Chapter 18

Concurrency Control

Interactions among concurrently executing transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure. Thus, the timing of individual steps of different transactions needs to be regulated in some manner. This regulation is the job of the *scheduler* component of the DBMS, and the general process of assuring that transactions preserve consistency when executing simultaneously is called *concurrency control*. The role of the scheduler is suggested by Fig. 18.1.

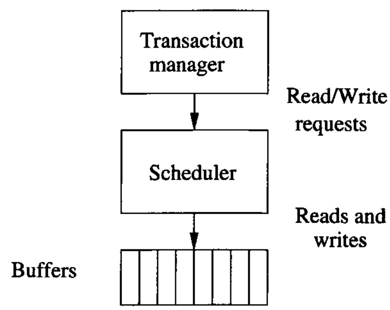

Figure 18.1: The scheduler takes read/write requests from transactions and either executes them in buffers or delays them

As transactions request reads and writes of database elements, these requests are passed to the scheduler. In most situations, the scheduler will execute the reads and writes directly, first calling on the buffer manager if the desired database element is not in a buffer. However, in some situations, it is not safe for the request to be executed immediately. The scheduler must delay the request; in some concurrency-control techniques, the scheduler may even abort the transaction that issued the request.

We begin by studying how to assure that concurrently executing transactions preserve correctness of the database state. The abstract requirement is called *serializability*, and there is an important, stronger condition called *conflict-serializability* that most schedulers actually enforce. We consider the most important techniques for implementing schedulers: locking, timestamping, and validation. Our study of lock-based schedulers includes the important concept of "two-phase locking," which is a requirement widely used to assure serializability of schedules.

18.1 Serial and Serializable Schedules

Recall the "correctness principle" from Section 17.1.3: every transaction, if executed in isolation (without any other transactions running concurrently), will transform any consistent state to another consistent state. In practice, transactions often run concurrently with other transactions, so the correctness principle doesn't apply directly. This section introduces the notion of "schedules," the sequence of actions performed by transactions and "serializable schedules," which produce the same result as if the transactions executed one-at-a-time.

18.1.1 Schedules

A *schedule* is a sequence of the important actions taken by one or more transactions. When studying concurrency control, the important read and write actions take place in the main-memory buffers, not the disk. That is, a database element A that is brought to a buffer by some transaction T may be read or written in that buffer not only by T but by other transactions that access A.

T_1	T_2
`READ(A,t)`	`READ(A,s)`
`t := t+100`	`s := s*2`
`WRITE(A,t)`	`WRITE(A,s)`
`READ(B,t)`	`READ(B,s)`
`t := t+100`	`s := s*2`
`WRITE(B,t)`	`WRITE(B,s)`

Figure 18.2: Two transactions

Example 18.1: Let us consider two transactions and the effect on the database when their actions are executed in certain orders. The important actions of the transactions T_1 and T_2 are shown in Fig. 18.2. The variables t and s are local variables of T_1 and T_2, respectively; they are *not* database elements.

We shall assume that the only consistency constraint on the database state is that $A = B$. Since T_1 adds 100 to both A and B, and T_2 multiplies both

A and B by 2, we know that each transaction, run in isolation, will preserve consistency. □

18.1.2 Serial Schedules

A schedule is *serial* if its actions consist of all the actions of one transaction, then all the actions of another transaction, and so on. No mixing of the actions is allowed.

T_1	T_2	A	B
		25	25
`READ(A,t)`			
`t := t+100`			
`WRITE(A,t)`		125	
`READ(B,t)`			
`t := t+100`			
`WRITE(B,t)`			125
	`READ(A,s)`		
	`s := s*2`		
	`WRITE(A,s)`	250	
	`READ(B,s)`		
	`s := s*2`		
	`WRITE(B,s)`		250

Figure 18.3: Serial schedule in which T_1 precedes T_2

Example 18.2: For the transactions of Fig. 18.2, there are two serial schedules, one in which T_1 precedes T_2 and the other in which T_2 precedes T_1. Figure 18.3 shows the sequence of events when T_1 precedes T_2, and the initial state is $A = B = 25$. We shall take the convention that when displayed vertically, time proceeds down the page. Also, the values of A and B shown refer to their values in main-memory buffers, not necessarily to their values on disk.

Figure 18.4 shows another serial schedule in which T_2 precedes T_1; the initial state is again assumed to be $A = B = 25$. Notice that the final values of A and B are different for the two schedules; they both have value 250 when T_1 goes first and 150 when T_2 goes first. In general, we would not expect the final state of a database to be independent of the order of transactions. □

We can represent a serial schedule as in Fig. 18.3 or Fig. 18.4, listing each of the actions in the order they occur. However, since the order of actions in a serial schedule depends only on the order of the transactions themselves, we shall sometimes represent a serial schedule by the list of transactions. Thus, the schedule of Fig. 18.3 is represented (T_1, T_2), and that of Fig. 18.4 is (T_2, T_1).

T_1	T_2	A	B
		25	25
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(A,t)			
t := t+100			
WRITE(A,t)		150	
READ(B,t)			
t := t+100			
WRITE(B,t)			150

Figure 18.4: Serial schedule in which T_2 precedes T_1

18.1.3 Serializable Schedules

The correctness principle for transactions tells us that every serial schedule will preserve consistency of the database state. But are there any other schedules that also are guaranteed to preserve consistency? There are, as the following example shows. In general, we say a schedule S is *serializable* if there is a serial schedule S' such that for every initial database state, the effects of S and S' are the same.

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Figure 18.5: A serializable, but not serial, schedule

Example 18.3: Figure 18.5 shows a schedule of the transactions from Example 18.1 that is serializable but not serial. In this schedule, T_2 acts on A after T_1 does, but before T_1 acts on B. However, we see that the effect of the two transactions scheduled in this manner is the same as for the serial schedule (T_1, T_2) from Fig. 18.3. To convince ourselves of the truth of this statement, we must consider not only the effect from the database state $A = B = 25$, which we show in Fig. 18.5, but from any consistent database state. Since all consistent database states have $A = B = c$ for some constant c, it is not hard to deduce that in the schedule of Fig. 18.5, both A and B will be left with the value $2(c + 100)$, and thus consistency is preserved from any consistent state.

T_1	T_2	A	B
		25	25
`READ(A,t)`			
`t := t+100`			
`WRITE(A,t)`		125	
	`READ(A,s)`		
	`s := s*2`		
	`WRITE(A,s)`	250	
	`READ(B,s)`		
	`s := s*2`		
	`WRITE(B,s)`		50
`READ(B,t)`			
`t := t+100`			
`WRITE(B,t)`			150

Figure 18.6: A nonserializable schedule

On the other hand, consider the schedule of Fig. 18.6, which is not serializable. The reason we can be sure it is not serializable is that it takes the consistent state $A = B = 25$ and leaves the database in an inconsistent state, where $A = 250$ and $B = 150$. Notice that in this order of actions, where T_1 operates on A first, but T_2 operates on B first, we have in effect applied different computations to A and B, that is $A := 2(A + 100)$ versus $B := 2B + 100$. The schedule of Fig. 18.6 is the sort of behavior that concurrency control mechanisms must avoid. □

18.1.4 The Effect of Transaction Semantics

In our study of serializability so far, we have considered in detail the operations performed by the transactions, to determine whether or not a schedule is serializable. The details of the transactions do matter, as we can see from the following example.

T_1	T_2	A	B
		25	25
`READ(A,t)`			
`t := t+100`			
`WRITE(A,t)`		125	
	`READ(A,s)`		
	`s := s+200`		
	`WRITE(A,s)`	325	
	`READ(B,s)`		
	`s := s+200`		
	`WRITE(B,s)`		225
`READ(B,t)`			
`t := t+100`			
`WRITE(B,t)`			325

Figure 18.7: A schedule that is serializable only because of the detailed behavior of the transactions

Example 18.4: Consider the schedule of Fig. 18.7, which differs from Fig. 18.6 only in the computation that T_2 performs. That is, instead of multiplying A and B by 2, T_2 adds 200 to each. One can easily check that regardless of the consistent initial state, the final state is the one that results from the serial schedule (T_1, T_2). Coincidentally, it also results from the other serial schedule, (T_2, T_1). □

Unfortunately, it is not realistic for the scheduler to concern itself with the details of computation undertaken by transactions. Since transactions often involve code written in a general-purpose programming language as well as SQL or other high-level-language statements, it is impossible to say for certain what a transaction is doing. However, the scheduler does get to see the read and write requests from the transactions, so it can know what database elements each transaction reads, and what elements it *might* change. To simplify the job of the scheduler, it is conventional to assume that:

- Any database element A that a transaction T writes is given a value that depends on the database state in such a way that no arithmetic coincidences occur.

An example of a "coincidence" is that in Example 18.4, where $A + 100 + 200 = B + 200 + 100$ whenever $A = B$, even though the two operations are carried out in different orders on the two variables. Put another way, if there is something that T could do to a database element to make the database state inconsistent, then T will do that.

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 consist of 4 and 6 actions, respectively, 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.

b) Two writes of the same database element by different transactions conflict. That is, $w_i(X)$; $w_j(X)$ is a conflict. The reason is that as written, the value of X remains afterward as whatever T_j computed it to be. If we swap the order, as $w_j(X)$; $w_i(X)$, then we leave X with the value computed by T_i. Our assumption of "no coincidences" tells us that the values written by T_i and T_j will be different, at least for some initial states of the database.

c) A read and a write of the same database element by different transactions also conflict. That is, $r_i(X)$; $w_j(X)$ is a conflict, and so is $w_i(X)$; $r_j(X)$. If we move $w_j(X)$ ahead of $r_i(X)$, then the value of X read by T_i will be that written by T_j, which we assume is not necessarily the same as the previous value of X. Thus, swapping the order of $r_i(X)$ and $w_j(X)$ affects the value T_i reads for X and could therefore affect what T_i does.

The conclusion we draw is that any two actions of different transactions may be swapped unless:

1. They involve the same database element, and

2. At least one is a write.

Extending this idea, we may take any schedule and make as many nonconflicting swaps as we wish, with the goal of turning the schedule into a serial schedule. If we can do so, then the original schedule is serializable, because its effect on the database state remains the same as we perform each of the nonconflicting swaps.

We say that two schedules are *conflict-equivalent* if they can be turned one into the other by a sequence of nonconflicting swaps of adjacent actions. We shall call a schedule *conflict-serializable* if it is conflict-equivalent to a serial schedule. Note that conflict-serializability is a sufficient condition for serializability; i.e., a conflict-serializable schedule is a serializable schedule. Conflict-serializability is not required for a schedule to be serializable, but it is the condition that the schedulers in commercial systems generally use when they need to guarantee serializability.

Example 18.6: Consider the schedule

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

from Example 18.5. We claim this schedule is conflict-serializable. Figure 18.8 shows the sequence of swaps in which this schedule is converted to the serial schedule (T_1, T_2), where all of T_1's actions precede all those of T_2. We have underlined the pair of adjacent actions about to be swapped at each step. □

$$r_1(A);\ w_1(A);\ r_2(A);\ \underline{w_2(A);\ r_1(B)};\ w_1(B);\ r_2(B);\ w_2(B);$$
$$r_1(A);\ w_1(A);\ \underline{r_2(A);\ \overline{r_1(B)}};\ w_2(A);\ w_1(B);\ r_2(B);\ w_2(B);$$
$$r_1(A);\ w_1(A);\ r_1(B);\ r_2(A);\ \underline{w_2(A);\ w_1(B)};\ r_2(B);\ w_2(B);$$
$$r_1(A);\ w_1(A);\ r_1(B);\ \underline{r_2(A);\ \overline{w_1(B)}};\ \overline{w_2(A)};\ r_2(B);\ w_2(B);$$
$$r_1(A);\ w_1(A);\ r_1(B);\ \overline{w_1(B)};\ \overline{r_2(A)};\ w_2(A);\ r_2(B);\ w_2(B);$$

Figure 18.8: Converting a conflict-serializable schedule to a serial schedule by swaps of adjacent actions

18.2.2 Precedence Graphs and a Test for Conflict-Serializability

It is relatively simple to examine a schedule S and decide whether or not it is conflict-serializable. When a pair of conflicting actions appears anywhere in S, the transactions performing those actions must appear in the same order in any conflict-equivalent serial schedule as the actions appear in S. Thus, conflicting pairs of actions put constraints on the order of transactions in the hypothetical, conflict-equivalent serial schedule. If these constraints are not contradictory, we can find a conflict-equivalent serial schedule. If they are contradictory, we know that no such serial schedule exists.

Given a schedule S, involving transactions T_1 and T_2, perhaps among other transactions, we say that *T_1 takes precedence over T_2*, written $T_1 <_S T_2$, if there are actions A_1 of T_1 and A_2 of T_2, such that:

1. A_1 is ahead of A_2 in S,

2. Both A_1 and A_2 involve the same database element, and

3. At least one of A_1 and A_2 is a write action.

Notice that these are exactly the conditions under which we cannot swap the order of A_1 and A_2. Thus, A_1 will appear before A_2 in any schedule that is conflict-equivalent to S. As a result, a conflict-equivalent serial schedule must have T_1 before T_2.

We can summarize these precedences in a *precedence graph*. The nodes of the precedence graph are the transactions of a schedule S. When the transactions are T_i for various i, we shall label the node for T_i by only the integer i. There is an arc from node i to node j if $T_i <_S T_j$.

Example 18.7: The following schedule S involves three transactions, T_1, T_2, and T_3.

$$S\colon\ r_2(A);\ r_1(B);\ w_2(A);\ r_3(A);\ w_1(B);\ w_3(A);\ r_2(B);\ w_2(B);$$

If we look at the actions involving A, we find several reasons why $T_2 <_S T_3$. For example, $r_2(A)$ comes ahead of $w_3(A)$ in S, and $w_2(A)$ comes ahead of both

Why Conflict-Serializability is not Necessary for Serializability

Consider three transactions T_1, T_2, and T_3 that each write a value for X. T_1 and T_2 also write values for Y before they write values for X. One possible schedule, which happens to be serial, is

$$S_1\text{: } w_1(Y);\ w_1(X);\ w_2(Y);\ w_2(X);\ w_3(X);$$

S_1 leaves X with the value written by T_3 and Y with the value written by T_2. However, so does the schedule

$$S_2\text{: } w_1(Y);\ w_2(Y);\ w_2(X);\ w_1(X);\ w_3(X);$$

Intuitively, the values of X written by T_1 and T_2 have no effect, since T_3 overwrites their values. Thus, X has the same value after either S_1 or S_2, and likewise Y has the same value after either S_1 or S_2. Since S_1 is serial, and S_2 has the same effect as S_1 on any database state, we know that S_2 is serializable. However, since we cannot swap $w_1(Y)$ with $w_2(Y)$, and we cannot swap $w_1(X)$ with $w_2(X)$, therefore we cannot convert S_2 to any serial schedule by swaps. That is, S_2 is serializable, but not conflict-serializable.

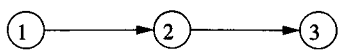

Figure 18.9: The precedence graph for the schedule S of Example 18.7

$r_3(A)$ and $w_3(A)$. Any one of these three observations is sufficient to justify the arc in the precedence graph of Fig. 18.9 from 2 to 3.

Similarly, if we look at the actions involving B, we find that there are several reasons why $T_1 <_S T_2$. For instance, the action $r_1(B)$ comes before $w_2(B)$. Thus, the precedence graph for S also has an arc from 1 to 2. However, these are the only arcs we can justify from the order of actions in schedule S. □

To tell whether a schedule S is conflict-serializable, construct the precedence graph for S and ask if there are any cycles. If so, then S is not conflict-serializable. But if the graph is acyclic, then S is conflict-serializable, and moreover, any topological order of the nodes[1] is a conflict-equivalent serial order.

[1] A *topological order* of an acyclic graph is any order of the nodes such that for every arc $a \to b$, node a precedes node b in the topological order. We can find a topological order for any acyclic graph by repeatedly removing nodes that have no predecessors among the remaining nodes.

Example 18.8: Figure 18.9 is acyclic, so the schedule S of Example 18.7 is conflict-serializable. There is only one order of the nodes or transactions consistent with the arcs of that graph: (T_1, T_2, T_3). Notice that it is indeed possible to convert S into the schedule in which all actions of each of the three transactions occur in this order; this serial schedule is:

$$S': r_1(B);\ w_1(B);\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);\ r_3(A);\ w_3(A);$$

To see that we can get from S to S' by swaps of adjacent elements, first notice we can move $r_1(B)$ ahead of $r_2(A)$ without conflict. Then, by three swaps we can move $w_1(B)$ just after $r_1(B)$, because each of the intervening actions involves A and not B. We can then move $r_2(B)$ and $w_2(B)$ to a position just after $w_2(A)$, moving through only actions involving A; the result is S'. □

Example 18.9: Consider the schedule

$$S_1: r_2(A);\ r_1(B);\ w_2(A);\ r_2(B);\ r_3(A);\ w_1(B);\ w_3(A);\ w_2(B);$$

which differs from S only in that action $r_2(B)$ has been moved forward three positions. Examination of the actions involving A still give us only the precedence $T_2 <_{S_1} T_3$. However, when we examine B we get not only $T_1 <_{S_1} T_2$ [because $r_1(B)$ and $w_1(B)$ appear before $w_2(B)$], but also $T_2 <_{S_1} T_1$ [because $r_2(B)$ appears before $w_1(B)$]. Thus, we have the precedence graph of Fig. 18.10 for schedule S_1.

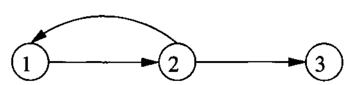

Figure 18.10: A cyclic precedence graph; its schedule is not conflict-serializable

This graph evidently has a cycle. We conclude that S_1 is not conflict-serializable. Intuitively, any conflict-equivalent serial schedule would have to have T_1 both ahead of and behind T_2, so therefore no such schedule exists. □

18.2.3 Why the Precedence-Graph Test Works

If there is a cycle involving n transactions $T_1 \to T_2 \to \ldots \to T_n \to T_1$, then in the hypothetical serial order, the actions of T_1 must precede those of T_2, which precede those of T_3, and so on, up to T_n. But the actions of T_n, which therefore come after those of T_1, are also required to precede those of T_1 because of the arc $T_n \to T_1$. Thus, if there is a cycle in the precedence graph, then the schedule is not conflict-serializable.

The converse is a bit harder. We must show that if the precedence graph has no cycles, then we can reorder the schedule's actions using legal swaps of adjacent actions, until the schedule becomes a serial schedule. If we can do so, then we have our proof that every schedule with an acyclic precedence graph is conflict-serializable. The proof is an induction on the number of transactions involved in the schedule.

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

$$T_1, T_2, \ldots, T_n$$

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.

T_1: `READ(A,t); t:=t+2; WRITE(A,t); READ(B,t); t:=t*3; WRITE(B,t);`
T_2: `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) 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.

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

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

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

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:

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

Answer the following:

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

b) How many schedules of the eight actions are 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(d) different?

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

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

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 conflict-serializable schedules.

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

Exercise 18.2.4: For each of the following schedules:

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

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

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

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

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

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 18.2.5:** 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.

! **Exercise 18.2.6:** Explain how, for any $n > 1$, one can find a schedule whose precedence graph has a cycle of length n, but no smaller cycle.

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.

18.3.1 Locks

In Fig. 18.11 we see a scheduler that uses a lock table to help perform its job. Recall from the chapter introduction that the responsibility of the scheduler is to take requests from transactions and either allow them to operate on the database or block the transaction until such time as it is safe to allow it to continue. A lock table will be used to guide this decision in a manner that we shall discuss at length.

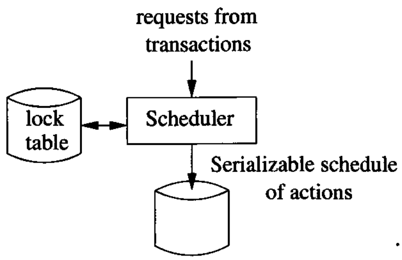

Figure 18.11: A scheduler that uses a lock table to guide decisions

Ideally, a scheduler would forward a request if and only if its execution cannot possibly lead to an inconsistent database state after all active transactions commit or abort. A locking scheduler, like most types of scheduler, instead enforces conflict-serializability, which as we learned is a more stringent condition than correctness, or even than serializability.

When a scheduler uses locks, transactions must request and release locks, in addition to reading and writing database elements. The use of locks must be proper in two senses, one applying to the structure of transactions, and the other to the structure of schedules.

- *Consistency of Transactions*: Actions and locks must relate in the expected ways:
  1. A transaction can only read or write an element if it previously was granted a lock on that element and hasn't yet released the lock.
  2. If a transaction locks an element, it must later unlock that element.
- *Legality of Schedules*: Locks must have their intended meaning: no two transactions may have locked the same element without one having first released the lock.

We shall extend our notation for actions to include locking and unlocking actions:

$l_i(X)$: Transaction T_i requests a lock on database element X.

$u_i(X)$: Transaction T_i releases ("unlocks") its lock on database element X.

Thus, the consistency condition for transactions can be stated as: "Whenever a transaction T_i has an action $r_i(X)$ or $w_i(X)$, then there is a previous action $l_i(X)$ with no intervening action $u_i(X)$, and there is a subsequent $u_i(X)$." The legality of schedules is stated: "If there are actions $l_i(X)$ followed by $l_j(X)$ in a schedule, then somewhere between these actions there must be an action $u_i(X)$."

Example 18.10: Let us consider the two transactions T_1 and T_2 that we introduced in Example 18.1. Recall that T_1 adds 100 to database elements A and B, while T_2 doubles them. Here are specifications for these transactions, in which we have included lock actions as well as arithmetic actions to help us remember what the transactions are doing.[2]

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

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

Each of these transactions is consistent. They each release the locks on A and B that they take. Moreover, they each operate on A and B only at steps where they have previously requested a lock on that element and have not yet released the lock.

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$; $u_2(A)$;	250	
	$l_2(B)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		50
$l_1(B)$; $r_1(B)$;			
`B := B+100;`			
$w_1(B)$; $u_1(B)$;			150

Figure 18.12: A legal schedule of consistent transactions; unfortunately it is not serializable

Figure 18.12 shows one legal schedule of these two transactions. To save space we have put several actions on one line. The schedule is legal because

[2] Remember that the actual computations of the transaction usually are not represented in our current notation, since they are not considered by the scheduler when deciding whether to grant or deny transaction requests.

the two transactions never hold a lock on A at the same time, and likewise for B. Specifically, T_2 does not execute $l_2(A)$ until after T_1 executes $u_1(A)$, and T_1 does not execute $l_1(B)$ until after T_2 executes $u_2(B)$. As we see from the trace of the values computed, the schedule, although legal, is not serializable. We shall see in Section 18.3.3 the additional condition, "two-phase locking," that we need to assure that legal schedules are conflict-serializable. □

18.3.2 The Locking Scheduler

It is the job of a scheduler based on locking to grant requests if and only if the request will result in a legal schedule. If a request is not granted, the requesting transaction is delayed; it waits until the scheduler grants its request at a later time. To aid its decisions, the scheduler has a *lock table* that tells, for every database element, the transaction (if any) that currently holds a lock on that element. We shall discuss the structure of a lock table in more detail in Section 18.5.2. However, when there is only one kind of lock, as we have assumed so far, the table may be thought of as a relation `Locks(element, transaction)`, consisting of pairs (X,T) such that transaction T currently has a lock on database element X. The scheduler has only to query and modify this relation.

Example 18.11: The schedule of Fig. 18.12 is legal, as we mentioned, so the locking scheduler would grant every request in the order of arrival shown. However, sometimes it is not possible to grant requests. Here are T_1 and T_2 from Example 18.10, with simple but important changes, in which T_1 and T_2 each lock B before releasing the lock on A.

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

In Fig. 18.13, when T_2 requests a lock on B, the scheduler must deny the lock, because T_1 still holds a lock on B. Thus, T_2 is delayed, and the next actions are from T_1. Eventually, T_1 executes $u_1(B)$, which unlocks B. Now, T_2 can get its lock on B, which is executed at the next step. Notice that because T_2 was forced to wait, it wound up multiplying B by 2 after T_1 added 100, resulting in a consistent database state. □

18.3.3 Two-Phase Locking

There is a surprising condition, called *two-phase locking* (or *2PL*) under which we can guarantee that a legal schedule of consistent transactions is conflict-serializable:

- In every transaction, all lock actions precede all unlock actions.

T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
`A := A+100;`			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$;		
	`A := A*2;`		
	$w_2(A)$;	250	
	$l_2(B)$ **Denied**		
$r_1(B)$; `B := B+100;`			
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$;		
	`B := B*2;`		
	$w_2(B)$; $u_2(B)$;		250

Figure 18.13: The locking scheduler delays requests that would result in an illegal schedule

The "two phases" referred to by 2PL are thus the first phase, where locks are obtained, and the second phase, where locks are relinquished. Two-phase locking is a condition, like consistency, on the order of actions in a transaction. A transaction that obeys the 2PL condition is said to be a *two-phase-locked transaction*, or 2PL transaction.

Example 18.12: In Example 18.10, the transactions do not obey the two-phase locking rule. For instance, T_1 unlocks A before it locks B. However, the versions of the transactions found in Example 18.11 *do* obey the 2PL condition. Notice that T_1 locks both A and B within the first five actions and unlocks them within the next five actions; T_2 behaves similarly. If we compare Figs. 18.12 and 18.13, we see how the 2PL transactions interact properly with the scheduler to assure consistency, while the non-2PL transactions allow non-conflict-serializable behavior. □

18.3.4 Why Two-Phase Locking Works

Intuitively, each two-phase-locked transaction may be thought to execute in its entirety at the instant it issues its first unlock request, as suggested by Fig. 18.14. Thus, there is always at least one conflict-equivalent serial schedule for a schedule S of 2PL transactions: the one in which the transactions appear in the same order as their first unlocks.

We shall show how to convert any legal schedule S of consistent, two-phase-locked transactions to a conflict-equivalent serial schedule. The conversion is best described as an induction on n, the number of transactions in S. In what follows, it is important to remember that the issue of conflict-equivalence refers

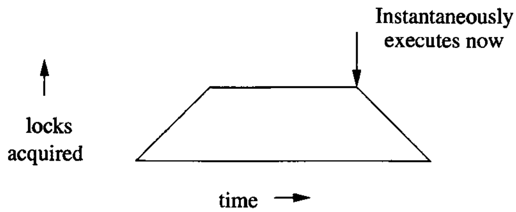

Figure 18.14: Every two-phase-locked transaction has a point at which it may be thought to execute instantaneously

to the read and write actions only. As we swap the order of reads and writes, we ignore the lock and unlock actions. Once we have the read and write actions ordered serially, we can place the lock and unlock actions around them as the various transactions require. Since each transaction releases all locks before its end, we know that the serial schedule is legal.

BASIS: If $n = 1$, there is nothing to do; S is already a serial schedule.

INDUCTION: Suppose S involves n transactions $T_1, T_2, \ldots, T_n$, and let T_i be the transaction with the first unlock action in the entire schedule S, say $u_i(X)$. We claim it is possible to move all the read and write actions of T_i forward to the beginning of the schedule without passing any conflicting reads or writes.

Consider some action of T_i, say $w_i(Y)$. Could it be preceded in S by some conflicting action, say $w_j(Y)$? If so, then in schedule S, actions $u_j(Y)$ and $l_i(Y)$ must intervene, in a sequence of actions

$$\cdots w_j(Y); \cdots ;\ u_j(Y); \cdots ;\ l_i(Y); \cdots ;\ w_i(Y); \cdots$$

Since T_i is the first to unlock, $u_i(X)$ precedes $u_j(Y)$ in S; that is, S might look like:

$$\cdots ;\ w_j(Y); \cdots ;\ u_i(X); \cdots ;\ u_j(Y); \cdots ;\ l_i(Y); \cdots ;\ w_i(Y); \cdots$$

or $u_i(X)$ could even appear before $w_j(Y)$. In any case, $u_i(X)$ appears before $l_i(Y)$, which means that T_i is *not* two-phase-locked, as we assumed. While we have only argued the nonexistence of conflicting pairs of writes, the same argument applies to any pair of potentially conflicting actions, one from T_i and the other from another T_j.

We conclude that it is indeed possible to move all the actions of T_i forward to the beginning of S, using swaps of nonconflicting read and write actions, followed by restoration of the lock and unlock actions of T_i. That is, S can be written in the form

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

The tail of $n - 1$ transactions is still a legal schedule of consistent, 2PL transactions, so the inductive hypothesis applies to it. We convert the tail to a

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

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

!! 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:** 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 18.3.3: For each of the schedules of Exercise 18.2.4, 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 18.3.4:** 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) $r_1(A)$; $w_1(B)$;

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

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:

1. *Consistency of transactions*: A transaction may not write without holding an exclusive lock, and you may not read without holding some lock. More precisely, in any transaction T_i,

   (a) A read action $r_i(X)$ must be preceded by $sl_i(X)$ or $xl_i(X)$, with no intervening $u_i(X)$.

   (b) A write action $w_i(X)$ must be preceded by $xl_i(X)$, with no intervening $u_i(X)$.

   All locks must be followed by an unlock of the same element.

2. *Two-phase locking of transactions*: Locking must precede unlocking. To be more precise, in any two-phase locked transaction T_i, no action $sl_i(X)$ or $xl_i(X)$ can be preceded by an action $u_i(Y)$, for any Y.

3. *Legality of schedules*: An element may either be locked exclusively by one transaction or by several in shared mode, but not both. More precisely:

   (a) If $xl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$ or $sl_j(X)$, for some j other than i, without an intervening $u_i(X)$.

   (b) If $sl_i(X)$ appears in a schedule, then there cannot be a following $xl_j(X)$, for $j \neq i$, without an intervening $u_i(X)$.

   Note that we *do* allow one transaction to request and hold both shared and exclusive locks on the same element, provided its doing so does not conflict with the lock(s) of other transactions. If transactions know in advance their needs for locks, then only the exclusive lock would have to be requested, but if lock needs are unpredictable, then it is possible that one transaction would request both shared and exclusive locks at different times.

Example 18.13: Let us examine a possible schedule of the following two transactions, using shared and exclusive locks:

$$T_1\text{: } sl_1(A);\ r_1(A);\ xl_1(B);\ r_1(B);\ w_1(B);\ u_1(A);\ u_1(B);$$
$$T_2\text{: } sl_2(A);\ r_2(A);\ sl_2(B);\ r_2(B);\ u_2(A);\ u_2(B);$$

Both T_1 and T_2 read A and B, but only T_1 writes B. Neither writes A.

In Fig. 18.15 is an interleaving of the actions of T_1 and T_2 in which T_1 begins by getting a shared lock on A. Then, T_2 follows by getting shared locks on both A and B. Now, T_1 needs an exclusive lock on B, since it will both read and write B. However, it cannot get the exclusive lock because T_2 already has a shared lock on B. Thus, the scheduler forces T_1 to wait. Eventually, T_2 releases the lock on B. At that time, T_1 may complete. □

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$xl_1(B)$ **Denied**	
	$u_2(A)$; $u_2(B)$
$xl_1(B)$; $r_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

Figure 18.15: A schedule using shared and exclusive locks

Notice that the resulting schedule in Fig 18.15 is conflict-serializable. The conflict-equivalent serial order is (T_2, T_1), even though T_1 started first. The argument we gave in Section 18.3.4 to show that legal schedules of consistent, 2PL transactions are conflict-serializable applies to systems with shared and exclusive locks as well. In Fig. 18.15, T_2 unlocks before T_1, so we would expect T_2 to precede T_1 in the serial order.

18.4.2 Compatibility Matrices

If we use several lock modes, then the scheduler needs a policy about when it can grant a lock request, given the other locks that may already be held on the same database element. A *compatibility matrix* is a convenient way to describe lock-management policies. It has a row and column for each lock mode. The rows correspond to a lock that is already held on an element X by another transaction, and the columns correspond to the mode of a lock on X that is requested. The rule for using a compatibility matrix for lock-granting decisions is:

- We can grant the lock on X in mode C if and only if for every row R such that there is already a lock on X in mode R by some other transaction, there is a "Yes" in column C.

		Lock requested	
		S	X
Lock held	S	Yes	No
in mode	X	No	No

Figure 18.16: The compatibility matrix for shared and exclusive locks

Example 18.14: Figure 18.16 is the compatibility matrix for shared (S) and exclusive (X) locks. The column for S says that we can grant a shared lock on an element if the only locks held on that element currently are shared locks. The column for X says that we can grant an exclusive lock only if there are no other locks held currently. □

18.4.3 Upgrading Locks

A transaction T that takes a shared lock on X is being "friendly" toward other transactions, since they are allowed to read X at the same time T is. Thus, we might wonder whether it would be friendlier still if a transaction T that wants to read and write a new value of X were first to take a shared lock on X, and only later, when T was ready to write the new value, *upgrade* the lock to exclusive (i.e., request an exclusive lock on X in addition to its already held shared lock on X). There is nothing that prevents a transaction from issuing requests for locks on the same database element in different modes. We adopt the convention that $u_i(X)$ releases all locks on X held by transaction T_i, although we could introduce mode-specific unlock actions if there were a use for them.

Example 18.15: In the following example, transaction T_1 is able to perform its computation concurrently with T_2, which would not be possible had T_1 taken an exclusive lock on B initially. The two transactions are:

T_1: $sl_1(A)$; $r_1(A)$; $sl_1(B)$; $r_1(B)$; $xl_1(B)$; $w_1(B)$; $u_1(A)$; $u_1(B)$;
T_2: $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$; $u_2(A)$; $u_2(B)$;

Here, T_1 reads A and B and performs some (possibly lengthy) calculation with them, eventually using the result to write a new value of B. Notice that T_1 takes a shared lock on B first, and later, after its calculation involving A and B is finished, requests an exclusive lock on B. Transaction T_2 only reads A and B, and does not write.

T_1	T_2
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$sl_1(B)$; $r_1(B)$;	
$xl_1(B)$ **Denied**	
	$u_2(A)$; $u_2(B)$
$xl_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_2(B)$;	

Figure 18.17: Upgrading locks allows more concurrent operation

Figure 18.17 shows a possible schedule of actions. T_2 gets a shared lock on B before T_1 does, but on the fourth line, T_1 is also able to lock B in shared mode. Thus, T_1 has both A and B and can perform its computation using their values. It is not until T_1 tries to upgrade its lock on B to exclusive that the scheduler must deny the request and force T_1 to wait until T_2 releases its lock on B. At that time, T_1 gets its exclusive lock on B, writes B, and finishes.

Notice that had T_1 asked for an exclusive lock on B initially, before reading B, then the request would have been denied, because T_2 already had a shared lock on B. T_1 could not perform its computation without reading B, and so T_1 would have more to do after T_2 releases its locks. As a result, T_1 finishes later using only an exclusive lock on B than it would if it used the upgrading strategy. □

Example 18.16: Unfortunately, indiscriminate use of upgrading introduces a new and potentially serious source of deadlocks. Suppose, that T_1 and T_2 each read database element A and write a new value for A. If both transactions use an upgrading approach, first getting a shared lock on A and then upgrading it to exclusive, the sequence of events suggested in Fig. 18.18 will happen whenever T_1 and T_2 initiate at approximately the same time.

T_1	T_2
$sl_1(A)$	
	$sl_2(A)$
$xl_1(A)$ **Denied**	
	$xl_2(A)$ **Denied**

Figure 18.18: Upgrading by two transactions can cause a deadlock

T_1 and T_2 are both able to get shared locks on A. Then, they each try to upgrade to exclusive, but the scheduler forces each to wait because the other has a shared lock on A. Thus, neither can make progress, and they will each wait forever, or until the system discovers that there is a deadlock, aborts one of the two transactions, and gives the other the exclusive lock on A. □

18.4.4 Update Locks

We can avoid the deadlock problem of Example 18.16 with a third lock mode, called *update locks*. An update lock $ul_i(X)$ gives transaction T_i only the privilege to read X, not to write X. However, only the update lock can be upgraded to a write lock later; a read lock cannot be upgraded. We can grant an update lock on X when there are already shared locks on X, but once there is an update lock on X we prevent additional locks of any kind — shared, update, or exclusive — from being taken on X. The reason is that if we don't deny such locks, then the updater might never get a chance to upgrade to exclusive, since there would always be other locks on X.

This rule leads to an asymmetric compatibility matrix, because the update (U) lock looks like a shared lock when we are requesting it and looks like an exclusive lock when we already have it. Thus, the columns for U and S locks are the same, and the rows for U and X locks are the same. The matrix is shown in Fig. 18.19.[3]

	S	X	U
S	Yes	No	Yes
X	No	No	No
U	No	No	No

Figure 18.19: Compatibility matrix for shared, exclusive, and update locks

Example 18.17: The use of update locks would have no effect on Example 18.15. As its third action, T_1 would take an update lock on B, rather than a shared lock. But the update lock would be granted, since only shared locks are held on B, and the same sequence of actions shown in Fig. 18.17 would occur.

However, update locks fix the problem shown in Example 18.16. Now, both T_1 and T_2 first request update locks on A and only later take exclusive locks. Possible descriptions of T_1 and T_2 are:

$$T_1\text{: } ul_1(A);\ r_1(A);\ xl_1(A);\ w_1(A);\ u_1(A);$$
$$T_2\text{: } ul_2(A);\ r_2(A);\ xl_2(A);\ w_2(A);\ u_2(A);$$

The sequence of events corresponding to Fig. 18.18 is shown in Fig. 18.20. Now, T_2, the second to request an update lock on A, is denied. T_1 is allowed to finish, and then T_2 may proceed. The lock system has effectively prevented concurrent execution of T_1 and T_2, but in this example, any significant amount of concurrent execution will result in either a deadlock or an inconsistent database state. □

T_1	T_2
$ul_1(A);\ r_1(A);$	
	$ul_2(A)$ **Denied**
$xl_1(A);\ w_1(A);\ u_1(A);$	
	$ul_2(A);\ r_2(A);$
	$xl_2(A);\ w_2(A);\ u_2(A);$

Figure 18.20: Correct execution using update locks

[3]Remember, however, that there is an additional condition regarding legality of schedules that is not reflected by this matrix: a transaction holding a shared lock but not an update lock on an element X cannot be given an exclusive lock on X, even though we do not in general prohibit a transaction from holding multiple locks on an element.

18.4.5 Increment Locks

Another interesting kind of lock that is useful in some situations is an "increment lock." Many transactions operate on the database only by incrementing or decrementing stored values. For example, consider a transaction that transfers money from one bank account to another.

The useful property of increment actions is that they commute with each other, since if two transactions add constants to the same database element, it does not matter which goes first, as the diagram of database state transitions in Fig. 18.21 suggests. On the other hand, incrementation commutes with neither reading nor writing; If you read A before or after it is incremented, you leave different values, and if you increment A before or after some other transaction writes a new value for A, you get different values of A in the database.

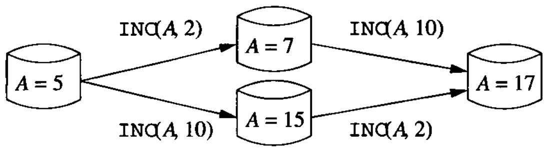

Figure 18.21: Two increment actions commute, since the final database state does not depend on which went first

Let us introduce as a possible action in transactions the *increment* action, written `INC(A,c)`. Informally, this action adds constant c to database element A, which we assume is a single number. Note that c could be negative, in which case we are really decrementing A. In practice, we might apply `INC` to a component of a tuple, while the tuple itself, rather than one of its components, is the lockable element. More formally, we use `INC(A,c)` to stand for the atomic execution of the following steps: `READ(A,t); t := t+c; WRITE(A,t);`.

Corresponding to the increment action, we need an *increment lock*. We shall denote the action of T_i requesting an increment lock on X by $il_i(X)$. We also use shorthand $inc_i(X)$ for the action in which transaction T_i increments database element X by some constant; the exact constant doesn't matter.

The existence of increment actions and locks requires us to make several modifications to our definitions of consistent transactions, conflicts, and legal schedules. These changes are:

a) A consistent transaction can only have an increment action on X if it holds an increment lock on X at the time. An increment lock does not enable either read or write actions, however.

b) In a legal schedule, any number of transactions can hold an increment lock on X at any time. However, if an increment lock on X is held by some transaction, then no other transaction can hold either a shared or exclusive lock on X at the same time. These requirements are expressed by the compatibility matrix of Fig. 18.22, where I represents a lock in increment mode.

c) The action $inc_i(X)$ conflicts with both $r_j(X)$ and $w_j(X)$, for $j \neq i$, but does not conflict with $inc_j(X)$.

	S	X	I
S	Yes	No	No
X	No	No	No
I	No	No	Yes

Figure 18.22: Compatibility matrix for shared, exclusive, and increment locks

Example 18.18: Consider two transactions, each of which read database element A and then increment B.

$$T_1\text{: } sl_1(A);\ r_1(A);\ il_1(B);\ inc_1(B);\ u_1(A);\ u_1(B);$$
$$T_2\text{: } sl_2(A);\ r_2(A);\ il_2(B);\ inc_2(B);\ u_2(A);\ u_2(B);$$

Notice that the transactions are consistent, since they only perform an incrementation while they have an increment lock, and they only read while they have a shared lock. Figure 18.23 shows a possible interleaving of T_1 and T_2. T_1 reads A first, but then T_2 both reads A and increments B. However, T_1 is then allowed to get its increment lock on B and proceed.

T_1	T_2
$sl_1(A);\ r_1(A);$	
	$sl_2(A);\ r_2(A);$
	$il_2(B);\ inc_2(B);$
$il_1(B);\ inc_1(B);$	
	$u_2(A);\ u_2(B);$
$u_1(A);\ u_1(B);$	

Figure 18.23: A schedule of transactions with increment actions and locks

Notice that the scheduler did not have to delay any requests in Fig. 18.23. Suppose, for instance, that T_1 increments B by A, and T_2 increments B by $2A$. They can execute in either order, since the value of A does not change, and the incrementations may also be performed in either order.

Put another way, we may look at the sequence of non-lock actions in the schedule of Fig. 18.23; they are:

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

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

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

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

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

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

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:** 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 18.4.3: 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) $r_1(A);\ r_2(B);\ inc_1(B);\ inc_2(C);\ w_1(C);\ w_2(D);$

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

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

Exercise 18.4.4: 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 18.4.5: 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 18.4.6:** 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 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) Write (c) Update (d) Increment.

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

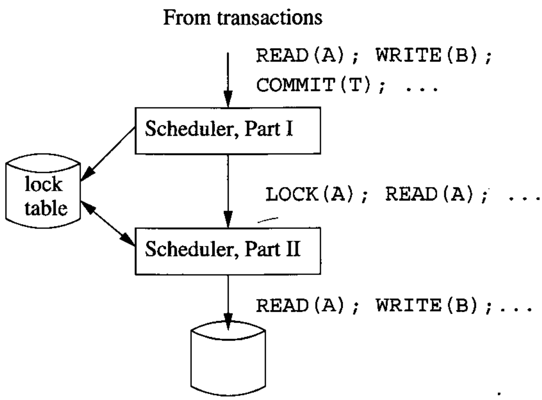

Figure 18.24: A scheduler that inserts lock requests into the transactions' request stream

allocated space by the operating system like other code and internal data of the DBMS.

Actions requested by a transaction are generally transmitted through the scheduler and executed on the database. However, under some circumstances a transaction is *delayed*, waiting for a lock, and its requests are not (yet) transmitted to the database. The two parts of the scheduler perform the following actions:

1. Part I takes the stream of requests generated by the transactions and inserts appropriate lock actions ahead of all database-access operations, such as read, write, increment, or update. The database access actions are then transmitted to Part II. Part I of the scheduler must select an appropriate lock mode from whatever set of lock modes the scheduler is using.

2. Part II takes the sequence of lock and database-access actions passed to it by Part I, and executes each appropriately. If a lock or database-access request is received by Part II, it determines whether the issuing transaction T is already delayed, because a lock has not been granted. If so, then the action is itself delayed and added to a list of actions that must eventually be performed for transaction T. If T is *not* delayed (i.e., all locks it previously requested have been granted already), then

   (a) If the action is a database access, it is transmitted to the database and executed.

   (b) If a lock action is received by Part II, it examines the lock table to see if the lock can be granted.

      i. If so, the lock table is modified to include the lock just granted.

ii. If not, then an entry must be made in the lock table to indicate that the lock has been requested. Part II of the scheduler then delays transaction T until such time as the lock is granted.

3. When a transaction T commits or aborts, Part I is notified by the transaction manager, and releases all locks held by T. If any transactions are waiting for any of these locks, Part I notifies Part II.

4. When Part II is notified that a lock on some database element X is available, it determines the next transaction or transactions that can now be given a lock on X. The transaction(s) that receive a lock are allowed to execute as many of their delayed actions as can execute, until they either complete or reach another lock request that cannot be granted.

Example 18.19: If there is only one kind of lock, as in Section 18.3, then the task of Part I of the scheduler is simple. If it sees any action on database element X, and it has not already inserted a lock request on X for that transaction, then it inserts the request. When a transaction commits or aborts, Part I can forget about that transaction after releasing its locks, so the memory required for Part I does not grow indefinitely.

When there are several kinds of locks, the scheduler may require advance notice of what future actions on the same database element will occur. Let us reconsider the case of shared-exclusive-update locks, using the transactions of Example 18.15, which we now write without any locks at all:

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

The messages sent to Part I of the scheduler must include not only the read or write request, but an indication of future actions on the same element. In particular, when $r_1(B)$ is sent, the scheduler needs to know that there will be a later $w_1(B)$ action (or might be such an action). There are several ways the information might be made available. For example, if the transaction is a query, we know it will not write anything. If the transaction is a SQL database modification command, then the query processor can determine in advance the database elements that might be both read and written. If the transaction is a program with embedded SQL, then the compiler has access to all the SQL statements (which are the only ones that can access the database) and can determine the potential database elements written.

In our example, suppose that events occur in the order suggested by Fig. 18.17. Then T_1 first issues $r_1(A)$. Since there will be no future upgrading of this lock, the scheduler inserts $sl_1(A)$ ahead of $r_1(A)$. Next, the requests from T_2 — $r_2(A)$ and $r_2(B)$ — arrive at the scheduler. Again there is no future upgrade, so the sequence of actions $sl_2(A)$; $r_2(A)$; $sl_2(B)$; $r_2(B)$ are issued by Part I.

Then, the action $r_1(B)$ arrives at the scheduler, along with a warning that this lock may be upgraded. The scheduler Part I thus emits $ul_1(B)$; $r_1(B)$ to

Part II. The latter consults the lock table and finds that it can grant the update lock on B to T_1, because there are only shared locks on B.

When the action $w_1(B)$ arrives at the scheduler, Part I emits $xl_1(B)$; $w_1(B)$. However, Part II cannot grant the $xl_1(B)$ request, because there is a shared lock on B for T_2. This and any subsequent actions from T_1 are delayed, stored by Part II for future execution. Eventually, T_2 commits, and Part I releases the locks on A and B that T_2 held. At that time, it is found that T_1 is waiting for a lock on B. Part II of the scheduler is notified, and it finds the lock $xl_1(B)$ is now available. It enters this lock into the lock table and proceeds to execute stored actions from T_1 to the extent possible. In this case, T_1 completes. □

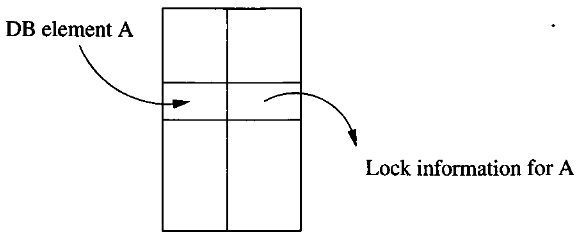

Figure 18.25: A lock table is a mapping from database elements to their lock information

18.5.2 The Lock Table

Abstractly, the lock table is a relation that associates database elements with locking information about that element, as suggested by Fig. 18.25. The table might, for instance, be implemented with a hash table, using (addresses of) database elements as the hash key. Any element that is not locked does not appear in the table, so the size is proportional to the number of locked elements only, not to the size of the entire database.

In Fig. 18.26 is an example of the sort of information we would find in a lock-table entry. This example structure assumes that the shared-exclusive-update lock scheme of Section 18.4.4 is used by the scheduler. The entry shown for a typical database element A is a tuple with the following components:

1. The *group mode* is a summary of the most stringent conditions that a transaction requesting a new lock on A faces. Rather than comparing the lock request with every lock held by another transaction on the same element, we can simplify the grant/deny decision by comparing the request with only the group mode.[4] For the shared-exclusive-update (SXU) lock scheme, the rule is simple: the group mode:

[4]The lock manager must, however, deal with the possibility that the requesting transaction already has a lock in another mode on the same element. For instance, in the SXU lock system discussed, the lock manager may be able to grant an X-lock request if the requesting

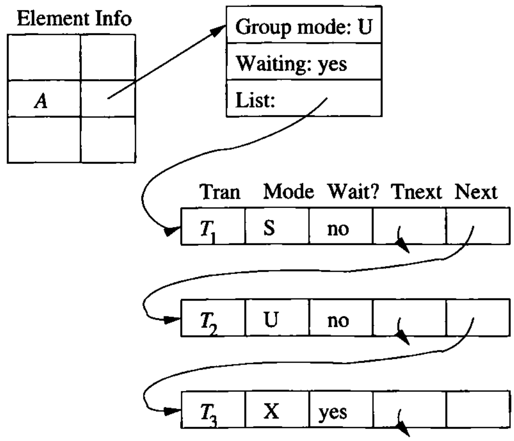

Figure 18.26: Structure of lock-table entries

(a) S means that only shared locks are held.

(b) U means that there is one update lock and perhaps one or more shared locks.

(c) X means there is one exclusive lock and no other locks.

For other lock schemes, there is usually an appropriate system of summaries by a group mode; we leave examples as exercises.

2. The *waiting* bit tells that there is at least one transaction waiting for a lock on A.

3. A list describing all those transactions that either currently hold locks on A or are waiting for a lock on A. Useful information that each list entry has might include:

   (a) The name of the transaction holding or waiting for a lock.

   (b) The mode of this lock.

   (c) Whether the transaction is holding or waiting for the lock.

   We also show in Fig. 18.26 two links for each entry. One links the entries themselves, and the other links all entries for a particular transaction (Tnext in the figure). The latter link would be used when a transaction commits or aborts, so that we can easily find all the locks that must be released.

transaction is the one that holds a U lock on the same element. For systems that do not support multiple locks held by one transaction on one element, the group mode always tells what the lock manager needs to know.

Handling Lock Requests

Suppose transaction T requests a lock on A. If there is no lock-table entry for A, then surely there are no locks on A, so the entry is created and the request is granted. If the lock-table entry for A exists, we use it to guide the decision about the lock request. We find the group mode, which in Fig. 18.26 is U, or "update." Once there is an update lock on an element, no other lock can be granted (except in the case that T itself holds the U lock and other locks are compatible with T's request). Thus, this request by T is denied, and an entry will be placed on the list saying T requests a lock (in whatever mode was requested), and `Wait? = 'yes'`.

If the group mode had been X (exclusive), then the same thing would happen, but if the group mode were S (shared), then another shared or update lock could be granted. In that case, the entry for T on the list would have `Wait? = 'no'`, and the group mode would be changed to U if the new lock were an update lock; otherwise, the group mode would remain S. Whether or not the lock is granted, the new list entry is linked properly, through its `Tnext` and `Next` fields. Notice that whether or not the lock is granted, the entry in the lock table tells the scheduler what it needs to know without having to examine the list of locks.

Handling Unlocks

Now suppose transaction T unlocks A. T's entry on the list for A is deleted. If the lock held by T is not the same as the group mode (e.g., T held an S lock, while the group mode was U), then there is no reason to change the group mode. On the other hand, if T's lock is in the group mode, we may have to examine the entire list to find the new group mode. In the example of Fig. 18.26, we know there can be only one U lock on an element, so if that lock is released, the new group mode could be only S (if there are shared locks remaining) or nothing (if no other locks are currently held).[5] If the group mode is X, we know there are no other locks, and if the group mode is S, we need to determine whether there are other shared locks.

If the value of `Waiting` is `'yes'`, then we need to grant one or more locks from the list of requested locks. There are several different approaches, each with its advantages:

1. *First-come-first-served*: Grant the lock request that has been waiting the longest. This strategy guarantees no *starvation*, the situation where a transaction can wait forever for a lock.

2. *Priority to shared locks*: First grant all the shared locks waiting. Then, grant one update lock, if there are any waiting. Only grant an exclusive lock if no others are waiting. This strategy can allow starvation, if a transaction is waiting for a U or X lock.

[5] We would never actually see a group mode of "nothing," since if there are no locks and no lock requests on an element, then there is no lock-table entry for that element.

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: 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.

Exercise 18.5.2: For each of the schedules of Exercise 18.2.4, tell the steps that the locking scheduler described in this section would execute.

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