locking the root node of the B-tree. If the transaction is 2PL, then it cannot unlock the root until it has acquired all the locks it needs, both on B-tree nodes and other database elements.[7] Moreover, since in principle any transaction that inserts or deletes could wind up rewriting the root of the B-tree, the transaction needs at least an update lock on the root node, or an exclusive lock if update mode is not available. Thus, only one transaction that is not read-only can access the B-tree at any time.

However, in most situations, we can deduce almost immediately that a B-tree node will not be rewritten, even if the transaction inserts or deletes a tuple. For example, if the transaction inserts a tuple, but the child of the root that we visit is not completely full, then we know the insertion cannot propagate up to the root. Similarly, if the transaction deletes a single tuple, and the child of the root we visit has more than the minimum number of keys and pointers, then we can be sure the root will not change.

Thus, as soon as a transaction moves to a child of the root and observes the (quite usual) situation that rules out a rewrite of the root, we would like to release the lock on the root. The same observation applies to the lock on any interior node of the B-tree. Unfortunately, releasing the lock on the root early will violate 2PL, so we cannot be sure that the schedule of several transactions accessing the B-tree will be serializable. The solution is a specialized protocol for transactions that access tree-structured data such as B-trees. The protocol violates 2PL, but uses the fact that accesses to elements must proceed down the tree to assure serializability.

18.7.2 Rules for Access to Tree-Structured Data

The following restrictions on locks form the *tree protocol*. We assume that there is only one kind of lock, represented by lock requests of the form $l_i(X)$, but the idea generalizes to any set of lock modes. We assume that transactions are consistent, and schedules must be legal (i.e., the scheduler will enforce the expected restrictions by granting locks on a node only when they do not conflict with locks already on that node), but there is no two-phase locking requirement on transactions.

1. A transaction's first lock may be at any node of the tree.[8]

2. Subsequent locks may only be acquired if the transaction currently has a lock on the parent node.

3. Nodes may be unlocked at any time.

4. A transaction may not relock a node on which it has released a lock, even if it still holds a lock on the node's parent.

[7] Additionally, there are good reasons why a transaction will hold all its locks until it is ready to commit; see Section 19.1.

[8] In the B-tree example of Section 18.7.1, the first lock would always be at the root.

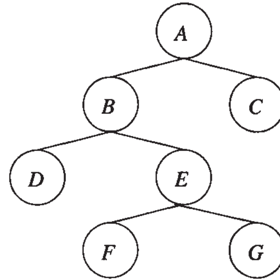

Figure 18.30: A tree of lockable elements

Example 18.23: Figure 18.30 shows a hierarchy of nodes, and Fig. 18.31 indicates the action of three transactions on this data. T_1 starts at the root A, and proceeds downward to B, C, and D. T_2 starts at B and tries to move to E, but its move is initially denied because of the lock by T_3 on E. Transaction T_3 starts at E and moves to F and G. Notice that T_1 is not a 2PL transaction, because the lock on A is relinquished before the lock on D is acquired. Similarly, T_3 is not a 2PL transaction, although T_2 happens to be 2PL. □

18.7.3 Why the Tree Protocol Works

The tree protocol implies a serial order on the transactions involved in a schedule. We can define an order of precedence as follows. Say that $T_i <_S T_j$ if in schedule S, the transactions T_i and T_j lock a node in common, and T_i locks the node first.

Example 18.24: In the schedule S of Fig 18.31, we find T_1 and T_2 lock B in common, and T_1 locks it first. Thus, $T_1 <_S T_2$. We also find that T_2 and T_3 lock E in common, and T_3 locks it first; thus $T_3 <_S T_2$. However, there is no precedence between T_1 and T_3, because they lock no node in common. Thus, the precedence graph derived from these precedence relations is as shown in Fig. 18.32. □

If the precedence graph drawn from the precedence relations that we defined above has no cycles, then we claim that any topological order of the transactions is an equivalent serial schedule. For example, either (T_1, T_3, T_2) or (T_3, T_1, T_2) is an equivalent serial schedule for Fig. 18.31. The reason is that in such a serial schedule, all nodes are touched in the same order as they are in the original schedule.

To understand why the precedence graph described above must always be acyclic if the tree protocol is obeyed, observe the following:

- If two transactions lock several elements in common, then they are all locked in the same order.

T_1	T_2	T_3
$l_1(A)$; $r_1(A)$;		
$l_1(B)$; $r_1(B)$;		
$l_1(C)$; $r_1(C)$;		
$w_1(A)$; $u_1(A)$;		
$l_1(D)$; $r_1(D)$;		
$w_1(B)$; $u_1(B)$;		
	$l_2(B)$; $r_2(B)$;	
		$l_3(E)$; $r_3(E)$;
$w_1(D)$; $u_1(D)$;		
$w_1(C)$; $u_1(C)$;		
	$l_2(E)$ **Denied**	
		$l_3(F)$; $r_3(F)$;
		$w_3(F)$; $u_3(F)$;
		$l_3(G)$; $r_3(G)$
		$w_3(E)$; $u_3(E)$;
	$l_2(E)$; $r_2(E)$;	
		$w_3(G)$; $u_3(G)$
	$w_2(B)$; $u_2(B)$;	
	$w_2(E)$; $u_2(E)$;	

Figure 18.31: Three transactions following the tree protocol

To see why, consider some transactions T and U, which lock two or more items in common. First, notice that each transaction locks a set of elements that form a tree, and the intersection of two trees is itself a tree. Thus, there is some one highest element X that both T and U lock. Suppose that T locks X first, but that there is some other element Y that U locks before T. Then there is a path in the tree of elements from X to Y, and both T and U must lock each element along the path, because neither can lock a node without having a lock on its parent.

Consider the first element along this path, say Z, that U locks first, as suggested by Fig. 18.33. Then T locks the parent P of Z before U does. But then T is still holding the lock on P when it locks Z, so U has not yet locked

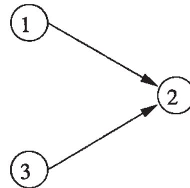

Figure 18.32: Precedence graph derived from the schedule of Fig. 18.31

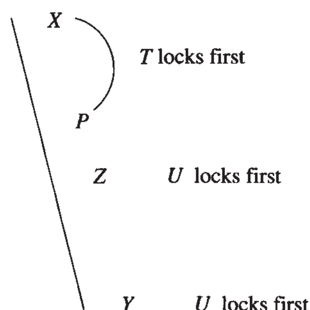

Figure 18.33: A path of elements locked by two transactions

P when it locks Z. It cannot be that Z is the first element U locks in common with T, since they both lock ancestor X (which could also be P, but not Z). Thus, U cannot lock Z until after it has acquired a lock on P, which is after T locks Z. We conclude that T precedes U at every node they lock in common.

Now, consider an arbitrary set of transactions $T_1, T_2, \ldots, T_n$ that obey the tree protocol and lock some of the nodes of a tree according to schedule S. First, among those that lock the root, they do so in some order, and by the rule just observed:

- If T_i locks the root before T_j, then T_i locks every node in common with T_j before T_j does. That is, $T_i <_S T_j$, but not $T_j <_S T_i$.

We can show by induction on the number of nodes of the tree that there is some serial order equivalent to S for the complete set of transactions.

BASIS: If there is only one node, the root, then as we just observed, the order in which the transactions lock the root serves.

INDUCTION: If there is more than one node in the tree, consider for each subtree of the root the set of transactions that lock one or more nodes in that subtree. Note that transactions locking the root may belong to more than one subtree, but a transaction that does not lock the root will belong to only one subtree. For instance, among the transactions of Fig. 18.31, only T_1 locks the root, and it belongs to both subtrees — the tree rooted at B and the tree rooted at C. However, T_2 and T_3 belong only to the tree rooted at B.

By the inductive hypothesis, there is a serial order for all the transactions that lock nodes in any one subtree. We have only to blend the serial orders for the various subtrees. Since the only transactions these lists of transactions have in common are the transactions that lock the root, and we established that these transactions lock every node in common in the same order that they lock the root, it is not possible that two transactions locking the root appear in different orders in two of the sublists. Specifically, if T_i and T_j appear on the list for some child C of the root, then they lock C in the same order as they lock

the root and therefore appear on the list in that order. Thus, we can build a serial order for the full set of transactions by starting with the transactions that lock the root, in their appropriate order, and interspersing those transactions that do not lock the root in any order consistent with the serial order of their subtrees.

Example 18.25: Suppose there are 10 transactions $T_1, T_2, \ldots, T_{10}$, and of these, T_1, T_2, and T_3 lock the root in that order. Suppose also that there are two children of the root, the first locked by T_1 through T_7 and the second locked by T_2, T_3, T_8, T_9, and T_{10}. Hypothetically, let the serial order for the first subtree be $(T_4, T_1, T_5, T_2, T_6, T_3, T_7)$; note that this order must include T_1, T_2, and T_3 in that order. Also, let the serial order for the second subtree be $(T_8, T_2, T_9, T_{10}, T_3)$. As must be the case, the transactions T_2 and T_3, which locked the root, appear in this sequence in the order in which they locked the root.

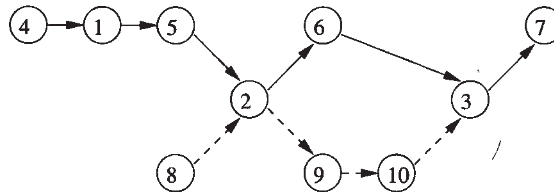

Figure 18.34: Combining serial orders for the subtrees into a serial order for all transactions

The constraints imposed on the serial order of these transactions are as shown in Fig. 18.34. Solid lines represent constraints due to the order at the first child of the root, while dashed lines represent the order at the second child. $(T_4, T_8, T_1, T_5, T_2, T_9, T_6, T_{10}, T_3, T_7)$ is one of the many topological sorts of this graph. □

18.7.4 Exercises for Section 18.7

Exercise 18.7.1: Suppose we perform the following actions on the B-tree of Fig. 14.13. If we use the tree protocol, when can we release a write-lock on each of the nodes searched?

(a) Insert 10 (b) Insert 20 (c) Delete 5 (d) Delete 23.

! **Exercise 18.7.2:** Consider the following transactions that operate on the tree of Fig. 18.30.

T_1: $r_1(A)$; $r_1(B)$; $r_1(E)$;
T_2: $r_2(A)$; $r_2(C)$; $r_2(B)$;
T_3: $r_3(B)$; $r_3(E)$; $r_3(F)$;