temporary variables with an initial _, you can use this form of the drop all the time.

The statement "**`if 0 then set mbr_group;`** " establishes the variables in the dataset Mbr_group in the program data vector without actually reading any records. This is important so that when the hash table is declared, SAS knows what type and length each of the variables should be. A hash table named hh is created, with the specification that the rows be ordered in ascending sequence by the key value. Immediately afterward, a hash iterator object named hi is declared pointing to the hash table hh. The iterator object allows the use of methods like .first() and next() seen in the last DO loop.

Once the hash table is declared, we fill it with the records from Mbr_group (the "do until (eof2)" loop). Note that we are assuming there are no duplicates on Member_id in either table. The method hh.add() will reject any row with a key value equal to an existing row.

The next DO loop (do until (EOF)) processes the records from Mbr_plan, one at a time. For each record, group_id is initialized to avoid carrying forward the value from the last successful hh.find(). The statement "**`if hh.find() = 0 then do;`**" both executes the find() method and tests the result for success. If it is successful (if the member_id on the record we just read from Mbr_plan exists on the hash table), the value of group_id from the hash table will be brought into the Data step variable group_id. We set the flag _f to 1, signaling that we have found that record, and .replace() that row in the hash table. Since the current values of Member_id and Group_id in the Data step match what we just brought from the hash table, the only change the .replace() method accomplishes is to set the flag _f to 1 in the hash table. Once we do that and output the record to ALL, we reset the flag _f back to missing to prepare for reading the next record. If the rc from executing the .find() method on this record is not 0, meaning that member_id does not exist in the Mbr_group table, all we have to do is output the record to ALL (since we want records from either dataset, whether or not it matches a record in the other dataset).

After the completion of the **`do until (eof)`** loop, we have finished reading all the records from Mbr_plan (and writing them to ALL). The last step is to take advantage of the methods available to the hash iterator object. These methods access the hash table that the iterator object is linked to (hh) and move a pointer through its

rows, bringing the values from the hash table into the SAS program data vector. They are then ready for output to the SAS dataset. We can check each row of the hash table sequentially to see whether it was matched to a record from Mbr_plan. If not, we can output it to ALL. If it is flagged, we know it already exists in ALL and there is no need to output it.

At the bottom of the data step, we know we have output all the records in Mbr_plan and all the records in Mbr_group to ALL, matching those with the same Member_id. Voilà, an outer join via hash tables!

CONCLUSIONS

In this paper you have seen a variety of applications for the new Data Step Component Object Interface tools called hash tables. This small set just scratches the surface of their potential utility. Hash tables deliver amazing efficiency for remarkably little investment in coding. They can be used in a variety of circumstances, and they solve at least one heretofore intractable problem. Limited only by available RAM and your imagination, these new Data Step Component Objects have revolutionized data processing. This paper provides you with the means and the motivation to use hash tables. Why not try them out today?

REFERENCES

Dorfman, Paul and Vyverman, Koen. 2005. "Data Step Hash Objects as Programming Tools." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Available **http://www2.sas.com/proceedings/sugi30/236-30.pdf**

Loren, Judy and Gaudana, Sandeep, 2005. "Join, Merge or Lookup? Expanding your toolkit." *NorthEast SAS Users Group, Inc.18th Annual Conference Proceedings.* Available **http://www.nesug.org/html/Proceedings/nesug05.pdf**

DISCLAIMER

ACKNOWLEDGMENTS

This paper and the author owe an enormous debt to Paul Dorfman, who showed infinite kindness and patience to someone learning not just about hash tables but about various intricacies of the Data step.

Thanks also to developers at SAS Institute for consultation on specific topics. Any errors in this paper are those of the author not of the information suppliers.

CONTACT INFORMATION

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

Judy Loren
Health Dialog Analytic Solutions
2 Monument Square
Portland, ME 04101
JLoren@healthdialog.com