TREE LOCKING ON CHANGING TREES

Vladimir Lanin and Dennis Shasha

Courant Institute of Mathematical Sciences, New York University

lanin@csd2.nyu.edu, shasha@nyu.edu

ABSTRACT: The tree locking protocol is a deadlock-free method of concurrency control defined and verified by Silberschatz and Kedem for data organized in a directed tree. Can the tree protocol work for applications that change the tree? We define a set of three operations capable of changing any tree to any other tree and show that the tree protocol continues to ensure serializability and deadlock-freedom in the presence of these operations.

1. Introduction

A locking protocol is a set of rules for locking data items such that any concurrent computation following those rules is guaranteed to satisfy some set of conditions. Typically, these conditions may include serializability, deadlock freedom, or order preservation, which are all rigorously defined below. For example, the two-phase protocol guarantees serializability and order preservation, but not deadlock freedom, by forbidding an action (a term we use interchangeably with ‘‘transaction’’) to place a new lock after releasing a lock.

In [SK80], Silberschatz and Kedem introduced a locking protocol that guaranteed serializability and deadlock freedom without requiring two-phasedness. It has since become known as the *tree protocol* since it is based on the assumption that that the data resides in a set of nodes organized in a directed tree. In brief, the protocol allows an action to begin by locking any node, but to place subsequent locks only on the children of its currently locked nodes, as long as it does not lock a node it has previously unlocked. No restrictions are placed on unlocking nodes.

This work was partially supported by the National Science Foundation under grant number IRI-8901699 and by the Office of Naval Research under grant number N00014-85-K-0046.

It is an unstated assumption of the tree protocol that the tree graph remain the same throughout a computation. This would seem to be a major limitation, since many database applications, such as B-tree algorithms, require the on-line restructuring of a tree. But does the tree really have to be static?

Several conflict-preserving concurrent B-tree algorithms, including those in [Sa76], [BS77], and [MR85], do, in fact, bear a striking resemblance to the tree protocol. They always retain the lock on a parent node until after its child has been locked, and keep a node locked as long as there is any possibility it might have to be modified. Each has been shown correct by various ad-hoc methods.

In this paper, we define a set of operations for modifying trees, extend the tree protocol to computations that include these operations, and then show that the resulting protocol continues to guarantee serializability and deadlock freedom. In addition, we examine the conditions under which it is order preserving.

2. Tree Editing Operations

We must now decide on a set of operations powerful enough to introduce arbitrary changes to the tree graph, yet restricted enough to ensure that the graph remain a tree in all intermediate states. B-tree splits and merges, for example, are clearly too restrictive in that they can only produce balanced trees and can not even change the height of any given node. Addition or removal of a single edge, on the other hand, is sure to disrupt the tree property.

Consider, however, an operation that changes (switches) the parent of a node c from p_1 to p_2 (simultaneously removing edge (p_1, c) and adding edge (p_2, c), see Fig. 1). If p_2 is not c or a descendant of c, the graph is sure to remain a tree. And yet a sequence of these *switch* operations can rearrange any given tree to any other given tree with only two limitations: the same node would remain the root and the set of nodes would remain the same.

The first limitation is not important, since the root can always be used as just a pointer to the ‘‘real’’ root. To eliminate the second limitation, we introduce another two operations. The *add_leaf* adds a new leaf c to the graph, along with the edge (p, c) from some old node p. The *remove_leaf* is the inverse, removing edge (p, c) to some leaf c and removing c from the set of nodes. Both, of course, maintain the

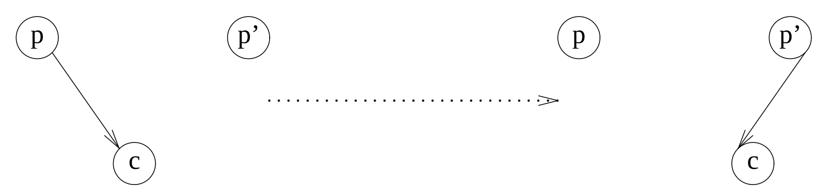


Figure 1.

The *switch*(p,p',c) operation.

tree property. Since the *switch* operation can be used to reposition a new leaf to an arbitrary position in the tree, or to reposition an arbitrary node to a leaf position where it can be removed, the three operations together can truly restructure the tree arbitrarily.

Since the tree editing operations treat the tree graph itself as shared data to be examined and modified, we must require that certain locks be held on the affected nodes before these operations may be executed. The *switch* operation requires that the executing action hold write locks on the two parent nodes. (Incidentally, the locks on the parent nodes make it possible for the *switch* to appear atomic even though it is actually likely to be implemented in at least two steps.) Note that the *switch* need not hold any lock on the child being moved. The *add_leaf* requires a write lock on the parent, and grants a write lock on the new child. The *remove_leaf* requires a write lock on both parent and child. These requirements are quite natural if the graph data is stored in the form of a list of children in each node.

As it turns out, certain subclasses of the *switch* operation have interesting properties to be discussed below. Let the *switch*$'(p,p',c)$ be a *switch* prior to which the executing action has never held a lock on c. Let the *switch*$''(p,p',c)$ be a *switch* where either p is a child of p' or vice-versa. A *switch* operation belonging to either of these two classes is called *restricted.* Similarly, a computation where every *switch* operation is restricted is called *switch restricted.*

It is noteworthy that the *switch′′* subclass alone should be sufficient for most tree-restructuring applications, since it is unusual to transfer a child between two completely unrelated nodes. To transfer a child from a node to its sibling, for example, as in a B-tree split or merge, we can use one *switch′′* to first transfer it to the common parent, then another *switch′′* to transfer it from the parent to the sibling.

3. Goals

It is now our aim to explore the properties of computations containing the tree editing operations and following a set of rules (to be explicitly defined below) akin to those of the static tree protocol. We will eventually show that such computations, just like static tree computations, are always serializable and deadlock-free, but not necessarily order-preserving. However, we will define conditions under which order is preserved between certain actions.

One property of the static tree protocol that is not always preserved in dynamic trees is the predetermination of the serialization ordering. As we will show, the order in which two actions can appear to serialize at the end of the computation can be determined in static trees as soon as both have locked their first nodes. For this property to hold in dynamic trees, the computation must not contain any unrestricted *switch* operations.

4. Notation

To proceed further, we must introduce unambiguous notation. To simplify both this notation and the following discussion, let us initially restrict the protocol to exclusive locks.

4.1. States, Operations, and Specifications

We consider a computation to be a sequence of operations, each operation belonging to some higher-level action. (Since we are dealing with concurrent computations, the operations of concurrent actions will be interleaved.) The operations we are concerned with here are those locking and unlocking nodes and/or modifying the tree, i.e. *lock_first*, *lock_child*, *unlock*, *switch*, *add_leaf*, and *remove_leaf*. Each operation changes the state of the underlying data in accordance with the operation's specification. To formulate the specifications, we must rigorously define the states to which they refer.

We consider a state of the computation to consist of three components: *T*, *has*, and *had*. $T = (E,N)$ is the current tree graph, where *E* is the set of edges and *N* is the set of nodes. *Has* is a function mapping each action to the set of nodes on which it currently holds locks. *Had* is a function mapping each action to the set of nodes on which it either holds or has ever held locks. (Thus, $has(a) \subseteq had(a)$.) Alternatively, we shall consider *has* and *had* to be sets of pairs of the form (a,n) where *a* is an action and *n* is a node. Thus, if $n \in has(a)$, then $(a, n) \in has$. Let $ancestors(n,T)$ be the set of ancestors of node *n* in tree *T*.

We express an operation's specification in two parts: a transformation from the state in which the operation starts to the state in which it finishes, and a condition on the starting state for which the operation waits to become true. (For example, a lock operation waits until its node is not locked, i.e. is not in the *has* of any other action.) We assume that the operations are implemented correctly, i.e. that in any concurrent computation containing the above operations, the operations can be placed in an interleaved order such that the conditions (both transforming and waiting) of each operation's specification are fulfilled for the state preceding and following the operation in the interleaving.

We now list the operations and their specifications. Only *lock_first* and *lock_child* have waiting conditions. Within the context of a specification, let *r* be the state preceding the operation, and *s* be the state following it. We will use state names as subscripts to denote the state to which some particular entity refers, i.e. T_s for the tree in state *s*. We subscript the operations with the name of the executing action, i.e. $lock\_first_a(n)$ is the *lock_first* operation performed by action *a* on node *n*. In both cases, we will sometimes omit the subscript when the meaning is made clear by other means.

$lock\_first_a(n)$ and $lock\_child_a(p,n)$:

waiting condition: $\forall_b \; n \notin has_r(b)$

transformation: $T_s = T_r \qquad has_s = has_r \cup (a,n) \qquad had_s = had_r \cup (a,n)$.

$unlock_a(n)$:

transformation: $T_s = T_r \qquad has_s = has_r - (a,n) \qquad had_s = had_r$.

$switch_a(p,p',c)$:

transformation: $E_s = ((E_r - (p,c)) \cup (p',c))$ $N_s = N_r$ $has_s = has_r$ $had_s = had_r$.

$add\_leaf_a(p,c)$:

transformation: $N_s = N_r \cup c$ $E_s = E_r \cup (p,c)$ $has_s = has_r \cup (a,c)$

$had_s = had_r \cup (a,c)$.

$remove\_leaf_a(p,c)$:

transformation: $N_s = N_r - c$ $E_s = E_r - (p,c)$ $has_s = has_r - (a,c)$ $had_s = had_r$.

4.2. The Protocol

A locking protocol is an additional set of restrictions on the allowed computations. We shall express these restrictions as additional conditions on the starting state of each operation. For example, for a $lock\_child_a(p,c)$, p must be the parent of c, a must already hold a lock on p, and a must never have held a lock on c. As opposed to an operations' specification, the protocol conditions are achieved not by the operation itself but by the operations preceding it in the computation. The protocol conditions are:

$lock\_first_a(n)$

$had_r(a) = \emptyset$ $n \in N_r$

$lock\_child_a(p,n)$:

$(p,c) \in E_r$ $p \in has_r(a)$ $c \notin had_r(a)$

$unlock_a(n)$:

$n \in has_r(a)$

$switch_a(p,p',c)$:

$(p,c) \in E_r$ $\{p,p'\} \subseteq has_r(a)$ $c \neq p'$ $c \notin ancestors(p',T_r)$

$switch'_a(p,p',c)$:

same as *switch*, plus $c \notin had_r(a)$

$switch''_a(p,p',c)$:

same as *switch*, plus $(p,p') \in T_r$ $(p',p) \in T_r$.

$add\_leaf_a(p,c)$:

$$p \in has_r(a) \qquad c \notin N_r \qquad \forall_b \; c \notin had_r(b)$$

$remove\_leaf_a(p,c)$:

$$(p,c) \in E_r \qquad \forall_n \; (c,n) \notin E_r \qquad \{p,c\} \subseteq has_r(a)$$

A computation satisfies the dynamic tree locking protocol if and only if

1) no action accesses a node on which it does not hold a lock, and

2) the operations listed above are the only ones that place or remove locks on nodes or modify the tree, and

3) the protocol's conditions hold for each occurrence of the above operation in the state in which it begins.

For short, such a computation is called a *dynamic tree computation.* In practice, we will not need to make use of the distinction between transformation, waiting, and protocol conditions. For any dynamic tree computation, they all hold equally true.

Let us name a computation's initial state *init* and its final state *fin*. For every computation, we assume that T_{init} is a tree and $has_{init} = had_{init} = \emptyset$. By this assumption, by the operations' specifications, and by the rules of the protocol, it is easy to verify that in every subsequent state *s*, T_s is still a tree and $has_s(a) \cap has_s(b) = \emptyset$ for all distinct actions *a* and *b*.

4.3. Ordering Relations

To prove serializability (as well as other properties), we must show that the conflict graph on the actions in a dynamic tree computation remains acyclic. Instead, it turns out to be more convenient to show that the transitive closure of a certain superset of the conflict relation remains irreflexive. We now proceed to define the various relations.

Let $a \underset{s}{\longrightarrow} b$ for actions *a* and *b* and state *s* if there exists a state *r* prior or equal to *s* such that $has_r(b) \cap had_r(a) \neq \emptyset$. In other words, $a \underset{s}{\longrightarrow} b$ if *a* and *b* both locked some node *n* prior to *s*, and *a* locked it before *b*. Let $\overset{+}{\underset{s}{\longrightarrow}}$ be the transitive closure of $\underset{s}{\longrightarrow}$, and let $\overset{*}{\underset{s}{\longrightarrow}}$ be its transitive and reflexive closure. We will indicate the closures of other relations in the same manner.

Thus, $\xrightarrow[s]{}$ is simply the conflicts-with relation on actions achieved up to state s. For this reason, a computation is (conflict-preserving) serializable if and only if its $\xrightarrow[fin]{+}$ is irreflexive.

Let $a \underset{s}{\twoheadrightarrow} b$ if there is a path q from node n to node m in T_s such that $n \in has_s(b)$, $m \in had_s(a)$, and the successor of n on q is not in $had_s(b)$ (see Fig. 2).

The $\twoheadrightarrow$ relation reflects conflicts that may arise via future *lock_child* operations. Thus, when $a \twoheadrightarrow b$, b may lock the nodes on q, starting with the successor of n and continuing down. Then, after it locks m, $a \longrightarrow b$ becomes true. Note that the above definition allows $a \twoheadrightarrow a$. For example, a may hold a lock on a node and its grandchild, but not on the interceding child. This is significant because if some other action b then locks the child node, $a \twoheadrightarrow b \twoheadrightarrow a$ will result.

Finally, let $_s$ be the union of $\xrightarrow[s]{}$ and $\underset{s}{\twoheadrightarrow}$. Thus, this relation reflects both conflicts that have already occurred and that could occur in the immediate future.

5. Intermediate Results

As stated above, an intermediate step to our goal of proving serializability and other properties is showing that $^+$ remains irreflexive throughout a dynamic tree computation. To achieve this, we must examine the effects of the various operations on $\longrightarrow$ and $\twoheadrightarrow$.

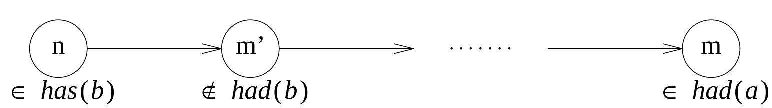


Figure 2.

$a \twoheadrightarrow b$ by the path from n to m.
If $a \neq b$, m' and m may be the same node.

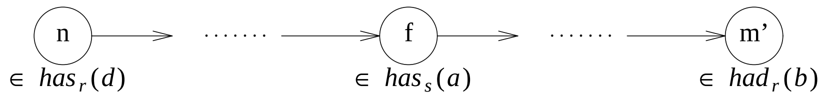


Figure 3.

T_s after a $lock\_first_a(f)$.
If $b \underset{s}{\twoheadrightarrow} a \;_s\; d$ then $b \underset{r}{\twoheadrightarrow} d$ by the path from n to m'.
(If $a \underset{s}{\longrightarrow} d$, then m' is f).

Lemma 1: if $lock\_first_a(f)$ maps state r to state s, and a, b and d are actions, then:

1) $a \;_s\; a$ does not hold.

2) if $b \neq a \quad d \neq a \quad b \;_s\; d$, then $b \;_r\; d$.

3) if $b \;_s\; a \;_s\; d$, then $b \;_r\; d$.

Proof:

1) $a \underset{s}{\longrightarrow} a$ is impossible by definition, and $a \underset{s}{\twoheadrightarrow} a$ can not hold since T_s is a tree and there is only one node in $had_s(a)$.

2) s and r are identical in all respects other than $had(a)$ and $has(a)$, and these play no part in $_s$ on actions distinct from a.

3) Since, as of s, a has yet to release any lock, $a \underset{s}{\longrightarrow} d$ is impossible for any d. Thus, let $a \underset{s}{\twoheadrightarrow} d$ by path q from n to m as defined above (see Fig. 3). Note that since $had_s(a) = \{f\}$, m must be f. If $b \underset{s}{\longrightarrow} a$ then $f \in had_r(b)$. But then $b \underset{r}{\twoheadrightarrow} d$ via q. Now let $b \underset{s}{\twoheadrightarrow} a$ by path q' from n' to m'. Once again, note $n' = f$. But then $b \underset{r}{\twoheadrightarrow} d$ via q followed by q'. □

Lemma 2: Let $lock\_first_a(f)$ map state r to state s, let a, b and d be actions, and let $b \ {}^{+}_{s}\ d$, but not $b \ {}^{+}_{r}\ d$. Then

1) either $b = a$, or $d = a$, and

2) if $b = d$, then $e \ {}^{+}_{r}\ e$ for some action e.

Proof:

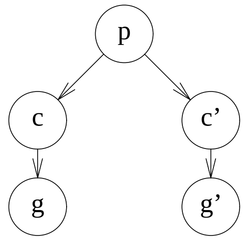
1) Assume $b \neq a \quad d \neq a$. Then, by lemma 1 part 2, there exists a b' and d' such that $b \ {}^{*}_{r}\ b' \ {}_{s}\ a \ {}_{s}\ d' \ {}^{*}_{r}\ d$. But then, by lemma 1 part 3, $b' \ {}_{r}\ d'$, thus $b \ {}^{+}_{r}\ d$, which is a contradiction.

2) Let $b = d$. Then, by part 1 of this lemma, $b = d = a$, thus $a \ {}^{+}_{s}\ a$. Then, by lemma 1 part 1, $a \ {}^{+}_{s}\ e \ {}^{+}_{s}\ a$ for some $e \neq a$. Thus, $e \ {}^{+}_{s}\ a \ {}^{+}_{s}\ e$. Then, by lemma 1 part 2, there exist an e' and e'' such that $e \ {}^{*}_{r}\ e' \ {}_{s}\ a \ {}_{s}\ e'' \ {}^{*}_{r}\ e$. But then by lemma 1 part 3, $e' \ {}_{r}\ e''$, thus $e \ {}^{+}_{r}\ e$. □

As we are about to show, it is the property of the static tree protocol that *lock_first* is the only operation that introduces new edges to ${}^{+}$. Together with lemma 2, this is sufficient to prove serializability and go a long way toward deadlock freedom. The remainder of this paper would be much simpler if this property also held true for the dynamic case.

Unfortunately, however, the *switch* operation can, under certain circumstances, add new edges to ${}^{+}$. Consider, for example, the scenario in Fig. 4. Up to the *switch* operation, b and d are unrelated in . After it, however, $b \twoheadrightarrow d$. Furthermore, the last operation could have just as easily been $switch_a(p,g,c')$, resulting in $d \twoheadrightarrow b$.

This is not just an artifact of a poor definition for . The *switch* performed by a makes it possible for d to go on to lock c, resulting in $b \xrightarrow[fin]{} d$. Had a performed $switch_a(p,g,c')$, b could go on to lock c', resulting in $d \xrightarrow[fin]{} b$. As we will show, in the static case, one can determine an order in which b and d can appear



T_{init}

a	b	d
$lock\_first(p)$		
$lock\_child(p,c)$		
$lock\_child(c,g)$		
$lock\_child(p,c')$		
$lock\_child(c',g')$		
$unlock(c)$		
$unlock(c')$		
	$lock\_first(c)$	
		$lock\_first(c')$
$switch(p,g',c)$		

Figure 4.

The non-restricted *switch* makes $b \twoheadrightarrow d$.

in the serialization ordering as soon as both have locked their first node.

Still, this property of the static protocol can be retained by restricting the *switch* operation to $switch'$ and $switch''$. Thus, we now show that operations other than $lock\_first$ and non-restricted *switch* do not expand $\overset{+}{}$.

Lemma 3: Let o be any operation other than $lock\_first$ or a non-restricted *switch*, and let o map state r to state s. Then $\overset{+}{\underset{s}{}} \subseteq \overset{+}{\underset{r}{}}$.

Proof: Let us consider each operation in turn.

• $unlock_a(n)$ and $remove\_leaf_a(p,c)$: Note that $has_s \subseteq has_r$ and $had_s = had_r$, thus $\underset{s}{\longrightarrow} = \underset{r}{\longrightarrow}$. Furthermore, since $E_s \subseteq E_r$, $\underset{s}{\twoheadrightarrow} \subseteq \underset{r}{\twoheadrightarrow}$.

• $add\_leaf_a(p,c)$: Since $had_s = had_r \cup (a,c)$, and c has never been locked before, $\underset{s}{\longrightarrow} = \underset{r}{\longrightarrow}$. Now assume $b \underset{s}{\twoheadrightarrow} d$ but not $b \underset{r}{\twoheadrightarrow} d$. Let q be the path from n to m by which $b \underset{s}{\twoheadrightarrow} d$. Since $E_s = E_r \cup (p,c)$, and the only newly locked node is c, which is a leaf, it must be that $c = m \quad a = b$. Note that p must be the predecessor of c in q, and $p \in has_r(b)$. Note $p \neq n$ (otherwise $b = a = d$, but $a \underset{s}{\twoheadrightarrow} a$ can not hold for the single-edge path from $n = p$ to $m = c$). But then $b \underset{r}{\twoheadrightarrow} d$ by the path from n to p.

• $lock\_child_a(p,c)$: Let $b \underset{s}{\longrightarrow} d$ but not $b \underset{r}{\longrightarrow} d$. Then the conflict occurred at c, thus $a = d$, and $c \in had_r(b)$. Since $p \in has_r(d)$, $b \underset{r}{\twoheadrightarrow} d$ by the path from p to c.

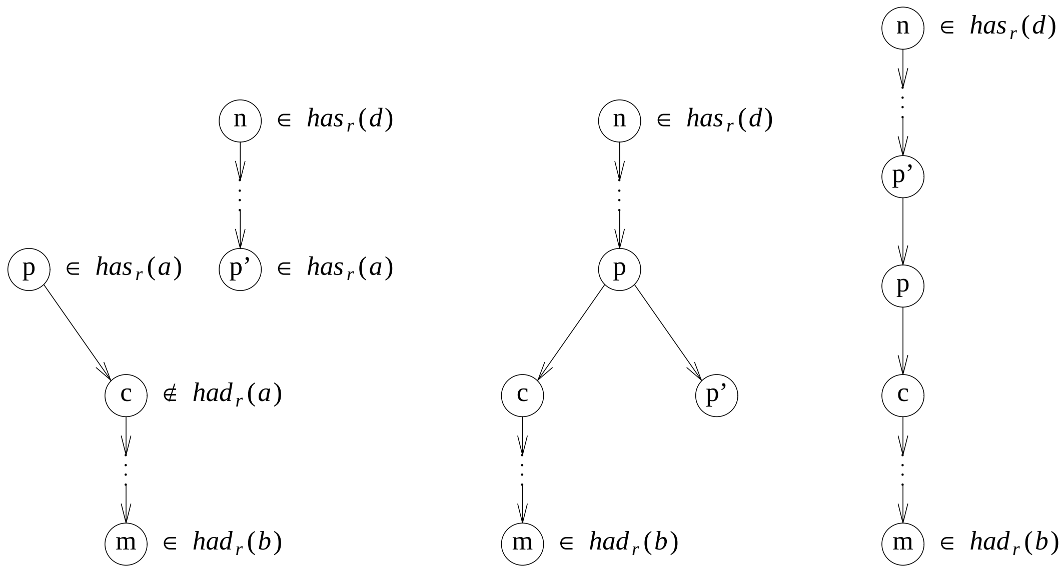
Now let $b \underset{s}{\twoheadrightarrow} d$ but not $b \underset{r}{\twoheadrightarrow} d$. Let q be the path from n to m by which $b \underset{s}{\twoheadrightarrow} d$. Since $T_s = T_r$, and the only newly locked node is c, either $c = n \quad a = d$, or $c = m \quad a = b$. In the first case, since $c \notin had_r(a)$, $b \underset{r}{\twoheadrightarrow} d$ by the path from p to c to m. For the second case, see the corresponding argument under add_leaf.

• $switch'_a(p,p',c)$ and $switch''_a(p,p',c)$: There are no newly locked nodes in s, thus $\underset{s}{\longrightarrow} = \underset{r}{\longrightarrow}$. Assume that indeed $b \underset{s}{\twoheadrightarrow} d$, but not $b \underset{r}{\twoheadrightarrow} d$, for some actions b and d. The only new edge in T_s is (p',c). Then q (the path from n to m defined above by which $b \underset{s}{\twoheadrightarrow} d$) must include (p',c) (see Fig. 5).

If $n = p'$, then $c \notin had_r(d)$. Furthermore, since $n \in has_r(d)$, and $p' \in has_r(a)$, a must then be d. But then $b \underset{r}{\twoheadrightarrow} d$ by the path from p to c to m. Thus, we assume $n \neq p'$, and therefore $a \underset{r}{\twoheadrightarrow} d$ by the path from n to p'. (We know the successor of n on this path is not in $had_r(d)$ because the path from n to m by which $b \underset{s}{\twoheadrightarrow} d$ goes through p').

For $switch'$, since $c \notin had_r(a)$, $b \underset{r}{\twoheadrightarrow} a$ by the path from p to c to m. Thus, $b \underset{r}{\twoheadrightarrow} a \underset{r}{\twoheadrightarrow} d$.

For $switch''$, if $(p',p) \in E_r$, then $b \underset{r}{\twoheadrightarrow} d$ by the path from n to p' to p to c to m. And if $(p,p') \in E_r$, then p must be the predecessor of p' on q, thus $b \underset{r}{\twoheadrightarrow} d$ by the path from n to p to c to m.



$switch'$: $c \notin had_r(a)$ $switch''$: $(p,p') \in E_r$ $switch''$: $(p',p) \in E_r$

Figure 5.

T_r prior to restricted $switch_a(p,p',c)$ resulting in $b \underset{s}{\twoheadrightarrow} d$.

In all cases, $b \overset{+}{\underset{r}{\twoheadrightarrow}} d$.

(We know that p' must have a predecessor on q since $p' \neq n$.) □

As mentioned above, one consequence of lemmas 2 and 3 is that the switch-restricted tree protocol (but not the unrestricted one) determines an action's position in the serialization ordering as soon as the action locks its first node:

Lemma 4: In any switch-restricted dynamic tree computation, if $b \,{}^{+}_{fin}\, d$, then $b \,{}^{+}\, d$ as soon as both b and d have executed their *lock_first* operations, and in all subsequent states.

Proof: Assume otherwise. Then at least one operation mapping state r to state s where $b \,{}^{+}_{s}\, d$ but not $b \,{}^{+}_{r}\, d$ must occur after both b and d have started. Let o be that operation. By lemma 3, o must be a

$lock\_first_a$ for some a. But by lemma 2 part 1, a is either b or d, which is a contradiction. □

Another immediate consequence of lemmas 2 and 3 is that $^{+}$ remains irreflexive in all states, thus guaranteeing serializability for switch-restricted computations.

We now proceed to show that this result also holds for non-restricted computations, even though lemma 3 is not sufficient for them. Doing this requires some extra work.

Definition: let $lca(v,v',T)$ be the lowest common ancestor of nodes v and v' in tree T, i.e. the node in $(\{v\} \cup ancestors(v,T)) \cap (\{v'\} \cup ancestors(v',T))$ that is furthest from the root of T. Let $\Lambda(v,v',T)$ be the set of nodes on the paths in T from $lca(v,v',T)$ to v and v' (including v, v', and $lca(v,v',T)$).

Lemma 5: In any state s of a dynamic tree computation, let v and v' be any two nodes in $had_s(b)$ for any action b. Then for every node u in $\Lambda(v,v',T_s)$ there exists some action b' such that $u \in had_s(b')$ and $b \xrightarrow[s]{*} b'$.

Proof: by induction on the length of the computation. The lemma is trivially true for the initial state. We will now consider each operation in turn, assuming the lemma holds in the state r preceding it. It is helpful to remember that, by definition, for all actions a and a', if $a \xrightarrow[r]{} a'$, then $a \xrightarrow[s]{} a'$, and, for all nodes v, if $v \in had_r(a)$, then $v \in had_s(a)$.

• $unlock_a(n)$ and $remove\_leaf_a(p,c)$: for any two nodes, the lemma holds in s the same way it holds in r.

• $lock\_first_a(f)$: for any two nodes distinct from f, or if $b \neq a$, the lemma holds in s the same way it holds in r. As for $v = f \quad b = a$, $had_s(a) = \{f\}$, and $\Lambda(f,f,T_s) = \{f\}$, and $f \in had_s(a)$.

• $lock\_child_b(p,c)$ and $add\_leaf_b(p,c)$: for any two nodes distinct from c, the lemma holds in s the same way it holds in r. Also, if $v = c$, but $b \neq a$, the lemma holds in s as in r (in the add_leaf case, $c \notin had_r(b) \cup had_s(b)$ for any $b \neq a$). As for $v = c \quad b = a$, let v' be any other node in $had_s(a)$. Note that $\Lambda(c,v',T_s) = \Lambda(p,v',T_r) \cup \{c\}$. Since p and v' are in $had_r(a)$, we know the lemma holds for the nodes in $\Lambda(p,v',T_r)$, and $c \in had_s(a)$.

• $switch_a(p,p',c)$: For any $\{v,v'\} \subseteq had_s(b)$ for some action b, if $\Lambda(v,v',T_s)$ does not include both p'

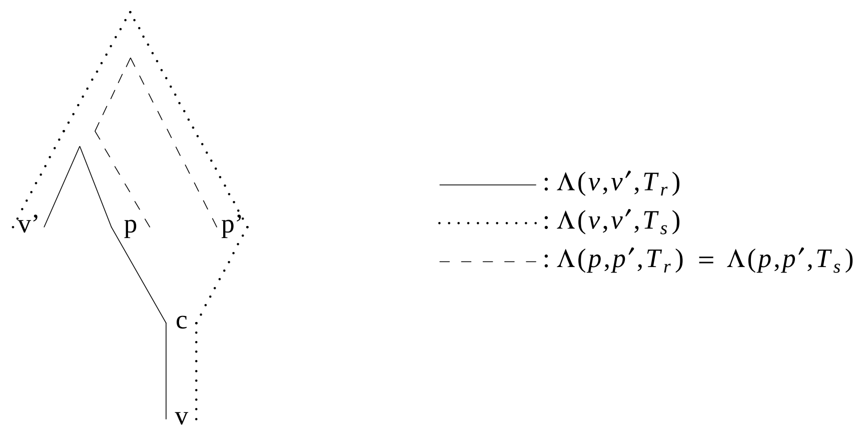


Figure 6.

$$\Lambda(v,v',T_s) \subseteq \Lambda(v,v',T_r) \cup \Lambda(p,p',T_r)$$

and c, the lemma holds in s as in r. Thus, let the path from $lca(v,v',T_s)$ to v include edge (p',c). Then, by definition of lca, v' is not in the subtree dominated by c (see Fig. 6). Then, in T_r, the path from $lca(v,v',T_r)$ to v had to include (p,c).

By the inductive hypothesis, since $p \in \Lambda(v,v',T_r)$, there exists a b' such that $p \in had_r(b')$ and $b \overset{*}{\underset{r}{\longrightarrow}} b'$. Thus, since $p \in has_r(a)$, $b \overset{*}{\underset{r}{\longrightarrow}} b' \overset{*}{\underset{r}{\longrightarrow}} a$.

Note that $\Lambda(v,v',T_s) \subseteq \Lambda(v,v',T_r) \cup \Lambda(p,p',T_r)$ (see Fig. 6). Since $\{v,v'\} \subseteq had_r(b)$, the lemma holds in s as in r for all nodes in $\Lambda(v,v',T_r)$. As for nodes u in $\Lambda(p,p',T_r)$, since $\{p,p'\} \subseteq had_r(a)$, there exists an a' such that $u \in had_r(a')$ and $a \overset{*}{\underset{r}{\longrightarrow}} a'$. Thus, $b \overset{*}{\underset{r}{\longrightarrow}} a \overset{*}{\underset{r}{\longrightarrow}} a'$. $\square$

Corollary: In any state s of a dynamic tree computation, if nodes v and v' are in $had_s(a)$ for some action a, and some node u is in $has_s(b)$ for some distinct action b, and $u \in \Lambda(v,v',T_s)$, then $a \overset{+}{\underset{s}{\longrightarrow}} b$.

Lemma 6: Let $switch_a(p,p',c)$ map state r to state s in a dynamic tree computation containing no $remove\_leaf$ operations, let b and d be actions, and let $b \;{}_{s}\; d$, but not $b \;\overset{+}{{}_{r}}\; d$. Let S_c be the set of nodes in the subtree dominated by c, and note that S_c is the same in T_s and T_r. Then

1) $b \underset{s}{\twoheadrightarrow} d$ by a path that includes (p',c), and

2) $had_s(b) \subseteq S_c$, and

3) $b \neq d$, and

4) if $b' \ {}_s\ b$ for any action b', then $b' \ {}_s\ d$.

Proof: Since $\underset{s}{\longrightarrow} = \underset{r}{\longrightarrow}$, $b \underset{s}{\twoheadrightarrow} d$ by a path from node n to node m that includes edge (p',c) (part 1).

See Figure 7.

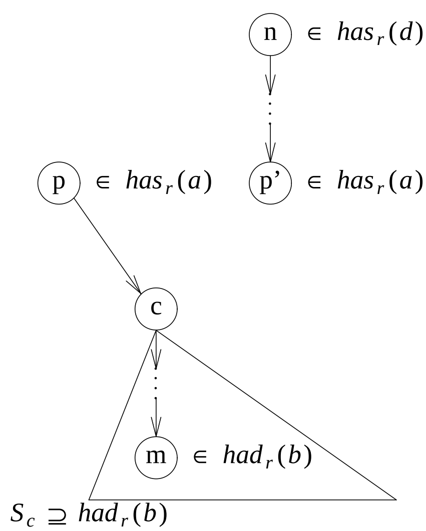


Figure 7.

T_r prior to an (unrestricted) $switch_a(p,p',c)$ resulting in $b \underset{s}{\twoheadrightarrow} d$.
As shown in lemma 6, as long as $had_s(b) \subseteq N_s$,
$had_s(b) \subseteq S_c$.

If $n = p'$, then $c \notin had_r(d)$. Furthermore, since $n \in has_r(d)$, and $p' \in has_r(a)$, a must then be d. But then $b \underset{r}{\twoheadrightarrow} d$ by the path from p to c to m, which is a contradiction. Thus, $n \neq p'$, and therefore $a \underset{r}{\twoheadrightarrow} d$ by the path from n to p'.

Assume there exists a node m' in $had_s(b) - S_c$. Since the computation contains no *remove_leaf* operations, $had_s(b) \subseteq N_s$. Then $p \in \Lambda(m, m', T_r)$. Then by the corollary to lemma 5, $b \overset{*}{\underset{r}{\longrightarrow}} a$. But since $a \underset{r}{\twoheadrightarrow} d$, this is a contradiction. Thus, $had_s(b) \subseteq S_c$ (part 2). Since $n \in had_s(d)$ but $n \notin S_c$, $b \neq d$ (part 3).

Case 1: $b' \underset{s}{\longrightarrow} b$ for some node m'. Then $m' \in had_s(b) \subseteq S_c$. But then $b' \underset{s}{\twoheadrightarrow} d$ by the path from n to c to m' (part 4).

Case 2: $b' \underset{s}{\twoheadrightarrow} b$ by a path from some node n' to some node m'. Then $n' \in had_s(b) \subseteq S_c$. But then $b' \underset{s}{\twoheadrightarrow} d$ by the path from n to c to n' to m' (part 4). □

Lemma 7: In any dynamic tree computation C, ${}^{+}$ is irreflexive in all states.

Proof: Assume otherwise. Let C' be the computation derived from C by replacing every $remove\_leaf_a(p,c)$ operation with an $unlock_a(c)$. (Since the specification of *remove_leaf* includes releasing a's lock on c anyway, this replacement only has the effect of keeping c in the tree.) Clearly, C' is a perfectly legal dynamic tree computation, and, for any state s in C and its corresponding state s' in C', $had_{s'} = had_s$, $has_{s'} = has_s$, and $E_{s'} \supseteq E_s$. Therefore, $\underset{s'}{\longrightarrow} = \underset{s}{\longrightarrow}$, and $\underset{s'}{\twoheadrightarrow} \supseteq \underset{s}{\twoheadrightarrow}$.

Thus, if there is a state with a reflexive ${}^{+}$ in C, there is such a state in C'. Let s be the first such state. Since $had_{init} = \emptyset$, $s \neq init$. Thus, let o be the operation preceding s, and let r be the state preceding o, with ${}^{+}_{r}$ irreflexive. Lemma 3 shows that o is either a *lock_first* or a *switch*. Lemma 2 part 2 shows that o is not a *lock_first*. Thus, o is a *switch*.

Let $b_0 \;{}_s\; b_1 \;{}_s\; \cdots \;{}_s\; b_{k-1} \;{}_s\; b_0$ be a minimal cycle in ${}_s$. Since ${}^{+}_{r}$ is irreflexive, $b_j \;{}_r\; b_{(j+1) \bmod k}$ does not hold for some j.

By lemma 6, part 3, $k > 1$. Since $b_{(j-1) \bmod k} \;_{s}\; b_j$, by lemma 6, part 4, $b_{(j-1) \bmod k} \;_{s}\; b_{(j+1) \bmod k}$. But then the cycle is not minimal, which is a contradiction. □

6. Serializability in the Presence of Readers

We say that a computation is conflict-preserving serializable if its $\xrightarrow[fin]{+}$ relation is irreflexive. Our first main result, a corollary of lemma 7, is:

Theorem 1: Every dynamic tree computation is conflict-preserving serializable.

The original tree protocol as defined in [SK80] dealt only with exclusive locks. Since this is a drastic limitation on a protocol's practicality, [KS83] considered the problem of extending a locking protocol to the use of read-locks. As a first step, they adopted the following:

Segregation Rule: an action may place either only read-locks or only write-locks.

In the first case, the action is known as a reader, in the second, a writer. Thus, Let R be the set of readers and W be the set of writers.

With this innovation, certain alterations must be made to the specifications of the operations. The waiting condition $\underset{b}{\forall}\ n \notin has_r(b)$ for $lock\_first_a(n)$ and $lock\_child_a(p,n)$ has to be replaced with $\underset{b}{\forall}\ n \notin has_r(b) \quad \{a,b\} \subseteq R$. The protocol conditions of $switch_a$, add_leaf_a, and $remove\_leaf_a$ must now be modified to include $a \in W$. The definitions of $\underset{s}{\longrightarrow}$ and $\underset{s}{\twoheadrightarrow}$ need also be revamped. For $a \underset{s}{\longrightarrow} b$ to hold now, at least one of a and b must be a writer. The same is true for $a \underset{s}{\twoheadrightarrow} b$, except $a \underset{s}{\twoheadrightarrow} a$ must still hold even if a is a reader.

Surprisingly, as shown in [KS83], the segregation rule alone is insufficient to guarantee serializability even in the original tree protocol. In the presence of readers, $^{+}$ may indeed cease to be irreflexive. (The problem occurs in the proof of lemma 1. It may be that $b \;_{s}\; a \underset{s}{\twoheadrightarrow} d$, but not $b \underset{r}{\twoheadrightarrow} d$ because both b and d are readers.)

Fortunately, one of the theorems in [KS83] shows that any serializable write-lock protocol can be converted to the use of read-locks if the segregation rule is combined with the following:

Transitive Conflict Rule: if $w \underset{fin}{\longrightarrow} r \underset{fin}{\longrightarrow} w'$ where r is a reader and $w \neq w'$, then $w \underset{fin}{\longrightarrow} w'$ must also hold.

This method may be applied to the dynamic tree protocol as well, thus producing the *segregated dynamic tree protocol.* We paraphrase and extend the proof of the relevant theorem in [KS83] below:

Lemma 8: The segregated dynamic tree protocol is serializable.

Proof: Assume there exists a computation C where $a_0 \underset{fin}{\longrightarrow} a_1 \underset{fin}{\longrightarrow} \cdots \underset{fin}{\longrightarrow} a_{n-1} \underset{fin}{\longrightarrow} a_0$ is a minimal cycle in $\underset{fin}{\longrightarrow}$. Note that since readers can not conflict with other readers, $a_i \in R$ implies $\{a_{i-1}, a_{i+1}\} \subseteq W$. Then there are two cases.

Case 1: $n > 2$. Then $i-1 \neq i+1$ (modulo n) for all $0 \leq i \leq n-1$. Thus, since the cycle is minimal, $a_{(i-1) \bmod n} \neq a_{(i+1) \bmod n}$. Then by the transitive conflict rule, if a_i is a reader, then $a_{(i-1) \bmod n} \underset{fin}{\longrightarrow} a_{(i+1) \bmod n}$. But then the cycle is not minimal, so all the a_i's must be writers. Let C' be C restricted to the writer actions. Since reader actions are not allowed tree-modifying operations, C' is a perfectly legal writer-only dynamic tree computation. But now $a_0 \underset{fin}{\longrightarrow} \cdots \underset{fin}{\longrightarrow} a_{n-1} \underset{fin}{\longrightarrow} a_0$ in C', which contradicts Theorem 1.

Case 2: $n \leq 2$. Since $a \longrightarrow a$ is impossible by definition, $n = 2$. If both a_0 and a_1 are writers, proceed as in case 1. Otherwise, since both can't be readers, let a_0 be the reader. Let C' be C restricted to the writer actions and a_0, and let a_0 now be a writer. Since the readers are not allowed tree modifying operations, and the locks of a_0 conflicted with the locks of all the writers anyway, C' is a legal writer-only dynamic tree computation. But now $a_0 \underset{fin}{\longrightarrow} a_1 \underset{fin}{\longrightarrow} a_0$ in C' which contradicts Theorem 1. □

We should note that the transitive conflict rule may be enforced by having all writers start at the root (as suggested in [KS83]), or by restricting readers to locking only one node (i.e. disallowing them *lock_child* operations).

7. Deadlock Freedom

Deadlock is a state where there exists a set of actions $a_0, \cdots, a_{n-1}$ such that a_i is waiting for a resource held by $a_{(i+1) \bmod n}$. In our context, locks are the only resource for which an action can wait. Although waiting can be initiated by both the *lock_first* and *lock_child* operations, the *lock_first* can not play a part in the deadlock cycle because an action can never be holding other locks while waiting for its first lock.

Thus, in a segregated dynamic tree protocol, state r is said to be *deadlock-prone* if there exists a sets of actions $a_0, \cdots, a_{n-1}$, nodes $p_0, \cdots, p_{n-1}$, and nodes $c_0, \cdots, c_{n-1}$ such that, for $0 \le i \le n-1$, $(p_i, c_i) \in E_r$, $p_i \in has_r(a_i)$, $c_i \notin had_r(a_i)$, $c_i \in has_r(a_{(i+1) \bmod n})$, and $a_i \in R$ implies $a_{(i+1) \bmod n} \in W$. >From such a state, deadlock would result if each a_i issued $lock\_child(p_i, c_i)$. Conversely, since *lock_child* operations are the only ones that can be involved in the deadlock cycle, deadlock can only be reached by going through a deadlock-prone state.

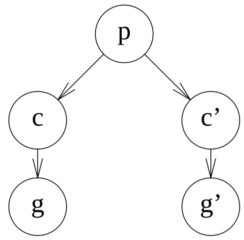
Theorem 2: A writer-only dynamic tree computation C never enters a deadlock-prone state s.

Proof: Assume otherwise. By definition of deadlock-prone state, $a_i \underset{s}{\twoheadrightarrow} a_{(i+1) \bmod n}$ by the path from p_i to c_i. Thus, $a_0 \overset{+}{\underset{s}{\twoheadrightarrow}} a_0$, which contradicts lemma 7. □

As with serializability, problems arise when we try to extend this result to computations containing readers. In fact, as illustrated by Fig. 8, even a segregated computation satisfying the original (non-dynamic) tree protocol and the transitive conflict rule may reach a deadlock-prone state. Thus, to guarantee deadlock freedom, we need to enforce some variation of the transitive conflict rule, such as:

For any state s, if $w \underset{s}{\twoheadrightarrow} r \underset{s}{\twoheadrightarrow} w'$ where r is a reader and $w \ne w'$, then $w \underset{s}{\twoheadrightarrow} w'$ must also hold.

It is easily shown by an argument similar to the proof of lemma 8 that this guarantees that $\overset{+}{\twoheadrightarrow}$ stays irreflexive in all states, thus making a deadlock-prone state impossible. We should also note that the methods mentioned above for enforcing the transitive conflict rule also work to enforce this variation.

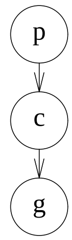


T_{init}

$r1 \in R$	$r2 \in R$	$w1 \in W$	$w2 \in W$
$lock\_first(p)$			
$lock\_child(p,c)$			
$lock\_child(c,g)$			
$unlock(c)$			
	$lock\_first(p)$		
	$lock\_child(p,c')$		
	$lock\_child(c',g')$		
	$unlock(c')$		
		$lock\_first(c)$	
			$lock\_first(c')$

Figure 8.

A serializable segregated computation satisfying the transitive conflict rule
with a deadlock-prone final state:
$r2$ may wait for $w1$ at c,
$w1$ may wait for $r1$ at g,
$r1$ may wait for $w2$ at c',
$w2$ may wait for $r2$ at g'.



T_{init}

a	*b*	*d*
$lock\_first(p)$		
$lock\_child(p,c)$		
$unlock(p)$		
	$lock\_first(p)$	
	$unlock(p)$	
		$lock\_first(g)$
		$unlock(g)$
$lock\_child(c,g)$		

Figure 9.

Even though b completely precedes d, $d \underset{fin}{\longrightarrow} a \underset{fin}{\longrightarrow} b$.

8. Order Preservation

We say that action *a completely precedes* action *b* if *a* releases all its locks before *b* places its first lock. A computation is *order-preserving* if, whenever $a \overset{+}{\underset{fin}{\longrightarrow}} b$, b does not completely precede a. In other words, if two actions actually executed in a certain order, they shouldn't appear by the results of the computation to execute in the opposite order.

The dynamic tree locking protocol, like the original tree protocol, does not in general guarantee order preservation. (See Fig. 9 for an example.) However, both protocols can be shown to preserve the temporal ordering between those pairs of actions related to each other in one of a number of ways.

In the case of the static tree protocol, the three relationships known to us can be stated succinctly. Order will be preserved between action b and a subsequent action d if d either

1) locked a node also locked by b, or

2) locked a node that is an ancestor of a node locked by *b*, or

3) locked a node that is a child of a node locked by *b*.

In the dynamic tree case, both the statement and proof of parts 2 and 3 above are complicated by the transiency of child and ancestor relationships: one is forced to state precisely when these relationships must have held in relation to the time that *b* and *d* locked the nodes involved. Thus, we break up this statement into three separate theorems:

Theorem 3: Let *C* be a computation of the segregated tree protocol wherein action *b* completely precedes action *d*. If there exists a node $n \in (had_{fin}(b) \cap had_{fin}(d))$, then $d \xrightarrow[fin]{+} b$ does not hold.

Proof: Let us construct a new computation, C', consisting only of the writers in *C*, as well as *b* and *d*. Since reader actions are not allowed *switch*, *add_leaf*, or *remove_leaf* operations, C' is a perfectly legal segregated dynamic tree computation.

Now, make both *b* and *d* writers in C'. Since *b* and *d* are not concurrent, and their locks conflict with all the other (writer) actions anyway, C' is still a perfectly legal dynamic tree computation, but now consisting only of writers. And since *C* satisfied the transitive conflict rule, $d \xrightarrow[fin]{+} b$ can hold in *C* only if it holds in C'.

Note that even if both *b* and *d* were readers in *C*, they are writers in C', thus $b \xrightarrow[fin]{} d$ in C' (at *n*). Therefore, by lemma 7, $d \xrightarrow[fin]{+} b$ can not hold in C'. □

It is noteworthy that the most common method of enforcing the Transitive Conflict Rule — having all actions start at the same node — also makes the dynamic tree protocol order-preserving by Theorem 3. Thus, Theorems 1, 2, and 3 can serve as rigorous correctness proofs for the B-tree algorithms in [Sa76], [BS77], and [MR85].

Theorem 4: Let *C* be a computation of the segregated tree protocol wherein action *b* completely precedes action *d*. If *C* is switch-restricted, and, in some state *s*, some node $n \in has_s(d)$ is an ancestor of some node $m \in had_s(b)$, then $d \xrightarrow[fin]{+} b$ does not hold.

Proof: Let us construct C' from C just as in Theorem 3. Let s be the first state in C' where $n \in has_s(d)$, $m \in had_s(b)$, and n is an ancestor of m. Let o be the operation preceding s, and let r be the state preceding o. Since r does not satisfy the above conditions, o must be either $lock\_first_d(n)$ or $switch_a(p,p',c)$ for some a where (p',c) is an edge on the path from n to m.

If o is a *lock_first*, then the successor of n on the path from n to m is not in $had_s(d) = \{n\}$, and thus $b \underset{s}{\twoheadrightarrow} d$ by the path from n to m. Unfortunately, we can make no assumption about the successor of n in the $o = switch_a(p,p',c)$ case. However, we can show that $b \overset{+}{\underset{s}{}} d$ also holds there anyway (see Fig. 10).

Since p is an ancestor of $m \in had_r(b)$ in r, and the conditions of the case do not hold in r, $n \in has_r(d)$ can not be p or an ancestor of p. Furthermore, since $p \in has_r(a)$, a can not be d. And since $p' \in has_r(a)$ but $n \in has_r(d)$, p' also isn't n. However, since (p',c) is an edge on the path from n to m in s, n is an ancestor of p'. Thus, $n \in \Lambda(p,p',T_s) - \{p,p'\}$. Then, by the corollary to lemma 5, $a \overset{+}{\underset{s}{\longrightarrow}} d$.

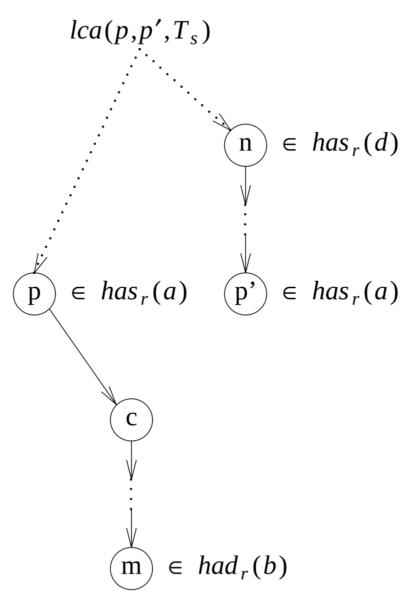


Figure 10.

T_r prior to a restricted *switch* resulting in $n \in ancestors(m,T_s)$.

For a *switch*′′, $\Lambda(p, p', T_s) = \{p, p'\}$. Thus, o is not a *switch*′′. Since C is switch-restricted, o is therefore a *switch*′, and $c \notin had_s(a)$. Then $b \underset{s}{\twoheadrightarrow} a$ by the path from p' to c to m. Thus, $b \underset{s}{\twoheadrightarrow} a \overset{+}{\underset{s}{\longrightarrow}} d$, and $b \mathop{}\limits^{+}_{s} d$ whether o is a *lock_first* or a *switch*.

For $d \overset{+}{\underset{fin}{\longrightarrow}} b$ to hold in C, it must hold in C'. Then, since C' is writer-only and switch-restricted, $d \mathop{}\limits^{+}_{s} b$ must hold by lemma 4. But by lemma 7 and the previous result, this is impossible. □

We should note that, as illustrated by the counterexample in Fig. 11, the condition that C be switch-restricted really is necessary for Theorem 4 to hold.

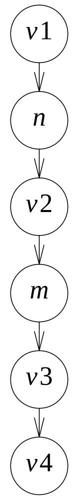
The final installment of the order preservation story requires some preparatory work.

Lemma 9: Let p be the parent of some node c in some state r of a dynamic tree computation, and let p' be the parent of c in some subsequent state s. If $p \in had_r(a)$ for some action a, then there exists some action a' such that $p' \in had_s(a')$, and $a \overset{*}{\underset{s}{\longrightarrow}} a'$.

Proof: Trivial if $p = p'$. If $p \neq p'$, then between r and s occurred the operations $w_i = switch_{e_i}(p_{i-1}, p_i, c)$, $1 \leq i \leq k$, where $p_0 = p$ and $p_k = p'$. Since e_1 held a lock on p as of w_1, and e_1 is a writer, either $a = e_1$, or $a \underset{s}{\longrightarrow} e_1$. And whenever $e_i \neq e_{i+1}$, e_i held a lock on p_i before e_{i+1}, therefore $e_i \underset{s}{\longrightarrow} e_{i+1}$. Thus, $a \overset{*}{\underset{s}{\longrightarrow}} e_k$. Since $p' \in had_s(e_k)$, the lemma holds for $a' = e_k$. □

Lemma 10: If $d \underset{s}{\twoheadrightarrow} a$ by a path from n to m, and, in some previous state, some action $b \neq a$ held a lock on the parent of m, then $b \mathop{}\limits^{+}_{s} a$.

Proof: Let p be the parent of m in s. By lemma 9, there exists a b' such that $b \overset{*}{\underset{s}{\longrightarrow}} b'$ and $p \in had_s(b')$ (see Fig. 12). If $b' = a$, we are done. Thus, assume $b' \neq a$. If $n \neq p$, then $b' \underset{s}{\twoheadrightarrow} a$ by the path from n to p. And if $n = p$, $b' \underset{s}{\longrightarrow} a$ at n. Thus, $b \overset{*}{\underset{s}{\longrightarrow}} b' \mathop{}\limits_{s} a$. □



T_{init}

e	*a*	*b*	*d*
lock_first(*v*1)			
lock_child(*v*1,*n*)			
lock_child(*n*,*v*2)			
lock_child(*v*2,*m*)			
lock_child(*m*,*v*3)			
lock_child(*v*3,*v*4)			
unlock(*n*)			
unlock(*m*)			
unlock(*v*3)			
	lock_first(*m*)		
	lock_child(*m*,*v*3)		
	unlock(*m*)		
		lock_first(*m*)	
		unlock(*m*)	
			lock_first(*n*)
			unlock(*n*)
switch(*v*2,*v*1,*m*)			
switch(*v*1,*v*4,*n*)			
unlock(*v*4)			
	lock_child(*v*3,*v*4)		
	lock_child(*v*4,*n*)		

Figure 11.

Because the computation is not switch-restricted,

$$d \underset{fin}{\longrightarrow} a \underset{fin}{\longrightarrow} b$$

even though $b \twoheadrightarrow d$ right after *d*'s *lock_first*,
and even though *b* completely precedes *d*.

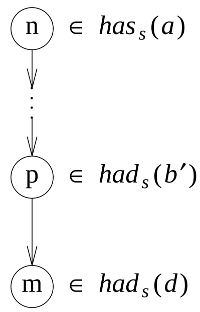


Figure 12.

Theorem 5: Let C be a computation of the segregated tree protocol wherein action b completely precedes action d. If, in some state q of C, some node $p \in had_q(b)$ is the parent of node $c \notin had_q(d)$, but $c \in had_{fin}(d)$, then $d \overset{+}{\underset{fin}{\longrightarrow}} b$ does not hold.

Proof: Let us construct C' from C as in Theorem 3, and let us further modify C' by replacing every $remove\_leaf_e(p,c)$ with an $unlock_e(c)$. By the same simple argument as in the proof of lemma 7, C' is still a legal dynamic tree computation, and $d \overset{+}{\underset{fin}{\longrightarrow}} b$ holds in C only if it holds in C'. The terms of the theorem must also still apply to C'.

Since $c \in had_{fin}(d)$, d executed either a $lock\_first_d(c)$ or a $lock\_child_d(p',c)$ for some p'. (An $add\_leaf_d(p',c)$ is impossible since c was in the tree before d locked it.)

First, consider the $lock\_child$ case. Let s be the state right before the $lock\_child$. By lemma 9, $p' \in had_s(b')$ for some b' such that $b \overset{*}{\underset{s}{\longrightarrow}} b'$. But since $p' \in has_s(d)$, $b' \overset{*}{\underset{s}{\longrightarrow}} d$. Thus, since $b \neq d$, $b \overset{+}{\underset{s}{\longrightarrow}} d$, thus $b \overset{+}{\underset{fin}{\longrightarrow}} d$. Therefore, by lemma 7, $d \overset{+}{\underset{fin}{\longrightarrow}} b$ can not hold.

Now, consider the $lock\_first_d(c)$ case. Assume $d \overset{+}{\underset{fin}{\longrightarrow}} b$ and proceed by contradiction. Let s be the first state in C' such that $d \underset{s}{\overset{+}{}} b$. Note that q comes before s. Let o be the operation preceding s. By lemma

3, o must be either a $lock\_first_e$ or a non-restricted $switch_e$ for some action e.

First, let o be a $lock\_first_e$. Then, by lemma 2, $e = d$, and o is $lock\_first_d(c)$. Note that $d \underset{s}{\twoheadrightarrow} b$ is impossible since $has_s(b) = \emptyset$. Also note that $d \underset{s}{\longrightarrow} a$ is impossible for any a since d has yet to unlock any node. Thus, $d \underset{s}{\twoheadrightarrow} a \;{}^{+}_{s}\; b$ for some a. Since $had_s(d) = \{c\}$, $d \underset{s}{\twoheadrightarrow} a$ by a path from some n to c. But then, by lemma 10, $b \;{}^{+}_{s}\; a$, which contradicts lemma 7.

Now, let o be a $switch_e(p_e, p'_e, c_e)$, and let r be the state preceding o. Let d also be known as a_0, b as a_k, and let $a_0 \;{}_{s}\; a_1 \;{}_{s}\; \cdots \;{}_{s}\; a_k$ be the shortest-length path connecting d to b in ${}_{s}$. Note that $d \underset{s}{\twoheadrightarrow} b$ is impossible since $has_s(b) = \emptyset$, and $d \underset{s}{\longrightarrow} b$ is impossible since b completely precedes d. Thus, $k > 1$.

By lemma 6 part 4, if $a_i \;{}^{+}_{r}\; a_{i+1}$ does not hold for some i between 1 and $k-1$ inclusive, then $a_{i-1} \;{}_{s}\; a_{i+1}$. But then the path was not of minimal length, which is a contradiction. Thus, $a_1 \;{}^{+}_{r}\; b$. And since $d \;{}^{+}_{r}\; b$ does not hold, neither does $d \;{}_{r}\; a_1$.

Thus, by lemma 6, part 1, $d \underset{s}{\twoheadrightarrow} a_1$ by a path from n to m that includes (p'_e, c_e). By part 2 of lemma 6, c is in the subtree dominated by c_e. Thus, $d \underset{s}{\twoheadrightarrow} a_1$ also holds by the path from n to c_e to c. But then, by lemma 10, $b \;{}^{+}_{s}\; a_1$, which contradicts lemma 7. □

9. Conclusion

The Silberschatz and Kedem tree protocol can be extended to dynamic trees by allowing the general-purpose operations *switch*, *add_leaf*, and *remove_leaf*. The resulting protocol is serializable in both its exclusive-lock only and segregated varieties, and is no less deadlock-free then the original tree protocol. However, unless the *switch* operation is further restricted, it no longer has the original protocol's property of determining an action's position in the serialization ordering as soon as the action locks its first node. We have also explored under what circumstances either protocol guarantees order preservation, and found that these too can depend on restricting the *switch* operation.

REFERENCES

[BS77] BAYER, R. and SCHKOLNIK, M. Concurrency of operations on B-trees. Acta Informatica 9,1 (January 1977) 1-21.

[KS83] KEDEM, Z. and SILBERSCHATZ, A. Locking protocols: from exclusive to shared locks. J. of the ACM 30,4 (October 1983) 787-804

[MR85] MOND, Y. and RAZ, Y. Concurrency control in B^+-tree databases using preparatory operations. Proc. of International Conference on Very Large Data Bases, 1985, 331-334

[Sa76] SAMADI, B. B-trees in systems with multiple users. Inf. Processing Letters 5, 4 (October 1976) 107-112

[SK80] SILBERSCHATZ, A. and KEDEM, Z. Consistency in hierarchical database systems. J. of the ACM 27,1 (January 1980) 72-80