Paper 3101-2019

Re-Mapping A Bitmap

Paul M. Dorfman, Independent SAS® Consultant

Lessia S. Shajenko, Senior Data Scientist, SVP, Bank of America

ABSTRACT

Bitmap lookup was introduced into SAS® programming in 2000. Yet, its virtues started to gradually fade from the mind of SAS programmers after the advent of the SAS hash object a few years later. One reason is that the hash object requires no custom coding. The other is that its table lookup functionality seems to supplant the utility of bitmapping. However, this impression is false: In its niche of searching against a massive number of limited-range integer keys, a properly constructed and accessed bitmap is both much faster and incurs much smaller memory footprint. In this paper, we will illustrate programming methods of allocating, mapping, and searching bitmaps based on both character and numeric arrays and discuss their relative pros and cons in terms of lookup speed and memory usage.

INTRODUCTION

The word "*bitmap*" has a number of meanings. In this paper, we use it in the sense of a SAS lookup table based on mapping a range of integer key-values into *bits*. Accordingly, by *bitmapping* we mean the algorithms used to organize, load, and search a bitmap in order to find if a given key-value is mapped.

At first glance, the utility of bitmapping may seem rather narrow: (a) it works only with integer keys, whose range equals the number of bits fitting into memory; and (b) searching a bitmap mainly answers the question whether a given key-value is present or absent (its ability to store and retrieve non-key data is marginal). However, in many real-world use cases these constraints are naturally satisfied. Furthermore, per *unit of memory used*, a bitmap (a) can hold far more key-values than any other type of lookup table and (b) its key mapping and searching operations vastly outperform any other lookup method in SAS.

Bitmapping was introduced into SAS nearly 20 years ago as part of the direct-addressing family of searching methods [Dorfman, 2000]. In this paper, in addition to the original bitmapping algorithms that utilize the mantissa bits of a numeric array, we present a recently developed technique making use of *every* bit of bytes in memory. Though this method is not as fast, it reduces the bitmap memory footprint, thus offering a choice between the speed and extended key-value range.

TWO WAYS TO SEARCH

Search algorithms can be classified in many ways. The scheme most relevant to bitmapping is to segregate search methods into those that rely (a) on comparisons between keys and (b) on digital properties of the keys. Consider the following list of 10 integer key-values to search:

```
03 15 06 04 12 11 14 05 01 08
```

and let us say that we want to find out if K=11 and K=02 are in the list. Looking at different ways of approaching this seemingly simple problem will help understand the distinction.

1. BY COMPARISONS BETWEEN KEYS

The simplest idea is to store the list in an array just the way it is and compare each item with K. If we find a match (as with K=11), we stop; otherwise, if we have visited every item

without finding a match (as with K=02), K is not in the array. Code-wise, it is done best by moving K into an extra array location (below, with index 0) as a *sentinel*:

```
data _null_ ;
  array qs [0:10] _temporary_ (. 03 15 06 04 12 11 14 05 01 08) ;
  K = 11 ;
  qs[hbound (SS)] = K ;
  do x = hbound (qs) by -1 until (qs[x] = K) ;
  end ;
  found = (X > 0) ;
  put K= found= ;
run ;
```

This variation of *sequential search* (sometimes called "quick" sequential search due to the sentinel trick) is as simple as it gets. Also, the table is easy and fast to load: We merely place the key-value in the next unoccupied position. However, search-wise, the scheme scales very poorly: If the number of items in the list is N, then on the average, we need N/2 key comparisons if we have a match and N+1 if we do not. Thus, if the list should grow T times, the number of comparisons (and the run time with it) would increase T times, too. Formally, this is expressed in so-called "big *O*" notation by saying that sequential search runs in *O*(N) time.

Apparently, from the standpoint of performance this type of table lookup organization and related search algorithm will not win any trophies. Search-wise, we can do much better by *ordering* the list first and then using *binary search*. For example:

```
data _null_ ;
  array bs [10] _temporary_ (03 15 06 04 12 11 14 05 01 08) ;
  call sortN (of bs[*]) ;
  K = 11 ;
  lo = lbound (bs) ;
  hi = hbound (bs) ;
  found = 0 ;
  do until (lo > hi or found) ;
    m = floor (divide (lo + hi, 2)) ;
    if K = bs[m] then found = 1 ;
    else if K < bs[m] then hi = m - 1 ;
    else                   lo = m + 1 ;
  end ;
  put K= found= ;
run ;
```

This is indeed much better, for every time the size of the key list is doubled, the number of key comparisons needed to find K (or find that it is not in the list) increases only by 1. Formally, this is expressed by saying that binary search runs in *O*(log(N)) time. In fact, *in general*, no search based on key comparisons can do better. However, the key comparisons are still there: For a table with 1 million key-values (not really all that big) and a random K, 21 key comparisons are needed on the average, whether it is hit or miss.

Besides, there are two other wrinkles. First, the table needs to be pre-sorted. Second, it is not at all simple to get the code right - for instance, merely forgetting to FLOOR the middle index M will cause the algorithm to fail.

2. BY DIRECT ADDRESSING

It raises the question whether or not a search table can be organized in such a way that searching it needs *no key comparisons at all*. At first glance, such a possibility may appear

counterintuitive. However, imagine that *all* the keys we are dealing with are limited to a restricted range of values. For example, suppose that our key-values are integers only in the range from 1 to 16. In this case, we can structure our lookup table differently. Namely, let us allocate a separate array item for *every possible key-value* and initially set all of them to zeroes:

```
array KX [16] _temporary_ (16 * 0) ;
```

Next, for each key-value in the list, let us replace 0 with 1 in the position of array KX whose *index is equal to the key-value* K itself. In other words, going through the original list, let us set `KX[03]=1, KX[15]=1,..., KX[08]=1.` The resulting table will look like this:

Position	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
Value	1	0	1	1	1	1	0	1	0	0	1	1	0	1	1	0

Note that we have not actually *stored* any key-value in the array: We have merely *mapped* each key-value to the array position with the index equal to the key-value itself. To put it yet another way, we have *marked* the array position with 1 as *occupied*, i.e. registered the key-value as *present*. Accordingly, the indices of the array positions filled with zeroes indicate the values *absent* from our search list.

Now, how can the table populated in this manner facilitate our search? Suppose we want to find whether or not K=11 is among the search keys we have mapped. Looking at KX[11], we see that KX[11]=1; therefore, K=11 is found. If, on the other hand, we want to search for K=2, we will see that KX[2]=0; therefore, it is not found. Apparently, by using this type of table lookup organization, we can tell whether a key is found or not without comparing it with any key in our list even once. In DATA step code:

```
data _null_ ;
  array KX [16] _temporary_ (16 * 0) ;
  *--- Mapping ----;
  do K = 03, 15, 06, 04, 12, 11, 14, 05, 01, 08 ;
    KX[K] = 1 ;
  end ;
  *--- Searching --;
  found = KX[K] ;
run ;
```

This kind of search is called, quite aptly, *key-indexed* search or just *key-indexing*. It is a most basic representative of the class of search methods done, not by using comparisons between keys, but by using their digital properties - in this case, the value of the key itself. Since search methods in this class map key-values directly into specific locations in memory - i.e., effectively, to memory addresses - the approach is termed *direct addressing*.

It should be quite obvious that no other search scheme can match key-indexing in terms of its speed of "inserting" a key-value in the table or searching for one, as both are executed merely by a single array reference. Moreover, it is obvious that an act of key-indexed search, whether it succeeds or fails, takes the same time *regardless of the number of key-values* mapped to the table, for every array item is referenced by its index exactly as fast as any other. In "big *O*" notation, this property is expressed by saying that the algorithm (in this case, key-indexing) runs in *O*(1), or *constant*, time.

FROM KEY-INDEXING TO BITMAPPING

However, nobody has ever proven the truism "There's ain't no such thing as a free lunch" wrong: By constructing key-indexed search, we have paid for its incredible speed by

stretching the array size to the entire range of key-values we can possibly have. Of course, with the range of [1:16] this is utterly insignificant; yet it demonstrates the principle: To make key-indexing work, we must allocate an index position for *every possible integer key-value*. In the intentionally simple case above, we had to use 16 positions, i.e. 6 more than the number of the key-values to search. However, as the key range grows, having a separate slot for every possible key-value becomes more and more onerous from the standpoint of memory usage.

For example, imagine that our keys are SAS dates reflecting daily events for, say, 100 years. It presents no challenge to key-indexing, as we would have to allocate a numeric temporary array with only about 37,000 items, which at 8 bytes per items is less than 300K of memory. However, if our keys were SAS datetimes in seconds, allocating an array for every second in 1 year alone would mean about 86400*365 8-byte items with memory usage close to 240M. It is still not all that bad; but if our keys were, for instance, 10-digit phone numbers, we would need an array with the memory footprint of almost 80G.

An astute SAS user might think that we could do better by allocating a *character* array with the item length $1 and mapping key-values using 1-byte literals (e.g. "1" and "0") and thus using 8 times less memory. Alas, as odd as it may seem, a $1 temporary array takes up *twice* the memory of a numeric array with 8-byte items (we will return to this topic later on for a different, albeit related, reason).

BIT ARRAYS?

In order to arrive at a better remedy, let us note that mapping key-values in our key-indexed table occurs in the *binary* fashion. In other words, each item in our table needs to be able to store only two values: One to indicate that a key-value is present and one - that it is absent. We may as well recall that the smallest piece of digital information capable of doing just that is a *bit*, as it can assume two values only: 0 and 1. It means that for a given integer key-value in a range, 1 bit is all we need to indicate its presence or absence.

Thus, the obvious way to address the problem would be to use a *bit array* instead and write *imaginary* code:

```
data _null_ ;
  array bits [16] _bit_ (16 * 0) ;
  *--- Mapping ----;
  do K = 03, 15, 06, 04, 12, 11, 14, 05, 01, 08 ;
    bits[K] = 1 ;
  end ;
  *--- Searching --;
  found = bits[K] ;
run ;
```

Well, *if it were possible*, we could stop right here. The problem is, there are no bit arrays in SAS - at least, not just yet; and so, the step above *will not work*. However, when SAS programmers need something not yet available, they try to make it happen in a roundabout way with what *is* available. In this case, it is possible: Bit arrays can be *emulated* thanks to SAS software being richly laden with other powerful features.

However, to emulate something, we ought to understand first what it is that we are trying to emulate. A bit array is just a string of *consecutive bits* in memory that can be referenced *programmatically* by an index in the same manner as a regular array. In SAS, we cannot do that because the smallest addressable piece of information is a byte - that is, each byte has its own physical address by which it can be accesses. But its 8 constituent bits do not; hence, we cannot access them directly. However, we can *expose* them from their byte using

various SAS tools, and manipulate them *as if* they were indexed. Let us see how it can be done.

BYTES AND BITS

Let us return to our simple sample of 10 denary key-values in the range of [1:16] and see how we can map them into bits. Since 1 byte has 8 bits, we shall need only 2 bytes - that is, 16 bits - to address the entire range. First, we need to allocate 2 bytes and initialize all their bits to 0. The most natural way to do that would be to use SAS bit literals, such as "01100110"b. Unfortunately, they can be used in comparisons but not in assignments. However, it can be done in many other ways, for example:

```
  bm = "0000"x ;
  bm = put (0, binary16.) ;
  bm = input ("0000000000000000", $binary16.) ;
```

The next step is to map our 10 key-values to the 2-byte (16 bits) *bitmap* BM above. It means that for K=(03, 15, 06, ...) we have to set the bits # 03, 15, 06, and so on of BM to 1. This is usually expressed by saying that they need to be *"turned on"* or *"set"* (in this argot, turning a bit "on" or setting a bit mean setting it to 1; and turning it "off" or unsetting it means setting it to 0). A brute-force way of doing it is:

1. Extract BM into a $8 variable (named BIN, say) using the $BINARY8. format.
2. Replace the needed positions in BIN with "1".
3. Use the $BINARY8. informat to rewrite BM with the needed bit is turned on.

Or, in the SAS language:

```
data _null_ ;
  bm = "0000"x ;
  bin = put (bm, $binary16.) ;
  *--- Mapping ----;
  do K = 03, 15, 06, 04, 12, 11, 14, 05, 01, 08 ;
    substr (bin, K, 1) = "1" ;
  end ;
  bm = input (bin, $binary16.) ;
  put bm=$binary16. ;
run ;
```

The PUT statement prints in the log:

```
bm=1011110100110110
```

and confirms that BM is properly *mapped* (or *compiled*). Now that we have out bitmap BM compiled, we can search it for any K in the [1:16] range. In order to do that, we only need to look at the Kth bit of BM:

```
data _null_ ;
  bm = "0000"x ;
  bin = put (bm, $binary16.) ;
  *--- Mapping ----;
  do K = 03, 15, 06, 04, 12, 11, 14, 05, 01, 08 ;
    substr (bin, K, 1) = "1" ;
  end ;
  bm = input (bin, $binary16.) ;
  *--- Searching --;
  do K = 11, 2 ;
    found = char (put (bm, $binary16.), K) = "1" ;
```

```
    put K=z2. found= ;
  end ;
run ;
```

The step prints in the log:

```
K=11 found=1
K=02 found=0
```

This rather *naive* approach is far from perfect, yet it demonstrates the principle: With the key-value range of [1:16], we need only 2 bytes to map all the key-values in the range and search them with a single *direct access* to the bitmap. Now let us convert this principle into something more practical and faster.

FORMING A BETTER BITE

The speed of mapping and searching is the name of the game. From this viewpoint, there are a few issues with the naive approach above:

1. Casting the whole bitmap BM to the PDV as a character variable BIN, 8 times the length of BM (using the $BINARY. format), is computationally expensive.
2. The SUBSTR pseudo-function is relatively slow.
3. Casting BIN back to BM is expensive, too.
4. If the key-value range should exceed 32767, storing the bitmap in a single character variable presents a problem (unless you are using a release higher than SAS 9.4 TS Level 1M4 and can allocate a character string of any length).

For the time being, let us set aside issue #4 (addressed at length later) and concentrate on performance. It is intuitively obvious (and actually true) that the fewer bytes we cast back and forth between BM and BIN, the faster bitmapping will work. Since the shortest piece we can transfer in this manner is 1 byte, it makes sense to consider working with only 1 byte at a time. Deciding on which byte and bit to work on is simple arithmetic:

- N_byte = ceil (divide (K, 8) = the number of byte we need
- N_bit = mod (K+7, 8) = the number of bit we need from N_byte

To check if the arithmetic is correct, let us calculate N_byte and N_bit for all K=1 to 16:

```
 K  N_byte  N_bit |  K  N_byte  N_bit
-----------------+-----------------
 01    01    01   |  09    02    01
 02    01    02   |  10    02    02
 03    01    03   |  11    02    03
 04    01    04   |  12    02    04
 05    01    05   |  13    02    05
 06    01    06   |  14    02    06
 07    01    07   |  15    02    07
 08    01    08   |  16    02    08
```

It does check out. Thus, given the value of K, we know which byte (N_byte) we need to extract from BM. Also, we know which bit (N_bit) within this byte we need to work on. We can now focus on issue #2, i.e. on finding a faster way to turn the bit in position N_bit on (in order to map K) or to find whether the bit is on or off (in order to search for K).

GOING BITWISE

Once again, the richness of the SAS toolset comes to rescue. Note that every byte of the bitmap is a character corresponding to an integer in the range [0:255]. We can cast that character as a SAS number using the RANK function or PIBw. informat; and once it is done, use the SAS *bitwise functions* to both turn the needed bit on and check whether it is on. Now let us consider different bitwise operations (e.g. mapping or search) separately.

Bitwise Mapping

To see how it can be done, let us start with a bitmap initialized to all zeroes and map K=03. To do it, we need to set bit #3 in byte #1 (see above) to 1. So, let us extract byte #1, compute its numeric equivalent N using the RANK function, and display the 8 least significant bits of its mantissa using the BINARY8. format:

```
data _null_ ;
  bm = "0000"x ;
  K = 03 ;
  N_byte = ceil (divide (K, 8)) ;
  N_bit = mod (K + 7, 8) ;
  byte = char (bm, N_byte) ;
  N = rank (byte) ;
  put N binary8. ;
run ;
```

The step prints:

```
00000000
```

This is as should be expected, as the bitmap bits at this point is all zeroes. Now we have to create a numeric variable whose third bit from the left (N_bit=3) is set to 1 regardless of the bits present in N. To do so, let us prepare a *bit mask* where all bits are zeroes except the third from the left:

```
00100000
```

In denary, this binary string corresponds to 2**5=32, so all we have to do to prepare the bit mask is to create a numeric variable M=32 whose binary image is `00100000`. Let us put the binary image of M underneath that of N:

```
00000000 (N)
00100000 (M)
```

To generate a value R with the third leftmost bit set to 1 (irrespective of any bits of N) with the bit mask M, we need to pair N and M using the so-called *bitwise OR* operation. In SAS, it is done by using the function BOR (i.e. Bitwise OR):

```
R = BOR (N, M) ;
```

The result of this operation will be:

```
00000000 (N)
BOR
00100000 (M)
------------
00100000 (R)
```

Having done that, we only need to cast R back to a 1-byte character value using the BYTE function and replace byte #N_byte in BM with this new value. At this point, the bit pattern in the first byte of BM is `00100000`, which means that K=3 has been mapped. Now if we

need to map the next key-value in our list that maps to byte 1 in BM, K=6, the bitwise OR operation will look as follows:

```
00100000 (N)
BOR
00000100 (M)
------------
00100100 (R)
```

After the bit pattern R is cast back to the first byte of BM, the latter's binary image will be 00100100, and so at this point both K=3 and K=6 will have been mapped successfully.

Note that the bit mask that needs to be applied depends on K - or, more exactly, on the value of N_bit it generates. Since we have 8 different bits, we have to use 8 different bit masks depending on which bit needs to be turned on, from 1 to 8:

```
Bit Mask | Binary Power | Denary Value
---------+--------------+-------------
10000000 |    2**7      |     128
01000000 |    2**6      |      64
00100000 |    2**5      |      32
00010000 |    2**4      |      16
00001000 |    2**3      |       8
00000100 |    2**2      |       4
00000010 |    2**1      |       2
00000001 |    2**0      |       1
-------------------------------------
```

These bit masks are in effect nothing but integer numbers that can be calculated on the fly as 2**(8 - N_bit). Better yet, to avoid on-the-fly computations, we can just pre-store them in a temporary array (named below BITMASK) indexed by N_bit. This way, we can use N_bit as an index into BITMASK to point at the correct bit pattern automatically.

Now let us apply this scheme to all key-values in our key list, one at a time. (Note that the code can be much simplified by nesting function calls; below, it is presented unnested, step-by-step, for the sake of clarity.)

```
data _null_ ;
  bm = "0000"x ;
  array bitmask [8] _temporary_ (128 64 32 16 8 4 2 1) ;
  *--- Mapping ----;
  do K = 03, 15, 06, 04, 12, 11, 14, 05, 01, 08 ;
    N_byte = ceil (divide (K, 8)) ;
    N_bit = 1 + mod (K + 7, 8) ;
    byte = char (bm, N_byte) ;
    N = rank (byte) ;
    R = BOR (N, bitmask[N_bit]) ;
    byte = byte (R) ;
    substr (bm, N_byte, 1) = byte ;
    put K=z2. bm $binary16. ;
  end ;
run ;
```

The SAS log shows the process of filling the bitmap with each successive key-value:

```
K=03 0010000000000000
K=15 0010000000000010
K=06 0010010000000010
K=04 0011010000000010
```

```
K=12 0011010000010010
K=11 0011010000110010
K=14 0011010000110110
K=05 0011110000110110
K=01 1011110000110110
K=08 1011110100110110
```

Now that the bitmap BM is compiled, we are ready to search it for a given key-value K.

Bitwise Search - Bit Checking

To search the bitmap, we first need to perform the same steps as during its mapping:

- Calculate N_byte and extract byte #N_byte
- Convert it to the respective number N using the RANK function
- Calculate N_bit and select the corresponding bit mask BITMASK[N_bit]

At this point, instead of *setting* the bit #N_bit to 1, we need to *detect* whether it is set to 1 (i.e. K is found) or `0` (i.e. K is not found). This, too, can be done by comparing the bit pattern of N with the bit mask where the bit #N_bit is the only bit set to 1. But this time around, we need a bitwise operation that would result in 1 only in the bit position where the *bits in both N and the bitmask are* 1. Effectively, we need to *multiply* the two bits in each position - and this is done by the operation known as *bitwise AND*. The corresponding function supplied by SAS is the BAND function (short for Bitwise AND).

For example, let us search for K=3. For this key-value, N_byte=1, and the bit pattern of byte #1 of BM is `10111101`. For K=3, N_bit=3. To see whether or not bit #3 in the first byte of our bitmap is on, it can be coupled with the bit mask `00100000` (denary M=32) this way:

```
10111101 (N)
BAND
00100000 (M)
------------
00100000 (R)
```

Since the result has 1 only in position #3, we thus know that bit #3 of N we are examining is set, and so our search is complete. Suppose that the next key-value we need to search for is K=6. This time, we need to apply the bit mask `00000100` (denary M=4) whose 6th bit is set:

```
10111101 (N)
BAND
00000100 (M)
------------
00000100 (R)
```

Again, since the result has the bit in position 6 on, K=6 is found. Now let us apply the same principle to look for K=2 using the bit mask `01000000` (denary M=64):

```
10111101 (N)
BAND
01000000 (M)
------------
00000000 (R)
```

This time, there are no bit positions where *both* N and M have 1, and so the result of BAND is all zeroes. Therefore, K=2 is not in the list. Note that though above, we have dealt only with the first byte of BM, in case when K > 8 (and so it maps to the 2nd byte with

N_byte=2), the same considerations remain fully valid. The only difference is that now BAND would compare the requisite bit masks with the bit pattern in the second byte of BM - that is, with 00110110.

After the value of R is returned, the only test we need to tell whether K is found is to see whether the numeric value of R=0 (K is not found) or R≠0 (K is found) because as a SAS numeric variable, R=0 only when all of its bits are set to 0. In fact, we do not even have to assign the value to R, as the result returned by BAND can be evaluated directly.

Bitwise Key Deletion - Turning A Set Bit Off

Bit setting and bit checking are analogous to the standard table operations *Insert* and *Search*. However, some tasks require an item already inserted into a table to be *deleted*. In terms of bitmapping, it means that the bit set to 1 for some key-value K should now be set back to 0, leaving the state of the rest of the bits intact.

To accomplish that, we need to compare the bits of BM with the bit mask where all bits are off except in the position where we need to turn our bit off. This is done using the BXOR (Binary Exclusive OR) SAS function.

For example, let us "delete" K=3 from the map - that is, turn the corresponding bit off. For K=3, N_byte=1, and the bit pattern of byte #1 of BM is 10111101, and N_bit=3. To set this bit off, we have to couple the first byte of BM with the bit mask 00100000 this way:

```
10111101 (N)
BXOR
00100000 (M)
------------
10011101 (R)
```

Likewise, if we then would like to turn bit #6 off, we would couple the first byte of BM with the bit mask 00000100:

```
10011101 (N)
BXOR
00000100 (M)
------------
10011001 (R)
```

As a result, we now have both K=3 and K=6 "deleted" from the bitmap BM. It is important that this method of turning a bit off does not affect any other bits of BM.

Let us now codify out findings: (1) map a number of key-values, (2) search the mapped bitmap for every K in the range from 1 to 16, and (3) turn off the bits set for K=03, 08, 15.

```
data _null_ ;
  bm = "0000"x ;
  array bitmask [8] _temporary_ (128 64 32 16 8 4 2 1) ;
  *--- Mapping ----;
  do K = 03, 15, 06, 04, 12, 11, 14, 05, 01, 08 ;
    N_byte = ceil (divide (K, 8)) ;
    N_bit = 1 + mod (K + 7, 8) ;
    byte = char (bm, N_byte) ;
    N = rank (byte) ;
    R = BOR (N, bitmask[N_bit]) ;
    byte = byte (R) ;
    substr (bm, N_byte, 1) = byte ;
  end ;
    *--- Search ----;
```

```
  do K = 1, 4, 12 ;
    N_byte = ceil (divide (K, 8)) ;
    N_bit = 1 + mod (K + 7, 8) ;
    byte = char (bm, N_byte) ;
    N = rank (byte) ;
    found = BAND (N, bitmask[N_bit]) ne 0 ;
    put K=z2. bm=binary16. found= ;
  end ;
  put "--------- Deletion -----" ;
  do K = 03, 08, 15 ;
    N_byte = ceil (divide (K, 8)) ;
    N_bit = 1 + mod (K + 7, 8) ;
    byte = char (bm, N_byte) ;
    N = rank (byte) ;
    R = BXOR (N, bitmask[N_bit]) ;
    byte = byte (R) ;
    substr (bm, N_byte, 1) = byte ;
    put K=z2. bm=binary16. ;
  end ;
run ;
```

The PUT statements print:

```
K=01 bm=1011110100110110 found=1
K=04 bm=1011110100110110 found=1
K=12 bm=1011110100110110 found=1
--------- Deletion -----
K=03 bm=1001110100110110
K=08 bm=1001110000110110
K=15 bm=1001110000110100
```

An alert reader will undoubtedly notice that code-wise, mapping, search, and deletion differ only beginning from lines where BOR, BAND, or BXOR are called. To avoid repetitive code, the remainder can be segregated into a LINK routine. Alternatively, the function calls can be nested. For example, on the search side, all 5 lines of code can be consolidated into a single line:

```
  found = ^^ BAND(rank(char(bm,ceil(divide(K,8)))),mask[1+mod(K+7,8)]) ;
```

(This style is particularly good for proverbial job security or if you want to amuse the guy who will have to maintain your code after you are gone.)

LONG-RANGE CHARACTER BITMAPPING

Thus far, we have proven the bitmapping concept by using integer key-values in the paltry [1:16] range and only 16 bits (2 bytes) of the character variable BM used as a bitmap. The same code will work without any alterations as long as our range lies within no more than 32767 contiguous integers because of the limit on the character variable length. (As a side note, it may be possible to work with longer strings for this purpose in SAS releases newer than SAS 9.4 TS Level 1M4; but we have not tested it.)

So, the next question is, how to emulate a bit array that would accept a longer key range. The answer to the question is that (a) you can create a temporary character array with an arbitrary number of items, as long as it fits in memory, and (b) the items of temporary arrays are contiguous in memory. Since their bytes occupy adjacent positions in memory, all of their bits do, too.

ARRAY MEMORY VAGARIES

The first natural inclination to use a character array for bitmapping is to create a character array with the item length $1, 8 bits per item. For example, suppose that we need to map the range from 1 to 2**20= 1,048,576 (about 1 million). It means that we need 2**20 consecutive contiguous bits in memory. If we create an array:

```
array bm [1048576] $ 1 _temporary_ ;
```

we will have what we need. Moreover, it would be programmatically convenient to use since each byte can be accessed simply as BM[N_byte]. Unfortunately, such an alluring proposition is not so good from the standpoint of memory usage. If you turn FULLSTIMER on and measure the memory footprint of such an array on a 64-bit Windows machine, you will see after some simple calculations that its memory usage is about 17 bytes of RAM per 1 item - that is, per 1 usable byte allocated via the array. By comparison, a *numeric* temporary array uses almost exactly 1 byte of RAM per 1 byte of allocated array storage. So, a numeric array with the same number of items as above (and so with 8 times the number of allocated bytes, as each numeric item has 8 bytes) actually uses less than 1/2 the memory. (It is a good hint at the possibility of using numeric arrays for bitmapping, and we will sure discuss it at length later.)

For the time being, let us see if we still can use a character array to allocate the same number of bytes/bits as in array BM above - but hopefully at a much reduced memory cost. First, it is obvious that the same number of usable bytes can be allocated by using an array with N times fewer items but N times greater item length. For example, these arrays:

```
array bm [1048576] $    1 _temporary_ ;
array bm [ 131072] $    8 _temporary_ ;
array bm [  16384] $   64 _temporary_ ;
array bm [   2048] $  512 _temporary_ ;
array bm [    256] $ 4096 _temporary_ ;
```

all allocate the same exact number of usable bytes (and bits) as far as bitmapping is concerned. However, their memory footprints differ quite dramatically. It can be observed in the table below, which we have compiled by measuring their RAM usage with various numbers of items and proportionally sized item lengths.

N_ITEMS	ITEM_LENGTH	RAM_KB	RAM_PER_BYTE
1,048,576	1	17633.62	17.22
131,072	8	3297.62	3.22
16,384	64	1505.62	1.47
8,192	128	1373.9	1.34
4,096	256	1309.9	1.28
2,048	512	1277.90	1.25
...	...	...	...
128	8,192	1251.03	1.22

In fact, our testing shows that the array memory footprint keeps falling all the way to the item length 2**15=16,384; but after it reaches 512, the reduction is insignificant. At this item length, memory usage per allocated byte almost matches that of the numeric array (approximately 1.2 vs 1.1). Note that the choice of item lengths as powers of 2 is not

incidental: In our tests, this is where memory usage per byte was optimal, and among those, the lowest values correspond to the multiples of 8.

We might speculate that the underlying reason for these phenomena lies in the way SAS organizes its internal temporary arrays bookkeeping; or that more array items mean more overhead. However, all that matters with our goal in mind is that we can use an array with item length 256 or greater to allocate as many contiguous bits in memory as we shall want at the cost of 1.2 or so byte of RAM per 1 byte of allocated bitmap storage.

However, using an array with item length greater than $1 means that we cannot extract the bitmap byte we need for mapping (or searching) simply as an array reference BM[N_byte]. Instead, now we have two choices:

1. Access the entire array item. Use the CHAR function to get our byte from it, and use the SUBSTR pseudo function to stick it back into the array item after mapping.
2. Come up with some way of extracting the needed byte directly from memory and put it back where it belongs, also directly, after it has been mapped.

Route #1 is seemingly simpler; but it has its shortcomings. First, an extra calculation is needed to determine, based on the item length, which array item to access. Second, character functions (and the SUBSTR pseudo-function in particular) are relatively slow when working against long strings - and per above, our array items have to be as long as $256 or longer to save memory.

Route #2 is perhaps a bit more involved, yet it is definitely much faster and more direct. To take it, we only need to be somewhat familiar with the SAS functions and call routines sometimes called "A-P-P functions" (an acronym for "Address-Peek-Poke") that facilitate direct read and write access to physical memory. In our case, they are just the tool for the job since our bitmap bytes, thanks to being part of a temporary array, have contiguous memory addresses running up relative to the address of the first (leftmost) array byte. So, let us go this route.

HAVING AN A-P-P BYTE

Since we are now aiming at the real-data bitmapping (rather than just a POC), we need to do some preliminary parameterization and other prep work:

- Set the range or our bitmap, R, based on the range of our integer keys.
- Decide on the array item length W. We can just set it to $256 or $512, as increasing it further has little impact on the memory footprint.
- Calculate the number of items in the array dictated by the two factors above.
- Prepare the bit masks.
- Initialize the array items to binary zeroes, one item (W bytes) at a time.
- Get the leftmost (lowest) array address = address of its lower bound item.
- Calculate N_byte and N_bit.

As we have indicated earlier, we can no longer use the assignments for BYTE and BM as we have done before:

```
byte = char (bm, N_byte) ;
substr (bm, N_byte, 1) = byte ;
```

and need to replace them. This is where the A-P-P functionality steps in. So, given the number of byte to extract, N_byte, how do we get it without any array references? Well, note that our bitmap is a contiguous stream of adjacent bytes (and their bits) in memory,

stretching from the leftmost byte of BM[1] all the way to the rightmost byte of BM[&D]. To calculate the physical memory address of its leftmost byte, we can use the A-P-P function ADDRLONG:

```
a1 = addrlong (bm[1]) ;
```

In order to reach the address of byte #N_byte, we need to advance (N_byte) bytes from A1 to the right of the array. Specifically for this purpose, SAS supplies the PTRLONGADD function. Using it in the statement:

```
a = ptrlongadd (a1, N_byte - 1) ;
```

assigns the address of byte #N_byte to A (note that we need to subtract 1 lest we advance 1 byte too far). Now that we have the address of the byte, we can extract the byte itself using another A-P-P function, PEEKCLONG:

```
byte = peekclong (a, 1) ;
```

This expression returns the value of the byte we have sought to extract and is to replace the expression for BYTE above. Now it can be plugged into the RANK function to determine N, compute R, and rewrite BYTE with the value mapped by BOR.

The last bullet on our mapping agenda is to find a substitute for the SUBSTR pseudo function (which we can no longer use) in order to insert BYTE back into the bitmap at the memory address A whence it came. Again, SAS comes to rescue with the A-P-P call routine CALL POKELONG, specially designed to write a value directly into physical memory at a specified address:

```
call pokelong (byte, a, 1) ;
```

At this point, the properly mapped BYTE is back in the bitmap, and so bitmapping of the current key-value K is complete. Now we can loop back, map the next key-value, and keep looping until all key-values are mapped.

Let us now turn to *searching* the compiled bitmap. But in order to do that, we do not have to invent anything new: We have already devised all the tools needed for the job, since in order to search for K, we only have to extract byte #N_byte (and already know how to do that) and check if bit #N_bit is at 0 or 1 using the BAND function. So, we can now proceed directly to complete code:

```
%let R =        1048576 ; /* key range: 2**20, ~1M       */
%let W =            256 ; /* array item character length */
%let D = %eval(&R/&W/8) ; /* calculated array dimension  */

data _null_ ;
  array bm [1:&D] $ &W _temporary_ ;
  *--- Initialize bitmap ---;
  array bitmask [8] _temporary_ (128 64 32 16 8 4 2 1) ;
  bz = put (repeat ("00"x, &W - 1), $&W..) ;  /* BZ = W bytes with binary zeroes  */
  do i = 1 to &D ; bm[i] = bz ; end ;         /* Fill array BM with binary zeroes */
  a1 = addrlong (bm[1]) ;                     /* Get first address of array BM    */
  *--- Mapping ---;
  do K = 1, 200001, 500001, 700001, 1000001 ; /* Let us map these 4 key-values */
    N_byte = ceil (divide (K, 8)) ;           /* Compute N_byte                */
    N_bit = 1 + mod (K + 7, 8) ;              /* Compute N_bit                 */
    a = ptrlongadd (a1, N_byte - 1) ;         /* Address of byte to extract    */
    byte = peekclong (a, 1) ;                 /* Extract the byte              */
    N = rank (byte) ;                         /* Compute its numeric value     */
    R = BOR (N, bitmask[N_bit]) ;             /* Apply BOR to map bit #N_bit   */
```

```
    byte = byte (R) ;                          /* Rewrite BYTE with mapped bit  */
    call pokelong (byte, a, 1) ;               /* Put mapped byte back into BM  */
  end ;
  *--- Search ---;
  do K = 1, 200001, 500001, 700001, 1000001   /* Bitmapped key-values */
       , 2, 200002, 500002, 700002, 1000002 ; /* Unmapped  key-values */
    N_byte = ceil (divide (K, 8)) ;
    N_bit = 1 + mod (K + 7, 8) ;
    a = ptrlongadd (a1, N_byte - 1) ;
    byte = peekclong (a, 1) ;
    N = rank (byte) ;
    found = BAND (N, bitmask[N_bit]) ne 0 ;
    put K= found= ;
  end ;
run ;
```

Running this step prints in the log (the K-values are reformatted to line up):

```
K=      1 found=1
K= 200001 found=1
K= 500001 found=1
K= 700001 found=1
K=1000001 found=1
K=      2 found=0
K= 200002 found=0
K= 500002 found=0
K= 700002 found=0
K=1000002 found=0
```

and confirms that the scheme we have devised works as intended. Note that once again, we have 5 lines of exactly the same code in the mapping and search phases. As before, the repetition can be eliminated by nesting function calls. In fact, the entire block of statements in the mapping loop can be reduced to mere 2 lines of code:

```
a = ptrlongadd(a1,ceil(divide(K,8))-1) ;
call pokelong(byte(BOR(rank(peekclong(a,1)),bitmask[1+mod(k+7,8)])),a,1) ;
```

(Actually, it could be reduced to 1 line but we do not want to calculate A twice.) And the block of statements in the searching loop can be reduced to a single statement:

```
found = ^^ BAND(rank(peekclong(ptrlongadd(a1,ceil(divide(K,8))-1),1)),
               bitmask[1+mod(k+7,8)]) ;
```

since here we do not have to compute A. Such a monstrosity is sure not pretty to stare at, but if you want to make someone looking at your code feel insane (or convince that you are), it can be just the ticket. On the other hand, the nested call fosters somewhat better performance by eliminating a number of assignment statements. Finally, if they are encapsulated (e.g. in a macro), nobody cares what is inside, as long as it works.

LONG-RANGE NUMERIC BITMAPPING

But wait! If you think that the character-based bitmap is great (though somewhat algorithmically confusing), let us turn to the idea - hinted at earlier - of using *numeric* variables to map bits.

The main hindrance of the character bitmap hampering its performance is the necessity to extract the byte within which we need to map the required bit - and, during the process of mapping, to insert it back into the bitmap. Basically, all our efforts to render the character-

based bitmap more agile have been centered on extracting and inserting the needed byte as efficiently as possible.

As we already know, in terms of memory usage per allocated byte, a numeric temporary array, with its item length of 8, is comparable to the most memory-efficient character arrays with the same overall number of bytes. However, in addition to this benefit, it has another unique property making it particularly suitable for bitmapping: Namely, by using a numeric item, we can turn its bits on and/or examine whether any one of its bits is set by performing a rapid *calculation directly on the entire item*.

NUMERIC BIT SETTING AND TESTING

To understand how it can be done, consider a single numeric SAS variable, such as a numeric array item, with 8-bytes, 8 bits apiece. Under ASCII systems, 53 bits out of 64 are allocated to the mantissa, and the remaining 11 - to the exponent; and under the EBCDIC systems, this balance is 56 to 8. Though it is in principle possible to use some exponent bits for our purpose, too, it is excessively tricky and hardly worth the trouble. Hence, let us settle on the mantissa bits only: Thus, we will have 53 (ASCII) or 56 (EBCDIC) usable bits for bitmapping at the cost of 11 to 8 unusable bits depending on the encoding. Accepting this compromise means that, compared to character bitmapping using all 64 bits, our integer key range is reduced (or memory usage increased) by about 17%.

Numeric Bit Setting By Adding A Binary Power

To understand the principle of numeric bitmapping, let us temporarily leave the bits (i.e. binary digits) alone, picture a *decimal* integer D initially set to D=0, and ask: Can we set its digit in position P to 1 using a single arithmetic operation on the *entire* number D? The answer is yes, we can - by simply adding 10**(P-1) to D. For example, let us do it for P=1, 3, 6, 7, 9 and display the results in the log:

```
data _null_ ;
  do P = 1, 3, 6, 7, 9 ;
    D + 10**(P - 1) ;
    put P= D z9. ;
  end ;
run ;
```

The log shows:

```
P=1 000000001
P=3 000000101
P=6 000100101
P=7 001100101
P=9 101100101
```

It should be an easy guess where we are driving with this analogy: If for a *decimal* number - that is, base 10 - we can set the digit in a specific position P by adding 10**(P-1), then by induction we can infer that the same can be done with a *binary* number B by adding 2**(P-1) to it since its base is 2. Let us check it out:

```
data _null_ ;
  do P = 1, 3, 6, 7, 9 ;
    B + 2**(P - 1) ;
    put P= B binary9. ;
  end ;
run ;
```

Expectedly, the log shows:

```
P=1 000000001
P=3 000000101
P=6 000100101
P=7 001100101
P=9 101100101
```

Thus, we can set any of the 53/56 mantissa bits of a numeric variable by merely adding a power of 2 to the variable - which can be justifiably expected to be much faster than extracting the needed byte, setting its bit, and putting it back into a character bitmap.

For demonstrating a concept, computing the binary powers on the fly is perfectly fine. However, when we need to map millions or billions of key-values, the expense of exponentiation repeated so many times can accumulate and become burdensome. So, as we have done with character bitmapping, it makes more sense to pre-compute the powers of 2 beforehand, store them in an array indexed by P, and turn bits on by adding the values of the corresponding array items to B:

```
data _null_ ;
  array bitmask [0:9] _temporary_ (1 2 4 8 16 32 64 128 256 512) ;
  do P = 1, 3, 6, 7, 9 ;
    B + bitmask[P - 1] ;
    put B binary9. ;
  end ;
run ;
```

The result will be exactly the same:

```
000000001
000000101
000100101
001100101
101100101
```

The array holding the binary powers is called "bitmask" for a reason (we will expound on that theme later). And surely, the array does not have to be initialized with hard coded values as done above. Instead, when it is more convenient - for instance, when the binary exponents range from 0 all the way to (M-1) - they can be calculated in the same step in a DO loop without any performance penalty, provided that care is taken to do it only once. Also, there exists a much weighty reason to initialize the array at run time rather than at compile time: As odd as it may sound, an array initialized at compile time takes up much more RAM than the same exact uninitialized array.

Making A Bit Mess - Not!

An alert reader may have already noticed that every time we used this method to set a bit, *that bit was always at* `0`. This is an *absolutely necessary condition* of setting a bit right without messing up the rest of the bitmap. Indeed, if after having already set a bit at position P, we added 2**(P-1) to the bitmap again, it would set (to 1) another bit that we did not intend to set at all! Moreover, it could unset the bit we have already set.

For example: If, after setting the first (P=1) bit of B to 1, we added 2**(P-1)=1 to B again, its binary value would become `000000010`. In other words, by doing this, we would *unset* the bit we had already set at P=1 and *set* the bit at P=2 that we want to remain at 0.

The moral is, before setting any bit by adding a binary power to a numeric variable without an iron-clad guarantee that input key-values have no duplicates, for every bit we are intending to set, we need to check whether it has already been set - and if so, do nothing.

Note that in this sense, this "add binary power" method of bit setting is quite different from that used for a character bitmap: The latter *overwrites* the byte where a bit has just been set with the updated bit pattern, so there is no danger in setting the bit again for a duplicate key-value.

Numeric Bit Setting: BOR-Bit-Mask

Let us recall how we were using the BOR function with corresponding bit masks to set the bits in the needed byte of the character bitmap. Since we had only 1 byte to work with, we used only 8 bits of the numeric variable into which the byte was cast. However, there is no reason why this method should not work for a greater number of bits. How many exactly, we will see later. For the time being, in our numeric bitmapping examples we have had only 9 bits, so let us see if the same approach will work here:

```
data _null_ ;
  array bitmask [0:9] _temporary_ ;
  do P = 0 to 9 ;
    bitmask[P] = 2**P ;
  end ;
  B = 0 ;
  do P = 1, 3, 6, 7, 9 ;
    B = BOR (B, bitmask[P - 1]) ;
    put B binary9. ;
  end ;
run ;
```

The log output shows exactly the same result as from the "add binary power" approach (so there is no reason to duplicate it here). Note that this time, the values of array BITMASK are not preset at compile time but computed at run time. Also note that B has to be initially set to 0 since the BOR function does not accept missing values (in the previous examples, it was of no concern since B was the receptacle of the SUM statement and so set to 0 automatically at compile time).

This BOR-bit-mask method of bit-setting is surely elegant and plenty fast. However, as our testing has shown, the "add binary power" method is so fast that time-wise, it outperforms its competitor by the ratio of 2:3. Furthermore, the BOR-bit-mask method is limited to 32 bits - as we will see later, it is the same as with the BAND-bit-mask method of bit-checking. So, why bring it up at all?

Well, admittedly, there is a degree of geeky pleasure in it on part of the authors. However, much more importantly, the BOR-bit-mask method is impervious to havoc duplicate key-values can wreak when the "add binary power" method is used. Here is a proof:

```
data _null_ ;
  array bitmask [0:9] _temporary_ ;
  do P = 0 to 9 ;
    bitmask[P] = 2**P ;
  end ;
  B = 0 ;
  do P = 1, 1, 5, 5, 9, 9 ;
    B = BOR (B, bitmask[P - 1]) ;
    put B binary9. ;
  end ;
run ;
```

Note that all input key-values have been duplicated; and here is the result:

```
000000001
000000001 <-- Duplicate
000010001
000010001 <-- Duplicate
100010001
100010001 <-- Duplicate
```

While the "add binary power" operation always needs to be preceded by a bit-checking operation to make sure the bit is not yet set, the BOR-bit-mask operation does not need it. So, even though by itself the "add binary power" method is about 50 per cent faster, if we account for the extra bit-checking operation that must precede it, BOR-bit-mask comes on top as 50 per cent faster. It means that if a bitmap is not used as a tool specifically for unduplication purposes and we have enough memory to get away with using only 32 mantissa bits, this method, especially combined with the BAND-bit-mask bit-checking method (described later), can deliver speed comparable to key-indexing at a fraction of memory cost.

Numeric Bit Checking - Inequality Method

The next question is: After the mantissa bits in a numeric item are mapped as shown above, can we use another single arithmetic operation *on the whole item* to discover if its specific bit is on (1) or off (0)? To find out, let us turn to our decimal analogy once again. Suppose that we need to know which digits of our decimal variable D=101100101 (as mapped above) are 0 or 1. For each P=1 to 9, let us calculate MOD(D,10**P) and 10**(P-1), and compare the two for every digit. Note that we are testing the positions from 9 to 1 since the most significant digit is on the left:

```
data _null_ ;
  do P = 1, 3, 6, 7, 9 ;
    D + 10**(P - 1) ;
  end ;
  put D=z9. / "C=" @ ;
  do P = 9 to 1 by -1 ;
    C = mod (D, 10**P) => 10**(P - 1) ;
    put C +(-1) @ ;
  end ;
run ;
```

The step prints in the log:

```
D=101100101
C=101100101
```

It demonstrates that the inequality test:

```
  mod (D, 10**P) => 10**(P - 1)
```

evaluates to true if, and only if, the digit in position P is set to 1. By induction, we can expect than the same is true for a *binary* number - if, of course, base 2 is used instead of 10. Indeed, running the step analogous to the above with base 2:

```
data _null_ ;
  do P = 1, 3, 6, 7, 9 ;
    B + 2 ** (P - 1) ;
  end ;
  put B=binary9. / "C=" @ ;
  do P = 9 to 1 by -1 ;
    C = mod (B, 2**P) => 2**(P - 1) ;
    put C +(-1) @ ;
```

```
  end ;
run ;
```

we get the following log output:

```
B=101100101
C=101100101
```

which confirms that the expression:

```
  mod (B, 2**P) => 2**(P - 1)
```

evaluates to 1 if the bit in position P is set to 1; otherwise, it evaluates to 0.

Note that raising 2 to the needed power does not have to be done every time a bit is mapped or checked. Instead, just as we did with character bitmapping, we can pre-compute the needed powers of 2, store them in an array indexed by P, and thus avoid the relatively expensive exponentiation on the fly. For example, the step above can be rewritten as follows - and generates the same results:

```
data _null_ ;
  array bitmask [0:9] _temporary_ ;
  do P = 0 to 9 ;
    bitmask[P] = 2**P ;
  end ;
  do P = 1, 3, 6, 7, 9 ;
    B + bitmask[P - 1] ;
  end ;
  put B=binary9. / "C=" @ ;
  do P = 9 to 1 by -1 ;
    C = mod (B, bitmask[P]) => bitmask[P - 1] ;
    put C +(-1) @ ;
  end ;
run ;
```

Note that we call the array holding the powers of 2 "bitmask". This is not incidental because its elements, in essence, *are* bit masks, whose bits are all off (i.e. at 0) but for a single position where the bit is set (i.e. at 1). Recalling how we made use of bit masks for checking the bits in a $1 variable (after turning it to a number) immediately suggests that in the case of a numeric variable we could probably resort to the same method as well.

Numeric Bit Checking - BAND-Bit-Mask Method

And indeed we can. The only difference is that now, instead of dealing with 8 bits, we will have to deal with many more, since our intent, as you may recall, is to use as many of the mantissa bits as possible. But let us first verify the principle of using the BAND function for bit checking; only this time, expand the number of bits - for example, from 9 to 13:

```
data _null_ ;
  array bitmask [0:13] _temporary_ ;
  do P = 0 to 13 ;
    bitmask [p] = 2 ** P ;
  end ;
  do P = 1, 3, 6, 7, 9, 11, 13  ;
    B ++ bitmask[P - 1] ;
  end ;
  put B=binary13. / 15 * "-" ;
  do P = 13 to 1 by -1 ;
    C1 = band (B, bitmask[P - 1]) ne 0 ;
```

```
    C2 = mod (B, bitmask[P]) => bitmask[P-1] ;
    put P=z2. C1= C2= ;
  end ;
run ;
```

As a result, we get the following in the log (again, remember that P=1 is on the right):

```
B=1010101100101
---------------
P=13 C1=1 C2=1
P=12 C1=0 C2=0
P=11 C1=1 C2=1
P=10 C1=0 C2=0
P=09 C1=1 C2=1
P=08 C1=0 C2=0
P=07 C1=1 C2=1
P=06 C1=1 C2=1
P=05 C1=0 C2=0
P=04 C1=0 C2=0
P=03 C1=1 C2=1
P=02 C1=0 C2=0
P=01 C1=1 C2=1
```

As we see, the result of bit checking by using the BAND-bit-mask method is the same as with the inequality approach. The former even offers a prettier expression. However, the big question is: How long a bit mask can we apply when using the BAND method?

A simple experiment shows that it is good all the way up to P=32, after which the first argument to the BAND function becomes *invalid*. To think of it, it should come as no surprise, since bit-testing numeric variables using a SAS bit mask literals is also limited to 32 bits - beyond that, SAS truncates the variable being compared to a 32-bit integer.

This 32-bit limit is rather unfortunate since, as our tests have shown, up to this limit the BAND-bit-mask method works about 1.5 times faster than the inequality method. Reducing the numeric bits available to bitmapping from 53/56 to 32 reduces the range of key-values that can be mapped (or, given a key range, increases the required memory footprint) by about 40 per cent. However, in a computing environment where memory is not overly tight, numeric bitmapping using the BAND-bit-mask method can be just the ticket.

Suppose, for example, that we have a 10-digit key (such as a phone number) and would like to map all of its possible key-values and search them using key-indexed search. The corresponding key-indexed array would require 8*10**10≈74G of RAM. However, a 32-bit numeric bitmap would need only 74/32≈2.3G, while a 53-bit bitmap would need 74/53≈1.4G. With memories currently available even on laptops, the difference between 2.3G and 1.4G may be insignificant; but the 50 per cent faster lookup of the 32-bit numeric bitmap using the BAND-bit-mask method can be just what the doctor ordered.

Numeric Bit Unsetting - Subtracting A Binary Power

If some key-value K has been mapped in a bitmap, the corresponding bit is set to 1. It indicates that K is "present". Under a number of realistic scenarios, we may want to erase that "presence" by turning the bit off. This is analogous to deleting a key from the table, through in this case we, of course, neither "insert" nor "delete" K physically: We merely toggle the bit, to which K maps. As with turning a bit of a numeric variable on, we can set a bit off using two methods: (a) via binary arithmetic or (b) via a bitwise function. Let us talk about method (a) first.

Since we know that we can set a bit (initially at 0) by adding a specific power of 2 to the entire numeric item, it is obvious that if a bit is set to 1, we should be able to turn it off by *subtracting* the corresponding binary power from the whole item. Let us check if this is actually true by using our example in the bit-setting section. The step below first sets the bits P=1, 3, 6, 7, 9 to 1 and then unsets them back to 0 in the reverse order:

```
data _null_ ;
  array bitmask [0:9] _temporary_ (1 2 4 8 16 32 64 128 256 512) ;
  do P = 1, 3, 6, 7, 9 ;
    B ++ bitmask[P - 1] ;
    put P= B=binary9. ;
  end ;
  put "--- Deleting --";
  do P = 9, 7, 6, 3, 1 ;
    B +- bitmask[P - 1] ;
    put P= B=binary9. ;
  end ;
run ;
```

Correspondingly, the log shows:

```
P=1 B=000000001
P=3 B=000000101
P=6 B=000100101
P=7 B=001100101
P=9 B=101100101
--- Deleting --
P=9 B=001100101
P=7 B=000100101
P=6 B=000000101
P=3 B=000000001
P=1 B=000000000
```

So, yes, it is just that simple. However, this method contains the same caveat as with bit setting using the "add binary power" method. Recall that when using it, in order to avoid making a bit mess, we need to be dead sure the bit we are about to turn on is initially off - otherwise the bit map will be hopelessly broken. In this case, the same principle applies, only kind of in reverse: Namely, before subtracting the needed power of 2 from B, we need to be certain that the bit we are about to turn off is originally set to 1.

For example, if we attempt to turn the 2nd bit from B=101100**10**1 (where this bit at `0`) by `subtracting 2**1=2,` it will result in B=101100**01**1, i.e. we will actually set bit #2 on and unset bit #3 instead. Hence, if we are supplied a list of key-values to be "erased" from the bitmap, we must always precede the "deletion" operation with bit checking to make certain the bit in question is set to 1.

Numeric Bit Unsetting - BXOR-Bit-Mask

As we already know from the character bitmapping part, we can perform the same operation using the BXOR function coupled with the proper bit mask. Note that in the step below, we are using the BOR-bit-mask method to set the bits, just for the sake of conceptual symmetry:

```
data _null_ ;
  array bitmask [0:9] _temporary_ (1 2 4 8 16 32 64 128 256 512) ;
  B = 0 ;
  do P = 1, 3, 6, 7, 9 ;
```

```
    B = BOR (B, bitmask[P - 1]) ;
  end ;
  put @5 B=binary9. ;
  put "--- Deleting --";
  do P = 9, 7, 6, 3, 1 ;
    B = BXOR (B, bitmask[P - 1]) ;
    put P= B=binary9. ;
  end ;
run ;
```

The SAS log reports:

```
    B=101100101
--- Deleting --
P=9 B=001100101
P=7 B=000100101
P=6 B=000000101
P=3 B=000000001
P=1 B=000000000
```

So, the method works as intended. Not surprisingly, the BXOR-bit-mask method shares common traits with the BOR-bit-mask method. First, it is safe: it cannot make a bit mess and will not break the bitmap, even if the bit to be turned off is already off. Second, it is limited to 32 bits, so it can be used only with numeric bitmaps where the number of usable bits per numeric item is preset to 32.

BITMAPPING INTO A NUMERIC ARRAY

Now that we know how to map numeric bits - and check if they are set, we must allocate a number of numeric variables sufficient to accommodate all needed bits and organize rapid access to them. As with character bitmapping, a natural way to do so is via a temporary array. Since we are dealing with numeric variables, each array item must be 8 bytes in length. However, in this case, it is not a limitation because (a) we do not need items longer than 8 to reduce memory usage and (b) access to every byte is as simple and rapid as a single array reference by its index. But before we can bitmap into a numeric array, we need to determine its dimension and bounds.

Numeric Array Dimension And Bounds

First, we must decide on the number of usable bits per array element. From the discussion above, we know that there are only two sensible choices: (1) use all mantissa bits to save memory or extend the key range; or (2) use only first 32 mantissa bits to gain speed at the expense of memory. Let us assume that we have chosen to use:

```
%let M = 53 ; * 53:ASCII, 56:EBCDIC, 32:More speed with BAND ;
```

Second, to determine the array size, we need to calculate the key range from the lowest and highest possible key-values we are to deal with and, based on the range, figure out whether we have enough memory. Suppose, for example, that we need to record certain events as happened on any second within the years between 1955 and 2015; so that later, given a datetime value as a key, they could be looked up as "occured" or "not occurred". Thus, we would have to let:

```
%let KL = -157766400 ; * lowest K-value ;
%let KH = 1767225600 ; * hihest K-value ;
```

The key range - that is, how many bits we shall need - is the difference plus 1:

```
%let R = %eval (&KH - &KL + 1) ; * Key-value range ;
```

Thus, R=1,924,992,001 is how many bits we need for bitmapping. Having computed R, we can now calculate the number of array items we need to accommodate them, i.e. the array dimension:

```
%let D = %sysfunc (ceil (&R / &M)) ; * Number of array items (dimension) ;
```

The rest of the array setup depends on its base, i.e. the starting index of its lower bound. By default, SAS arrays have base 1. However, in the DATA step they can be used with any integer base, be it negative, zero, or positive (unfortunately, in the FCMP procedure only 1-based arrays are allowed). In this paper, we are working with base 1; however, to avoid hard coding, let us assign it to a parameter:

```
%let LB = 1 ; * bitmap array base, its lower bound ;
```

The array base and its dimension determine its upper bound:

```
%let HB = %eval (&LB + &D - 1) ;
```

Now, to create the array, we can allocate:

```
  array BM [&LB:&HB] _temporary_ ;
```

This way, we have enough room to house all the bits we need. If R is not a multiple of M, we will just have some bits in the upper bound array element left over unused. It is not a deal breaker, as there can be no more than (M-1) of unused bits (i.e. for M=53, no more than 52). For our values of M and R, D=36,320,604. Estimating the memory footprint as 8*DIM bytes, we get about 277 MB. For D as computed above, the step:

```
data _null_ ;
  array bm [&LB:&HB] _temporary_ ;
run ;
```

reports real memory usage of almost exactly 277 MB, which nowadays can be easily handled by almost any hardware. (Note that to key-index, rather than bitmap, this range, we would need 53 times more memory, i.e. about 15 GB. Thinking of it another way, with this kind of memory footprint, a bitmap can accommodate more than 50 times the key range.)

Aligning Array Bits With Key-Values

We already know how to set and check bits within a single numeric variable. Now that we have a number of numeric variables as array items, we need to know how to locate the item to work on. To wit, given a key-value K, we need to find: (1) the correct numeric array item for K, and (2) within the item, the correct bit to be set or checked.

The obvious answer to the question (1) seems just to divide K by M. However, that would work only if the array base LB and the minimal key-value KL were the same. But we are working with a 1-based array, and our minimal key-value KL≠1. Hence, to bitmap properly into a LB-based array and any KL, we have to *shift* K, so that its *shifted* key-values in the range from K=KL to K=(KL+M-1) would map to BM[LB]. This way, K=KL will map to the 1st bit of BM[LB], K=KL+M - to the 1st bit of BM[LB+1], and so on.

Luckily, this is simple arithmetic. First, we need to shift the value of K back by KL places:

```
  KS = K - &KL ;
```

Next, we can compute the index of the 8-byte array element containing the needed bit:

```
  X = int (divide (KS, &M)) + &LB ;
```

The shift up by LB places is required to align the values of X, corresponding to the key-values from K to (K+M-1), with the [1:M] bits of BM[X]. Finally, we can compute the position of the needed bit in the array element with index X:

```
  P = 1 + mod (KS, &M) ;
```

Note that these formulae account for any KL value (negative, zero, or positive) and any array base LB (negative, zero, or positive) we may want to specify. Hence, to align the array bits with our key-values, we only need to work with the *shifted* values KS instead of the original key-values K - and, of course, do so on both the mapping and searching side.

NUMERIC ARRAY BITMAP IN ACTION

Now we are ready to put it all together. Let us create a bitmap to account for all the seconds from 1955 to 2015 and do the following:

- Map approximately each 10th datetime second as an "occurred" event.
- Search the bitmap for every datetime second from KL to KH to find and count the seconds corresponding to the "occurred" events.

Here is bitmapping code using a 53-bit bitmap, replete with parameterization:

```
%let M  = 53 ;
%let KL = -157766400 ;
%let KH = 1767225600 ;
%let R  = %eval (&KH - &KL + 1) ;
%let D  = %sysfunc (ceil (&R / &M)) ;
%let LB = 1 ;
%let HB = %eval (&LB + &D - 1) ;
data _null_ ;
  array BM      [&LB:&HB] _temporary_ ;
  array bitmask [  0: &M] _temporary_ ;
  do x = &LB to &HB ;
    BM[x] = 0 ;
  end ;
  do P = 0 to &M ;
    bitmask[P] = 2**P ;
  end ;
  *--- Mapping ---;
  t = time() ;
  do K = &KL TO &KH by 10 ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    if mod (BM[x], bitmask[P]) < bitmask[P-1] then BM[x] ++ bitmask[P - 1] ;
    N_mapped ++ 1 ;
  end ;
  mtime = time() - t ;
  *--- Search ---;
  t = time() ;
  do K = &KL TO &KH ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    N_found = mod (BM[x], bitmask[P]) => bitmask[P-1] ;
  end ;
  stime = time() - t ;
  put N_Mapped= N_found= / mtime= stime= ;
run ;
```

With M=53, as set above, we had no option but to use the "add binary power" method for bit setting and the inequality method for bit checking. Also, during the mapping phase, it was paramount to check if the bit to be set is already on (note the respective CONTINUE directive). With M=32, we can use the BOR-bit-mask and BAND-bit-mask methods for bit setting and checking, respectively; and it is also unnecessary to check for dupes during the mapping phase:

```
%let M = 32 ;
%let KL = -157766400 ;
%let KH = 1767225600 ;
%let R  = %eval (&KH - &KL + 1) ;
%let D  = %sysfunc (ceil (&R / &M)) ;
%let LB = 1 ;
%let HB = %eval (&LB + &D - 1) ;
data _null_ ;
  array BM      [&LB:&HB] _temporary_ ;
  array bitmask [  0: &M] _temporary_ ;
  do x = &LB to &HB ;
    BM[x] = 0 ;
  end ;
  do P = 0 to &M ;
    bitmask[P] = 2**P ;
  end ;
  t = time() ;
  *--- Mapping ---;
  do K = &KL TO &KH by 10 ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    BM[x] = BOR (BM[x], bitmask[P - 1]) ;
    N_mapped ++ 1 ;
  end ;
  mtime = time() - t ;
  *--- Search ---;
  t = time() ;
  do K = &KL TO &KH ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    N_found ++ BAND (BM[x], bitmask[P - 1]) ne 0 ;
  end ;
  stime = time() - t ;
  put N_Mapped= N_found= / mtime= stime= ;
run ;
```

Running both steps reveals how the two modes stack up in terms of mapping and search times and memory usage (for the sake of comparison, the results for a character bitmap are thrown in, too, though the code is not shown):

Bitmapping Mode	Real Time, seconds		RAM Footprint, MB
	Load	Search	
Numeric 53-Bit	21	169	277
Numeric 32-Bit	15	140	459
Character W=256	45	300	268

One might be inclined to think that these run times seem longish. Note, however, that each program has effectively "loaded" about 192,499,201 key-values in a lookup table, checking every time if it is a duplicate, and then searched the table for 1,924,992,001 distinct key-values - all on a run-of-the-mill laptop (2.4 GHz, 8 GB, running SAS9.4 TS Level 1M4 under X64_7PRO). To put it into perspective, under this hardware no other SAS technique can tackle this task without running out of memory in the first place.

Therefore, it is interesting to compare bitmapping from the standpoint of speed and resource usage with other well-known load-and-search techniques (of which SAS is so rich) of against a task every technique actually can deal with. To do that, we have compared loading 10 million integer 8-digit key-values into a lookup table and searching the latter 100 million times, i.e. for every integer in the 8-digit range, recording the loading (mapping) and search times and memory utilization. The full program is given in the Appendix; here are the results:

Technique	Variation	Real Time, seconds		RAM Footprint, MB
		Load	Search	
Bitmap	Numeric Array 32-Bit	0.7	6.7	24
	Character W=256, A-P-P	2.4	15.3	13
Hash Object	Hashexp:20	2.1	15.4	592
Format	Numeric, $1 Response	39.4	26.0	687

CONCLUSION

Bitmapping is a memory-resident table lookup technique based on the principle of addressing integer key-values directly to bits. Its applicability may appear limited since (a) it is confined to integer keys, (b) their range is restricted by available memory, and (c) it answers only the yes-no question, i.e. whether a given key-value is present or absent.

However, in the niche where the nature of the data and the task fits its nature, bitmapping is peerless from the standpoint of speed and memory usage. For example, its "restricted" integer range can easily map 10 billion integer key-values using just about 1.25 GB of RAM, and the speed of searching for any key-value in the range does not depend on how many keys are "inserted" (actually, mapped) into it, as bitmapping runs in *O*(1) time. It can be organized using both character and numeric variables to string its bits in memory, which lets the user balance its speed and memory usage.

REFERENCES

Dorfman, P. 2000. "Table Look-Up by Direct Addressing: Key-Indexing -- Bitmapping -- Hashing." *Proceedings of the SUGI 2000 Conference*, Long Beach, CA: SAS Institute, Inc. Available at https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p008-26.pdf

Dorfman, P. 1999. "Array Lookup Techniques: From Sequential Search To Key-Indexing." Proceedings of the SESUG 1999 Conference, Mobile, AL: SAS Institute, Inc. Available at:

https://analytics.ncsu.edu/sesug/1999/016.pdf

Dorfman, P. and Henderson, D. 2018. *Data Management Solutions Using SAS Hash Table Operations. A Business Intelligence Study*. Cary, NC: SAS Institue, Inc.

ACKNOWLEDGMENTS

The authors would like to thank Pete Lund for inviting them to present this paper.

CONTACT INFORMATION

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

Paul M. Dorfman
sashole@gmail.com

APPENDIX

1. COMPARISON: BITMAPPING, HASH OBJECT, AND FORMAT LOAD-AND-SEARCH

```
*--- Character Bitmapping ---;
%let R =      100000000 ;
%let W =            256 ;
%let D = %eval(&R/&W/8) ;
data _null_ ;
  array bm [1:&D] $ &W _temporary_ ;
  *--- Initialize bitmap ---;
  array bitmask [8] _temporary_ (128 64 32 16 8 4 2 1) ;
  bz = put (repeat ("00"x, &W - 1), $&W..) ;
  do i = 1 to &D ; bm[i] = bz ; end ;
  a1 = addrlong (bm[1]) ;
  *--- Mapping ---;
  t = time() ;
  do K = 1 to &R by 10 ;
    a = ptrlongadd(a1,ceil(divide(K,8))-1) ;
    call pokelong(byte(BOR(rank(peekclong(a,1)),bitmask[1+mod(k+7,8)])),a,1)
  end ;
  mtime = time() - t ;
  *--- Search ---;
  t = time() ;
  do K = 1 to &R ;
    N_found ++ BAND(rank(peekclong(ptrlongadd(a1,ceil(divide(K,8))-1),1)),
                bitmask[1+mod(k+7,8)]) ne 0 ;
  end ;
  stime = time() - t ;
  put N_found = / mtime= stime= ;
run ;
*--- 53-bit Numeric Array Bitmapping ---;
%let M = 32 ;
%let KL =         1 ;
%let KH = 100000000 ;
%let R  = %eval (&KH - &KL + 1) ;
%let D  = %sysfunc (ceil (&R / &M)) ;
%let LB = 1 ;
%let HB = %eval (&LB + &D - 1) ;
data _null_ ;
  array BM      [&LB:&HB] _temporary_ ;
  array bitmask [  0: &M] _temporary_ ;
  do x = &LB to &HB ;
    BM[x] = 0 ;
  end ;
```

```
  do P = 0 to &M ;
    bitmask[P] = 2**P ;
  end ;
  *--- Mapping ---;
  t = time() ;
  do K = &KL TO &KH by 10 ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    if mod (BM[x], bitmask[P]) < bitmask[P-1] then BM[x] ++ bitmask[P - 1] ;
    N_mapped ++ 1 ;
  end ;
  mtime = time() - t ;
  *--- Search ---;
  t = time() ;
  do K = &KL TO &KH ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    N_found = mod (BM[x], bitmask[P]) => bitmask[P-1] ;
  end ;
  stime = time() - t ;
  put N_Mapped= N_found= / mtime= stime= ;
run ;
*--- 32-bit Numeric Array Bitmapping ---;
%let M = 32 ;
%let KL =         1 ;
%let KH = 100000000 ;
%let R  = %eval (&KH - &KL + 1) ;
%let D  = %sysfunc (ceil (&R / &M)) ;
%let LB = 1 ;
%let HB = %eval (&LB + &D - 1) ;
data _null_ ;
  array BM      [&LB:&HB] _temporary_ ;
  array bitmask [  0: &M] _temporary_ ;
  do x = &LB to &HB ;
    BM[x] = 0 ;
  end ;
  do P = 0 to &M ;
    bitmask[P] = 2**P ;
  end ;
  t = time() ;
  *--- Mapping ---;
  do K = 1 to &R by 10 ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    BM[x] = BOR (BM[x], bitmask[P - 1]) ;
    N_mapped ++ 1 ;
  end ;
  mtime = time() - t ;
  *--- Search ---;
  t = time() ;
  do K = &KL TO &KH ;
    x = int (divide (K - &KL, &M)) + &LB  ;
    P = 1 + mod (K - &KL, &M) ;
    N_found ++ BAND (BM[x], bitmask[P - 1]) ne 0 ;
  end ;
```

```
  stime = time() - t ;
  put N_Mapped= N_found= / mtime= stime= ;
run ;
*--- Hash Object Load and Search ---;
data _null_ ;
  dcl hash H (hashexp:20) ;
  h.defineKey ("K") ;
  h.defineDone () ;
  *--- Load ---;
  t = time() ;
  do K = 1 to &R by 10 ;
    h.add() ;
  end ;
  mtime = time() - t ;
  t = time() ;
  *--- Search ---;
  do K = 1 to &R ;
    N_found ++ h.check() = 0 ;
  end ;
  stime = time() - t ;
  put N_found = / mtime= stime= ;
run ;
*--- Format load ---;
data vcntlin / view = vcntlin ;
  retain fmtname "lkup" label "1" hlo "" ;
  do start = 1 to &R by 10 ;
    output ;
  end ;
  label = "0" ;
  hlo = "o" ;
  output ;
run ;
proc format cntlin = vcntlin ;
run ;
*--- Format search ---;
data _null_ ;
  do K = 1 to 1e8 ;
    N_found ++ put (K, lkup.) = "1" ;
  end ;
  put N_found= ;
run ;
```

2. RANDOM SELECTION COMPARISON: HASH OBJECT, BITMAPPING, KEY-INDEXING

```
%let N =  20e6 ; *  20 million ;
%let R = 100e6 ; * 100 million ;

*** SAS Hash Object *** ;

data sample (keep = K) ;
  call streaminit (7) ;
  dcl hash h () ;
  h.defineKey ("K") ;
  h.defineDone () ;
  do until (Q = &N) ;
```

```
    K = rand ("integer", &R) ;
    if h.check() = 0 then continue ;
    h.add() ;
    output ;
    Q + 1 ;
  end ;
run ;

*** 32-bit Numeric Array Bitmap *** ;

%let M = 32 ;
%let D  = %sysfunc (ceil (&R / &M)) ;
data sample (keep = K) ;
  call streaminit (7) ;
  array BM      [1:&D] _temporary_ ;
  array bitmask [0:&M] _temporary_ ;
  do P = 0 to &M ;
    bitmask[P] = 2 ** P ;
  end ;
  do until (Q = &N) ;
    K = rand ("integer", &R) ;
    x = int (divide (K - 1, &M)) + 1 ;
    BM[x] = BM[x] max 0 ;
    P = mod (K - 1, &M) ;
    if BAND (BM[x], bitmask[P]) then continue ;
    BM[x] = BOR (BM[x], bitmask[P]) ;
    output ;
    Q + 1 ;
  end ;
run ;

*** FCMP-Encapsulated 32-bit Numeric Array Bitmap

proc fcmp outlib=work.f.f ;
  function bm32find (BM[*], K) ;
    outargs BM ;
    array bitmask [32] / nosym
   (0000000001 0000000002 0000000004 0000000008
    0000000016 0000000032 0000000064 0000000128
    0000000256 0000000512 0000001024 0000002048
    0000004096 0000008192 0000016384 0000032768
    0000065536 0000131072 0000262144 0000524288
    0001048576 0002097152 0004194304 0008388608
    0016777216 0033554432 0067108864 0134217728
    0268435456 0536870912 1073741824 2147483648) ;
    return (BAND (max (BM [int (divide (K - 1, 32)) + 1], 0), bitmask [1 + mod (K -
1, 32)]) > 0) ;
  endsub ;
  subroutine bm32add (BM[*], K) ;
    outargs BM ;
    array bitmask [32] / nosym
   (0000000001 0000000002 0000000004 0000000008
    0000000016 0000000032 0000000064 0000000128
    0000000256 0000000512 0000001024 0000002048
    0000004096 0000008192 0000016384 0000032768
```

```
    0000065536 0000131072 0000262144 0000524288
    0001048576 0002097152 0004194304 0008388608
    0016777216 0033554432 0067108864 0134217728
    0268435456 0536870912 1073741824 2147483648) ;
    x = int (divide (K - 1, 32)) + 1 ;
    BM[x] = BOR (max (BM[x], 0), bitmask[1 + mod (K - 1, 32)]) ;
  endsub ;
quit ;

option cmplib=work.f ;

%let M = 32 ;
%let D  = %sysfunc (ceil (&R / &M)) ;
data sample (keep = K) ;
  call streaminit (7) ;
  array BM [1:&D] _temporary_ ;
  do until (Q = &N) ;
    K = rand ("integer", &R) ;
    if bm32find (BM, K) then continue ;
    call bm32add (BM, K) ;
    output ;
    Q + 1 ;
  end ;
run ;

*** Key-Indexing *** ;

%let D = %sysfunc (int (&R)) ;
data sample (keep = K) ;
  call streaminit (7) ;
  array KX [1:&D] _temporary_ ;
  do until (Q = &N) ;
    K = rand ("integer", &R) ;
    if KX[K] then continue ;
    KX[K] = 1 ;
    output ;
    Q + 1 ;
  end ;
run ;

*** 256-Byte Array Character Bitmap *** ;

%let D = %sysfunc (ceil (&R / 256 / 8)) ;
data sample (keep = K) ;
  call streaminit (7) ;
  array BM [1:&D] $256  _temporary_ ;
  array bitmask [8] _temporary_ (128 64 32 16 8 4 2 1) ;
  b00x = put (repeat ("00"x, 255), $256.) ;
  do i = 1 to &D ;
    bm[i] = b00x ;
  end ;
  a1 = addrlong (BM[1]) ;
  do until (Q = &N) ;
    K = rand ("integer", &R) ;
    a = ptrlongadd (a1, ceil (divide (K, 8)) - 1) ;
```

```
    N_bit = 1 + mod (K + 7, 8) ;
    N = rank (peekclong (a, 1)) ;
    if BAND (N, bitmask[N_bit]) then continue  ;
    call pokelong (byte (BOR (N, bitmask[N_bit])), a, 1) ;
    output ;
    Q + 1 ;
  end ;
run ;
```

REFERENCES

Dorfman, P. 2000. "Table Look-Up by Direct Addressing: Key-Indexing -- Bitmapping -- Hashing." *Proceedings of the SUGI 2000 Conference*, Long Beach, CA: SAS Institute, Inc. Available at https://support.sas.com/resources/papers/proceedings/proceedings/sugi26/p008-26.pdf

Dorfman, P. 1999. "Array Lookup Techniques: From Sequential Search To Key-Indexing." Proceedings of the SESUG 1999 Conference, Mobile, AL: SAS Institute, Inc. Available at:

https://analytics.ncsu.edu/sesug/1999/016.pdf

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

ACKNOWLEDGMENTS

The authors would like to thank Peter Lund for inviting them to present this paper.

CONTACT INFORMATION

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

Paul M. Dorfman	Lessia S. Shajenko
sashole@gmail.com	lessia.s.shajenko@bankofamerica.com