# 17

# Introduction to Transaction Processing Concepts and Theory

The two subsequent chapters continue with more details on the techniques used to support transaction processing. Chapter 18 describes the basic concurrency control techniques, and Chapter 19 presents an overview of recovery techniques.

# 17.1 Introduction to Transaction Processing

In this section we informally introduce the concepts of concurrent execution of transactions and recovery from transaction failures. Section 17.1.1 compares single-user and multiuser database systems and demonstrates how concurrent execution of transactions can take place in multiuser systems. Section 17.1.2 defines the concept of transaction and presents a simple model of transaction execution, based on read and write database operations, that is used to formalize concurrency control and recovery concepts. Section 17.1.3 shows by informal examples why concurrency control techniques are needed in multiuser systems. Finally, Section 17.1.4 discusses why techniques are needed to permit recovery from failure by discussing the different ways in which transactions can fail while executing.

## 17.1.1 Single-User Versus Multiuser Systems

One criterion for classifying a database system is according to the number of users who can use the system **concurrently**—that is, at the same time. A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Systems in banks, insurance agencies, stock exchanges, supermarkets, and the like are also operated on by many users who submit transactions concurrently to the system.

Multiple users can access databases—and use computer systems—simultaneously because of the concept of **multiprogramming,** which allows the computer to execute multiple programs—or **processes**—at the same time. If only a single central processing unit (CPU) exists, it can actually execute at most one process at a time. However, **multiprogramming operating systems** execute some commands from one process, then suspend that process and execute some commands from the next process, and so on. A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved,** as illustrated in Figure 17.1, which shows two processes A and B executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

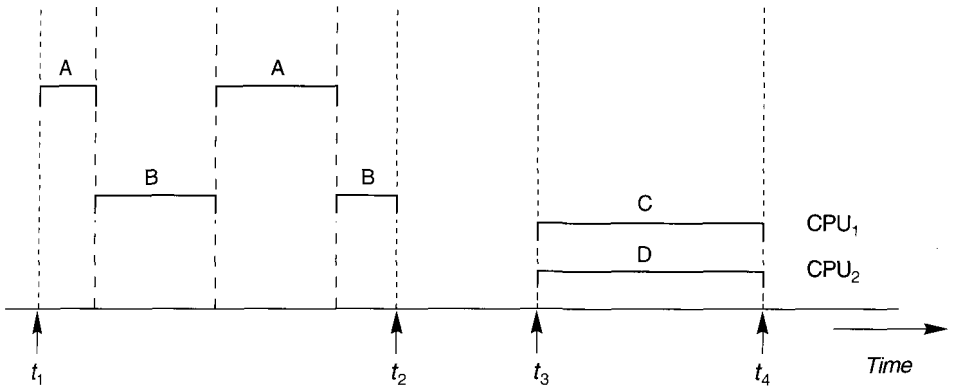


**FIGURE 17.1** Interleaved processing versus parallel processing of concurrent transactions.

If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 17.1. Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency,** so for the remainder of this chapter we assume this model. In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

## 17.1.2 Transactions, Read and Write Operations, and DBMS Buffers

A **transaction** is an executing program that forms a logical unit of database processing. A transaction includes one or more database access operations—these can include insertion, deletion, modification, or retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transaction if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction.**

The model of a database that is used to explain transaction processing concepts is much simplified. A **database** is basically represented as a collection of **named data items.** The size of a data item is called its **granularity,** and it can be a field of some record in the database, or it may be a larger unit such as a record or even a whole disk block, but the concepts we discuss are independent of the data item granularity. Using this simplified

database model, the basic database access operations that a transaction can include are as follows:

- **read_item(X):** Reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named* X.
- **write_item(X):** Writes the value of program variable X into the database item named X.

As we discussed in Chapter 13, the basic unit of data transfer from disk to main memory is one block. Executing a read_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the buffer to the program variable named X.

Executing a write_item(X) command includes the following steps:

1. Find the address of the disk block that contains item X.
2. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
3. Copy item X from the program variable named X into its correct location in the buffer.
4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Step 4 is the one that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store back a modified disk block that is in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system. The DBMS will generally maintain a number of **buffers** in main memory that hold database disk blocks containing the database items being processed. When these buffers are all occupied, and additional database blocks must be copied into memory, some buffer replacement policy is used to choose which of the current buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.[1]

A transaction includes read_item and write_item operations to access and update the database. Figure 17.2 shows examples of two very simple transactions. 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 the transaction writes. For example, the read-set of $T_1$ in Figure 17.2 is {X, Y} and its write-set is also {X, Y}.

Concurrency control and recovery mechanisms are mainly concerned with the database access commands in a transaction. Transactions submitted by the various users may

1. We will not discuss buffer replacement policies here as these are typically discussed in operating systems textbooks.

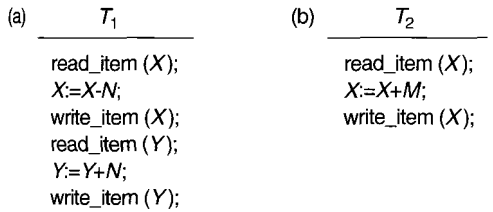


**FIGURE 17.2** Two sample transactions. (a) Transaction $T_1$. (b) Transaction $T_2$.

execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database. In the next section we informally introduce some of the problems that may occur.

## 17.1.3 Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner. We illustrate some of these problems by referring to a much simplified airline reservations database in which a record is stored for each airline flight. Each record includes the number of reserved seats on that flight as a *named data item*, among other information. Figure 17.2a shows a transaction $T_1$ that *transfers* $N$ reservations from one flight whose number of reserved seats is stored in the database item named X to another flight whose number of reserved seats is stored in the database item named Y. Figure 17.2b shows a simpler transaction $T_2$ that just *reserves* M seats on the first flight (X) referenced in transaction $T_1$.[2] To simplify our example, we do not show additional portions of the transactions, such as checking whether a flight has enough seats available before reserving additional seats.

When a database access program is written, it has the flight numbers, their dates, and the number of seats to be booked as parameters; hence, the same program can be used to execute many transactions, each with different flights and numbers of seats to be booked. For concurrency control purposes, a transaction is a *particular execution* of a program on a specific date, flight, and number of seats. In Figure 17.2a and b, the transactions $T_1$ and $T_2$ are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items X and Y in the database. We now discuss the types of problems we may encounter with these two transactions if they run concurrently.

The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions $T_1$ and $T_2$ are submitted at approximately the same time, and suppose that their operations are interleaved as shown

2. A similar, more commonly used example assumes a bank database, with one transaction doing a transfer of funds from account X to account Y and the other transaction doing a deposit to account X.

in Figure 17.3a; then the final value of item X is incorrect, because $T_2$ reads the value of X *before* $T_1$ changes it in the database, and hence the updated value resulting from $T_1$ is lost. For example, if X = 80 at the start (originally there were 80 reservations on the flight), N = 5 ($T_1$ transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and M = 4 ($T_2$ reserves 4 seats on X), the final result should be X = 79; but in the interleaving of operations shown in Figure 17.3a, it is X = 84 because the update in $T_1$ that removed the five seats from X was *lost*.

The Temporary Update (or Dirty Read) Problem. This problem occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4). The updated item is accessed by another transaction before it is changed

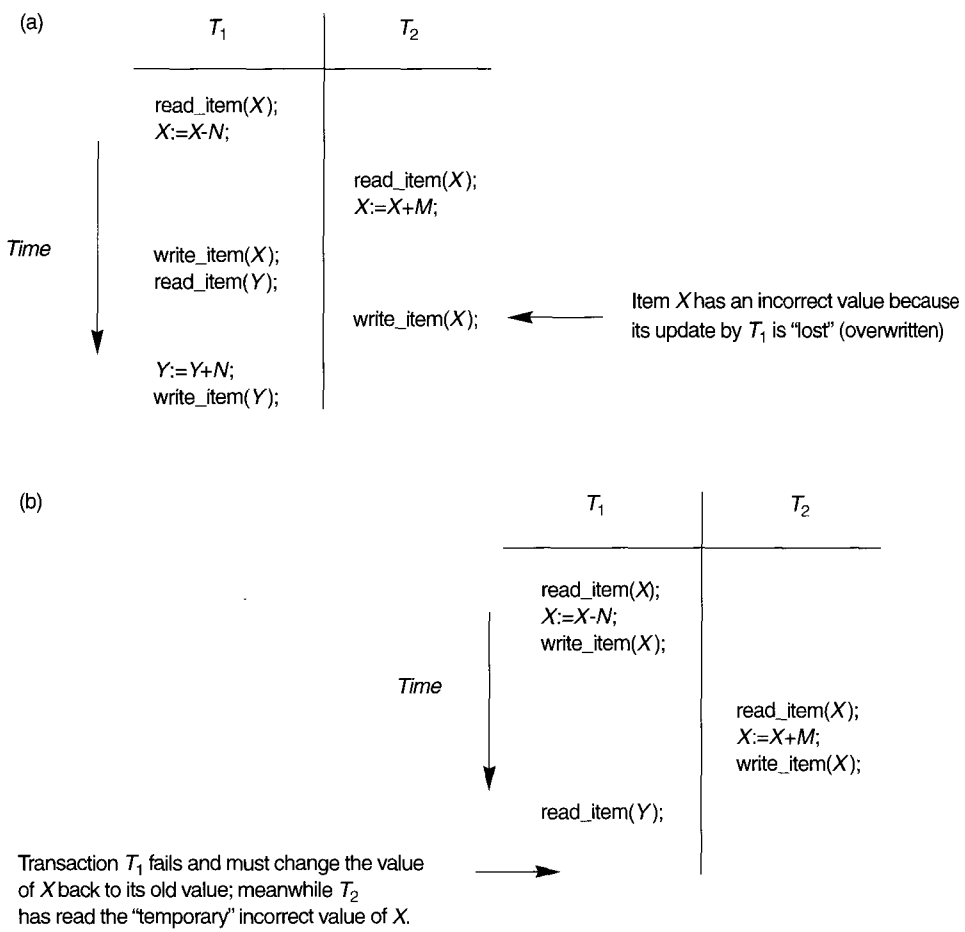


**FIGURE 17.3** Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem.

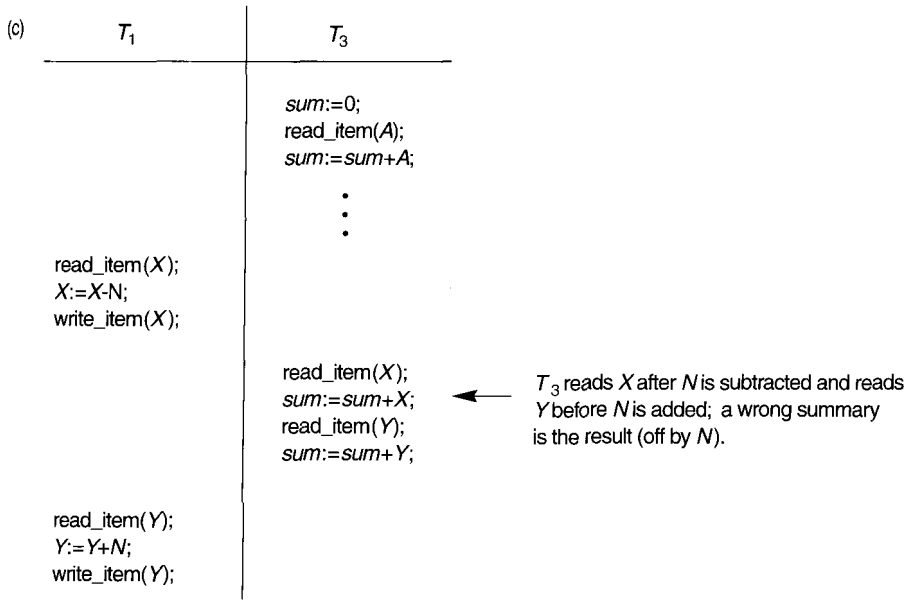


**FIGURE 17.3(CONTINUED)** Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.

back to its original value. Figure 17.3b shows an example where $T_1$ updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction $T_2$ reads the "temporary" value of X, which will not be recorded permanently in the database because of the failure of $T_1$. The value of item X that is read by $T_2$ is called *dirty data*, because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem*.

The Incorrect Summary Problem. If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction $T_3$ is calculating the total number of reservations on all the flights; meanwhile, transaction $T_1$ is executing. If the interleaving of operations shown in Figure 17.3c occurs, the result of $T_3$ will be off by an amount N because $T_3$ reads the value of X *after* N seats have been subtracted from it but reads the value of Y *before* those N seats have been added to it.

Another problem that may occur is called **unrepeatable read,** where a transaction T reads an item twice and the item is changed by another transaction T′ between the two reads. Hence, T receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.

### 17.1.4 Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or (2) the transaction has no effect whatsoever on the database or on any other transactions. The DBMS must not permit some operations of a transaction $T$ to be applied to the database while other operations of $T$ are not. This may happen if a transaction **fails** after executing some of its operations but before executing all of them.

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the middle of execution:

1. *A computer failure (system crash):* A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
2. *A transaction or system error:* Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error.[3] In addition, the user may interrupt the transaction during its execution.
3. *Local errors or exception conditions detected by the transaction:* During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exception condition,[4] such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception should be programmed in the transaction itself, and hence would not be considered a failure.
4. *Concurrency control enforcement:* The concurrency control method (see Chapter 18) may decide to abort the transaction, to be restarted later, because it violates serializability (see Section 17.5) or because several transactions are in a state of deadlock.
5. *Disk failure:* Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. *Physical problems and catastrophes:* This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

3. In general, a transaction should be thoroughly tested to ensure that it has no bugs (logical programming errors).

4. Exception conditions, if programmed correctly, do *not* constitute transaction failures.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task. We discuss recovery from failure in Chapter 19.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

# 17.2 Transaction and System Concepts

In this section we discuss additional concepts relevant to transaction processing. Section 17.2.1 describes the various states a transaction can be in, and discusses additional relevant operations needed in transaction processing. Section 17.2.2 discusses the system log, which keeps information needed for recovery. Section 17.2.3 describes the concept of commit points of transactions, and why they are important in transaction processing.

## 17.2.1 Transaction States and Additional Operations

A transaction is an atomic unit of work that is either completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts (see Section 17.2.3). Hence, the recovery manager keeps track of the following operations:

- `BEGIN_TRANSACTION`: This marks the beginning of transaction execution.
- `READ OR WRITE`: These specify read or write operations on the database items that are executed as part of a transaction.
- `END_TRANSACTION`: This specifies that `READ` and `WRITE` transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability (see Section 17.5) or for some other reason.
- `COMMIT_TRANSACTION`: This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- `ROLLBACK (OR ABORT)`: This signals that the transaction has *ended unsuccessfully*, so that any changes or effects that the transaction may have applied to the database must be *undone*.

Figure 17.4 shows a state transition diagram that describes how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can issue `READ` and `WRITE` operations. When the transaction ends, it moves to the **partially committed state.** At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the

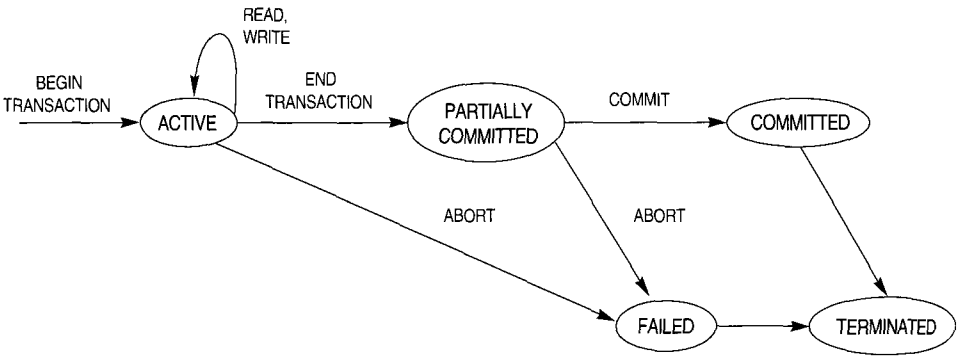


**FIGURE 17.4** State transition diagram illustrating the states for transaction execution.

transaction permanently (usually by recording changes in the system log, discussed in the next section).[5] Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state.** Commit points are discussed in more detail in Section 17.2.3. Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

However, a transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The **terminated state** corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions.

## 17.2.2 The System Log

To be able to recover from failures that affect transactions, the system maintains a log[6] to keep track of all transaction operations that affect the values of database items. This information may be needed to permit recovery from failures. The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures. We now list the types of entries—called **log records**—that are written to the log and the action each performs. In these entries, $T$ refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction:

1. [start_transaction,$T$]: Indicates that transaction $T$ has started execution.

5. Optimistic concurrency control (see Section 18.4) also requires that certain checks be made at this point to ensure that the transaction did not interfere with other executing transactions.

6. The log has sometimes been called the DBMS journal.

2. [write_item,T,X,old_value,new_value]: Indicates that transaction T has changed the value of database item X from old_value to new_value.
3. [read_item,T,X]: Indicates that transaction T has read the value of database item X.
4. [commit,T]: Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
5. [abort,T]: Indicates that transaction T has been aborted.

Protocols for recovery that avoid cascading rollbacks (see Section 17.4.2)—which include nearly all practical protocols—do not require that READ operations be written to the system log. However, if the log is also used for other purposes—such as auditing (keeping track of all database operations)—then such entries can be included. In addition, some recovery protocols require simpler WRITE entries that do not include new_value (see Section 17.4.2).

Notice that we assume here that *all* permanent changes to the database occur within transactions, so the notion of recovery from a transaction failure amounts to either undoing or redoing transaction operations individually from the log. If the system crashes, we can recover to a consistent database state by examining the log and using one of the techniques described in Chapter 19. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to **undo** the effect of these WRITE operations of a transaction T by tracing backward through the log and resetting all items changed by a WRITE operation of T to their old_values. **Redoing** the operations of a transaction may also be needed if all its updates are recorded in the log but a failure occurs before we can be sure that all these new_values have been written permanently in the actual database on disk.[7] Redoing the operations of transaction T is applied by tracing forward through the log and setting all items changed by a WRITE operation of T to their new_values.

## 17.2.3 Commit Point of a Transaction

A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database have been recorded in the log. Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database. The transaction then writes a commit record [commit,T] into the log. If a system failure occurs, we search back in the log for all transactions T that have written a [start_transaction,T] record into the log but have not written their [commit,T] record yet; these transactions may have to be *rolled back* to undo their effect on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

7. Undo and redo are discussed more fully in Chapter 19.

Notice that the log file must be kept on disk. As discussed in Chapter 13, updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk. It is common to keep one or more blocks of the log file in main memory buffers until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file block. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost. Hence, *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called **force-writing** the log file before committing a transaction.

## 17.3 Desirable Properties of Transactions

Transactions should possess several properties. These are often called the **ACID properties,** and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

The atomicity property requires that we execute a transaction to completion. It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.

The preservation of consistency is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints that should hold on the database. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

Isolation is enforced by the concurrency control subsystem of the DBMS.[8] If every transaction does not make its updates visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks (see Chapter 19). There have been attempts to define the *level of isolation* of a transaction. A transaction is said to have level 0 (zero) isolation if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates; and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to degree 2 properties, repeatable reads.

Finally, the durability property is the responsibility of the recovery subsystem of the DBMS. We will discuss how recovery protocols enforce durability and atomicity in Chapter 19.

# 17.4 Characterizing Schedules Based on Recoverability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from the various transactions is known as a **schedule** (or **history**). In this section, we first define the concept of schedule, and then we characterize the types of schedules that facilitate recovery when failures occur. In Section 17.5, we characterize schedules in terms of the interference of participating transactions, leading to the concepts of serializability and serializable schedules.

## 17.4.1 Schedules (Histories) of Transactions

A **schedule** (or **history**) $S$ of $n$ transactions $T_1, T_2, \ldots, T_n$ is an ordering of the operations of the transactions subject to the constraint that, for each transaction $T_i$ that participates in $S$, the operations of $T_i$ in $S$ must appear in the same order in which they occur in $T_i$. Note, however, that operations from other transactions $T_j$ can be interleaved with the operations of $T_i$ in $S$. For now, consider the order of operations in $S$ to be a *total ordering*, although it is possible theoretically to deal with schedules whose operations form *partial orders* (as we discuss later).

For the purpose of recovery and concurrency control, we are mainly interested in the `read_item` and `write_item` operations of the transactions, as well as the `commit` and `abort` operations. A shorthand notation for describing a schedule uses the symbols $r$, $w$, $c$, and $a$ for the operations `read_item`, `write_item`, `commit`, and `abort`, respectively, and appends as subscript the transaction id (transaction number) to each operation in the schedule. In this notation, the database item X that is read or written follows the $r$ and $w$

8. We will discuss concurrency control protocols in Chapter 18.

operations in parentheses. For example, the schedule of Figure 17.3(a), which we shall call $S_a$, can be written as follows in this notation:

$S_a$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

Similarly, the schedule for Figure 17.3(b), which we call $S_b$, can be written as follows, if we assume that transaction $T_1$ aborted after its `read_item(Y)` operation:

$S_b$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; $a_1$;

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions: (1) they belong to different transactions; (2) they access the same item X; and (3) at least one of the operations is a `write_item(X)`. For example, in schedule $S_a$, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$. However, the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations; the operations $w_2(X)$ and $w_1(Y)$ do not conflict, because they operate on distinct data items X and Y; and the operations $r_1(X)$ and $w_1(X)$ do not conflict, because they belong to the same transaction.

A schedule S of n transactions $T_1, T_2, \ldots, T_n$, is said to be a **complete schedule** if the following conditions hold:

1. The operations in S are exactly those operations in $T_1, T_2, \ldots, T_n$, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction $T_i$, their order of appearance in S is the same as their order of appearance in $T_i$.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.[9]

The preceding condition (3) allows for two *nonconflicting operations* to occur in the schedule without defining which occurs first, thus leading to the definition of a schedule as a **partial order** of the operations in the n transactions.[10] However, a total order must be specified in the schedule for any pair of conflicting operations (condition 3) and for any pair of operations from the same transaction (condition 2). Condition 1 simply states that all operations in the transactions must appear in the complete schedule. Since every transaction has either committed or aborted, a complete schedule will not contain any active transactions at the end of the schedule.

In general, it is difficult to encounter complete schedules in a transaction processing system, because new transactions are continually being submitted to the system. Hence, it is useful to define the concept of the **committed projection** C(S) of a schedule S, which includes only the operations in S that belong to committed transactions—that is, transactions $T_i$ whose commit operation $c_i$ is in S.

9. Theoretically, it is not necessary to determine an order between pairs of *nonconflicting* operations.

10. In practice, most schedules have a total order of operations. If parallel processing is employed, it is theoretically possible to have schedules with partially-ordered nonconflicting operations.

### 17.4.2 Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction failures, whereas for other schedules the recovery process can be quite involved. Hence, it is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple. These characterizations do not actually provide the recovery algorithm but instead only attempt to theoretically characterize the different types of schedules.

First, we would like to ensure that, once a transaction $T$ is committed, it should *never* be necessary to roll back $T$. The schedules that theoretically meet this criterion are called *recoverable schedules* and those that do not are called **nonrecoverable,** and hence should not be permitted. A schedule $S$ is recoverable if no transaction $T$ in $S$ commits until all transactions $T'$ that have written an item that $T$ reads have committed. A transaction $T$ **reads** from transaction $T'$ in a schedule $S$ if some item X is first written by $T'$ and later read by $T$. In addition, $T'$ should not have been aborted before $T$ reads item X, and there should be no transactions that write X after $T'$ writes it and before $T$ reads it (unless those transactions, if any, have aborted before $T$ reads X).

Recoverable schedules require a complex recovery process as we shall see, but if sufficient information is kept (in the log), a recovery algorithm can be devised. The (partial) schedules $S_a$ and $S_b$ from the preceding section are both recoverable, since they satisfy the above definition. Consider the schedule $S_a'$ given below, which is the same as schedule $S_a$ except that two commit operations have been added to $S_a$:

$S_a'$: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $w_1(Y)$; $c_1$;

$S_a'$ is recoverable, even though it suffers from the lost update problem. However, consider the two (partial) schedules $S_c$ and $S_d$ that follow:

$S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $a_1$;

$S_d$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $c_1$; $c_2$;

$S_e$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $a_1$; $a_2$;

$S_c$ is not recoverable, because $T_2$ reads item X from $T_1$, and then $T_2$ commits before $T_1$ commits. If $T_1$ aborts after the $c_2$ operation in $S_c$, then the value of X that $T_2$ read is no longer valid and $T_2$ must be aborted *after* it had been committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the $c_2$ operation in $S_c$ must be postponed until after $T_1$ commits, as shown in $S_d$; if $T_1$ aborts instead of committing, then $T_2$ should also abort as shown in $S_e$, because the value of X it read is no longer valid.

In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule $S_e$, where transaction $T_2$ has to be rolled back because it read item X from $T_1$, and $T_1$ then aborted.

Because cascading rollback can be quite time-consuming—since numerous transactions can be rolled back (see Chapter 19)—it is important to characterize the schedules where this phenomenon is guaranteed not to occur. A schedule is said to be **cascadeless,** or to **avoid cascading rollback,** if every transaction in the schedule reads only items that were

written by committed transactions. In this case, all items read will not be discarded, so no cascading rollback will occur. To satisfy this criterion, the $r_2(X)$ command in schedules $S_d$ and $S_e$ must be postponed until after $T_1$ has committed (or aborted), thus delaying $T_2$ but ensuring no cascading rollback if $T_1$ aborts.

Finally, there is a third, more restrictive type of schedule, called a **strict schedule,** in which transactions can *neither read nor write* an item X until the last transaction that wrote X has committed (or aborted). Strict schedules simplify the recovery process. In a strict schedule, the process of undoing a `write_item(X)` operation of an aborted transaction is simply to restore the **before image** (*old_value* or BFIM) of data item X. This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules. For example, consider schedule $S_f$:

$S_f$: $w_1(X, 5)$; $w_2(X, 8)$; $a_1$;

Suppose that the value of X was originally 9, which is the before image stored in the system log along with the $w_1(X, 5)$ operation. If $T_1$ aborts, as in $S_f$, the recovery procedure that restores the before image of an aborted write operation will restore the value of X to 9, even though it has already been changed to 8 by transaction $T_2$, thus leading to potentially incorrect results. Although schedule $S_f$ is cascadeless, it is not a strict schedule, since it permits $T_2$ to write item X even though the transaction $T_1$ that last wrote X had not yet committed (or aborted). A strict schedule does not have this problem.

We have now characterized schedules according to the following terms: (1) recoverability, (2) avoidance of cascading rollback, and (3) strictness. We have thus seen that those properties of schedules are successively more stringent conditions. Thus condition (2) implies condition (1), and condition (3) implies both (2) and (1). Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

## 17.5 Characterizing Schedules Based on Serializability

In the previous section, we characterized schedules based on their recoverability properties. We now characterize the types of schedules that are considered correct when concurrent transactions are executing. Suppose that two users—two airline reservation clerks—submit to the DBMS transactions $T_1$ and $T_2$ of Figure 17.2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction $T_1$ (in sequence) followed by all the operations of transaction $T_2$ (in sequence).
2. Execute all the operations of transaction $T_2$ (in sequence) followed by all the operations of transaction $T_1$ (in sequence).

These alternatives are shown in Figure 17.5a and b, respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transactions. Two possible schedules are shown

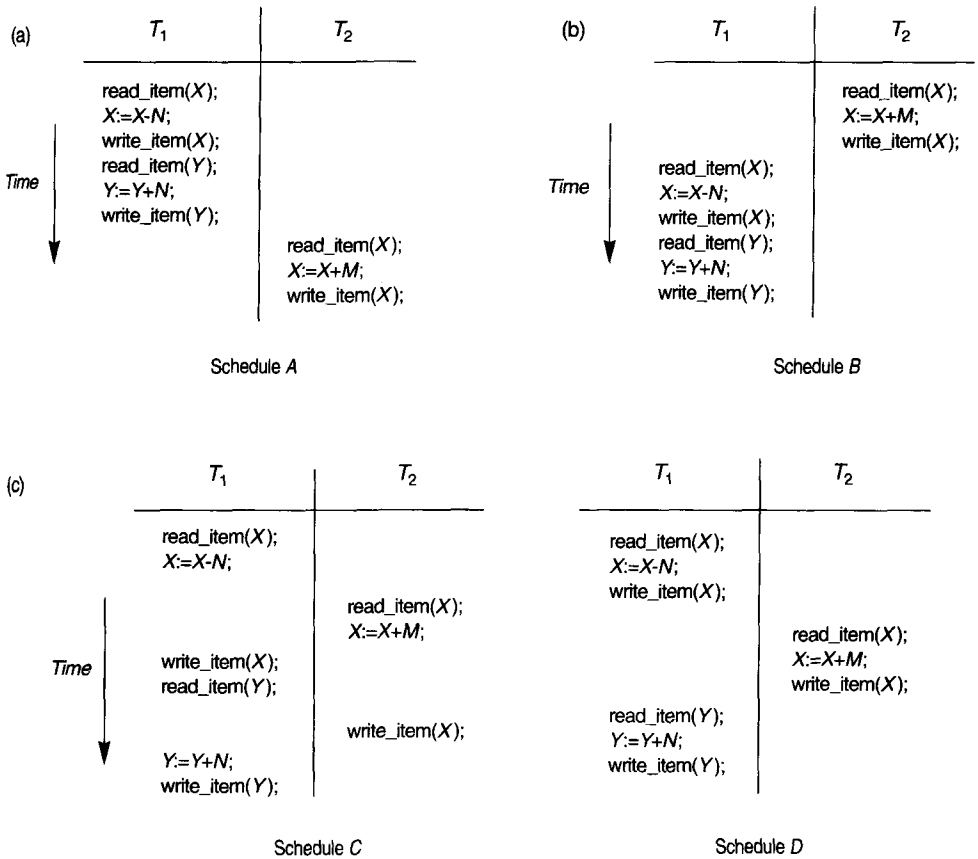


**FIGURE 17.5** Examples of serial and nonserial schedules involving transactions $T_1$ and $T_2$. (a) Serial schedule *A*: $T_1$ followed by $T_2$. (b) Serial schedule *B*: $T_2$ followed by $T_1$. (c) Two nonserial schedules *C* and *D* with interleaving of operations.

in Figure 17.5c. The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules. This section defines serializability and discusses how it may be used in practice.

## 17.5.1 Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figure 17.5a and b are called *serial* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: $T_1$ and then $T_2$ in Figure 17.5a, and $T_2$ and then $T_1$ in Figure 17.5b. Schedules C and *D*

in Figure 17.5c are called *nonserial* because each sequence interleaves operations from the two transactions.

Formally, a schedule $S$ is **serial** if, for every transaction $T$ participating in the schedule, all the operations of $T$ are executed consecutively in the schedule; otherwise, the schedule is called **nonserial.** Hence, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that every serial schedule is considered correct. We can assume this because every transaction is assumed to be correct if executed on its own (according to the consistency preservation property of Section 17.3). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end without any interference from the operations of other transactions, we get a correct end result on the database. The problem with serial schedules is that they limit concurrency or interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. In addition, if some transaction $T$ is quite long, the other transactions must wait for $T$ to complete all its operations before commencing. Hence, serial schedules are generally considered unacceptable in practice.

To illustrate our discussion, consider the schedules in Figure 17.5, and assume that the initial values of database items are X = 90 and Y = 90 and that $N = 3$ and $M = 2$. After executing transactions $T_1$ and $T_2$, we would expect the database values to be X = 89 and Y = 93, according to the meaning of the transactions. Sure enough, executing either of the serial schedules A or B gives the correct results. Now consider the nonserial schedules C and D. Schedule C (which is the same as Figure 17.3a) gives the results X = 92 and Y = 93, in which the X value is erroneous, whereas schedule D gives the correct results.

Schedule C gives an erroneous result because of the lost update problem discussed in Section 17.1.3; transaction $T_2$ reads the value of X *before* it is changed by transaction $T_1$, so only the effect of $T_2$ on X is reflected in the database. The effect of $T_1$ on X is *lost*, overwritten by $T_2$, leading to the incorrect result for item X. However, some nonserial schedules give the correct expected result, such as schedule D. We would like to determine which of the nonserial schedules *always* give a correct result and which may give erroneous results. The concept used to characterize schedules in this manner is that of serializability of a schedule.

A schedule $S$ of $n$ transactions is **serializable** if it is *equivalent to some serial schedule* of the same $n$ transactions. We will define the concept of equivalence of schedules shortly. Notice that there are $n!$ possible serial schedules of $n$ transactions and many more possible nonserial schedules. We can form two disjoint groups of the nonserial schedules: those that are equivalent to one (or more) of the serial schedules, and hence are serializable; and those that are not equivalent to *any* serial schedule and hence are not serializable.

Saying that a nonserial schedule $S$ is serializable is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct. The remaining question is: When are two schedules considered "equivalent"? There are several ways to define equivalence of schedules. The simplest, but least satisfactory, definition of schedule equivalence involves comparing the effects of the schedules on the

database. Two schedules are called **result equivalent** if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state. For example, in Figure 17.6, schedules $S_1$ and $S_2$ will produce the same final database state if they execute on a database with an initial value of $X = 100$; but for other initial values of X, the schedules are *not* result equivalent. In addition, these two schedules execute different transactions, so they definitely should not be considered equivalent. Hence, result equivalence alone cannot be used to define equivalence of schedules. The safest and most general approach to defining schedule equivalence is not to make any assumption about the types of operations included in the transactions. For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules *in the same order.* Two definitions of equivalence of schedules are generally used: *conflict equivalence* and *view equivalence*. We discuss conflict equivalence next, which is the more commonly used definition.

Two schedules are said to be **conflict equivalent** if the order of any two *conflicting operations* is the same in both schedules. Recall from Section 17.4.1 that two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and at least one of the two operations is a `write_item` operation. If two conflicting operations are applied in *different orders* in two schedules, the effect can be different on the database or on other transactions in the schedule, and hence the schedules are not conflict equivalent. For example, if a read and write operation occur in the order $r_1(X)$, $w_2(X)$ in schedule $S_1$, and in the reverse order $w_2(X)$, $r_1(X)$ in schedule $S_2$, the value read by $r_1(X)$ can be different in the two schedules. Similarly, if two write operations occur in the order $w_1(X)$, $w_2(X)$ in $S_1$, and in the reverse order $w_2(X)$, $w_1(X)$ in $S_2$, the next $r(X)$ operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different.

Using the notion of conflict equivalence, we define a schedule S to be **conflict serializable**[11] if it is (conflict) equivalent to some serial schedule S′. In such a case, we can reorder the *nonconflicting* operations in S until we form the equivalent serial schedule S′. According to this definition, schedule *D* of Figure 17.5c is equivalent to the serial

| $S_1$ | $S_2$ |
|---|---|
| read_item(*X*); | read_item(*X*); |
| *X*:=*X*+10; | *X*:=*X**1.1; |
| write_item(*X*); | write_item(*X*); |

**FIGURE 17.6** Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

11. We will use *serializable* to mean conflict serializable. Another definition of serializable used in practice (see Section 17.6) is to have repeatable reads, no dirty reads, and no phantom records (see Section 18.7.1 for a discussion on phantoms).

schedule A of Figure 17.5a. In both schedules, the `read_item`(X) of $T_2$ reads the value of X written by $T_1$, while the other `read_item` operations read the database values from the initial database state. In addition, $T_1$ is the last transaction to write Y, and $T_2$ is the last transaction to write X in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a *serializable schedule*. Notice that the operations $r_1(Y)$ and $w_1(Y)$ of schedule D do not conflict with the operations $r_2(X)$ and $w_2(X)$, since they access different data items. Hence, we can move $r_1(Y)$, $w_1(Y)$ before $r_2(X)$, $w_2(X)$, leading to the equivalent serial schedule $T_1$, $T_2$.

Schedule C of Figure 17.5c is not equivalent to either of the two possible serial schedules A and B, and hence is *not serializable*. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails, because $r_2(X)$ and $w_1(X)$ conflict, which means that we cannot move $r_2(X)$ down to get the equivalent serial schedule $T_1$, $T_2$. Similarly, because $w_1(X)$ and $w_2(X)$ conflict, we cannot move $w_1(X)$ down to get the equivalent serial schedule $T_2$, $T_1$.

Another, more complex definition of equivalence—called *view equivalence*, which leads to the concept of *view serializability*—is discussed in Section 17.5.4.

## 17.5.2 Testing for Conflict Serializability of a Schedule

There is a simple algorithm for determining the conflict serializability of a schedule. Most concurrency control methods do *not* actually test for serializability. Rather protocols, or rules, are developed that guarantee that a schedule will be serializable. We discuss the algorithm for testing conflict serializability of schedules here to gain a better understanding of these concurrency control protocols, which are discussed in Chapter 18.

Algorithm 17.1 can be used to test a schedule for conflict serializability. The algorithm looks at only the `read_item` and `write_item` operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph** G = (N, E) that consists of a set of nodes $N = \{T_1, T_2, \ldots, T_n\}$ and a set of directed edges $E = \{e_1, e_2, \ldots, e_m\}$. There is one node in the graph for each transaction $T_i$ in the schedule. Each edge $e_i$ in the graph is of the form $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, where $T_j$ is the **starting node** of $e_i$ and $T_k$ is the **ending node** of $e_i$. Such an edge is created if one of the operations in $T_j$ appears in the schedule *before* some *conflicting operation* in $T_k$.

**Algorithm 17.1:** Testing conflict serializability of a schedule S.

1. For each transaction $T_i$ participating in schedule S, create a node labeled $T_i$ in the precedence graph.
2. For each case in S where $T_j$ executes a `read_item`(X) after $T_i$ executes a `write_item`(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where $T_j$ executes a `write_item`(X) after $T_i$ executes a `read_item`(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where $T_j$ executes a `write_item`(X) after $T_i$ executes a `write_item`(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

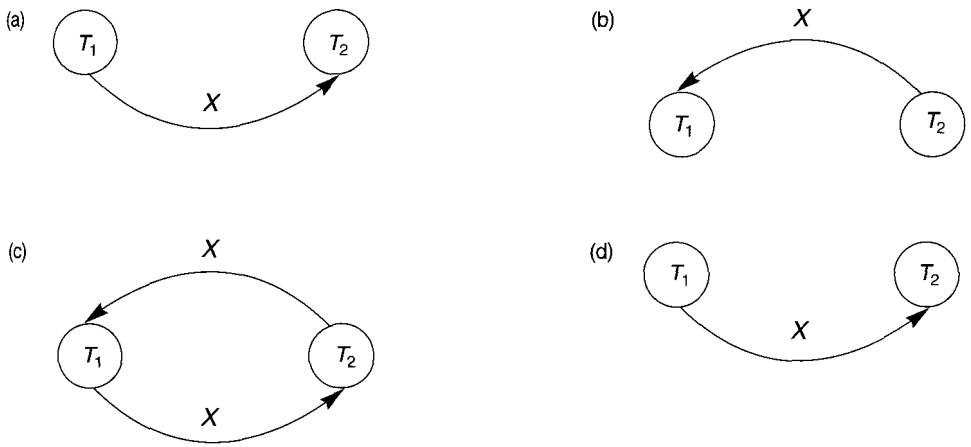


**FIGURE 17.7** Constructing the precedence graphs for schedules *A* to *D* from Figure 17.5 to test for conflict serializability. (a) Precedence graph for serial schedule *A*. (b) Precedence graph for serial schedule *B*. (c) Precedence graph for schedule *C* (not serializable). (d) Precedence graph for schedule *D* (serializable, equivalent to schedule *A*).

The precedence graph is constructed as described in Algorithm 17.1. If there is a cycle in the precedence graph, schedule $S$ is not (conflict) serializable; if there is no cycle, $S$ is serializable. A **cycle** in a directed graph is a **sequence of edges** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \ldots, (T_i \rightarrow T_j))$ with the property that the starting node of each edge—except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node).

In the precedence graph, an edge from $T_i$ to $T_j$ means that transaction $T_i$ must come before transaction $T_j$ in any serial schedule that is equivalent to $S$, because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an **equivalent serial schedule** $S'$ that is equivalent to $S$, by ordering the transactions that participate in $S$ as follows: Whenever an edge exists in the precedence graph from $T_i$ to $T_j$, $T_i$ must appear before $T_j$ in the equivalent serial schedule $S'$.[12] Notice that the edges $(T_i \rightarrow T_j)$ in a precedence graph can optionally be labeled by the name(s) of the data item(s) that led to creating the edge. Figure 17.7 shows such labels on the edges.

In general, several serial schedules can be equivalent to $S$ if the precedence graph for $S$ has no cycle. However, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so $S$ is not serializable. The precedence graphs created for schedules *A* to *D*, respectively, of Figure 17.5 appear in Figure 17.7a to d. The

12. This process of ordering the nodes of an acyclic graph is known as topological sorting.

graph for schedule C has a cycle, so it is not serializable. The graph for schedule *D* has no cycle, so it is serializable, and the equivalent serial schedule is $T_1$ followed by $T_2$. The graphs for schedules *A* and *B* have no cycles, as expected, because the schedules are *serial* and hence serializable.

Another example, in which three transactions participate, is shown in Figure 17.8. Figure 17.8a shows the `read_item` and `write_item` operations in each transaction. Two schedules *E* and *F* for these transactions are shown in Figure 17.8b and c, respectively, and the precedence graphs for schedules *E* and *F* are shown in parts d and e. Schedule *E* is not serializable, because the corresponding precedence graph has cycles. Schedule *F* is serializable, and the serial schedule equivalent to *F* is shown in Figure 17.8e. Although only one equivalent serial schedule exists for *F*, in general there may be *more than one equivalent serial schedule* for a serializable schedule. Figure 17.8f shows a precedence graph representing a schedule that has two equivalent serial schedules.

### 17.5.3 Uses of Serializability

As we discussed earlier, saying that a schedule *S* is (conflict) serializable—that is, *S* is (conflict) equivalent to a serial schedule—is tantamount to saying that *S* is correct. Being *serializable* is distinct from being *serial*, however. A serial schedule represents inefficient processing because no interleaving of operations from different transactions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for another transaction to terminate, thus slowing down processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness. In practice, it is quite difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transactions—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler, which allocates resources to all processes. Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule. Hence, it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability.

If transactions are executed at will and then the resulting schedule is tested for serializability, we must cancel the effect of the schedule if it turns out not to be serializable. This is a serious problem that makes this approach impractical. Hence, the approach taken in most practical systems is to determine methods that ensure serializability, without having to test the schedules themselves. The approach taken in most commercial DBMSs is to design **protocols** (sets of rules) that—if followed by *every* individual transaction or if enforced by a DBMS concurrency control subsystem—will ensure serializability of *all schedules in which the transactions participate*.

Another problem appears here: When transactions are submitted continuously to the system, it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule *S*. Recall from Section 17.4.1 that the *committed projection* C(*S*) of a schedule *S* includes only the operations in *S* that belong to committed transactions. We can theoretically define a schedule *S* to be serializable if its committed projection C(*S*) is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

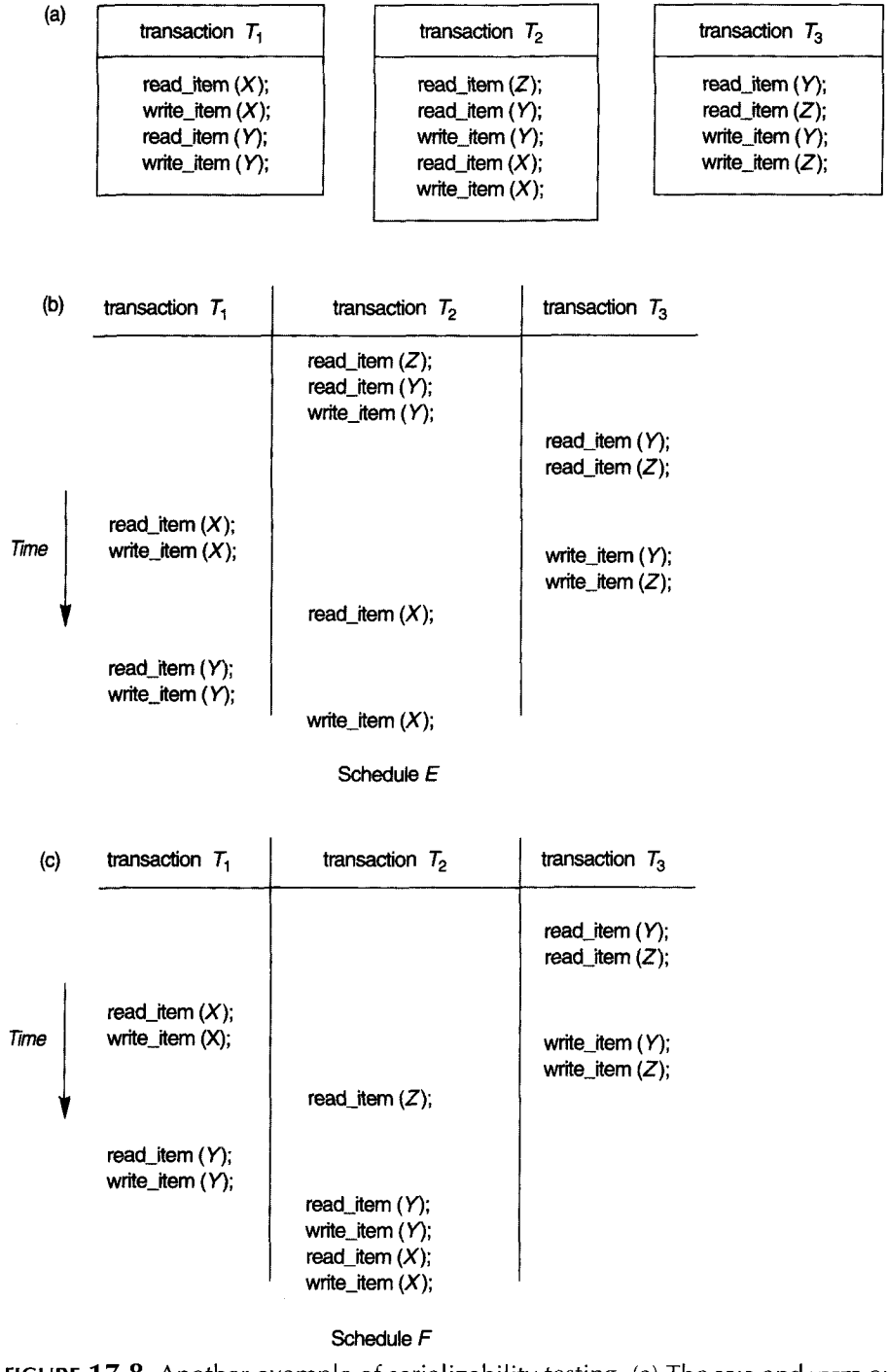


**FIGURE 17.8** Another example of serializability testing. (a) The READ and WRITE operations of three transactions $T_1$, $T_2$, and $T_3$. (b) Schedule *E*. (c) Schedule *F*.

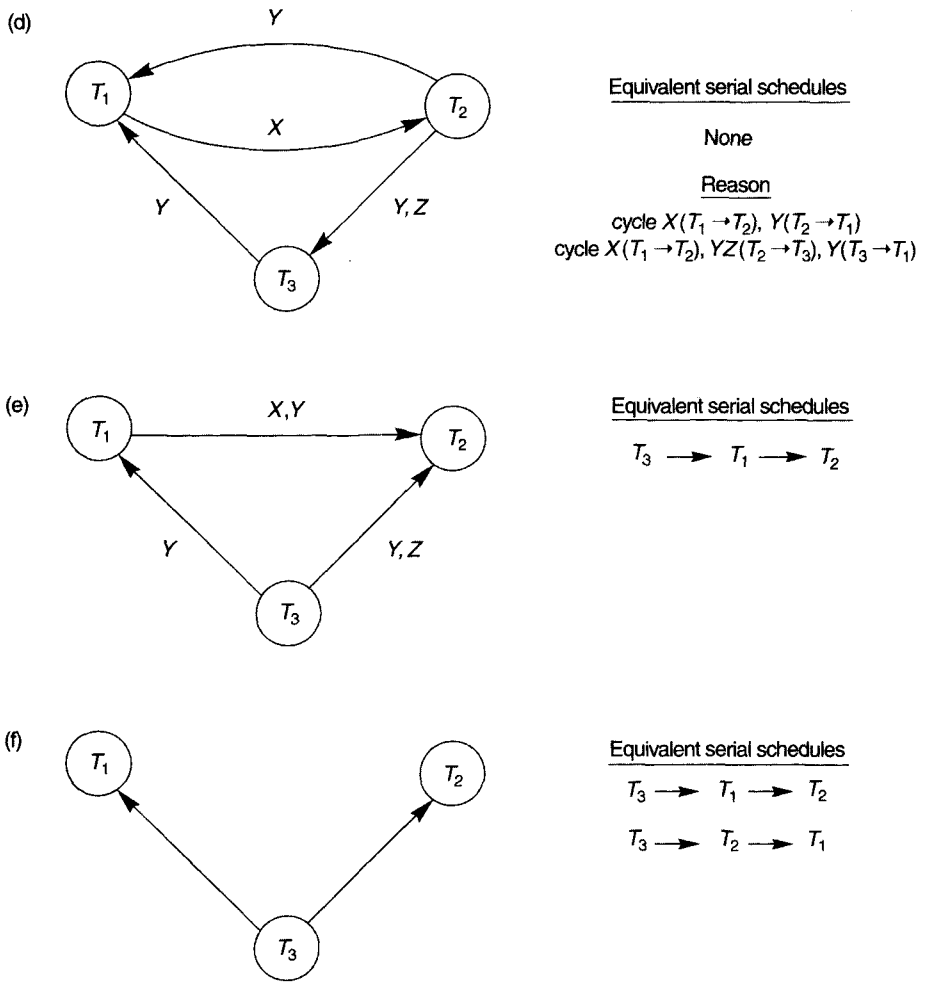


**FIGURE 17.8(CONTINUED)** Another example of serializability testing. (d) Precedence graph for schedule *E*. (e) Precedence graph for schedule *F*. (f) Precedence graph with two equivalent serial schedules.

In Chapter 18, we discuss a number of different concurrency control protocols that guarantee serializability. The most common technique, called *two-phase locking*, is based on locking data items to prevent concurrent transactions from interfering with one another, and enforcing an additional condition that guarantees serializability. This is used in the majority of commercial DBMSs. Other protocols have been proposed;[13] these

13. These other protocols have not been used much in practice so far; most systems use some variation of the two-phase locking protocol.

include *timestamp ordering*, where each transaction is assigned a unique timestamp and the protocol ensures that any conflicting operations are executed in the order of the transaction timestamps; *multiversion protocols*, which are based on maintaining multiple versions of data items; and *optimistic* (also called *certification* or *validation*) *protocols*, which check for possible serializability violations after the transactions terminate but before they are permitted to commit.

### 17.5.4 View Equivalence and View Serializability

In Section 17.5.1, we defined the concepts of conflict equivalence of schedules and conflict serializability. Another less restrictive definition of equivalence of schedules is called *view equivalence*. This leads to another definition of serializability called *view serializability*. Two schedules $S$ and $S'$ are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in $S$ and $S'$, and $S$ and $S'$ include the same operations of those transactions.
2. For any operation $r_i(X)$ of $T_i$ in $S$, if the value of $X$ read by the operation has been written by an operation $w_j(X)$ of $T_j$ (or if it is the original value of $X$ before the schedule started), the same condition must hold for the value of $X$ read by operation $r_i(X)$ of $T_i$ in $S'$.
3. If the operation $w_k(Y)$ of $T_k$ is the last operation to write item $Y$ in $S$, then $w_k(Y)$ of $T_k$ must also be the last operation to write item $Y$ in $S'$.

The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to *see the same view* in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules. A schedule $S$ is said to be **view serializable** if it is view equivalent to a serial schedule.

The definitions of conflict serializability and view serializability are similar if a condition known as the **constrained write assumption** holds on all transactions in the schedule. This condition states that any write operation $w_i(X)$ in $T_i$ is preceded by a $r_i(X)$ in $T_i$ *and* that the value written by $w_i(X)$ in $T_i$ depends only on the value of $X$ read by $r_i(X)$. This assumes that computation of the new value of $X$ is a function $f(X)$ based on the old value of $X$ read from the database. However, the definition of view serializability is less restrictive than that of conflict serializability under the **unconstrained write assumption,** where the value written by an operation $w_i(X)$ in $T_i$ can be independent of its old value from the database. This is called a **blind write,** and it is illustrated by the following schedule $S_g$ of three transactions $T_1$: $r_1(X)$; $w_1(X)$; $T_2$: $w_2(X)$; and $T_3$: $w_3(X)$:

$S_g$: $r_1(X)$; $w_2(X)$; $w_1(X)$; $w_3(X)$; $c_1$; $c_2$; $c_3$;

In $S_g$ the operations $w_2(X)$ and $w_3(X)$ are blind writes, since $T_2$ and $T_3$ do not read the value of $X$. The schedule $S_g$ is view serializable, since it is view equivalent to the serial

schedule $T_1$, $T_2$, $T_3$. However, $S_g$ is not conflict serializable, since it is not conflict equivalent to any serial schedule. It has been shown that any conflict-serializable schedule is also view serializable but not vice versa, as illustrated by the preceding example. There is an algorithm to test whether a schedule $S$ is view serializable or not. However, the problem of testing for view serializability has been shown to be NP-hard, meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

### 17.5.5 Other Types of Equivalence of Schedules

Serializability of schedules is sometimes considered to be too restrictive as a condition for ensuring the correctness of concurrent executions. Some applications can produce schedules that are correct by satisfying conditions less stringent than either conflict serializability or view serializability. An example is the type of transactions known as **debit-credit transactions**—for example, those that apply deposits and withdrawals to a data item whose value is the current balance of a bank account. The semantics of debit-credit operations is that they update the value of a data item X by either subtracting from or adding to the value of the data item. Because addition and subtraction operations are commutative—that is, they can be applied in any order—it is possible to produce correct schedules that are not serializable. For example, consider the following two transactions, each of which may be used to transfer an amount of money between two bank accounts:

$T_1$: $r_1(X)$; X := X − 10; $w_1(X)$; $r_1(Y)$; Y := Y + 10; $w_1(Y)$;

$T_2$: $r_2(Y)$; Y := Y − 20; $w_2(Y)$; $r_2(X)$; X := X + 20; $w_2(X)$;

Consider the following nonserializable schedule $S_h$ for the two transactions:

$S_h$: $r_1(X)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$;

With the additional knowledge, or **semantics,** that the operations between each $r_i(I)$ and $w_i(I)$ are commutative, we know that the order of executing the sequences consisting of (read, update, write) is not important as long as each (read, update, write) sequence by a particular transaction $T_i$ on a particular item $I$ is not interrupted by conflicting operations. Hence, the schedule $S_h$ is considered to be correct even though it is not serializable. Researchers have been working on extending concurrency control theory to deal with cases where serializability is considered to be too restrictive as a condition for correctness of schedules.

## 17.6 Transaction Support in SQL

The definition of an SQL-transaction is similar to our already defined concept of a transaction. That is, it is a logical unit of work and is guaranteed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without error or it fails and leaves the database unchanged.

With SQL, there is no explicit `Begin_Transaction` statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK. Every transaction has certain characteristics attributed to it. These characteristics are specified by a SET TRANSACTION statement in SQL. The characteristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.

The **access mode** can be specified as READ ONLY or READ WRITE. The default is READ WRITE, unless the isolation level of READ UNCOMMITTED is specified (see below), in which case READ ONLY is assumed. A mode of READ WRITE allows update, insert, delete and create commands to be executed. A mode of READ ONLY, as the name implies, is simply for data retrieval.

The **diagnostic area size** option, DIAGNOSTIC SIZE $n$, specifies an integer value $n$, indicating the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or exceptions) to the user or program on the most recently executed SQL statement.

The **isolation level** option is specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.[14] The default isolation level is SERIALIZABLE, although some systems use as READ COMMITTED their default. The use of the term SERIALIZABLE here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms,[15] and it is thus not identical to the way serializability was defined earlier in Section 17.5. If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three violations may occur:

1. **Dirty read:** A transaction $T_1$ may read the update of a transaction $T_2$, which has not yet committed. If $T_2$ fails and is aborted, then $T_1$ would have read a value that does not exist and is incorrect.
2. **Nonrepeatable read:** A transaction $T_1$ may read a given value from a table. If another transaction $T_2$ later updates that value and $T_1$ reads that value again, $T_1$ will see a different value.
3. **Phantoms:** A transaction $T_1$ may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction $T_2$ inserts a new row that also satisfies the WHERE-clause condition used in $T_1$, into the table used by $T_1$. If $T_1$ is repeated, then $T_1$ will see a phantom, a row that previously did not exist.

Table 17.1 summarizes the possible violations for the different isolation levels. An entry of "yes" indicates that a violation is possible and an entry of "no" indicates that it is not possible.

14. These are similar to the *isolation levels* discussed briefly at the end of Section 17.3.

15. The dirty read and unrepeatable read problems were discussed in Section 17.1.3. Phantoms are discussed in Section 18.6.1.

Possible Violations Based on Isolation Levels as Defined in SQL

| Isolation level | Type of Violation | | |
|---|---|---|---|
| | Dirty read | Nonrepeatable read | Phantom |
| READ UNCOMMITTTED | yes | yes | yes |
| READ COMMITTED | no | yes | yes |
| REPEATABLE READ | no | no | yes |
| SERIALIZABLE | no | no | no |

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
     READ WRITE
     DIAGNOSTIC SIZE 5
     ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
     VALUES ('ROBERT', 'SMITH', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
     SET SALARY = SALARY * 1.1 WHERE DNO = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ...;
```

The above transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2. If an error occurs on any of the SQL statements, the entire transaction is rolled back. This implies that any updated salary (by this transaction) would be restored to its previous value and that the newly inserted row would be removed.

As we have seen, SQL provides a number of transaction-oriented features. The DBA or database programmers can take advantage of these options to try improving transaction performance by relaxing serializability if that is acceptable for their applications.

## 17.7 SUMMARY

In this chapter we discussed DBMS concepts for transaction processing. We introduced the concept of a database transaction and the operations relevant to transaction processing. We compared single-user systems to multiuser systems and then presented examples of how uncontrolled execution of concurrent transactions in a multiuser system can lead to incorrect results and database values. We also discussed the various types of failures that may occur during transaction execution.

We then introduced the typical states that a transaction passes through during execution, and discussed several concepts that are used in recovery and concurrency control methods. The system log keeps track of database accesses, and the system uses this information to recover from failures. A transaction either succeeds and reaches its commit point or it fails and has to be rolled back. A committed transaction has its changes permanently recorded in the database. We presented an overview of the desirable properties of transactions—namely, atomicity, consistency preservation, isolation, and durability—which are often referred to as the ACID properties.

We then defined a schedule (or history) as an execution sequence of the operations of several transactions with possible interleaving. We characterized schedules in terms of their recoverability. Recoverable schedules ensure that, once a transaction commits, it never needs to be undone. Cascadeless schedules add an additional condition to ensure that no aborted transaction requires the cascading abort of other transactions. Strict schedules provide an even stronger condition that allows a simple recovery scheme consisting of restoring the old values of items that have been changed by an aborted transaction.

We then defined equivalence of schedules and saw that a serializable schedule is equivalent to some serial schedule. We defined the concepts of conflict equivalence and view equivalence, which led to definitions for conflict serializability and view serializability. A serializable schedule is considered correct. We then presented algorithms for testing the (conflict) serializability of a schedule. We discussed why testing for serializability is impractical in a real system, although it can be used to define and verify concurrency control protocols, and we briefly mentioned less restrictive definitions of schedule equivalence. Finally, we gave a brief overview of how transaction concepts are used in practice within SQL.

We will discuss concurrency control protocols in Chapter 18, and recovery protocols in Chapter 19.

## Review Questions

17.1. What is meant by the concurrent execution of database transactions in a multiuser system? Discuss why concurrency control is needed, and give informal examples.

17.2. Discuss the different types of failures. What is meant by catastrophic failure?

17.3. Discuss the actions taken by the `read_item` and `write_item` operations on a database.

17.4. Draw a state diagram, and discuss the typical states that a transaction goes through during execution.

17.5. What is the system log used for? What are the typical kinds of records in a system log? What are transaction commit points, and why are they important?

17.6. Discuss the atomicity, durability, isolation, and consistency preservation properties of a database transaction.

17.7. What is a schedule (history)? Define the concepts of recoverable, cascadeless, and strict schedules, and compare them in terms of their recoverability.

17.8. Discuss the different measures of transaction equivalence. What is the difference between conflict equivalence and view equivalence?

17.9. What is a serial schedule? What is a serializable schedule? Why is a serial schedule considered correct? Why is a serializable schedule considered correct?

17.10. What is the difference between the constrained write and the unconstrained write assumptions? Which is more realistic?

17.11. Discuss how serializability is used to enforce concurrency control in a database system. Why is serializability sometimes considered too restrictive as a measure of correctness for schedules?

17.12. Describe the four levels of isolation in SQL.

17.13. Define the violations caused by each of the following: dirty read, nonrepeatable read, and phantoms.

## Exercises

17.14. Change transaction $T_2$ in Figure 17.2b to read

```
read_item(X);
X:= X+M;
if X > 90 then exit
else write_item(X);
```

Discuss the final result of the different schedules in Figure 17.3(a) and (b), where M = 2 and N = 2, with respect to the following questions. Does adding the above condition change the final outcome? Does the outcome obey the implied consistency rule (that the capacity of X is 90)?

17.15. Repeat Exercise 17.14, adding a check in $T_1$ so that Y does not exceed 90.

17.16. Add the operation commit at the end of each of the transactions $T_1$ and $T_2$ from Figure 17.2; then list all possible schedules for the modified transactions. Determine which of the schedules are recoverable, which are cascadeless, and which are strict.

17.17. List all possible schedules for transactions $T_1$ and $T_2$ from Figure 17.2, and determine which are conflict serializable (correct) and which are not.

17.18. How many *serial* schedules exist for the three transactions in Figure 17.8(a)? What are they? What is the total number of possible schedules?

17.19. Write a program to create all possible schedules for the three transactions in Figure 17.8(a), and to determine which of those schedules are conflict serializable and which are not. For each conflict serializable schedule, your program should print the schedule and list all equivalent serial schedules.

17.20. Why is an explicit transaction end statement needed in SQL but not an explicit begin statement?

17.21. Describe situations where each of the different isolation levels would be useful for transaction processing.

17.22. Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.

a. $r_1(X); r_3(X); w_1(X); r_2(X); w_3(X);$
b. $r_1(X); r_3(X); w_3(X); w_1(X); r_2(X);$
c. $r_3(X); r_2(X); w_3(X); r_1(X); w_1(X);$
d. $r_3(X); r_2(X); r_1(X); w_3(X); w_1(X);$

17.23. Consider the three transactions $T_1$, $T_2$, and $T_3$, and the schedules $S_1$ and $S_2$ given below. Draw the serializability (precedence) graphs for $S_1$ and $S_2$, and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).

$T_1$: $r_1$ (X); $r_1$ (Z); $w_1$ (X);

$T_2$: $r_2$ (Z); $r_2$ (Y); $w_2$ (Z); $w_2$ (Y);

$T_3$: $r_3$ (X); $r_3$ (Y); $w_3$ (Y);

$S_1$: $r_1$ (X); $r_2$ (Z); $r_1$ (Z); $r_3$ (X); $r_3$ (Y); $w_1$ (X); $w_3$ (Y); $r_2$ (Y); $w_2$ (Z); $w_2$ (Y);

$S_2$: $r_1$ (X); $r_2$ (Z); $r_3$ (X); $r_1$ (Z); $r_2$ (Y); $r_3$ (Y); $w_1$ (X); $w_2$ (Z); $w_3$ (Y); $w_2$ (Y);

17.24. Consider schedules $S_3$, $S_4$, and $S_5$ below. Determine whether each schedule is strict, cascadeless, recoverable, or nonrecoverable. (Determine the strictest recoverability condition that each schedule satisfies.)

$S_3$: $r_1$ (X); $r_2$ (Z); $r_1$ (Z); $r_3$ (X); $r_3$ (Y); $w_1$ (X); $c_1$; $w_3$ (Y); $c_3$; $r_2$ (Y); $w_2$ (Z); $w_2$ (Y); $c_2$;

$S_4$: $r_1$ (X); $r_2$ (Z); $r_1$ (Z); $r_3$ (X); $r_3$ (Y); $w_1$ (X); $w_3$ (Y); $r_2$ (Y); $w_2$ (Z); $w_2$ (Y); $c_1$; $c_2$; $c_3$;

$S_5$: $r_1$ (X); $r_2$ (Z); $r_3$ (X); $r_1$ (Z); $r_2$ (Y); $r_3$ (Y); $w_1$ (X); $c_1$; $w_2$ (Z); $w_3$ (Y); $w_2$ (Y); $c_3$; $c_2$;

## Selected Bibliography

The concept of transaction is discussed in Gray (1981). Bernstein, Hadzilacos, and Goodman (1987) focus on concurrency control and recovery techniques in both centralized and distributed database systems; it is an excellent reference. Papadimitriou (1986) offers a more theoretical perspective. A large reference book of more than a thousand pages by Gray and Reuter (1993) offers a more practical perspective of transaction processing concepts and techniques. Elmagarmid (1992) and Bhargava (1989) offer collections of research papers on transaction processing. Transaction support in SQL is described in Date and Darwen (1993). The concepts of serializability are introduced in Gray et al. (1975). View serializability is defined in Yannakakis (1984). Recoverability of schedules is discussed in Hadzilacos (1983, 1988).

# 18

# 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 (see Section 17.5), using **protocols** (that is, sets of rules) that guarantee serializability. One important set of 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 Section 18.1. Locking protocols are used in most commercial DBMSs. Another set of concurrency control protocols use **timestamps.** A timestamp is a unique identifier for each transaction, generated by the system. Concurrency control protocols that use timestamp ordering to ensure serializability are described in Section 18.2. In Section 18.3, we discuss **multiversion** concurrency control protocols that use multiple versions of a data item. In Section 18.4, 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.**

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 in Section 18.5. In Section 18.6, we discuss concurrency control issues that arise when indexes are used to process transactions. Finally, in Section 18.7 we discuss some additional concurrency control issues.

It is sufficient to cover Sections 18.1, 18.5, 18.6, and 18.7, and possibly 18.3.2, if the main emphasis is on introducing the concurrency control techniques that are used most often in practice. The other techniques are mainly of theoretical interest.

# 18.1 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 18.1.1 we discuss the nature and types of locks. Then, in Section 18.1.2, we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 18.1.3 we discuss two problems associated with the use of locks—namely, deadlock and starvation—and show how these problems are handled.

## 18.1.1 Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking concepts gradually, we first discuss binary locks, which are simple but restrictive and so are not used in practice. We then discuss shared/exclusive locks, which provide more general locking capabilities and are used in practical database locking schemes. In Section 18.3.2, we describe a certify lock and 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. 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(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) 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 18.1.

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 18.1, the wait command within the `lock_`

```
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 18.1** Lock and unlock operations for binary locks.

`item`(X) operation is usually implemented by putting the transaction on 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 on the same queue. Hence, the wait command is considered to be outside the `lock_item` operation.

Notice that it is quite 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. 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**

1. This rule may be removed if we modify the lock_item(X) operation in Figure 18.1 so that if the item is currently locked *by the requesting transaction*, the lock is granted.

**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*. 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 three 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. Each record in the lock table will have four fields: <data item name, LOCK, no_of_reads, locking_transaction(s)>. Again, to save space, the system need 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 18.2.[2] As before, each of the three 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 on a waiting queue for the item.

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]
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, as we discuss shortly.

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.

3. This rule may be relaxed to allow a transaction to unlock an item, then lock it again later.

```
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 if 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 18.2** Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

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 be relaxed, 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 of Locks. Sometimes 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 on 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 on 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 18.2 must be changed appropriately. 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 18.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 18.3a 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 18.3c 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.

### 18.1.2 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 expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase. Hence, a `read_lock(X)` operation that downgrades an already held write lock on X can appear only in the shrinking phase.

Transactions $T_1$ and $T_2$ of Figure 18.3a do not follow the two-phase locking protocol. This is 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 18.4. Now, the schedule shown in Figure 18.3(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 18.1.1. This is because $T_1'$ will issue its `write_lock(X)` *before* it

4. This is unrelated to the two-phase commit protocol for recovery in distributed databases (see Chapter 25).

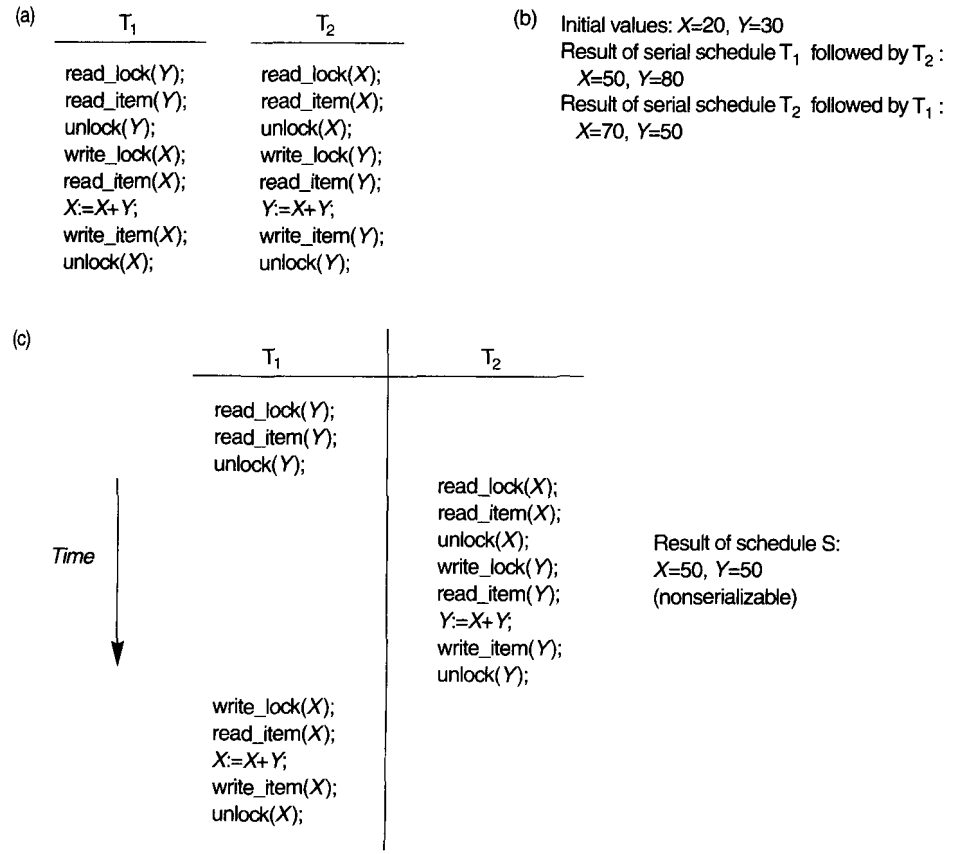


**FIGURE 18.3** Transactions that do not obey two-phase locking. (a) Two transactions $T_1$ and $T_2$. (b) Results of possible serial schedules of $T_1$ and $T_2$. (c) A nonserializable schedule S that uses locks.

| $T_1'$ | $T_2'$ |
|---|---|
| read_lock (Y); | read_lock (X); |
| read_item (Y); | read_item (X); |
| write_lock (X); | write_lock (Y); |
| unlock (Y); | unlock (X); |
| read_item (X); | read_item (Y); |
| X:=X+Y; | Y:=X+Y; |
| write_item (X); | write_item (Y); |
| unlock (X); | unlock (Y); |

**FIGURE 18.4** Transactions $T_1'$ and $T_2'$, which are the same as $T_1$ and $T_2$ of Figure 18.3 but which follow the two-phase locking protocol. Note that they can produce a deadlock.

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.

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 any more. The locking mechanism, by enforcing two-phase locking rules, also enforces serializability.

Two-phase locking may limit the amount of concurrency that can occur in a schedule. This is 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 on; 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.

**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 17.1.2 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 shall see in Section 18.1.3 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 most situations.

In practice, the most popular variation of 2PL is **strict 2PL,** which guarantees strict schedules (see Section 17.4). 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 conservative and rigorous 2PL; the former must lock all its items *before it starts* so once the transaction starts it is in its shrinking phase, whereas 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.

In many cases, 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 on 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 on 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.

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). In addition, the use of locks can cause two additional problems: deadlock and starvation. We discuss these problems and their solutions in the next section.

## 18.1.3 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 on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. A simple example is shown in Figure 18.5a, where the two transactions $T_1'$ and $T_2'$ are deadlocked in a partial schedule; $T_1'$ is on the waiting queue for X, which is locked by $T_2'$, while $T_2'$ is on 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.

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*

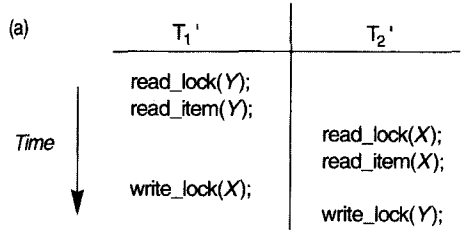


**FIGURE 18.5** Illustrating the deadlock problem. (a) A partial schedule of $T_1'$ and $T_2'$ that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

5. These protocols are not generally used in practice, either because of unrealistic assumptions or because of their possible overhead. Deadlock detection and timeouts (see below) are more practical.

*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. This solution obviously 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) be 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? 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 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 as follows:

- **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.

In wait-die, an older transaction is allowed to wait on 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 on 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 that *may be involved* in a deadlock. It can be shown that these two techniques are deadlock-free, since in wait-die, transactions only wait on younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait on 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. Because 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 rules are 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, 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 on $T_j$, then $b(T_i) < b(T_j)$, since $T_i$ can only wait on $T_j$ at a time when $T_j$ is not blocked. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Deadlock Detection and Timeouts. A second—more practical—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 quite 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. 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. Figure 18.5b shows the wait-for graph for the (partial) schedule shown in Figure 18.5a. 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.

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 or not.

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, giving 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.

# 18.2 Concurrency Control Based on Timestamp Ordering

The use of locks, 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. A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule. In Section 18.2.1 we discuss timestamps and in Section 18.2.2 we discuss how serializability is enforced by ordering transactions based on their timestamps.

## 18.2.1 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.

## 18.2.2 The Timestamp Ordering Algorithm

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the equivalent serial schedule 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 order-

ing, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps. The algorithm must ensure that, for each item accessed by *conflicting operations* in the schedule, the order in which the item is accessed does not violate the serializability 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; this 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; this 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.

Basic Timestamp Ordering. 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. Transaction T issues a `write_item`(X) operation:
   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. Transaction T issues a `read_item`(X) operation:
   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).

Hence, 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, like the 2PL protocol. However, some schedules are possible under each protocol that are not allowed under the other. Hence, neither protocol allows *all possible* serializable schedules. 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. 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 that 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 TS(T) > 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. Hence, 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).

## 18.3 Multiversion Concurrency Control Techniques

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control,** because several versions (values) of an item are maintained. When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea 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 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. However, older versions may have to be

maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a *temporal database* (see Chapter 24), 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.

### 18.3.1 Multiversion Technique Based on Timestamp Ordering

In this method, several versions $X_1$, $X_2$, ..., $X_k$ of each data item X are maintained. For *each version*, the value of version $X_i$ and the following two timestamps 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 two 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 is attempting 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.

### 18.3.2 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 18.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 18.6a. 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 must always have been written by some committed transaction. The second version X′ is created when a transaction T acquires a write lock on the item. 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

(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 18.6** Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

currently holds write locks on before it can commit. 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 18.6b.

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 if upgrading of a read lock to a write lock is allowed, and these must be handled by variations of the techniques discussed in Section 18.1.3.

## 18.4 Validation (Optimistic) Concurrency Control Techniques

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 proposed concurrency control methods use the validation technique. We will describe only one scheme here. In this scheme, updates in the transaction are *not* applied directly to the database items until the transaction reaches its end. 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.

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

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 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 these circumstances, optimistic techniques do not work well. The techniques are called "optimistic" because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.

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. In addition, *start* and *end* times for some of 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 committed transactions or with any other transactions currently in their validation phase. The validation phase for $T_i$ checks that, for *each* such transaction $T_j$ that is either 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 $T_i$, the first condition is checked first for each transaction $T_j$, 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, there is no interference and $T_i$ is validated successfully. If none of these three conditions holds, the validation of transaction $T_i$ fails and it is aborted and restarted later because interference *may* have occurred.

## 18.5 Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assumed that the database was 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 granularity can affect the performance of concurrency control and recovery. In Section 18.5.1, we discuss some of the tradeoffs with regard to choosing the granularity level used for locking, and, in Section 18.5.2, we discuss a multiple granularity locking scheme, where the granularity level (size of the data item) may be changed dynamically.

## 18.5.1 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 shall 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 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 block X in a conflicting lock mode, it is forced to wait. If the data item size was a single record, 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.

## 18.5.2 Multiple Granularity Level Locking

Since the best granularity size depends on the given transaction, it seems appropriate that a database system support multiple levels of granularity, where the granularity level can be different for various mixes of transactions. Figure 18.7 shows a simple granularity hierarchy with a database containing two files, each file containing several pages, and each page containing several records. This can be used to illustrate a **multiple granularity level** 2PL

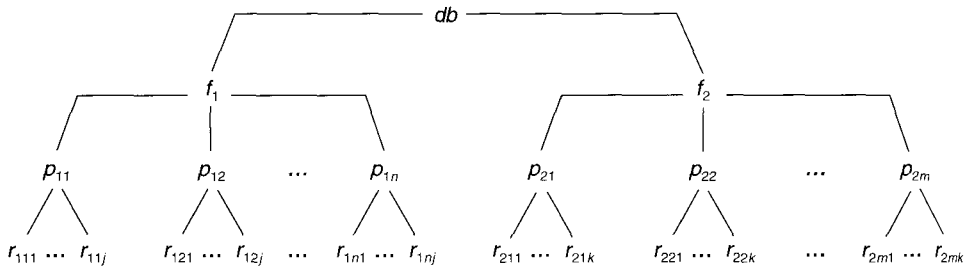


**FIGURE 18.7** A granularity hierarchy for illustrating multiple granularity level locking.

protocol, where a lock can be requested at any level. However, additional types of locks will be needed to efficiently support such a protocol.

Consider the following scenario, with only shared and exclusive lock types, that refers to the example in Figure 18.7. 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 individual record. 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 is quite difficult for the lock manger 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 a shared lock(s) will be requested on some descendant node(s).
2. Intention-exclusive (IX) indicates that an exclusive lock(s) will be requested on some descendant node(s).
3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendant node(s).

The compatibility table of the three intention locks, and the shared and exclusive locks, is shown in Figure 18.8. Besides the introduction of 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 18.8) 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. To illustrate the MGL protocol with the database hierarchy in Figure 18.7, 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 $r_{11j}$ and the entire $f_2$ file.

Figure 18.9 shows a possible serializable schedule for these three transactions. Only the lock operations are shown. The notation <lock_type>(<item>) is used to display the locking operations in the schedule.

| | 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 18.8** Lock compatibility matrix for multiple granularity locking.

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| 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($r_{211}$) | | |
| 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 18.9** Lock operations to illustrate a serializable 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 is incurred by such a protocol when compared to a single level granularity locking approach.

## 18.6 Using Locks for Concurrency Control in Indexes

Two-phase locking can also be applied to indexes (see Chapter 14), 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. This is because searching an index always *starts at the root*, so 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 can be released. Second, 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 a higher level node(s). Then, the locks on the higher level node(s) 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 together 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. In addition, 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.

# 18.7 Other Concurrency Control Issues

In this section, we discuss some other issues relevant to concurrency control. In Section 18.7.1, we discuss problems associated with insertion and deletion of records and the so-called *phantom problem*, which may occur when records are inserted. This problem was described as a potential problem requiring a concurrency control measure in Section 17.6. Then, in Section 18.7.2, 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.

## 18.7.1 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 **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, while 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 18.6. Recall from Chapter 14 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. This is 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* they 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.

### 18.7.2 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.

### 18.7.3 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 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 is released.

## 18.8 SUMMARY

In this chapter we discussed DBMS techniques for concurrency control. We started by discussing lock-based protocols, which are by far the most commonly used in practice. 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, 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, which can occur with locking. In practice, it is common to use timeouts and deadlock detection (wait-for graphs).

We then presented other concurrency control protocols that are not used often in practice but are important for the theoretical alternatives they show for solving this

problem. These include the timestamp ordering protocol, 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 conflict serializability. The strict timestamp ordering protocol was also presented. We then discussed two multiversion protocols, 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. We then presented an example of an optimistic protocol, which is also known as a certification or validation protocol.

We then 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, which is to develop locking protocols for indexes so that indexes do not become a hindrance to concurrent access. Finally, we introduced the phantom problem and problems with interactive transactions, and briefly described the concept of latches and how it differs from locks.

In the next chapter, we give an overview of recovery techniques.

## Review Questions

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

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

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

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

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

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

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

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

18.9. Discuss two multiversion techniques for concurrency control.

18.10. What is a certify lock? What are the advantages and disadvantages of using certify locks?

18.11. 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.

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

18.13. What type of locks are needed for insert and delete operations?
18.14. What is multiple granularity locking? Under what circumstances is it used?
18.15. What are intention locks?
18.16. When are latches used?
18.17. What is a phantom record? Discuss the problem that a phantom record can cause for concurrency control.
18.18. How does index locking resolve the phantom problem?
18.19. What is a predicate lock?

## Exercises

18.20. 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.)
18.21. 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.)
18.22. Prove that strict two-phase locking guarantees strict schedules.
18.23. Prove that the wait-die and wound-wait protocols avoid deadlock and starvation.
18.24. Prove that cautious waiting avoids deadlock.
18.25. Apply the timestamp ordering algorithm to the schedules of Figure 17.8(b) and (c), and determine whether the algorithm will allow the execution of the schedules.
18.26. Repeat Exercise 18.25, but use the multiversion timestamp ordering method.
18.27. Why is two-phase locking not used as a concurrency control method for indexes such as $B^+$-trees?
18.28. The compatibility matrix of Figure 18.8 shows that IS and IX locks are compatible. Explain why this is valid.
18.29. 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 was 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 et al. (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 (1978, 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).

Other recent work on concurrency control includes semantic-based concurrency control (Badrinath and Ramamritham, 1992), transaction models for long running activities (Dayal et al., 1991), and multilevel transaction management (Hasse and Weikum, 1991).

# 19

# Database Recovery Techniques

In this chapter we discuss some of the techniques that can be used for database recovery from failures. We have already discussed the different causes of failure, such as system crashes and transaction errors, in Section 17.1.4. We have also covered many of the concepts that are used by recovery processes, such as the system log and commit points, in Section 17.2.

We start Section 19.1 with an outline of a typical recovery procedures and a categorization of recovery algorithms, and then discuss several recovery concepts, including write-ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of an incomplete or failed transaction. In Section 19.2, we present recovery techniques based on *deferred update*, also known as the NO-UNDO/REDO technique. In Section 19.3, we discuss recovery techniques based on immediate update; these include the UNDO/REDO and UNDO/NO-REDO algorithms. We discuss the technique known as shadowing or shadow paging, which can be categorized as a NO-UNDO/NO-REDO algorithm in Section 19.4. An example of a practical DBMS recovery scheme, called ARIES, is presented in Section 19.5. Recovery in multidatabases is briefly discussed in Section 19.6. Finally, techniques for recovery from catastrophic failure are discussed in Section 19.7.

Our emphasis is on conceptually describing several different approaches to recovery. For descriptions of recovery features in specific systems, the reader should consult the bibliographic notes and the user manuals for those systems. Recovery techniques are often intertwined with the concurrency control mechanisms. Certain recovery techniques are best used with specific concurrency control methods. We will attempt to discuss recovery

concepts independently of concurrency control mechanisms, but we will discuss the circumstances under which a particular recovery mechanism is best used with a certain concurrency control protocol.

# 19.1 Recovery Concepts

## 19.1.1 Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state just before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the **system log,** as we discussed in Section 17.2.2. A typical strategy for recovery may be summarized informally as follows:

1. If there is extensive damage to a wide portion of the database due to catastrophic failure, such as a disk crash, the recovery method restores a past copy of the database that was *backed up* to archival storage (typically tape) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed up* log, up to the time of failure.
2. When the database is not physically damaged but has become inconsistent due to noncatastrophic failures of types 1 through 4 of Section 17.1.4, the strategy is to reverse any changes that caused the inconsistency by *undoing* some operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database, as we shall see. In this case we do not need a complete archival copy of the database. Rather, the entries kept in the online system log are consulted during recovery.

Conceptually, we can distinguish two main techniques for recovery from noncatastrophic transaction failures: (1) deferred update and (2) immediate update. The **deferred update** techniques do not physically update the database on disk until *after* a transaction reaches its commit point; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace (or buffers). During commit, the updates are first recorded persistently in the log and then written to the database. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm.** We discuss this technique in Section 19.2.

In the **immediate update** techniques, the database may be updated by some operations of a transaction *before* the transaction reaches its commit point. However, these operations are typically recorded in the log *on disk* by force writing *before* they are applied to the database, making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its

operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* may be required during recovery. This technique, known as the UNDO/REDO **algorithm,** requires both operations, and is used most often in practice. A variation of the algorithm where all updates are recorded in the database before a transaction commits requires *undo* only, so it is known as the UNDO/NO-REDO **algorithm.** We discuss these techniques in Section 19.3.

## 19.1.2 Caching (Buffering) of Disk Blocks

The recovery process is often closely intertwined with operating system functions—in particular, the buffering and caching of disk pages in main memory. Typically, one or more disk pages that include the data items to be updated are **cached** into main memory buffers and then updated in memory before being written back to disk. The caching of disk pages is traditionally an operating system function, but because of its importance to the efficiency of recovery procedures, it is handled by the DBMS by calling low-level operating systems routines.

In general, it is convenient to consider recovery in terms of the database disk pages (blocks). Typically a collection of in-memory buffers, called the **DBMS cache,** is kept under the control of the DBMS for the purpose of holding these buffers. A **directory** for the cache is used to keep track of which database items are in the buffers.[1] This can be a table of `<disk page address, buffer location>` entries. When the DBMS requests action on some item, it first checks the cache directory to determine whether the disk page containing the item is in the cache. If it is not, then the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to **replace** (or **flush**) some of the cache buffers to make space available for the new item. Some page-replacement strategy from operating systems, such as least recently used (LRU) or first-in-first-out (FIFO), can be used to select the buffers for replacement.

Associated with each buffer in the cache is a **dirty bit,** which can be included in the directory entry, to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, the cache directory is updated with the new disk page address, and the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit is 1.* Another bit, called the **pin-unpin** bit, is also needed—a page in the cache is **pinned** (bit value 1 (one)) if it cannot be written back to disk as yet.

Two main strategies can be employed when flushing a modified buffer back to disk. The first strategy, known as **in-place updating,** writes the buffer back to the *same original disk location,* thus overwriting the old value of any changed data items on disk.[2] Hence, a single copy of each database disk block is maintained. The second strategy, known as **shadowing,** writes an updated buffer at a different disk location, so multiple versions of

1. This is somewhat similar to the concept of *page tables* used by the operating system.

2. In-place updating is used in most systems in practice.

data items can be maintained. In general, the old value of the data item before updating is called the **before image (BFIM),** and the new value after updating is called the **after image (AFIM).** In shadowing, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering. We briefly discuss recovery based on shadowing in Section 19.4.

### 19.1.3 Write-Ahead Logging, Steal/No-Steal, and Force/No-Force

When in-place updating is used, it is necessary to use a log for recovery (see Section 17.2.2). In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as **write-ahead logging.** Before we can describe a protocol for write-ahead logging, we need to distinguish between two types of log entry information included for a write command: (1) the information needed for UNDO and (2) that needed for REDO. A **REDO-type log entry** includes the **new value** (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database to its AFIM). The **UNDO-type log entries** include the **old value** (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both types of log entries are combined. In addition, when cascading rollback is possible, `read_item` entries in the log are considered to be UNDO-type entries (see Section 19.1.5).

As mentioned, the DBMS cache holds the cached database disk blocks, which include not only *data blocks* but also *index blocks* and *log blocks* from the disk. When a log record is written, it is stored in the current log block in the DBMS cache. The log is simply a sequential (append-only) disk file and the DBMS cache may contain several log blocks (for example, the last $n$ log blocks) that will be written to disk. When an update to a data block—stored in the DBMS cache—is made, an associated log record is written to the last log block in the DBMS cache. With the write-ahead logging approach, the log blocks that contain the associated log records for a particular data block update must first be written to disk before the data block itself can be written back to disk.

Standard DBMS recovery terminology includes the terms **steal/no-steal** and **force/no-force,** which specify when a page from the database can be written to disk from the cache:

1. If a cache page updated by a transaction *cannot* be written to disk before the transaction commits, this is called a **no-steal approach.** The pin-unpin bit indicates if a page cannot be written back to disk. Otherwise, if the protocol allows writing an updated buffer *before* the transaction commits, it is called **steal.** Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed.
2. If all pages updated by a transaction are immediately written to disk when the transaction commits, this is called a **force approach.** Otherwise, it is called **no-force.**

The deferred update recovery scheme in Section 19.2 follows a *no-steal* approach. However, typical database systems employ a *steal/no-force* strategy. The advantage of steal is that it avoids the need for a very large buffer space to store all updated pages in memory. The advantage of no-force is that an updated page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to read that page again from disk. This may provide a substantial saving in the number of I/O operations when a specific page is updated heavily by multiple transactions.

To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the logon disk before changes are applied to the database. For example, consider the following **write-ahead logging (WAL)** protocol for a recovery algorithm that requires both UNDO and REDO:

1. The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log records for the updating transaction—up to this point in time—have been force-written to disk.
2. The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force-written to disk.

To facilitate the recovery process, the DBMS recovery subsystem may need to maintain a number of lists related to the transactions being processed in the system. These include a list for **active transactions** that have started but not committed as yet, and it may also include lists of all **committed** and **aborted transactions** since the last checkpoint (see next section). Maintaining these lists makes the recovery process more efficient.

### 19.1.4 Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint.**[3] A `[checkpoint]` record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their `[commit,T]` entries in the log before a `[checkpoint]` entry do not need to have their WRITE operations *redone* in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing.

The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time—say, every $m$ minutes—or in the number $t$ of committed transactions since the last checkpoint, where the values of $m$ or $t$ are system parameters. Taking a checkpoint consists of the following actions:

1. Suspend execution of transactions temporarily.
2. Force-write all main memory buffers that have been modified to disk.
3. Write a `[checkpoint]` record to the log, and force-write the log to disk.
4. Resume executing transactions.

3. The term *checkpoint* has been used to describe more restrictive situations in some systems, such as DB2. It has also been used in the literature to describe entirely different concepts.

As a consequence of step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction. This can facilitate undoing transaction operations in the event that a transaction must be rolled back.

The time needed to force-write all modified memory buffers may delay transaction processing because of step 1. To reduce this delay, it is common to use a technique called **fuzzy checkpointing** in practice. In this technique, the system can resume transaction processing after the `[checkpoint]` record is written to the log without having to wait for step 2 to finish. However, until step 2 is completed, the previous `[checkpoint]` record should remain valid. To accomplish this, the system maintains a pointer to the valid checkpoint, which continues to point to the previous `[checkpoint]` record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

## 19.1.5 Transaction Rollback

If a transaction fails for whatever reason after updating the database, it may be necessary to **roll back** the transaction. If any data item values have been changed by the transaction and written to the database, they must be restored to their previous values (BFIMs). The undo-type log entries are used to restore the old values of data items that must be rolled back.

If a transaction $T$ is rolled back, any transaction $S$ that has, in the interim, read the value of some data item $X$ written by $T$ must also be rolled back. Similarly, once $S$ is rolled back, any transaction $R$ that has read the value of some data item $Y$ written by $S$ must also be rolled back; and so on. This phenomenon is called **cascading rollback,** and can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules (see Section 17.4.2). Cascading rollback, understandably, can be quite complex and time-consuming. That is why almost all recovery mechanisms are designed such that cascading rollback *is never required.*

Figure 19.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 19.1a. Figure 19.1b shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of data items $A$, $B$, $C$, and $D$, which are used by the transactions, are shown to the right of the system log entries. We assume that the original item values, shown in the first line, are $A = 30$, $B = 15$, $C = 40$, and $D = 20$. At the point of system failure, transaction $T_3$ has not reached its conclusion and must be rolled back. The `WRITE` operations of $T_3$, marked by a single * in Figure 19.1b, are the $T_3$ operations that are undone during transaction rollback. Figure 19.1c graphically shows the operations of the different transactions along the time axis.

We must now check for cascading rollback. From Figure 19.1c we see that transaction $T_2$ reads the value of item $B$ that was written by transaction $T_3$; this can also be determined by examining the log. Because $T_3$ is rolled back, $T_2$ must now be rolled back, too. The `WRITE` operations of $T_2$, marked by ** in the log, are the ones that are undone. Note that only `write_item` operations need to be undone during transaction rollback; `read_item` operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.

(a)

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read_item($A$) | read_item($B$) | read_item($C$) |
| read_item($D$) | write_item($B$) | write_item($B$) |
| write_item($D$) | read_item($D$) | read_item($A$) |
| | write_item($D$) | write_item($A$) |

(b)

| | | $A$ | $B$ | $C$ | $D$ |
|---|---|---|---|---|---|
| | | 30 | 15 | 40 | 20 |
| | [start_transaction, $T_3$] | | | | |
| | [read_item, $T_3$, $C$] | | | | |
| * | [write_item, $T_3$, $B$, 15, 12] | | 12 | | |
| | [start_transaction, $T_2$] | | | | |
| | [read_item, $T_2$, $B$] | | | | |
| ** | [write_item, $T_2$, $B$, 12, 18] | | 18 | | |
| | [start_transaction, $T_1$] | | | | |
| | [read_item, $T_1$, $A$] | | | | |
| | [read_item, $T_1$, $D$] | | | | |
| | [write_item, $T_1$, $D$, 20, 25] | | | | 25 |
| | [read_item, $T_2$, $D$] | | | | |
| ** | [write_item, $T_2$, $D$, 25, 26] | | | | 26 |
| | [read_item, $T_3$, $A$] | | | | |
| | ← system crash | | | | |

*$T_3$ is rolled back because it did not reach its commit point.
**$T_2$ is rolled back because it reads the value of item $B$ written by $T_3$.

(c)

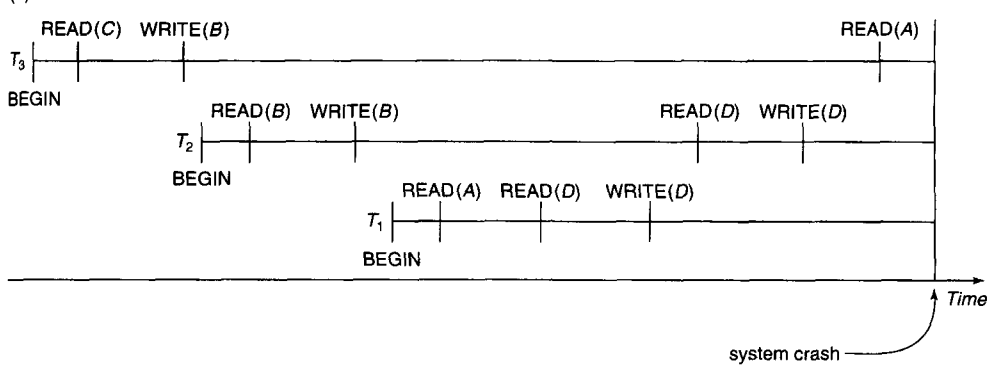


**FIGURE 19.1** Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

In practice, cascading rollback of transactions is *never* required because practical recovery methods guarantee cascadeless or strict schedules. Hence, there is also no need to record any `read_item` operations in the log, because these are needed only for determining cascading rollback.

## 19.2 Recovery Techniques Based on Deferred Update

The idea behind deferred update techniques is to defer or postpone any actual updates to the database until the transaction completes its execution successfully and reaches its commit point.[4] During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. If a transaction fails before reaching its commit point, there is no need to undo any operations, because the transaction has not affected the database on disk in any way. Although this may simplify recovery, it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held in the cache buffers until the commit point.

We can state a typical deferred update protocol as follows:

1. A transaction cannot change the database on disk until it reaches its commit point.
2. A transaction does not reach its commit point until all its update operations are recorded in the log *and* the log is force-written to disk.

Notice that step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. Hence, this is known as the **NO-UNDO/REDO recovery algorithm.** REDO is needed in case the system fails after a transaction commits but before all its changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries.

Usually, the method of recovery from failure is closely related to the concurrency control method in multiuser systems. First we discuss recovery in single-user systems, where no concurrency control is needed, so that we can understand the recovery process independently of any concurrency control method. We then discuss how concurrency control may affect the recovery process.

### 19.2.1 Recovery Using Deferred Update in a Single-User Environment

In such an environment, the recovery algorithm can be rather simple. The algorithm RDU_S (Recovery using Deferred Update in a Single-user environment) uses a REDO procedure, given subsequently, for redoing certain `write_item` operations; it works as follows:

PROCEDURE RDU_S: Use two lists of transactions: the committed transactions since the last checkpoint, and the active transactions (at most one transaction will fall in this category, because the system is single-user). Apply the REDO operation to all the

4. Hence deferred update can generally be characterized as a *no-steal approach*.

WRITE_ITEM operations of the committed transactions from the log in the order in which they were written to the log. Restart the active transactions.

The REDO procedure is defined as follows:

**REDO(WRITE_OP)**: Redoing a `write_item` operation WRITE_OP consists of examining its log entry `[write_item,T,X,new_value]` and setting the value of item X in the database to `new_value`, which is the after image (AFIM).

The REDO operation is required to be **idempotent**—that is, executing it over and over is equivalent to executing it just once. In fact, the whole recovery process should be idempotent. This is so because, if the system were to fail during the recovery process, the next recovery attempt might REDO certain `write_item` operations that had already been redone during the first recovery process. The result of recovery from a system crash *during recovery* should be the same as the result of recovering *when there is no crash during recovery!*

Notice that the only transaction in the active list will have had no effect on the database because of the deferred update protocol, and it is ignored completely by the recovery process because none of its operations were reflected in the database on disk. However, this transaction must now be restarted, either automatically by the recovery process or manually by the user.

Figure 19.2 shows an example of recovery in a single-user environment, where the first failure occurs during execution of transaction $T_2$, as shown in Figure 19.2b. The recovery process will redo the `[write_item,T1,D,20]` entry in the log by resetting the value of item $D$ to 20 (its new value). The `[write,T2, ...]` entries in the log are ignored by the recovery process because $T_2$ is not committed. If a second failure occurs during recovery from the first failure, the same recovery process is repeated from start to finish, with identical results.

(a)

| $T_1$ | $T_2$ |
|---|---|
| read_item($A$) | read_item($B$) |
| read_item($D$) | write_item($B$) |
| write_item($D$) | read_item($D$) |
| | write_item($D$) |

(b)

[start_transaction, $T_1$]
[write_item, $T_1$, $D$, 20]
[commit, $T_1$]
[start_transaction, $T_2$]
[write_item, $T_2$, $B$, 10]
[write_item, $T_2$, $D$, 25] ← system crash

The [write_item,...] operations of $T_1$ are redone.
$T_2$ log entries are ignored by the recovery process.

**FIGURE 19.2** An example of recovery using deferred update in a single-user environment. (a) The READ and WRITE operations of two transactions. (b) The system log at the point of crash.

### 19.2.2 Deferred Update with Concurrent Execution in a Multiuser Environment

For multiuser systems with concurrency control, the recovery process may be more complex, depending on the protocols used for concurrency control. In many cases, the concurrency control and recovery processes are interrelated. In general, the greater the degree of concurrency we wish to achieve, the more time consuming the task of recovery becomes.

Consider a system in which concurrency control uses strict two-phase locking, so the locks on items remain in effect *until the transaction reaches its commit point*. After that, the locks can be released. This ensures strict and serializable schedules. Assuming that `[checkpoint]` entries are included in the log, a possible recovery algorithm for this case, which we call RDU_M (Recovery using Deferred Update in a Multiuser environment), is given next. This procedure uses the REDO procedure defined earlier.

> PROCEDURE RDU_M (WITH CHECKPOINTS): Use two lists of transactions maintained by the system: the committed transactions $T$ since the last checkpoint (**commit list**), and the active transactions $T'$ (**active list**). REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log*. The transactions that are active and did not commit are effectively canceled and must be resubmitted.

Figure 19.3 shows a possible schedule of executing transactions. When the checkpoint was taken at time $t_1$, transaction $T_1$ had committed, whereas transactions $T_3$ and $T_4$ had not. Before the system crash at time $t_2$, $T_3$ and $T_2$ were committed but not $T_4$ and $T_5$. According to the RDU_M method, there is no need to redo the `write_item` operations of transaction $T_1$—or any transactions committed before the last checkpoint time $t_1$. `Write_item` operations of $T_2$ and $T_3$ must be redone, however, because both transactions reached

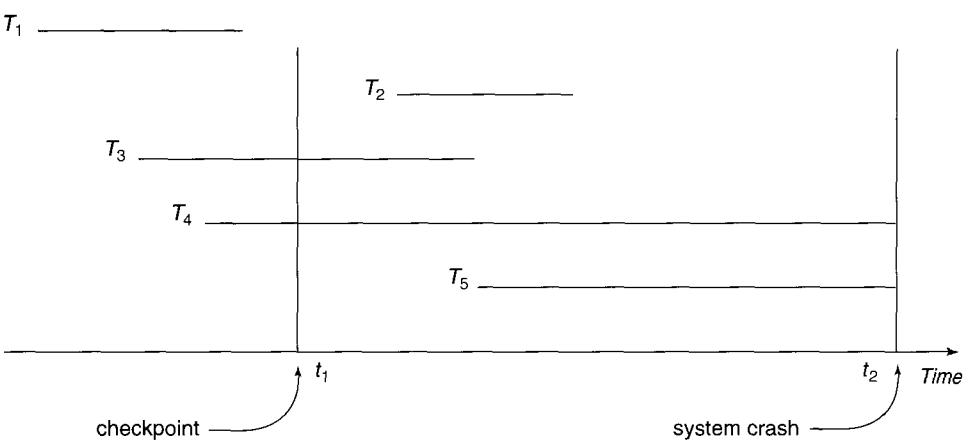


**FIGURE 19.3** An example of recovery in a multiuser environment.

their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions $T_4$ and $T_5$ are ignored: They are effectively canceled or rolled back because none of their `write_item` operations were recorded in the database under the deferred update protocol. We will refer to Figure 19.3 later to illustrate other recovery protocols.

We can make the NO-UNDO/REDO recovery algorithm *more efficient* by noting that, if a data item X has been updated—as indicated in the log entries—more than once by committed transactions since the last checkpoint, it is only necessary to REDO *the last update of* X from the log during recovery. The other updates would be overwritten by this last REDO in any case. In this case, we start from *the end of the log*; then, whenever an item is redone, it is added to a list of redone items. Before REDO is applied to an item, the list is checked; if the item appears on the list, it is not redone again, since its last value has already been recovered.

If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk. A drawback of the method described here is that it limits the concurrent execution of transactions because *all items remain locked until the transaction reaches its commit point*. In addition, it may require excessive buffer space to hold all updated items until the transactions commit. The method's main benefit is that transaction operations *never need to be undone*, for two reasons:

1. A transaction does not record any changes in the database on disk until after it reaches its commit point—that is, until it completes its execution successfully. Hence, a transaction is never rolled back because of failure during transaction execution.
2. A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.

Figure 19.4 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

### 19.2.3 Transaction Actions That Do Not Affect the Database

In general, a transaction will have actions that do *not* affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action based on these reports that affects the database. Hence, such reports should be generated only *after the transaction reaches its commit point*. A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

(a)

| $T_1$ | $T_2$ | $T_3$ | $T_4$ |
|---|---|---|---|
| read_item($A$) | read_item($B$) | read_item($A$) | read_item($B$) |
| read_item($D$) | write_item($B$) | write_item($A$) | write_item($B$) |
| write_item($D$) | read_item($D$) | read_item($C$) | read_item($A$) |
| | write_item($D$) | write_item($C$) | write_item($A$) |

(b)

[start_transaction, $T_1$]
[write_item, $T_1$, $D$, 20]
[commit, $T_1$]
[checkpoint]
[start_transaction, $T_4$]
[write_item, $T_4$, $B$, 15]
[write_item, $T_4$, $A$, 20]
[commit, $T_4$]
[start_transaction, $T_2$]
[write_item, $T_2$, $B$, 12]
[start_transaction, $T_3$]
[write_item, $T_3$, $A$, 30]
[write_item, $T_2$, $D$, 25] ← system crash

$T_2$ and $T_3$ are ignored because they did not reach their commit points.
$T_4$ is redone because its commit point is after the last system checkpoint.

**FIGURE 19.4** An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

# 19.3 Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database can be updated "immediately," without any need to wait for the transaction to reach its commit point. In these techniques, however, an update operation must still be recorded in the log (on disk) *before* it is applied to the database—using the write-ahead logging protocol—so that we can recover in case of failure.

Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction*. This is accomplished by rolling back the transaction and undoing the effect of the transaction's `write_item` operations. Theoretically, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits*, there is never a need to REDO any operations of committed transactions. This is called the **UNDO/NO-REDO recovery algorithm.** On the other hand, if the

transaction is allowed to commit before all its changes are written to the database, we have the most general case, known as the **UNDO/REDO recovery algorithm.** This is also the most complex technique. Next, we discuss two examples of UNDO/REDO algorithms and leave it as an exercise for the reader to develop the UNDO/NO-REDO variation. In Section 19.5, we describe a more practical approach known as the ARIES recovery technique.

### 19.3.1 UNDO/REDO Recovery Based on Immediate Update in a Single-User Environment

In a single-user system, if a failure occurs, the executing (active) transaction at the time of failure may have recorded some changes in the database. The effect of all such operations must be undone. The recovery algorithm RIU_S (Recovery using Immediate Update in a Single-user environment) uses the REDO procedure defined earlier, as well as the UNDO procedure defined below.

PROCEDURE RIU_S

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions (at most one transaction will fall in this category, because the system is single-user).
2. Undo all the `write_item` operations of the *active* transaction from the log, using the UNDO procedure described below.
3. Redo the `write_item` operations of the *committed* transactions from the log, in the order in which they were written in the log, using the REDO procedure described earlier.

The UNDO procedure is defined as follows:

UNDO(WRITE_OP): Undoing a `write_item` operation `write_op` consists of examining its log entry `[write_item,T,X,old_value,new_value]` and setting the value of item X in the database to `old_value` which is the before image (BFIM). Undoing a number of `write_item` operations from one or more transactions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

### 19.3.2 UNDO/REDO Recovery Based on Immediate Update with Concurrent Execution

When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure RIU_M (Recovery using Immediate Updates for a Multiuser environment) outlines a recovery algorithm for concurrent transactions with immediate update. Assume that the log includes checkpoints and that the concurrency control protocol produces *strict schedules*—as, for example, the strict two-phase locking protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that last wrote the item has committed (or aborted and rolled back). However, deadlocks can occur in strict two-phase locking, thus

requiring abort and UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM).

PROCEDURE RIU_M

1. Use two lists of transactions maintained by the system: the committed transactions since the last checkpoint and the active transactions.
2. Undo all the `write_item` operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.
3. Redo all the `write_item` operations of the *committed* transactions from the log, in the order in which they were written into the log.

As we discussed in Section 19.2.2, step 3 is more efficiently done by starting from the *end of the log* and redoing only *the last update of each item* X. Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2.

## 19.4 SHADOW PAGING

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say, *n*—for recovery purposes. A **directory** with *n* entries[5] is constructed, where the $i^{th}$ entry points to the $i^{th}$ database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins executing, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory.** The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never* modified. When a `write_item` operation is performed, a new copy of the modified database page is created, but the old copy of that page is *not overwritten*. Instead, the new page is written elsewhere—on some previously unused disk block. The current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure 19.5 illustrates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory, and the new version by the current directory.

To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the database before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state

5. The directory is similar to the **page table** maintained by the operating system for each process.

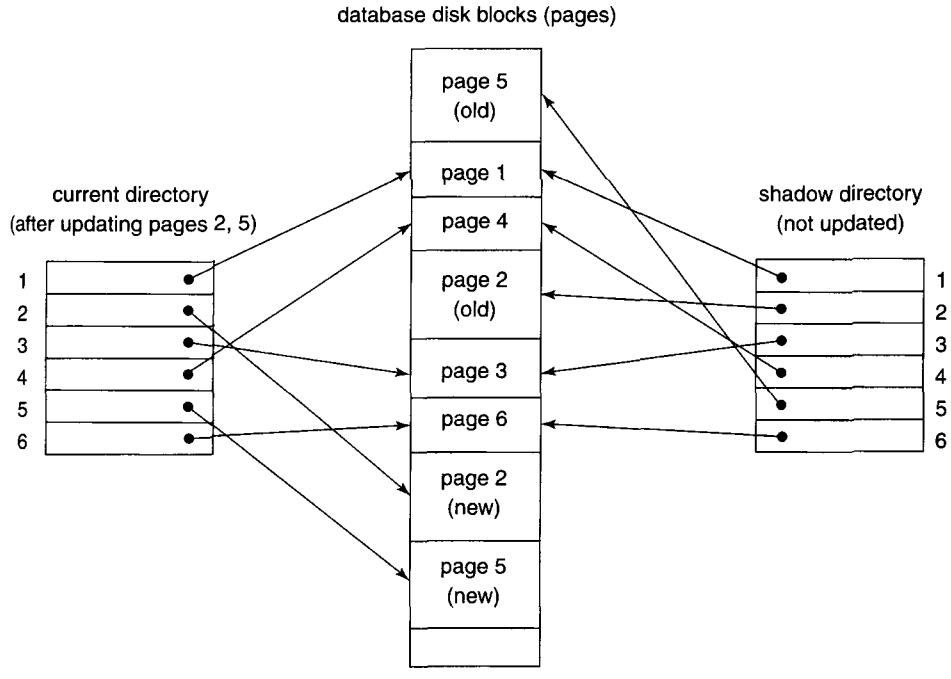


**FIGURE 19.5** An example of shadow paging.

prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corresponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it difficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writing shadow directories to disk as transactions commit is significant. A further complication is how to handle **garbage collection** when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between current and shadow directories must be implemented as an atomic operation.

## 19.5 THE ARIES RECOVERY ALGORITHM

We now describe the ARIES algorithm as an example of a recovery algorithm used in database systems. ARIES uses a steal/no-force approach for writing, and it is based on three concepts: (1) write-ahead logging, (2) repeating history during redo, and (3) logging

changes during undo. We already discussed write-ahead logging in Section 19.1.3. The second concept, **repeating history,** means that ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state *when the crash occurred.* Transactions that were uncommitted at the time of the crash (active transactions) are undone. The third concept, **logging during undo,** will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery procedure consists of three main steps: (1) analysis, (2) REDO and (3) UNDO. The **analysis step** identifies the dirty (updated) pages in the buffer,[6] and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The **REDO phase** actually reapplies updates from the log to the database. Generally, the REDO operation is applied to only committed transactions. However, in ARIES, this is not the case. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. In addition, information stored by ARIES and *in the data pages will allow ARIES to determine whether the operation to be redone has* actually been applied to the database and hence need not be reapplied. Thus *only the necessary REDO operations* are applied during recovery. Finally, during the **UNDO phase,** the log is scanned backwards and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. In addition, checkpointing is used. These two tables are maintained by the transaction manager and written to the log during checkpointing.

In ARIES, every log record has an associated **log sequence number (LSN)** that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a *specific change* (action) of some transaction. In addition, each data page will store the LSN of the *latest log record corresponding to a change for that page.* A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end). The need for including the first three actions in the log has been discussed, but the last two need some explanation. When an update is undone, a *compensation log record* is written in the log. When a transaction ends, whether by committing or aborting, an *end log record* is written.

Common fields in all log records include: (1) the previous LSN for that transaction, (2) the transaction ID, and (3) the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write) action, additional fields in the log record include: (4) the page ID for the page that includes the item, (5) the length of the updated item, (6) its offset from the beginning of the page, (7) the before image of the item, and (8) its after image.

6. The actual buffers may be lost during a crash, since they are in main memory. Additional tables stored in the log during checkpointing (Dirty Page Table, Transaction Table) allow ARIES to identify this information (see next page).

Besides the log, two tables are needed for efficient recovery: the **Transaction Table** and the **Dirty Page Table,** which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for *each active transaction,* with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The Dirty Page Table contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

**Checkpointing** in ARIES consists of the following: (1) writing a `begin_checkpoint` record to the log, (2) writing an `end_checkpoint` record to the log, and (3) writing *the LSN of* the `begin_checkpoint` record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the `end_checkpoint` record, the contents of both the Transaction Table and Dirty Page Table are appended to the end of the log. To reduce the cost, **fuzzy checkpointing** is used so that the DBMS can continue to execute transactions during checkpointing (see Section 19.1.4). In addition, the contents of the DBMS cache do not have to be flushed to disk during checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery. Notice that if a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which is used for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The **analysis phase** starts at the `begin_checkpoint` record and proceeds to the end of the log. When the `end_checkpoint` record is encountered, the Transaction Table and Dirty Page Table are accessed (recall that these tables were written in the log during checkpointing). During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction $T$ in the Transaction Table, then the entry for $T$ is deleted from that table. If some other type of log record is encountered for a transaction $T'$, then an entry for $T'$ is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page $P$, then an entry would be made for page $P$ (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The **REDO phase** follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk.* It can determine this by finding the smallest LSN, M, of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to a LSN $< M$, for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN = M and scans forward to the end of the log. For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page $P$ that is not in the Dirty Page Table, then this change is already on disk and need not be reapplied. Or, if a change recorded in the log (with LSN = $N$, say) pertains to page $P$ and the Dirty Page Table contains an entry for P

with LSN greater than $N$, then the change is already present. If neither of these two conditions hold, page $P$ is read from disk and the LSN stored on that page, LSN($P$), is compared with $N$. If $N <$ LSN($P$), then the change has been applied and the page need not be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the **undo_set**—has been identified in the Transaction Table during the analysis phase. Now, the **UNDO phase** proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the undo_set has been undone. When this is completed, the recovery process is finished and normal processing can begin again.

Consider the recovery example shown in Figure 19.6. There are three transactions: $T_1$, $T_2$, and $T_3$. $T_1$ updates page C, $T_2$ updates pages B and C, and $T_3$ updates page A. Figure 19.6 (a) shows the partial contents of the log and (b) shows the contents of the Transaction Table and Dirty Page Table. Now, suppose that a crash occurs at this point.

(a)

| LSN | LAST_LSN | TRAN_ID | TYPE | PAGE_ID | OTHER INFORMATION |
|---|---|---|---|---|---|
| 1 | 0 | T1 | update | C | ... |
| 2 | 0 | T2 | update | B | ... |
| 3 | 1 | T1 | commit | | ... |
| 4 | begin checkpoint | | | | |
| 5 | end checkpoint | | | | |
| 6 | 0 | T3 | update | A | ... |
| 7 | 2 | T2 | update | C | ... |
| 8 | 7 | T2 | commit | | ... |

(b)

| TRANSACTION TABLE | | | DIRTY PAGE TABLE | |
|---|---|---|---|---|
| TRANSACTION ID | LAST LSN | STATUS | PAGE ID | LSN |
| T1 | 3 | commit | C | 1 |
| T2 | 2 | in progress | B | 2 |

(c)

| TRANSACTION TABLE | | | DIRTY PAGE TABLE | |
|---|---|---|---|---|
| TRANSACTION ID | LAST LSN | STATUS | PAGE ID | LSN |
| T1 | 3 | commit | C | 1 |
| T2 | 8 | commit | B | 2 |
| T3 | 6 | in progress | A | 6 |

**FIGURE 19.6** An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

Since a checkpoint has occurred, the address of the associated `begin_checkpoint` record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end. The `end_checkpoint` record would contain the Transaction Table and Dirty Page table in Figure 19.6b, and the analysis phase will further reconstruct these tables. When the analysis phase encounters log record 6, a new entry for transaction $T_3$ is made in the Transaction Table and a new entry for page A is made in the Dirty Page table. After log record 8 is analyzed, the status of transaction $T_2$ is changed to committed in the Transaction Table. Figure 19.6c shows the two tables after the analysis phase.

For the REDO phase, the smallest LSN in the Dirty Page table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. The LSNs {1, 2, 6, 7} corresponding to the updates for pages C, B, A, and C, respectively, are not less than the LSNs of those pages (as shown in the Dirty Page table). So those data pages will be read again and the updates reapplied from the log (assuming the actual LSNs stored on those data pages are less then the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure 19.6c), UNDO is applied only to the active transaction $T_3$. The UNDO phase starts at log entry 6 (the last update for $T_3$) and proceeds backward in the log. The backward chain of updates for transaction $T_3$ (only log record 6 in this example) is followed and undone.

## 19.6 Recovery in Multidatabase Systems

So far, we have implicitly assumed that a transaction accesses a single database. In some cases a single transaction, called a **multidatabase transaction,** may require access to multiple databases. These databases may even be stored on different types of DBMSs; for example, some DBMSs may be relational, whereas others are object-oriented, hierarchical, or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction may have its own recovery technique and transaction manager separate from those of the other DBMSs. This situation is somewhat similar to the case of a distributed database management system (see Chapter 25), where parts of the database reside at different sites that are connected by a communication network.

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager,** or **coordinator,** is needed to maintain information needed for recovery, in addition to the local recovery managers and the information they maintain (log, tables). The coordinator usually follows a protocol called the **two-phase commit protocol,** whose two phases can be stated as follows:

- **Phase 1:** When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message "prepare for commit" to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery to disk and then send a "ready to commit" or "OK" signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a "cannot commit" or "not OK" signal to the coordinator. If the coordinator does not

receive a reply from a database within a certain time out interval, it assumes a "not OK" response.

- **Phase 2:** If *all* participating databases reply "OK," and the coordinator's vote is also "OK," the transaction is successful, and the coordinator sends a "commit" signal for the transaction to the participating databases. Because all the local effects of the transaction and information needed for local recovery have been recorded in the logs of the participating databases, recovery from failure is now possible. Each participating database completes transaction commit by writing a [commit] entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases or the coordinator have a "not OK" response, the transaction has failed, and the coordinator sends a message to "roll back" or UNDO the local effect of the transaction to each participating database. This is done by undoing the transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating databases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before Phase 1 usually requires the transaction to be rolled back, whereas a failure during Phase 2 means that a successful transaction can recover and commit.

## 19.7 Database Backup and Recovery from Catastrophic Failures

So far, all the techniques we have discussed apply to noncatastrophic failures. A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure. Similarly, the shadow directory must be stored on disk to allow recovery when shadow paging is used. The recovery techniques we have discussed use the entries in the system log or the shadow directory to recover from failure by bringing the database back to a consistent state.

The recovery manager of a DBMS must also be equipped to handle more catastrophic failures such as disk crashes. The main technique used to handle such crashes is that of **database backup.** The whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. Thus users do not lose all transactions they have performed since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up to tape can have their effect on the database redone. A new log is started

after each database backup. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

## 19.8 SUMMARY

In this chapter we discussed the techniques for recovery from transaction failures. The main goal of recovery is to ensure the atomicity property of a transaction. If a transaction fails before completing its execution, the recovery mechanism has to make sure that the transaction has no lasting effects on the database. We first gave an informal outline for a recovery process and then discussed system concepts for recovery. These included a discussion of caching, in-place updating versus shadowing, before and after images of a data item, UNDO versus REDO recovery operations, steal/no-steal and force/no-force policies, system checkpointing, and the write-ahead logging protocol.

Next we discussed two different approaches to recovery: deferred update and immediate update. Deferred update techniques postpone any actual updating of the database on disk until a transaction reaches its commit point. The transaction force-writes the log to disk before recording the updates in the database. This approach, when used with certain concurrency control methods, is designed never to require transaction rollback, and recovery simply consists of redoing the operations of transactions committed after the last checkpoint from the log. The disadvantage is that too much buffer space may be needed, since updates are kept in the buffers and are not applied to disk until a transaction commits. Deferred update can lead to a recovery algorithm known as NO-UNDO/REDO. Immediate update techniques may apply changes to the database on disk before the transaction reaches a successful conclusion. Any changes applied to the database must first be recorded in the log and force-written to disk so that these operations can be undone if necessary. We also gave an overview of a recovery algorithm for immediate update known as UNDO/REDO. Another algorithm, known as UNDO/NO-REDO, can also be developed for immediate update if all transaction actions are recorded in the database before commit.

We discussed the shadow paging technique for recovery, which keeps track of old database pages by using a shadow directory. This technique, which is classified as NO-UNDO/NO-REDO, does not require a log in single-user systems but still needs the log for multiuser systems. We also presented ARIES, a specific recovery scheme used in some of IBM's relational database products. We then discussed the two-phase commit protocol, which is used for recovery from failures involving multidatabase transactions. Finally, we discussed recovery from catastrophic failures, which is typically done by backing up the database and the log to tape. The log can be backed up more frequently than the database, and the backup log can be used to redo operations starting from the last database backup.

## Review Questions

19.1. Discuss the different types of transaction failures. What is meant by catastrophic failure?

19.2. Discuss the actions taken by the `read_item` and `write_item` operations on a database.

19.3. (*Review from Chapter 17*) What is the system log used for? What are the typical kinds of entries in a system log? What are checkpoints, and why are they important? What are transaction commit points, and why are they important?

19.4. How are buffering and caching techniques used by the recovery subsystem?

19.5. What are the before image (BFIM) and after image (AFIM) of a data item? What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?

19.6. What are UNDO-type and REDO-type log entries?

19.7. Describe the write-ahead logging protocol.

19.8. Identify three typical lists of transactions that are maintained by the recovery subsystem.

19.9. What is meant by transaction rollback? What is meant by cascading rollback? Why do practical recovery methods use protocols that do not permit cascading rollback? Which recovery techniques do not require any rollback?

19.10. Discuss the UNDO and REDO operations and the recovery techniques that use each.

19.11. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?

19.12. How can recovery handle transaction operations that do not affect the database, such as the printing of reports by a transaction?

19.13. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?

19.14. What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update? Develop the outline for an UNDO/NO-REDO algorithm.

19.15. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?

19.16. Describe the three phases of the ARIES recovery method.

19.17. What are log sequence numbers (LSNs) in ARIES? How are they used? What information does the Dirty Page Table and Transaction Table contain? Describe how fuzzy checkpointing is used in ARIES.

19.18. What do the terms steal/no-steal and force/no-force mean with regard to buffer management for transaction processing.

19.19. Describe the two-phase commit protocol for multidatabase transactions.

19.20. Discuss how recovery from catastrophic failures is handled.

## Exercises

19.21. Suppose that the system crashes before the `[read_item,T3,A]` entry is written to the log in Figure 19.1b. Will that make any difference in the recovery process?

19.22. Suppose that the system crashes before the `[write_item,T2,D,25,26]` entry is written to the log in Figure 19.1b. Will that make any difference in the recovery process?

19.23. Figure 19.7 shows the log corresponding to a particular schedule at the point of a system crash for four transactions $T_1$, $T_2$, $T_3$, and $T_4$. Suppose that we use the *immediate update protocol* with checkpointing. Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone, and whether any cascading rollback takes place.

19.24. Suppose that we use the deferred update protocol for the example in Figure 19.7. Show how the log would be different in the case of deferred update by removing the unnecessary log entries; then describe the recovery process, using your modified log. Assume that only REDO operations are applied, and specify which operations in the log are redone and which are ignored.

19.25. How does checkpointing in ARIES differ from checkpointing as described in Section 19.1.4?

19.26. How are log sequence numbers used by ARIES to reduce the amount of REDO work needed for recovery? Illustrate with an example using the information shown in Figure 19.6. You can make your own assumptions as to when a page is written to disk.

[start_transaction, $T_1$]
[read_item, $T_1$, $A$]
[read_item, $T_1$, $D$]
[write_item, $T_1$, $D$, 20, 25]
[commit, $T_1$]
[checkpoint]
[start_transaction, $T_2$]
[read_item, $T_2$, $B$]
[write_item, $T_2$, $B$, 12, 18]
[start_transaction, $T_4$]
[read_item, $T_4$, $D$]
[write_item, $T_4$, $D$, 25, 15]
[start_transaction, $T_3$]
[write_item, $T_3$, $C$, 30, 40]
[read_item, $T_4$, $A$]
[write_item, $T_4$, $A$, 30, 20]
[commit, $T_4$]
[read_item, $T_2$, $D$]
[write_item, $T_2$, $D$, 15, 25] ← system crash

**FIGURE 19.7** An example schedule and its corresponding log.

19.27. What implications would a no-steal/force buffer management policy have on checkpointing and recovery?

Choose the correct answer for each of the following multiple-choice questions:

19.28. Incremental logging with deferred updates implies that the recovery system must necessarily
   a. store the old value of the updated item in the log.
   b. store the new value of the updated item in the log.
   c. store both the old and new value of the updated item in the log.
   d. store only the Begin Transaction and Commit Transaction records in the log.

19.29. The write ahead logging (WAL) protocol simply means that
   a. the writing of a data item should be done ahead of any logging operation.
   b. the log record for an operation should be written before the actual data is written.
   c. all log records should be written before a new transaction begins execution.
   d. the log never needs to be written to disk.

19.30. In case of transaction failure under a deferred update incremental logging scheme, which of the following will be needed:
   a. an undo operation.
   b. a redo operation.
   c. an undo and redo operation.
   d. none of the above.

19.31. For incremental logging with immediate updates, a log record for a transaction would contain:
   a. a transaction name, data item name, old value of item, new value of item.
   b. a transaction name, data item name, old value of item.
   c. a transaction name, data item name, new value of item.
   d. a transaction name and a data item name.

19.32. For correct behavior during recovery, undo and redo operations must be
   a. commutative.
   b. associative.
   c. idempotent.
   d. distributive.

19.33. When a failure occurs, the log is consulted and each operation is either undone or redone. This is a problem because
   a. searching the entire log is time consuming.
   b. many redo's are unnecessary.
   c. both (a) and (b).
   d. none of the above.

19.34. When using a log based recovery scheme, it might improve performance as well as providing a recovery mechanism by
   a. writing the log records to disk when each transaction commits.
   b. writing the appropriate log records to disk during the transaction's execution.
   c. waiting to write the log records until multiple transactions commit and writing them as a batch.
   d. never writing the log records to disk.

19.35. There is a possibility of a cascading rollback when
   a. a transaction writes items that have been written only by a committed transaction.
   b. a transaction writes an item that is previously written by an uncommitted transaction.
   c. a transaction reads an item that is previously written by an uncommitted transaction.
   d. both (b) and (c).

19.36. To cope with media (disk) failures, it is necessary
   a. for the DBMS to only execute transactions in a single user environment.
   b. to keep a redundant copy of the database.
   c. to never abort a transaction.
   d. all of the above.

19.37. If the shadowing approach is used for flushing a data item back to disk, then
   a. the item is written to disk only after the transaction commits.
   b. the item is written to a different location on disk.
   c. the item is written to disk before the transaction commits.
   d. the item is written to the same disk location from which it was read.

## Selected Bibliography

The books by Bernstein et al. (1987) and Papadimitriou (1986) are devoted to the theory and principles of concurrency control and recovery. The book by Gray and Reuter (1993) is an encyclopedic work on concurrency control, recovery, and other transaction-processing issues.

Verhofstad (1978) presents a tutorial and survey of recovery techniques in database systems. Categorizing algorithms based on their UNDO/REDO characteristics is discussed in Haerder and Reuter (1983) and in Bernstein et al. (1983). Gray (1978) discusses recovery, along with other system aspects of implementing operating systems for databases. The shadow paging technique is discussed in Lorie (1977), Verhofstad (1978), and Reuter (1980). Gray et al. (1981) discuss the recovery mechanism in SYSTEM R. Lockeman and Knutsen (1968), Davies (1972), and Bjork (1973) are early papers that discuss recovery. Chandy et al. (1975) discuss transaction rollback. Lilien and Bhargava (1985) discuss the concept of integrity block and its use to improve the efficiency of recovery.

Recovery using write-ahead logging is analyzed in Jhingran and Khedkar (1992) and is used in the ARIES system (Mohan et al. 1992a). More recent work on recovery includes compensating transactions (Korth et al. 1990) and main memory database recovery (Kumar 1991). The ARIES recovery algorithms (Mohan et al. 1992) have been quite successful in practice. Franklin et al. (1992) discusses recovery in the EXODUS system. Two recent books by Kumar and Hsu (1998) and Kumar and Son (1998) discuss recovery in detail and contain descriptions of recovery methods used in a number of existing relational database products.