18.1.5 A Notation for Transactions and Schedules

If we assume "no coincidences," then only the reads and writes performed by the transaction matter, not the actual values involved. Thus, we shall represent transactions and schedules by a shorthand notation, in which the actions are $r_T(X)$ and $w_T(X)$, meaning that transaction T reads, or respectively writes, database element X. Moreover, since we shall usually name our transactions $T_1, T_2, \ldots$, we adopt the convention that $r_i(X)$ and $w_i(X)$ are synonyms for $r_{T_i}(X)$ and $w_{T_i}(X)$, respectively.

Example 18.5: The transactions of Fig. 18.2 can be written:

$$T_1\text{: } r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);$$
$$T_2\text{: } r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);$$

As another example,

$$r_1(A);\ w_1(A);\ r_2(A);\ w_2(A);\ r_1(B);\ w_1(B);\ r_2(B);\ w_2(B);$$

is the serializable schedule from Fig. 18.5. □

To make the notation precise:

1. An *action* is an expression of the form $r_i(X)$ or $w_i(X)$, meaning that transaction T_i reads or writes, respectively, the database element X.

2. A *transaction* T_i is a sequence of actions with subscript i.

3. A *schedule* S of a set of transactions $\mathcal{T}$ is a sequence of actions, in which for each transaction T_i in $\mathcal{T}$, the actions of T_i appear in S in the same order that they appear in the definition of T_i itself. We say that S is an *interleaving* of the actions of the transactions of which it is composed.

For instance, the schedule of Example 18.5 has all the actions with subscript 1 appearing in the same order that they have in the definition of T_1, and the actions with subscript 2 appear in the same order that they appear in the definition of T_2.

18.1.6 Exercises for Section 18.1

Exercise 18.1.1: A transaction T_1, executed by an airline-reservation system, performs the following steps:

i. The customer is queried for a desired flight time and cities. Information about the desired flights is located in database elements (perhaps disk blocks) A and B, which the system retrieves from disk.

ii. The customer is told about the options, and selects a flight whose data, including the number of reservations for that flight is in B. A reservation on that flight is made for the customer.

iii. The customer selects a seat for the flight; seat data for the flight is in database element C.

iv. The system gets the customer's credit-card number and appends the bill for the flight to a list of bills in database element D.

v. The customer's phone and flight data is added to another list on database element E for a fax to be sent confirming the flight.

Express transaction T_1 as a sequence of r and w actions.

! **Exercise 18.1.2:** If two transactions each consist of 5 actions, how many interleavings of these transactions are there?

18.2 Conflict-Serializability

Schedulers in commercial systems generally enforce a condition, called "conflict-serializability," that is stronger than the general notion of serializability introduced in Section 18.1.3. It is based on the idea of a *conflict*: a pair of consecutive actions in a schedule such that, if their order is interchanged, then the behavior of at least one of the transactions involved can change.

18.2.1 Conflicts

To begin, let us observe that most pairs of actions do *not* conflict. In what follows, we assume that T_i and T_j are different transactions; i.e., $i \neq j$.

1. $r_i(X);\ r_j(Y)$ is never a conflict, even if $X = Y$. The reason is that neither of these steps change the value of any database element.

2. $r_i(X);\ w_j(Y)$ is not a conflict provided $X \neq Y$. The reason is that should T_j write Y before T_i reads X, the value of X is not changed. Also, the read of X by T_i has no effect on T_j, so it does not affect the value T_j writes for Y.

3. $w_i(X);\ r_j(Y)$ is not a conflict if $X \neq Y$, for the same reason as (2).

4. Similarly, $w_i(X);\ w_j(Y)$ is not a conflict as long as $X \neq Y$.

On the other hand, there are three situations where we may not swap the order of actions:

a) Two actions of the same transaction, e.g., $r_i(X);\ w_i(Y)$, always conflict. The reason is that the order of actions of a single transaction are fixed and may not be reordered.

BASIS: If $n = 1$, i.e., there is only one transaction in the schedule, then the schedule is already serial, and therefore surely conflict-serializable.

INDUCTION: Let the schedule S consist of the actions of n transactions

$$T_1, T_2, \dots, T_n$$

We suppose that S has an acyclic precedence graph. If a finite graph is acyclic, then there is at least one node that has no arcs in; let the node i corresponding to transaction T_i be such a node. Since there are no arcs into node i, there can be no action A in S that:

1. Involves any transaction T_j other than T_i,
2. Precedes some action of T_i, and
3. Conflicts with that action.

For if there were, we should have put an arc from node j to node i in the precedence graph.

It is thus possible to swap all the actions of T_i, keeping them in order, but moving them to the front of S. The schedule has now taken the form

$$(\text{Actions of } T_i)(\text{Actions of the other } n-1 \text{ transactions})$$

Let us now consider the tail of S — the actions of all transactions other than T_i. Since these actions maintain the same relative order that they did in S, the precedence graph for the tail is the same as the precedence graph for S, except that the node for T_i and any arcs out of that node are missing.

Since the original precedence graph was acyclic, and deleting nodes and arcs cannot make it cyclic, we conclude that the tail's precedence graph is acyclic. Moreover, since the tail involves $n - 1$ transactions, the inductive hypothesis applies to it. Thus, we know we can reorder the actions of the tail using legal swaps of adjacent actions to turn it into a serial schedule. Now, S itself has been turned into a serial schedule, with the actions of T_i first and the actions of the other transactions following in some serial order. The induction is complete, and we conclude that every schedule with an acyclic precedence graph is conflict-serializable.

18.2.4 Exercises for Section 18.2

Exercise 18.2.1: Below are two transactions, described in terms of their effect on two database elements A and B, which we may assume are integers.

```
T1: READ(A,t); t:=t+2; WRITE(A,t); READ(B,t); t:=t*3; WRITE(B,t);
T2: READ(B,s); s:=s*2; WRITE(B,s); READ(A,s); s:=s+3; WRITE(A,s);
```

We assume that, whatever consistency constraints there are on the database, these transactions preserve them in isolation. Note that $A = B$ is *not* the consistency constraint.

a) Give examples of a serializable schedule and a nonserializable schedule of the 12 actions above.

b) How many serial schedules of the 12 actions are there?

!! c) How many serializable schedules of the 12 actions are there?

d) It turns out that both serial orders have the same effect on the database; that is, (T_1, T_2) and (T_2, T_1) are equivalent. Demonstrate this fact by showing the effect of the two transactions on an arbitrary initial database state.

Exercise 18.2.2: The two transactions of Exercise 18.2.1 can be written in our notation that shows read- and write-actions only, as:

$$T_1\colon\ r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);$$
$$T_2\colon\ r_2(B);\ w_2(B);\ r_2(A);\ w_2(A);$$

Answer the following:

a) Among the possible schedules of the eight actions above, how many are equivalent to the serial order (T_1, T_2)?

! b) How many schedules of the eight actions are conflict-equivalent to the serial order (T_2, T_1)?

!! c) How many schedules of the eight actions are equivalent (not necessarily conflict-equivalent) to the serial schedule (T_1, T_2), assuming the transactions have the effect on the database described in Exercise 18.2.1?

! d) Why are the answers to (c) above and Exercise 18.2.1(c) different?

! **Exercise 18.2.3:** Suppose the transactions of Exercise 18.2.2 are changed to be:

$$T_1\colon\ r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);$$
$$T_2\colon\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);$$

That is, the transactions retain their semantics from Exercise 18.2.1, but T_2 has been changed so A is processed before B. Give:

a) The number of serializable schedules, assuming the transactions have the same effect on the database state as in Exercise 18.2.1.

b) The number of conflict-serializable schedules.

! Exercise 18.2.4: Explain how, for any $n > 1$, one can find a schedule whose precedence graph has a cycle of length n, but no smaller cycle.

Exercise 18.2.5: For each of the following schedules:

a) $w_3(A)$; $r_1(A)$; $w_1(B)$; $r_2(B)$: $w_2(C)$; $r_3(C)$;

b) $r_1(A)$; $r_2(A)$; $w_1(B)$; $w_2(B)$; $r_1(B)$; $r_2(B)$; $w_2(C)$; $w_1(D)$;

c) $r_1(A)$; $r_2(A)$; $r_1(B)$; $r_2(B)$; $r_3(A)$; $r_4(B)$; $w_1(A)$; $w_2(B)$;

d) $r_1(A)$; $r_2(A)$; $r_3(B)$; $w_1(A)$; $r_2(C)$; $r_2(B)$; $w_2(B)$; $w_1(C)$;

e) $r_1(A)$; $w_1(B)$; $r_2(B)$: $w_2(C)$; $r_3(C)$; $w_3(A)$;

Answer the following questions:

i. What is the precedence graph for the schedule?

ii. Is the schedule conflict-serializable? If so, what are all the equivalent serial schedules?

! *iii.* Are there any serial schedules that must be equivalent (regardless of what the transactions do to the data), but are not conflict-equivalent?

!! Exercise 18.2.6: Say that a transaction T *precedes* a transaction U in a schedule S if every action of T precedes every action of U in S. Note that if T and U are the only transactions in S, then saying T precedes U is the same as saying that S is the serial schedule (T, U). However, if S involves transactions other than T and U, then S might not be serializable, and in fact, because of the effect of other transactions, S might not even be conflict-serializable. Give an example of a schedule S such that:

i. In S, T_1 precedes T_2, and

ii. S is conflict-serializable, but

iii. In every serial schedule conflict-equivalent to S, T_2 precedes T_1.

18.3 Enforcing Serializability by Locks

In this section we consider the most common architecture for a scheduler, one in which "locks" are maintained on database elements to prevent unserializable behavior. Intuitively, a transaction obtains locks on the database elements it accesses to prevent other transactions from accessing these elements at roughly the same time and thereby incurring the risk of unserializability.

In this section, we introduce the concept of locking with an (overly) simple locking scheme. In this scheme, there is only one kind of lock, which transactions must obtain on a database element if they want to perform any operation whatsoever on that element. In Section 18.4, we shall learn more realistic locking schemes, with several kinds of lock, including the common shared/exclusive locks that correspond to the privileges of reading and writing, respectively.

A Risk of Deadlock

One problem that is not solved by two-phase locking is the potential for deadlocks, where several transactions are forced by the scheduler to wait forever for a lock held by another transaction. For instance, consider the 2PL transactions from Example 18.11, but with T_2 changed to work on B first:

T_1: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;

T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $u_2(B)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $u_2(A)$;

A possible interleaving of the actions of these transactions is:

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
	$l_2(B)$; $r_2(B)$;		
`A := A+100;`			
	`B := B*2;`		
$w_1(A)$;		125	
	$w_2(B)$;		50
$l_1(B)$ **Denied**	$l_2(A)$ **Denied**		

Now, neither transaction can proceed, and they wait forever. In Section 19.2, we shall discuss methods to remedy this situation. However, observe that it is not possible to allow both transactions to proceed, since if we do so the final database state cannot possibly have $A = B$.

conflict-equivalent serial schedule, and now all of S has been shown conflict-serializable.

18.3.5 Exercises for Section 18.3

Exercise 18.3.1: Below are two transactions, with lock requests and the semantics of the transactions indicated. Recall from Exercise 18.2.1 that these transactions have the unusual property that they can be scheduled in ways that are not conflict-serializable, but, because of the semantics, are serializable.

T_1: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $u_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(B)$;

T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $u_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(A)$;

In the questions below, consider only schedules of the read and write actions, not the lock, unlock, or assignment steps.

a) Give an example of a schedule that is prohibited by the locks.

! b) Of the $\binom{8}{4} = 70$ orders of the eight read and write actions, how many are legal schedules (i.e., they are permitted by the locks)?

! c) Of those schedules that are legal and serializable, how many are conflict-serializable?

! d) Of the legal schedules, how many are serializable (according to the semantics of the transactions given)?

!! e) Since T_1 and T_2 are not two-phase-locked, we would expect that some nonserializable behaviors would occur. Are there any legal schedules that are unserializable? If so, give an example, and if not, explain why.

Exercise 18.3.2: For each of the schedules of Exercise 18.2.5, assume that each transaction takes a lock on each database element immediately before it reads or writes the element, and that each transaction releases its locks immediately after the last time it accesses an element. Tell what the locking scheduler would do with each of these schedules; i.e., what requests would get delayed, and when would they be allowed to resume?

! **Exercise 18.3.3:** Here are the transactions of Exercise 18.3.1, with all unlocks moved to the end so they are two-phase-locked.

T_1: $l_1(A)$; $r_1(A)$; `A := A+2`; $w_1(A)$; $l_1(B)$; $r_1(B)$; `B := B*3`; $w_1(B)$; $u_1(A)$; $u_1(B)$;

T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $r_2(A)$; `A := A+3`; $w_2(A)$; $u_2(B)$; $u_2(A)$;

How many legal schedules of all the read and write actions of these transactions are there?

! **Exercise 18.3.4:** For each of the transactions described below, suppose that we insert one lock and one unlock action for each database element that is accessed.

a) $r_2(A)$; $w_2(A)$; $w_2(B)$;

b) $r_1(A)$; $w_1(B)$;

Tell how many orders of the lock, unlock, read, and write actions are:

i. Consistent and two-phase locked.

ii. Consistent, but not two-phase locked.

iii. Inconsistent, but two-phase locked.

iv. Neither consistent nor two-phase locked.

18.4 Locking Systems With Several Lock Modes

The locking scheme of Section 18.3 illustrates the important ideas behind locking, but it is too simple to be a practical scheme. The main problem is that a transaction T must take a lock on a database element X even if it only wants to read X and not write it. We cannot avoid taking the lock, because if we didn't, then another transaction might write a new value for X while T was active and cause unserializable behavior. On the other hand, there is no reason why several transactions could not read X at the same time, as long as none is allowed to write X.

We are thus motivated to introduce the most common locking scheme, where there are two different kinds of locks, one for reading (called a "shared lock" or "read lock"), and one for writing (called an "exclusive lock" or "write lock"). We then examine an improved scheme where transactions are allowed to take a shared lock and "upgrade" it to an exclusive lock later. We also consider "increment locks," which treat specially write actions that increment a database element; the important distinction is that increment operations commute, while general writes do not. These examples lead us to the general notion of a lock scheme described by a "compatibility matrix" that indicates what locks on a database element may be granted when other locks are held.

18.4.1 Shared and Exclusive Locks

The lock we need for writing is "stronger" than the lock we need to read, since it must prevent both reads and writes. Let us therefore consider a locking scheduler that uses two different kinds of locks: *shared locks* and *exclusive locks*. For any database element X there can be either one exclusive lock on X, or no exclusive locks but any number of shared locks. If we want to write X, we need to have an exclusive lock on X, but if we wish only to read X we may have either a shared or exclusive lock on X. If we want to read X but not write it, it is better to take only a shared lock.

We shall use $sl_i(X)$ to mean "transaction T_i requests a shared lock on database element X" and $xl_i(X)$ for "T_i requests an exclusive lock on X." We continue to use $u_i(X)$ to mean that T_i unlocks X; i.e., it relinquishes whatever lock(s) it has on X.

The three kinds of requirements — consistency and 2PL for transactions, and legality for schedules — each have their counterpart for a shared/exclusive lock system. We summarize these requirements here:

We may move the last action, $inc_1(B)$, to the second position, since it does not conflict with another increment of the same element, and surely does not conflict with a read of a different element. This sequence of swaps shows that S is conflict-equivalent to the serial schedule $r_1(A)$; $inc_1(B)$; $r_2(A)$; $inc_2(B)$;. Similarly, we can move the first action, $r_1(A)$ to the third position by swaps, giving a serial schedule in which T_2 precedes T_1. □

18.4.6 Exercises for Section 18.4

Exercise 18.4.1: For each of the schedules of transactions T_1, T_2, and T_3 below:

a) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;

b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(C)$; $w_2(D)$; $w_3(E)$;

c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

d) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;

e) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

do each of the following:

i. Insert shared and exclusive locks, and insert unlock actions. Place a shared lock immediately in front of each read action that is not followed by a write action of the same element by the same transaction. Place an exclusive lock in front of every other read or write action. Place the necessary unlocks at the end of every transaction.

ii. Tell what happens when each schedule is run by a scheduler that supports shared and exclusive locks.

iii. Insert shared and exclusive locks in a way that allows upgrading. Place a shared lock in front of every read, an exclusive lock in front of every write, and place the necessary unlocks at the ends of the transactions.

iv. Tell what happens when each schedule from (*iii*) is run by a scheduler that supports shared locks, exclusive locks, and upgrading.

v. Insert shared, exclusive, and update locks, along with unlock actions. Place a shared lock in front of every read action that is not going to be upgraded, place an update lock in front of every read action that will be upgraded, and place an exclusive lock in front of every write action. Place unlocks at the ends of transactions, as usual.

vi. Tell what happens when each schedule from (*v*) is run by a scheduler that supports shared, exclusive, and update locks.

Exercise 18.4.2: For each of the following schedules, insert appropriate locks (read, write, or increment) before each action, and unlocks at the ends of transactions. Then tell what happens when the schedule is run by a scheduler that supports these three types of locks.

a) $r_1(A)$; $r_2(B)$; $inc_1(B)$; $inc_2(A)$; $w_1(C)$; $w_2(D)$;

b) $inc_1(A)$; $inc_2(B)$; $inc_1(B)$; $inc_2(C)$; $w_1(C)$; $w_2(D)$;

c) $r_1(A)$; $r_2(B)$; $inc_1(B)$; $inc_2(C)$; $w_1(C)$; $w_2(D)$;

Exercise 18.4.3: In Exercise 18.1.1, we discussed a hypothetical transaction involving an airline reservation. If the transaction manager had available to it shared, exclusive, update, and increment locks, what lock would you recommend for each of the steps of the transaction?

Exercise 18.4.4: The action of multiplication by a constant factor can be modeled by an action of its own. Suppose `MC(X,c)` stands for an atomic execution of the steps `READ(X,t); t := c*t; WRITE(X,t);`. We can also introduce a lock mode that allows only multiplication by a constant factor.

a) Show the compatibility matrix for read, write, and multiplication-by-a-constant locks.

! b) Show the compatibility matrix for read, write, incrementation, and multiplication-by-a-constant locks.

! **Exercise 18.4.5:** Consider the two transactions:

$$T_1\text{: } r_1(A);\ r_1(B);\ inc_1(A);\ inc_1(B);$$
$$T_2\text{: } r_2(A);\ r_2(B);\ inc_2(A);\ inc_2(B);$$

Answer the following:

a) How many interleavings of these transactions are serializable?

b) If the order of incrementation in T_2 were reversed [i.e., $inc_2(B)$ followed by $inc_2(A)$], how many serializable interleavings would there be?

! **Exercise 18.4.6:** Suppose for sake of argument that database elements are two-dimensional vectors. There are four operations we can perform on vectors, and each will have its own type of lock.

i. Change the value along the x-axis (an X-lock).

ii. Change the value along the y-axis (a Y-lock).

iii. Change the angle of the vector (an A-lock).

iv. Change the magnitude of the vector (an M-lock).

Answer the follow

a) Which pairs so its angle the same as angle to 120

b) Based on yo types of loc

!! c) Suppose we values for a 10 to the x would the c

! **Exercise 18.4.7**

Your problem is ??? and make th replacements for (c) Update (d)

18.5 An

Having seen a n scheduler that us a simple schedul

1. The transa upon to do stream of r

2. Transactio when the t abort.

18.5.1 A S

Figure 18.24 sh write, commit, table, which, alt or completely in table is not part Rather, the loc

Answer the following questions.

a) Which pairs of operations commute? For example, if we rotate the vector so its angle is 120^o and then change the x-coordinate to be 10, is that the same as first changing the x-coordinate to 10 and then changing the angle to 120^o?

b) Based on your answer to (a), what is the compatibility matrix for the four types of locks?

!! c) Suppose we changed the four operations so that instead of giving new values for a measure, the operations incremented the measure (e.g., "add 10 to the x-coordinate," or "rotate the vector 30^o clockwise"). What would the compatibility matrix then be?

! **Exercise 18.4.7:** Here is a schedule with one action missing:

$$r_1(A);\ r_2(B);\ ???;\ w_1(C);\ w_2(A);$$

Your problem is to figure out what actions of certain types could replace the ??? and make the schedule not be serializable. Tell all possible nonserializable replacements for each of the following types of action: (a) Read (b) Increment (c) Update (d) Write.

18.5 An Architecture for a Locking Scheduler

Having seen a number of different locking schemes, we next consider how a scheduler that uses one of these schemes operates. We shall consider here only a simple scheduler architecture based on several principles:

1. The transactions themselves do not request locks, or cannot be relied upon to do so. It is the job of the scheduler to insert lock actions into the stream of reads, writes, and other actions that access data.

2. Transactions do not release locks. Rather, the scheduler releases the locks when the transaction manager tells it that the transaction will commit or abort.

18.5.1 A Scheduler That Inserts Lock Actions

Figure 18.24 shows a two-part scheduler that accepts requests such as read, write, commit, and abort, from transactions. The scheduler maintains a lock table, which, although it is shown as secondary-storage data, may be partially or completely in main memory. Normally, the main memory used by the lock table is not part of the buffer pool that is used for query execution and logging. Rather, the lock table is just another component of the DBMS, and will be

3. *Priority to upgrading*: If there is a transaction with a U lock waiting to upgrade it to an X lock, grant that first. Otherwise, follow one of the other strategies mentioned.

18.5.3 Exercises for Section 18.5

Exercise 18.5.1: For each of the schedules of Exercise 18.2.5, tell the steps that the locking scheduler described in this section would execute.

Exercise 18.5.2: What are suitable group modes for a lock table if the lock modes used are:

a) Shared and exclusive locks.

! b) Shared, exclusive, and increment locks.

!! c) The lock modes of Exercise 18.4.6.

18.6 Hierarchies of Database Elements

Let us now return to the exploration of different locking schemes that we began in Section 18.4. In particular, we shall focus on two problems that come up when there is a tree structure to our data.

1. The first kind of tree structure we encounter is a hierarchy of lockable elements. We shall discuss in this section how to allow locks on both large elements, e.g., relations, and smaller elements contained within these, such as blocks holding several tuples of the relation, or individual tuples.

2. The second kind of hierarchy that is important in concurrency-control systems is data that is itself organized in a tree. A major example is B-tree indexes. We may view nodes of the B-tree as database elements, but if we do, then as we shall see in Section 18.7, the locking schemes studied so far perform poorly, and we need to use a new approach.

18.6.1 Locks With Multiple Granularity

Recall that the term "database element" was purposely left undefined, because different systems use different sizes of database elements to lock, such as tuples, pages or blocks, and relations. Some applications benefit from small database elements, such as tuples, while others are best off with large elements.

Example 18.20: Consider a database for a bank. If we treated relations as database elements, and therefore had only one lock for an entire relation such as the one giving account balances, then the system would allow very little concurrency. Since most transactions will change an account balance either positively or negatively, most transactions would need an exclusive lock on the

18.6.4 Exercises for Section 18.6

Exercise 18.6.1: Change the sequence of actions in Example 18.22 so that the $w_4(D_3)$ action becomes a write by T_4 of the entire relation `Movie`. Then, show the action of a warning-protocol-based scheduler on this sequence of requests.

Exercise 18.6.2: Consider, for variety, an object-oriented database. The objects of class C are stored on two blocks, B_1 and B_2. Block B_1 contains objects O_1, O_2, and O_3, while block B_2 contains objects O_4 and O_5. The entire set of objects of class C, the blocks, and the individual objects form a hierarchy of lockable database elements. Tell the sequence of lock requests and the response of a warning-protocol-based scheduler to the following sequences of requests. You may assume all requests occur just before they are needed, and all unlocks occur at the end of the transaction.

a) $r_1(O_5)$; $w_2(O_5)$; $r_2(O_3)$; $w_1(O_4)$;

b) $r_1(O_1)$; $r_1(O_3)$; $r_2(O_1)$; $w_2(O_4)$; $w_2(O_5)$;

c) $r_1(O_1)$; $w_2(O_2)$; $r_2(O_3)$; $w_1(O_4)$;

d) $r_1(O_1)$; $r_2(O_2)$; $r_3(O_1)$; $w_1(O_3)$; $w_2(O_4)$; $w_3(O_5)$; $w_1(O_2)$;

!! **Exercise 18.6.3:** Show how to add increment locks to a warning-protocol-based scheduler.

18.7 The Tree Protocol

Like Section 18.6, this section deals with data in the form of a tree. However, here, the nodes of the tree do not form a hierarchy based on containment. Rather, database elements are disjoint pieces of data, but the only way to get to a node is through its parent; B-trees are an important example of this sort of data. Knowing that we must traverse a particular path to an element gives us some important freedom to manage locks differently from the two-phase locking approaches we have seen so far.

18.7.1 Motivation for Tree-Based Locking

Let us consider a B-tree index in a system that treats individual nodes (i.e., blocks) as lockable database elements. The node is the right level of lock granularity, because treating smaller pieces as elements offers no benefit, and treating the entire B-tree as one database element prevents the sort of concurrent use of the index that can be achieved via the mechanisms that form the subject of this section.

If we use a standard set of lock modes, like shared, exclusive, and update locks, and we use two-phase locking, then concurrent use of the B-tree is almost impossible. The reason is that every transaction using the index must begin by 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.

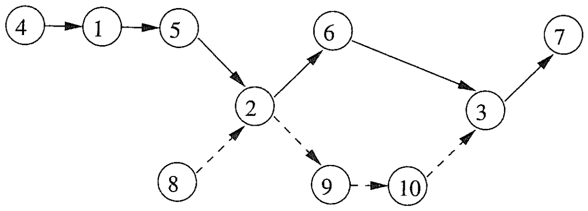

Figure 18.34: Combining serial orders for the subtrees into a serial order for all transactions

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 4 (b) Insert 30 (c) Delete 37 (d) Delete 7.

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

$$
\begin{array}{l}
T_1\text{: } r_1(A);\ r_1(B);\ r_1(E);\\
T_2\text{: } r_2(A);\ r_2(C);\ r_2(B);\\
T_3\text{: } r_3(B);\ r_3(E);\ r_3(F);
\end{array}
$$

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

!! **Exercise 18.7.3:** 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?

! **Exercise 18.7.4:** 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?

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.* Assign a "timestamp" to each transaction. Record the timestamps of the transactions that last read and write each database element, and compare these values with the transactions timestamps, 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.* 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.[9] Generally, optimistic schedulers are

[9]That is not to say that a system using a locking scheduler will never abort a transaction; for instance, Section 19.2 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.

18.8.7 Exercises for Section 18.8

Exercise 18.8.1: Below are several sequences of events, including *start* events, where st_i means that transaction T_i starts. These sequences represent real time, and the timestamp scheduler will allocate timestamps to transactions in the order of their starts. Tell what happens as each executes.

a) st_1; $r_1(A)$; st_2; $w_2(B)$; $r_2(A)$; $w_1(B)$;

b) st_1; st_2; $r_1(A)$; $r_2(B)$; $w_2(A)$; $w_1(B)$;

c) st_1; st_3; st_2; $r_1(A)$; $r_3(B)$; $w_1(C)$; $r_2(B)$; $r_2(C)$; $w_3(B)$; $w_2(A)$;

d) st_1; st_2; st_3; $r_1(A)$; $r_3(B)$; $w_1(C)$; $r_2(B)$; $r_2(C)$; $w_3(B)$; $w_2(A)$;

!! **Exercise 18.8.2:** We observed in our study of lock-based schedulers that there are several reasons why transactions that obtain locks could deadlock. Can a timestamp scheduler using the commit bit $C(X)$ have a deadlock?

Exercise 18.8.3: Tell what happens during the following sequences of events if a multiversion, timestamp scheduler is used. What happens instead, if the scheduler does not maintain multiple versions?

a) st_1; st_2; st_3; st_4; $w_1(A)$; $w_3(A)$; $r_4(A)$; $r_2(A)$;

b) st_1; st_2; st_3; st_4; $w_1(A)$; $w_4(A)$; $r_3(A)$; $w_2(A)$;

c) st_1; st_2; st_3; st_4; $w_1(A)$; $w_2(A)$; $w_3(A)$; $r_2(A)$; $r_4(A)$;

18.9 Concurrency Control by Validation

Validation is another type of optimistic concurrency control, where we allow transactions to access data without locks, and at the appropriate time we check that the transaction has behaved in a serializable manner. Validation differs from timestamping principally in that the scheduler maintains a record of what active transactions are doing, rather than keeping read and write times for all database elements. Just before a transaction starts to write values of database elements, it goes through a "validation phase," where the sets of elements it has read and will write are compared with the write sets of other active transactions. Should there be a risk of physically unrealizable behavior, the transaction is rolled back.

18.9.1 Architecture of a Validation-Based Scheduler

When validation is used as the concurrency-control mechanism, the scheduler must be told for each transaction T the sets of database elements T reads and writes, the *read set*, RS(T), and the *write set*, WS(T), respectively. Transactions are executed in three phases:

1. *Read.* In
elements
space all

2. *Validate.*
comparin
shall desc
then the
phase.

3. *Write.* In
for the el

Intuitively, we
ecuting at the
has an assume
decision to vali
with this serial

To support
maintains thre

1. *START*,
validatio
START(T

2. *VAL*, the
the writi
maintain
Note tha
the hypo

3. *FIN*, the
transacti
time at w
we do no
active tr
may thus
beyond b

18.9.2 Th

The informati
potential viola
in which the tr
what can be w

1. Suppose

- When a rollback is necessary, timestamps catch some problems earlier than validation, which always lets a transaction do all its internal work before considering whether the transaction must rollback.

18.9.4 Exercises for Section 18.9

Exercise 18.9.1: In the following sequences of events, we use $R_i(X)$ to mean "transaction T_i starts, and its read set is the list of database elements X." Also, V_i means "T_i attempts to validate," and $W_i(X)$ means that "T_i finishes, and its write set was X." Tell what happens when each sequence is processed by a validation-based scheduler.

a) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(B)$; $W_3(C)$;

b) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(C)$; $W_2(B)$; $W_3(A)$;

c) $R_1(A,B)$; $R_2(B,C)$; $R_3(C)$; V_1; V_2; V_3; $W_1(A)$; $W_2(C)$; $W_3(B)$;

d) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(B)$;

e) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(A)$; V_2; $W_2(A)$; $W_3(D)$;

f) $R_1(A,B)$; $R_2(B,C)$; V_1; $R_3(C,D)$; V_3; $W_1(C)$; V_2; $W_2(A)$; $W_3(D)$;

18.10 Summary of Chapter 18

✦ *Consistent Database States*: Database states that obey whatever implied or declared constraints the designers intended are called consistent. It is essential that operations on the database preserve consistency, that is, they turn one consistent database state into another.

✦ *Consistency of Concurrent Transactions*: It is normal for several transactions to have access to a database at the same time. Transactions, run in isolation, are assumed to preserve consistency of the database. It is the job of the scheduler to assure that concurrently operating transactions also preserve the consistency of the database.

✦ *Schedules*: Transactions are broken into actions, mainly reading and writing from the database. A sequence of these actions from one or more transactions is called a schedule.

✦ *Serial Schedules*: If transactions execute one at a time, the schedule is said to be serial.

✦ *Serializable Schedules*: A schedule that is equivalent in its effect on the database to some serial schedule is said to be serializable. Interleaving of actions from several transactions is possible in a serializable schedule that is not itself serial, but we must be very careful what sequences of actions

we allow, or
state.

✦ *Conflict-Seri*
ability is tha
of adjacent a
serializable.
transaction, c
at least one o

✦ *Precedence G*
a precedence
and there is
with a later
the precedenc

✦ *Locking*: The
to lock datab
after finishin
transactions

✦ *Two-Phase L*
ever, two-pha
they only ac
locks, will gu

✦ *Lock Modes*:
usually use s
when a lock
locks for rea
writing.

✦ *Compatibilit*
when it is le
may be othe

✦ *Update Lock*
then write a
the lock to e
shared locks
locks from b

✦ *Increment L*
add or subtr
Increment l
although th