Find() the power of Hash - How, Why and When to use the SAS® Hash Object

John Blackwell

ABSTRACT

The SAS® hash object has come of age in SAS 9.2, giving the SAS programmer the ability to quickly do things never before possible within a single data step. This paper will demonstrate how easy it is to get started using the powerful features of the hash object and show examples of how these features can improve programmer productivity and system performance.

INTRODUCTION

The SAS hash object is commonly referred to as an in-memory look-up table. While this is accurate enough, thinking of it as merely a look-up table does not give programmers an intuitive sense of everything a hash object can be used for. This is particularly true in SAS 9.2, where two new features in particular have greatly expanded its usefulness. This paper will provide a brief introduction to the SAS hash object, showing a typical "look-up" usage and then examining situations where hash objects can be the difference between a job failing and completing.

USING THE HASH OBJECT AS LOOK-UP TABLE

In this example I am using a data set from a direct-marketing organization that contains the account_id of the customer and a source code representing various catalog mailings.

Sample Customer_Mail data:

Account_ID	Source
8800544	SP45635
8803002	WT4563
9330220	SP45635
5996600	WT4563
5588221	WT4563
5996600	SP45635

The second data set will be used as the look-up table for the source code description and loaded into a hash object.

Sample Source_Code data:

Source	Description
SP45635	Spring Catalog 2009
WT4563	Winter Catalog 2009

In order to append the descriptions from the Source_Code data set to the Customer_Mail data using a hash object I executed the following code. In cases where descriptions were missing from the source_code table the description is populated with "Missing."

```
data xmpl.mailing_w_descriptions;
length description $35;
```

```
        set xmpl.customer_mail;
        if _n_ =1 then do;
                declare hash sc (data set: 'xmpl.source_code');
                sc.definekey('source');
                sc.definedata('description');
                sc.definedone();
        end;
        if sc.find() ge 0;
        if description ='' then description='Missing';
run;
```

The resulting data set would be identical to what would have been achieved by sorting each data set by source and then using the data step to merge the sorted tables (or by using an outer join in proc sql on the source field). Using the hash object utilizes significantly less cpu time because the data from xmpl.source_code is loaded into memory and minimizes the number of costly disk reads. The above example is a good illustration of the hash object at its most simple; it is a good starting point for the programmer currently unfamiliar with how to use them.

The benefits in performance grow exponentially when you are combining multiple large data sets and, in some cases, can be the difference between a job failing and running successfully. For example, programmers often need more than one look-up table, and adding multiple hash objects is fairly simple. In the following example we are adding a region code based on the account_id of the customer as well as appending the source description.

```
data xmpl.mailing_w_descriptions_region;
length description region_code $35;
        set xmpl.customer_mail;
        if _n_ =1 then do;
                declare hash sc (data set: 'xmpl.source_code');
                sc.definekey('source');
                sc.definedata('description');
                sc.definedone();
/*second hash object*/
                declare hash rg (data set: 'xmpl.customer_geographic');
                rg.definekey('account_id');
                rg.definedata('region_code');
                rg.definedone();

        end;
        if sc.find() ge 0;
        if rg.find() ge 0;
        if description ='' then description='Missing';
run;
```

COMMON ERRORS

It is important to note that the DEFINEDONE and DEFINEKEY methods do not affect the PDV. So if you were to execute the above code without the length statement, the log would show the following error message:

```
ERROR: Undeclared data symbol description for hash object…
ERROR: DATA STEP Component Object failure.  Aborted during the
EXECUTION phase.
```

Since we are only adding two variables (description and region_code) through the hash object in the preceding code, including a length statement is a relatively efficient way to add them to the PDV. However, altering the PDV to accommodate *several* variables of different lengths is not. The most efficient way to alter the PDV is to issue a SET statement for the data set(s) being loaded into the hash object(s) with conditional logic that will never be true. For example, the following line will cause the PDV to add space for the variables in 'xmpl.source_code' and 'xmpl.customer_geographic':

```
if 1=2 then do;
        set 'xmpl.customer_geographic';
        set 'xmpl.source_code';
end;
```

Obviously "1" will never equal "2," so the SET statements will never actually execute. But they will still cause the PDV to be altered. This method renders the length statement unnecessary and, since the variables will be read automatically through the SET statements, the programmer will not even need to know their lengths.

THINKING BEYOND LOOK-UP TABLES

As of SAS 9.2, the Hash Object has an "ordered" method which can be used to sort a data set. While sorting is not necessary for "look-ups", many SAS procedures that use by-group processing either require, or run faster on, sorted data sets. For example, consider a series of reports that is created from a multi-million-row data set and generated from several proc-tabulate steps. Each proc-tabulate step uses different "by groups" and therefore requires multiple sorts. The sorts on this data set are extremely costly in terms of CPU time and take several minutes each using proc sort. You can use proc tabulate with the "unsorted" option, but the performance time on large data sets often makes this impractical. The following code shows how sorted subsets of data (xmpl.market_sort) can be created using proc-sort-and-through hashing:

Sorting with proc sort

```
proc sort data=xmpl.marketing_rev out=xmpl.market_sort (keep=account_id contact_date);
        by account_id contact_date;
run;
```

Sorting with hash

```
data _null_;
        length account_id contact_date 8;
        if _n_=1 then do;
        declare hash hh (data set: 'xmpl.marketing_rev', ordered:'a');
        hh.definekey('account_id','contact_date');
        hh.definedone();
        end;
        hh.output(data set: 'dash.hpg24');
run;
```

Prior to SAS 9.2, the above hash object code would not have worked if duplicate values for account_id and contact date had needed to be preserved in the resulting data set. The hash object would simply take the first value for each key and ignore duplicates. As of 9.2, you can specify that it take the last value or preserve all duplicate-key values with the "multidata" option. The previous example could be re-written as follows to preserve duplicate values:

Sorting with hash (preserving duplicates with the "multidata" option)

```
data _null_;
        length account_id contact_date 8;
        if _n_=1 then do;
```

```
        declare hash hh (data set: 'dash.apps_ints_raw', multidata:'y', ordered:'a');
        hh.definekey('account_id','contact_date');
        hh.definedone();
        end;
        hh.output(data set: 'dash.hpg24');
run;
```

When using the hash object to sort, it is useful to remember that you can specify as many fields as you want to sort by as the "key." The key is not necessarily a combination of values that would normally be considered a key, but rather any combination of values (character, numeric, or a combination), by which you would like to sort your data. In the above example the cpu time for the hash object sort is 35 seconds versus 7 minutes for the proc sort.

USING HASH TO ACCESS DATA OUTSIDE OF SAS

One of the most compelling reasons to use hash objects involves working with large data sets stored outside of SAS. For example, most marketing organizations will use SAS to access data from a data warehouse in an RDBMS. Typically SAS programmers will extract data into temporary data sets and then sort and merge them in subsequent steps. But because the data in the RDBMS cannot be sorted or indexed, it is often not possible to use the data step with a merge statement when working with non-SAS data. While proc sql can be used with varying degrees of efficiency, depending on the size of the data, it is often impractical. In the scenario where a programmer wants to combine large SAS data sets with large Oracle® tables, the proc sql step will often fail. In the following example, code is shown that combines data from SAS (xmpl.account_names) and an Oracle table (orcl.account_purchases). The SAS data set contains the name information for all accounts, and the oracle table is a list of total purchase amount for all accounts.

```
data highest_amt;
        length total_purchase_amount 8;
        set xmpl.account_names;
        if _n_=1 then do;
        declare hash a(data set:'orcl.account_purchases');
        a.definekey('account_id');
        a.definedata('total_purchase_amount');
        a.definedone();
        end;
if a.find() ge 0;
run;
```

In its simplicity, this is similar to the first example provided in the paper. However, in this case, the proc-sql version of the code failed to complete, and the multi-step method of doing an initial extract from the Oracle table, sorting, and then merging using the data step took almost 30 minutes. The preceeding code took only three minutes to run. to run versus about 3 minutes using the above code. In addition, because this method does not call on any of the more complex methods associated with hash, the coding time is significantly quicker even for programmers new to the SAS hash object . While this is technically a case of using the hash object as a "look up", it is not what people typically associate with look-up tables since the data being joined through the hash object has the same cardinality as the data set specified in the set statement.

CONCLUSIONS

The SAS hash object should be thought of as more than just an in-memory table look-up. While traditional look-ups (appending information from a small table to a bigger table) can be performed efficiently with the SAS hash object, in many situations the improvement in performance is only a few

seconds. For many users, gains of a few minutes in performance may not be critical and are often not sufficient motivation to learn how to use the hash object. However, in situations where you are dealing with very large data sets and multi-field keys, hashing may be the only viable solution. SAS programmers should see the hash object as an alternative to any proc-sql-join or data-step merge. The only limit to the amount of data that can be loaded into a hash object is the amount of memory available to the SAS session.

REFERENCES:

Bhat, Gajanan, and Raj Suligavi. 2001. "Merging Tables in DATA Step vs. PROC SQL: Convenience and Efficiency Issues." Proceedings of the Twenty-Sixth SAS Users Group International Meeting. Cary, NC: SAS Institute Inc.

Loren, Judy. 2006. "How Do I Love Hash Tables? Let Me Count The Ways!" Proceedings of the Nineteenth Northeast SAS Users Group Meeting. Cary, NC: SAS Institute Inc.

Secosky, Jason, and Janice Bloom. 2007. "Getting Started with the DATA Step Hash Object." Proceedings of the First SAS Global Forum. Cary, NC: SAS Institute Inc.

ACKNOWLEDGMENTS

CONTACT INFORMATION

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

John Blackwell
The Nature Conservancy
4245 N Fairfax Drive ste 100
Arlington, VA 22203
jblackwell@tnc.org