This page intentionally left blank

chapter 21

Concurrency Control Techniques

In this chapter, we discuss a number of concurrency control techniques that are used to ensure the noninterference or isolation property of concurrently executing transactions. Most of these techniques ensure serializability of schedules—which we defined in Section 21.5—using **concurrency control protocols** (sets of rules) that guarantee serializability. One important set of protocols—known as *two-phase locking protocols*—employs the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently; a number of locking protocols are described in Sections 21.1 and 21.3.2. Locking protocols are used in some commercial DBMSs, but they are considered to have high overhead. Another set of concurrency control protocols uses **timestamps**. A timestamp is a unique identifier for each transaction, generated by the system. Timestamp values are generated in the same order as the transaction start times. Concurrency control protocols that use timestamp ordering to ensure serializability are introduced in Section 21.2. In Section 21.3, we discuss **multiversion** concurrency control protocols that use multiple versions of a data item. One multiversion protocol extends timestamp order to multiversion timestamp ordering (Section 21.3.1), and another extends timestamp order to two-phase locking (Section 21.3.2). In Section 21.4, we present a protocol based on the concept of **validation** or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols**, and they also assume that multiple versions of a data item can exist. In Section 21.4, we discuss a protocol that is based on the concept of **snapshot isolation**, which can utilize techniques similar to those proposed in validation-based and multiversion methods; these protocols are used in a number of commercial DBMSs and in certain cases are considered to have lower overhead than locking-based protocols.

Another factor that affects concurrency control is the **granularity** of the data items—that is, what portion of the database a data item represents. An item can be as small as a single attribute (field) value or as large as a disk block, or even a whole file or the entire database. We discuss granularity of items and a multiple granularity concurrency control protocol, which is an extension of two-phase locking, in Section 21.5. In Section 21.6, we describe concurrency control issues that arise when indexes are used to process transactions, and in Section 21.7 we discuss some additional concurrency control concepts. Section 21.8 summarizes the chapter.

It is sufficient to read Sections 21.1, 21.5, 21.6, and 21.7, and possibly 21.3.2, if your main interest is an introduction to the concurrency control techniques that are based on locking.

21.1 Two-Phase Locking Techniques for Concurrency Control

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the database. Locks are used as a means of synchronizing the access by concurrent transactions to the database items. In Section 21.1.1, we discuss the nature and types of locks. Then, in Section 21.1.2, we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 21.1.3, we describe two problems associated with the use of locks—deadlock and starvation—and show how these problems are handled in concurrency control protocols.

21.1.1 Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking concepts gradually, first we discuss binary locks, which are simple but are also *too restrictive for database concurrency control purposes* and so are not used much. Then we discuss *shared/exclusive* locks—also known as *read/write* locks—which provide more general locking capabilities and are used in database locking schemes. In Section 21.3.2, we describe an additional type of lock called a *certify lock*, and we show how it can be used to improve performance of locking protocols.

Binary Locks. A **binary lock** can have two **states** or **values**: locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item X. If the value of the lock on X is 1, item X *cannot be accessed* by a database operation that requests the item. If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item X as **lock(*X*)**.

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item X by first issuing a **lock_item(*X*)** operation. If

```
lock_item(X):
B:  if LOCK(X) = 0                (*item is unlocked*)
        then LOCK(X) ←1           (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                  (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 21.1
Lock and unlock operations for binary locks.

LOCK(X) = 1, the transaction is forced to wait. If LOCK(X) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item X. When the transaction is through using the item, it issues an **unlock_item(X)** operation, which sets LOCK(X) back to 0 (**unlocks** the item) so that X may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock_item(X) and unlock_item(X) operations is shown in Figure 21.1.

Notice that the lock_item and unlock_item operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 21.1, the wait command within the lock_item(X) operation is usually implemented by putting the transaction in a waiting queue for item X until X is unlocked and the transaction can be granted access to it. Other transactions that also want to access X are placed in the same queue. Hence, the wait command is considered to be outside the lock_item operation.

It is simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database. In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item. The system needs to maintain *only these records for the items that are currently locked* in a **lock table**, which could be organized as a hash file on the item name. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.

3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.[1]
4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

These rules can be enforced by the lock manager module of the DBMS. Between the lock_item(X) and unlock_item(X) operations in transaction T, T is said to **hold the lock** on item X. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

Shared/Exclusive (or Read/Write) Locks. The preceding binary locking scheme is too restrictive for database items because at most one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for *reading purposes only*. This is because read operations on the same item by different transactions are *not conflicting* (see Section 21.4.1). However, if a transaction is to write an item X, it must have exclusive access to X. For this purpose, a different type of lock, called a **multiple-mode lock**, is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read_lock(X), write_lock(X), and unlock(X). A lock associated with an item X, LOCK(X), now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table, as well as a list of transaction ids that hold a shared lock. Each record in the lock table will have four fields: <Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>. The system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If LOCK(X) = write-locked, the value of locking_transaction(s) is a *single transaction* that holds the exclusive (write) lock on X. If LOCK(X)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on X. The three operations read_lock(X), write_lock(X), and unlock(X) are described in Figure 21.2.[2] As before, each of the three locking operations should be considered indivisible; no interleaving should be allowed once one of the operations is started until either the operation terminates by granting the lock or the transaction is placed in a waiting queue for the item.

---

[1]This rule may be removed if we modify the lock_item (X) operation in Figure 21.1 so that if the item is currently locked *by the requesting transaction*, the lock is granted.

[2]These algorithms do not allow *upgrading* or *downgrading* of locks, as described later in this section. The reader can extend the algorithms to allow these additional operations.

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Figure 21.2
Locking and unlocking operations for two-mode (read/write, or shared/exclusive) locks.

When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.[3]

[3]This rule may be relaxed to allow a transaction to unlock an item, then lock it again later. However, two-phase locking does not allow this.

4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed for downgrading of locks, as we discuss shortly.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed for upgrading of locks, as we discuss shortly.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion (Upgrading, Downgrading) of Locks. It is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to **upgrade** the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking_transaction(s) field) to store the information on which transactions hold locks on the item. The descriptions of the read_lock(X) and write_lock(X) operations in Figure 21.2 must be changed appropriately to allow for lock upgrading and downgrading. We leave this as an exercise for the reader.

Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 21.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 21.3(a) the items Y in T_1 and X in T_2 were unlocked too early. This allows a schedule such as the one shown in Figure 21.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction. The best-known protocol, two-phase locking, is described in the next section.

21.1.2 Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction.[4] Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the

[4]This is unrelated to the two-phase commit protocol for recovery in distributed databases (see Chapter 23).

(a)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b)

Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

Time

Result of schedule S: X=50, Y=50 (nonserializable)

Figure 21.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

Transactions T_1 and T_2 in Figure 21.3(a) do not follow the two-phase locking protocol because the write_lock(X) operation follows the unlock(Y) operation in T_1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T_2. If we enforce two-phase locking, the transactions can be rewritten as T_1' and T_2', as shown in Figure 21.4. Now, the schedule shown in Figure 21.3(c) is not permitted for T_1' and T_2' (with their modified order of locking and unlocking operations) under the rules of locking described in Section 21.1.1 because T_1' will issue its write_lock(X) *before* it unlocks item Y; consequently, when T_2' issues its read_lock(X), it is forced to wait until T_1' releases the lock by issuing an unlock (X) in the schedule. However, this can lead to deadlock (see Section 21.1.3).

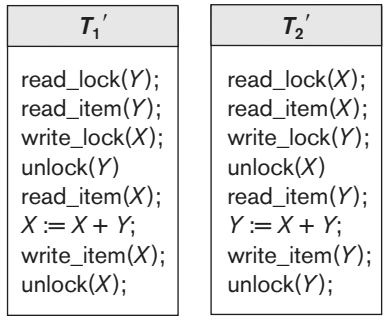

Figure 21.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

It can be proved that, if *every* transaction in a schedule follows the two-phase locking protocol, the schedule is *guaranteed to be serializable*, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.

Two-phase locking may limit the amount of concurrency that can occur in a schedule because a transaction T may not be able to release an item X after it is through using it if T must lock an additional item Y later; or, conversely, T must lock the additional item Y before it needs it so that it can release X. Hence, X must remain locked by T until all items that the transaction needs to read or write have been locked; only then can X be released by T. Meanwhile, another transaction seeking to access X may be forced to wait, even though T is done with X; conversely, if Y is locked earlier than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit *all possible* serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

Basic, Conservative, Strict, and Rigorous Two-Phase Locking. There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. Recall from Section 21.1.2 that the **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a *deadlock-free protocol*, as we will see in Section 21.1.3 when we discuss the deadlock problem. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations.

In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules (see Section 21.4). In this variation, a transaction T does not release any

of its exclusive (write) locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free. A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction T does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL.

Notice the difference between strict and rigorous 2PL: the former holds write-locks until it commits, whereas the latter holds all locks (read and write). Also, the difference between conservative and rigorous 2PL is that the former must lock all its items *before it starts*, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until *after it terminates* (by committing or aborting), so the transaction is in its expanding phase until it ends.

Usually the **concurrency control subsystem** itself is responsible for generating the read_lock and write_lock requests. For example, suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction T issues a read_item(X), the system calls the read_lock(X) operation on behalf of T. If the state of LOCK(X) is write_locked by some other transaction T', the system places T in the waiting queue for item X; otherwise, it grants the read_lock(X) request and permits the read_item(X) operation of T to execute. On the other hand, if transaction T issues a write_item(X), the system calls the write_lock(X) operation on behalf of T. If the state of LOCK(X) is write_locked or read_locked by some other transaction T', the system places T in the waiting queue for item X; if the state of LOCK(X) is read_locked and T itself is the only transaction holding the read lock on X, the system upgrades the lock to write_locked and permits the write_item(X) operation by T. Finally, if the state of LOCK(X) is unlocked, the system grants the write_lock(X) request and permits the write_item(X) operation to execute. After each action, the system must *update its lock table* appropriately.

Locking is generally considered to have a high overhead, because every read or write operation is preceded by a system locking request. The use of locks can also cause two additional problems: deadlock and starvation. We discuss these problems and their solutions in the next section.

21.1.3 Dealing with Deadlock and Starvation

Deadlock occurs when *each* transaction T in a set of *two or more transactions* is waiting for some item that is locked by some other transaction T' in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock. A simple example is shown in Figure 21.5(a), where the two transactions T_1' and T_2' are deadlocked in a partial schedule; T_1' is in the waiting queue for X, which is locked by T_2', whereas T_2' is in the waiting queue for Y, which is locked by T_1'. Meanwhile, neither T_1' nor T_2' nor any other transaction can access items X and Y.

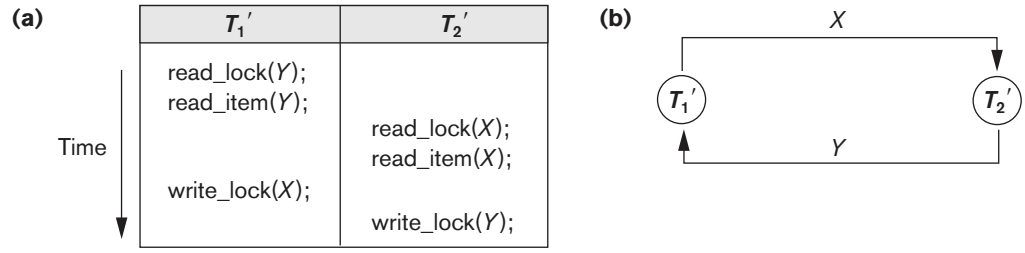

Figure 21.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

Deadlock Prevention Protocols. One way to prevent deadlock is to use a **deadlock prevention protocol**.[5] One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously, this solution further limits concurrency. A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction? Some of these techniques use the concept of **transaction timestamp** TS(T'), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which transactions are started; hence, if transaction T_1 starts before transaction T_2, then TS(T_1) < TS(T_2). Notice that the *older* transaction (which starts first) has the *smaller* timestamp value. Two schemes that prevent deadlock are called *wait-die* and *wound-wait*. Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:

- **Wait-die.** If TS(T_i) < TS(T_j), then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i *dies*) and restart it later *with the same timestamp*.
- **Wound-wait.** If TS(T_i) < TS(T_j), then (T_i older than T_j) abort T_j (T_i *wounds* T_j) and restart it later *with the same timestamp;* otherwise (T_i younger than T_j) T_i is allowed to wait.

[5]These protocols are not generally used in practice, either because of unrealistic assumptions or because of their possible overhead. Deadlock detection and timeouts (covered in the following sections) are more practical.

In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it. Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing. It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created. However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may *never actually cause a deadlock*.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the **no waiting algorithm**, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly. The **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock. The cautious waiting rule is as follows:

- **Cautious waiting.** If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time $b(T)$ at which each blocked transaction T was blocked, if the two transactions T_i and T_j above both become blocked and T_i is waiting for T_j, then $b(T_i) < b(T_j)$, since T_i can only wait for T_j at a time when T_j is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Deadlock Detection. An alternative approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is heavy, it may be advantageous to use a deadlock prevention scheme.

A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge $(T_i \rightarrow T_j)$ is created in the wait-for graph. When T_j releases the lock(s) on the items that T_i was

waiting for, the directed edge is dropped from the wait-for graph. We have a state of deadlock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining *when* the system should check for a deadlock. One possibility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 21.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 21.5(a).

If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as **victim selection**. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

Timeouts. Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists.

Starvation. Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

21.2 Concurrency Control Based on Timestamp Ordering

The use of locking, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted

because of the deadlock problem. A different approach to concurrency control involves using transaction timestamps to order transaction execution for an equivalent serial schedule. In Section 21.2.1, we discuss timestamps; and in Section 21.2.2, we discuss how serializability is enforced by ordering conflicting operations in different transactions based on the transaction timestamps.

21.2.1 Timestamps

Recall that a **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction T as **TS(*T*)**. Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, … in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

21.2.2 The Timestamp Ordering Algorithm for Concurrency Control

The idea for this scheme is to enforce the equivalent serial order on the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**. Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps. The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of *conflicting operations* in the schedule, the order in which the item is accessed must follow the timestamp order. To do this, the algorithm associates with each database item X two timestamp (TS) values:

1. **read_TS(*X*).** The **read timestamp** of item X is the largest timestamp among all the timestamps of transactions that have successfully read item X—that is, read_TS(X) = TS(T), where T is the *youngest* transaction that has read X successfully.
2. **write_TS(*X*).** The **write timestamp** of item X is the largest of all the timestamps of transactions that have successfully written item X—that is, write_TS(X) = TS(T), where T is the *youngest* transaction that has written X successfully. Based on the algorithm, T will also be the last transaction to write item X, as we shall see.

Basic Timestamp Ordering (TO). Whenever some transaction T tries to issue a read_item(X) or a write_item(X) operation, the **basic TO** algorithm compares the timestamp of T with read_TS(X) and write_TS(X) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction T is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If T is aborted and rolled back, any transaction T_1 that may have used a value written by T must also be rolled back. Similarly, any transaction T_2 that may have used a value written by T_1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction T issues a write_item(X) operation, the following check is performed:
   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
   b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).
2. Whenever a transaction T issues a read_item(X) operation, the following check is performed:
   a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already written the value of item X before T had a chance to read X.
   b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the *larger* of TS(T) and the current read_TS(X).

Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO). A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable. In this variation, a transaction T issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation *delayed* until the transaction T' that *wrote* the value of X (hence TS(T') = write_TS(X)) has committed or aborted.

To implement this algorithm, it is necessary to simulate the locking of an item X that has been written by transaction T' until T' is either committed or aborted. This algorithm *does not cause deadlock*, since T waits for T' only if $\mathsf{TS}(T) > \mathsf{TS}(T')$.

Thomas's Write Rule. A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).
3. If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

21.3 Multiversion Concurrency Control Techniques

These protocols for concurrency control keep copies of the old values of a data item when the item is updated (written); they are known as **multiversion concurrency control** because several versions (values) of an item are kept by the system. When a transaction requests to read an item, the *appropriate* version is chosen to maintain the serializability of the currently executing schedule. One reason for keeping multiple versions is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version(s) of the item is retained. Some multiversion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. In some cases, older versions can be kept in a temporary store. It is also possible that older versions may have to be maintained anyway—for example, for recovery purposes. Some database applications may require older versions to be kept to maintain a history of the changes of data item values. The extreme case is a *temporal database* (see Section 26.2), which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Several multiversion concurrency control schemes have been proposed. We discuss two schemes here, one based on timestamp ordering and the other based on 2PL. In addition, the validation concurrency control method (see Section 21.4) also maintains multiple versions, and the *snapshot isolation* technique used in

several commercial systems (see Section 21.4) can be implemented by keeping older versions of items in a temporary store.

21.3.1 Multiversion Technique Based on Timestamp Ordering

In this method, several versions $X_1, X_2, \ldots, X_k$ of each data item X are maintained. For *each version*, the value of version X_i and the following two timestamps associated with version X_i are kept:

1. **read_TS(X_i).** The **read timestamp** of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.
2. **write_TS(X_i).** The **write timestamp** of X_i is the timestamp of the transaction that wrote the value of version X_i.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version X_i, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) and TS(T).

To ensure serializability, the following rules are used:

1. If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).
2. If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

As we can see in case 2, a read_item(X) *is always successful*, since it finds the appropriate version X_i to read based on the write_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T attempts to write a version of X that should have been read by another transaction T' whose timestamp is read_TS(X_i); however, T' has already read version X_i, which was written by the transaction with timestamp equal to write_TS(X_i). If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created. Notice that if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

21.3.2 Multiversion Two-Phase Locking Using Certify Locks

In this multiple-mode locking scheme, there are *three locking modes* for an item—read, write, and *certify*—instead of just the two modes (read, write) discussed previously. Hence, the state of LOCK(X) for an item X can be one of read-locked, write-locked, certify-locked, or unlocked. In the standard locking scheme, with only read and write locks (see Section 21.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 21.6
Lock compatibility tables. (a) Lock compatibility table for read/write locking scheme. (b) Lock compatibility table for read/write/certify locking scheme.

by means of the **lock compatibility table** shown in Figure 21.6(a). An entry of *Yes* means that if a transaction T holds the type of lock specified in the column header on item X and if transaction T' requests the type of lock specified in the row header on the same item X, then T' *can obtain the lock* because the locking modes are compatible. On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so T' *must wait* until T *releases* the lock.

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions T' to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing *two versions* for each item X; one version, the **committed version**, must always have been written by some committed transaction. The second **local version** X' can be created when a transaction T acquires a write lock on X. Other transactions can continue to read the *committed version* of X while T holds the write lock. Transaction T can write the value of X' as needed, without affecting the value of the committed version X. However, once T is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit; this is another form of **lock upgrading**. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version X of the data item is set to the value of version X', version X' is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 21.6(b).

In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version X that was written by a committed transaction. However, deadlocks may occur, and these must be handled by variations of the techniques discussed in Section 21.1.3.

21.4 Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done *before* a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing. Several concurrency control methods are based on the validation technique. We will describe only one scheme in Section 21.4.1. Then, in Section 21.4.2, we discuss concurrency control techniques that are based on the concept of **snapshot isolation**. The implementations of these concurrency control methods can utilize a combination of the concepts from validation-based techniques and versioning techniques, as well as utilizing timestamps. Some of these methods may suffer from anomalies that can violate serializability, but because they generally have lower overhead than 2PL, they have been implemented in several relational DBMSs.

21.4.1 Validation-Based (Optimistic) Concurrency Control

In this scheme, updates in the transaction are *not* applied directly to the database items on disk until the transaction reaches its end and is *validated*. During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction.[6] At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation

[6]Note that this can be considered as keeping multiple versions of items!

phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later; under such circumstances, optimistic techniques do not work well. The techniques are called *optimistic* because they assume that little interference will occur and hence most transaction will be validated successfully, so that there is no need to do checking during transaction execution. This assumption is generally true in many transaction processing workloads.

The optimistic protocol we describe uses transaction timestamps and also requires that the write_sets and read_sets of the transactions be kept by the system. Additionally, *start* and *end* times for the three phases need to be kept for each transaction. Recall that the write_set of a transaction is the set of items it writes, and the read_set is the set of items it reads. In the validation phase for transaction T_i, the protocol checks that T_i does not interfere with any recently committed transactions or with any other concurrent transactions that have started their validation phase. The validation phase for T_i checks that, for *each* such transaction T_j that is either recently committed or is in its validation phase, *one* of the following conditions holds:

1. Transaction T_j completes its write phase before T_i starts its read phase.
2. T_i starts its write phase after T_j completes its write phase, and the read_set of T_i has no items in common with the write_set of T_j.
3. Both the read_set and write_set of T_i have no items in common with the write_set of T_j, and T_j completes its read phase before T_i completes its read phase.

When validating transaction T_i against each one of the transactions T_j, the first condition is checked first since (1) is the simplest condition to check. Only if condition 1 is false is condition 2 checked, and only if (2) is false is condition 3—the most complex to evaluate—checked. If any one of these three conditions holds with each transaction T_j, there is no interference and T_i is validated successfully. If *none* of these three conditions holds for any one T_j, the validation of transaction T_i fails (because T_i and T_j may violate serializability) and so T_i is aborted and restarted later because interference with T_j *may* have occurred.

21.4.2 Concurrency Control Based on Snapshot Isolation

As we discussed in Section 20.6, the basic definition of **snapshot isolation** is that a transaction sees the data items that it reads based on the committed values of the items in the *database snapshot* (or database state) when the transaction starts. Snapshot isolation will ensure that the phantom record problem does not occur, since the database transaction, or, in some cases, the database statement, will only see the records that were committed in the database at the time the transaction started. Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction. In addition, snapshot isolation does not allow the problems of dirty read and nonrepeatable read to occur. However, certain anomalies that violate serializability can occur when snapshot isolation is used as the basis for

concurrency control. Although these anomalies are rare, they are very difficult to detect and may result in an inconsistent or corrupted database. The interested reader can refer to the end-of-chapter bibliography for papers that discuss in detail the rare types of anomalies that can occur.

In this scheme, read operations do not require read locks to be applied to the items, thus reducing the overhead associated with two-phase locking. However, write operations do require write locks. Thus, for transactions that have many reads, the performance is much better than 2PL. When writes do occur, the system will have to keep track of older versions of the updated items in a **temporary version store** (sometimes known as tempstore), with the timestamps of when the version was created. This is necessary so that a transaction that started before the item was written can still read the value (version) of the item that was in the database snapshot when the transaction started.

To keep track of versions, items that have been updated will have pointers to a list of recent versions of the item in the *tempstore,* so that the correct item can be read for each transaction. The tempstore items will be removed when no longer needed, so a method to decide when to remove unneeded versions will be needed.

Variations of this method have been used in several commercial and open source DBMSs, including Oracle and PostGRES. If the users require guaranteed serializability, then the problems with anomalies that violate serializability will have to be solved by the programmers/software engineers by analyzing the set of transactions to determine which types of anomalies can occur, and adding checks that do not permit these anomalies. This can place a burden on the software developers when compared to the DBMS enforcing serializability in all cases.

Variations of snapshot isolation (SI) techniques, known as **serializable snapshot isolation (SSI)**, have been proposed and implemented in some of the DBMSs that use SI as their primary concurrency control method. For example, recent versions of the PostGRES DBMS allow the user to choose between basic SI and SSI. The tradeoff is ensuring full serializability with SSI versus living with possible rare anomalies but having better performance with basic SI. The interested reader is referred to the end-of-chapter bibliography for more complete discussions of these topics.

21.5 Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:

- A database record
- A field value of a database record
- A disk block
- A whole file
- The whole database

The particular choice of data item type can affect the performance of concurrency control and recovery. In Section 21.5.1, we discuss some of the tradeoffs with regard to choosing the granularity level used for locking; and in Section 21.5.2, we discuss a multiple granularity locking scheme, where the granularity level (size of the data item) may be changed dynamically.

21.5.1 Granularity Level Considerations for Locking

The size of data items is often called the **data item granularity**. *Fine granularity* refers to small item sizes, whereas *coarse granularity* refers to large item sizes. Several tradeoffs must be considered in choosing the data item size. We will discuss data item size in the context of locking, although similar arguments can be made for other concurrency control techniques.

First, notice that the larger the data item size is, the lower the degree of concurrency permitted. For example, if the data item size is a disk block, a transaction T that needs to lock a single record B must lock the whole disk block X that contains B because a lock is associated with the whole data item (block). Now, if another transaction S wants to lock a different record C that happens to reside in the same disk block X in a conflicting lock mode, it is forced to wait. If the data item size was a single record instead of a disk block, transaction S would be able to proceed, because it would be locking a different data item (record).

On the other hand, the smaller the data item size is, the more the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead. In addition, more storage space will be required for the lock table. For timestamps, storage is required for the read_TS and write_TS for each data item, and there will be similar overhead for handling a large number of items.

Given the above tradeoffs, an obvious question can be asked: What is the best item size? The answer is that it *depends on the types of transactions involved*. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record. On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items.

21.5.2 Multiple Granularity Level Locking

Since the best granularity size depends on the given transaction, it seems appropriate that a database system should support multiple levels of granularity, where the granularity level can be adjusted dynamically for various mixes of transactions. Figure 21.7 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records. This can be used to illustrate a **multiple granularity level** 2PL protocol, with shared/exclusive locking modes, where a lock can be requested at any level. However, additional types of locks will be needed to support such a protocol efficiently.

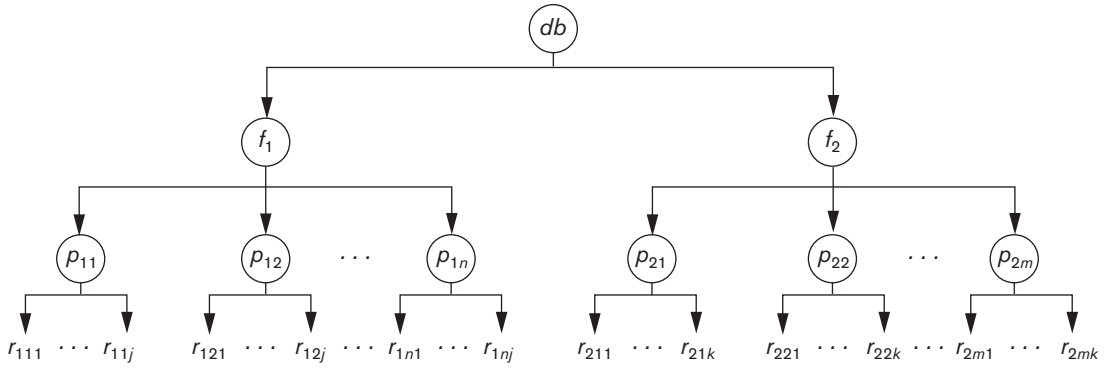

Figure 21.7
A granularity hierarchy for illustrating multiple granularity level locking.

Consider the following scenario, which refers to the example in Figure 21.7. Suppose transaction T_1 wants to update *all the records* in file f_1, and T_1 requests and is granted an exclusive lock for f_1. Then all of f_1's pages (p_{11} through p_{1n})—and the records contained on those pages—are locked in exclusive mode. This is beneficial for T_1 because setting a single file-level lock is more efficient than setting n page-level locks or having to lock each record individually. Now suppose another transaction T_2 only wants to read record r_{1nj} from page p_{1n} of file f_1; then T_2 would request a shared record-level lock on r_{1nj}. However, the database system (that is, the transaction manager or, more specifically, the lock manager) must verify the compatibility of the requested lock with already held locks. One way to verify this is to traverse the tree from the leaf r_{1nj} to p_{1n} to f_1 to *db*. If at any time a conflicting lock is held on any of those items, then the lock request for r_{1nj} is denied and T_2 is blocked and must wait. This traversal would be fairly efficient.

However, what if transaction T_2's request came *before* transaction T_1's request? In this case, the shared record lock is granted to T_2 for r_{1nj}, but when T_1's file-level lock is requested, it can be time-consuming for the lock manager to check all nodes (pages and records) that are descendants of node f_1 for a lock conflict. This would be very inefficient and would defeat the purpose of having multiple granularity level locks.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants. There are three types of intention locks:

1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 21.8
Lock compatibility matrix for multiple granularity locking.

3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The compatibility table of the three intention locks, and the actual shared and exclusive locks, is shown in Figure 21.8. In addition to the three types of intention locks, an appropriate locking protocol must be used. The **multiple granularity locking (MGL)** protocol consists of the following rules:

1. The lock compatibility (based on Figure 21.8) must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.
4. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.
5. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

Rule 1 simply states that conflicting locks cannot be granted. Rules 2, 3, and 4 state the conditions when a transaction may lock a given node in any of the lock modes. Rules 5 and 6 of the MGL protocol enforce 2PL rules to produce serializable schedules. Basically, the locking *starts from the root* and goes down the tree until the node that needs to be locked is encountered, whereas unlocking *starts from the locked node* and goes up the tree until the root itself is unlocked. To illustrate the MGL protocol with the database hierarchy in Figure 21.7, consider the following three transactions:

1. T_1 wants to update record r_{111} and record r_{211}.
2. T_2 wants to update all records on page p_{12}.
3. T_3 wants to read record $r_{11}j$ and the entire f_2 file.

T_1	T_2	T_3
IX(*db*)		
IX(f_1)		
	IX(*db*)	
		IS(*db*)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(*db*)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(*db*)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(*db*)

Figure 21.9
Lock operations to illustrate a serializable schedule.

Figure 21.9 shows a possible serializable schedule for these three transactions. Only the lock and unlock operations are shown. The notation <lock_type>(<item>) is used to display the locking operations in the schedule.

The multiple granularity level protocol is especially suited when processing a mix of transactions that include (1) short transactions that access only a few items (records or fields) and (2) long transactions that access entire files. In this environment, less transaction blocking and less locking overhead are incurred by such a protocol when compared to a single-level granularity locking approach.

21.6 Using Locks for Concurrency Control in Indexes

Two-phase locking can also be applied to B-tree and B^+-tree indexes (see Chapter 19), where the nodes of an index correspond to disk pages. However, holding locks on index pages until the shrinking phase of 2PL could cause an undue amount of transaction blocking because searching an index always *starts at the root*. For example, if a transaction wants to insert a record (write operation), the root would be locked in exclusive mode, so all other conflicting lock requests for the index must wait until the transaction enters its shrinking phase. This blocks all other transactions from accessing the index, so in practice other approaches to locking an index must be used.

The tree structure of the index can be taken advantage of when developing a concurrency control scheme. For example, when an index search (read operation) is being executed, a path in the tree is traversed from the root to a leaf. Once a lower-level node in the path has been accessed, the higher-level nodes in that path will not be used again. So once a read lock on a child node is obtained, the lock on the parent node can be released. When an insertion is being applied to a leaf node (that is, when a key and a pointer are inserted), then a specific leaf node must be locked in exclusive mode. However, if that node is not full, the insertion will not cause changes to higher-level index nodes, which implies that they need not be locked exclusively.

A conservative approach for insertions would be to lock the root node in exclusive mode and then to access the appropriate child node of the root. If the child node is not full, then the lock on the root node can be released. This approach can be applied all the way down the tree to the leaf, which is typically three or four levels from the root. Although exclusive locks are held, they are soon released. An alternative, more **optimistic approach** would be to request and hold *shared* locks on the nodes leading to the leaf node, with an *exclusive* lock on the leaf. If the insertion causes the leaf to split, insertion will propagate to one or more higher-level nodes. Then, the locks on the higher-level nodes can be upgraded to exclusive mode.

Another approach to index locking is to use a variant of the B^+-tree, called the **B-link tree**. In a B-link tree, sibling nodes on the same level are linked at every level. This allows shared locks to be used when requesting a page and requires that the lock be released before accessing the child node. For an insert operation, the shared lock on a node would be upgraded to exclusive mode. If a split occurs, the parent node must be relocked in exclusive mode. One complication is for search operations executed concurrently with the update. Suppose that a concurrent update operation follows the same path as the search and inserts a new entry into the leaf node. Additionally, suppose that the insert causes that leaf node to split. When the insert is done, the search process resumes, following the pointer to the desired leaf, only to find that the key it is looking for is not present because the split has moved that key into a new leaf node, which would be the *right sibling* of the original leaf

node. However, the search process can still succeed if it follows the pointer (link) in the original leaf node to its right sibling, where the desired key has been moved.

Handling the deletion case, where two or more nodes from the index tree merge, is also part of the B-link tree concurrency protocol. In this case, locks on the nodes to be merged are held as well as a lock on the parent of the two nodes to be merged.

21.7 Other Concurrency Control Issues

In this section, we discuss some other issues relevant to concurrency control. In Section 21.7.1, we discuss problems associated with insertion and deletion of records and we revisit the *phantom problem*, which may occur when records are inserted. This problem was described as a potential problem requiring a concurrency control measure in Section 20.6. In Section 21.7.2, we discuss problems that may occur when a transaction outputs some data to a monitor before it commits, and then the transaction is later aborted.

21.7.1 Insertion, Deletion, and Phantom Records

When a new data item is **inserted** in the database, it obviously cannot be accessed until after the item is created and the insert operation is completed. In a locking environment, a lock for the item can be created and set to exclusive (write) mode; the lock can be released at the same time as other write locks would be released, based on the concurrency control protocol being used. For a timestamp-based protocol, the read and write timestamps of the new item are set to the timestamp of the creating transaction.

Next, consider a **deletion operation** that is applied on an existing data item. For locking protocols, again an exclusive (write) lock must be obtained before the transaction can delete the item. For timestamp ordering, the protocol must ensure that no later transaction has read or written the item before allowing the item to be deleted.

A situation known as the **phantom problem** can occur when a new record that is being inserted by some transaction T satisfies a condition that a set of records accessed by another transaction T' must satisfy. For example, suppose that transaction T is inserting a new EMPLOYEE record whose Dno = 5, whereas transaction T' is accessing all EMPLOYEE records whose Dno = 5 (say, to add up all their Salary values to calculate the personnel budget for department 5). If the equivalent serial order is T followed by T', then T' must read the new EMPLOYEE record and include its Salary in the sum calculation. For the equivalent serial order T' followed by T, the new salary should not be included. Notice that although the transactions logically conflict, in the latter case there is really no record (data item) in common between the two transactions, since T' may have locked all the records with Dno = 5 *before* T inserted the new record. This is because the record that causes the conflict is a **phantom record** that has suddenly appeared in the database on being inserted. If other operations in the two transactions conflict, the conflict due to the phantom record may not be recognized by the concurrency control protocol.

One solution to the phantom record problem is to use **index locking**, as discussed in Section 21.6. Recall from Chapter 19 that an index includes entries that have an attribute value plus a set of pointers to all records in the file with that value. For example, an index on Dno of EMPLOYEE would include an entry for each distinct Dno value plus a set of pointers to all EMPLOYEE records with that value. If the index entry is locked before the record itself can be accessed, then the conflict on the phantom record can be detected, because transaction T' would request a read lock on the *index entry* for Dno = 5, and T would request a write lock on the same entry *before* it could place the locks on the actual records. Since the index locks conflict, the phantom conflict would be detected.

A more general technique, called **predicate locking**, would lock access to all records that satisfy an arbitrary *predicate* (condition) in a similar manner; however, predicate locks have proved to be difficult to implement efficiently. If the concurrency control method is based on snapshot isolation (see Section 21.4.2), then the transaction that reads the items will access the database snapshot at the time the transaction starts; any records inserted after that will not be retrieved by the transaction.

21.7.2 Interactive Transactions

Another problem occurs when interactive transactions read input and write output to an interactive device, such as a monitor screen, before they are committed. The problem is that a user can input a value of a data item to a transaction T that is based on some value written to the screen by transaction T', which may not have committed. This dependency between T and T' cannot be modeled by the system concurrency control method, since it is only based on the user interacting with the two transactions.

An approach to dealing with this problem is to postpone output of transactions to the screen until they have committed.

21.7.3 Latches

Locks held for a short duration are typically called **latches**. Latches do not follow the usual concurrency control protocol such as two-phase locking. For example, a latch can be used to guarantee the physical integrity of a disk page when that page is being written from the buffer to disk. A latch would be acquired for the page, the page written to disk, and then the latch released.

21.8 Summary

In this chapter, we discussed DBMS techniques for concurrency control. We started in Section 21.1 by discussing lock-based protocols, which are commonly used in practice. In Section 21.1.2 we described the two-phase locking (2PL) protocol and a number of its variations: basic 2PL, strict 2PL, conservative 2PL, and rigorous 2PL. The strict and rigorous variations are more common because of

their better recoverability properties. We introduced the concepts of shared (read) and exclusive (write) locks (Section 21.1.1) and showed how locking can guarantee serializability when used in conjunction with the two-phase locking rule. We also presented various techniques for dealing with the deadlock problem in Section 21.1.3, which can occur with locking. In practice, it is common to use timeouts and deadlock detection (wait-for graphs). Deadlock prevention protocols, such as no waiting and cautious waiting, can also be used.

We then presented other concurrency control protocols. These include the timestamp ordering protocol (Section 21.2), which ensures serializability based on the order of transaction timestamps. Timestamps are unique, system-generated transaction identifiers. We discussed Thomas's write rule, which improves performance but does not guarantee serializability. The strict timestamp ordering protocol was also presented. We discussed two multiversion protocols (Section 21.3), which assume that older versions of data items can be kept in the database. One technique, called multiversion two-phase locking (which has been used in practice), assumes that two versions can exist for an item and attempts to increase concurrency by making write and read locks compatible (at the cost of introducing an additional certify lock mode). We also presented a multiversion protocol based on timestamp ordering. In Section 21.4.1, we presented an example of an optimistic protocol, which is also known as a certification or validation protocol.

We then discussed concurrency control methods that are based on the concept of snapshot isolation in Section 21.4.2; these are used in several DBMSs because of their lower overhead. The basic snapshot isolation method can allow nonserializable schedules in rare cases because of certain anomalies that are difficult to detect; these anomalies may cause a corrupted database. A variation known as serializable snapshot isolation has been recently developed and ensures serializable schedules.

Then in Section 21.5 we turned our attention to the important practical issue of data item granularity. We described a multigranularity locking protocol that allows the change of granularity (item size) based on the current transaction mix, with the goal of improving the performance of concurrency control. An important practical issue was then presented in Section 21.6, which is to develop locking protocols for indexes so that indexes do not become a hindrance to concurrent access. Finally, in Section 21.7, we introduced the phantom problem and problems with interactive transactions, and we briefly described the concept of latches and how this concept differs from locks.

Review Questions

21.1. What is the two-phase locking protocol? How does it guarantee serializability?

21.2. What are some variations of the two-phase locking protocol? Why is strict or rigorous two-phase locking often preferred?

21.3. Discuss the problems of deadlock and starvation, and the different approaches to dealing with these problems.

21.4. Compare binary locks to exclusive/shared locks. Why is the latter type of locks preferable?

21.5. Describe the wait-die and wound-wait protocols for deadlock prevention.

21.6. Describe the cautious waiting, no waiting, and timeout protocols for deadlock prevention.

21.7. What is a timestamp? How does the system generate timestamps?

21.8. Discuss the timestamp ordering protocol for concurrency control. How does strict timestamp ordering differ from basic timestamp ordering?

21.9. Discuss two multiversion techniques for concurrency control. What is a certify lock? What are the advantages and disadvantages of using certify locks?

21.10. How do optimistic concurrency control techniques differ from other concurrency control techniques? Why are they also called validation or certification techniques? Discuss the typical phases of an optimistic concurrency control method.

21.11. What is snapshot isolation? What are the advantages and disadvantages of concurrency control methods that are based on snapshot isolation?

21.12. How does the granularity of data items affect the performance of concurrency control? What factors affect selection of granularity size for data items?

21.13. What type of lock is needed for insert and delete operations?

21.14. What is multiple granularity locking? Under what circumstances is it used?

21.15. What are intention locks?

21.16. When are latches used?

21.17. What is a phantom record? Discuss the problem that a phantom record can cause for concurrency control.

21.18. How does index locking resolve the phantom problem?

21.19. What is a predicate lock?

Exercises

21.20. Prove that the basic two-phase locking protocol guarantees conflict serializability of schedules. (*Hint*: Show that if a serializability graph for a schedule has a cycle, then at least one of the transactions participating in the schedule does not obey the two-phase locking protocol.)

21.21. Modify the data structures for multiple-mode locks and the algorithms for read_lock(X), write_lock(X), and unlock(X) so that upgrading and downgrading of locks are possible. (*Hint*: The lock needs to check the transaction id(s) that hold the lock, if any.)

21.22. Prove that strict two-phase locking guarantees strict schedules.

21.23. Prove that the wait-die and wound-wait protocols avoid deadlock and starvation.

21.24. Prove that cautious waiting avoids deadlock.

21.25. Apply the timestamp ordering algorithm to the schedules in Figures 21.8(b) and (c), and determine whether the algorithm will allow the execution of the schedules.

21.26. Repeat Exercise 21.25, but use the multiversion timestamp ordering method.

21.27. Why is two-phase locking not used as a concurrency control method for indexes such as B^+-trees?

21.28. The compatibility matrix in Figure 21.8 shows that IS and IX locks are compatible. Explain why this is valid.

21.29. The MGL protocol states that a transaction *T* can unlock a node *N*, only if none of the children of node *N* are still locked by transaction *T*. Show that without this condition, the MGL protocol would be incorrect.

Selected Bibliography

The two-phase locking protocol and the concept of predicate locks were first proposed by Eswaran et al. (1976). Bernstein et al. (1987), Gray and Reuter (1993), and Papadimitriou (1986) focus on concurrency control and recovery. Kumar (1996) focuses on performance of concurrency control methods. Locking is discussed in Gray et al. (1975), Lien and Weinberger (1978), Kedem and Silbershatz (1980), and Korth (1983). Deadlocks and wait-for graphs were formalized by Holt (1972), and the wait-wound and wound-die schemes are presented in Rosenkrantz et al. (1978). Cautious waiting is discussed in Hsu and Zhang (1992). Helal et al. (1993) compares various locking approaches.

Timestamp-based concurrency control techniques are discussed in Bernstein and Goodman (1980) and Reed (1983). Optimistic concurrency control is discussed in Kung and Robinson (1981) and Bassiouni (1988). Papadimitriou and Kanellakis (1979) and Bernstein and Goodman (1983) discuss multiversion techniques. Multiversion timestamp ordering was proposed in Reed (1979, 1983), and multiversion two-phase locking is discussed in Lai and Wilkinson (1984). A method for multiple locking granularities was proposed in Gray et al. (1975), and the effects of locking granularities are analyzed in Ries and Stonebraker (1977). Bhargava and Reidl (1988) presents an approach for dynamically choosing among various concurrency control and recovery methods. Concurrency control methods for indexes are presented in Lehman and Yao (1981) and in Shasha and Goodman (1988). A performance study of various B^+-tree concurrency control algorithms is presented in Srinivasan and Carey (1991).