Paper 128-26

QUICK DISK TABLE LOOKUP VIA HYBRID INDEXING INTO A DIRECTLY ADDRESSED SAS® DATA SET

Paul M. Dorfman
CitiCorp AT&T Universal Card, Jacksonville, FL

ABSTRACT

When a large, disk-resident, periodically refreshed lookup table has to be frequently searched, a natural way of providing a quick access to the data is to store the table as a SAS data set with a SAS index built on the search key. The speed of SAS index lookup depends on the nature of the key. More often than not, the key is not perfect for being indexed, i.e. discriminant, uniform, and ascending. Besides, even a 'good' indexed key may not be quick enough for some time-sensitive applications.

In this paper, it is proposed to structure the lookup file for rapid direct addressing via a hash index, which is 'good' regardless of the search key distribution: Each discrete index value points to about the same number of search keys. Such a cluster of keys is loaded into memory via a single direct read as a prehashed table, where the search key can be located or rejected immediately.

INTRODUCTION

This paper has originated from the practical necessity to accelerate the following process. A SAS subroutine described below is called from a variety of programs run under OS/390. Basically, it does the following:

1. Accepts a driver file with up to 1 million so-called shadow accounts from a vendor (not allowed to operate on real account numbers for data security reasons).
2. Replaces the shadow accounts with real accounts from a disk-resident lookup table. The table is a SAS data set, sorted and indexed by the shadow account, with upwards of 40 million rows.

The lookup file is refreshed overnight from a small, about 100,000 records, transaction file, by MODIFYing old real accounts with new ones in the records with matching keys. The refresh time does not really matter, and it is very short anyway.

The problem lies with the step 2. If the driver file has under 100,000 accounts, the lookup takes 7-10 CPU seconds to complete. As the number of the keys in the driver grow, the response time grows approximately linearly, and with 1,000,000 keys, it exceeds 1 CPU minute. For a program run infrequently, it is more than sufficiently fast. However, on a typical business day, the subroutine may be executed hundreds of times in batch (sometimes concurrently). Even worse, it is called by some online applications.

It was asked to research the possibility of reducing response time, particularly with large driver sizes, 2 to 3 times. The question therefore is: Is it possible to organize a lookup process that would search considerably faster than a SAS index working under favorable conditions?

PROBLEM PURIFIED

Formally, the problem boils down to the following. A file DRIVER (up to 1,000,000 records) contains a unique 16-digit integer numeric variable K (the shadow account number) serving as a key. Another file, LOOKUP (up to 40 million records), also has the unique key K juxtaposed to a unique 16-digit integer numeric variable S (the real account number). We need to replace K in DRIVER by the corresponding values of S in LOOKUP, based on the values of the common key K, as rapidly as possible. No particular limitations are imposed on the time required to organize a lookup structure, since it is done either once or very infrequently.

SAS INDEXING

The approach taken in the original application was to sort LOOKUP by K, build a SAS index on K, and let it do the job. It appears to be right on the money, for the conditions under which the index operates, are close to the ideal. Indeed, the indexed key is:

- Discriminant
- Uniformly distributed
- Sorted ascending

The code that originally loaded the lookup file (distilled for the purposes of this paper) is simple. A program, similar to one shown below, was run to load the lookup file, sort and index it. Note that there is the libref USER lurking behind the scenes and pointing to a permanent SAS library different from LOADLIB. So, all unprefixed data sets actually reside in the library associated with USER.

```
proc sort data=loadlib.lookup out=lookup nodupkey;
   by k;
run;
proc sql;
   create index k on lookup(k);
quit;
```

In the subroutine that replaces K in DRIVER with S from LOOKUP, one of two well-known methods are used. The first is a SQL join. After it has executed, the SAS log, prompted by msglevel=i and _method, confirms that the index has been used, indeed, to satisfy the query.

```
option msglevel=i;
proc sql _method;
   create table realacct as
   select l.s
   from lookup l, driver d
   where l.k = d.k
   ;
quit;
```

The second approach is to use SET with the KEY= option, making SAS use the index regardless of its desire:

```
data realacct (keep=s);
   set driver (keep=k);
   set lookup key=k;
   if _iorc_ = 0 then output;
   else _error_ = 0;
run;
```

This method runs about 20% faster than SQL. With 40 million lookup records and 100,000 keys in the driver, the step finishes in 8.28 CPU seconds. Given the amount of data to search, it should be considered quite remarkable. However, already with 500,000 keys, the response time increases to 34.15 CPU seconds, and with 1 million keys - to almost a minute. The question is, is there anything in Base SAS that can be done to keep the response under 10 seconds?

ALTERNATIVES APPROACHES

Under the circumstances, the SAS index is searched for a key much faster than a serial pass through the file would, dramatically reducing I/O traffic. However, no matter how efficient the index lookup algorithm is, it is still predominantly a disk search. Estimating extremely crudely that log2(NL) probes are necessary to find or reject a key stored in an index (where NL is the number of lookup keys), about 26 disk accesses will be necessary to complete the search for NL=40,000,000.

A lucrative alternative would be to store all the lookup keys in some kind of memory table, for example, a format. However, with tens of millions of keys, it is virtually impossible, even with the amounts of real storage (RAM) available nowadays: Tests have shown that already at the level of 10 million keys, formats require upwards of 600 MB.

Another option is to store the shadow and real accounts in a DB2 table indexed by K and use it as the search medium. This approach has actually been proposed and even reinforced by the claim that DB2 has a speed demon inside, making it 'an order of magnitude faster than SAS' for this kind of task. Later on in the performance section, we will see how much water the claim holds (for the time being, suffices it to say that after some testing, the proposal did not pass).

Finally, we can try a hybrid approach that would somehow marry disk (external) and memory (internal) searching methods. Conceivably, this may be achieved by splitting the lookup file into buckets (strata, partitions: not that the terminology really matters here) small enough to fit in memory completely. That is, while still having the entire file, due to its size, resident on disk, we can attempt to distribute its records between some M buckets in such a way that:

- A bucket is small enough to fit in real storage (RAM), and hence all keys belonging to it can be searched completely in memory.
- The bucket is large enough to embrace as many records, compete with the keys and satellites, as can be brought into memory via a single direct addressing instruction.

PLAN OF ATTACK

In other words, under such an organization, each bucket would belong to its own observation. In order for this to work, we need some concrete rule, or function, that would send a particular key to the particular bucket in the lookup file. Given a key to search, the function could then be used to locate the bucket (observation) the key belongs to, in order to read the observation into memory and search it there. As to the memory search, the data can be organized in each bucket in such a way that searching for a key will require no more than 2-3 key comparisons, irrespective of the number of keys in the bucket.

Now imagine that the keys in the driver file are tagged with the bucket numbers using the same distribution rule as the one applied to the keys in the lookup file. Then, if we re-group the driver file by the bucket number, each by-group will correspond to one, and only one, record in the stratified lookup file.

How would this scheme change the lookup philosophy compared to SAS indexing? With the latter, searching the index file for a key (associated with its own I/O traffic) would return a pointer to the proper record, and the record would be fetched. So, if there are ND keys in the DRIVER, and all of them are present in LOOKUP, ND direct internal read accesses would be needed. The stratified scheme having M buckets would perform no more than M direct explicit reads, because all driver keys mapping to the same bucket number could possibly be found only in the corresponding bucket of the (reshaped) lookup file. If H is a bucket number, a single POINT=H read would move the entire bucket into memory, where the driver keys from the current group could be rapidly searched.

Effectively, it would replace a 'long and narrow' lookup file, with a separate direct read for every key found in the index, by a 'short and wide' lookup file, with the maximum of M direct accesses to the lookup file, the rest of the search occurring completely in memory. As the latter can be programmed extremely efficiently, there is a hope that the 'short and wide' could outperform 'long and narrow' enough to deliver the required decrease in the run-time.

To find out if these speculations hold any water, we need to express them using the SAS language and test the concoction. However, first it is necessary to decide on the reasonable method of distributing the keys among the buckets.

DIVIDE . . .

Intuitively, it is apparent that the more uniformly the keys will be spread, the better. We certainly do not need a stratified lookup file with a couple of records containing the majority of keys and the rest having none, for it would defy the entire idea of distribution. Moreover, there is another advantage of mapping the keys to their buckets evenly: It makes the distribution of the keys themselves (whose skewness can be detrimental to SAS index performance) unimportant.

One of the easiest ways to obtain an even distribution is dividing the key (or its numeric representation, if the key is character) by a prime number and computing the remainder:

```
h = mod (k, M) + 1;
```

The added unity is only needed to shift the range of H from [0:M-1] to [1:M] and thus make it possible to access the bucketed lookup file via POINT=H in the future.

Now it is time to decide on the value of M, which will become the number of records in the final lookup file and thus define its 'aspect ratio', directly affecting the number of keys per bucket/observation. At first glance, it seems that the more keys per bucket (the lower M), the better, since fewer buckets would result in fewer direct accesses, shifting the balance towards searching in memory. However, as the number of variables per observation grows, so does the compilation time. Usually negligible, in this particular task, the compilation time may become part of the

game: With 20,000 variables in PDV, say, it may take seconds to compile a step, while we are trying to shave seconds off the combined compilation and execution time. Keeping no more than 1,000-2,000 variables per record, split equally between the shadow and real accounts, makes this concern insignificant. Extensive testing has shown that throughout the range of 150 to 5000 keys per bucket, the actual number has a very little impact on the run time. Choosing the aspect ratio rather lean, with 150-1000 keys per lookup observation, might be even beneficial to different users searching the file concurrently.

The importance of M being prime and not too close to a power of 2 is paramount, because it guarantees a fairly even distribution of NL keys across M buckets. To simplify things, we can first pick the number of key items per bucket, let us call it IPR, tentatively, and then use a little fast SAS program to compute the first prime number greater than NL/IPR:

```
%let ipr =  150 ; *** items per lookup record;

data _null_;
   do M=ceil(n/&ipr) by 1 until (j=up+1);
      up = ceil(sqrt(M));
      do j=2 to up until (not mod(M,j)); end;
   end;
   call symput('M',compress(put(M,best.)));
   stop;
   set lookup nobs=n;
run;
```

Now, the final number of buckets in the future hybrid-indexed lookup file (let us call it HLOOKUP), is stored in the macro variable M, which can also help determine the size of the arrays we will need to declare in a step where HLOOKUP is actually organized and written out.

DIRECT ADDRESSING INTO DISK

The next step is tagging each key K in the original lookup file with the bucket number H, to which it will eventually belong, and produce a file SPLIT that can be subsequently grouped by H and re-organized 'horizontally' forming HLOOKUP.

This is done by reading LOOKUP and applying the modulo formula discussed above to K in every record. However, there is a subtle hidden caveat, although it is quite unlikely to manifest itself unless IPR is quite low, approximately under 10. With very few items per bucket on the average, it might happen that no keys at all map to some bucket (whose number is, say, HGAP), and the final lookup file will have fewer records than the number of available buckets M. Therefore, it will be impossible to address the buckets directly using POINT= option. To account for such a situation, a dummy empty record with H=HGAP must be output to file SPLIT. It is taken into consideration in the step below:

```
data split (keep=h k s);
   array f(1:&M) _temporary_;
   do until (eof);
      set lookup end=eof;
      h = 1 + mod(k,&M);
      f(h) ++ 1;
      output;
   end;
   k = .K; s = .S;
   do h=1 to dim(f);
      if f(h) > hs then hs = f(h);
      else if f(h) = . then output;
   end;
   do hs=ceil(1.25*hs) by +1 until (j=up+1);
      up = ceil(sqrt(hs));
      do j=2 to up until (not mod(hs,j)); end;
   end;
   call symput ('hs',left(put(hs,best.)));
run;

proc sort data=split; by h; run;
```

In the DATA step, above, the first DO-loop reads LOOKUP, tags its records with H-values, and outputs them. At the same time, it uses the array F(1:&M) to track the number of keys falling into each bucket. The second DO-loop scans the array and outputs a dummy record with missing K and S for each empty bucket, thus closing the gaps, should any happen. Special missing values .K identify dummy records as such and allow to skip them in the future without searching. The second DO-loop also determines HS, the number of keys falling into the most populated bucket. The third, nested, DO-loop finds the first prime number greater than 1.25*HS and assigns it to the macro variable HS. Finally, in the sort step, file SPLIT is ordered by the bucket number H.

DIRECT ADDRESSING INTO MEMORY

The reason why the third DO-loop above is necessary deserves a good explanation. In the subsequent steps, we intend to read the file SPLIT and transpose it, writing K and S values from each H-group 'horizontally' as separate variables. Given a record from the driver file with its own K tagged with H, we are going to use POINT=H to access the only bucket (represented by an observation in HLOOKUP) where K can possibly be found, and search the lookup keys within the bucket to find K and its S-counterpart.

The question is, how should the lookup keys be organized within each bucket in order to facilitate the fastest possible search in memory? The simplest efficient solution is to order SPLIT, in the sort step above, by H K (not just by H), with the intention to use a hand-coded binary search. With 1024 keys per lookup record, say, it would need only about 11 key comparisons to locate a search key.

However, we can do much better than that by organizing the keys within each bucket in the form of a hash table. Compared to binary search, this type of in-memory search locates a search key, on the average, only after 2-3 key comparisons, regardless of the number of keys per record. By preparing 25% more slots within each bucket than the maximum number of keys a bucket can possibly have (hence the 1.25*HS in the code above), we are making the table sparse and thus can use the simplest collision resolution policy, open addressing with linear probing, in order to cut down on the amount of computations. The hash tables in each HLOOKUP record can be organized on the fly in the same DATA step where the file is prepared:

```
data hlookup (keep=kk: ss:);
   array kk(1:&hs);
   array ss(1:&hs);
   do until (last.h);
      set split end=eof;
      by h;
      if k = .K then leave;
      do j=1+mod(k,&hs) by 1 until (kk(j)=. or kk(j)=k);
         if j > &hs then j = 1;
      end;
      kk(j) = k;
      ss(j) = s;
   end;
run;
```

Note that keeping H would be redundant, since by the very nature of the algorithm, its values would exactly coincide with the observation numbers in HLOOKUP.

Before the keys K in DRIVER can be searched in HLOOKUP for their S-counterparts, the driver itself must be stratified and ordered by H. This is a simple operation:

```
data hdriver;
   set driver;
   h = 1 + mod(k, &M);
run;

proc sort data=hdriver; by h; run;
```

Reorganizing the driver file this way does drain certain resources, but it results in markedly improved lookup performance. Besides, in the real-life situation described in the introduction, the vendor can be required to shape the file in any specific way beforehand, so that the on-line application will accept the driver file in its ready-to-search form.

. . . AND CONQUER

At this point, HLOOKUP is hybrid-indexed and HDRIVER is grouped properly. The final step replaces the shadow account K with the real account S, creating file REALACCT:

```
data realacct (keep=s);
   array kk (1:&hs);
   array ss (1:&hs);
   set hdriver;
   by h;
   if first.h then set hlookup point=h;
   do j=1+mod(k,&hs) by 1 until (kk(j) = .);
      if j > &hs then j = 1;
      if k = kk(j) then do;
         s = ss(j);
         output;
         delete;
      end;
   end;
run;
```

According to the plan outlined above, HLOOKUP is read only once at the beginning of each group by H in DRIVER. It loads the entire bucket (a particular observation in HLOOKUP) into memory. The bucket contains lookup keys stored in the cells of the array KK in the form of a hash table. From that point on and until the end of the current H-group, no input from HLOOKUP occurs, because all the keys read from the group are searched completely in memory.

IT'S BETTER TO SEE ONCE . .

One look at a simple data sample may tell more that tons of wordy explanations. Imagine that LOOKUP has just 50 keys valued 1 through 50, and HDRIVER has 20 unique random keys chosen from K=[1:99]; not all of them have a match in LOOKUP. Let us take a peek at HDRIVER and HLOOKUP concocted from the sample set of keys by the process described above. The files are juxtaposed to each other in Table 1. Matching keys are shown in boldface.

Note for a key in any given H-group from HDRIVER, a match can be found only in the observation of HLOOKUP whose number is H. It means, first off, that no keys from the first HLOOKUP record-bucket can be present anywhere in HDRIVER, for the latter has no records with H=1 at all. In fact, during the search phase, this record will never have to be even looked at. The table should make the mechanics of searching clear.

Table 1. Sample HDRIVER and HLOOKUP.

HDRIVER			HLOOKUP (keys only)							
K	H		Obs	KK1	KK2	KK3	KK4	KK5	KK6	KK7
			1	.	22	44	.	11	33	.
23	2	-->	2	.	01	**23**	**45**	.	12	34
45	2									
46	3	-->	3	35	.	02	24	**46**	.	13
57	3									
36	4	-->	4	14	**36**	.	03	25	47	.
26	5	-->	5	.	15	37	.	04	**26**	**48**
48	5									
49	6	-->	6	**49**	.	16	38	.	05	27
28	7	-->	7	**28**	50	.	17	39	.	06
72	7									
07	8	-->	8	**07**	29	.	.	18	**40**	.
40	8									
08	9	-->	9	.	**08**	30	.	.	19	41
52	9									
63	9									
74	9									
20	10	-->	10	**42**	.	09	**31**	.	.	**20**
31	10									
42	10									

Let us consider, for instance, what happens for H=3. Once the H-group in HDRIVER with H=3 is hit, a read from HLOOKUP with POINT=H is performed, and the third record is moved to PDV. The first key to search for is K=46. MOD(46,7)+1 yields 5. Since KK(5)=46, we have an instant match. The corresponding satellite residing in SS(5) (not shown) is copied to S, and a record is written out to REALACCT. Next key in the group H=3 is K=57. MOD(57,7)+1 yields 2. Since KK(2)=., we immediately conclude that 57 is not in the bucket, and therefore not in HLOOKUP at all.

Thus, even though we have 20 keys to search for and 14 matching keys, they fall in only 9 distinct H-groups, and therefore HLOOKUP has be to directly accessed 9 times, strictly in order. In the case of the binary search, LOOKUP would have to be set (log2(50)+1)*20, i.e. about 140 times, all out of order. With SAS indexing, first, the index should be searched for each key in DRIVER, and since there are 14 matches, 14 direct accesses would have to be executed. Hence, fewer buckets result in higher performance, of course, with the limitations already discussed above.

SEMANTICS AND BEYOND

Just to clarify any semantic questions that might arise, the name for the indexing method described here has been chosen without any connection to existing algorithmic naming conventions. Hopefully, 'direct-addressed SAS data set' will not cause any confusion: A specific bucket in the data set is accessed directly, through POINT=H, hence the term. As far as 'hybrid-indexed' is concerned, one way to classify searching methods in general is distinguish between external (disk, tape) and internal (memory only) searches. The approach adopted in this paper combines the two: It uses a hash index into records on disk to locate a specific block

of keys and move them, together with their satellites, to memory. The keys in the block are already hash-indexed, which is why the final search can be performed very rapidly.

Summarizing, the hybrid indexing comprises two direct-addressed stages:

1. POINT=H disk access to a record holding a complete pre-hashed table. This is done only once for each by-group in the driver file grouped by H.
2. Direct access via hash index in the hash table in memory to locate (of reject) the current driver key and retrieve the satellite if a match is found.

Intuitively, we should be able to reap benefits from such a scheme for a simple reason. If there are 40 million records in LOOKUP, and it is transformed into HLOOKUP partitioned in 50,000 buckets, we will never have to read HLOOKUP (execute SET POINT=H instruction) more than 50,000 times regardless of the number of keys in the driver file. Let us now see what the verdict of the Supreme Judge, the Experiment, is.

THE NAME OF THE GAME

Performance, that is. Hybrid hash indexing was tested along with SAS indexing on the same machine that was used in the real application, that is S/390 9672/R36 under OS/390, SAS Version 8 (TS M0). As explained above, the influence of IPR value (items per record) on performance is very weak, so the benchmarks shown below were obtained with a static IPR set to 600, which translates into about 700 keys per lookup record. With that value fixed, the number of buckets depends on the number of keys in the original LOOKUP. LOOKUP files with NL=10, 20, 30, and 40 million keys were tested. Another varying parameter was the number of keys in the driver file DRIVER. The latter was tested with ND=100K, 500K, and 900K keys. Test files with the properties close to the real situation, were simulated in the following manner:

```
%let nl  =  4e7   ; *** records in lookup;
%let nd  =  1e5   ; *** records in driver;
%let kr  =  1e16  ; *** numeric key range;

data lookup (keep=k s) driver (keep=k);
   do s=1 to &nl;
      k = int(ranuni(1)*&kr);
      output lookup;
      if not mod(s,ceil(&nl/&nd)) then output driver;
   end;
run;
```

This way, it is guaranteed that all keys K in DRIVER will be found in LOOKUP. From the performance standpoint, this is the worst case scenario, for unsuccessful searches occur faster than successful ones, and greater number of hits leads to heavier output traffic.

The results are summarized in performance Table 1. It appears that the common-sense strategy devised above works pretty successfully. Even in the worst case scenario, at 40 million records in LOOKUP and almost 1 million keys in DRIVER, the response time stays under 10 CPU seconds. Hybrid indexing thus beats the SAS index quite soundly, especially in the case of relatively large driver files.

It should also be noted that in the experiments described above, the relative performance of hybrid indexing was, in fact, underestimated, because the test data had been contrived in every possible way to benefit SAS index usage. The indexed key K was uniform and discriminant, and both files were sorted into ascending order. Substantial skewness in the distribution of the keys across their range may impact B-tree performance quite adversely. But it has no impact whatsoever on performance of the hybrid-indexing scheme, because it provides its own randomization mechanism: The hash modulo function spreads the keys among the buckets evenly, no matter how the keys are distributed themselves.

Table 2. Hybrid Hash Index vs SAS Index.

		Observations in Driver		
		100K	500K	900K
Observations in Lookup	Lookup Technique			
10M	Set Key=	4.45	17.52	28.15
	SQL Join	6.72	28.96	45.61
	Hybrid Index	1.20	2.81	4.15
20M	Set Key=	5.54	18.91	30.74
	SQL Join	7.91	29.23	49.41
	Hybrid Index	1.86	3.65	5.13
30M	Set Key=	6.60	20.41	32.28
	SQL Join	9.08	31.07	51.28
	Hybrid Index	2.59	4.28	5.83
40M	Set Key=	8.28	34.15	54.64
	SQL Join	10.97	37.78	62.15
	Hybrid Index	3.34	5.41	7.21

To fully appreciate the lookup speed delivered by hybrid indexing (and SAS indexing, too!), let us return to the DB2 alternative mentioned above. On the same computer, a DB2 table structured exactly like LOOKUP and indexed by K was loaded with 40 million rows. With 100,000 keys in the driver file, it took DB2 (optimized for the index usage) 35.68 CPU seconds to do the same job the hybrid index and SAS index had finished in 3.34 and 8.28 CPU seconds, respectively.

LOADING HLOOKUP FROM SCRATCH

The run-time required to organize a hybrid-indexed lookup table by loading it from scratch is insignificant in the context of the current task. However, it may be of some importance if a hybrid-indexed structure is used, for example, as a repository for a data mart. A lookup table that needs an infinite time to be prepared hardly represents any practical value, even if it can be searched with a jaw-dropping speed. Fortunately, in the case of hybrid indexing, it takes roughly the same time to stratify, sort, hash, and finally shape the table, as it does to sort a file and create a SAS index. For instance, with 40 million keys in the original unsorted lookup file, sorting the file by K and building a SAS index on it takes approximately 400 CPU seconds. The three steps preparing the hybrid-index file HLOOKUP take about 420 CPU seconds, in all.

UPDATING HLOOKUP IN PLACE

Much more important than loading is whether it is possible to update HLOOKUP from a relatively small transaction file. It is certainly undesirable to rewrite the entire master

file having 40 million keys, to only update 100,000 of them. In the original setting, where LOOKUP has a SAS index defined on the key, the file is easily updated directly via the MODIFY statement. Can the same be done against a hybrid-indexed file?

The answer is 'yes', and in fact it is quite simple. Let us assume that the file TRANS contains the key K and the new real account S that is supposed to overwrite the old one corresponding to the same key. Just like the driver file, the transaction file has to be tagged with the bucket number H and sorted by H first:

```
data trans (keep=h k s);
   set trans;
   h = 1 + mod(k,&M);
run;
proc sort data=trans; by h; run;
```

Now, the MODIFY statement can be used to update HLOOKUP in place:

```
data hlookup (keep=kk: ss:);
   array kk(1:&hs);
   array ss(1:&hs);
   set trans;
   modify hlookup point=h;
   do j=1+mod(k,&hs) by 1 until (kk(j)=. or kk(j)=k);
      if j > &hs then j = 1;
   end;
   kk(j) = k;
   ss(j) = s;
run;
```

By the nature of the algorithm, the MODUFY instruction will always be successful, since H always lies between 1 and &M, and the number of observations in HLOOKUP always equals &M.

INSERTING NEW ACCOUNTS

If a key in the transaction file is new and is not present in HLOOKUP, it will be inserted automatically by the updating program shown above into the first missing array node encountered when KK(J)=. evaluates to true. It will never increase the number of HLOOKUP records &M. Instead, the unused slack left in the hash table when the file was being loaded will be used. Thus, the sparsity of the observations in HLOOKUP plays a double role: It promotes good hashing performance and leaves some room where new keys and satellites can be inserted. The factor 1.25 in the program that created HLOOKUP is somewhat arbitrary, and was used instead of, say, 1.5 just to conserve disk space. However, if numerous insertions can be anticipated in the future, the factor can be increased accordingly.

CONCLUSION

Hybrid indexing is a method of organizing lookup data in a SAS data file, where each row serves as a bucket containing approximately equal number of keys and corresponding satellites stored in a hash table. It takes about the same time to organize a hybrid-indexed data set as it does to build a SAS index. However, by shifting the main burden of searching from I/O to memory, hybrid indexing results in 3-7 times faster lookup times. Unlike SAS indexing, hybrid indexing provides its own randomization mechanism and is thus insensitive to the distribution of keys. It also scales much better as the size of the subset returned from the lookup file grows.

It might seem amazing that with nothing but Base SAS, it is possible, in under 8 seconds, to read a file with 1 million keys, find their attributes among 40 million entries stored in another file, and write the output. However, in reality, it is a direct product of the power and flexibility of the SAS Language. The statue was already inside the piece of marble, and we only had to remove the excess of the material. Which requires nothing more than the fun of SAS programming.

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 SESUG'00, Charlotte, NC.
6. M.A. Raithel. *Tuning SAS Applications in the MVS Environment*, Cary,NC:SAS Institute Inc.,1995.

AUTHOR CONTACT INFORMATION

Paul M. Dorfman
10023 Belle Rive Blvd. 817
Jacksonville, FL 32256
(904) 564-1931 (h)
(904) 954-8533 (o)
sashole@bellsouth.net
paul.dorfman@citicorp.com
paul_dorfman@hotmail.com