Paper 232-31

CRAFTING YOUR OWN INDEX: WHY, WHEN, HOW

Paul M. Dorfman, Independent SAS® Consultant, Jacksonville, FL
Lessia S. Shajenko, Bank of America, Boston, MA

ABSTRACT

A SAS index is a search file associated with a data file. Given a key, the file is searched and, if the key is found, its record ID is returned and used to retrieve the data file record. For many practical tasks, this works sufficiently fast. But fast is never fast enough - SAS indexes are disk-resident and so leave much room for acceleration. SAS auto-maintains its indexes, which is splendid. But oftentimes the trade-off is programmatic rigidity. When either overweighs the pain to program, the indexing scheme suddenly appears to virtually lend itself to custom crafting. Being no longer tied to a particular search algorithm or storage medium, you can let your programming fancy fly, the power of the SAS language being at your disposal. The pain disappears. The disk-resident index file yields to a memory-based look-up returning a file pointer. B-trees are defenestrated in favor of search algorithms chosen for speed and/or to fit a particular key structure. This is a how-to paper about "external" indexing. Whys and whens are discussed. Coding details are furnished in all gore.

INTRODUCTION

Imagine two, not necessarily ordered, SAS data files created as follows:

```
data driver ;
   input kn kc $ ;
cards ;
5 kc5
1 kc1
3 kc3
run ;

data lookup (index = (k = (kc kn))) ;
   input kn kc: $9. sat $ ;
cards ;
7 kc7 s71
5 kc5 s51
1 kc1 s11
2 kc2 s21
3 kc3 s31
1 kc1 s12
3 kc3 s32
run ;
```

where KN and KC are a numeric and character keys forming a composite key (KN KC). For every record in DRIVER, we want to find all matching records by (KN KC) in LOOKUP *by overtly using the index K*, and write a join file MATCH with the following content:

```
kn  kc         sat
------------------
 1  kc1        s11
 1  kc1        s12
 3  kc3        s32
 3  kc3        s31
 5  kc5       s51
```

Since LOOKUP is indexed, the following Data step solution seems to be the most natural:

```
data iorc_match ;
   set driver ;
   do _iorc_ = 0 by 0 while ( _iorc_ = 0 ) ;
      set lookup key = k ;
      if _iorc_ eq 0 then output ;
      else _error_ = 0 ;
   end ;
run ;
```

(Of course, an SQL equijoin can be used; however SQL has its own mind as to index usage and may well sort the files behind the scenes instead.)

Now imagine that our real files are much, much larger, which means that the index will have to be searched much more often. Since SAS indexes are disk-resident (unless LOOKUP is loaded in memory using the SASFILE statement, which, contrary to expectations, adds no agility whatsoever to the step above), and use B-trees as the index structure, SAS index lookup is restricted to its search algorithm on disk.

It can be therefore reasonable to speculate that crafting an external, artificial index, whose lookup algorithm is restricted only by the programmer's fancy and skill, may work substantially faster and be much more flexible. This paper is the result of an attempt to establish (i) whether such assertion holds water; (ii) if it does, under which circumstances; and (iii) how to make your own index worth its key.

WHY

Let us further elaborate on selected points briefly touched upon in the Introduction. Any index, SAS indexes included, is nothing more than a search table structure consisting of two principal components:

- Keys
- Record identifiers (RID)

Given a key (which can be simple or composite, single-type or mixed-type), the index is searched. If the search is successful, all RIDs with the key can be identified and used [internally] to retrieve the corresponding rows from the table over which the index has been created. If the search is unsuccessful, it is signified by setting the automatic variable _IORC_ to something different from zero and _ERROR_ to 1.

With a SAS index, all those things happen behind the scenes, and this is a good thing. But by the same token, you have no say in which search mechanism to be used, even if the speed of the internal search mechanism seems to be insufficient, and the application needs to be accelerated. This is not such a good thing. Now, there exist dozens of searching algorithms, especially memory-based ones, working times faster than the search intrinsic to the disk-resident B-tree structure. What if we used one of these schemes, for example, the hash object available in SAS9? It is not unlikely that then we could find the key being searched (or determine that it is not in the table) significantly faster.

It is also easy to explicitly code the second part of the job done by the SAS index if the key has been found, namely to retrieve corresponding records from the indexed file by their record identifiers. The mechanism of achieving this, known as the POINT= option on SET or MODIFY statements, has been around for years, and with the course of time, it has only become better: SAS have made it faster and capable of accessing compressed files.

Thus, we can reasonably hope that by combining a fast, memory-based search scheme and direct access to a SAS data file by the observation number we can attain the level of performance significantly exceeding that of an intrinsic SAS index - a sufficient incentive for any inquisitive SAS programmer to stop speculating and start coding away.

HOW

An alert reader has no doubt noticed that the order of WHEN and HOW announced in the title has just been reversed. Well, there is a good reason for it: After having consumed and understood the paper, a user would first ask “when?” and then “how?”, just as declared above, whereas at this point, it is impossible to comprehend when to use your own index (and whether to create it at all) without first delving into gory details of constructing one and testing its relative performance.

1. Short Satellites, No Duplicate Keys in LOOKUP

The simplest case for making an artificial index is the situation when neither DRIVER nor LOOKUP have duplicate keys, and the satellite(s) in LOOKUP are so short that they can be held together with the keys in memory. This case has been investigated a number of times (see, e.g., [7]). First, let us model the data sets DRIVER and LOOKUP more suited to real-life performance testing than the little files used for demonstration purposes above. Note that:

1. Any SAS log notes not directly relevant to the issue being discussed are from now on omitted.
2. Code shown, unless otherwise noted, is executed using SAS9.1.3 under Windows XP Professional on dual PIII, 1GHz desktop with 1G of RAM.

```
data driver ;                           *concoct test file DRIVER                ;
   do kn = 1 to 1e6 ;                   *num  key KN in integer range [1:1000000] ;
      do kc = put (kn * 10, z9.) ;      *char key KC as 10*KN, with leading zeros ;
         output ;
      end ;
   end ;
run ;

NOTE: The data set WORK.DRIVER has 1000000 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           1.32 seconds
      user cpu time       1.17 seconds
      system cpu time     0.14 seconds
      Memory                             152k


data lookup (index = (k = (kn kc))) ; *concoct test file LOOKUP indexed by (kn kc) ;
   do kn = 0.5e6 + 1 to 1.5e6 by 2 ;  *KN to hit match half of KN values in DRIVER ;
      do kc = put (kn * 10, z9.) ;    *char key KC as 10*KN, with leading zeros    ;
         sat = 'sat' || kc ;          *dummy $char 12 data                         ;
         output ;
      end ;
   end ;
run ;

NOTE: The data set WORK.LOOKUP has 500000 observations and 3 variables.
NOTE: Composite index k has been defined.
```

Note that LOOKUP is deliberately populated with keys which would result in about half matches and half mismatches with the keys from DRIVER. Such test design is usually called for because successful and unsuccessful searches may (and do) bear different CPU costs. Let us see how the task of matching the files described in the Introduction is handled by the SAS index K:

```
data match ;                      *write DRIVER-LOOKUP matches by (kn kc) here ;
   set driver ;                   *read next record from DRIVER                ;
   set lookup key = k ;           *search index K for (kc kn)                  ;
   if _iorc_ eq 0 then output ;   *if found write PDV content out              ;
   else _error_ = 0 ;             *else prevent error msg from polluting log   ;
run ;

NOTE: There were 1000000 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.MATCH has 250000 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           14.00 seconds
      user cpu time       13.82 seconds
      system cpu time     0.14 seconds
      Memory                             220k
```

To find out whether the labor of crafting one’s own index could offer any advantage compared to the SAS index usage above, let us tailor our first hand-made index. Namely, let us store the entire LOOKUP in a SAS9 hash table using (KN KC) as a composite key and SAT as data. In the step below, the hash table is loaded [at the run-time] automatically by passing the input data set name LOOKUP to the table constructor as a parameter. Then file DRIVER is read in an explicit DO-UNTIL loop. Amongst other advantages, it provides the convenience of not having to deal with IF _N_=1 condition in order to isolate the loading of the table to the single instance - before DRIVER is read. For each [composite] key coming from DRIVER, HH.FIND() method is used to locate the key in hash table HH. If the key is found, SAT value from the table is implicitly moved into the “host” PDV variable SAT overwriting its current content, and the observation is written out; otherwise program control proceeds to reading the next record from DRIVER:

```
data match ;
   length sat $ 12 ;                                *match hash parameter type     ;
   dcl hash hh   (dataset: 'lookup', hashexp: 16) ; *load HH directly from LOOKUP  ;
   hh.definekey  ('kn', 'kc') ;                     *key HH by (kn kc) - unique!   ;
   hh.definedata ('sat') ;                          *keep SAT in HH as data        ;
   hh.definedone () ;                               *done instantiating HH         ;

   do until ( end_driver ) ;                        *read DRIVER in explicit loop  ;
      set driver end = end_driver ;                 *read record from DRIVER       ;
      if hh.find() = 0 then output ;                *found? move SAT->PDV, output  ;
   end ;                                            *pgm cntrl checks end_driver   ;
   stop ;                                           *stop to not load LOOKUP again ;
run ;

NOTE: There were 500000 observations read from the data set WORK.LOOKUP.
NOTE: There were 1000000 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.MATCH has 250000 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           3.00 seconds
      user cpu time       2.75 seconds
      system cpu time     0.21 seconds
      Memory                             28603k
```

Clearly, in this simplest case of unique search key (KN KC), the hand-made [hash] index wins by a wide margin. However, it should be no surprise, since the entire index together with the satellite data are completely held in memory (of which the memory usage note in the log leaves no doubts, either), and the index is searched by an extremely rapid hybrid [hash -> AVL tree search] algorithm.

2. The Choice of Search Algorithm [Digression]

At this point, let us step aside for a second to note that above, the hash table as the algorithm of choice was selected chiefly because SAS9 makes it so simple. If you are still running Version 8.2 (and many a folk still do), the artificial index similar to the one used above can still be constructed, albeit instead of the built-in hash object, a different structure would have to be used, for example:

- If the key is an integer in approximately [1:1E7] range (or wider, depending on RAM), key-indexed search [5] is times faster than any hash table, built-in or not -- indeed, than any search method at all.
- For wider ranges and very low-cardinality (for example, dichotomous) satellite variable, parallel bitmaps [5] can be used for constructing a very rapid memory-based index.
- Hand-coded hash schemes developed in [5] can be used with any production SAS version.
- It is not impracticable, in this simple case, to use a format.
- The keys and satellite(s) from sorted LOOKUP can be loaded in temporary arrays and searched using a binary or an interpolation search (see, e.g., [8]).
- Whatever lookup technique should strike your fancy.

3. Short Satellites, Duplicate Keys in LOOKUP [Back on track]

The situation when LOOKUP has duplicate keys in need to be collected upon a successful search poses no threat to a SAS index (just loop through the index using code given in Introduction), but does present an immediate difficulty for a number of schemes listed above. Foremost, if we want to continue using the SAS9 hash as the lookup table underlying the index, key duplicates have to be dealt with for two reasons:

1. The SAS9 hash object does not allow storing and retrieving duplicate keys.
2. It does not facilitate methods for partial key lookup (at least, as of yet).

So, if we want to keep using the hash table as the search portion of the artificial index, we need to:

1. Find a way to extend the [already composite] key in order to discriminate between different occurrences of the same key.
2. Find a method to search for the extended keys, whose values are not known from DRIVER file beforehand.

One way of solving the problem is to enumerate the keys with the same values and for each unique key, keep track of its group size (the number of duplicates) in a separate hash table keyed by the same key. But before writing code to solve the problem, we need another test file LOOKUP with duplicates in it. The following step duplicates each unique key randomly up to 2 times:

```
data lookup (index = (k = (kn kc))) ;
   do kn = 0.5e6 + 1 to 1.5e6 by 4 ;             *concoct KN key as before         ;
      do kc = put (kn * 10, z9.) ;               *form KC from KN as before        ;
         do _n_ = 1 to ceil (ranuni (1) * 3) ;   *now repeat (kn kc) up to 3 times ;
            sat = 'sat' || kc || put (_n_, 1.) ; *and concoct dummy atellite data  ;
            output ;                             *output each key duplicate / data ;
         end ;
      end ;
   end ;
run ;

NOTE: The data set WORK.LOOKUP has 500020 observations and 3 variables.
NOTE: Composite index k has been defined.
```

DRIVER file remains intact. Now we can follow the scheme outlined above with real-world code:

```
data match ;
   dcl hash hh   (hashexp: 16) ;       *HH to hold enumerated composite key and SAT ;
   hh.definekey  ('kn', 'kc', '_n_') ; *key HH by unique composite (kn kc _n_)      ;
   hh.definedata ('sat') ;             *SAT as data tied to unique enumerated key   ;
   hh.definedone () ;                  *finish instantiating HH                     ;

   dcl hash hs   (hashexp: 16) ;       *HS to hold (kn kc) as key tied to _N_ in HH ;
   hs.definekey  ('kn', 'kc') ;        *key HS by unique composite (kn kc)          ;
   hs.definedata ('_n_') ;             *keep enumerator _n_ in HS as data           ;
   hs.definedone () ;                  *finish instantiating HS                     ;

   do until ( end_lookup ) ;           *use explicit DO-UNTIL loop to read LOOKUP   ;
      set lookup end = end_lookup ;    *read next record from LOOKUP                ;
      if hs.find() ne 0 then _n_ = 0 ; *for each new key (kn kc) key  set _n_ =  0  ;
      _n_ ++ 1 ;                       *for each new (kn kc) occurrence set _n_+ 1  ;
      hs.replace() ;                   *store _N_ in HS keyed by unique (kn kc    ) ;
      hh.add() ;                       *store SAT in HH keyed by unique (kn kc _n_) ;
   end ;                               *here pgm cntrl checks for end_lookup        ;

   do until ( end_driver ) ;           *use explicit DO-UNTIL loop to read DRIVER   ;
      set driver end = end_driver ;    *if (kn kc) is not in HS, do nothing         ;
      if hs.find() = 0 then           /*if (kn kc) is not in HS do nothing, else:  */
      do _n_ = 1 to _n_ ;              *form composite (kn kc _n_) from 1 to _n_    ;
         hh.find() ;                   *get SAT from HH for each (kn kc) occurrence ;
         output ;                      *write a record to MATCH                     ;
      end ;                            *close iterative DO                          ;
   end ;                               *here pgm cntrl checks for end_driver        ;
   stop ;                              *no need for pgm cntrl to go to top of step  ;
run ;

NOTE: There were 500020 observations read from the data set WORK.LOOKUP.
NOTE: There were 1000000 observations read from the data set WORK.DRIVER.
```

```
NOTE: The data set WORK.MATCH has 249944 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           5.17 seconds
      user cpu time       4.70 seconds
      system cpu time     0.28 seconds
      Memory                             45149k
```

To compare this step with one making use of the composite non-unique SAS index K defined over (KN KC), we will need to enclose the index-reading directive

```
set lookup key = k ;
```

in a DO loop, so that for each distinct composite (KN KC) value, all RIDs from the index could be collected. The code line below,

```
do _iorc_ = 0 by 0 while ( _iorc_ = 0 ) ;
```

where **`BY 0`** is necessary to keep the loop iterating, is functionally equivalent to the combination of:

```
_iorc_ = 0 ;
do while ( _iorc_ = 0 ) ;
```

but the former form, besides being more concise, can foster other, not purely aesthetical, benefits under different circumstances; so let us keep it that way.

```
data iorc_match ;
   set driver ;
   do _iorc_ = 0 by 0 while ( _iorc_ = 0 ) ; *loop thru non-unique SAS index till no-match ;
      set lookup key = k ;                    *output each key occurrence if match is found ;
      if _iorc_ eq 0 then output ;
      else _error_ = 0 ;
   end ;
run ;
```

```
NOTE: There were 1000000 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.IORC_MATCH has 249944 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           15.21 seconds
      user cpu time       14.85 seconds
      system cpu time     0.32 seconds
      Memory                             212k
```

Once again, performance of the [hash-based] custom-made index dwarfs that of the SAS index despite the fact that the test case was contrived to its advantage: The index is well-discriminating and sits atop of an ascending key. However, it also dwarfs the SAS index in the code verbosity, and its apparent complexity.

4. Short Satellites, Duplicate Keys in Both DRIVER and LOOKUP

This seemingly complex code, though, may seem much less intimidating once we consider the case when duplicate keys are present in both DRIVER and LOOKUP files, coupled with the requirement that every combination of key-matching records from the two files must be output (once again, an SQL equijoin would make a short work of it, but here, we are expressly limiting ourselves with overt index usage for table lookup purposes). To make it clear, if we had input as follows:

```
data driver ;
   input kn kc $ ;
cards ;
5 c5
5 c5
1 c1
3 c3
3 c3
run ;
```

```
data lookup (index = (k = (kc kn))) ;
   input kn kc: $9. sat $ ;
cards ;
7 c7 s7
5 c5 s5
1 c1 s11
2 c2 s2
3 c3 s31
3 c3 s32
1 c1 s12
3 c3 s33
run ;
```

we would like to generate the following output:

```
kn  kc         sat
------------------
 1  c1         s11
 1  c1         s12
 3  c3         s31
 3  c3         s31
 3  c3         s32
 3  c3         s32
 3  c3         s33
 3  c3         s33
 5  c5         s5
 5  c5         s5
```

Now that duplicates are present in both LOOKUP and DRIVER, what alterations do we have to make to the code above utilizing the hand-made index in order to accommodate the new requirement? The answer is: *None whatsoever*, because duplicate key values in DRIVER have absolutely no effect on the operation of the artificial hash index. Each new key coming from DRIVER is merely looked up in the hash index (as many times as dictated by the enumerator _N_), and corresponding SAT values are output.

However, with the SAS index we run into a problem in this situation. The reason is that if DRIVER has two adjacent records with the same key value, the SAS index pointer will have been wound down to the end of the index after all index values have been fetched for the first such record coming from DRIVER. Thus, for the next record with the same key, index will find, and hence output, nothing. In order to rectify the situation, the index needs to be rewound to the point in the index file where the key begins - and this poses a problem, for which SAS does not offer a programming statement or (better yet) an option on the SET statement.

The only way to "rewind" the index is to supply it with a key, on which index search is guaranteed to fail, but to any professional programmer, such a method immediately gives off a thick smell of a kludge for the simple reason that it is a kludge. Indeed, in order for the method to work correctly, the fake key value must not be present in either DRIVER or LOOKUP, thus making the correctness of the program depend on input data. However, since at the moment this is the only option, here is one way of implementing it (another, which requires to sort DRIVER, is offered by Raithel [9]), where the "guaranteed to fail" key is made by setting its numeric portion to a special missing value - in [always fallible] hope that no such value will be found in the either input file:

```
data match (drop = _:) ;                          *kill auxiliary variables           ;
   set driver (rename = (kn=_kn kc=_kc)) ;        *read DRIVER renaming key variables ;

   if _kn = lag (_kn) and _kc = lag (_kc) then do ; *do adjacent records have same key? ;
      kn = .z ;                                   *if yes, make one key missing       ;
      link fetch ;                                *rewind index by failing the search ;
      _error_ = 0 ;                               *ok we know, do not pollute SAS log ;
   end ;

   kn = _kn ;                                     *restore key value from DRIVER      ;
   kc = _kc ;                                     *restore key value from DRIVER      ;

   do _iorc_ = 0 by 0 until ( _iorc_ ne 0 ) ;     *set _iorc_=0, loop through index   ;
      link fetch ;                                *search the index for real keys now ;
      if _iorc_ = 0 then output ;                 *output if found                    ;
      else _error_ = 0 ;                            *else prevent error log messages    ;
   end ;

   return ; fetch: set lookup key = k ; return ;    *no need for 2 LOOKUP buffers here  ;
run ;
```

As noted above, it no longer looks and/or feels as clean and neat as it does when one of the files has unique keys. Another important question is how such *modus operandi* affects performance. To test, we need another DRIVER file, now with duplicate keys, which we will generate in the manner already used to produce LOOKUP:

```
option fullstimer ;
data driver ;
   do kn = 1 to 1e6 ;
      do kc = put (kn * 10, z9.) ;
         do _n_ = 1 to ceil (ranuni (1) * 3) ; *now repeat (kn kc) up to 3 times ;
            output ;
         end ;
      end ;
   end ;
run ;

NOTE: The data set WORK.DRIVER has 1999727 observations and 2 variables.
```

The program using the artificial hash index remains intact; running it against the newly created DRIVER and already available last version of LOOKUP yields the following log notes:

```
NOTE: There were 500020 observations read from the data set WORK.LOOKUP.
NOTE: There were 1999727 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.MATCH has 499725 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           6.62 seconds
      user cpu time       6.01 seconds
      system cpu time     0.61 seconds
      Memory                            45149k
```

Let us now run the step where SAS index is rewound via searching for a fake key; the stats from the SAS log after the run are displayed below:

```
NOTE: There were 1999727 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.MATCH has 499725 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           1:01.75
      user cpu time       50.98 seconds
      system cpu time     10.65 seconds
      Memory                            214k
```

The only good thing that can be said about it when compared to the artificial index performance is that (i) it runs; and (ii) it uses 210 times less RAM. This would be a phenomenal advantage ten years ago; but nowadays trading 60 seconds using 0.25M of memory for 6 seconds using 50M of memory is a pretty fair deal.

Besides, we have not yet exhausted the possibilities of artificial indexing, which lend themselves to fostering much better memory usage than we have seen so far.

5. Long Satellites, Duplicates in Both DRIVER and LOOKUP

High memory usage by the hand-crafted index observed above is a direct consequence of the fact that out of sheer convenience and coding slothfulness, we have stored both the key portion of the index (KN KC) and the data it points to (SAT) in the same lookup table. In SAS index terms, this would be tantamount to the loading of the whole indexed file in memory, whilst in reality, a SAS index keeps both the lookup table (with the keys and RID pointers) and data associated with the RID pointers on disk, unless the SASFILE statement is used as discussed above.

Hand-tailored indexes are not limited to the restrictions imposed on SAS indexes by the underlying software, and so we are at liberty to hold the index search table and/or data anywhere we see fit. Obviously, the best place to keep the lookup structure is memory. However, in the majority of real-world scenarios, satellite information is usually way too voluminous to be stored in RAM, except for special cases when it comprises variables with relatively short summary length. In our example above, there is only one character satellite SAT with length 13, which, if repeated 500020 times, adds up to only 6M of RAM penalty, i.e. virtually no penalty at all. However, for lookup files with much longer satellites, it is very difficult to make case for keeping both keys and data in memory.

A viable compromise is easy to find in the very concept of an index, according to which an index contains only keys and pointers to disk records containing satellite information. If we decide - quite reasonably - to hold the lookup table in memory and satellites on disk, it is very easy to implement this concept in the SAS language:

- Keep the record keys and their observation numbers (RIDs) in RAM (e.g. a hash table, format, etc.).
- Keep LOOKUP satellite information in a SAS data file.
- For each key from DRIVER, search the table and return record pointer(s).
- Use the pointer(s) with SET POINT= instruction to get satellite information from the SAS file.

The only change we have to make to the already presented code in order to implement this scheme is to replace SAT as data portion of the lookup table with a record pointer (let us call it PTR, say), create this variable before a record is stored in the table, and finally use PTR to fetch records from disk-resident LOOKUP whenever a key from DRIVER is found in LOOKUP. Below, the matching step altered in the described way (right) is presented alongside its parent:

```
*-------------------------------;
* Keys and satellites in memory ;
*-------------------------------;
data match ;
   dcl hash hh   (hashexp: 16) ;
   hh.definekey  ('kn', 'kc', '_n_') ;
   HH.DEFINEDATA ('SAT') ;
   hh.definedone () ;

   dcl hash hs   (hashexp: 16) ;
   hs.definekey  ('kn', 'kc') ;
   hs.definedata ('_n_') ;
   hs.definedone () ;

   DO UNTIL ( END_LOOKUP ) ;
      set lookup end = end_lookup ;
      if hs.find() ne 0 then _n_ = 0 ;
      _n_ ++ 1 ;
      hs.replace() ;
      hh.add() ;
   end ;

   do until ( end_driver ) ;
      set driver end = end_driver ;
      if hs.find() = 0 then
      do _n_ = 1 to _n_ ;
         hh.find() ;

         output ;
      end ;
   end ;
   stop ;
run ;
```

```
*------------------------------------;
* Keys in memory, satellites on disk ;
*------------------------------------;
data match ;
   dcl hash hh   (hashexp: 16) ;
   hh.definekey  ('kn', 'kc', '_n_') ;
   HH.DEFINEDATA ('PTR') ;
   hh.definedone () ;

   dcl hash hs   (hashexp: 16) ;
   hs.definekey  ('kn', 'kc') ;
   hs.definedata ('_n_') ;
   hs.definedone () ;

   DO PTR = 1 BY 1 UNTIL ( END_LOOKUP ) ;
      set lookup end = end_lookup ;
      if hs.find() ne 0 then _n_ = 0 ;
      _n_ ++ 1 ;
      hs.replace() ;
      hh.add() ;
   end ;

   do until ( end_driver ) ;
      set driver end = end_driver ;
      if hs.find() = 0 then
      do _n_ = 1 to _n_ ;
         hh.find() ;
         SET LOOKUP POINT = PTR ;
         output ;
      end ;
   end ;
   stop ;
run ;
```

The differing lines of code above are given in the uppercase. Note that the variable PTR will be dropped from MATCH automatically, as it is used with POINT= option.

One would think that the extra I/O will set performance back significantly, yet this is not the case. Running the step on the right against the same DRIVER and LOOKUP files as before yields the following:

```
NOTE: There were 500020 observations read from the data set WORK.LOOKUP.
NOTE: There were 1999727 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.MATCH has 499725 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           7.95 seconds
      user cpu time       7.37 seconds
      system cpu time     0.54 seconds
      Memory                             40690k
```

which is but a second or so slower compared to the step on the left. However, its memory usage is now determined by and limited to keys only, since the pointer always takes the same 8 bytes per table row.

6. Looking Back: A Quick Summary

Let us step back for a moment, abstract from the concrete lookup algorithm (V9 hash) used in the examples shown so far, and crystallize how, in essence, we go about creating and using an artificial index. For the sake of generality, let us assume that the index is too large to fit in memory completely together with all satellite variables, and so we shall use RAM only to hold the lookup table pointing to satellite information left resident on disk. This coincides with the scheme considered in paragraph 5 above.

First, let us consider the *index creation phase*:

1. Read a record from LOOKUP table keeping the key only
2. Store the key in a RAM-resident lookup table (e.g. Hash table)
3. Store the corresponding observation number N alongside with the key
4. If there are any records left, go to #1, else stop.

Now, let us consider the *index usage phase* given a DRIVER file:

1. Read a record from DRIVER
2. Search its key in the lookup table
3. If not found go to #1
4. Else use the pointer alongside it with POINT= option to retrieve the lookup satellites
5. Write the record out
6. If the lookup table has more matches for the key go to #4
7. If there are any records left in DRIVER, go to #1, else stop.

The above schemes are not unique to artificial indexes: Automated SAS indexes use the same behind-the-scenes, except that the lookup table is kept both disk-resident and disk-searched.

This quick review should make it all the more plain, in addition to the brief notes to the same effect outlined in paragraph 2 (“Digression”), that principally, it can accommodate any particular lookup algorithm. The latter then can be chosen from whatever considerations the developer deems to be paramount. For example, the reason for a particular algorithm selection may include (but not be limited to) the following:

- the lookup method is the only one the developer knows (perhaps in this case, it behooves one to learn more)
- the developer feels “comfortable” with it (if this is a V9 hash, this is easy to understand)
- lookup keys possess unique properties making them suitable for implementing an especially quick and/or simple lookup algorithm

7. A Special Case: Keys Good for Key-Indexing

The last point may deserve special consideration. If the lookup key is *known and guaranteed* to be integer and fall in a limited range, then it lends itself to the fastest existing search algorithm known as key-indexed search (see *[5]* for detailed description and discussion). By “limited” range, we mean a range which permits to compile a numeric temporary array with enough cells to accommodate all possible key values. With the modern memories, such range is realistically limited to integer keys which may take up on about 1E+8 discrete values. For example, a 7-digit positive patient ID is definitely fine, but a 9-digit SSN number would perhaps be too large.

For instance, a key with such properties can be obtained for our test purposes by eliminating the KC part from the composite key in the test data sets used previously:

```
41   data driver ;                        *concoct test file DRIVER                 ;
```

```
   do kn = 1 to 1e6 ;                 *num  key KN in integer range [1:1000000] ;
      output ;
   end ;
run ;

NOTE: The data set WORK.DRIVER has 1000000 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.20 seconds
      cpu time            0.20 seconds


data lookup ;                         *concoct test file LOOKUP indexed by (kn kc) ;
   do kn = 0.5e6 + 1 to 1.5e6 by 2 ;  *KN to hit match half of KN values in DRIVER ;
      sat = 'sat' || put (kn, z12.) ; *dummy $char 12 data                         ;
      output ;
   end ;
run ;

NOTE: The data set WORK.LOOKUP has 500000 observations and 2 variables.
```

By design, such a key has the integer range from 1 to 1.5 million. In the program excerpt below, the key-indexed search is implemented on the base of the temporary array PTR. Since we have no more than 1.5E+6 possible integer key values, the table need not contain more than 2 million cells, so the size margin used below should be more than comfortable:

```
data match ;
   array ptr [0 : 1999999] _temporary_ ; *key-indexed table ;

   do _n_ = 1 by 1 until (end_lookup) ;
      set lookup end = end_lookup ;
      ptr [kn] = _n_ ;
   end ;

   do until (end_driver) ;
      set lookup end = end_driver ;
      _n_ = ptr [kn] ;
      if missing (_n_) then continue ;
      set lookup point = _n_ ;
      output ;
   end ;
run ;

NOTE: There were 500000 observations read from the data set WORK.LOOKUP.
NOTE: There were 1000000 observations read from the data set WORK.DRIVER.
NOTE: The data set WORK.MATCH has 250000 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           1.85 seconds
      cpu time            1.81 seconds
      Memory                             16108k
```

Owing to the lightning-quick nature of the key-indexed search, this is fast by any comparison - even though for every matching key, we had to randomly read a record from the disk-resident satellite container. An alert reader must have noticed that this code does not deal with possible lookup duplicates. However, due to the utter simplicity of the scheme, it is not difficult to accommodate the possibility using additional parallel array(s).

As with all things of this nature, a question arises: Do such “perfect” keys exist in reality to even spend time on coding artificial indices based on their non-generic properties? Luckily, there are many practical situations where key-indexing can be of tremendous value. From the few of them listed in [5], let us mention the obvious example of the SAS date value with the intrinsic natural range of ‘01jan1582’d=-138061 to, say, ‘25mar2506’d=199506 (hoping that the SAS System will be in use for another 500 years), or the SAS time values which can never be out of [0:86400] range.

More importantly, virtually all contemporary ETL processes include a mechanism (often hash-based!) providing for the generation of so-called surrogate keys, by which the fact table and dimensions are joined. By design, the surrogate keys are always positive integers; by the nature of the natural keys they replace, they are always limited in range. For example, high-watermark surrogate key values may correspond to the never-too-large maximum number of procedure codes, diagnosis codes, provider identification keys, etc., available in the system. This makes artificial indexes based on the key-indexed search an ideal join tool in a properly designed data warehouse.

8. Best Index - NO Index?

So far, we have assumed that the DRIVER file does not have its records in any particular order, which has been the primary reason to create an index on a smaller file, LOOKUP. Of course, in all of the examples above, the entire DRIVER has had to be read under all circumstances, the observed performance differences having been due to the organization of the index created over LOOKUP.

But what if the DRIVER file is already known to be in order by its key? Since this is a SAS data set, we can read its records at random; and since it is sorted, such read can be implemented as the binary, interpolation, or Fibonacci search. With a relatively few records to be examined in LOOKUP against DRIVER, we can reverse their roles and binary-search DRIVER for every record coming from LOOKUP, and potentially end up reading only a handful of DRIVER records randomly instead of reading the entire thing serially.

Consider the following test data and matching steps:

```
%let n = 1e6 ; *records in lookup (ex-ddriver) ;
%let r =   4 ; *records in driver (ex-lookup) as &n/&r ;

data DRIVER LOOKUP (drop = sat:) ;
  do kn = 1 to &n ;
    sat = put (kn, z12.) ;
    kc = put (substr (sat, 8), $5.) ;
    output DRIVER ;
    if ranuni (1) < 1/&r then output LOOKUP ;
  end ;
run ;

NOTE: The data set WORK.DRIVER has 1000000 observations and 3 variables.
NOTE: The data set WORK.LOOKUP has 249643 observations and 2 variables.

data match (drop = _:) ;
   set LOOKUP ;
   _l = 1 ;
   _h = n ;
   do until ( _l > _h | _match ) ;
      m = floor ((_l + _h) * .5) ;
      set DRIVER (rename = (kn = _k1 kc = _k2)) point = m nobs = n ;
      if      kn <= _k1 and kc < _k2 then _h = m - 1 ;
      else if kn => _k1 and kc > _k2 then _l = m + 1 ;
      else _match = 1 ;
   end ;
   if not _match then call missing (sat) ;
run ;

NOTE: There were 249643 observations read from the data set WORK.LOOKUP.
NOTE: The data set WORK.MATCH has 249643 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           6.98 seconds
      user cpu time       6.76 seconds
      system cpu time     5.10 seconds
      Memory                            215k
```

Now this is not particularly speedy under this concrete arrangement, but we did not have to create any index at all - SAS or otherwise! To get matching records from the larger file, we just read it randomly. If this is not impressive, note that for each record read from LOOKUP, DRIVER has to be read randomly no more than log2(N)=20 times if N=1E+6 (as above), and no more than 29 times if N=1E+9. Try to extrapolate the result to DRIVER with a billion records (not uncommon nowadays), and it is easy to see how such a technique can potentially save huge resources.

Once again, the code is not difficult to amend in order to retrieve potential duplicate keys in (this time) DRIVER. A successful binary search lands its pointer M somewhere inside the cluster of identically keyed adjacent records. A simple loop, scrolling the file first from M down while the key is the same and then from M up while the key is the same, will suffice. Such code with all the gory details is readily found in the SAS-L archives (2005) located at:

http://www.listserv@listserv.uga.edu/archives/sas-l.html

WHEN

Now that we know a bit or two about artificial index performance under certain scenarios, it is not difficult to develop a few tentative conditions, under which creating and using such a structure appears more beneficial than using a SAS index proper:

- Lackluster performance of a SAS index overtly used as a lookup table
- User community, to which cutting a response time by a minute makes real difference
- Sufficient expertise in understanding indexes in general on part of SAS programming staff
- Lack of programming sloth and fear of relatively sophisticated coding on part of the same
- Availability of SAS9 (hash object) or local expertise in high-performance table lookup programming (any lookup algorithm)

CONCLUSION

Crafting your own index is not difficult, and under a number of circumstances it may offer a stark difference in performance compared to overtly used SAS indexes. An artificial index is not bound to a predetermined lookup scheme, and index search can be performed entirely in memory, while data to which the index points can be either memory-resident or disk-resident, depending on the available resources, performance needs, and, ultimately, common programming sense.

REFERENCES

[1] D.E. Knuth, *The Art of Computer Programming*, 2.
[2] D.E. Knuth, *The Art of Computer Programming*, 3.
[3] R. Sedgewick, *Algorithms in C*, 1-4.
[4] T.A. Standish. *Data Structures, Algorithms and Software Principles in C.*
[5] P.M. Dorfman. *Table Lookup via Direct Addressing: Key-Indexing, Bitmapping, Hashing.* Proceedings of SUGI 26, Long Beach, CA, 2001.
[6] M.A. Raithel. *Tuning SAS Applications in the MVS Environment*, Cary,NC:SAS Institute Inc.,1995.
[7] P.M. Dorfman, K.Vyverman. *Data Step Hash Objects as Programming Tools*. Proceedings of SUGI 30, Philadelphia, PA, 2004.
[8] P.M. Dorfman. *From Sequential Search to Key-Indexing*. Proceedings of SESUG'99, Mobile, AL, 1999.
[9] M.A. Raithel. *The Complete Guide to Creating and Using Indexes*, Cary,NC:SAS Institute Inc.,2005.

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
4437 Summer Walk Court
Jacksonville, FL 32258
(904) 260-6509 (h)
(904) 791-6889 (o)
(904) 226-0743 (c)
sashole@bellsouth.net

Lessia S. Shajenko
10 Vale Rd.,

Belmont, MA 02478-4301
(781) 788-7819
Lessia.S.Shajenko@bankofamerica.com