Compact balanced tries

Pierre Nicodeme

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1533

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

COMPACT BALANCED TRIES

Pierre NICODÈME

Octobre 1991

RR-1533

Compact Balanced Tries

Pierre Nicodème

September 30, 1991

Abstract.

This document proposes a compact structure for ordered indexes. Applied to typical English language data, this structure needs only two bytes per key. Compression is provided by a compact bit-map representation of tries or of subtries. Flexibility is provided by balancing methods between subtries. The global structure aggregating subtries in block-nodes is $B-$tree like. As for a $B-$tree, shrinking the index may be done for any key distribution, leading to an almost 100% storage utilization. All the properties of $B-$trees are kept, except the possibility of local binary searching inside the block-nodes. The present algorithm ensures a 50% minimum storage utilization for any block-node (when shrinking of the index is not performed); it is fit for multi-user usage and is a candidate as a basic tool for memory database ordered indexes.

Tries equilibrés et compacts

Résumé.

Ce document propose une structure compacte pour les index ordonnés. Appliquée à des données typiques du langage anglais, cette structure n'utilise que 2 octets par clé. La compression est obtenue au moyen d' une représentation compacte par bit-map de tries ou de sous-tries. La structure globale regroupant des sous-tries en noeuds de type blocs est similaire à un arbre$-B$. De même que pour les arbres$-B$, on peut procéder à une réduction de l'index pour n'importe quelle distribution de clé, conduisant à une utilisation mémoire avoisinant les 100%. Toutes les propriétés des arbres$-B$ sont conservées, à l'exception de la possibilité de faire une recherche binaire à l'intérieur d'un noeud bloc. L'algorithme proposé assure une utilisation minimale de 50% pour tout noeud bloc (quand la réduction d'index n'est pas utilisée), convient pour le multi-transactionnel, et est candidat comme outil de base pour les index ordonnés des bases de données mémoire.

Compact Balanced Tries

Pierre Nicodème

nicodeme@margaux.inria.fr

Copernique–Cap Gemini
8, Mail de l'Europe
78170 La Celle Saint Cloud
France

INRIA–Rocquencourt
Domaine de Voluceau
78153 Le Chesnay Cedex
France

Abstract

This paper proposes a compact structure for ordered indexes. Applied to typical English language data, our structure needs only two bytes per key. Compression is provided by a compact bit-map representation of tries or of subtries. Flexibility is provided by balancing methods between subtries. The global structure aggregating subtries in block-nodes is $B-$tree like. As in a $B-$tree, shrinking the index may be done for any key distribution, leading to an almost 100% storage utilization. All the properties of $B-$trees are kept, except the possibility of local binary searching inside the block-nodes. The present algorithm is better than all ordered index algorithm known at the moment, when considering both its compactness and its flexibility; it ensures a 50% minimum storage utilization for any block-node (when shrinking of the index is not performed), fits for multi-user usage and is a candidate as a basic tool for memory database ordered indexes.

1 Introduction

Industrial or commercial applications, such as keeping track of the customers references for a national electric distribution or managing a phone book for a whole country, require huge indexes. When handling the customer or supplier files of a company, it is natural to provide ordered indexes of these files; when looking for a name in a phone book, a human being will first look for a name with similar spelling, get to a page of the dictionary, and proceed further scanning forward or backward the phone book until the desired name is reached. Therefore both capabilities of fast direct addressing (first part of the research) and sequential processing (second part of the research) are necessary to perform such a natural research. The classical *$B-$tree* [1] exhibits these both capabilities; it offers good storage utilization (at least 50%) and its balanced structure provides a guarantee of uniform performance. Moreover, when a $B-$tree is composed of partially filled nodes, it is possible to scan the leaves, forcing some of them to a 100% storage utilization, and deleting the leaves which become empty; this maximum shrinking method leads to $B-$trees approaching a 100% storage utilization. Finally the segmented structure of the $B-$tree makes it suitable for multi-user environment. The only drawback of the B-tree is its relative lack of compactness. As later version of the $B-$tree, the *prefix $B-$tree* [2] provides a 25% improvement of disk access when compared to an ordinary B-tree, which corresponds to a substantial (about 50%) saving of storage utilization.

Other methods have been proposed in the last decade; they do not offer all the B-tree properties.

The *bounding disorder method* [12] uses a mixing of hashing and tree indexing; the upper part of the index is tree ordered, while the lower part is hashed in multi-bucket nodes; this method provides good results for equi-join queries and for large range queries; however sequential processing may result in weak performances and delicate overflow handling is necessary. Average storage utilization is about 80%.

In *Trie Hashing* [11], an in-core trie-directory pilots the hashing scheme and allows sequential processing; the storage utilization is about 80%. However, the directory size is $O\left(n^{1+1/s}\right)$ [5] (n being the number of records inserted and s the bucket size), which is superlinear in n, and the directory is not segmentable; this implies low parallelism in case of directory updating, and limitation of the data size in relationship with the memory size. Moreover, unfavourable key distributions result in a large increase of the directory size.

Our work come as a descendant of the classical B-tree and of the *Compact Tries* [9], [4]. We present in section 2 the *Compact 0-complete Tree* [13], also a B-tree descendant, which keeps almost all the property of B-trees and produces very compact storage (4 bytes per key structure with 60% storage utilization).

We detail in section 3 the Compact Tries which provide a very compact indexing method (4-5 bytes per key structure) based on a bit-map representation of tries, but does not have a B-tree structure and is not segmented.

Our research contribution is given in section 4. We show how it is possible to split the Compact Tries in a segmented structure of B-tree type, keeping the compactness advantages of the Compact Tries and retrieving all the important properties of B-trees. Maximum shrinking index methods may be applied to these *Compact Balanced Tries*, allowing 100% storage utilization; in contrast, the minimal splitting rule of the Compact 0-complete Trees forbids such methods and does not guarantee the storage utilization in unfavourable cases of key distribution.

Interesting experimental results are given in section 5.

The Compact-Balanced Tries may be viewed as a powerful way to compress B-tree nodes, while retaining the 50% minimum storage and the capacity of forcing a maximum shrinking of the index size.

On the other hand, the limitations of the Compact Tries are directly linked to the 1-piece shape of the structure: heavy sequential processing is needed and parallel processing is forbidden; the scope of the Balanced Compact Tries is to override these limitations.

The basic features of the Compact-Balanced Trie are the ability of generating an edge-key (or starting-key) for the new block-node after a block splitting, of updating it when keys are exchanged between brother block-nodes, and the capacity of handling this edge-key in the Compact Trie algorithms.

One of the B-tree properties is not kept by the Compact Balanced Tries: binary search is not possible inside the block-nodes and only sequential processing is permitted at this level; however this drawback is not so important when considering that sequential-processing is only local and performed as in-core processing.

2 Compact 0-complete Trees (C_0–Trees)

The Compact 0-complete Trees, outside of the worst case splitting problem discussed in this chapter, have the best characteristics of an ordered index, and their architecture is similar to the architecture of a classical B–tree.

For random keys, they can represent 32 bytes long keys with only one byte; this result will be the point of comparison for the Compact-Balanced Tries we present in section 4.

2.1 Brief survey of the C_0–Trees

2.1.1 0-complete trie

A 0-complete trie is a trie where each node has at-least a 0-node son; (a 0-node is a node accessed by a 0 bit in the trie).

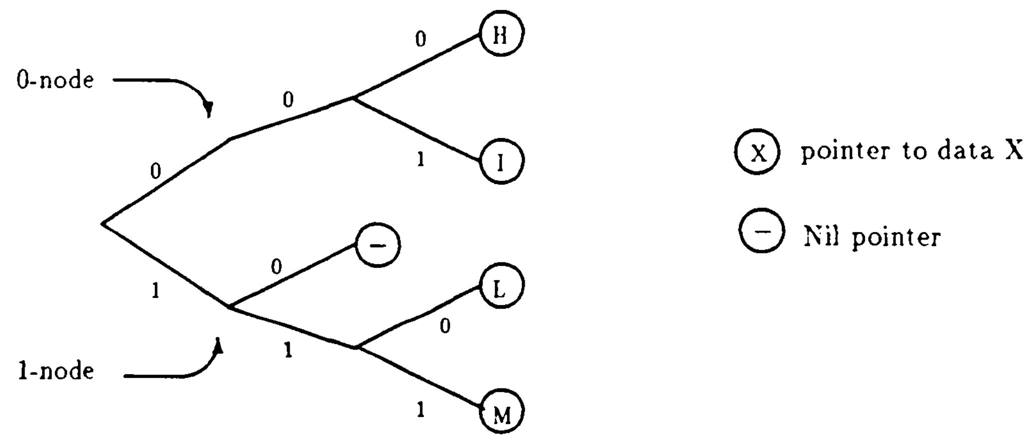


Figure 1: 0-complete trie

2.1.2 Bounding nodes

The following property stands for a 0-complete trie: each leaf node has a 1-node as a successor in the preorder traversal of the trie, which is called its bounding node; bounding nodes are represented with doubles circles in Figure 2.

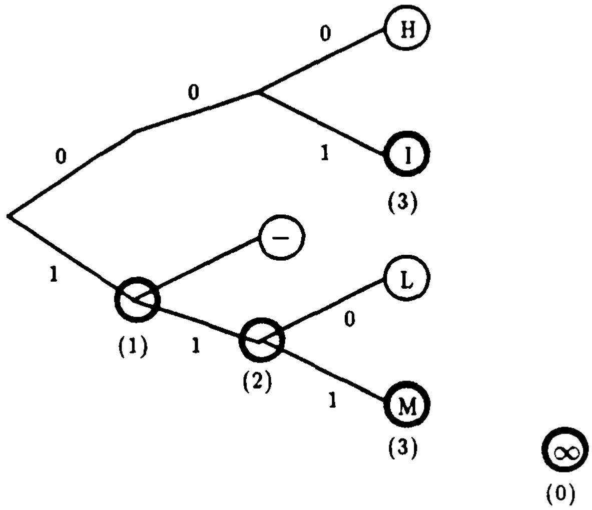


Figure 2: Bounding nodes

The depths of the bounding nodes are in parentheses in Figure 2; a special bounding node with depth 0 is added to bound the last leaf. With these preliminaries, Orlandic and Pfaltz produce an algorithm which provides the key of a leaf as a function of the depth of its bounding node and of the depths of all the preceding bounding nodes. This algorithm may be summarized as follows: scanning of the depths of the bounding nodes is sequentially processed; the starting key is null; set to 1 the bit with position equal to the current depth; set to 0 all the bit to the right of this bit.

Hence, we get the following table, corresponding to the trie of Figure 2:

depth	depths sequence	key	leaf
3	3	001	H
1	3, 1	100	I
2	3, 1, 2	110	–
3	3, 1, 2, 3	111	L
0	3, 1, 2, 3, 0	∞	M

Figure 3: Key construction

2.1.3 C_0–node representation and C_0–node splitting

A C_0–node is a list of entries, each entry consisting of a bounding node depth and a data (or NIL) pointer; a node is a simple trie-node.

Node	
depth	data pointer
3	H
1	I
2	–
3	L
0	M

Figure 4: C_0–compact node

The splitting rule for an overflowing C_0–node is as follows: any entry may be chosen for the splitting, with the condition than no preceding entry in the C_0–node has a depth inferior to the depth of this entry.

In case of Figure 4, the two possible splitting entries are entry 1 (depth 3), and entry 2 (depth 1). This rule, called minimal depth splitting rule, allows the handling of the upper parts of the C_0–tree with the same algorithm as used for the leaves, but it does not insure, in worst cases, a good filling rate of the C_0–nodes; the worst case is a kind of 1-comb which degenerates in a succession of 1-entry C_0–nodes (Figure 5).

Moreover, the minimal depth splitting rule does not allow the transfer of entries at will between adjacent brother nodes, when this B–trees capability allows by a progressive scanning of the leaves to reach any desired filling rate.

For these reasons, we consider that the C_0–compact trees do not offer the complete security required for standard indexes of B–tree type.

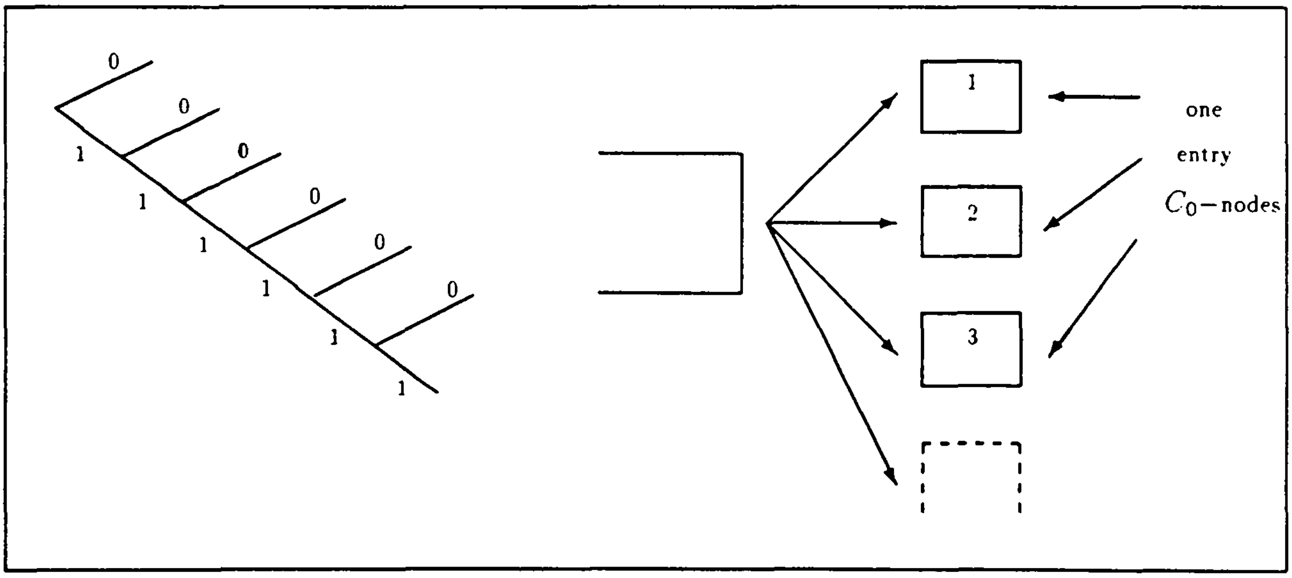


Figure 5: worst splitting case for the C_0–Trees

3 Compact Trie

Jean Kouacou-Kouadio proposes a compact trie representation by bit-map, based on a scan along the nodes of the trie in respect with the preorder traversal [9].

W. de Jonge, A. S. Tanenbaum and R. P. van de Riet [4] propose a very similar representation; they call it linear representation. The Compact Balanced Trie we present in the next section is an extension of the Compact Trie, in terms of its ability to split the trie into subtries and to represent each of these in a compact way.

We present in this section the Compact Tries, and a data structure to handle them.

3.1 Compact-Trie Representation

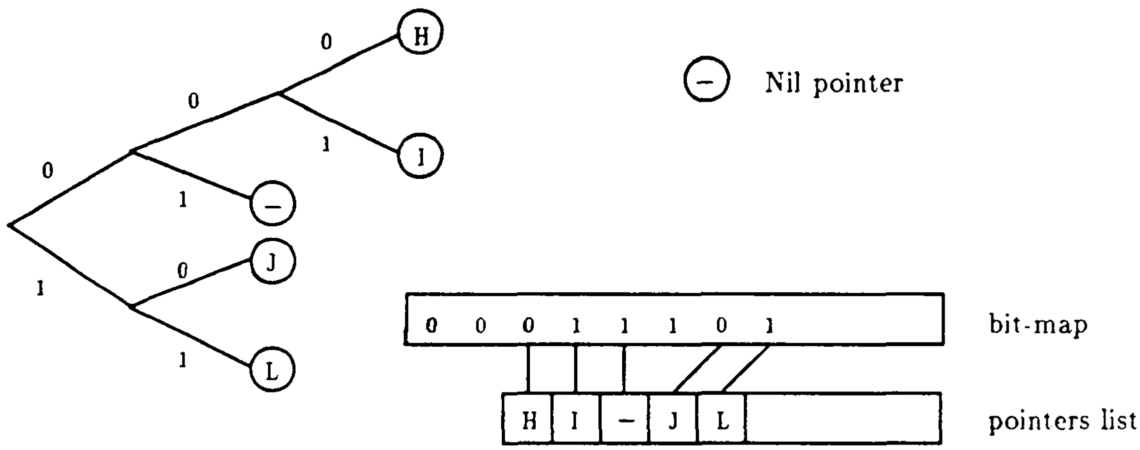


Figure 6: Compact-Trie Representation

The Compact Trie representation is composed of a bit-map and of a pointer-list. The 0 or 1 bit value of the trie corresponds to the digital values of the keys, data I beginning by 001 (Figure 6).

The bit-map is the sequence of 0 and 1 obtained by preorder traversal of the trie (Figure 6). The pointer list associates a pointer to each leaf of the trie.

A basic property of this representation is that any bit of the bit-map followed by a 1-bit represents a leaf node.

This property is true for a trie with a root and two leaves; note that replacing a leaf by a subtrie with an interior node and two leaves corresponds to the insertion of a "01" bit sequence in the bit-map just after the bit representing the replaced leaf (Figure 7), and supposing the property true for any trie of $2n$ leaves, it is also true, by induction, for any trie of $2n + 2$ leaves.

3.2 Insertion

We suppose the key J has the value 10001001 and we want to insert a key K of value 10101111 in the trie of Figure 6; this will lead to the trie and the representation of Figure 7.

The insertion of another key X of value 10001000 will lead to Figure 8; in this figure, the presence of 4 new NIL-pointers is noticeable.

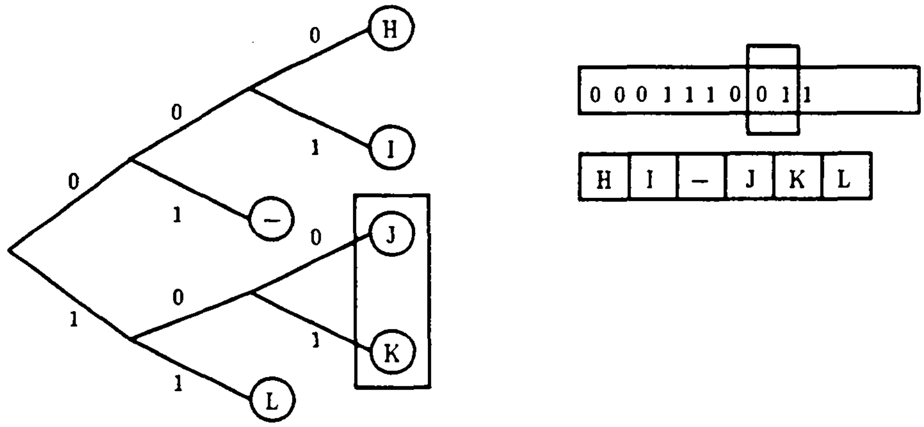


Figure 7: Insertion of key K (10101111)

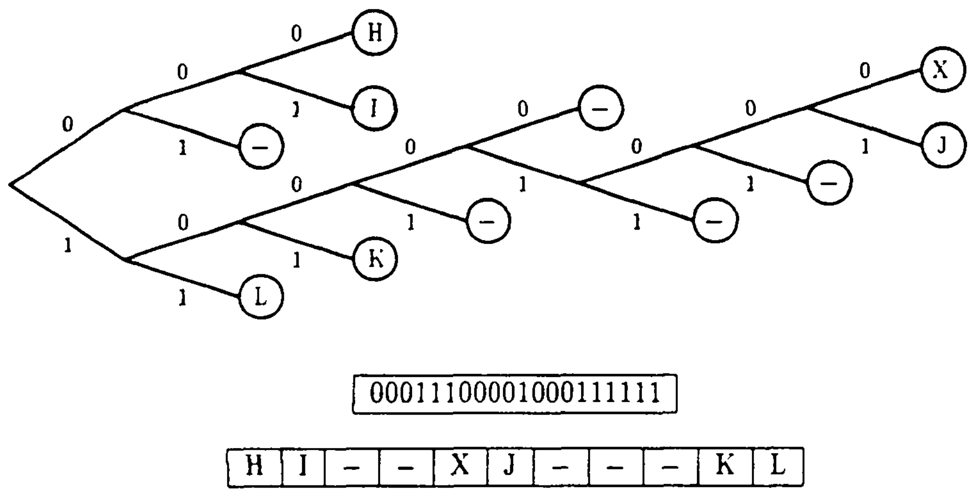


Figure 8: Insertion of key X (10001000)

3.3 Deletion

A key can be immediately deleted by setting to NIL the pointer to the suppressed key.

However it is useful to eliminate the unnecessary NIL-pointers; this is done by iteratively replacing the subtries composed of an internal node, a data-leaf and a NIL-leaf, by a data-leaf, and by updating consequently the bit-map and the pointer list.

Deletion of key X in Figure 8 leads back to Figure 7.

3.4 Retrieval

The retrieval algorithm is based on a joint processing of the bit-map and of the pointers-list.

Each bit of the retrieval key is checked from left to right; for each 1-bit found, the corresponding 0-subtrie is skipped over.

Skipping over a subtrie is straightforward since in any subtrie the number of leaves exceeds by 1 the number of interior nodes.

Therefore, the following retrieval algorithm stands:

```
int         bx;         /* index in the bit-map                    */
int         bk;         /* bit being checked in the retrieved key */
int         px;         /* index in the pointers-list              */
int         nbbit;      /* number of bits in the bit-map           */

retrieve()
{
        for ( bk=1, bx=1, px=1;  bx <= nbbit;   bk++, bx++ ) {
          if ( bitkey(bk) == 0 )                skipstrie();
          if ( bitmap(bx+1) == 1)               break;
        }
}

skipstrie()
{
        int         interior = 0;
        int         leaf     = 0;

        do {
          if ( bitmap(bx+1) == 1 )          { leaf++;  px++; }
          else                                interior++;
          bx++;
        }
        while( leaf < interior + 1 );
}
```

3.5 Data Structure for Compact-Tries

We propose the structure of Figure 9 to handle a significant presence of NIL-pointers (other structures could be chosen, based upon a predicted amount of NIL-pointers).

The first byte of each pointer is only devoted to NIL-pointers; its value is the number of consecutive NIL-pointers at this point.

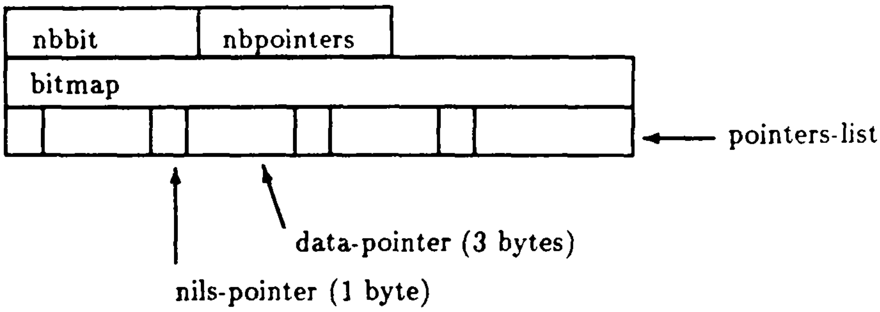


Figure 9: data structure for Compact-Tries

20		6			
00011100001000111111					
0	H	0	I	2	X
0	J	3	K	0	L

Figure 10: Compact Representation corresponding to trie of figure 8

The drawback of this structure is that sequential processing all along the bit-map and the pointer-list is needed, which has two disadvantages: first, the processing time become very important as the number of keys increases; moreover, updating the structure implies locking the whole structure, which is a serious drawback for parallel processing.

These defaults will be corrected in the Compact-Balanced Tries we now present.

4 Compact-Balanced Tries (CB-Tries)

The Compact-Balanced Tries is a natural descendant of the Compact Trie of Jean Kouacou-Kouadio.

A trie is split into pieces and each piece is represented in a compact way by a node comparable to a $B-$tree node.

An edge key is generated at each trie or subtrie splitting and a corresponding starting depth is calculated. The edge key is the key value of the splitting point. The edge depth indicates the number of bits of the binary representation of the edge key to take in account before coming back to the ordinary processing of the bit-map. The edge handling modifies a part of the insertion, deletion and retrieval algorithm, but the global aspect of these algorithms is maintained.

The data structure exhibited to handle the Compact-Balanced Tries closely follows the one presented for handling the Compact Tries.

We will differentiate the trie-nodes or nodes from the Compact-Balanced CB-nodes containing trie-nodes.

4.1 Trie Splitting

We will split the trie T of Figure 6, assuming a 10001001 value for data J (Figure 11).

This key J is the edge key of the subtrie $T2$; the associated edge-depth is 2; the subtrie $T1$ is the beginning part (using preorder traversal) of the trie T; hence it has no edge-depth; the only difference between such a beginning trie and an ordinary trie is that its last leaf may be missing.

The biggest part of Compact-Trie representation is the pointer-list; for this reason, the splitting point will be chosen at middle of the pointer-list, and the first leaf of a subtrie may be a 1-leaf, just as the last one may be a 0-leaf; therefore the subtries generated may remain incomplete, at the beginning, or at the end, or at both extremities.

Insertion of two more keys Y (10100000) and Z (10110000) would produce a splitting of subtrie $T2$ and of the corresponding CB-node $N2$ (Figure 12); subtrie $T3$ is beginning-incomplete.

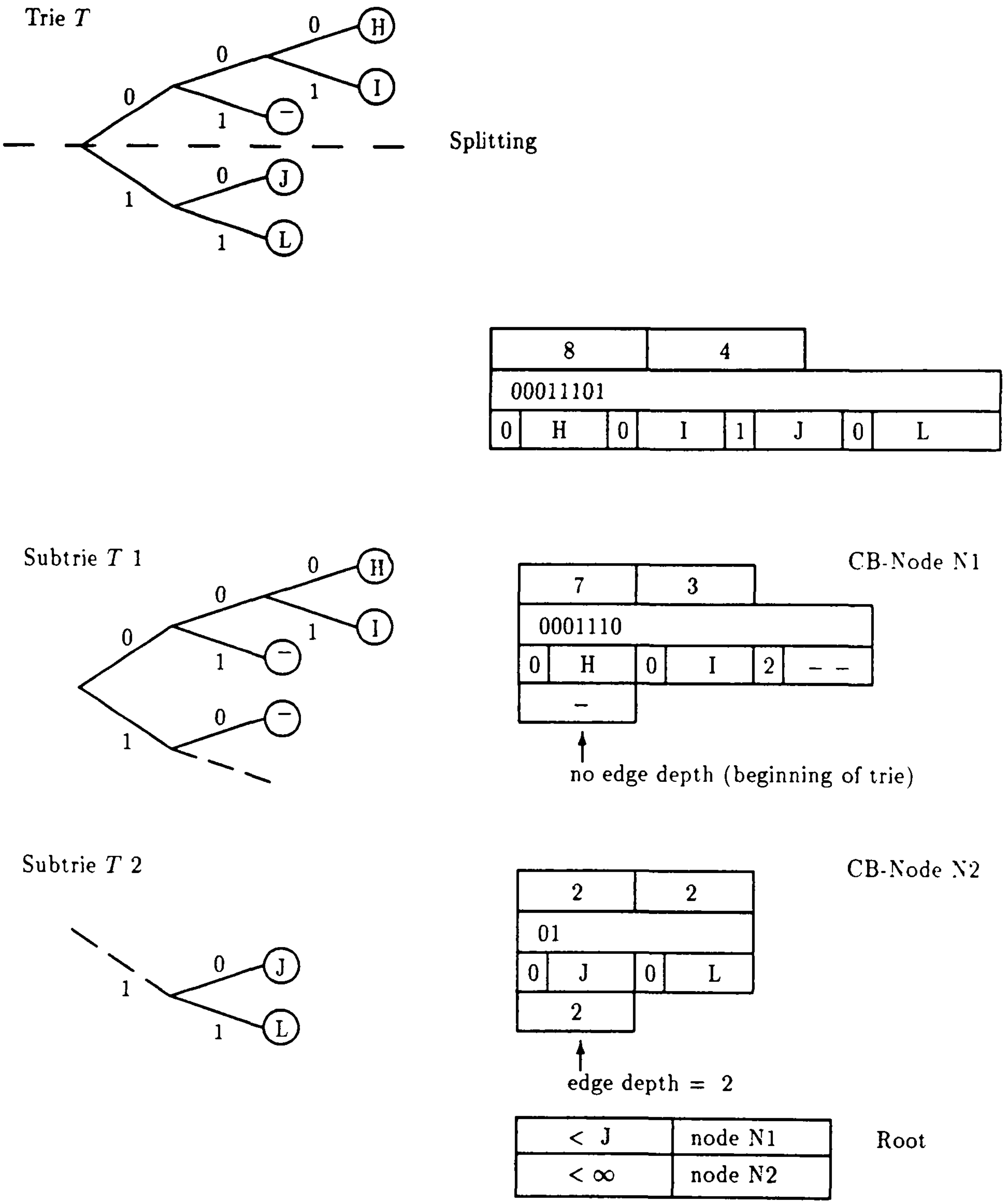


Figure 11: Trie Splitting and corresponding CB-nodes

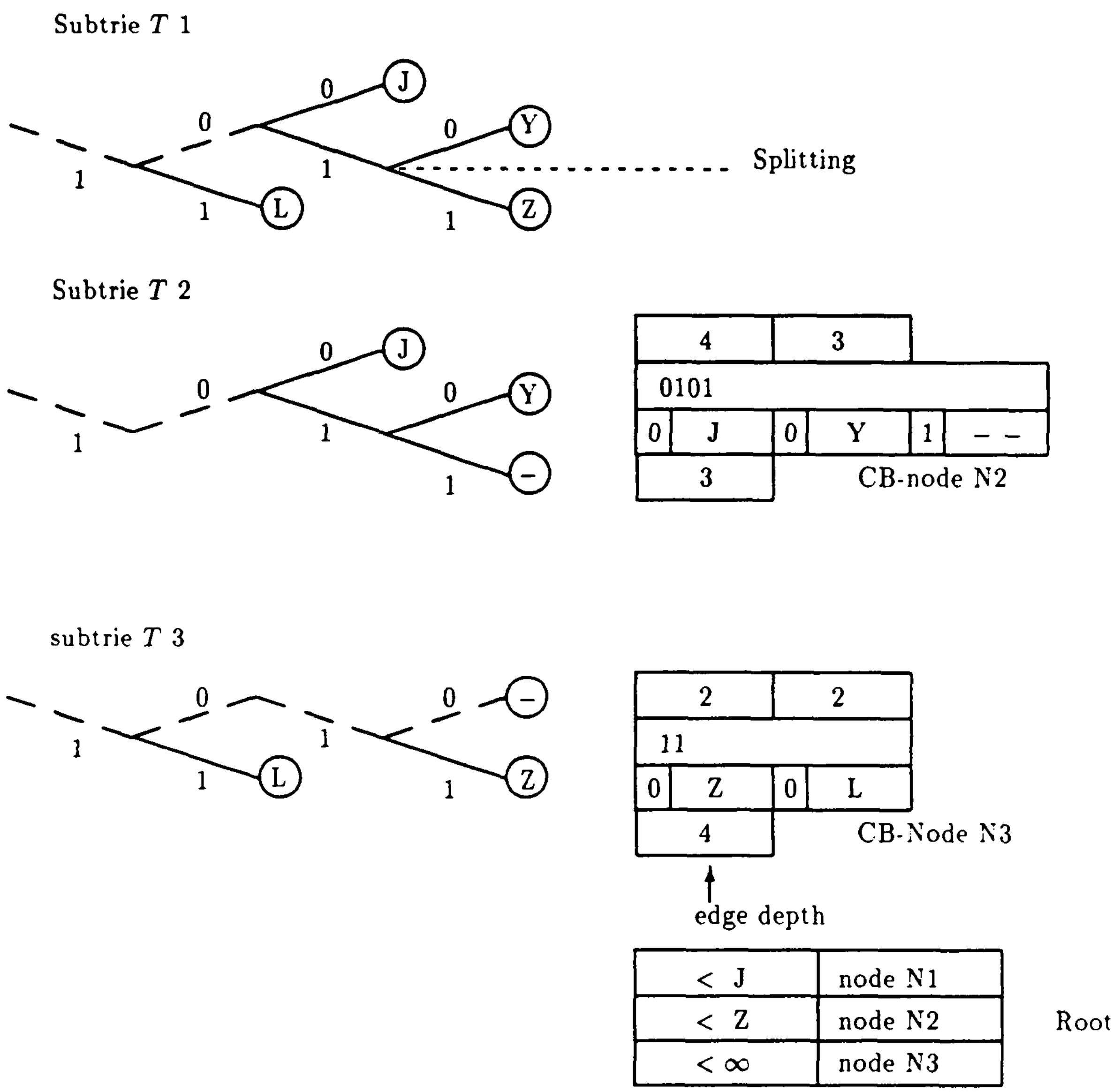


Figure 12: Subtrie Splitting

Between the Figure 11 and the Figure 12, the edge-depth of the CB-node $N2$ (subtrie $T2$) has been modified by insertion of the key Y; this depth modification has to be handled by the insertion process.

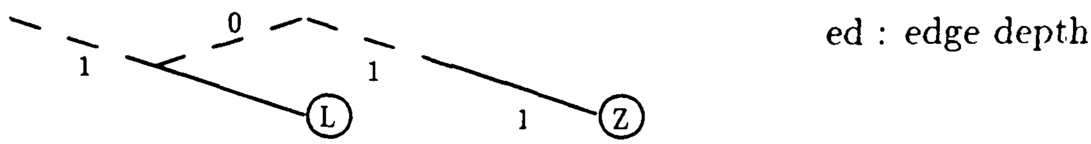
4.2 Root

For the sake of simplicity, the root has been handled in experimental realizations of the algorithm as a single B–tree node and the height of the resulting tree has been limited to two levels; there are no theoretical necessities implying such limitations.

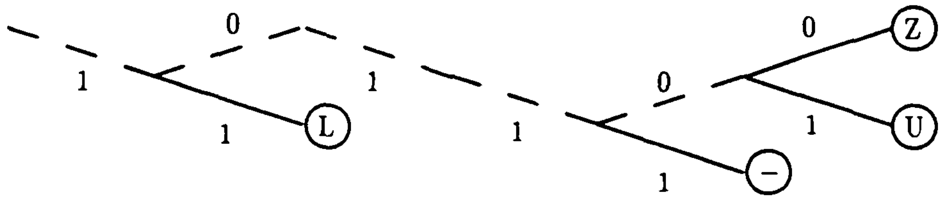
4.3 Insertions

When the edge-depth remains unchanged, the insertion algorithm is the same as the one used for a compact-trie.

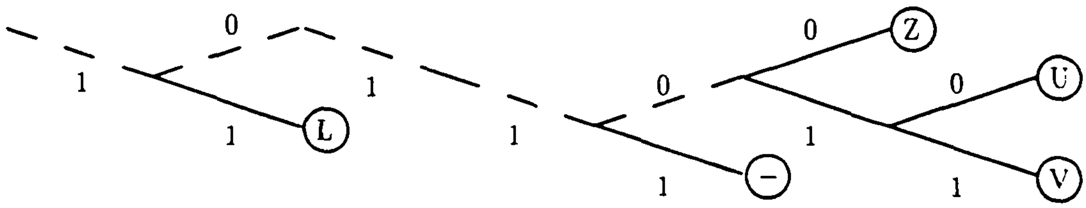
When the edge-depth is modified, the new edge-depth is the position of the first bit that differs in the binary representation of the edge key and of the inserted key; 1-NIL-leaves created by the insertion have to be handled in the bit-map and in the pointers-list; 0-NIL-leaves created by the insertion before the edge key have to be handled only if the inserted key is smaller than the edge key.



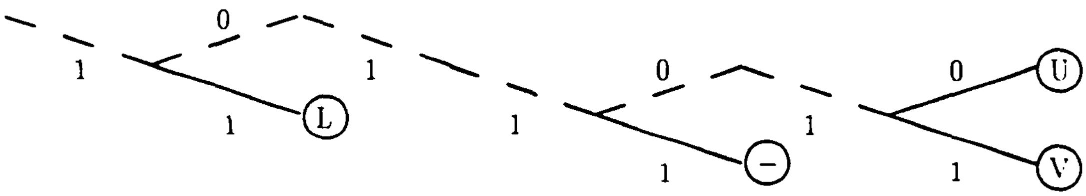
Subtrie T 3 : Z = 10110000, L = 11000000 (ed = 4)



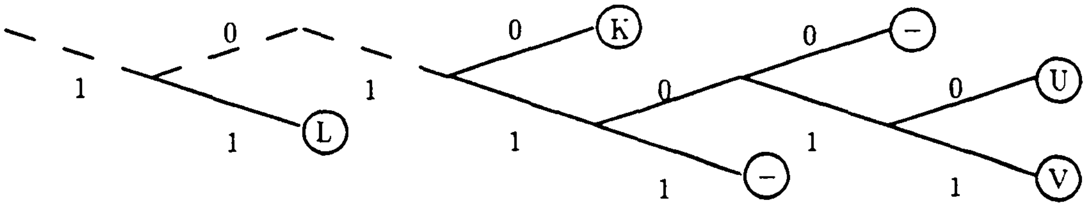
insert U = 10110100 (ed = 6)



insert V = 10110110 (ed = 7)



suppress Z (ed = 7)



insert K = 10100000 (ed = 4)

Figure 13: edge-depth modifications corresponding to key insertions and suppressions

4.4 Deletions

When the edge key is not deleted, the algorithm is the same as in the case of compact trie.

When the edge key is deleted, compute the depth and the position inside the bit-list of the next data leaf (pointer 2 of the pointers-list). Then iterate upwards by checking the key bits of this data with decreasing depth; stop iterating when a 1-brother is either an interior node (detected in bit-map) or a data-leaf (detected in pointers-list). Update consequently the bit-map, the edge-depth and the pointers-list.

4.5 Retrieval

the retrieval algorithm implies subtrie skipping; therefore shadow interior nodes and shadow 0-leaves must be handled along the edge.

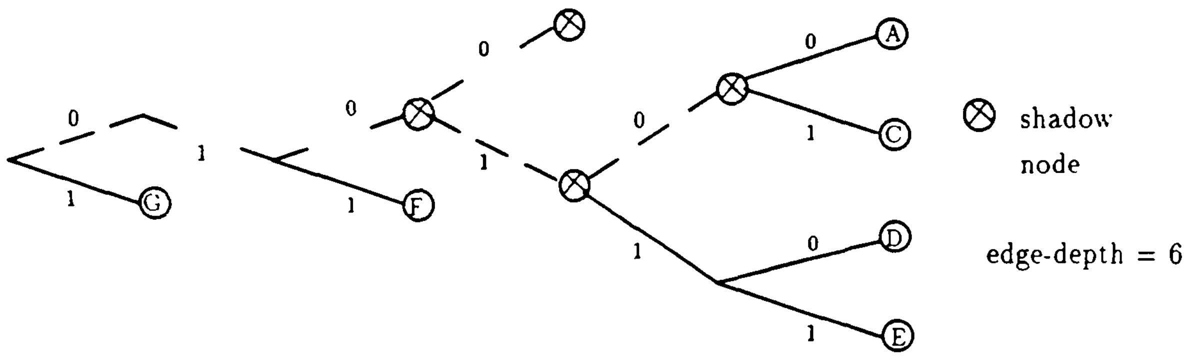


Figure 14: Edge-subtrie skipping (retrieve key F)

Retrieval of key F is proceeded by skipping the subtrie containing the keys A, C, D, E, and four shadow nodes; this subtrie contains 4 interior nodes (3 shadow), and 5 leaves (1 shadow); there is no subtrie to skip if the retrieval key is inferior to the key-edge; there is one and only one edge-subtrie to skip if the retrieval key is superior to the edge-key, and if the position of the first bit which differs in the binary representation of the retrieval key and of the edge-key is inferior or equal to the edge-depth.

There may be several ordinary subtrie skipping during the retrieval processing.

The edge-subtrie-skipping algorithm is given in Appendix A.

4.6 Depth edge calculus

The splitting process requires the computation of the depth of the new edge key (first key of the rightmost of the two brothers CB-nodes exchanging entries).

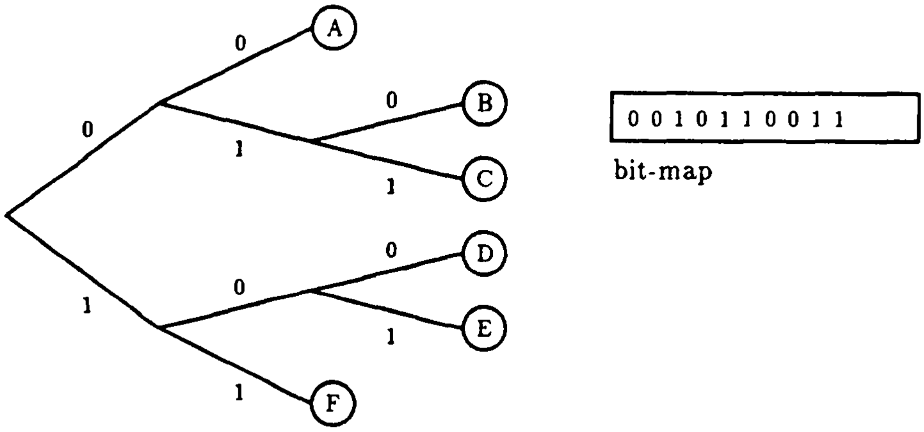
This computation makes use of a pile algorithm, as shown in Figure 15.

The pile is built from the positions of the 0-bits of the last node being accessed in preorder traversal of the trie, along the bit-map.

For each 0-node (0-bit in the bit-map), the depth is increased by one and empiled.

For a 1-node, if the preceding bit is a zero, the depth remains unchanged, elsewhere the depth is equal to the top of the pile; then unpiling is performed.

When proceeding with an edge key, the starting depth is the edge-depth and the pile has to be loaded with the 0-bit positions of the edge-key (limited to the edge-depth); then the node depth computing may be processed as previously.



bit-map index	bit value	node depth	pile of 0-bit positions
1	0	1	1
2	0	2	1, 2
3	1	2	1
4	0	3	1, 3
5	1	3	1
6	1	1	-
7	0	2	2
8	0	3	2, 3
9	1	3	2
10	1	2	-

Figure 15: node depth computing

4.7 Merging and Balancing

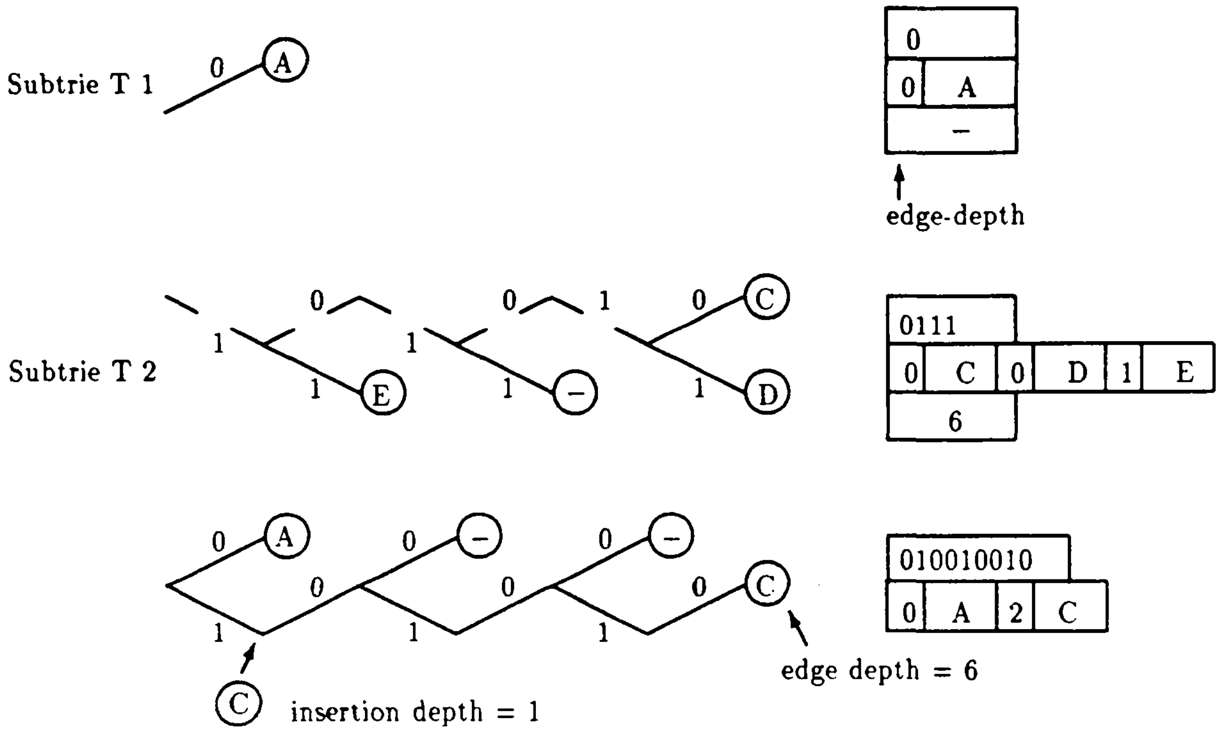
The balancing process between two CB-nodes may be done by merging the 2 CB-nodes in a double-size one, and then by splitting this node in two CB-nodes; therefore we only have to consider the merging process.

We will name the CB-node containing the keys of inferior (resp. superior) value as inferior (resp. superior) CB-node; the merging process will be performed by a special kind of insertion of the first key of the superior CB-node in the inferior CB-node, and then by a concatenation of the two bit-maps, and by a concatenation of the two pointers-lists (with exclusion of the first bit and of the first pointer of the superior CB-node).

Figure 16 and Figure 17 show the two different cases of merging insertion; in case A (insertion depth < edge depth), an ordinary insertion would give a two bits 01 bit-map; this bit-map (and the pointer-list) is extended with interior nodes and NIL-$\emptyset$−leaves until the edge depth of the superior node is reached; no 1-leaves have to be handled.

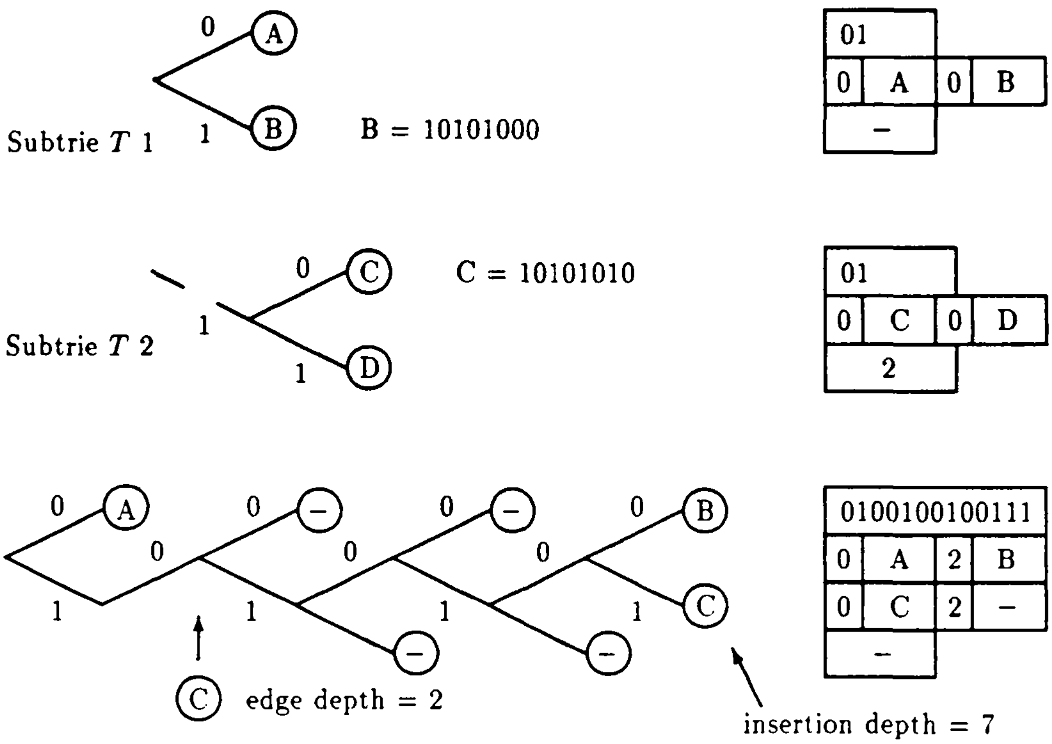
In case B (insertion depth > edge depth), neither interior nodes nor NIL-$\emptyset$−leaves have to be handled, but NIL-1-leaves have to be handled in the bit-map and in the pointer-list, between the edge depth and the insertion depth.

During pointer-list concatenation, if the last pointer of the CB-inferior node is a NIL-pointer (no valid data), this NIL-pointer (number of NIL-leaves) has to be added to the NIL-part of the second NIL-pointer of the CB-superior node.



Case A : insertion depth < edge depth

Figure 16: Merging Insertion of data C in subtrie $T1$ for subtries-merging of $T1$ and $T2$



Case B : insertion depth > edge depth

Figure 17: Merging Insertion of data C in subtrie $T1$ for subtries-merging of $T1$ and $T2$

5 Experimental results

5.1 Experimental results on the Unix Dictionary

Experimental results have been obtained on the Unix "words" dictionary.

The CB-node structure is given in Appendix A.

The experimental results (Appendix B) show that the number of NIL-leaves is about 400% of the number of data-leaves; this gives an average of 10 bits per key in the bit-map, and an average of 18 bits per key when counting the 8 bits of the NIL-pointers. These results are strongly dependent on the structure of alphanumeric data.

It is noticeable that purely sequential keys would produce no NIL-leaves, and therefore would request 2 bits per key in the bit-map, or 10 bits per key when including the bits of the NIL-pointers.

Randomly distributed keys would request an average number of bit per key intermediate between 10 bits and 18 bits.

The distribution of the NIL-pointers values provides indications for further improvement of the structure; for instance, handling the NIL-pointer-list apart of the data pointer list, it is simple to add a secondary bit-list where each 1-bit corresponds to the presence of a NIL-pointer: the NIL-pointer list is reduced to the non-zero NIL pointers; in this case, the number of bit used for representing the Unix dictionary reduces to 16 bits. Improvement is still possible: NIL-pointers may be represented as half a byte, one byte or two bytes long fields, (of the form 0xxx, 10xxxxxx, 11xxxxxxxxxxxxxx); this representation would require only 14 bits per key for the Unix dictionary;

such a representation would require 3 bits per key for sequential data and an average of 8–9bits per key for random data.

Performances of bit-string handling are not optimal and experimental results provide better performances with 50-entries CB-nodes than with 100-entries CB-nodes (this could also be an effect of the locally sequential algorithm); this points that the number of entries in a CB-node is a parameter to choose carefully; programming the bit-string handling in assembly language would certainly improve the performances; propositions are also made in [4] for improving the bit-handling.

5.2 Comparison of the Compact-Balanced Trie and of the Compact 0-complete Trees

The Compact 0-complete Tree [13] uses a 4-byte structure per key, one byte being the current bounding node depth and the 3 next bytes being data or NIL-pointer.

Orlandic and Pfaltz have done asymptotic analysis of the Compact 0-complete Tree [14]; the average storage utilization is 57%; as we did point out in chapter 2, there is no possibility to increase this storage utilization by progressive merging of nodes.

In contrast, our experimental results exhibit an average of 5 bytes per key (bit-map, NIL-pointer, data-pointer), but $B-$tree like flexibility allows by a progressive scanning of the index to force a 100% filling rate, for any key-distribution.

Therefore, a 1000 byte Node would contain an average of 150 entries in case of the Compact 0-complete Trees and about 200 entries in case of the Compact-Balanced Tries (after forcing a 100% filling rate).

The difference would be more important with heavier structures.

Moreover, like in $B-$tree case, a 50% storage utilization is guaranteed for any Node of a Compact-Balanced Trie.

The Compact 0-complete Tree has the advantage in terms of algorithm simplicity and of implementation ease, while the Compact-Balanced Trie has the advantage in terms of storage utilization and of worst cases handling.

6 Conclusions

The Compact-Balanced Tries require between one and two bytes to represent a key; in respect with this excellent compaction result, the relative moderate performance of the bit-map handling and the relative complexity of the algorithm are slight drawbacks.

The $B-$tree type structure of the Compact-Balanced Trie allows partial locking of the structure and multi-user parallel processing; for this reason, Compact-Balanced Tries could be of efficient usage in a parallel machine like Bubba [3].

Balanced Compact Tries offer compactness and all the $B-$tree properties; they are therefore a secure choice for implementation of memory databases.

Balanced Compact Tries can also manage clustering ordered indexes without any special difficulty.

The worst case of insertion corresponds to pairs of keys having N first bits in common; in that case, the number of NIL-leaves is about N, and the number of bits used to represent these keys is about $2N$ (in the bit-map representation); therefore, if N is large, and if there is an important presence of such pairs of key, a Patricia-tree like method [8] would locally be more space-saving than the bit-map representation. This worst case would require about the same amount of memory in a Compact-Balanced Trie as in a classical $B-$tree.

When coping with random key distributions, and remarking that the bit-map representation is a minimal representation for a trie, it is not unreasonable to think that the indexing method we propose do approach a theoretical optimum in terms of compactness.

We proved that the linear representation of tries can be segmented and handled with a complete flexibility; this important novelty allows to step down from global linearity of the algorithms to local linearity and insures that worst cases of insertions are handled as well as a $B-$tree could do; these flexibility and compactness results improve the best results obtained at the moment for such indexes.

The research presented in this paper could be extended in different ways: it is not easy to conjecture if a compact representation may be found for multidimensional data; however, as it has be done with $B-$trees, a Compact Balanced Trie may be used for each dimension of such data; it would be interesting to see if an extension of $B-$trees to multidimensional case, as done with *KB-Trees* [7] could be applied to Compact Balanced Tries; the comparison with *grid files*, as done by Kriegel [10], or with *BANG files* [6], would then be an interesting subject.

Another field of research would be the probabilistic analysis of the NIL-pointers, in both cases of random keys and of alphanumeric keys; results in this direction would allow a further tuning of the structure.

Acknowledgment

I want to thank F. Michel, J.Y. Caleca, C. Frotté and D. Muysers of Copernique who allowed this work to begin.

I am very grateful to Philippe Flajolet for having welcomed me in the "Algorithm project" of INRIA, and having given me decisive hints and continuous encouragement.

I thank also Patrick Valduriez and Mordecai Golin for rereading this document, Virginie Collette for patiently typing it and for realizing the numerous LaTeX drawings, and all the "Algorithm team" for its everyday help and a joyful environment in which it is easier to work.

References

[1] BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 4 (1972), 173–189.

[2] BAYER, R., AND UNTERAUER, K. Prefix $b-$trees. *A.C.M. Transactions on Database Systems 2*, 1 (March 1977), 11–26.

[3] BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. Prototyping Bubba, a highly parallel database system. *I.E.E.E. Transactions on Knowledge and Data Engineering 2*, 1 (March 1990).

[4] DE JONGE, W., TENENBAUM, A., AND VAN DE RIET, R. Two access methods using compact binary trees. *I.E.E.E. Transactions on Software Engineering 13*, 7 (July 1987), 799–809.

[5] Flajolet, P. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica 20* (1983), 345–369.

[6] Freeston, M. The bang file, a new kind of grid file. In *Proc. of SIGMOD Conference* (May 1987), pp. 260–269. San Francisco.

[7] Gütting, H., and Kriegel, H. Multidimensional b–tree: an efficient dynamic file structure for exact match queries. In *Proc. of the Tenth Gesellschaft für Informatik Conference* (Sept. 1980), R. Wilhem, ed., pp. 375–388. Saarbrucken, Germany.

[8] Knuth, D. *The Art of Computer Programming*, vol. 3: Sorting and Searching. Addison-Wesley, 1973. Patricia Trees.

[9] Kouacou-Kouadio, J. *Adressage d'Informations Structurées.* PhD thesis, Université Paris IX Dauphine, Jan. 1986.

[10] Kriegel, H. Performance comparison of index structures for multikey retrieval. In *Proc. of SIGMOD Conference* (June 1984), pp. 186–196. Boston, Massachusetts.

[11] Litwin, W. Trie hashing. In *Proc. of ACM-SIGMOD Conference* (Apr. 1981), pp. 12–29. Ann Arbor, Michigan.

[12] Litwin, W., and Lomet, D. A new method for fast data searches with keys. *I.E.E.E. Software* (March 1987), 16–24.

[13] Orlandic, R., and Pfaltz, J. Compact $\emptyset$–complete trees. In *Proc. 14th VLDB conference* (Aug. 1988), pp. 372–381. Los Angeles, California.

[14] Orlandic, R., and Pfaltz, J. Analysis of compact $\emptyset$–complete trees. In *Lecture Notes in Computer Science* (Aug. 1989), Springer Verlag, pp. 362–371. Szeged, Hungaria.

A Skip-edge-subtrie algorithm

```
struct node {

        int ed;                 /* edge-depth                                   */
        int nbbit               /* current number of bit of the bit-map         */
        int nbpointer;          /* current number of pointers of the poin-      */
                                /* ters-list                                    */

        int         bit-map [BMAPSIZE];
        int         pointer [PTRSSIZE]; /* first  character is NIL-pointers  */
                                        /* next 3 characters are DATA-pointer*/
};
int bx;                         /* bit-map index                                */
int px;                         /* pointer index                                */

skipestrie (differ)             /*          JUMP EDGE SUBTRIE                   */
```

```
int differ;                     /* position of the first bit differing      */
                                /* between the retrieval-key and the edge-key*/
{

      struct node *n = &currentnode;

      char        *edgekey = n->pointer[0];

      int  interior;          /* number of interior nodes (shadows included) */
      int  leaf;              /* number of leaf     nodes (shadows included) */

      int  eb = differ;       /* (key) edge bit position-number              */

      bx = px = 0;

      /* HANDLING OF THE SHADOW NODES */

      for ( interior=0, leaf=0; eb < n->ed; eb++, interior++ )
        if ( bit( edgekey, eb ) == 1 )  leaf++;
      if ( bit( edgekey, n->ed ) == 1 ) leaf++;

      /* HANDLING OF NODES ALONG THE BIT-MAP CONTENT */

      do {
        if ( bx == n->nbbit-1 || bit( n->bitmap, bx+1) == 1 )
                      { leaf++; px++; }
        else interior++;
      }
      while( leaf < interior + 1 );
}

bit( adr, posi )           /*                 BIT VALUE                */

                           /* return the value of the bit at position */
                           /* "posi", first bit being highest bit of  */
                           /* address "adr"                           */
```

B Experimental results

Experimentals Results on the Unix Words Dictionary

Number of words inserted = 24259

Average size of the words	= 7 bytes = 56 bits
Bit-map size	= 249719 bits
Number of non-zero NIL-pointers	= 14796
Average number of bits used for a word	= 10 + 8 (NIL-pointer-size) = 18
Number maximum of entries per node	= 100
Number of NIL-leaves	= 101125

Distribution of NIL-sequence :

(a NIL-sequence is a number of consecutive -- in sense of pre-order traversal -- NIL-leaves, represented in the NIL-pointers; NIL-sequence[11] stands for a sequence of 11 consecutive NIL-pointers)

```
NIL-sequence[ 1] = 2303
NIL-sequence[ 2] = 1779
NIL-sequence[ 3] = 1557
NIL-sequence[ 4] = 1206
NIL-sequence[ 5] = 1045
NIL-sequence[ 6] = 963
NIL-sequence[ 7] = 761
NIL-sequence[ 8] = 692
NIL-sequence[ 9] = 551
NIL-sequence[10] = 535
NIL-sequence[11] = 454
NIL-sequence[12] = 371
NIL-sequence[13] = 342
NIL-sequence[14] = 318
NIL-sequence[15] = 277
NIL-sequence[16] = 190
NIL-sequence[17] = 178
NIL-sequence[18] = 151
NIL-sequence[19] = 138
NIL-sequence[20] = 127
NIL-sequence[21] = 106
NIL-sequence[22] = 87
NIL-sequence[23] = 63
NIL-sequence[24] = 52
NIL-sequence[25] = 64
NIL-sequence[26] = 53
NIL-sequence[27] = 39
NIL-sequence[28] = 45
NIL-sequence[29] = 26
NIL-sequence[30] = 17
NIL-sequence[31] = 22
NIL-sequence[32] = 17
NIL-sequence[33] = 19
NIL-sequence[34] = 8
NIL-sequence[35] = 9
NIL-sequence[36] = 4
NIL-sequence[37] = 7
NIL-sequence[38] = 3
NIL-sequence[39] = 4
NIL-sequence[40] = 2
NIL-sequence[41] = 1
NIL-sequence[42] = 2
NIL-sequence[44] = 2
NIL-sequence[46] = 2
NIL-sequence[47] = 2
NIL-sequence[48] = 1
NIL-sequence[52] = 1
NIL-sequence[65] = 1
```

ISSN 0249-6399