Paper 3125-2019

HASH Beyond Lookups – Take Another Look

Elizabeth Axelrod, Abt Associates Inc.

ABSTRACT

Using the hash object in SAS® – beyond table lookup and data retrieval – enables you to do some pretty cool stuff, solving complicated problems with code that runs efficiently and flows intuitively. And… it's easy to maintain and modify. What's not to like? This is an updated and expanded version of my original paper "Hash Beyond Lookups - Your Code Will Never Be the Same". It provides a further exploration of some useful Hash techniques, and offers more situations that can be elegantly solved using Hash. The main message? Don't be afraid of Hash!

INTRODUCTION

Most SAS users who learn about the hash object first learn how to do lookups with a hash object (table). This paper assumes that you already have some familiarity with this technique. But did you know that in addition to looking up data in a hash table, you can also:

- *build* tables in memory during program execution,
- *traverse* tables,
- *aggregate* data,
- *sort* data, and
- *save* the contents of a table to a file.

Hash tables are well suited to solve a variety of problems that can also be solved by other more traditional methods. But once you get a taste of the power, efficiency, and elegance of these techniques, you'll be hooked. Perhaps because it has its own nomenclature – object.dot syntax – so different from the standard DATA step, many programmers do not venture beyond using hash tables for lookup or retrieval of data. But if you're already using hash tables to do lookups and data retrieval, you're almost there - the trick is just to look at your problems in a new way, and recognize the opportunity to use hash as a solution. This paper is intended to introduce you to some of these other fabulous and surprisingly simple-to-code things that hash objects can do. At the end of this paper you will find references and suggested readings which provide extensive details and more technical explanations.

GETTING STARTED

Using hash methods to build tables in memory enables you to solve complex problems – often in a single DATA step, which otherwise might require multiple DATA and PROC steps. This paper uses fairly simple examples, so we can focus on the basic techniques.

In this first example, we'll create several summary tables from very a large non-sorted file – all in a single DATA step, with no pre-sorting. Let's say you have a large file of inpatient hospital claims (CLAIMS), and a list of hospital IDs (HOSPITALS). In this example it doesn't matter how the input files are sorted, but I present the input data in a sorted fashion, so you can more easily see what the output files should look like.

Here are the input files:

CLAIMS

PATID	ADMIT	DISCH	HOSPID	PAYMENT
P1	03/05/15	03/08/15	H111	1111.11
P1	01/03/16	01/09/16	H333	222.22
P1	10/12/16	10/15/16	H333	333.33
P2	05/05/15	05/06/15	H222	444.44
P2	09/16/15	10/01/15	H222	11.11
P2	12/01/15	12/15/15	H555	5555.55
P2	04/04/16	05/06/16	H111	7777.77
P2	06/06/16	06/08/16	H444	111.11

HOSPITALS

HOSPID	HOSPTYPE
H111	ACUTE
H222	ACUTE
H333	REHAB
H444	PSYCH

Table 1. Input Data Sets

Your mission is to create two output files:

1) *Summarized measures at the hospital level.* This file will contain one record for every hospital encountered in the CLAIMS file, with these variables:

• HOSPID	Hospital ID
• HOSPTYPE	Type of hospital, found in the HOSPITALS data set
• N_CLMS	Number of records found in the CLAIMS data set for each hospital
• TOT_PAY	The sum of payments found in CLAIMS for each hospital

2) *Summarized measures at the patient/hospital level*. This file will have one record for each unique patient/hospital combination, with these variables:

• PATID	Patient ID
• HOSPID	Hospital ID
• MIN_ADMIT	Earliest admission date for this Patient/Hospital
• MAX_DISCH	Latest discharge date for this Patient/Hospital

Here are the desired output files:

HOSPITAL SUMMARY (HSUM)

HOSPID	HOSPTYPE	TOT_PAY	N_CLMS
H111	ACUTE	8888.88	2
H222	ACUTE	455.55	2
H333	REHAB	555.55	2
H444	PSYCH	111.11	1
H555	?	5555.55	1

PATIENT/HOSPITAL SUMMARY (PSUM)

PATID	HOSPID	MIN_ADMIT	MAX_DISCH
P1	H111	03/05/15	03/08/15
P1	H333	01/03/16	10/15/16
P2	H111	04/04/16	05/06/16
P2	H222	05/05/15	10/01/15
P2	H444	06/06/16	06/08/16
P2	H555	12/01/15	12/15/15

Table 2. Desired Output Data Sets

SOLUTION

You could certainly solve this without using hash objects, but you'd have to do some sorting or use some PROC steps, and if the specifications were more complicated than this, you might even need to pass through your data multiple times. By using hash tables, we can do the whole thing in a single DATA step…and we don't have to do any sorting!

We will be using three hash tables – one will be a lookup table of hospitals so we can retrieve HOSPTYPE, and the other two are the summary tables.

Here is the logic of what we'll be doing:

1. Define all the variables in our hash tables for the PDV (more about this later).
2. Declare (create) a hash table (hash object) for the hospital lookup table.
3. Declare an ordered hash table to hold the hospital summary table.
4. Declare an ordered hash table to hold the patient/hospital summary table.
5. Read each record from the CLAIMS data set.
6. Look for HOSPID in the hospital hash table, in order to retrieve its HOSPTYPE.
7. Accumulate info from the current claim into the hospital summary table (HSUM).
8. Accumulate info from the current claim into the patient/hospital summary table (PSUM).
9. If we're done reading all the claims, write out the two summary tables.

Done!

One of the things I love about hash is that your code flows through your DATA step in a logical progression that reflects the simplest path in solving a problem.

The code is below, followed by a brief explanation of each numbered section.

CODE

```
data HOSP_SUM (keep=HOSPID HOSPTYPE N_CLMS TOT_PAY)
     PAT_SUM  (keep=PATID  HOSPID MIN_ADMIT MAX_DISCH);
    ************************************************;
    * Define all vars used in hash tables for the PDV,
    * using LENGTH and SET statements.
    ************************************************;
    attrib MIN_ADMIT    length=4 format=MMDDYY8.                 ❶
           MAX_DISCH    length=4 format=MMDDYY8.
           N_CLMS       length=4
           TOT_PAY      length=8;
    if 0 then
        set CLAIMS
            HOSPITALS;
    ************************************************;
    * Declare all hash tables.
    ************************************************;
    if _n_=1 then do;
        ********************************************;
        * HID: HOSPITAL ID TABLE, FOR LOOKUP AND DATA RETRIEVAL.
        ********************************************;          ❷
        declare hash HID(DATASET:'HOSPITALS');
        HID.definekey('HOSPID');
        HID.definedata('HOSPTYPE');
        HID.definedone();

        ********************************************;
        * HSUM: HOSPITAL SUMMARY TABLE.
        ********************************************;
        declare hash  HSUM(ORDERED:'A');
        declare hiter HIX('HSUM');                              ❸
        HSUM.definekey('HOSPID');
        HSUM.definedata('HOSPID','HOSPTYPE','N_CLMS','TOT_PAY');
        HSUM.definedone();

        ********************************************;
        * PSUM: PATIENT/HOSPITAL SUMMARY TABLE.
        ********************************************;
        declare hash  PSUM(ordered:'A');
        declare hiter PIX('PSUM');                              ❹
        PSUM.definekey('PATID','HOSPID');
        PSUM.definedata('PATID','HOSPID','MIN_ADMIT','MAX_DISCH');
        PSUM.definedone();
    end;

    set CLAIMS end=EOF;                                         ❺

    ********************************************;
    * Is the hopsital ID from the claim in the
    * list of hospital IDs in the HID hash table?
    ********************************************;               ❻
    RC=HID.find();
    found=(RC=0);
    if not(FOUND) then
        HOSPTYPE='?';
    *******************************************;
    * Now see if this HOSPID is already in the HSUM
    * hash table. If not, initialize the vars and add
    * a new entry in the HSUM hash table.
    *******************************************;
    RC=HSUM.find();
    found=(RC=0);
    if not(FOUND) then do;
        * This HOSPID is not yet in the hsum hash table... Initialize
        * the summary vars, then add a new entry in the HSUM hash table;
        N_CLMS  = 0;
        TOT_PAY = 0;
        HSUM.add();
    end;
    * In all cases, sum the vars, then replace the data in the hash table;
    N_CLMS  = sum(N_CLMS,1);
    TOT_PAY = sum(TOT_PAY,PAYMENT);
    HSUM.replace();
```

❼

```
    *******************************************;
    * Is this patient/hospital combo already in
    * the PSUM hash table? If not, initialize the
    * vars and add a new entry in the PSUM hash table.
    *******************************************;
    RC=PSUM.find();
    FOUND=(RC=0);
    if not(FOUND) then do;
      * This patid/hospid combo is not yet in the PSUM hash table.
      * Initialize the summary vars and add a new entry for this
      * PATID/HOSPID to the PSUM hash table;
        MIN_ADMIT=.;
        MAX_DISCH=.;
        PSUM.add();
    end;
    * Hold onto the earliest ADMIT and the latest DISCH dates for this
    * PATID/HOSPID combo, then replace these values in the table;
    MIN_ADMIT = min(ADMIT,MIN_ADMIT);
    MAX_DISCH = max(DISCH,MAX_DISCH);
    PSUM.replace();
```

❽

```
    *******************************************;
    * Are we done reading CLAIMS? If yes, write out
    * the contents of the HSUM and PSUM hash tables.
    *******************************************;
    if eof then do;
        RC=HIX.first();
        do while(RC=0);
            output HOSP_SUM;
            RC=HIX.next();
        end;
        RC=PIX.first();
        do while(RC=0);
            output PAT_SUM;
            RC=PIX.next();
        end;
    end;
run;
```

❾

CODE WALK-THROUGH

1. Specify the two output data sets. Use a non-executed SET statement and a LENGTH statement to define all variables that are in any of the hash tables, so they are included in the PDV. (See the *Invisible Curtain* section, below, for why this is necessary.)

   The next three sections declare all 3 hash tables we'll be using.

2. The first one (HID) will hold the list of Hospitals and their HOSPTYPEs. This code should look familiar, as it just defines a lookup table. We are populating the hash table HID with data from the HOSPITALS file. This happens just by the way we've declared the hash table, specifying the input data set name to the *argument tag* DATASET. During the first iteration of the DATA step (_n_=1), the data from HOSPITALS is loaded into the hash table in one fell swoop – vacuum-cleaner style – Whoosh!

3. The next hash table (HSUM) will hold our hospital summary data, and is declared as an *empty* table – more like a *template* for a table - defining keys and data, but not specifying an input source with which to populate it. Instead, we will control how the data get into this table using hash methods during program execution.

   To define an empty table, we just have to make a slight modification to our declaration statement. Instead of specifying the input source data with the argument tag DATASET, we could just say:

   ```
   declare hash  HSUM();
   ```

   But since we want our output files to be sorted, we specify a sort order:

   ```
   declare hash  HSUM(ordered:'A');
   ```

   Specifying an order ('Y', 'Yes', or 'A' for ascending, or 'D' for descending), provides us with the ability to *retrieve* items from the hash table in sorted order. We do not care how they are actually *stored* in memory – this is for SAS to worry about!

   We also specify a *hash iterator* (hiter):

   ```
   declare hiter HIX('HSUM');
   ```

   Think of this as a pointer - dedicated to this hash table - which will enable us to traverse the table forwards and backwards. We name the hash iterator (in this case, HIX) and associate it with a specific hash table (HSUM).

   Note that HOSPID is defined as both the *key* and *data* hash variables. This is because I want HOSPID to populate in the HSUM table. So even though HOSPID is already defined as the hash table's *key*, I also have to define it as a hash *data* variable.

4. This hash table (PSUM) is also an empty hash table, but this one has two keys: PATID and HOSPID, and will hold summary measures for all unique combinations of PATID/HOSPID encountered on the claims.

5. Our hash tables are all defined, so... we can finally start reading CLAIMS.

6. The FIND() method is used to see if this claim's HOSPID is in the HID hash table, and if it is, it retrieves the value for HOSPTYPE (which we'll need for our HSUM hash table). If we do not find HOSPID in the HID hash table, we assign a value of `'?'` to HOSPTYPE. It is extremely important to remember that if the result of a FIND() call is unsuccessful, then the value of the data item you are trying to retrieve, in this case HOSPTYPE, is not automatically set to missing. Instead, it is the value of your last successful call. That is why I'm explicitly assigning a value to HOSPTYPE if I did not find the HOSPID in the HID

hash table. Alternatively, I could have set HOSPTYPE to missing before I issued the FIND():

```
call missing(HOSPTYPE);
RC = HID.find();
```

Either of these approaches – setting HOSPTYE to missing before issuing the FIND(), or explicitly assigning it a value after the FIND() if the call is unsuccessful – will ensure that you're not holding onto a value of HOSPTYPE that belongs to another HOSPID.

Note that the variable RC holds the return code for the result of a call, and a value of 0 equals success. This is the opposite of the Boolean convention (where 0 equals FALSE). Therefore, to help minimize my own confusion, I use this convention:

```
RC = H.find();
FOUND = (RC=0);
if FOUND then /* do something */;
```

I'm just converting the successful return code of 0 to a 1, so I can use standard DATA step coding. With this one teeny extra line of coding (FOUND=(RC=0)), I save myself from the inevitable unintended swip-swapping of my intended actions.

7. Next, we look to see if the HOSPID from the current claim is in the HSUM hash table. If it's not in there yet, we initialize the summary variables N_RECS and TOT_PAY to zero (0). We then use the ADD method to add this new HOSPID to the hash table. Then, in all cases, we add 1 to N_RECs, add the payments from the current claim, and use the REPLACE method to replace the entry in the hash table.

   Note that we do not need a RETAIN for these accumulated variables. Why not? What's going on here? I called the FIND method prior to doing any summing. If the key-value of HOSPID (currently in the PDV) is not found in the table, I'm just initializing variables, then adding a new entry to the table. But if the key-value *is* found, then the current values of N_RECS and TOT_PAY are the *accumulated* values for that key variable so far. I then add 1 to N_RECS and add the current value of PAYMENT to the accumulated TOT_PAY, and put the updated values back into the hash table using the REPLACE method. Items in a hash table are not reinitialized at each execution of a DATA step.

8. This section is similar to the previous section – but we are now summarizing to the patient/hospital level. Is this PATID/HOSPID combination from the CLAIMS already in the hash table PSUM? If not, initialize the summarization variables to missing (.). Then use standard DATA step functions to determine the earliest admission date and latest discharge date, and use the REPLACE method to put these values back into the hash table.

9. When we've finished reading all the CLAIMS, the HSUM table will contain one entry for each unique HOSPID encountered on the CLAIMS, with a count of that hospital's claims and a sum of its payments. The PSUM hash table will contain one entry for each patient/hospital combo, with the earliest admission date and latest discharge date for each combo. Now we just have to write out each entry from these hash tables. If we don't, all our work will be lost, since the contents of a hash table disappears when the DATA step finishes processing.

   To write out these entries to a SAS data set we will traverse the table, one item at a time, starting with the 'FIRST'. Since we defined the HSUM hash table as ORDERED: 'A', the 'first' item is the one with the smallest value of HOSPID. We then move on to the NEXT (logical) entry, and continue along until the entire table is written out to a file. Again, we don't know or care how SAS is *storing* our data in memory – we just care about how it's *retrieved* for us.

The next block of code writes out items from the PSUM hash table using the same technique as described above. The resulting data set holds the earliest admission date and latest discharge date for each unique combination of PATID / HOSPID.

Here is another way to write the contents of a hash table:

```
if EOF then
     RC=HSUM.output(dataset:'HOSPITAL_SUMMARY');
```

This construct uses a leaf-blower technique – Whoosh! The entire hash table is output to a file in our desired sort order. No need to traverse the table one item at a time using the hash iterator; the OUTPUT method does it automatically. Keep in mind that if you use this method to output your data to a file, you cannot specify the name of the output data set on your DATA statement. In the code above, I chose to show you the one-item-at-a-time technique for two reasons – a) to familiarize you with how the hash iterator works, and b) because I think it's more intuitive to see the output data sets specified on the DATA statement. In this example, either technique is fine to use because we are writing out the contents of our hash tables just once – after we've read all our records. But if you call the OUTPUT method more than once (naming the same output data set) in your DATA step, each subsequent OUTPUT method call will result in overwriting that file. For further details on this, I refer you to the book by Paul Dorfman and Don Henderson, listed at the end of this paper.

WHY THIS IS SO WONDERFUL – AKA HASH LOVE

The beauty of creating summary tables using hash objects is that our incoming data does not need to be pre-sorted – yet we are building a table which we can retrieve in sorted order. Additionally, since the entire table is in memory, we have access to all of the data during program execution. We can traverse the table, examining entries as we go, and perform other DATA step operations if desired. And as mentioned above, the code follows a logical and intuitive flow that is easy to modify. Imagine if you were asked to create another summary table. No problem! Just throw in another hash table to gather the summarized data.

THE INVISIBLE CURTAIN

Did you know that the hash variables that you define in the key and data portions of your hash table do not automatically appear in the Program Data Vector (PDV)? Wow! To me, this fact is both perplexing and magical. I imagine an invisible curtain between these two environments. So how does data in your hash table communicate with the PDV? How do you pass information between them? You must create PDV variables at compile time with the same exact names as all the hash variables defined in your hash tables. You can use SET or LENGTH statements to do this. Using any of these techniques achieves the goal of providing a 'conduit' for communication between your hash table variables and their PDV counterparts. It's helpful to remember this as you develop and test your code, and I encourage (URGE) you to refer to Paul Dorfman's 2014 paper for further explanation of this.

MORE COOL STUFF

Here's another kind of problem hash tables can help you solve: Supposing you had to select all of the claims for patients, where some condition was met on one or more claims for that patient. You might not know if the condition has been met until you read through all of the claims for a patient. Programmers frequently solve this problem by passing through their data twice – once to identify which patients have one or more claims that meet the condition(s), and then a second time to pull *all* claims for those patients. With a hash approach, you can do this all in one DATA step! Let's look at an example.

Our task is to create a sorted file of all inpatient claims for patients who meet these criteria:

- they have at least one claim with *(condition X)* … and
- they have at least one claim with *(condition Y)*

where the *conditions* are stand-ins for whatever selection criteria you want. For example, condition X could be (DIAG_CD='1234') and condition Y could be (HOSPID='5678'), and the conditions need not be met on the same claim. The output file is to be sorted by PATID and ADMIT date.

In this example, I'm assuming that my hospital claims are at least partially sorted - that is, they are sorted by patient (PATID). We'll solve this by inserting all claims for a patient into a hash table. Once we've assembled all the claims for that patient, we'll decide if they've met our selection criteria – in this case *conditions X and Y*. If they have, we'll write out the contents of the hash table to a permanent data set. Then we'll move on to the claims for the next patient.

SOLUTION

Here is the logic of what we'll be doing:

1. Define all the variables in our hash table for the PDV.
2. Declare (create) a hash table for the hospital claims.
3. Read each record from the CLAIMS data set, and determine if the current claim meets the selection criteria.
4. Whether or not a condition has been met, insert the claim into the hash table.
5. If we're on the last claim for this patient, decide if this patient has met all the selection criteria. If YES, write out all of the table entries to an external file (in sorted order).
6. Empty out the contents of the hash table, to prepare for the next patient's claims. Continue until we're done reading all the claims.

Done!

The code starts on the next page, followed by a code walk-through.

CODE

```
data OUT.FOO;
    ************************************************;
    * Define all vars used in HASH table for the PDV.
    ************************************************;
     if 0 then                                                          ❶
        set CLAIMS;

    ************************************************;
    * Declare the hash table.
    ************************************************;
    if _N_=1 then do;
        ********************************************;
        * CLM: Hash table that holds data from CLAIMS.
        ********************************************;                   ❷
        declare hash CLM(ordered: 'A', multidata:'Y');
        declare hiter CIX('CLM');
        CLM.definekey('PATID','ADMIT');
        CLM.definedata('PATID','ADMIT','HOSPID','DISCH','PAYMENT');
        CLM.definedone();
    end;

    ********************************************;
    * Read the claims. Set indicators if my conditions are met.
    ********************************************;
    set CLAIMS;
    by PATID;
                                                                        ❸
    length EVER_X EVER_Y 3;
    retain EVER_X EVER_Y 0;
    if /* (condition x is TRUE) */ then EVER_X=1;
    if /* (condition y is TRUE) */ then EVER_Y=1;
    ********************************************;
    * Insert the claim into the hash table.
    ********************************************;
    CLM.add();                                                          ❹

    ********************************************;
    * If we are on the last record for this PATID, check if this
    * patient meets the selection criteria. If YES, write out all
    * claims for this PATID. Then clear out the hash table, to
    * prepare it for the next patient's claims.
    ********************************************;
    if last.PATID then do;
       if EVER_X & EVER_Y then do;
            RC=CIX.first();
            do while(RC=0);                                             ❺
                  output OUT.FOO;
                  RC=CIX.next();
            end;
       end; /* EVER_X & EVER_Y */
       RC=CLM.clear();
       EVER_X=0; EVER_Y=0;                                              ❻
    end; /* last.PATID */
    drop EVER_X EVER_Y;
run;
```

CODE WALK-THROUGH

1. Specify the output data set. Remember: This file will hold all claims for patients who meet the selection criteria. We then use a non-executed SET statement to define all variables that are in the hash table, so they are included in the PDV.

2. Next we define an empty hash table that will hold data from the CLAIMS file. The ORDERED=‘A’ argument tag indicates that I‘ll want my output data file to be sorted by my keys, which in this case are PATID and ADMIT. We often think of a key as having unique values, but in this case I know that my CLAIMS data set contains multiple records with the same value for PATID (and possibly even multiple values for the same PATID ADMIT combo.). Using the argument tag MULTIDATA =‘Y’ will allow me to insert records with the same key values into my hash table without causing any problems. I also declare a hash iterator, because I’ll need to traverse this table. And just a reminder: Although I’ve already defined PATID and ADMIT as my keys, I must also define them as DATA items since I want these items to appear in my output record.

3. Now we can start reading the CLAIMS data set. Since this file is sorted by PATID, and since I will be processing these claims in groups by patient, I use by-group processing, with the BY PATID statement. I use the RETAIN statement to initialize a few indicators (EVER_X and EVER_Y) to zero. Then I set these indicators to 1 if my various conditions are met. For the purpose of this example, I’m using pseudo-code (if */* condition X is TRUE) */* to make the point, but of course your real code would be something like: if DIAG_CD='1234' then...

4. I use the ADD method to insert the claim into the hash table. Note that whether or not my conditions are met, I insert the claim. This is for two reasons: a) If the selection criteria are met, I’ll need to have access to all claims for this patient, so they all have to be in the table, and b) I might not know if this PATID will be selected until I’ve read all claims for this patient.

5. Are we on the last record for this PATID? If yes, we can determine if the patient has met our selection criteria. If they have, we output the all the claims for this PATID, one record-at-a-time, using the hash iterator, just as we did in the first example. Note that in this example, I cannot use the OUTPUT method described above. (Believe me, I tried it!) The resulting file would only contain data for the last patient who met all my selection criteria. Since I want to add records to my output data set as I process each patient, I need to use the hash iterator to cycle through the hash table’s entries, and write them out one-record-at-a-time.

   If the selection criteria were *not* met, we simply don’t output the records. Note that outputting entries from the hash table does not clear out its contents. We’ll do this in our next step.

6. In all cases (whether or not the selection criteria were met for this PATID, and whether or not we output the entries from the hash table to an external data set), we must clear out the contents of the hash table. We are not deleting the table, we are just clearing out its contents, making it ready to receive the next patient’s claims. We must also re-initialize our indicators (EVER_X and EVER_Y), making them ready for the next patient as well. We can drop these variables, because by definition, they will always have values of 1 in the output file.

   We’re at the bottom of the DATA step. If there are more records to be read, we loop back up to the top and read the next record. If there are no more records to be read, we’re all done. Note that unlike our first example, we’re not doing any end-of-file processing. Instead, we are (optionally) outputting the entries from the hash table, clearing out the hash table, and reinitializing our indicators for each patient, when we're at LAST.PATID.

MORE HASH LOVE

Using this approach, we were able to solve this task in a single DATA step. How cool is that? Could we have done it using other approaches? Certainly. But by using hash tables, we are able to take advantage of all the goodies available to us in standard DATA step coding. And since we're only placing one patient's claims into our hash table at a time, it's extremely unlikely that we would approach memory limit issues.

There are many situations where using hash tables to store data in memory can be a useful technique, and I've only mentioned a few. Hash tables are of course limited by available memory, and this can be an obstacle. But even so, there are ways to deal with this. I refer you to the paper and book by Paul Dorfman and Don Henderson, listed at the end of this paper, for methods to deal with the limits of memory.

The examples I've presented here are based on a few of the real-world problems I've been asked to solve. I chose these to share with you because they are common problems that we, as programmers, are faced with, and it is most likely that using hash techniques to solve them will not result in memory problems. So they're a nice way to jump in and start using hash beyond lookups!

CONCLUSION

We did it! Starting with some un-sorted data (first example) or partially sorted data (second example), we built tables in memory, created several aggregated files, traversed tables, and extracted data based on various conditions - all without creating intermediate results, without sorting, and by passing through the incoming data just once.

And because the hash processing is embedded within the DATA step, we are able to take full advantage of the power and capabilities of the DATA step. Code flows in a logical progression that reflects the simplest path in solving a problem, making your code readable, maintainable, and flexible.

This is by no means an exhaustive overview of what you can do with the hash object – there are many excellent papers that will take you further. But I hope you can see that the code to build hash tables during program execution is really quite straightforward and not difficult to write. The first time I tried this, I was faced with a very complex set of specifications and with enormous data sets as input. I was apprehensive to try it, but I was pretty sure it was the best approach. I was amazed at how easy it was to write, and how efficient it was to run. No sorting! No multiple passes through the data! The result was elegant and extremely satisfying. So I encourage you to take another look at your programming assignments, and see if a solution using hash tables could work for you too.

REFERENCES & RECOMMENDED READING

Dorfman, Paul, and Don Henderson. 2018. *Data Management Solutions Using SAS Hash Table Operations: A Business Intelligence Case Study.* Cary, NC: SAS Institute Inc.

Dorfman, Paul, and Don Henderson. 2015. "Data Aggregation Using the SAS Hash Object". Available at https://support.sas.com/resources/papers/proceedings15/2000-2015.pdf.

Dorfman, Paul. 2014. "The SAS® Hash Object in Action". Available at: http://www.lexjansen.com/wuss/2014/113_Final_Paper_PDF.pdf

Burlew, Michele M. 2012. *SAS® Hash Object Programming Made Easy.* Cary, NC: SAS Institute Inc.

Secosky, Jason and Janice Bloom. 2007.   "Getting Started with the DATA Step Hash Object". Available at: http://www2.sas.com/proceedings/forum2007/271-2007.pdf

ACKNOWLEDGEMENTS

The author wishes to thank Paul Grant, for his review and advice, and Paul Dorfman and Don Henderson for their 2016 presentation at BASUG, which got me hooked on hash.

CONTACT INFORMATION

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

Elizabeth Axelrod
Abt Associates Inc.
10 Fawcett Street
Cambridge, MA 02138
elizabeth_axelrod@abtassoc.com