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.

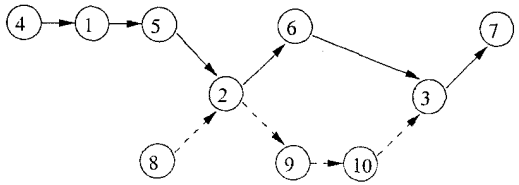

Figure 18.34: Combining serial orders for the subtrees into a serial order for all transactions

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. 13.23. 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: $r_3(B)$; $r_3(E)$; $r_3(F)$;

If schedules follow the tree protocol, in how many ways can we interleave: *a) T_1 and T_2 b) T_1 and T_3 !! c) all three?

! **Exercise 18.7.3:** Suppose there are eight transactions $T_1, T_2, \ldots, T_8$, of which the odd-numbered transactions, T_1, T_3, T_5, and T_7, lock the root of a tree, in that order. There are three children of the root, the first locked by T_1, T_2, T_3, and T_4 in that order. The second child is locked by T_3, T_6, and T_5, in that order, and the third child is locked by T_8 and T_7, in that order. How many serial orders of the transactions are consistent with these statements?

!! **Exercise 18.7.4:** Suppose we use the tree protocol with shared and exclusive locks for reading and writing, respectively. Rule (2), which requires a lock on the parent to get a lock on a node, must be changed to prevent unserializable behavior. What is the proper rule (2) for shared and exclusive locks? *Hint*: Does the lock on the parent have to be of the same type as the lock on the child?

18.8 Concurrency Control by Timestamps

Next, we shall consider two methods other than locking that are used in some systems to assure serializability of transactions:

1. *Timestamping.* We assign a "timestamp" to each transaction, record the timestamps of the transactions that last read and write each database element, and compare these values to assure that the serial schedule according to the transactions' timestamps is equivalent to the actual schedule of the transactions. This approach is the subject of the present section.

2. *Validation.* We examine timestamps of the transaction and the database elements when a transaction is about to commit; this process is called "validation" of the transaction. The serial schedule that orders transactions according to their validation time must be equivalent to the actual schedule. The validation approach is discussed in Section 18.9.

Both these approaches are *optimistic*, in the sense that they assume that no unserializable behavior will occur and only fix things up when a violation is apparent. In contrast, all locking methods assume that things will go wrong

unless transactions are prevented in advance from engaging in nonserializable behavior. The optimistic approaches differ from locking in that the only remedy when something does go wrong is to abort and restart a transaction that tries to engage in unserializable behavior. In contrast, locking schedulers delay transactions, but do not abort them.[11] Generally, optimistic schedulers are better than locking when many of the transactions are read-only, since those transactions can never by themselves cause unserializable behavior.

18.8.1 Timestamps

In order to use timestamping as a concurrency-control method, the scheduler needs to assign to each transaction T a unique number, its *timestamp* $\text{TS}(T)$. Timestamps must be issued in ascending order, at the time that a transaction first notifies the scheduler that it is beginning. Two approaches to generating timestamps are:

a) One possible way to create timestamps is to use the system clock, provided the scheduler does not operate so fast that it could assign timestamps to two transactions on one tick of the clock.

b) Another approach is for the scheduler to maintain a counter. Each time a transaction starts, the counter is incremented by 1, and the new value becomes the timestamp of the transaction. In this approach, timestamps have nothing to do with "time," but they have the important property that we need for any timestamp-generating system: a transaction that starts later has a higher timestamp than a transaction that starts earlier.

Whatever method of generating timestamps is used, the scheduler must maintain a table of currently active transactions and their timestamps.

To use timestamps as a concurrency-control method, we need to associate with each database element X two timestamps and an additional bit:

1. $\text{RT}(X)$, the *read time* of X, which is the highest timestamp of a transaction that has read X.

2. $\text{WT}(X)$, the *write time* of X, which is the highest timestamp of a transaction that has written X.

3. $\text{C}(X)$, the *commit bit* for X, which is true if and only if the most recent transaction to write X has already committed. The purpose of this bit is to avoid a situation where one transaction T reads data written by another transaction U, and U then aborts. This problem, where T makes a "dirty read" of uncommitted data, certainly can cause the database

[11] That is not to say that a system using a locking scheduler will never abort a transaction; for instance, Section 19.3 discusses aborting transactions to fix deadlocks. However, a locking scheduler never uses a transaction abort simply as a response to a lock request that it cannot grant.

state to become inconsistent, and any scheduler needs a mechanism to prevent dirty reads.[12]

18.8.2 Physically Unrealizable Behaviors

In order to understand the architecture and rules of a timestamp-based scheduler, we need to remember that the scheduler assumes that the timestamp order of transactions is also the serial order in which they must appear to execute. Thus, the job of the scheduler, in addition to assigning timestamps and updating RT, WT, and C for the database elements, is to check that whenever a read or write occurs, what happens in real time *could* have happened if each transaction had executed instantaneously at the moment of its timestamp. If not, we say the behavior is *physically unrealizable*. There are two kinds of problems that can occur:

1. *Read too late*: Transaction T tries to read database element X, but the write time of X indicates that the current value of X was written after T theoretically executed; that is, $\text{TS}(T) < \text{WT}(X)$. Figure 18.35 illustrates the problem. The horizontal axis represents the real time at which events occur. Dotted lines link the actual events to the times at which they theoretically occur — the timestamp of the transaction that performs the event. Thus, we see a transaction U that started after transaction T, but wrote a value for X before T reads X. T should not be able to read the value written by U, because theoretically, U executed after T did. However, T has no choice, because U's value of X is the one that T now sees. The solution is to abort T when the problem is encountered.

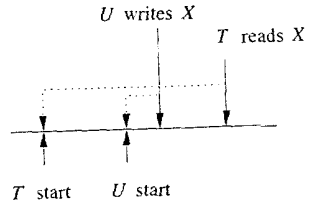

Figure 18.35: Transaction T tries to read too late

2. *Write too late*: Transaction T tries to write database element X, but the read time of X indicates that some other transaction should have read the value written by T but read some other value instead. That is, $\text{WT}(X) < \text{TS}(T) < \text{RT}(X)$. The problem is shown in Fig. 18.36. There we see a transaction U that started after T, but read X before T got a chance to write X. When T tries to write X, we find $\text{RT}(X) > \text{TS}(T)$, meaning that X has already been read by a transaction U that theoretically executed

[12] Although commercial systems generally give the user an option to allow dirty reads.

later than T. We also find $\text{WT}(X) < \text{TS}(T)$, which means that no other transaction wrote into X a value that would have overwritten T's value. thus, negating T's responsibility to get its value into X so transaction U could read it.

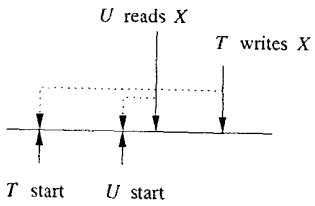

Figure 18.36: Transaction T tries to write too late

18.8.3 Problems With Dirty Data

There is a class of problems that the commit bit is designed to help deal with. One of these problems, a "dirty read," is suggested in Fig. 18.37. There, transaction T reads X, and X was last written by U. The timestamp of U is less than that of T, and the read by T occurs after the write by U in real time. so the event seems to be physically realizable. However, it is possible that after T reads the value of X written by U, transaction U will abort; perhaps U encounters an error condition in its own data, such as a division by 0. or as we shall see in Section 18.8.4, the scheduler forces U to abort because it tries to do something physically unrealizable. Thus, although there is nothing physically unrealizable about T reading X, it is better to delay T's read until U commits or aborts. We can tell that U is not committed because the commit bit $\text{C}(X)$ will be false.

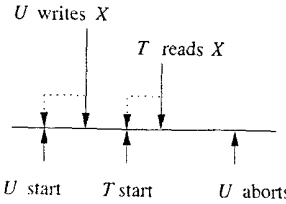

Figure 18.37: T could perform a dirty read if it reads X when shown

A second potential problem is suggested by Fig. 18.38. Here, U, a transaction with a later timestamp than T, has written X first. When T tries to write, the appropriate action is to do nothing. Evidently no other transaction V that should have read T's value of X got U's value instead, because if V

tried to read X it would have aborted because of a too-late read. Future reads of X will want U's value or a later value of X, not T's value. This idea, that writes can be skipped when a write with a later write-time is already in place, is called the *Thomas write rule*.

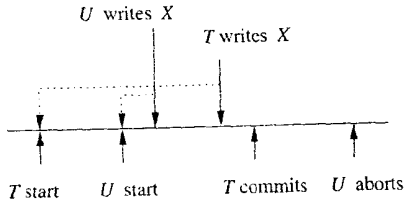

Figure 18.38: A write is cancelled because of a write with a later timestamp, but the writer then aborts

There is a potential problem with the Thomas write rule, however. If U later aborts, as is suggested in Fig. 18.38, then its value of X should be removed and the previous value and write-time restored. Since T is committed, it would seem that the value of X should be the one written by T for future reading. However, we already skipped the write by T and it is too late to repair the damage.

While there are many ways to deal with the problems just described, we shall adopt a relatively simple policy based on the following assumed capability of the timestamp-based scheduler.

- When a transaction T writes a database element X, the write is "tentative" and may be undone if T aborts. The commit bit $C(X)$ is set to false, and the scheduler makes a copy of the old value of X and its previous $\text{WT}(X)$.

18.8.4 The Rules for Timestamp-Based Scheduling

We can now summarize the rules that a scheduler using timestamps must follow to make sure that nothing physically unrealizable may occur. The scheduler, in response to a read or write request from a transaction T has the choice of:

a) Granting the request.

b) Aborting T (if T would violate physical reality) and restarting T with a new timestamp (abort followed by restart is often called *rollback*), or

c) Delaying T and later deciding whether to abort T or to grant the request (if the request is a read. and the read might be dirty, as in Section 18.8.3).

The rules are as follows:

1. Suppose the scheduler receives a request $r_T(X)$.
   (a) If $\text{TS}(T) \geq \text{WT}(X)$, the read is physically realizable.
      i. If $\text{C}(X)$ is true, grant the request. If $\text{TS}(T) > \text{RT}(X)$, set $\text{RT}(X) := \text{TS}(T)$; otherwise do not change $\text{RT}(X)$.
      ii. If $\text{C}(X)$ is false, delay T until $\text{C}(X)$ becomes true or the transaction that wrote X aborts.
   (b) If $\text{TS}(T) < \text{WT}(X)$, the read is physically unrealizable. Rollback T; that is, abort T and restart it with a new, larger timestamp.
2. Suppose the scheduler receives a request $w_T(X)$.
   (a) If $\text{TS}(T) \geq \text{RT}(X)$ and $\text{TS}(T) \geq \text{WT}(X)$, the write is physically realizable and must be performed.
      i. Write the new value for X,
      ii. Set $\text{WT}(X) := \text{TS}(T)$, and
      iii. Set $\text{C}(X) :=$ `false`.
   (b) If $\text{TS}(T) \geq \text{RT}(X)$, but $\text{TS}(T) < \text{WT}(X)$, then the write is physically realizable, but there is already a later value in X. If $C(X)$ is true, then the previous writer of X is committed, and we simply ignore the write by T; we allow T to proceed and make no change to the database. However, if $C(X)$ is false, then we must delay T as in point 1(a)ii.
   (c) If $\text{TS}(T) < \text{RT}(X)$, then the write is physically unrealizable, and T must be rolled back.
3. Suppose the scheduler receives a request to commit T. It must find (using a list the scheduler maintains) all the database elements X written by T, and set $\text{C}(X) :=$ `true`. If any transactions are waiting for X to be committed (found from another scheduler-maintained list), these transactions are allowed to proceed.
4. Suppose the scheduler receives a request to abort T or decides to rollback T as in 1b or 2c. Then any transaction that was waiting on an element X that T wrote must repeat its attempt to read or write, and see whether the action is now legal after the aborted transaction's writes are cancelled.

Example 18.26: Figure 18.39 shows a schedule of three transactions, T_1, T_2, and T_3 that access three database elements, A, B, and C. The real time at which events occur increases down the page, as usual. However, we have also indicated the timestamps of the transactions and the read and write times of the elements. We assume that at the beginning, each of the database elements has both a read and write time of 0. The timestamps of the transactions are acquired when they notify the scheduler that they are beginning. Notice that even though T_1 executes the first data access, it does not have the least

T_1	T_2	T_3	A	B	C
200	150	175	RT=0	RT=0	RT=0
			WT=0	WT=0	WT=0
				RT=200	
$r_1(B)$;					
	$r_2(A)$;		RT=150		
		$r_3(C)$;			RT=175
				WT=200	
$w_1(B)$;					
$w_1(A)$;			WT=200		
	$w_2(C)$;				
	Abort;				
		$w_3(A)$;			

Figure 18.39: Three transactions executing under a timestamp-based scheduler

timestamp. Presumably T_2 was the first to notify the scheduler of its start, and T_3 did so next, with T_1 last to start.

In the first action, T_1 reads B. Since the write time of B is less than the timestamp of T_1, this read is physically realizable and allowed to happen. The read time of B is set to 200, the timestamp of T_1. The second and third read actions similarly are legal and result in the read time of each database element being set to the timestamp of the transaction that read it.

At the fourth step, T_1 writes B. Since the read time of B is not bigger than the timestamp of T_1, the write is physically realizable. Since the write time of B is no larger than the timestamp of T_1, we must actually perform the write. When we do, the write time of B is raised to 200, the timestamp of the writing transaction T_1.

Next, T_2 tries to write C. However, C was already read by transaction T_3, which theoretically executed at time 175, while T_2 would have written its value at time 150. Thus, T_2 is trying to do something that would result in physically unrealizable behavior, and T_2 must be rolled back.

The last step is the write of A by T_3. Since the read time of A, 150, is less than the timestamp of T_3, 175, the write is legal. However, there is already a later value of A stored in that database element, namely the value written by T_1, theoretically at time 200. Thus, T_3 is not rolled back, but neither does it write its value. □

18.8.5 Multiversion Timestamps

An important variation of timestamping maintains old versions of database elements in addition to the current version that is stored in the database itself. The purpose is to allow reads $r_T(X)$ that otherwise would cause transaction T to abort (because the current version of X was written in T's future) to proceed by reading the version of X that is appropriate for a transaction with

T's timestamp. The method is especially useful if database elements are disk blocks or pages, since then all that must be done is for the buffer manager to keep in memory certain blocks that might be useful for some currently active transaction.

Example 18.27: Consider the set of transactions accessing database element A shown in Fig. 18.40. These transactions are operating under an ordinary timestamp-based scheduler, and when T_3 tries to read A, it finds WT(A) to be greater than its own timestamp, and must abort. However, there is an old value of A written by T_1 and overwritten by T_2 that would have been suitable for T_3 to read; this version of A had a write time of 150, which is less than T_3's timestamp of 175. If this old value of A were available, T_3 could be allowed to read it, even though it is not the "current" value of A. □

T_1	T_2	T_3	T_4	A
150	200	175	225	RT=0
				WT=0
$r_1(A)$				RT=150
$w_1(A)$				WT=150
	$r_2(A)$			RT=200
	$w_2(A)$			WT=200
		$r_3(A)$		
		Abort		
			$r_4(A)$	RT=225

Figure 18.40: T_3 must abort because it cannot access an old value of A

A multiversion timestamping scheduler differs from the scheduler described in Section 18.8.4 in the following ways:

1. When a new write $w_T(X)$ occurs, if it is legal, then a new version of database element X is created. Its write time is TS(T), and we shall refer to it as X_t, where t = TS(T).

2. When a read $r_T(X)$ occurs, the scheduler finds the version X_t of X such that $t \leq$ TS(T), but there is no other version $X_{t'}$ with $t < t' \leq$ TS(T). That is, the version of X written immediately before T theoretically executed is the version that T reads.

3. Write times are associated with *versions* of an element, and they never change.

4. Read times are also associated with versions. They are used to reject certain writes, namely one whose time is less than the read time of the

previous version. Figure 18.41 suggests the problem, where X has versions X_{50} and X_{100}, the former was read at time 80, and a new write by a transaction T whose timestamp is 60 occurs. This write must cause T to abort, because its value of X should have been read by the transaction with timestamp 80, had T been allowed to execute.

5. When a version X_t has a write time t such that no active transaction has a timestamp less than t, then we may delete any version of X *previous* to X_t.

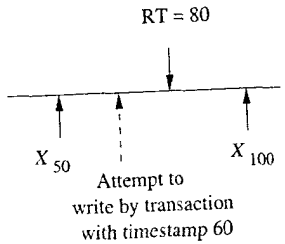

Figure 18.41: A transaction tries to write a version of X that would make events physically unrealizable

Example 18.28: Let us reconsider the actions of Fig. 18.40 if multiversion timestamping is used. First, there are three versions of A: A_0, which exists before these transactions start, A_{150}, written by T_1, and A_{200}, written by T_2. Figure 18.42 shows the sequence of events, when the versions are created, and when they are read. Notice in particular that T_3 does not have to abort, because it can read an earlier version of A. □

T_1	T_2	T_3	T_4	A_0	A_{150}	A_{200}
150	200	175	225			
				Read		
$r_1(A)$					Create	
$w_1(A)$					Read	
	$r_2(A)$					Create
	$w_2(A)$				Read	
		$r_3(A)$				Read
			$r_4(A)$			

Figure 18.42: Execution of transactions using multiversion concurrency control