MULTILEVEL TRIE HASHING

W. LITWIN, D.E ZEGOUR, G. LEVY

INRIA 78153 LECHESNAY, FRANCE

ABSTRACT

Trie hashing is one of the fastest access methods to primary key ordered dynamic files. The key address is computed through a trie usually in core. Key search needs then at most one disk access. For very large files, trie size may however become prohibitive. We present an extension of the method, where the trie is split into subtries stored each in a page on the disk. Address computation requires the core for a single page. Two disk accesses may suffice for any key search in a Gbyte file.

I. INTRODUCTION

Trie hashing (TH) manages dynamic and ordered files of records identified by a key. The access function generated by the method is a dynamic trie whose size is usually proportional to the file size. As long as the trie is in a buffer in the main memory (core), any key search takes at most one disk access. This is usually the case, especially for files on personal computers. The method is then among the most efficient. In particular, it is usually faster than B-trees /BAY72/,/BAY77/ because of higher branching factor. Properties of the method are discussed in /TOR83/, /GON84/. /KRI84/, /DAT86/ and /L1T81, 84, 85/,

However, the requirement that the entire trie is in core may be inconvenient for very large files or when several files should be used simultaneously. Below, we describe an extension of the method adapted to these cases. The extension was called *multilevel trie hashing* (MLTH). Its principle is to split the trie into subtries small enough to be fitted in the buffer. Subtries are stored in pages on the disk, organized into a tree. Pages are created through the principle of splitting of overflowing subtries.

Below, Section II recalls at first the principles of TH. The presentation differs from the original one in /LIT8I/ by the discussion of properties of the method that appeared in the meantime. Section III defines the MLTH algorithm for trie splitting and Section IV discusses the operations on MLTII file. Section V analyzes the performance of the method. In particular, it shows that two accesses should usually suffice for a key search in Gbyte files. Section VI shows that MLTH provides a better search performance or is less sensitive to adverse key distributions than the related methods. Section VII concludes the discussion.

II. TRIE HASHING

II.1 File structure

For TH, a file is a set of records identified by primary keys. Keys consist of digits of a finite and ordered alphabet, where the smallest digit, called *space,* is denoted '__' and the largest digit is denoted ':' . All possible keys constitute the *key space*. Inside a record, only the key is relevant to access computation. Records are stored in *buckets* numbered 0,1,2,…,N that are units of transfer between the file and the core. The bucket number is called, its *address*. Each bucket may contain the same number of records called *bucket capacity* and denoted b ; $b \geq 2$.

Fig 1 shows TH file of 31 most used English words /KNU73/. The file is addressed through the trie at the fig 1-c, created dynamically by splits of overflowing buckets in the way shown later on. A trie is classically presented as an M-ary tree whose nodes correspond to digits /FRE60/, /KNU73/. This structure is however inefficient for dynamic files (/KNU73/ pp.481-485). In TH, the M-ary structure is embedded into a binary structure, as at the figure, called below if needed *TH-trie* and axiomatically defined as *Litwin trie* in /T'OR83/ TH-trie is a particular b-tree in the terminology of /KNU73/ (p. 315) that is a single node called *root* plus 0 or 2 disjoint b-trees (not to be confused with the well known B-tree /BAY72/). Each trie has an odd. number of nodes and a node usually noted *n* is either a leaf or an internal node with either 0 or 2 sons. Internal nodes form a binary tree (this is a general relationship between b-trees and binary trees/KNU73/(p, 315)),An internal node contains a pair of values called *digit value* and *digit number,* usually noted below *(d, i).* The trie is *empty* when it has no internal nodes. Otherwise, i = 0 for the root. A leaf either contains a value noted A that points to bucket A or the value called *nil* that indicates that no bucket corresponds to the leaf, Accordingly, the leaf is called leaf *A* or node *A or* nil leaf (node).

To any n corresponds a string called a logical *path* to *n* noted below C_n and defined recursively as follows:

Let (c)$_l$ be (l+1)-digit prefix of a string c (an empty string for l <0)
- If *n* is the root, then C_n =':'.
- Otherwise, let $p = (d, i)$ be the parent of n. If *n* is the right child then $C_n = \mathbf{C}_p$ else $C_n=(C_p)_{i-1}$ d.

Fig l.c shows the logical paths in the example trie,

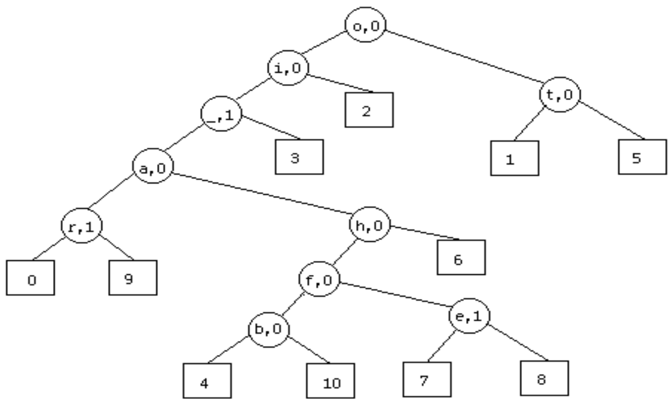
The logical paths define an M-ary structure on TH-trie called below a *logical structure,* whose internal nodes are digits and leaves are bucket addresses. Fig 2 shows the logical structure of the example trie. All *d's* with the same *i* in TH-trie constitute level *i* in the logical Structure. Each node (*d,* 0) corresponds to a (unique) digit *d* at the level 0, ordered from left to right according to the digit value order; for instance (i, 0) corresponds to digit 'i' at level 0. The edges link d_i either to d_{i+l} next on the logical path, like 'i' and '_' or to the leaf terminating the path, like 'i' and leaf 3 for instance- This M-ary structure is characteristic to tries (see Fig 31, p. 484 in/KNU73/), except that in TH leaves are pointers to buckets and not the keys themselves.

Figure 1 : Example file.

(a) : Keys sequence

the, of, and, to, a, in, that, is, i, it, for, as, with,was, his, he, be, not, by, but, have, you, which, are, on, or, her, had, at, from, this

(b) : Trie



(c) File :

are
and
a
	to
	this
	the
	that
		or
		on
		of
		not
			it
			is
			in
				by
				but
				be
					you
					with
					which
					was
						i	her
							he
							have
							had
								his	at
									as
										from
										for

0	1	2	3	4	5	6	7	8	9	10

(d) Node structure

UP
DV,DN
LP

(e) Trie representation

- 4	2	3	- 5	5	6	- 7	8	9	10
o, 0	i, 0	_, 1	a, 0	t, 0	h, 0	f, 0	e, 1	r, 1	b, 0
- 1	- 2	- 3	- 8	1	- 6	- 9	7	0	4
0	1	2	3	4	5	6	7	9	10

The basic memory representation of TH-trie is a linked list called *standard representation* /LIT8l/. Fig l.d and 1.e show the standard representation of the example trie. Each element of the list is called a *cell* that consists of four fields. Fields DV (digit value) and DN (digit number) represent an internal node of the trie. The pointer LP represents the le ft leaf or edge under this node and the pointer RP represents the right leaf or edge. A positive pointer value A, means that the field represents the leaf A. A negative value -A, represents an edge and points to cell A representing the child node.

Cell 0 represents the trie root if the root is not leaf O. The empty trie, corresponding either to the empty **file or** with bucket 0 only (after the first insertion), is represented as cell 0 with DV = ':', **DN** = 0 and LP = 'nil' or LP = 0, Otherwise, there is exactly one cell per internal node. The number of cells is also equal to that of the leaves minus one.

The standard representation is not the only one possible. Other representations exist and may have desirable properties /LIT85/

II 2 Kev search

The keys are mapped to the addresses through the logical paths, according to the following rules:

(i) all keys are mapped to the root,

(ii) let n be a node and S_n the set of keys mapped to *n*. Let p = *(d, i)* be some parent with l and *r* its left and right children. Then, S_l contains all keys *c* in S_p for which $(c)_i \leq C_l$, and all other keys of S_p

(iii) For any key, its address is the pointer reached through the application of rules (3) and of (ii).

In the example trie, all keys are thus mapped *to* node (o, 0). Then, only The keys with $(c)_0$ £ *'o'* are mapped to node (i, 0), all others are mapped to (t, 0). From $S_{(i,\ 0)}$, the keys with $(c)_0$ <= ‘i’ are mapped to (_, 1), others are mapped to leaf 2. From $S_{(\_\ ,\ 1)}$, the keys with $(c)_1 \leq$'i' go to (a, 0), others are mapped to leaf 3, etc.

These principles lead to the following algorithm for key search, determining the successive nodes on which the key is mapped:

(A1) TH key search. Let *c* be the searched key, *r* the trie root and *n* the visited node ; n = *(d, i) for* internal nodes and *n=r* initially. Let *L(n)* and R(n) be two operators providing the left and the right child of *n*. Let ***C, C'*** be string variables, C =C'=’:’ initially,

```
While B is an internal node do:
    Set C" $\leftarrow$ $(C)_{i-1}$d
    if $(c)_i$ £ C' then C $\leftarrow$ C'; n $\leftarrow$ L(n) else n $\leftarrow$ R{n)
Endwhile ; Return n,C
```

For the example trie, c = *'s'* for instance, returns n = 1 and C = 't', while c = 'he' or c = 'gun^1 return $n = 7$ and C = 'he'. If Algorithm Al ends up in a nil node, then c is not in the file. The values of C are the logical paths to the successively visited nodes. They are returned for the splitting algorithm A2 below,

While Algorithm Al is probably the simplest, one may optimize the search. The following algorithm may be faster, as it compares only one digit at the time, skips unnecessary comparisons and does not need C'. It compares at first only the leftmost digit C_0 of c and only to nodes with $i = 0$, until $C_0 = d$ or a leaf is reached. In the case of the matching, the comparison switches to C_1 and to nodes with $i = 1$ only etc. The notation and initial values are basically as in Algorithm Al.

<u>(Al bis) TH key search.</u> Let c_J denote digit j of c = $C_0C_1,..,C_J, ..C_K$. Let it be j =0 initially.

While n is an internal node do **:**

```
        If j=i then

            If cj £ d then
                C ← $(C)_{i-1}d$ ; If $c_j$ =d then set $j$ ←j+1;
                set $n$ ← L(n);
            else $n$ ← R(n)
        else
            if j <$i$ then
                set n ← L(n);C ← $(C)_{i-1}$d
            else n ← R(n)
Endwhile
return n,C
```

The search for 'he' for instance, compares at first C_0 = 'h' to digits in nodes with $i = 0$, and to only to such digits. Thus the comparison of 'h' to '_' is skipped. When (h, 0) is reached, the comparison switches to 'e' and to nodes with $i = 1$. TH key search differs thus from the usual key search in a binary search tree or in a usual trie, where the key is compared to each node value. Both algorithms compare indeed only a part of the key either to a value computed from the visited nodes (Algorithm Al) or to selected nodes (Algorithm A2), They apply the idea in hashing that is " to chop off some aspects of the key and to use this partial information as a basis for searching" (/KNU73/, vol 3, p. 507). Deeper discussion of these aspects of TH is in /LIT85/.

<u>II.3 Range queries</u>

A range query searches all keys c within some bounds c_1 and c_2 ; $c_1 < c_2$; **in** descending or in ascending order. The rule (ii) above for the key-to-address mapping to implies that the **TH** file is ordered and thus keys may be searched using only one access per bucket. more efficiently than for a classical hashing. If leaf n points to the visited bucket, then the successor of n in ascending key order is the leftmost leaf in the right subtrie of the (unique) node for which n is the rightmost leaf in the left subtrie. In the trie on Fig 1-b, the successive leaves and buckets are 0, 9, 4, 10, 7, 8, 6, 3, 2, 1, 5.

This order of leaves corresponds to the *inorder* traversal of the trie. It at so corresponds to the preorder and postorder traversals, since all these traversals coincide with respect to the leaves order. The inorder traversal is recursively defined, as : (1) Traverse the left subtree in inorder, (2) Process the root node, (3) Traverse the right subtree in inorder. Simple algorithms for inorder traversal, as well as for the other traversals are widely known /KNU73/, /TRE85/ etc. The following algorithm is derived from Algorithm T in /KNU73/, Vol 1, p 317. The notation is this from Algorithm Al and *S* denotes an initially empty stack.

(T) Inorder traversal of the trie

```
1, Set n ← r,
2.If n is a leaf. go to step 4.
3. Set S ← n, i.e., push n onto the stack. Set n ← L(n) and return to step 2.
4- [n <= Stack] Visit n. Then, if S is empty, then return else set n ← S, i.e. pop n from the stack.
5- Visit n . Then, set n ← R(n) and go to step 2.
```

The leaves in ascending key order are delivered by step 4, To visit them in descending key order, one has to traverse the trie in *converse* inorder (preorder, postorder), The converse orders simply correspond to the interchange of the words "left" and "right" in the original definitions. The corresponding conversion of Algorithm T is trivial.

One way to process a range query is therefore (i) to compute by Algorithm A1 the addresses A_1 and A2 for c_1 and c_2 and (ii) to read the buckets in the inorder of leaves between these bounds. The Algorithm T will usually visit more leaves, as it starts the traversal from leaf 0 while usually $A_1 > 0$. One may however start the visits directly from A_1. It suffices to put on a stack the nodes whose left edges were used while computing Algorithm A_1 for c_1. The stack would then contain the same nodes as stack S of Algorithm T when the traversal reaches A_1. The adaptation of Algorithm T is trivial. This strategy may speed up the query processing, though the trie traversal is a memory operation that thus should be anyhow usually much faster than bucket accesses,

To illustrate the range query processing, consider for example the search for keys *c* starting with 'h', **e.g.** 'h' ≤ c ≤ 'i' . Algorithm Al would return addresses 7 and 6. If Algorithm T is not modified, it will also compute the traversal for all nodes preceding leaf *7.* If Algorithms T and Al are modified to avoid this part of the traversal. Algorithm T will start with stack S with nodes : (o, 0), (i, 0), *(_,l),* (h, 0), (e, 1), The accessed buckets will be in both cases: bucket 7, 8 and 6, in this order .

II.4 File expansion

Insertions expand the file through the splits of the overflowing buckets. Each split usually moves about a half of keys in the overflowing bucket A, into a new bucket, appended to the file end. The move results from the trie expansion, usually by one internal node and one leaf pointing to the new bucket. The effect is that C_A decreases or that it is extended with some digits. If j + 1 is the length of new C_A, some prefixes $(c)_j$ in bucket *A* become then greater than C_A. The moved keys are these with such prefixes.

(A2) Th bucket splitting

Let C result from Algorithm Al for the key to be inserted-Let *N* be the last current address in the file. Let *B* be the ordered sequence of *b+1* keys to split, including the new key and let *c"* be the last key in *B*. Let *c'* denote a key in *B* called the split key usually near the middle of *B* .

1. |Determine the split string] Find the shortest $(c')_i$, called the *split string* that is smaller than $(c'')_i$

2. [Split the bucket] Set N $\leftarrow$ *N* + 1. Append bucket *N* and move to it all keys *c* in *B* Whose $(c)_i > (c')_i$
3. [Trie expansion] :

3.1 [Find the digits of the split siring that are already in the trie] If $i > 0$, then cut from $(c')_i$ the largest $(c')_l$; $l < i$; such that $(c')_l = (C)_l$ If $I < i - 1$, then *go* to step (3-3),

3.2 [Usual case : expand the trie by one internal node and leaf N] Replace leaf *A* with node (c', i). Attach leaf A again, as the left child of (c'_i, i). Attach leaf N as the right child of $(c'_i$, i) Return,

3-3 [Expand the trie by more than one internal node] Replace leaf A with the following subtrie:

- $(c'_{l+1}, l+1)$ is the root,

- the left child of each (C_j, j) ;j =l+1, …, i-1; is $(c'_{j+1}, {}_{j+1})$; the right child **is** a nil node.

- the left and right children of (c_i, i) are leaves A and N.

3.4 Return.

The file at Fig l.b was Created by splits generated by insertions of Fig l.a to buckets with the capacity $b = 4$, For each split, the split key position *m* in the sequence *B* was m = INT(b/2 + 1)=3. The initial file consisted of bucket 0 and of leaf 0. The first three splits were as follows :

- key 'a' generated the 1st split. The split key was "of and the split string was 'o', as it was the shortest prefix of 'of smaller than the same length prefix of 'the[1], the last key in B, Step 2 appended bucket 1 and moved to it keys 'to' and 'the', leaving thus three keys in bucket 0. Step 3 expanded the yet empty trie to node (o, 0) with two children ; leaf 0 and leaf 1.

- the insertion of key 'is' generated the 2nd split of bucket 0. The split key was then 'in' and the split string 'i'. Step 2 moved to bucket 2 key 'of', leaving thus key 'is' in bucket 0, unlike it would be the case in a B-tree split. The split resolved the collision, but left bucket 0 full- Step 3 replaced leaf 0 with node (i, 0) and appended to it leaf 0 as the left child and leaf 2 as the right one,

- Key "i" generated the 3rd split of bucket 0. Algorithm A2 chose key T as the split key, The split siring was then 'i_' since the last key of *B* was 'is' (note that the split siring was **in** this case longer that the split key itself and. that 'i' = 'i_'='i__….._') . Step 2 appended bucket 3 and moved there keys 'in' and 'is'. Step

3.1 found, for the first time, that some front digits of the split string are already in the logical path C_0 and thus in the trie, in the occurrence digit T. This digit was then cut off and the split string shortened to the single digit '_', Step 3,2 replaced leaf 0 by node (_, 1) with leaf 0 as left child and leaf 3 as the right one,

It should appear from these splits that the idea in Algorithm A2 is to minimize the number of nodes created by the split. Step (1) chooses for this purpose the shortest split string $(c')_i$. Step (3.1) removes further all front digits already in *C*. Usually, this reduces $(c')_i$ to a single digit, leading to a single new leaf that is leaf *N* and a single new internal node (d_l, *I*); / £ *j+1* ; where (j+l)is the current length of C. Otherwise, Step (3.3) creates one internal node and one leaf per remaining digit. Except for the bottom right leaf M all other right leaves are then nil leaves, as the corresponding buckets would be empty, A nil leaf is replaced with an actual address N+l at the first insertion choosing it. The corresponding bucket is then appended and the key inserted. An example of nil leaf processing is in /LIT81/.

Since a split usually adds only one digit and one address, as long as digits are atomic values, TH-trie is basically the most compact representation in a form of a binary tree of the set of split strings used for the file expansion, (we say basically, since one may optimize Algorithm A2 in a way providing sometimes shorter split strings or even avoiding some splits through a recalculation of the existing node values). To obtain a more compact representation, one has to either break digits to bits, or has to use sequential representations of the logical structure, i.e, without LP or RP /LIT81/, /L1T84/, /TOR83/. These representations are however much less efficient for internal search and dynamic modifications. Another possibility is to avoid, to use digit values at all, while calculating the splits using an interpolation like in /BUR83/, but then performance is again less efficient for internal search and. more sensible to adverse key distributions. This aspects of TH design are discussed in Section VI.2.

The keys *c* that move to bucket *N* are all those whose $(c)_i > (c')_i$. Not only the split key stays in bucket *A*, but may be some but not all keys above it in *B*, as it appeared in the example. TH splits have thus a random tendency to load bucket *A* more than it would be the case of a B-tree with the same position of the split key. If the split key is the middle one, TH splits tend to be on average asymmetric. This asymmetry has no practical effect on bucket load for random insertion, but reveals beneficial to sorted insertions (/LIT85/ and Section II.6). It makes TH splits *half random* /LIT85/ in the sense that if *c'* is chosen in the middle of the bucket, then bucket *A* surely keeps each *c* £ *c'* and thus at least half of keys. It also situates the method between these using die deterministic splits, like B-trees, and "pure" dynamic hashing methods discussed in Section VI-2 using random splits.

II.5 File contraction

Buckets and leaves A and *A'* are *siblings* if they have the same parent node. Siblings that after some deletions contain together at most *b* keys may *be* merged in the way inverse **to** splitting, freeing then bucket *A*' and shrinking the trie. Deletions may also render empty a bucket *A* that has no sibling, like bucket 6 in Fig 1. Leaf *A* is then made nil and the bucket disallocated.

II.6 Performance

Performance analysis of TH is in /L1T85/. It mainly concerns the load factor a == **x** / ((*b* (*N* +1)), where *x* is the number of keys in the file. The values of a are determined for random, ascending and descending insertions, coming unexpectedly, and noted below respectively a_r, a_a and a_d. It appears that, for the split key near the middle of the bucket, e-g, $m \approx 0.5$ b , a_r stays on the average close to 70% as in a B-tree. For the same m, a_a is within 60 - 70%, depending on *b*. This is substantially better man the well known $a_a = 50$ % of a B-tree and in particular allow to load the file through the usual insertion algorithm. Finally, the corresponding a_d is 40 - 50 %. It increases over 50 % if *m* is lowered to m ≈ 0.4 b. at the expense of a_a, but for some *m*, both a_a and a_d are *over* 50%, The reason for this surprising property that seems unique to TH is that the split asymmetry for the same *m* and keys is on the average larger for ascending insertions than for descending ones. The value of a_r is almost unaffected when *m* decreases to $m \approx 0.4$b, though it may slightly improve for some optimal *m* values under 0.5% , depending on *b*- The percentage of nil leaves is negligible, under 0,5% for random insertions- In practice, the trie grows at the rate of one cell per split and there are *N* cells in the list

When the trie is in core, any successful search for a key costs 1 disk access. An unsuccessful search costs at most 1 access, as no access is needed to a nil leaf. The practical cell size is six bytes ; two bytes per LP and RP and one byte per DV and DN. Therefore, 6K byte buffer suffices for about **1K** bucket file, while 64K byte buffer suffices for 1K bucket file- Since typical values *of b* are between 10 and 200, the corresponding TH files may contain about $10^4 - 10^6$ records. In particular, if the bucket is the MS-Dos hard disk allocation unit (cluster) that is 4K bytes, then 30K byte buffer suffices for the file covering the 20M byte disk of IBM-AT.

A typical IBM-PC compatible or a Mac has now at least 512K of core or some Mbytes. Macintosh Plus has in particular at least 32K bytes for the cache memory. The assumption that the trie is in core is therefore reasonable, especially for personal computers.

III. MULTILEVEL TRIE HASHING

III. 1 The idea

The requirement that the trie is in core may in contrast be inconvenient when the file **is** very large or many files should, be open simultaneously. MLTH is intended for these situations. It keeps TH schema for bucket splitting, but also splits the trie. The subtries are stored in buckets called *pages* structured into an M-ary tree (Fig 3). A page size may be small, usually a few Kbytes. A subtrie splits dynamically when it overflows its page. As for the buckets, page addresses are 0,1, ... but they correspond to a distinct physical area. The number of cells per page is called *page capacity* and. is noted *b'*.

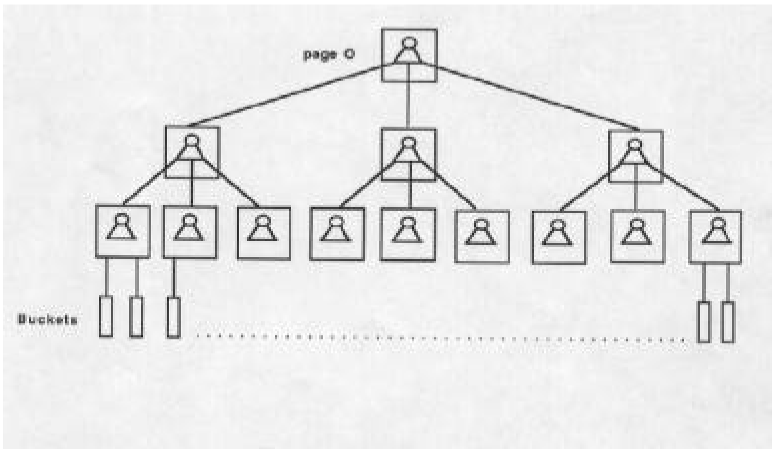


Fig 3 : Multilevel Trie

III. 2 Trie splitting

IIL2.1 Overall principles

Let *P* be the page that overflows, T the (sub)trie in P and r the root of *T.* Unless *P* = 0 that is always the root page, there is a parent page *P'* that points to *P*. The simplest way to split the trie is to move *r* to *P'* or to keep it in page 0 and to make ii pointing to two pages containing respectively the left and the right subtrie of *r*. Node r would become a *split node*. For the trie at Fig l.b, node *r*= (o, 0) would become the split node, the subtrie rooted by (i, 0) would enter the left page and the subtrie rooted by *(t*, 0) would enter the right page. However, as this trie shows, such a *straight split* could be uneven, lowering the page load to some extent for random insertions and largely for adverse distributions. One needs a method making splits usually even. The MLTH split consists of two phases :

1- *Choice of the best split node* noted r', making the number of internal nodes that precede r' in inorder in r the closest to that of nodes following r',

2. *Trie splitting using r'* that is the splitting of the (sub)trie *T* into two subtries : T'_j whose nodes precede r' in inorder and T'_r whose nodes follow r'.

The inorder keeps the trie splits ordered. The node *r'* moves to *P'* or slays in page 0. The pointers in the cell representing *r'* are set to the addresses of the pages with T'_l and T'_r becoming *page pointers*. Page pointers are positive, but they cannot be mistaken with bucket pointers, as it will appear. Details of the steps are as follows.

III.2.2 Choice of split node

We call logical parent of node *n = (d, i)* in T the node *p* in T mapped to the parent d_p of node d_n in the logical structure of T. For instance, (i, 0) is the logical parent of (_, 1) in the example trie, since node 'i' is the

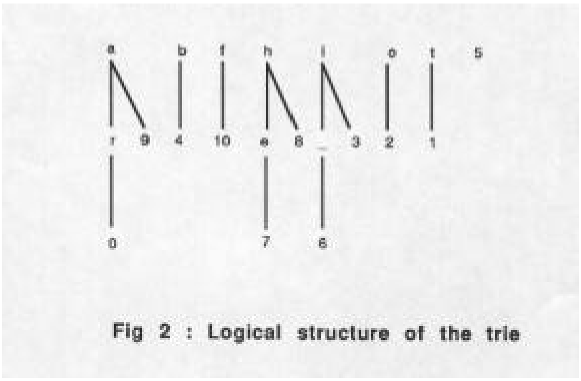


Fig 2 : Logical structure of the trie

parent of node '_' in the logical structure at Fig 2, as well as (h, 0) is the logical parent of (e, 1). No node $n = (d,$ 0) has the logical parent. For i > 0, p is an ascendant of n in T that is the parent of n if n is a left child, otherwise it is the last node on the path E from r to n whose *left* edge is in E . The former case applies to (_, 1), the latter applies to (e, 1).

We further call tries T_1 and T_2 *equivalent* if they have the same logical structure. For instance the tries at Fig 1 and Fig. 4 are equivalent. Equivalent tries provide the same key-to-address mapping, as they have the same logical paths and nodes, but may differ by edges and roots. In particular, if $r'' = (d_{r''}, i_{r''})$ is the root of T_2, then the node $(d_{r''}, i_{r''})$ logically corresponding to r'' in T_2 i.e. mapped to the same digit $d_{r''}$ in the logical structure, cannot have the logical parent P in T_2. Since node p in T_2 would indeed be a descendant of r'' in T_2, as any other node in T_2, p could not be the logical parent of r'' in T_2 and the logical structures would have to differ. For instance, any node $(d,\ i)$ with $i = 0$ in the example trie may be the root of an equivalent trie, but not a node with $i > 0$. For this trie, there is exactly seven choices for r'' , including $r'' = r =$ (o, 0).

The triplet $(T_l$, r', $T_r)$ may be considered as a trie T rooted by r' and equivalent to T, as will be shown. Let N +1 be the size of T and thus of T' and L the size of T_l. The best split node r' rooting T is the node of T that (a) minimizes !L - *N I* 2! , provided (b) it has **no** logical parent in T. At least one choice of r' always exists, namely $r' = r$. It may happen that it is the only choice.

The test of the condition (a) is trivial, but not this of (b). However, the following properties make it simple and fast:

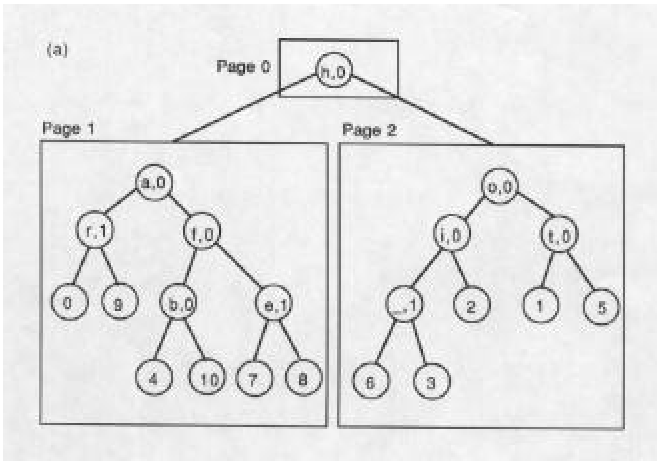
(i) - *if r'* is j -th internal node in inorder in T ;j= 0,1,2,...; then **L=j**

(ii) - if the traversal of T uses algorithm T, then the visited node n has a logical parent p in T iff p is in stack S.

(iii) - n has no logical parent if $i_n = 0$ or S is empty and it has a logical parent if $i' < i_n$, where i' is the digit number of the node at the bottom of S. Otherwise, n may have the logical parent or not depending on the trie.

Property (i) results from the definition of the best split node itself. Property (ii) holds since S contains all nodes with the left edge in E which is in particular the case *of p*. Property (iii) results from the trie expansion principles in Algorithm A2.

Figure 4 : Trie split



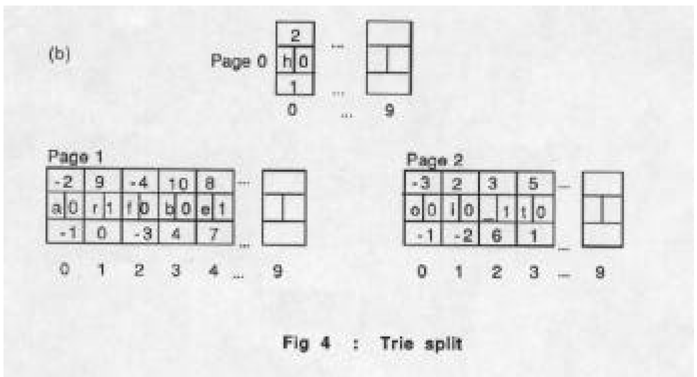


Fig 4 : Trie split

The following algorithm applies these properties to find *r'*. In fact, it computes the whole path *E* until **r',** needed for phase 2. The path is computed from stack *S* produced by Algorithm T assumed slightly modified to contain *E*. The algorithm already puts on *S* all nodes with left edge in *E* . We assume further that each node *n* stored on *S* has a flag F that is set to $F \leftarrow$ .false when *n* is pushed onto A in step 3 of Algorithm T. Then, the operation $n \leftarrow S$ in step 4 is repeated until *n* with *F* = .false. The flag is then set to $F \leftarrow$ .true and *n* is pushed onto A again. Step 5 is then performed for *n* as presently. The node *r'* itself is the last **in E.**

(A3) **MLTH split node search.** Let it be : L = -1, v =∞, *t* = *N* /2, n the visited node, i (n) the digit number of n, *i'* the digit number at the bottom of *S* if *S* is not empty.

```
While traversing T in inorder:
    If n is an internal node then :
        set $L \leftarrow L + 1$; set $v' \leftarrow |L - 1|$
        (a)  **if v** $\leq$ v' then return E
             **if** $i(n) = 0$ or S is empty then set $v \leftarrow v'$ ; set $E \leftarrow S$ **else**
        (b)  if i' $\geq$ i(n) then
                   if S has no node n' with i(n') = i(n) - 1 and F = .false then
                         set $v \leftarrow v'$ ; set $E \leftarrow S$
```

Endwhile ; Return E

The algorithm always terminates, returning E with either a node providing a better split than r or r itself. For the example trie, the execution of Algorithm A3 would be as follows :

- Node (r, 1) would be visited first. The test *of* i' would show that (r, 1) has the logical parent and v value would not change.

- visits to nodes (a, 0), (b, 0), (f, 0) would progressively decrease v .

- node (e, 1) that could correspond to one of the best values of L that are $L = 4$ and $L = 5$, would not change *v*,

- node (h, 0) would further decrease v . Node *(_,* 1) would terminate the computation and the algorithm would return E = (o, 0), (i, 0), („. 1), (a, 0), (h, 0) with thus **r'** = (h. **0).**

III. 2. 3 Trie splitting

III. 2.3.1 The algorithm

The splitting algorithm is derived from the algorithm for splitting a list into two parts whose concatenation is the original list, in /KNU73/ (p. 466-467). It constructs progressively T'_l and T'_r while traversing nodes of T in path E from r to r'. Like at Fig 4, it then assimilates the triplet *(T'_l, r', T'_r)* to a (sub)trie equivalent to T to split in the straight way (with r' as the root).

(A4) **MLTH trie splitting.** **Let** P be the page that overflows. Let N' be the last page in the file. Let n be the visited node and n' the child of n in E. Let n_j be a node called a *juncture node* in /KNU73/. **In** general, if T_1 s a trie with n_j then we say that one *concatenated* a trie T_2 with T_1 when the root of T_2 replaced n_j rendering both tries a single one. Initially, T'_l and T'_r. consist each of a single juncture node.

1. [Build T'_l **and** T'_r**] :**

- While **visiting successively each $n \# r'$ in E,** starting **from $n = r$, *do* :**
- If n' is the left (right) child then concatenate T'_r (T'_l) with n and its right (left) subtrie ; create then the left (right) child of n and set it to current n,.
- Concatenate T, with the right subtrie of r' and T'_l with the left subtrie of r'.

2. [Write T'_l and/or T'_r to the disk if they do not overflow] :

- If T'_l does not overflow then if P is not page 0, then write T'_j into P else write T'_l into page N' and set $N' \leftarrow N' + 1$.

- If T'_r does not overflow, then write T'_r into page N' and set $N' \leftarrow N' + 1$.

3. [Connect r' to the parent of T if any and to T'_l and T'_r]

- If T has the parent, in page let it be P', then read $\boldsymbol{P'}$; insert r' value into DV and DN of the first free cell $N'_{p'}$ in the buffer of P'; set $LP(N'_{p'}) \leftarrow P$ and $RP(N'_{p'}) \leftarrow N''$; set the page pointer that pointed to P within the parent cell to $P \leftarrow -N'_{p'}$.
- **If P** is **page 0, then** replace **the content of P with r' and connect** pages N' **-1 and** N'' **to '.**

4. [Split the overflowing subtrie, if any]. If either T'_l or T'_r overflows, then apply A4 *to* T'_l *or* to $\boldsymbol{T'_r}$.

5. If the buffer with **r'** does not overflow, then write it to page $\boldsymbol{P'}$ or to page 0 and return. Else apply A4 to the subtrie in the buffer.

III.2.3.2 Discussion

Fig 4 shows the split of the example trie. The algorithm worked as follows, from the initial values returned by Algorithm 3 that were : E = (o, 0). (i, 0), (_. 1)> (a. 0)> (h, 0) and thus r' = (h, 0) .

- Step 1 started with node **(o, 0) whose child (i. 0) in E is a** left child. **Therefore** the step concatenated T'_r *with* the node (o, 0) and its right subtrie in T. The left child of (o, 0) in $\boldsymbol{T'_r}$ became new n_j.

- Step 1 was then iterated for node (i, 0). Since its child in A is again a left child, this node and its right subtrie in T became the left subtrie of (o, 0) in T'_r. By the same token, the node (_, *1)* and its right subtrie in T became the left subtrie of (i, 0) in T'_r and the left child of **(_,** 1) in T'_r became new n_j.

- In contrast, (a, 0) became the root of T'_l since its child in E is a right child. Also its-left subtrie entered T'_l, while n_j became its right child.

• The left subtrie of (h, 0) in T replaced n, in T'_l and the right subtrie replaced nj in T'_r

• Step 2 wrote T'_l into page 1 and T'_r into page 2.

• Step 3 replaced the content of buffer of page 0 with cell 0 with values DV = h, DN = 0, **LP= 1,RP=2.**

- Step 5 wrote the buffer to page 0 and terminated the algorithm.

The algorithm may generate a multilevel trie, but in practice P' should stay page 0 only. Page 0 should overflow only at the first split of the trie, when the trie becomes bilevel. T'_l (T'_r.) overflows iff the bucket split that triggered the trie split appended more internal nodes to the left (right) subtrie of r than resulted for T'_r (T'_l) from the split. The corresponding recursive call to A4 is very unlikely.

It is also unlikely, although may happen that T'_l *or* T'_r consists of a single leaf. This is the case if one chooses (t, 0) as the split node in the example trie. The empty subtrie is represented in the new page by the cell that is a copy of this created for r'. However, the value of the pointer other than this pointing to the new page is set to nil. The cell is replaced with the usual one when the corresponding bucket is split.

The choice of r' did not violate the logical structure. The trie T' produced by the split preserves in addition the inorder. Therefore, all logical paths are preserved as well. The trie T' is thus equivalent to T .

Pages with bucket pointers are called *leaf pages* . Other pages are called *branch pages.* The number of pages that a search examines until it reaches a page is called the *page level.* Leaf pages are all at the same level called *the file level.* MLTH file is in this sense *balanced,* as is a B-tree file. TH file level is of course 0.

Finally, one may observe that the basic action of the algorithms A3 and A4 on T as the whole, may be seen as its transformation into to a better balanced T'. The process balances the respective sizes of T'_l and of

T'_r as equally as possible, but does not attempt **to** further balance each subtrie. This would have no effect on the page load factor, only on the trie traversal time that is usually anyhow very small compared to the disk access time. Also, experiences show the subtries are usually already rather good. If one wants nevertheless the best balancing, then the couple A3 - A4 should be applied recursively to the subtries. The corresponding extension is obvious. It may also be applied to the balancing of TH trie as well, as the algorithm proposed in /TOR83/. It remains an open question which algorithm is more efficient. It is however likely that it should be ours, as there is no transformation of T into a canonical form prior to the balancing process.

IV. OPERATIONS ON MLTH FILE

IV.l Search and insertion.

The search moves to another page when a positive (page) pointer is encountered as long the level of the visited page is smaller than the file level. The page at the file level must be a leaf page and the pointer is a bucket pointer. With respect to insertions, the only new feature consists of page splits.

IV.2 Ranye query

Range queries that exceed one bucket require the determination of the address of the bucket that follows or precedes the current one with respect to key order. This address may be determined in two manners :

(a) - from the trie. The next bucket to be searched corresponds to the next leaf (in inorder or converse inorder etc.). Usually this leaf is within the same page and sometimes within the sibling leaf page. The general way to find the sibling page is to visit upper level page(s). The root page 0 should be the only such page in practice. The address of the parent page may be found also in stack S.

(b) - from the chain linking logically consecutive buckets, same as this frequently implemented for a B-tree. Accesses to the trie are then no longer needed, except of course for the first key search.

IV.3 Deletions

Deletions may merge sibling pages. Let P' be the page that contains the internal node to be replaced with the value of its left child because of the bucket merge. Let P be its parent page that is the page that contained the split node (thus P' is not the root page). Finally let P'' be the sibling page of P' with respect to P. The following situations may occur:

- The number of cells in P' and P" is together less than b' and there is no other page at the file level. P' and P'' are then merged into P'. In addition, the split node moves from P into P'. All the corresponding pointers are updated in consequence.

-P' *and* P'' are the last ones at the current file level and their cells may jointly enter the free space in P.

The cells move then to *P* . The pointers within *P* are updated in consequence.

Deletions may render the page load uneven, although this phenomena should not have practical importance. In particular, one may rebalance the set of nodes in sibling pages, applying the split algorithm to the set of nodes in both pages and to the parent node. A new parent node may result and a better distribution.

V. PERFORMANCE

V.I Load factor

The paging does not influence the bucket load factor. Bucket load for MLTH is thus this of TH. The pages themselves are buckets of a dynamic file whose records are cells. For random insertions the load factor α'_r should thus be on the average around 65 - 70%, as page splits should be usually even. Fig 5 shows the shape of α'_r as a function of the number of insertions *x* and its average values obtained by simulations, for three typical values of *b'*. These values correspond to the page size *p* = 0.5, 1, 2 Kbytes. As one could expect, the average values of α'r are almost independent of *b* and *b'* values, though only *b = 5,* 10 appear at the figure. The split balancing improves the average load by 5 - 8 %, the load factor for straight splits only, being noted α'n at the figure. The α'_r curve shape shows the usual periodical behavior in $\log_2$ (x) and oscillations due to the tendency of pages to be filled up and to split rather simultaneously. The curve shape for other values of *b* and of *b'* is very similar.

For sorted insertions, the average values of α' are between 40 - 80. Usually, one has $\alpha'_a > \alpha'_d$ but not always. Generally, the actual root was the only choice for the split node for ascending insertions, e.g. split balancing was inoperative. This, because of the structural condition (b) in section III.2.2, despite the existence of nodes better respecting the condition (a) alone. The effect is nevertheless usually positive, as it makes $\alpha'_a \approx 60$ % and sometimes much higher. The large variation of α'_a values for the same *b',* but different *b* values, including *b* = 20, 50 not shown at the figure, was confirmed by several simulations. It seems to indicate a complex relationship between the sorted file size and page and bucket sizes, not yet understood enough.

Unlike for ascending insertions, the split balancing revealed crucial for descending ones. Instead of being stable usually around 45 %, α'_d had tendency to decrease towards almost zero. **To** obtain higher values of α'_d or of α'_a, for instance for the initial loading, the choice of the split node should enlarge the right or the left subtrie.

V.2 Fanout

We call a fanout of level *i* ; i =0,1,2,...; the average number of records filling up the file of level *i.* The fanout will be denoted *x(i).* For the usual random insertions the fanout of MLTH file is about :

x(i) » *b* (0.7 b) $^{i+1}$. **(V.I)**

Indeed, since α » 0.7 then the fanout of the root page is :

$\boldsymbol{x}$ **(0)** = 0.7 b b'.

as this page may contain up to b' cells. Then, the value of $x(l)$ corresponds to the root

page full and to the average load of the other pages equal to about 70 %. Thus :

$x\,(1) = 0.7b\, b'\,(0.7\, b') = b(\,0.7b'\,)^2$

etc.

Two byte pointers suffice for files of 2^{15} = 32K buckets. The corresponding cell size is six bytes. Let p be the page size in Kbytes. It results from (V.I) that for 32K bucket file, p = 1.3 usually suffices for two level trie. Indeed, as:

32K » 0.7 $b'^{\,2}$ = 0.7 $(Kp)^2$ / 36, so: p »1.3.

To provide a buffer of this size is generally easy. **It** also means that a larger page for such a file is unnecessary, unless one wishes to have page 0 only. Assuming the MS-Dos 4K byte clusters for buckets, it furthermore means that the file may span over 130 Mbytes. Finally, assuming b between 10 and 100 as usually, means that the file may grow up to at least 220 000 records and even 2 200 000 records.

For larger files, the pointer size may be three bytes. Such pointers suffice for 1.6M bucket files. This size probably suffices for the largest known files. Cell size is then eight bytes. The fanout becomes about:

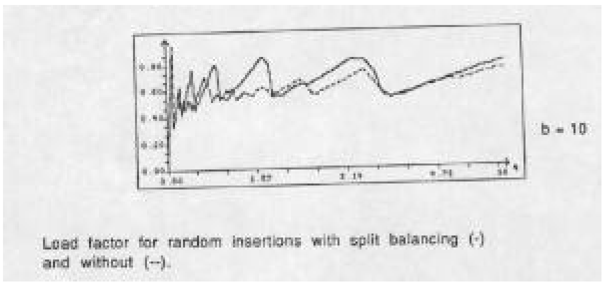
$x\,(i)$ » b $(90p)^{i+1}$.

Modest b = 20 and p = 10 suffices thus for a bi-level file of almost 16 million records. A larger page like p = 32 , leads to such a file of 160 million records. Then, p = 64 leads to a more than six hundred million record file etc. In particular, page and bucket sizes equal to the MS-Dos 4Kbytes lead to the file spanning over 0.75G bytes. This size suffices not only for IBM-PC magnetic disks, but also for the current optical disks (about 500M to 1G byte per disk).

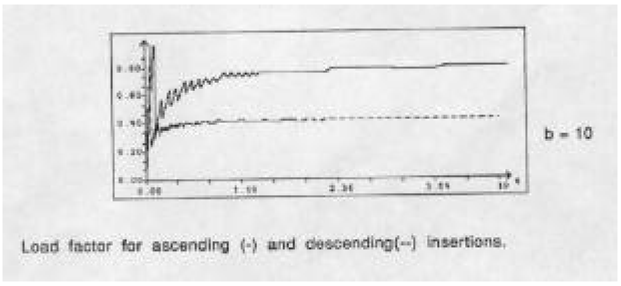
V.3 Access performance

The root page may be assumed in core. The successful search for a key needs then i + 1 (disk) accesses, in practice two accesses for the largest files, and the unsuccessful one at most i+l accesses. As nil leaves should be rare, the average cost of an unsuccessful search should be in practice two accesses as well.

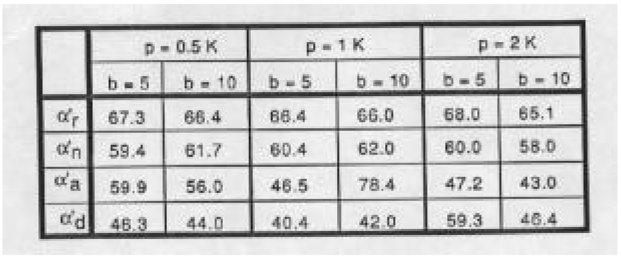
hi general, the formula of various access costs to MLTH file in the case of the random insertions are those known, of a B-tree with the same fanout and the same splitting policy. However, they lead to larger pages for a B-tree for the same fanout or to a better performance of MLTH file for the same page size, because of the larger fanout. The reason for that will be discussed.



a)



b)



	p = 0.5 K		p = 1 K		p = 2 K	
	b = 5	b = 10	b = 5	b = 10	b = 5	b = 10
α'_r	67.3	66.4	66.4	66.0	68.0	65.1
α'_n	59.4	61.7	60.4	62.0	60.0	58.0
α'_a	59.9	56.0	46.5	78.4	47.2	43.0
α'_d	46.3	44.0	40.4	42.0	59.3	46.4

Fig. 5 : Page load factor

a) Curves for p = 1

b) Average load factor for random and sorted insertions

VL COMPARTSON TO RELATED METHODS

VI.1 R.trees

We will call **B-cell** data produced by one split within a B-tree file, or within a B*-tree or within a prefix B-tree,... A B-cell contains mainly a record, or a key or a part of a key like a separator /BAY77/, then a pointer and a field indicating the cell length or end, unless the main field is of fixed length. Given the practical sizes of records, of keys and of separators, a B-cell is usually larger than that of the MLTH cell. The cell is largest for a basic B-tree, where it usually contains the whole record, and the smallest for a prefix B-tree. **For** random insertions, the load factor α'_r of a B-tree (B*-tree,...) is about the same as that of MLTH, i. e. near 70%. Prefix B-tree is an exception, since the separators length optimization may decrease α'r. For sorted insertions, the load factor is also usually higher for the ascending case and only a few percent lower for the descending ones. The same choice of page and bucket sizes leads thus for MLTH to a fanout generally larger than this of a B-tree. More precisely, it appears from (V. 1) that if B-cell is *m* times larger than TH-ceIl, then for the same page size the fanout of MLTH is in general at least m^{i+1} times larger. For a B-tree the keys are usually about 20-40 bytes, which means that two level MLTH file may be about 10 to 45 times larger. For a prefix B-tree, $m = 2$ should be quite typical, which means that two level MLTH file may be 4 times larger. For smaller files, the disk access costs of MLTH should then be at most those of a B-tree. For larger files, the access costs of MLTH should be smaller by at least one access. Internal search should be faster as well, especially if Algorithm Al bis is applied.

While a prefix B-tree minimizes *m* among B-trees, it introduces algorithmic constraints with respect to both TH and a usual B-tree. The search is more cumbersome, as the cells are of variable length. The binary search in the page requires the existence of the delimiter noted '*' in /BAY77/ that must not appear inside a separator or a pointer, while modem applications use usually the full ASCII code for separators. Finally, the deletion is complex, as the merge may expand the separators, triggering overflows /BAY77/. Usually, one knows whether the merge is possible only after recalculating the separators. For all these reasons, while B-trees and their basic variants are among data structures most used, as far as we know no commercial system uses prefix B-trees (except might be the new recently announced system Merkur).

B-trees have several variants /KNU73/, /YA078/, /COM79/, /LOM81/, /ROS81/, /SH085/,.... Each variant optimizes some performance, usually the value of a_B. However, the ideas in these improvements may also be applied to MLTH and should lead to similar improvements.

VI.2 Dynamic hashing.

Among various dynamic hashing access methods, /LAR78/, /LIT78,80/, /FAG79/,..., some are designed for ordered files. Most known are probably the interpolation hashing (IH) /BUR83/, the grid file (GF) /NIE84/ and the bounded disorder method (BD) /LIT86/. Although IH and GF are basically designed as multikey access methods, one may apply them to primary key ordered files as well.

IH may provide higher load **factor** than MLTH, as it may attain about 90%. It may also **attain a** better average search cost, which may be under two accesses. However, while no key search in MLTH may need more than two accesses, keys in overflow chains of IH need several accesses. Finally, IH does not support sorted insertions and is more sensitive to adverse distributions.

GF load factor is similar to that of MLTH. There is no overflow record and so search cost is also two accesses. However, the method is also more sensitive than MLTH **to** adverse distributions. Insertions may then lead to an exponential growth of the directory.

The nature of BD is different from GF and from IH, since it may be used in connection with any principle of indexing. The principle of BD is indeed to replace in an access method the concept of a bucket pointed by an index cell, with the concept of a multibucket node addressed internally through hashing. The fanout may then increase and so the search costs may decrease. The price to pay is a higher cost of splits and of range queries, since there is no order within the nodes.

One way to compare BD and MLTH is to consider that BD is applied to TH instead of trie splitting. The load factor of BD should therefore be a few percent smaller than this of MLTH, as BD nodes cannot be filled up to 100%. The average search cost may in contrast be smaller, close to one access per search. However, the search cost through overflow chains within nodes may be much higher than with MLTH. When the index attained the maximal size (one page), node sizes grow indeed at least proportionally to the file size.

VTT. CONCLUSION

Multilevel trie hashing is a new access method for large ordered and dynamic files. The method applies when the usage of the basic trie hashing is inconvenient. **It** is based on sophisticated properties of tries, which do not seem to be known yet. Performance analysis shows that two accesses per key search suffice for largest practical files. This makes the method usually faster than a B-tree. As the algorithms are simple to implement and fast, the method should reveal among the most practical known.

A prototype implementation of MLTH in Pascal is presented in /ZEG86/. Present work concerns deeper behavior analysis under various conditions. Future work should concern the concurrent usage of MLTH files and the recovery, on the basis of /ELL83/ and /TR083/. Also, one should study the design of variants optimizing the load factor, applying the ideas of "overflow", of uneven splitting etc. that worked well for B-trees. Finally, one should extend MLTH to the multi-key case.

Acknowledgements

We thank Dr. Vidyasankar for helpful comments,

REFERENCES

/BAY72/ Bayer, R., Me. Creight, E. Organization and maintenance of large ordered indexes. Acta Informatica, 1, 3, (1972), 173-189

/BAY77/ Baycr.R., Unlcrauer, K. Prefix B-Trees. ACM TODS. 2,1,(Mar 1977), 11-26.

/BRI59/ Briandais (de la), R. File Searching Using Variable Length Keys. Proc. of Est. Joint Comp. Conf, 295-298.

/BUR83/ Burkhard, W. Interpolation-Based Index Maintenance. PODS 83.ACM, (March 1983), 76-89.

/COM79/ Comer, D. The ubiquitous B-tree. ACM Comp. Surv. 11, 2 (June 1979), 121-137.

/DAT86/ Date, C., J. An Introduction to Relational Database Systems. 4-lh cd,, Addison-Wesley, 1986,639

/ELL83/Ellis, C., S. Extendible Hashing for Concurrent Operation and Distributed Data. PODS 83. ACM, (March 1983), 106-116.

/FAG79/ Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R. Extendible hashing - a fast access method for dynamic files. ACM-TODS, 4, 3, (Sep 1979). 315-344.

/FLA83/ Ph. Flajolet; On the Performance Evaluation of Extendible Hashing and Trie Searching. Acta Infonnalica, 20, 345-369 (1983).

/FRE60/ Fredkin, E. Trie Memory, CACM. 3,490-499.

/GON84/Gonnet, G., H. Handbook of ALGORITHMS and DATA STRUCTURES. Addison-Wesley, 19S4.

/KNU73/ Knuth, D.E. : The Art of Computer Programming. Addison-Wesley, 1973.

/KRI84/ Krishnamuny, R., Morgan S., P. Query Processing on Personal Computers - A Pragmatic Approach, VLDB-84, Singapore (Aug. 1984), 26-29.

/ION81/ de Jonge, W., Tanenbaum, A., S., Van de Riet R. A Fast, Tree-based Access Method for Dynamic Files. Rapp IR-70, Vrije Univ. Amsterdam, (Jul 1981), 20.

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

/LAR82a/ Larson. P., A. A single file version of linear hashing with partial expansions. VLDB 82, ACM, (Sep 1982). 300-309.

/LIT78/Litwin, W. Virtual hashing : a dynamically changing hashing. VLDB 78. ACM, (Sep 1978), 517-523.

/LIT80/Litwin, W. Linear hashing : A new tool for files and tables addressing. VLDB 80, ACM, (Sep 1980),
212-223. /UTSl/ Litwin.W. Trie hashing. SIGMOD 81. ACM. (May 1981).
19-29.

/LIT84/ Litwin, W. Data Access Methods and Structures to Enhance Performance. Database performance. Slate of the An Report 12:4. Pergamon Infotech, 1984,93-108.

/LIT85/ Litwin, Witold. Trie hashing : Further properties and performances. Int. Conf. on Foundation of Data Organisation. Kyoto, May 1985. Plenum Press.

/LIT86/ Litwin, W., Lomet, D. Bounded Disorder Access Method. 2-nd Int. Conf. on Data Eng. IEEE, Los Angeles, (Feb. 1986).

/LOM79/ Lomel, D... B. Multi-table search for B-tree Files. ACM-SIGMOD, 1979, 35-42.

/LOM81/ Lomet, D. Digital B-trees. VLDB 81. ACM, (Sep 1981), 333-344.

/LOM83a/Lomet, D. Bounded Index Exponential Hashing. ACM TODS, 8, 1. (Mar 1983), 136-165.

/MUL81/ Mullin, j., K. Tightly controlled linear bashing without separate overflow storage. BIT, 21,4, (1891), 389-400.

/NIE84/ Nievergelt.J., Himerberger, H., Sevcit, K.. C. The Grid File: An Adaptable, Symmetric Multi-key File Structure. ACM TODS, (March 1984).

/ORE83/ Orensiein, J. A Dynamic Hash File for Random and Sequential Accessing. VLDB 83, (Nov 1983). 132-141.

/OUK83/ Ouksel, M. Scheuerman, P. Storage Mapping for Multidimensional Linear Dynamic Hashing. PODS 83. ACM, (March 1983), 90-105.

/RAM84/ RamamonohanaraO, K., Sacks-Davis, R. Recursive Linear Hashing. ACM-TODS, 9, 3, (Sep. 1984).

/REG82/ Regnier, M. Linear hashing with groups of reorganization. An algorithm for files without history. In Sheuennann P. (ed) : Improving Database Usability and Responsiveness, Academic Press, (1982). 257-272.

/ROS81/Rosenberg, A., L.,Snyder,L. Time and space optimality in B-trees. ACM-TODS, 6,1 (1981), 174-193.

/SAM84/ Samet, H. The Quadtree and Related Hierarchical Data Structures. ACM Computing Surveys, 16, 2 (June 1984), 187-260.

/SCH81/ Scholl, M. New File Organizations Based on Dynamic Hashing. ACM TODS, 6, 1, (March 1981), 194-211.

/SH085/ Shou-Hsuan Stephen Huang. Height-Balanced Trees. ACM TODS, 10,2 (1985), 261-284.

/TAM82/ Tamminen, M. Extendible hashing with overflow. Inf. Proc. Lett. 15, 5,1982,227-232.

/TOR83/ Torenvliet, L., Van Emde Boas, P. The Reconstruction and Optimization of Trie Hashing Functions. VLDB 83, (Nov. 1983), 142-157.

/TRE85/ Tremblay, J-P., Sorenson, P., G. An Introduction to Data Structures. 2-nd ed., McGraw-Hill, 1984, 861.

/TR081/ Tropf, H., Herzog, H. Multidimensional range search in dynamically balanced trees. Agnew. hif.2, 71-77.

/WIE83/ Wiederhold, G. Database design. Mcdraw-hill Book Company, 1983.

/YA078/Yao, A., C. On random 2-3 trees. Acta Inf. 18, (1983), 159-170.

/ZEG86/ Zegour, D . Implementation du hachage digital multiniveaux. Techn. Rep.. (Sep. 1986). INRIA.