SRC TR 89-13
UMIACS TR 89-11
CS TR-2189

Trie Hashing with Controlled Load

by

W. Litwin, N. Roussopoulos,
G. Levy and H. Wang

Trie Hashing with Controlled Load

W. Litwin, N. Roussopoulos
UMIACS/SRC, Univ. of Maryland, College Park Md 20742, USA

G. Levy, H. Wang
INRIA 78153 Le Chesnay, France

ABSTRACT

Trie hashing is an access methods to primary key ordered dynamic files. The key address is computed through a trie. Key search needs usually one disk access since the trie may be in core and needs two accesses for very large files, when the trie has to be on the disk. We present a new variant of the method that allows to set up an arbitrary load factor for ordered insertions. In particular, one may create compact files, loaded up to 100 %. We show that the capabilities of trie hashing make the method preferable to a B-tree by most of criteria that motivated the latter method supremacy over the database world.

1. INTRODUCTION

Trie hashing (TH) is an access method to dynamic and ordered files of records identified by a key. The records are stored in buckets. The access function of the method is a dynamic trie whose size is proportional to the file size. The trie results from splits of buckets that overflow. It is usually compact enough to fit a main memory, especially for medium size files, typical on workstations. Key search takes then at most one disk access. For larger files, the trie may be stored on the disk as a dynamic multilevel hierarchy of subtries, called Multilevel Trie Hashing (MLTH). Each subtrie occupies one page whose size is usually a few Kbytes. Because of the high branching factor, two levels usually suffice for a Gbyte file, leading to two accesses per any key search, provided that the root page is in main memory. These properties make the method among the most efficient and usually faster than B-trees /BAY72/, /BAY77/. Properties of the method are discussed in /TOR83/, /GON84/, /KRI84/, /DAT86/, /LIT81, 84, 85, 87/ and /OTO87/.

Under random insertions, the bucket load factor of TH and MLTH file is about 70 %. The method also supports ordered (sorted) insertions. The load factor depends on the bucket size, key distribution and the parameter of the splitting algorithm, called split key position. If the split key is near the middle key of the bucket, the usual position to generate even splits for random insertions, then the load factor is between 50 and 70 % for the ascending order of incoming keys and between 40 - 55 % for the descending order. In comparison, it is 50 % for a B-tree.

If one knows in advance that insertions are ordered, as for the initial loading, then the split key may be intentionally shifted above the middle key for ascending insertions, or below it for the descending ones to increase the load factor. Simulations showed that TH could than provide in general the load factor between 70 -80 %. They also showed that the exact value is hard to predict. A B-tree performs better with respect to this aspect. The bucket load factor is the linear function of the split key position and one easily sets up a given load /ROS81/. The load factor may further attain 100 %, provided the split key is the highest key in the bucket for ascending insertions and the lowest for the the descending ones. Such a *compact B-tree* /ROS81/ is not useful for dynamic applications with random updates, as even a few random insertions may decrease the load near 50 %. However, it is useful for files that are dynamically created, but remains afterwards static or are thrown away at the end of a transaction. Modern database applications need such files, for the processing of selections and joins, back-up copies, versions, file transfer between distributed sites,...

Below, we present a refinement called *Trie Hashing with Controlled Load* (THCL). It is designed to control the load factor of TH file as tightly as of a B-tree file. It allows to fix the bucket load for ordered insertions to any predefined value that may reach 100 %. Furthermore, THCL guarantees no less than 50 % load for deletions. It allows also to use redistributions of keys between existing buckets before a new one is appended. The load factor of THCL file may then increase also for unexpected ordered insertions ie when the split key was not shifted. The corresponding value is that of a B-tree file with the same redistributions ie even up to 100 %. They also increse it for random insertions, again to the same values as in a B-tree ie up to 87 %.

The main conceptual change in THCL with respect to the basic TH is that there is no nil nodes and that several trie leaves may point to the same bucket. The modification is simple to implement and does not change key search performance. It may require additional accesses during a page split, but the increase is marginal. The trie size may become moderately larger, especially for high load, but in practice does not change TH access performance and remains usually much smaller than the B-tree that would be required for the same file. The refinement makes TH more ubiquitous and especially more suitable than a B-tree for most applications. It also brings out interesting properties of tries.

Below, Section 2 recalls the principles of TH. Section 3 discusses the file behavior. Section 4 introduces THCL. Section 5 compares the method to B-trees. Section 6 concludes the discussion.

2. PRINCIPLES OF TRIE HASHING

2.1. File structure

For TH, a *file* is a set of records identified by primary keys belonging to some *key space* of all possible 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 ':'. Inside a record, only the key is relevant to the address computation. Records are stored in *buckets* numbered 0,1,2,...,N that are units of transfer between the file and buffers in main memory. The bucket number is called its *address*. Each bucket may contain up to a fixed number of records called *bucket capacity* and denoted by b ; $b \geq 2$.

Fig 1 shows a 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 inefficient for dynamic files (/KNU73/ pp. 481-485). In TH, the M-ary structure is embedded into a particular binary data structure, called below *TH-trie* or simply *trie* whenever no confusion is possible. TH-tries are introduced in /LIT81/ and axiomatically defined in /TOR83/. TH-trie has an odd number of nodes and each node (usually noted n) is either a leaf or an internal node with either 0 or 2 sons. 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. If the trie is not empty, then $i = 0$ for the root. A leaf either contains an address A that points to bucket A or the value *nil* indicating that no bucket corresponds to the leaf. Accordingly, the leaf is called leaf or node A or nil leaf (node).

To any node n in TH-trie corresponds a string called a *logical path* to n. It is noted below C_n and is 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 $C_n = C_p$ else $C_n = (C_p)_{i-1}d$.

Fig 1.c shows the logical paths in the example trie. In particular, we say that node n' is a *logical child* of node (d, i), if (i) n' is a descendant of (d, i) such that $C_{n'}$ ends up with d as i-th digit, and (ii) n' is a leaf or of the form $(d', i+1)$. Thus (_, 1) is the logical child of (i, 0) and (e, 1) is the logical child of (h, 0), while (i, 0) is not the logical child of (o, 0). In turn, (d, i) is the logical parent of n'. Note that the logical path through the left edge of the the logical child, if any, is of the form *..dd'..* .

The logical paths define the M-ary structure embedded into TH-trie. This structure is characteristic to tries, except that in TH leaves are pointers to buckets and not the keys themselves (see Fig 31, p. 484 in /KNU73/). It is called the *logical structure* and Fig 2 shows it for the example trie. The internal nodes are digits and leaves are bucket addresses. 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 logical parents and children.

The basic memory representation of TH-trie is a linked list called *standard representation* /LIT81/. Fig 1.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) store the value of an internal node of the trie. The pointer LP represents the left leaf or the edge to the left child and the pointer RP represents the right leaf or edge. A positive pointer value A represents the leaf A. A negative value $-A$, represents an edge and points to cell A representing the child node and its leaves or edges. Cell 0 represents the trie root if the root is not leaf 0. The empty trie, corresponding either to the empty file or bucket 0 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.

2.2. Key search

The keys are mapped to the corresponding addresses through the logical paths. The rules are as follows :

(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 S_r all other keys of S_p.

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

In the example trie, all keys are thus mapped to node (o, 0). Then, only the keys with $(c)_0 \le$ 'o' are mapped to node (i, 0), all others are mapped to (t, 0). From $S_{(i, 0)}$, the keys with $(c)_0 \le$ 'i' are mapped to (_, 1), others are mapped to leaf 2. From $S_{(\_, 1)}$, the keys with $(c)_1 \le$ 'i_' go to (a, 0), others are mapped to leaf 3, etc. Logical paths partition thus the key space. This partition preserves the order and thus TH supports range queries. See /LIT88/ for the range query processing discussion.

One way to find a key is to determine the successive nodes on which the key is mapped, until a leaf is found. This is done, particularly efficiently, by the following algorithm (see /LIT88/ for another algorithm). The algorithm returns also the value of the logical path to the leaf, used by the splitting algorithm presented later on.

(A1) TH key search. Let c be the searched key, r the root, n the visited node ; $n = (d, i)$ for internal nodes. Let L(n) and R(n) be two operators providing the left and the right child of n. Let C be a string variable ; C = ':' initially. Let c_j denote digit j of $c = c_0c_1...c_j...c_k$.

```
$n$ <- $r$ ; $j = 0$
While $n$ is an internal node do :
if $j = i$ then
  if $c_j \le d$ then
     set $n$ <= L($n$) ; $C$ <= $(C)_{i-1}d$ ;
  if $c_j = d$ then set $j$ <= $j + 1$ 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, where the key is compared to each node value. Algorithm A1 compares in contrast only a single digit and to selected nodes. It applies the idea in hashing "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 this aspect of TH may be found in /LIT88/ and /LIT85/.

2.3. Bucket splitting

The file and the trie expand the file through the splits of the overflowing buckets. The algorithm is as follows.

(A2) TH bucket splitting. Let A be the overflowing bucket and C its logical path. Let N be the last current bucket 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. [Find the split string] Cut from the split key the shortest prefix $(c')_i$ called the *split string*, that is smaller than the $(c'')_i$.

2. [Split the bucket] Set $N <= N + 1$. Append bucket N. Move to N all keys c in B where $(c)_i > (c')_i$.

3. [Expand the trie] :

3.1 [Cut the digits of the split string that are already in the logical path, if there is some] If $i > 0$, then cut from $(c')_i$ the largest $(c')_l$ such that $(c')_l = (C)_l$.

3.2 [Usual case : only one digit is new which is c'_i . Expand the trie by the internal node representing c'_i and by leaf N] If $i = 0$ or $l = i - 1$, then do :

- Replace leaf A with node (c'_i, i).
- Attach leaf A again, as the left child of (c'_i, i) and attach leaf N as the right child of (c'_i, i).

3.3 [Rare case : several digits that are digits $c'_{l+1}..c'_i$ remain in the split string. Expand the trie by internal nodes representing these digits, by leaf N and by some nil nodes]. Otherwise, replace leaf A with the 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 .

Return.

The split strings define a new logical path to A that is smaller than the old one. Usually both paths coincide for all digits, but the last one ie c'_i , whose value decreases for the new path. The keys moving to bucket N are these whose i-th digit prefix exceeds the new path. This set contains usually about half of keys in B which are all greater than those staying in bucket A. That is why, TH file is ordered and supports range queries.

The example file at Fig 1 has buckets with the capacity $b = 4$. The split key position m is $m =$ INT(b /2 + 1) = 3. The initial file consisted of bucket 0 and of leaf 0. The insertions that generated splits are underlined. Fig. 2-5 in /LIT81/ shows the first three splits. Fig. 3 here shows the split triggered by insertion of key 'hat' to bucket 7 of file at Fig. 1. The key 'have' is the split key, as it became 3rd key within B . The split string, underlined, is 'ha'. Since 'h' is already in the logical path 'he' to bucket 7, the only new internal node for the trie is (a, 1).

Nil nodes avoid the allocation of empty buckets when a split appends several internal nodes to the trie. A nil leaf is replaced with an actual address N+1 at the first insertion choosing it. The corresponding bucket is then appended and the key inserted. An example of nil leaf will be discussed in Section 3.2, see also /LIT81/. The creation of nil nodes in Step 3 was isolated as a separate step for didactic purpose, in practice Steps 3.1 and 3.2 may obviously be combined into a more efficient specification.

The keys c that move to bucket N are all those whose $(c)_i > (c')_i$. Not only the split key c' stays in bucket A, but may be some but not all keys above it in B . This would be the case of key 'have' at Fig. 3, if one chose $m = 2$, ie 'hat' as the split key. 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 3.2 below). It makes TH splits *partly random* /LIT85/ in the sense that bucket A surely keeps each $c \leq c'$, but the decision is random for some keys above c' . Splits of other methods for dynamic hashing are fully random, as all keys may stay or move. In contrast, B-tree split is deterministic, as any c above c' moves to the new bucket, while all others remain. Thus TH principles position the method somewhere between tree based methods and usual dynamic hashing methods /ENB88/.

2.4. Bucket merging

Buckets and leaves A and A' are *siblings* if they have the same parent node ie share a cell like 0 and 9 for instance. 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 freed.

The shrinking of the trie may correspond to the physical shrinking of the table of cells, through the move of the last cell to the empty one. Another approach is to only mark deleted leaves through a special value. This is preferable for the efficiency of the concurrency control, /VID87/.

2.5. Trie splitting

When the trie becomes too large for the main memory, it may be split, leading to the *multilevel trie* /LIT88/. The splits create then a hierarchy of *pages* containing subtries, as it is shown at Fig. 4. Pages split when they overflow. They constitute *levels* of the same depth with respect to the root page. All leave nodes are in pages of the lowest level, called *file level.* While this schema called *multilevel trie hashing* (MLTH) looks like an organization of the trie into a B-tree, splitting algorithm differ because of structural constraints of the trie. Details are in /LIT88/, guidelines that follow suffice here.

Each page split consists of two phases. The 1st phase is called *choice of split node*. The 2nd phase is called *trie splitting*. The split node, let it be *r'*, is a node respecting the following conditions :

(i) - the number of internal nodes that precede *r'* in inorder in the subtrie to split, let it be *T*, is the closest to that of nodes following *r'*.

(ii) - *r'* has no logical parent in *T*.

The split node may nevertheless have a parent in *T* or a logical parent in the subtrie of upper level. The root of *T* always respects (ii). It becomes thus the split node if there is no better choice.

The trie splitting phase moves *r'* to the upper level (parent) page and creates a new page (if *T* is in the root page, then a new root page is created). The corresponding cell is appended to these already in the page. The left pointer of the cell is set to point to *T* page and the right pointer is set to point to the new page. The pointer of the parent cell, if any, that pointed to *T* page, is set to point to *r'* inside the parent page. *T* is then split into two subtries. Details of the splitting algorithm are in /LIT88/. One subtrie has all nodes preceding *r'* in inorder in *T*. It remains in the*T* page, unless it was the root page in which case it also moves to a new page. The other subtrie has the nodes that follow *r'* in *T*. This trie always move to a new page.

The split preserves the inorder to allow range queries. Fig 4 shows the splitting of the example trie from Fig 1, assuming that page capacity *b'* is *b'* = 9 cells. The split occured when key 'from' in the sequence at Fig 1a created a collision. The node (h, 0) was chosen as the split node. The node (e, 1) would respect the condition (i) above equally well, but fails to respect the condition (ii), as it has the logical parent (h, 0) in the trie. The reason for that condition is that the logical parent of the split node would become as all other nodes a physical descendant of *r'* in the new trie, while it is impossible in a TH-trie. The splitting phase created in page 1 the subtrie from all nodes preceeding (h, 0) in inorder at Fig 1 and in page 2 the subtrie with all nodes following it. Page 0 became the root page with only one cell, whose pointers are set up to point to the subtries.

2.6. Trie balancing

The trie, or subtrie in a page, are usually not best balanced, especially if key distribution is skewed or insertions are ordered. For instance, the example trie is unbalanced to the left, as the left subtrie of (o, 0) is larger. If the trie is well balanced, the time for in-memory search through the trie is about O($\log_2 N$). The search through an unbalanced trie is usually longer. This time remains nevertheless in practice of order of miliseconds and is thus only a fraction of the disk access time.

An unbalanced trie can usually be balanced. For instance, the example trie may be transformed to this at Fig 4, abstraction made of the paging.The balancing shortens only the node search time. Disk access performance, load factor and trie size are unaffected. Three techniques are known for the trie balancing. They are particular to tries, as balancing have to preserve the logical ancestorship. The first method performs the overall balancing of the trie or of subtrie in a page, using an intermediate canonical form /TOR83/. The second approach also balances the whole (sub)trie, but through the recursive application of the principles of trie splitting in /LIT88/. It makes r' the root of the new trie with left and right subtrie having therefore sizes closer to each other, then the procedure applies to each subtrie etc. Finally, one may balance the trie incrementally during each split, like an AVL tree. The balancing algorithm is nevertheless particular to preserve the trie correctness /OTO88/.

3. FILE BEHAVIOR

3.1 Random insertions

Behavior and performance analysis of TH can be found in /LIT85/, /LIT88/ and /ZEG88/. It mainly concerns the load factor $a = x / (b\ (N+1))$, where x is the number of keys in the file. The value of a is studied for buckets and, for MLTH, also for pages containing the trie nodes. It is determined under random, ascending and descending insertions and is denoted below respectively a_r, a_a and a_d. The split key is near the middle of the bucket, e.g. $m \approx 0.5b$. It appears that for both, buckets and pages, a_r stays on the average close to 70%, though page a_r is usually 2 - 3 % lower, as trie splits are less even than bucket splits. Bucket a_r may be slightly higher for some m values under $0.5b$, depending on b, as these value make the split more equal in the presence of split asymmetry. All together, the load factor of TH under random insertions is thus about that of a B-tree. Both methods use thus under random insertions about the same space for records.

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. This is also the number of branching nodes in the B-tree for the same insertions (one such node contains one key plus one pointer). The practical size of TH cell is six bytes : two bytes per LP and RP and one byte per DV and DN. The trie requires therefore usually much less space than the corresponding B-tree for its branching nodes whose size is in general several times larger than the cell size. In particular, 6 Kbyte buffer for the trie suffices to address about 1 000 bucket file, while 64 Kbyte buffer suffices for 11 000 bucket file. Since typical values of b are between 10 and 200, the corresponding TH files may contain about 104 - 10^6 records. In particular, if the bucket is the MS-Dos hard disk allocation unit (cluster) that is 4 Kbytes, then 30 Kbyte buffer suffices for the file covering the 20 Mbyte disk of IBM-AT.

The tries of the discussed size fit easily into a typical main memory. Any successful key search requires then only one disk access. An unsuccessful one costs at most one access, as there is no need for access if the leaf is nil. If the file is larger and MLTH is used, then two accesses per search should suffice in practice, as two levels for the trie should be enough. If p is page size, then for instance $p = 10$ Kbytes suffices for a bi-level file of almost 16 million records, assuming a modest $b = 20$ /LIT88/. Then, $p = 64$ Kbytes leads to a more than six hundred million record file. In particular, if page and bucket sizes are equal to the MS-DOS 4 Kbytes, then the file may span over 1G byte. This usually suffices even for optical disks.

3.2 Ordered insertions

Ordered insertions may be expected or unexpected. In the latter case, the m value is the same as for random insertions ie $m \approx 0.5b$ usually. The analysis in /LIT85/ and LIT88/ shows that the bucket load a_a is then within 60 - 73%, depending on b. This is substantially better than the well known $a_a = 50$ % of a B-tree for the same case and allows in practice to load the file through the usual insertion algorithm. On the other hand, the corresponding a_d is 40 – 55 %. This may be under the corresponding $a_d = 50$ % of a B-tree. The value of a_d increases over 50 % if m is lowered to about $m \approx 0.4b$, at the expense of a_a that may decrease under 50 %. For some m however, both a_a and a_d may be over 50 %. The reason for this nice 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 $m \approx 0.4b$. The percentage of nil leaves increases to 1-3 %.

The page load factor a_a is usually about 52 %. However, for some combinations of bucket and page sizes it may even reach 72 % or may drop to 40 %. This comes from the shape of the tries resulting from the split that improves a_a when the left subtrie is usually larger and vice versa. The factor a_d varies less, being usually about 45 %, within the interval 40 -53 %. See /LIT88/ and /ZEG88/ for details.

When the ordered insertions are expected, one may choose *m* value that increases the bucket load with respect to that implied by *m* tuned for random insertions /LIT85/. The *m* value should then increase for ascending insertions. The reason is that (i) the split loads then better the overflowing bucket, leaving it even 100 % full, if one sets $m = b$ and (ii) further insertions do not address this bucket anymore. Being ascending, they go indeed only to the nodes that follows the overflowing one in inorder. For opposite reasons, *m* value should be lowered for the descending order.

The performance analysis shows for these cases the following values :

(i) - the value of a_a remains between 60 - 80 %, depending on file parameters and key distribution, even if one sets $m = b$, . This is worse than for a B-tree that provides then a_a up to 100 %.

(ii) - the value of a_d is about 60 - 80 %, even if one sets $m = 1$, ie only the first key in the overflowing bucket remains in it after the split. Again, a B-tree may provide a_d up to 100 %.

This behavior of TH results from subtle influence of nil nodes for (i) and of the partial randomness for (ii). Fig 5 shows the behavior for the ascending insertions, assuming $m = b$. The insertion of key 'oszh' triggers the split of bucket 0. The split leaves the bucket full, but creates nil nodes, as Step 3.1 of Algorithm 2 leaves several digits in the split string (they are underlined and in fact are the whole split key). Further insertions go to bucket 1. However, when key 'ota' is inserted, it goes to the nil node under (s, 1) and bucket 2 is allocated. Bucket 1 is not yet full and no other insertion will come into. As keys come in ascending order only, all of them now have to address leaf 2 at least. This process usually repeats for several buckets and $a_a = 100$ % cannot attained.

Fig 6 shows the file behavior for expected descending insertions. To attain 100 % load of the buckets, it is in contrast necessary here to leave only one key in bucket, ie to set $m = 1$. However, it does not suffice because of the split randomness and two keys : 'orba' and 'orbf' remain. Bucket 1 is not fully loaded and no further keys will go into. The process usually repeats, as for the next split key 'mama', and the file cannot attain 100 % load neither.

When the ordered insertions are expected, one may also choose another split node to increase the page load. The position of the split node should then be shifted towards the last node of *T* for ascending insertions and towards the first one for the descending case. It is shown in /ZEG88/ that the page load increases then to 70 -87 %, depending on page and bucket sizes. This refinement has however only a marginal importance in practice, as the trie size is only a small fraction of the file size.

3.3 Deletions

The basic algorithm allows to freed a bucket either when it is empty or it may be merged with its successor or predecessor, provided that the corresponding leaves are siblings. From ten successor - predecessor "couples" in the example graph, four may thus merge. Others have to wait until the trie shrinks. For instance, bucket 10 and bucket 7 may merge only after the merges of nodes 4 with 10 and of nodes 7 with 8.

In contrast, there is no similar constraint in a B-tree. As any couple of successive nodes are siblings, they may always merge (except for the prefix B-tree /BAY77/). B-tree may therefore guarantee at least 50 % load of the file also under deletions, while not TH.

The condition for merge in TH file may be relaxed, providing more frequent merges. The technique is to rotate the trie, making the successive leaves siblings. The rotation may be performed classically, involving thus basically a change to a few internal pointers under the closest common ancestor. To be valid for a trie, it must however not violate the logical ancestorship. A logical parent of a node must thus not become its physical descendant in the new tree which then would not be a trie anymore. This condition leads to cases where two successive buckets still cannot merge. In the example trie, using rotations one doubles the number of mergeable couples to eight, but two couples which are bucket 9 and 4 and bucket 2 and 3 still cannot be merged.

Therefore, even this refinement does not allow TH to attain a B-tree performance. The difference to load factor not known, since bucket merging in TH file was not studied in detail. As for a B-tree, it was considered as a secondary peformance. Applications usually either delete records only logically and rarely to a point where the file shrinks heavily.

4. LOAD CONTROL

The afore mentioned performances of TH are sufficient for applications when ordered insertions are seldom compared to the random ones. However, more and more applications heavily use ordered insertions for which it is useful to have more control over the load factor. For instance, one may wish to get 50 % load no matter whether the incoming key order is ascending or descending and what is *b* value or key distribution. Also, one may need to create a file dynamically from sorted insertions, leaving it then for read access only, (if it is for instance a back-up, or a log file or a version), or dropping it with the transaction end, in case of query processing or of file transfer. If the file can then be loaded to 90 - 100 %, not only the disk space is saved, but also the efficiency of range queries improves. On the other hand, one may wish to be sure of the 50 % load even for extensive deletions.

To achieve these goals, the basic TH needs to be refined. One solution termed *Trie Hashing with Controlled Load* (THCL) is proposed below. THCL provides more control over bucket splitting, making it deterministic if required. It also eliminates nil nodes, avoiding the problem of underloaded buckets. Furthermore, THCL allows to merge successive buckets. Finally, it also allows to apply to TH the concept of the redistribution, well known for B-trees. The benefits are similar ie the load factor increases also for random and unexpected ordered insertions.

4.1. Elimination of nil nodes

Nil nodes avoid empty buckets, delaying the allocation until corresponding keys present for storage. As it appeared, the created buckets may however remain only partly loaded. For a higher load, it is better to avoid the creation of these buckets at all. One solution is to use instead of nil value the address of an existing bucket. The main rule below, is that this address is N for all nil leaves that Algorithm 2 would create in Step 3.3. It is thus the same address for all right leaves it creates, instead of only for the bottom right one. Fig 7 illustrates this rule. All right leaves now carries the same address 1. All new ascending insertions, in particular the key 'ota' from Fig 5, now goes to bucket 1, instead of creating bucket 2 and, all together, up to four new possibly underloaded buckets. Bucket 1 may now be filled up, no matter what are the incoming keys. When it overflows, because of key 'ovm' at the figure, it is split and bucket 2 is initiated.

Main algorithmic consequence is now that keys sharing a bucket may be mapped to different leaves. This affects Algorithm A2 in Step 3 as follows.

3. [Expand the trie] :

3.0 Identify the leaf to which the split key is mapped (for instance, use Algorithm A1 and retain the position of the last cell). Compute the logical path C to this leaf that will be still called leaf A .

3.1 [Cut the digits of the split string that are already in the logical path, if there is some]. Cut from $(c')_i$ the largest $(c')_l$ such that $(c')_l = (C)_l$. Let k be the remaining number of digits.

3.2 [Case of the single new digit which is c'_i . Expand the trie by the internal node representing c'_i and by leaf N]. If $k =$ 1, then do :

- Replace leaf A with node (c'_i,i).

- Attach leaf A again, as the left child of (c'_i,i) and attach leaf N as the right child of (c'_i, i).

3.3 [Case of several new digits. Expand the trie by internal nodes representing these digits, and by leaves pointing to N]. If k > 1, then replace leaf A with the 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 N.
- the left and right children of (c_i, i) are leaves A and N .

3.4 [New case : all digits of the split string are already in$C$]. If $k = 0$, then set to N the pointer in the successor of leaf A (that also pointed to A).

3.5 If some leaves that follow the leaves pointing now to N still point to A , then set these pointers to$N$.

Step 3.0 is defined informally only, since there are several ways to implement it. One needs it, since the split key may now be mapped to a different leaf than the key that triggered the overflow. At Fig 7, the key 'ovm' is mapped to the leaf under (s, 1), while if 'oszr' with its usual middle position was the split key, then it would be mapped to the right son of (a, 3).

Step 3.1 is the same, but may finish with an empty split string even if the split string had a single digit ie the value of i was $i = 0$. It will be the case of 'ota' if the new key was for instance 'vm'. Step 3.2 is basically the same, except that the test uses k value, as $i = 0$ is no more a sufficient condition. Idem for Step 3.3, except that it puts N value instead of nils. Step 3.4 deals with empty split strings. It does not enlarge the trie, only updates the successor of leaf A . It acts in the manner similar to that leading to an allocation of an actual bucket to a nil node.

Finally, Step 3.4 keeps consistency with Step 2, that moved to bucket N any key above the cut key in bucket A.. If the new key was for instance 'vm', it would thus also move to bucket 2 through step 2. Without Step 3.4, the trie would however continue to map it to bucket 1, as 'vm' is mapped to the right son of (o, 0).

When MLTH is applied, Step 3.4 implies that one may need to access next page, if the last node in the current page carries A. The refinement also slightly modifies the algorithmic of range queries. Next leaf may now carry the address of the bucket already in main memory, lowering thus slightly the access cost of corresponding queries.

4.2. Split control

We will call *cut key* the last key that a split leaves in the overflowing bucket. Cut key will be noted c'''. Through Step 1, Algorithm A2 leads to $c' \leq c''' < c''$ which implies that c''' may fall anywhere within $[m, b]$, as the position of c'' is $b + 1$. Therefore, TH split may lead to A deterministic split in contrast occurs when any split key is also the cut key.

One way to decrease the split randomness in TH, is to slightly modify Step 1. Instead of using only *c''*, one may allow the choice of a given key *c* above *c'* and until *c''* , called below *bounding key*. Closer is the position of *c* to that *c'*, less the split is random. If the bounding key is next to the split key, then the split is deterministic.

Fig 8 shows the application of split control to expected descending insertions. If the bounding key position is is $m + 1$ and the split key position is the usual one ie $m = 3$, then exactly two keys move at each split to the new bucket and the bucket load of $a_d = 50\ \%$ is guaranteed. At the figure, the bounding key is 'osca'. If *m* is set to $m = 1$, then exactly four keys move to next bucket and the load reaches $a_d = 100\ \%$. If insertions were ascending, then the choice of the position $m + 1$ for the bounding key with *m* about $b/2$ would also lead to the 50 % load.

4.3. Bucket merging

The principle of making successive leaves pointing to the same bucket, may be applied to deletions. Successive buckets that cannot be merged by the basic TH may then merge. It suffices to make all the corresponding leaves pointing to the same bucket. One may then in particular also guarantee 50 % load, as for a B-tree.

Bucket merging may be accompanied by trie node merging. Unlike in the basic method, both processes may now however be decoupled. The basic strategy for node merge is to remove the the siblings pointing to the same bucket and their parent. Using valid rotations, one may further also merge successive leaves that are not siblings. The trade-off between all the choices is a larger trie versus simplified and faster algorithmic. What are precisely the corresponding figures is an open research problem. It is clear however, that it may be frequently better not to merge the nodes at all. The gains in memory space are in this case rather unimportant with respect to the main file, while deletions often alternate with insertions. Also, leaving nodes where they are, simplifies the concurrency control and makes it more efficient /VID87/.

4.4. Redistribution

This term designates the capability of a B-tree to redistribute keys between existing buckets, instead of systematically allocating a new bucket for each split. The discussed refinements allow to apply the concept also to TH files. Early analysis of the idea of redistributing keys in TH file to improve the load factor is already in /DEL84/. The buckets to be used may be the predecessor or the successor in inorder of the overflowing one, let it be O. If the redistribution uses the successor, let it be S, then one should at first set the position of the split key high enough to move no more keys than there is available space in bucket S (this is hard to control in the basic TH or cannot be done at all, because of split randomness). Then, one simply applies Algorithm A2, using bucket S instead of new bucket N. If the redistribution uses the predecessor, let it be P, then the split key should be the lowest one in bucket O ($m' = 1$). The bounding key should be lowered to position m'' such that if bucket O has room for p keys, then $m'' \leq p + 1$. The keys under the bounding one move then to bucket P and the pointers in the leaf or leaves O to which they were mapped are set to P.

The redistribution may increase the size of the trie or may leave it unchanged. Perhaps surprisingly, it may even allow to shrink it. It happens when Step 3.1 ends up with the empty string. One of the parents of a leaf set to S or P by the redistribution algorithm may then become pointing to S or P through both edges. One may leave this node as is or may replace it and its leaves by a single leaf S or leaf P respectively.

Fig 9 illustrates such situation. The redistribution is set to load the buckets as high as possible, so it moves to bucket 2 only the highest key mapped to the overflowing bucket 1. The key 'oszr' is therefore chosen as the split key at the figure, as it becomes the highest in the bucket, once the new key 'ost' properly enters the sequence to split. As the result of the operation, node (t, 1) points to bucket 2 through its both leaves. It may stay as is if it is unconvenient to move cells or may be suppressed.

4.5. File behavior

If the splits are deterministic and without nil nodes, then the load factor for ordered insertions has a guaranteed value. In particular, as for a B-tree, one may attain a = 100 % for the expected case. Also, one may provide $a_d = a_a$ = 50 % for the unexpected case for any b and the same lower bound under deletions. The load factor for random insertions remains in practice unaffected ie $a_r \approx$ 70 %.

Performance for unexpected and random cases improve further if redistributions are used. The results reported for a B-tree in /KNU73/ for instance apply indeed also to THCL files. Obviously, a_a and a_d may approach 100 % for unexpected insertions. Then, a_r may reach almost 87 %. This is however in practice only a peak result. The 70 % value of a_r is an average of an oscillation, as in fact buckets under insertions have tendency to fill up almost simultaneously to a high value and then to split, also almost simultaneously, especially for larger b.. This phenomenon lowers the load almost to 50 % and all together makes the usage of redistributions not as efficient as it could seem. That is why this refinement is uncommon in B-trees in practice.

The characteristic that has to be affected by the load control is the size M of the trie. As it appears hard to find out analytically how it depends on various parameters, the file behavior was studied through simulations. They consisted of ordered insertions of 5 000 keys, randomly drawed and then sorted (same number as for the reported experiments with B-trees). Fig 10 shows the results for the ascending insertions. The file behavior appears as follows.

- The parameter d is defined as $d = b - m$, where m is the position of the split key. The most compact file ie with a = 100 % is thus achieved for d = 0 (the load factor value is denoted a% at the figure). This choice enlarges the trie with respect to the basic case of ascending insertions in the presence of nil nodes with $m \approx (b+1) / 2$. The reason is that adjacent keys usually share more digits than more distant ones and a longer split string is generally needed. The increase is by 20 -30 %. For the typical b = 20 for instance, the absolute values are respectively 414 and 420 nodes.

- One may therefore expect, that to lower m ie to increase d, even slightly, may be greatly beneficial to M value, while affecting a only a little. Curves confirm this expectation. Substantial savings, over 30 % of peak M value (for d = 0) may be achieved, while a remains over 90 % anyhow. The resulting trie is then even sizably smaller than that generated by TH. Globally, THCL provides then about 20 - 30 % higher load factor for about 20 % smaller trie ! For b = 20 for instance, the best size is 344 nodes instead of 414.

- An even more surprising fact is that there is always a minimum of M while split key position m moves down towards the middle key. The minimum apparently corresponds to the balance between two phenomena accompanying the lowering of the split key position :

(i) - split strings shorten and thus M grows less. While the decrease is rapid for small d's, the length of a split string becomes however rapidly close to a minimum.

(ii) - a split moves more keys to the new bucket. The frequency of splits increases and so M grows faster. Unlike (i), this phenomena is however almost proportional to d and its negative influence on M size always prevails for larger d's.

- also surprisingly, if the split key is the middle one, then for any b, M is over 20 % smaller for the basic TH and its a_a is slightly higher (2 - 4 %). The explanation is that THCL split in general removes about half of keys from the last bucket supporting the last insertion and add a new node to the trie. The expansion through the allocation of an actual buckets to a nil node in the basic TH is in contrast smoother. It does not remove any key from this bucket and does not add any new node. Apparently, such buckets remain then also generally loaded better to the point that slightly less new buckets is finally created.

This characteristic justifies the use of the basic TH instead of THCL if the load control is not needed. Alternatively, one may refine Algorithm 2, to seek more for splits without a new node through Step 3.4. For this purpose, one should scan for the candidate split key above the basic position $b / 2$, whenever more than one leave point to the overflowing bucket.

Fig 11 shows the behavior for descending insertions. The meaning of d is there $d = m''' - m - 1$ where m is set to$m = 1$, ie the split key is the bottom one in the bucket and m''' is the position of the bounding key. The increase of d corresponds to a higher position of the bounding key. There is no more a minimum of M, thought important savings to M size with respect to this for $d = 0$ (around 30 %) correspond to small d's only. Afterwards, the curve is almost flat; as well as that of a_d that remains over 90 % or close to. It is thus relatively costly with respect to M to approach the 100 % load, because of much longer split strings, while all others positions of the bounding key are still very efficient. The most practical m''' appears to be that providing anyhow an excellent load over 95 %, corresponding to d value where M curve becomes flat. The size of the trie is then again smaller than for the basic TH, but the difference is also smaller, under 10 %.

The ratio M / N, denoted s, is the average number of cells created by a split ie the growth rate. It appears that s value for the minimal M value for ascending insertions is $s = 1.25 \div 1.6$, for $b = 10$ and$b = 50$ respectively. For the descending insertions and the values of a pointed at Fig 11 by arrows, the values of s are $s = 1.2 \div 1.5$. Assuming the cell size of 6 bytes, it means that the trie grows at the rate of $7.2 \div 9$ bytes per split on the average. A B-tree grows by the size of the key plus the pointer size. This is typically 20 to 50 bytes ie requires several times more space for the branching nodes to the same file. For the full load of a = 100 %, the s values grow to $s = 1.6 \div 2.13$ for the trie. The trie growth rate increases thus about 1.5 times to $10 \div 13$ bytes per split. This still makes the trie usually much smaller than the B-tree.

5. COMPARISON TO B-TREES

The "ubiquitous" supremacy of B-trees or, more specifically, of their most used implementation that is a B^+-tree, comes from the universality of this method with respect to all the currently important requirements /COM79/. The legion of variants or of competitors that cannot be referenced all here, are much less used or stayed research proposals. While they usually outperform a B-tree in some aspects, they also introduce drawbacks which make a B-tree still in general preferable. This is in particular the case of prefix B-trees, where the gain with respect to the index size requires branching nodes of variable length, a basically sequential search in the page, deletions that may lead to a page overflow,... /LOM79/, /BAY77/, /COM79/).

Most of the methods propose indeed variations of the same paradigm of m-ary trees of keys built through deterministic half to half splitting using key value. It is then at least tough to optimize some performance factors, without deteriorating others. TH starts from a different paradigm that is a binary tree and a digit at the time partly random splitting. This paradigm provides an inherently higher fan-out. The work on TH and now THCL, shows that this property in general allows the method to outperform a B-tree or to perform about equally well. If a precise comparison for a requirement is not known, at least one knows that TH fulfils it fairly well. The method is also more ubiquitous in the sense that a single setting of parameters may suffice for broader usage conditions. All these features make TH an attractive alternative to a B-tree as a general purpose method. We now sum up the reasons for this claim.

- Behavior under random insertions. Most applications work with random, though not necessarily uniform, insertions. TH and a B-tree provide then the same load factor. Because of its usually much higher fan out, TH typically needs however less disk accesses for a key search or not more. Alternatively, for the same fan out, it requires a smaller page size than the B-tree, including the prefix B-tree. Furthermore, it usually requires less accesses per insertion or split. It may require more accesses for some splits, but this case is unlikely in practice. Finally, it supports deletions and allows the file to shrink. Shrinking is usually faster than in a B-tree, as there is only one or two levels of pages to deal with and there no shift of half of the keys in the page on the average.

- Behavior under expected ordered insertions. The user of a B-tree may shift the split key to create a file with even 100 % load. TH user may do the same. In addition, he may use a unique split key setting for both random and ordered insertion, if the 70 % load suffices. This is the case of a file once loaded and then supporting only random insertions, which is the most frequent one.

- Behavior under unexpected ordered insertions. B-tree file load decreases then to 50 %, as the split key is the middle one, unless a redistribution is used (which is seldom). This position of the split key is the prevalent choice. TH allows to obtain the same load, still providing better access performance. However, if the descending case is unlikely or unimportant, TH may provide the load 10 - 20 % higher than a B-tree, almost at the level of random insertions, without a redistribution. The load for descending insertions remains fair, as it is over 40 %. For smaller *b*, it may even be also higher than 50 %. Higher load factor and higher fan out again lead to better access performance.

- Fast search within a page. B-tree allows easily the binary search, at least when keys are of fixed size. This is also the usual case in TH, but the search is usually faster, as branching compares single digits, instead of chunks of keys. As with any method related to ideas in hashing, one may however probably find the "worst case" insertions creating an almost linear trie with O(N) worst search time. As usually for hash files, this is nevertheless highly unlikely in practice. Also, it would not create any catastrophic effect in usual conditions, the time for search in the page being a fraction of disk access time.

- Simplicity of implementation. Both methods are rather simple to implement, though there is no precise criterion for simplicity comparison. TH is particularly easy to implement if one level tries suffice for the application files which should be the common case. Trie splitting algorithms have more subtleties than those rather straightforward of a B-tree. They remain nevertheless fast, compact and easy to program. The implementation used for simulations in Turbo Pascal needed about 50 Kb for the load module.

6. CONCLUSION

Trie hashing with refinements introduced above allows tighter control of memory occupancy. The main benefit is the possibility to create compact files, loaded up even to 100 % and to guarantee the 50 % load under any conditions. The high load improves also the efficiency of range queries and may improve that of the key search, especially if the more compact trie may entirely fit to the main memory. The compact files are suitable for applications where insertions are ordered. It may be a historical file, a back-up or a log file, a transfered file, or a temporary file for a query evaluation etc. The price to pay is 25 - 30 % larger trie, especially if the load should approach 100 %. All together the capabilities of trie hashing, make it appearing more universal and generally more efficient than a B-tree.

Trie hashing looks promising also with respect to other requirements upon an access method such as concurrency. In /VID87/ an algorithm is proposed for the concurrent access the basic TH with one level trie. It is shown that TH may allow for higher degree of concurrency than a B-tree, especially when no cell is physically deleted. One needs then to lock only the leaf A and the variable N , which makes the algorithm in addition very simple. In a B-tree one needs instead to lock at least one whole page and known algorithms are far more complex /SAG85/. This constraint of a B-tree probably cannot be overcome, as a part of the sequence of the keys in the page have to be shifted upward to make the room for the new one. In TH in contrast, the new cell is appended to the end of the table of cells, so there is no need to block any access to another cell. This results should now be refined for MLTH and for the new variant.

By the same token, /TOR83/ shows an efficient algorithm for the reconstruction of the TH trie that was accidentally destroyed. The algorithm uses the logical paths, assumed stored on the disk, for instance in the headers of the buckets. The reconstructed trie may be in addition better balanced than the original one. One conjectures in /TOR83/ that the new trie is the optimal one. This analysis should also be extended now to MLTH and to the new refinements.

Furthermore, the current trend towards large main memories, more and more often allows to create large main memory files. Database processing become then much more efficient, especially for workstations. Again, TH appears attractive, especially because of lower overall space needed for the trie and faster digit at a time search than in a B-tree /KRI84/. Research issues related to algorithms for trie balancing, less important in the case of disk files, become then naturally of major concern.

The TH-trie was first defined rather indirectly, as the structure built by the splitting algorithm specified in /LIT81/. Further work triggered an effort towards a formal definition of the structure, independent of the algorithm, and towards formal studies of structural properties of the tries /TOR83/, /LIT88/. Important notions appeared from these investigations, like these of the logical structure or of equivalent tries. They allowed to improve the original algorithms for search and splitting and led to the discovery of these for trie paging and balancing.

Such studies should now concern THLC, especially the idea of several leaves pointing to the same bucket. This property may also be seen as providing the structure of a lattice /DEL84/, which is a type of structures rather unexplored in comparison to trees. Some aspects of the THLC algorithmic are also only globally defined, as there are several design choices to implement them. It was pointed out in the corresponding sections and should now be precised and the trade-offs compared. There are also several interesting open problems with respect to the basic TH which applies to THCL, as well. Some are listed in /TOR83/, others are related to the correctness and optimality of algorithms for trie balancing.

Performance characteristics of THLC are basically known through simulations. On one hand, one should now test THCL's behavior especially with respect to the trie size M, for a large set of real data, (the 20 000 word UNIX dictionnary for instance). This was done for the basic TH and confirmed the theoretical figures /ZEG88/. One the other hand, one should study them also through analytical methods. These in /JAC88/ and /REG87/, giving the mean and the variance of the size of the trie using Mellin transform, seem particularly promising. The results are nevertheless only for binary digits. They should be thus extended to Q-ary alphabets ; $Q > 2$ and to THCL. This is clearly a difficult goal.

Furthermore, one should design variants of the method optimizing a particular performance. The ideas of "overflow", of digital B-trees /LOM81/, of uneven splitting etc. that worked fine for a B-tree, should reveal equally useful. Finally, one should extend TH to the multikey case, for instance on the basis of the ideas in /OTO87/ and /OUK83/ and of some of general ones in /SAM84/. As tries remain compact in presence of uneven distributions, one may expect them to offer an alternative to the grid files /NIE84/ without the phenomenon of exponential growth of the directory.

REFERENCES

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

/BAY77/ Bayer, R., Unterauer, 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-th ed., Addison-Wesley, 1986, 639

/DEL84/ Delafosse, L. Improving the load factor of Trie Hashing. Unpublished memorandum, in French. (Dec. 1984), 16.

/ELL85/ Ellis, C., S. Concurrency and Linear Hashing. ACM-PODS 85. (March 1985), 1-7. To appear in ACM-TODS.

/ENB88/ Enbody, R., J., Du, H., C. Dynamic Hashing Schemes. ACM Computing Surveys, 20, 2, (June 1988).

/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 Informatica, 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, 1984.

/JAC88/ Jacquet, Ph, Régnier, M. New Results on the Size of Tries. to app. in IEEE Trans. on Inf. Theory.

/JON81/ 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.

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

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

/KUN80/ Kung, H. T., Lehman, L., P. Concurrent Manipulation of Binary Search Trees. ACM-TODS, 5, 3, (Sept. 1980), 354-382.

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

/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.

/LIT81/ Litwin, W. Trie hashing. SIGMOD 81. ACM, (May 1981), 19-29.

/LIT84/ Litwin, W. Data Access Methods and Structures to Enhance Performance. Database performance, State of the Art 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).

/LIT88/ Litwin, W. Zegour, D. , Levy G. Multilevel trie hashing. Extending Database Technology, 1-st European Conf. on Databases, (March 1988), Springer Verlag.

/LOM79/ Lomet, 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.

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

/NIE84/ Nievergelt, J., Hinterberger, H., Sevcik, K., C. The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM TODS, (March 1984).

/OTO87/ Otoo E. Multikey Trie Hashing for Scientific and Statistical Databases. CODATA 87, P.S. Glazer (ed). Elsevier Sc. Publ. B.V. (north Holland), 35-40.

/OTO88/ Otoo E. Locally Balanced Compact Trie Hashing. 3rd Int. Conf. on Data and Knowledge Bases. Jerusalem (June 1988), IPA.

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

/REG887/ Regnier, M. Trie Hashing Analysis. 4-th IEEE Int. Conf. on Data Eng. (March 1987).

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

/SAG85/ Sagiv, Y. Concurrent Operations on B-trees with Overtaking. ACM-PODS, (March 1985), 28-37.

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

/SHO85/ 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 Reconstructive 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.

/TRO81/ Tropf, H., Herzog, H. Multidimensional range search in dynamically balanced trees. Agnew. Inf. 2, 71-77.

/VID87/ Vidyasankar, K., Litwin, W. Sagiv, Y. Concurrency and Trie Hashing. Techn. Rep. 8704. Mem. Univ. of Newfoundland, Canada, (Aug. 1987), 28. Subm. to a journal.

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

/YAO78/ Yao, A., C. On random 2-3 trees. Acta Inf. 18, (1983),159-170.

/ZEG88/ Zegour, D . Extensions de Hachage Digital. Dr of Univ. Thesis., Dauphine Univ., Paris, (June, 1988), ed. by INRIA, 246.

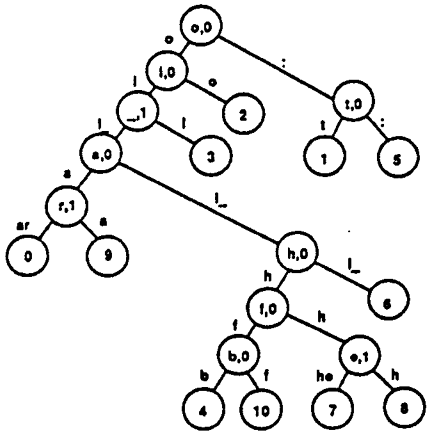
(a) the, of, and, to, a, in, that, is, i, it, for, as, with, was, his, he, be, not, by, but, have, you, which, are, on, or, her, had, at, from, this

(b)

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

(c)



(d)

<table>
<tr><td colspan="2">RP</td></tr>
<tr><td>DV</td><td>DN</td></tr>
<tr><td colspan="2">LP</td></tr>
</table>

RP - Right pointer
LP - Left pointer
DV - Digit value
DN - Digit number

(e)

-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	8	9

Fig 1 Example file
(a) insertions
(b) buckets
(c) trie with logical paths
(d) cell structure
(e) trie representation

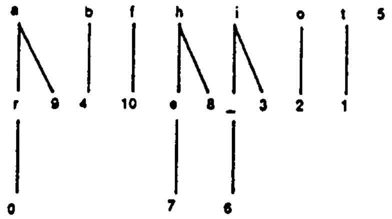


Fig 2 : Logical structure of the trie

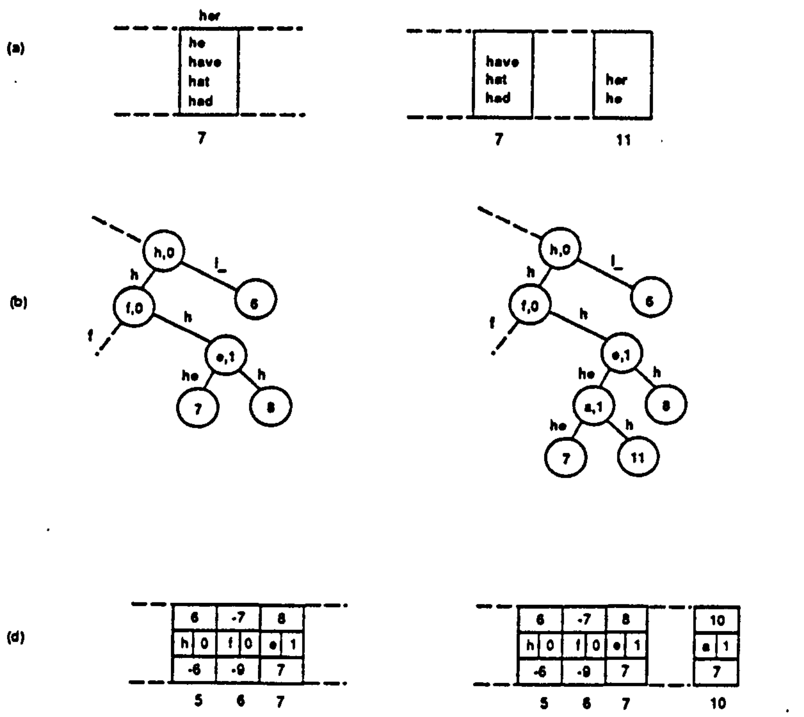


Fig 3 Bucket split

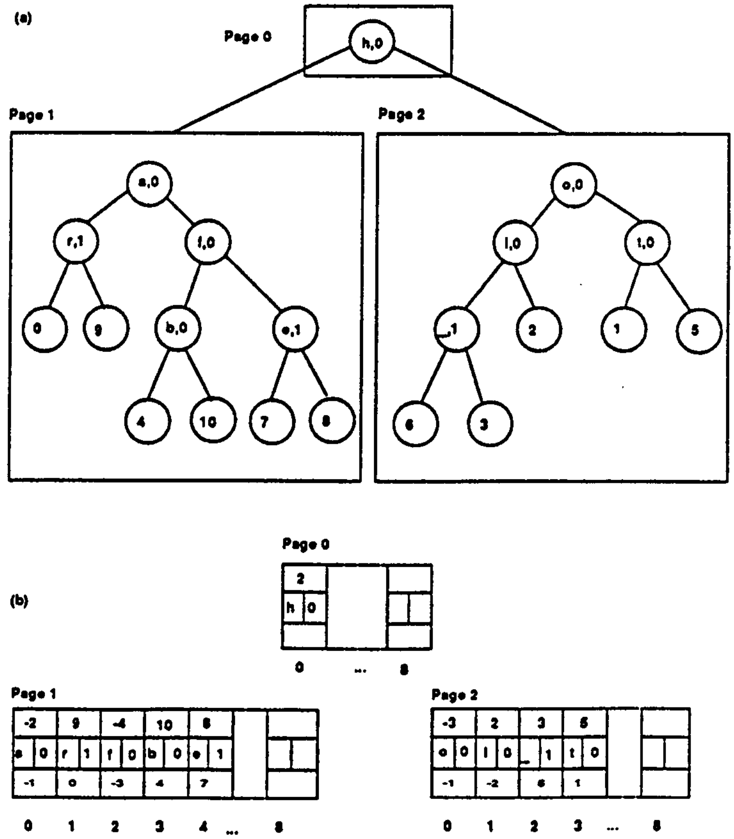


Fig 4 Trie split

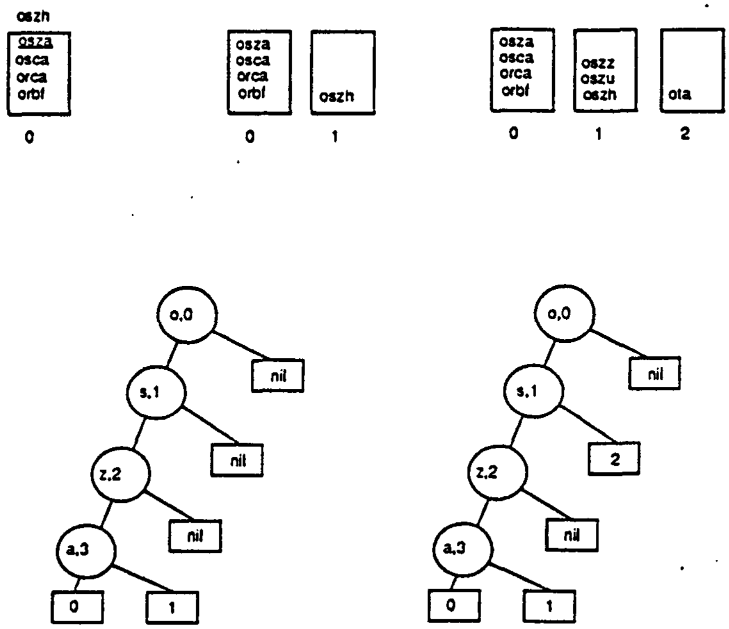


Fig. 5 Best splitting under expected ascending insertions

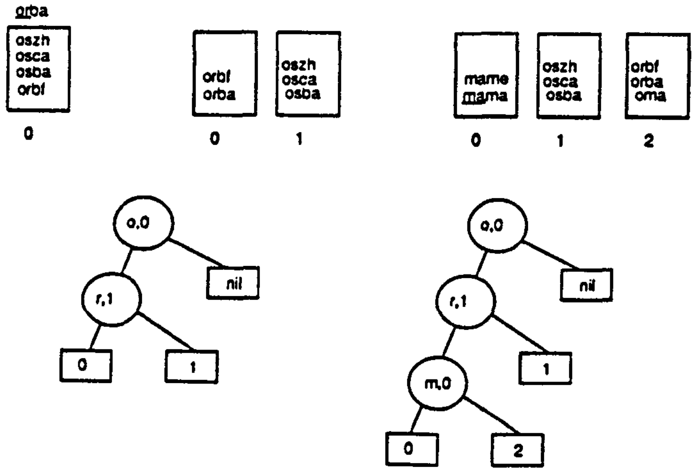


Fig. 6 Best splitting under expected descending insertions

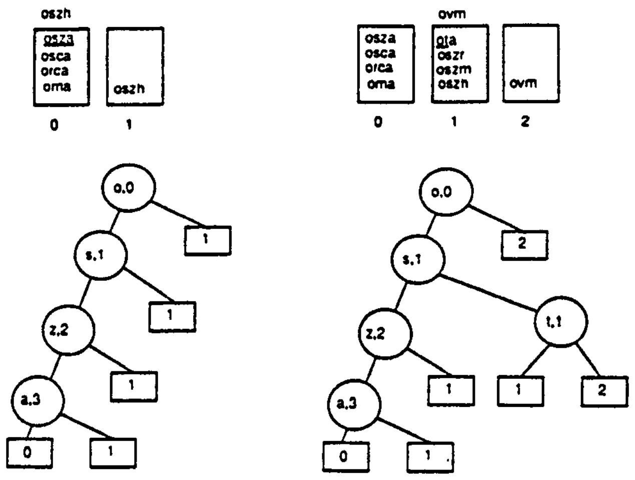


Fig. 7 Splitting without nil nodes

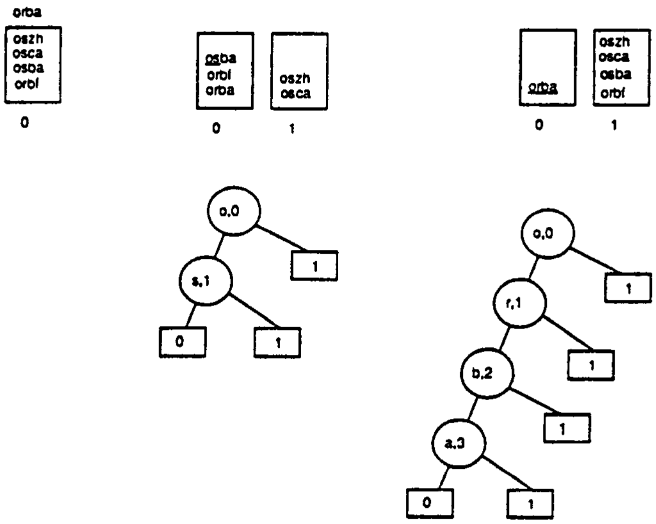


Fig. 8 Controlled splitting for descending insertions

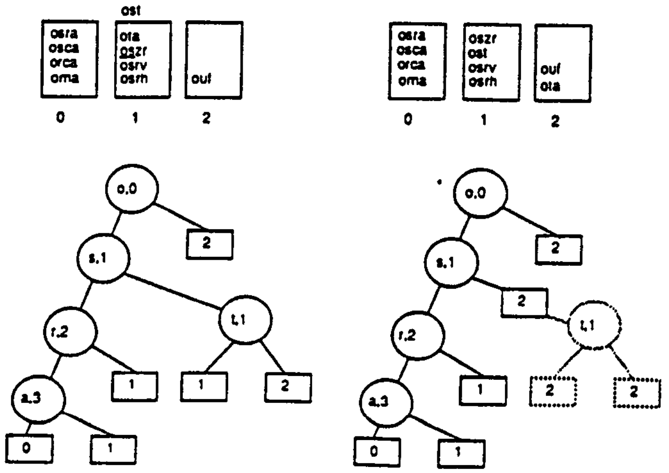


Fig. 9 Redistribution shrinking the trie

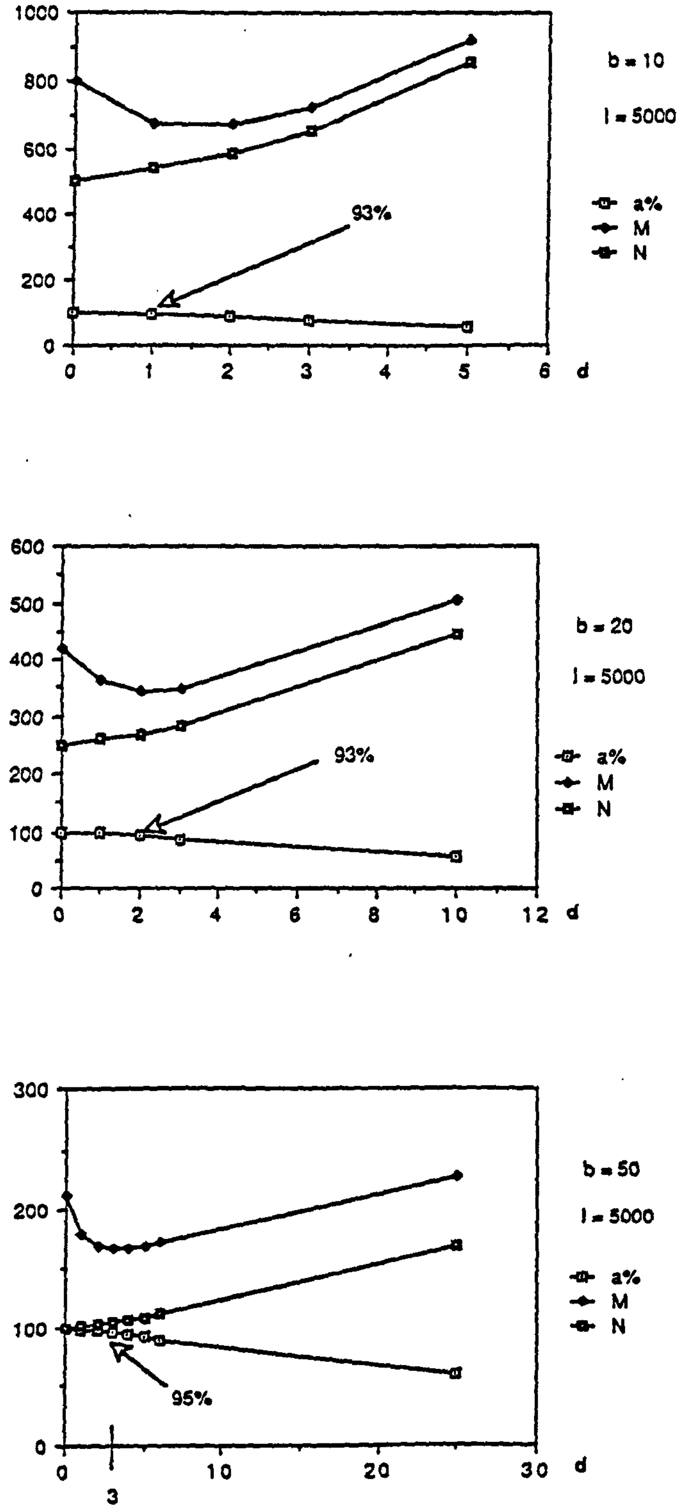


Fig 10 Performance for ascending insertions :

- load factor (a%)
- trie size (M)
- file size (N)

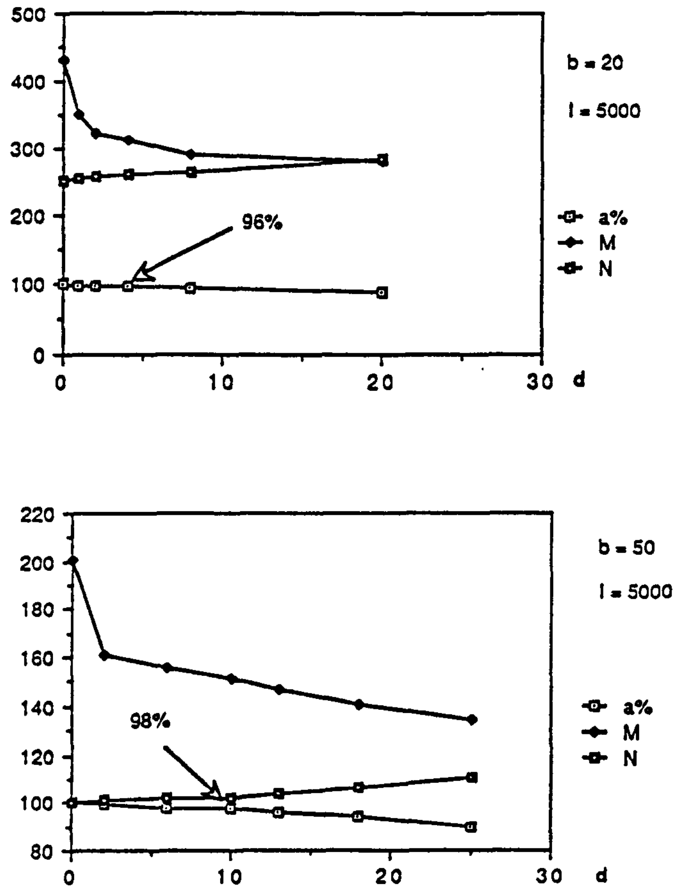


Fig 11 Performance for descending insertions :

- load factor (a%)
- trie size (M)
- file size (N)