CHAPTER 18

Concurrency Control

We saw in Chapter 17 that one of the fundamental properties of a transaction is isolation. When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes. In this chapter, we consider the management of concurrently executing transactions, and we ignore failures. In Chapter 19, we shall see how the system can recover from failures.

As we shall see, there are a variety of concurrency-control schemes. No one scheme is clearly the best; each one has advantages. In practice, the most frequently used schemes are *two-phase locking* and *snapshot isolation*.

18.1 Lock-Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item. We introduced the concept of locking in Section 17.9.

18.1.1 Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. **Shared**. If a transaction T_i has obtained a **shared-mode lock** (denoted by S) on item Q, then T_i can read, but cannot write, Q.
2. **Exclusive**. If a transaction T_i has obtained an **exclusive-mode lock** (denoted by X) on item Q, then T_i can both read and write Q.

We require that every transaction **request** a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q. The transaction

	S	X
S	true	false
X	false	false

Figure 18.1 Lock-compatibility matrix comp.

makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

To state this more generally, given a set of lock modes, we can define a **compatibility function** on them as follows: Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is **compatible** with mode B. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure 18.1. An element comp(A, B) of the matrix has the value *true* if and only if mode A is compatible with mode B.

Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(Q) instruction. A transaction can unlock a data item Q by the unlock(Q) instruction.

To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency-control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, T_i is made to **wait** until all incompatible locks held by other transactions have been released.

Transaction T_i may unlock a data item that it had locked at some earlier point. Note that a transaction must hold a lock on a data item as long as it accesses that item. Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured.

As an illustration, consider again the banking example that we introduced in Chapter 17. Let A and B be two accounts that are accessed by transactions T_1 and T_2. Transaction T_1 transfers \$50 from account B to account A (Figure 18.2). Transaction T_2 displays the total amount of money in accounts A and B—that is, the sum $A + B$ (Figure 18.3).

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Figure 18.2 Transaction T_1.

Suppose that the values of accounts A and B are \$100 and \$200, respectively. If these two transactions are executed serially, either in the order T_1, T_2 or the order T_2, T_1, then transaction T_2 will display the value \$300. If, however, these transactions are executed concurrently, then schedule 1, in Figure 18.4, is possible. In this case, transaction T_2 displays \$250, which is incorrect. The reason for this mistake is that the transaction T_1 unlocked data item B too early, as a result of which T_2 saw an inconsistent state.

The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks. The transaction making a lock request cannot execute its next action until the concurrency-control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction. Exactly when within this interval the lock is granted is not important; we can safely assume that the lock is granted just before the following action of the transaction. We shall therefore drop the

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure 18.3 Transaction T_2.

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figure 18.4 Schedule 1.

column depicting the actions of the concurrency-control manager from all schedules depicted in the rest of the chapter. We let you infer when locks are granted.

Suppose now that unlocking is delayed to the end of the transaction. Transaction T_3 corresponds to T_1 with unlocking delayed (Figure 18.5). Transaction T_4 corresponds to T_2 with unlocking delayed (Figure 18.6).

You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of $250 being displayed, is no longer possible with T_3 and T_4. Other schedules are possible. T_4 will not print out an inconsistent result in any of them; we shall see why later.

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule of Figure 18.7 for T_3 and T_4. Since T_3 is holding an exclusive-mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**. When deadlock occurs, the system must roll back one of

```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

Figure 18.5 Transaction T_3 (transaction T_1 with unlocking delayed).

the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution. We shall return to the issue of deadlock handling in Section 18.2.

If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states. On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur. There are ways to avoid deadlock in some situations, as we shall see in Section 18.1.5. However, in general, deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system.

We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such

```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

Figure 18.6 Transaction T_4 (transaction T_2 with unlocking delayed).

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 18.7 Schedule 2.

schedules is a proper subset of all possible serializable schedules. We shall present several locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation. Before doing so, we introduce some terminology.

Let $\{T_0, T_1, \ldots, T_n\}$ be a set of transactions participating in a schedule S. We say that T_i **precedes** T_j in S, written $T_i \rightarrow T_j$, if there exists a data item Q such that T_i has held lock mode A on Q, and T_j has held lock mode B on Q later, and comp(A,B) = false. If $T_i \rightarrow T_j$, then that precedence implies that in any equivalent serial schedule, T_i must appear before T_j. Observe that this graph is similar to the precedence graph that we used in Section 17.6 to test for conflict serializability. Conflicts between instructions correspond to noncompatibility of lock modes.

We say that a schedule S is **legal** under a given locking protocol if S is a possible schedule for a set of transactions that follows the rules of the locking protocol. We say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable; in other words, for all legal schedules the associated $\rightarrow$ relation is acyclic.

18.1.2 Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it

is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be **starved**.

We can avoid starvation of transactions by granting locks in the following manner: When a transaction T_i requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that:

- There is no other transaction holding a lock on Q in a mode that conflicts with M.
- There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i.

Thus, a lock request will never get blocked by a lock request that is made later.

18.1.3 The Two-Phase Locking Protocol

One protocol that ensures serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase**. A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase**. A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions T_3 and T_4 are two phase. On the other hand, transactions T_1 and T_2 are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction T_3, we could move the unlock(B) instruction to just after the lock-X(A) instruction and still retain the two-phase locking property.

We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions. We leave the proof as an exercise for you to do (see Practice Exercise 18.1).

Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions T_3 and T_4 are two phase, but, in schedule 2 (Figure 18.7), they are deadlocked.

Recall from Section 17.7.2 that, in addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure 18.8. Each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7.

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure 18.8 Partial schedule under two-phase locking.

Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits. We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit.

Consider the following two transactions, for which we have shown only some of the significant read and write operations:

T_8: read(a_1);
read(a_2);
...
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

If we employ the two-phase locking protocol, then T_8 must lock a_1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that T_8 needs an exclusive lock on a_1 only at the end of

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Figure 18.9 Incomplete schedule with a lock conversion.

its execution, when it writes a_1. Thus, if T_8 could initially lock a_1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since T_8 and T_9 could access a_1 and a_2 simultaneously.

This observation leads us to a refinement of the basic two-phase locking protocol, in which **lock conversions** are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Returning to our example, transactions T_8 and T_9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure 18.9, where only some of the locking instructions are shown.

Note that a transaction attempting to upgrade a lock on an item Q may be forced to wait. This enforced wait occurs if Q is currently locked by *another* transaction in shared mode.

Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol. However, to obtain conflict-serializable schedules through non-two-phase locking protocols, we need either to have additional information about the transactions or to impose some structure or ordering on the set of data items in the database. We shall see examples when we consider other locking protocols later in this chapter.

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a read(Q) operation, the system issues a lock-S(Q) instruction followed by the read(Q) instruction.
- When T_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the write(Q) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

18.1.4 Implementation of Locking

A **lock manager** can be implemented as a process that receives messages from transactions and sends messages in reply. The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlocks). Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.

The lock manager uses this data structure: For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived. It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the **lock table**. Each record of the linked list for a data item notes which transaction made the request, and what lock mode it requested. The record also notes if the request has currently been granted.

Figure 18.10 shows an example of a lock table. The table contains locks for five different data items, I4, I7, I23, I44, and I912. The lock table uses overflow chaining, so there is a linked list of data items for each entry in the lock table. There is also a list of transactions that have been granted locks, or are waiting for locks, for each of the data items. Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade. We have omitted the lock mode to keep the figure simple. It can be seen, for example, that T23 has been granted locks on I912 and I7 and is waiting for a lock on I4.

Although the figure does not show it, the lock table should also maintain an index on transaction identifiers so that it is possible to determine efficiently the set of locks held by a given transaction.

The lock manager processes requests this way:

- When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.

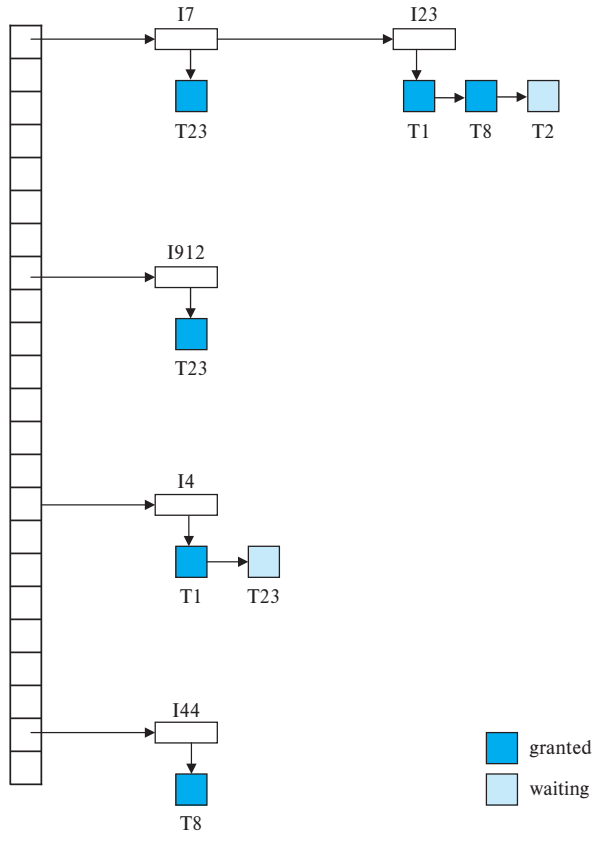

Figure 18.10 Lock table.

It always grants a lock request on a data item that is not currently locked. But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.

- When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transaction. It tests the record that follows, if any, as described in the previous paragraph, to see if that request can now be granted. If it can, the lock manager grants that request and processes the record following it, if any, similarly, and so on.
- If a transaction aborts, the lock manager deletes any waiting request made by the transaction. Once the database system has taken appropriate actions to undo the transaction (see Section 19.3), it releases all locks held by the aborted transaction.

This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted. We study how to detect and handle deadlocks later, in Section 18.2.2. Section 20.3.1 describes an alternative implementation—one that uses shared memory instead of message passing for lock request/grant.

18.1.5 Graph-Based Protocols

As noted in Section 18.1.3, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set $\mathbf{D} = \{d_1, d_2, \ldots, d_h\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j. This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

The partial ordering implies that the set **D** may now be viewed as a directed acyclic graph, called a **database graph**. In this section, for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We shall present a simple protocol, called the *tree protocol*, which is restricted to employ only *exclusive* locks. References to other, more complex, graph-based locking protocols are in the online bibliographical notes.

In the **tree protocol**, the only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

All schedules that are legal under the tree protocol are conflict serializable.

To illustrate this protocol, consider the database graph of Figure 18.11. The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

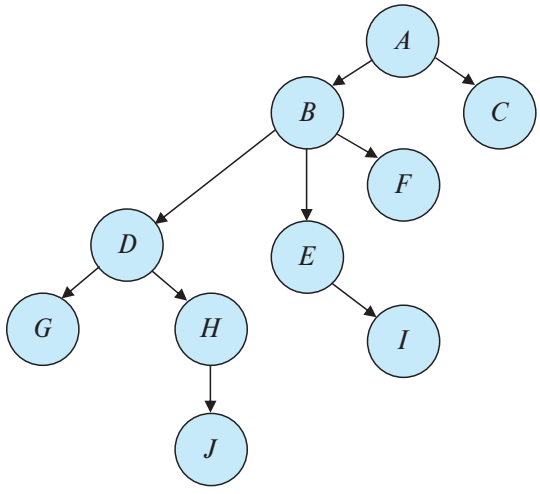

Figure 18.11 Tree-structured database graph.

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

One possible schedule in which these four transactions participated appears in Figure 18.12. Note that, during its execution, transaction T_{10} holds locks on two *disjoint* subtrees.

Observe that the schedule of Figure 18.12 is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.

The tree protocol in Figure 18.12 does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency. Here is an alternative that improves concurrency, but ensures only recoverability: For each data item with an uncommitted write, we record which transaction performed the last write to the data item. Whenever a transaction T_i performs a read of an uncommitted data item, we record a **commit dependency** of T_i on the transaction that performed the last write to the data item. Transaction T_i is then not permitted to commit until the commit of all transactions on which it has a commit dependency. If any of these transactions aborts, T_i must also be aborted.

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

Figure 18.12 Serializable schedule under the tree protocol.

The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times and to an increase in concurrency.

However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items A and J in the database graph of Figure 18.11 must lock not only A and J, but also data items B, D, and H. This additional locking results in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency. Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

For a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the tree protocol. Indeed, there are schedules possible under the two-phase locking protocol that are not possible under the tree protocol, and vice versa. Examples of such schedules are explored in the exercises.

18.2 Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and … , and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.

There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme. As we shall see, both methods may result in transaction rollback. Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

18.2.1 Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery, and it performs transaction rollback instead of waiting for a lock whenever the wait could potentially result in a deadlock.

The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages to this protocol: (1) it is often hard to predict, before the transaction begins, what data items need to be locked; (2) data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Another approach for preventing deadlocks is to impose an ordering of all data items and to require that a transaction lock data items only in a sequence consistent with the ordering. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items.

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering. This scheme is easy to implement,

as long as the set of data items accessed by a transaction is known when the transaction starts execution. There is no need to change the underlying concurrency-control system if two-phase locking is used: All that is needed is to ensure that locks are requested in the right order.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction T_j requests a lock that transaction T_i holds, the lock granted to T_i may be **preempted** by rolling back of T_i, and granting of the lock to T_j. To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its *old* timestamp when restarted. Two different deadlock-prevention schemes using timestamps have been proposed:

1. The **wait–die** scheme is a nonpreemptive technique. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j (i.e., T_i is older than T_j). Otherwise, T_i is rolled back (dies).

   For example, suppose that transactions T_{14}, T_{15}, and T_{16} have timestamps 5, 10, and 15, respectively. If T_{14} requests a data item held by T_{15}, then T_{14} will wait. If T_{16} requests a data item held by T_{15}, then T_{16} will be rolled back.

2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp larger than that of T_j (i.e., T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is *wounded* by T_i).

   Returning to our example, with transactions T_{14}, T_{15}, and T_{16}, if T_{14} requests a data item held by T_{15}, then the data item will be preempted from T_{15}, and T_{15} will be rolled back. If T_{16} requests a data item held by T_{15}, then T_{16} will wait.

The major problem with both of these schemes is that unnecessary rollbacks may occur.

Another simple approach to deadlock prevention is based on **lock timeouts**. In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts. If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed. This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery, which Section 18.2.2 discusses.

The timeout scheme is particularly easy to implement, and it works well if transactions are short and if long waits are likely to be due to deadlocks. However, in general it is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources.

Starvation is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

18.2.2 Deadlock Detection and Recovery

If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock. To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

In this section, we elaborate on these issues.

18.2.2.1 Deadlock Detection

Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. This graph consists of a pair $G = (V, E)$, where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair $T_i \rightarrow T_j$. If $T_i \rightarrow T_j$ is in E, then there is a directed edge from transaction T_i to T_j, implying that transaction T_i is waiting for transaction T_j to release a data item that it needs.

When transaction T_i requests a data item currently being held by transaction T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction T_j is no longer holding a data item needed by transaction T_i.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

To illustrate these concepts, consider the wait-for graph in Figure 18.13, which depicts the following situation:

- Transaction T_{17} is waiting for transactions T_{18} and T_{19}.
- Transaction T_{19} is waiting for transaction T_{18}.
- Transaction T_{18} is waiting for transaction T_{20}.

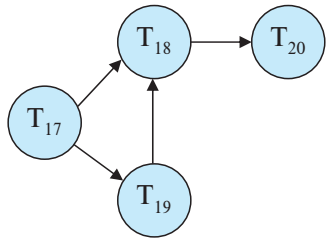

Figure 18.13 Wait-for graph with no cycle.

Since the graph has no cycle, the system is not in a deadlock state.

Suppose now that transaction T_{20} is requesting an item held by T_{19}. The edge $T_{20} \rightarrow T_{19}$ is added to the wait-for graph, resulting in the new system state in Figure 18.14. This time, the graph contains the cycle:

$$T_{18} \rightarrow T_{20} \rightarrow T_{19} \rightarrow T_{18}$$

implying that transactions T_{18}, T_{19}, and T_{20} are all deadlocked.

Consequently, the question arises: When should we invoke the detection algorithm? The answer depends on two factors:

1. How often does a deadlock occur?
2. How many transactions will be affected by the deadlock?

If deadlocks occur frequently, then the detection algorithm should be invoked more frequently. Data items allocated to deadlocked transactions will be unavailable to other transactions until the deadlock can be broken. In addition, the number of cycles in the graph may also grow. In the worst case, we would invoke the detection algorithm every time a request for allocation could not be granted immediately.

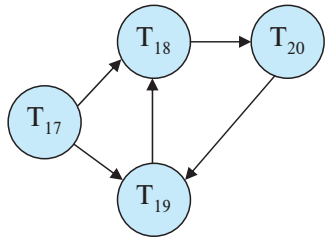

Figure 18.14 Wait-for graph with a cycle.

18.2.2.2 Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim**. Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including:
   a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
   b. How many data items the transaction has used.
   c. How many more data items the transaction needs for it to complete.
   d. How many transactions will be involved in the rollback.
2. **Rollback**. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
   The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the online bibliographical notes for relevant references.
3. **Starvation**. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**. We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

18.3 Multiple Granularity

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction T_i needs to access an entire relation, and a locking protocol is used to lock tuples, then T_i must lock each tuple in the relation. Clearly, acquiring many such locks is time-consuming; even worse, the lock table may become very large and no longer fit in memory. It would be better if T_i could issue a *single* lock request to lock the entire relation. On the other hand, if transaction T_j needs to access only a few tuples, it should not be required to lock the entire relation, since otherwise concurrency is lost.

What is needed is a mechanism to allow the system to define multiple levels of **granularity**. This is done by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Note that the tree that we describe here is significantly different from that used by the tree protocol (Section 18.1.5). A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.

As an illustration, consider the tree of Figure 18.15, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type *area*; the database consists of exactly these areas. Each area in turn has nodes of type *file* as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type *record*. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol, we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction T_i gets an **explicit lock** on file F_c of Figure 18.15, in exclusive mode, then it has an **implicit**

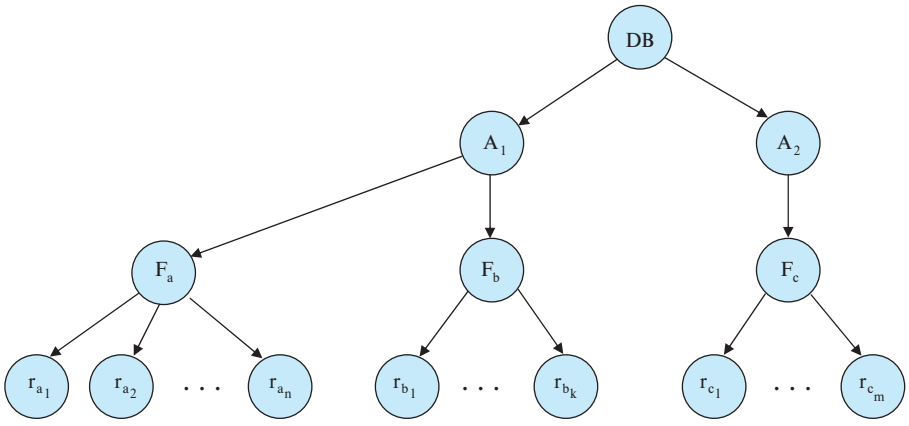

Figure 18.15 Granularity hierarchy.

lock in exclusive mode on all the records belonging to that file. It does not need to lock the individual records of F_c explicitly.

Suppose that transaction T_j wishes to lock record r_{b_6} of file F_b. Since T_i has locked F_b explicitly, it follows that r_{b_6} is also locked (implicitly). But, when T_j issues a lock request for r_{b_6}, r_{b_6} is not explicitly locked! How does the system determine whether T_j can lock r_{b_6}? T_j must traverse the tree from the root to record r_{b_6}. If any node in that path is locked in an incompatible mode, then T_j must be delayed.

Suppose now that transaction T_k wishes to lock the entire database. To do so, it simply must lock the root of the hierarchy. Note, however, that T_k should not succeed in locking the root node, since T_i is currently holding a lock on part of the tree (specifically, on file F_b). But how does the system determine if the root node can be locked? One possibility is for it to search the entire tree. This solution, however, defeats the whole purpose of the multiple-granularity locking scheme. A more efficient way to gain this knowledge is to introduce a new class of lock modes, called **intention lock modes**. If a node is locked in an intention mode, explicit locking is done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node–say, Q–must traverse a path in the tree from the root to Q. While traversing the tree, the transaction locks the various nodes in an intention mode.

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks. The compatibility function for these lock modes is shown in Figure 18.16.

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 18.16 Compatibility matrix.

The **multiple-granularity locking protocol** uses these lock modes to ensure serializability. It requires that a transaction T_i that attempts to lock a node Q must follow these rules:

- Transaction T_i must observe the lock-compatibility function of Figure 18.16.
- Transaction T_i must lock the root of the tree first and can lock it in any mode.
- Transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either IX or IS mode.
- Transaction T_i can lock a node Q in X, SIX, or IX mode only if T_i currently has the parent of Q locked in either IX or SIX mode.
- Transaction T_i can lock a node only if T_i has not previously unlocked any node (i.e., T_i is two phase).
- Transaction T_i can unlock a node Q only if T_i currently has none of the children of Q locked.

Observe that the multiple-granularity protocol requires that locks be acquired in *top-down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order. Deadlock is possible in the multiple-granularity protocol, as it is in the two-phase locking protocol.

As an illustration of the protocol, consider the tree of Figure 18.15 and these transactions:

- Suppose that transaction T_{21} reads record r_{a_2} in file F_a. Then, T_{21} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a_2} in S mode.
- Suppose that transaction T_{22} modifies record r_{a_9} in file F_a. Then, T_{22} needs to lock the database, area A_1, and file F_a (and in that order) in IX mode, and finally to lock r_{a_9} in X mode.
- Suppose that transaction T_{23} reads all the records in file F_a. Then, T_{23} needs to lock the database and area A_1 (and in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T_{24} reads the entire database. It can do so after locking the database in S mode.

We note that transactions T_{21}, T_{23}, and T_{24} can access the database concurrently. Transaction T_{22} can execute concurrently with T_{21}, but not with either T_{23} or T_{24}.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of:

- Short transactions that access only a few data items.
- Long transactions that produce reports from an entire file or set of files.

The number of locks that an SQL query may need to acquire can usually be estimated based on the relation scan operations performed by a query. A relation scan, for example, would acquire a lock at a relation level, while an index scan that is expected to fetch only a few records may acquire an intention lock at the relation level and regular locks at the tuple level. In case the a transaction acquires a large number of tuple locks, the lock table may become overfull. To deal with this situation, the lock manager may perform **lock escalation**, replacing many lower level locks by a single higher level lock; in our example, a single relation lock could replace a large number of tuple locks.

18.4 Insert Operations, Delete Operations, and Predicate Reads

Until now, we have restricted our attention to read and write operations. This restriction limits transactions to data items already in the database. Some transactions require not only access to existing data items, but also the ability to create new data items. Others require the ability to delete data items. To examine how such transactions affect concurrency control, we introduce these additional operations:

- **delete**(Q) deletes data item Q from the database.
- **insert**(Q) inserts a new data item Q into the database and assigns Q an initial value.

An attempt by a transaction T_i to perform a read(Q) operation after Q has been deleted results in a logical error in T_i. Likewise, an attempt by a transaction T_i to perform a read(Q) operation before Q has been inserted results in a logical error in T_i. It is also a logical error to attempt to delete a nonexistent data item.

18.4.1 Deletion

To understand how the presence of **delete** instructions affects concurrency control, we must decide when a **delete** instruction conflicts with another instruction. Let I_i and I_j be instructions of T_i and T_j, respectively, that appear in schedule S in consecutive order. Let I_i = **delete**(Q). We consider several instructions I_j.

- I_j = read(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the read operation successfully.
- I_j = write(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_j can execute the write operation successfully.
- I_j = **delete**(Q). I_i and I_j conflict. If I_i comes before I_j, T_j will have a logical error. If I_j comes before I_i, T_i will have a logical error.
- I_j = **insert**(Q). I_i and I_j conflict. Suppose that data item Q did not exist prior to the execution of I_i and I_j. Then, if I_i comes before I_j, a logical error results for T_i.

If I_j comes before I_i, then no logical error results. Likewise, if Q existed prior to the execution of I_i and I_j, then a logical error results if I_j comes before I_i, but not otherwise.

We can conclude the following:

- Under the two-phase locking protocol, an exclusive lock is required on a data item before that item can be deleted.
- Under the timestamp-ordering protocol, a test similar to that for a write must be performed. Suppose that transaction T_i issues **delete**(Q).
  - If TS(T_i) < R-timestamp(Q), then the value of Q that T_i was to delete has already been read by a transaction T_j with TS(T_j) > TS(T_i). Hence, the **delete** operation is rejected, and T_i is rolled back.
  - If TS(T_i) < W-timestamp(Q), then a transaction T_j with TS(T_j) > TS(T_i) has written Q. Hence, this **delete** operation is rejected, and T_i is rolled back.
  - Otherwise, the **delete** is executed.

18.4.2 Insertion

We have already seen that an **insert**(Q) operation conflicts with a **delete**(Q) operation. Similarly, **insert**(Q) conflicts with a read(Q) operation or a write(Q) operation; no read or write can be performed on a data item before it exists.

Since an **insert**(Q) assigns a value to data item Q, an **insert** is treated similarly to a write for concurrency-control purposes:

- Under the two-phase locking protocol, if T_i performs an **insert**(Q) operation, T_i is given an exclusive lock on the newly created data item Q.
- Under the timestamp-ordering protocol, if T_i performs an **insert**(Q) operation, the values R-timestamp(Q) and W-timestamp(Q) are set to TS(T_i).

18.4.3 Predicate Reads and The Phantom Phenomenon

Consider transaction T_{30} that executes the following SQL query on the university database:

```
select count(*)
from instructor
where dept_name = 'Physics' ;
```

Transaction T_{30} requires access to all tuples of the *instructor* relation pertaining to the Physics department.

Let T_{31} be a transaction that executes the following SQL insertion:

```
insert into instructor
    values (11111, 'Feynman', 'Physics', 94000);
```

Let S be a schedule involving T_{30} and T_{31}. We expect there to be potential for a conflict for the following reasons:

- If T_{30} uses the tuple newly inserted by T_{31} in computing **count**(*), then T_{30} reads a value written by T_{31}. Thus, in a serial schedule equivalent to S, T_{31} must come before T_{30}.
- If T_{30} does not use the tuple newly inserted by T_{31} in computing **count**(*), then in a serial schedule equivalent to S, T_{30} must come before T_{31}.

The second of these two cases is curious. T_{30} and T_{31} do not access any tuple in common, yet they conflict with each other! In effect, T_{30} and T_{31} conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. As a result, the system could fail to prevent a nonserializable schedule. This problem is an instance of the **phantom phenomenon**.

Phantom phenomena can occur not just with inserts, but also with updates. Consider the situation we saw in Section 17.10, where a transaction T_i used an index to find only tuples with *dept_name* = "Physics", and as a result did not read any tuples with other department names. If another transaction T_j updates one of these tuples, changing its department name to Physics, a problem similar to the above problem occurs: even though T_i and T_j have not accessed any tuples in common, they do conflict with each other. This problem too is an instance of the phantom phenomenon. In general, the phantom phenomenon is rooted in predicate reads that conflict with inserts or updates that result in new/updated tuples that satisfy the predicate.

We can prevent these problems by allowing transaction T_{30} to prevent other transactions from creating new tuples in the *instructor* relation with *dept_name* = "Physics", and from updating the department name of an existing *instructor* tuple to Physics.

To find all *instructor* tuples with *dept_name* = "Physics", T_{30} must search either the whole *instructor* relation, or at least an index on the relation. Up to now, we have assumed implicitly that the only data items accessed by a transaction are tuples. However, T_{30} is an example of a transaction that reads information about what tuples are in a relation, and T_{31} is an example of a transaction that updates that information.

Clearly, it is not sufficient merely to lock the tuples that are accessed; the information used to find the tuples that are accessed by the transaction must also be locked.

Locking of information used to find tuples can be implemented by associating a data item with the relation; the data item represents the information used to find the tuples in the relation. Transactions, such as T_{30}, that read the information about what tuples are in a relation would then have to lock the data item corresponding to the

relation in shared mode. Transactions, such as T_{31}, that update the information about what tuples are in a relation would have to lock the data item in exclusive mode. Thus, T_{30} and T_{31} would conflict on a real data item, rather than on a phantom. Similarly, transactions that use an index to retrieve tuples must lock the index itself.

Do not confuse the locking of an entire relation, as in multiple-granularity locking, with the locking of the data item corresponding to the relation. By locking the data item, a transaction only prevents other transactions from updating information about what tuples are in the relation. Locking is still required on tuples. A transaction that directly accesses a tuple can be granted a lock on the tuples even when another transaction has an exclusive lock on the data item corresponding to the relation itself.

The major disadvantage of locking a data item corresponding to the relation, or locking an entire index, is the low degree of concurrency— two transactions that insert different tuples into a relation are prevented from executing concurrently.

A better solution is an **index-locking** technique that avoids locking the whole index. Any transaction that inserts a tuple into a relation must insert information into every index maintained on the relation. We eliminate the phantom phenomenon by imposing a locking protocol for indices. For simplicity we shall consider only B^+-tree indices.

As we saw in Chapter 14, every search-key value is associated with an index leaf node. A query will usually use one or more indices to access a relation. An insert must insert the new tuple in all indices on the relation. In our example, we assume that there is an index on *instructor* for attribute *dept_name*. Then, T_{31} must modify the leaf containing the key "Physics". If T_{30} reads the same leaf node to locate all tuples pertaining to the Physics department, then T_{30} and T_{31} conflict on that leaf node.

The **index-locking protocol** takes advantage of the availability of indices on a relation, by turning instances of the phantom phenomenon into conflicts on locks on index leaf nodes. The protocol operates as follows:

- Every relation must have at least one index.
- A transaction T_i can access tuples of a relation only after first finding them through one or more of the indices on the relation. For the purpose of the index-locking protocol, a relation scan is treated as a scan through all the leaves of one of the indices.
- A transaction T_i that performs a lookup (whether a range lookup or a point lookup) must acquire a shared lock on all the index leaf nodes that it accesses.
- A transaction T_i may not insert, delete, or update a tuple t_i in a relation r without updating all indices on r. The transaction must obtain exclusive locks on all index leaf nodes that are affected by the insertion, deletion, or update. For insertion and deletion, the leaf nodes affected are those that contain (after insertion) or contained (before deletion) the search-key value of the tuple. For updates, the leaf nodes affected are those that (before the modification) contained the old value of the search key, and nodes that (after the modification) contain the new value of the search key.

- Locks are obtained on tuples as usual.
- The rules of the two-phase locking protocol must be observed.

Note that the index-locking protocol does not address concurrency control on internal nodes of an index; techniques for concurrency control on indices, which minimize lock conflicts, are presented in Section 18.10.2.

Locking an index leaf node prevents any update to the node, even if the update did not actually conflict with the predicate. A variant called key-value locking, which minimizes such false lock conflicts, is presented in Section 18.10.2 as part of index concurrency control.

As noted in Section 17.10, it would appear that the existence of a conflict between transactions depends on a low-level query-processing decision by the system that is unrelated to a user-level view of the meaning of the two transactions. An alternative approach to concurrency control acquires shared locks on predicates in a query, such as the predicate "*salary* > 90000" on the *instructor* relation. Inserts and deletes of the relation must then be checked to see if they satisfy the predicate; if they do, there is a lock conflict, forcing the insert or delete to wait till the predicate lock is released. For updates, both the initial value and the final value of the tuple must be checked against the predicate. Such conflicting inserts, deletes, and updates affect the set of tuples selected by the predicate, and they cannot be allowed to execute concurrently with the query that acquired the (shared) predicate lock. We call this protocol **predicate locking**;[1] predicate locking is not used in practice since it is more expensive to implement than the index-locking protocol and does not give significant additional benefits.

18.5 Timestamp-Based Protocols

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

18.5.1 Timestamps

With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i). This timestamp is assigned by the database system before the transaction T_i starts execution. If a transaction T_i has been assigned timestamp TS(T_i), and a new transaction T_j enters the system, then TS(T_i) < TS(T_j). There are two simple methods for implementing this scheme:

[1]The term *predicate locking* was used for a version of the protocol that used shared and exclusive locks on predicates, and was thus more complicated. The version we present here, with only shared locks on predicates, is also referred to as **precision locking**.

1. Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if $\text{TS}(T_i) < \text{TS}(T_j)$, then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

To implement this scheme, we associate with each data item Q two timestamp values:

1. **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully.
2. **R-timestamp**(Q) denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

18.5.2 The Timestamp-Ordering Protocol

The **timestamp-ordering protocol** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

- Suppose that transaction T_i issues read(Q).
  - If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and T_i is rolled back.
  - If $\text{TS}(T_i) \geq \text{W-timestamp}(Q)$, then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and $\text{TS}(T_i)$.
- Suppose that transaction T_i issues write(Q).
  - If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
  - If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls T_i back.
  - Otherwise, the system executes the write operation and sets W-timestamp(Q) to $\text{TS}(T_i)$.

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions T_{25} and T_{26}. Transaction T_{25} displays the contents of accounts A and B:

T_{25}: read(B);
read(A);
display($A + B$).

Transaction T_{26} transfers \$50 from account B to account A, and then displays the contents of both:

T_{26}: read(B);
$B := B - 50$;
write(B);
read(A);
$A := A + 50$;
write(A);
display($A + B$).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3 of Figure 18.17, $\text{TS}(T_{25}) < \text{TS}(T_{26})$, and the schedule is possible under the timestamp protocol.

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa (see Exercise 18.27).

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order.

The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If a transaction is suffering from repeated restarts, conflicting transactions need to be temporarily blocked to enable the transaction to finish.

The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

- Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Figure 18.17 Schedule 3.

- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits (see Exercise 18.28).
- Recoverability alone can be ensured by tracking uncommitted writes and allowing a transaction T_i to commit only after the commit of any transaction that wrote a value that T_i read. Commit dependencies, outlined in Section 18.1.5, can be used for this purpose.

If the timestamp-ordering protocol is applied only to tuples, the protocol would be vulnerable to the phantom problems that we saw in Section 17.10 and Section 18.4.3.

To avoid this problem, the timestamp-ordering protocol could be applied to all data that is read by a transaction, including relation metadata and index data. In the context of locking-based concurrency control, the index-locking protocol, described in Section 18.4.3, is a more efficient alternative for avoiding the phantom problem; recall that the index-locking protocol obtains locks on index nodes, in addition to obtaining locks on tuples. The timestamp-ordering protocol can be similarly modified to treat each index node as a data item, with associated read and write timestamps, and to apply the timestamp-ordering tests on these data items, too. This extended version of the timestamp-ordering protocol avoids phantom problems and ensures serializability even with predicate reads.

18.5.3 Thomas' Write Rule

We now present a modification to the timestamp-ordering protocol that allows greater potential concurrency than does the protocol of Section 18.5.2. Let us consider schedule 4 of Figure 18.18 and apply the timestamp-ordering protocol. Since T_{27} starts before T_{28}, we shall assume that $\text{TS}(T_{27}) < \text{TS}(T_{28})$. The read($Q$) operation of T_{27} succeeds, as does the write(Q) operation of T_{28}. When T_{27} attempts its write(Q) operation,

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

Figure 18.18 Schedule 4.

we find that TS(T_{27}) < W-timestamp(Q), since W-timestamp(Q) = TS(T_{28}). Thus, the write(Q) by T_{27} is rejected and transaction T_{27} must be rolled back.

Although the rollback of T_{27} is required by the timestamp-ordering protocol, it is unnecessary. Since T_{28} has already written Q, the value that T_{27} is attempting to write is one that will never need to be read. Any transaction T_i with TS(T_i) < TS(T_{28}) that attempts a read(Q) will be rolled back, since TS(T_i) < W-timestamp(Q). Any transaction T_j with TS(T_j) > TS(T_{28}) must read the value of Q written by T_{28}, rather than the value that T_{27} is attempting to write.

This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances. The protocol rules for read operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol of Section 18.5.2.

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this: Suppose that transaction T_i issues write(Q).

1. If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

The difference between these rules and those of Section 18.5.2 lies in the second rule. The timestamp-ordering protocol requires that T_i be rolled back if T_i issues write(Q) and TS(T_i) < W-timestamp(Q). However, here, in those cases where TS(T_i) ≥ R-timestamp(Q), we ignore the obsolete write.

By ignoring the write, Thomas' write rule allows schedules that are not conflict serializable but are nevertheless correct. Those non-conflict-serializable schedules allowed satisfy the definition of *view serializable* schedules (see Note 18.1 on page 867). Thomas' write rule makes use of view serializability by, in effect, deleting obsolete write operations from the transactions that issue them. This modification of transactions makes it possible to generate serializable schedules that would not be possible under the other

protocols presented in this chapter. For example, schedule 4 of Figure 18.18 is not conflict serializable and, thus, is not possible under the two-phase locking protocol, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T_{27} would be ignored. The result is a schedule that is *view equivalent* to the serial schedule $<T_{27}, T_{28}>$.

18.6 Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state. A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for *monitoring* the system.

The **validation protocol** requires that each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order:

1. **Read phase**. During this phase, the system executes transaction T_i. It reads the values of the various data items and stores them in variables local to T_i. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. The validation test (described below) is applied to transaction T_i. This determines whether T_i is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
3. **Write phase**. If the validation test succeeds for transaction T_i, the temporary local variables that hold the results of any write operations performed by T_i are copied to the database. Read-only transactions omit this phase.

Each transaction must go through the phases in the order shown. However, phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction T_i:

1. ***StartTS***(T_i), the time when T_i started its execution.
2. ***ValidationTS***(T_i), the time when T_i finished its read phase and started its validation phase.
3. ***FinishTS***(T_i), the time when T_i finished its write phase.

Note 18.1 VIEW SERIALIZABILITY

There is another form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be **view equivalent** if three conditions are met:

1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.
2. For each data item Q, if transaction T_i executes read(Q) in schedule S, and if that value was produced by a write(Q) operation executed by transaction T_j, then the read(Q) operation of transaction T_i must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of transaction T_j.
3. For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) in schedule S'.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation. Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state.

The concept of view equivalence leads to the concept of view serializability. We say that a schedule S is **view serializable** if it is view equivalent to a serial schedule.

As an illustration, suppose that we augment schedule 4 with transaction T_{29} and obtain the following view serializable (schedule 5):

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

Indeed, schedule 5 is view equivalent to the serial schedule $<T_{27}, T_{28}, T_{29}>$, since the one read(Q) instruction reads the initial value of Q in both schedules and T_{29} performs the final write of Q in both schedules.

Every conflict-serializable schedule is also view serializable, but there are view-serializable schedules that are not conflict serializable. Indeed, schedule 5 is not conflict serializable, since every pair of consecutive instructions conflicts, and, thus, no swapping of instructions is possible.

Note 18.1 VIEW SERIALIZABILITY (Cont.)

Observe that, in schedule 5, transactions T_{28} and T_{29} perform write(Q) operations without having performed a read(Q) operation. Writes of this sort are called **blind writes**. Blind writes appear in any view-serializable schedule that is not conflict serializable.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp $ValidationTS(T_i)$. Thus, the value $TS(T_i) = ValidationTS(T_i)$ and, if $TS(T_j) < TS(T_k)$, then any produced schedule must be equivalent to a serial schedule in which transaction T_j appears before transaction T_k.

The **validation test** for transaction T_i requires that, for all transactions T_k with $TS(T_k) < TS(T_i)$, one of the following two conditions must hold:

1. $FinishTS(T_k) < StartTS(T_i)$. Since T_k completes its execution before T_i started, the serializability order is indeed maintained.
2. The set of data items written by T_k does not intersect with the set of data items read by T_i, and T_k completes its write phase before T_i starts its validation phase ($StartTS(T_i) < FinishTS(T_k) < ValidationTS(T_i)$). This condition ensures that the writes of T_k and T_i do not overlap. Since the writes of T_k do not affect the read of T_i, and since T_i cannot affect the read of T_k, the serializability order is indeed maintained.

As an illustration, consider again transactions T_{25} and T_{26}. Suppose that $TS(T_{25}) < TS(T_{26})$. Then, the validation phase succeeds in the schedule 6 in Figure 18.19. Note that the writes to the actual variables are performed only after the validation phase of T_{26}. Thus, T_{25} reads the old values of B and A, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked to enable the long transaction to finish.

Note also that the validation conditions result in a transaction T only being validated again the set of transactions T_i that finished after T started, and, further, are serialized before T. Transactions that finished before T started can be ignored in the validation tests. Transactions T_i that are serialized after T (that is, they have $ValidationTS(T_i) > ValidationTS(T)$) can also be ignored; when such a transaction T_i is validated, it would be validated against T if T finished after T_i started.

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
<validate>	
display($A + B$)	
	<validate>
	write(B)
	write(A)

Figure 18.19 Schedule 6, a schedule produced by using validation.

This validation scheme is called the **optimistic concurrency-control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

It is possible to use $\text{TS}(T_i) = \textit{StartTS}(T_i)$ instead of $\textit{ValidationTS}(T_i)$ without affecting serializability. However, doing so may result in a transaction T_i entering the validation phase before a transaction T_j that has $\text{TS}(T_j) < \text{TS}(T_i)$. Then, the validation of T_i would have to wait for T_j to complete, so its read and write sets are completely known. Using *ValidationTS* avoids this problem.

18.7 Multiversion Schemes

The concurrency-control schemes discussed thus far ensure serializability by either delaying an operation or aborting the transaction that issued the operation. For example, a read operation may be delayed because the appropriate value has not been written yet; or it may be rejected (that is, the issuing transaction must be aborted) because the value that it was supposed to read has already been overwritten. These difficulties could be avoided if old copies of each data item were kept in a system.

In **multiversion concurrency-control** schemes, each write(Q) operation creates a new **version** of Q. When a transaction issues a read(Q) operation, the concurrency-control manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

18.7.1 Multiversion Timestamp Ordering

The timestamp-ordering protocol can be extended to a multiversion protocol. With each transaction T_i in the system, we associate a unique static timestamp, denoted by TS(T_i). The database system assigns this timestamp before the transaction starts execution, as described in Section 18.5.

With each data item Q, a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$ is associated. Each version Q_k contains three data fields:

1. **Content** is the value of version Q_k.
2. **W-timestamp**(Q_k) is the timestamp of the transaction that created version Q_k.
3. **R-timestamp**(Q_k) is the largest timestamp of any transaction that successfully read version Q_k.

A transaction—say, T_i—creates a new version Q_k of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by T_i. The system initializes the W-timestamp and R-timestamp to TS(T_i). It updates the R-timestamp value of Q_k whenever a transaction T_j reads the content of Q_k and R-timestamp(Q_k) < TS(T_j).

The **multiversion timestamp-ordering scheme** presented next ensures serializability. The scheme operates as follows: Suppose that transaction T_i issues a read(Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1. If transaction T_i issues a read(Q), then the value returned is the content of version Q_k.
2. If transaction T_i issues write(Q), and if TS(T_i) < R-timestamp(Q_k), then the system rolls back transaction T_i. On the other hand, if TS(T_i) = W-timestamp(Q_k), the system overwrites the contents of Q_k; otherwise (if TS(T_i) > R-timestamp(Q_k)), it creates a new version of Q.

The justification for rule 1 is clear. A transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is "too late" in doing a write. More precisely, if T_i attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed.

The **valid interval** of a version Q_i of Q with W-timestamp t is defined as follows: if Q_i is the latest version of Q, the interval is $[t, \infty]$; otherwise let the next version of Q have timestamp s; then the valid interval is $[t, s)$. You can easily verify that reads by a transaction with timestamp t_i return the content of the version whose valid interval contains t_i.

Versions that are no longer needed are removed according to the following rule: Suppose that there are two versions, Q_k and Q_j, of a data item, and that both versions

have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again, and can be deleted.

The multiversion timestamp-ordering scheme has the desirable property that a read request never fails and is never made to wait. In typical database systems, where reading is a more frequent operation than is writing, this advantage may be of major practical significance.

The scheme, however, suffers from two undesirable properties. First, the reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one. Second, the conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive. Section 18.7.2 describes an algorithm to alleviate this problem.

This multiversion timestamp-ordering scheme does not ensure recoverability and cascadelessness. It can be extended in the same manner as the basic timestamp-ordering scheme to make it recoverable and cascadeless.

18.7.2 Multiversion Two-Phase Locking

The **multiversion two-phase locking protocol** attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between **read-only transactions** and **update transactions**.

Update transactions perform rigorous two-phase locking; that is, they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order. Each version of a data item has a single timestamp. The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the ts-counter, that is incremented during commit processing.

The database system assigns read-only transactions a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp-ordering protocol for performing reads. Thus, when a read-only transaction T_i issues a read(Q), the value returned is the contents of the version whose timestamp is the largest timestamp less than or equal to TS(T_i).

When an update transaction reads an item, it gets a shared lock on the item and reads the latest version of that item. When an update transaction wants to write an item, it first gets an exclusive lock on the item and then creates a new version of the data item. The write is performed on the new version, and the timestamp of the new version is initially set to a value ∞, a value greater than that of any possible timestamp.

When the update transaction T_i completes its actions, it carries out commit processing; only one update transaction is allowed to perform commit processing at a time. First, T_i sets the timestamp on every version it has created to 1 more than the value of ts-counter; then, T_i increments ts-counter by 1, and commits.

Read-only transactions see the old value of ts-counter until T_i has successfully committed. As a result, read-only transactions that start after T_i commits will see the values updated by T_i, whereas those that start before T_i commits will see the value before the updates by T_i. In either case, read-only transactions never need to wait for

Note 18.2 MULTIVERSIONING AND DATABASE IMPLEMENTATION

Consider a database system that implements a primary key constraint by ensuring that only one tuple exists for any value of the primary key attribute. The creation of a second version of the record with the same primary key would appear to be a violation of the primary key constraint. However, it is logically not a violation, since the two versions do not coexist at any time in the database. Therefore, primary constraint enforcement must be modified to allow multiple records with the same primary key, as long as they are different versions of the same record.

Next, consider the issue of deletion of tuples. This can be implemented by creating a new version of the tuple, with timestamps created as usual, but with a special marker denoting that the tuple has been deleted. Transactions that read such a tuple simply skip it, since it has been deleted.

Further, consider the issue of enforcing foreign-key dependencies. Consider the case of a relation r whose attribute $r.B$ is a foreign-key referencing attribute $s.B$ of relation s. In general, deletion of a tuple t_s in s or update of a primary key attribute of tuple t_s in s causes a foreign-key violation if there is an r tuple t_r such that $t_r.B = t_s.B$. With multiversioning, if the timestamp of the transaction performing the deletion/update is ts_i, the corresponding condition for violation is the existence of such a tuple version t_r, with the additional condition that the valid interval of t_r contains ts_i.

Finally, consider the case of an index on attribute $r.B$ of relation r. If there are multiple versions of a record t_i with the same value for B, the index could point to the latest version of the record, and the latest version could have pointers to earlier versions. However, if an update was made to attribute $t_i.B$, the index would need to contain separate entries for different versions of record t_i; one entry for the old value of $t_i.B$ and another for the new value of $t_i.B$. When old versions of a record are deleted, any entry in the index for the old version must also be deleted.

locks. Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

Versions are deleted in a manner like that of multiversion timestamp ordering. Suppose there are two versions, Q_k and Q_j, of a data item, and that both versions have a timestamp less than or equal to the timestamp of the oldest read-only transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again and it can be deleted.

18.8 Snapshot Isolation

Snapshot isolation is a particular type of concurrency-control scheme that has gained wide acceptance in commercial and open-source systems, including Oracle,

PostgreSQL, and SQL Server. We introduced snapshot isolation in Section 17.9.3. Here, we take a more detailed look into how it works.

Conceptually, snapshot isolation involves giving a transaction a "snapshot" of the database at the time when it begins its execution. It then operates on that snapshot in complete isolation from concurrent transactions. The data values in the snapshot consist only of values written by committed transactions. This isolation is ideal for read-only transactions since they never wait and are never aborted by the concurrency manager.

Transactions that update the database potentially have conflicts with other transactions that update the database. Updates performed by a transaction must be validated before the transaction is allowed to commit. We describe how validation is performed, later in this section. Updates are kept in the transaction's private workspace until the transaction is validated, at which point the updates are written to the database.

When a transaction T is allowed to commit, the transition of T to the committed state and the writing of all of the updates made by T to the database must be conceptually done as an atomic action so that any snapshot created for another transaction either includes all updates by transaction T or none of them.

18.8.1 Multiversioning in Snapshot Isolation

To implement snapshot isolation, transactions are given two timestamps. The first timestamp, $StartTS(T_i)$, is the time at which transaction T_i started. The second timestamp, $CommitTS(T_i)$ is the time when the transaction T_i requested validation.

Note that timestamps can be wall clock time, as long as no two transactions are given the same timestamp, but they are usually assigned from a counter that is incremented every time a transaction enters its validation phase.

Snapshot isolation is based on multiversioning, and each transaction that updates a data item creates a version of the data item. Versions have only one timestamp, which is the write timestamp, indicating when the version was created. The timestamp of a version created by transaction T_i is set to $CommitTS(T_i)$. (Since updates to the database are also only made after validation of the transaction T_i, $CommitTS(T_i)$ is available when a version is created.)[2]

When a transaction T_i reads a data item, the latest version of the data item whose timestamp is $\leq StartTS(T_i)$ is returned to T_i. Thus, T_i does not see the updates of any transactions that committed after T_i started, while it does see the updates of all transactions that commit before it started. As a result, T_i effectively sees a snapshot of the database as of the time it started.[3]

---

[2]Many implementations create versions even before the transaction starts validation; since the version timestamp is not available at this point, the timestamp is set to infinity initially, and is updated to the correct value at the time of validation. Further optimizations are used in actual implementations, but we ignore them for simplicity.

[3]To efficiently find the correct version of a data item for a given timestamp, many implementations store not only the timestamp when a version was created, but also the timestamp when the next version was created, which can be considered an *invalidation timestamp* for that version; the version is valid between the creation and invalidation timestamps. The current version of a data item has the invalidation timestamp set to infinity.

18.8.2 Validation Steps for Update Transactions

Deciding whether or not to allow an update transaction to commit requires some care. Potentially, two transactions running concurrently might both update the same data item. Since these two transactions operate in isolation using their own private snapshots, neither transaction sees the update made by the other. If both transactions are allowed to write to the database, the first update written will be overwritten by the second. The result is a **lost update**. This must be prevented. There are two variants of snapshot isolation, both of which prevent lost updates. They are called *first committer wins* and *first updater wins*. Both approaches are based on testing the transaction against concurrent transactions.

A transaction T_j is said to be **concurrent with** a given transaction T_i if it was active or partially committed at any point from the start of T up to the point when validation of T_i started. Formally, T_j is concurrent with T_i if either

$$StartTS(T_j) \leq StartTS(T_i) \leq CommitTS(T_j), \text{ or}$$
$$StartTS(T_i) \leq StartTS(T_j) \leq CommitTS(T_i).$$

Under **first committer wins**, when a transaction T_i starts validation, the following actions are performed as part of validation, after its *CommitTS* is assigned. (We assume for simplicity that only one transaction performs validation at a time, although real implementations do support concurrent validation.)

- A test is made to see if any transaction that was concurrent with T has already written an update to the database for some data item that T intends to write. This can be done by checking for each data item d that T_i intends to write, whether there is a version of the data item d whose timestamp is between $StartTS(T_i)$ and $CommitTS(T_i)$.[4]
- If any such data item is found, then T_i aborts.
- If no such data item is found, then T commits and its updates are written to the database.

This approach is called "first committer wins" because if transactions conflict, the first one to be tested using the above rule succeeds in writing its updates, while the subsequent ones are forced to abort. Details of how to implement these tests are addressed in Exercise 18.15.

Under **first updater wins**, the system uses a locking mechanism that applies only to updates (reads are unaffected by this, since they do not obtain locks). When a transaction T_i attempts to update a data item, it requests a *write lock* on that data item. If the lock is not held by a concurrent transaction, the following steps are taken after the lock is acquired:

- If the item has been updated by any concurrent transaction, then T_i aborts.

[4]There are alternative implementations, based on keeping track of read and write sets for transactions.

- Otherwise T_i may proceed with its execution, including possibly committing.

If, however, some other concurrent transaction T_j already holds a write lock on that data item, then T_i cannot proceed, and the following rules are followed:

- T_i waits until T_j aborts or commits.
  - If T_j aborts, then the lock is released and T_i can obtain the lock. After the lock is acquired, the check for an update by a concurrent transaction is performed as described earlier: T_i aborts if a concurrent transaction had updated the data item, and it proceeds with its execution otherwise.
  - If T_j commits, then T_i must abort.

Locks are released when the transaction commits or aborts.

This approach is called "first updater wins" because if transactions conflict, the first one to obtain the lock is the one that is permitted to commit and perform its update. Those that attempt the update later abort unless the first updater subsequently aborts for some other reason. (As an alternative to waiting to see if the first updater T_j aborts, a subsequent updater T_i can be aborted as soon as it finds that the write lock it wishes to obtain is held by T_j.)

18.8.3 Serializability Issues and Solutions

Snapshot isolation is attractive in practice because transactions that read a lot of data (typically for data analysis) do not interfere with shorter update transactions (typically used for transaction processing). With two-phase locking, such long read-only transactions would block update transactions for long periods of time, which is often unacceptable.

It is worth noting that integrity constraints that are enforced by the database, such as primary-key and foreign-key constraints, cannot be checked on a snapshot; otherwise it would be possible for two concurrent transactions to insert two tuples with the same primary key value, or for a transaction to insert a foreign key value that is concurrently deleted from the referenced table. This problem is handled by checking these constraints on the current state of the database, rather than on the snapshot, as part of validation at the time of commit.

Even with the above fix, there is still a serious problem with the snapshot isolation scheme as we have presented it and as it is implemented in practice: *snapshot isolation does* **not** *ensure serializability!*

Next we give examples of possible nonserializable executions under snapshot isolation. We then outline the *serializable snapshot isolation* technique that is supported by some databases, which extends the snapshot isolation technique to ensure serializability. Snapshot isolation implementations that do not support serializable snapshot isolation often support SQL extensions that allow the programmer to ensure serializability even with snapshot isolation; we study these extensions at the end of the section.

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Figure 18.20 Nonserializable schedule under snapshot isolation.

- Consider the transaction schedule shown in Figure 18.20. Two concurrent transactions T_i and T_j both read data items A and B. T_i sets $A = B$ and writes A, while T_j sets $B = A$ and writes B. Since T_i and T_j are concurrent, under snapshot isolation neither transaction sees the update by the other in its snapshot. But, since they update different data items, both are allowed to commit regardless of whether the system uses the first-update-wins policy or the first-committer-wins policy.

  However, the execution is not serializable, since it results in swapping of the values of A and B, whereas any serializable schedule would set both A and B to the same value: either the initial value of A or the initial value of B, depending on the order of T_i and T_j.

  It can be easily seen that the precedence graph has a cycle. There is an edge in the precedence graph from T_i to T_j because T_i reads the value of A that existed before T_j writes A. There is also an edge in the precedence graph from T_j to T_i because T_j reads the value of B that existed before T_i writes B. Since there is a cycle in the precedence graph, the result is a nonserializable schedule.

  This situation, where each of a pair of transactions has read a data item that is written by the other, but the set of data items written by the two transactions do not have any data item in common, is referred to as **write skew**.

- As another example of write skew, consider a banking scenario. Suppose that the bank enforces the integrity constraint that the sum of the balances in the checking and the savings account of a customer must not be negative. Suppose the checking and savings balances for a customer are \$100 and \$200, respectively. Suppose that transaction T_{36} withdraws \$200 from the checking account, after verifying the integrity constraint by reading both balances. Suppose that concurrently transaction T_{37} withdraws \$200 from the savings account, again after verifying the integrity constraint. Since each of the transactions checks the integrity constraint on its own snapshot, if they run concurrently each will believe that the sum of the balances after the withdrawal is \$100, and therefore its withdrawal does not violate

the constraint. Since the two transactions update different data items, they do not have any update conflict, and under snapshot isolation both of them can commit.

Unfortunately, in the final state after both T_{36} and T_{37} have committed, the sum of the balances is \$100, violating the integrity constraint. Such a violation could never have occurred in any serial execution of T_{36} and T_{37}.

- Many financial applications create consecutive sequence numbers, for example to number bills, by taking the maximum current bill number and adding 1 to the value to get a new bill number. If two such transactions run concurrently, each would see the same set of bills in its snapshot, and each would create a new bill with the same number. Both transactions pass the validation tests for snapshot isolation, since they do not update any tuple in common. However, the execution is not serializable; the resultant database state cannot be obtained by any serial execution of the two transactions. Creating two bills with the same number could have serious legal implications.

  The above problem is in fact an example of the phantom phenomenon, which we saw in Section 18.4.3, since the insert performed by each transaction conflicts with the read performed by the other transaction to find the maximum bill number, but the conflict is not detected by snapshot isolation.[5]

The problems listed above seem to indicate that the snapshot isolation technique is vulnerable to many serializability problems and should never be used. However, serializability problems are relatively rare for two reasons:

1. The fact that the database must check integrity constraints at the time of commit, and not on a snapshot, helps avoid inconsistencies in many situations. For example, in the financial application example that we saw earlier, the bill number would likely have been declared as a primary key. The database system would detect the primary key violation outside the snapshot and roll back one of the two transactions.

   It was shown that primary key constraints ensured that all transactions in a popular transaction processing benchmark, TPC-C, were free from nonserializability problems, when executed under snapshot isolation. This was viewed as an indication that such problems are rare. However, they do occur occasionally, and when they occur they must be dealt with.[6]

2. In many applications that are vulnerable to serializability problems, such as skew writes, on some data items, the transactions conflict on other data items, ensuring

[5] The SQL standard uses the term *phantom problem* to refer to nonrepeatable predicate reads, leading some to claim that snapshot isolation avoids the phantom problem; however, such a claim is not valid under our definition of phantom conflict.

[6] For example, the problem of duplicate bill numbers actually occurred several times in a financial application in I.I.T. Bombay, where (for reasons too complex to discuss here) the bill number was not a primary key, and it was detected by financial auditors.

such transactions cannot execute concurrently; as a result, the execution of such transactions under snapshot isolation remains serializable.

Nonserializable may nevertheless occur with snapshot isolation. The impact of nonserializable execution due to snapshot isolation is not very severe for many applications. For example, consider a university application that implements enrollment limits for a course by counting the current enrollment before allowing registration. Snapshot isolation could allow the class enrollment limit to be exceeded. However, this may happen very rarely, and if it does, having one extra student in a class is usually not a major problem. The fact that snapshot isolation allows long read transactions to execute without blocking updaters is a large enough benefit for many such applications to live with occasional glitches.

Nonserializability may not be acceptable for many other applications, such as financial applications. There are several possible solutions.

- A modified form of snapshot isolation, called serializable snapshot isolation, can be used if it is supported by the database system. This technique extends the snapshot isolation technique in a way that ensures serializability.
- Some systems allow different transactions to run under different isolation levels, which can be used to avoid the serializability problems mentioned above.
- Some systems that support snapshot isolation provide a way for SQL programmers to create artificial conflicts, using a **for update** clause in SQL, which can be used to ensure serializability.

We briefly outline each of these solutions below.

Since version 9.1, PostgreSQL implements a technique called serializable snapshot isolation, which ensures serializability; in addition, PostgreSQL versions from 9.1 onwards include an index-locking-based technique to provide protection against phantom problems.

The intuition behind the **serializable snapshot isolation** (**SSI**) protocol is as follows: Suppose we track all conflicts (i.e., write-write, read-write, and write-read conflicts) between transactions. Recall from Section 17.6 that we can construct a transaction precedence graph which has a directed edge from T_1 to T_2 if transactions T_1 and T_2 have conflicting operations on a tuple, with T_1's action preceding T_2's action. As we saw in Section 17.6, one way to ensure serializability is to look for cycles in the transaction precedence graph and roll back transactions if a cycle is found.

The key reason for loss of serializability with snapshot isolation is that read-write conflicts, where a transaction T_1 writes a version of an object, and a transaction T_2 subsequently reads an earlier version of the object, are not tracked by snapshot isolation. This conflict can be represented by a read-write conflict edge from T_2 to T_1.

It has been shown that in all cases where snapshot isolation allows nonserializable schedules, there must be a transaction that has both an incoming read-write conflict

edge and an outgoing read-write conflict edge (all other cases of cycles in the conflict graph are caught by the snapshot isolation rules). Thus, serializable snapshot isolation implementations track all read-write conflicts between concurrent transactions to detect if a transaction has both an incoming and an outgoing read-write conflict edge. If such a situation is detected, one of the transactions involved in the read-write conflicts is rolled back. This check is significantly cheaper than tracking all conflicts and looking for cycles, although it may result in some unnecessary rollbacks.

It is also worth mentioning that the technique used by PostgreSQL to prevent phantoms uses index locking, but the locks are not held in a two-phase manner. Instead, they are used to detect potential conflicts between concurrent transactions and must be retained for some time even after a transaction commits, to allow checks against other concurrent transactions. The index-locking technique used by PostgreSQL also does not result in any deadlocks.

SQL Server offers the option of allowing some transactions to run under *snapshot* isolation, while allowing others to run under the *serializable* isolation level. Running long read-only transactions under the snapshot isolation level while running update transactions under the serializable isolation level ensures that the read-only transaction does not block updaters, while also ensuring that the above anomalies cannot occur.

In Oracle versions till at least Oracle 12c (to the best of our knowledge), and in PostgreSQL versions prior to 9.1, the *serializable* isolation level actually implements snapshot isolation. As a result, even with the isolation level set to serializable, it is possible that the database permits some schedules that are not serializable.

If an application has to run under snapshot isolation, on several of these databases an application developer can guard against certain snapshot anomalies by appending a **for update** clause to the SQL select query as illustrated below:

```
select *
from instructor
where ID = 22222
for update;
```

Adding the **for update** clause causes the system to treat data that are read as if they had been updated for purposes of concurrency control. In our first example of write skew shown in Figure 18.20, if the **for update** clause were appended to the select queries that read the values of A and B, only one of the two concurrent transactions would be allowed to commit since it appears that both transactions have updated both A and B.

Formal methods exist (see the online bibliographical notes) to determine whether a given mix of transactions runs the risk of nonserializable execution under snapshot isolation and to decide on what conflicts to introduce (using the **for update** clause, for example) to ensure serializability. Such methods can work only if we know in advance what transactions are being executed. In some applications, all transactions are from a predetermined set of transactions, making this analysis possible. However, if the application allows unrestricted, ad hoc transactions, then no such analysis is possible.

18.9 Weak Levels of Consistency in Practice

In Section 17.8, we discussed the isolation levels specified by the SQL standard: serializable, repeatable read, read committed, and read uncommitted. In this section, we first briefly outline some older terminology relating to consistency levels weaker than serializability and relate it to the SQL standard levels. We then discuss the issue of concurrency control for transactions that involve user interaction, an issue that we briefly discussed in Section 17.8.

18.9.1 Degree-Two Consistency

The purpose of **degree-two consistency** is to avoid cascading aborts without necessarily ensuring serializability. The locking protocol for degree-two consistency uses the same two lock modes that we used for the two-phase locking protocol: shared (S) and exclusive (X). A transaction must hold the appropriate lock mode when it accesses a data item, but two-phase behavior is not required.

In contrast to the situation in two-phase locking, S-locks may be released at any time, and locks may be acquired at any time. Exclusive locks, however, cannot be released until the transaction either commits or aborts. Serializability is not ensured by this protocol. Indeed, a transaction may read the same data item twice and obtain different results. In Figure 18.21, T_{32} reads the value of Q before that value is written by T_{33}, and again after it is written by T_{33}.

Reads are not repeatable, but since exclusive locks are held until transaction commit, no transaction can read an uncommitted value. Thus, degree-two consistency is one particular implementation of the read-committed isolation level.

It is interesting to note that with degree-two consistency, a transaction that is scanning an index may potentially see two versions of a record that was updated while the scan was in progress and may also potentially see neither version! For example,

T_{32}	T_{33}
lock-S(Q)	
read(Q)	
unlock(Q)	
	lock-X(Q)
	read(Q)
	write(Q)
	unlock(Q)
lock-S(Q)	
read(Q)	
unlock(Q)	

Figure 18.21 Nonserializable schedule with degree-two consistency.

consider a relation $r(A, B, C)$, with primary key A, with an index on attribute B. Now consider a query that is scanning the relation r using the index on attribute B, using degree-two consistency. Suppose there is a concurrent update to a tuple $t_1 \in r$ that updates attribute $t_1.B$ from v_1 to v_2. Such an update requires deletion of an entry corresponding to value v_1 from the index and insertion of a new entry corresponding to v_2. Now, the scan of r could possibly scan the index node corresponding to v_1 after the old tuple is deleted there but visit the index node corresponding to v_2 before the updated tuple is inserted in that node. Then, the scan would completely miss the tuple, even though it should have seen either the old value or the new value of t_1. Further, a scan using degree-two consistency could possibly visit the node corresponding to v_1 before the delete, and the node corresponding to v_2 after the insert, and thereby see two versions of t_1, one from before the update and one from after the update. (This problem would not arise if the scan and the update both used two-phase locking.)

18.9.2 Cursor Stability

Cursor stability is a form of degree-two consistency designed for programs that iterate over tuples of a relation by using cursors. Instead of locking the entire relation, cursor stability ensures that:

- The tuple that is currently being processed by the iteration is locked in shared mode. Once the tuple is processed, the lock on the tuple can be released.
- Any modified tuples are locked in exclusive mode until the transaction commits.

These rules ensure that degree-two consistency is obtained. But locking is not done in a two-phase manner, and serializability is not guaranteed. Cursor stability is used in practice on heavily accessed relations as a means of increasing concurrency and improving system performance. Applications that use cursor stability must be coded in a way that ensures database consistency despite the possibility of nonserializable schedules. Thus, the use of cursor stability is limited to specialized situations with simple consistency constraints.

When supported by the database, snapshot isolation is a better alternative to degree-two consistency as well as cursor stability, since it offers a similar or even better level of concurrency while reducing the risk of nonserializable executions.

18.9.3 Concurrency Control Across User Interactions

Concurrency-control protocols usually consider transactions that do not involve user interaction. Consider the airline seat selection example from Section 17.8, which involved user interaction. Suppose we treat all the steps from when the seat availability is initially shown to the user, until the seat selection is confirmed, as a single transaction.

If two-phase locking is used, the entire set of seats on a flight would be locked in shared mode until the user has completed the seat selection, and no other transaction would be able to update the seat allocation information in this period. Such locking

would be a very bad idea since a user may take a long time to make a selection, or even just abandon the transaction without explicitly cancelling it. Timestamp protocols or validation could be used instead, which avoid the problem of locking, but both these protocols would abort the transaction for a user *A* if any other user *B* has updated the seat allocation information, even if the seat selected by *B* does not conflict with the seat selected by user *A*. Snapshot isolation is a good option in this situation, since it would not abort the transaction of user *A* as long as *B* did not select the same seat as *A*.

However, snapshot isolation requires the database to remember information about updates performed by a transaction even after it has committed, as long as any other concurrent transaction is still active, which can be problematic for long-duration transactions.

Another option is to split a transaction that involves user interaction into two or more transactions, such that no transaction spans a user interaction. If our seat selection transaction is split thus, the first transaction would read the seat availability, while the second transaction would complete the allocation of the selected seat. If the second transaction is written carelessly, it could assign the selected seat to the user, without checking if the seat was meanwhile assigned to some other user, resulting in a lost-update problem. To avoid the problem, as we outlined in Section 17.8, the second transaction should perform the seat allocation only if the seat was not meanwhile assigned to some other user.

The above idea has been generalized in an alternative concurrency control scheme, which uses version numbers stored in tuples to avoid lost updates. The schema of each relation is altered by adding an extra *version_number* attribute, which is initialized to 0 when the tuple is created. When a transaction reads (for the first time) a tuple that it intends to update, it remembers the version number of that tuple. The read is performed as a stand-alone transaction on the database, and hence any locks that may be obtained are released immediately. Updates are done locally and copied to the database as part of commit processing, using the following steps which are executed atomically (i.e., as part of a single database transaction):

- For each updated tuple, the transaction checks if the current version number is the same as the version number of the tuple when it was first read by the transaction.
    1. If the version numbers match, the update is performed on the tuple in the database, and its version number is incremented by 1.
    2. If the version numbers do not match, the transaction is aborted, rolling back all the updates it performed.
- If the version number check succeeds for all updated tuples, the transaction commits. It is worth noting that a timestamp could be used instead of the version number without impacting the scheme in any way.

Observe the close similarity between the preceding scheme and snapshot isolation. The version number check implements the first-committer-wins rule used in snapshot isolation, and it can be used even if the transaction was active for a very long time. However, unlike snapshot isolation, the reads performed by a transaction may not correspond to a snapshot of the database; and unlike the validation-based protocol, reads performed by the transaction are not validated.

We refer to the above scheme as **optimistic concurrency control without read validation**. Optimistic concurrency control without read validation provides a weak level of serializability, and it does not ensure serializability. A variant of this scheme uses version numbers to validate reads at the time of commit, in addition to validating writes, to ensure that the tuples read by the transaction were not updated subsequent to the initial read; this scheme is equivalent to the optimistic concurrency-control scheme which we saw earlier.

This scheme has been widely used by application developers to handle transactions that involve user interaction. An attractive feature of the scheme is that it can be implemented easily on top of a database system. The validation and update steps performed as part of commit processing are then executed as a single transaction in the database, using the concurrency-control scheme of the database to ensure atomicity for commit processing. The scheme is also used by the Hibernate object-relational mapping system (Section 9.6.2), and other object-relational mapping systems, where it is referred to as optimistic concurrency control (even though reads are not validated by default). Hibernate and other object-relational mapping systems therefore perform the version number checks transparently as part of commit processing. (Transactions that involve user interaction are called **conversations** in Hibernate to differentiate them from regular transactions; validation using version numbers is particularly useful for such transactions.)

Application developers must, however, be aware of the potential for non-serializable execution, and they must restrict their usage of the scheme to applications where non-serializability does not cause serious problems.

18.10 Advanced Topics in Concurrency Control

Instead of using two-phase locking, special-purpose concurrency control techniques can be used for index structures, resulting in improved concurrency. When using main-memory databases, conversely, index concurrency control can be simplified. Further, concurrency control actions often become bottlenecks in main-memory databases, and techniques such as latch-free data structures have been designed to reduce concurrency control overheads. Instead of detecting conflicts at the level of reads and writes, it is possible to consider operations, such as increment of a counter, as basic operations, and perform concurrency control on the basis of conflicts between operations. Certain applications require guarantees on transaction completion time. Specialized concurrency control techniques have been developed for such applications.

18.10.1 Online Index Creation

When we are dealing with large volumes of data (ranging in the terabytes), operations such as creating an index can take a long time—perhaps hours or even days. When the operation finishes, the index contents must be consistent with the contents of the relation, and all further updates to the relation must maintain the index.

One way of ensuring that the data and the index are consistent is to block all updates to the relation while the index is created, for example by getting a shared lock on the relation. After the index is created, and the relation metadata are updated to reflect the existence of the index locks can be released. Subsequent update transactions will find the index, and carry out index maintenance as part of the transaction.

However, the above approach would make the system unavailable for updates to the relation for a very long time, which is unacceptable. Instead, most database systems support **online index creation**, which allows relation updates to occur even as the index is being created. Online index creation can be carried out as follows:

1. Index creation gets a snapshot of the relation and uses it to create the index; meanwhile, the system logs all updates to the relation that happen after the snapshot is created.
2. When the index on the snapshot data is complete, it is not yet ready for use, since subsequent updates are missing. At this point, the log of updates to the relation is used to update the index. But while the index update is being carried out, further updates may be happening on the relation.
3. The index update then obtains a shared lock on the relation to prevent further updates and applies all remaining updates to the index. At this point, the index is consistent with the contents of the relation. The relation metadata are then updated to indicate the existence of the new index. Subsequently all locks are released.
   Any transaction that executes after this will see the existence of the index; if the transaction updates the relation, it will also update the index.

Creation of materialized views that are maintained immediately, as part of the transaction that updates any of the relations used in the view, can also benefit from online construction techniques that are similar to online index construction. The query defining the view is executed on a snapshot of the participating relations, and subsequent updates are logged. The updates are applied to the materialized view, with a final phase of locking and catching up similar to the case of online index creation.

Schema changes such as adding or deleting attributes or constraints can also have a significant impact if relations are locked while the schema change is implemented on all tuples.

- For adding or deleting attributes, a version number can be kept with each tuple, and tuples can be updated in the background, or whenever they are accessed; the version number is used to determine if the schema change has already been applied

to the tuple, and the schema change is applied to the tuple if it has not already been applied.

- Adding of constraints requires that existing data must be checked to ensure that the constraint is satisfied. For example, adding a primary or unique key constraint on an attribute ID requires checking of existing tuples to ensure that no two tuples have the same ID value. Online addition of such constraints is done in a manner similar to online index construction, by checking the constraints on a relation snapshot, while keeping a log of updates that occur after the snapshot. The updates in the log must then be checked to ensure that they do not violate the constraint. In a final catch-up phase, the constraint is checked on any remaining updates in the log and added to the relation metadata while holding a shared lock on the relation.

18.10.2 Concurrency in Index Structures

It is possible to treat access to index structures like any other database structure and to apply the concurrency-control techniques discussed earlier. However, since indices are accessed frequently, they would become a point of great lock contention, leading to a low degree of concurrency. Luckily, indices do not have to be treated like other database structures; it is desirable to release index locks early, in a non-two-phase manner, to maximize concurrency. In fact, it is perfectly acceptable for a transaction to perform a lookup on an index twice and to find that the structure of the index has changed in between, as long as the index lookup returns the correct set of tuples. Informally, it is acceptable to have nonserializable concurrent access to an index, as long as the accuracy of the index is maintained; we formalize this notion next.

Operation serializability for index operations is defined as follows: A concurrent execution of index operations on an index is said to be serializable if there is a serialization order of the operations that is consistent with the results that each index operation in the concurrent execution sees, as well as with the final state of the index after all the operations have been executed. Index concurrency control techniques must ensure that any concurrent execution of index operations is serializable.

We outline two techniques for managing concurrent access to B^+-trees as well as an index-concurrency control technique to prevent the phantom phenomenon. The online bibliographical notes reference other techniques for B^+-trees as well as techniques for other index structures. The techniques that we present for concurrency control on B^+-trees are based on locking, but neither two-phase locking nor the tree protocol is employed. The algorithms for lookup, insertion, and deletion are those used in Chapter 14, with only minor modifications.

The first technique is called the **crabbing protocol**:

- When searching for a key value, the crabbing protocol first locks the root node in shared mode. When traversing down the tree, it acquires a shared lock on the child node to be traversed further. After acquiring the lock on the child node, it releases the lock on the parent node. It repeats this process until it reaches a leaf node.

- When inserting or deleting a key value, the crabbing protocol takes these actions:
  - It follows the same protocol as for searching until it reaches the desired leaf node. Up to this point, it obtains (and releases) only shared locks.
  - It locks the leaf node in exclusive mode and inserts or deletes the key value.
  - If it needs to split a node or coalesce it with its siblings, or redistribute key values between siblings, the crabbing protocol locks the parent of the node in exclusive mode. After performing these actions, it releases the locks on the node and siblings.

    If the parent requires splitting, coalescing, or redistribution of key values, the protocol retains the lock on the parent, and splitting, coalescing, or redistribution propagates further in the same manner. Otherwise, it releases the lock on the parent.

The protocol gets its name from the way in which crabs advance by moving sideways, moving the legs on one side, then the legs on the other, and so on alternately. The progress of locking while the protocol both goes down the tree and goes back up (in case of splits, coalescing, or redistribution) proceeds in a similar crab-like manner.

Once a particular operation releases a lock on a node, other operations can access that node. There is a possibility of deadlocks between search operations coming down the tree, and splits, coalescing, or redistribution propagating up the tree. The system can easily handle such deadlocks by restarting the search operation from the root, after releasing the locks held by the operation.

Locks that are held for a short duration, instead of being held in a two-phase manner, are often referred to as **latches**. Latches are used internally in databases to achieve mutual exclusion on shared data structures. In the above case, locks are held in a way that does not ensure mutual exclusion during an insert or delete operation, yet the resultant execution of index operations is serializable.

The second technique achieves even more concurrency, avoiding even holding the lock on one node while acquiring the lock on another node; thereby, deadlocks are avoided, and concurrency is increased. This technique uses a modified version of B^+-trees called **B-link trees**; B-link trees require that every node (including internal nodes, not just the leaves) maintain a pointer to its right sibling. This pointer is required because a lookup that occurs while a node is being split may have to search not only that node but also that node's right.

Unlike the crabbing protocol, the **B-link-tree locking protocol** holds locks on only one internal node at a time. The protocol releases the lock on the current internal node before requesting a lock on a child node (when traversing downwards), or on a parent node (while traversing upwards during a split or merge). Doing so can result in anomalies: for example, between the time the lock on a node is released and the lock on a parent is requested, a concurrent insert or delete on a sibling may cause a split or merge on the parent, and the original parent node may no longer be a parent of the

child node when it is locked. The protocol detects and handles such situations, ensuring operation serializability while avoiding deadlocks between operations and increasing concurrency compared to the crabbing protocol.

The phantom phenomenon, where conflicts between a predicate read and an insert or update are not detected, can allow nonserializable executions to occur. The index-locking technique, which we saw in Section 18.4.3, prevents the phantom phenomenon by locking index leaf nodes in a two-phase manner. Instead of locking an entire index leaf node, some index concurrency-control schemes use **key-value locking** on individual key values, allowing other key values to be inserted or deleted from the same leaf. Key-value locking thus provides increased concurrency.

Using key-value locking naïvely, however, would allow the phantom phenomenon to occur; to prevent the phantom phenomenon, the **next-key locking** technique is used. In this technique, every index lookup must lock not only the keys found within the range (or the single key, in case of a point lookup) but also the next-key value—that is, the key value just greater than the last key value that was within the range. Also, every insert must lock not only the value that is inserted, but also the next-key value. Thus, if a transaction attempts to insert a value that was within the range of the index lookup of another transaction, the two transactions would conflict on the key value next to the inserted key value. Similarly, deletes must also lock the next-key value to the value being deleted to ensure that conflicts with subsequent range lookups of other queries are detected.

18.10.3 Concurrency Control in Main-Memory Databases

With data stored on hard disk, the cost of I/O operations often dominates the cost of transaction processing. When disk I/O is the bottleneck cost in a system, there is little benefit from optimizing other smaller costs, such as the cost of concurrency control. However, in a main-memory database, with disk I/O no longer the bottleneck, systems benefit from reducing other costs, such as query processing costs, as we saw in Section 15.8; we now consider how to reduce the cost of concurrency control in main-memory databases.

As we saw in Section 18.10.2, concurrency-control techniques for operations on disk-based index structures acquire locks on individual nodes, to increase the potential for concurrent access to the index. However, such locking comes at the increased cost of acquiring the locks. In a main-memory database, where data are in memory, index operations take very little time for execution. Thus, it may be acceptable to perform locking at a coarse granularity: for example, the entire index could be locked using a single latch (i.e., short duration lock), the operation performed, and the latch released. The reduced overhead of locking has been found to make up for the slightly reduced concurrency, and to improve overall performance.

There is another way to improve performance with in-memory indices, using atomic instructions to carry out index updates without acquiring any latches at all.

```
insert(value, head) {
        node = new node
        node–>value = value
        node–>next = head
        head = node
}
```

Figure 18.22 Insertion code that is unsafe with concurrent inserts.

Data structures implementations that support concurrent operations without requiring latches are called **latch-free data structure** implementations.

Consider a linked list, where each node has a value *value* and a *next* pointer, and the head of the linked list is stored in the variable *head*. The function *insert*() shown in Figure 18.22 would work correctly to insert a node at the head of the list, if there are no concurrent invocations of the code for the same list.[7]

However, if two processes execute the *insert*() function concurrently on the same list, it is possible that both of them would read the same value of variable *head*, and then both would update the variable after that. The final result would contain one of the two nodes being inserted, while the other node being inserted would be lost.

One way of preventing such a problem is to get an exclusive latch (short term lock) on the linked list, perform the *insert*() function, and then release the latch. The *insert*() function can be modified to acquire and release a latch on the list.

An alternative implementation, which is faster in practice, is to use an atomic *compare-and-swap*() instruction, abbreviated to CAS, which works as follows: The instruction CAS(*var*, *oldval*, *newval*) takes three arguments: a variable *var* and two values, *oldval* and *newval*. The instruction does the following atomically: check if the value of *var* is equal to *oldval*, and if so, set *var* to *newval*, and return success. If the value is not equal, it returns failure. The instruction is supported by most modern processor architectures, and it executes very quickly.

The function *insert_latchfree*(), shown in Figure 18.23 is a modification of *insert*() that works correctly even with concurrent inserts on the same list, without obtaining any latches. With this code, if two processes concurrently read the old value of *head*, and then both execute the CAS instruction, one of them will find the CAS instruction returning success, while the other one will find it returning failure since the value of *head* changes between the time it is read and when the CAS instruction is executed. The repeat loop then retries the insert using the new value of *head*, until it succeeds.

Function *delete_latchfree*(), shown in Figure 18.23, similarly implements deletion from the head of the list using the compare and swap instruction, without requiring latches. (In this case, the list is used as a stack, since deletion occurs at the head of

[7] We assume all parameters are passed by reference.

```
insert_latchfree(head, value) {
    node = new node
    node->value = value
    repeat
        oldhead = head
        node->next = oldhead
        result = CAS(head, oldhead, node)
    until (result == success)
}

delete_latchfree(head) {
    /* This function is not quite safe; see explanation in text. */
    repeat
        oldhead = head
        newhead = oldhead->next
        result = CAS(head, oldhead, newhead)
    until (result == success)
}
```

Figure 18.23 Latch-free insertion and deletion on a list.

the list.) However, it has a problem: it does not work correctly in some rare cases. The problem can occur when a process $P1$ is performing a delete, with node $n1$ at the head of the list, and concurrently a second process $P2$ deletes the first two elements, $n1$ and $n2$, and then reinserts $n1$ at the head of the list, with some other element, say $n3$ as the next element. If $P1$ read $n1$ before $P2$ deleted it, but performs the CAS after $P2$ has reinserted $n1$, the CAS operation of $P1$ will succeed, but set the head of the list to point to $n2$, which has been deleted, leaving the list in an inconsistent state. This problem is known as the *ABA problem*.

One solution is to keep a counter along with each pointer, which is incremented every time the pointer is updated. The CAS instruction is applied on the (pointer, counter) pair; most CAS implementations on 64 bit processors support such a double compare-and-swap on 128 bits. The ABA problem can then be avoided since although the reinsert of $n1$ would result in the head pointing to $n1$, the counter would be different, resulting in the CAS operation of $P1$ failing. See the online solutions to Practice Exercise 18.16 for more details of the ABA problem and the above solution. With such a modification, both inserts and deletes can be executed concurrently without acquiring latches. There are other solutions that do not require a double compare-and-swap, but are more complicated.

Deletion from the tail of the list (to implement a queue) as well as more complex data structures such as hash indices and search trees can also be implemented in a latch-

free manner. It is best to use latch-free data structure implementations (more often referred to as **lock-free data structure** implementations) that are provided by standard libraries, such as the Boost library for C++, or the ConcurrentLinkedQueue class in Java; do not build your own, since you may introduce bugs due to "*race conditions*" between concurrent accesses, that can be very hard to detect or debug.

Since today's multiprocessor CPUs have a large number of cores, latch-free implementations have been found to significantly outperform implementations that obtain latches, in the context of in-memory indices and other in-memory data structures

18.10.4 Long-Duration Transactions

The transaction concept developed initially in the context of data-processing applications, in which most transactions are noninteractive and of short duration. Serious problems arise when this concept is applied to database systems that involve human interaction. Such transactions have these key properties:

- **Long duration**. Once a human interacts with an active transaction, that transaction becomes a **long-duration transaction** from the perspective of the computer, since human response time is slow relative to computer speed. Furthermore, in design applications, the human activity may involve hours, days, or an even longer period. Thus, transactions may be of long duration in human terms, as well as in machine terms.
- **Exposure of uncommitted data**. Data generated and displayed to a user by a long-duration transaction are uncommitted, since the transaction may abort. Thus, users—and, as a result, other transactions—may be forced to read uncommitted data. If several users are cooperating on a project, user transactions may need to exchange data prior to transaction commit.
- **Subtasks**. An interactive transaction may consist of a set of subtasks initiated by the user. The user may wish to abort a subtask without necessarily causing the entire transaction to abort.
- **Recoverability**. It is unacceptable to abort a long-duration interactive transaction because of a system crash. The active transaction must be recovered to a state that existed shortly before the crash so that relatively little human work is lost.
- **Performance**. Good performance in an interactive transaction system is defined as fast response time. This definition is in contrast to that in a noninteractive system, in which high throughput (number of transactions per second) is the goal. Systems with high throughput make efficient use of system resources. However, in the case of interactive transactions, the most costly resource is the user. If the efficiency and satisfaction of the user are to be optimized, response time should be fast (from a human perspective). In those cases where a task takes a long time, response time

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figure 18.24 A non-conflict-serializable schedule.

should be predictable (i.e., the variance in response times should be low) so that users can manage their time well.

Snapshot isolation, described in Section 18.8, can provide a partial solution to these issues, as can the *optimistic concurrency control without read validation* protocol described in Section 18.9.3. The latter protocol was in fact designed specifically to deal with long-duration transactions that involve user interaction. Although it does not guarantee serializability, optimistic concurrency control without read validation is quite widely used.

However, when transactions are of long duration, conflicting updates are more likely, resulting in additional waits or aborts. These considerations are the basis for the alternative concepts of correctness of concurrent executions and transaction recovery that we consider in the remainder of this section.

18.10.5 Concurrency Control with Operations

Consider a bank database consisting of two accounts A and B, with the consistency requirement that the sum $A + B$ be preserved. Consider the schedule of Figure 18.24. Although the schedule is not conflict serializable, it nevertheless preserves the sum of $A + B$. It also illustrates two important points about the concept of correctness without serializability.

1. Correctness depends on the specific consistency constraints for the database.
2. Correctness depends on the properties of operations performed by each transaction.

While two-phase locking ensures serializability, it can result in poor concurrency in case a large number of transactions conflict on a particular data item. Timestamp and validation protocols also have similar problems in this case.

Concurrency can be increased by treating some operations besides **read** and **write** as fundamental low-level operations and to extend concurrency control to deal with them.

Consider the case of materialized view maintenance, which we saw in Section 16.5.1. Suppose there is a relation *sales*(*date, custID, itemID, amount*), and a materialized view *daily_sales_total*(*date, total_amount*), that records total sales on each day. Every sales transaction must update the materialized view as part of the transaction if immediate view maintenance is used. With a high volume of sales, and every transaction updating the same record in the *daily_sales_total* relation, the degree of concurrency will be quite low if two-phase locking is used on the materialized view.

A better way to perform concurrency control for the materialized view is as follows: Observe that each transaction increments a record in the *daily_sales_total* relation by some value but does not need to see the value. It would make sense to have an operation increment(v, n), that adds a value n to a variable v without making the value of v visible to the transaction; we shall see shortly how this is implemented. In our sales example, a transaction that inserts a *sales* tuple with amount n invokes the increment operation with the first argument being the *total_amount* value of the appropriate tuple in the materialized view *daily_sales_total*, and the second argument being the value n.

The increment operation does not lock the variable in a two-phase manner; however, individual operations should be executed serially on the variable. Thus, if two increment operations are initiated concurrently on the same variable, one must finish before the other is allowed to start. This can be ensured by acquiring an exclusive latch (lock) on the variable v before starting the operation and releasing the latch after the operation has finished its updates. Increment operations can also be implemented using compare-and-swap operations, without getting latches.

Two transactions that invoke the increment operation should be allowed to execute concurrently to avoid concurrency control bottlenecks. In fact, increment operations executed by two transactions do not conflict with each other, since the final result is the same regardless of the order in which the operations were executed. If one of the transactions rolls back, the increment(v, n) operation must be rolled back by executing an operation increment($v, -n$), which adds a negative of the original value; this operation is referred to as a **compensating operation**.

However, if a transaction T wishes to read the materialized view, it clearly conflicts with any concurrent transaction that has performed an increment operation; the value that T reads depends on whether the other transaction is serialized before or after T.

We can define a locking protocol to handle the preceding situation by defining an **increment lock**. The increment lock is compatible with itself but is not compatible with shared and exclusive locks. Figure 18.25 shows a lock-compatibility matrix for three lock modes: share mode, exclusive mode, and increment mode.

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

Figure 18.25 Lock-compatibility matrix with increment lock mode.

As another example of special-purpose concurrency control for operations, consider an insert operation on a B^+-tree index which releases locks early, as we saw in Section 18.10.2. In this case, there is no special lock mode, but holding locks on leaf nodes in a two-phase manner (or using next-key locking) as we saw in Section 18.10.2 ensures serializability. The insert operation may have modified several nodes of the B^+-tree index. Other transactions may have read and updated these nodes further while processing other operations. To roll back the insertion, we would have to delete the record inserted by T_i; deletion is the compensating action for insertion. The result is a correct, consistent B^+-tree, but not necessarily one with exactly the same structure as the one we had before T_i started.

While operation locking can be done in a way that ensures serializability, in some cases it may even be used in a way that does not guarantee serializability, but where violations may be acceptable. Consider the case of concert tickets, where every transaction needs to access and update the total ticket sales. We can have an operation increment_conditional(v, n) which increments v by n, provided the resultant value would be ≥ 0; the operation returns a status of success in case the resultant value is ≥ 0 and returns failure otherwise. Consider a transaction T_i executed to purchase tickets. To book three tickets, where variable *avail_tickets* indicates the number of available tickets, the transaction can execute increment_conditional(*avail_tickets*, -3). A return value of success indicates that there were enough tickets available, and decrements the available tickets, while failure indicates insufficient availability of tickets.

If the variable *avail_tickets* is locked in a two-phase manner, concurrency would be very poor, with customers being forced to wait for bookings while an earlier transaction commits, even when there are many tickets available. Concurrency can be greatly increased by executing the increment_conditional operation, without holding any locks on *avail_tickets* in a two-phase manner; instead, an exclusive lock is obtained on the variable, the operation is performed, and the lock is then released.

The transaction T_i also needs to carry out other steps, such as collecting the payment; if one of the subsequent steps, such as payment, fails, the increment operation must be rolled back by executing a compensating operation; if the original operation added $-n$ to *avail_tickets*, the compensating operation adds $+n$ to *avail_tickets*.

It may appear that two increment_conditional operations are compatible with each other, similar to the increment operation that we saw earlier. But that is not

the case. Consider two concurrent transactions to purchase a single ticket, and assume that there is only one ticket left. The order in which the operations are executed has an obvious impact on which one succeeds and which one fails. Nevertheless, many real-world applications allow operations that hold short-term locks while they execute and release them at the end of the operation to increase concurrency, even at the cost of loss of serializability in some situations.

18.10.6 Real-Time Transaction Systems

In certain applications, the constraints include **deadlines** by which a task must be completed. Examples of such applications include plant management, traffic control, and scheduling. When deadlines are included, correctness of an execution is no longer solely an issue of database consistency. Rather, we are concerned with how many deadlines are missed, and by how much time they are missed. Deadlines are characterized as follows:

- **Hard deadline**. Serious problems, such as system crash, may occur if a task is not completed by its deadline.
- **Firm deadline**. The task has zero value if it is completed after the deadline.
- **Soft deadlines**. The task has diminishing value if it is completed after the deadline, with the value approaching zero as the degree of lateness increases.

Systems with deadlines are called **real-time systems**.

Transaction management in real-time systems must take deadlines into account. If the concurrency-control protocol determines that a transaction T_i must wait, it may cause T_i to miss the deadline. In such cases, it may be preferable to pre-empt the transaction holding the lock, and to allow T_i to proceed. Pre-emption must be used with care, however, because the time lost by the pre-empted transaction (due to rollback and restart) may cause the pre-empted transaction to miss its deadline. Unfortunately, it is difficult to determine whether rollback or waiting is preferable in a given situation.

Due to the unpredictable nature of delays when reading data from disk, main-memory databases are often used if real-time constraints have to be met. However, even if data are resident in main memory, variances in execution time arise from lock waits, transaction aborts, and so on. Researchers have devoted considerable effort to concurrency control for real-time databases. They have extended locking protocols to provide higher priority for transactions with early deadlines. They have found that optimistic concurrency protocols perform well in real-time databases; that is, these protocols result in fewer missed deadlines than even the extended locking protocols. The online bibliographical notes provide references to research in the area of real-time databases.

18.11 Summary

- When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the in-

teraction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.

- To ensure serializability, we can use various concurrency-control schemes. All these schemes either delay an operation or abort the transaction that issued the operation. The most common ones are locking protocols, timestamp-ordering schemes, validation techniques, and multiversion schemes.
- A locking protocol is a set of rules that state when a transaction may lock and unlock each of the data items in the database.
- The two-phase locking protocol allows a transaction to lock a new data item only if that transaction has not yet unlocked any data item. The protocol ensures serializability, but not deadlock freedom. In the absence of information concerning the manner in which data items are accessed, the two-phase locking protocol is both necessary and sufficient for ensuring serializability.
- The strict two-phase locking protocol permits release of exclusive locks only at the end of transaction, in order to ensure recoverability and cascadelessness of the resulting schedules. The rigorous two-phase locking protocol releases all locks only at the end of the transaction.
- Various locking protocols do not guard against deadlocks. One way to prevent deadlock is to use an ordering of data items and to request locks in a sequence consistent with the ordering.
- Another way to prevent deadlock is to use preemption and transaction rollbacks. To control the preemption, we assign a unique timestamp to each transaction. The system uses these timestamps to decide whether a transaction should wait or roll back. The wound – wait scheme is a preemptive scheme.
- If deadlocks are not prevented, the system must deal with them by using a deadlock detection and recovery scheme. To do so, the system constructs a wait-for graph. A system is in a deadlock state if and only if the wait-for graph contains a cycle. When the deadlock detection algorithm determines that a deadlock exists, the system rolls back one or more transactions to break the deadlock.
- There are circumstances where it would be advantageous to group several data items and to treat them as one aggregate data item for purposes of working, resulting in multiple levels of granularity. We allow data items of various sizes, and we define a hierarchy of data items where the small items are nested within larger ones. Such a hierarchy can be represented graphically as a tree. In such multigranularity locking protocols, locks are acquired in root-to-leaf order; they are released in leaf-to-root order. Intention lock modes are used at higher levels to get better concurrency, without affecting serializability.

- A timestamp-ordering scheme ensures serializability by selecting an ordering in advance between every pair of transactions. A unique fixed timestamp is associated with each transaction in the system. The timestamps of the transactions determine the serializability order. Thus, if the timestamp of transaction T_i is smaller than the timestamp of transaction T_j, then the scheme ensures that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j. It does so by rolling back a transaction whenever such an order is violated.
- A validation scheme is an appropriate concurrency-control method in cases where a majority of transactions are read-only transactions, and thus the rate of conflicts among these transactions is low. A unique fixed timestamp is associated with each transaction in the system. The serializability order is determined by the timestamp of the transaction. A transaction in this scheme is never delayed. It must, however, pass a validation test to complete. If it does not pass the validation test, the system rolls it back to its initial state.
- A multiversion concurrency-control scheme is based on the creation of a new version of a data item for each transaction that writes that item. When a read operation is issued, the system selects one of the versions to be read. The concurrency-control scheme ensures that the version to be read is selected in a manner that ensures serializability by using timestamps. A read operation always succeeds.
    - In multiversion timestamp ordering, a write operation may result in the rollback of the transaction.
    - In multiversion two-phase locking, write operations may result in a lock wait or, possibly, in deadlock.
- Snapshot isolation is a multiversion concurrency-control protocol based on validation, which, unlike multiversion two-phase locking, does not require transactions to be declared as read-only or update. Snapshot isolation does not guarantee serializability but is nevertheless supported by many database systems. Serializable snapshot isolation is an extension of snapshot isolation which guarantees serializability.
- A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted. A transaction that inserts a new tuple into the database is given an exclusive lock on the tuple.
- Insertions can lead to the phantom phenomenon, in which an insertion logically conflicts with a query even though the two transactions may access no tuple in common. Such conflict cannot be detected if locking is done only on tuples accessed by the transactions. Locking is required on the data used to find the tuples in the relation. The index-locking technique solves this problem by requiring locks on certain index nodes. These locks ensure that all conflicting transactions conflict on a real data item, rather than on a phantom.

- Weak levels of consistency are used in some applications where consistency of query results is not critical, and using serializability would result in queries adversely affecting transaction processing. Degree-two consistency is one such weaker level of consistency; cursor stability is a special case of degree-two consistency and is widely used.
- Concurrency control is a challenging task for transactions that span user interactions. Applications often implement a scheme based on validation of writes using version numbers stored in tuples; this scheme provides a weak level of serializability and can be implemented at the application level without modifications to the database.
- Special concurrency-control techniques can be developed for special data structures. Often, special techniques are applied in B^+-trees to allow greater concurrency. These techniques allow nonserializable access to the B^+-tree, but they ensure that the B^+-tree structure is correct, and they ensure that accesses to the database itself are serializable. Latch-free data structures are used to implement high-performance indices and other data structures in main-memory databases.

Review Terms

- Concurrency control
- Lock types
  - Shared-mode (S) lock
  - Exclusive-mode (X) lock
- Lock
  - Compatibility
  - Request
  - Wait
  - Grant
- Deadlock
- Starvation
- Locking protocol
- Legal schedule
- Two-phase locking protocol
  - Growing phase
  - Shrinking phase
  - Lock point
  - Strict two-phase locking
  - Rigorous two-phase locking
- Lock conversion
  - Upgrade
  - Downgrade
- Graph-based protocols
  - Tree protocol
  - Commit dependency
- Deadlock handling
  - Prevention
  - Detection
  - Recovery
- Deadlock prevention
  - Ordered locking
  - Preemption of locks
  - Wait–die scheme

- Wound–wait scheme
- Timeout-based schemes

- Deadlock detection
  - Wait-for graph
- Deadlock recovery
  - Total rollback
  - Partial rollback
- Multiple granularity
  - Explicit locks
  - Implicit locks
  - Intention locks
- Intention lock modes
  - Intention-shared (IS)
  - Intention-exclusive (IX)
  - Shared and intention-exclusive (SIX)
- Multiple-granularity locking protocol
- Timestamp
  - System clock
  - Logical counter
  - W-timestamp(Q)
  - R-timestamp(Q)
- Timestamp-ordering protocol
  - Thomas' write rule
- Validation-based protocols
  - Read phase
  - Validation phase
  - Write phase
  - Validation test
- Multiversion timestamp ordering
- Multiversion two-phase locking
  - Read-only transactions
  - Update transactions
- Snapshot isolation
  - Lost update
  - First committer wins
  - First updater wins
  - Write skew
  - Select for update
- Insert and delete operations
- Phantom phenomenon
- Index-locking protocol
- Predicate locking
- Weak levels of consistency
  - Degree-two consistency
  - Cursor stability
- Optimistic concurrency control without read validation
- Conversations
- Concurrency in indices
  - Crabbing protocol
  - B-link trees
  - B-link-tree locking protocol
  - Next-key locking
- Latch-free data structures
- Compare-and-swap (CAS) instruction

Practice Exercises

18.1 Show that the two-phase locking protocol ensures conflict serializability and that transactions can be serialized according to their lock points.

18.2 Consider the following two transactions:

T_{34}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{35}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Add lock and unlock instructions to transactions T_{31} and T_{32} so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?

18.3 What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?

18.4 Consider a database organized in the form of a rooted tree. Suppose that we insert a dummy vertex between each pair of vertices. Show that, if we follow the tree protocol on the new tree, we get better concurrency than if we follow the tree protocol on the original tree.

18.5 Show by example that there are schedules possible under the tree protocol that are not possible under the two-phase locking protocol, and vice versa.

18.6 Locking is not done explicitly in persistent programming languages. Rather, objects (or the corresponding pages) must be locked when the objects are accessed. Most modern operating systems allow the user to set access protections (no access, read, write) on pages, and memory access that violate the access protections result in a protection violation (see the Unix `mprotect` command, for example). Describe how the access-protection mechanism can be used for page-level locking in a persistent programming language.

18.7 Consider a database system that includes an atomic **increment** operation, in addition to the read and write operations. Let V be the value of data item X. The operation

increment(X) by C

sets the value of X to $V + C$ in an atomic step. The value of X is not available to the transaction unless the latter executes a read(X).
Assume that increment operations lock the item in increment mode using the compatibility matrix in Figure 18.25.

a. Show that, if all transactions lock the data that they access in the corresponding mode, then two-phase locking ensures serializability.

b. Show that the inclusion of **increment** mode locks allows for increased concurrency.

18.8 In timestamp ordering, **W-timestamp**(Q) denotes the largest timestamp of any transaction that executed write(Q) successfully. Suppose that, instead, we defined it to be the timestamp of the most recent transaction to execute write(Q) successfully. Would this change in wording make any difference? Explain your answer.

18.9 Use of multiple-granularity locking may require more or fewer locks than an equivalent system with a single lock granularity. Provide examples of both situations, and compare the relative amount of concurrency allowed.

18.10 For each of the following protocols, describe aspects of practical applications that would lead you to suggest using the protocol, and aspects that would suggest not using the protocol:

- Two-phase locking
- Two-phase locking with multiple-granularity locking.
- The tree protocol
- Timestamp ordering
- Validation
- Multiversion timestamp ordering
- Multiversion two-phase locking

18.11 Explain why the following technique for transaction execution may provide better performance than just using strict two-phase locking: First execute the transaction without acquiring any locks and without performing any writes to the database as in the validation-based techniques, but unlike the validation techniques do not perform either validation or writes on the database. Instead, rerun the transaction using strict two-phase locking. (Hint: Consider waits for disk I/O.)

18.12 Consider the timestamp-ordering protocol, and two transactions, one that writes two data items p and q, and another that reads the same two data items.

Give a schedule whereby the timestamp test for a write operation fails and causes the first transaction to be restarted, in turn causing a cascading abort of the other transaction. Show how this could result in starvation of both transactions. (Such a situation, where two or more processes carry out actions, but are unable to complete their task because of interaction with the other processes, is called a **livelock**.)

18.13 Devise a timestamp-based protocol that avoids the phantom phenomenon.

18.14 Suppose that we use the tree protocol of Section 18.1.5 to manage concurrent access to a B^+-tree. Since a split may occur on an insert that affects the root, it appears that an insert operation cannot release any locks until it has completed the entire operation. Under what circumstances is it possible to release a lock earlier?

18.15 The snapshot isolation protocol uses a validation step which, before performing a write of a data item by transaction T, checks if a transaction concurrent with T has already written the data item.

a. A straightforward implementation uses a start timestamp and a commit timestamp for each transaction, in addition to an *update set*, that, is the set of data items updated by the transaction. Explain how to perform validation for the first-committer-wins scheme by using the transaction timestamps along with the update sets. You may assume that validation and other commit processing steps are executed serially, that is, for one transaction at a time,

b. Explain how the validation step can be implemented as part of commit processing for the first-committer-wins scheme, using a modification of the above scheme, where instead of using update sets, each data item has a write timestamp associated with it. Again, you may assume that validation and other commit processing steps are executed serially.

c. The first-updater-wins scheme can be implemented using timestamps as described above, except that validation is done immediately after acquiring an exclusive lock, instead of being done at commit time.

   i. Explain how to assign write timestamps to data items to implement the first-updater-wins scheme.

   ii. Show that as a result of locking, if the validation is repeated at commit time the result would not change.

   iii. Explain why there is no need to perform validation and other commit processing steps serially in this case.

18.16 Consider functions *insert_latchfree*() and *delete_latchfree*(), shown in Figure 18.23.

a. Explain how the ABA problem can occur if a deleted node is reinserted.

b. Suppose that adjacent to *head* we store a counter *cnt*. Also suppose that DCAS((*head,cnt*), (*oldhead, oldcnt*), (*newhead, newcnt*)) atomically performs a compare-and-swap on the 128 bit value (*head,cnt*). Modify the *insert_latchfree*( ) and *delete_latchfree*( ) to use the DCAS operation to avoid the ABA problem.

c. Since most processors use only 48 bits of a 64 bit address to actually address memory, explain how the other 16 bits can be used to implement a counter, in case the DCAS operation is not supported.

Exercises

18.17 What benefit does strict two-phase locking provide? What disadvantages result?

18.18 Most implementations of database systems use strict two-phase locking. Suggest three reasons for the popularity of this protocol.

18.19 Consider a variant of the tree protocol called the *forest* protocol. The database is organized as a forest of rooted trees. Each transaction T_i must follow the following rules:

- The first lock in each tree may be on any data item.
- The second, and all subsequent, locks in a tree may be requested only if the parent of the requested node is currently locked.
- Data items may be unlocked at any time.
- A data item may not be relocked by T_i after it has been unlocked by T_i.

Show that the forest protocol does *not* ensure serializability.

18.20 Under what conditions is it less expensive to avoid deadlock than to allow deadlocks to occur and then to detect them?

18.21 If deadlock is avoided by deadlock-avoidance schemes, is starvation still possible? Explain your answer.

18.22 In multiple-granularity locking, what is the difference between implicit and explicit locking?

18.23 Although SIX mode is useful in multiple-granularity locking, an exclusive and intention-shared (XIS) mode is of no use. Why is it useless?

18.24 The multiple-granularity protocol rules specify that a transaction T_i can lock a node Q in S or IS mode only if T_i currently has the parent of Q locked in either

IX or IS mode. Given that SIX and S locks are stronger than IX or IS locks, why does the protocol not allow locking a node in S or IS mode if the parent is locked in either SIX or S mode?

18.25 Suppose the lock hierarchy for a database consists of database, relations, and tuples.

a. If a transaction needs to read a lot of tuples from a relation r, what locks should it acquire?

b. Now suppose the transaction wants to update a few of the tuples in r after reading a lot of tuples. What locks should it acquire?

c. If at run-time the transaction finds that it needs to actually update a very large number of tuples (after acquiring locks assuming only a few tuples would be updated). What problems would this cause to the lock table, and what could the database do to avoid the problem?

18.26 When a transaction is rolled-back under timestamp ordering, it is assigned a new timestamp. Why can it not simply keep its old timestamp?

18.27 Show that there are schedules that are possible under the two-phase locking protocol but not possible under the timestamp protocol, and vice versa.

18.28 Under a modified version of the timestamp protocol, we require that a commit bit be tested to see whether a `read` request must wait. Explain how the commit bit can prevent cascading abort. Why is this test not necessary for `write` requests?

18.29 As discussed in Exercise 18.15, snapshot isolation can be implemented using a form of timestamp validation. However, unlike the multiversion timestamp-ordering scheme, which guarantees serializability, snapshot isolation does not guarantee serializability. Explain the key difference between the protocols that results in this difference.

18.30 Outline the key similarities and differences between the timestamp-based implementation of the first-committer-wins version of snapshot isolation, described in Exercise 18.15, and the optimistic-concurrency control-without-read-validation scheme, described in Section 18.9.3.

18.31 Consider a relation $r(A, B, C)$ and a transaction T that does the following: find the maximum A value in r, and insert a new tuple in r whose A value is 1+ the maximum A value. Assume that an index is used to find the maximum A value.

a. Suppose that the transaction locks each tuple it reads in S mode, and the tuple it creates in X mode, and performs no other locking. Now suppose two instances of T are run concurrently. Explain how the resultant execution could be non-serializable.

b. Now suppose that $r.A$ is declared as a primary key. Can the above nonserializable execution occur in this case? Explain why or why not.

18.32 Explain the phantom phenomenon. Why may this phenomenon lead to an incorrect concurrent execution despite the use of the two-phase locking protocol?

18.33 Explain the reason for the use of degree-two consistency. What disadvantages does this approach have?

18.34 Give example schedules to show that with key-value locking, if lookup, insert, or delete does not lock the next-key value, the phantom phenomenon could go undetected.

18.35 Many transactions update a common item (e.g., the cash balance at a branch) and private items (e.g., individual account balances). Explain how you can increase concurrency (and throughput) by ordering the operations of the transaction.

18.36 Consider the following locking protocol: All items are numbered, and once an item is unlocked, only higher-numbered items may be locked. Locks may be released at any time. Only X-locks are used. Show by an example that this protocol does not guarantee serializability.

Further Reading

[Gray and Reuter (1993)] provides detailed textbook coverage of transaction-processing concepts, including concurrency-control concepts and implementation details. [Bernstein and Newcomer (2009)] provides textbook coverage of various aspects of transaction processing including concurrency control.

The two-phase locking protocol was introduced by [Eswaran et al. (1976)]. The locking protocol for multiple-granularity data items is from [Gray et al. (1975)]. The timestamp-based concurrency-control scheme is from [Reed (1983)]. The validation concurrency-control scheme is from [Kung and Robinson (1981)]. Multiversion timestamp order was introduced in [Reed (1983)]. A multiversion tree-locking algorithm appears in [Silberschatz (1982)].

Degree-two consistency was introduced in [Gray et al. (1975)]. The levels of consistency—or isolation—offered in SQL are explained and critiqued in [Berenson et al. (1995)]; the snapshot isolation technique was also introduced in the same paper. Serializable snapshot-isolation was introduced by [Cahill et al. (2009)]; [Ports and Grittner (2012)] describes the implementation of serializable snapshot isolation in PostgreSQL.

Concurrency in B^+-trees was studied by [Bayer and Schkolnick (1977)] and [Johnson and Shasha (1993)]. The crabbing and B-link tree techniques were introduced by [Kung and Lehman (1980)] and [Lehman and Yao (1981)]. The technique of key-value locking used in ARIES provides for very high concurrency on B^+-tree access and is de-

scribed in [Mohan (1990)] and [Mohan and Narang (1992)]. [Faerber et al. (2017)] provide a survey of main-memory databases, including coverage of concurrency control in main-memory databases. The ABA problem with latch-free data structures as well as solutions for the problem are discussed in [Dechev et al. (2010)].

Bibliography

[Bayer and Schkolnick (1977)] R. Bayer and M. Schkolnick, "Concurrency of Operating on B-trees", *Acta Informatica*, Volume 9, Number 1 (1977), pages 1–21.

[Berenson et al. (1995)] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels", In *Proc. of the ACM SIGMOD Conf. on Management of Data* (1995), pages 1–10.

[Bernstein and Newcomer (2009)] P. A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd edition, Morgan Kaufmann (2009).

[Cahill et al. (2009)] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases", *ACM Transactions on Database Systems*, Volume 34, Number 4 (2009), pages 20:1–20:42.

[Dechev et al. (2010)] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs", In *IEEE Int'l Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing, (ISORC)* (2010), pages 185–192.

[Eswaran et al. (1976)] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", *Communications of the ACM*, Volume 19, Number 11 (1976), pages 624–633.

[Faerber et al. (2017)] F. Faerber, A. Kemper, P.-A. Larson, J. Levandoski, T. Neumann, and A. Pavlo, "Main Memory Database Systems", *Foundations and Trends in Databases*, Volume 8, Number 1-2 (2017), pages 1–130.

[Gray and Reuter (1993)] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann (1993).

[Gray et al. (1975)] J. Gray, R. A. Lorie, and G. R. Putzolu, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", In *Proc. of the International Conf. on Very Large Databases* (1975), pages 428–451.

[Johnson and Shasha (1993)] T. Johnson and D. Shasha, "The Performance of Concurrent B-Tree Algorithms", *ACM Transactions on Database Systems*, Volume 18, Number 1 (1993), pages 51–101.

[Kung and Lehman (1980)] H. T. Kung and P. L. Lehman, "Concurrent Manipulation of Binary Search Trees", *ACM Transactions on Database Systems*, Volume 5, Number 3 (1980), pages 339–353.

[Kung and Robinson (1981)] H. T. Kung and J. T. Robinson, "Optimistic Concurrency Control", *ACM Transactions on Database Systems*, Volume 6, Number 2 (1981), pages 312–326.

[Lehman and Yao (1981)] P. L. Lehman and S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, Volume 6, Number 4 (1981), pages 650–670.

[Mohan (1990)] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operations on B-Tree indexes", In *Proc. of the International Conf. on Very Large Databases* (1990), pages 392–405.

[Mohan and Narang (1992)] C. Mohan and I. Narang, "Efficient Locking and Caching of Data in the Multisystem Shared Disks Transaction Environment", In *Proc. of the International Conf. on Extending Database Technology* (1992), pages 453–468.

[Ports and Grittner (2012)] D. R. K. Ports and K. Grittner, "Serializable Snapshot Isolation in PostgreSQL", *Proceedings of the VLDB Endowment*, Volume 5, Number 12 (2012), pages 1850–1861.

[Reed (1983)] D. Reed, "Implementing Atomic Actions on Decentralized Data", *Transactions on Computer Systems*, Volume 1, Number 1 (1983), pages 3–23.

[Silberschatz (1982)] A. Silberschatz, "A Multi-Version Concurrency Control Scheme With No Rollbacks", In *Proc. of the ACM Symposium on Principles of Distributed Computing* (1982), pages 216–223.

Credits

The photo of the sailboats in the beginning of the chapter is due to ©Pavel Nesvadba/Shutterstock.