Paper 129-25

Private Detectives In A Data Warehouse: Key-Indexing, Bitmapping, And Hashing

Paul M. Dorfman

Citibank Universal Card, Jacksonville, FL

ABSTRACT

In data processing in general, and numerous aspects of Data Warehousing, in particular, searching is one of the most frequently performed operations. Base SAS offers a rich collection of built-in searching techniques. Merging and SQL joins, formats and SAS indexes - all serve the purpose of looking up relevant data. In addition, SAS Language incorporates arrays – the data structures ideal for implementing just about any searching algorithm. An array-based lookup is not a ready-to-go recipe: It has to be hand-coded and tuned. However, this "private detective" approach is more flexible and may result in programs searching much faster and using fewer resources than the "heavy artillery".

In this paper, SAS arrays are viewed as a DATA step structure ideally suited for quick data lookup based on the idea of direct addressing. Three such techniques - key-indexing, bitmapping, and hashing - are considered in their logical sequence using a real-life example of matching data files by a common key. The results of benchmarking presented in the paper show that these methods beat even the fastest ready-to-use tools like popular "large formats", by a wide margin. Thus, they can be indispensable in any massive data processing setting such as a Data Warehouse, where speed and efficiency considerations are paramount.

INTRODUCTION

In this paper, we will concentrate on a group of in-memory lookup techniques based on direct or almost direct addressing into a temporary SAS array. First, we shall consider *key-indexed search.* Then we will try to expand its domain by viewing an *array as a bitmap*. Finally, we will see how the core idea of key-indexing can be generalized to arrive at a hybrid search method called *hashing*.

To make the discussion more practical, we will consider a common task of matching two files by a common variable. This will help us see how different lookup methods compare to each other and to some of the ready-to-go methods offered in SAS.

Imagine an unsorted file SMALL containing N_SMALL records with a key variable KEY and satellite variable S_SAT. Another unsorted file, LARGE, with N_LARGE records, also contains KEY and a satellite field L_SAT. *Let us assume, for the time being, that the keys are integers.* Assume that LARGE is so big that sorting is not an option; however, also assume that we have enough memory to hold all keys from SMALL at once. Given these conditions, *What is the most efficient way to subset LARGE based on the values of KEY in SMALL* to produce a file MATCH? SAS offers a number of ready-to-go tools based on in-memory table lookup. For example:

1. Compile unduplicated keys from SMALL into a format via CNTLIN=, and search it for each KEY read from LARGE.
2. Join the files using BUFFERSIZE large enough to make the SQL optimizer use SQXJHSH access method.
3. Load the keys from SMALL into a sorted array, and use a hand-coded binary or interpolation search.

With plenty of working methods available, why try something else? Because there are faster and more efficient ways to do the trick!

KEY-INDEXING

Most of the ready-to-go and hand-coded searching methods are based on *comparing a search key to all or some keys in a memory table.* It makes them *principally limited* since generally, no comparison-based method can search in fewer iterations than binary search. We could therefore try to remove the limitation by *doing away with the key comparisons* altogether. But is it possible to search for a key without at least one comparison? The answer is "yes", and it is given by a radically different searching philosophy called *direct addressing,* that finds its pure expression in *key-indexed search.* The idea is simple. Imagine that all keys are 1-digit numbers from 0 to 9, and that SMALL has just 9 records:

```
KEY   | 2 | 3 | 5 | 2 | 7 | 9 | 5 | 7 | 3 |
------+-----------------------------------
S_SAT | 1 | 2 | 3 | 0 | 4 | 5 | 6 | 9 | 7 |
```

Let us create a temporary array HKEY with *one node (location, address) allocated for each possible key value*:

```
 0   1   2   3   4   5   6   7   8   9
[.] [.] [.] [.] [.] [.] [.] [.] [.] [.]
```

Now, for each key from SMALL, let us look at the array location *whose index is equal to the value of the KEY. If the node is empty,* we move S_SAT there:

```
 0   1   2   3   4   5   6   7   8   9
[.] [.] [1] [2] [.] [3] [.] [4] [.] [5]
```

We have just created a *key-indexed table* comprising two types of *entries*: *empty and occupied*. Inserting a satellite only if the node is empty retains its first instance corresponding to a repeating key value, otherwise the last instance would be used. Either way, *duplicate keys are deleted automatically* as the table is loaded. If SMALL has no satellites or they are of no interest, the entries of the key-indexed table could be marked as occupied by moving 1 into the node, the unduplication effect remaining intact.

To search for a key, we simply examine the table location *whose index is equal to the key*. If it is missing, the key is not in the table; if not, then the key is found, and the node contains the respective satellite value. For example, if KEY=1, the search fails since the address 01 of the table is empty. If KEY=7, the node 07 is occupied, so the key is found, and the node returns the satellite value HKEY(KEY)=4. Note that the loading process *implicitly incorporates searching*: To determine if the node empty we in effect search the table to find out if key has already been marked in the table. If yes, it is a duplicate, and the next record is to be read. The nature of the process makes it unnecessary to sort SMALL or insert

the keys themselves, as effectively the keys are "inserted" by making their corresponding nodes occupied by the satellites or 1. Now suppose that all keys are integers from –4E6 to +4E6. It naturally defines the bounds of the key-indexed table. Using the key-indexing scheme given above, the sample problem can be solved straightforwardly as follows:

```
*** KEY-INDEXED TABLE LOAD AND SEARCH ***;

DATA MATCH;
   ARRAY HKEY (-4000000:4000000) _TEMPORARY_;
   DO UNTIL (EOF1);
      SET SMALL END=EOF1;
      IF HKEY(KEY) = . THEN HKEY(KEY) = S_SAT;
   END;
   DO UNTIL (EOF2);
      SET LARGE END=EOF2;
      S_SAT = HKEY(KEY);
      IF S_SAT > . THEN OUTPUT;
   END;
RUN;
```

From the nature of the algorithm, it is clear that *no lookup method is simpler or can run faster than key-indexing*: It completes any search, hit or miss*, without comparing any keys,* via a single array reference. It also possesses the fundamental property: *Its speed does not depend on the number of keys* "inserted" into the table, i.e. any single act of key-indexed search takes precisely identical time.

To see how well key-indexing performs, it was compared in load and search phases to formatting, SQXJHSH, and other methods presented below, for N_LARGE=2E6 and a number of N_SMALL values using SMALL and LARGE. The results shown in the Section "Benchmarking" testify that at the huge memory expense, key-indexing completely dominates the competition, for instance, out-performing MERGE running against *already sorted* input as 5:1.

If key-indexed search is that fast, why not use it all the time? The fly on the ointment is that it is only *applicable when lookup keys are integers falling in a limited range.* Our test keys can only take on 8,000,001 distinct values, so sufficient array space can be allocated for all keys at the expense of 64 MB of memory. With even more memory, we could get away with 7-digit keys. But to deal with a 9-digit SSN, an almost unfeasible array with 1 billion elements would be needed, while a 16-digit credit card number would make key-indexing a technical fantasy. On the other hand, in many real-world applications when keys do indeed fall in a limited range, key-indexing, with its blazing speed and simplicity, is beyond reproach. Here are some examples:

1. SAS dates. Any date value up to year 4000 can be accommodated by a [-138061:380217] table.
2. SAS times. An array sized as [0:86400] will suffice.
3. ICD9/CPT4 codes. If some character is a letter, it can be converted into a number.
4. PIB2. informat maps any 1- and 2-byte character key onto the [0:65535] range.
5. Any fractional keys if limited in range when rescaled.

The usefulness of key-indexing in such situations does not rid it of its inherent limitation. It would be too bad, though, to let such a neat idea go underused just because of its greed for memory. But to expand the universe of usable keys, we must find a practical way of keeping memory consumption at bay. Note that both the speed and limitation of key-indexing stem from the following factors:

- The lookup table is directly addressed.
- The entire set of possible key values is addressable: A separate node is allocated for each possible key value.
- No comparisons between keys are made.

Therefore, we can choose one of two principally different paths:

1. Still keep all possible key values addressed, but *expand the addressable range of keys* by addressing them into the bits of array items rather than the items themselves.
2. Eliminate *any restrictions* imposed on the nature of lookup keys by dropping the requirements that a) no two distinct keys shall reside in one node, and b) no comparisons are allowed.

The first approach results in a technique called *bitmapping*. The second path leads to a more versatile hybrid searching method known as *hashing*. Let us discuss bitmapping first.

BITMAPPING

Suppose we do not care about pulling S_SAT along with a key found in SMALL. Then a key-indexed table is only required to tell whether a node corresponding to the key is empty or occupied. Key-indexing uses full 8 *bytes*, the memory length of a numeric array item, to store a binary value, while obviously a single *bit* will suffice. But how do we make efficient use of bits for such a purpose?

The question would not even arise were it possible to have a temporary array with 1 bit per item, as then the bits would be addressed directly via an index. Unfortunately, the shortest memory length reserved by a temporary SAS array is always 64 bits per item regardless of the declared expression length. Hence, to index the bits composing array elements, additional computations are needed.

At first glance, it seems natural to create a *character array* with the shortest allowable memory length, $8, and the number of elements equal to the number of all possible key values divided by 64. This would obviously allow cutting memory usage 64 times. Say we are dealing with 8-digit natural keys, thus needing 1E8 bits to address all of them. The equivalent amount of storage, about 12 MB, is nowadays insignificant. An array with 1,562,500 8-byte items will thus cover the entire universe of possible keys. (Note that key-indexing would require about 760 MB, mostly wasted.) To mark a key as "present", we might then to proceed as follows:

1. ARRAY BITMAP (0:1562500) $8 _TEMPORARY_.
2. X= INT(KEY/64). Thus X points to the array item containing the bit to be turned on.
3. R=1+MOD(KEY, 64). Now R points to the correct bit.

To turn the bit on, we would compute X and R, set BITMAP(X) to binary zeroes as "0000000000000000"X if it is still blank, convert it to a 64-*byte* BITSTR mirroring the bit content of BITMAP(X) , set R-th byte to 1, reconstruct BITMAP(X), and insert it back:

```
BITSTR = PUT(BITMAP(X),$BINARY64.);
SUBSTR(BITSTR,R,1) = '1';
BITMAP(X) = INPUT(BITSTR,$BINARY64.);
```

Searching for a key, we would find X and test BITMAP(X). If it is blank, we have a miss; otherwise, we would compute R and use either of the expressions

```
SUBSTR(PUT(BITMAP(X),$BINARY64.),R,1) EQ "1"
INPUTN(BITMAP(X),'BITS1.'||PUT(R,Z2.))
```

to tell whether the key is in the table.

Unfortunately, this scheme returns lackluster performance: To examine *a bit* or turn it on, we have to either memory-write full *64 bytes* or use the slowly working modified informat. So, to achieve a decent search speed, we must find a way to locate individual bits by performing a *rapid computation on the entire item* rather than breaking it apart. Hence we could try using a numeric array instead. That brings about two interrelated questions:

1. What kind of operation should be performed on a numeric array item in order to turn its R-th bit on?
2. How to find out if R-th bit is on using a direct computation?

The first issue is easily resolved by adding 2**(R-1) to the item, all the more that a series of binary powers can be stored in an auxiliary array beforehand. However, there are several subtle caveats:

Caveat 1. It must be never attempted to turn a bit on if it is already on since otherwise the added unity becomes a carry turning one or more of the higher bits to 1, wreaking havoc in the entire bitmap. So, the answer to question 1 above cannot be found without answering question 2 first. Of course, the bit checking is superfluous if BITMAP(X) is missing, meaning that none of the bits has been set to 1 yet.

Caveat 2. Before a bit can be turned on, a missing item must be set to 0. (However, initializing the entire array to zeroes is not necessary, as at the search stage, missing items can be skipped.)

Caveat 3. How many bits per array item can be used in this manner? Since they *must* be limited to the mantissa, it leaves us with 56 usable bits under OS/390 and 53 bits under NT, so the divisor 64 in the formulae above must be changed accordingly. Thus, by switching to a numeric array, we sacrifice about 15 per cent of memory utilization for the sake of speed. To answer question 2, let us think first how to determine whether, say, the 4th least significant *decimal* digit of a variable N is not zero. Naturally, we would divide by 1E4 and find the remainder; if it is no less than 1E3, the digit is not 0, otherwise it is 0. By induction, a boolean expression

```
MOD(N,10**R) GE 10**(R-1)
```

indicates whether R-th decimal place is "on" or "off". Similarly,

```
MOD(N,2**R) GE 2**(R-1)
```

is set to the value of R-th bit of the mantissa of N. Now we should have no problem "keying" a numeric-array bitmap:

Step 1. Allocate a numeric temporary array BITMAP bound from 0 to 10**(number of key digits)/M where M=56 or M=53. Also, create an auxiliary array B and fill it with the consecutive powers of 2.
Step 2. Read a record with KEY from SMALL.
Step 3. Locate the array element: X=INT(KEY/M). If BITMAP(x) is missing, then set it to 0.
Step 4. Locate the bit: R=MOD(KEY, M). If BITMAP(X) is missing, go straight to Step 5. Otherwise check the bit. If it is already 1, the key is duplicate: return to Step 2.
Step 5. Turn the bit on: BITMAP(X) ++ B(R). Go to Step 2.

Searching for a key in such a bitmap table is equally simple:

Step 1. Read a record from LARGE.
Step 2. Locate the array element: X=INT(KEY/M). If BITMAP(X) is missing, the search is unsuccessful - hence return to Step 1.
Step 3. Locate the bit: R = MOD(KEY, M).
Step 4. Check the bit. If it is 0, the key is not found - go to Step 1. Otherwise, write the record and return to Step 1.

Before translating the algorithm into SAS, the possibility of negative keys should be accounted for. With key-indexing, it is easy: Just assign the lower bound of the table the lowest negative key value. With a bitmap, we can achieve it by shifting all keys up by the absolute value of the lowest key MINKEY, rescaling the upper BITMAP bound accordingly, and leaving the lower bound at 0.

```
*** BITMAP TABLE LOAD AND SEARCH ***;

%LET M      =   56; *! 53 if not S/390 ;
%LET MINKEY = -4E6;
%LET MAXKEY = +4E6;
%LET LO = %SYSFUNC(FLOOR(&MINKEY/&M));
%LET HI = %SYSFUNC(FLOOR(&MAXKEY/&M));
%LET HB = %EVAL(&HI - &LO);

DATA MATCH (KEEP=KEY L_SAT);
   ARRAY BITMAP (0:&HB) _TEMPORARY_;
   ARRAY B      (0:&M ) _TEMPORARY_;
   DO X=0 TO &M; B(X) = 2**X; END;
   DO UNTIL (EOF1);
      SET SMALL END=EOF1;
      X = INT((KEY - &MINKEY) / &M);
      IF BITMAP(X) EQ . THEN BITMAP(X) = 0;
      R = KEY - &MINKEY - X*&M;
      IF MOD(BITMAP(X),B(R+1)) LT B(R) THEN
         BITMAP(X) ++ B(R);
   END;
   DO UNTIL (EOF2);
      SET LARGE END=EOF2;
      X = INT((KEY - &MINKEY) / &M);
      IF BITMAP(X) EQ . THEN CONTINUE;
      R = KEY - &MINKEY - X*&M;
      IF MOD(BITMAP(X),B(R+1)) GE B(R) THEN
         OUTPUT;
   END;
RUN;
```

The macro assignments at the top take care of the shifting and rescaling. M=53 will work under any OS; however, under OS/390, it is possible to choose M=56 and save memory. The first loop populates B with powers of 2. The <EOF1> loop can be thought of as "bitmap compilation". The <EOF2> loop performs the actual bitmap searching and matching. Because of the extra computations, bitmapping runs about 50 per cent slower than key-indexing, yet it uses 53 to 56 times less memory and is still twice as fast as MERGE!

Above, the bitmap compilation loop resides in the same step where it is searched. However, a bitmap can also be compiled in a separate step, stored in a file, and reused thereafter. To do so, we would only have to change the DATA statement to, for instance,

```
DATA LIB.BITMAP (KEEP=BYTE8);
```

and replace the <EOF2> loop with

```
   DO X=0 TO &HB;
      BYTE8 = BITMAP(X);
      OUTPUT;
   END;
```

A saved bitmap can be subsequently utilized any time a file needs to be subset, validated, scrubbed, etc., based on the key pattern stored in the bitmap, by loading the bitmap into memory from LIB.BITMAP. The <EOF1> loop would only have to be replaced with

```
   DO X=0 TO BITOBS-1;
      SET LIB.BITMAP NOBS=BITOBS;
      BITMAP(X) = BYTE8;
   END;
```

There are a few notable differences between straight key-indexing and bitmapping:

- Bitmapping expands the allowable key range 53 to 56 times, depending on the OS, given the same memory resource.
- A bitmap cannot be used in a non-contrived manner for dragging lookup satellites through memory into MATCH.
- An unsuccessful search occurs slightly faster than successful.

However, the similarities are more pronounced:

- The size of a bitmap is dictated solely by the overall key range and completely independent of the number of lookup keys.
- Neither file has to be sorted.

- Duplicate lookup keys are eliminated automatically.
- The speed of search does not depend on the number of keys.

Because of its relatively wide key range and pure direct addressing nature, bitmapping operates with incredible speed in a niche, no other searching method can touch. Imagine SMALL file with 50 million 8-digit keys (hardly small but let us stick with the name), and LARGE with mere 100 million records – figures not unusual in Data Warehousing today. Sorting either one for MERGE is not exactly painless, especially if L_SAT tail is long. Storing the lookup keys in a format is practically hopeless, as already at 10 million keys, it usually tops 600 MB. Key-indexing would consume about 800 MB. A hash table (described later) would need at least 400 MB. On the other hand, a bitmap can be safely compiled with 120 million bits of temporary array storage in the worst case scenario (M=53); that is, it can be easily swallowed by mere 15 MB of memory! If we decide to store the bitmap on disk, it will take about 1.9 million 1-column rows; loading such a file into an array is a matter of several seconds. Moreover, the lookup speed and memory usage will remain exactly the same, no matter whether we have 100, 100 thousand, or 100 million keys to search.

Thus, the bitmapping niche can be defined as "no-matter-how-many-short-keys". But how can we capitalize on direct addressing in the case of keys with a huge, say 16-digit, range, that neither key-indexing or bitmapping can accommodate? Welcome to *hashing*!

HASHING

In order to fix the main drawback of key-indexing, bitmapping changes nothing principally – it simply expands the workable universe of keys about 53+ times by using memory more efficiently. Hashing approaches the problem quite differently: It drops the requirement of a separate slot for each possible key *and* allows some amount of comparisons between keys. A simple example might be the easiest way of making the idea transparent. Let us suppose that SMALL contains just ten 3-digit keys:

```
185 971 400 260 922 970 543 532 050 067
```

To use key-indexing we would have to allocate a table sized as [0:999] and map each key to the node corresponding to its value. Out of 1000 table nodes, only 10 end up occupied while the rest are simply wasted! Can we get away with a smaller table of a reasonable size, *only slightly larger than the number of keys*, and still be able to take advantage of direct addressing? Let us choose some number HSIZE somewhat greater than the number of keys N_SMALL, for instance 17, and allocate an array sized as HKEY(0:17). Let us agree to call the array HKEY the *hash table*, HSIZE - the *hash table size*, and the ratio N_SMALL/HSIZE - the *load factor*. Thus, the load factor shows the number of lookup keys relative to the total number of nodes in the hash table, in other words, how *sparse* the hash table is. In our example, the load factor equals 0.588, that is, the hash table is about 41 percent sparse.

Imagine some fast function H(KEY) taking a key as an argument and returning an address into HKEY, unique to each key supplied, so that H(KEY) maps each key to its own location one-to-one. Were such *perfect hash function* available, we would only have to plug it in the code for key-indexed search. However, although perfect hash functions *are* possible, they are quite difficult to discover, and once found, can only be used for the same set of keys.

A lot more flexible method can be obtained by giving up the requirement of one-to-one correspondence between the keys and table addresses, and letting H(KEY) *map two distinct keys to the same location* in HKEY. However, if two different keys are sent to the same node, a phenomenon termed a *collision* occurs, and we must invoke some *collision resolution policy* in order to tell the keys apart in the process of inserting a key or searching for one.

Thus, we arrive at the *core concept behind hashing*. If the hash function H(KEY) is good enough to *map only a few keys* to a particular hash address H, we can adopt the following strategy:

1. Use H(KEY) to *hash* KEY to some address H in the table.
2. If the address H is empty, the search is unsuccessful.
3. If the address is occupied, search the keys residing at H *sequentially*.

Clearly, hashing is a typical *hybrid algorithm*: It combines direct addressing with *sequential search*, a comparison-based method. The *average* number of keys mapping to a hash address equals N_SMALL/HSIZE, the load factor. If the hash table is not full and the keys are spread uniformly, the average number of key comparisons required to find or reject a key will thus be less than 1! Also, searching for a key should be the faster, the sparser the table is. So, in order to use of a hash table, we have to:

1. Choose a proper hash function H(KEY).
2. Find an efficient way to resolve collisions.

Before trying to formulate the requirements for a good hash function let us see first what it *should not be*. On one extreme, if a function is lightning fast but maps the majority of keys to the same hash address, it defies the very purpose of distributing keys among different addresses since then we would have to search all these keys attached to the same node sequentially! Hence, it is paramount that a good hash function should map the keys evenly across all hash table nodes without burdening some addresses with huge clusters and leaving the rest of them empty. On the other extreme, if a function maps the keys extremely uniformly but is computed slowly, it is of no good use, either, since in this case, direct addressing itself will become a bottleneck. Now we can have some rational requirements a *good hash function* should satisfy:

1. Its computation should be as fast as possible.
2. It should distribute the keys uniformly to minimize collisions.
3. It must return addresses in the range [0:HSIZE].

There is a number of mapping methods conforming to these requirements [1]. We will discuss the simplest technique called *the division method.* It makes use of the remainder modulo:

```
H = MOD (KEY, HSIZE).
```

It certainly fits #3, since this function *always* returns an integer in the range from 0 to HSIZE-1. It also satisfies #1, for although it incorporates a division, its computation is quite fast. However, to satisfy #2, the value for HSIZE must be chosen rather carefully. Suppose that R is the radix of the character set, and X and Y - small integers. It can be proven [1] that if HSIZE is a *prime number* such that R**X is not equal to Y, the keys will be spread satisfactorily evenly. Let us see how it would work for our sample set of the ten keys. If we choose the target load factor as 0.625 and divide it into the number of keys, we will obtain 16. The first prime greater or equal to 16 is 17, so let us select it as HSIZE. The actual load factor is now 10/17 = 0.588. We should therefore allocate the hash table as

```
ARRAY HKEY (0:17) _TEMPORARY_;
```

To obtain the hash addresses, we divide each key by HSIZE=17, compute the remainder H and insert KEY into H-th slot:

```
00     .    .    .
01   970    .    .
02   971    .    .
03     .    .    .
04   922    .    .
05   260  532    .
```

```
06     .    .    .
07     .    .    .
08     .    .    .
09   400    .    .
10     .    .    .
11     .    .    .
12     .    .    .
13     .    .    .
14     .    .    .
15   185    .    .
16   543  050  067
17     .    .    .
```

The keys 970, 971, 922, 400, and 185 map uniquely. The keys 260 and 532 produce a single collision at the address 05, and the keys 543, 050, and 067 result in a double collision in the node 16. If this table is to be stored in memory and searched, the collisions at the locations 05 and 16 are to be resolved. Instead of finding HSIZE by hand or from a table of primes, it can be dynamically computed and stored in a macro variable in order to size the table:

```
%LET LOAD = 0.5;
DATA _NULL_;
   P = CEIL(SIZE/&LOAD);
   DO UNTIL (J = U + 1);
      P ++ 1;
      U = CEIL(SQRT(P));
      DO J=2 TO U UNTIL (MOD(P,J)=0); END;
   END;
   CALL SYMPUT('HSIZE',LEFT(PUT(P,BEST.)));
   STOP;
RUN;
```

As we already know, selecting a decent hash function is just part of the deal. No matter how good the function is, it is *practically guaranteed* that some addresses will be burdened with more than one key. Therefore, we have to devise a method to resolve the collisions. This is another point where hashing deviates from key-indexing and bitmapping, where keys are not stored in the table. With hashing, the keys themselves have to reside in the table because they will have to be compared to a search key (unless it hashes to an empty node). Various *collision resolution policies* differ in the ways colliding keys are stored, linked as related to the same hash address, and traversed.

1. Separate Chaining

One way of resolving collisions naturally suggests itself once we look at the distribution of our 10 keys among the addresses of the 17-node hash table shown in the previous section. The keys hashing to the same address form a visible "chain". Using such chains to resolve collisions is thus logically called *separate chaining*.

There are two ways such chains can be utilized. First, they can be stored *outside the table* by placing them in the occurrences of a two-dimensional array. A huge drawback of this method is poor memory utilization. If we have 100,000 keys in SMALL and a single "bad" address colliding 10 keys, we will be forced to create a 2-dimensional array sized as (0:10, 0:100000) to resolve the collisions. Even with the load factor 1, it requires 10 times the memory the keys would occupy by themselves. On the positive side, the 2-dimensional chaining is quite fast, and it can work with load factors greater than 1. So, if good memory utilization is not too paramount, the method could be well recommended. (Feel free to contact the author for the details of implementation.)

However, the idea of chaining can be exploited in a much neater fashion! Once the philosophy of allocating the storage for colliding keys is changed from sequential to *linked* , we arrive at an extremely elegant collision resolution policy, both very fast and reasonably memory-efficient.

2. Coalesced Chaining

The *core idea* here is to place all chains of colliding keys into the hash table itself and combine the keys hashing to the same node into a linked list with the head residing at the colliding address. Setting the last link of each chain to null designates the end of the chain, thus helping us tell where to stop when the list is traversed serially. Since the linked lists are thus allowed to overlap in the hash table sharing the same storage locations, this approach is termed *coalesced list chaining*, or *coalesced chaining* for short. To make it possible, all we need is a numeric array of link items, say, LINK, parallel to the hash table. Note that in order for this method to work, *at least one entry in the table must be empty.* Otherwise if the table is full, there would be no room to place the final null link needed to terminate the traversing loop. Since a table allocated as (0:HSIZE) has HSIZE+1 entries, but the modulo-based hash function only addresses HSIZE nodes from 0 to HSIZE-1, this requirement will always be satisfied. Let us agree to *always leave the address 0 empty* by hashing a key as

```
MOD (KEY, HSIZE) + 1.
```

That is, if KEY modulo HSIZE is 05, it will map to the address 06. As stated above, we now need two parallel arrays, one for the hash table itself and another one to hold the links:

```
ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
ARRAY LINK (0:&HSIZE) _TEMPORARY_;
```

Now we are ready to spell a detailed plan of loading the table:

STEP 1. Set a counter variable R to the top address: R=HSIZE.
STEP 2. Hash: H=MOD(KEY,HSIZE) + 1.
STEP 3. If LINK(H)= ., the node is empty, no list is attached to it. Go to step 7 and insert the key.
A. Otherwise *traverse the chain* to find if the key is already in the table. If KEY=HKEY(H) the key is duplicate. Get the next key and return to step 2.
B. Else If HKEY(H) is not 0, it is not the end of the list yet. Set H= HLINK(H) and repeat step from A.
STEP 4. Find an empty address closest to the top: Decrement R until LINK(R)= .
STEP 5. Store the key at this address: HKEY(R)=KEY.
STEP 6. Memorize where KEY actually belongs: LINK(H)=R; H=R.
STEP 7. Insert KEY into the address H and set its link to null: HKEY(H)=KEY; LINK(H)=0. Now the node is occupied.

Applied to our 10 sample keys and HSIZE=17, this process results in the following loaded table (colliding keys are shown in bold):

H	HLINK	HKEY
00	.	.
01	.	.
02	00	970
03	00	971
04	.	.
05	00	922
06	**15**	**260**
07	.	.
08	.	.
09	.	.
10	00	400
11	.	.
12	.	.
13	**00**	**067**
14	**13**	**050**
15	**00**	**532**
16	00	185
17	**14**	**543**

A brief look at table reveals how the colliding keys are linked. The keys 543, 050, and 067 all hash to the address 17. The first key of the chain, 543, must be stored at this address, and there it is. The link of the address 17 is not 0, hence, it is not the end of the list. Instead, LINK(17)=14. This is the node where the next key in the chain, 050, must reside, and it is there, indeed. But once again, the list must continue because the address 14 contains a non-zero link, LINK(14)=13. Finally, we find the key 067 in the node 13, and it is the last key colliding at the hash address 17, for LINK(04)=0.

Likewise, the keys 260 and 532 both hash to the address 06. Looking at this node we find that it is exactly where 260 has been inserted. However, LINK(06) points to the node 15 containing the key 532, the last key in this linked list since the node is marked by a null link, LINK(15)=0.

Contrary to the colliding keys, the keys hashing to their addresses uniquely bump in a null link at once: For example, the key 922 hashes to the address 05, with LINK(05)=0.

At this point, it should be clear how this table organization facilitates searching. Suppose that we need to look for KEY=051. It hashes to the address 01 where the link field LINK(01) is missing, that is, none of the keys in the table has ever hashed to this address. Hence, the key is not in the table. However, searching for KEY=047 that hashes to the address 14, we encounter a non-empty LINK(14)=13. Hence, more keys in the table have also hashed to this address, so the entire chain must be examined for the presence of 047. Since the key 050 in the node 14 does not match the search key, we have to look at the next key in the chain located at the address 13. The key 067 in the node 13 does not match the search key 047, either, and it is the end of the list since LINK(13)=0. Hence, 047 is not present in the table.

As an example of a successful search, let us try to find KEY= 050. It hashes to the address 17 with the key 543, different from 050. But it is not the end of story: LINK(17)=14 is not null telling that the next comparison should be made with HKEY(14) = 050. At this point, the search key is found, the list no longer needs to be traversed, and the process terminates successfully.

After this walk-through, it should not take a Certified SAS Programmer to schedule *hash searching:*

STEP 1. Hash: H = MOD(KEY,HSIZE) + 1.
STEP 2. If LINK(H) = . then search terminates unsuccessfully.
STEP 3. Traverse the list. If KEY = HKEY(H) then search terminates successfully.
STEP 4. Else examine LINK(H). If LINK(H) = 0 then KEY is not found. Stop.
STEP 5. Next link. Set H = LINK(H) and go to step 3.

Now we can finally give a solution to the matching problem using coalesced chain hashing. An additional array parallel to the hash table and links, HSAT, is used to pull the satellite from SMALL if KEY coming from LARGE is found. If we do not need S_SAT it may be omitted along with the corresponding instructions.

```
*** COALESCED LINKED LIST CHAIN HASHING ***;

DATA MATCH (KEEP=KEY S_SAT L_SAT);
   ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
   ARRAY LINK (0:&HSIZE) _TEMPORARY_;
   ARRAY HSAT (0:&HSIZE) _TEMPORARY_;
   R = &HSIZE;
   DO UNTIL (EOF1);
      SET SMALL END=EOF1;
      H = MOD(KEY,&HSIZE) + 1;
      FOUND = 0;
      IF LINK(H) > . THEN DO;
         LINK TRAVERSE;
         IF FOUND THEN CONTINUE;
         DO WHILE (LINK(R) > .);
            R +-1;
         END;
         LINK(H) = R;
         H       = R;
      END;
      HKEY(H) = KEY;
      HSAT(H) = S_SAT;
      LINK(H) = 0;
   END;
   DO UNTIL (EOF2);
      SET LARGE END=EOF2;
      FOUND = 0;
      H = MOD(KEY,&HSIZE) + 1;
      IF LINK(H) > . THEN LINK TRAVERSE;
      IF FOUND THEN DO;
         S_SAT = HSAT(H);
         OUTPUT;
      END;
   END;
   STOP;
   TRAVERSE:
      IF KEY = HKEY(H) THEN FOUND = 1;
      ELSE IF LINK(H) NE 0 THEN DO;
         H = LINK(H);
         GOTO TRAVERSE;
      END;
   RETURN;
RUN;
```

This code intentionally parallels the algorithm above, so one should not be surprised to find the GOTO. Those believing that "GOTO" and "structured programming" are antonyms may prefer to rewrite the TRAVERSE block (at the expense of an extra comparison) as

```
DO UNTIL (FOUND);
   IF HKEY(H) = KEY THEN FOUND = 1;
   ELSE IF LINK(H) = 0 THEN LEAVE;
   ELSE H = LINK(H);
END;
```

The routine has been tested using the process described in the Section "Benchmarking". The results of Table 1 confirm that:

1. The sparser the table, the faster the search, but it consumes proportionally more memory.
2. Lookup time does not depend on the number of keys.
3. Hashing runs a bit slower than key-indexing but is much more memory efficient.
4. Just like key-indexing and bitmapping, hashing needs neither sorting nor removing duplicates.

3. Open Addressing

Chain hashing is just one possible way to resolve collisions. It performs very well and is not too sensitive to the load factor. However, it requires an extra array to hold links, as large as the hash table itself, and if SMALL is actually large, the additional memory burden can be quite significant. In a different class of collision resolution policies, *open addressing,* memory utilization is improved by doing away with the links altogether.

The main idea behind open addressing can be described as follows. Just like in coalesced hashing, keys are stored in the hash table itself. If we try loading a key and it produces a collision, we simply step down the table one node at a time until an empty location is found, and place the colliding key there. If the bottom of the table is reached, we wrap to its top and continue the process. Then at the time of search, if a key hashes to an occupied node, we can use the same exact path until either a match or empty node is encountered, in which case the key is not in the table.

However, this technique called *linear probing* performs quite miserably if the table gets more than half full, which is due to the phenomenon termed *primary clustering*. Since we fill out the very first empty location we come across, the groups of adjacent

occupied addresses tend to aggregate forming clusters. Hence, if the table is not quite sparse, we will eventually have to travel through the entire table before finding an empty location to either insert a key or stop in the case of an unsuccessful search.

To alleviate the problem, linear probing can be replaced by stepping through the table more than one node at a time in a pseudo-random manner, deterministic in the sense that for a given key hashing to a given address, the sequence of probes is the same each time it is invoked. That is, the size of the step called *probe decrement* should randomly depend on the key rather than being always 1. Also, the process *must* be able to examine all nodes in the table in an unbroken wrap-around cycle. Random sequencing virtually eliminates primary clustering by making it unlikely for a new empty location to be chosen next to an occupied one.

One way to implement this plan is to hash a key, after it has been hashed once, again, this time using a slightly different hash function to obtain a probe decrement for the key. Such a policy of resolving collisions called *double hashing* has the overhead of the second hashing, but leads to separate probing sequences for different keys spreading empty and occupied nodes evenly across the table. The question is, how to select the second hash function to satisfy the requirements formulated above? It follows from the number theory that if the probe decrement C and the hash table size HSIZE are *relatively prime*, then if, starting at some address, we step through the table C nodes at a time in a wrap-around manner, each node will be covered exactly once. For instance, if HSIZE is prime, C can be any integer between 1 and HSIZE-1 inclusive, or if HSIZE is a power of 2, C can be any odd integer between 1 and log(HSIZE)-1. In the program below solving our sample file match problem by means of double hashing, HSIZE is left prime and C is calculated as

```
C = 1 + MOD(KEY, HSIZE-2).
```

This has been proven to work satisfactorily in most cases.

```
*** DOUBLE HASHING TABLE LOAD AND SEARCH ***;

DATA MATCH (KEEP=KEY S_SAT L_SAT);
ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
ARRAY HSAT (0:&HSIZE) _TEMPORARY_;
DO UNTIL (EOF1);
       SET SMALL END=EOF1;
       H = MOD(KEY,&HSIZE);
       IF      HKEY(H) EQ . THEN LINK INSERT;
       ELSE IF HKEY(H) NE KEY THEN DO;
          C = 1 + MOD(KEY,&HSIZE-2);
          IPROBE:
          H +- C;
          IF H LT 0 THEN H ++ &HSIZE;
          IF HKEY(H) EQ . THEN LINK INSERT;
          ELSE IF HKEY(H) NE KEY THEN GOTO IPROBE;
       END;
   END;
   DO UNTIL (EOF2);
       SET LARGE END=EOF2;
       H = MOD(KEY,&HSIZE);
       IF      HKEY(H) EQ .   THEN FOUND = 0;
       ELSE IF HKEY(H) EQ KEY THEN FOUND = 1;
       ELSE DO;
          C = 1 + MOD(KEY,&HSIZE-2);
          SPROBE:
          H +- C;
          IF H LT 0 THEN H ++ &HSIZE;
          IF      HKEY (H) EQ .   THEN FOUND = 0;
          ELSE IF HKEY (H) EQ KEY THEN FOUND = 1;
          ELSE GOTO SPROBE;
       END;
       IF NOT FOUND THEN CONTINUE;
       S_SAT = HSAT(H); OUTPUT;
   END;
   STOP;
   INSERT: HKEY(H) = KEY; HSAT(H) = S_SAT;
RUN;
```

Once again, those revering "gotoless" programming are encouraged to make the code "prettier" without hindering performance. If the satellites are not needed the array HSAT and instructions referring to it should be omitted.

Let us take a look at the performance Table 1. With a 50 per cent sparse table, double hashing runs just a bit slower than coalesced chaining with 20 percent sparsity but uses less memory. So, double hashing is quite fast; it even loads an equally sparse table a tad faster than chaining. That a method based on stepping through the table before an empty node is found works so well, may seem surprising; however, this is a direct result of the double hashing probing methodology. In fact, independent experiments show that if the table is less than half full, double hashing makes on the average no more than 2 comparisons per miss, and no more than 1.3 comparisons per hit.

HASHING WITH NON-INTEGER KEYS

As the tests show, hashing performs admirably by any account regardless of the collision resolution policy used. To this end, however, we have only eliminated the range restriction of key-indexing and bitmapping, since the assumption of the keys being non-negative integers has been used in both implementations. Fortunately, with hashing, the nature of the keys is not critical because in its final stage, it is strictly comparison-based. Both hashes and traversals are used merely to minimize the number of comparisons necessary to carry out a search, yet the final decision (if a hash address is not empty) is made by comparing some keys in the table to the search key. Therefore, in order to be able to operate on keys of any type, we only have to figure out how to hash a key if it is not a positive integer. For the hash function to remain uniform and fast, it is critical to adhere to the following simple rules of efficient hashing:

- In order to minimize collisions, hashing a key should involve as many key characters as possible.
- String operations and conversions must be minimized.

Let us consider a number of distinct situations.

1. Fractional Signed Keys

In this case, we can simply rescale each key before hashing it by multiplying it by a suitable integer constant and adding another constant to the result if necessary. For instance, if our keys are in the decimal form X.Y, multiplying each key by Y would suffice. If, in addition, they can be negative, we would simply add an integer Z known to exceed the largest absolute value a negative key can assume. So, the entire change to the programs above needed to accommodate fractional signed keys would be using

```
MOD ( KEY*Y + Z, HSIZE)
```

in the hashing formulae instead of the straight modulo. It will not cause any noticeable deterioration in performance, since in SAS this kind of computation is very fast.

2. Digital Strings

It is a simple matter of using the INPUT function and an appropriate numeric informat. For example, if the keys were 16-digit account numbers stored in a character variable, we could simply choose

```
MOD (INPUT(KEY,16.), HSIZE)
```

as our hash function. Accordingly, the hash table itself would now have to be organized as a *character* array:

```
ARRAY HKEY(0:&HSIZE) $16 _TEMPORARY_;
```

3. Alphanumeric Keys

Numerous techniques of hashing character keys have been developed [2, 3, 4]. Almost all of them are based on breaking a character key apart and then involving the individual bytes into a sort of computation resulting in an integer in the range [0:HSIZE-1]. Some of these methods (like universal hashing) actually guarantee to hash *any* input evenly. However, they are based on the assumption that the process of extracting individual bytes from a string is very fast. Unfortunately, this is not fast in SAS. We would be much better off converting a character string to an integer in a single shot, and PIBw. informat is just the tool:

```
MOD (INPUT(LEFT(KEY),PIBw.).
```

Generally, the wider is the informat, the better, but selecting it too wide may result in a large integer rendering the result produced by MOD function incorrect. Experimentally, it has been found that under NT, the maximum allowable width, 8, works fine. Under OS/390, it should not exceed 7, and under HP-UNIX, 6 is the limit. The method has an extra advantage of avoiding the slow SUBSTR function. Note that we use PIBw. instead of S370FPIBw. because it is faster, and in hashing the order of bytes does not matter: We only want to use as many key bytes as possible to minimize collisions. The LEFT function may help by squeezing leading blanks to the right. If a key is longer than the practical informat width, the trick still works, provided that the input characters distinguish the keys well. However, if they have a good chance of being identical, they can be selected from a different portion of the key.

4. Composite Keys

This situation arises quite often. A natural inclination is to concatenate the components and hash the result. Principally, there is nothing wrong about it; however, there are two pitfalls. First, in the context of hashing, where computing a hash function fast is paramount, concatenation is slow. Second, the components may concatenate into an integer lying beoynd SAS integer precision.

Consider a situation when records are uniquely identified by two numeric variables, a 16-digit ID and 9-digit MEM, while particular ID can point to multiple accounts. Concatenating the keys as ID || MEM and hashing the result would have the effect of scrambling the entire MEM. All keys with the same ID would then hash to the same address regardless of MEM and lead to multiple collisions and horrible performance. Luckily, it can be avoided since we are not interested in the value of the key itself, but only in its remainder modulo HSIZE. Hence, Horner's algorithm can be used to hash the components separately and combine the results in the final address. The outcome is the same as if we had enough integer precision to store the combined key accurately. For the ID and MEM, it means that the hash function can be computed in the form:

```
MOD(MOD(ID,HSIZE)*1E9 + MEM, HSIZE) .
```

If the partial keys are longer or the range is wider, they can be split further, and Horner's rule can be applied to the components. Certainly, the key parts must be kept in parallel hash arrays, loaded and tested separately. If, for instance, ID and MEM were hashed by chaining, the HKEY declaration would have to be replaced with

```
ARRAY HID  (0:&HSIZE) _TEMPORARY_;
ARRAY HMEM (0:&HSIZE) _TEMPORARY_;
```

and HKEY(H)=KEY would become

```
HID  (H) = ID;
HMEM (H) = MEMNO;
```

while in the TRAVERSE subroutine, the instruction KEY=HKEY(H) would transform into the line

```
IF ID = HID(H) AND MEMNO = HMEM(H);
```

Similar modifications can be done in the case of double hashing.

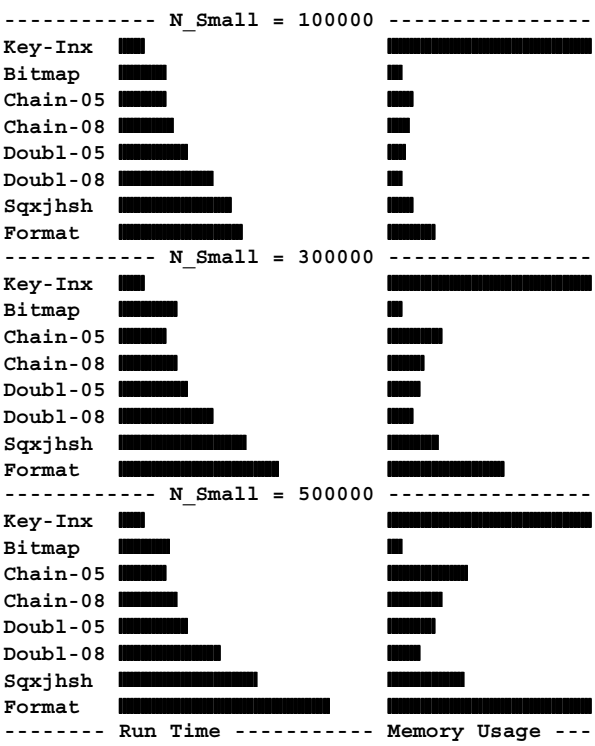
BENCHMARKING

Each technique presented above is a "private detective" that operates best in its own "area of expertise" defined by the number of lookup keys and key range. To compare them to each other and two SAS-supplied methods, SMALL and LARGE were created with random integer keys in [0:8E6] range where all methods could work within the system imposed memory limit of 70 MB. To include bitmapping into the comparison group, the satellite S_SAT was omitted from SMALL in all methods tested. The input was prepared in such a way that hits and misses were equally likely to occur. LARGE with fixed N_LARGE=2E6 was then matched against SMALL varying the number of records in S/390 G5 R36 Enterprise Server batch running SAS Version 6.09E. For key-indexing, bitmapping, and hashing LOAD represents the time needed to load a table from SMALL (<EOF1> loop). In the case of formatting, LOAD is the time required to unduplicate SMALL and compile the format. For MERGE, it is the time needed to sort the files. Suffices "05" and "08" indicate the load factors used in the test. LOAD, SEARCH, and TOTAL are given in CPU seconds, MEMORY – in KB.

Table 1. Benchmarking.

N_Small	Method	Load	Search	Run	Memory
100,000	Key-Inx	0.42	12.18	12.60	65261
	Bitmap	0.36	22.64	23.00	3925
	Chain-05	0.31	21.78	22.09	5997
	Chain-08	0.34	26.28	26.62	4829
	Doubl-05	0.28	32.78	33.06	4445
	Doubl-08	0.33	47.91	48.24	3961
	Sqxjhsh	0.00	52.66	52.66	5881
	Format	6.27	51.92	58.19	10866
	Merge	19.74	46.16	65.90	3276
300,000	Key-Inx	0.67	12.07	12.74	65261
	Bitmap	0.99	26.17	27.16	3925
	Chain-05	0.87	21.47	22.34	12093
	Chain-08	0.95	26.03	26.98	8637
	Doubl-05	0.80	31.24	32.04	7493
	Doubl-08	0.97	47.03	48.00	5769
	Sqxjhsh	0.00	58.38	58.38	11401
	Format	18.71	55.19	73.90	26199
	Merge	20.91	46.63	67.54	3267
500,000	Key-Inx	0.92	12.09	13.01	65261
	Bitmap	1.59	26.56	28.15	3925
	Chain-05	1.37	21.60	22.97	18033
	Chain-08	1.53	25.30	26.83	12357
	Doubl-05	1.29	31.90	33.19	10465
	Doubl-08	1.57	42.17	43.74	7625
	Sqxjhsh	0.00	64.49	64.49	16921
	Format	29.77	67.26	97.03	57289
	Merge	21.59	47.27	68.85	3267

The series of charts below gives a more transparent idea about the performance of the different lookup techniques:



Note that the run times exhibited by key-indexing, bitmapping, and hashing, in agreement with their direct-addressing nature, are virtually independent from the number of lookup keys. For key-indexing and bitmapping, memory usage is always fixed since the number of keys has no effect on the universe of keys they embrace. Hashing uses memory strictly proportional to the number of keys in the table and sparsity of the table.

APPLICATIONS

1. Subsetting

From the test results, it is evident that when it comes to subsetting, direct-addressing methods result in lookup speeds unmatched even by methods written in the underlying software and specifically designed for searching. As an icing on the cake, hashing is significantly more memory-efficient than formatting and SQL. The latter is extremely important when the number of keys in SMALL grows beyond a couple of million. Hash memory is strictly proportional to the number of lookup keys and can be accurately estimated beforehand. Contrary to that, the amount of memory used by formats or SQL seems to grow uncontrollably after a certain threshold has been reached. The author has been exposed to a number of situations not unlike the following truest story.

Once upon a time, in a Data WarePalace, the King summoned two Knights, Format and Hash, handed each 400 MB of memory, two files, SMALL and LARGE, and ordered to match them. At 5 million records in SMALL, Format took about 10 CPU minutes to compile using all 400 MB, and then 25 CPU minutes to match. Hash finished the job in 10 CPU minutes with 240 MB to spare. At 10 million records, Format ran out of memory; Hash completed the task in 11 minutes still with 80 MB left. Guess who married the Princess? (The correct answer is Format: Hash was burned at the stake for sorcery.)

On a different note, we should exercise caution dragging satellites from SMALL through the memory. If there is more than one satellite, one may be tempted to create a separate parallel satellite array for each, but this is not always the right thing to do since temporary SAS arrays are allocated in 8-byte multiples per item. If we have four 8-byte character satellites, a separate array can be declared as $8 for each with 100 per cent memory utilization. However, if we have four 2-byte satellites and create 4 parallel arrays $2 each, it will waste gobs of memory, for SAS will allocate the arrays with 8 bytes per item, anyway. So, in this case, we will be much better off memory-wise allocating one array as $8, stringing the satellites together in the load phase, and unstringing them into separate variables prior to output.

2. DATA Step Data Dictionaries

Let us take a look at direct addressing from a different standpoint. A key-indexed, bitmap, or hash table can be viewed as *some abstract data type* (*ADT*) in memory allowing to efficiently perform certain operations on its *entries* containing keys and maybe satellites. There are two operations we have learned in the process of solving our sample problem: *Insert and search*. Many kinds of ADTs other than a directly addressed table can facilitate these operations: A plain sequentially searched array, sorted array, AVL tree, just to name a few. The difference between ADTs lies in the time it takes to insert or search for a key. For example, a plain array requires O(1), i.e. constant time, independent from the number of entries N, to insert a new entry - we simply append it to the right. On the other hand, searching occurs in O(N) time, i.e. is proportional to N. The sorted array ADT requires O(N) time to insert an entry because it is necessary to shift a number of items proportional to N to free a node keeping the table sorted. In exchange, searching an ordered array occurs only in O(log(N)), or even O(loglog(N)) time. An AVL tree performs both operations in O(log(N)) time in the worst case scenario.

In comparison, direct-addressing methods have the clear advantage of supporting both insertion and search in constant time O(1), because, as we have seen, it does not depend on the number of keys in the table. This property makes them ideal for implementing dynamic DATA step data dictionaries. Imagine that in the course of a DATA step, we need to maintain the history of key values as we go, and at different points in the program, answer questions like:

1. Has the current key already been used before?
2. If not, how to rapidly memorize it to be able to answer question 1 momentarily?
3. How to replace a stored key with another one?
4. If key is no longer needed, how to delete it?

If the nature of the keys makes key-indexing applicable, there is no better tool for the job, as all actions above are performed in O(1) time and do not get any simpler:

1. See if the KEY-th node contains a missing value.
2. Move 1 into the KEY-th node.
3. If the KEY-th node is busy, do nothing, else move 1 there.
4. Move a missing value into the KEY-th node.

If the key range is too wide for key-indexing, bitmapping might be just the ticket; otherwise we can organize a hash table using either of the collision resolution policies given in the text. In both programs, the body of the <EOF1> loop constitutes a ready-to-go combined hash search-and-insertion. That answers questions 1 and 2, or 1 and 3. The <EOF2> loop is a pure hash search thus providing an answers to question 1 standing alone.

A practical application of these principles immediately coming to mind is obtaining frequency counts in the case of a huge number of distinct levels of a categorical variable, when FREQ or SUMMARY either run out of memory or take too long to run. To compute frequencies without sorting, we must be able to maintain a table in memory allowing to immediately locate the value coming with the

next record and add a unity to its count. The following question was asked in SAS-L: "I have an unsorted dataset with 100 million records. It has a numeric integer variable FLDR_ID in -500000 to 500000 range. How to create a file with frequencies, cumulative frequencies, percents and cumulative percents for all values of FLDR_ID given the 50 MB RAM limit?" Below is a key-indexing solution Ian Whitlock and the author offered independently:

```
DATA DS_FREQ (KEEP=FLDR_ID FREQ CFREQ PCNT CPCNT);
   ARRAY F (-500000:500000) _TEMPORARY_;
   DO UNTIL(END);
      SET IDS END=END NOBS=NOBS;
      IF FLDR_ID = . THEN FLDR_ID = LBOUND(F);
      F(FLDR_ID) ++ 1;
   END;
   PTOT = 1/NOBS * 100;
   DO I=LBOUND(F) TO HBOUND(F);
      IF F(I) = . THEN CONTINUE;
      FREQ =  F(I);
      CFREQ ++ FREQ;
      PCNT  = FREQ * PTOT;
      CPCNT ++ PCNT;
      IF I > LBOUND(F) THEN FLDR_ID = I;
      ELSE FLDR_ID = .;
      OUTPUT;
   END;
RUN;
```

This simple program uses 10.5 MB of memory and runs *at least an order of magnitude* faster than either FREQ or SUMMARY.

3. Sortless Unduplication

Usually duplicates are removed as a side effect of PROC SORT. However, if sorting is not the main purpose, the computer has to do a lot of heavy-duty work for no reason. PROC SQL with DISTINCT is "pure" but runs rather sluggishly.

Discussing hashing, we saw that in the process of loading a key-indexed, bitmap, or hash table, duplicates coming from SMALL are removed automatically by the search-and-insert routine embedded in the <EOF1> loop. Thus, it can be used to remove duplicates directly without sorting, provided that there is enough memory for a table to hold all keys. All we have to do is load the hash table and dump its contents skipping empty nodes. For instance, by selecting double hashing, omitting L_SAT from the KEEP list, and replacing the <EOF2> loop with

```
DO H=0 TO &HSIZE;
   IF HKEY(H) = . THEN CONTINUE;
   KEY   = HKEY(H);
   S_SAT = HSAT(H);
   OUTPUT;
END;
```

we will obtain the output filel containing no duplicates.

CONCLUSION

Key-indexing and bitmapping are in-memory searching techniques strictly based on direct addressing into array elements or their bits. Their area of applicability is limited to keys falling in a limited integer range defined by the available memory resources. However, when applicable, these methods exhibit unmatched performance; their implementations are straightforward. *Hashing* helps direct addressing work on keys of any type and range by bringing serial search and collision resolution policies into the equation. A bit slower that pure direct addressing, hashing searches times faster than SAS formats and SQL, and uses significantly less memory. Data Warehousing is one of the widest fields where the unmatched speed and efficiency of direct-addressing methods can be fully utilized. Compared to "traditional" techniques, they can successfully supplant formats in eliminating costly table joins, and accelerate the processes of data extraction, scrubbing, and validation, based on a large predetermined set of keys, many times. The larger the data, the bigger advantage direct addressing offers. Finally, direct addressing searching methods are just smart and intent "private detectives" that can be used at no extra cost by any Decision Support or Data Warehouse SAS programmer interested in delivering data to the user in a speedy manner given restricted memory and disk resources.

Key-indexing, bitmapping, and hashing are cool. They allow operating in the niches where "standard" approaches either run out of memory or take a frustrating time to run. The author encourages other SAS users to use these tools, modify them, tweak them, improve the code, and discover new areas of application. Karsten M. Self wrote once after having tried hashing in a real-world application: "Hash rocks, Dude!" Hopefully, if you also try direct addressing in practice, you will hardly disagree.

SAS is a registered trademark or trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

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, Parts 1-4.
4. T. A. Standish. Data Structures, Algorithms and Software Principles in C.

ACKNOWLEDGEMENTS

Thanks to Karsten M. Self, Ian Whitlock, F. Joseph Kelley, and Sigurd Hermansen for their enthusiastic support of direct-addressing methods in SAS, valuable discussions full of ideas, wit, and vigor, and giving the author an opportunity to apply the techniques to solve practical problems. Thanks to Greg Barnes Nelson for his kind invitation to present the topic in the Data Warehousing Section. The author gratefully acknowledges the contribution of the individuals who have, directly or indirectly, encouraged the author and supported his efforts of making direct addressing an accepted and practically used DATA step philosophy:

Doris H. Bogar
Michael V. Dorfman
Eugenia P. Kravchenko
Victor P. Dorfman
Vera Voloshin
Vladimir A. Kirillov
Yuri Katsnelson
Gennady Taratut
Jane King
Jay Melesky
Bob Abelsor
Ashiru Babatunde
Peter Crawford
Colin Earle
Christoph Edel
Ronald J. Fehd
Paul Gorell
Steven Kleiman
Alex V. Martchenko
Tom Mendicino
Diana Noble
Gerard Pauline
Ray Pass
Michael A. Raithel
Mike Rhoads
Dianne Rhodes
Don Stanley
Mark Terjeson
William W. Viergever
John Whittington
Thomas Zicafoose
Shiling Zhang

David Pider

AUTOR CONTACT INFORMATION

Paul M. Dorfman
10023 Belle Rive Blvd. 817
Jacksonville, FL 32256
(904) 564-1931 (h)
(904) 954-8533 (o)
sashole@mediaone.net