Comparisons of B-trees and trie hashing for multidimensional access*

G.LEVY ** **- D.E ZEGOUR - W.K HIDOUCI**

Institut National d'Informatique, Oued Smar - Alger,

0. Abstract :

With the apparition of dynamic hashing, a multitude of access methods has submerged. These methods have been conceived for multiple keys as well as for monokey access. In addition they give good performance searching time. In this paper, we have analyzed, mainly by simulation, two methods in a multiple attributes environment in order to compare them : multidimensional B-trees (MBT) proposed by M. OUKSEL and P. SCHEUERMANN [Ouk81,Sch82] and multidimensional trie hashing (MTH) conceived by E.J. OTOO [Oto87]. The most significant results were :
1- The access performance for exact, partial and range queries defined on MTH are better than those defined on the MBT method.
2- The average costs of MTH insertion's and deletion's are better than the MBT one's.
3- The load factor is lower for MTH since it is about 40% .

Key Words :

Access method, Dynamic hashing, Trie hashing, B-trees, Multidimensional access.

1. Introduction

Multikey access allows to address files with several attributes. Classic methods use the inverted lists. In other words, to localize a record with a given secondary key we begin by consulting the secondary index, then the primary one before finding the searched record. These methods require as many secondary index tables as secondary keys appearing in a request.

It is clear that the method limits itself to small files since the index tables are assumed to be in RAM. Further, these methods involve periodical reorganizations for dynamic files, because of repeated records deletion by flag.

At the beginning of the 80's, with the introduction of dynamic hashing, several algorithms have been developed for addressing multidimensional files. These methods use either very little RAM for their index or practically nothing at all. We can cite : grid files [Nie84], MTH [Oto87], Interpolated hashing [Bur83]. During the same period, B-trees, considered as the most popular file structure, have been right away extended to multiple dimensions. [Ouk81,Sch82]

We can compare the access method on the following points :

1. Direct access organization or tree structured organization
2. Single attribute access or multiple attributes access
3. Static organization or dynamic organization
4. Order preserved or not.

The ideal is to have a dynamic direct access organization addressing files with multiple attributes and in which the order is preserved. In what follows, we will consider that all records are reduced to their keys. Each

* Master thesis, INI-1993
** Professor at Paris Dauphine University

record can be viewed as an ordered n-uplet (k_1, k_2, ...k_n) of values corresponding to attributes A_1, A_2, ...A_n. As for ordinary B-trees and other tree-indexing methods, we shall assume that the file and the index reside on disk. The bloc is the unit of transfer between RAM and disk. Let us also recall the following definitions corresponding to requests on multiple dimension files :

- an exact match query consists of listing all the keys which respect the conditions :
(A_1=K_1) and (A_2=K_2) and .....and (A_n=K_n) where K_i is a value of attribute A_i 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.
($L_1 \leq A_1 \leq U_1$) and ($L_2 \leq A_2 \leq U_2$) ... and ($L_n \leq A_n \leq U_n$)

The rest of the paper is organized as follows : section 2 and 3 recall the schemes for MBT and MTH organizations and describe briefly the principles of construction and the other related operations. Section 4 presents the comparative study. Section 5 concludes the paper.

2. Multidimensional B-trees (MBT)

2.1 B-trees

Before we proceed with the MBT organization, we briefly review B-trees.

A B-tree of order m is a multiway tree which satisfies the following properties :

- the root node has at least 2 sons
- every other node has between m/2 and m sons
- all leaves appear on the same level

A nonleaf node with k sons contains (k-1) keys and has the following format :

$p_0\ k_1\ p_1\ k_2 \ldots\ k_{k-1}\ p_{k-1}$ where $k_1 < k_2 < ... < k_{k-1}$

p_i points to a subtree containing values between k_i and k_{i+1}, for i=1,2 ..., k-2
p_0 points to a subtree with values less than k_1; p_{k-1} points to a subtree with values greater than k_{k-1}.

Several variants of B-trees have been proposed, as B*trees, where each node is at least 2/3 full and prefixed B-trees which use compression techniques. [Bay77]

More details on B-trees are to be found in [Knu73].

2.2 MBT organizations :

This organization is depicted in figure 1. It uses a directory allowing to index all attributes of the file. The directory is a M-ary tree where each internal node (represented by a triangle) is a B-tree. The internal nodes at the same level in the directory correspond to B-trees indexing different values of a same attribute. Then the root node in the first level corresponds to the B-tree containing all the values of attribute A_1. Each value V_1, in this B-tree, points to one of the sons' nodes in the next level (B-tree of level 2) containing values of A_2 appearing with V_1 in the file. These values of A_2 form the filial set of V_1 at level 2. In addition, each value V_2 of this set, points to one of the sons' nodes (B-tree at level 3) containing all the values appearing together with V_1 and V_2 in the file. This process continues until the leaf node is reached, containing all record addresses which respect the condition :
(A_1=V_1) and (A_2=V_2) and . . . (A_n=V_n)

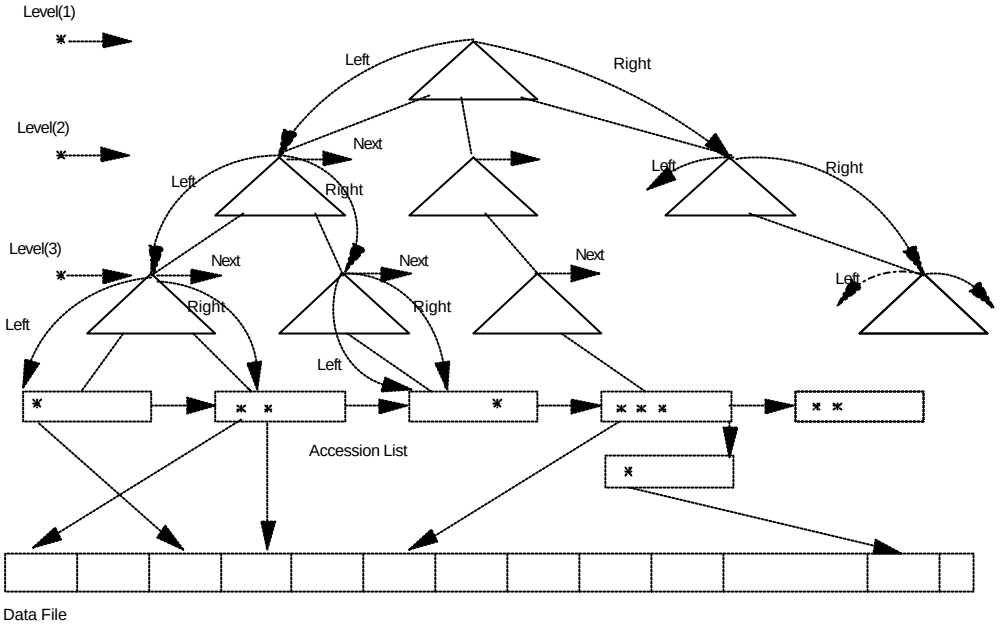
A page inside a B-tree at level h have the following structure :

$P_0\ K_1\ F_1\ P_1\ K_2\ F_2\ P_2$ ... $Km_{h-1}\ Fm_{h-1}\ Pm_{h-1}$ where :

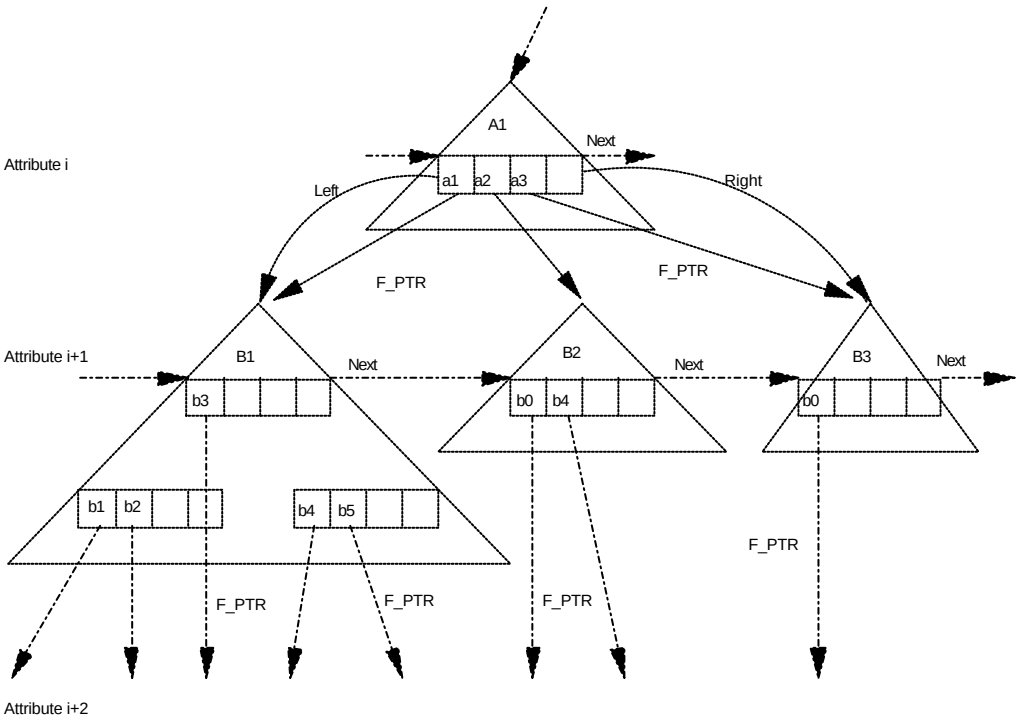
K_j : attribute value.
P_j : Points to a sub-tree, in the same B-tree, containing some values between K_j and K_{j+1}.

F_j : Points to the filial set of K_j in the next level (F_PTR).

Fig 1 : MBT STRUCTURE



- a -



- b -

Figure 1-b shows the relationship between a B-tree at level i and its filial sets at level i+1. The pointers of the filial set associated with values of B-tree at level i are drawn with solid lines. This figure corresponds to the file with keys :

$a_1b_1\ a_1b_2\ a_1b_3\ a_1b_4\ a_1b_5\ a_2b_0\ a_2b_4\ a_3b_0$

$\{b_1,b_2,b_3,b_4,b_5\}$ is the filial set of a_1, and $\{b_0,b_4\}$ is the filial set of a_2. The leaf nodes consist of accession pages containing each pointers to file records. These pages are organized as a linked list (horizontal lists). If one of these pages becomes full, an orthogonal linked list is created for this page.

The root of each B-tree holds three additional pointers : LEFT, RIGHT and NEXT which have the following meaning :

NEXT : pointer to the right B-tree at the same level.
LEFT : pointer to the leftmost node (B-tree) at level k+1 of its filial set.
RIGHT : pointer to the rightmost node (B-tree) at level k+1 of its filial set.
In figure 1-b, these pointers are drawn with broken lines.

In order to access directly to any level i of the directory, we dispose of pointer LEVEL(i) giving the first B-tree of each level i. Thus, with the pointers LEVEL and NEXT, we obtain a linked list of B-trees of a same level.

2.2.1 Construction principle :

It consists of the traversal of the directory from the root down to the leaves. Thus, we determine among the multiple key (a_1, a_2, ...a_n), the first a_j which does not exist on the path. This attribute value (a_j) will be inserted in the filial set of a_{j-1}. The next values (a_{j+1}, a_{j+2}, ... a_n) will be inserted in the new B-trees pointed to by F_j, F_{j+1}, ... F_{n-1}.

In order to maintain the link between the filial sets, we also localize during the traversal the filial sets which will precede and will succeed the new ones.

2.2.2 Other operations :

Deleting consists of removing from the directory all values a_j such as all filial set containing a_{j+1}, a_{j+2}, ...a_n are singletons. In figure 1-b, deleting key (a_2, b_4) consists only of removing b_4 from B2 at level i+1. Deleting key (a_3, b_0) consists of removing a_3 and b_0 from levels i and i+1 respectively.

Exact query consists simply of the traversal of the directory from the root down to the accession pages. At level i, only one B-tree is accessed in order to find the value corresponding to attribute A_i.

The treatment of Partial query become very efficient with the use of pointers LEVEL, NEXT, LEFT and RIGHT.

In order to find all records satisfying the condition :
attribute i_1 = v1 AND attribute i_2 = v2 ... AND attribute i_q = vq where q < n.
We first look for v1 in the linked list of B-trees pointed by LEVEL(i_2). For each successful search, the F-pointers associated with v1, will delimit the filial sets which must be treated in the next level. If this level is specified in the query (i.e. $i_1+1=i_2$), we then look for the next value (v2) in the same manner. If this level is not specified, the LEFT and RIGHT pointers will delimit portions of linked lists containing filial sets which must be treated in the next level.

Because of the ordering imposed on the filial sets on a given level and the pointers used, we can avoid the traversal of the whole directory in the range query. At each level, only two attribute values have to be retrieved : the closest value superior or equal to the lower bound and the closest value inferior or equal to the upper bound. Also, thanks to F-pointers associated with upper and lower bounds, we can delimit the portion of linked list of B-trees at the next level. The process terminates when accession pages are reached. This algorithm uses a breath first search and consequently uses a queue for saving the pairs of pointers which delimit the portions of accession lists containing qualified record addresses.

3. Multikey trie hashing (MTH)

3.1 Trie hashing

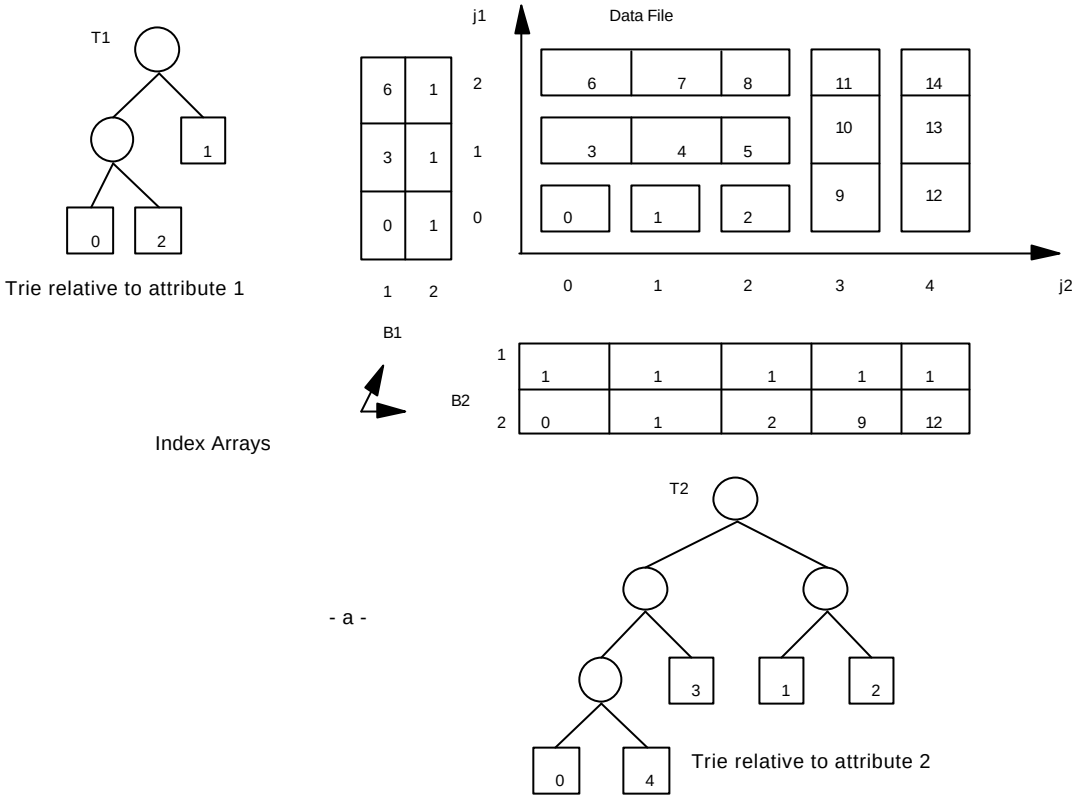
The file is addressed through a trie created dynamically by splitting the overflowing buckets. Every trie node contains a digit value and a digit number which will direct the search process. A leaf either holds a pointer to a file bucket or a Nil value, which indicates that no bucket correspond to the leaf.

More detail may be found in [Lit81,Lit85] and [Zeg87].

3.2 MTH organization

MTH is a new method allowing the access with composite keys. It consists of keeping in core, d digital tries separately, indexing the d different attribute values of the file. The leaf nodes retain index numbers instead of bucket logical addresses, as described in the original version. To localize a record with key (K_1, K_2, ... ,K_d), we map the d keys by applying the d tries, obtaining thus a d-uplet (i_1, i_2, ...,i_d). Finally, the bucket address containing the searched record is computed through the mapping function F which transforms the d-uplet to a linear address. Conceptually, we can say that the file buckets are represented in a d-dimensional space where the d axes are defined by the d attributes (see fig 2-a). A point with coordinates (i_1, i_2, ..., i_d) in the space represent the file bucket numbered F(i_1,i_2, ...,i_d) .

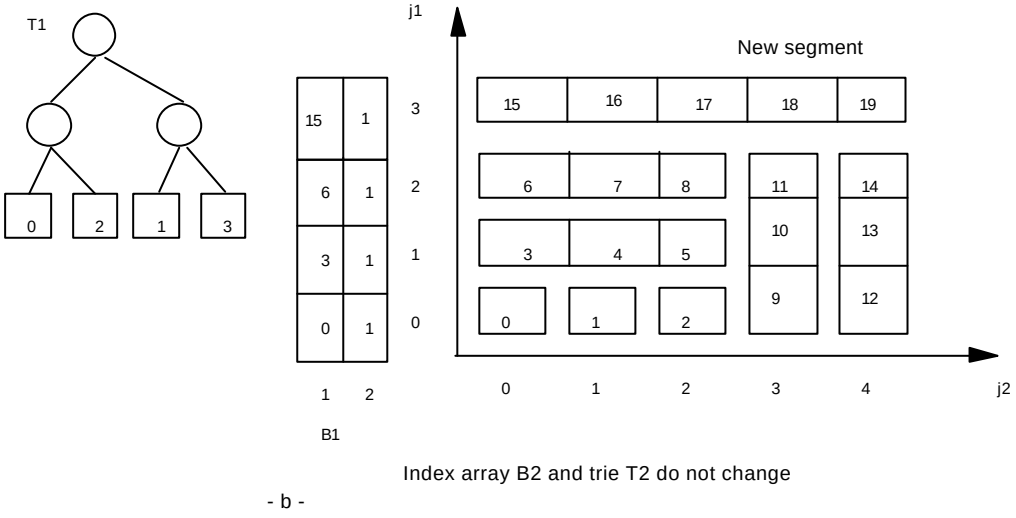
Fig 2 : MTH STRUCTURE (Dim=2)



- a -

After some insertions, buckets 3,4,5,10 and 13 split

with a new segment (buckets 15,16,17,18 and 19) added in J1 direction



- b -

The mapping function uses the technique of extensible arrays [Oto83] which allows to deal with the d-dimensional array, extensible in any direction, with the addition of (d-1) dimensional bloc of buckets at each extension. For MTH, the file represents the array with dimension d stored linearly on disk. The mapping function uses d bi-dimensional arrays holding the first address of each bloc of buckets added. They also hold the multiplicatif factor used in the address computation.

3.2.1 Construction Principle

A record insertion can lead to a collision in a bucket if this one is full. The file is then extended by a whole (d-1) dimensional bloc of buckets added at the end of the file whose size is :

$$\prod (U_j + 1),\ j=1,2,...d \text{ and } j <> t.$$

t being the axis on which the extension is made and U_j being the maximal index trie T_j.
This bloc of buckets (called also segment) represents, in fact, an hyper-plane. Consequently, all the buckets which previously had i_j as common index in their d-uplet addresses are split.
We retrieve this structure in grid files[NIE 84]. Figure 2-b shows an example of file extension by adding a (d-1) dimensional segment of buckets.

3.2.2 Other operations

The process of deleting simply inverts the insertion procedure. If after deleting a record from a bucket, it becomes half-full, we coalesce the segment in question with another.

Exact query is straightforward. For each value v_j specified in the request, we apply trie T_j in order to obtain index i_j. If one of i_j (j=1, ...,d) is Nil, the search fails, else the bucket address which should contain the searched record is given by $F(i_1, i_2, ...i_d)$ where F is the mapping function.

Partial query is one in which only some attribute values are specified. In order to find the others, the traversal of corresponding dimensions in the d-space is necessary.

In order to have the intervals in question in the range query, we proceed in each trie as follow : we first search the key superior or equal to the specified lower bound to find the lower index. During this search, we built the *inorder* stack. Then, we use the latter to have the next indexes.

4. Comparison of the methods :

We can resume the comparison of access costs in the following table (more details are in [Hid93]) :

Operations	MBT [Ouk81,Sch82]	MTH
Exact query	O(Log N)	0 or 1
Partial query	O(N/N' Log N')	0 or 1
Range query	O(Log N)	1
Insertion	O(Log N)	2 or $O(N^{1-1/d})$
Deletion	O(Log N)	between 2 and $O(N^{1-1/d})$

where :

N = File Size
$N' = \prod N_j$ j = d-q+1, d-q+2, ... d
d = the keys dimension
N_j = the average filial set size at level j

4.1 MBT size

We will present in this section an analytic study on the index size (directory) according to the number of inserted keys in the file. The relationship between the number of inserted keys N and the average number of keys contained in a B-tree is given by the following formula :
$N = N_1.N_2...N_d$ where d is the space dimension and N_j is the average filial set size at level j.

By assuming that the inserted keys are uniformly distributed and the range of each attribute is of the same size, we will have :

$N_1 = N_2 = ... = N_d = N^{1/d} = A$

Let α be the load factor of a B-tree. the number of pages contained in a B-tree is evaluated to:

nbp = A / αb , b being the page capacity.

The total number of pages contained in the directory (Npage) is the product of nbp by the number of B-trees (na) in this directory. Since each B-tree of level i holds, on average, A keys, it filial set (at level i+1) consists of A different B-trees.

Thus we have :

the number of B-trees at level 1 is one
the number of B-trees at level 2 is A
the number of B-trees at level 3 is A^2
And that way, we obtain A^{j-1} at level j.

The total number of B-trees is then : $na = 1 + A + A^2 + ... + A^{d-1}$

We then have $Npages = nbp * (1 + A + A^2..A^{d-1}) = (A + A^2..A^d)/\alpha b$

Thus for 100 000 uniformly distributed keys with the following considerations : d=2 ; page size = 512 Bytes (41 keys) ; $\alpha = 70\%$

We will have Npage = 3495.34
As one page needs 512 bytes, 3500 pages will need 1.7 Mbyte on average.

The size occupied by the linked list of accession pages is disregarded in the above calculation.

4.2 MTH size

Let U_j be the number of leaves in Tries T_j, j=1,2...d. Since the file represents a d-dimensional array of size $(U_1*U_2*....U_d)$ the number of file buckets is $N = U_1 * U_2 *....U_d$

The index array IXA(1..U, 1..d, 1..d) where U is Max $\{U_i\}$ takes up Ud^2S where S is the size in bytes of an array item. Trie T_j needs $6U_j$ if the standard representation is chosen (the number of internal nodes is the same as the number of leaves). By assuming the keys uniformly inserted, we can consider:

$U_1 = U_2 = ... = U_d$. We then have $U = N^{1/d}$

The memory space needed by MTH is $d6U + Ud^2S$. Thus for 100 000 uniformly distributed keys, with the following considerations : d=2 ; $\alpha = 70\%$ and page size = 512 Bytes (63 keys)
we need less than 1 Kbytes (960 bytes) in RAM.

4.3 MBT Load factor

We have made many simulations with different numbers of inserted keys, different dimensions and different capacity page in order to achieve to the following conclusions : as for the others methods, the load factor is between 60 and 70 % for random insertions and it is only about 5O% for sorted insertions. The curves depicted in figure 3 and 4 show the evolution of directory load factor for 80 000 and 100 000 random keys insertions with page capacity varying from 10 keys per page (128 Bytes) to 41 keys per page (512 Bytes).
The file load factor is 100%.

Fig 3 : MBT Load Factor
Dim=2 PageSize=128 Bytes

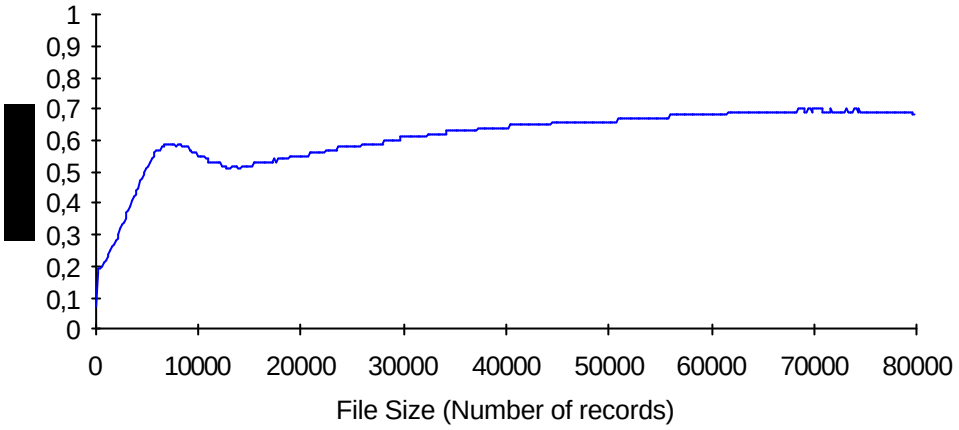
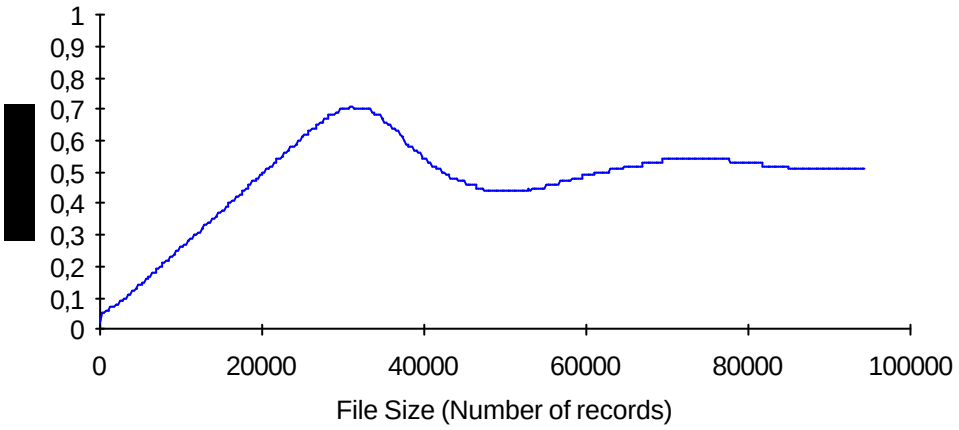


Fig 4 : MBT Load Factor
Dim=2 PageSize=512 Bytes



4.4 MTH load factor

The following curve (Fig 5) shows the average occupancy of file buckets in terms of the file size. First, we observe a low percentage (about 38%) which is mainly due to the insertion principle. The latter consists of extending the file by a (d-1)- dimensional bloc whose size is proportional to the file size. Second, we notice oscillations of strong intensity of load factor at the beginning, that become week.

This behavior is due probably to the fact that the growth factor (GF) of the number of file buckets decreases as soon as this number increases.

In other words, during the first d splits (d being the dimension) the value of GF is one. During the next d splits, GF is only 0.5. GF is 1/3 for the next d splits, and so on...

For example, if d=2 the number of buckets increases as follows (e_j represents the j^{ith} extension) :

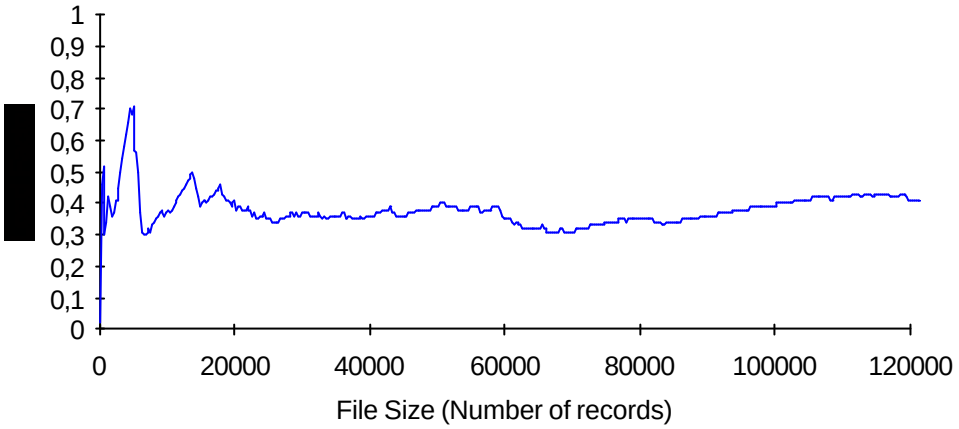
e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	...	...
1/1	2/2	2/4	3/6	3/9	4/12	4/16	5/20	...	...
GF=1	GF=1	GF=1/2	GF=1/2	GF=1/3	GF=1/3	GF=1/4	GF=1/4	...	...

X/Y means that X new buckets are added to the Y buckets of the file during the extension e_j.

In general, the number of file buckets of a d-dimensional file having undergoing n splits, can be approached by :

$N = (1 + n/d)^d$

Fig 5 : MTH Load Factor
Dim=2 PageSize=512 Bytes



Although the segment size added to the file increases in terms of the number of inserted records, these segments represent, on the other hand, fractions smaller and smaller (in 1/n) of the whole file buckets at each extension, which implies the load factor oscilles less and less. The dimension has also an influence on the load factor. Indeed, when the number of attributes is great, the segment size added to the file is greater. Consequently, the load factor lowers. For the sorted insertion, the load factor is yet lower (23%). Indeed, they have many successive splits on the same axis, before extending the others dimensions, which implies many empty buckets or weakly loaded ones.

5. Conclusion

Relatively to access time, MTH offers good performances for the insertion, deletion and different requests compared to the same operations defined on MBT. In the latter, the partial request could be very expensive under some cases. MBT offers, on the other hand, good load factor, which is not the case for MTH. In the latter, more than 60 % of the whole space needed by the file is empty. this is due, mainly, to the poor collision resolution technique used. Indeed, at each collision, the file is extended by segments bigger and bigger. MTH uses RAM to keep the whole index and the array needed for the mapping function. On the other hand, MBT keeps in memory only some buffers to achieve the access operations. Also, MTH is very sensitive to a system crash.

By deeply studying the two methods, we have proposed certain extensions; the orthogonal chaining of accession pages, the notion of logic pages for MBT, and the mechanism of deletion in the case of MTH. Thus, an other way to avoid the weak load factor in MTH, would be to consider the multidimensional array as an index in disk, where each item holds a bucket address.

REFERENCES

[Bay77] : Bayer R. and Unterauer K. Prefix B-trees. ACM TODS,2,1,(Mar 1977),11-26.

[Bur83] : Burkhard W.A. Interpolation-based index maintenance. BIT 23 (1983), 274-294.

[Hid93] : Hidouci W.K. Etude et comparaison des Arbres-B et du Hachage Digital pour l'accès multidimensionnel. Mémoire de Magister - INI 93.

[Knu73] : Knuth D.E. : The Art of Computer Programming. Vol 3, Addison-Wesley, 1973.

[Lit81] : Litwin W. : Trie hashing. SIGMOD 91, ACM (May 1981).

[Lit85] : Litwin W.Trie hashing:Further properties and performance. Int. Conf. F.D.O Kyoto May 1985.

[Nie84] : Nievergelt J., Hinterberger H. and Sevcik K.C. The Grid File:An adaptable, symmetric, multikey file structure. ACM Trans.Database syst. 9,1 (Mar 1984).

[Oto83] : Otoo E.J.and Merrett T.H. A storage scheme for extendible arrays. Computing, 31 (1983) 1-9.

[Oto87] : Otoo E.J. Multikey trie hashing for scientific and statistical databases. CODATA (North Holland) 1987.

[Ouk81] : Ouksel M. and Scheuermann P. Multidimensional B-trees : Analysis of dynamic behavior. BIT 21 (1981), 401-418.

[Sch82] : Scheuermann P. and Ouksel M.Multidimensional B-trees for associative searching in database systems. INFORM Systems Vol 7,2 (1982).

[Zeg87] : Zegour D.E., Litwin W. and Levy G. Multilevel Trie hashing. Rapp. Rech., INRIA, 1987.