

# **Trie hashing with the sequential representations of the trie**

D.E Zegour, W. Litwin\*

INSTITUT NATIONAL D'INFORMATIQUE

**Abstract :** Trie hashing is one of the fastest methods for accessing data on the disk. As long as the trie is in core, any key search takes at most one disk access. The trie size depends linearly on the file size and on the representation chosen for the trie. The representation considered until now was called standard representation. We propose two representations that are about two times more compact. The same buffer in core, suffice then for about two times larger file. The price is that the algorithmic is more complex and needs more processing time.

**Keywords :** Algorithms, Data structures, File structures, B-trees, Hashing, Dynamic hashing.

## **Plan**

- 1. Introduction**
- 2. Basic principles**
- 3. New graphs**
- 4. Sequential representations**
- 5. The LL-TB sequential representations**
  - 5.1. The principle**
  - 5.2. Insertion**
  - 5.3. Key search**
  - 5.4. Sequential search**
- 6. The TB-LR sequential representations**
  - 6.1. The principle**
  - 6.2. Insertion**
  - 6.3. Key search**
  - 6.4. Sequential search**
- 7. Analysis**
  - 7.1. Standard representation**
  - 7.2. The LL-TB sequential representations**
  - 7.3. The TB-LR sequential representations**
- 8. Implementation**
- 9. Comparison with others methods**
- 10. Conclusion**
- Bibliography, Figures.**

\* Researcher at INRIA, France

## 1. Introduction

Trie hashing is a new access method proposed in [LIT 81]. The method allows to constitute dynamic ordered files of records identified with a primary key. As for the related others methods, the load factor is on the average 70% for random insertions. It's about 60-70% for sorted insertions [LIT 85]. A key search is performed in one disk access at most, even for files attaining millions of records. These properties place the method among the fastest access ones.

The trie can be represented in the memory in several ways. The basic representation in [LIT 81], called standard representation, needs about six bytes/node. In this representation, any bucket needs at least one node. It is shown in [LIT 81] that a file of 100 000 buckets needs then about 10K buffer for its trie. For large files, trie size needed may become prohibitive for some applications.

The idea explored in this paper is to increment the ratio of file size to trie size by using more compact trie representations. We called the proposed representations LL-TB and TB-LR sequential representations. We will show that with these representations, we can double the file for the same size of the memory allocated for the trie. The counterpart, this involves a more complicated algorithmic. In addition, the computing time will be higher. Nevertheless, the trade-of should be interesting for many applications.

Section 2 recalls the basic principles of [LIT 81]. Section 3 and 4 introduce the representations. Sections 5 and 6 present briefly the corresponding search, splitting and sequential search algorithms. Section 7 discusses performance. Section 8 gives a possible implementation. In section 9, comparisons with others methods are made. Section 10 concludes the paper.

## 2. Basic principles

Trie hashing uses a dynamic hashing function that is represented by a trie. The function maps the key values on the bucket addresses. Initially, i.e before the first collision, all keys are mapped to the address zero. Then, when collisions occur, the key values extend linearly. Collisions are resolved by splitting. Each split extends the file by one bucket. The trie consists of two kinds of nodes: internal and external nodes. Each split creates one external and one internal node, at least. The trie may be represented in storage in several manners. Two types of such representations were called respectively standard and sequential representation [LIT 81].

In the standard representation the trie is represented as a binary tree. To each internal node correspond two fields : a value field and a pointer field. Value field is a pair (DV,DN) where DV is a digit value and DN is a digit number. Pointer field is a pair (LP,UP) where LP and UP are pointers either to internal or external nodes. Buckets are designated by external nodes. A negative pointer indicates a pointer to an internal node. The trie is extended almost linearly, i. e, at each split, at least, one node is added at the end of the file. A split may end more than one node, but this event is rare /LIT 85/. Fig(1) shows the graph G of the trie after insertions of 31 keys. The search and splitting algorithms corresponding to this representation are in /LIT 81/.

The principle of a sequential representation of a graph is that nodes are in some predefined order. We then save the space needed by the pointers. This renders the sequential representations more compact than standard ones. In contrast, the algorithmic has to be more complex as it involves shifting of nodes so that to preserve the predefined order.

In /LIT 85/, two sequential representations were mentioned:

- (i) -the nodes correspond to the leaves of the graph G visited in ascendant (left-to-right) order,
- (ii)-The nodes correspond first to all nodes of level 1 in ascendant order of digits; then, in the same order, all nodes of level 2. And so on..

More details may be found in /LIT 81/.

Below, we consider that the file is a collection of buckets numbered from 0 to N. Each bucket has a capacity b.

Each key is of the form  $C = d_0d_1...d_l$

where  $d_i$  is a digit of a given alphabet.  $i$  is the digit level. In what follows, the alphabet is the usual one. The maximal digit value will be denoted ' $\cdot$ '. The symbol ' $\_$ ' denote one space. We will also suppose that the method principles of trie hashing defined in /LIT 81/ or /LIT 85/ are known. We will be interested mainly in the manner in which the trie is constructed. So, we will designate by  $C'$  the middle key of the sequence of the  $b+1$  keys and by  $C''$  the greatest one. We will also assume that the sequence chosen for the split is  $c'0c'1...c'k$  and  $I$  is the number of digits that already exist in the trie. The variable  $m$  denotes the address of the bucket undergoing the split and the variable  $M$  the next bucket to allocate for the file.

### 3. New graph

In our study, we consider the hashing function as a  $m$ -ary trie. Fig(2) shows the new graph  $G'$  representing the trie for the example file. In what follows, we give the correspondence between the binary and  $m$ -ary tries. Let us recall that at each collision, one determines the shortest sequence of digits which permits to split the  $b+1$  keys. At a given time, let  $S$  be the sequel of the all sequences. A digit  $d_i$  is told logical descendant of  $d_k$  of order  $h$  if there exist a sequence in  $S$  containing both  $d_i$  and  $d_k$  such as  $i=k+h$ .

The new graph  $G'$  we have considered is a  $m$ -ary trie. It contains as much nodes as internal nodes in the graph  $G$  representing the binary trie. In the graph  $G'$ , a level consists of all the internal nodes having the same number digit. In a level, the nodes are divided in classes so-called sublevels. The digits of nodes of a sublevel in the level  $i$  are the logical descendants of order 1 of a digit of a node in the level  $i-1$ . Then, at the level 0, there is only one sublevel, and usually, there are many sublevels by level. Furthermore, within the sublevels, the digits are in ascendant order.

Each node consists of two fields: a node digit and a node pointer. Node digits concatenation of each path from root node to a given node represents the maximal key of a bucket. If we choose LP as node pointer, the maximal key corresponds to the bucket pointed by the node pointer of the same node (case of Fig 2a). if we choose UP as node pointer of a node, the maximal key corresponds to the bucket addressed by the node pointer of the next node with respect to the traversal in *postorder* (case of Fig 2b ).

Note the interesting property that if we list the node pointer of the  $m$ -ary trie in *postorder* (i.e, for each node  $n$  of the trie we apply the rule  $T_1T_2...T_K n) / KNU 73/$ , then the bucket addresses stored at these node pointers are listed in sorted order.

We can extend the graph  $G'$  by external nodes. We obtain thus an extended graph so-called below  $G'e$ . Fig3 gives the extended graphs corresponding to the graphs of fig2. In this graph, there are two sorts of nodes. The internal nodes are digits and the external ones are pointers to the buckets.

In the next section, we present the sequential representations we can define in the graphs  $G'$  and  $G'e$ .

### 4. Sequential representations

The sequential representations we will analyze correspond then, on the one hand, to the traversal of the new graph G' in the following orders:

(i) - Left to right in a level, then from top to bottom.

(ii)- From top to bottom, then left to right.

and, on the other hand, to the traversal of the graph G'e in the order (ii).

We called the representation corresponding to (i) the LL-TB sequential representation because the nodes are represented as follows: The nodes of the level 0 left to right, then the nodes of the level 1 in the same order; and so forth...

We called the representation corresponding to (ii) the TB-LR sequential representation because the nodes are represented as follows: The nodes of the most left path from top to bottom, then those of the path immediately to the right; and so forth...

To distinguish the two representations corresponding to (ii), we will call the one defined in G'e the TB-LR' sequential representation.

In the graph G', a node may be represented as a 3-uplet with the fields DV, UP and B. DV is the digit value, UP is the upper pointer and B is the leaf bit equal to 0 if the internal node is a leaf and 1 otherwise. In this graph, for the level 0 and for each sublevel we also consider a counter of the number of nodes.

With these considerations, if we represent the counter between < > and if we take UP as node pointer of a node (fig2a), then the LL-TB sequential representation of the graph G' will be :

(I) : <7> (a,4,1) (b,10,0) (f,7,0) (h,6,1) (i,2,1) (o,1,0) (t,5,0) <1> (r,9,0) <1> (e,8,0) <1> (\_,3,0)

the TB-LR sequential representation of the graph G' will be

(II) : <7> (a,4,1) <1> (r,9,0) (b,10,0) (f,7,0) (h,6,1) <1> (e,8,0) (i,2,0) <1> (\_,3,0) (o,1,0) (t,5,0)

In the graph G'e, there are two sorts of nodes. If we represent the external node between < > and if we take LP as node pointer of a node (fig3b), then the TB-LR' sequential representation could be:

(III) : a r <0> <9> b <4> f <10> h e <7> <8> i \_ <6> <3> o <2> t <1> <5>

We shall see, latter on, how the distinction between an internal and an external node could be done.

Below, we analyze the algorithmic corresponding to the two types of representation. I. e, one representation defined in the graph  $G'$  and one defined in the graph  $G'e$ . Thus, we make this, in detail, only for the LL-TB and TB-LR' sequential representation ( I) and (III) ). Indeed, we shall see, latter on, that the algorithmic of the TB-LR sequential representation is similar to the TB-LR' one.

## **5. The LL-TB sequential representation:**

### **5.1 The principle**

This representation is the one defined in /LIT 85/. An internal node is a 3-uplet with the fields DV , UP and B. DV is the digit value, UP is the upper pointer and B is the leaf bit equal to 0 if the internal node is a leaf and 1 otherwise. We represent neither the level of node nor the pointer to other internal nodes. The level of digit is the level of node in the m-ary trie. Furthermore, as the order of nodes is predefined, we do not need pointers.

### **5.2 Insertion**

We first search the bucket that should contain the key to insert. If the key is not in it and if the bucket is full, then a collision occurs. It will be processed as follows:

Let us consider the shortest sequence  $c^0c^1...c^k$  selected for the split. We first retrieve the first  $I$  digits which already exist in the trie. Let  $Node^i$  be the last node (with the digit  $c^{i-1}$ ) visited at the level  $i-1$ . Then, we insert  $(K-I+1)$  digits, each one of them in the corresponding level. If a single digit is selected by the split, i. e, no nil nodes, one inserts either the node  $(c^k, M, 0)$  into a sublevel or a sublevel with this node in the level  $k$  according to the value of the leaf bit of the last visited node in level  $i-1$  (1 or 0). For the insertion of the nil nodes, if any, we consider two cases:

-Insertion of the first nil node :

The leaf bit of the last visited node in level  $i+1$  is either 1 or 0. For the first case, the sublevel of level  $i$  exists. We insert then the nil node in this sublevel and we bring up-to-date the leaf bit of the node Node'. For the second case, we create a sublevel with the nil node in level  $i$ .

-Insertion of the others nil nodes:

We create sublevels in the level  $i+1, i+2, \dots, k-1$  successively. Every time we create a node in a sublevel, we increase the counter of the number of nodes.

Suppose we want to insert the key 'help' in Fig2a. A collision will occur in bucket 7. The shortest sequence selected for the split is 'he\_'. The sequence 'he' exists already in the trie. We create a node with the following values:

DV='\_' ; UP=11 ; B=0

at level 2. Since the latter does not exist, we create then a sublevel consisting of this node. Further, we associate to it a counter with the value 1. Note that the node's leaf bit with digit value 'i' at level 1 is set to 1.

We obtain then

<7> (a,4,1) (b,10,0) (f,7,0) (h,6,1) (i,2,1) (o,1,0) (t,5,0) <1> (r,9,0) <1> (e,8,1) <1> (\_,3,0) <1> (\_,11,0)

### 5.3 Key search

key search is performed as follows: let  $C=c_1c_2\dots$  be the searched key. We search at level 0 a node with a digit  $d_0$  such as  $C \leq d_0\dots\dots$ . If this node does not exist then the field UP of the last visited node gives the bucket which should contain the key C. Otherwise we analyze the two following cases:

- the node is a leaf and we stop the search. The field UP of the preceding node holds the address of the bucket which should contain the key C.
- the node is not a leaf. We process its sons which are obviously at level 1. Likewise, either we find a node with digit  $d_1$  such as  $C \leq d_0d_1\dots\dots$  or not. We treat this node as previously. And so on...

## 5.4 Sequential search

As we have outlined it above, the traversal of the graph  $G'$  in *postorder* gives the sorted sequence of buckets. In our example file, if we traverse the graph  $G'$  of Fig2a (or Fig2b) in that order, we obtain the following sorted sequence of buckets:

0, 9, 4, 10, 7, 8, 6, 3, 2, 1, 5

As nodes are represented in a predefined order, for each node  $Node$  belonging to level  $i$ , we must compute the address of the next level  $i+1$ , its number of sublevels and the number of the sublevel that contains the sons of the node  $Node$ . This allows us to use a recursive call.

## 6. The TB-LR sequential representation

### 6.1 The principle

In this representation, the trie is a sequel of external and internal nodes. An internal node is a digit; an external node is a pointer to a bucket. The internal nodes are stored by paths. We represent first the internal nodes of the most left path, from top to bottom. Then those of the path immediately to the right in the same order, and so forth. The external nodes follow the internal ones associated with a path. As we represent the internal nodes from top to bottom in a path, their level is shown implicitly in the path. So, in the path  $b_0b_1\dots b_n$ ,  $i$  is the level of digit  $b_i$ . Further, the digits (or internal nodes) are such that  $b_0 < b_1 < \dots < b_n$ . Usually, there are common nodes to many paths. In this manner, they are not duplicated.

### 6.2 Insertion

As described previously, at each new collision, we make the following operations : Let  $m$  be the bucket undergoing the split,  $M$  the next bucket to allocate,  $K$  and  $I$  corresponding the usual parameters. We first search the path of the trie containing the first  $I$  digits. Let  $c'_0c'_1\dots c'_i$  be this path. Then, we insert the sequence  $c'_{i+1}c'_{i+2}\dots c'_k$  such as  $c'_{i+1}$  would be a son of  $c'_i$ ,  $c'_{i+2}$  a son of  $c'_{i+1}$ , and so forth. To respect the order of nodes at each level, the son must be inserted at its appropriate position among this brothers.



Then, we replace the old bucket  $m$  by  $M$ . Finally, we generate  $(K-I-1)$  nil nodes. On the average, an internal node and an external one are created by collision.

As seen previously, if we want to insert the key 'help' in the case of Fig3a, a collision occurs in bucket 7. The sequence chosen for the split is 'he\_'. The internal nodes 'h' and 'e' already exist in the trie. Node '\_' becomes a son of node 'e'. We associate to node '\_' an external node with value 11. The path is then extended by nodes '\_' and 11. We obtain the following representation:

a r <0> <9> b <4> f <10> h e \_ <11> <7> <8> i \_ <6> <3> o <2> t <1> <5>

### 6.3 Key search

Key search is performed as follows: we start by the most left path. Then we go over all the internal nodes, i.e, until an external node is encountered. The maximal key of this bucket is thus  $d_1d_2...d_n::...$  where  $d_1, d_2, ...d_n$  are the nodes of the path. Either the searched key  $C$  is less or equal than this maximal key and the concerned bucket is found, or the encountered bucket is not the right one. In the last case, we take the following external node and the maximal key becomes  $d_1d_2...d_{n-1}::...$ . Then, we repeat the same steps. If there are no external node and the concerned bucket is not found, we go to the path immediately to right, and so forth. The traversal is stopped when the bucket is found.

### 6.4 Sequential search

Sequential search is performed easily. It consist of reading the pointers in order in the linear representation. Although in the TB-LR sequential representation there is only one type of node, the corresponding algorithmic is similar to the one defined in the TB-LR' sequential representation, except that the sequential search algorithm needs, in addition, a stack.

## 7. Analysis.

We will study in this section the memory space and the necessary computing time of key-address transformation's algorithm for the two sequential representations seen above. In addition, we recall this for the standard representation for comparison purposes.

Let  $x$  be the number of records in the file.

- $b$  be the capacity of a bucket

- $f$  be the load factor

$f$  is  $x/(n.b)$ ,  $n$  being the number of buckets of the file

We will take  $b=100$  and  $f=0.7$ .

The ratio  $x/f.b$  represents also the number of buckets in the file. Indeed, we have  $x/(f.b) = x/((x/(n.b)).b) = n$ .

At each split:

-the file is extended by one bucket.

-the trie is extended on the average by one internal node for the standard representation and for the sequential representations defined in the graph  $G'$  (LL-TB and TB-LR) and by one internal node and one external node for the one defined in the graph  $G'e$  (TB-LR').

The size and the structure of the node depend on the chosen representation. We designate by  $M''$  the number of bytes needed for a given representation. For every representation, we will set in the following cases of digits: character, numerical and binary. Then, we will observe the space needed by the tries corresponding to various representations for 35 000, 140 000, 800 000 and two millions of records.  $a$  will designate the number of nil nodes. The field address at a level with nodes takes two bytes, which allows to address a file of 32K buckets.

## 7.1 standard representation

The trie is a linked list of internal nodes. An internal node is a 4-uplet (LP,UP,DV,DN).

-If the digits are alphabetic, we may choose 1 byte for DV and 1 byte for DN. A node is then represented using 6 bytes.

We then have :

$$M'' = 6 (M + a)$$

-4 bits per DV are sufficient for numerical digits. Usually, the numerical key does not exceed 16 digits.

Thus, 4 others bits suffice for DN. It results :

$$M'' = 5 (M + a)$$

-If the digits are bits, the field DV is not necessary since we always have DV=0. Likewise, 4 bits suffice to indicate the number of digits. Therefore :

$$M'' = 4,5 (M + a)$$

The time corresponding to the key-address transformation is  $\text{LOG}_2(M'+a)$ .

## 7.2 The LL-TB sequential representation

The trie is a collection of M-ary vectors of variable length, where each vector is associated to one sublevel (or level 0). An item of a vector is a 3-uplet (DV,UP,B). Let L be the length in bytes of such a node. There is a counter NC in each sublevel which gives the number of nodes. We may choose 1 byte for NC, which allows us to have 256 nodes per vector. Further, for every collision :

- (i)- either we insert a node at a sublevel in which case the trie is extended by L bytes,
- (ii)-or we insert a sublevel at a level in which case the trie is extended by (L+1) bytes.

We can suppose that when the number of keys to insert increases, the insertions of sublevels become rare(to confirm possibly by simulation).

We then have :

$$M'' = L (M' + a) + s$$

s, being the number of insertions of sublevels.

Indeed, if n is the number of nodes inserted at level i, we have

$$M'' = Ln + (L + 1)s$$

$$\text{or } M'' = LM' + s$$

and with respect to nil nodes

$$M'' = L (M'+a) + s$$

We also can write

$$M'' = (L+h) (M'+a) \text{ with } 0 < h < 1.$$

-for the alphabetic digits we have :

$$M'' = (3 + h) (M' + a)$$

-for the numerical digits we have :

$$M'' = (2,5 + h) (M' + a)$$

-For the binary digits we have :

$$M'' = (2 + h) (M' + a)$$

The nodes are visited sequentially, level by level, until the searched key is found. At a level ,

(a) -we consult the fields DV, UP and B of a node belonging to a single level.

(aa) -we only consult the field B of nodes of all the others sublevels.

The number of visited nodes is of the order of  $N/2$ . However, on the average, if we consider time spending in (aa) negligible compared to the one in (a), it would be  $\text{LOG}_b(M'+a)$ , b being the number of sublevels by level.

### 7.3 The TB-LR sequential representation

The trie is a sequel of internal and external nodes. The internal nodes are digits and the external ones are pointers to the buckets. At each collision the trie is extended by one internal node and one external node.

Thus, with the above considerations:

- for the alphabetic digits we have :

$$M'' = 3 (M' + a)$$

- for the numerical digits we have :

$$M'' = 2,5 (M' + a)$$

- for the binary digits we have :

$$M'' = 2 (M' + a)$$

The search is made path by path. On the average, half of the paths is traversed before finding the right bucket. The number of visited nodes is thus of the order of  $N/2$ ,  $N$  being the number of both internal and external nodes. Fig 4 gives some examples.

To sum up, the sequential representation defined in the graph  $G'e$  (TB-LR') is the most compact. But the difference with the ones defined in the graph  $G'$  (LL-TB and TB-LR) is not very important. On the other hand, The algorithmic corresponding to the sequential representations by path (TB-LR and TB-LR') is less complex than the one corresponding to the representation by level (LL-TB). As for the time of calculation of an address, it is the same for the TB-LR and TB-LR' sequential representations, i. e, of the order of  $N/2$ . However, as we have outlined it above, it could be much faster in the LL-TB sequential representation.

## 8. Implementation

If the standard representation does not set problems for its implementation in a language such as PASCAL, FORTRAN, BASIC, the sequential representation could set serious problems. First, access at the bit level is necessary in such representations. Further, the length node, in bits, is not usually a multiple of 8, i.e, the nodes are not on the frontier of byte. As a result, we will have to use a table of bits for the trie representation and, consequently, to write the procedures of management of the table in assembly language.

The problem in the TB-LR' sequential representation is that of the distinction between an internal node and an external one. If we use a code at 8 bits which is very frequent, the presence of a 9th bit is necessary. It is almost impossible to use an array of an evolved programming language because first, the items have the same attributes and second, the problem of frontier will always appear.

An implementation of these two representations has been developed in PL1 with the algorithms described in this paper. We recall that PL1 allows the access of a bit. Further, we dispose of functions of conversion such as

- transformation of a table of bits to a string of bits and vice versa.
- transformation of a string of bits to a character and vice versa.
- transformation of a string of bits to a number and vice versa.

Which facilitates the management of the table.

In our implementation the distinction between an internal node and a external node has been made as follows : An external node is represented by a positive number. the most left bit is thus to 1. The value 0 represents the nil. An internal node is a digit which is represented in EBCDIC. For the alphabetic and numerical digits the left bit is to 1. We represented conventionally the space by X'CO'.

## **9. Comparison to others methods**

Trie hashing was conceived by W. Litwin in 1981. The analysis of the method, mainly by simulation, have been made in /LIT 85/. we present below some results. We first compare the method with b-tree, the most used actually, then with others methods.

### b-tree

In a b-tree, a key is typically found in three to five access disk, depending logarithmically on the file size. The load factor stays close to 70 per cent. The file is ordered. These properties have made b-tree one of the most popular data structures.

In Trie hashing, the search is 2-4 times faster than the one in a b-tree. If for random insertions, the load factor is typically the same, it is 10-20 per cent better than the 50 per cent of a b-tree. This is, of course, in

the case where the trie can fit in the core, i.e. for files attaining millions of records. For larger files, an extension of the method exists /ZEG 87/.

### Others methods

Trie hashing is typically faster than the known algorithms for classical hashing. Contrary to these methods, trie hashing keeps the file ordered and it is devoted also for dynamic files. Among the methods recently conceived, trie hashing is at least two times faster than in Grid files /NIE 84/ and slightly faster than in interpolation hashing /BUR 83/.

## **10. Conclusion**

The sequential representation requires two times less space than the standard one. On the other hand, it presents certain major inconvenient. The representation of a node of a length that is not a multiple of 8 makes their implementation not evident for most of programming languages. Unless to program certain parts in assembly language. The algorithmic is complex. Especially for the LL-TB sequential representation in which we handle arrays of variable length. So, for the machines of 8 bits, we recommend the sequential representations as defined in G'. The LL-TB sequential representation uses more space than the TB-LR one but it could be faster. On the other hand, the TBLR' one is not recommended in such machines. In machines of 7 bits, the TB-LR' is the most efficient. It is the more compact and its algorithmic is simple in comparison with the LL-TB one. In the extension of trie hashing, called multilevel trie hashing /ZEG 87/, we recommend the sequential representations by paths (TB-LR or TB-LR'). In this case, the algorithmic could be simpler than if the standard representation is used. Indeed, the split of the trie is simple and we have not to rotate the trie. As for the time of key-address transformation the standard representation is the faster.

## **Bibliography :**

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

/KNU 73/

D.E KNUTH

The art of computer programming. Vol 3  
Addison Wesley, 1973

/LIT 81/

W. LITWIN

Trie hashing

SIGMOD 81, ACM, (May 1981), 19-29

/LIT 85/

W. LITWIN

Trie hashing, further properties and performances

Int. Conf. On foundation of Data organisation. KYOTO, May 1985

/NIV 84/

Nievergelt, J., Hinterberger, H., Sevcik, K., C.

The grid file : an adaptable, symetric multikey file structure.

ACM TODS, (March 1984).

/ZEG 87/

D.E ZEGOUR - W. LITWIN

Multilevel trie hashing

Int. Conf. of database. Venise, Italy, December 1987.