LINEAR HASHING : A NEW TOOL FOR FILE AND TABLE ADDRESSING.

Witold Litwin

I. N. R. I. A.
78 150 Le Chesnay, France.

Abstract.

Linear hashing is a hashing in which the address space may grow or shrink dynamically. A file or a table may then support any number of insertions or deletions without access or memory load performance deterioration. A record in the file is, in general, found in one access, while the load may stay practically constant up to 90 %. A record in a table is found in a mean of 1.7 accesses, while the load is constantly 80 %. No other algorithms attaining such a performance are known.

1. INTRODUCTION.

The most fundamental data structures are files and tables of records identified by a primary key. Hashing and trees (B-tree, binary tree,..) are the basic addressing techniques for those files and tables, thousands of publications dealed with this subject. If a file or a table is almost static, hashing allows a record to be found in general in one access. A tree always requires several accesses. However, when the file or the table is, as usually, dynamic, then a tree still works reasonably well, while the performance of hashing may become very bad. It may even become necessary to rehash all records into a new file.

We have shown, however, in /LIT77/ that hashing may be a tool for dynamic files, if the hashing function is dynamically modified in the course of insertions or deletions. We have called this new type of hashing virtual hashing (VH), in contrast to the well known hashing with a static function, which we will refer to as classical. Through an algorithm called VHO /LIT77/, we have shown that a record in a dynamic file may typically be found in two accessses, while the load stays close to 70 % /LIT77a/. Another algorithm, called VH1 /LIT78/, / LIT78a/, has shown that a record may even be found typically in one access, while the load during insertions oscillates between 45 % and 90 % and is 67.5 % on average. It showed also that the average load during insertions may be always greater than 63 % and almost always greater than 85 %, if we accept that the average successful search requires 1.6 accesses /LIT79a/. Finally, a generalisation of VH1, called VH2, has shown that for a similar load, the average successful search requires very close to one access /LIT79a/. Two other algorithms similar to VHO have been proposed, Dynamic Hashing (DH) /LAR78/ and Extendible Hashing (EH) /FAG78/. Since trees typically lead to more than 3 or 4 accesses per search and to a load close to 70 % /KNU74/, /COM79/, all these VH algorithms offer better access performance for similar or higher load factors.

VHO, DH and EH require at least two accesses per search because the data structure which represents the dynamically created hashing functions must be on the disk. VH1 and VH2 are faster, because the functions are represented by a bit table which, depending on file parameters, needs 1 kbyte of core (main storage), per file for 7 000 to more than 1 500 000 records. In this paper, we present the algorithm which goes further, only a few bytes of core suffice now for a file of any size. For any number of insertions or deletions, the load of a file may therefore be high and a record may be found, in general, in one access. No other algorithms attaining such a performance are known.

The algorithm is called Linear Virtual Hashing or Linear Hashing in short (LH). The choice of file parameters may lead to a mean number of accesses per successful search not greater than 1.03, while the load stays close to 60 %. It may also lead to a load staying equal to 90 % while the successful search requires 1.35 accesses in

the average. Even if the buffer in core may contain only one record, a search in the file needs 1.7 accesses in the average while the load remains at 80 %. This property makes LH probably the best performing tool for dynamic tables as well.

The next section describes the principles of LH. We first show the basic schema for hashing. We then discuss the computing of the physical addresses of buckets, when the storage for them is allocated in a non-contiguous manner. Finally, we present some variants of the basic schema.

Section 3 shows performance of the Linear Hashing. First, we show access and memory load performance of the basic schema. Next, the performance are analysed for a variant with a, so-called, load control.

Section 4 concludes the paper. We sum up the advantages which Linear Hashing brings, we show some application areas and, finally, we indicate directions for further research.

2. PRINCIPLES OF THE LINEAR HASHING.

2.1. Basic schema.

We recall that hashing is a technique which addresses records provided with an identifier called primary key or, simply, key. The key, let it be c, is usually a non-negative integer and, in a work on the addressing by primary key, we may disregard the rest of the record. A simple pseudo-random function, let it be h, called a hashing function, assigns to c the memory cell identified by the value h(c). The hashing by division c /--> c mod M ; M = 2,3,.. ; is an exemple of a hashing function. The cells are called buckets and may contain b records, b = 1,2,.. . The record is inserted into the bucket h(c), called primary for c, unless the bucket is already full. The search for c always starts with the access to the bucket h(c).

If the bucket is full when c should be stored, we speak about a collision. An algorithm called collision resolution method (CRM) is then applied which, typically, stores c in a bucket m such that $m \neq h(c)$. c then becomes an overflow record and the bucket m is called overflow bucket for c. If (i) overflow buckets are not primary for any c,(ii) each of them is devoted to only one h(c) and (iii) a new overflow bucket for an h(c) is chained to the existing ones, then we have a bucket chaining CRM. If, in particular, the capacity b' of an overflow bucket is b' = 1, we call this CRM separate chaining /KNU74/.

A search for an overflow record requires at least two accesses. If all collisions are resolved only by overflow record creations, as it was assumed until recently /LIT77/, then access performance must rapidly deteriorate when primary buckets become full. If the insertion of c leads to a collision and no records already stored in the bucket h(c) should become overflow records, then c may be stored in its primary bucket only if a new hashing function is chosen. The new function, let it be h', should assign new addresses to some of the records hashed with h on h(c) and the file should be reorganized in consequence. If h = h' for all other records, the reorganizing needs to move only a few records and so may be performed dynamically. The new function is then called by us dynamically created hashing function or, shortly, a dynamic hashing function. The modification to the hashing function and to the file is called a split, h(c) is, under the circumstances, split address . The idea in VH in general and so, in particular, in LH is to use splits in order to avoid the accumulation of overflow records. Splits are typically performed during some insertions. All splits result from the application of split functions. For LH, as well as for VH0 and for VH1, the basic split functions are defined as follows /LIT79/, /LIT80/ :

- Let C be the key space. Let $h_0 : C \rightarrow \{0, 1,.., N-1\}$ be the function that is used to load the file. The functions h_1, h_2,.., h_i,.. are called split functions for h_0 if they obey the following requirements :

(1)

$$h_i : C \rightarrow \{0, 1,.., 2^i N-1\}$$

(2)

For any c either :

$$h_i(c) = h_{i-1}(c) \quad (2.1)$$

or :

$$h_i(c) = h_{i-1}(c) + 2^{i-1}N \quad (2.2)$$

We assume that, typically, each h_i ; i = 0,1,.. ; hashes randomly. This means that the probability that c is mapped by h_i to a given address is $1/2^i N$. This also means that (2.1) and (2.2) are equiprobable events.

Fig. 1 illustrates the use of split functions. The file is created with h0 : c --> c mod N, where N = 100. The bucket capacity is b = 5 records. For split functions we choose the hashing by division, namely we put :

$$h_i : c \mapsto c \bmod 2^i N.$$

This choice respects (2) since, obviously, for any non-negative integers k, L, either :

k mod 2L = k mod L

or :

k mod 2L = k mod L + L.

We assume that a collision occurs during the insertion of c = 4900. Instead of simply storing c as an overflow record, we change h_0 to the following h :

$h(c) = h_1(c)$ if $h_0(c) = 0$
$h(c) = h_0(c)$ otherwise.

We then reorganize the file. We thus have applied h_1 as the split function and we have performed the split for the address 0. The hashing function h results from the split and is a dynamic hashing function.

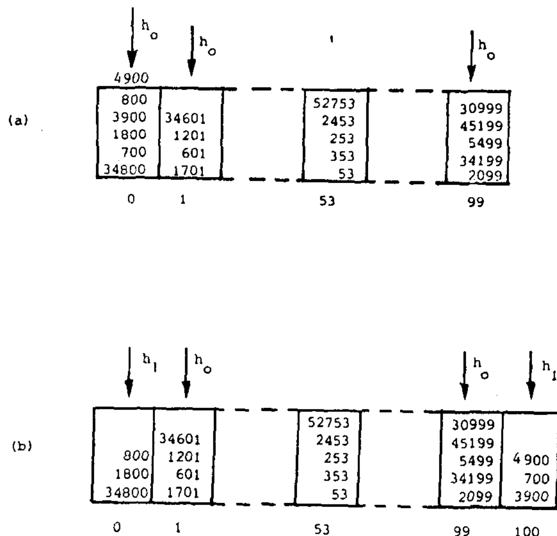


Fig. 1. The use of a split for a collision resolution. (a) - a collision occurs for the bucket 0. (b) - the collision is resolved without creating an overflow record and the address space is extended.

Fig. 1.b shows the new state of the file. Since $h = h_0$ for each address except 0, no records other than these hashed to 0 have been moved. Since $h = h_1$ for the records which h0 hashed to the address 0, for approximatively half the number of these records the address has been changed. It followed from (2) that all these records had the same new address which, under the circumstances, had to be 100. A new bucket has therefore been appended to the last bucket of the file to which all the records have been moved in one access. Since for all these records the bucket 100 is henceforward the primary bucket, they are all accessible in one access. In particular, this is also the case of the new record, i. e., 4 900. On the other hand, the records which remained in the bucket 0 continue to be accessible in one access. In contrast to what could be done if a classical hashing was used, the split has resolved the collision without creating an overflow record and without access performance deterioration.

Let us now assume that the file addressed with h0 undergoes a sequence of insertions which did not yet lead to overflow records. Furthermore, let us assume that a split is performed iff a collision occurs. The natural idea would be to split the bucket which undergoes the collision, this was implicit for all algorithms for VH. However, split addresses must then be random and this must lead to dynamic hashing functions using tables. Dynamic hashing functions which do not need tables may be obtained only if the split addresses are chosen in a predefined order. To perform splits in some predefined order, instead of split the bucket which undergoes the collision, is the main new idea in the linear hashing.

Let m be the address of a collision. Let n be the address of a split to be performed in the course of the resolution of this collision. Since the values of m are random while these of n are predefined, usually $n \neq m$. If so, we assume that the new record is stored as an overflow record from the bucket m through a classical CRM, bucket chaining for instance. Next, we assume that n is given by a pointer which thus indicate the bucket to be split. For the first N collisions, the buckets are pointed in the linear order 0,1,2,..,N-1 and all splits use h_1. (2.2) implies then, that the file becomes progressively larger, including one after another the buckets N+1, N+2,...,2N-1. A record to be inserted undergoes a split usually not when it leads to the collision, but with some delay. The delay corresponds to the number of buckets which has to be pointed while the pointer travels up, from the address indicated in the moment of the collision, to the address of this collision.

With this mechanics, no matter what is the address, let it be m1, of the first collision, LH performs the first split using h_1 and for the address 0. The records from the bucket 0 are then randomly distributed between the bucket 0 and a new bucket N, while, unless $m_1=0$, an overflow record is created for the bucket m_1. The second collision, no matter what is its address, let us say m_2, leads to an analogous result, except that,

first, it splits for the address 1 and appends the bucket N+1. Next, it may constitute the delayed split for the first collision, suppressing therefore the corresponding overflow record. This process continues for each of the N first collisions, moving thus the pointer, step by step, up to the bucket N-1. Sooner or later, the pointer points to each m and the splits, despite of being delayed, move thus most of the overflow records to the primary buckets. We may therefore reasonably expect that, for any $n < N$, only a few overflow records exist.

After N collisions, we have $h = h_1$. (1) implies then that, instead of the hashing on N addresses, we now hash on 2N addresses. (2) implies that h_2 has on the hashing with h_1, the action analogous to that of h_1 on the hashing with $h = h_0$, except that it hashes on 4N addresses. We therefore assume that $n = 0$ again, that now we split with h_2 and that the upper bound on n is now 2N-1. For further insertions, we use $h_3, h_4, .., h_j, ..$ while the pointer travels each time from 0 to $2^{j-1}N$.

It results from the above principles that, first, the address space increases linearly and is as large as needed. Next, for any number of insertions, most of overflow records is moved to the primary buckets by the delayed splits. On one hand, we may thus reasonably expect that the rate of overflow records remains always small. If $b >> 1$, the rate should even be neglectable small. Thus, we may expect the linear hashing to find a record usually in one access to the bucket, no matter how few buckets were provided when the file was created and how high the number of insertions finally is.

The highest index of a split function currently used, let it be j ; j=0,1,.. ; is called *file level*. If n=0, we always have $h = h_j$ for some j. Otherwise, first, $h = h_{j-1}$ for the buckets not yet split with h_j, i. e., n, n+1,..,$2^{j-1}N$. Next, $h = h_j$ for all the others. The algorithm computing the primary address of a c is therefore trivial :

```
(A1)
     if n = 0 : m <-- h_j(c)
     else
          m <-- h_{j-1}(c)
          if m < n : m <-- h_j(c) endif
     endif
endA1
```

2.2. Physical address computing.

The address given by hashing must be transformed into the physical address of the bucket in the memory. The memory for files is usually divided into quanta of let it be q buckets ; q = 1,2,.. Quanta may be all of the same size or different sizes may be available. It then may be particularly worthwhile to use sizes which are 2^i of a certain minimal q (buddy system /KNU74/).

When the file is loaded some quanta are statically allocated. Then, if a file increases dynamically, quanta are sometimes added. If all quanta for a file are contiguous, then the the physical address of a bucket m is as follows :

$$m'(m) = m'(0) + dm \quad (3)$$

where d is the number of memory elements, i. e., bytes, words or sectors,.. per bucket. Thus the advantage of a contiguous allocation is that only the address of the first quantum is needed and that the computing of the physical address is trivial.

However, if several dynamic files should share a memory, it may be better to allocate non-contiguous quanta. For each file, the addresses of these quanta must then be collected. The address of the i-th quantum may be the value T(i) of a table T. For the quanta of a fixed size, we then have the following formula :

$$i(m) = INT(m/q) \quad (4)$$

$$m'(m) = T(i(m)) + d(m - i(m)q)$$

where INT denotes the integer part. In particular, if q=1, i. e., if the allocation is totally distributed, then we have simply :

$$m'(m) = T(m). \quad (5)$$

For the quanta of different sizes, we particularly recommend the following schema :

- let K be a parameter; K = 1,2,.. . The sizes $q_0, q_1, ...$ of successive quanta of the file should be :

$$q_0 = N \quad (6)$$

$$q_1 = q_2 = ... = q_K = q_0/K$$

. . .

$$q_{lk+1} = q_{lk+2} = .. = q_{(l+1)k} = 2^l q_1$$

where l=0,1,.. For instance, if N = 20 and K = 4, then the sizes of the quanta dynamically allocated are 5, 5, 5, 5, 10, 10, 10, 10, 20, 20,.. . Dynamic allocations take thus place when LH starts to use the addresses 20, 25, 30, 35, 40, 50, 60, 70, 80, 100,... Higher is the value of K, smaller is the drop in memory load when a new quantum is allocated, but T is larger. The practical values of K are between 1 and 10.

Let m_i be the smallest logical address in the i-th quantum. Next, let it be ;

$$m'' = m - m_{i(m)}.$$

Therefore we have :

$$m'(m) = T(i(m)) + dm''. \quad (7)$$

In the case of (6), i(m) and m'' may then be computed by the following obvious algorithm :

```
(A2)
     if m < N : i <-- 0 ; m'' <-- m
     else
          i <-- j-1 ; M <-- 2^i N
          while M > m and i > 0 :
               M <-- M/2 ; i <-- i-1
          endwhile
          i' <-- M/k ; m'' <-- m-M
          i <-- INT(m''/i') +ik +1
          m'' <-- m'' mod i'
     endif
endA2
```

2.3. Variants of the basic schema.

2.3.1. Split control.

Splits that are performed iff a collision occurs are called *uncontrolled*. Splits are called *controlled* if they also depend on other conditions or are performed even if there is no collision. A particularly useful control is called *load control*. Under this control, a split is performed when a collision occurs, but only if the load factor is superiour to some threshold. This may concern the load factor, let it be $\hat{a}$, defined as usual /KNU74/ :

$$\hat{a} = x/bM \quad (8)$$

where x is the number of records in the file, b is the bucket capacity and M is the number of primary buckets. The control may also take in to account the overflow buckets in which case the load factor, let it be $\hat{a}'$, is defined as :

$$\hat{a}' = x/(bM + b'M') \quad (9)$$

where M' is the number of overflow buckets and b' is the overflow bucket capacity.

In what follows the thresholds are denoted as g and g', respectively. We will show that, when the file undergoes insertions, the load control usually keeps the load factor almost equal to the chosen threshold. A similar control may keep the load factor greater than or almost equal to a threshold, when the file undergoes deletions. Each time a deletion brings the load below this threshold, we may simply perform an operation called *grouping* which is inverse to splitting. A grouping moves thus the pointer one address backward and so decreases M. If the threshold for deletions is equal to the one for insertions, *the load of a LH file usually stays almost constant*.

2.3.2. Pointer independent address computing.

It may surprise, but primary address may be computed in fact without the knowledge of the value of n. The following algorithm, analogous to that of VH1 /LIT78a/, proves it :

```
(A3)
     m <-- h_j(c)
     if m >= M : m <-- h_{j-1}(c) endif
endA3
```

If $n = 0$, then it is obvious that (A3) works. If $n \neq 0$ then, if :

$$h(c) = h_j(c),$$

then :

$$h(c) < n < M$$

or :

$$2^{j-1} < n < M.$$

Thus (A3) terminates correctly. Else, we have :

$$h(c) = h_{j-1}(c).$$

Then, if :

$$h_{j-1}(c) = h_j(c)$$

then :

$$h(c) < 2^{j-1}N < M.$$

Thus (A3) also terminates correctly. Else :

$$h_j(c) >= M,$$

since the bucket h(c) is not yet split. Therefore, in this last case, the algorithm terminates correctly as well.

In particular, we may assume $N = 1$. The file level j is then a function of M. It follows, first, that the size of address space may be the only one parameter which LH needs for addresses

computing. It follows, next, that the parameters of a classical hashing may suffice in order to construct a linear one. For instance, the knowledge of the number of the addresses for the hashing function and of the fact that a hashing by division is used, suffice in both cases.

2.3.3. Other split functions.

Let $b_1, b_2, \dots, b_i, \dots$ be a sequence of randomly generated bits, with equiprobability of $b_i=0$ and of $b_i=1$. Such a sequence may be obtained using a random number generator. Let B_i be the integer with binary representation $b_i, b_{i-1}, \dots, b_1$. The functions h_i defined as follows :

$$h_i : c \mapsto h_0(c) + B_i N \qquad (10)$$

are, obviously, split functions for <u>any</u> h_0. Thus, LH may be constructed not only for a hashing by division, but for any usual hashing function.

Split functions with B_i given by a random number generator may be particularly interesting for $N = 1$. For this value each h_i hashes on 2^i addresses. If the hashing by division is applied, the address of c is, simply, the i least significant bits of c. The hashing by division may then be sometimes rather non-random, while a random number generator may still perform well. The choice of $N = 1$ is particularly useful since, first, there is no more problem to choose among several possible h_0. Next, LH covers all possible address space sizes. Finally, since the file may then be constituted even from only one bucket, a good load factor may be provided even for very few records.

B_i may also result from the <u>multiplication function</u> /KNU74/, let it be h'. If w is the word size and A is an integer relatively prime to w, then the hashing with h' on 2^i addresses is defined as follows :

$$h'_i(c) = INT(2^i((Ac/w) \bmod 1)) \qquad (11)$$

B_i is in this case constituted from the bits of $h'_i(c)$, taken in the reverse order. In other terms, the most significant bit of h'_i becomes the least significant in B_i etc

Knuth shows that a particularly good choice for A is A = 6 125 423 371. He also shows an algorithm computing (11) in only four instructions of the MIX assembler. Finally, he shows that his algorithm is usually faster than the hashing by division. To compute B_i through the multiplication function may thus be faster than through a random number generator.

In particular, Knuth shows that h' is a <u>scrambling function</u>. This means, first, that its partial result, let it be f(c) ; f(c) = Ac mod w ; is such that if $c' \neq c''$, then $f(c') \neq f(c'')$. Next, this means that the transformation $c \rightarrow f(c)$ tenders to randomize the keys. Therefore, the following split functions may be constructed :

$$h_i(c) = f(c) \bmod 2^i N \qquad (12)$$

which may perform better than the direct hashing by division.

Finally, split functions may also be constructed for alphabetic or variable-length keys. In particular, the individual words of such a key may be simply combined into a single word, to which any of the previously discussed functions may then be applied. Any of the combinations suggested by Knuth may be used, the addition mod w for instance.

2.3.4. General definition of split functions.

LH may be seen as VH1 in which split addresses have been predefined. VH1 may be generalized into an algorithm called VH2. Furthermore, it may be assumed that split addresses are, in fact, predefined for VH2. The conditions (1) and (2) may then be generalized follows ;

- let K be a parameter which value is fixed when the file is created ; K=1,2,.. . Let it be $k_i = K + i \bmod K$. Let N be an integer, $N > 1$. The hashing functions h_i ; i=1,2,.. ; are split functions for a hashing functions h_0, if the following condition are respected :

- for i = 0,1,.. : (13)

$$h_i : C \rightarrow \{0,1,\dots,N_i-1\}$$

$$N0 = KN$$

$$N_{i+1} = N_i + N_i/k_i$$

(14)

For any c, either :

$$h_i(c) = h_{i-1}(c)$$

or :

$$h_i(c) = N_{i-1} + INT(h_{i-1}(c) / k_i).$$

For example, if N=5 and K=4, then the successive N_is are : 20, 25, 30, 35, 40, 50, 60, 70, 80, 100,... (note the similarity to (6)).

As previously, we assume that each h_i should hash randomly, It follows that the probability that a key changes the address after a split, let it be p, is now $p_k = 1/(k_i+1)$. If K = 1, then (13) and (14) are, simply, (1) and (2) and $p_k = 0.5$. For greater K, p_k decreases and the distribution of records within LH file becomes more uniform. First, higher load factor obviously results for uncontrolled splits. Next, when the threshold increases, load control with K > 1 should lead to a better search performance. However, it is easy to see that for such K, to perform a split, usually needs more accesses. Therefore, insertions and deletions will be more costly than for K = 1 as well.

K > 1 implies that the address space doubles not after one, but after K trips of the pointer. Each of K trips may then be called a *partial expansion* of the address space. Partial expansions may result from formula others than the above, these introduced by /LAR80/ in particular. Performance resulting from (13) and (14) for K > 1 being quite similar to these of the Larson's schema, only the case of K = 1 is discussed in what follows.

3. *PERFORMANCE ANALYSIS*.

3.1. *Address computing*.

If the allocation is contiguous, LH is obviously almost as simple and fast as the classical hashing. If the allocation is non-contiguous and (3) or (4) are used, then this is also the case, as long as T may be entirely in core. The use of (6) needs few more instructions, but the computing of (A2) also very fast ; it is quite clear that, for any j, the "while" loop is in the average executed at most twice.

A four byte word allows the values of n and of j to go up to 2^{32}, i. e., allows the LH file to grow up to more than four billions buckets. For any number of insertions, (A1) enters thus even a very small core. If the allocation is contiguous, since (3) is used, the computing of the physical address also needs only a few bytes. If the allocation is non-contiguous, the core is mainly needed for T. If (A2) is used, the size of T is, obviously, not greater than jk+1. For k = 10 which is largely sufficient in practice, 301 words are then sufficient for a file which increases even a billion times. Thus, in practice, no matter if the allocation is contiguous or not, no matter how small is the core and how high is the number of insertions, the computing of a address resulting from LH, *never requires a disk access*.

However, the disk storage is usually needed if the allocation is totally distributed. (5) shows then clearly that any address is computed in no more than one access. On the other hand, since the disk is required only for T and since T contains only the pointers to buckets, the disk storage required then by LH is the minimal one. It is usually much smaller than that of the index of a VSAM file, since the index contains keys and internal pointers and since its load foactor is lower. It is also several times smaller than the storage required by the tables of VHO, DH and EH, either because of their much lower load factor (VHO, EH) or because of the internal pointers (DH).

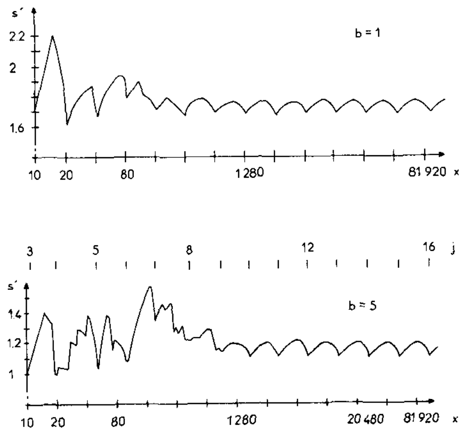
3.2. *Uncontrolled split*.

3.2.1. *Access performance*.

We now assume that overflow records are addressed through separate chaining and that the file is created by x insertions ; x=0,1,.. . We also assume that each h_i hashes randomly. Finally, we assume (and we have now right to do it) that the computing of an address never requires a disk access. By s', s'', s''' we denote the mean number of accesses per successful search, per unsuccessful search and per insertion. By $\hat{s}$ we denote the mean number of accesses per split. These coefficients will be called *costs*.

Fig. 2 shows curves of s'(x) obtained through simulations for bucket capacities b = 1,5,10,50. For b =1,5 we have not shown the first splits, since they correspond to x < 10 and to s' practically equal to 1. For b = 5, we have also shown the evolution of the file level j. The curves describe the file which is created with only one bucket and which undergoes $x > 2^{13}b$ insertions. At the end, the number of records in the file is thus more than 8 000 times greater than the number of records which could be found in one access in the initial one. It does not even make sence to compute what would be the deterioration of access performance, if the classical hashing would have been used.

For each b, when x increases, the curve is first irregular and therefore displays the existence of a *transient state*. After a few insertions, the file reaches a *stable state* , where the curve is a periodic function of $\log_2 x$. On one hand, it means that in the stable state s' does not depend on x, but on the relative position of the pointer. On the other hand, it means that s' does not increase for whatever the number of



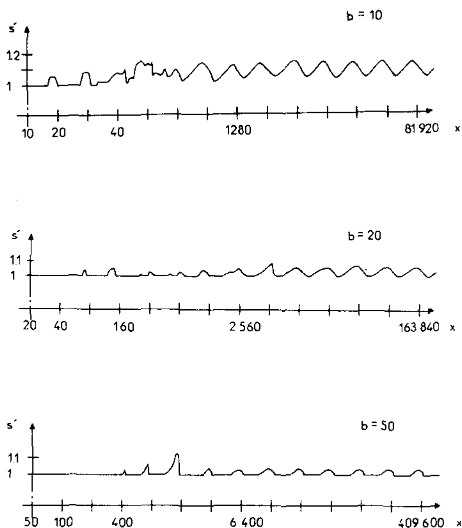


Fig.2. Mean number of accesses per successful search for linear hashing with uncontrolled split.

insertions could be. We therefore may conclude that for any bucket capacity and any number of insertions, the mean number of accesses per successful search stays close to 1. For large buckets, we may even consider that we have always $s'=1$! LH is thus a very important algorithm, since, first, <u>no other algorithms attaining such an almost ideal performance are known</u>. Next, it performs several times better than trees. We recall that B-trees need in general 4,5 accesses while binary trees need typically more than 10, since $s'(x) \hat{=} \log_2 x$ /KNU74/.

The transient state may be disregarded, since the performance is good and x is neglectable small. Since the curves in the stable state are periodical, they may be characterized by the values of s'_{min}, s'_{max} and of s'_{ave}, which is the mean value of s' over one period. The values corresponding to the curves are displayed in the table 1.a.

b	1	5	10	20	50	
s'_{ave}	1.73	1.16	1.07	1.02	1.00	(a)
s'_{max}	1.77	1.20	1.11	1.06	1.03	
s'_{min}	1.68	1.10	1.02	1.00	1.00	
s''_{ave}	1.62	1.28	1.19	1.12	1.06	(b)
s''_{max}	1.63	1.32	1.26	1.24	1.23	
s''_{min}	1.6	1.20	1.08	1.02	1.00	
s'''_{ave}	7.91	3.97	3.14	2.67	2.35	(c)
s'''_{max}	8.92	4.07	3.45	3.09	2.71	
s'''_{min}	6.45	3.34	2.48	2.11	2.00	
$\hat{s}_{ave}$	6.94	6.09	5.99	5.98	5.98	(d)
$\hat{s}_{min}$	5.80	5.10	5.00	5.00	5.00	
$\hat{s}_{max}$	8.62	7.45	8.10	8.85	10.85	
$\hat{a}_{ave}$	1.30	0.66	0.61	0.59	0.59	(e)
$\hat{a}_{max}$	1.30	0.67	0.63	0.65	0.70	
$\hat{a}_{min}$	1.30	0.65	0.59	0.55	0.51	
$\hat{a}'_{ave}$	0.8	0.63	0.59	0.59	0.59	

<u>Table 1</u> - Performance of linear hashing with uncontrolled splits :

(a) successful search,
(b) unsuccessful search,
(c) insertion,
(d) split,
(e) load factor.

The analysis of s'' through simulations and through modelling /LIT79/ shows also a transient state and a stable state. Both states correspond of course with these of s'. The performance of unsuccessful search with LH in the stable state are displayed in the table 1.b. As in the case of classical hashing using the separate chaining (see /KNU74/), they are better than the performance of the successful search for b = 1 and poorer for

b > 1. However, as before, for any bucket capacity and any number of insertions s'' stays close to 1. The limit value for s''_{max} when b increases is 1.24 /LIT79/.

Table 1.d shows the characteristics of the split cost. Unless split is performed for the address of the collision, this cost is at least five accesses (two accesses in order to store the overflow record for the bucket m, three accesses in order to split the bucket n). The split needs more accesses if the bucket n overflows. This case is obviously the most frequent for b = 1 ; that is why $\hat{s}_{ave}$ is maximal. The number of overflow records on n is obviously the shortest for n close to 0, $\hat{s} = s_{min}$ corresponds to such values of n. Inversely, the longest chains exist for n close to 2^i, $\hat{s} = s_{max}$ corresponds thus to the end of a trip of the pointer. $\hat{s}_{max}$ increases with b, since the chains become longer while the use of the separate chaining implies one access per every record in the chain. However, $\hat{s}_{ave}$ reveals practically independent of b.

Finally, table 1.c lists s''', i. e., the cost of an insertion. For b = 1 almost 8 accesses are needed, since split cost is the highest and since a split results from almost any insertion. For larger b, s''' falls down quickly, since the proportion of insertions leading to a split decreases and the others need typically 2 to 3 accesses. If b >> 20, which is a typical value for files, s''' oscillates between 2 and 3. Thus, first, as it was the case of the other costs, insertion cost of LH may also stay always close to its theoretical minimum. Next, even in the worst case, i. e., for b = 1, the insertion cost is still typically much smaller than for trees, since, for instance, for $x = 10^5$, a binary tree leads to s''' > 16 /KNU74/.

3.2.2. Load factor.

The characteristics of load factors $\hat{a}$ and $\hat{a}'$ are shown in the table 1.e. For b = 1 the load factor is constantly equal to 80 %. This conjunction of such a good load and of the previously shown access performance makes LH probably the best known tool for dynamic tables. For higher values of b, the load is going down to the average value of almost 60 % and so the load of LH with uncontrolled split may be almost 10 % worse than the load of a B-tree. However, better access performance is usually prefered to a sligtly better use of the increasingly cheaper disk space.

3.3. Controlled split.

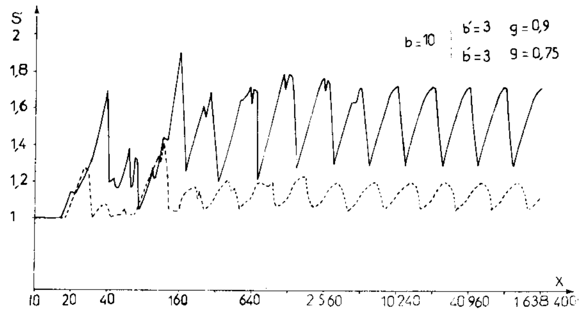
If the load is controlled and the threshold g is greater than $\hat{a}_{max}$, then the load is practically equal to g. It is obvious that the higher g is, the worse must be the access performance, since the ratio of overflow records increases. Simulation studies show, however, that substantial increases to the load factor may be achieved while the value of the access performance still stays excellent.

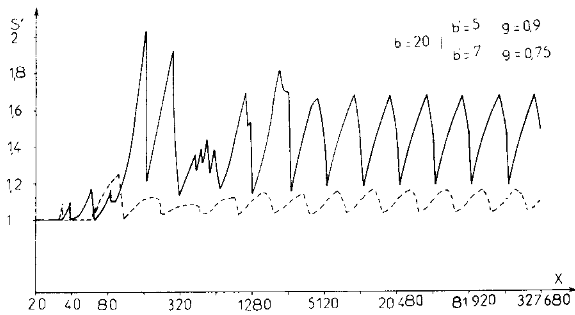
For instance g = 0.75 and b = 5 leads to a load which is almost 10 % higher than the one for the uncontrolled load. The correspondig access performance is still very good, since $s'_{ave} = 1.25$, $s''_{ave} = 1.43$ and $s'''_{ave} = 3.84$. For b = 50, the same g leads to a 16 % improvement, while the access performance becomes : $s'_{ave} = 1.28$, $s''_{ave} = 2.38$ and $s'''_{ave} = 3.46$. For many applications the above trade off may be significant.

Higher thresholds increase the length of overflow record chains. On one hand, the storage occupied by overflow buckets is then no more negligible. On the other hand, larger overflow buckets must lead to better access performance. For higher thresholds, more stable load factor results thus from the control on $\hat{a}'$ and it is better to choose b' > 1.

The threshold corresponding to the load control on $\hat{a}'$ is denoted g'. For given values of b and of g', access performance depend on b'. If g' is higher than $\hat{a}_{max}$ of uncontrolled split, then, obviously, neither b' = 1 nor b' >> 1 can provide the best access performance. Therefore, they are b's which are the optimal ones. Fig. 3 shows curves of s' corresponding to the minimal s'_{ave} for b = 10,20,50 while g' = 0.75,0.9. It also indicates the corresponding optimal b's. Table 2 displays the performance of the corresponding stable states and the performance for g' = 0.85. All these results are obtained through simulations.

It first become apparent from the figure, that if g' = 0.75, then s' is always almost 1. It is also apparent that s' stays close to one even for g' = 0.9 ! In other words, LH does not only find a record in general in one access independently of the number of insertions, but may also use almost the minimal storage ! In particular LE achieves not only a much better access performance than B-trees but also saves more than 20 % in storage !





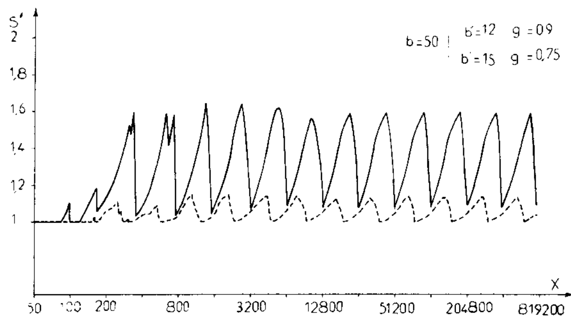


Fig.3. Mean number of accesses per successful search for linear hashing with the load kept equal to 75 % (- -) and to 90 % (——)

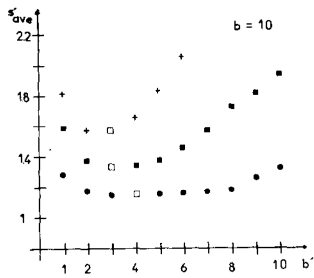
Furthermore, with respect to table 1, table 2 shows that it is rather worthwhile to choose a high threshold, even if one seriously cares about performance other than s'. For g' = 0.9, load control improves the load factor up to 31 % in mean and up to 40 % with respect to the worst value. The price to pay for such a significant improvement in load seems rather low, since, first, s''_{ave} increases only by 1.3 accesses. Next, s'''_{ave} increases only by 1.5 accesses. Only $\hat{s}_{ave}$ deteriorates more substantially, since it increases by 3.4 accesses. However, for b = 50 and g' = 0.85, this deterioration stays small, since it not exceed 1.3 accesses. Finally, $\hat{s}_{max}$ may do not deteriorate at all, being, even, for g' = 0.75, better for all each b. For b = 50 the gain is quite important, since it reaches 3.4 accesses. These gains are obviously due to b' > 1.

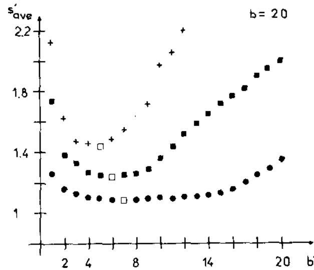
b	10			20			50			
b'	4	3	3	7	6	5	15	14	12	
g'	0.75	0.85	0.90	0.75	0.85	0.90	0.75	0.85	0.90	
s'_{ave}	1.14	1.33	1.57	1.08	1.24	1.44	1.05	1.20	1.35	
s'_{max}	1.22	1.47	1.73	1.15	1.35	1.65	1.13	1.35	1.59	(a)
s'_{min}	1.06	1.15	1.31	1.02	1.07	1.17	1.00	1.02	1.09	
s''_{ave}	1.37	1.99	2.48	1.29	1.80	2.45	1.27	1.78	2.37	
s''_{max}	1.48	2.15	2.75	1.43	2.11	2.87	1.49	2.19	2.95	(b)
s''_{min}	1.21	1.52	2.06	1.10	1.38	1.85	1.02	1.18	1.66	
s'''_{ave}	3.42	4.05	4.68	2.91	3.42	4.17	2.62	3.10	3.73	
s'''_{max}	3.67	4.71	5.43	3.11	3.60	4.43	2.68	3.38	4.15	(c)
s'''_{min}	2.91	3.29	3.73	2.51	2.82	3.25	2.27	2.43	2.91	
$\hat{s}_{min}$	5.05	5.60	6.15	5.05	5.75	5.9	5.00	5.45	6.20	
$\hat{s}_{ave}$	6.34	7.82	9.48	6.24	7.6	9.47	6.24	7.27	9.02	(d)
$\hat{s}_{max}$	7.10	9.50	11.1	7.35	8.75	11.6	7.15	8.5	10.50	

Table 2 : Performance of linear hashing with controlled load :

(a) successful search,
(b) unsuccessful search,
(c) insertion,
(d) split cost.

Fig. 4 displays s'_{ave} in function of b', with values of b and of g' from table 2 as parameters. It appears that for a file loaded up to 75 %, access performance are almost the same for a large number of values of b'. For these values, performance are, in addition, almost independent of b. For example, s'ave is smaller than 1.2 accesses, for all b' between 2 and 8 when b = 10, for all b' between 2 and 16 when b = 20 and for all b' between 2 and 50 when b = 50 ! Since practical constraints may frequently impose bucket





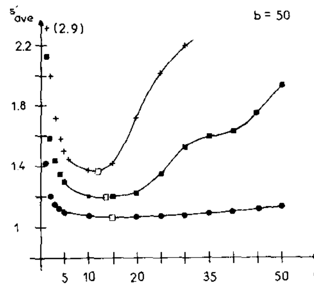


Fig.4. Mean number of accesses per successful search as a function of the size of the overflow bucket.

capacities which are not the optimal ones, this stability of excellent access performance is one more important property of LH.

The figure shows, however, that when the load becomes higher, b' should be kept closer to the optimal one. The practical rule which appears is then :

$b/5 \le b' \le b/3$.

For $b > 20$, even if the lower bound is $b/7$, we stay under a mean of 1.5 accesses.

It also becomes apparent that the access performance deteriorates less when g' increases from 75 % to 85 %, as it deteriorates when g' increases from 85 % to 90 %. In other terms, the last 5 % are the most expensive ones and it is not recommended to further increase g'.

4. CONCLUSIONS.

If a file or a table addressed with LH is static, then the performance is simply that of classical hashing, i. e., the best known. If they are dynamic, then the mean number of accesses per search stays close to 1 independently of the number of insertions and for load factors reaching 90 %. If the bucket capacity is greater than 10 records, then almost any record is found in one access. Finally, address computing is almost as simple and rapid as for classical hashing. The comparison of these performance with those of the classical hashing, of trees and even of other algorithms for virtual hashing, shows that for the search by the primary key, Linear Hashing is the best performing technique known.

High and constant load factor means that LH store records always in an almost minimal storage. A sequential search scans thus an almost minimal number of buckets, i. e., is almost as fast as possible. If the classical hashing is used, the number of primary buckets is fixed when the file is created. If the number of records is then i times less than expected, a sequential search with LH is almost i times faster. This property of LH is also important, since sequential searches are quite frequent.

With respect to trees, in addition to much faster search, LH provides much simpler algorithmic. This is, first, the case of the algorithms for a search, for an insertion and, especially, for a deletion. This is also the case of the algorithms for concurrency control, since only the key and the pointer must be locked, instead of a path in the tree. Also, there is no a problem of an inconsistency which may occur in a tree because of keys duplicated between the file and its index. Thus trees stay more advantageous only when the file must be searched in one order.

LH is, of course, primarly devoted to applications where the file may heavily grow or shrink or where the number of records is unknown when the file is created. It may thus be very useful for compilers and text processing systems. It may avoid the painful estimations of the file sizes in a DBMS, from which files the deleted records are, usually, not physically removed. It may also avoid performance deterioration for such files,

rendering thus the very annoying reorganizations of the whole database /SCH73/ unnecessary or less frequent. It is a good tool for the management of working spaces for queries to a DBMS, since the number of records retrieved by a query to a working space is, usually, unknown in advance. It may also be used for a virtual memory management. On the other hand, since it works with very small cores, LH renders dynamic files usable on micro-computers. Clearly, the applications of the Linear Hashing are very numerous.

Research on LH has just started and many possibilities are still open. Other criteria for control may be useful, /SHO79/ for instance shows that the performance may be excellent if we simply split one time for any gb insertions. M. Girault (Institut de Programmation) has suggested to consider splits and groupings as operations which are, finally, completely out of the algorithmic for a record insertion or deletion. He suggests further to leave splits and groupings to the competence of a dedicated processor which task would thus be to take care of performance of all files. This idea obviously leads to a new and interesting type of an associative memory.

Furthermore, methods other than bucket chaining should be explored for overflow record addressing. The first investigations of open addressing show that it may work pretty well /KAR79/. Also, the properties of split functions should be investigated. Finally, much work is needed in modelling, since classical methods do not apply to dynamically created hashing functions. Especially, there is no models for the transient state and for a file with small buckets.

ACKNOWLEDGMENTS.

This work was sponsored by project SIRIUS.

REFERENCES

AHO75 Aho, A.V., Hopcroft, I.E., Ullman, J.D. The design and analysis of computer algorithms, Addison-Wesley Reading Mass. 1975.

BLA77 Blake, I.F., Konhein, A.G. Big buckets are (are not) better! Journal ACM 24, 4 (Oct 1977), 591-605

CAR73 Carter, B. The reallocation of hash-coded tables. Com. ACM 16, 1 (Jan 1973), 11-14

COM79 Comer, D. The ubiquous B-trees. ACM Comp. Surv., 11, 2, (Jun 1979), 121-138.

FAG78 Fagin, R., Nievergelt, J., Pippenger, N., Strong, H. R. Extendible hashing - a fast access method for dynamic files. IBM Res. Rep. RJ2305, (Jul 31, 1978).

GHO75 Ghost, S.P.,Lum, V.Y.Analysis of collisions when hashing by division. Information Systems, 1-B (1975), 15-22

GUI72 Guiho G. Sur l'etude de collisions dans les methodes de hash-coding, CRAS 274 (Feb 14, 1972)

GUI73 Guiho, G. Organisation des memoires, Influence d'une structure et etude d'une optimisation. These de Doctorat d'Etat. Univ. Paris VI, (Jun 1973), 278.

KAR79 Karlsson, K. Resolution de collisions du hachage virtuel lineaire par une methode du type adressage ouvert. Rap. D.E.A. Inf. Institut de Programmation, (Jun 1979), 81.

KNO71 Knott, G. D. Expandable open addressing hash table storage and retrieval. SIGFIDET Workshop on Data Description, Access and Control, ACM, (1971), 186-206.

KNU74 Knuth D.E. The Art of Computer Programing, Vol 3. Addison-Wesley, Reading Mass. 1974

LAR78 Larson, P. Dynamic hashing. BIT 18 (1978), (184,201).

LAR80 Larson, P. Linear hashing with partial expansions. Proc 6-th Conf on Very Large Databases, Montreal (Oct 1980).

LIT77 Litwin, W. Auto-structuration d'un fichier : methodologie, organisation d'acces, extension du hash-coding. Res. Rep 77/11, Institut de Programmation, Paris , (Avr 1977), 102

LIT77a Litwin, W. Methode d'access par hash-coding virtuel (VHAM) : Modelisation, Application a la gestion de memoire. Res. Rep. 77/16, Institut de Programmation, Paris, (Nov 1977), 50.

LIT78 Litwin, W. Une nouvelle methode d'acces par codage decoupe a un fichier. Compte-rendus de l'Academie des Sciences, Paris, t. 286, (Avr 1978), 695,698.

LIT78a Litwin, W. Virtual hashing : a dynamically changing hashing. Proc 4-th Conf on Very Large Databases, Berlin, (Sep 1978), 517-523

LIT79 Litwin, W. Linear virtual hashing : a new tool for files and tables implementation. Res. Rep. MAP-I-021, I.R.I.A , (Jan 1979), 24.

LIT79a Litwin, W. Hachage Virtuel : une nouvelle technique d'adressage de memoires. These de Doctorat d'Etat. Univ. Paris VI, (Mar 1979), 248.

LIT80 Litwin, W. Linear hashing : a new algorithm for files and tables addressing. Proc Int Conf. On Data Bases, Aberdeen, (Jul 1980).

LUM73 Lum V.Y. General performance analysis of key to address transformation methods. Com ACM 16, (Oct 1973).

MAR77 Martin, J. Computer Datbase Organization. Prentice Hall, Inc. Englewood Cliffs, New Jersey, 1977.

ROS77 Rosenberg, A.L., Stockmayer, L.J. Hashing shemes for extendible arrays. JACM 24, 2, (Apr 1977), 199-221.

SCH73 Schneiderman, B. Optimum data reorganization points, Com ACM 16, 6 (Jun 1973), 23-28.

SHO79 Scholl, M. Performance analysis of new file organizations based on dynamic hash-coding. Res. Rep 347, I.R.I.A - Laboria, (Mar 1979), 28.

SPR77 Sprugnoli, R. Perfect hashing functions : a single probe retrieving method for static sets. Com ACM 20, 11 (Nov 1977), 841-850.