I cut my processing time by 90% using hash tables - You can do it too!
Jennifer K. Warner-Freeman, PJM® Interconnection, Norristown, PA

ABSTRACT
Hash tables are an exciting new feature in SAS® 9. Their use can achieve great I/O efficiencies and enormous time savings when merging tables with the DATA Step. Besides that, they are easy to use! The motivation for writing this paper came when, after incorporating hash tables, a frequently run process went from taking two hours to complete to taking a mere even minutes. This allows my team and I the luxury of doing more analysis in less time and the ability to ask and answer many more “what if” questions.

This paper will:
- introduce hash tables with a brief discussion of the benefits of their use
- describe hash table use with a straight-forward example
- compare hash tables to other merging and lookup techniques and discuss memory issues
- define a more complex business scenario where hash tables are coupled with macro variables and looping in order to perform more flexible joins based on dynamic by-variables

This paper is appropriate for any intermediate level SAS programmer who wants to return to work and impress their co-workers and command the amazement of their bosses with how much faster their processes run.

INTRODUCTION
Hash tables are a great and easy to use tool introduced in SAS 9. If the use of hash tables offered no benefit other than speeding up merges, they would still be a welcome addition to any programmer’s toolbox. So why is it that so few people are using them? I think that there are a few reasons that can explain this.

The first, and probably most prevailing reason people aren’t using hash tables is lack of awareness. If you are reading this paper then that probably isn’t an issue for you. However, despite being what I thought was “on top of things” when it came to my SAS programming skills, I had not heard of hash tables being used in SAS until about a year ago. I had heard of hash tables and hash functions, but I didn’t have any idea what they actually were and I certainly didn’t know I could use them in SAS.

The second reason I think more people aren’t using hash tables is the syntax for their use is different from the traditional procedural SAS syntax. Hash tables employ the syntax of objects and methods and attributes (oh my!). That alone is enough to make any procedural programmer think twice about abandoning their nice familiar PROC SORT with a DATA step merge or PROC SQL steps.

Third, there are many excellent sources out there on the topic of hash tables. However, many of them get into levels of detail that might be overwhelming for someone who just wants to try the darn things out without having to get into some of the more technical details. I am not a computer scientist. I wouldn’t even call myself a programmer. I have a background in economics, analytics and forecasting and I use SAS as a tool. I get intimidated when I start reading things about binary trees and for some reason reading the words “associative arrays” in relation to what I am trying to learn about makes me lose all hope for ever understanding the topic.

Last, while many sources have great examples of how to use hash tables with simple scenarios, few offer examples that tackle more complicated joins or merges. For me, that is where the power of this tool really lies. I know that SAS programmers are very clever folks, but sometimes having an example to work from, especially when attempting something in uncharted waters, can be terribly helpful.

To address the reasons above, this paper will familiarize the reader with what hash tables are, what the syntax means and how to use hash tables in simple, as well as more complex, scenarios. At the same time, it will attempt to avoid information overload. I believe hash tables are a wonderful timesaver. My goal is to make hash tables as accessible to the general SAS user as they are to seasoned SAS pros who have known about them and have been manually coding their own for years.

This paper focuses on how to use hash tables for faster merges. Other things you can do with hash tables will also be discussed briefly. This paper will not discuss the genesis of hash tables, the fundamentals behind them or how programmers manually make their own. Instead, this paper is a gentle introduction and practical guide for anyone who wants to dip their toe into the hash table pool to see how the water feels before jumping in.
BACKGROUND

So what are hash tables and why are they, in many cases, faster than using other merging techniques? A hash table is a collection of records, typically from a SAS dataset, that is loaded into memory, instead of on disk, and is available for use by the DATA step that created it. Hash table records have unique keys and take advantage of direct addressing instead of sequential addressing. Umm...OK, so what does that really mean?

Let's start with loading a dataset into memory. This is analogous to doing a research paper the really old-fashioned way – with actual books at the library. Loading the dataset into memory would be like going to the shelf, grabbing all the books on your research topic, putting them onto a library cart and rolling them over to the table or desk where you are working. The books, collectively, are like the dataset, with individual books being like records in the dataset. The library cart is like memory. Now instead of having to get up and walk over to the shelf (disk) each time you want to examine another book (record) from the collection of books that the library has on the topic (dataset) you can simply turn to the cart (memory) which is right at your fingertips.

Next, direct addressing as opposed to sequential addressing can also be analogous to our library example. When datasets are merged or joined it is typically done on a by, or key, variable. I have a quote (key value) that I found on the internet somewhere. I know it is from one of the books (records) in the library’s collection of books on my topic (dataset). I want to go to the book (record) that the quote came from and get some more information (another variable) from it. If all I have is the quote, the only place I have to go is through each book (record) until I find it. Maybe the quote will be in the first book (record) but maybe it will be in the last. SAS is doing the same thing when it goes through each record, one by one, until it finds a match for the key value. This is sequential addressing. If only I had the book title, author and page number I could go directly to the quote. SAS is doing the same thing when it keeps track of the key values on the hash table records, where they “live” in memory and can go directly to them when it needs to. This is direct addressing.

Thus, having the data right at our fingertips in memory and knowing exactly where to find each observation when we need to access it will greatly speed up most merges. Additionally, when using direct addressing, there is no need to sort any of the datasets prior to merging. When merging large or multiple datasets, just not having to sort can be an enormous time savings all by itself.

A SIMPLE EXAMPLE

Now that we know why hash tables can be faster for merging, let’s jump right in and see how to actually use them. For the purposes of the examples in this paper, suppose that I am an ice cream vendor. I have trucks and stands where I sell ice cream. I keep track of my sales for each channel (trucks and stands), as well as calendar and weather variables. I currently operate in only one zone or territory, but am planning to expand my operations to two additional zones for the purposes of the next example. I want to build an econometric model that will help me identify what factors most affect sales. To do this, I must first merge three datasets (sales, calendar and weather).

Below is an example showing the structure of the three tables I need to merge. The common key between all three datasets is the date. Note, however, that the key field, date, is named something different on all three datasets. Also, none of the three datasets are sorted the same way.

<table>
<thead>
<tr>
<th>Sales</th>
<th>Calendar</th>
<th>Weather</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sales_Date</td>
<td>Stand_ Sales</td>
<td>Truck_Sales</td>
</tr>
<tr>
<td>7/1/2007</td>
<td>$1,023.00</td>
<td>$879.00</td>
</tr>
<tr>
<td>7/2/2007</td>
<td>$954.00</td>
<td>$1,045.00</td>
</tr>
<tr>
<td>7/3/2007</td>
<td>$1,053.00</td>
<td>$857.00</td>
</tr>
<tr>
<td>7/4/2007</td>
<td>$1,143.00</td>
<td>$988.00</td>
</tr>
<tr>
<td>7/5/2007</td>
<td>$785.00</td>
<td>$859.00</td>
</tr>
<tr>
<td>7/6/2007</td>
<td>$732.00</td>
<td>$811.00</td>
</tr>
<tr>
<td>7/7/2007</td>
<td>$426.00</td>
<td>$398.00</td>
</tr>
</tbody>
</table>
To merge these datasets with hash tables, the syntax would look like this:

```sas
data sales_cal_weath (drop = date DT);
  if 0 then set lib1.sales lib1.calendar lib1.weather;
  if _N_ = 1 then do;
    declare hash cal(dataset:'lib1.calendar');
    cal.defineKey('Date');
    cal.defineData('Day_of_week', 'Holiday_indicator');
    cal.defineDone();
    declare hash w(dataset:'lib1.weather');
    w.defineKey('DT');
    w.defineData('Temp', 'Precip');
    w.defineDone();
  end;
  set lib1.sales;
  if cal.find(key:sales_date) = 0 and w.find(key:sales_date) = 0
    then output;
run;
```

OK – that doesn’t look so friendly does it? Let’s look at each piece individually. I will describe each of them briefly here and then get into more detail about steps 2, 3 and 4 in the following section.

1. **Data, set and run** - Those three statements should look familiar and are just as you would expect. In this case we are creating a dataset named sales_cal_weath and we are using lib1.sales as our “base” dataset. A subsequent section of this paper will discuss how to decide which tables should be base tables and which should be hash tables.

2. **Load properties of variables in the hash tables** – SAS needs to know about the variables that are present in a hash table before it can use them. You can either use a LENGTH statement to define the variables, their types and lengths, or simply use a SET statement like this one. This SET statement will be read at compile time, information about the variables in the datasets will be extracted, but the statement will never actually execute due to the “if 0” condition.

3. **Declare hash tables** – This is where you tell SAS what tables you want loaded into memory. You are naming your in-memory locations, telling SAS what datasets to load into them, what unique key variables you will be using to join with and what other variables from the datasets you want to have available for use in memory.

4. **Join to hash tables** – Here, the DATA step is reading observations from the sales dataset and using the variable specified, sales_date, to join to the hash table. You already defined the “key” variable (Date and DT, respectively) on each hash table’s declaration. SAS now attempts to find matches between sales_date and whatever variable was specified in the defineKey statement from the hash table declaration. The if condition in this statement will result in an inner join – in other words, only matching observations between the three datasets will be on the output dataset.
A MORE DETAILED EXPLANATION

STEP 2. \texttt{if 0 then set lib1.sales lib1.calendar lib1.weather;}

What is the need for this strange looking set statement? First, a little background about the DATA step and how it interacts with the hash table objects. In simple terms, to the DATA step, the hash object is like a black box. The DATA step knows it is providing some information to the black box, the black box does something with the information, and the DATA step should expect something back from the black box. It really doesn’t know what that something may be. Thus, if your hash table is returning some variables, and the DATA step doesn’t know about those variables and have slots set up for them in the program data vector (PDV), then you are going to get a grumpy error message in your log like the one at the bottom of this page.

There are a few ways to tell the DATA step about variables that are on the hash tables you will be declaring. The first is a LENGTH statement. Often times, programmers use LENGTH statements before SET statements to control the ordering of variables on the output dataset among other reasons. The LENGTH statement tells SAS to set up a place in the PDV for the variables, what type of variable will occupy the space and how long the variable will be. The LENGTH statement does not actually create the values, it only saves a place for them. The actual values get created with assignment statements later on in the DATA step. Thus, if I ran this DATA step, where I set up a spot for a variable “Celsius”, but never actually create, or initialize it, then I would get the following result and message in my log.

\begin{verbatim}
data weather2;
    length celsius 8.;
    set weather;
run;
\end{verbatim}

\textbf{NOTE: Variable celsius is uninitialized.}

The same is true when using a LENGTH statement with hash tables. If I had changed my code to include the LENGTH statement below, instead of using the \texttt{if 0 then set...} from above, then I would get the following messages in my log. This is because SAS doesn’t know that these variables are being initialized in the hash object. These aren’t errors and the output table still has all of the correct information on it, but sometimes people don’t like all those scary messages in the log.

\begin{verbatim}
length date
    holiday_indicator
    DT temp precip 8.
    day_of_week $9.;
\end{verbatim}

\textbf{NOTE: Variable date is uninitialized.}
\textbf{NOTE: Variable holiday_indicator is uninitialized.}
\textbf{NOTE: Variable DT is uninitialized.}
\textbf{NOTE: Variable temp is uninitialized.}
\textbf{NOTE: Variable precip is uninitialized.}
\textbf{NOTE: Variable day_of_week is uninitialized.}

To avoid these messages in the log, you can use a CALL MISSING routine. The CALL MISSING can go anywhere in the code following the LENGTH statement and prior to the RUN statement. However, it seems to be common form to place the statement immediately following the \texttt{defineDone()} statement in the hash object declaration. You can also specify a NONOTES system option, but that will turn off all notes to the log until you specify NOTES to turn them back on and that may not be desirable.

\begin{verbatim}
call missing (date, day_of_week, holiday_indicator, DT, temp, precip);
\end{verbatim}

In addition to those pesky notes, or the need to use a CALL MISSING routine to avoid them, one must use caution with a LENGTH statement if it is to work correctly. If I had accidentally left \texttt{day_of_week} (or any other variable from either of my hash tables) off of the LENGTH statement, then I would get an error like the one below. SAS didn’t know anything about the \texttt{day_of_week} variable on the calendar table and consequently, it didn’t know to set up a spot in the PDV for it and instead it gave me an error message. I would get the same message if I had misspelled one of the variables and would get a similar error if I had specified the wrong length or type for a variable in the LENGTH statement.

\textbf{ERROR: Undeclared data symbol Day_of_week for hash object at line 730 column 7.}
\textbf{ERROR: DATA STEP Component Object failure. Aborted during the EXECUTION phase.}
In addition to the notes in the log about uninitialized variables and potential for human error, using a LENGTH statement has a few other limitations. What if you have 500 variables on your hash table and don’t want to do all that typing? Not to mention, your program won’t be flexible because of all of those hard coded variable names.

To solve all of these issues, you can use a SET statement. The SET statement is a compile time statement. Thus, when the code compiles, SAS will read this statement and think to itself, “Oh, when I execute this block of code the user is going to want to read in some datasets. I better set up some room in the program data vector for the variables in those datasets.”

So what about the if 0? In this conditional test, a value of 0 represents false. The statement is saying “If false then set...”. If what is false? It doesn’t matter what “it” is. “It” will never be false and the statement will never execute. Thus, SAS will never actually set these datasets together (nor do you want it to) at execution, but at compile time it will set up the slots for the variables in the PDV.

In addition to if 0 then set dsn;, you can also use

set dsn point = _n_;
or set dsn (obs = 1);.

However, set dsn (obs = 0); will not work.

This method is not without its own drawbacks as well. First, setting the dataset will set up slots in the PDV and on the output dataset for every variable in the dataset. You may not want all of those variables on your output dataset. This can be controlled with a DROP= or KEEP= in your DATA statement. Also, if you do want all of those variables, but don’t include all of them in the defineData method, then they will be missing in the output dataset.

For example, if I ran my DATA step without this DROP = statement (drop = date DT), then the resulting table would look like the example below.

```
data sales_cal_weath;
  if 0 then set lib1.sales lib1.calendar lib1.weather;
  ...
```

Note that there are columns for Date and DT. This is because those variables are on the datasets that were specified in the SET statement. But there are no values in these columns. This is because while I specified these two variables as key fields in the defineKey method, I did not specify them in the defineData method. Consequently, they did not get initialized in the hash object and are missing in the output.

Last, you may have noticed that I put the sales table on this SET statement. This was not necessary and the only reason I did it is because I wanted those columns to be the left most columns on my output dataset. Had I not put that table on this set statement then those columns would have been the last columns on the table. Again, this is due to what actions take place at compile time as opposed to execution. Had I left the sales table off of that SET statement, the results would look as they do below. Note that I did add the DROP = back in.

```
data sales_cal_weath (drop = DT date);
  if 0 then set work.calendar work.weather;
  ...
```

```
<table>
<thead>
<tr>
<th>Sales Date</th>
<th>Stand Sales</th>
<th>Truck Sales</th>
<th>Date</th>
<th>Day_of_week</th>
<th>Holiday Indicator</th>
<th>DT</th>
<th>Temp</th>
<th>Precip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 01/01/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
<td>Sunday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 01/02/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
<td>Monday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 01/03/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
<td>Tuesday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 01/04/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
<td>Wednesday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 01/05/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
<td>Thursday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 01/06/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
<td>Friday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 01/07/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
<td>Saturday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

For example, if I ran my DATA step without this DROP = statement (drop = date DT), then the resulting table would look like the example below.

```
data sales_cal_weath (drop = DT date);
  if 0 then set work.calendar work.weather;
  ...
```

```
<table>
<thead>
<tr>
<th>Day_of_week</th>
<th>Holiday Indicator</th>
<th>Temp</th>
<th>Precip</th>
<th>Sales Date</th>
<th>Stand Sales</th>
<th>Truck Sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td>01/01/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
</tr>
<tr>
<td>Monday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td>01/02/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
</tr>
<tr>
<td>Tuesday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td>01/03/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
</tr>
<tr>
<td>Wednesday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td>01/04/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
</tr>
<tr>
<td>Thursday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td>01/05/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
</tr>
<tr>
<td>Friday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td>01/06/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
</tr>
<tr>
<td>Saturday</td>
<td>6</td>
<td>75</td>
<td>0.25</td>
<td>01/07/2007</td>
<td>$1,000.00</td>
<td>$1,000.00</td>
</tr>
</tbody>
</table>
```
**STEP 3. if _N_ = 1 then do;**

```sas
declare hash cal(dataset:'lib1.calendar');
cal.defineKey('Date');
cal.defineData('Day_of_week', 'Holiday_indicator');
cal.defineDone();
```

This is where you are telling SAS what tables you want loaded into memory. You are naming your in-memory locations, telling SAS what datasets to load in them, what the unique key variables are and what other variables from the datasets you want to have in memory.

a. **if _N_ = 1 then do;** Hash tables only need to be loaded once. The Declare statement is an executable statement and not a compile-time statement. If you didn’t have this conditional statement to load them only on the first iteration of the DATA step then the tables would be reloaded with each iteration. That would more than negate any benefit you would gain by using hash tables in the first place.

b. **declare hash cal(dataset:'lib1.calendar');**

i. In this statement, the key word `declare` is telling SAS that we are going to declare something.

ii. The word `hash` tells SAS that it is a hash table object that is being declared. You can also declare hash iterator objects, however, iterator objects are not discussed in this paper.

iii. Once you tell SAS that you want to declare a hash table object, you must tell it what you want to call it. This will be a human-readable alias, in this case, `cal`. You can name your hash table objects whatever you wish as long as you stay within the bounds of SAS variable naming conventions i.e. the name must be 32 characters or fewer and begin with a letter or underscore.

iv. Now that we told SAS to set up a space in memory for a hash table that we will refer to as `cal`, we need to instantiate the hash table by actually putting something in it. In this example, we are loading a SAS dataset, `lib1.calendar`, into the hash table. In parentheses, we tell SAS that we are loading a dataset by using the word `dataset:` followed by a colon. Then, in quotes, simply specify the dataset name - `lib1.calendar`.

As with other SAS procedures and data steps, if a libref is left off, SAS will assume that the dataset resides in the work directory. Additionally, data can be loaded to a hash table by manually typing it in. An example of that will be shown in the More Hash Object Methods section of this paper. You can also supply macro variables as arguments provided they are enclosed in double quotes – `(dataset:"&dsn")`.

v. **declare hash cal(dataset:'lib1.calendar', hashexp:'10');**

You can also add a second argument, `hashexp`, to specify the internal hash table size. According to the SAS Help and Documentation, "The value of hashexp is used as a power-of-two exponent to create the hash table size. For example, a value of 4 for hashexp equates to a size of 2^4, or 16. The maximum value for hashexp is 16, which equates to a hash table size of 2^16 or 65,536."

The hash table size is not equal to the number of items that can be stored. Think of the hash table as an array of containers. A hash table size of 16 would have 16 containers. Each container can hold an infinite number of items. The efficiency of the hash tables lies in the ability of the hash function to map items to and retrieve items from the containers.

In order to maximize the efficiency of the hash object lookup routines, you should set the hash table size according to the amount of data in the hash object. Try different hashexp values until you get the best result. For example, if the hash object contains one million items, a hash table size of 16 (hashexp = 4) would not be very efficient. A hash table size of 512 or 1024 (hashexp = 9 or 10) would result in better performance." I was able to trim another minute off of my own process by experimenting with this argument.
c. `c. cal.definekey('Date');` We have now told SAS that we want it to make us a hash table object that we will refer to as `cal` and it will hold observations from the dataset `lib1.calendar`. At this point, we still aren’t finished. SAS needs a little more information. First, what has this dataset keyed on? In other words, what variable, or combination of variables, will identify a unique observation? This could be something like social security number, part numbers, or a composite key like date and hour or area code and phone number.

i. Because `cal`, the hash table object we just declared, is an object, it has attributes and methods associated with it. In simple terms, imagine a dog as an object. Things like the color of its fur and the length of its tail would be attributes. What it can do, like sit or roll over, are its methods. Another example may be a `%include` program and some macros that are defined within it. In another program you might include the first program and then make calls to that program’s macros. This would make the `%include` program like the object and the program’s macros like the object’s methods.

The hash table object has methods as well. For one, it can assign one of its variables to be the key variable using the `definekey` method. Instead of saying “Rover, come!” or `%include progl.sas; %example_macro(parm);`, you would use the syntax

```sas
object.method('parameter')
```

to tell the hash table object that you want it to define a particular variable as its key. Ensure that the variable name is in quotes.

ii. If my weather table was keyed on an hourly level, instead of on a daily level of detail, the statement would look like this: `cal.definekey('Date', 'Hour');` Enclose each variable name in quotes and separate them with commas.

d. `d. cal.defineData('Day_of_week', 'Holiday_indicator');` In addition to using a method to tell SAS what variable from the dataset is to be the key variable, we need to use another method to tell SAS which variables from the dataset we want to be loaded into the hash object.

i. To tell the hash table object which variables it should include from the dataset, use the `defineData` method. Enclose each variable name in quotes and separate them with commas.

ii. It should be noted that just because you defined a variable as your key (as was done with `Date` above) this does not mean that it will be included on your output dataset. If you want your key to be included as well, then you must specify it in the `defineData` method call. However, because it is a key, most times you will already have that variable on your base dataset and won’t need to have it twice.

iii. To include all of the variables in a dataset without having to type them all, the syntax would look like this: `cal.defineData(all:'YES')`

e. `e. cal.defineDone();` `DefineDone` simply completes the initialization of the hash object. No arguments are required between the parentheses, but the parentheses are still needed.

**STEP 4.**

```sas
if cal.find(key: sales_date) = 0
and w.find(key: sales_date) = 0 then output;
```

Now that we have declared our hash table objects and instantiated them by putting some data into them, how do we actually use them? To use them, we need another method. This time we will be using the `Find` method.

There are a few additional methods that will be discussed in the More Hash Object Methods section of this paper.

a. With each iteration of the datastep, the value of `sales_date` from the sales table (or whatever table is specified on the set statement) is being plugged into the `Find` method call as a parameter or argument. The method takes the value of `sales_date` from the sales table and looks for a match on the calendar table. The method knows to look for a match in the `Date` field on the Calendar table and the `DT` field on the weather table because these are the fields that we specified as keys in our hash table declaration.
b. If the key field was called date (instead of sales_date) on the sales table, the same as it is on the calendar table, then you could shorten the statement to look like this.

```plaintext
if cal.find() = 0 and w.find(key:date) = 0 then output;
```

Because the key field is named date on both the sales and the calendar table, you don’t need to specify it in the Find method for the calendar table. However, the date field is still named DT on the weather table so it is still necessary to specify the key to use from the sales table in the Find method for the weather table. This is because the key fields are not named the same on both tables.

c. You will notice that the condition here is `if 'condition' = 0`. Usually 0 means false. It would seem that we are saying if we don’t find a match then output the record. That is not the case for this method. The Find method has return codes. A return code of 0 means that a match was found. If you are certain that there is a match for every value on your base table in your hash tables then you can leave the if / then logic off and shorten the code to this:

```plaintext
cal.find(key:sales_date); w.find(key:sales_date);
```

d. You should know your data well before using hash tables to avoid unintended results. If you use the code above and don’t have a match on your hash table for every value on your base table, you will get an error. As an example, let’s say that my calendar table only had odd numbered days on it. When I go to join to it from my sales table, which has all days on it, I get the error messages in the log like this one: **ERROR: Key not found** and my output table does not get created.

To remedy this, the code at the top of this page outputs only matching records. If I want all observations from my base table on my output table, I might try something like the code below which forces matches and non-matches to be on the output dataset. However, as you can see, that causes problems as well. If the Find method doesn’t find a match, it fills in the value from the last observation. For example, for calendar data on July 20th the output dataset has data from July 1st which instead you would expect the values to be missing since no match was found on the calendar table.

```plaintext
w.find(key:sales_date);
if cal.find(key:sales_date) = 0 then output;
if cal.find(key:sales_date) ne 0 then output;
```

To prevent that from happening, use the return codes to test for a match. In the code below, if no match is found on the calendar hash table, then the value of the variable will be set to missing.

```plaintext
if cal.find(key:sales_date) = 0 then do;
  day_of_week = day_of_week;
  holiday_indicator = holiday_indicator;
end;
else do;
  day_of_week = .;
  holiday_indicator = .;
end;
```

e. If you are merging with a composite key made up of multiple variables, the syntax would look like this:

```plaintext
Cal.find(key:sales_date, key:sales_hour);
```

Note that the Find method does not require its arguments to be in quotes. The same would be true if a macro variable were used as an argument like this: `Cal.find(key:&var1, &var2);`

f. Finally, if you have the same variable on your base table and one of you hash tables, and the variable is specified in the DefineData method, values from the hash table will overwrite values on the base table.
COMPARISON TO OTHER LOOKUP OR MERGING TECHNIQUES

While hash tables are a great tool, and have the potential to be much faster for merges, there is a drawback to them as well. Hash tables are very memory intensive. Thus, choosing the right lookup or merging technique is somewhat dependent on your system constraints. Some techniques listed below are not memory intensive at all, but instead are very CPU or I/O intensive.

In addition, there are tradeoffs between programmability and run-time efficiency. Some techniques may be very fast and efficient when it comes to CPU usage or I/O savings. However, if you only need to do something once, or on an infrequent ad-hoc basis, then it might make more sense to code what is most efficient in terms of programmer time and not CPU time. I can still write up a PROC SQL step faster than I can a DATA step with hash tables. Because my time costs my company more than the CPU’s, unless I am going to run the code over and over again, sometimes it will make more sense to do what takes me less time. Below is an outline of some benefits and drawbacks of the more popular merging and lookup techniques.

SORT AND MERGE WITH BY (MATCH MERGING):
+ This method is one that all SAS programmers are familiar with and is typically easy to understand and read.
+ Not memory intensive because only the current observation from each input dataset is loaded into memory.
+ Joins can be conditionally based on if statements.
- Data sets must be indexed or sorted.
- By variables must have the same name.
- You can only merge on one by variable, or set of by variables, in each DATA step.

PROC SQL JOIN:
+ Not very memory intensive.
+ Does not have any of the same negatives listed for the DATA step merge.
+ Allows flexibility in join types (inner, outer, full, union etc.).
- Slower than in-memory look-up techniques.
- May not be as fast as DATA step for some kinds of merges.
- Syntax is not as easy to understand if one is not familiar with SQL.

PROC FORMAT:
+ In-memory technique that is faster than DATA step merge or PROC SQL join.
+ Once created, formats can be used in other procs.
+ Lookup key can be numeric or character.
- Large memory requirement to load entire format into memory.
- Key can only be one variable, i.e. no composite keys.
- Formats require more disk space than datasets for storage.

TEMPORARY ARRAY:
+ Fastest in-memory technique. Even faster than hash tables.
+ Can use non-sorted and non-indexed base datasets.
+ Can use mathematical expressions for array element lookup.
- Not intuitive to code.
- Large memory requirement to load the entire array.
- Can only return a single value from the lookup operation.
- Lookup value can only be numeric.
- Array is a compile time construct and can’t be used in if/then conditions.

HASH TABLE:
+ No sorting or indexing required.
+ Can join to multiple tables, using different keys, in one step.
+ Can return multiple values from one lookup operation.
+ Lookup values can be character or numeric.
+ Faster than all techniques except temporary arrays.
- Not intuitive to code (until you practice a few times!).
- Large memory requirement to load entire table.
MEMORY USAGE
With all this talk about memory usage, you may be wondering if you need to worry about it. If so, how does one determine how much memory their potential hash table will use or how much memory is available to use. If you calculate that you don’t have enough memory for your hash table, what are some ways to make it smaller? If you have attempted to use hash tables and have memory-related questions, this section will attempt to address them for you.

ESTIMATE AVAILABLE MEMORY
If you attempt to load a hash table for which you have insufficient memory you will get an error. According to http://support.sas.com/techsup/unotes/SN/016/016920.html, “The following messages will be written to the SAS log when you run out of memory when adding keys and associated data items to a hash object.

FATAL: Insufficient memory to execute data step program. Aborted during the EXECUTION phase.
NOTE: The SAS System stopped processing this step because of insufficient memory.”

To determine approximately how much memory is available, you can run the following code which will report, in bytes, your available memory.

```
data _null_;  
mem = getoption('xmrlmem');  
put mem=;  
run;```

ESTIMATE HASH TABLE SIZE
Once you know how much room you have to work with, the next step is determining how much room your hash table will need. The size of a hash table can be estimated by multiplying the record size by the number of records. The record size is simply a sum of the number of bytes in each variable in the record. If the record has an 8 byte customer number key, a 30 byte name field and an 8 byte age field, then the total record size would be approximately 46 bytes. If there are 100,000 records to be loaded into the table, then the approximate size of the hash table will be 4,600,000 bytes or 4.6MB.

You can also take advantage of two system options to aid in diagnosing a potential memory issue. The first, MSGLEVEL = I, can be used when attempting to load the full set of records. Note, the message level is the letter I and not the number 1. This option will report the number of items added to the hash table when the shortage occurred. To turn off this additional information, use MSGLEVEL = N.

The second option, FULLSTIMER, would be more helpful if you are experimenting with different sized subsets to see how small the subset needs to be before you will have sufficient memory. FULLSTIMER will report the memory usage of the DATA step to the log. Thus, if you can successfully load ¼ of your total records, you could multiply the value reported by FULLSTIMER by 4 and get an approximation of the memory needed for the full table. To turn this option off, use NOFULLSTIMER.

TIPS FOR MAKING HASH TABLES SMALLER
Of course, knowing how much memory your hash table needs doesn’t really matter if you are like most of us and have no control over how much memory is on the machine that you use. Some people may not even be able to change system options, like MEMSIZE, that control how much memory is available to their sessions and thus is also available for hash tables. In cases where you can’t just magically add memory if you need to, you need to get more creative.

The first thing to do is to ensure that the smaller tables are the ones being loaded to hash objects. Make your largest table the base table, and other, smaller, tables the hash tables. Second, try to make your variables smaller. If you have a numeric, 8 byte 0/1 indicator of something, change it to a 1 byte character ‘0’/’1’ or ‘y’/’n’.

There are also tools available that can reduce the size of a SAS data set by calculating the minimum possible number of bytes for all variables in a data set. One such tool, Minimizer, is available from Qualex Consulting Services at www.qlx.com/minimizer. Finally, make sure that you are only using the records and variables that you need. Perhaps a subset of the data will get the job done if you don’t have a dense merge.
MORE HASH OBJECT METHODS

There are more methods available to the hash objects. Referring back to my previous examples, the dog knows more tricks or the %include program has more macros defined in it. Thus far, this paper has gone into excruciating detail about the defineKey, defineData, defineDone and Find methods. Because these are the methods that I use for merging, and I wanted to limit my topic to using hash tables to speed up merges, this section only briefly outlines some other methods. There are many sources available that go into far more detail about ways to use these other methods. However, if this paper is your first exposure to hash tables, then I thought it was important for you to know that hash tables have more than just the one capability.

ADD – This method adds key and data values to the hash object. If you were loading values manually, as opposed to from a dataset you would use this method.

DELETE – This method deletes a hash table from memory.

OUTPUT – This method enables you to output the contents of your hash table to a permanent SAS dataset.

REMOVE – This method allows you to remove the data associated with a specific key in your hash table.

REPLACE – This method allows you to replace the data associated with a specific key in your hash table.

The following example shows the use of all of these methods.

data _null_
length dt Temp Precip 8.;
format dt date9.;
if _N_ = 1 then do;
Declare hash w (dataset:'weather', ordered:'Y');
w.defineKey ("Dt");
w.defineData("dt","Temp","Precip");
w.defineDone();
w.remove (key:’01JUL2007’d);
w.replace(key:'02JUL2007'd, data:'02JAN2007'd, data:99, data:99);
w.add (key:'08JUL2007'd, data:'08JUL2007'd, data:50, data:50);
w.output (dataset:'weather_new');
w.delete ();
end;
run;

Note the addition of this ordered option.
* You can specify ‘Y’, ‘YES’, ‘a’ or ‘ascending’ for output to be in ascending order by the key.
* Specify ‘d’ or ‘descending’ for output in descending order.
* ‘N’ or ‘NO’ outputs rows in unsorted order. This is the default.
A MORE COMPLEX EXAMPLE

Thus far, the examples in this paper have been relatively simplistic. Within the ice cream example, there is now a more complex join that needs to take place. After joining tables similar to those in the first example, I used the resulting table as an input to PROC REG. I now have two regression models per zone (one each for stands and trucks) that explain sales and an output dataset with the model coefficients. Sales are explained by a combination of calendar effects, weather effects and economic effects. By multiplying my model coefficients by data for future dates, I would now like to use these models to forecast future sales. The equations look something like this:

\[
\begin{align*}
\text{Zone 1 Truck sales} & \quad \text{Jan 01, 2008} = 0.3(\text{econ Z1 Jan 01, 2008}) + 0.2(\text{calendar Jan 01, 2008}) + 0.5(\text{weather Z1 Jan 01, 2008}) \\
\text{Zone 2 Stand Sales} & \quad \text{Jan 01, 2008} = 0.4(\text{econ Z2 Jan 01, 2008}) + 0.3(\text{calendar Jan 01, 2008}) + 0.3(\text{weather Z2 Jan 01, 2008})
\end{align*}
\]

The same coefficients, by zone, are used each time the model is solved. Future calendar data, such as the day of the week for particular date, is known. Economic forecasts are easily obtained. The only input left to deal with is the weather. Unfortunately, there are no reliable weather forecasts for ten years, or even ten days, into the future. To handle this issue, we have chosen to solve the model with a range of historical weather values, thus creating a distribution of sales forecasts from which we can choose a median, high and low sales value.

For any given date in the future, the sales model's are solved with nine historical weather scenarios. For example, for January 1\(^{st}\), 2008, the model will be solved with the economic forecast and calendar data for that day. In each of the nine instances of solving for that day, a different historical weather scenario gets plugged in. Weather from the same date for the previous three years is used, as well as weather from the following two days. Thus, January 1\(^{st}\), 2005's weather will be used the first time the model is solved, January 2\(^{nd}\), 2005 the second time and January 3\(^{rd}\), 2005 the third. Repeat the process for 2006 and 2007 thus solving each model 9 times for each forecast date.

1. Zone 1 Truck sales Jan 01, 2008 = 0.3(\text{econ Z1 Jan 01, 2008}) + 0.2(\text{calendar Jan 01, 2008}) + 0.5(\text{weather Z1 Jan 01, 2005})
2. Zone 1 Truck sales Jan 01, 2008 = 0.3(\text{econ Z1 Jan 01, 2008}) + 0.2(\text{calendar Jan 01, 2005}) + 0.5(\text{weather Z1 Jan 02, 2005})
3. Zone 1 Truck sales Jan 01, 2008 = 0.3(\text{econ Z1 Jan 01, 2008}) + 0.2(\text{calendar Jan 01, 2005}) + 0.5(\text{weather Z1 Jan 03, 2005})

\[
\ldots
\]

9. Zone 1 Truck sales Jan 01, 2008 = 0.3(\text{econ Z1 Jan 01, 2008}) + 0.2(\text{calendar Jan 01, 2008}) + 0.5(\text{weather Z1 Jan 03, 2007})

You can probably see that joining these tables with DATA step merges would be time consuming. First of all, one of the tables is keyed on the same variables, so it would require multiple sorts and merges. Second, while the economic and weather data is zone specific, the calendar data isn’t. To get the correct calendar data associated with each of the zone’s economic and weather data, the join needs to be one-to-many, which a DATA step merge won’t do (at least not in its simple form). This scenario seems ideally suited to a PROC SQL join, but hash tables can get it done much faster. To account for the DATA step’s inability to do a one-to-many join, hash tables are coupled with some looping for each of the individual zones as well as for the two models that are being solved.

<table>
<thead>
<tr>
<th>Forecast date</th>
<th>weather_scenario</th>
<th>dayofweek</th>
<th>month</th>
<th>holiday_ind</th>
<th>Zone DEPVAR</th>
<th>Intercept</th>
<th>Day_of_week</th>
<th>Month</th>
<th>Holiday_indicator</th>
<th>Precip</th>
<th>Temp</th>
<th>GMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zone 1 Truck Sales</td>
<td>21,963.98</td>
<td>(32.62)</td>
<td>0.00</td>
<td>134.31</td>
<td>(780.90)</td>
<td>(6.31)</td>
<td>470.23</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zone 1 Stand Sales</td>
<td>20,696.69</td>
<td>(39.48)</td>
<td>0.00</td>
<td>206.11</td>
<td>(739.84)</td>
<td>(4.88)</td>
<td>433.61</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zone 2 Truck Sales</td>
<td>26,746.62</td>
<td>(43.34)</td>
<td>0.00</td>
<td>147.74</td>
<td>(780.90)</td>
<td>(6.31)</td>
<td>517.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zone 2 Stand Sales</td>
<td>25,206.62</td>
<td>(43.43)</td>
<td>0.00</td>
<td>226.73</td>
<td>(739.84)</td>
<td>(4.88)</td>
<td>487.97</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zone 3 Truck Sales</td>
<td>15,206.99</td>
<td>(34.50)</td>
<td>0.00</td>
<td>127.73</td>
<td>(807.39)</td>
<td>(2.43)</td>
<td>345.41</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zone 3 Stand Sales</td>
<td>12,398.27</td>
<td>(41.24)</td>
<td>0.00</td>
<td>190.92</td>
<td>(752.39)</td>
<td>0.28</td>
<td>282.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In the code to the left, zonelist and filesuffix are macro variables. Typically, in my process, these are values supplied by the users through a GUI, but for this example, we will just hard code them. The macro variables col1 and col2 are for columns I want on my final dataset. Because these values are unlikely to ever change, I am comfortable leaving them in my code like this.

The macro, makefilerefs, reads the zonelist and creates global macro variables from each element in the zonelist that will be dataset names for use by the next step. It also counts the number of zones in the zonelist. For example, this macro will produce the following macro variables:

&d1 = zone1_testhash
&d2 = zone2_testhash
&d3 = zone3_testhash
&zonecnt = 3

The first part of this step is pretty standard. The loop in the DATA statement is dynamically creating the number of datasets that are specified by the zonecnt macro variable – a separate output dataset for each zone in the list. For this example it is the equivalent of "data zone1_testhash zone2_testhash zone3_testhash;". While this may seem like overkill for this example, my users are often running the process for different selections of zones and I needed this step to be very flexible. Also in this section, three hash tables are being declared and the base table, work.calendar, is being set. The DATA step is embedded in a macro so that the %do loops will work.
The second part of this DATA step utilizes the macro variables that were supplied in the code and were created from the %makefilerefs macro. The outer loop, \( z \), loops \&zonecnt times, in this case, three times – once for each zone specified in the \&zonelist macro variable. Because the calendar table, the table on the SET statement, does not have zone on it, and we need to merge to hash tables that use zone in their keys, we need to get zone onto the calendar table. In each iteration of the loop, \( z \) is provided as an argument to the SCAN function which is reading elements of the macro variable \&zonelist. The SCAN function returns the \( z \)th zone from \&zonelist, assigns the value to the variable zone, and now the calendar table has the zone variable it needs to match to the hash tables.

The inner loop, \( m \), iterates twice, once for each model (truck_sales and stand_sales). In this loop, the Find method uses a key of zone and \&col&m to look up observations in the coefficient dataset. Depending on the iteration of the inner loop, the value of \&m will resolve to be 1 or 2. Once \&m resolves, then the value of \&col1 and \&col2 will resolve to be stand_sales and truck_sales, respectively. The code takes the data from the calendar dataset, which has already been merged with the economic and weather data in the outer loop, and multiplies these data values by the corresponding coefficients from the coefficient dataset to generate forecasts for future dates. A column for each forecast, truck_sales and stand_sales, is output to the final data set for the zone. My results are shown below. It would seem that while my regression models still need a lot of work, my DATA step with loops and hash tables performed perfectly!
CONCLUSIONS

I hope that this paper has convinced you that while hash tables may seem difficult at first, they can be an easy to use and valuable tool once you have had some practice. In my own experience I took a process, similar in structure to the more complex example in this paper, that was taking between 2 and 4 hours (depending on network traffic) to run using a PROC SQL join, and using hash tables cut the execution time to a consistent 11 minutes. You will note that aside from this anecdotal evidence, I do not provide any benchmarking results in this paper. That seemed, to me, to be beyond the scope of a practical guide for the use of hash tables. However, there are many sources that do address this issue. I simply encourage everyone to just try hash tables and see how your own execution times improve.

REFERENCES

Getting Started with the DATA Step Hash Object

Hashing Performance Time with Hash Tables
Elena Muriel, Amadeus Software Ltd., UK

SAS Programming 3: Advanced Techniques and Efficiencies Course Notes
SAS Institute, Inc., Cary, North Carolina

SAS Help and Documentation
Search -> keyword “Hash Object”

SAS Technical Support
http://support.sas.com/techsup/unotes/SN/016/016920.html

ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration. Other brand and product names are trademarks of their respective companies.

PJM® is a registered servicemark of PJM Interconnection, Norristown PA. Used with permission, all other rights reserved.

I would like to thank Fabian Robinson who first suggested I try using hash tables in my process; Jane Stroupe, Rick Bell and Jason Secosky from SAS who helped me get started on how to actually use them; John Slivka, Molly Mooney and John Reynolds - my team members who have provided valuable feedback and have been very patient with me while I have been writing this paper; Bridgid Cummings, Andy Ford and Robert Jones – co-workers who helped with the review of this paper; Kellie Bach and Troy Wolfe - friends and former supervisors who have been very influential in my career using SAS; and last but not least, John Freeman - my dear husband who knows absolutely nothing about SAS yet has graciously read this paper more times than he probably cares to think about.

CONTACT INFORMATION

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

Jennifer K. Warner-Freeman
PJM Interconnection
955 Jefferson Avenue
Norristown, PA 19403-2497
610-666-4713
freemj@pjm.com
www.pjm.com

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *