SESUG Paper 276-2018

Efficient DATA Step Random Sampling Out Of Thin Air

Paul M. Dorfman, Independent Consultant
Lessia S. Shajenko, Senior Quantitative Analyst, Bank of America

ABSTRACT

Many ETL and data analysis tasks require to generate a random sample of unique K out of N available integers such that K<<N. While it can be done using the SURVEYSELECT procedure, it needs an input data set with N observations. Unfortunately, when N is large enough, it is not a practical option. For example, at N=1E14, merely looping from 1 to N would take close to six days. However, the required sample can be created in a DATA step "out of thin air" using algorithms that require looping only from 1 to K regardless of how large N is. Moreover, their bookkeeping memory storage can be kept strictly bounded by K. In this paper, we will discuss a number of such algorithms and their DATA step implementations based both on arrays and the SAS® hash object.

INTRODUCTION

Suppose that we want to generate K random integers in the range of [1:N] without replacement. An easy way is to create a data set with N consecutive integers and feed it into the SURVEYSELECT procedure. However, if N is very large - such as N=1E15, for example - it is practically impossible to create an input data set with so many records. Moreover, if SAS/STAT is not licensed, the procedure is not available. In this paper, we discuss how it can nevertheless be done in a single DATA step without any initial input, i.e. "*out of thin air*", and compare different approaches from the standpoint of their viability and efficiency depending on K, N, and the K/N ratio.

SCENARIOS

Let us assume that the value of K is reasonably limited, so that an output with K records can be created. This is a fairly relaxed constraint since a data set with a single numeric variable can be easily created with K as large as, say, 1E9 (1 billion) even on modest hardware. For the purposes of this discussion and to keep test run times in check, we will keep K under K=1E8 (100 million), dealing mostly with K up to 2E7 (20 million).

With K thus constrained, the choice of algorithm is dictated by the range [1:N] of the pool from which the random integers are selected. There are several possibilities:

1. N is not too large to allocate a numeric temporary array with N items. As an estimate, 1 GB of RAM can accommodate about 2**30/8 ~ 134 million such array items.
2. N is not too large to loop from 1 to N in a reasonable time. A DO loop scans from 1 to N=1E9 (1 billion) in a few seconds, so it can scan as far as to N=1E11 (100 billion) in a few minutes. Obviously, if N is not too large for case #1, it is an easy fit for this case as well. Conversely, the values of N near the upper limit for this case are too large for case #1.
3. N is too large even to loop from 1 to N in a reasonable time. For example, if we are to pick K random integers from the range [1:1E15] (i.e. 1 quadrillion), looping through this range on typical hardware would take about 30 days. Needless to say, such values of N negate the possibility of using algorithms good for either case above, so some other approach is needed.

In the course of the paper, we will see how different tools, such as arrays, full scan, different kinds of hash tables, and bitmaps can be chosen depending on the particular random selection algorithm with an eye on maximizing efficiency and keeping memory usage in check.

RAND FUNCTION PRELIMINARIES

There are a great variety of SAS functions capable of generating random numbers. In this paper, the RAND function is used. Its advantages over other random functions, such as RANUNI, have been amply described (Wickling, 2013).

If the RAND function is called with the first argument "UNIFORM", it generates random numbers from a uniform distribution in the range of [0:1]. In order to turn them into integers in a specific range [MIN:MAX], a little bit of extra arithmetic is needed:

```
R = MIN + floor (rand ("uniform") * (1 + MAX - MIN)) ;
```

The range from which this expression can select a random integer is defined by the range of *consecutive* integers that can be stored with full precision in a standard SAS numeric variable. This range depends on the encoding used under different computing platforms:

Encoding	Binary		Decimal	
	MIN	MAX	MIN	MAX
ASCII	- 2 ** 53	+ 2 ** 53	- 9,007,199,254,740,992	+ 9,007,199,254,740,992
EBCDIC	- 2 ** 56	+ 2 ** 56	- 72,057,594,037,927,936	+ 72,057,594,037,927,936

Table 1. RAND ("uniform") Integer Selection Limits

In other words, on the ASCII platforms (used to test the programs presented here) the maximum positive range from which a random integer is from 1 to approximately 9E15.

Since the advent of SAS 9.4 TS Level 1M4, the first argument of the RAND function can also accept the value of "INTEGER" and two additional arguments indicating the lower and upper integer bounds for the selection. These arguments make it easy to generate a random integer from the uniform distribution exactly between two integers MIN and MAX without any extra arithmetic:

```
R = rand ("integer", MIN, MAX) ;
```

If MIN=1, the second argument can be omitted, in which case MIN=1 is assumed by default, and so the above can be coded simply as:

```
R = rand ("integer", MAX) ;
```

However, it should be kept in mind that for all the convenience of this form of the RAND function, it limits the integer selection range on *any* computing platform to the following:

Power of 2		Decimal	
MIN	MAX	MIN	MAX
- 2 ** 32 + 1	+ 2 ** 32 - 1	- 2,147,483,647	+ 2,147,483,647

Table 2. RAND ("integer") Integer Selection Limits

Therefore, if random integers are to be selected from the range [1:N] where N > 2,147,483,647, the RAND function should be used with the first argument "UNIFORM" and the scaling formula shown above.

NAIVE SELECTION

Naive selection means that we repeatedly pick a random integer from the range [1:N], reject it if it has already been selected before, and proceed in this manner until the number of selected items is equal to K. In more formal terms, the algorithm can be described as follows:

1. Select a random integer from range [1:N] and store it in some kind of lookup table.
2. Select a random integer from the same range [1:N] again.
3. If the selected integer is in the lookup table, reject it.
4. Otherwise, add it to the output and to the lookup table.
5. If the output count is still less than K, go to step 2; otherwise, stop.

With this method, every time we pick an item from [1:N] we do it "with replacement" since the item is not taken out of the selection pool. The sampling *without* replacement part comes here from rejecting duplicate picks. Let us see how this method can be implemented in DATA step code by sampling K=5 integers out of N=9 and choosing the lookup table in the form of the *key-indexed array*:

```
%let K = 6 ;
%let N = 9 ;

data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;                 /* Set a seed for the RAND function      */
  array t [1:&N] _temporary_ ;          /* Key-indexed array T as lookup table   */
    do HIT = 1 by 1 until (Q = K) ;     /* Loop. HIT = number of attempted picks */
    R = rand ("integer", 1, N) ;        /* Pick random R in [1:N] range          */
  * R = ceil (rand ("uniform") * N) ;   /* Use for releases before SAS9.4 1M4    */
    if t[R] then continue ;             /* If R in table T, DO iterates again    */
    t[R] = 1 ;                          /* Mark R in table T as already selected */
    output ;                            /* Output R as not already selected      */
    Q + 1 ;                             /* Increment output count Q              */
  end ;                                 /* DO loop stops here when Q = K         */
  put HIT= ;                            /* Print number of attempted picks       */
run ;
```

The following table shows the pick-reject process for each attempted pick:

HIT	R PICKED	R SELECTED	T1	T2	T3	T4	T5	T6	T7	T8	T9	OUTPUT
1	**8**	8	.	.	.	.	.	.	.	.	.	Yes
2	**6**	6	.	.	.	.	.	.	.	1	.	Yes
3	**2**	2	.	.	.	.	.	1	.	1	.	Yes
4	**3**	3	.	1	.	.	.	1	.	1	.	Yes
5	**3**	Dup	.	1	**1**	.	.	1	.	1	.	No
6	**9**	9	.	1	1	.	.	1	.	1	.	Yes
7	**9**	Dup	.	1	1	.	.	1	.	1	**1**	No
8	**6**	Dup	.	1	1	.	.	**1**	.	1	1	No
9	**2**	Dup	.	**1**	1	.	.	1	.	1	1	No
10	**5**	5	.	1	1	.	.	1	.	1	1	Yes

Table 3. Naive Random Selection Step-by-Step

At the end of the selection process, the variable HIT contains the number of attempts needed to select 6 unique random integers. In the case above, HIT=10, which means that the range of [1:9] has had to be probed 10 times before reaching the needed sample size K=6. It is obvious that the nearer K is to N, the higher is the probability of duplicate picks and the more attempts to complete the process are needed. In the upper limit case of K=N=9, the program prints HIT=21, meaning that it spends most of its time rejecting the duplicate picks.

The main shortcoming of the naive approach is having to reject many duplicates as K nears N. On the other hand, in most sampling scenarios K << N, so the naive scheme offers a number of advantages:

1. It is very simple.
2. It does not have to scan through the entire [1:N] range.
3. Though it may call the RAND function HIT > K times, we will see later that if the lookup table is chosen appropriately, it remains surprisingly efficient even at K as high as ~0.8 *N.

LOOKUP TABLE CHOICE

The efficiency of rejecting duplicate picks depends on the lookup table used to store the selected integers and search them to see if the next pick is already there. It should satisfy two requirements:

1. It should be fast in terms of inserting integers and searching for them - preferably, both should occur in *O(1)* time. In other words, the time needed to insert a key (or search for it) should be the same regardless of the number of items stored in the table.
2. It should fit in memory with a good margin for other processes to spare.

To satisfy the *O(1)* criterion, a lookup table must be based on the principle of *direct addressing*. In SAS, there are 3 types of such lookup tables:

1. Key-indexed lookup.
2. Bitmap lookup.
3. Hash table lookup.

Let us look at them separately.

1. Key-Indexed Lookup

The array T shown in the program above represents a *key-indexed* table. It allocates a temporary array with all possible items indexed from 1 to N. Initially, all of them are missing. If integer R is picked, the R-th array item is marked as T[R]=1. Thus, if the same integer R is picked again, we immediately know it because T[R]=1; otherwise, T[R] would be missing.

Obviously, when we deal with integer keys, there is no lookup scheme simpler and/or faster than key-indexing. But it is also obvious that these advantages come at a price: We need to pay for memory usage. Indeed, to be able to store any integer from 1 to N in the table, we need to allocate N array items, 8 bytes each. This is of no concern for N=9, as in the example above. Even at N=1E8, the array memory footprint would be around 800 MB of RAM, which nowadays can be handled by practically any computing device. However, as N grows, the memory usage toll gets more burdensome, and the more questionable becomes the "wisdom" of using full 8 numeric bytes to mark a single integer in the table.

An efficiency-minded programmer could logically ask at this point: Why not use a character $1 temporary array instead, with practically no change to the code? *Seemingly*, it would allow to cut the memory footprint 8 times since now we would be using only 1 byte per each digit in the [1:N] range. Right? The answer is that unfortunately, for each item of a $1 temporary array, SAS allocates 16 [sic!] bytes of memory; and so not only it would not be saving any memory but it would actually double its usage.

2. Bitmap Lookup

However, it is also obvious that for the task at hand, the information about the random integers we need to store is confined to "picked" or "not picked". In other words, this is binary type data, and thus it can be

stored in a *single bit* whose values 1 and 0 can represented the two respective states. Therefore, we can proceed as follows:

1. Allocate N *bits* in memory, initially all set to 0. In other words, create an *empty bitmap*.
2. Pick a random integer R and look into the R-th bit in the bitmap.
3. If the R-th bit is at the state of 0, the integer has not yet been picked, so select R for the output and set the R-th bit to 1 (also expressed as "turn it on").
4. Otherwise if the R-th bit is turned on, R has already been selected, so reject it.
5. Repeat the process from Step #1 until Q=K.

As we see, the process is principally identical to the key-indexed approach. The difference is that now, to mark a pick as selected, we are using *1 bit* instead of *8 bytes* (i.e. instead of *64 bits*) and therefore can expect the corresponding reduction in memory usage. The remaining, purely technical, questions are:

- What kind of data structure to use for bitmap allocation?
- How to turn a specific bit on?
- How to find out if a specific bit is on or off?

Luckily, the answers to these questions have already been given (Dorfman, 2001), so we will only present executable code. Below, we are using K and N much larger than before to see how this program scales up with respect to the run time and memory usage compared to the other approaches in this section.

```
%let K = 2E7 ; *  20 million ;
%let N = 1E8 ; * 100 million ;
%let B =  53 ; * Number of usable ASCII numeric bits (56 for EBCDIC) ;
%let H = %sysfunc (floor (&N / &B)) ; * Compute array size ;

data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;
  array bmp [0:&H] _temporary_ (0, &H * 0) ; /* initialize bitmap     */
  array bin [0:&B] _temporary_;              /* pre-store powers of 2 */
  do _x = 0 to &B ;
    bin[_x] = 2 ** _x ;
  end ;
  do hit = 1 by 1 until (Q = K) ;
    R = rand ("integer", N) ;
    _x = int (divide (R - 1, &B)) ;
    _r = R - 1 - _x * &B ;
    if mod (bmp[_x], bin[_r+1]) => bin[_r]    /* R is not in bitmap */
      then continue ;
    bmp[_x] + bin[_r] ;                       /* Turn R-th bit on   */
    Q + 1 ;
    output ;
  end ;
  put hit= ;
run ;
```

Running this step shows that the number of attempted picks HIT=22,320,136. In other words, we needed extra 2,320,136 random picks before arriving at the needed sample size. Though it may seem like gobs of extra work, the number of extra picks is only 12 percent of K, and (on a garden-variety Windows laptop) the step takes under 7 seconds to run, using 29 MB of RAM in the process. By comparison, the key-indexed step run with the same (K,N) just less than 1 second faster - but requires 763 MB of RAM to run. The huge difference in memory usage should come at no surprise: If we examine the value of the macro variable H, it turns out that the bitmap BMP needs 1,886,792 array items as opposed to 100 million items for the key-indexed array.

The advantage of the bitmap does not end here because its range can be extended further. For example, N as high as 1E10 (10 billion) can be accommodated at the expense of less than 3 GB of RAM. Also, for

a given value of N, the bitmap memory footprint can be cut almost by half by basing the bitmap on character arrays and using all the 64 bits of every byte. But this is a subject for another paper.

3. Hash Table Lookup

The glaring shortcoming of both the key-indexed and bitmap methods is rooted in the fact that the size of the lookup table is defined by the full range [1:N]. Indeed, it is apparent from the nature of the algorithm that there will never be more than K items in the lookup table marked as already selected; and if so, why do we need a table with N elements, be they bytes or bits? The question is even more relevant if we consider that the SAS hash object also offers nearly *O(1)* insert and search behavior. And since with the task at hand it will never have to store more than K items, using it to weed out duplicate picks appears to be a natural choice. Without further ado, let us see then how this can be done using the hash object:

```
%let K = 2E7 ; *  20 million ;
%let N = 1E8 ; * 100 million ;

data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;
  dcl hash T () ;                    /* Create an instance of hash object T */
  t.defineKey ("R") ;                /* Define R as its key                 */
  t.defineDone () ;
  do HIT = 1 by 1 until (Q = K) ;
    R = rand ("integer", N) ;        /* Use with SAS9.4 1M4 and later       */
  * R = ceil (rand ("uniform") * N) ; /* Use with any SAS9.4 release        */
    if t.check() = 0 then continue ; /* If key R is in table T, loop again  */
    t.add() ;                        /* Else insert R into table T          */
    output ;
    Q + 1 ;
  end ;
  put hit= ;
run ;
```

Comparing the actual performance metrics of this run to those for the key-indexing and bitmapping may at first leave one dumbfounded. Indeed, the step finishes the job in 39 seconds and ends up using 960 MB of RAM - far slower than either key-indexing and bitmapping and exceeding both in memory usage. So, *what gives*?

Well, as far as lookup speed is concerned, nothing can realistically compete with the ferocious agility of a key-indexed table or bitmap in their narrow specialized niche of searching for limited-range integer keys.

But what of the memory footprint? We had hoped that since the hash table needs to store only K items, it would surely use a lot less memory to accommodate 20 million of them than it takes for the key-indexed table to have enough room for 100 million keys, would it not? Unfortunately, the answer is "no", and there are two reasons for it:

1. A hash object instance cannot store just the lookup key in its key portion and nothing else. Its internal structure requires it to have at least one hash variable in its data portion. If no data portion hash variable is defined explicitly (as it is done above since there is no DEFINEDATA method call), the same hash variable - in this case, R - defined for the key portion is automatically defined for the data portion as well. Thus, in actuality we have to store two hash variables, 8 bytes apiece.
2. On the 64-bit ASCII platforms, the minimal hash table entry size - i.e. the total number of memory bytes occupied by a single hash item - is 48 bytes, even if the table has only two $1 hash variables; and it stays at 48 bytes until the total length of all hash variables exceeds 16. So, with one numeric key portion variable R (length 8) and one numeric data portion variable R, each hash item inserted into the table occupies 48 bytes in memory. So, 20 million of them take up 960 MB.

These numbers cannot help but raise two questions:

1. Does using a hash table to track duplicate random picks make sense?
2. If it does, why and under which circumstances?

The answer to the first question is: *Yes, it does*. The answer to the second question (which also explains the answer to the first) is that for a given value of K, hash memory usage, though considerable, will stay unchanged regardless of the value of N, all the way up to N=2**53 (~9E15). In other words, the program above would run within the same resource limits (and actually even a tad faster thanks to hitting on fewer duplicate picks) even for N=9E15. On the other hand, allocating a bitmap array to hold 9E15 bits (i.e. allocating ~1000 TB of RAM) is sheer fantasy.

The takeaway from this section is that a decision on using a particular kind of lookup table for the naive selection approach should be judiciously tailored to the values of K and N and available memory resources. Roughly speaking, it can be recommended that:

- At approximately N ≤ 5E7 (50 million), a key-indexed table trumps everything else, given its utter simplicity and speed.
- At approximately 5E7 ≤ N ≤ 5E9, a bitmap table stays within reasonable memory limits and is almost as fast as a key-indexed table.
- Beyond that, a hash table should be used, provided that K/46 bytes can fit comfortably in memory.

It also matters what *kind of hash table* is being used. Above, we have taken the easy route of using the hash object whose built-in functionality makes coding a breeze. As we will see later on, the efficiency of the naive approach can be increased quite a bit by using an array-based hash table.

DYNAMIC POOL SELECTION

The most eminent drawback of the naive selection method is having to employ one means or another to reject duplicate random picks. Naturally, it raises the question of whether we can construct an algorithm that would avoid picking duplicate items in the first place. Fortunately, algorithms of this kind have been indeed devised. They are based on discarding the integer just selected from the selection pool before the next selection is made.

To illustrate the concept, let us return to the very first example with N=9 and K=6 and the same array T with N elements used there. This time, however, let us change the procedure and proceed as follows:

1. Populate array T with consecutive integers from 1 to N (in this, case, from 1 to 9).
2. Set selection pool size S=N, and set selection counter Q=1.
3. If Q > K, stop.
4. Pick a random array index X in range [1:S], set R=T[X], and output R.
5. Replace item T[X] with the end-of-range item T[S]: T[X]=T[S]. Optionally, replace T[S]=R.
6. Decrement S by 1.
7. Increment Q by 1 and repeat from Step #3.

This algorithm represents selection without replacement *literally*, as no item selected for the sample is ever returned to the selection pool: The pool size is adjusted down by 1 with every new selection. It can be proven mathematically that even though every new item is picked from the reduced range, each of the N items is equally likely to be randomly selected. Translating this algorithm, verbatim, into the DATA step language, we have:

```
%let K = 6 ;
%let N = 9 ;

data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (5) ;
  array T [1:&N] _temporary_ ;          /* Allocate key-indexed array T[1:N]    */
  do Q = 1 to N ;                       /* Populate T with integers from 1 to N */
    T[Q] = Q ;
  end ;
  S = N ;                               /* Initially, set pool size S to N      */
  do Q = 1 to K ;                       /* Increment Q by 1. If Q > K, stop     */
```

```
    X = ceil (rand ("uniform") * S) ; /* Pick random index X in [1:S] range   */
  * X = rand ("integer", S) ;         /* Alternative for SAS9.4 1M4 or later  */
    R = T[X] ;                        /* Select item with index X for output  */
    output ;                          /* Add the record to file Sample        */
    T[X] = T[S] ;                     /* Replace T[X] with end-of-range T[S]  */
    S +- 1 ;                          /* Reduce pool size by 1 for next pick  */
  end ;
run ;
```

The process rendered by this program can be visualized for each of the K=6 selection steps in the following table. The pool size available at each step is indicated by the shaded area.

Example: K=6, N=9												
Pick #	Pool Size	Index Picked	Item Sampled	State of Key-Indexed Table T[1:9]								
Q	S	X	R	T1	T2	T3	T4	T5	T6	T7	T8	T9
1	9	2	2	1	**2**	3	4	5	6	7	8	9
2	8	1	1	**1**	9	3	4	5	6	7	8	2
3	7	7	7	8	9	3	4	5	6	**7**	1	2
4	6	6	6	8	9	3	4	5	**6**	7	1	2
5	5	1	8	**8**	9	3	4	5	6	7	1	2
6	4	4	4	5	9	3	**4**	8	6	7	1	2

Table 4. Dynamic Pool Selection Step-by-Step

The main advantage of the method is not having to reject duplicate picks at all. For the same reason, the scheme, as opposed to the naive selection methods, is equally good at any K/N ratio all the way up to K=N. For example, running the key-indexed naive and dynamic pool selection methods side by side for N=1E8 and a number of different K/N ratios results in the picture shown in the table below:

Key-Indexed Selection	Real Run Time for N = 1E8, seconds			
	K = 2E7	K = 4E7	K = 6E7	K = 8E7
Naive	3.6	8.0	14.5	24.4
Dynamic Pool	4.6	7.6	10.6	12.4

Table 5. Naive vs Dynamic Pool Key-Indexed Selection

Note that both methods use the same memory footprint (~760 MB) since both are based on the key-indexed array T of the same size. As we see, the closer K comes to N, the more time the naive scheme spends rejecting duplicate picks and the more its run time performance deteriorates relative to the dynamic pool selection method.

Of course, the dynamic pool selection technique as shown is not without shortcomings itself:

- Since it is based on a numeric key-indexed array of size N, it requires 8*N bytes of memory. It means that if you have M bytes of RAM available for the task, N is limited to approximately M/8.
- Furthermore, in this case the array cannot be replaced with a bitmap since it has to store non-binary data.

- In order to initialize the array, the entire range of integers from 1 to N be scanned. It is not a really big drawback in this case, as N is limited by memory anyway. But the scanning still takes time, and this is why at low K/N ratios the method runs a tad slower than naive key-indexing or bitmapping.

The impact of the dynamic pool selection method on the memory footprint raises the question: Can it be reduced from being bound by N to being bound by K? To find an answer, let us observe that if we used this method to select N from N, i.e. continued the selection process all the way to K=N, the output would contain a set of N randomly picked and disordered integers from 1 to N. Therefore, if our array T were populated with already randomly chosen *ordered* K items, we would only need to use the dynamic pool method to select *all* of them, i.e. K from K, in which case we would only need an array sized as [1:K].

But how can we get the randomly chosen, albeit ordered, set of K items in the first place? Here the venerable so-called "classic" K/N method comes to rescue.

SHUFFLED K/N SELECTION

The "classic" K/N method of selecting a random sample of size K out of N items without replacement is well known and has been used by SAS programmers for generations. Essentially, its algorithm is also based on the concept of a dynamically reduced selection pool size. Adapted to our purpose here, it proceeds in the following manner:

1. Set R = 1, and set the pool size to N.
2. If R > N, stop.
3. Generate a random variate from a uniform distribution in the [0:1] range.
4. If this variate is < K/N, select the current R for the sample; then decrement K by 1.
5. Decrease the pool size N by 1.
6. Increment R by 1 and go to step #2.

Since the probability of selection is continually adjusted depending on the number of already selected items and the remaining pool size, the algorithm guarantees that in the end, exactly K items out of N will have been selected. Again, it can be mathematically proven that the scheme provides each integer in the [1:N] with the same chance of being selected. In the DATA step language:

```
data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;
  do R = 1 to N ;
    if rand ("uniform") < divide (K, N) then do ;
      output ;
      K +- 1 ;
    end ;
    N +- 1 ;
  end ;
run ;
```

This approach does require to scan through the entire range [1:N]. However, compared to the array-based dynamic pool selection method (which also needs to scan through the whole range), its memory usage is virtually nil, as it does not need any lookup table for internal bookkeeping.

The only *problem*, from the standpoint of our stated goal, is that the selected values of R are output in perfect *ascending order*, whereas we need them to come out completely randomly *disordered*. The remedy is to pass the values of R to the dynamic pool selection routine. Only now, since the K/N selection method always generates exactly K items, all we need is to place them into an array of size [1:K] rather than [1:N] and execute the routine for K items out of K:

```
data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;
  array T [1:&K] _temporary_ ;
  do R = 1 to N ;
    if rand ("uniform") < divide (K, N) then do ;
```

```
      Q + 1 ;
      T[Q] = R ;
      K +- 1 ;
    end ;
    N +- 1 ;
  end ;
  do Q = 1 to &K ;
    S = &K - Q + 1 ;
    X = ceil (rand ("uniform") * S) ;
    R = T[X] ;
    output ;
    T[X] = T[S] ;
  end ;
run ;
```

Essentially, we (a) randomly select an ordered set of K integers out of N and then (b) *shuffle* them using the dynamic pool selection routine on their way to the output. This way, the memory footprint is bound by K, while at the same time we avoid the need to hit on duplicate picks and reject them. Being bound by K, this approach keeps memory usage at 8*K bytes. For example, for our typical test values of (K,N)=(2E7,1E8), it uses ~150 MB of RAM compared to ~760 MB required by the pure dynamic pool selection method.

The combined method runs about 50 per cent slower than pure dynamic pool selection. The reason is apparent: The former calls the RAND function (N+K) times while the latter calls it just K times. However, this deficiency is offset by the fact that the combined method will run within the memory limits defined by K as long as the value of N allows for scanning from 1 to N in a reasonable time. For example, it selects K=2E7 out of N=5E9 in under 3 minutes with the memory footprint still under ~150 MB of RAM.

K-BOUND HASH SELECTION

Every random selection method described thus far has at least one of the following shortcomings:

1. Checking for duplicate random picks and reject them.
2. Scanning the full range of the interval [1:N].
3. Calling the RAND function N or more times.
4. Having its memory footprint bound by N rather than K.

It raises the question whether it is possible to devise a scheme that would:

1. Require no checking for duplicate random picks.
2. Have its memory footprint bound by K.
3. Scan only from 1 to K and call the RAND function no more than K times.

The answer to this question is "yes". However, since "there ain't no such thing as a free lunch" in general and in computing in particular, such a scheme comes at the expense of a more complex bookkeeping data structure and much trickier logic - even though both are largely based on the concepts already presented above. Here is the main idea of the algorithm:

1. Keep a hash table T[P,R] with integer P as its key and integer R - as its data.
2. Randomly pick P from the range [1:S] where S is initially set to S=N, i.e. the full sampling range.
3. If P is not in the table, select R=P. Otherwise, choose R=T(P) from the table.
4. Add thus chosen R to the sample.
5. If an item with P=S (i.e. equal to the end of the current range) is not in the table, set R=S.
6. If P is not yet in the table, insert the T[P,R] pair into it.
7. Otherwise, update R for the table item for this value of key P with the value chosen in Step #5.
8. Decrement S by 1 and repeat from Step #2.

Essentially, the algorithm uses the same idea as the dynamic pool selection method, except that rather than keeping the entire [1:N] range in the lookup table, it keeps at most K items keyed by the already

selected picks P. If it hits a duplicate P, it selects the end-of-range value previously assigned to this P as a key-value in the table. The table below illustrates the process of randomly selecting K=9 out of N=9, i.e. every integer in the [1:N] range. If K < N items were to be selected, the process shown below would simply terminate at the corresponding value of K.

K(pick)	S(range)	P(picked)	R(sampled)	Hash Table State						
1	**9**	**8**	**8**	**P**						
				R						
2	8	**5**	**5**	**P**	**8**					
				R	**9**					
3	**7**	**1**	**1**	**P**	**5**	8				
				R	9	9				
4	**6**	**2**	**2**	**P**	**1**	5	8			
				R	**7**	9	9			
5	5	**2**	**6**	**P**	1	**2**	5	8		
				R	7	**6**	9	9		
6	**4**	**4**	**4**	**P**	1	**2**	5	8		
				R	7	**9**	9	9		
7	**3**	**3**	**3**	**P**	1	2	**4**	5	8	
				R	7	9	**4**	9	9	
8	2	**2**	**9**	**P**	1	**2**	**3**	4	5	8
				R	7	**9**	**3**	4	9	9
9	1	**1**	**7**	**P**	**1**	2	3	4	5	8
				R	**7**	9	3	4	9	9

Table 6. Dynamic Pool Hash Selection Step-by-Step

A DATA step implementation of the algorithm offered below is much more concise than its explanation. Note that in order for the program to work, table T does *not* have to be ordered - here it is done only to make it replicate the step-by-step content illustrated above. Also, though the *argument tags* used in the first FIND and REPLACE method calls can be omitted, their inclusion maps the program more clearly to the description of the underlying algorithm. Thus, in the DATA step language:

```
%let K = 9 ;
%let N = 9 ;

data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;

  dcl hash T () ;
  T.definekey ("P") ;
  T.definedata ("R") ;
  T.definedone () ;

  do K = 1 to K ;
    S = N - K + 1 ;
```

```
    P = ceil (rand ("uniform") * S) ;
    if T.find (key:P) ne 0 then R = P ;
    output sample ;
    if T.find (key:S) ne 0 then R = S ;
    T.replace (key:P, data:R) ;
  end ;
run ;
```

From the *purely algorithmic* standpoint, this is the most efficient scheme presented yet - and, at least as far as the authors are concerned, it is the most aesthetically elegant, too. Indeed, it does not have to check for duplicate picks and retry, nor does it have to burden the memory with more than K table items. In fact, most of the time the hash table ends up containing *fewer* items than even K (for instance, in the 9 out of 9 selection above, it only needed 6 items to maintain the bookkeeping).

That said, at K not too close to N and N moderate enough to allocate a key-indexed array or a bitmap, the sophistication of this method cannot stand up to the brute force of the key-indexed or bitmap insertion/search, even when they need to reject millions of duplicate picks. One reason is their incomparable insertion and search speed. The other is that for every pick, the K-bound hash selection has to search the hash table three times, plus perform one insertion or update.

However, the hash-based methods, whose memory usage, though considerable, is bound by K, stand alone when the sheer magnitude of N precludes any practical possibility of either allocating a lookup data structure with N items (even if they are just N *bits*) in memory or scanning through the [1:N] range in a reasonable time. For example, at K=1E7 (10 million) and N ≃ 1E12, i.e. 1 trillion, it is still possible, in principle, to use the shuffled K/N selection method; but it would take upward of 2 hours to scan from 1 to N and call the RAND function as many times. But doing so is hardly practical since selecting a sample of the same size takes between 1 and 2 minutes using the hash-based techniques (at the expense of ~500 MB of RAM). And when N approaches 1E13 and above, the hash-based methods realistically become the only sane choice.

HASH ARRAYS

To an efficiency- and performance-minded SAS programmer, it is still rather disconcerting when an algorithmically superior method runs significantly slower than brute force approaches. The reason the hash object seems to underperform in this particular situation is rooted in its versatility. Table lookup is just one of its many functions, and accommodating a host of other useful features it is laden with inevitably involves a good deal of overhead. In particular, it is designed to handle *any keys*, whether they are numeric, character, integer, fractional, simple or composite, etc., using the same internal algorithm. Thus, we cannot ask the hash object to use a specific approach if the keys we are dealing with have specific properties. Likewise, we cannot ask it to discard the data portion to save memory because our algorithm happens to need nothing but the key-values to be stored and searched.

Our present situation is a case in point, since here we are dealing exclusively with *integer* keys. Such *specific* keys are exceptionally well suited for *specific* hashing algorithms based on SAS arrays. This is because (a) the hash function can be rapidly computed using a single modulo operation and (b) we can control the table size and thus balance performance versus memory usage to suit our needs. Since a number of such schemes and the ways of coding them in the DATA step have been already worked out (e.g., Dorfman, 2001), we only need to choose one of them and adapt to the task at hand.

Here we will select the simplest hashing scheme based on so-called *collision resolution policy by linear probing*. In order to implement it, we need to do the following first:

1. Choose a number KL slightly larger than K - for example, KL=1.25*K or so. Traditionally, this is done by dividing K by a fraction LF termed the *load factor*. For example, in this case, LF=0.8 and KL=K/LF.
2. Find the first *prime number* greater than KL and assign it to a macro variable H, which then can be used to size the hash table array.

Given that we already have macro variable K populated, it is easy to do using a simple DATA step:

```
%let K  = 2E7 ;
%let LF = 0.8 ;

data _null_ ;
  do p = ceil (&K / &LF) by 1 until (j = up + 1) ;
    up = ceil (sqrt (p)) ;
    do j = 2 to up until (not mod (p,j)) ;
    end ;
  end ;
  call symputx ("H", p) ;
run ;
```

For example, for the numbers above, the step will generate H=25,000,009, which is the first prime greater than 2E7/0.8. By choosing LF, we effectively control the hash table size. Generally speaking, lower LF values result in a faster hash search at the expense of more memory. However, experiments show that setting LF=0.8 is a good compromise, below which memory usage grows much faster than speed. It is convenient to encapsulate the above step in a macro:

```
  %macro hsize (Number, LF, Result) ;
  %global &Result ;
  data _null_ ;
    do p = ceil (&Number / &LF) by 1 until (j = up + 1) ;
      up = ceil (sqrt (p)) ;
      do j = 2 to up until (not mod (p,j)) ;
      end ;
   end ;
   call symputx ("&Result", p) ;
  run ;
  %mend ;
```

After the macro variable K is assigned, it can be then called, for example, as follows:

```
  %hsize (&K, 0.8, H)
```

In this example, it will generate the needed prime number and assign it to macro variable H. Now, without further ado, we can recode the two hash programs already presented above using hash arrays.

NAIVE SELECTION VIA HASH ARRAYS

To recap, with naive selection we repeatedly generate random numbers in the [1:N] range and reject duplicate picks until we have arrived at the required sample size K. In the program below, the already selected integers are stored and tracked using temporary array T organized as a classic hash table:

```
%let K  = 2E7 ;
%let N  = 1E8 ;
%hsize (&K, 0.8, H)

data sample (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;
  /* Array T represents a hash table */
  array T [0:&H] _temporary_ ;
  do HIT = 1 by 1 until (Q = K) ;
    R = ceil (rand ("uniform") * N) ;
    /* Next 3 lines: Search hash table T using linear probing */
    do x = mod (R, &h) by -1 until (t[x] = . or t[x] = R) ;
      if x < 0 then x = &h - 1 ;
    end ;
    /* Key R is in hash table T: Loop for next pick R */
    if t[x] = R then continue ;
    output ;
    /* Key R is not in hash table T: Insert it */
```

```
    t[x] = R ;
    Q + 1 ;
  end ;
run ;
```

We will present the entire performance comparison picture later. For the time being, suffices it to say that this step runs in under 6 seconds and uses ~190 MB of RAM. By comparison, the step that uses the hash object to execute exactly the same naive scheme with the same K and N runs in 35 seconds and uses ~960 MB of RAM. Now let us see how the dynamic pool selection approach implemented with the hash object earlier may look if recoded using array-based hashing.

DYNAMIC POOL SELECTION VIA HASH ARRAYS

Let us recall that to implement this scheme, it is not enough to store just the key because we also need to store - and, later on, retrieve - some end-of-range integers. This is why the hash object table used earlier has P as its key and R - as its data. Correspondingly, to mirror the scheme using hash arrays, we need either two of them or one 2-dimensional array. Below two arrays are opted for for the sake of clarity:

```
%let K  = 2E7 ;
%let N  = 1E8 ;
%hsize (&K, 0.8, H)

data _null_ (keep = R) ;
  retain K &K N &N ;
  call streaminit (7) ;
  /* Hash table TP stores P as key, parallel array TR stores R as data */
  array TP [0 : &h] _temporary_ ;
  array TR [0 : &h] _temporary_ ;
  do K = 1 to K ;
    S = N - K + 1 ;
    P = ceil (rand ("uniform") * S) ;
    /* Search hash TP for P as key */
    do x = mod (P, &h) by -1 until (tp[x] = . or tp[x] = P) ;
      if x < 0 then x = &h - 1 ;
    end ;
    /* If P is not found, R=P, else R comes from the table */
    R = ifN (tp[x] = ., P, tr[x]) ;
    /* Memorize search index for insertion later on */
    xP = x ;
    output ;
    /* Search hash TP for S as key */
    do x = mod (S, &h) by -1 until (tp[x] = . or tp[x] = S) ;
      if x < 0 then x = &h - 1 ;
    end ;
    /* If S is not found, set R=S, else set R to table value */
    R = ifN (tp[x] = ., S, tr[x]) ;
    /* If P was not in table TP, insert (P,R) pair from PDV   */
    /* Else if it was, update TR with PDV value of R          */
    /* Either way, use index xP memorized from earlier search */
    tp [xP] = P ;
    tr [xP] = R ;
  end ;
run ;
```

This step finishes the job in under 8 seconds with the memory load of ~380 MB. By comparison, the hash object step, from which this scheme has been replicated verbatim, takes about 40 seconds with the memory footprint of ~960 MB.

HASH OBJECT vs HASH ARRAYS

The fact that in this special case the hash arrays outperform the hash object by a significant margin does not make the latter somehow "bad" or "slow". What it rather means is that within the scope of this particular task - specifically, when we have to deal with integer keys only - the hand-coded hash has the same kind of advantage any specialized tool does over a more versatile tool when the task suits its particular profile. The same can be said of key-indexing and bitmapping if N is small enough to comfortably fit N array elements or N bits in memory. The hash arrays work better than the hash object for selecting random integers for the same reason a dedicated 100 meter sprinter runs 100 meters much faster than a decathlete.

Furthermore, for a special tool to be effective, one has to know a thing or two about its internal intricacies. To code array-based hashing from scratch, one must understand the nuts and bolts of the algorithm, while to achieve the result using the hash object, one only needs to know how to push its buttons, whose actions are described in the SAS documentation and elsewhere (e.g., Dorfman and Henderson, 2018).

That said, since the hash array routines are already coded and tested, anyone can be use them in a copy-and-paste manner (informally known as "code cannibalization") in situations where they offer better run times coupled with smaller memory footprints.

RELATIVE EFFICIENCY COMPARISON

Since the efficiency of random selection is a stated goal of the paper, we feel compelled to present the results of some tests to show how the different methods described above stack against each other in terms of run time and memory footprint.

Every piece of code presented in this paper has been run and tested using SAS 9.4 TS level 1M4 under the X64_7PRO platform on a W520 Lenovo (Intel Core i7 2.4 GHz processor) with 8 GB of RAM and 471 GB SSD hard drive. The run time figures shown below have been obtained with the output turned off (using the DATA _NULL_ statement) in order to compare only the efficiency of the underlying algorithms unmarred by the time needed to write the output files.

The comparison figures below are given for K=2E7 (20 million) out of N=1E8 (100 million). These figures are chosen to make sure that all programs being compared:

1. Can handle a [1:N] scan.
2. Fit into 1 GB of RAM.
3. K is not too close to N, so the methods rejecting duplicate picks are not disadvantaged.

Now, the results:

Selection Type	Method	Run Time, sec	Memory, MB	Realistic Limits	
				N, ≤	K/N, ≤
Naive	Key-Index	3.8	760	1E9	0.9
	Bitmap	4.2	30	2.5E10	0.9
	Hash Object	34.0	960	9E15	0.7
	Hash Array	5.2	190	9E15	0.8
Dynamic Pool	Key-Index	4.4	760	1E9	1.0
	Hash Object	40.0	960	9E15	1.0
	Hash Array	7.3	380	9E15	1.0
Hybrid [1:N] Scan	K/N × Shuffle	6.0	150	1E12	1.0

Table 7. Random Selection Methods: Relative Efficiency

CONCLUSION

As is usually the case with SAS, when it comes to randomly selecting K unique integers out of N, there are many more ways than one to skin a cat. The approaches and algorithms presented in this paper cover various scenarios depending on K, N, and the K/N ratio and aim to maximize programming or machine efficiency depending on one's purpose and available resources.

Another reason one might want to peruse the paper is that while programming methods and their supporting data structures presented here are geared towards a particular task, they may be useful to anyone who would like to learn new high-performance SAS techniques or otherwise is generally keenly interested in nose-to-the-grindstone DATA step programming.

The authors also hope that readers will take a fresh look at the techniques like key-indexing, bitmapping, and array hashing. Though they have been relegated, for obvious reasons, to the back stage after the advent of the SAS hash object, the performance results presented in the paper remind that for certain types of data surgeries, a specifically shaped scalpel is more suitable than even the best Swiss knife.

REFERENCES

Wickling, R. 2013. "Six reasons you should stop using the RANUNI function to generate random numbers." Available at https://blogs.sas.com/content/iml/2013/07/10/stop-using-ranuni.html.

Dorfman, P. 2001. "Table Look-Up by Direct Addressing: Key-Indexing - Bitmapping - Hashing." *Proceedings of SUGI 26*, Long Beach, CA. Available at http://www2.sas.com/proceedings/sugi26/p008-26.pdf.

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

ACKNOWLEDGMENTS

The impetus for penning this paper was a May 2018 SAS-L discussion on the subject of generating random IDs started by Jimmy Lee and joined by Joe Matise, Rick Wickling, Vincent Martin, Ian Wakeling, Mark Keintz, Roger DeAngelis, and Daniel Nordlung. The authors would like to thank all these individuals for their ideas and specifically Ian Wakeling for offering the duplicate-rejection hash object approach presented here.

CONTACT INFORMATION

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

Paul M. Dorfman
Independent Consultant
sashole@gmail.com

Lessia S. Shajenko
Bank of America
lessia.s.shajenko@bankofamerica.com