

# A NEW MULTI-ATTRIBUTES ACCES METHOD FOR VOLUMINOUS FILES

M. ARIDJ<sup>1</sup> & D.E ZEGOUR<sup>2</sup>

Institut National d'Informatique  
Alger, Algeria

## Abstract :

*Multidimensional trie hashing (MTH) access method is an extension of the trie hashing for multikey dynamic files (or databases). Its formulation consists on maintaining in main memory (d) separated tries, every one indexes an attribute. The data file represents an array of dimension (d), orderly in a linear way on the disk. The correspondence between the physical addresses and indexes resulting of the application of the tries is achieved by the mapping function. In average, a record may be found in one disk access, what ranks the method among the most efficient known. The inconvenient of MTH is double :*

*a weak occupancy of file buckets (40-50%) and an increasing of memory space according to the file size (tries in memory). We propose a refinement of MTH on two levels. First, by using of the compact representations of tries suggested in [6], then by applying the phenomenon of delayed splitting ( partial expansion ) as introduced in the first methods of dynamic hashing and as used in [16]. The analysis of performances of this new scheme, mainly by simulation, shows on the one hand a high load factor (70-80%) with an access time practically equal to one disk access and on the other hand an increasing of file size with a factor of two with the same space used by MTH.*

## 1. Introduction

The first multidimensional access methods used the inverted lists technique which associates a secondary index to each attribute. It is clear that this type of methods is limited to static files with modest sizes. This is mainly due to the memory space consumed by the index tables. In order to respond to the requirements of new applications which require innovative solutions to storage and access problems, several multidimensional access methods have been developed. These methods are usually based on B-trees [2] or dynamic hashing.[9,10,11,12,13]. On dynamic hashing, we can take as example grid files [15] and the MTH [19], where the main idea is that the records are represented in multidimensional space . Further, the relationship between the points of this space and file buckets, is achieved by a projection function which makes to each point correspond to bucket address on disk. While other appear methods as interpolation hashing [17] or the EXCEL method [22] by a unique search key, consisting of

the combination of all the attributes. Among the multidimensional access methods based on the B-trees structure we can cite : K-D-tree[1], K-D-B-trees[20], multidimensional B-trees MBT [3,5], R-trees,R+tree [21], HB-tree [4].

In this paper, we suggest a new multidimensional access scheme called Multidimensional compact trie hashing with partial expansion ( MCTHE) which certainly, represents an improved version of multidimensional trie hashing.

We organized the paper as follows: we begin with recalling the multidimensional concepts in section 2. Section 3 recalls the principle of trie hashing, then the one of multidimensional trie hashing, the basis of our work. In section 4, we introduce the new version of MTH, with the refinements evoked beforehand. We will develop all the related algorithms once the compact representation and recalling the principle of partial expansion introducing. Section 5 is devoted to the study of the performances of the proposed schema, mainly by simulation. Section 6 compares MCTHE with MTH at all levels. Finally, section 7 draws a concludes to the paper.

## 2. Concepts of multidimensional

The new applications such as those proposed for meteorological, astronomical and design data base for CAD and VLSI systems, process more and more complex data. It often happens that these treatments are based on several attributes (multikey queries). The new access methods are responsible for such kinds of data allowing thus a multidimensional access to data in order to facilitate the insertion, deletion and different queries operations.

Possible strategies for the developing of the multidimensional access are :

- **The “use multiple (k) single-attribute”**: It consists of using for each key attribute a data structure (B-tree or hashing) independently of other attributes (
- **The “mapping multiple attributes to one”** : It consists of projecting the d-dimensional space formed by the attributes on one-dimensional
- **The “using explicitly multi-attribute indexes”** : The third approach consists of developing methods specially for multidimensional data.

<sup>1</sup> : chargé de cours UHBC chlef  
[aridj\\_moh@yahoo.fr](mailto:aridj_moh@yahoo.fr)

<sup>2</sup> : professeur INI Oued Smar alger  
[dzegour@yahoo.fr](mailto:dzegour@yahoo.fr)

Let us recall the following definitions corresponding to requests on multidimensional files :

- **an exact match query** :determines all the keys which respect the conditions
- $(A1=K1)$  and  $(A2=K2)$  and .....and  $(An=Kn)$  where  $Ki$  is a value of attribute  $Ai$  and  $n$  is the dimension.
- **a partial match query** :is one which specifies values for only some of indexed attributes.
- **a region query** :is one in which a lower and upper bound is specified for each one of the attributes.  $(L1 \leq A1 \leq U1)$  and  $(L2 \leq A2 \leq U2)$  ... and  $(Ln \leq An \leq Un)$

### 3. Multidimensional trie hashing (MTH)

#### 3.1 Trie hashing (TH)[13]

For trie hashing, the file is a set of records identified by primary keys. A key is a sequence of digits of a given alphabet. The records are stored in the buckets on the disk. Each bucket is referred by an unique address and holds a fixed or variable number of records.

The file is addressed through dynamic hashing function which generates a particular binary tree called binary trie or Litwin's tree. The trie consists of two kinds of nodes:

1. The internal nodes are represented by couples  $(d,i)$ , where  $d$  is a digit of a given alphabet and  $i$  the position of this digit in the searched key.

2. The external nodes or leaves represent the addresses of file buckets containing the data file.

In the original paper of Litwin[13], trie hashing is represented in its standard form where all the links are explicit. Therefore each internal node consists of 4 fields. UP :Upper(Right) pointer, LP :Lower (left)Pointer , DV: digit value and DN: digit number. In trie hashing, searching is made by the traversal of the trie down to leaves which contain the record addresses. At most, one access disk is necessary to have the record.[13,14] Record insertion may involve a file extension while record deleting a file contraction. The load factor is about 70 % for random insertions and about 60% - 70% for sorted insertions.

#### 3.2 Multidimensional trie hashing (MTH) [19]

MTH [19] is an extension of trie hashing for the multidimensional access. Its formulation consists to maintain in main memory  $d$  separates tries  $(T_j)_{j=1..d}$ , indexing the various values of the  $d$  key attributes of data file respectively(fig.1). Conceptually, the buckets of the data file are represented in  $d$ -dimensional space where the  $(d)$  axes are defined by the  $d$  attributes of the data file. So, a space point with coordinates  $\langle I_1, I_2, \dots, I_d \rangle$  represents the bucket returned by the mapping function applied to  $d$ -tuple  $\langle I_1, I_2, \dots, I_d \rangle$ . The mapping function uses the technique of extensible arrays[18]. It may be implemented by using  $(d)$  two-dimensional vectors, or a single three-dimensional array , extensible(s) in only one direction and called index array. These are useful for storing the addresses of each block beginning of buckets added, and the multiplicity factors allowing to compute the offsets to add to the base address in order to locate a multidimensional array item.

More formally, the  $d$  index arrays are of the form :  $B_j[0..U_j, 1..d]$ , with  $U_j$  : maximal Index of the trie  $T_j$ . or

the equivalent index array IXA :  $[0..X, 1..d, 1..d]$ , where  $X = \max\{u_j\}_{j=1..d}$ . the mapping function is given by the following algorithm :

*Algorithm  $f(j_1, j_2, \dots, j_d)$  :*

1. Choose  $t=m$  such as  $B_m[j_m, m] = \max\{B_r[j_r, r]\}_{r=1..d}$
2.  $adr = B_t[j_t, t] + B_t[j_t, r] * j_r$  with  $r = 1 \dots d$  et  $r < t$  ;
3. Return ( $adr$ )

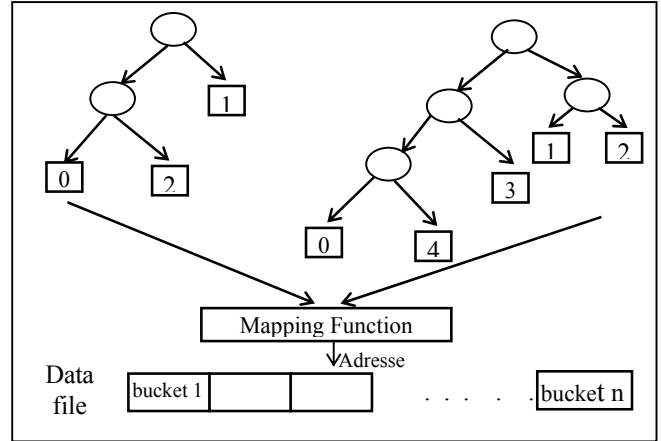


Fig 1 Principe of MTH

The insertion mechanism is as follows : At the beginning, an exact match query is made to find the bucket which must contain the record to insert. Two cases are possible. The bucket is not full and then we simply add the record. A splitting operation is necessary involving then an extension of the multidimensional array(file).

An exact match query is realized by applying to each value  $V_i$  to the trie  $(T_i)$  associated to attribute  $A_i$ . We obtain then a  $d$  tuples  $\langle I_1, I_2, \dots, I_d \rangle$  on which we apply a mapping function to retrieve the bucket address which must contain the records verifying the query condition

A Partial match query of the form  $(A_{i1}=V_1)$  and  $(A_{i2}=V_2) \dots \dots A_{iq}=V_q$  with  $1 \leq q < d$ .

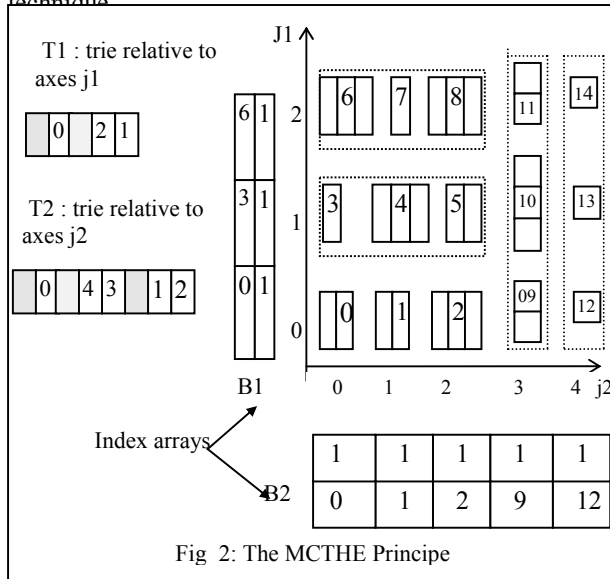
We apply on each specified attribute  $A_{ij}$  in the query by its value  $V_j$  the corresponding trie  $T_{ij}$ . We obtain then a  $d$ -tuples of the form  $\langle *, \dots, I_1, \dots, *, \dots, I_d, \dots, * \rangle$  where  $(*)$  denotes the values of indexes of non specified attributes in the query. The potential bucket addresses to contain the records satisfying the query are obtained by applying the mapping function to the  $d$  tuples  $\langle *, \dots, I_1, \dots, *, \dots, I_d, \dots, * \rangle$  where the  $(*)$  are replaced by all possible values of the corresponding attribute ( These values can be retrieved by traversing the binary trie).

A Region query consists of sweeping all the attributes  $A_j$  between the specified interval in the query.

The analyses of performance made on MTH in [7,18,19] have shown that the bucket load factor is about 40 % or less for random insertion and about 20 % for sorted insertions. This is the major inconvenient of MTH. On the other hand, the access cost of MTM places the method among the fast ones known for the multidimensional access. Indeed, searching a record (exact match query) is accomplished in one disk access in average, while inserting is made in two disk access in average.

#### 4. The proposed schema : MCTHE.

MCTHE -which we suggest- is a variant of MTH, where the tries are stored in a special form on the one hand and the splitting are treated by the partial expansion principle on the other hand.(fig.2) The principle of MCTHE is similar to the one of MTH, except for two following reasons: Firstly the trie are stored in memory in a compact form, secondly, we delay the bucket splitting in order to improve the load factor of the file. So, we recall below, the PP-LR representation suggested in [6] with the corresponding algorithms, then the partial expansion technique



In this example, the data file is composed by 15 buckets which are clustered in 15 segments. The number of partial expansions  $r=3$ . The keys dimension  $d=2$ .

The address computation of an element  $F[j_1, j_2]$  (example  $F[1,3]$ ) is achieved as follows:

1. Find a segment which may contains the bucket's record. Its base addresses is giving by:  
 $\text{Max}(B1[j_1, 1], B2[j_2, 2])$ .
2. Compute the exact address of the elements. For our example  $f(1,3)=B2[3,2]+B2[3,1]*j_1=9+1*1=10$

##### 4.1 PP-LR representation [6]

One of the originality of MCTHE is that the tries are stored in main memory in a compact form where the links are implicit. The basic principle of this kind of representation consists in placing the internal nodes in a predefined order. Thus, we have not need of pointers. The representation that we have used in MCTHE is the called the PP-LR (Path by Path, Then Left to Right) one. For the interested reader, other representations are presented in [6].

In this representation, the trie is a sequence 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 first represent 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  $b_i$ . Then 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.

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  $I$  correspond 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.

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  $d_1d_2\dots d_n\dots$  where  $d_1, d_2, \dots d_n$  are the nodes of the path. Either the searched key  $C$  is less or equal to the maximal key and the concerned bucket is found, or the encountered bucket is not the right one. In the latter case, we take the following external node and the maximal key becomes  $d_1d_2\dots d_{n-1}\dots$ . 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.

Sequential search is easily performed. It consist in reading the pointers respectively in the linear representation.

The PP-LR representations are about two times more compact. The same buffer in core, holds then about two times larger files. But the algorithms are more complex and need more processing time.

##### 4.2 Partial expansion principle (PE)

In the basic access methods (B-tree, hashing), the overloading bucket is immediately split in two buckets implying a local reorganization. This may involve a substantial deterioration of the load factor. In order to avoid this, several propositions have been made. We can cite as example, virtual hashing [11,12] in where a splitting is made in case the load factor attains a predefined threshold (a) This allows to control the load factor of the file and then to keep a good loading [11]. In B-trees, we can cite The B\*-trees proposed by Knuth [8] where a bucket is split only when the brother buckets are full. In this situation there is a splitting of two buckets in to three. This warrants a minimal load factor of 66% and in average it is about 81% [8]. These propositions have been generalized by the end of 1980's and have introduced a new concept called : **partial expansion** (PE). PE consists in gradually increasing the size of the overloading bucket instead of splitting it immediately. The splitting is made when a maximal size is reached. Each expansion is called a

partial expansion. The process which allows a bucket to grow from its minimal size until its splitting (maximal size) is called a full expansion. We call a growth sequence of a bucket the sequence  $s_0, s_1, \dots, s_{r-1}$  of  $r$  ( $r$ : number of steps or expansions) of different bucket sizes with  $s_0 < s_1 < \dots < s_{r-1}$ . A growth sequence is valid if  $2s_0 > s_{r-1}$ . In practice, we choose the sequence  $: rb, (r+1)b, \dots, (2r-1)b$ ,  $r$  being the number of expansions and  $b$  the size bucket.

An analysis of performances on B+-trees with the partial expansion [16] have shown that :

- With 2 partial expansions we have a load factor 81 % , improving thus of 10 % the one of B+-trees. The cost of the insertion is practically the same for the B+-trees and BE-trees.

- the BE-trees with 2 partial expansions performs practically the same load factor than the one of B\*-trees [8] but with an access time significantly lower down.

#### 4.3 Algorithm of MCTHE

##### Insertion

When inserting a new key  $K=(k_1, k_2, \dots, k_j, \dots, K_d)$ , we search first the bucket  $C$  which must containing  $K$ . We have two cases Case 1 : bucket  $C$  is not full, the record is inserted in  $C$ . Case 2: bucket  $C$  is full. Let  $tc=ib$  ( $i : 1..2r-1$ ), the size of bucket  $C$ . we have two situations :

1.  $i < 2r-1$  : we proceed with a partial expansion as follows: the bucket size is augmented with one unit, ie. it passes from  $ib$  to  $(i+1)b$ , the record is inserted

2.  $i = 2r-1$  : the expansion is full, bucket  $C$  is split implying an extension of the file as follows :

We choose an axis  $j$  ( cyclically by example), on which the multidimensional array (the file ) extends. The leave node  $I_j$ , the result of the application of the attribute value  $k_j$  on trie  $T_j$ , is split in to two nodes  $I_j$  et  $m_j$ . Then, we must rehash the keys of all the buckets whose have  $I_j$  as value to  $j$ th coordinate ( this group of buckets forms the segment  $I_j$ ) in order to know the buckets which will stay in the segment ( $I_j$ ) and the ones which will be reloaded in the new segment ( $m_j$ ). If during the rehashing of a key, we reach on a nil node (newly generated'), this will be replaced by the next free index in the axis  $J(m_j+1, \dots)$  and a new segment is added at the end of the file.

##### Searching

The exact match query is very simple. For each value  $v_j$ , we apply trie  $T_j$  (hashing function) to obtain the index  $ij$ . If one of the indexes is NIL, the searching algorithm is stopped with failure. Otherwise, the bucket which could contain the record is given by the application of the projection mapping on the  $d$ -tuples of indexes.

In order to respond to a partial match query, we proceed as follows :

1. We apply on each specified value  $V_j(j=1..q)$  the corresponding trie  $T_j$  to obtain a  $d$ -tuples of the form  $\langle *, \dots, i_1, *, \dots, i_q, * \rangle$  (1), where (\*) denotes an index related to a non specified attribute in the query.

2. We compute the addresses of buckets susceptible to contain the searched records by applying the projection function on the  $d$ -tuples (1), where (\*) is replaced by the values, results of traversals of tries associated to the unknown attributes.

In a region query we specify for each key attribute  $k_j$  a whole interval delimited by two values :  $Inf_j$  and  $Sup_j$ . In order to respond to this query we proceed as follows :

1. To each attribute  $k_j$ , we apply the corresponding trie  $T_j$  (mapping function) on the two values

$Inf_j$  and  $Sup_j$ .

2. sweep each attribute  $k_j$  between  $Inf_j$  and  $Sup_j$ .

Notice here that the traversal of tries is very simple because the nodes are stored in *inorder*.

#### 5. Performances

The study of performance on the proposed scheme concerns essentially the load factor, the access cost of insertion, deletion and query operations. It is based mainly on the simulation. So, several parameters are considered : bucket capacity ( $b$ ), the number of partial expansions ( $R$ ), the dimension ( $d$ ) etc. We have observed the behavior of the method through random insertions.

##### 5.1 Load Factor

It is about to observing the changes in the behavior of the load factor according to the maximal number of keys of a bucket ( $b$ ), the dimension of keys ( $d$ ) and the number of partial expansions, let  $r$ . The test of the figure (fig 3) is obtained by the insertion of 400,000 records in a file with the following parameters maximal size of a bucket  $b=75$ ; number of partial expansion  $r=3$ ; the dimension  $d=2$ .

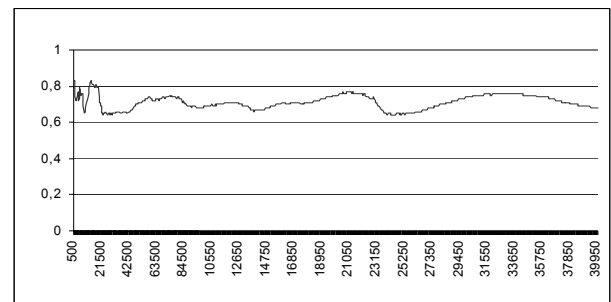


Fig 3 : Variations of the load factor

The main results obtained are :

1. The application of the partial expansion principle involves broadly the load factor It reaches easily the 70 %
2. the bucket size ( $b$ ) have not a great influence on the behavior of the load factor.
3. the load factor for MCTHE is less sensible when the dimension increases.
4. For the sorted insertions the load factor of the file data is weaker compared to random insertions

##### 5.2 Operations

###### Insertion

The tests we have obtained allow to establish the following results :

1. The access cost of an insertion operation is known in advance. It is practically stable and about 2 when the partial expansions principle is not applied, otherwise it is between 3 and 4.
2. The access cost is not according to the number of insertions.
3. The application of the partial expansion principle has not a great influence on the access cost of insertions

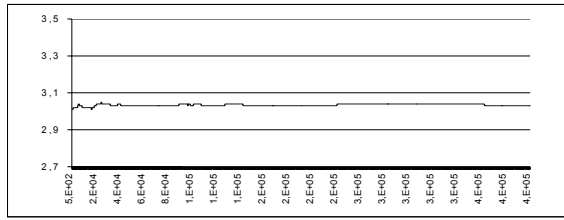


Fig 4: Variations of The access cost of an insertion

### Searching

The access cost of an exact match query (Fig.5-A) is the number of disk accesses necessary to find one or several records responding to a condition. It is practically equal to 1 if the we do not apply the partial expansion principle. It is otherwise near of  $(r/2)$  ( $r$  being the number of partial expansion applied. Note finally that this cost is neither influenced by the dimension  $d$  nor by the number of record present in the file.

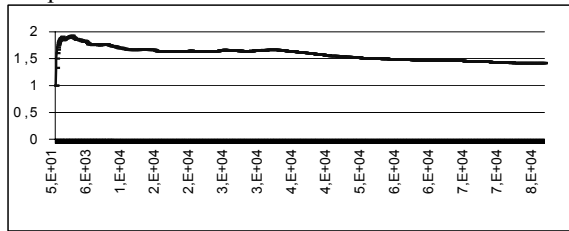


Fig 5.A: Variations of the access cost of exact match query

The cost in access number of a partial match query (Fig.5-B) where we specify  $q$  attribute values on the  $d$  indexes, is in average  $r/2$  access to obtain the first bucket verifying the query. The other buckets are obtained between 1 and  $r$  disk accesses.

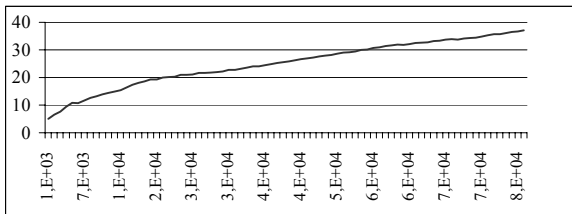


Fig 5.B: Variations of the access cost of partial match query

The access cost of query (Fig.5-C) is goes accordingly with the number of buckets verifying the query. It is between 1 and  $r$  disk access for each bucket visited.

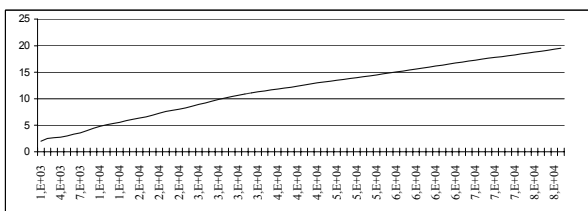


Fig 5.C: Variations of the access cost of a region query

## 6. Comparison with MTH

In this section, we compare the performances of MCTHE with those of MTH[19]. We make this essentially by simulation.

### Load factor

It is certain that the application of the partial expansion principle involves the performances of the load factor.[16]. In order to verify this, we have inserted 80 000 records in a file with size bucket equal to  $b=75$  records by using MTH, then MCTHE for the following various values of  $r=1, r=2, r=3$ .

Notice also that :

1. the performances in load factor for MCTHE with  $r=1$  are practically equal to those of MTH 40 % for the both methods. This is due to the fact that the two methods use the same technique to solve the collisions

2. the application of the partial expansion principle involves broadly the performances in load factor. Morose , we can attain 70 % while it is under 38 for MTH.

### Insertion

For MTH an insertion may cost 2 accesses disk if this it does not imply a splitting. Otherwise, more the blocks size is important, more the access cost is great. The tests we have realized show that the access cost of an insertion is in average equal to 2.

For MCTHE, the insertion cost is equal to 2 disk accesses if no expansion (partial or full) is made. It is of 3 disk accesses when a partial expansion is accomplished, while it is according to the block size added in the case of full expansion The simulation tests realized show that the number of disk access needed for an insertion operation is near of 3 in average .

Notice that MTH and MCTHE offers both promising access performances for the insertion operation

### Searching

One of the advantage of hashing access methods is that offer very good access performances for the search operations broadly best than any other kinds of methods.

Notes :

- for an exact match query : in average less than one disk access is necessary for a record search in a MTH file or MCTHE file without expansion , However, for MCTHE the average access cost of search operation is near of  $r/2$  ( $r$  being the number of partial expansion applied, generally  $r < 4$ )
- partial and region query : the simulation tests show that the two methods (MTH, MCTHE) offer practically the same access performances.

### Memory space

In MTH, tries are implemented by using the standard representation where each trie is a list of internal nodes . At each collision one of the ( $d$ ) tries is extended by one internal node in average. We may use 6 bytes to represent an internal node ( 2 bytes for UP, 2 bytes for LP, 1 for DV and 1 for DN). If  $M$  denotes the number of extensions made on an axis  $A_i$  and  $F$  the number of internal NIL in trie  $T_i$  then the size  $TA_i$  of trie  $T_i$  is given by :

$$TA_i = 6(M+F).$$

Thus for the  $d$  tries we have

$$\text{Size of occupied memory} = \sum_{i=1}^d T A_i \approx d.6.(M + F)$$

In MCTHE we have used the PP-LR representation PP-LR to implement the tries. In this representation, a trie is a sequence of internal and external nodes. An internal node takes up 1 byte while an external one two bytes. At each collision in average an internal node and an external one are added to the trie. With the same suppositions, the size  $T A_i$  of a trie  $T_i$  is  $T A_i = 3(M + F)$ . for the  $D$  tries we have

$$\text{Size of occupied memory} = \sum_{i=1}^d T A_i \approx d.3.(M + F)$$

We may notice for MCTHE the increase of file size by a 2-factor with the same memory space used in MTH.

#### Time of tries traversal

MTH : With the supposition that the tries are balanced, what is the case when the records insertion are random, searching is in  $O(\log_2 N)$ , ( $N$  being the number of nodes in the trie). Thus, the traversal of  $d$  tries is in  $d(O(\log_2 N))$ .

MCTHE : the traversal of trie in PP-LR representation is made path by path. Therefore, to attain an external node, we visit in average  $N/2$  nodes ( $N$  being the number of nodes in the trie). The search algorithm is then in  $O(N)$ , and the traversal of the  $d$  tries is in  $d(O(N))$ .

To sum up, we can see that the time consumed by the traversal of the trie realized by MTH is better than the one realized MCTHE. This has not a great importance because the tries are present in memory.

#### 8. Conclusion

MCTHE is a new scheme of file structures intended to multidimensional dynamic files. It consists on keeping in RAM  $d$  tries under their compact form and delaying the buckets splitting by using the partial expansion principle. The performance analysis of MCTHE shows that:

1. The tries compact form allows to double the file size for the same memory space used by MTH.
  2. The load factor is between 70-80 %, what is broadly better than the one of MTH and even than the most concurrent methods such as ABMD structure.
  3. The insertion access cost varies between 3 and 4 access disk accesses.
  4. The exact match query access cost varies between  $0$  and  $r$  accesses  $r$  being the number of partial expansions.
- All this places the methods among the most efficient known until now. Nevertheless, we think that we may improve this new scheme by controlling the load factor, or by the balancing of tries as made in [19], or yet by envisaging an extension to distributed environment.

#### 9. References

- [1] : J.L. Bentley. « Multidimensional binary search trees in database applications ». IEEE Trans on software engineering SE5(4) :333-340, July 1979.
- [2] : R. Bayer, E. McCreight. « Organization and maintenance of large ordered indexes ». acta informatica 1(3) :173-189, 1972
- [3] : M. Ouksel, P. Scheuermann « Multidimensional B-tree for associative searching in database systems » INFORM systems Vol 7,2 1982
- [4] : G. Evangelidis, B. Salzberg. « using Holey Brick tree for spatial data in general purpose DBMSs ». IEEE database Engineering Bulletin 16(3) : 34-39, Sep 1993.
- [5] : M. Ouksel, P. Scheuermann « Multidimensional B-tree: analysis of dynamic behavior » BIT21 401-418, 1981
- [6] : D. Zegour « Extensions du hachage digital : hachage multiniveaux hachage digital avec représentation séquentielles » thèse de doctorat, université de Paris IX Dauphine 1988
- [7] : W.K. Hidouci - D.E Zegour « Comparisons of B-trees and trie hashing for multidimensional access » 4TH. Maghrebien Conf. On Software Engineering and artificial intelligence. Algiers'96.
- [8] D.E Knuth « The Art of Computer Programming » Vol3 : Sorting and searching, Addison-Wesley, 1993.
- [9] : P. Larson « Dynamic Hashing » BIT 18 (1978),
- [10] : P. Larson « Dynamic Hash Tables » Communications of ACM 4-1988 volume 31.
- [11] : W. Litwin « Virtual hashing a dynamically changing hashing » VLDB 80, ACM, Sep 1978, 517-523
- [12] : W. Litwin « Hachage Virtuel » thèse de doctorat d'état, Paris VI, 1979.
- [13] : W. Litwin « Trie Hashing » SIGMOD 81, ACM, may 1981, 19-29.
- [14] W. Litwin « Trie hashing : Further Properties and Performances » Int. Conf. on Foundation of Data Organization. Kyoto, May 1985, Plenum press.
- [15] : Nievergelt, Hinterberger, Sevcic « the grid file : an adaptable symmetric multikey file structure » ACM Trans data base sys 9 (1) : 38-71 march 1984.
- [16] : A. Ricardo, Baeza-Yates, P. Larson « Performance of B<sup>+</sup>-trees with partial expansion » IEEE Tran. On knowledge and data engineering Vol 1 No 2 June 89
- [17] : W. Bukhard : « interpolation-based index Maintenance ». ACM Trans On knowledge and data engineering Vol 3 1983.
- [18] : E.J. Otoo : « A multidimensional digital hashing scheme for files with composite keys » SIGMOD vol 14,4 (Dec 1985)
- [19] E.J. Otoo : « Multikey trie for scientific and statistical databases » CODATA (North Holland) 1987
- [20] : J.T. Robinson « the K-D-B-trees : a search structure for large multidimensional dynamic indexes » In proceedings of ACM-SIGMOD annual conf on management of data, pp 10-18 New York, April 1981
- [21] : T. Sellis, N. Roussopoulos, C. Faloutsos « The R+-tree : dynamic index for multidimensional objects ». In international conf on very large database page 1-24, Brighton, England, 1987
- [22] : M. Tamminen « The EXELL method for efficient geometric access to data » Acta polytechnica SCANDINAVIA, Mathematics and Computing Science series N 34, Helsinki 1981.