part 9

Transaction Processing, Concurrency Control, and Recovery

This page intentionally left blank

chapter 20

Introduction to Transaction Processing Concepts and Theory

The concept of *transaction* provides a mechanism for describing logical units of database processing. **Transaction processing systems** are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users. In this chapter, we present the concepts that are needed in transaction processing systems. We define the concept of a transaction, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correctness. A transaction is typically implemented by a computer program that includes database commands such as retrievals, insertions, deletions, and updates. We introduced some of the basic techniques for database programming in Chapters 10 and 11.

In this chapter, we focus on the basic concepts and theory that are needed to ensure the correct executions of transactions. We discuss the concurrency control problem, which occurs when multiple transactions submitted by various users interfere with one another in a way that produces incorrect results. We also discuss the problems that can occur when transactions fail, and how the database system can recover from various types of failures.

This chapter is organized as follows. Section 20.1 informally discusses why concurrency control and recovery are necessary in a database system. Section 20.2 defines the term *transaction* and discusses additional concepts related to transaction

processing in database systems. Section 20.3 presents the important properties of atomicity, consistency preservation, isolation, and durability or permanency—called the ACID properties—that are considered desirable in transaction processing systems. Section 20.4 introduces the concept of schedules (or histories) of executing transactions and characterizes the *recoverability* of schedules. Section 20.5 discusses the notion of *serializability* of concurrent transaction execution, which can be used to define correct execution sequences (or schedules) of concurrent transactions. In Section 20.6, we present some of the commands that support the transaction concept in SQL, and we introduce the concepts of isolation levels. Section 20.7 summarizes the chapter.

The two following chapters continue with more details on the actual methods and techniques used to support transaction processing. Chapter 21 gives an overview of the basic concurrency control protocols and Chapter 22 introduces recovery techniques.

20.1 Introduction to Transaction Processing

In this section, we discuss the concepts of concurrent execution of transactions and recovery from transaction failures. Section 20.1.1 compares single-user and multiuser database systems and demonstrates how concurrent execution of transactions can take place in multiuser systems. Section 20.1.2 defines the concept of transaction and presents a simple model of transaction execution based on read and write database operations. This model is used as the basis for defining and formalizing concurrency control and recovery concepts. Section 20.1.3 uses informal examples to show why concurrency control techniques are needed in multiuser systems. Finally, Section 20.1.4 discusses why techniques are needed to handle recovery from system and transaction failures by discussing the different ways in which transactions can fail while executing.

20.1.1 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**. 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 users and travel agents concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. In these systems, hundreds or thousands of users are typically operating on the database by submitting transactions concurrently to the system.

Multiple users can access databases—and use computer systems—simultaneously because of the concept of **multiprogramming**, which allows the operating system of the computer to execute multiple programs—or **processes**—at the same time. A single

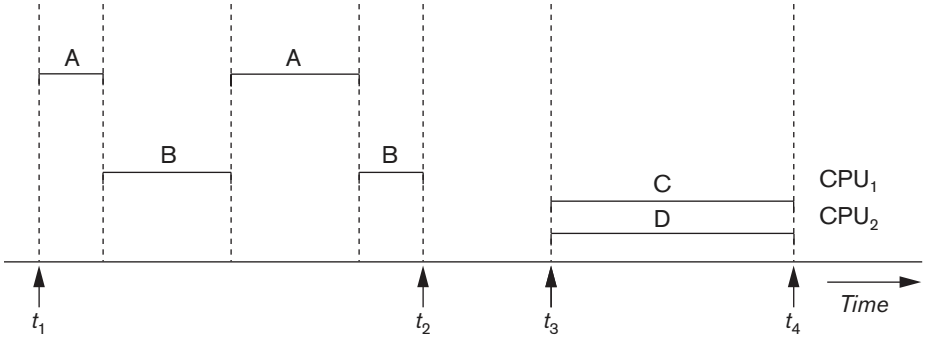


Figure 20.1
Interleaved processing versus parallel processing of concurrent transactions.

central processing unit (CPU) can only 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 20.1, 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.

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 20.1. 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.

20.1.2 Transactions, Database Items, 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 (update), 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**; otherwise it is known as a **read-write transaction**.

The *database model* that is used to present transaction processing concepts is simple when compared to the data models that we discussed earlier in the book, such as the relational model or the object model. A **database** is basically represented as a collection of *named data items.* The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are independent of the data item granularity (size) and apply to data items in general. Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. If the item granularity is a single record, then the record id can be the item name. 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 16, the basic unit of data transfer from disk to main memory is one disk page (disk 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). The size of the buffer is the same as the disk block size.
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 disk block from the buffer back to disk (either immediately or at some later point in time).

It is step 4 that actually updates the database on disk. Sometimes 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 a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in cooperation with the underlying operating system. The DBMS will maintain in the **database cache**

a number of **data buffers** in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed. When these buffers are all occupied, and additional database disk blocks must be copied into memory, some **buffer replacement policy** is used to choose which of the current occupied buffers is to be replaced. Some commonly used buffer replacement policies are **LRU** (least recently used). If the chosen buffer has been modified, it must be written back to disk before it is reused.[1] There are also buffer replacement policies that are specific to DBMS characteristics. We briefly discuss a few of these in Section 20.2.4.

A transaction includes read_item and write_item operations to access and update the database. Figure 20.2 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 20.2 is {*X*, *Y*} and its write-set is also {*X*, *Y*}.

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

20.1.3 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 (uniquely identifiable) data item*, among other information. Figure 20.2(a) 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 20.2(b) 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 number, the flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many different transactions*, each with a different flight number, date, and number 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

[1]We will not discuss general-purpose buffer replacement policies here because they are typically discussed in operating systems texts.

[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*.

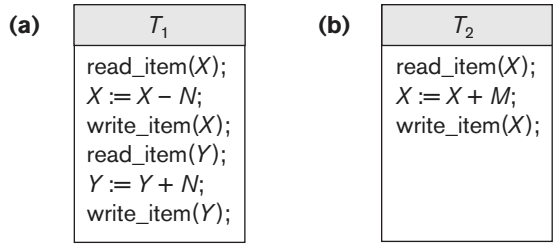


Figure 20.2
Two sample transactions.
(a) Transaction T_1.
(b) Transaction T_2.

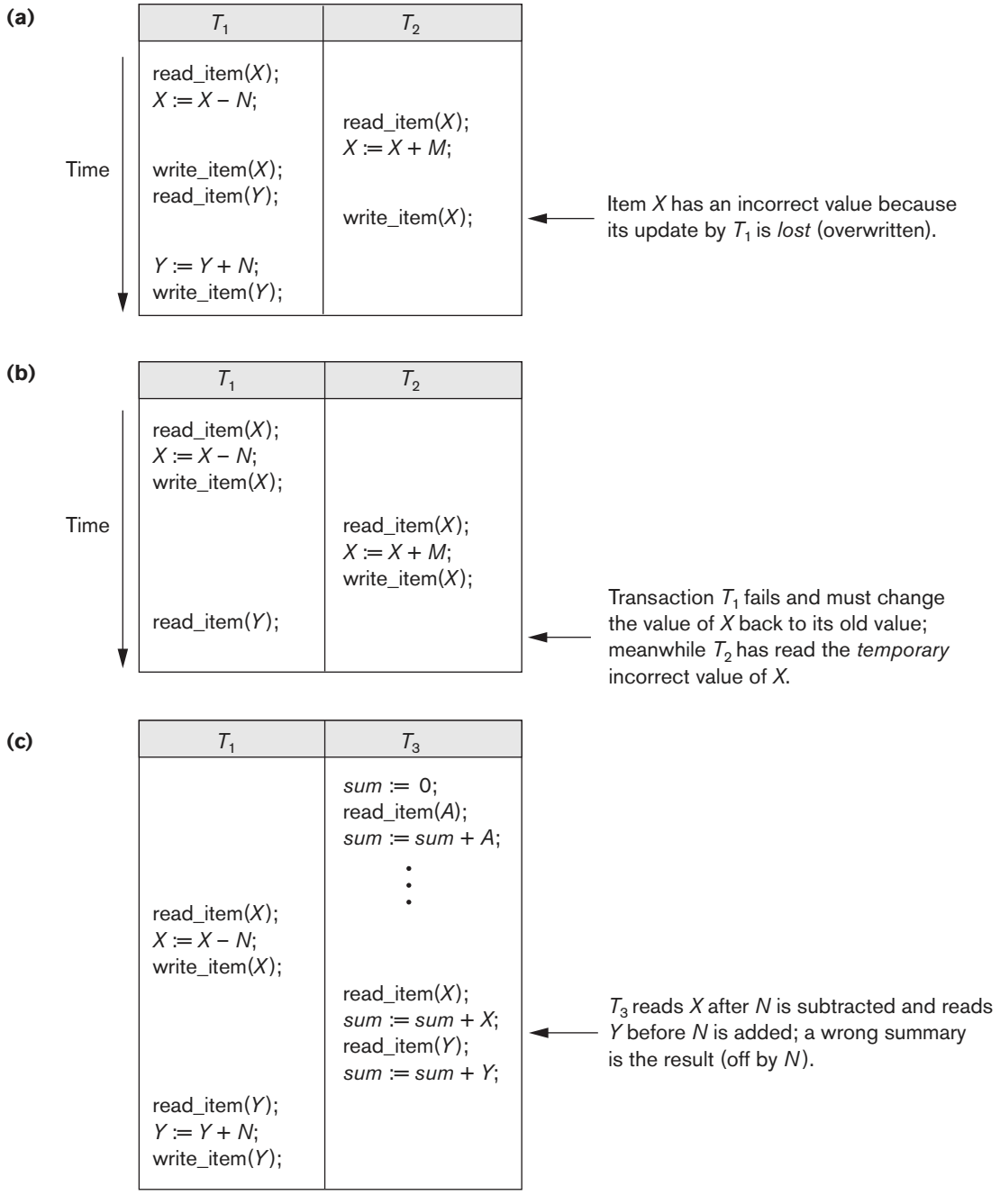
of seats. In Figures 20.2(a) and (b), the transactions T_1 and T_2 are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items X and Y in the database. Next we discuss the types of problems we may encounter with these two simple transactions 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 in Figure 20.3(a); 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 $X = 79$. However, in the interleaving of operations shown in Figure 20.3(a), 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 20.1.4). Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value. Figure 20.3(b) shows an example where T_1 updates item X and then fails before completion, so the system must roll back X 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 database items while other transactions are updating some of these items, 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 20.3(c) 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.

Figure 20.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



The Unrepeatable Read Problem. Another problem that may occur is called *unrepeatable read*, where a transaction T reads the same 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 inquires 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, and it may end up reading a different value for the item.

20.1.4 Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions. In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**. The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not, because *the whole transaction* is a logical unit of database processing. If a transaction **fails** after executing some of its operations but before executing all of them, the operations already executed must be undone and have no lasting effect.

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] Additionally, 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. 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 could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

[3]In general, a transaction should be thoroughly tested to ensure that it does not have any bugs (logical programming errors).

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

4. **Concurrency control enforcement.** The concurrency control method (see Chapter 21)may abort a transaction because it violates serializability (see Section 20.5), or it may abort one or more transactions to resolve a state of deadlock among several transactions (see Section 21.1.3). Transactions aborted because of serializability violations or deadlocks are typically restarted automatically at a later time.
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.

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 quickly 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 22.

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

20.2 Transaction and System Concepts

In this section, we discuss additional concepts relevant to transaction processing. Section 20.2.1 describes the various states a transaction can be in and discusses other operations needed in transaction processing. Section 20.2.2 discusses the system log, which keeps information about transactions and data items that will be needed for recovery. Section 20.2.3 describes the concept of commit points of transactions and why they are important in transaction processing. Finally, Section 20.2.4 briefly discusses DBMS buffer replacement policies.

20.2.1 Transaction States and Additional Operations

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts (see Section 20.2.3). Therefore, the recovery manager of the DBMS needs to keep 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

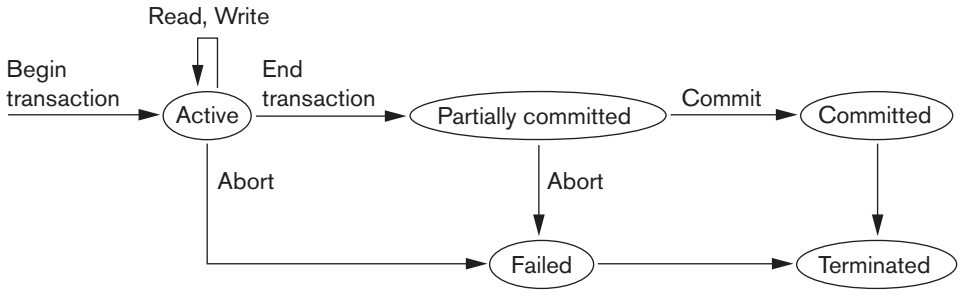


Figure 20.4
State transition diagram illustrating the states for transaction execution.

whether the transaction has to be aborted because it violates serializability (see Section 20.5) 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 20.4 shows a state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the **partially committed state**. At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not. Also, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log, discussed in the next section).[5] If these checks are 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 20.2.3. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

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.

[5]Optimistic concurrency control (see Section 21.4) also requires that certain checks are made at this point to ensure that the transaction did not interfere with other executing transactions.

20.2.2 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, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers, called the **log buffers**, hold the last part of the log file, so that log entries are first added to the log main memory buffer. When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures. The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record. In these entries, *T* refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:

1. **[start_transaction, *T*].** Indicates that transaction *T* has started execution.
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 20.4.2)—which include nearly all practical protocols—*do not require* that READ operations are 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. Additionally, some recovery protocols require simpler WRITE entries that only include one of new_value or old_value instead of including both (see Section 20.4.2).

Notice that we are assuming 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 22. 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. **Redo** of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the sys-

[6]The log has sometimes been called the *DBMS journal*.

tem can be sure that all these new_values have been written to the actual database on disk from the main memory buffers.[7]

20.2.3 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 must be *permanently recorded* in the database. The transaction then writes a commit record [commit, T] into the log. If a system failure occurs, we can 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.

Notice that the log file must be kept on disk. As discussed in Chapter 16, 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. As we mentioned earlier, it is common to keep one or more blocks of the log file in main memory buffers, called the **log buffer**, 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 buffer. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process if the contents of main memory are 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 buffer to disk before committing a transaction.

20.2.4 DBMS-Specific Buffer Replacement Policies

The DBMS cache will hold the disk pages that contain information currently being processed in main memory buffers. If all the buffers in the DBMS cache are occupied and new disk pages are required to be loaded into main memory from disk, a **page replacement policy** is needed to select the particular buffers to be replaced. Some page replacement policies that have been developed specifically for database systems are briefly discussed next.

Domain Separation (DS) Method. In a DBMS, various types of disk pages exist: index pages, data file pages, log file pages, and so on. In this method, the DBMS cache is divided into separate domains (sets of buffers). Each domain handles one type of disk pages, and page replacements within each domain are han-

[7]Undo and redo are discussed more fully in Chapter 22.

dled via the basic LRU (least recently used) page replacement. Although this achieves better performance on average that basic LRU, it is a *static algorithm,* and so does not adapt to dynamically changing loads because the number of available buffers for each domain is predetermined. Several variations of the DS page replacement policy have been proposed, which add dynamic load-balancing features. For example, the **GRU** (Group LRU) gives each domain a priority level and selects pages from the lowest-priority level domain first for replacement, whereas another method dynamically changes the number of buffers in each domain based on current workload.

Hot Set Method. This page replacement algorithm is useful in queries that have to scan a set of pages repeatedly, such as when a join operation is performed using the nested-loop method (see Chapter 18). If the inner loop file is loaded completely into main memory buffers without replacement (the hot set), the join will be performed efficiently because each page in the outer loop file will have to scan all the records in the inner loop file to find join matches. The hot set method determines for each database processing algorithm the set of disk pages that will be accessed repeatedly, and it does not replace them until their processing is completed.

The DBMIN Method. This page replacement policy uses a model known as **QLSM** (query locality set model), which predetermines the pattern of page references for each algorithm for a particular type of database operation. We discussed various algorithms for relational operations such as SELECT and JOIN in Chapter 18. Depending on the type of access method, the file characteristics, and the algorithm used, the QLSM will estimate the number of main memory buffers needed for each file involved in the operation. The DBMIN page replacement policy will calculate a **locality set** using QLSM for each file instance involved in the query (some queries may reference the same file twice, so there would be a locality set for each file instance needed in the query). DBMIN then allocates the appropriate number of buffers to each file instance involved in the query based on the locality set for that file instance. The concept of locality set is analogous to the concept of *working set*, which is used in page replacement policies for processes by the operating system but there are multiple locality sets, one for each file instance in the query.

20.3 Desirable Properties of Transactions

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

- **Atomicity.** A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
- **Consistency preservation.** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

- **Isolation.** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
- **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. On the other hand, write operations of a committed transaction must be eventually written to disk.

The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs and 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 on the database that should hold. 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.

The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS.[8] If every transaction does not make its updates (write operations) 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 22) but does not eliminate all other problems.

The *durability property* is the responsibility of the *recovery subsystem* of the DBMS. In the next section, we introduce how recovery protocols enforce durability and atomicity and then discuss this in more detail in Chapter 22.

Levels of Isolation. 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 level 2 properties, repeatable reads.[9] Another type of isolation is called **snapshot isolation**, and several practical concurrency control methods are based on this. We shall discuss snapshot isolation in Section 20.6, and again in Chapter 21, Section 21.4.

[8]We will discuss concurrency control protocols in Chapter 21.

[9]The SQL syntax for isolation level discussed in Section 20.6 is closely related to these levels.

20.4 Characterizing Schedules Based on Recoverability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or **history**). In this section, first we define the concept of schedules, and then we characterize the types of schedules that facilitate recovery when failures occur. In Section 20.5, we characterize schedules in terms of the interference of participating transactions; this discussion leads to the concepts of serializability and serializable schedules.

20.4.1 Schedules (Histories) of Transactions

A **schedule** (or **history**) S of n transactions T_1, T_2, … , T_n is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule S. However, for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i. The order of operations in S is considered to be a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders*, but we will assume for now total ordering of the operations in a schedule.

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 b, r, w, e, c, and a for the operations begin_transaction, read_item, write_item, end_transaction, commit, and abort, respectively, and appends as a *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 operations in parentheses. In some schedules, we will only show the *read* and *write* operations, whereas in other schedules we will show additional operations, such as commit or abort. The schedule in Figure 20.3(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 20.3(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:

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

Conflicting Operations in a Schedule. 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.

Intuitively, two operations are conflicting if changing their order can result in a different outcome. For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, then the value of X that is read by transaction T_1 changes, because in the second ordering the value of X is read by $r_1(X)$ *after* it is changed by $w_2(X)$, whereas in the first ordering the value is read *before* it is changed. This is called a **read-write conflict**. The other type is called a **write-write conflict** and is illustrated by the case where we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$. For a write-write conflict, the *last value* of X will differ because in one case it is written by T_2 and in the other case by T_1. Notice that two read operations are not conflicting because changing their order makes no difference in outcome.

The rest of this section covers some theoretical definitions concerning schedules. 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 relative 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.[10]

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.[11] 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.

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

[11]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.

20.4.2 Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved. In some cases, it is even not possible to recover correctly after a failure. 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; they 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. This ensures that the durability property of transactions is not violated (see Section 20.3). The schedules that theoretically meet this criterion are called *recoverable schedules.* A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS. The condition for a **recoverable schedule** is as follows: A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written some item X 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).

Some recoverable schedules may require a complex recovery process, as we shall see, but if sufficient information is kept (in the log), a recovery algorithm can be devised for any recoverable schedule. 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; this problem is handled by serializability theory (see Section 20.5). 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, but T_2 commits before T_1 commits. The problem occurs 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 is 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 S_e, aborting T_2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule S_c.

In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, 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 time-consuming—since numerous transactions can be rolled back (see Chapter 22)—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 because the transactions that wrote them have committed, 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.

It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable. Suppose we have i transactions $T_1, T_2, \ldots, T_i$, and their number of operations are $n_1, n_2, \ldots, n_i$, respectively. If we make a set of *all possible schedules* of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable. The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

Most recovery protocols allow only strict schedules, so that the recovery process itself is not complicated (see Chapter 22).

20.5 Characterizing Schedules Based on Serializability

In the previous section, we characterized schedules based on their recoverability properties. Now we characterize the types of schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as *serializable schedules*. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T_1 and T_2 in Figure 20.2 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 two schedules—called *serial schedules*—are shown in Figures 20.5(a) 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 in Figure 20.5(c). 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.

20.5.1 Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figures 20.5(a) 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 20.5(a), and T_2 and then T_1 in Figure 20.5(b). Schedules C and D in Figure 20.5(c) 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**. Therefore, 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 20.3). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, we get a correct end result.

The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O

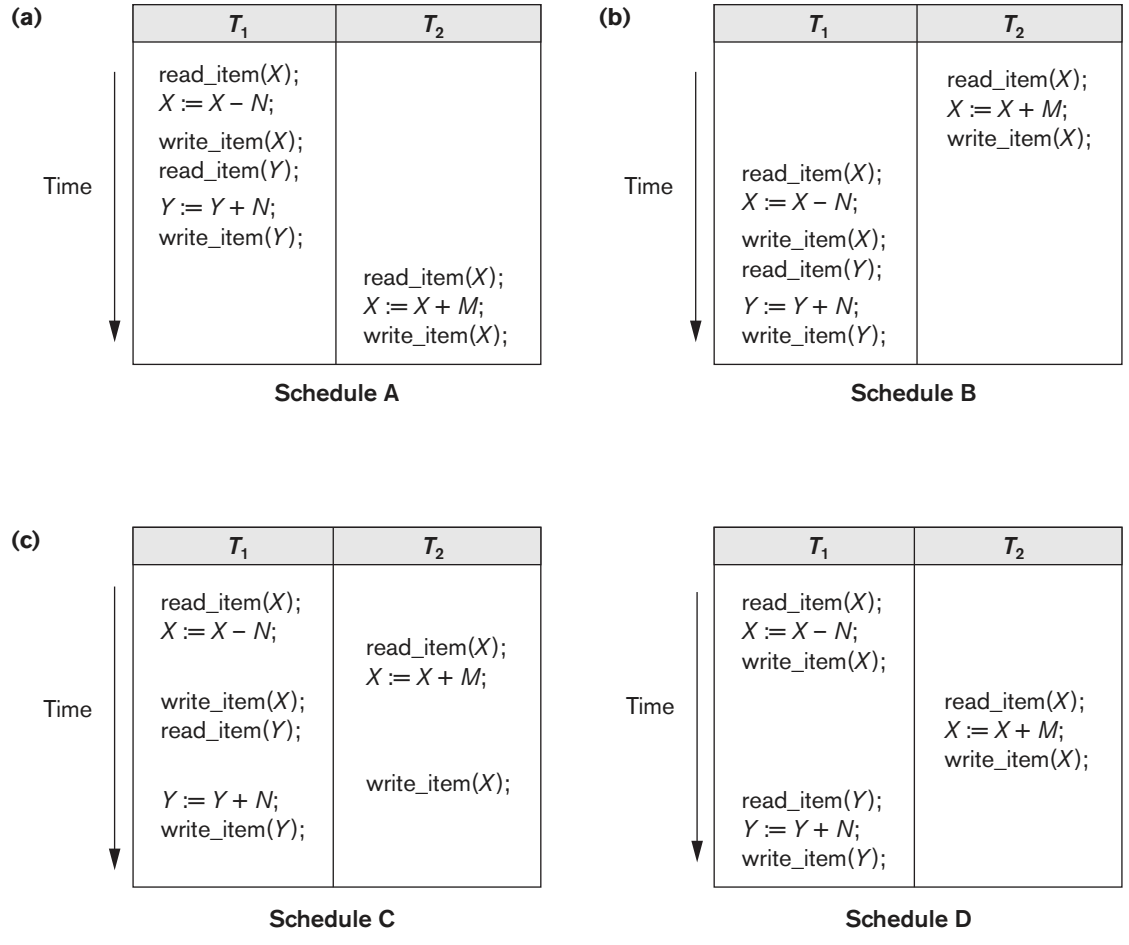


Figure 20.5
Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. Additionally, if some transaction T is long, the other transactions must wait for T to complete all its operations before starting. Hence, serial schedules are *unacceptable* in practice. However, if we can determine which other schedules are *equivalent* to a serial schedule, we can allow these schedules to occur.

To illustrate our discussion, consider the schedules in Figure 20.5, 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 20.3(a)) 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 20.1.3; 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.

The definition of *serializable schedule* is as follows: 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 schedule equivalence. The simplest but least satisfactory definition 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 20.6, schedules S_1 and S_2 will produce the same final database state if they execute on a database with an initial value of $X = 100$; however, for other initial values of X, the schedules are *not* result equivalent. Additionally, these 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 to focus only on the read_item and write_item operations of the transactions, and not make any assumptions about the other internal 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.

Conflict Equivalence of Two Schedules. Two schedules are said to be **conflict equivalent** if the relative order of any two *conflicting operations* is the same in both schedules. Recall from Section 20.4.1 that two operations in a schedule are said to

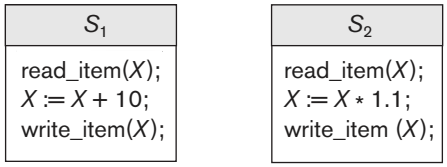


Figure 20.6
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

conflict if they belong to different transactions, access the same database item, and either both are write_item operations or one is a write_item and the other a read_item. If two conflicting operations are applied in *different orders* in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent. For example, as we discussed in Section 20.4.1, 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.

Serializable Schedules. Using the notion of conflict equivalence, we define a schedule S to be **serializable**[12] 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 in Figure 20.5(c) is equivalent to the serial schedule A in Figure 20.5(a). In both schedules, the read_item(X) of T_2 reads the value of X written by T_1, whereas the other read_item operations read the database values from the initial database state. Additionally, 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. Therefore, 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 in Figure 20.5(c) 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 20.5.4.

[12]We will use *serializable* to mean conflict serializable. Another definition of serializable used in practice (see Section 20.6) is to have repeatable reads, no dirty reads, and no phantom records (see Section 22.7.1 for a discussion on phantoms).

20.5.2 Testing for Serializability of a Schedule

There is a simple algorithm for determining whether a particular schedule is (conflict) serializable or not. Most concurrency control methods do *not* actually test for serializability. Rather protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable. Some methods guarantee serializability in most cases, but do not guarantee it absolutely, in order to reduce the overhead of concurrency control. 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 21.

Algorithm 20.1 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 from node T_j to node T_k is created by the algorithm if a pair of conflicting operations exist in T_j and T_k and the conflicting operation in T_j appears in the schedule *before* the *conflicting operation* in T_k.

Algorithm 20.1. 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.

The precedence graph is constructed as described in Algorithm 20.1. 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

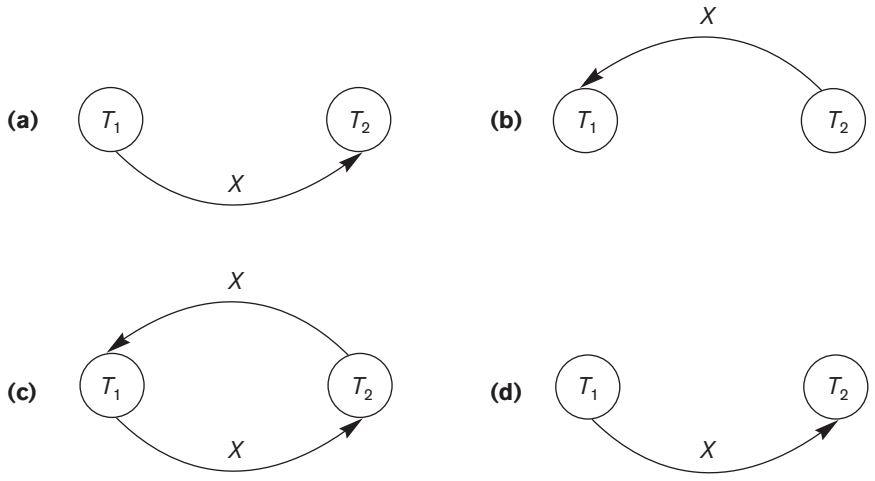


Figure 20.7
Constructing the precedence graphs for schedules A to D from Figure 20.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

in the precedence graph from T_i to T_j, T_i must appear before T_j in the equivalent serial schedule S'.[13] 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 20.7 shows such labels on the edges. When checking for a cycle, the labels are not relevant.

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, in Figure 20.5 appear in Figures 20.7(a) to (d). The 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 20.8. Figure 20.8(a) shows the read_item and write_item operations in each transaction. Two schedules E and F for these transactions are shown in Figures 20.8(b) and (c), respectively, and the precedence graphs for schedules E and F are shown in Figures 20.8(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 20.8(e). 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 20.8(f) shows a precedence graph representing a schedule

[13]This process of ordering the nodes of an acrylic graph is known as *topological sorting*.

Figure 20.8
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.

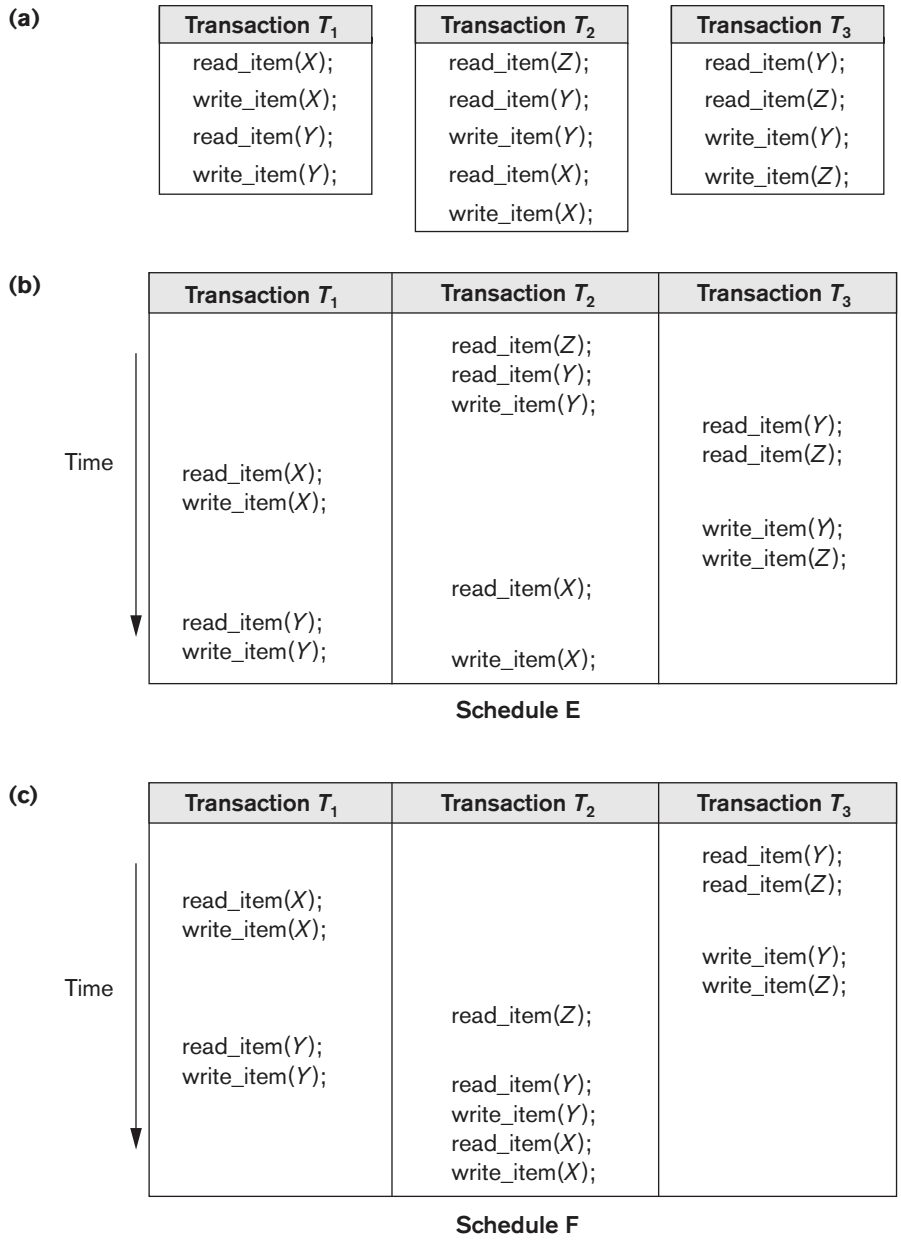
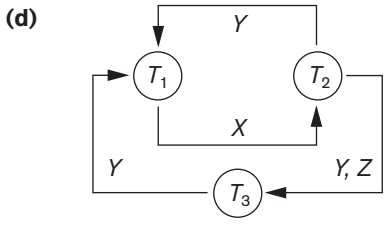


Figure 20.8 (continued)
Another example of serializability testing. (d) Precedence graph for schedule E. (e) Precedence graph for schedule F. (f) Precedence graph with two equivalent serial schedules.



that has two equivalent serial schedules. To find an equivalent serial schedule, start with a node that does not have any incoming edges, and then make sure that the node order for every edge is not violated.

20.5.3 How Serializability Is Used for Concurrency Control

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 a long transaction to delay other transactions, thus slowing down transaction processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness. In practice, it is 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. 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*. Some protocols may allow nonserializable schedules in rare cases to reduce the overhead of the concurrency control method (see Section 20.6).

Another problem is that transactions are submitted continuously to the system, so 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 20.4.1 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.

In Chapter 21, 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 some commercial DBMSs. We will also discuss a protocol based on the concept of *snapshot isolation* that ensures serializability in most but not all cases; this is used in some commercial DBMSs because it has less overhead than the two-phase locking protocol. Other protocols have been proposed[14]; these 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.

20.5.4 View Equivalence and View Serializability

In Section 20.5.1, 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

[14]These other protocols have not been incorporated much into commercial systems; most relational DBMSs use some variation of two-phase locking or snapshot isolation.

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** (or **no blind writes**) 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. A **blind write** is a write operation in a transaction T on an item X that is not dependent on the old value of X, so it is not preceded by a read of X in the transaction T.

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. This is possible when *blind writes* are allowed, 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 (as an exercise, the reader should construct the serializability graph for S_g and check for cycles). 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.

20.5.5 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 transactions, each of which may be used to transfer an amount of money between two bank accounts:

T_1: $r_1(X)$; X :{equal} $X - 10$; $w_1(X)$; $r_1(Y)$; Y :{equal} $Y + 10$; $w_1(Y)$;
T_2: $r_2(Y)$; Y :{equal} $Y - 20$; $w_2(Y)$; $r_2(X)$; X :{equal} $X + 20$; $w_2(X)$;

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. Also, in certain domains of applications, such as computer-aided design (CAD) of complex systems like aircraft, design transactions last over a long time period. In such applications, more relaxed schemes of concurrency control have been proposed to maintain consistency of the database, such as *eventual consistency*. We shall discuss eventual consistency in the context of distributed databases in Chapter 23.

20.6 Transaction Support in SQL

In this section, we give a brief introduction to transaction support in SQL. There are many more details, and the newer standards have more commands for transaction processing. The basic 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 an 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 select, 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*, which indicates 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 *n* 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.[15] The default isolation level is SERIALIZABLE, although some systems use READ COMMITTED as their default. The use of the term SERIALIZABLE here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms,[16] and it is thus not identical to the way serializability was defined earlier in Section 20.5. 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 *r* that also satisfies the WHERE-clause condition used in T_1, into the table used by T_1. The record *r* is called a **phantom record** because it was not there when T_1 starts but is there when T_1 ends. T_1 may or may not see the phantom, a row that previously did not exist. If the equivalent serial order is T_1 followed by T_2, then the record *r* should not be seen; but if it is T_2 followed by T_1,then the phantom record should be in the result given to T_1. If the system cannot ensure the correct behavior, then it does not deal with the phantom record problem.

[15]These are similar to the *isolation levels* discussed briefly at the end of Section 20.3.

[16]The dirty read and unrepeatable read problems were discussed in Section 20.1.3. Phantoms are discussed in Section 22.7.1.

Table 20.1 Possible Violations Based on Isolation Levels as Defined in SQL

	Type of Violation		
Isolation Level	**Dirty Read**	**Nonrepeatable Read**	**Phantom**
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Table 20.1 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. READ UNCOMMITTED is the most forgiving, and SERIALIZABLE is the most restrictive in that it avoids all three of the problems mentioned above.

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.

Snapshot Isolation. Another isolation level, known as snapshot isolation, is used in some commercial DBMSs, and some concurrency control protocols exist that are based on this concept. The basic definition of **snapshot isolation** is that a transaction sees the data items that it reads based on the committed values of the items in the *database snapshot* (or database state) when the transaction starts. Snapshot isolation will ensure that the phantom record problem does not occur, since

the database transaction, or in some cases the database statement, will only see the records that were committed in the database at the time the transaction starts. Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction. We will discuss a concurrency control protocol based on this concept in Chapter 21.

20.7 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 in Section 20.1. 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 in Section 20.1.1. We also discussed the various types of failures that may occur during transaction execution in Section 20.1.4.

Next, in Section 20.2, we 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 (Section 20.2.2) keeps track of database accesses, and the system uses this information to recover from failures. A transaction can succeed and reach its commit point, or it can fail and has to be rolled back. A committed transaction (Section 20.2.3) has its changes permanently recorded in the database. In Section 20.3, we presented an overview of the desirable properties of transactions—atomicity, consistency preservation, isolation, and durability—which are often referred to as the ACID properties.

Then we defined a schedule (or history) as an execution sequence of the operations of several transactions with interleaving in Section 20.4.1. We characterized schedules in terms of their recoverability in Section 20.4.2. 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.

Then in Section 20.5 we defined the 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. A serializable schedule is considered correct. We presented an algorithm for testing the (conflict) serializability of a schedule in Section 20.5.2. We discussed why testing for serializability is impractical in a real system, although it can be used to define and verify concurrency control protocols in Section 20.5.3, and we briefly mentioned less restrictive definitions of schedule equivalence in Sections 20.5.4 and 20.5.5. Finally, in Section 20.6, we gave a brief overview of how transaction concepts are used in practice within SQL, and we introduced the concept of snapshot isolation, which is used in several commercial DBMSs.

Review Questions

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

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

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

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

20.5. 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?

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

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

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

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

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

20.11. 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?

20.12. Describe the four levels of isolation in SQL. Also discuss the concept of snapshot isolation and its effect on the phantom record problem.

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

Exercises

20.14. Change transaction T_2 in Figure 20.2(b) 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 Figures 20.3(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)?

20.15. Repeat Exercise 20.14, adding a check in T_1 so that Y does not exceed 90.

20.16. Add the operation commit at the end of each of the transactions T_1 and T_2 in Figure 20.2, and then list all possible schedules for the modified transactions. Determine which of the schedules are recoverable, which are cascadeless, and which are strict.

20.17. List all possible schedules for transactions T_1 and T_2 in Figure 20.2, and determine which are conflict serializable (correct) and which are not.

20.18. How many *serial* schedules exist for the three transactions in Figure 20.8(a)? What are they? What is the total number of possible schedules?

20.19. Write a program to create all possible schedules for the three transactions in Figure 20.8(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.

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

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

20.22. 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. $r_3(X); r_2(X); r_1(X); w_3(X); w_1(X);$

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

20.24. 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 serializability and related ideas to maintain consistency in a database were introduced in Gray et al. (1975). The concept of the database transaction was first discussed in Gray (1981). Gray won the coveted ACM Turing Award in 1998 for his work on database transactions and implementation of transactions in relational DBMSs. Bernstein, Hadzilacos, and Goodman (1988) 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) offers collections of research papers on transaction processing for advanced applications. Transaction support in SQL is described in Date and Darwen (1997). View serializability is defined in Yannakakis (1984). Recoverability of schedules and reliability in databases is discussed in Hadzilacos (1983, 1988). Buffer replacement policies are discussed in Chou and DeWitt (1985). Snapshot isolation is discussed in Ports and Grittner (2012).

This page intentionally left blank

chapter 21

Concurrency Control Techniques

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

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

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

21.1 Two-Phase Locking Techniques for Concurrency Control

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

21.1.1 Types of Locks and System Lock Tables

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

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

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

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

Figure 21.1
Lock and unlock operations for binary locks.

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

21.1.2 Guaranteeing Serializability by Two-Phase Locking

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

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

(a)

T_1	T_2

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

(b)

Initial values: X=20, Y=30

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

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

(c)

T_1	T_2

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

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

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

Time

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

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

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

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

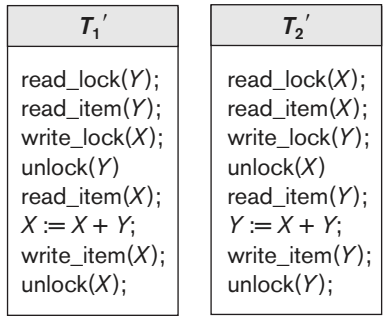


Figure 21.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

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

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

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

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

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

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

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

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

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

21.1.3 Dealing with Deadlock and Starvation

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

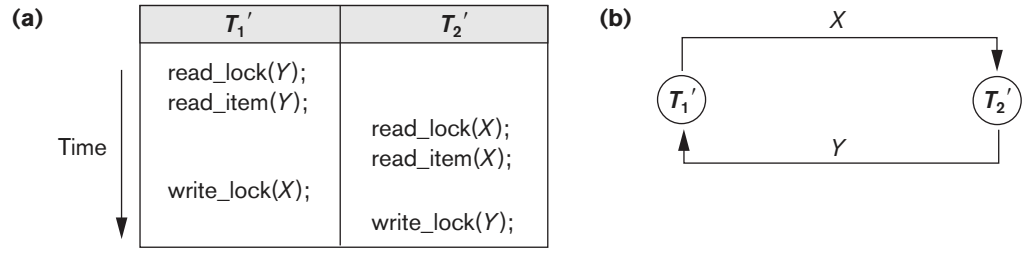


Figure 21.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

21.2 Concurrency Control Based on Timestamp Ordering

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

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

21.2.1 Timestamps

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

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

21.2.2 The Timestamp Ordering Algorithm for Concurrency Control

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

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

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

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

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

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

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

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

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

21.3 Multiversion Concurrency Control Techniques

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

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

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

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

21.3.1 Multiversion Technique Based on Timestamp Ordering

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

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

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

To ensure serializability, the following rules are used:

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

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

21.3.2 Multiversion Two-Phase Locking Using Certify Locks

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

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

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

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

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

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

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

21.4 Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

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

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

21.4.1 Validation-Based (Optimistic) Concurrency Control

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

There are three phases for this concurrency control protocol:

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

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

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

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

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

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

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

21.4.2 Concurrency Control Based on Snapshot Isolation

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

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

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

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

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

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

21.5 Granularity of Data Items and Multiple Granularity Locking

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

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

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

21.5.1 Granularity Level Considerations for Locking

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

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

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

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

21.5.2 Multiple Granularity Level Locking

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

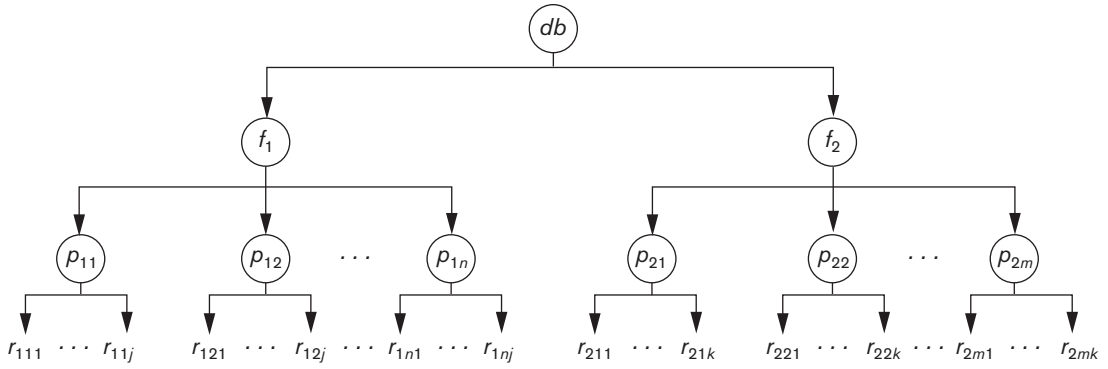


Figure 21.7
A granularity hierarchy for illustrating multiple granularity level locking.

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

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

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

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

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

Figure 21.8
Lock compatibility matrix for multiple granularity locking.

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

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

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

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

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

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

Figure 21.9
Lock operations to illustrate a serializable schedule.

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

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

21.6 Using Locks for Concurrency Control in Indexes

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

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

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

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

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

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

21.7 Other Concurrency Control Issues

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

21.7.1 Insertion, Deletion, and Phantom Records

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

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

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

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

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

21.7.2 Interactive Transactions

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

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

21.7.3 Latches

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

21.8 Summary

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

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

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

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

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

Review Questions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

21.15. What are intention locks?

21.16. When are latches used?

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

21.18. How does index locking resolve the phantom problem?

21.19. What is a predicate lock?

Exercises

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

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

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

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

21.24. Prove that cautious waiting avoids deadlock.

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

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

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

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

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

Selected Bibliography

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

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

Anomalies that can occur with basic snapshot isolation are discussed in Fekete et al. (2004), Jorwekar et al. (2007), and Ports and Grittner (2012), among others. Modifying snapshot isolation to make it serializable is discussed in Cahill et al. (2008), Fekete et al. (2005), Revilak et al. (2011), and Ports and Grittner (2012).

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

This page intentionally left blank

chapter 22

Database Recovery Techniques

In this chapter, we discuss some of the techniques that can be used for database recovery in case of system failure. In Section 20.1.4 we discussed the different causes of failure, such as system crashes and transaction errors. In Section 20.2, we introduced some of the concepts that are used by recovery processes, such as the system log and commit points.

This chapter presents additional concepts that are relevant to recovery protocols and provides an overview of the various database recovery algorithms. We start in Section 22.1 with an outline of a typical recovery procedure and a categorization of recovery algorithms, and then we 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 22.2, we present recovery techniques based on *deferred update,* also known as the NO-UNDO/REDO technique, where the data on disk is not updated until *after* a transaction commits. In Section 22.3, we discuss recovery techniques based on *immediate update*, where data can be updated on disk during transaction execution; these include the UNDO/REDO and UNDO/NO-REDO algorithms. In Section 22.4, we discuss the technique known as shadowing or shadow paging, which can be categorized as a NO-UNDO/NO-REDO algorithm. An example of a practical DBMS recovery scheme, called ARIES, is presented in Section 22.5. Recovery in multidatabases is briefly discussed in Section 22.6. Finally, techniques for recovery from catastrophic failure are discussed in Section 22.7. Section 22.8 summarizes the chapter.

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 at the end of the chapter and the online and printed 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 discuss recovery concepts independently of concurrency control mechanisms.

22.1 Recovery Concepts

22.1.1 Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state 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 21.2.2. 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 or other large capacity offline storage media) 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 on disk is not physically damaged, and a noncatastrophic failure of types 1 through 4 in Section 21.1.4 has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing* its write operations. It may also be necessary to *redo* some operations in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet been written to disk. For noncatastrophic failure, the recovery protocol does not need a complete archival copy of the database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish two main policies for recovery from noncatastrophic transaction failures: deferred update and immediate update. The **deferred update** techniques do not physically update the database on disk until *after* a transaction commits; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache; see Section 20.2.4). Before commit, the updates are recorded persistently in the log file on disk, and then after commit, the updates are written to the database from the main memory buffers. If a transaction fails before reaching its commit point, it will not have changed the database on disk 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 on disk. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm**. We discuss this technique in Section 22.2.

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 must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk 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 during recovery and is used most often in practice. A variation of the algorithm where all updates are required to be recorded in the database on disk *before* a transaction commits requires *undo* only, so it is known as the **UNDO/NO-REDO algorithm**. We discuss these two techniques in Section 22.3.

The UNDO and REDO operations are required to be **idempotent**—that is, executing an operation multiple times is equivalent to executing it just once. In fact, the whole recovery process should be idempotent because if the system were to fail during the recovery process, the next recovery attempt might UNDO and REDO certain write_item operations that had already been executed 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*!

22.1.2 Caching (Buffering) of Disk Blocks

The recovery process is often closely intertwined with operating system functions—in particular, the buffering of database disk pages in the DBMS main memory cache. Typically, multiple 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 (see Section 20.2.4).

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, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache. If it is not,

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

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 (see Section 20.2.4).

The entries in the DBMS cache directory hold additional information relevant to buffer management. 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, a new entry is inserted in the cache directory 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). Additional information, such as the transaction id(s) of the transaction(s) that modified the buffer, are also kept in the directory. 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. For example, the recovery protocol may restrict certain buffer pages from being written back to the disk until the transactions that changed this buffer have committed.

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 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 data items can be maintained, but this approach is not typically used in practice.

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)**. If shadowing is used, 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 22.4.

22.1.3 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 21.2.2). 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** and is necessary so we can UNDO the operation if this is required during recovery. 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: the information needed for UNDO

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

and the information 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 on disk 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 BFIM and AFIM are recorded into a single log entry. Additionally, when cascading rollback (see Section 22.1.5) is possible, read_item entries in the log are considered to be UNDO-type entries.

As mentioned, the DBMS cache holds the cached database disk blocks in main memory buffers. The DBMS cache includes not only *data file blocks*, but also *index file blocks* and *log file blocks* from the disk. When a log record is written, it is stored in the current log buffer in the DBMS cache. The log is simply a sequential (append-only) disk file, and the DBMS cache may contain several log blocks in main memory buffers (typically, the last *n* log blocks of the log file). When an update to a data block—stored in the DBMS cache—is made, an associated log record is written to the last log buffer in the DBMS cache. With the write-ahead logging approach, the log buffers (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 from its main memory buffer.

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

1. If a cache buffer page updated by a transaction *cannot* be written to disk before the transaction commits, the recovery method is called a **no-steal approach**. The pin-unpin bit will be set to 1 (pin) to indicate that a cache buffer cannot be written back to disk. On the other hand, if the recovery 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. The *no-steal rule* means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.
2. If all pages updated by a transaction are immediately written to disk *before* the transaction commits, the recovery approach is called a **force approach**. Otherwise, it is called **no-force**. The *force rule* means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.

The deferred update (NO-UNDO) recovery scheme discussed in Section 22.2 follows a *no-steal* approach. However, typical database systems employ a *steal/no-force* (UNDO/REDO) 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 write that page multiple times to disk and possibly having to read it again from disk. This may provide a substantial saving in the number of disk 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 log on 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 entries for the updating transaction—up to this point—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 they may also include lists of all **committed** and **aborted transactions** since the last checkpoint (see the next section). Maintaining these lists makes the recovery process more efficient.

22.1.4 Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint**.[3] A [checkpoint, *list of active transactions*] 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. As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.

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]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.

3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.

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, which is not acceptable in practice. To overcome this, it is common to use a technique called **fuzzy checkpointing**. In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end_checkpoint, …] record is written in the log with the relevant information collected during checkpointing. However, until step 2 is completed, the previous checkpoint record should remain valid. To accomplish this, the system maintains a file on disk that contains 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.

22.1.5 Transaction Rollback and Cascading Rollback

If a transaction fails for whatever reason after updating the database, but before the transaction commits, 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 on disk, 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 it can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules (see Section 20.4.2). Understandably, cascading rollback can be complex and time-consuming. That is why almost all recovery mechanisms are designed so that cascading rollback *is never required.*

Figure 22.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 22.1(a). Figure 22.1(b) 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 22.1(b), are the T_3 operations that are undone during transaction rollback. Figure 22.1(c) graphically shows the operations of the different transactions along the time axis.

(a)

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(C)
write_item(B)
read_item(A)
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)

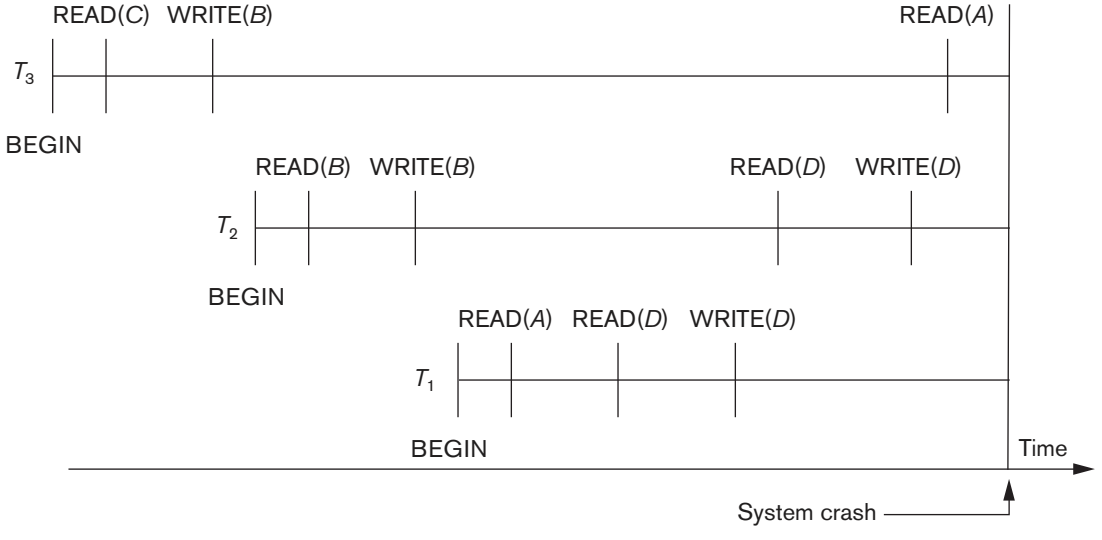


Figure 22.1
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

We must now check for cascading rollback. From Figure 22.1(c), 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.

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.

22.1.6 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 that is based on these reports and 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.

22.2 NO-UNDO/REDO Recovery Based on Deferred Update

The idea behind deferred update is to defer or postpone any actual updates to the database on disk 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. Therefore, only **REDO-type log entries** are needed in the log, which include the **new value** (AFIM) of the item written by a write operation. The **UNDO-type log entries** are not needed since no undoing of operations will be required during recovery. Although this may simplify the recovery process, 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

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

in the cache buffers until the commit point, so many cache buffers will be *pinned* and cannot be replaced.

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; hence all buffers that have been changed by the transaction must be pinned until the transaction commits (this corresponds to a *no-steal policy*).
2. A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and* the log buffer 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. 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 during recovery.

For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated. Consider a system in which concurrency control uses strict two-phase locking, so the locks on written 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.

> **Procedure RDU_M (NO-UNDO/REDO 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.

The REDO procedure is defined as follows:

> **Procedure 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).

Figure 22.2 illustrates a timeline for 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. The write_item operations of T_2 and T_3 must be redone, however, because both transactions reached 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 on disk under the deferred update protocol (no-steal policy).

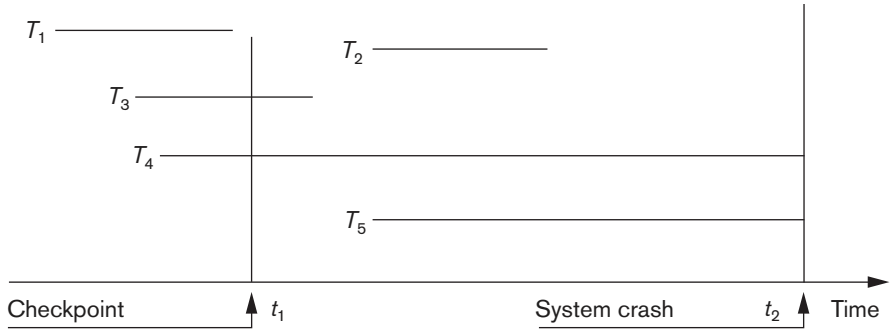


Figure 22.2
An example of a recovery timeline to illustrate the effect of checkpointing.

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 because the other updates would be overwritten by this last REDO. 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 latest 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 write-locked items remain locked until the transaction reaches its commit point.* Additionally, 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 22.3 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

22.3 Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point. Notice that it is *not a requirement* that every update be

(a)

T_1
read_item(A)
read_item(D)
write_item(D)

T_2
read_item(B)
write_item(B)
read_item(D)
write_item(D)

T_3
read_item(A)
write_item(A)
read_item(C)
write_item(C)

T_4
read_item(B)
write_item(B)
read_item(A)
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 22.3
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.

applied immediately to disk; it is just possible that some updates are applied to disk *before the transaction commits*.

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. Therefore, the **UNDO-type log entries**, which include the **old value** (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a **steal strategy** for deciding when updated main memory buffers can be written back to disk (see Section 22.1.3).

Theoretically, we can distinguish two main categories of immediate update algorithms.

1. 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**. In this method, all updates by a transaction must be recorded on disk *before the transaction commits*, so that REDO is never needed. Hence, this method must utilize the **steal/force**

strategy for deciding when updated main memory buffers are written back to disk (see Section 22.1.3).

2. 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**. In this case, the **steal/no-force strategy** is applied (see Section 22.1.3). This is also the most complex technique, but the most commonly used in practice. We will outline an UNDO/REDO recovery algorithm and leave it as an exercise for the reader to develop the UNDO/NO-REDO variation. In Section 22.5, we describe a more practical approach known as the ARIES recovery technique.

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 (UNDO/REDO recovery). 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 wrote the item has committed. 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 (UNDO/REDO with checkpoints).

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, using the REDO procedure defined earlier.

The UNDO procedure is defined as follows:

Procedure 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.

As we discussed for the **NO-UNDO/REDO** procedure, 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 so that an item can be undone at most once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the forward direction (starting from

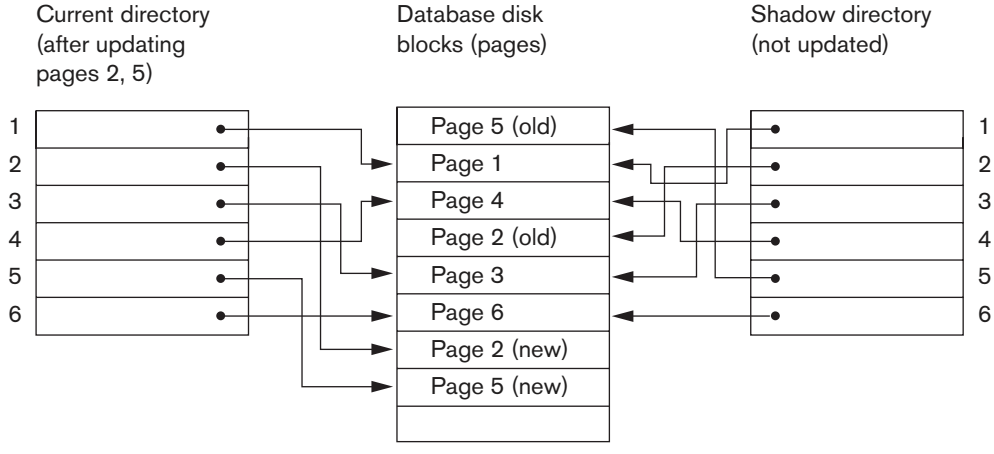
the beginning of the log). Whenever an item is undone, it is added to a list of undone items and is not undone again.

22.4 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 22.4 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.

Figure 22.4
An example of shadow paging.



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

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 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.

22.5 The ARIES Recovery Algorithm

We now describe the ARIES algorithm as an example of a recovery algorithm used in database systems. It is used in many relational database-related products of IBM. ARIES uses a steal/no-force approach for writing, and it is based on three concepts: write-ahead logging, repeating history during redo, and logging changes during undo. We discussed write-ahead logging in Section 22.1.3. 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: analysis, REDO, and 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 only to committed transactions. However, this is not the case in ARIES.

[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 (as discussed later in this section).

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. Additionally, 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 therefore does not need to be reapplied. Thus, *only the necessary REDO operations* are applied during recovery. Finally, during the **UNDO phase**, the log is scanned backward 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. Additionally, checkpointing is used. These 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. Also, 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 so that the undo does not have to be repeated. When a transaction ends, whether by committing or aborting, an *end log record* is written.

Common fields in all log records include the previous LSN for that transaction, the transaction ID, and 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 the page ID for the page that contains the item, the length of the updated item, its offset from the beginning of the page, the before image of the item, and its after image.

In addition to 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 DBMS cache, which includes the page ID and the LSN corresponding to the earliest update to that page.

Checkpointing in ARIES consists of the following: writing a begin_checkpoint record to the log, writing an end_checkpoint record to the log, and 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 22.1.4). Additionally, 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. Note that if a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which would be 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 an 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 does not need to 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 holds, 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 does not need to 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.

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	*C*	...
2	0	T_2	update	*B*	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	*A*	...
7	2	T_2	update	*C*	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	7
B	2
A	6

Figure 22.5
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.

Consider the recovery example shown in Figure 22.5. 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 22.5(a) shows the partial contents of the log, and Figure 22.5(b) shows the contents of the Transaction Table and Dirty Page Table. Now, suppose that a crash occurs at this point. 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 contains the Transaction Table and Dirty Page Table in Figure 22.5(b), 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 22.5(c) 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 than the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure 22.5(c)), 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.

22.6 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 23), 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 the 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, local 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. Conversely, 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 local 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.

22.7 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 a **database backup**, in which the whole database and the log are periodically copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices. 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.

Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations. Subterranean storage vaults have been used to protect such data from flood, storm, earthquake, or fire damage. Events like the 9/11 terrorist attack in New York (in 2001) and the Katrina hurricane disaster in New Orleans (in 2005) have created a greater awareness of *disaster recovery of critical databases*.

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. Therefore, 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.

22.8 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. First in Section 22.1 we gave an informal outline for a recovery process, and then we 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 (Section 22.2) and immediate update (Section 22.3). 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 in Section 22.4, 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 in Section 22.5, which is a specific recovery scheme used in many of IBM's relational database products. Then in Section 22.6 we discussed the two-phase commit protocol, which is used for recovery from failures involving multidatabase transactions. Finally, we discussed recovery from catastrophic failures in Section 22.7, 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

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

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

22.3. 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?

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

22.5. 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?

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

22.7. Describe the write-ahead logging protocol.

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

22.9. 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?

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

22.11. 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?

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

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

22.14. 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.

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

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

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

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

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

22.20. Discuss how disaster recovery from catastrophic failures is handled.

Exercises

22.21. Suppose that the system crashes before the [read_item, T_3, A] entry is written to the log in Figure 22.1(b). Will that make any difference in the recovery process?

22.22. Suppose that the system crashes before the [write_item, T_2, D, 25, 26] entry is written to the log in Figure 22.1(b). Will that make any difference in the recovery process?

22.23. Figure 22.6 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.

[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 22.6
A sample schedule and its corresponding log.

22.24. Suppose that we use the deferred update protocol for the example in Figure 22.6. 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.

22.25. How does checkpointing in ARIES differ from checkpointing as described in Section 22.1.4?

22.26. 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 22.5. You can make your own assumptions as to when a page is written to disk.

22.27. 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:

22.28. Incremental logging with deferred updates implies that the recovery system must

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

22.29. The write-ahead logging (WAL) protocol simply means that

a. 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

22.30. 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

22.31. For incremental logging with immediate updates, a log record for a transaction would contain

a. a transaction name, a data item name, and the old and new value of the item
b. a transaction name, a data item name, and the old value of the item
c. a transaction name, a data item name, and the new value of the item
d. a transaction name and a data item name

22.32. For correct behavior during recovery, undo and redo operations must be

a. commutative
b. associative
c. idempotent
d. distributive

22.33. 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 redos are unnecessary
c. both (a) and (b)
d. none of the above

22.34. Using a log-based recovery scheme might improve performance as well as provide 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

22.35. 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)

22.36. 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

22.37. 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. Lockemann and Knutsen (1968), Davies (1973), 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., 1992). 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 successful in practice. Franklin et al. (1992) discusses recovery in the EXODUS system. Two books by Kumar and Hsu (1998) and Kumar and Song (1998) discuss recovery in detail and contain descriptions of recovery methods used in a number of existing relational database products. Examples of page replacement strategies that are specific for databases are discussed in Chou and DeWitt (1985) and Pazos et al. (2006).