Multidimensional Data Indexing in BIG DATA

Aridj Mohamed
University Hassiba benbouli Chlef Algeria

Abstract:

Multidimensional trie hashing (MTH) access method is an extension of the trie hashing for dynamic multi-key files (or databases). Its formulation consists in maintaining in main memory (d) separate tries, every one indexes an attribute. The data file represents an array of dimension (d), in an orderly, linear way on the disk. The correspondence between the physical addresses and indexes resulting of the application of the tries is achieved through the mapping function. In average, a record may be found in one disk access, which places the method among the most efficient known. Yet MTH has the double disadvantage of a low occupancy of file buckets (40-50%) and a greater memory space in relation to the file size (tries in memory).

We propose a refinement of MTH on two levels. First, by using the compact representations of tries suggested in [23], then by applying the phenomenon of delayed splitting (partial expansion) as introduced in the first methods of dynamic hashing and as used in [25]. 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 increase in the file size with a factor of two with the same space used by MTH.

Keywords : Data structure, BigData , hashing , Multidimensional data, data storage

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 low-volume files. This is mainly due to the memory space consumed by the index tables. In order to respond to the requirements of new applications that 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. [5,9,10,11,12,13]. On dynamic hashing, we can take as example grid files [26] and the MTH [19], where the main idea is that the records are represented in multidimensional space . Furthermore, the relationship between the points of this space and file buckets, is achieved by a projection function that makes each point corresponds to a specific bucket address on disk. While other methods are considered such as interpolation hashing [24] 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 [27, 28], R-trees [6], R+tree [21], HB-tree [4, 15].

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

The paper is organized 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 mentioned beforehand. We will develop all the related algorithms after introducing the compact representation and recalling the principle of partial expansion. Section 5 is devoted to the study of the performances of the proposed scheme, mainly by simulation. Section 6 compares MCTHE with MTH at all levels. Finally, Section 7 contains conclusion based on the results of our research.

2. Concepts of multidimensional

The new applications such as those proposed for meteorological, astronomical and design database for CAD and VLSI systems, process more and more complex data. It often happens that such data processing is 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 query operations.

Possible strategies for the developing of the multidimensional access are [3]:

- **The "use of multiple (k) single-attribute":** It consists in using for each key attribute a data structure (B-tree or hashing) independently of other attributes (ABMD[27,28],MDM[19])
- **The "mapping multiple attributes to one"**: It consists in projecting the d-dimensional space formed by the attributes on one-dimensional space. Interpolation hashing [24], EXCEL [22],Z-ordonning [16].

- **The "use of explicitly multi-attribute indexes"**: The third approach consists in developing methods specially for multidimensional data such as the K-D-B-trees[20], R-trees [6], R+tree [21], HB-tree [4,15], grid files[26].

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

- **an exact match query** :determines all the keys which fulfill 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 the indexed attributes.
- **a region query** :is one in which a lower and upper bound is specified for each one of the attributes. (L1≤A1≤U1) and (L≤A2≤U2) ... and (Ln≤An≤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 to by a unique address and holds a fixed or variable number of records.

The file is addressed through dynamic hashing function that 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 involves 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 in maintaining in main memory d separates tries (Tj) 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 <I1, I2 ...Id> represents the bucket returned by the mapping function applied to d-tuple <I1,I2,.....Id>. The mapping function uses the technique of extensible arrays [17]. It may be implemented by using (d) two-dimensional vectors, or a single three-dimensional array, extensible in only one direction and called index array. These are useful for storing the addresses of each block beginning of added buckets, and the multiplicity factors allowing computing 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 : Bj[0..Uj,1..d], with Uj : maximal Index of the trie Tj. or the equivalent index array IXA : [0..X,1..d,1..d], where X=max{uj} j...d. the mapping function is given by the following algorithm :

Algorithm f(j1,j2, ... jd) :
1. Choose t=m such as Bm[jm,m]=max{Br[jr,r]} r=1..d ;
*2. adr=Bt[jt,t]+ ? Bt[jt,r]*jr with r =1 .. d et r<>t ;*
3. Return (adr)

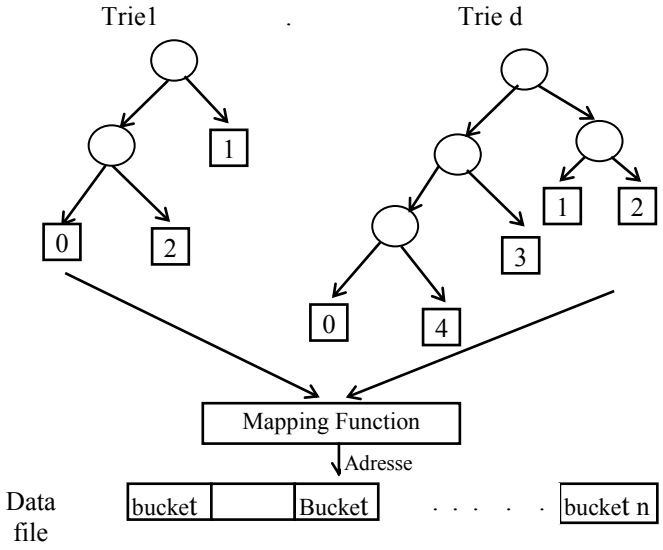


Fig 1 Principle of MTH

The insertion mechanism is as follows: At the beginning, an exact match query is made to find the bucket that must contain the record to be inserted. 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 achieved through applying to each value Vi to the trie (Ti) associated to attribute Ai. We obtain then a d tuples <I1,I2,.....Id> 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 (Ai1=V1) and (Ai2=V2) .....Aiq=Vq) with 1≤q<d.

We apply on each specified attribute Aij in the query by its value Vj the corresponding trie Tij. We obtain then a d-tuples of the form <*,...I1,....,*,....Id,...,*> 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 <*,...I1,....,*,....Id,...,*> 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 in sweeping all the attributes Aj between the specified intervals 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 fastest 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 scheme: 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 is processed by the partial expansion principle on the other hand. (fig.2). The principle of MCTHE is similar to the one of MTH, except for the 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 [23] with the corresponding algorithms, then the partial expansion technique.

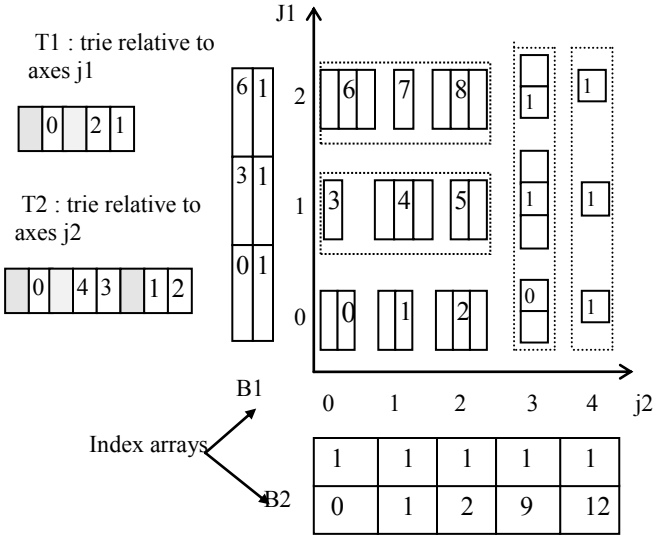


Fig 2: The MCTHE Principle

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[j1,j2] (example F[1,3]) is achieved as follows:

1. Find a segment that may contain the bucket's record. Its base address is given by: Max (B1 [j1,1],B2[j2,2]).
2. Compute the exact address of the elements. For our example f(1,3)=B2[3,2]+B2[3,1]*j1=9+1*1=10

4.1 PP-LR representation [23]

Amongst the characteristics that constitutes 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 no need of pointers. The representation that we have used in MCTHE is the one called the PP-LR (Path by Path, Then Left to Right). For the interested reader, other representations are presented in [23].

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 b0b1....bn, i is the level of digit b. Then the digits (or internal nodes) are such that b0 < b1...< bn. Usually, there are common nodes to many paths. In this way, 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 to the usual parameters. We first search the path of the trie containing the first I digits. Let c'0c'1...c'i be this path. Then, we insert the sequence c'i+1 c'i+2....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 his 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 shown earlier.

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 d1d2...dn::..: where d1, d2, ...dn 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 d1d2...dn-1::..:. Then, we repeat the same steps. If there are no external nodes and the relevant bucket is not found, we go to the path immediately to the 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 twice as compact. The same buffer in core holds then files almost twice as large. However, 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 proposals have been made. We can cite for example, virtual hashing [11,12] in which 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 into three ones. This warrants a minimal load factor of 66% and in average, it is about 81% [8]. These proposals 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 the maximum 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 s0,s1,... sr-1 of r (r :number of steps or expansions) of different bucket sizes with s0<s1<... <sr-1. A growth sequence is valid if 2s0>sr-1.in practice, we choose the sequence: rb,(r+1)b,.... (2r-1)b, r being the number of expansions and b the bucket size.

An analysis of performances on B+-trees with the partial expansion (be) [25] has shown that:

- With 2 partial expansions we have a load factor of 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=(k1,k2..kj...Kd), we search first the bucket C which must contain 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 carry out a partial expansion as follows: the bucket size is increased by one unit, i.e. 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 Ij, the result of the application of the attribute value kj on trie Tj, is split in two nodes Ij et mj. Then, we must rehash the keys of all the buckets that have Ij as value to jth coordinate (this group of buckets forms the segment Ij) in order to know the buckets which will stay in the segment (Ij) and the ones which will be reloaded in the new segment (mj). 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(mj+1 ,...) and a new segment is added at the end of the file.

Searching

The exact match query is very simple. For each value vj, we apply trie Tj (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 through 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 Vj(j=1..q) the corresponding trie Tj to obtain a d-tuples of the form <*,...,i1,*...iq,..*> (1), where (*) denotes an index related to a non-specified attribute in the query.
2. We compute the addresses of buckets that are likely 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 kj a whole interval outline by two values: Infj and Supj. In order to respond to this query we proceed as follows:

1. To each attribute kj, we apply the corresponding trie Tj (mapping function) on the two values

Infj and Supj.

2. Sweep each attribute kj between Infj and Supj.

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

Deletion

Deleting a record with key K consists in retrieving first the bucket, which could contain it by the search algorithm, then delete it from this bucket, if it is present. This operation can involve a deterioration of load factor and the creation of holes. In order to avoid these two problems we can proceed in some situations to merging two blocs of bucket into one. (File contraction)

5. Performances

The study of performance on the proposed scheme mainly concerns 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 observing the changes in the behavior of the load factor in relation 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 performed 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 of keys is d=2.

The main results obtained are:

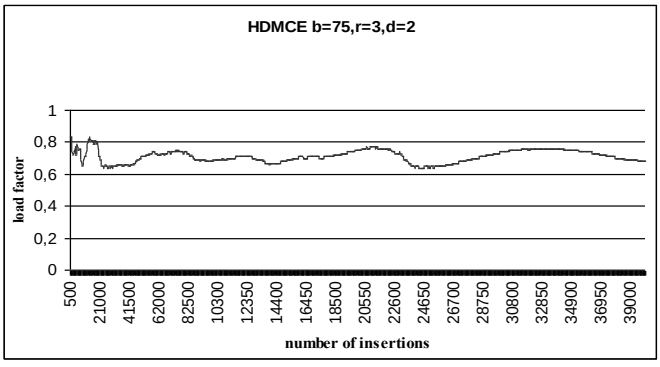


Fig 3: Variations of the load factor

1. The application of the partial expansion principle broadly involves the load factor. It easily reaches 70 %
2. The bucket size (b) does not have a great influence on the behavior of the load factor.
3. The load factor for MCTHE is less sensitive 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 carried out allow us 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 does not depend on the number of insertions.
3. The application of the partial expansion principle does not have a great impact on the access cost of insertions.

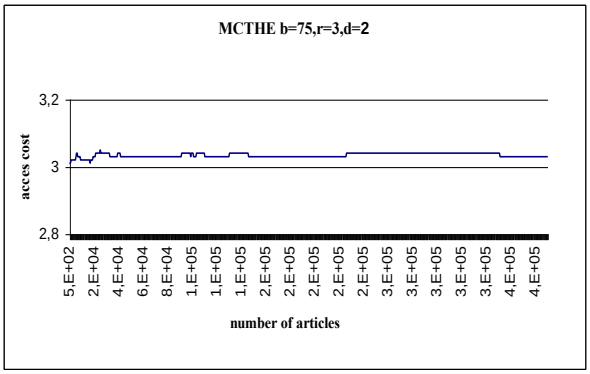


Fig 4: Variations of The access cost of an insertion

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 satisfying 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 influenced neither by the dimension d nor by the number of record present in the file.

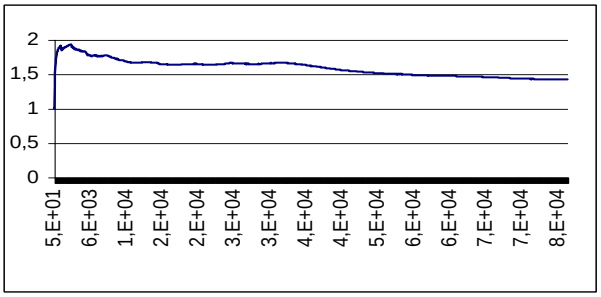


Fig 5.A: Variations of the access cost of an exact match query

The cost in access number of a partial match query (Fig.5-B) in which 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.

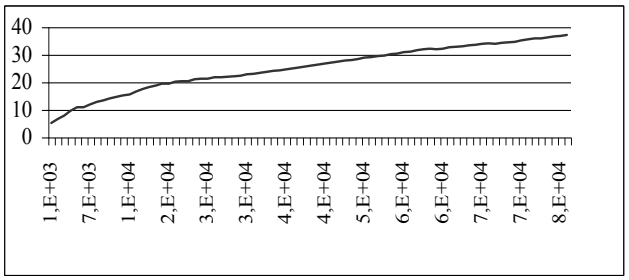


Fig 5.B: Variations of the access cost of partial match query

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

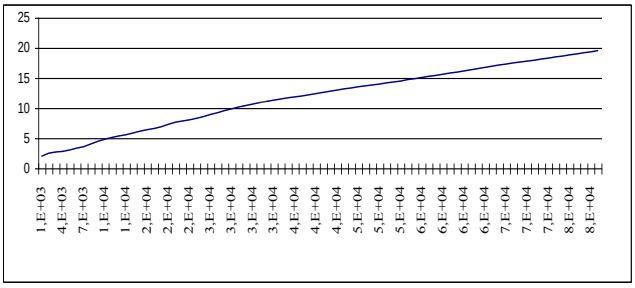


Fig 5.C: Variations of the access cost of a region query

Deletion

A deletion may cost between 2 and 3 disk accesses when the merge process of blocks is not applied. Otherwise, it depends on the number of buckets contained in the merged blocks. The access cost is high enough compared to one of the insertion operation (Fig 5-D)

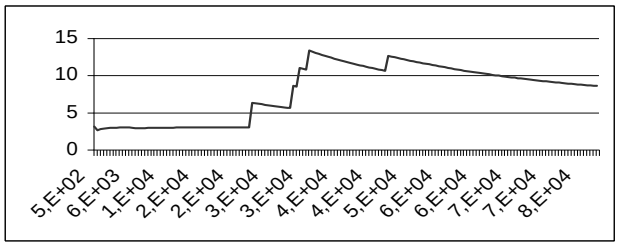


Fig 5.D: Variations of the access suppression

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

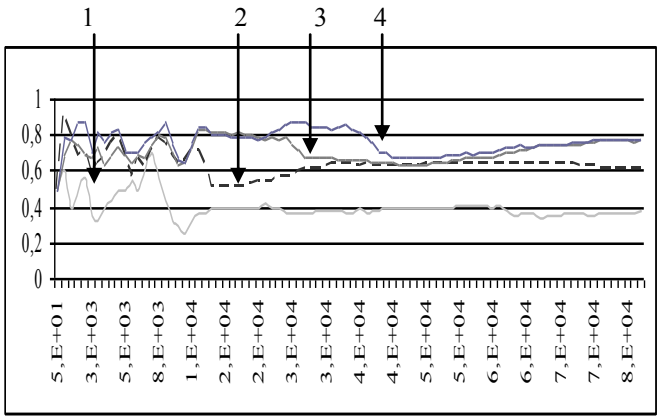
It is certain that the application of the partial expansion principle involves the performances of the load factor. [25]. In order to verify this, we have inserted 80 000 records in a file with bucket size equal to b=75 records by using MTH, then MCTHE for the following various values of

r=1,r=2,r=3. The following figure summarizes this experience.(Fig 6)

Notice also that:

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

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



1: MTH/MCTHE r=1 2: MCTHE r=2
3: MCTHE r=3 4: MCTHE r=4

Fig 6: The load factor comparison (MTH / MCTHE)

Insertion

For MTH an insertion may cost 2 disk accesses if this does not involve a splitting. Otherwise, the larger the size of the blocks is; the more the access cost increases. The tests we have carried out show that the access cost of an insertion is in average equal to 2. (See Fig 7)

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 matches the added block size in the case of full expansion The simulation tests carried out show that the number of disk access needed for an insertion operation is close to 3 in average (Fig 7). Notice that both MTH and MCTHE offer promising access performances for the insertion operation

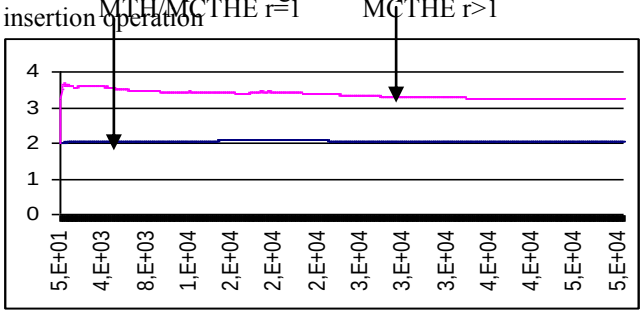


Fig 7: comparison of the insertion cost. (MTH/MCTHE)

Searching

One of the advantages of hashing access methods is that it offers very good access performances for the search operations much better than any other kind 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 close to 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 (Fig 1.d). 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 Ai and F the number of internal NIL in trie Ti then the size TAi of trie Ti is given by : TAi =6(M+F).

Thus for the d tries we have

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

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 TAi of a trie Ti is TAi =3(M+F). For the D tries we have

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

We may notice for MCTHE the increase in the 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, which is the case when the records insertion are random, searching is in O (Log2 N), (N being the number of nodes in the trie). Thus, the traversal of d tries is in d(O(Log2 N)

MCTHE: the traversal of trie in PP-LR representation is made path by path. Therefore, to reach an external node, we visit in average N/2 nodes ((N being the number of nodes in the trie).). Th 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 achieved by MTH is better than the one achieved by MCTHE. This does not have much influence because the tries are present in memory.

8. Conclusion

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

1. The tries compact form allows doubling the file size for the same memory space used by MTH.
2. The load factor is between 70-80 %, which 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 disk accesses.
4. The exact match query access cost varies between 0 and r accesses r being the number of partial expansions.

All this place the methods among the most efficient known until now. Nevertheless, we think that we may involve this new scheme by controlling the load factor as the one proposed in [LRLW91], or through balancing tries as madein [19], or yet by envisaging an extension to distributed environment.

9. References

[1] : J.L.benthty. « Multidimensional binary search trees in database applications ».IEEE Trans on software engineering SE5(4) :333-340, July 1979.

[2] :R.Bayer, E. McCreigth. « Organization and maintenance of large ordered indexes » acta informatica 1(3) :173-189,1972

[3] Georgios Evangelidis «The hB-tree a concurrent and recoverable Multiattribute index structure » . PHD thesis presented to faculty of Graduate school of the college of computer science of Northeastern university.1994.

[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]:R.Fagin, J.Nievergett,N.Pippenger,H.Rstrong «Extendible hashing- a fast method for dynamic files » ACM-TODS,4,3 (Sep 1979), 315-344.

[6]:A.Guttman. « R-trees:dynamic index structures for spatial searching»in proceedings of ACM-SIGMOD annual conf on management of data pp47-57 Boston June 1984.

[7] : W.K.Hidouci - D.E Zegour «Comparisons of B-trees and trie hashing for multidimensional access » 4TH. Maghrebian 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 se 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] : D.Lomet, B.Salzberg. «hB-tree a multiattribute indexing method with good guaranteed performance » ACM Trans on database systems 15(4) : 625-58,Dec 1990.

[16] : J.A.Orenstein, T.Merrett « a class of data structures for associative searching » . in proceedings of SIGART-SIGMOD 3rd symposium on principles of database systems PP 181-190 Waterloo Canada 1984

[17] : E.J. otoo , T.H Merrett. «A storage scheme for extensible arrays » . computing, 31 (1983) 1-9.

[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.Roussopoulus,C.Faloutos « The R+-tree :dynamic index for multidimensional objects » . In international conf on very large database page 1-24, Brighten, 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.

[23] : D.Zegour «Extensions du hachage digital : hachage multiniveaux hachage digital avec repr'sentation séquentielles » thèse de doctorat, université' de Paris IX Dauphine 1988

[24] : W. Bukhard : « interpolation –based index Maintenance ». ACM Trans On knowledege and data engineering Vol 3 1983.

[25] : A.Ricardo,Baeza-Yates,P.Larson « Performance of B^+-trees with partial expansion » IEEE Tran. On knowledege and data engineering Vol 1 No 2 june 89

[26]:Nievergelt,Hinterberger,Sevci « the grid file : an adaptable symmetric multikey file structure » ACM Trans data base sys 9 (1) : 38-71 march 1984.

[27]: M.Ouksel, P.Scheuermann «Multidimensional B-tree: analysis of dynamic behavior» BIT21 401-418, 1981

[28] : M.Ouksel, P.Scheuermann « Multidimensional B-tree for associalive searching in data base systems » INFORM systems Vol 7,2 1982