Concurrent Use of B-trees with Variable-Length Entries

Arthur M. Keller and Gio Wiederhold
Stanford University, Computer Science Dept.

Introduction Recent published work provides a solution to a vexing problem in implementing concurrent access to B-trees. The problem occurs when the dichromatic technique to enhance concurrent acces [Guibas and Sedgewick 78] is applied to B-trees with variable-length entries. This technique involves presplitting of B-tree blocks when the tree is traversed from the root down to its leaves to prepare for an update. Any block encountered which is full, e.g., it will not be able to accept another entry, is split during the downward traversal. After such a split, or when a split is not needed, the ancestor block of the index can be unblocked, permitting access to the other $y - 1$ subtrees for blocks of size B containing y entries [Wiederhold 87].

Unfortunately, the dichromatic technique fails when the entries are of variable length, since it is impossible to determine with absolute certainty that a new entry will fit into a partially full block. Variable-length entries are essential to commercial quality B-tree systems for two reasons:

1 Keys represented as character-string entries are naturally variable. Allocating a maximum string size imposes an unnatural constraint on the users, and long fixed sizes, v, to minimize this constraint, drastically reduce the fanout $y = B/v$, causing increases in the depth, $x = \log_y n$, for n entries of the tree and hence longer searches $T = \mathcal{O}(x)$.

2 In order to achieve high fanout, it is quite profitable to abbreviate the key entries. Front abbreviation omits repetitive prefix information, and rear abbreviation omits data which does not descriminate among successor entries. Abbreviation assumes variable-length entries.

A conservative approach that guarantees that the next key to be inserted will always fit is quite wasteful of space. Every index block will have blank space equal to the size of the largest key, and this space can never be effectively utilized. Wasting space also reduces fanout considerably.

Authors' current address: Stanford University, Computer Science Dept., Stanford, CA 94305-2140.

The work reported in this document was conducted under contract N39-84-C-0211 from the Department of Defense.

Pieces of a Solution A recent paper paper [Nurmi, Soisalon-Soininen, and Wood 87] expands a technique proposed by [Sagiv 86] that deals with reducing the cost of handling deletions in B-trees. [Sagiv 86] and others wish to minimize costly sideways checking of block density after entry deletions, needed to re-merge sibling blocks. The probability that blocks can be merged is always small. The proposals favor deferred block deletion, to be done at times when the cost is less. In practice this objective is achieved today by delaying sideways checking until the probability of a successful merge is quite high. Checking may be deferred until the number of entries $y_e < 0.33y$ or even until $y_e = 0$.

In the same spirit, [Nurmi et al 87] propose that for insertions which would lead to an overflow in, the ancestor block, a temporary block be created. This block will have only two entries, one for each of the two results of the split. Now splitting will also not propagate upwards, and concurrent operations are enhanced. To indicate that a node should be split in the future, so that the temporary block interposed can be removed, a flag-bit is inserted in the ancestor block. Any future traversal, encountering this block, can split it and remove the interspersed block. The action can alternatively be deferred to a low cost time.

An Improved Solution: Sibling Promotion We propose to create a temporary sibling pointer when there is insufficient room in the parent block for another child pointer. We take at least half of the child block that needs to be split and place it in a sibling block. We replace the portion copied out with a sibling pointer to the new block. We mark the parent block to indicate that it needs to be split. This clean up split is done on the next update access to the parent block. During the split, the sibling pointer is moved into the new parent. This last clean up step may cause the grandparent block to overflow. In this last case, the algorithm is used again, so that the parent blocks become siblings, with a split pending on the grandparent block. Note that this technique works for any trees including binary trees.

Figure 1 shows the grandparent block, the parent block, and the child block that needs to be split. Figure 2 shows the grandparent block, the parent block, and the child block and its sibling block after the split. Figure 3 shows, after a clean up operation, the grandfather block, the two parent blocks and the two child blocks.

Retrieval is unaffected by this process because a sibling pointer looks and behaves on retrieval exactly like a child pointer. In fact, the sibling pointer is always the last key and pointer pair in a block.

Prior to clean up, the path length through the sibling is one more than normal, and the path length through the original child is equal to normal. In the [Nurmi et al 87] algorithm, it is always one greater than normal. Furthermore, our algorithm requires no additional space, while the [Nurmi et al 87] algorithm uses extra temporary blocks.

Summary Our technique, sibling promotion, combined with recent techniques,. can make B-tree access even more viable in high-concurrency environments with variable-length keys.

References

Guibas,L.J. and Sedgewick,R.: "A Dichromatic Framework for Balanced Trees"; *Proc. IEEE FOCS* 19, 1978, pp.8–21.

Nurmi,O., Soisalon-Soininen,E., and Wood,D.: "Concurrency Control in Database Structures with Relaxed Balance"; *Proc. ACM PODS*, Mar.1987, pp.170–176.

Sagiv,Y.: "Concurrent Operations on B-trees with Overtaking"; *J. of Computer and System Sciences*, Vol.33, 1986.

Wiederhold,G.: *File Organization for Database Design*; Mc-Graw Hill, 1987, pp.155–156 and 262–269.

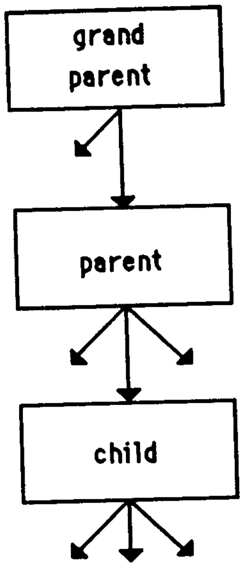

Figure 1

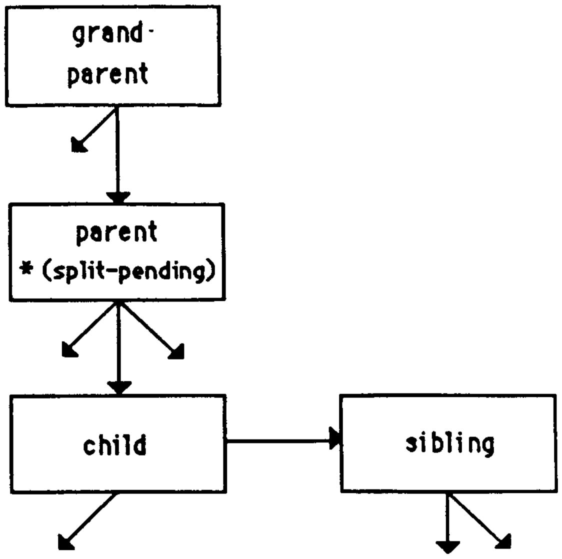

Figure 2

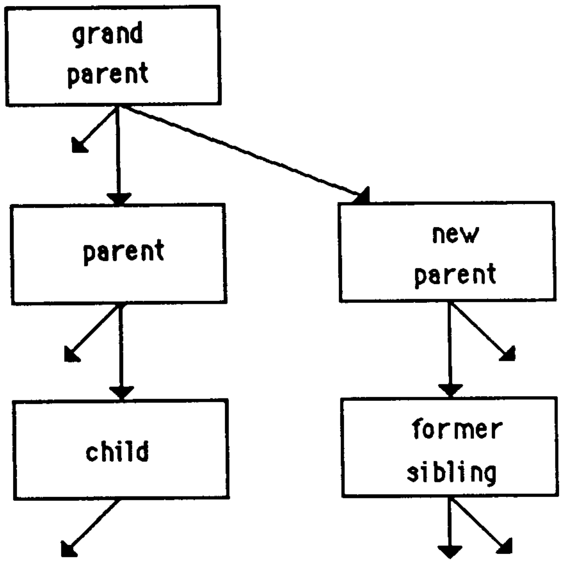

Figure 3