# A COMPARATIVE STUDY OF CONCURRENCY CONTROL METHODS IN B—TREES

Alexandros Biliris
Computer Science Department
Boston University
Boston, MA 02215

## ABSTRACT

In this paper, we described a number concurrency control methods for B-trees. Among them, a protocol which allows a number of independent search, insertion, and deletion processes, acting concurrently on a B-tree, to operate even when multiple insertions or deletions are pending. A number of properties have been also proposed to compare such protocols. The set of properties includes the number of lock types being used, degree of data sharing, number of processes that are permitted to access a node simultaneously, how fast a reader can reach the leaf node, number of nodes locked by the three processes during their operation, number of nodes being accessed sequentially, and the number an updater passes through the tree. Based on these properties, seven protocols are compared and discussed.

## 1. INTRODUCTION

In many computing systems, there is a need for sharing data and resources among processes. The problem of synchronizing such processes that compete with one another for shared objects is called, in the database literature, the concurrency control problem. Given that the system has not failed, the concurrency control mechanism must ensure that consistency of objects is preserved and that all processes will complete their operations in finite time. [KOHL81].

An important part of any database is its index mechanism which speeds up the retrieval process by directing the searcher to a small portion of the database containing the desired item. Hashing and its variants provide a mean for data accessing. On the other hand, multilevel (tree) structures in which each index at some level points to another index in the level below until the actual data have been reached, have became extremely popular. While no single scheme can be best for all applications a particular multilevel structure, called B-tree [BAYE72], has became the most widely used technique for storing large files of information, especially on external storage devices. Knuth [KNUT73] provides a survey of the basic techniques and Comer [COME79] discusses and analyzes the B-trees and its variants.

Although, simultaneous accesses to the same database component may be rare, there is a high probability that indexes will be repeatedly accessed, and therefore a great deal of the time spent during the database access is attributed to searching through indexes. Serializing access to the indexes of a data base may create an undesirable bottleneck and degrade the entire system. Since B-trees are the most widely used access aids, for both primary and secondary indexing, maximizing concurrency on them is the most contributing factor in the overall degree of concurrency.

The rest of the paper is organized as follows. Section 2 briefly presents some representative algorithms for concurrency control on B-trees. Section 3 describes the mU protocol and in section 4 twelve properties for comparison of similar algorithms are proposed. Finally, section 5 summarizes this work.

### 1.1. DEFINITIONS

**B-tree Definition:** *A B-tree of order m is a balanced tree which has the following properties.*

- *Every node, except for the root has between m and 2m children.*
- *The root has between two and 2m children.*

Proc. of the Aegean Workshop on VLSI and Algorithms,
Springer-Verlag Lecture Notes in Computer Science,
Vol. 227, July 1986, pp. 305-316

- *A nonleaf node consists of s pointers to its children ($P_0, P_1, \ldots, P_{s-1}$) and s-1 keys ($K_1, K_2, \ldots, K_{s-1}$) arranged in such way that for every key K in the subtree pointed to by $P_i$, the following relationships hold:*

$$K < K_1, i=0$$
$$K_i \leqslant K < K_{i+1}, 0<i<s-1$$
$$K_i \leqslant K, i=s-1$$

- *A leaf node with s children contains s keys.*

We assume that each node is organized sequentially, and that the set of all keys appears in the leaves. In case that $P_i$ belongs to a leaf node it may point to records keeping actual data associated with the key $K_i$. Data associated with each key are of no interest in the following discussion and are omitted. An example of a three level B-tree of order two is shown in Figure 1.

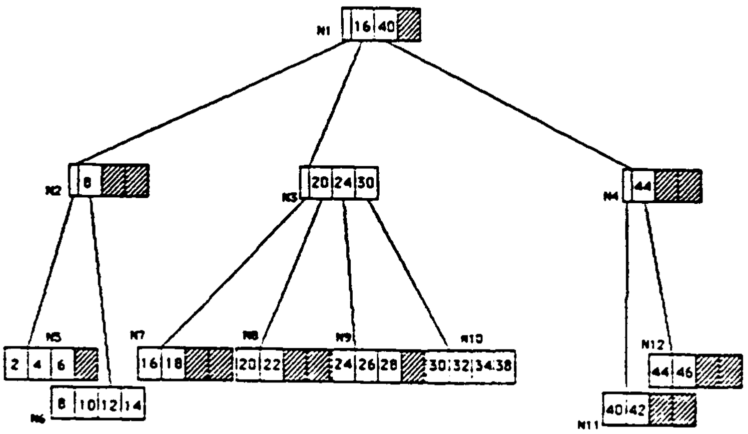


Fig 1. A B-tree structure.

The operations to be performed, concurrently, on a B-tree structure will be of three kinds: search for a key, insert a (key, pointer) pair, and delete a pair from the tree. The processes that perform these operations are called read process (RP), insertion process (IP) and deletion process (DP), respectively. IPs and DPs are collectively called *updaters*.

An updater goes through the following two phases. *Searching*: it reads the tree starting from the root to determine the leaf node for a particular key. *Restructuring*: Once the leaf node is found, it adds (or removes) the key and restructures the tree if necessary. It is exactly this restructuring phase that creates most of the problems in a concurrent environment.

Most of the solutions to the problem of supporting concurrent operations in B-trees make use of the following observations. There exists a node which is the root of a subtree above which no change in data and structure due to an update can propagate. This node is called a *safe node* [BAYE77b]. A node consisting of less than 2m children is called *insertion-safe* (i-safe), because a new key can be added without forcing a split. A node with more than m children is called *deletion-safe* (d-safe), because a key can be deleted without going below the m-children minimum. The portion of the access path from the deepest safe node to a leaf is called the *scope* of the Updater. We also refer to the child of the Updater's deepest safe node on its scope, as the *highest unsafe node*.

The protocols compared in this paper are defined using the Conditional Compatibility and Convertibility Graph (CCCG), [BILI85b].

**CCCG Definition**: *The CCCG is a weighted directed graph in which vertices represent lock types and edges with their associated weight specify the relation among these locks on a node, as follows:*

*if a and b are vertices (locks) then*

*a (solid edge. W) b <=>*

*A process may place an a-lock on a b-locked node iff condition W is satisfied. Absence of the condition W, implies that locks a and b are fully (that is always) compatible.*

*a (broken edge) b <=>*

*A process holding an a-lock on a node may convert it into a b-lock.*

## 2. RELATED RESEARCH

We may very easily show that taking no precautions against the anomalies of concurrency leads to incorrect (non serializable) results and to an inconsistent state of the index itself. The execution of a set of processes is called *serializable*. iff it produces the same effects as some serial execution of the same processes [BERN79. PAPA79].

The first algorithm to this problem was offered by Samadi. [SAMA76]. who uses semaphores to exclusively lock a node. Fig 2a. That is the only lock type that may be used by each process.

Three algorithms have been proposed by Bayer and Schkolnick [BAYE77b]. Their *first* protocol substantially improved the previous method by introducing separate lock types for readers (r-locks) and updaters (e-locks). Fig 2b. This. takes advantage of the fact that readers do not modify any node on the tree and therefore multiple read access to a node may be allowed. Their *second* solution. requires updaters to proceed down the tree as if they were readers using r-locks. until they reach a leaf node. They e-lock this node and examine its safeness: if it is not safe. they release all the locks and repeat access to the tree. this time using the protocol of their first solution. The *third* solution requires updaters. to use read-compatible locks. namely w-locks. during the searching phase. In case the leaf node is not safe the updaters should convert. from top to bottom. their w-locks to e-locks. The CCCG of this protocol is shown in Fig 2c.

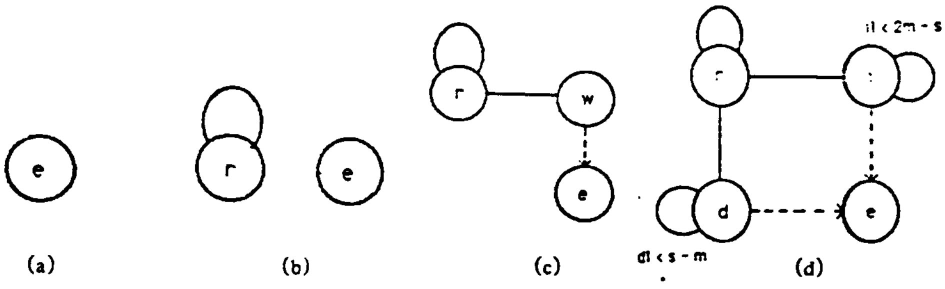


Fig 2. CCCG of various protocols.

In the Kwong and Wood scheme. [KWON82]. the relations among locks remain the same. Fig 2c. However. the modifications imposed by a key insertion on a full node are always done on a new not-yet-in-tree node (side branch) which has as a result to further delay the placement of the exclusive locks because of the update.

Guibas and Sedgewick proposed another solution in which an insertion process. as it goes down the tree. splits "almost full" nodes to avoid a bottom-up restructuring of the tree [GUIB78]. and Miller and Snyder [MILL78] use a kind of queue mechanism to help Readers to flow over locked regions of the tree.

Lehman and Yao presented a solution in which Readers use no locks and insertion processes place their exclusive locks in a bottom-up manner [LEHM81]. This protocol is based on the assumption that each process works on each node after it is fetched on its private space (no data sharing) and requires a slight modification of the usual B-tree structure. named $B^{link}$-tree in their paper. In addition. in this protocol. a node of fewer than m pairs (zero included) is permitted: in case of storage underutilization the entire tree is locked and re-organization of the tree is performed.

Kung and Lehman's work on binary trees. [KUNG80]. created a new class of concurrency control methods called optimistic. These methods assume that conflicting operations are the exceptional case and therefore no locking is needed. Instead. correctness is guaranteed in the validation step. performed at the end of each operation. in which a test is done to verify that nodes accessed by one process have not been modified by others. Kersten and Tebra. [KERS84]. have extended this idea for B-trees. and Lausen develop an intergrated concurrency control mechanism that switches from the "optimistic" method to locking (Samadi's method) depending on the number of the conflicting operations. [LAUS84].

## 3. THE mU PROTOCOL

The above locking algorithms characterize the safeness of a node by a single true-false value, making minimal use of the fact that a node may be 'very' or 'a little bit' safe at some particular instance for a particular kind of process. Since, no distinction is made between insertion and deletion processes, no advantage is taken from the fact that the first adds some data to the node while the second removes some. The mU protocol permits as many insertion (deletion) processes to place, concurrently, a lock on a node as will not, by their collective action, require the node to be split (merged).

The protocol uses four types of locks, viz. read-lock, insert-lock, delete-lock, and exclusive-lock (r-, i-, d-, and e-lock respectively). Readers use r-locks, IPs i-locks, DPs d-locks. Their CCCG is given in Figure 1. We use the variables rl, il, and dl to keep the number of r-, i-, and d-locks respectively currently placed on a node; s represents the number of children in the node. An important point that can not be seen from the CCCG is that a request to i- or d-lock a node implies also a request for an r-lock. When the reading is terminated the Updater must explicitly r-unlock the node. Also, i- or d- locks do not give the right to an Updater to modify a node. They are used as reservations of free slots in the node. Should an actual modification be required, these locks should be converted to e-locks, a technique used also in [BAYE78, KWON82].

A summary of the basic lock relations is following.

F1. A request of an RP to r-lock a node is granted when this node is not e-locked and no e-lock request is pending for that node.

F2. A request of an IP to i-lock a node is granted when this node is not e- or d-locked and

$$il < 2m-s \text{ or } il=0$$

F3. A request of a DP to d-lock a node is granted when this node is not e- or i-locked and

$$dl < s-m \text{ or } dl=0$$

F4. Request for i- or d-lock implies also request for r-lock.

F5. Only a process that already holds an i- or d-lock may convert it to e-lock; the lock will be assigned when all r-locks have been removed for that node.

It is noteworthy that in this protocol *compatibility relations among locks are not static*. The lock assignment on a node does not depend exclusively on the lock type, but also on the *status of the node* and the *number of processes* acting currently on that particular node. Many authors have observed that using semantic knowledge about the object that a process manipulates, or about the operations that a process performs on an object can increase concurrency. [BERN78, GARC83, HSU83, SPEC83, SPEC85].

The process that honors or denies the right of a process to access a particular node, is called *Lock Controller* (LC). It is assumed that lock requests of the same kind are treated by the LC in a first-in-first-out fashion and that processes being run on processors with comparable speeds.

### 3.1. SEARCHING

A Read process searches down the tree, using r-locks only, reporting success or failure. Locks are placed and released according to the locking-coupling technique [BAYE77b], in which an RP on its path to the leaf unlocks a node only after it has locked its child. The locking-coupling technique guarantees that there is (at least) one node in the tree that is not currently updated.

### 3.2. INSERTION

An Insertion Process uses i-locks on its passage to a leaf node. Since i-lock implies r-lock it can read the node with no other control. In each node it checks for the node's safeness, and if the node is safe, it i-unlocks all the ancestors. The outline of the IP's steps from root through the leaf node is shown in Fig 3a. On reaching the leaf node the IP's scope will be i-locked, but still free for other IPs and RPs. If the leaf node is safe, it e-locks it, adds the new key, unlocks the node and so its task is terminated. If the leaf node is full then the tree will be re-organized. We do that using the side branching technique which was first

```
i-lock(root);
current := root;
while not current is a leaf node loop
        find appropriate child of current;
        r-unlock(current);
        current := appropriate child;
        i-lock(current);
        if current is i-safe then
            i-unlock ancestors of current ;
        end if ;
end loop .
```

(a)

```
current := the full of pairs leaf node;
while current is not i-safe loop
        get a new node;
        add appropriate half of current into branch;
        add the new (k, p) on branch;
        current := father of current on IP's
                access path;
end loop ;
```

(b)

Fig 3. (a) IP steps from root to the leaf; (b) IP steps from leaf to the deepest safe.

reported by Kung and Lehman for binary trees [KUNG80], and Kwong and Wood for B-trees [KWON82].

### 3.2.1. Restructuring Phase

The IP goes in the bottom up direction creating left or right side branches as follows. ("left" and "right" declare the direction of the branch with respect to current node). Let C be the full node in which IP wants to add a new (k, p) pair. The IP procedure scans the current node (C) to find the "position" j where the key (k) should be added. It then gets a new node (B) from the free storage and writes the left, if $j \leqslant m$, or the right half of C, if $j > m$, into B. In either case the branch node (B) contains $m-1$ pairs. Since a new node has been created, a new pair should be added on C's parent and the overflow propagates until the deepest safe node. The outline of the IP's steps from the leaf to the deepest safe node is shown in Fig 3b. On reaching the deepest safe node the IP will have to add on that node a pointer to the newly created node one level below. The remaining task is to remove redundant halves from every node on its scope (e-locking one node at a time).

The highest unsafe node needs special care in the mU protocol. Consider the situation of Figure 1, where node N1 and N3 are the deepest safe and highest unsafe node respectively of an IP, called IP1. Assume, also an other IP, called IP2, which holding an i-lock on N1 (before IP1 converts its i-lock to e-lock), finds that its path passes through N3. IP2's request to i-lock node N3 is not granted (because of F2) and it waits for IP1 to free node N3. However, when IP1 unlocks this node, half of N3's (K,P) pairs have been removed to the newly created N3's branch. This raises the possibility of an incorrect path selection by IP2. For this reason, IP1 provides information about the side branch by means of an auxiliary pair (LINKK, LINKP), hereinafter referred to as the LINK pair, defined as follows, [BILI85a]:

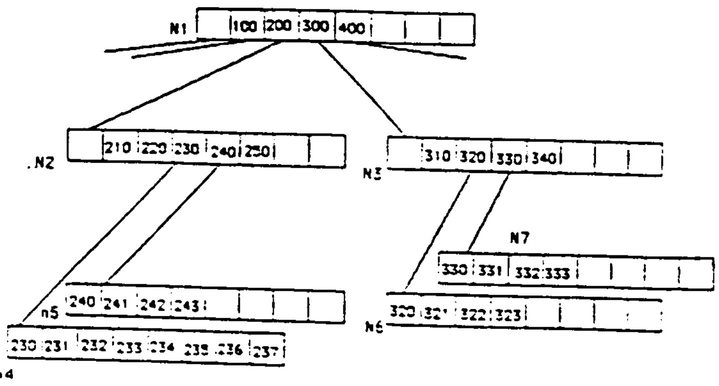
**LINK pair definition**: LINK is a pair (LINKK, LINKP) of a key and a pointer to a node which is set on the highest unsafe node (C) of an IP's scope when a left or right branch (B) of C has been created, such that:

- LINKP points to B, regardless of B's direction.
- LINKK is the key added to C's parent because of the splitting of node C (it may be the separator of C itself or its branch node).

That is, IP1 removes half of C's pairs and places (LINKK, LINKP) on the rightmost pair ($K_{2m-1}$, $P_{2m-1}$). Then it unlocks C and waits for IP2 to read it. When this is done it sets $P_{2m-1}$ to NIL. A NIL pointer at this position indicates a node with no branch. Since a FIFO discipline is assumed for lock requests of the same kind, IP1 will be able to place again an i-lock on this node only after all other IPs have already placed their i-locks. Proofs for the above statements can be found in [BILI85b]).

The example of Figure 4b shows the operations of four Insertion processes acting on the B-tree of Figure 4a using the mU protocol. To simplify the flow, it is assumed that the Lock Controller assigns lock requests from left to right. It is also assumed that other operations besides lock assignments are of no interest and they are treated on a single step (and briefly).

## 3.3. DELETION



(a)

| time | insert 238 | insert 239 | insert 328 | insert 338 |
|---|---|---|---|---|
| 1 | i-l(N1),G | i-l(N1),G | i-l(N1),G | i-l(N1),W |
| 2 | read N1 | read N1 | read N1 | W |
| 3 | r-ul(N1) | r-ul(N1) | r-ul(N1) | W |
| 4 | i-l(N2),G | i-l(N2),G | i-l(N3),G | W |
| 5 | i-ul(N1) | i-ul(N1) | i-ul(N1) | G |
| 6 | read N2 | read N2 | read N3 | read N1 |
| 7 | r-ul(N2) | r-ul(N2) | r-ul(N3) | r-ul(N1) |
| 8 | i-l(N4),G | i-l(N4),W | i-l(N6),G | i-l(N3),G |
| 9 | read N4 | W | i-ul(N3) | i-ul(N1) |
| 10 | r-ul(N4) | W | read N6 | read N3 |
| 11 | N8 right branch of N4; transfer N4's right half on N8; add 238 on N8 | W | r-ul(N6) | r-ul(N3) |
| 12 | e-l(N2),G | W | e-l(N6),G | i-l(N7),G |
| 13 | add N8 on N2 | W | add 328 on N6 | i-ul(N3) |
| 14 | e-l(N4),G | W | e-ul(N6) | read N7 |
| 15 | set LINK on N3 | W | | r-ul(N7) |
| 16 | e-ul(N4) | G | | e-l(N7),G |
| 17 | i-l(N4),G | i-ul(N2) | | add 338 on N7 |
| 18 | r-ul(N4) | read N4 | | e-ul(N7) |
| 19 | e-l(N4),W | i-l(N8),G | | |
| 20 | W | r-ul(N4) | | |
| 21 | G | i-ul(N4) | | |
| 22 | reset LINK | read N8 | | |
| 23 | e-ul(N2) | r-ul(N8) | | |
| 24 | e-ul(N4) | e-l(N8),G | | |
| 25 | | add 239 on N8 | | |
| 26 | | e-ul(N8) | | |

where,

i-l(N) : i-lock node N: e-l(N) : convert i- to e-lock
x-ul(N): x unlock node N, where x: r, i or e
W : lock not granted, process should wait; G : lock granted

(b)

Figure 4: An example of four concurrent Insertions

The outline of a Deletion Process steps from root to leaf node is the same with an IP, Fig 3a, except that DPs use d-locks on their passage to the leaf node.

### 3.3.1. Restructuring Phase

On reaching the leaf node, DP checks for its safeness. In case the leaf node is d-safe the DP removes the key and unlocks that node. Its task is finished. Otherwise it begins the tree re-organization.

On each node (C), the sibling node (B) is checked. If B has exactly m children then merging is performed. Node B is locked and all pairs from C (except the one that has to be removed) are read into B together with the separating key on the parent. Node C now becomes redundant but it remains intact for Readers. Merging results in a new (K,P) that should be deleted on the parent node and the problem propagates to the next (upper) level.

If C's sibling is d-safe rotation is performed. Nodes B,C and their parent (A) are e-locked and the keys on C and B together with the separating key on A are rotated. Once a rotation is performed on C, the DP's task is to remove the remaining d-locks on C's ancestors.

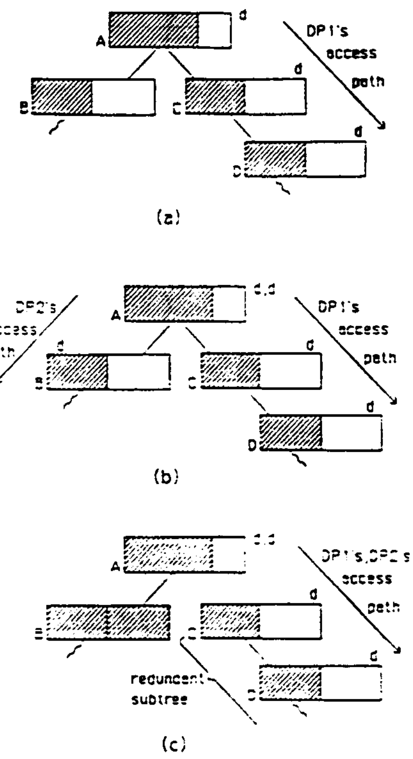


Fig 5: (a) initial configuration, (b) deadlock, (c) lost path problem

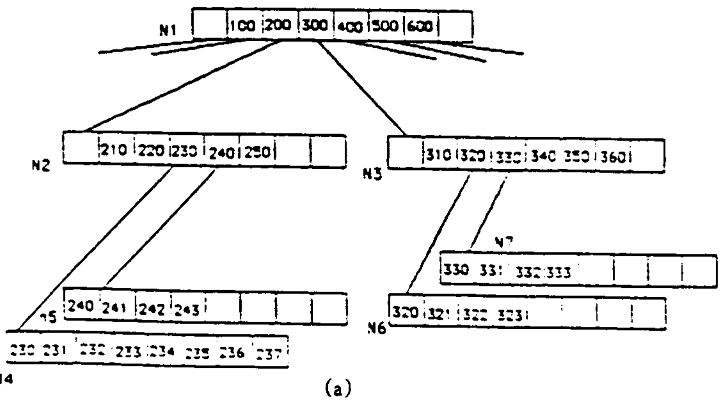
If C's sibling is not d-safe the DP goes up the tree (by node merging) until the deepest safe node is reached (node A in Figure 5a). DP1 wants to d-lock B to see if merging or rotation is appropriate. This action, however, can create deadlock problems, and also leave the tree on a non consistent state. These problems are examined in the next two paragraphs.

*Deadlock Problem*: Consider another DP (DP2) for which nodes A and B are part of its access path, and node A is its deepest safe node (Figure 5b). DP2 d-locks its scope and enters the restructuring phase, together with DP1 on A,C,... path. DP1, on reaching A, requests to d-lock node B, a request which can not be granted since B is unsafe and d-locked by DP2 (rule F3). DP2, on the other hand, also reaches node A and request to d-lock C. Therefore each DP is waiting for the other one to unlock the desired node (deadlock).

*The Lost Path Problem*: This problem is raised when DP2's path passes through C instead of B (Figure 5c). Then, DP2 tries to d-lock a node which will be removed from the tree structure by DP1.

For the above reasons all DPs are forced to operate in a different way on the highest unsafe node(C). They remove the appropriate (k,p) from C and immediately unlock that node. Merging or rotation will be done collectively in a following step by one DP only. This DP is the process that reduced the number of children of C from m to m-1. It might be said that it is *responsible* for that node.

The example of Figure 6a shows an execution of four Deletion processes acting on the B-tree of Figure 6b using the mU protocol. Again, it is assumed that the Lock Controller assigns lock requests from left to right, and that other operations besides lock assignments are of no interest and they are treated on a single step (and briefly).



(a)

| time | delete 322 | delete 323 | delete 333 | delete 234 |
|---|---|---|---|---|
| 1 | d-l(N1),G | d-l(N1),G | d-l(N1),G | d-l(n1),W |
| 2 | read N1 | read N1 | read N1 | W |
| 3 | r-ul(N1) | r-ul(N1) | r-ul(N1) | W |
| 4 | d-l(N3),G | d-l(N3),G | d-l(N3),G | W |
| 5 | d-ul(N1) | d-ul(N1) | d-ul(N1) | G |
| 6 | read N3 | read N3 | read N3 | read N1 |
| 7 | r-ul(N3) | r-ul(N3) | r-ul(N3) | r-ul(N1) |
| 8 | d-l(N6),G | d-l(N6),W | d-l(N7),G | d-l(N2),G |
| 9 | r-ul(N6) | W | r-ul(N7) | d-ul(N1) |
| 10 | e-l(N6),G | W | e-l(N7),G | read N2 |
| 11 | delete 322 | W | delete 333 | r-ul(N2) |
| 12 | D322 i responsible for node N6 | W | D333 is responsible for node N7 | d-l(N4) |
| 13 | e-ul(N6) | G | e-ul(N7) | d-ul(N2) |
| 14 | e-l(N3),G | r-ul(N6) | e-l(N3),W | r-ul(N4) |
| 15 | d-l(N6),W | e-l(N6),G | W | e-l(N4),G |
| 16 | W | delete 323 D323 is NOT responsible for node N6 | W | delete 234 |
| 17 | W | e-ul(N6) | W | e-ul(N4) |
| 18 | G | d-ul(N3) | W | |
| 19 | r-ul(N6) | | W | |
| 20 | e-l(N6),G | | W | |
| 21 | d-l(N7),G | | W | |
| 22 | r-ul(N7) | | W | |
| 23 | e-l(N7),G | | W | |
| 24 | merge N6 into N7 | | W | |
| 25 | remove N6 from N3 | | W | |
| 26 | e-ul(N3) | | G | |
| 27 | e-ul(N7) | | d-l(N7),G | |
| 28 | e-ul(N6) | | r-ul(N7) | |
| 29 | free N6 | | e-l(N7),G | |
| 30 | | | N7 has more than 4 pairs | |
| 31 | | | e-ul(N3) | |
| 32 | | | e-ul(N7) | |

where,

d-l(n) : d-lock node n; e-l(n) : convert d- to e-lock on n
x-ul(n): x unlock node n, where x is d, r, or e
G: lock granted; W: lock not granted, the process should wait

(b)

**Figure 5**: An example of four concurrent Deletions.

## 4. COMPARISON OF SOLUTIONS

Unfortunately, there is no universal approved measure of goodness of solution. In addition, most papers (including ours) in this field use terms that are intuitively understood, not precisely defined. Terms like "degree of concurrency" and "protocol simplicity" belong to to this category.

Many authors attempt to define the effective level of concurrency as the number of transactions doing useful work. (see i.e. [FRAN85]). The term "useful", however, may have a different interpretation for different concurrent algorithms. It could be the case that some operations performed by some processes using one protocol don't need to be performed on another. Consider, for example, the case where *restarts* are employed in some "optimistic" protocols as conflict resolution method and that the majority of the processes are forced to restart. Clearly, knowing that the majority of the process are doing "useful" work is not informative. Better measures such as, for example, the number of processes completing their work per unit of time (throughput) are very difficult to evaluate analytically.

Yet, there is a place for comparison of different solutions, examining some of their characteristics and getting a feeling of the protocol quality. Working in this direction, twelve protocol properties that a comparison should be based upon are proposed (a similar effort is done in [KWON82]). The set of properties includes the number of lock types being used, degree of data sharing, number of processes that are permitted to access a node simultaneously, how fast an RP can reach the leaf node, number of nodes locked by the three processes during their operation, number of nodes being accessed sequentially, and the number an updater passes through the tree.

These properties are used to construct Table 1, which compares six already proposed protocols with ours. It should be pointed out that it is the set of all these properties that give a feeling of the protocol quality rather than each individual property isolated from each other.

## 4.1. Number of Lock Types

The number of lock types being used gives a measure of the protocol complexity. The more lock types are used the more complex the protocol is expected.

Table 1: Comparison of solutions

| SOLUTIONS | S1 | S2 | S3 | S4 | S5 | S6 | S7 |
|---|---|---|---|---|---|---|---|
| PROPERTIES | Samadi | Bayer Schkolnick solution 1 | Bayer Schkolnick solution 2 | Bayer Schkolnick solution 3 | Lehman Yao | Side-branching | The mU protocol |
| P1. Number of lock types | 1 | 2 | 2 | 3 | 1 | 3 | 4 |
| P2. Special memory arrangement | no | no | no | no | yes, no sharing of memory | no | no |
| P3. Maximum number of RPs accessing the same node | 1 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| P4. Maximum number of IPs accessing the same node | 1 | 1 | 1 | 1 | ∞ | 1 | m |
| P5. Maximum number of DPs accessing the same node | 1 | 1 | 1 | 1 | X[1] | 1 | m |
| P6. RPs in the Updaters scope during the searching phase | no | no | no | yes | yes | yes | yes |
| P7. RPs in the Updaters scope during restructuring phase | no | no | no | no | yes | yes | yes |
| P8. Maximum number of nodes locked by RPs | 2 | 2 | 2 | 2 | 0 | 2 | 2 |
| P9. Maximum number of nodes locked by IPs | the whole scope | the whole scope | 1, or the whole scope | the whole scope | 3 | 2 | 2 |
| P10. Maximum number of nodes locked by DPs | the whole scope | the whole scope | 1, or the whole scope | the whole scope | X[1] | 3 | 3 |
| P11. Maximum number of nodes being accessed sequentially | 1 | 1 | 1 | 1 | ∞ | 1 | IPs: 2 RPs,DPs:1 |
| P12. Number of passages through the tree | ↓↑ | ↓↑ | ↓↓↑ | ↓↓↑ | ↓↑ | ↓↑↓ | ↓↑↓ |

X[1]: In this solution, a node may have less than m children and therefore a DP never restructures the tree.

For instance, solution S1 requires a fairly simple lock controller (a semaphore) that makes it attractive. However the concurrency that is permitted in this protocol is at a very low level. On the other hand, the mU protocol increases the degree of concurrency because, mainly, it distinguishes the locks used by IPs and DPs. Moreover the operations that are performed for these locks by the Lock Controller are trivial (additions and subtractions). Indeed, a recent work by Carey, [CARE83], indicates that concurrency control overhead is insignificant compared to other factors.

## 4.2. Special Memory Arrangement

Memory requirements give a measure of the protocol applicability under different logical configuration of the primary memory. Solutions which require special memory arrangement have more restrictive use than others that do not. Consider, for instance, solution S5 which does not permit data sharing at all. It requires that each process has part of the main memory for its own. In fact, this is a key issue in this solution and a major reason (together with a link pointer on each node to the right sibling) of how this protocol works using one lock type. For example, a Reader need not lock a node before reading, because it may read it on its own memory regardless if this node is updated at the same time in an other memory segment by an Updater. In systems where the main memory is shared among processes (use of cache memory for all the processes running on the system is an example), S5 can not be used or extra effort should be made to bridge the gap between the assumed and actual storage systems. The remaining solutions in the table , on the other hand, do not impose such a restriction.

## 4.3. Maximum Number of Processes Accessing Concurrently the Same Node

The maximum number of processes accessing concurrently the same node is a doubtless factor of the protocol's degree of concurrency. Solutions that permit a higher number of processes to operate on the same node at the same time are faster. The solution presented in this paper compares favorably with earlier solutions in that an Updater who visits a node of s children permits up to 2m-s-1 Updaters of the same kind to access that same node. Consider, for instance, the case of the deepest safe node. In all earlier solutions, except S5, this node is locked by the Updater who first happened to visit it and no other Updaters have access to that node until: (1) the leaf node is reached, (2) the key is added or deleted and, (3) the restructuring phase (get new nodes, or merge two halves) is performed. The mU protocol keeps this node free; therefore nodes belonging to the s-1 subtrees of the deepest safe node also remain free. It's easy to see what it is gained in concurrency, in the case where the deepest safe node is at a relatively higher level in the B-tree (closer to the root). Naturally, nodes in higher levels need be split (likewise deleted) infrequently compared to the number of insertions (deletions) performed. However, this statement implies that most of the time the higher level nodes are safe. In this case the mU protocol is still more efficient than previous solutions since it doesn't delay an Updater from accessing a node, waiting for an other Updater of the same kind holding already a lock on that node to find the next (deeper) safe node.

Solution S5 permits an infinite number of Updaters to visit a node at the same time. This solution, however, does not require each node to have at least m children. That is, a DP e-locks the leaf node, removes the appropriate pair, unlocks the node, and its task is terminated regardless of the number of pairs in this node. Consequently, repeated deletions may lead to space underutilization. For applications, however, where deleted pairs are evenly distributed on the tree, or for applications in which speed is of prime concern while space utilization and memory sharing is not, this may be the protocol of choice.

## 4.4. Reader Access During the Searching and Restructuring Phase

In general, RPs represent the majority of the processes entering the tree. Therefore it's important to see how fast a RP can reach the leaf node, in the presence of Updaters (the interaction among Readers is examined in P3). P7 is a very important issue since the restructuring phase of an Updater's life could take time, especially when the deepest safe node is in higher levels.

## 4.5. Maximum Number of Nodes Locked by an RP, IP , or DP

All protocols using more than one lock type have at least a read (r-) and an exclusive (e-) lock. Whatever is the CCCG of a particular protocol these two locks are incompatible (they can not coexist on the same node). Therefore, it is worth knowing how many nodes are locked at a time by each process. It might be suggested that it is a measure of the restrictions' spreading set by a process. Clearly, solutions S5, S6 and S7 dominate over the first four. It is noteworthy that the 'yes' answer in P7, for these three protocols, is an immediate consequence of P9 and P10. It is the fact that an Updater locks two or three nodes, and not its entire scope, which permits Readers to operate on this part of the tree.

## 4.6. Maximum Number of Nodes Accessed Sequentially by the Processes

Some solutions (S5, S7) use a link pointer to a sibling node as another means to reach that node. In S5 the link (to the right sibling) is permanent and part of the tree structure. It is used by all processes trying to reach the appropriate leaf node. As is pointed out in [LEHM81], this situation could force a process to run indefinitely having to follow link pointers created by other processes. Although this is extremely unlikely to happen in a practical implementation, it is still an undesirable factor for the searching time.

The mU protocol also uses a link to the left or right sibling. However, this link is not part of the tree structure. It is set by the IPs on the highest unsafe node, to address the newly created sibling node and for a time period sufficient for other processes acting on that node to complete their operation. In addition only IPs should be aware of this link. Readers and DPs need know nothing about it, and therefore they always access one node sequentially.

## 4.7. Number of Updater Passes Through the Tree

The first down arrow indicates that an Updater goes first through the whole tree. The other arrows indicate the passage from the leaf to the deepest safe node (up arrow) and vise versa (down arrow). This property is a factor of the Updaters' speed: it is desirable to keep the number of passes as small as possible. Without any intention to discharge S6 and S7 from being three-pass protocols, it may be said that it is the existence of the third pass that enables S6 and S7 to permit Readers to operate most of the time in the Updaters' scope.

## 5. CONCLUSIONS

In this paper, we first defined the Conditional Compatibility and Convertibility Graph (CCCG), which is a directed graph defining relations among locks that are not static. Then, we described few concurrency control methods for B-trees. Among them, it's the mU protocol which provides high concurrency among Read, Insertion or Deletion processes operating concurrently on a node. This is achieved by using three separate lock types for each of the above processes.

A number of properties have been also proposed to compare such protocols. The set of properties includes the number of lock types being used, degree of data sharing, number of processes that are permitted to access a node simultaneously, how fast a reader can reach the leaf node, number of nodes locked by the three processes during their operation, number of nodes being accessed sequentially, and the number an updater passes through the tree. Based on these properties, the seven protocols are compared. Although no precisely defined metric for the overall quality of a protocol exists, it was argued that the mU protocol permits more concurrency, without having to sacrifice space or to impose special memory requirements. The validity of the above argument has been also demonstrated by developing a simulation model, [BILI86].

## REFERENCES

[BAYE72] Bayer R., McCreight E., Organization and maintenance of large ordered indexes. *Acta Informatica*, Vol. 1, 1972.

[BAYE77b]Bayer R., Schkolnick M., Concurrency of operations on B-trees. *Acta Informatica*, Vol. 9, 1977.

[BERN78] Bernstein P. A., Rothnie J. B. Jr., Goodman N., Papadimitriou C. A., The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case). *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, May 1978.

[BERN79] Bernstein P., Shipman D., Wong W., Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, Vol 5, No 3, May 1979.

[BILI85a] Biliris A., Feldman M. B., Concurrent Insertions in Multiway Dynamic Structures, *Proceedings of the 19th Annual Conference on Information Sciences and Systems*, March 1985.

[BILI85b] Biliris A., *Concurrency Control on Database Indexes: Design and Evaluation*, Ph.D. Thesis, EECS Department, George Washington University, 1985. Available also as TR 85014-85015, Computer Science Dept., Boston University.

[BILI86] Biliris A., Feldman M. B., Using the Ada Tasking Model in Evaluating Concurrency Control Protocols in Databases, submitted to *IEEE, Transactions on Software Engineering*.

[CARE83] Carey M. J., *Modeling and Evaluation of Database Concurrency Control Algorithms*, Ph.D. Thesis, Computer Science Department, University of California, Berkley, 1983

[COME79]Comer D., The ubiquitous B-Tree. *ACM Computing Surveys*, Vol. 11, No. 2, and Vol. 11, No. 4, 1979.

[FRAN85] Franaszek P., Robinson J. T., Limitations of Concurrency in Transaction Processing. *ACM Transactions on Database Systems*, Vol 10, No 1, 1985.

[GARC83] Garcia-Molina H., Using Semantic Knowledge for Transaction Processing in a Distributed Database. *ACM Transactions on Database Systems*, Vol. 8, No. 2, June 1983.

[GUIB78] Guibas C. S., Sedgewick R., A Dichromatic Framework for Balanced Trees. *Proceedings of the 19th Annual Symp. Foundation Comp. Science*, 1978.

[HSU83] Hsu M., Madnick S., Hierarchical Database Decomposition — A Technique for Database Concurrency Control. *Proceedings of the second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Atlanta, Georgia, March 1983.

[KERS84] Kersten M. L., Tebra H., Application of an Optimistic Concurrency Control Method. *Software-Practice and Experience*. Eds. John Wiley and Sons, Vol. 14, No. 2, February 1984.

[KNUT73]Knuth D. E., *The Art of Computer Programming, vol 3: Sorting and Searching*, Reading MA : Addisson-Wesley, 1973.

[KOHL81]Kohler W. H., A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems. *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.

[KUNG80]Kung H. T., Lehman P. L., A concurrent database manipulation problem: binary search trees. *ACM Transactions on Database Systems*, Vol. 5, No 3, 1980.

[KUNG81]Kung H., Robinson J., On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981.

[KWON82]Kwong Y., Wood D., A new method for concurrency in B-trees. *IEEE Transactions on Soft. Engineering*, Vol 8, No 3, 1982.

[LAUS84] Lausen G., Integrated Concurrency Control in Shared B-Trees. *Computing*, Vol. 33, No. 1, Eds. Spring-Verlag, New York, 1984.

[LEHM81]Lehman P. L. Yao S. B., Efficient Locking for Concurrent Operation on B-Trees. *ACM Transactions on Database Systems*, Vol. 6, No. 4, 1981.

[MILL78] Miller R., Snyder L., Multiple access to B-trees. *Proceedings of the 12th Annual Conference in Information Science and Systems*, March 1978.

[PAPA79] Papadimitriou C., Serializability of Concurrent Updates. *Journal of the ACM*, Vol. 26, No. 4, October 1979.

[SAMA76]Samadi B., B-trees in a system with multiple users. *Information Processing letters*, Vol. 5, No. 4, 1976.

[SPEC83] Spector A., Schwartz P., Transactions: A Construct for Reliable Distributed Computing. *Operating Systems Review*, Vol. 17, No. 2, April 1983.

[SPEC85] Spector A. Z., Butcher J., Daniels D. S., Duchamp D. J., Eppinger J. L., Fineman C. E., Heddaya A., Schwarz P., Support for Distributed Transactions in the TABS Prototype. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, June 1985.