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 V-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 $T_1$ 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
subtrie \( T_2 \). 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
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)

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
<th>depth</th>
<th>pile (0-bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1, 2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>1, 2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Bits/Keys</th>
<th>number of keys</th>
<th>bit-map</th>
<th>NIL-pointers</th>
<th>compressed NIL-pointers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential data</td>
<td>any</td>
<td>2.0</td>
<td>8</td>
<td>1.0</td>
</tr>
<tr>
<td>Random data</td>
<td>25259</td>
<td>2.8</td>
<td>8</td>
<td>2.1</td>
</tr>
<tr>
<td>Unix Dictionary</td>
<td>25259</td>
<td>10.2</td>
<td>8</td>
<td>4.2</td>
</tr>
</tbody>
</table>

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


