Paper 12-27

Hashing Rehashed

Paul M. Dorfman, Independent Consultant
Gregg P. Snell, Data Savant Consulting

ABSTRACT

You know hashing works—and fast! You've been to the presentations, even read the papers. But you still have not tried any hashing techniques. Why? Is it because you had a hard time understanding all those hkeys, collisions, and prime modulo thingies? Well, you're not alone. Fortunately, just as you do not need to be an auto mechanic to learn how to drive, you do not need to be a mathematician to learn how to use hashing.

This paper is a re-packaging of the direct addressing, key-indexing and hashing concepts presented by one of the authors at previous SUGIs. The proofs of the algorithms and rational for the logic are accepted as previously presented, and we now focus, instead, on how to apply the methods, not why they work. Simple examples are presented with code and graphics. (There is a reason why picture books are easier to read!) Our goal is to help you understand how these concepts work and show you how to use them yourself. Once learned, you too will be able to leap tall databases in a single bound!

INTRODUCTION

What we are commonly referring to as "Hashing" is actually a number of searching techniques, based on one central concept, that complement each other in producing the fastest table-lookup, period! And while some of the techniques might be difficult to grasp at first, once understood, they can be incredible tools in your SAS® Software arsenal. It is our hope that this rehash of the SUGI 26 Paper *"Table Look-Up by Direct Addressing: Key-Indexing – Bitmapping – Hashing"* (hereinafter referred to as "Last Year's Paper") will cause a few more light bulbs to come on such that you will actually try to use these methods. Amaze your friends! Impress your boss! You'll be a Hashing Superman!

As part of our explanation, we will be tackling a common problem with very large SAS data sets. Primarily, there are millions of records each with hundreds of variables, in a data set called CUSTOMER, which is already sorted by the *primary* key variable SSN (9-digit numeric). But we need to add fields linked to our data by *secondary* keys, other than this SSN.

Our first problem is that we want to add some new demographic data we just purchased, the number of Sport Utility Vehicles (SUV) licensed, that is keyed to zip code. Unfortunately, CUSTOMER is too large to re-sort by zip code because we do not have enough disk space. Besides, your boss wants to include this new demographic data in his presentation tomorrow so we have got to figure out a way to do it now!

Our second problem is very similar, but this time the variable we want to add is SPOUSE_AGE, which is keyed to SPOUSE_SSN. Again, sorting is out due to disk and time limitations, but (you guessed it) the boss wants this new data on the same report due tomorrow!

Key-Indexing and hashing techniques can ideally solve both of these problems. Shall we?

DIRECT ADDRESSING

The central concept, which is the basis for organizing table lookups faster than a speeding bullet, is *direct addressing*. Understanding direct addressing is fundamental to understanding how you can make use of key-indexing, bit-mapping and hashing. The key concept here is that of accessing key values "directly" by their location (*address, node*) in a table, as opposed to searching for them by comparing the search key to all or some table values. And it is quite easy to understand how direct addressing is far superior to comparison-based searching in the following simple example.

Let us assume we have a small (array) table with 10 elements:

```
array var_array (0:9) _temporary_
     (1 2 2 3 0 4 5 6 9 7);
```

and we want to know if the value of VAR_ARRAY is ever equal to 7. You might notice that the values of the table are not sorted and contain duplicates—a rather common phenomenon in the real world. A search for 7 might look like this:

```
%let searchkey = 7;

data _null_;
  array var_array(0:9) _temporary_
       (1 2 2 3 0 4 5 6 9 7);
  do i=0 to 9;
     if var_array(i) = &searchkey then do;
        put 'hit';
        stop;
     end;
  end;
  put 'miss';
run;
```

While this example is absurdly small, it illustrates the Achilles heel of a comparison search: Some (or even all) values in the table are compared to our target value of 7 until (or if) we find a match. But how can we use direct addressing in this example by directly referencing the array element that contains 7?

<whisper>
sppissst! This is where the magic happens…it's called key-indexing!
</whisper>

KEY-INDEXING

Key-indexing is the concept of using the value of a table's key variables as the index into that table. This enables us to directly address the table, as we know exactly what the addresses are for each value! Consider the following:

```
data _null_;
  array var_array(0:9) _temporary_
        (0 1 2 3 4 5 6 7 . 9);
  if var_array(&searchkey) ne . then
       put 'hit';
  else put 'miss';
run;
```

It doesn't take a rocket scientist to realize that this code is simpler and will execute much faster as we have eliminated all but one comparison! Now, at this point, you may be saying to yourself ***"Hold on there! You guys changed the order of the table!"*** Yes, we did and thank you for noticing. By definition, the values in a key-indexed table must be stored such that the key is the index. However, this is **extremely easy** to do programmatically and there are some fringe benefits as well!

So with that in mind, let us do it for our zip code problem, shall we? To use key-indexing, we want an array (table) large enough to hold all of the SUV counts for all possible zip codes (ARRAY SUV(0:99999) _TEMPORARY_). Then we want to load the array from the demographic file such that the value of SUV(zip) is either missing (.) or contains the SUV count for that customer's zip code. For example…

SUV(0)=.
SUV(1)=.
…
SUV(27513)=214
SUV(32258)=88
SUV(66216)=67
…
SUV(99999)=.

So our code might look something like this:

```
data TESTLIB.CUSTOMER;
  ** load suv counts into the array;
  array suv(0:99999) _temporary_;
  do until(eof1); /* source not sorted */
     set MYLIB.SUVBYZIP(keep=zipcode
                             suv_count)
                         end=eof1;
     if suv(zipcode) = . then
        suv(zipcode) = suv_count;
  end;
```

That's it! Now we have an array, called SUV, which contains either missing values (no zip or the count is missing) or the number of SUVs for that zip. Did you notice some of the fringe benefits of loading a key-indexed array? The **source data did not require sorting** because, by definition, we always know exactly where the values will be stored in the array. Also, our table is **naturally de-duped** because we only set the array value once. The rest of our program is now very simple and straight-forward:

```
  ** add suv_count to master data set;
  do until(eof2);
     set PRODLIB.CUSTOMER end=eof2;
     /* assign count by directly
        addressing the array*/
     suv_count=suv(zip5);
     /* be sure to drop unwanted fields
        introduced during table load */
     drop zipcode;
     output;
  end;
run;
```

We used the *key* variable ZIPCODE as the *index* to the table and were able to assign the SUV counts without any comparisons by *directly addressing* the *key-indexed* table. Neither data set required sorting and had there been duplicates in our demographic data, they would not have presented a problem! Pretty slick, eh?

Unfortunately, there are two primary limitations to using key-indexing—*memory* and *index size*. For this example, we only needed enough memory to hold a temporary array that was 8-byte numeric by 100,000 rows, or 800K. But as the size of the data to be added gets larger, so does the memory requirement.

The second primary limitation to using key-indexing is how large the range of the key variable can become before exceeding memory (not to mention it must always be integer). 100,000 zips may not be a problem, but what about 9-digit Social Security Numbers (SSN)? That would require 8 GB, which is not chump change even with contemporary memories. And adding just one digit to the key range would increase memory usage ten-fold! How can one possibly use key-indexing when SSN (or any other large or non-integer value) is the key?

Since memory and index size represent our primary limitations, our solution must address one or both of these issues.

<whisper>

sppissst! This is where the REAL magic happens… actually, there are several pieces of magic but we won't try to explain them all, just enough to get you going…

</whisper>

BITMAPPING

Bitmapping dramatically improves memory utilization by using a single bit (1 bit) rather than 8 bytes (64 bits) of memory to the presence of absence of a numeric lookup key. Given the same memory resource, bitmapping has the addressable key range 53 to 56 times that of key-indexing alone. While using bitmapping for adding satellite data to a master file can be challenging, it is a champion when we only need to rapidly find out if the record with a given key should be selected, and if memory resources are sufficient for mapping the entire key range into memory bits.

In this paper, we have chosen to not expand on the bitmapping technique. A full and detailed explanation can be found in "Last Year's Paper".

HASHING

Hashing actually addresses *both* the memory and index size issues. Now, before we go any farther, does everybody know what *hashing* means? Hashing is the process of converting a long-range key (numeric or character) to a smaller-range integer number with a mathematical algorithm or function:

$$\text{LONG-RANGE KEY} \xrightarrow{\textbf{HASH}} \text{SMALLER-RANGE INTEGER}$$

Here, let us explain it to you with a simple example before we attempt to use it on our CUSTOMER database. Let us presume our database has only 10 rows and the key is 3-digits like this:

```
data small;
  input key;
  cards;
  185
  971
  400
  260
  922
  970
  543
  532
  050
  067
  ;
run;
```

Now, on such a small key, we could use key-indexing with a simple array of 1,000 elements (ARRAY KEY(0:999)) but only 10 cells out of 1,000 would be occupied while the rest is wasted memory. But what if we could "hash" these ten 3-digit keys to fit in a smaller array, take up less memory, and still make use of direct addressing?

Ideally, if we had a hash function that would map each of the ten 3-digit values to its own *hash address* from 0 to 9 we would have the *perfect hash function*. However, it would take a great deal of time to develop such a function, which might turn out to be quite slow to compute and would also fail as soon as the keys changed or if you have more than 10 records. In reality, we will use a less-than-perfect function. It will not guarantee a one-to-one mapping between the keys and their hash addresses, i.e. it will allow some keys to map to the same hash address – a phenomenon called a *collision*. However, the function will be able to not pile too many keys on one address, it will be fast to calculate, and good for any input set of keys, not just one. Any collisions that ensue will be dealt with separately by invoking a *collision resolution policy*.

A fairly fast, simple, and uniform hash function can be obtained by dividing the key (or its numeric representation if the key is character) by a prime number and using the remainder as a hash address. A detailed discussion of such a *division method* can be found in “Last Year’s Paper”. Essentially, there are **only three things** you will need to know in order to effectively use this method.

First, what is the load factor of the hashing array? In other words, after all the keys have been hashed and loaded into the array, how full do you want it to be? The sparser the array is populated (i.e. smaller load factor) the faster searching will be but at the expense of using more memory for the larger array. A more densely populated array is somewhat slower (due to more collisions) but requires less memory. For many applications of this technique, a load factor of 0.8 yields a good balance of performance and memory requirements. Let us assign that to a macro variable:

```
%let load = 0.8;
```

Secondly, you need to know what the optimal array size should be for the given load factor. This is one of those “magic” things we won’t spend much time on since it requires some knowledge of number theory to really understand. Fortunately, the author of this function has already provided us with a short and extremely fast SAS program to dynamically calculate and store the optimal value in a macro variable called HASH_SIZE. Let us use this program on the SMALL data set we used previously:

```
data _null_;
  do p=ceil(p/&load) by 1 until(j=up+1);
     up = ceil(sqrt(p));
     do j=2 to up until(not mod(p,j)); end;
  end;
  call symput('hash_size',left(put(p,best.)));
  stop;
  set small nobs=p;
run;

%put hash_size=&hash_size;

hash_size=13
```

So, if we use the load factor of 0.8, our optimal array size would be 13. Now, let us see what effect this has on the hashed values in our example:

```
data _null_;
  set small;
  hash_addr = mod(key,&hash_size)+1;
  if _n_=1 then put 'KEY        HASH_ADDR';
  put key Z3. '  ===>  ' hash_addr Z2.;
run;
```

```
KEY        HASH_ADDR
185   ===>   04
971   ===>   10
400   ===>   11
260   ===>   01
922   ===>   13
970   ===>   09
543   ===>   11
532   ===>   13
050   ===>   12
067   ===>   03
```

From this example you can see that our hash function has produced two collisions: 400 and 543 both hashed into 11 and 922 and 532 hashed into 13.

That brings up the **third** thing you need to know about this method: how to resolve collisions. But before we go any farther, you need to be aware that we will be deviating from pure direct addressing *because* of the potential collisions.

For with each collision, we will be altering the original hashed number in order to move our index to an empty node (un-collided address) in the array. Consequently, we can no longer rely upon the simple fact that ARRAY(HASH_ADDR) is not missing to identify a match. We will have to store the KEY in each node to enable further confirmation that ARRAY(HASH_ADDR)=KEY. And should this initial comparison fail, we will not yet know if KEY is missing or simply stored elsewhere due to a collision, until we follow our chosen resolution policy to its logical end. This makes hashing a *hybrid algorithm* because it combines direct addressing with short, and hence very fast, bursts of sequential searching.

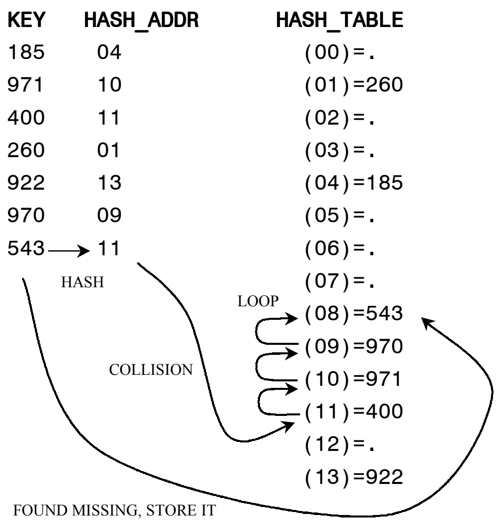
Now, hang with us folks! We know this part is a little tougher to follow but it is the key to understanding how to use hashing and we are only going to discuss two methods: linear probing and coalesced chaining.

COLLISION RESOLUTION POLICY: LINEAR PROBING

Collision resolution with Linear Probing is, perhaps, the simplest method of all. As the name implies, when a collision occurs, the remaining cells in the table are simply probed in a linear fashion (decrementing the index by 1) until the next empty address is found. If our index becomes less than 0 then we have stepped off the bottom of the table and need to reset the index to the top, and continue until encountering either a duplicate key or empty address. The simplicity of this method becomes quite evident when you realize this entire wrap-around cycle can be coded in a single DO loop. A program using SMALL might look like this:

```
data _null_;
array hash_table(0:&hash_size)
      _temporary_;
  ** load and link the hash table;
  do until(eof1);
     set small end=eof1;
                    /* hash the key */
     do hash_addr=mod(key,&hash_size)+1
       /* decrement loop */
        by -1 until
        /* missing found */
        (hash_table(hash_addr)=. or
       /* duplicate found */
        hash_table(hash_addr)=key);
        /* stepped off table, start over */
       if hash_addr < 0 then
          hash_addr=&hash_size-1;
     end;
     /* store the key */
     hash_table(hash_addr) = key;
  end;
  /* all done, write results to the log */
  put 'hash_table';
  do i=0 to &hash_size;
     put '(' i z2.')=' hash_table(i) z3.;
  end;
run;

hash_table
(00)=.
(01)=260
(02)=.
(03)=067
(04)=185
(05)=.
(06)=.
(07)=050
(08)=543
(09)=970
(10)=971
(11)=400
(12)=532
(13)=922
```

Our first collision occurred when 543 hashed into 11 because HASH_TABLE(11) was greater then missing. Visualizing our code at that point might look like this:



The main advantage of linear probing is its utter simplicity. And, if the table is sparse enough, it performs quite well! As a rule of thumb, linear probing performs best if about half of all nodes in the table are left empty, i.e. with the load factor of about 0.5. However, this represents a significant problem when you consider that a smaller load factor corresponds to a larger array and dramatically larger memory requirements. Also, linear probing performance rapidly deteriorates, as the table gets fuller.

Which leads us naturally to what might be considered the best collision resolution scheme:

COLLISION RESOLUTION POLICY: COALESCED CHAINING

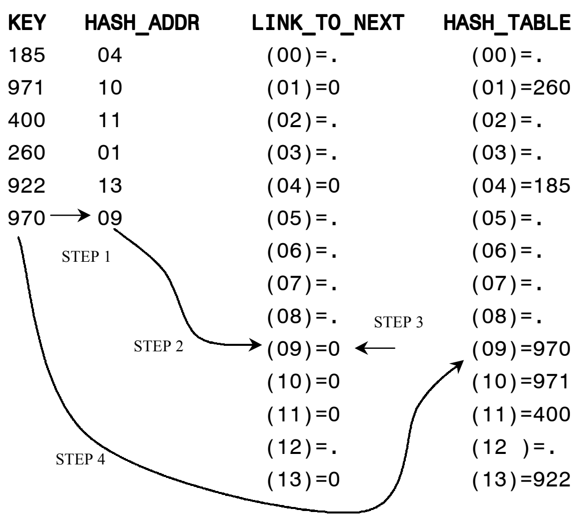
Coa-what? *Coalesce* simply means to unite into a whole. Earlier, we have seen that the key(s) hashing to the same address form a “chain”, which we need to traverse sequentially. We can create one array for each chain and write very simple code to implement such *separate chaining*. But since each array must be allocated to its maximum possible size, the simplicity comes at the expense of a huge memory cost. In coalesced chaining, we avoid this by coalescing all the chains together into a “whole” array. The idea is fairly simple in that we will maintain a parallel array, called LINK_TO_NEXT that will contain one of 3 distinct values:

- missing if the corresponding node of HASH_TABLE is empty,
- 0 if the corresponding node of HASH_TABLE contains a hashed value and no other keys have “collided” with it, or
- a number that points to the next cell with a hashed number.

Let us go back to our SMALL example. The first six values hash without collisions, so stepping through our process:

1. hash KEY into HASH_ADDR
2. check to see if LINK_TO_NEXT(HASH_ADDR) is missing
3. found a missing LINK_TO_NEXT cell so the corresponding HASH_TABLE cell is also missing; store 0 in LINK_TO_NEXT to mark it as filled
4. store KEY in HASH_TABLE(HASH_ADDR)

and our arrays would look like this:



KEY	HASH_ADDR	LINK_TO_NEXT	HASH_TABLE
185	04	(00)=.	(00)=.
971	10	(01)=0	(01)=260
400	11	(02)=.	(02)=.
260	01	(03)=.	(03)=.
922	13	(04)=0	(04)=185
970	09	(05)=.	(05)=.
		(06)=.	(06)=.
		(07)=.	(07)=.
		(08)=.	(08)=.
		(09)=0	(09)=970
		(10)=0	(10)=971
		(11)=0	(11)=400
		(12)=.	(12)=.
		(13)=0	(13)=922

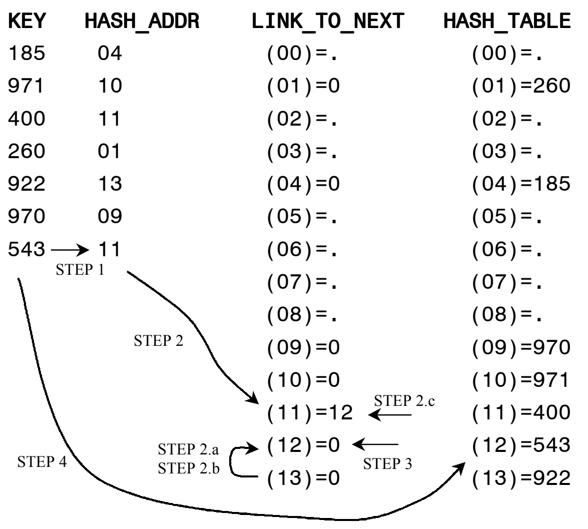
The next KEY of 543 hashes into 11, but 11 is already occupied because LINK_TO_NEXT(11) is greater than missing. So our process now becomes:

1. hash KEY into HASH_ADDR
2. check to see if LINK_TO_NEXT(HASH_ADDR) is missing
   a. if not, see if the key already exists by following the chained addresses in LINK_TO_NEXT until we find it in HASH_TABLE, or find a 0.
   b. If not found, then beginning with the top of LINK_TO_NEXT, search backward until we find a missing value.
   c. Store this index in the original LINK_T0_NEXT(HASH_ADDR) (it was a 0) before changing HASH_ADDR to this index
3. found a missing LINK_TO_NEXT cell so the corresponding HASH_TABLE cell is also missing; store 0 in LINK_TO_NEXT to mark it as filled
4. store KEY in HASH_TABLE(HASH_ADDR)

SAS code for this process becomes thus:

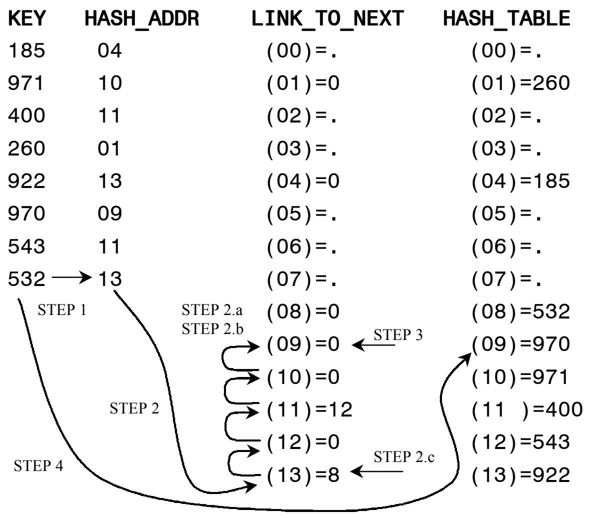
```
data _null_;
 array hash_table(0:&hash_size)
          _temporary_;
 array link_to_next(0:&hash_size)
          _temporary_;
 ** load and link hash table using SMALL;
 do until(eof1);
   set small end=eof1;
   /* STEP 1 (hash the key) */
   hash_addr = mod(key,&hash_size)+1;
   found = 0;
   /* STEP 2 (collision?) */
   if link_to_next(hash_addr) >. then do;
   /* STEP 2.a (check for duplicates) */
     link traverse;
     if found then continue;
     /* STEP 2.b (follow next_key to
          empty node) */
     do next_key = &hash_size by -1
        until(link_to_next(next_key)=.);
     end;
     /* STEP 2.c (change original link
          from 0 to next */
     link_to_next(hash_addr)=next_key;
     hash_addr = next_key;
    end;
    /* STEP 3 (mark link as occupied) */
    link_to_next(hash_addr) = 0;
    /* STEP 4 (store the key) */
    hash_table(hash_addr) = key;
 end;
 /* all done, write results to the log */
 put 'link_to_next   hash_table';
 do i=0 to &hash_size;
  put '(' I z2. ')=' link_to_next(i) z2.
  @15 '(' i z2. ')=' hash_table(i) z3.;
 end;
 stop;
 /* see if key already exists */
 traverse:
   if key =   hash_table(hash_addr)
      then found=1;
   else if link_to_next(hash_addr) ne 0
   then do;
      hash_addr=link_to_next(hash_addr);
      goto traverse;
   end;
run;

link_to_next   hash_table
(00)= .        (00)=   .
(01)=00        (01)=260
(02)= .        (02)=   .
(03)=00        (03)=067
(04)=00        (04)=185
(05)= .        (05)=   .
(06)= .        (06)=   .
(07)=00        (07)=050
(08)=00        (08)=532
(09)=00        (09)=970
(10)=00        (10)=971
(11)=12        (11)=400
(12)=07        (12)=543
(13)=08        (13)=922
```

Visualizing the process during the first collision might look like this:



KEY	HASH_ADDR	LINK_TO_NEXT	HASH_TABLE
185	04	(00)=.	(00)=.
971	10	(01)=0	(01)=260
400	11	(02)=.	(02)=.
260	01	(03)=.	(03)=.
922	13	(04)=0	(04)=185
970	09	(05)=.	(05)=.
543	11	(06)=.	(06)=.
		(07)=.	(07)=.
		(08)=.	(08)=.
		(09)=0	(09)=970
		(10)=0	(10)=971
		(11)=12	(11)=400
		(12)=0	(12)=543
		(13)=0	(13)=922

And during the subsequent collision of 532:



KEY	HASH_ADDR	LINK_TO_NEXT	HASH_TABLE
185	04	(00)=.	(00)=.
971	10	(01)=0	(01)=260
400	11	(02)=.	(02)=.
260	01	(03)=.	(03)=.
922	13	(04)=0	(04)=185
970	09	(05)=.	(05)=.
543	11	(06)=.	(06)=.
532	13	(07)=.	(07)=.
		(08)=0	(08)=532
		(09)=0	(09)=970
		(10)=0	(10)=971
		(11)=12	(11)=400
		(12)=0	(12)=543
		(13)=8	(13)=922

A piece of cake, right? Come on now, it's not that bad. Besides, you have to get those spouse ages added to CUSTOMER by this afternoon and hashing is the only way you can pull-off such a super-human feat! Recalling the three things we need to know about hashing, let us apply them to our problem:

1. The **load factor** (how full do you want the array to be?)
   The spouse age file is pretty big so we better start with the recommended factor of 0.8 and if we run out of memory, up it to 0.9 or higher (we may also have to invoke SAS with MEMSIZE= option greater than the default of 64M)
2. The **optimal array size** given our load factor
   This part is very easy because we just run the magic code provided in "Last Year's Paper".
3. **Collision resolution policy**
   Big keys (SSN) and a big table point us to using coalesced chaining.

Here we go…UP, UP and AWAY! <screeeech> Nope. Wait. Hold everything! We forgot something very necessary for this particular problem. The SPOUSE_AGE, remember? How are we going to add this field if we *have* to store SSN in the HASH_TABLE because of collisions? Simple! Just add another array that will hold the ages, but we only have to load or unload it *after* the collision stuff is all over and we are ready to work with the key.

Ok, now we are ready…

```
%let load = 0.8;

data _null_;
  do p=ceil(p/&load) by 1 until(j=up+1);
     up = ceil(sqrt(p));
     do j=2 to up until(not mod(p,j));
     end;
  end;
  call symput('hash_size',left(put(p,best.)));
  stop;
  set MYLIB.SPOUSE_AGE nobs=p;
run;

data TESTLIB.CUSTOMER;
  array hash_table(0:&hash_size)
        _temporary_;
  array link_to_next(0:&hash_size)
        _temporary_;
  ARRAY AGES(0:&hash_size) _temporary_;
  ** load and link hash table using SSN
     from spouses, then add ages;
  do until(eof1);
   set MYLIB.SPOUSE_AGE(keep=ssn age
         rename=(ssn=SPOUSE_SSN
                 age=AGE_TO_ADD)) end=eof1;
   /* STEP 1 (hash the key) */
   hash_addr=mod(SPOUSE_SSN,&hash_size)+1;
   found = 0;
   /* STEP 2 (collision?) */
   if link_to_next(hash_addr) > . then do;
     /* STEP 2.a (check for duplicates) */
     link traverse;
     if found then continue;
     /* STEP 2.b (follow next_key to
        empty node) */
     do next_key = &hash_size by -1
        until(link_to_next(next_key)=.);
     end;
     /* STEP 2.c (change original link
        from 0 to next) */
     link_to_next(hash_addr) = next_key;
     /* (set new hash_addr) */
     hash_addr = next_key;
  end;
```

```
   /* STEP 3 (mark link as occupied) */
   link_to_next(hash_addr) = 0;
   /* STEP 4 (store the key) */
   hash_table(hash_addr) = SPOUSE_SSN;
   AGES(HASH_ADDR) = AGE_TO_ADD;
   /* be sure to drop unwanted fields
      introduced during table load */
   drop AGE_TO_ADD;
   end;
   ** now add the values in hash_table to
      CUSTOMER;
   do until(eof2);
      set PRODLIB.CUSTOMER end=eof2;
      found = 0;
    hash_addr=mod(SPOUSE_SSN,&hash_size)+1;
      if link_to_next(hash_addr) > . then
         link traverse;
      if found then do;
         SPOUSE_AGE=AGES(hash_addr);
      end;
      /* remember to output each record */
      output;
   end;
   stop;
   /* see if key already exists */
   traverse:
      if SPOUSE_SSN=hash_table(hash_addr)
         then found=1;
      else if link_to_next(hash_addr) ne 0
      then do;
         hash_addr=link_to_next(hash_addr);
         goto traverse;
       end;
run;
```

Now, wasn't that easy? Outside of comments, we only had to change about 12 lines of code (see upper-case).

CONCLUSION

Hopefully we have been able to remove at least some of the confusion that may have prevented you from using key-indexing and hashing in the past. Most of the code is extremely straight-forward and requires but a few minor changes to address most problems. And, at the sake of sounding like a broken record, we strongly encourage you to reread the SUGI 26 Paper "Table Look-Up by Direct Addressing: Key-Indexing – Bitmapping – Hashing". We have re-addressed some of the original concepts, but there are *many* more we did not cover and certainly not in as much detail.

Now that you have a better understanding of what "hashing" is all about, we hope you will try it. But, how do you decide *when* to use key-indexing or hashing over the more traditional approaches? Under certain conditions, as in the examples we presented, the decision is made for you. But assuming you could choose between a traditional SORT and MERGE or Hashing, what other criteria should you consider? **Speed**!

Nothing is faster (not even a speeding bullet)! Whether your problem is a one time hurry-up-and-get-it-done job for the boss, or a web-deployed database application that requires sub second response time, hashing will save the day and you too could become a hero!

REFERENCES

1. D. E. Knuth, The Art of Computer Programming, 2.
2. D. D. Knuth, The Art of Computer Programming, 3.
3. R. Sedgewick, Algorithms in C, Parts 1-4.
4. T. A. Standish. Data Structures, Algorithms and Software Principles in C.
5. P.M. Dorfman. Table Look-Up by Direct Addressing: Key-Indexing– Bitmapping–Hashing. SUGI 26.
6. J. Morris, Data Structures and Algorithms, Hash Tables(http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/hash_tables.html)

ACKNOWLEDGMENTS

P.M.D.: Special thanks to Karsten M. Self, Ian Whitlock, F. Joseph Kelley, Sigurd Hermansen, and Base SAS R&D team for their support of direct-addressing methods in SAS, valuable discussions full of ideas, wit, and vigor, and giving the author an opportunity to apply the techniques to solve practical problems. The author gratefully acknowledges the contribution of all individuals who have, directly or indirectly, encouraged the author and supported his efforts of making direct addressing an accepted and practically used DATA step philosophy.

G.P.S.: A very special thanks to Paul Dorfman, author of "the" paper and a very gracious and forgiving coauthor as well a consummate expatiator of the English language ;-) You are my Hashing hero!

CONTACT INFORMATION

Paul M. Dorfman
4437 Summer Walk Ct.
Jacksonville, FL 32258
(904) 260-6509 (h)
(904) 905-5428 (o)
pdorfman@bellsouth.net
paul_dorfman@hotmail.com
paul.dorfman@bcbsfl.com

Gregg P. Snell
Data Savant Consulting
5632 Noland Road
Shawnee, KS 66216
(913) 638-4640 (o)
(208) 977-1943 (f)
gsnell@datasavantconsulting.com
www.datasavantconsulting.com

TRADEMARK CITATION