25

Compact Balanced Tries

Pierre Nicodème
Copernique and INRIA, Rocquencourt

[summary by Mireille Régnier]

Classical $B-$*trees* and *prefix* $B-$*trees* [1] offer both fast, direct addressing and easy sequential processing. They are balanced, segmented, and flexible. Flexibility means that a $B-$tree leaf splitting may be done at any position inside the leaf. This property is emphasised: one generates and suppresses empty leaves, while forcing the other leaves to a 100% storage utilisation. This property is important when memory utilisation is a crucial matter, as in memory databases. The segmentation allows parallel processing.
Other methods have been proposed in the last decade; they do not offer all the $B-$tree properties: the *bounding disorder method* [8], *Trie hashing* [7], *compact 0-complete tree* [10], the *compact trie* [5, 3] (section 1). This last structure is based on a bit-map representation of the tries. It is a simple, powerful, but not segmented structure.
It is shown (section 2) how to split the compact trie in a segmented and flexible structure of $B-$tree type: the *compact balanced trie*. Experimental results are given in section 3.

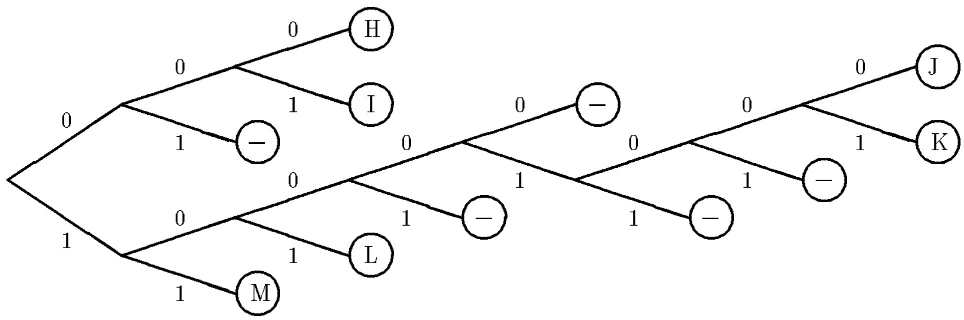
1 Compact Trie

The compact trie representation used by Kouacou-Kouadio, De Jonge, Tanenbaum and Van De Riet [5, 3] 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 (Figure 1). The bit-map is the $0-1$ sequence obtained by right to left preorder traversal of the trie (Figure 1). The pointer list associates a pointer to each leaf of the trie. The basic property of this representation is that any bit of the bit-map followed by a 1-bit, as well as the last bit, represents a leaf node.
Insertion and *deletion* imply updating of the bit-map and of the pointer-list. The *retrieval* algorithm is based on a joint processing of the bit-map and of the pointer-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. This skipping is straightforward: checking the type (internal node or leaf) is easily done with the characterisation above and in any subtrie the number of leaves exceeds by 1 the number of internal nodes. The structure of Figure 2 is used for a compact representation. Each pointer is four bytes long, and the first byte of each pointer is the number of consecutive NIL-pointers at this point. The compact trie is not a segmented structure; therefore the linear search algorithm is $O\left(n^2\right)$ key comparisons with n keys, and partial locking is impossible.

2 Compact-Balanced Tries

2.1 Trie Splitting

Figure 3 illustrates a trie splitting into two subtries. The subtrie $T1$ differs only from the trie T by an incomplete bit-map and pointer-list. An edge-key and a corresponding edge-depth are added to



00011100001000111111

H	I	–	–	J	K	–	–	–	L	M

Figure 1: binary trie (bit-list and pointer-list)

<table>
<tr><td colspan="2">20</td><td colspan="2">6</td><td colspan="2"></td></tr>
<tr><td colspan="6">00011100001000111111</td></tr>
<tr><td>0</td><td>H</td><td>0</td><td>I</td><td>2</td><td>J</td></tr>
<tr><td>0</td><td>K</td><td>3</td><td>L</td><td>0</td><td>M</td></tr>
</table>

Figure 2: compact representation corresponding to the trie of Figure 1

subtrie $T2$. This allows (subsection 2.2) to restart the linear search algorithm of the compact-trie from the node corresponding to the edge key. Therefore, splitting a trie (or equivalently a subtrie) implies the splitting of the bit-map and of the pointer-list, and the creation of an edge-key, with a corresponding edge-depth.

There are *no constraints* for the choice of the splitting point. It may be chosen at the middle of the pointer-list which is the biggest part of the compact trie representation. The iterative splitting of the trie (and of the corresponding CB-nodes) generates a balanced tree structure.

2.2 Retrieval algorithm and edge-depth calculation

Shadow interior nodes and 0-leaves, i.e. leaves accessed through a 0-bit, must be handled along the edge during the retrieval algorithm. Key F retrieval skips 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). The knowledge of the edge-key allows to rebuild the shadow nodes; the edge-subtrie skipping algorithm integrates this construction inside the ordinary subtrie skipping algorithm. The splitting process requires the computation of the depth of a new edge key, which corresponds to the computation of the depth of the corresponding node in the trie. This computation makes use of a stack algorithm (Figure 5 represents the stack evolution for trie T of Figure 3). The stack is

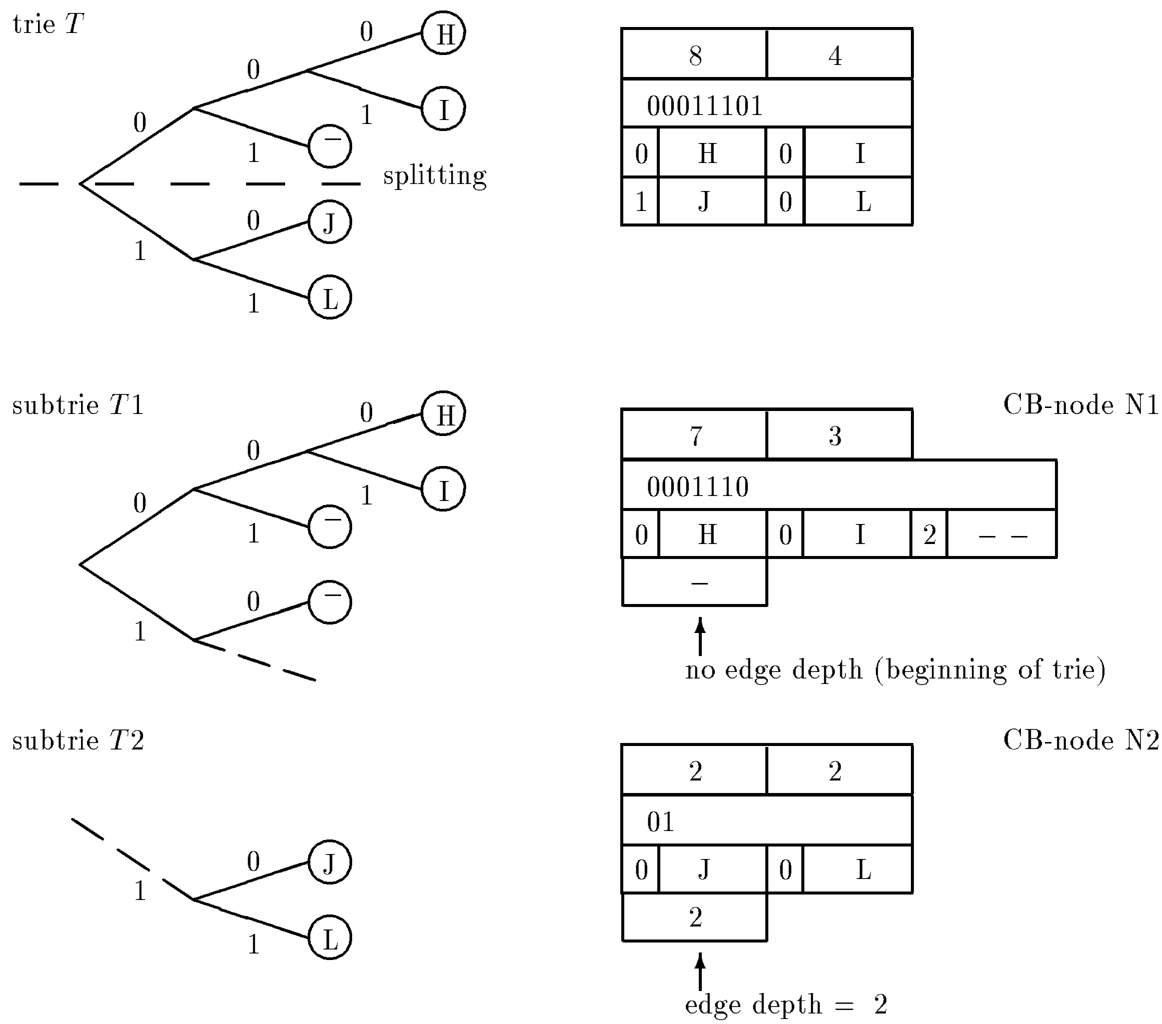


Figure 3: trie splitting and corresponding CB-nodes (compact-balanced nodes)

built from the positions of the 0-bits of the nodes accessed in a preorder traversal of the trie, by use of the bit-map.

The bit-map is scanned from left to right. For each 0-bit, the depth is increased by one and stacked. For a 1-bit, if the preceding bit is a zero, the depth remains unchanged, else the depth is equal to the depth of the top of the pile; thereafter unstacking 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 may be computed as previously. And the CB-Node splitting is achieved.

2.3 Merging and Balancing

The balancing process between two CB-nodes may be done by merging the 2 CB-nodes into a double size one, and then splitting this node into two CB-nodes; merging 2 CB-nodes $C1$ and $C2$

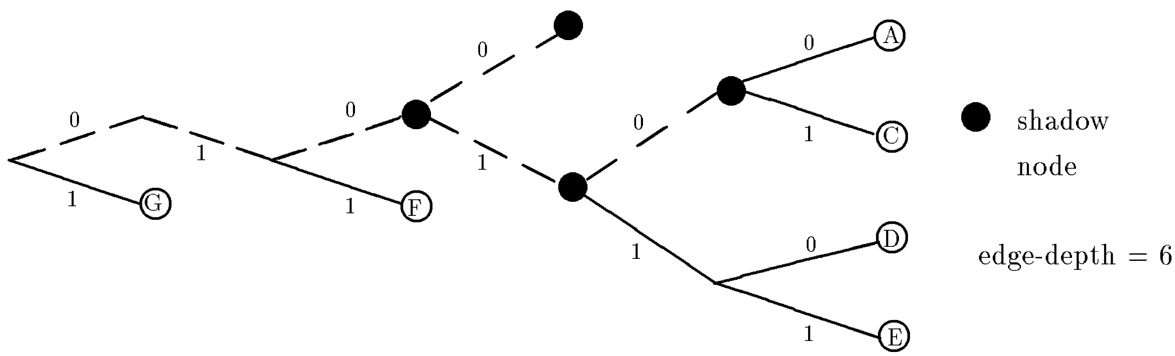


Figure 4: Edge-subtrie skipping (retrieve key F)

index	value	depth	pile (0-bits)
1	0	1	1
2	0	2	1, 2
3	0	3	1, 2, 3
4	1	3	1, 2
5	1	2	1
6	1	1	-
7	0	2	2
8	1	2	-

Figure 5: node depth computing

(keys of $C2$ being bigger than keys of $C1$) results in inserting the edge-key of $C2$ in $C1$, adjusting the bit-map and pointer-list of $C2$ and concatenating them to those of $C1$.

3 Experimental Results

Experimental results have been obtained on the Unix "words" dictionary and on an equivalent number of randomly distributed keys.

Bits/Keys	number of keys	bit-map	NIL-pointers	compressed NIL-pointers
Sequential data	any	2.0	8	1.0
Random data	25259	2.8	8	2.1
Unix Dictionary	25259	10.2	8	4.2

These results, (average number of bits per key in the bit-map and the NIL-pointers list), compare favorably to the results of the compact 0-complete tree (8 bits per key for random data with a 57% storage utilisation [11]); the storage utilisation of a compact balanced trie may be raised to 100%, thanks to the flexibility property (no constraints on the splitting point).
Sequential data do not generate NIL-leaves, while the high number of bits in the case of the Unix Dictionary is strongly dependent on the structure of alphanumeric data which generates many NIL-leaves.

4 Conclusions

These compact-balanced tries provide excellent compaction results. The relative moderate performance of the bit-map handling and the relative complexity of the algorithm are slight drawbacks. Suggestions (use of translation tables) are made in [3] to avoid bit-string handling.
It is 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. The compact-balanced tries may be considered compact $B-$trees and could easily be implemented when lazy deletions policies are used [4]; their segmentation allows parallel partial locking and parallel processing [2], while their compactness fits with the constraints of memory databases (performances should be compared to the ones for $T-$trees [6]).
Moreover, as the bit-map representation is a minimal representation for a trie, one may think that the compact-balanced trie approaches a theoretical optimum in terms of compactness.
The research presented in this paper could be extended in different ways: a compact representation for multidimensional data and a probabilistic analysis of the NIL-pointers, in both cases of random keys and of alphanumeric keys.

References

[1] R. Bayer and K. Unterauer. Prefix B-trees. *A.C.M. Transactions on Database Systems*, 2(1):11–26, March 1977.

[2] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *I.E.E.E. Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[3] W. de Jonge, A. S. Tenenbaum, and R. P. van de Riet. Two access methods using compact binary trees. *I.E.E.E. Transactions on Software Engineering*, 13(7):799–809, July 1987.

[4] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *Proc. of the 8th ACM SIGACT-SIGMOD Conference on Principles of Databases Systems*, pages 235–246, March 1989. Philadelphia.

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

[6] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In *Proc. of the 12th International VLDB Conference*, pages 294–303, June 1986. Kyoto.

[7] W. Litwin. Trie hashing. In *Proc. of ACM-SIGMOD Conference*, pages 12–29, April 1981. Ann Arbor, Michigan.

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

[9] P. Nicodème. Compact balanced tries. In J. van Leeuwen, editor, *Algorithms, Software, Architecture, Information Processing 92*, volume 1, pages 477–483. North-Holland, 1992. Proceedings of the IFIP 12th World Computer Congress, Madrid. Also available as INRIA Research Report n° 1533.

[10] R. Orlandic and J. Pfaltz. Compact 0-complete trees. In *Proc. 14th VLDB conference*, pages 372–381, August 1988. Los Angeles, California.

[11] R. Orlandic and J. Pfaltz. Analysis of compact 0-complete trees. In *Lecture Notes in Computer Science*, pages 362–371. Springer Verlag, August 1989. Szeged, Hungaria.