New Concurrency Control Algorithms for Accessing and Compacting B-Trees

V. W. Setzer
Department of Computer Science
University of São Paulo
C.P. 20570
01452-990 - São Paulo, Brazil
vwsetzer@ime.usp.br

A. Zisman
Department of Computer Science
University of São Paulo
C.P. 20570
01452-990 - São Paulo, Brazil
zisman@ime.usp.br

Abstract

This paper initially presents a brief but fairly exhaustive survey of solutions to the concurrency control problem for B-trees. We then propose a new solution, which is characterized by the use of variable-length indices, the employment of a single lock type for the usual access operations and preemptive splits as well as delayed catenations and subdivisions. We also introduce a new compaction algorithm and its concurrent execution, using a new lock type.

1 Introduction

The B-tree, introduced in 1972 by Bayer and McCreight [BM72] has become the standard structure for the implementations of indices for file and database management systems, because of its efficiency in sequential and random accesses using index values. (An extensive and up-to-date survey may be found in [Zis93].) The file sharing by multiuser environments lead to the need of introducing features for the concurrent access to data and index files. This has to be done preserving the information integrity obtained or given by the users.

This work was supported by the São Paulo Research Foundation (FAPESP) grant No. 93/0603-1.

**Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994**

This paper presents a new concurrency control method which can be applied to two types of B-trees and some of its variations. It has the following characteristics: 1. Explicitly locks a small number of nodes (maximum of 2) for each executing process accessing the tree, except during compaction. 2. Uses just 2 lock types, one of them dedicated to the compaction operation; this operation, which has not been covered in the literature of concurrency control methods, may be executed simultaneously with reading, inserting, deleting and updating operations. 3. Delays catenation operations and preempts rearrangement of elements among nodes as well as node splits, thus permitting a good load factor and avoiding the repetition of path traversals by each process. 4. Permits the use of variable-length indices.

The rest of this paper is organized as follows. Section 2 defines general terms necessary to understand the whole text and characterizes the B-tree types which will be used. Section 3 presents a brief but fairly complete survey of published papers covering concurrency control for B-trees; it is preceeded by specific definitions for this section. Section 4, after introducing specific definitions, informally presents the new method as well as a precise description of the involved algorithms and their extension to variable-length index values. Section 5 summarizes the results and makes proposals for future research.

2 General Definitions

D1 Given a data file composed of data records, an *index* is a simple or compound field of each of these records whose *value* is used for searching certain records.

D2 A B-tree of order K, where K is a natural number, is an oriented tree, with the following properties:

1. Each tree node corresponds to a single disk page;
2. Each node, except the root, has m fixed-length index values, where $K \leq m \leq 2K$, stored in the node in nondecreasing order;
3. The root contains from 1 to $2K$ index values;
4. Every tree node n is either a leaf, that is, has no descendents, or is an *internal node* and contains $m+1$ sons. To each of these sons, except the first one, is associated an index value i and a pointer p stored in n. If n is not a leaf, each subtree whose root is pointed to by some p_j in n has index values strictly greater than its corresponding i_j $(1 \leq j \leq m)$. An extra pointer p_0 in n points to the first son, wich is the root of the subtree that contains index values strictly smaller than the value of i_1.
5. For every index value i a pointer d is associated to the data record which contains i;
6. Every leaf node is in the same tree level, that is, each path from the root to a leaf has the same length.

D3 Let p_j and p_{j+1} $(0 \leq j \leq m-1)$ be two adjacent pointers in a nonleaf node n which points to n' and n'', respectively. The index value i_{j+1} is called the *separator* of n' and n''.
D4 *B^*-tree.* To decrease the B-tree height (cf. **D2**), Knuth suggested in 1973 [Knu73], a variation called B^*-tree[1]. In this structure, data records are stored directly into the leaves. The latter are connected by pointers forming a linked list. The internal nodes contain only index values and pointers to their sons. We will consider B^*-trees where the index values in internal nodes are organized as in a B-tree, that is, internal nodes do not duplicate index values but for father nodes of leafs.
D5 *B^+-tree.* A variation of the B^*-tree where the leaf nodes hold pointers to the data records instead of the records themselves. The leaf nodes also form a linked list through pointers to the immediate right-hand side neighbours.
D6 A node is *complete* if $m = 2K$.
D7 A node becomes *overflown* if $m = 2K$ and there is an attempt of inserting another index value into it.

[1] Actually, Knuth did not give any name to this kind of tree. Many authors call it B^*-tree. On the other hand, Comer [Com79], Korth [KS86], Boswell and Tharp [BT90] designate it as a B^+-tree.

D8 A node becomes *underflown* if $m = K$ and there is an attempt of deleting any of its index values.
D9 The *free space* of a node is the length in bytes (or words) of its portion which does not contain index values and pointers, and other information necessary to represent the tree.

When a node n' becomes overflown, it is *split* or its items are *subdivided* among its neighbours.
D10 In a B-tree, the *split* of a node n' consists of the creation of a new node n'', adjacent brother of n', and then dividing n''s items between n' and n''. n' will hold the K smaller index values and its asssociated pointers, n'' the K greater, and the median index value among all of the values of n' and n'' becomes the separator (cf. **D3**) of these nodes. The separator and a pointer to n'' is inserted in the father node of n'. In the case of B^*- and B^+-trees the split of a leaf node produces the duplication of the median index value, which is inserted into the father node. For the internal nodes, there is no duplication.
D11 A *2-3-split* is a technique introduced in [Knu73] where the items of an overflown node n' are united to those of an adjacent brother n'' and divided (as uniformly as possible) between n', n'' and a newly created node n''', adjacent to n''.
D12 The *suvdivision* of the items of n' consists of moving some of these items to one of its adjacent brothers n'' which is not complete, if it exists. This is done by merging the items of n' and n'', their separator and the new value being inserted into n', and then subdividing these values into n' and n'', inserting the new median value in n''s father as a separator, such that the number of values in n' and n'' differs at most by 1. In the literature, this operation is called *overflow* and was introduced by Bayer and McCreight [BM72].

When a node n' becomes underflown, it is *catenated* with one of its adjacent brothers n'' (the inverse process of splitting), or its items are *distributed* between the items of n' and n''.
D13 In a B-tree, the *catenation* consists of merging the items of n', n'' and their separator, when the total number of these items is less than or equal to $2K$. The resulting items are stored into one of the nodes and the other one is deleted. The separator is eliminated from the father of n' and n''. If the merge cardinality is greater than $2K$ a *distribution* is performed between n' and n'' analogously to the subdivision operation; this operation was called *underflow* by Bayer and McCreight (notice that we have changed this denomination and "overflow", because we use them to indicate a node status (cf. **D7, D8**).

D14 A process traversing the B-tree may be classified as *reader*, which executes the operation of searching a certain index value i, or as *updater*, which performs an insertion (deletion) operation. The latter consists of two subprocess: *updater-reader*, which searches for the node where (from where) i will be inserted (removed) and *updater-writer*, which produces the insertion (deletion) with an eventual restructuring of the tree.

D15 The nodes visited by a process P constitute P's *access path*.

D16 A *compaction* process traverses the whole tree producing its reorganization in order to diminish the free spaces (cf. **D9**) of its nodes.

D17 A process *locks* a node n when it associates to n a mark indicating to other processes some access restriction to n.

D18 A node contains an *exclusive* lock produced by a process P when it cannot be visited by any other process until P unlocks it. When P is about to modify some node, P has to mark it with an exclusive lock. When a process reaches a node with an exclusive lock it remains in a waiting state, in a queue associated to that node, until it is unlocked.

D19 A node n is *safe* when an insertion or deletion operation does not affect any of its antecedents; otherwise it is *unsafe*. It follows that for an insertion a node is safe when it contais less than $2K$ index values; for a deletion a node is safe when it has more than K index values.

3 Solutions for the Concurrency Control Problem

To permit the use of B-trees and its variations in multiprocessing and multiprogramming applications, many solutions have been proposed to the concurrency control problem. This solutions are described here briefly, in their chronological order of appearence in the literature.

3.1 Specific Definitions

Kwong and Wood [KW80b, KW80a, KW82] defined the following three locking techniques.

D20 *Lock-Coupling.* During the execution of a reader process (cf. **D14**) a node is unlocked only when its appropriate son is locked. During the execution of an updater process every node of its access path (cf. **D15**) is locked until it reaches a safe node (cf. **D19**). At this moment all antecedents of this safe node are unlocked.

D21 *Driving-off.* A process has its execution interrupted by an updater process when the latter inserts exclusive locks (cf. **D18**) into the nodes which it traverses.

D22 *Side-Branching.* During an insertion (deletion) operation, when a node becomes overflown (underflown) (cf. **D7**, **D8**) it is not split (catenated) (cf. **D10**, **D13**). An appropriate half of its items is copied into a new node and the new index value is inserted into this half (the items of its adjacent brother are joined to its items without the index value to be deleted). The traversal is backed-up until a safe node is reached; resuming the top-down traversal, the excess half of each original node is (the redundant nodes are) removed.

D23 *B^L-tree.* Introduced by Lehman and Yao in 1981 [LY81] the B^L-tree (from the original $B^{Linked} - tree$) is a variation of the B^+-tree (cf. **D5**). Each node contains an excess pair (index value, pointer) that is, for each node n there exists a pair (i_{m+1}, p_{m+1}), $K \leq m < 2K$, where i_{m+1} is the greatest index value in the subtree rooted by p_m in n, and p_{m+1}, called *link pointer*, points to n's adjacent right-hand side neighbour.

D24 *2-3-tree.* Introduced by John Hopcroft in 1970, (not published, referred by [Com79, Knu73]) it is a balanced tree where each nonleaf node has at least two and at most three sons. Its use is suitable for main storage devices.

In every method a process queue is created for each node subjected to an unfulfilled lock request (because it was already locked by another process). These requests are later on executed according to the queue order.

3.2 History

The first solution for the concurrency control problem was proposed by Samadi in 1976 [Sam76] and by Parr in 1977 [Par77]. In this solution only the locking and unlocking operations are valid. These operations are executed through the standard *semaphore* technique (Dijkstra [Dij68]) using only one lock type. Actually, reader and updater processes use the lock-coupling technique (cf. **D20**).

In 1977, Bayer and Schkolnick [BS77] presented four solutions for B^*-trees (cf. **D4**). These solutions use three types of locks: *read* (ρ-lock), *alternative* (α-lock) and *exclusive* (ε-lock). A process may place a ρ- or an α-lock into a node which already contains a ρ-lock. A process may only place an ε-lock in a node if this node is unlocked; other processes cannot access a node with such a lock. A process may convert an α-lock to an ε-lock. In all of their solutions, read processes place ρ-locks into the traversed nodes through the lock-coupling technique specific for this type of processes. The updater processes are executed in the following way.

Solution 1. Uses the lock-coupling technique employing ε-locks in the visited nodes.
Solution 2. The updater-reader subprocess is executed in the same way as a reader. Upon reaching a leaf node an ε-lock is placed into it. If this node is not safe, unlock it and its father; solution 1 is then applied.
Solution 3. The updater-reader subprocess is executed with the lock-coupling technique using α-locks. When a leaf node is reached a new traversal is initiated at the root, converting each α-lock to an ε-lock
Solution 4. This solution consists of a generalized combination of the other three solutions. Depending upon certain parameters it uses solutions 1, 2 or 3 for specific parts of the tree.

In 1978 Miller and Snyder [MS78] introduced a solution for B-trees which, in comparison to the previous Bayer and Schkolnick's solutions, is characterized by locking only those nodes that are going to be modified. It uses three lock types: *access* (a-lock), *pioneer* (p-lock) and *follower* (f-lock). A process may place an a-lock into a node which already contains an a-lock. The other two locks are exclusive. The reader part of any process traverses the tree locking the visited nodes with an a-lock, unlocking the current node before locking its son. When a leaf node n is reached, it is marked with a p-lock. One has to distinguish between insertion and deletion operations. An insertion operation performs p-locks into up to 3 ancestor nodes of n and f-locks its and its father's adjacent brothers. When a split propagates upwards, this block of locked nodes is moved up accordingly. A similar process is performed for deletion operations, with the difference that the sons of the brothers of n's father are also marked with f-locks.

In 1980 Ellis [Ell80] proposed a solution for 2-3-trees (cf. **D24**). This solution is based upon Bayer and Schkolnik's and in Lamport's idea [Lam77] of permitting reader and updater processes to simultaneously examine the same node, in opposite scanning directions.

In 1981 Lehman and Yao [LY81] defined a method for B^L-trees (cf. **D23**). This method uses just one lock type and executes a constant, small number of node locks for each executing process. Reader processes and updater-reader subprocesses do not execute any lock. When a reader process P reaches a leaf node n, this node is locked; if another process, simultaneous to P, has split n, P may require the visit of n's adjacent leaf node. The latter is locked, becoming the current node, and n is unlocked. This solution does not use catenations and distributions (cf. **D13**) and permits the existence of underflown nodes (cf. **D8**).

In 1986 Sagiv [Sag86] improved this solution, producing the locking of just one node by each executing process. His solution does not lock n's adjacent brother; in order to guarantee that it is using a node's latest version, it locks this node and rereads it just before its modification. This solution uses a "special block" storing the tree's height and a vector of pointers to every lefmost node of each level.

In order to improve Bayer and Schkolnick's, as well as Ellis' solutions, Kwong and Wood suggested from 1980 to 1982 [KW80b, KW80a, KW82] a new solution for B^*-trees which makes use of the 3 locking techniques defined in section 3.1 (cf. **D20, D21, D22**). The reader process is executed as in Bayer and Schkolnick's solutions; the updater-reader subprocess is executed as in their solution 3 (see above). But, when a leaf node has to be split (cf. **D10**) (catenated) the side-branching technique is employed, in the leaf-to-root direction, until the first safe node in the traversed path is reached. At this moment, the nodes are updated in the root-to-leaf direction, using the driving-off technique.

In 1985 Mond and Raz [MR85] proposed a solution for B^*-trees based upon the algorithm given by Guibas and Sedgewick in 1978 [GS78], who introduced preemptive splits for updater processes for 2-3 and 2-3-4-trees. Their trees do not complain with our definitions **D10** and **D13**: the split and catenation operations are performed like those we defined for B-trees. Mond and Raz's solution produces preemptive splits/catenations for complete/half-complete nodes traversed by reader and updater-reader subprocesses, leaving these nodes ready for future insertions and deletions. It uses limiting numbers of $K-1$ and $2K+1$ index values instead of the usual K and $2K$. Complete nodes contains $2K$ or $2K+1$ index values. Catenations are perfomerd for nodes with K or $K-1$ values. The method employs two lock types determinded by the type of process being executed, guaranteeing that at most two nodes are locked at each instant by each executing process. Differently from the lock-coupling technique, these locks are performed pairwise, in the current node n and in its father. Before locking n's appropriate son, n's father is unlocked.

We detected a problem with Mond and Raz's solution. The catenation operation may produce a complete node. If this node is not changed, a subsequent process visiting it will perform a split, starting a possible cicle of these operations.

In 1988 Keller and Wiederhold [KW88] afirmed that Mond and Raz's method cannot be applied to indices with variable-length values, and introduced the ***sibling promotion*** technique: upon reaching a complete node n, this technique creates a sibling node of n, including half of n's information in the new node, and a "mark" in n's father n_f, indicating the fact that it has to be split and the presence of that sibling. When another process reaches n_f, a separator between n and its created sibling node is introduced into n_f.

In 1992 Souza and Carvalho [SC92] proposed a method for B^*-trees in which the access path of each executing process remains in central storage. When a process reaches a locked node, it starts to successively traverse its path until it is allowed to proceed. This method permits the use of the subdivision technique when a node is overflown.

4 The Method

The concurrency method presented here may be applied to B^* and B^+-trees (cf. **D4, D5**) which will be referred to in the sequel simply as *trees*.

4.1 Specific Definitions

D25 The *load factor of a node* is given by $F = \frac{I}{2K}$, where I is the number of index values stored at that node. The *load factor of a tree level* l is $F = \frac{I}{2KN}$, where I and N are the number of index values and nodes stored at l respectively, and K the order of nodes at l. The *total factor of a tree* is $F = \frac{I}{2KN}$, where I and N are the number of index values and nodes stored in the whole tree respectively. Notice that for B^* and B^+-trees K differs from leaf to internal nodes because they store different types of informations. In these cases the total load factor is given by $F = \frac{I'+I}{2K'N_i+2KN_f}$, where I' and I are the numbers of index values stored in the internal and leaf nodes respectively, K' and K are the order of the internal and leaf nodes, N_i is the number of internal nodes and N_f is the number of leaf nodes.

D26 The *load factor vector* of a tree is a vector containing the load factor of each level of that tree.

D27 A *f-lock* is an exclusive lock introduced into a node by a reader or an updater process. A *c-lock* is introduced by a compaction process (cf. **D16**).

D28 The *compaction indicator* is a Boolean variable whose value indicates if the tree is (*true* value) or is not (*false* value) being compacted. This indicator is verified before the execution of a reader or updater process.

D29 A *underflown son flag* is a Boolean variable associated to each i_j index value ($1 \leq j \leq m$) of a non-leaf node n. This variable indicates if n's son pointed to by p_j is (*true* value) or is not (*false* value) underflown, avoiding unnecessary reading of that son. (This variable may be implemented as the lefmost bit of each pointer or as a bit vector stored in n).

D30 The *compaction queue* is a data structure which stores an identification of every process that was interrupted due to the execution of a compaction process. After finishing the compaction, the queued processes are automatically activated; they resume their action at the root node.

D31 A *3-2-catenation* is a technique where the items of 3 adjacent brother nodes are united with their correspondent separators (stored in their father) and divided (as uniformly as possible) between 2 of the 3 nodes. This is the contrary to a 2-3-split (cf. **D11**).

4.2 General Description

The method employs the technique of maintaining the tree nodes safe both for insertions as deletions, based upon Mond and Raz's [MR85] idea, who applied to B-trees the technique introduced by Guibas and Sedgewick [GS78] (see subsection 3.2).

Besides Mond and Raz's application of previous splits, our method executes previous subdivisions, as well as delayed catenations. Thus, it avoids propagations of tree restructurings. Moreover, it uses just one lock type, simplifying the algorithms.

During read, insertion and deletion operations, f-locks (cf. **D27**) are performed into the current node n and its father. Before visiting an appropriate son n' of n, the latter's father is unlocked. So, one guarantees that for each executing process only 2 nodes are explicitly locked at each instant. Notice that their descendants are implicitly locked for processes which have not reached n's father yet. Nevertheless, some of these descendants may contain f-locks due to other processes which have already passed through n'.

Let P be the executing reader process or updater-reader subprocess, and n its current internal node, where n is complete (cf. **D6**) and unlocked. In this case, in order to make n safe for future insertions and to unlock its father, one of the following operations is performed. If n has adjacent underflown sons, indicated by their underflown son flags (cf. **D29**), the latter are catenated. Otherwise, a subdivision (cf. **D12**) between n and some of its brothers, or a split of n is performed. As will be described later, the choice between a subdivision or a split depends on the load factor (cf. **D25**) of n's father level, stored in the ocupation factor vector (cf. **D26**).

The catenation operation tries to eliminate some separator of n and is executed on 2 of its unlocked underflown sons, which are either adjacent or separated by an unlocked, not underflown node. In the latter case, a 3-2-catenation is executed (cf. **D31**). In both cases, the resulting node or nodes will not be complete, because two of the sons were underflown, thus avoiding a cicle of consecutive catenation and split operations as happens with Mond and Raz's method (see subsection 3.2). If any of the involved n's sons has been locked by other processes, P enters into a wait state until these locks are removed.

The subdivision operation is performed when the catenation could not be executed, n is complete and the load factor of its father level is greater than a predetermined value f_s, which will be called *split factor limit* (for example, $f_s = 85\%$). This factor indicates the situation where it is preferable to subdivide instead of to split, because the latter produces the insertion of a new separator in n's father. If n has at least two not complete left and/or right brothers the operation tries to make a subdivision among the elements of n and its brothers, first examining the left and then the right ones. That is, the subdivision is performed inspecting at most four (or another predetermined number) consecutive brothers of n. If some of these brothers have been locked by another process, P remains in a wait state, until the lock is removed. When n does not have two brothers at the same side, only the existent nodes are examined.

If n is complete, the load factor of n's father level is greater than f_s and it was not possible to perform a subdivision of n (which means that the adjacent brothers are complete), then a 2-3-split (cf. **D11**) is done between n and one of its brothers. If one of the brothers involved is locked by another process, P waits until it becomes unlocked.

If n is complete and that load factor is less than or equal to f_s, then a normal split is executed, saving the access to its brothers. Note that the split operation does not produce a propagation of new splits in the bottom-up direction because, by construction, n's father has enough space for the insertion of a new element and, as it is locked, no other process has modified it. This is an essential point of this method, because this way one avoids the need to lock a great number of nodes.

Before catenating some of n's sons, it is not necessary to lock these nodes, because at this moment n is locked and no other process which has not reached it may lock its sons. The same applies to subdivisions and split operations, done at brothers and sons of n, as n's father is also locked.

Contrary to the solution of Mond and Raz, described in section 3, our method does not distinguish between reader and updater processes, that is, they have the same behavior.

Notice that in this method an underflown node may remain in this state until its father becomes complete. Then catenations (two nodes in one or 3-2) are tried. Considering that in practice the number of insertions tends to be asymptotically greater than or equal to the number of deletions (because, on the contrary, the file could disappear), it is convenient to delay those operations. (Lehman and Yao [LY81] have also pointed to the fact that the number of insertions surpass that of deletions.)

When the amount of underflown nodes is too big, the total load factor (cf. **D25**) tends to decrease. To circumvent this situation a compaction process should be automatically or manually triggered, which may be executed concurrently with the other operations. When a compaction process is started, it activates the compaction indicator (cf. **D28**), which is disabled only at the end of this process. The latter employs a special lock type called *c-lock* (cf. **D27**), performed only at leaf nodes. A node n with a c-lock may be read by other processes. The leaf nodes to the right of a c-locked node may be read and updated by other processes.

According to the proposal of Miller et al. [MPRS79] the compaction process builds a new, compacted tree in a new disk area, that is, the tree is not built over the original one. Upon completion of the new tree, it is copied into the area which contains the old one, and the temporary space is released. Miller used a compaction algorithm based upon a depth-first search. Our method starts from the leaves constructing a level at a time; this favors the physical contiguity of the tree being built and the mentioned concurrencies.

The process consists of visiting the leftmost leaf node n_1, executing a c-lock in it. Subsequently, n_1 is traversed, and its elements are inserted into a new disk page P_{G_1}. Through n_1's pointer to its right immediate neighbour node (cf. **D4**, **D5**), n_2 is reached. A c-lock is introduced into n_2 and, if possible, its elements are inserted into P_{G_1} maintaining an initial load factor for the nodes established as a parameter. Or, if not possible, into a new disk page P_{G_2}, making P_{G_1} point to the former. This process continues until the last leaf node is reached, its elements being copied into the last new disk page P_{G_m}. The pages $P_{G_1}, \cdots, P_{G_m}$ become the leaf nodes of the new tree being built. At this moment, the process starts traversing these new leaf nodes, building their father nodes. The last element of P_{G_j} is introduced into its father, becoming the separator between P_{G_j} and $P_{G_{j+1}}$ $(1 \leq j \leq m-1)$. This is repeated for the other tree levels, until the new tree's root is built.

In the new tree, the nodes at each level are physically built immediately to the right of the nodes of the previous level. This provides for a certain physical contiguity, increasing the efficiency of search procedures. During the construction of the new tree's internal nodes, other processes may perform only read operations in the old tree, because at this moment everyone of the latter's leaf nodes contains a c-lock.

When an updater process reaches a c-locked leaf node it must be interrupted and inserted into the compaction queue (cf. **D30**). When the compaction process ends, every queue request is automatically triggered and the correspondent process is restarted at the new tree's root, trying to accomplish the desired operation.

The possible existence of an active compaction process has to produce alterations in reader and updater processes. Before starting their executions, they should verify the compaction indicator. If it is on, those processes should not execute catenations, splits and subdivisions, because the tree is being compacted anyhow. Insertions should only be performed in non-c-locked, non-complete leaf nodes or in those leaves whose fathers are not complete, guaranteeing no propagations.

4.3 Processes and Algorithms

The processes described below use algorithms which are presented under the form of decision tables (see, for instance [PHH71], divided into two subtables (conditions and actions) where "y", "n" and "-" stand for "yes", "no" and "indifferent", respectively. Numbers in a certain column of an action section indicate the order in which the actions corresponding to conditions which are all true for that column, are to be performed. Conditions and actions are expressed as "macros" defined after the tables. Some algorithms are described using macros defined in previous algorithms. Let $i_1, \cdots, i_m$ be the index values in node n.

4.3.1 Reader process or updater-reader subprocess

Traverses the tree from the root down to a leaf, calling the Search algorithm for each visited node n.

4.3.2 Updater process

Calls the Search algorithm until it reaches an appropriate leaf node n. Returns n with the information that it has *found* or *not found* the index value being searched and the correct position of insertion or deletion inside n. Then, it executes the Insertion or Deletion algorithm, depending on the type of process.

4.3.3 Compaction Process

Sets the compaction indicator to *on* and calls the Compaction algorithm.

4.3.4 Search algorithm

Input: The root node n of a subtree and the index value i being searched.

Conditions:

compaction-indicator?: does the compaction indicator contain a *true* value?

complete?: is node n complete?

compaction-indicator?	n	n	n	n	n	n	n	y	y	y
complete?	n	-	-	y	y	y	y	-	-	-
underflown-sons?	-	-	-	y	n	n	n	-	-	-
father's-load-greater?	-	-	-	-	y	y	n	-	-	-
may-subdivide?	-	-	-	-	y	n	-	-	-	-
is-leaf?	n	y	y	-	n	n	n	n	y	y
found?	-	y	n	-	-	-	-	-	y	n
catenate-sons				1						
subdivide					1					
split							1			
2-3-split						1				
go-on	1			2	2	2	2	1		
return-found		1							1	
return-not-found			1							1

underflown-sons?: does node n contain underflown sons n' and n'' either adjacent or separated by an unlocked, not underflown son?

father's-load-greater?: is the load factor at n's father level greater than f_s?

may-subdivide?: is it possible to subdivide n expanding this operation to take into account at most 2 adjacent brothers at each side of n?

is-leaf?: is n a leaf node?

found?: was i found in n?

Actions:

catenate-sons: if the underflown sons are adjacent, catenate them; if they are separated by a not underflown node, execute a 3-2-catenation.

subdivide: subdivide n's elements expanding this operation to take into account at most 2 adjacent brothers at each side of n.

split: split n.

2-3-split: perform a 2-3-split on n.

go-on: unlock n's father, locate the appropriate n's son, lock this son and call it n.

return-found: return, indicating (through a *found indicator*) that i was *found*, and giving its position j in n, $1 \leq j \leq m$.

return-not-found: return, indicating that i was *not found* and giving j, $0 \leq j \leq m$, where $j = 0$ means that i must eventually be inserted to the left of i_1, otherwise ($j > 0$) between i_j and i_{j+1}.

4.3.5 Insertion algorithm

Input: The node n, the position j in n returned by Search, the *found indicator* and the index value i to be inserted.

Conditions:

found-indicator?: has the *found indicator* the value *found*?

compaction-indicator?	-	n	n	n	n	y	y	y	-
found-indicator?	y	n	n	n	n	n	n	n	-
safe-for-insertion?	-	y	n	n	n	y	n	n	-
father-safe-for-insertion?	-	-	-	-	-	-	y	n	-
father's-load-greater?	-	-	y	y	n	-	-	-	-
may-subdivide?	-	-	y	n	-	-	-	-	-
is-there-c-lock?	-	-	-	-	-	n	n	n	y
insert-value		1	1	1	1	1	1		
subdivide			2						
split					2		2		
2-3-split				2					
message-1	1								
insert-compaction-queue								1	1
unlock		2	3	3	3	2	3	2	

safe-for-insertion?: is the leaf node n safe for insertion, that is, n contains less than $2K$ values?
father-safe-for-insertion?: is the father of leaf node n safe for insertion?
is-there-c-lock?: does node n contain a c-lock?

Actions:
insert-value: insert i into the leaf node n at position j.
message-1: emit message "the index value is already present in the tree".
insert-compaction-queue: insert the process into the compaction queue.
unlock: unlock n's father and n, in this order.

4.3.6 Deletion algorithm

Input: The node n, the position j in n returned by Search and the *found indicator*.

compaction-indicator?	-	n	n	y
found-indicator?	n	y	y	-
safe-for-deletion?	-	y	n	-
is-there-c-lock?	-	n	n	y
delete-value		1	1	
underflown-son-flag			2	
message-2	1			
insert-compaction-queue				1
unlock		2	3	

Conditions:
safe-for-deletion?: is the leaf node n safe for deletion, that is, does it contain more than K values?
Actions:
delete-value: remove the idex value from the leaf node n at position j.
underflown-son-flag: update the underflown son flag corresponding to n at its father, indicating that the leaf node n has become underflown.
message-2: emit message "the index value is not present in the tree".

4.3.7 Compaction Algorithm

Comment: This algorithm uses, for each leaf node, its pointer to the next node located at its right-hand side (cf. **D5**)

Input: The lelfmost leaf node n.

null-right-pointer?	n	y
c-lock	1	
create-new-leaf	2	
fetch-right-pointer	3	
repeat-table	4	
create-internal-nodes		1
copy-new-tree		2
turn-off-compaction-ind		3
trigger-compaction-queue		4

Conditions:
null-right-pointer?: is n's pointer to its immediate right neighbour null?

Actions:
c-lock: lock the next leaf node with a c-lock and call it n.
create-new-leaf: copy n's contents into the new page being built.
fetch-right-pointer: fetch n's pointer to its immediate right neighbour.
repeat-table: go to the beginning of this table.
create-internal-node: build the internal nodes of the new tree.
copy-new-tree: copy the new compacted tree into the space occupied by the old one.
turn-off-compaction-ind: set the compaction indicator to *off*.
trigger-compaction-queue: trigger the processes stored in the compaction queue.

4.4 Deadlock and Consistency

Our methods are deadlock-free, due to the particular lock types employed, that is, always involving two consecutive nodes along the hierarchical path, and following a certain ordering corresponding to the tree structure (top-down).

Consider the situation of figure 1 which illustrates a part of a tree during the execution of some process P_1, where n is the current node, n_f is its father and $f(P_1)$

the f-lock executed by P_1. If there exists a process P_2 locking a node n', where n' is n's son, then n is not the current node of P_2, otherwise, n would have been locked by P_2. Let n'', the current node of P_2, be the son of n'. We assert that the updates preformed by P_2 in n'', and consequently in n' do not propagate until n, because n' has, by construction, some available free space for future insertions (P_2 has already visited n'), avoiding the deadlock situation.

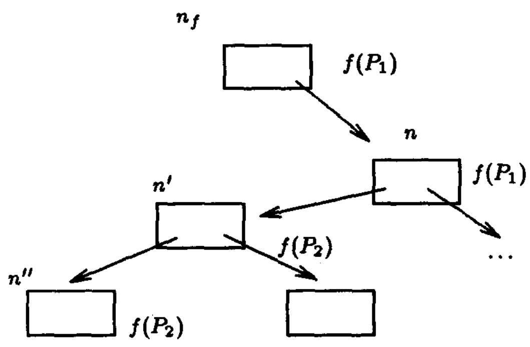

Figure 1: Processes P_1 and P_2 do not interefere

On the other hand, figure 2 illustrates the situation where n_1, one of n's brother, is locked by P_2, and P_1 is attempting to perform a split on n or a subdivision involving its elements. By construction, the current node of P_2 is a son n_1' of n_1, otherwise n's father n_f would not be locked by P_1. P_1 must wait until P_2 unlocks n_1. Notice that there are no deadlock problems, because n_1 contains free space for the insertion of a new element (P_2 has already visited n_1) and thus there is no insertion propagation until n_f (which is locked by P_1).

Notice that the attempt to split or subdivide was reduced to the verification of at most n's two immediately precedent or subsequent brothers, avoiding the possiblity of deadlock occurrence. A deadlock could happen had we visited n's "cousins" (sons of a brother of n_f).

Due to the utilization of c-locks and the initial construction of the compacted tree in a new area, the compaction process executed simultaneously with reader and updater processes does not cause dealock.

4.5 Variable-length Index Values

The use of front and rear index value compression (Wagner [Wag73]) and of Prefix B-trees (Bayer and Unterauer [BU77]) tend to diminish the tree's height. They require the use of variable-length index values.

Keller and Wiederhold [KW88] call the attention to the fact that Mond and Raz's method [MR85] cannot be used with variable-length index values. In this case,

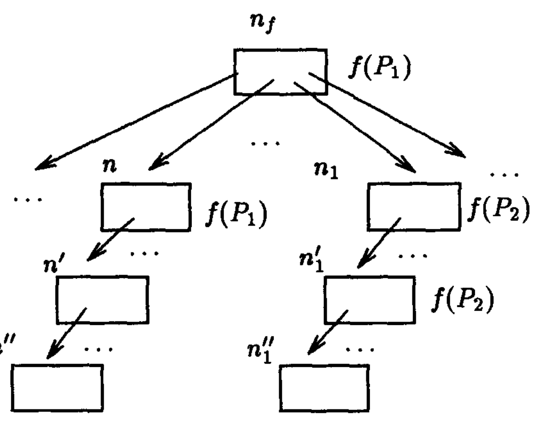

Figure 2: P_1 awaits P_2

the free space of each node may not be sufficient to accommodate a new value being inserted. In other words, it is difficult to characterize the fact that a node is or is not complete.

Our method may be extended to support these type of values. For this, we propose that the user specify the index value's maximum length in bytes or words, L_m. Let L_m be less than the half of a node's length. It is necessary to introduce some modifications in the present method.

The limits K and $2K$ are not characterized anymore. They have to be replaced by the length of a half-node and of a full node, respectively. Let us redefine the notion of a complete node.

D32 In a tree with variable-length index values and a maximum value length L_m, a node is *complete* when its free space is less than L_m.

D33 In a tree with variable-length index values a node is *uderflown* when its free space is greater than half of its total length.

The catenation, subdivision and split operations must be performed so that the nodes do not remain complete. When a node n is complete and has 2 underflown sons separated at most by 1 node, one should begin by trying to execute a catenation. If not possible or, if after its execution the node remains complete, one tries to perform a subdivision among its underflown sons and their brothers. Thus, one tries to replace the present separators by others of smaller length, eventually obtaining a non-complete n.

5 Conclusion and Future Research

Bayer and Schkolnick's [BS77], Miller and Snyder's [MS78] and Kwong and Wood's [KW80b, KW80a, KW82] solutions are characterized by the use of various lock types, with conversions among some of them.

Our method uses only one lock type, to perform read, insert, delete and update operations. It uses another lock type to execute a compaction process. The concurrency problem of running a compaction process together with other processes has not been found in the literature. Contrary to Bayer and Schkolnick's, Lehman and Yao's [LY81], Sagiv's [Sag86], Kwong and Wood's, and Souza and Carvalho's [SC92] solutions, in our method each executing process traverses the tree only once, except for some cases of processes that are being executed while the tree is being compacted.

The proposed solution applies to B^*- and B^+-trees and extends Mond and Raz's [MR85] method (see subsection 3.2). Contrary to their method, ours avoids the problem of producing cycles of catenation and split operations. It also performs 3-2-catenations as well as preemptive (eventually 2-3-) splits and subdivisions of complete nodes, but only when it is not possible to execute a delayed catenation of two of its (eventually non-adjacent) sons; the underflown son flags were introduced to improve performance in this case, reducing the number of nodes which have to be examined. We have also introduced a criterium to choose between subdivisions and splits, using a new parameter, the father's level load factor. Another item, not covered by them, is the use of variable-length index values permitting value compression and prefixing. The method favours a good total load factor, and provides a new, concurrent compaction technique. The latter guarantees physical contiguity of nodes in each level and of consecutive levels.

Without the compaction method, the concurrecy solution may also be employed to B-trees.

This work opens the following topics for future research. 1. Simulations to determine the ideal split factor limit (f_s) for each type of application, node load factor after compaction, and total load factor to trigger automatic compaction. 2. Extensions to avoid preemptive splits and delayed catenations in the nodes close to the root, because the probability of executing these operations in those nodes is very low in comparison to nodes close to the leaves. 3. Simulations to compare our method with other ones.

References

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[BS77] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[BT90] W. Boswell and A.L. Tharp. *Advances in Computing and Information*, volume 468 of *Lecture Notes in Computer Science*, chapter Alternatives to The B$^+$-Tree, pages 266–274. Springer, May 1990.

[BU77] R. Bayer and K. Unterauer. Prefix B-tree. *ACM Transactions on Databae Systems*, 2(1):12–26, March 1977.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw Hill Book Company, New York, 1990.

[Com79] D. Comer. The Ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.

[Dij68] E.W. Dijkstra. The structure of the multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[Ell80] C.S. Ellis. Concurrent search and insertion in 2-3-trees. *Acta Informatica*, 14(1):63–86, 1980.

[GS78] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annual Symposium Foundation of Computer Science*, pages 8–21, 1978.

[Knu73] D.E. Knuth. *The Art Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[KS86] H.F. Korth and A. Silberchatz. *Database System Concepts*. Mc. Graw Hill Inc., New York, 1986.

[KW80a] Y.S. Kwong and D. Wood. *In Proc. 4th International Symposium Programming*, volume 83 of *Lecture Notes in Computer Science*, chapter Concurrent Operations in Large Ordered Indexes, pages 208–221. Springer, 1980.

[KW80b] Y.S. Kwong and D. Wood. *In Proc. MFCS*, volume 88 of *Lecture Notes in Computer Science*, chapter Approaches to Concurrency in B-Trees, pages 402–413. Springer, 1980.

[KW82] Y.S Kwong and D. Wood. A new method of concurrency in B-trees. *IEEE Transactions on Software Engineering*, SE-8(3):211–222, May 1982.

[KW88] M.A. Keller and G. Wiederhold. Concurrent use of B-trees with variable-length entries. *Sigmod Records*, 17(2):89–90, June 1988.

[Lam77] L. Lamport. Concurrent reading and writing. *Communicdtions of the ACM*, 20(11):806–811, November 1977.

[LY81] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-tree. *ACM Transactions on Databae Systems*, 6(4):650–670, December 1981.

[MPRS79] E.R. Miller, N. Pippenger, A.L. Rosemberg, and L. Snyder. Optimal 2-3 trees. *Siam Journal of Computing*, 8(1):42–59, February 1979.

[MR85] Y. Mond and Y. Raz. Concurrency control in B^{+}-trees databases using preparatory operations. In *Proc. of the 11th International Conference on Very Large Database*, pages 331–334, Stockholm, 1985.

[MS78] E.R. Miller and L. Snyder. Multiple access to B-trees. In *Proc. Conference Information Sciences and Systems*, pages 400–407, John Hopkins University - Baltimore, March 1978.

[Nag90] S. Nagayama. Decision tables and the implementation of the I-M-E generator (in Portuguese). Master's thesis, Institute of Mathematics and Statistics, USP, São Paulo, 1990.

[Par77] J.R. Parr. An access method for concurrently sharing a B-tree index. Technical Report 36, University of Western Ontario, Departament of Computer Science, April 1977.

[PHH71] S.L. Pollack, H.J. Hicks, and W.J. Harrison. *Decision Tables: Theory and Practice.* John Wiley and Sons, New York, 1971.

[Sag86] Y. Sagiv. Concurrent operations on B^{*}-trees with overtacking. *Journal of Computer and Systems Science*, 33:275–296, 1986.

[Sam76] B. Samadi. B-trees in a system with multiple users. *Inf. Processing Letters*, 5(4):107–112, October 1976.

[SC92] R.M.F. Souza and O.S.F. Carvalho. Concurrency control of B-trees. *(in portuguese). IV- Simpósio Brasileiro de Arquitetura de Computadores - Processamento de Alto Desempenho (IV SBAC PAD)*, pages 397–412, 1992.

[Wag73] R.E. Wagner. Indexing design considerations. *IBM System Journal*, (4):351–367, 1973.

[Zis93] A. Zisman. The B-trees and an implementation proposal (in Portuguese). Master's thesis, Institute of Mathematics and Statistics, USP, São Paulo, 1993.