Paper 81-2008

Many-to-Many Merges in the DATA Step

Ed Heaton, Westat, Rockville, MD

ABSTRACT

When we want to join two tables on a many-to-many merge, we need a crosswalk table. The crosswalk table contains the keys for the two tables that we want to join. Once we have the crosswalk, we can use a nested join in **Proc sql** to create an SQL View. But sometimes we want a data view that utilizes features of the SAS® **Data** step that cannot be easily duplicated in **Proc sql**. We can perform this merge in a **Data** step with the **hash** object.

This paper will use an example where we join a data table about papers with a data table about authors. Some authors submitted more than one paper for consideration, and some papers were written by more than one author. We will create a **Data** step view that joins the two tables with a **Set** statement and two **hash** objects. This merge runs in half the time of the SQL join. Furthermore, it opens the door to **Data** step processing in the merge.

PREVIOUS WORK

Premier work on hash tables was championed by Paul Dorfman as early as 2000 [1] – before SAS gave us the tool as an implemented **Data** step object. Further work on the user-implemented hash table was presented by Dorfman and Gregg Snell in 2003.[2] Then, when SAS provided the **Data** step hash object with SAS 9, many authors wrote papers on the topic. (Dorfman & Vyverman, [3,4] Secosky & Bloom, [5] Muriel, [6] et al.) In 2005, Judy Loren and Sandeep Gaudana taught us how to join tables using a hash object at NESUG.[7] Then Greg Snell presented the technique at SUGI 31.[8]

INTRODUCTION

This paper will investigate a technique to join three tables on two independent keys. It will do so in a single **Data** step. This allows us to use the technique to create a data view, and still use the features that are unique to the **Data** step. We will look at two techniques.

One technique joins tables that are either sorted or indexed on the primary keys. This technique joins one table and the crosswalk with the familiar **Merge** statement. Then it includes the other table through a hash object.

The other technique is for tables that are not in any particular order. The techniques involve reading the crosswalk table with the familiar **Set** statement, and the inclusion of the other two tables through **hash** objects.

Let's look at an example that will illustrate both methods.

We have a table of data about SESUG 2007 authors with an author key. We have another table about SESUG 2007 papers with a paper key. Some authors submitted more than one paper for consideration, and some papers were written by more than one author. This is a classic many-to-many merge. To make sure things go together as we want, we created a crosswalk that has two columns - the author key and the paper key. (If we have a composite key, then we will need more than just these two columns.)

THE TABLES

The **Paper** table looks – in part – like the following.

Obs	Paper Key	Section	Title
9	9	AD	Integrating SAS® and Microsoft .NET for
10	10	AD	Developing Flexible Reports to Respond t
14	14	AD	Two Portals, One Sign-On: Gateway to Uni
47	47	DM	Steps to Consider for Preparation of Dat
60	60	HW	How to use PROC SQL SELECT INTO for List
63	63	HW	A Pragmatic Programmers Introduction to
74	74	IS	The Power of the BY Statement
83	83	PO	%TRAVERSE: A Knuth-style Recursive Proce
98	98	PO	Job Scheduling with the SLEEP Command
120	120	SD	Constructing Factor-Covering Designs
123	123	SD	"Powerful" SAS® Techniques from the Wind
135	135	SD	Smart SAS® Tricks
138	138	TU	Understanding Why Your Macros Don't Work
140	140	TU	Macro Bugs - How to Create, Avoid and De
151	151	TU	Point-and-Click Style Editing in SAS® En

The **Author** table looks somewhat like the following.

Obs	Author Key	FirstName	Sas Experience
9	9	Mike	15
10	10	Mai	2
14	14	Patricia	1
47	49	Jake	2
60	62	Jimmy	.
63	69	Ali	.
74	83	Andrew	26
83	100	Imelda	18

Obs	Author Key	FirstName	Sas Experience
98	116	Kevin	.
120	143	Tonya	0
123	146	Adrienne	.
135	159	Andrea	0
140	167	Monal	5
151	183	Linda	2
153	187	Eric	12
171	215	Brian	.

Finally, we need a crosswalk table – **PaperAuthor** – that matches paper keys with author keys. It looks somewhat like this.

Obs	Paper Key	Author Key
9	111	158
10	2	2
14	97	136
47	123	150
60	141	179
63	139	165
74	94	133
83	129	178
98	76	102

Obs	Paper Key	Author Key
120	25	34
123	110	154
135	110	155
138	8	9
140	148	84
151	128	176
153	111	157
171	130	180
184	69	92

METHODS

Three methods we will look at include the SQL join, the common sort-and-merge technique, and a one **Data** step method.

SQL JOIN

We want to join (merge) three tables on two different keys. Our usual Base SAS® approach is to join the tables with **Proc sql** – creating either a data file or a view.

```
Proc sql ;
    Create view PapersAndTheirAuthors as
        select *
        from Paper natural right join (
            select * from PaperAuthor natural left join Author
        )
    ;
Quit ;
```

The natural join in **Proc sql** is an equi-join assuming the join key is all the variables from both datasets where both the name and the data type is the same. These key variables will be written only once to the output dataset. If you have any like-named variables that you do not want to be in the key, you will need to use a dataset option to drop or rename one of them.

SORT, SORT, MERGE, RESORT, SORT, AND THEN MERGE AGAIN

Sometimes we want to do things after we put the files together that we can do in the **Data** step but not in **Proc sql**. This takes six steps by traditional methods and thus cannot be done as a data view.

```
Proc sort data=Paper ;        By PaperKey ;  Run ;
Proc sort data=PaperAuthor ;  By PaperKey ;  Run ;
Data PaperWithAuthorKey ;
    Merge
        Paper
        PaperAuthor( in=inPaperAuthor )
    ;
    By PaperKey ;
    If inPaperAuthor ;
Run ;
Proc sort data=PaperWithAuthorKey ;  By AuthorKey ;  Run ;
Proc sort data=Author ;              By AuthorKey ;  Run ;
Data PapersAndTheirAuthors ;
    Merge
        PaperWithAuthorKey( in=inPaperWithAuthorKey )
        Author
    ;
    By AuthorKey ;
    If inPaperWithAuthorKey ;
Run ;
```

If our tables come to us sorted or indexed, we might be able to skip three of the sorts. But the data from the first **Data** step will be in the wrong order for the second merge; so it will have to be sorted.

MERGE WITH HASH OBJECTS

The **hash** object allows us a one-step method that uses the **Data** step and thus can be written as a data view.

Tables in random order

We will need two hash objects – one for the table of papers and one for the table of authors. We will read the crosswalk dataset with a **Set** statement. Then we will use the hash object's **find()** method to include the data from each of the hash tables based on it's key.

```
Data PapersAndTheirAuthors / view=PapersAndTheirAuthors ;
    Set RandomPaperAuthor ;
    If ( _N_ eq 1 ) then do ;
        If 0 then set RandomPaper RandomAuthor ;
        Declare hash p( dataset:'RandomPaper' , hashExp:16 ) ;
        p.defineKey('PaperKey') ;
        p.defineData( all:'yes' ) ;
        p.defineDone() ;
        Declare hash a( dataset:'RandomAuthor' , hashExp:16 ) ;
        a.defineKey('AuthorKey') ;
        a.defineData( all:'yes' ) ;
        a.defineDone() ;
    End ;
    If not p.find() ;
    If not a.find() ;
Run ;
```

In the above code, the **If 0 then set** … statement allows us to create space in the Program Data Vector for the variables from the RandomPaper and RandomAuthor datasets without actually moving any data into the PDV. Without that, the **all:'yes'** option in the **defineData()** methods would not know what constituted all of the variables. The **If not p.find()** and **If not a.find()** statements are equivalent to using the **in=** dataset option to keep only records where there is contribution from both datasets as in the sort-sort-merge-sort-sort-merge example above.

The **find()** method returns zero (false) if successful and a non-zero number (true) if the key was not found. That's counter-intuitive for most SAS programmers. Hence, the condition if there is a problem is rather ungainly. That said, we want to continue to the implied **Output** statement if the key exists – if **find()** returns zero – in both hash tables.

With this **Data** step method, we can create a data view which automatically updates when it's used and which can further process the input data in complex ways that are only available through the **Data** step.

Presorted tables

We can join the Paper and Crosswalk tables with a **Merge** statement if they are both sorted on the PaperKey. Then we will need one hash object to include the Author table.

```
Data PapersAndTheirAuthors / view=PapersAndTheirAuthors ;
    Merge
        PaperAuthor( in=inPaperAuthor )
        Paper
    ;
    By PaperKey ;
    If ( _N_ eq 1 ) then do ;
        If 0 then set Author ;
        Declare hash a( dataset:'Author' , hashExp:16 ) ;
        a.defineKey('AuthorKey') ;
        a.defineData( all:'yes' ) ;
        a.defineDone() ;
    End ;
    If inPaperAuthor ;
    If not a.find() ;
Run ;
```

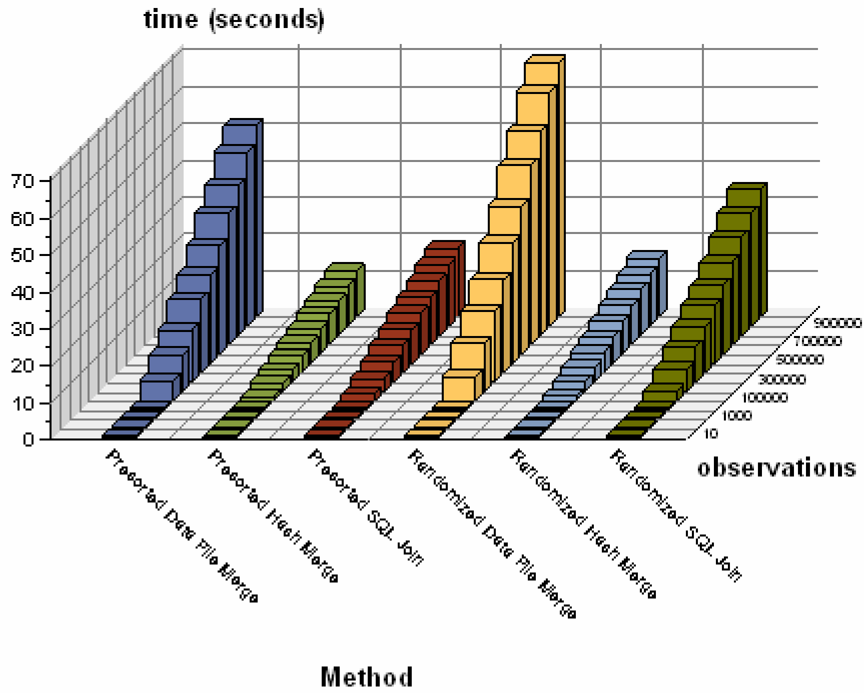
Although we are using only one hash table here, it's difficult to load because the data in the Author table is sorted and sorted data loads into a binary tree slowly because the nodes have to be continually rebalanced.

PERFORMANCE

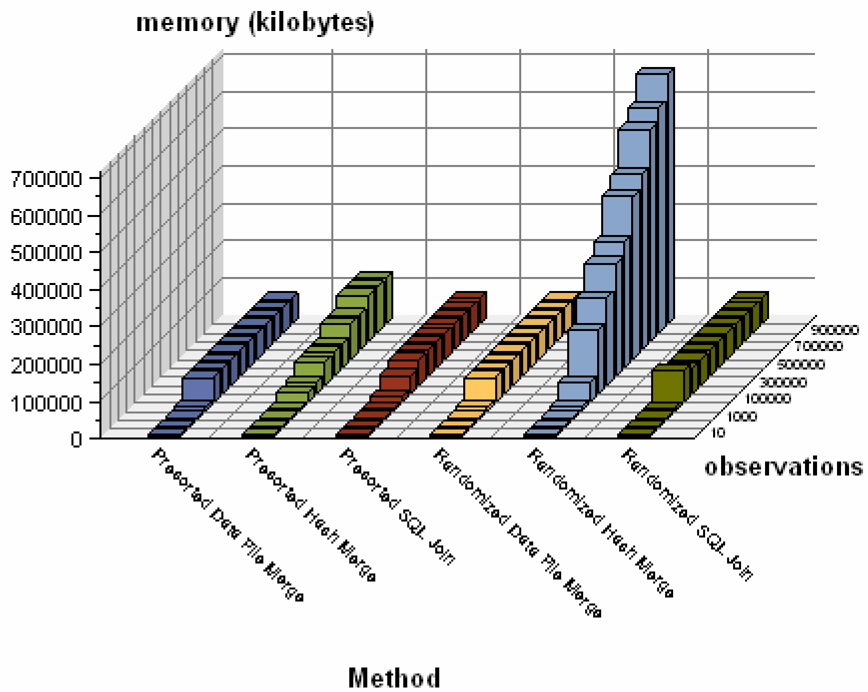
We gain not only the convenience of the option of using a **Data** step view, but performance gains, too. Let's look at test tables with equal numbers of observations in both the Papers and Authors tables, and nearly twice the number of observations in the crosswalk. The tables have as few as 10 observations in both the Authors and Papers tables and 16 rows in the crosswalk. The largest set of test tables had 1,048,576 rows in both the Authors and Papers tables and 2,096,710 rows in the crosswalk table. The Paper tables included one 2-byte character variable and twenty 25-byte character variables. The Author tables included one 2-byte character variable and ten 8-byte character variables.

Tests on the three outlined methods on both sorted and unsorted data are demonstrated in the following graphs. These compare the time to create the data view or data file and to then run a **Proc freq** on that data.

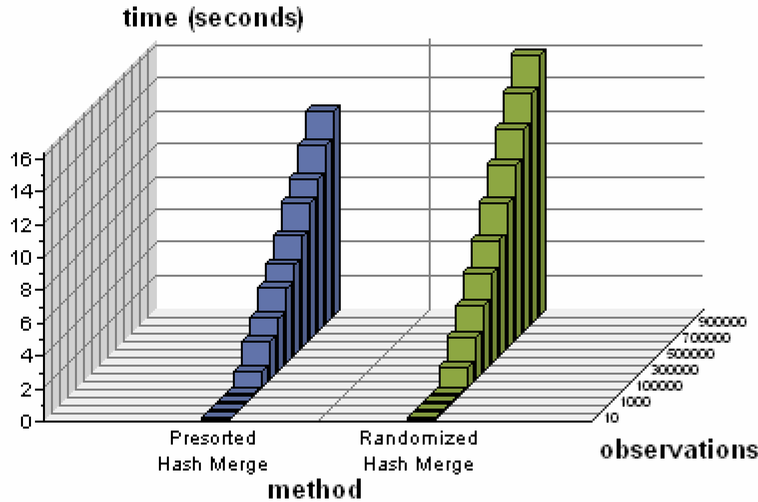
Total CPU Time



Gains in speed come at a cost, however. That price is memory.



We see relatively little gain (reduced CPU time) when we use presorted data and only one hash table.



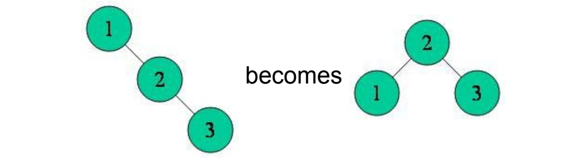
THE HASHEXP OPTION

The SAS hash object is an array of AVL trees [9] (self-balancing binary search trees). The hashing algorithm tells SAS where to store the data – which tree. The data for one key is stored in one node of the selected tree. The **hashExp** parameter tells SAS how many AVL trees we want. That is, **hashExp:0** requests 2^0 = 1 tree, **hashExp:1** requests 2^1=2 trees, and **hashExp:16** requests 2^{16} = 65,536 trees.

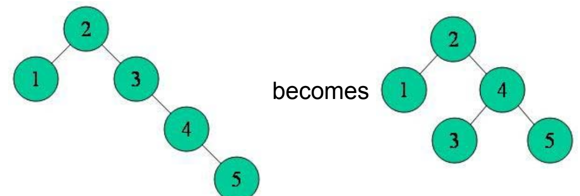
In an AVL tree, the height of the left and right child subtrees of any node can differ by no more than one. The process for adding sorted data to an AVL tree is somewhat like the following.[10]

First we put key=1 into the table. Then key=2 goes into the table as the right-child (since 2 is larger than 1). Then key=3 goes into the table as the right-child of the key=2 node.

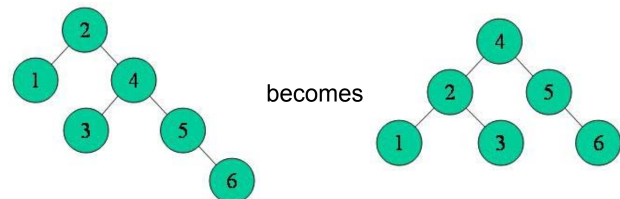
Now the height of the left child of the root node is zero and the height of the right child is two. So SAS performs a tree rotation – moving key=2 to the root node.



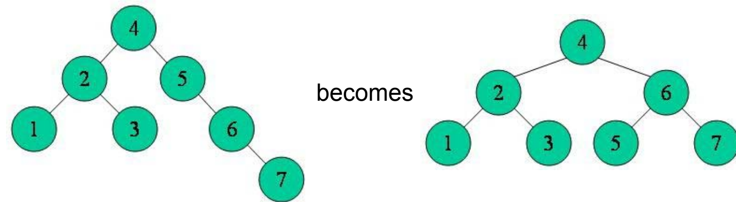
Additions of key=4 and key=5 requires a rotation in the right child subtree of the root node.



After key=6 is added the tree is rearranged using a rotation and a swap – to get a new root node and to move key=3 from the right child of the root node to the left child.



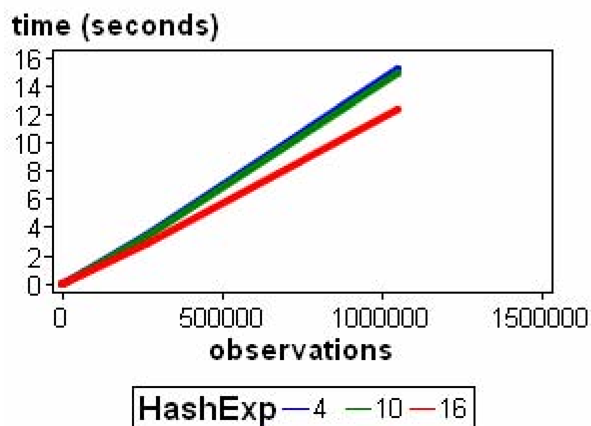
After key=7 is added, the tree needs to be rearranged again.



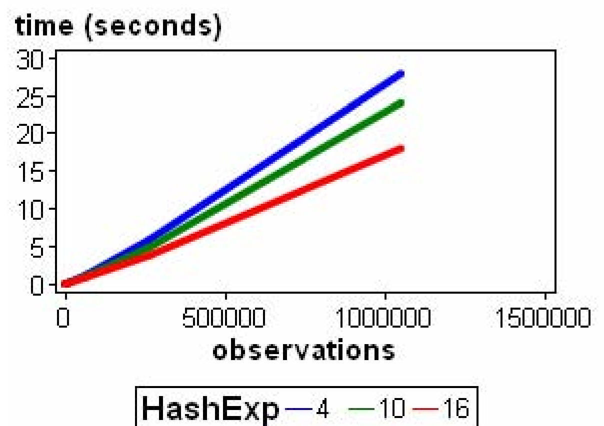
This would have been a lot more efficient if key=4 had been loaded first. In fact, the tree could loaded without a single rotation if the 4 were loaded first, 2 loaded before 1 and 3, and 6 loaded before 5 and 7. The sorted load is the least efficient method of all.

It makes sense that we could make this more efficient by keeping the binary trees small to minimize the amount of restructuring as the trees become unbalanced. To do this, we need more binary trees. Tests showed that using **hashExp:16** for the maximum number of binary trees yielded the fastest run times with very little cost in memory.

CPU Time with Presorted Data



CPU Time with Randomized Data



These graphs represent the median values from seven runs for each level of observations and **hashExp**. The tests were run under Windows XP Professional on a dual-core Intel® Pentium® D CPU clocked at 2.8 GHz with 1024 MB of RAM on a 200 MHz bus.

CONCLUSION

SAS programmers frequently need to implement a many-to-many join of two tables using an associated crosswalk table. While this is most commonly done using PROC SQL, other techniques exist that may offer advantages in convenience and/or performance. This is particularly true since the arrival of the **Data** step hash object. This paper has compared several of these techniques for creating a data view or data file. Creating the data view with **Proc sql** is programmer-efficient, but takes considerable time and memory when the view is used.

The merge-sort-merge method is considerably slower than either the **Proc sql** join or a one **Data** step method using hash tables. After we join the data with the merge-sort-merge method, use of the data can be much faster and can use much less memory, but the data view is impractical. So, the expensive merge-sort-merge must be performed each time the data changes.

If our crosswalk and one of the other tables are sorted on the same key, we can create a data view that joins the tables in a **Data** step by combining the **Merge** statement and the **hash** object. This runs considerably faster than the view created by **Proc sql** and with about half the memory. If none of our tables is sorted, we can join the tables with a **Set** statement for the crosswalk and two hash tables – one for each of the two tables we are joining. This runs twice as fast as the **Proc sql** method but the memory requirements are extensive.

If you are using the merged data more than once in a program, it makes sense to write to a data file rather than a data view.

REFERENCES

1. Dorfman, Paul M. Private Detectives in a Data Warehouse: Key-Indexing, Bitmapping, and Hashing. SUGI 25, Indianapolis, IN. 2000. (http://www2.sas.com/proceedings/sugi25/25/dw/25p129.pdf)
2. Dorfman, Paul M. and Gregg Snell. HASHING: GENERATIONS. SUGI 28. 2003. (http://www2.sas.com/proceedings/sugi28/004-28.pdf)
3. Dorfman, Paul M. and Koen Vyverman. Hash Component Objects: Dynamic Data Storage and Table Lookup. SUGI 29, 2004. (http://www2.sas.com/proceedings/sugi29/238-29.pdf)
4. Dorfman, Paul M. and Koen Vyverman. Data Step Hash Objects as Programming Tools. SUGI 31, 2006. (http://www2.sas.com/proceedings/sugi31/241-31.pdf)
5. Secosky, Jason and Janice Bloom. Getting Started with the DATA Step Hash Object. SAS Global Forum 2007. (http://support.sas.com/rnd/base/datastep/dot/hash-getting-started.pdf)
6. Muriel, Elena. Hashing Performance Time with Hash Tables. SAS Global Forum 2007. (http://www2.sas.com/proceedings/forum2007/039-2007.pdf)
7. Loren, Judy and Sandeep Gaudana. Join, Merge or Lookup? Expanding your toolkit. NESUG 2007, Portland ME. 2005. (http://www.nesug.org/proceedings/nesug05/pm/pm16.pdf)
8. Snell, Greg P. Think FAST! Use Memory Tables (Hashing) for Faster Merging. SUGI 31. 2006. (http://www2.sas.com/proceedings/sugi31/244-31.pdf)
9. Parman, Bill. How to implement the SAS® DATA Step Hash Object. SESUG 2006. (http://analytics.ncsu.edu/sesug/2006/SC19_06.PDF)
10. Gogeshvili, Arsen. AVL tree applet. (http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm)

CONTACT INFORMATION

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

Edward Heaton
Westat
1650 Research Boulevard
Rockville, MD 20850
Work Phone: (301) 610-4818
Fax: (301) 294-3992
Email: EdHeaton@Westat.com

The content of this paper is the work of the author and does not necessarily represent the opinions, recommendations, or practices of Westat.

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.