"Scalable Distributed Compact Trie Hashing" (CTH*)

DR. D.E ZEGOUR
National Computing High School, Algiers

D_zegour@ini.dz
www.multimania.com/zegour

Abstract

This last decade, a new class of data structures named Scalable Distributed Data Structures (SDDSs), is appeared completely dedicated to a distributed environment. This type of data structures opened an important axis of research, considering that the data management in a transparent manner is fundamental in a computer network. All the existing methods are mainly based on Linear hashing (LH) and Range-partitioning (RP*). In this paper, we propose a new method with the constraints of the SDDS. Our approach is an adaptation of the well known method Trie hashing(TH) for a distributed environment, i.e., a network of interconnected computers. The latter uses a digital tree (trie) as access function. Our major objective is the distribution of file buckets and the tree representing the hashing function. We have considered TH with the tree represented in compact form (CTH) because this option is probably more interesting for the reduction of the message size circulating on the network. Contrary to the majority of the existing methods, the proposed one provides the order of distributed files, then facilitates both the range query operations and the ordered traversal of files. Moreover, the following properties make our method a promising opening towards a new class of SDDS : - preservation of the order of records, - works without multicast - three bytes are sufficient to address a server, - the transfer of some bytes is enough for the update of the client trees. The access performances should exceed the ones of traditional files and some competitive scalable and distributed data structures.*

I. Introduction

Current research is emphasized, more and more, on the manner to cooperate several microcomputers (network) in order to make them as powerful (or more) as a supercomputer and this thanks to their accumulated storage capacity and their parallel processing. Such a system is called multi-computers. A striking example is the one of the HP Labs with a few thousands of workstations. Traditional data structures do not suffice on the multi-computer so defined because of the new following requirements: distributed and parallel processing, distributed RAMs, distributed disk files and scalability. Thus, it is necessary to find a data structure adequate to this new type of 'supercomputer'.

This last decade a new class of data structures was born totally devoted for the multi computer. This class of data structures is baptized Scalable Distributed Data Structures (SDDSs). They are based on a client/server architecture. The following properties characterize the SDDS :- Distribution of the file buckets on the servers (at a rate of one server per bucket), - No a main site, - No dialogue between the clients. Each client owns an image of the file. Then, the clients can make addressing errors. They are updated using messages called Image Adjustment Messages (IAMs). The client image is updated gradually until obtaining the real image.

Currently, there are two main classes of SDDS methods quite distinct : the ones which are based on LH and the ones on RP. The SDDS which are defined on LH are methods which do not preserve the order of records. They use some information at the level of the clients for addressing the file usually distributed on several servers [Lit 93], [Lit 93a], [Lit 00]. The SDDS which are based on RP preserve the order of records. There are several variants : RP*N uses no index and operates simply with multicast, RP*C uses a client index with limited multicast, RP*S uses a servers index with multicast optionally. [Lit 93b], [Lit 94]. It was shown that the files SDDS could be faster and larger than the traditional files.

Trie hashing [Lit 81] is one of the fastest access methods for the mono key, ordered and dynamic files. The technique uses a hashing function, which is variable and represented by a digital tree (trie) which grows and retracts according to the insertions and deletions. At the time of its creation, the technique had known much successes. Several works were devoted to it the ten years which followed its creation. We can cite [Lom83], [Tor83], [Lit85], [LIT86], [Zeg87], [Oto87], [Zeg88], [Hid93], [Zeg94]. Why not to distribute such a powerful file structure? Such is our concern in this paper.

There are several ways to represent the access function generated by TH in memory. The compact representations (CTH : Compact trie hashing) we have suggested before [Zeg88], [Zeg94] allow to double the files addressed by the standard representation (TH) for the same memory capacity. Moreover, for a distributed environment this option is probably more interesting, in particular for the transfer the tree parts from a site to another, considering this as a frequent operation in our schema. The idea of the compact representations is to represent the links in an implicit manner to the detriment of algorithms of maintenance slightly longer than those of the standard method. The method consumes only 3 bytes per file bucket, what makes possible to address some millions of records with a small memory capacity.

In this paper, our main interest is the distribution of CTH relatively to the properties of SDDS. We add symbol '*' for underlining the distributive character of the data structure. We will present the proposed schema and will show its validity. The results obtained show that the schema CTH* is promising for the following reasons : - the preservation of the order of records, - works without multicast for all the operations, - three bytes are sufficient to address a server, - the transfer of some bytes is enough for the updating of the trees at the level of the clients.

The next sections in short: II recalls CTH. III describes how the file buckets and the trie are distributed on several servers. IV presents the search and insert algorithms. V provides an illustration of the method and gives some properties of the proposed schema. In section VI, the range query operation is described. A simulation model is presented in section VII followed by some experimental results. Section VIII compares our method with the concurrent method Rp*S which use a B^+-tree. In section IX, some variants are introduced. Finally, section X concludes the paper.

II. Compact trie hashing

We briefly present, in what follows, a scenario showing the construction mechanism of a CTH file by inserting the following keys : " a ce dx ef h x y kx fe hx hy yya yyb yyc "
Let us take a capacity of 4 keys per bucket. At the start, the tree is '|0'. '| ' indicates the largest digit. This means that all the keys are mapped initially in bucket 0.

1. Keys' a' , 'ce', ' dx', and 'ef' are inserted in bucket 0.
The tree is : ***|0***
The file (keys per bucket): Bucket 0: ['a' 'ce' 'dx' 'ef']

2.Insert ' h'
The search of key 'h' gives the address 0 . As the bucket 0 is full, there is a collision. The maximal key of this bucket is '|'. Let us consider the ordered sequence "'a' 'ce' 'dx' 'ef' 'h'". C (the middle key) is 'dx' and C"(the last key) is 'h'. The smallest prefix of digits which makes possible to distinguish C' from C" is 'd'. It is called the sequence of division. The tree becomes ***d0|1****. All the keys lower or equal than 'd' stay on the bucket 0. The others are transferred in a new bucket. The file (keys per bucket) is then: Bucket 0: ['a' 'ce' 'dx']; Bucket 1: ['ef' ' h']*

1. Insert 'x' and 'y'
As 'x' and 'y' are greater than 'd', the search operation gives the address 1. They are inserted in bucket 1 since this one can contain them. Bucket 1: ['ef' 'h' 'x' 'y']

4. Insert 'kx'
The search of key 'kx' gives 1. Bucket 1 is full. There is thus a collision on this bucket. The sequence of division is 'k'. The tree becomes ***d0k1|2****. All the keys lower or equal than 'k' stay on the bucket 1. The others are transferred in a new bucket. The file (keys per bucket) is then :*
The file (keys per bucket): Bucket 0: ['a' 'ce' 'dx'] ; Bucket 1: [' ef' 'h' 'kx'] ; Bucket 2: ['x' 'y']

5. Insert 'fe'
The insertion of key 'fe' changes the bucket 1 as follows : Bucket 1: ['ef' 'fe' 'h' 'kx']

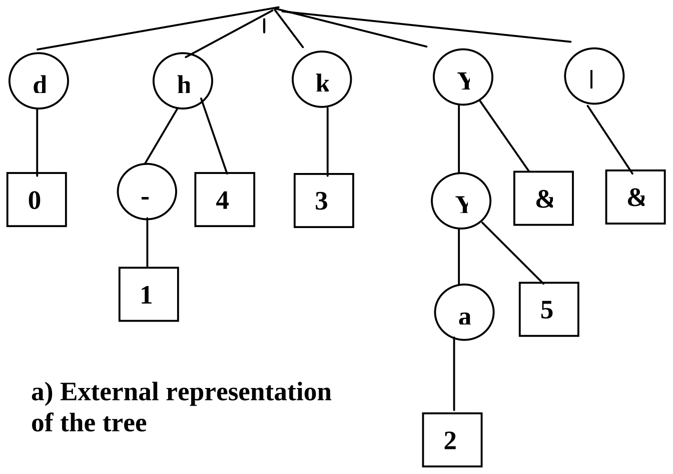
6. Insert 'hx'
The insertion of key 'hx' leads to a collision on bucket 1. The sequence of division is 'h'. The tree becomes: ***d0h1k3|2****. The keys of the sequence 'ef' 'fe' 'h' 'hx' 'kx' which are smaller than 'h' stay in bucket 1. The others migrate to the new bucket. The file (keys per bucket) is : Bucket 0: ['a' 'ce' 'dx']; Bucket 1: ['ef' 'fe' ''h' 'hx'] ; Bucket 3: ['kx'] ; Bucket 2: [' x' 'y']*

7. Insert 'hy'
The search of key 'hy' gives bucket 1. This undergoes thus a collision. The sequence of division is 'h_'. Digit 'h' already exists in the tree. Only digit '_' is added. The tree changes as follows: ***d0h_14k3|2****. All the keys smaller than 'h-' remain in bucket 1. The others go towards a new bucket. The file (keys per bucket) becomes :*
Bucket 0: ['a' 'ce' 'dx'] ; Bucket 1: ['ef' 'fe' 'h'] ; Bucket 4: ['hx' 'hy'] ; Bucket 3: ['kx'] ; Bucket 2: ['x' 'y']

8. Insert' yya' 'yyb'
The insertion of keys 'yya' and 'yyb' changes bucket 2 as follows :
Bucket 2: ['x' 'y' 'yya' 'yyb']

9. Insert 'yyc'
The insertion of key 'yyc' causes a collision on bucket 2. The sequence of division is 'yya'. None of the three digits exists in the tree. The three digits are thus inserted with the creation of two Nil nodes and a new bucket. What gives the tree: ***d0h_14k3yya25Nil|Nil*** *. The file (key per bucket) is altered as follows :*
Bucket 0: ['a' 'ce' 'dx'] ; Bucket 1: ['ef' 'fe' 'h'] ; Bucket 4: ['hx' 'hy'] ; Bucket 3: ['kx'] ;
Bucket 2: ['x' 'y' 'yya'] ; Bucket 5: ['yyb' 'yyc']

Figure 1 gives the file (buckets) and the final tree generated by the method for the above example. & represents the Nil node which means that, for the meantime, no file bucket is allocated for some path.



a) External representation of the tree

d0h_14k3yya25Nil|Nil

b) Internal representation of the tree

c) File

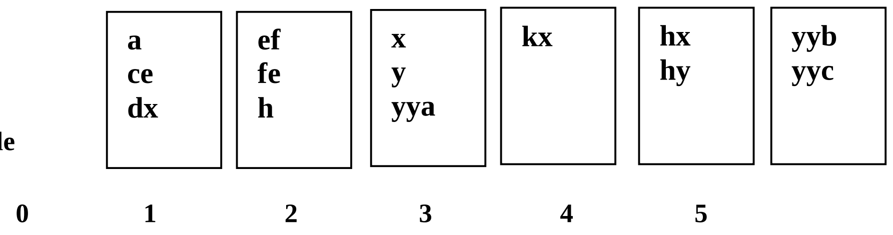


Fig1. File and tree in its internal and external representations.

The buckets are at the level of the leaves. The concatenation of the digits on a path of the tree represents the maximal key of the bucket appearing in the corresponding leaf. Thus, the representation corresponds to the *preorder* on this new tree. In this representation, the digital tree is a sequence of internal and external nodes. An internal node is a digit. An external node is a pointer towards a bucket of the file.
The buckets are ordered from the left towards the right. In the example: 0 1 4 3 2 5.

We present in what follows the insert process. More details on the other operations can be found in **[Zeg 94].**

Insertion

The process of an insertion can be defined as follows:
Let us consider **m** the bucket to be split.
Form the ordered sequence of the keys of this bucket with the key which caused the collision.
Let it be **C'** the key of the middle of this sequence and **C''** the last.
Determine the smallest prefix **Seq** of digits in **C'** which makes possible to distinguish **C'** from **C''**. Let it $C'_0C'_1 ..... C'_K$ be this prefix and **I** the first digits of this prefix which already exist in the tree.

The expansion of the tree is done as follows:
Consider that **Ind_d** is the index in the tree of the first digit in the maximal key associated to the overloaded bucket and **Ind_m** the index of the overloaded bucket.

```
If I <> 0
    If Cm is a prefix of Seq
        Ind_d := Ind_m
    Else
        Ind_d := the index in the tree of the first different digit of Seq in Cm
    Endif
Endif

Case K-I = 1 :
    Replace m by M
    Insert into the position Ind_d the sequence $C'_k M$

Case K-I > 1 :
    Replace m by Nil
    Insert into the position Ind_d the sequence $C'_{i+1} C'_{i+2} .....C'_k$ m M $Nil_1 Nil_2 .....Nil_{k-I-2}$
```

In the first case two nodes are added, in the second case 2(K-I) nodes.
M being the next bucket to be allocated to the file.

Many examples are in the scenario shown above.

III. The concepts of Distributed Compact Trie Hashing (DCTH)

At the level of each client there is a partial digital tree from which any operation on the file begins.
At all moment, any client can enter in scene with an empty tree (| 0)
During the search phase, common to all the operations, the tree is updated gradually until obtaining the real tree. The process of updating the clients is described further.

The Nil nodes at the level of the clients are represented by a negative server number, which means they reference secondary intervals inside the server.

At the level of each server there are:
- A digital tree
- A bucket containing the records of the file
- A primary interval [Min, Max]
- A list of secondary intervals

The Nil nodes at the level of the servers exist and are of two types : those which are associated to secondary intervals and those which are added when a empty tree is generated for a server newly created.

When a server bucket becomes full, it is split and then generates a new server. The digital tree of a server is extended to each division. Thus, it keeps the trace of all the splits on this server. The primary intervals of the two servers are also updated. Rarely, one or more secondary intervals can be added in the overloaded server if the Nil nodes are created. The process of splitting is given further.
The digital trees are represented in a *preorder sequential form* of the new tree as explained in the previous section.
The expansion of the file is done through collisions. At each collision, there is a distribution of the file (splitting of the server) on a logical server. The number of servers is conceptually infinite. The server can be determined in a static or dynamic way. We can have several logical servers for the same physical server.

The system is initialized with only server 0 with an empty bucket, the greater possible interval]λ, Λ] as primary interval and with tree : | 0 which maps server 0 for all keys. We suppose that λ is the smallest key and Λ is the largest one. Thus, Λ = '|||…' and λ ='____…' if '_'is the smallest digit and '|'the greatest one. The trees at the level of the clients are initialized to | 0.

IV. Search and insertion operations.

We organize this section as follows. The algorithm which transforms, from a tree of a given client, a key to a server address is first presented with a flowchart and comments. Hereafter, the updating algorithm is given with some examples. The insert process and the splitting algorithm will follow. We finish this section by underlying some properties of the proposed scheme.

Transformation (Client , key) to a server address

The following figure (Fig. 2) gives the algorithm which transforms the pair (Client, Key) to a couple (Server, Rank). If Rank = 1, the result is the primary interval of server, otherwise the result is a secondary interval of server and Rank designates then its rank.

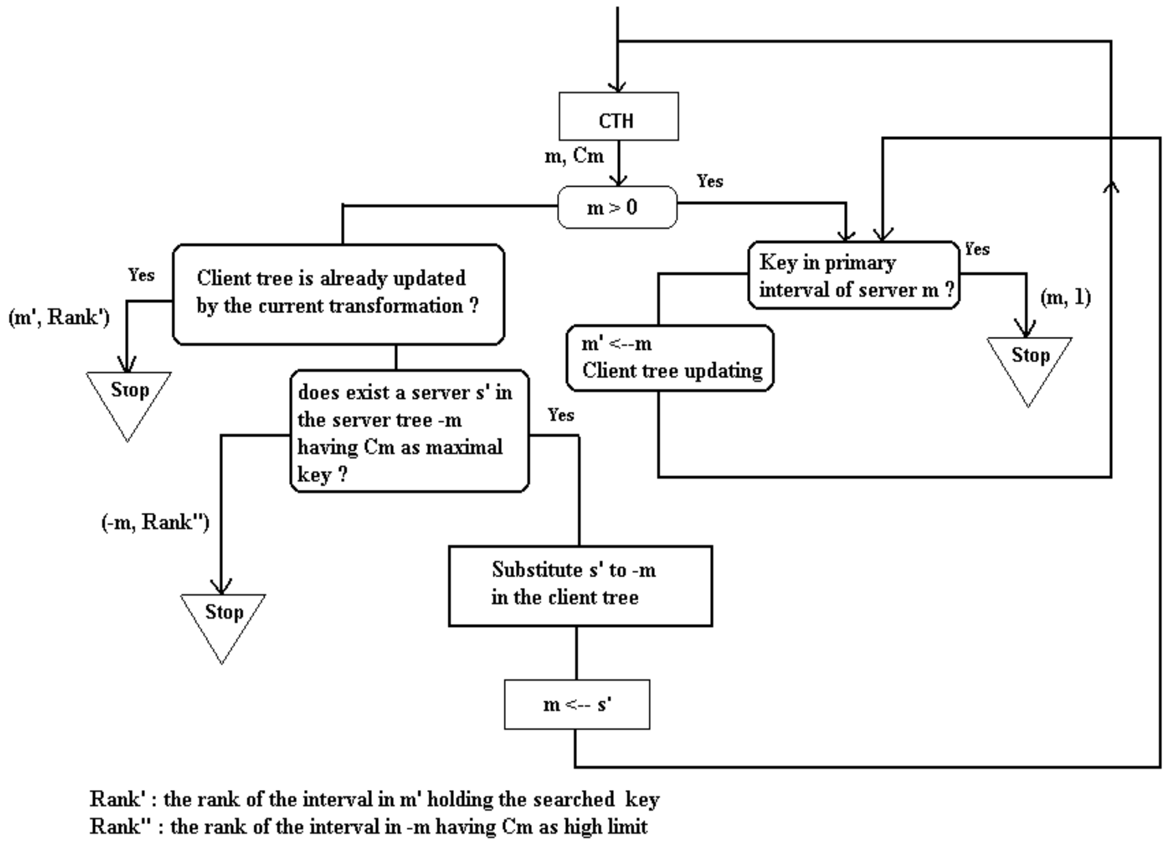

CTH
m, Cm
m > 0
Yes
Client tree is already updated by the current transformation ?
Yes
(m', Rank')
Stop
does exist a server s' in the server tree -m having Cm as maximal key ?
Yes
(-m, Rank")
Stop
Substitute s' to -m in the client tree
m <-- s'
Key in primary interval of server m ?
Yes
(m, 1)
Stop
m' <--m
Client tree updating
Rank' : the rank of the interval in m' holding the searched key
Rank" : the rank of the interval in -m having Cm as high limit

Fig 2. The transformation algorithm in CTH*.

Comments

A search emanating from a given client starts with the traversal of its local tree. A few servers can be traversed according to the following process in order to determine the correct one :

1. Apply the Key→ Server transformation algorithm on the client tree. Let it ***m*** *be the server found and* ***Cm*** *its maximal key.*

2. If ***m*** *< 0, it's question of a secondary interval inside server* ***m****. Two cases are to be considered :*
- The client tree is already updated by the current transformation and ***m'*** *designates the server on which the updating algorithm has been applied (the old value of* ***m****). In this situation the result of the transformation is the couple (****m'****,* ***Rank*** *),* ***Rank*** *being the secondary interval associated to the Nil node which has been transferred to the client by the updating algorithm)*
- The client tree has not yet been updated by the current transformation and therefore ***-m*** *is the server which held the Nil node corresponding to the maximal key* ***Cm****. If there is in the tree of the server* ***-m*** *a server* ***s'*** *corresponding to* ***Cm****, this means that the Nil node has been replaced by a server and then the transformation process continues in this server* ***s'*** *(step 3 with* ***m =s'****).* ***s'*** *replaces then* ***-m*** *at the level of the client. In the case*

*where **s'** does not exist, this means that the Nil node has not been replaced and the result is the couple **(-m, Rank)**. **Rank** is the secondary interval in the server **-m** having **Cm** as upper limit.*

*3. If **m** >0, it's question of a primary interval inside server **m**, then if the searched key is in the primary interval then the result is the couple **(m, 1)**.*

*4. Otherwise, set **m'=m**, Apply the algorithm of client tree updating and restart from 1.*

Client tree updating algorithm

We begin by defining some terms from examples. Then, we present the algorithm followed by some examples.

Terminology

The tree 'a r 0 9 b 5 f g h 3 2 1' holds
- three subtrees of levels 0 : 'a r 0 9', 'b 5' and 'f g h 3 2 1'.
- five subtrees of level 1 : 'r 0', '9', '5', 'g h 3 2' and '1'.
- Three subtrees of level 2 : '0', 'h 3' and '2'
- one subtree of level 3 : '3'.

Each subtree consists of paths. For example 'a r 0' and 'a 9' are the paths of the subtree 'a r 0 9'.
A path consists of two parts : digit part and number path.
If P is a path then Digit (P) represents its digit part and Number(P) represents its number part.
Example: Digit('a r 0') is 'a r' and Number ('a r 0') is 0. In others words, Digit(P) is the maximal key of bucket Number(P).

Algorithm

We give in what follows, the global algorithm which adjusts the client tree according to the server tree.
*Let **Cm** = $C_0C_1...C_k$ be the maximal key of the searched bucket (server **m**).*
*(i) Determine in the client tree the number of internal nodes which precede **m**. Let **Ni** be this number. Consider **Ne** : = length(**Cm**) – **Ni**. **Ne** represents the number of digits that exist in the server tree. Ne is also equal to **k+1 – Ni.***
*(ii)Determine in the server tree all the paths **b** of sub trees of level **Ne** such as Digits(**b**) + '|' ≤ ($C_{Ne}C_{Ne+1},\ldots C_k$) + '|' .*
*(iii) Replace in the client tree ($C_{Ne}C_{Ne+1},\ldots C_k$) **m** by the paths **b** Found*

The expression ($C_{Ne}C_{Ne+1},\ldots C_k$) is the null string if Ne > k.

Note : In order to minimize the number of bytes to transfer from a server to a client, the steps (ii) and (iii) of the above algorithm are refined as follows :

*(ii)Determine in the server all the paths in the subtrees of level **Ne** such as Digits(**b**) + '|' < ($C_{Ne}C_{Ne+1},\ldots C_k$) + '|' .*
(iii)
(a) Case where no path exists :
*Replace, in the client tree, **m** by **M**, **M** being the external node of the first path in the server tree.*
(b) Case where at least one path exists :

Let $\boldsymbol{b_{Last}}$ *be the last path,* **L** *the length of Digits(*$\boldsymbol{b_{Last}}$*) and* **M** *the external node of the path immediately to the right of* $\boldsymbol{b_{Last}}$*.*
(i)If $L \geq 2$ *Replace* $\boldsymbol{C_{Ne}} \dots \boldsymbol{C_{Ne+L-2}}$ *by the paths* **b***, otherwise insert the paths* **b** *before* $\boldsymbol{C_{Ne}}$*.*
(ii)Replace **m** *by* **M**

Some examples

Example 1
Suppose we have a client with the following part of the tree : … **q m 16 q c 41 36 35 r e** …
And suppose that the tree of server 41 is **q p h 41 238 q c 229 Nil Nil**
Let us apply the updating algorithm on server 41.
Ni = 2 ; **Ne**= 1 ; **Cm**='qqc' ; **k** = 2
The subtrees of level Ne = 1 are 'p h 41 238' ; 'q c 229 Nil'
The paths of subtrees of level **Ne**=1 less or equal to 'qc'+'|' are 'ph' + '|' , 'p'+'|', 'qc'+'|'
There is replacement of 'q c' 41 by 'p h 41 238 q c 229'
The client tree becomes … **q m 16 p h 41 238 q c 229 36 35 r e** …

Example 2
Suppose we have a client with the following part of the tree : **… t b … m 111 70 u f ...**
And suppose that the tree of server 70 is **t p 70 x 211 y 147 141**
Let us apply updating algorithm on server 70.
Ni = 0 ; **Ne**= 1 ; **Cm**='t' ; **k** = 0
The subtrees of level **Ne** = 1 are 'p 70' ; 'x 211' ; 'y 147' ; '141'
The paths of subtrees of level **Ne**=1 less or equal to '|' are 'p' + '|' , 'x'+'|', 'y'+'|', '|'
There is replacement of 70 by 'p 70 x 211 y 147 141'
The client tree becomes **… t b … m 111 p 70 x 211 y 147 141 u f ...**

Example 3
Suppose we have a client with the following part of the tree **… i g 3 u 33 18 j …**
And suppose that the tree of server 33 is **i t 33 u 64 Nil**
Let us apply updating algorithm on server 33.
Ni = 1 ; **Ne**= 1 ; **Cm**='iu' ; **k** = 1
The subtrees of level **Ne** = 1 are 't 33' ; 'u 64' ; 'Nil'
The paths of subtrees of level Ne=1 less or equal to 'u|' are 't' + '|' , 'u'+'|'
There is replacement of 'u 33' by 't 33 u 64'
The client tree becomes **… i g 3 t 33 u 64 18 j …**

Insertion process

Any insert operation begins with a phase of transformation (client, Key) ---> (m, Rank)

If ***Rank*** *<> 1, it's question of a secondary interval inside server* ***m***
(i) Transform the secondary interval to a server newly created, ***N****.*
(ii) Initialise this server with a bucket consisting of the new key, an empty tree and the same interval that the one of secondary interval. The empty tree is generated from the upper limit of the secondary interval.
(iii) Adjust the server tree by replacing Nil by ***N****.*
(iv) Adjust the client tree by replacing Nil by ***N****.*

if ***Rank*** *= 1, it's question of a primary interval inside server* ***m****.*

*(i) If **Key** is not in the server bucket and the server is not full then insert simply **Key** in the server bucket.*
*(ii) If **Key** is not in the server bucket and this one is full there is collision. We split only the server tree.*

In what follows, we give firstly how the tree of the server is modified and secondly how the server is split with the eventual generation of secondary intervals.

Splitting process

The process of splitting is as follows:

- *Form the ordered sequence consisted of the records in the server bucket and the new record.*
- *Determine the sequence of division **Sq**.*
- *Split the server bucket into 2 according to **Sq**.*
    - *The old server **m** contains the keys $\leq$ **Sq**.*
    - *Primary_Interval (**m**) becomes] Inf (Primary_interval(**m**)), **Sq**]*
    - *The new server **M**, contains the remainder. Primary_interval(**M**) is]**Sq**, $\mathbf{Sq}_{-1}$] .*
- *Eventually, generate the following secondary intervals*
  ]$\mathbf{Sq}_{-1}$, $\mathbf{Sq}_{-2}$]
  ]$\mathbf{Sq}_{-2}$, $\mathbf{Sq}_{-3}$]
  …
  ]$\mathbf{Sq}_{-i}$, Sup(**m**)]
- *Modify the server tree according to the sequence of division as this is made in section II.*

The notation $\mathbf{Sq}_{-k}$ designates the string **Sq** without the **K** last digits.

A particular treatment is made when there is creation of Nil nodes in order to maintain the partition. We must generate a secondary interval to each Nil node. These secondary intervals can be kept in the split server. further, a secondary interval becomes a server when a key is inserted in it. The number of secondary intervals is very weak compared to the number of servers because the method CTH generates 5% of Nil nodes on the average for the random insertions [LIT 85].

Properties of the schema

The following properties hold :
- The schema is deterministic : all the operations are performed without multicast.
- The primary and secondary intervals form a partition on the set of keys.
- Each client tree recovers the whole partition. This means that from each client tree, we can have all the records of the file.
- From server 0, we can built the real tree.
- Let us consider S_1,S_2, …, S_n the servers visited while transforming the pair (Client, Key) into a server. The updating algorithm of the client tree is then applied (n-1) times. The two properties hold :

1) : For all $i > 1$: Inf (Primary_Interval(S_i)) $\geq$ Sup(Primary_Interval(S_{i-1})
2) : From a server, we can retrieve all the servers which follow it in the partition.

V. Illustration of the method

Figure 3 gives the final states of the clients and servers after the insertion of 25 keys (strings) by the corresponding clients (4 clients):

We suppose that the capacity of a bucket (server) is equal to 4.

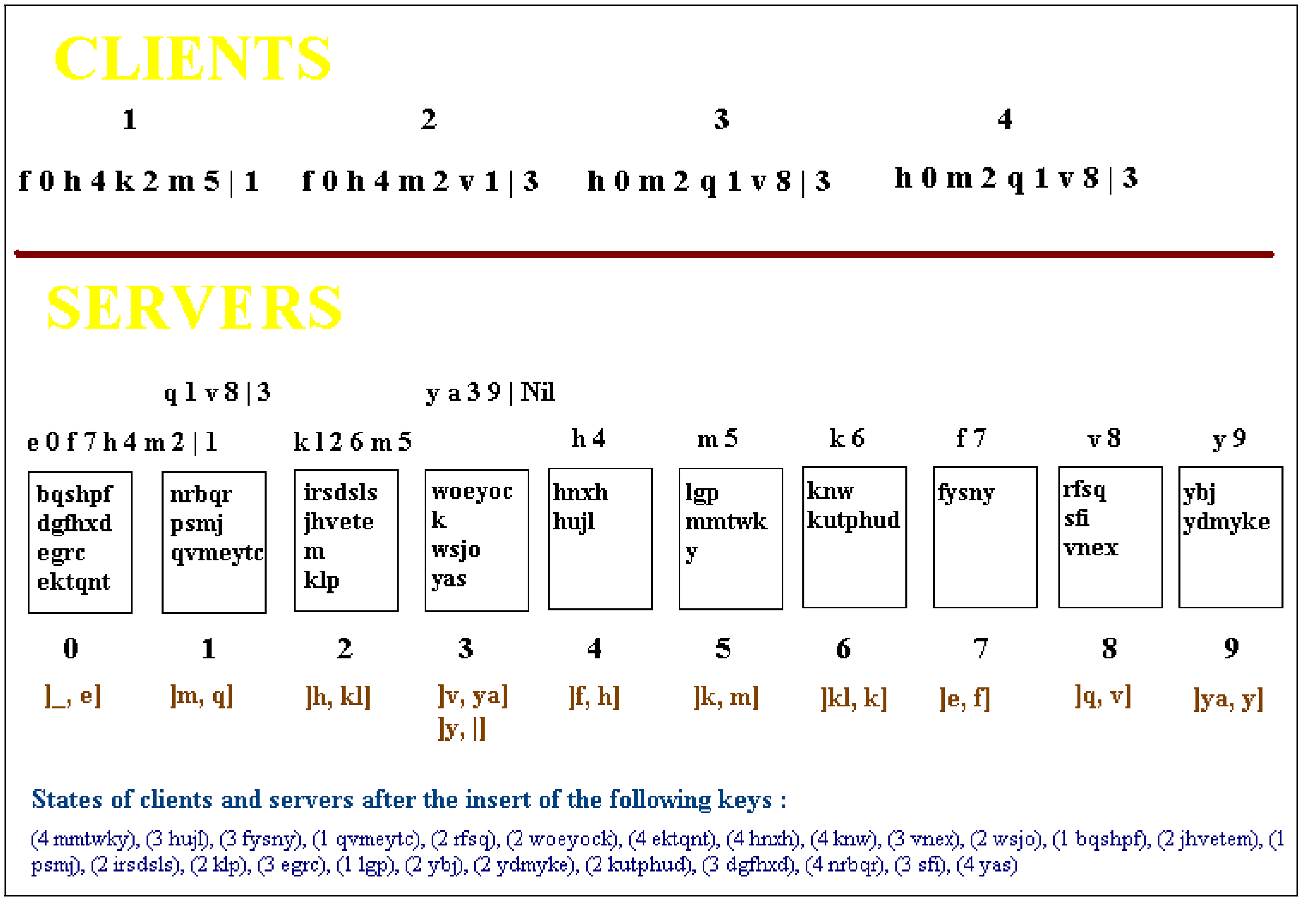

CLIENTS
1
2
3
4
f 0 h 4 k 2 m 5 | 1
f 0 h 4 m 2 v 1 | 3
h 0 m 2 q 1 v 8 | 3
h 0 m 2 q 1 v 8 | 3
SERVERS
q 1 v 8 | 3
y a 3 9 | Nil
e 0 f 7 h 4 m 2 | 1
k l 2 6 m 5
h 4
m 5
k 6
f 7
v 8
y 9
bqshpf dgfhxd egrc ektqnt
nrbqr psmj qvmeytc
irsdsls jhvetem klp
woeyock wsjo yas
hnxh hujl
lgp mmtwky
knw kutphud
fysny
rfsq sfi vnex
ybj ydmyke
0
1
2
3
4
5
6
7
8
9
]_, e]
]m, q]
]h, kl]
]v, ya]]y, |]
]f, h]
]k, m]
]kl, k]
]e, f]
]q, v]
]ya, y]
States of clients and servers after the insert of the following keys :
(4 mmtwky), (3 hujl), (3 fysny), (1 qvmeytc), (2 rfsq), (2 woeyock), (4 ektqnt), (4 hnxh), (4 knw), (3 vnex), (2 wsjo), (1 bqshpf), (2 jhvetem), (1 psmj), (2 irsdsls), (2 klp), (3 egrc), (1 lgp), (2 ybj), (2 ydmyke), (2 kutphud), (3 dgfhxd), (4 nrbqr), (3 sfi), (4 yas)

Fig3. Clients and servers

Figure 3. shows 4 clients and 10 servers. Each client has its own access function represented by a digital tree. The file is distributed on 10 servers at a rate of one bucket per server. Each server has a primary interval indicating the set of the possible keys on this server, a digital tree keeping the trace of the splitting on this server and a bucket containing the keys inserted. A server can contain one or several secondary intervals. The client tree represents its image of the file. Thus, for client 1, with digital trie **f 0 h 4 k 2 m 5 | 1,** the set of the keys is divided as follows: [λ, f] : 0 ;]f, h] : 4 ; [h, k] : 2 ;]k, m] :5,]m, Λ] :1. This client thus 'sees' only servers 0, 4, 2, 5 and 1. For client 2, with the tree **f 0 h 4 m 2 v 1 | 3,** 'sees' servers 0, 4, 2, 1 and 3. None of the two partitions is the real partition of the file. There are servers which are split (0, 1, 2 and 3) and others not yet. The servers 4, 5, 6, 7, 8 and 9 hold empty trees. Notice that server 3 has a secondary interval.

If Client 1 searches for key 'egrc', it finds server 1. Although the tree of client 1 does not reflect the real image of the file, client 2 has found the correct server since key ' egrc' is in the primary interval of this latter.

If client 1 searches for key 'wsjo', it finds first server 1. As the primary interval of server 1 does not cover this key, the client tree is updated according to the tree of server 1 and becomes **f 0 h 4 k 2 m 5 q 1 v 8 | 3**. The re application of CTH in this new tree gives now server 3 which is the correct.

Let us suppose now that client 1 wants to insert the key ' are'. The application of its tree (hashing function) to this key gives us server 1. As the bucket of server 1 is full, a collision occurs on this server. The collision is solved as follows: we consider the ordered sequence formed of the keys of server 1 and of the new key, that is to say 'are', ' bqsph', 'gfhxd', ' egrc', 'ektqnt'. Then, we determine the smallest sequence of digits which makes it possible to distinguish the key from the medium ('dgfhxd') and the last key ('ektqnt'). It is thus 'd' this sequence. The tree of server 1 which is **e 0 f 7 h 4 m 2 | 1** becomes **d 0 e 10 f 7 h 4 m 2 | 1** . Server 10 is created with an empty tree (**e 10**) and with the primary interval]d, e]. The keys are divided between servers 0 and 10 so that the keys strictly higher than 'd' migrate in the new server. Thus after splitting, the keys 'are', ' bqsph' and ' dgfhxd' remain in server 1 whereas the keys ' egrc' and 'ektqnt' go towards server 10.

VI Range query operation

The range query operation consists in listing all the keys in a given interval. This can be achieved by two manners according to the multicast can be used or not. If the multicast is used, the principle is simple. It suffices to send the request to all the servers. The servers concerned by the request respond to the client having formulated the request. If the multicast cannot be used, the following algorithm is proposed.
It allows to make a parallel process. In fact, the client determines first from its tree all the servers which could contain the keys requested. Then, the servers simultaneously work to perform the operation.

We suppose a client Client wants to determine all the keys in the interval [X, Y].

We begin by defining the module Scope(S) which recuperates all the keys visible from S by traversing all the descendant servers of S. This module is recursive and can be described as follows:

The module Scope (Server)

```
- Go to the server
- List all the keys  $\leq$ Y
- If it exists a key  $\geq$ Y then Stop

- Proceed by a sequential traversal of the tree of server Server  using a local stack.
   - When an internal node (digit) is met, it is stacked.
   - When an external node ( a server or Nil ) is met, the stack contains the maximal key associated to this external node ( concatenation of all the digits). If it is a server S, we make the recursive call Scope(S). Then make a pop operation on the stack.
```

Range query algorithm

We use the modules Scope(S) defined above and Treat_Server (S, Cmax) defined as follows :
Input :
- S : a server.
- Cmax : its maximal key according to the client tree having made the query.

```
    - Go to the server  S
    - List all the keys  $\leq$ Y
    - If it exists a key  $\geq$ Y then Stop
```

- *Traverse the tree of server **S**. For every server **S'** different of **S** such as its maximal key is less or equal than **Cmax** do Scope(**S'**)*

Finally, the range query operation is defined as follows :

*1- Apply the transformation algorithm on the client **Client** to search key **X**. Let it **M** be the server found and **Cm** its maximal key.*
*2- Initialize a stack with **Cm.***
*3- Traverse the client tree from node **M** as follows:*
When an internal node (digit) is met, it is stacked.
*When an external node is met, let it **S** be this external node and then **Cm** its maximal key.*
*(i) if **Cm** ≤ Y then if **S** > 0, perform Treat_Server(**S, Cm**). Pop a digit from the stack and advance in the client tree(we make this in the cases : **S** > 0 and **S** < 0)*
*(ii) if **Cm > Y** then stop*

In Figure 4, the client selects three servers S1, S2 and S3 with maximal keys Cmax1, Cmax2 and Cmax3 respectively. It sends then three messages with these maximal keys towards the corresponding servers. Every server performs simultaneously the following operations :
- sends to the client all the records in its bucket with keys in range [X, Y]
- traverses its tree and performs Scope on all the servers with maximal keys less or equal than Y. The module Scope, applied on a given server, goes to all its descendants which send then all the records in their buckets with keys in range [X, Y]. Server trees are represented by segments. the solid parts represent all the paths such as Digit(path) less than Y.

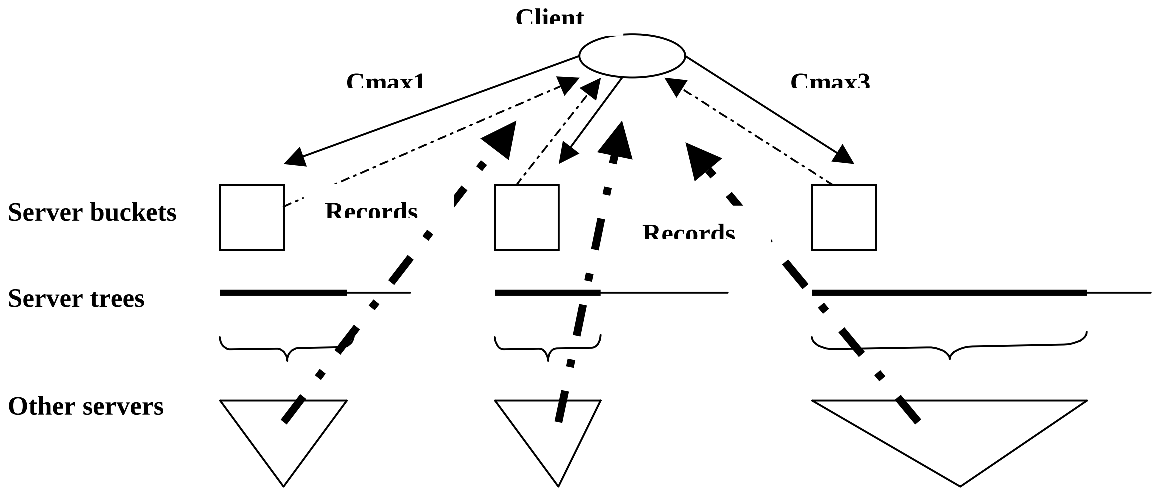


Fig.4. Example showing the rang query process

VII Simulation model and test

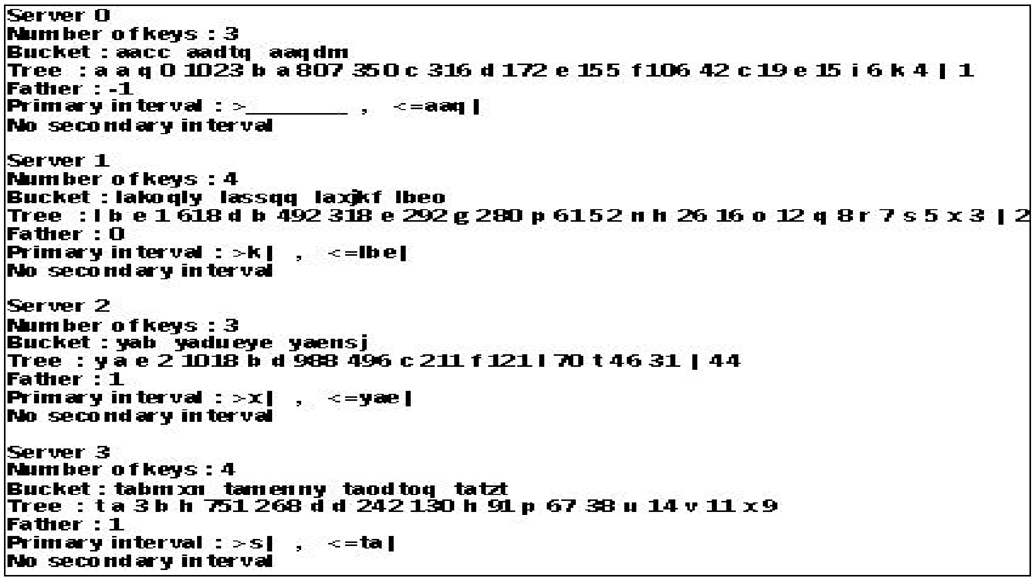
We worked on only one machine and we used Delphi as programming language. Our main objective is to test the validity of the method. Each client is materialized by a table which contains its tree. The servers are compared to the blocks of a file which evolves in a linear way dynamically according to the collisions. A server contains 4 fields: bucket for the keys, local tree represented by a table, a primary interval and a list of secondary intervals. We considered a very small capacity of the buckets (b=4). That allows to have a great number of servers in order to test the validity of the suggested

schema and analyze some parameters. We briefly give below some implementation considerations.
Clients **:** We considered 4 clients which can address around 1000 servers.
Servers **:** Each server is a simulated by a file block. The bucket capacity of a server is b = 4 keys. Expression 'Go to server' is traduced by a read operation.
Operations : Construction of a data file with pairs (Client, Key). Clients are generated randomly in set { 1, 2, 3, 4 }. The articles are reduced to their keys. Keys are strings of characters which are generated also randomly. A run consists to insert n pairs (client, key) from a data file. We can see the states of clients and servers at the end of a run.
Verification **:** We wrote a program that allows to verify the validity of the proposed schema. A first run allows to search all the keys inserted by each client. The client trees are then updated. In a second run we notice no modification on the client tree. We wrote also a program which verifies the validity of the tree after any replacement operation on the client tree. Another program verifies that the partition is maintained after each split of a server.
Trace **:** We can follow the complete trace of the program, What allows us to see the mechanism of the construction of the trees and the distribution of the file on the servers.

We give below a screen dump, some results and a curve showing the convergence of the client trees.

Screen Dump

Figure 5. shows the states of some servers after the insertion of 3000 keys with a server bucket capacity equal to 4. 1064 servers are generated with a load factor of 70,49%.

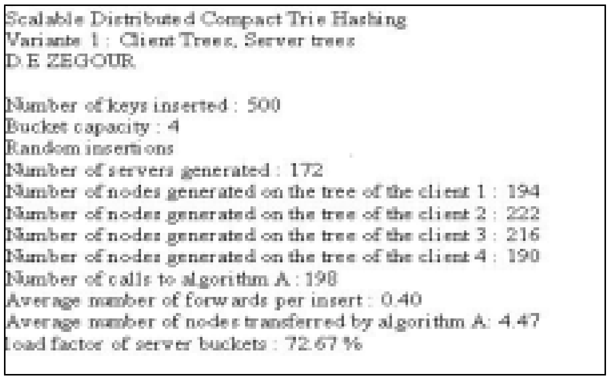


```
Server 0
Number of keys : 3
Bucket : aacc  aadtq  aaqdm
Tree  : a a q 0 1023 b a 807 350 c 316 d 172 e 155 f 106 42 c 19 e 15 i 6 k 4 | 1
Father : -1
Primary interval : >________ ,  <=aaq|
No secondary interval

Server 1
Number of keys : 4
Bucket : lakoqly  lassqq  laxjkf  lbeo
Tree  : l b e 1 618 d b 492 318 e 292 g 280 p 6152 n h 26 16 o 12 q 8 r 7 s 5 x 3 | 2
Father : 0
Primary interval : >k|  ,  <=lbe|
No secondary interval

Server 2
Number of keys : 3
Bucket : yab  yadueye  yaensj
Tree  : y a e 2 1018 b d 988 496 c 211 f 121 l 70 t 46 31 | 44
Father : 1
Primary interval : >x|  ,  <=yae|
No secondary interval

Server 3
Number of keys : 4
Bucket : tabmxn  tamenny  taodtoq  tatzt
Tree  : t a 3 b h 751 268 d d 242 130 h 91 p 67 38 u 14 v 11 x 9
Father : 1
Primary interval : >s|  ,  <=ta|
No secondary interval
```

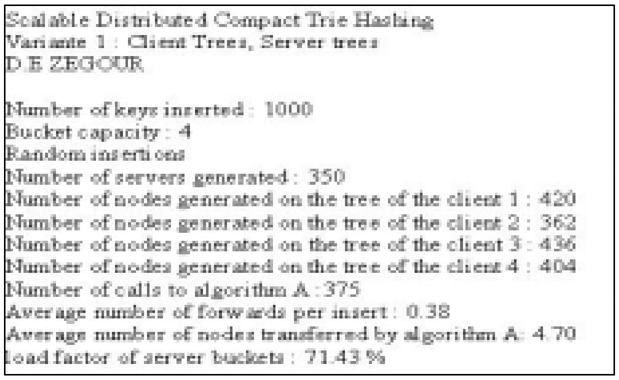
Fig.5 : Example of servers

Some results

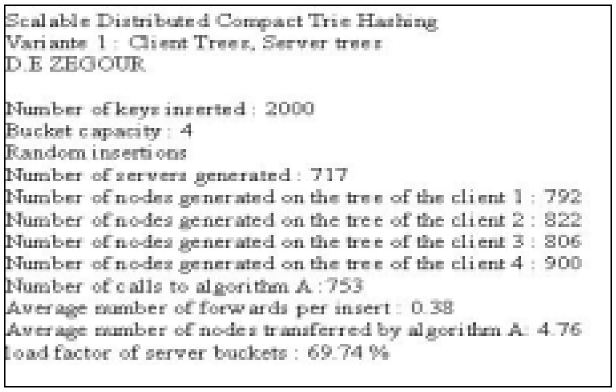
Figure 6 shows four tests with 500, 1000, 2000 and 3000 records inserted randomly. Algorithm A designates the updating algorithm described before.



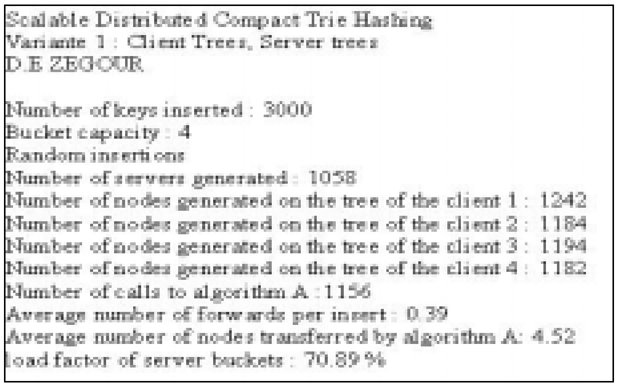
```
Scalable Distributed Compact Trie Hashing
Variante 1 : Client Trees, Server trees
D.E ZEGOUR

Number of keys inserted : 500
Bucket capacity : 4
Random insertions
Number of servers generated : 172
Number of nodes generated on the tree of the client 1 : 194
Number of nodes generated on the tree of the client 2 : 222
Number of nodes generated on the tree of the client 3 : 216
Number of nodes generated on the tree of the client 4 : 190
Number of calls to algorithm A : 198
Average number of forwards per insert : 0.40
Average number of nodes transferred by algorithm A: 4.47
load factor of server buckets : 72.67 %
```



```
Scalable Distributed Compact Trie Hashing
Variante 1 : Client Trees, Server trees
D.E ZEGOUR

Number of keys inserted : 1000
Bucket capacity : 4
Random insertions
Number of servers generated : 350
Number of nodes generated on the tree of the client 1 : 420
Number of nodes generated on the tree of the client 2 : 362
Number of nodes generated on the tree of the client 3 : 436
Number of nodes generated on the tree of the client 4 : 404
Number of calls to algorithm A : 375
Average number of forwards per insert : 0.38
Average number of nodes transferred by algorithm A: 4.70
load factor of server buckets : 71.43 %
```



```
Scalable Distributed Compact Trie Hashing
Variante 1 : Client Trees, Server trees
D.E ZEGOUR

Number of keys inserted : 2000
Bucket capacity : 4
Random insertions
Number of servers generated : 717
Number of nodes generated on the tree of the client 1 : 792
Number of nodes generated on the tree of the client 2 : 822
Number of nodes generated on the tree of the client 3 : 806
Number of nodes generated on the tree of the client 4 : 900
Number of calls to algorithm A : 753
Average number of forwards per insert : 0.38
Average number of nodes transferred by algorithm A: 4.76
load factor of server buckets : 69.74 %
```



```
Scalable Distributed Compact Trie Hashing
Variante 1 : Client Trees, Server trees
D.E ZEGOUR

Number of keys inserted : 3000
Bucket capacity : 4
Random insertions
Number of servers generated : 1058
Number of nodes generated on the tree of the client 1 : 1242
Number of nodes generated on the tree of the client 2 : 1184
Number of nodes generated on the tree of the client 3 : 1194
Number of nodes generated on the tree of the client 4 : 1182
Number of calls to algorithm A : 1156
Average number of forwards per insert : 0.39
Average number of nodes transferred by algorithm A: 4.52
load factor of server buckets : 70.89 %
```

Fig.6 : Some simulation results

The load factor is the same that the other file structures (B-trees, RP, …) for random insertions, i.e. 70%. The number of messages to search or insert a record is close to a few messages. Only some nodes are transferred to update the trees of the clients when addressing errors occur.

We observe a load factor close to 70%. The number of nodes transferred is about 4.5. The number of forwards is about 0.40. These results hold for any number of keys inserted.

With b=50, we have the following results :

Number of keys inserted	10 000	20 000	30 000
Load factor	67,11	69,56	72, 13
Number of forwards	0,10	0,09	0,10
Number of nodes transferred	2,40	2,54	2,47

Of course, these results are affirmative for the meantime. They will be validated or adjusted in a real environment of test. Therefore, it would be necessary to develop a communication protocol for the proposed method in a network of computers.

Convergence of client trees

We suppose a new client inserts 1000 keys already inserted. No record is inserted but the client tree is updated. Figure 7 shows the speed with which the client tree is updated during the insertions of records. The tree evolves very quickly for the first insertions, then evolves more and more lesser when we progressively insert records. The tree becomes steady as soon as a certain number of records is reached.

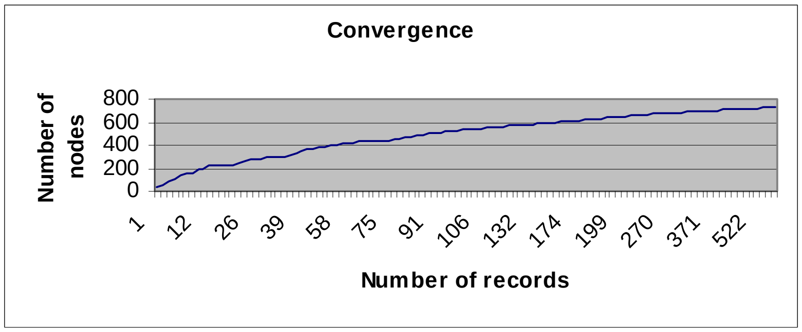


Fig.7 Convergence of the client trees

VIII. Variants of CTH

CTH* without server trees and without multicast (. CTH*CT)

At the level of each client there is a partial digital tree from which any operation starts on the file.
At the level of each server there are a bucket containing the records of the file and an interval]Min, Max].
There is a central server which contain the real tree.
During the search phase, the client tree is updated gradually until obtaining the real tree. The expansion of the file is done through collisions. At each collision there is distribution of the file on a new logical server. The digital tree of the central server is extended to each division of a server. It thus keeps the real tree.
Compared with the method described previously, in this schema there is no tree at the level of the servers. A particular server contains the real tree. With this manner, we avoid multicasting. The algorithm of transformation (Client, Key) to server is depicted in figure 8.

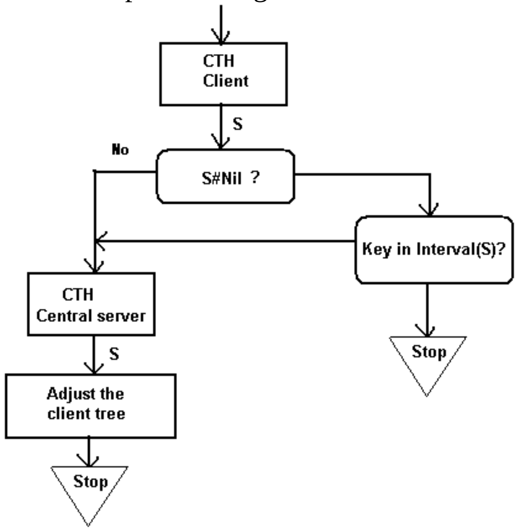


Fig8. The transformation algorithm in CTH*CT

CTH* without server trees (CTH*NST)

It is the same schema as the one proposed, except that there is no tree at the level of the servers. If the key is not in the interval of the server, we use multicasting in order to find the correct server. The image of the client is then updated with the B algorithm.. The algorithm of transformation (Client, Key) to server is depicted in figure 9.:

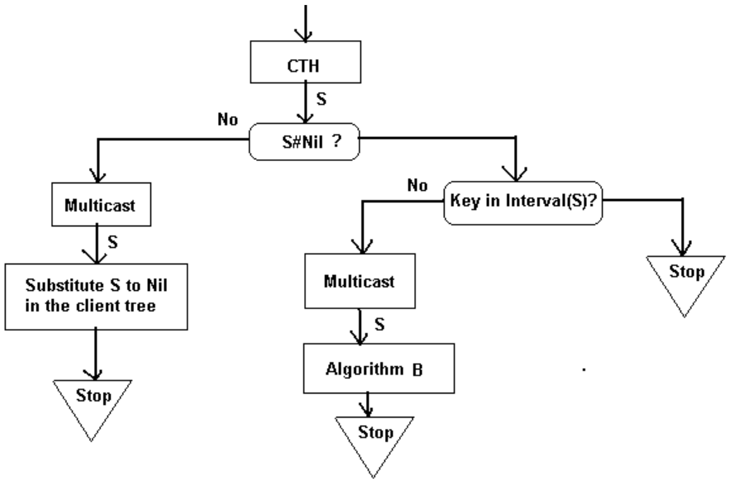


Fig9. The transformation algorithm in CTH*NST

IX. Comparison with RP*S

Among all the known SDDS, we were interested to the method RP*s. The two schemas are deterministic and preserve the order of records. By deterministic, we mean the possibility to perform the operations without the use of multicast. Rp*S is comparable to the variant of CTH* with the tree centralized. If we cut the tree on several levels as this is made in the method 'multilevel trie hashing' [Lit 87], we obtain an organization similar to Rp*S. On the other hand, the basic version of CTH* described in this paper is superior to RP*S at several levels of comparison.

Client : RP*s uses a table (Key, Server) for the access to the servers. CTH* uses a compact digital tree. The two perform a sequential traversal but the advantage of CTH* resides in the fact that it doesn't stock the keys integrally. Three bytes are sufficient on average to address a server. It results an important gain of space compared to RP*s.
Servers : RP*s uses two types of server: data server and index server. CTH* uses only one type of server which contains a part of the index and a part of data. This can avoid messages to access to data servers.
Transformation (Client, Key) to server : RP*s traverses a table, CTH* a tree. When an addressing error occurs, RP*s moves up the B^+-tree and moves upward again until the leaf in order to find the correct server. Once the correct server is found, there is an adjustment of the client table. When an addressing error occurs, CTH* goes only to the downward servers of the server on which the error is appeared. Therefore for CTH*, we don't have need to climb up the tree to recover the correct server. From the erroneous server, we can recover the correct server, while exploring only its descendants. CTH* adjusts the client tree by transferring only a few nodes from the server toward the client. For RP*s, there is adjustment only in some cases. In CTH*, there is always adjustment of the client tree.
Convergence : More faster on CTH* that on RP*S because in CTH* there is transfer of a part of the tree that represents several couples (key, address).
Insert operation : For RP*s, splitting operations imply more of work. Splitting can be in cascade (principle

of B-trees) and therefore can cause the modification of several servers. For CTH*, only one server is modified. The splitting algorithm is more simpler than the one of RP*s (B+ tree). It results an important gain in terms of the messages.
Searching the next : RP*s can require to climb up and move downward in the B-tree, what risks to imply several messages. For CTH*, in the worst case, the next is recovered in only one message.
Query operation : For CTH*, from the client tree, we can do any query operation. From the client tree we can determine the potential servers. We send a parallel request to all these servers. Then, the search continues only in the descendants of these servers. This should be better than on RP*s.

X. Conclusion

The first SDDSs were primarily based on dynamic hashing with the works of W. Litwin which proposed a generalization of his well-known popular method (LH) [Lit81] to a distributed environment, more precisely for the multicomputers. Thereafter, several variants of LH* have been proposed, each one allows to improve a certain aspect of the method. Although these SDDSs were shown very efficient, they present a major inconvenient which is the no preservation of the order of records. This can degrade the operations relative to the order of records. We proposed in this paper a generalization of another dynamic hashing method (TH)as well known as LH to a distributed environment.

Contrary to the majority of the existing methods, the proposed method provides the order of distributed files and then facilitates the range queries operations and the ordered traversal of files. Moreover, the following properties makes of our method a promising opening towards a new class of SDDS : - preservation of the order of records - works without multicast -three bytes are sufficient to address a server, - the transfer of some bytes is enough for the update to the trees at the level of the clients.

We focused our work mainly on the mechanism of construction and the range query operation. It would be of course more complete to study the suppression operation. Further work should concern the implementation of the communication protocol in order to test the method in a real network and proceed to deeper analysis of performance.

A basic schema and two variants have been proposed. According to the application, one of the proposed schemas can be used : CTH* works without multicast. CTH*CT uses the real image of the tree indexing all the servers and thus works also without multicast. CTH*NST works with an intense use of multicast but does not use trees on the servers.

Further research should concern also many other variants as it was made in LH*. As example, It will be interesting to study the security problems.

Now, the method is in its implementation phase under the Unix platform. It constitutes the storage layer of the project ' ACT 21' [Zeg01] which consists to the realization of a parallel databases management system based on the concept of actors.

Acknowledgements

We would like to express special thanks M.S Birech, Y. Laalaoui, A. Bekkouche and A. Khaled (students), who contributed by their reflection to the elaboration of some algorithms. A major expression of thanks must go to my dear colleagues W. Litwin and G. Lévy (Paris Dauphine university) for their advice and helpful comments and suggestions.

References

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

[Dev 93] Devine, R. Design and Implementation of DDH: Distributed Dynamic Hashing. Int. Conf. on Foundations of Data Organizations, FODO93. 12 VLDB1994, Chile
Lecture Notes in Comp. Sc., SpringerVerlag (publ.), Oct. 1993.

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

[Kro 94] Kroll, B., Widmayer, P. Distributing a Search Tree Among a Growing Number of Processors. To app. at ACMSIGMOD Int. Conf. On Management of Data, 1994.

[LIT80] Litwin, W. Linear hashing : A new tool for files and tables addressing. VLDB 80, ACM, (Sep 1980),

[Lit 81] Litwin, W. Trie hashing. SIGMOD 81. ACM, (May 1981), 19-29.

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

[LIT86] Litwin, W., Lomet, D. Bounded Disorder Access Method. 2-nd Int. Conf. on Data Eng. IEEE, Los Angeles, (Feb. 1986).
[LIT87] : Litwin W, Zegour D.E. and Levy G. Multilevel Trie hashing. Int. Conf. VLDB, Venise, Italy. 1987.

[Lit 93] Litwin, W. Neimat, MA., Schneider, D. LH* : Linear Hashing for Distributed Files. ACMSIGMOD Int. Conf. On Management of Data, 1993.

[Lit 93a] Litwin, W., Neimat, MA., Schneider, D. LH*: A Scalable Distributed Data Structure. (Nov. 1993). Submitted for journal publ.

[Lit 93b] Litwin, W., Neimat, MA., Schneider, D. RP* : a Scalable Distributed Data Structure using Multicast (extended abstract) HPLDTD93009, (Sept. 1993).

[Lit 94] Litwin, W., Neimat, MA., Schneider, D. RP* : A Family of Order Preserving Scalable Distributed Data Structures. Proc. Of 2O th conf. VLDB, chile 1994

[Lit 00] W. Litwin, T.J.E Schwarz. LH*RS : A High-Availability Scalable Distributed Data Structures using Reed Solomon Codes. ACM-Sigmod 2000, Dallas

[Lom 83] Lomet, D. Bounded Index Exponential Hashing. ACM TODS, 8, 1. (Mar 1983), 136-165.

[Mat 90] Matsliach, G., Shmueli, O. Distributing a B+tree in a loosely coupled environment. Inf. Proc. Letters, 34, 1990, 313321.

[Mat 91] Matsliach, G., Shmueli, O. An Efficient Method for Distributing Search Structures. IEEEPDIS Conf., 1991.

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

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

[Per 89] Perrizo, W., Lin, J., Hoffman, W. Algorithms for Distributed Query Processing in Broadcast Local Area Networks. IEEE TKDE, 1, 2, 1989, 215225.

[TOR83] Torenvliet, L., Van Emde Boas, P. The Reconstruction and Optimization of Trie Hashing Functions. VLDB 83, (Nov. 1983), 142-157.

[Zeg 88] D.Zegour «Extensions du hachage digital : hachage multiniveaux hachage digital avec représentations séquentielles » thèse de doctorat, université de Paris IX Dauphine 1988.

[Zeg 94] D.E Zegour, W. Litwin. Trie hashing with the sequential representations of the trie. Revue internationale des technologies avancées. CDTA, Alger.

[Zeg01] D.E Zegour. Présentation générale du projet 'ACT'. Rapport interne. Institut National d'Informatique, Alger, 2001.