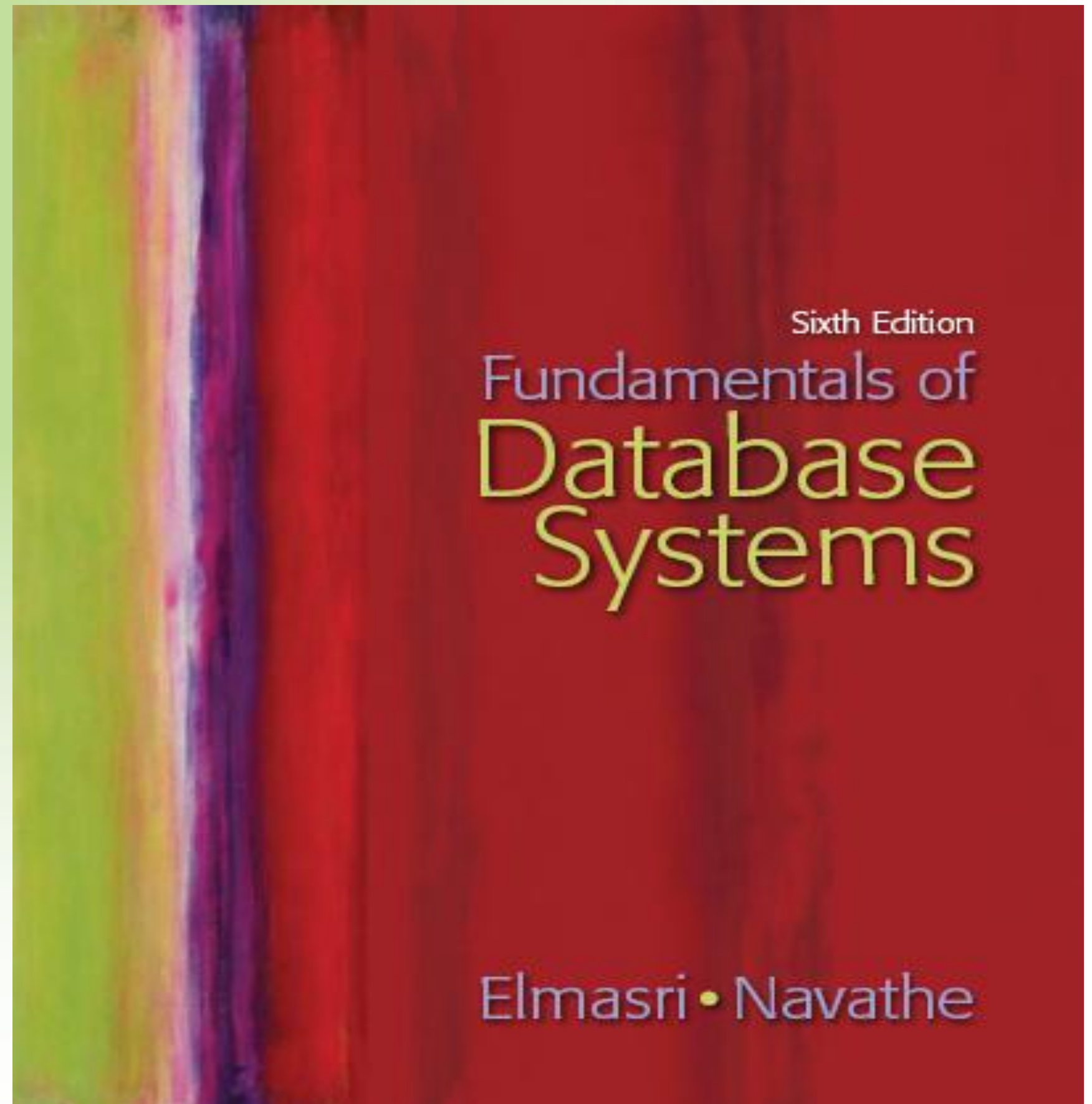


Chapter 21

Introduction to Transaction Processing Concepts and Theory



Addison-Wesley
is an imprint of

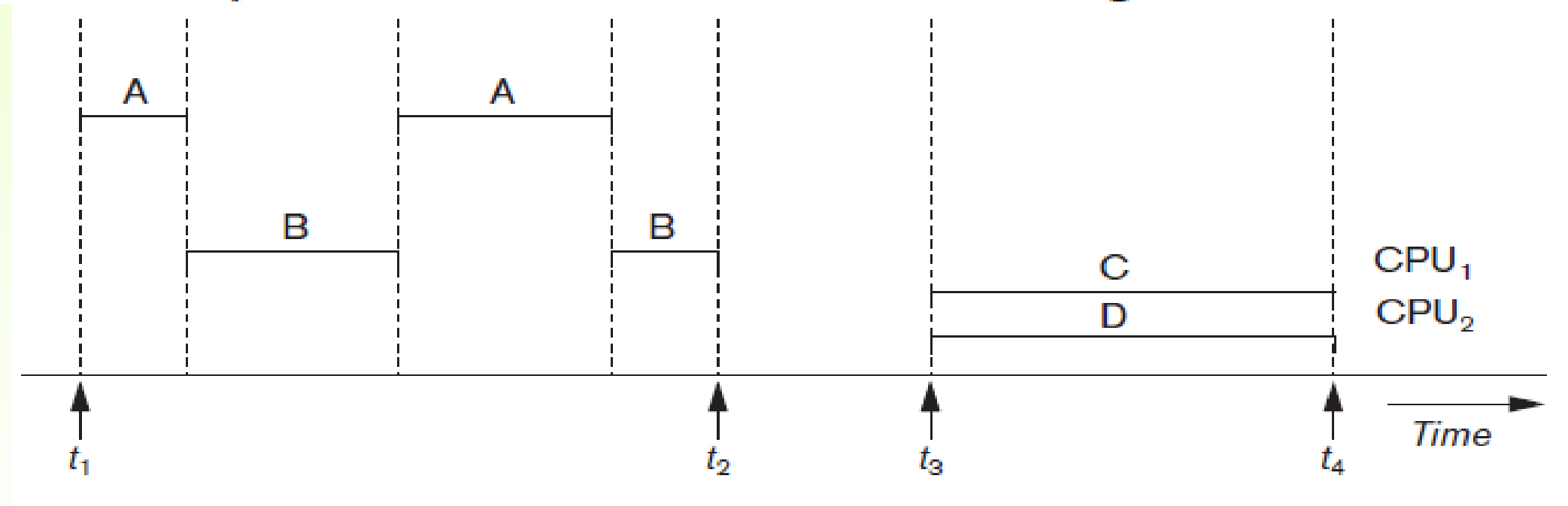
PEARSON

Copyright © 2011 Pearson Education, Inc. Publishing as Pearson Addison-Wesley

21.1.1 Single-User versus Multiuser Systems

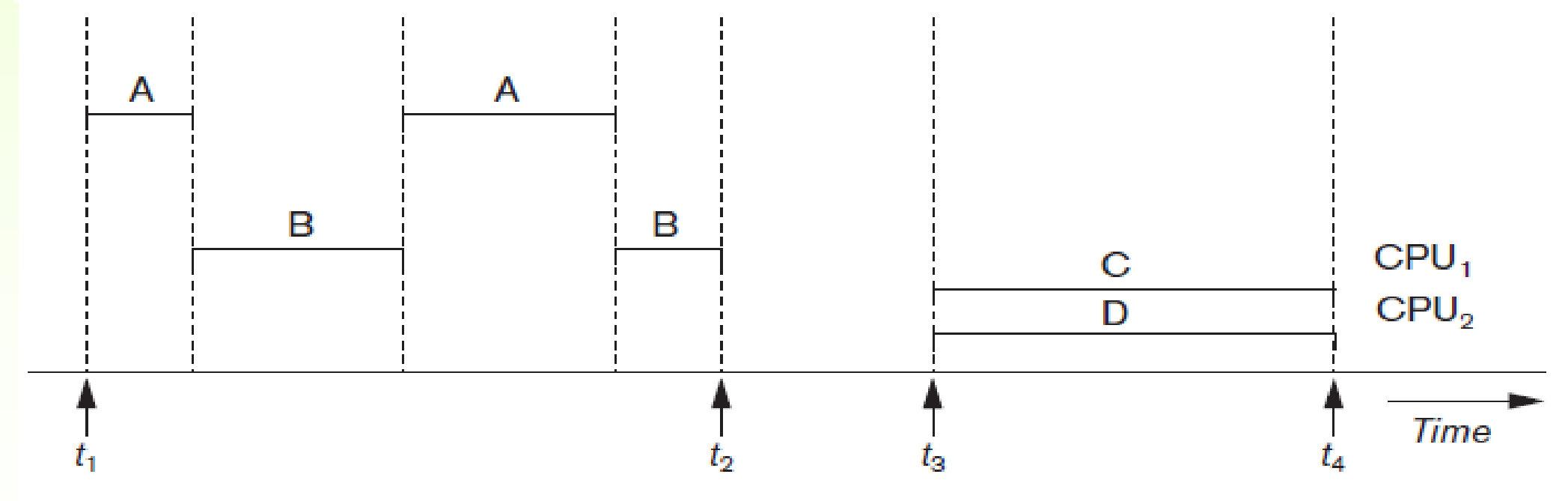
One criterion for classifying a database system is according to the number of users who can use the system concurrently. A DBMS is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks 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 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 21.1, which shows two processes, A



is actually interleaved, as illustrated in Figure 21.1, 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 21.1. 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



21.1.2 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, 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**.

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 inde-

data 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 .

Introduction to Transaction Processing (4)

READ AND WRITE OPERATIONS:

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.
- `read_item(X)` command includes the following steps:
 - **Find the address of the disk block that contains item X.**
 - **Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).**
 - **Copy item X from the buffer to the program variable named X.**

Introduction to Transaction Processing (5)

READ AND WRITE OPERATIONS (cont.):

- **write_item(X)** command includes the following steps:
 - Find the address of the disk block that contains item X.
 - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
 - Copy item X from the program variable named X into its correct location in the buffer.
 - Store the updated 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. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store 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 buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.¹

A transaction includes read_item and write_item operations to access and update the database. Figure 21.2 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 21.2 is $\{X, Y\}$ and its write-set is also $\{X, Y\}$.

(a)	T_1	(b)	T_2
	read_item(X); $X := X - N$; write_item(X); read_item(Y); $Y := Y + N$; write_item(Y);		read_item(X); $X := X + M$; write_item(X);

Figure 21.2
Two sample transactions. (a) Transaction T_1 . (b) Transaction T_2 .

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.

21.1.3 Why Concurrency Control Is Needed

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 21.3(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

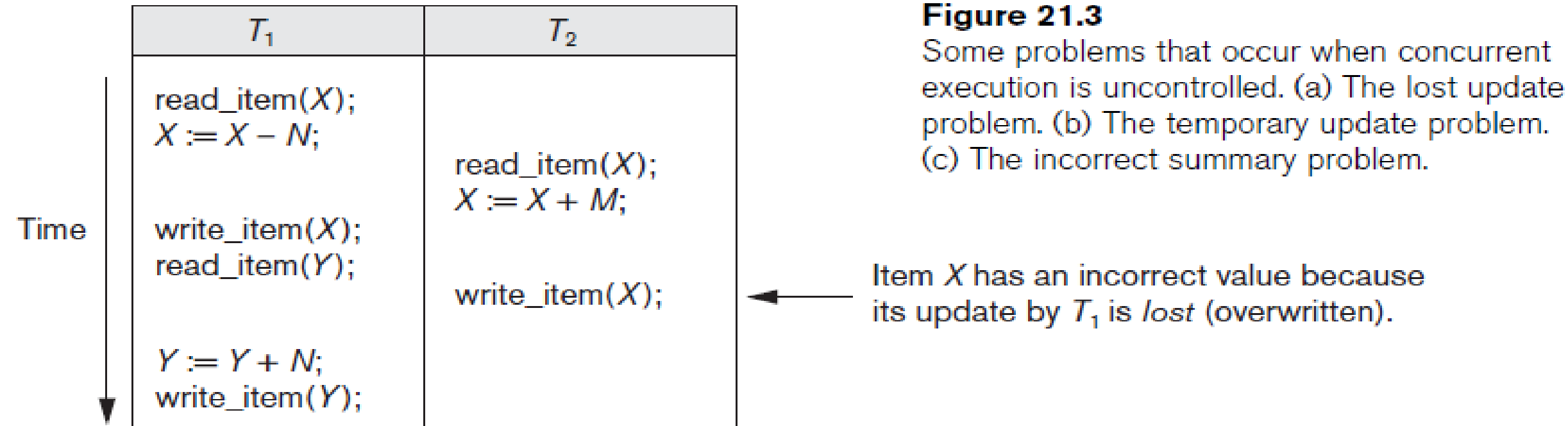



Figure 21.3

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

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 21.3(a), it is $X = 84$ because the update in T_1 that removed the five seats from X was *lost*.

T_1	T_2
read_item(X); $X := X - N$;	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	write_item(X);
$Y := Y + N$; write_item(Y);	

Final (X)= 84 

Final (X)= 79

X=80

N=5

M=4

T1

X=80

X= 80 - 5

Write(X)
= 75

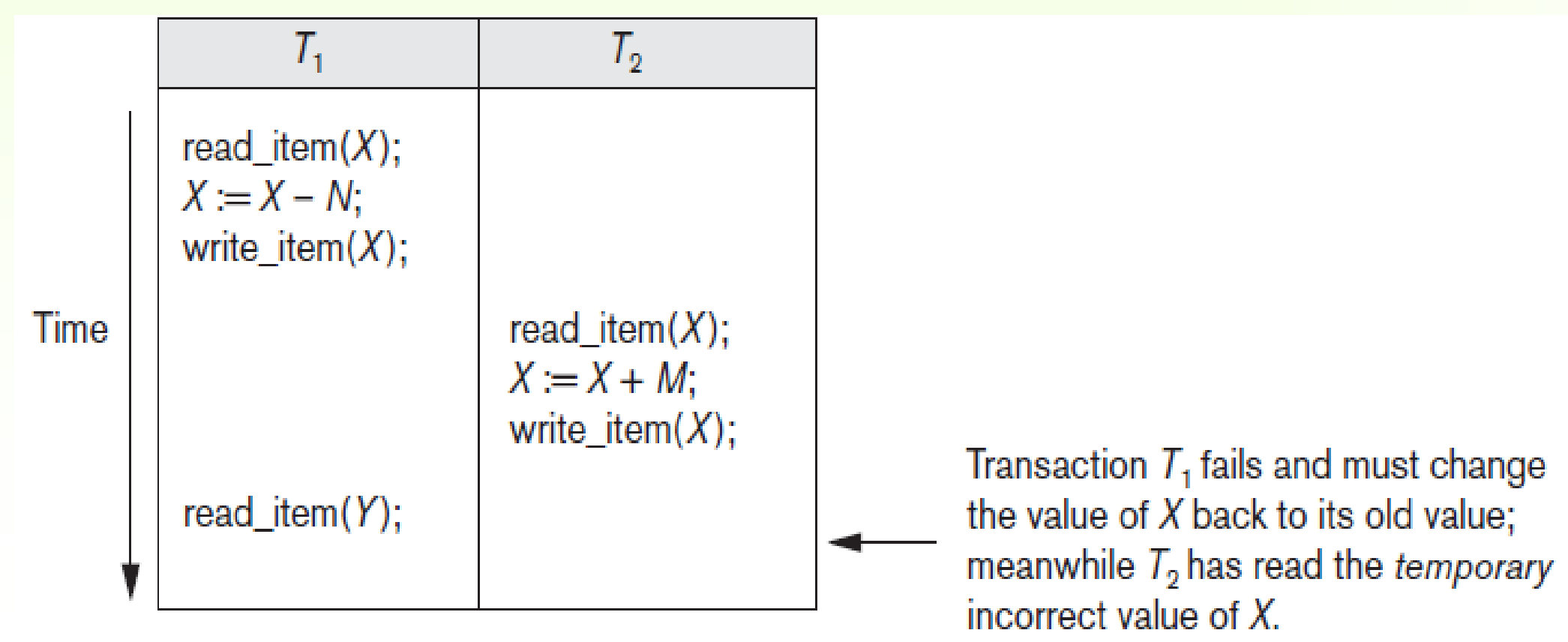
T2

X=80

X= 80 + 4

Write(X)
= 84

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 21.1.4). Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure 21.3(b) shows an example where T_1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T_2 reads the *temporary* value of X , which will not be recorded permanently in the database because of the failure of T_1 . The value of item X that is read by T_2 is called *dirty data* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem*.



The Incorrect Summary Problem. If one transaction is calculating an aggregate summary function on a number of 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 21.3(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.

T_1	T_3
	$sum := 0;$ $read_item(A);$ $sum := sum + A;$ \vdots
$read_item(X);$ $X := X - N;$ $write_item(X);$	$read_item(X);$ $sum := sum + X;$ $read_item(Y);$ $sum := sum + Y;$
$read_item(Y);$ $Y := Y + N;$ $write_item(Y);$	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

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.

21.1.4 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.

4. **Concurrency control enforcement.** The concurrency control method (see Chapter 22) may decide to abort a transaction because it violates serializability (see Section 21.5), or it may abort one or more transactions to resolve a state of deadlock among several transactions (see Section 22.1.3). 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 23.

Transaction Processing Part 2

21.2 Transaction and System Concepts

21.2.1 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 21.2.3). Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

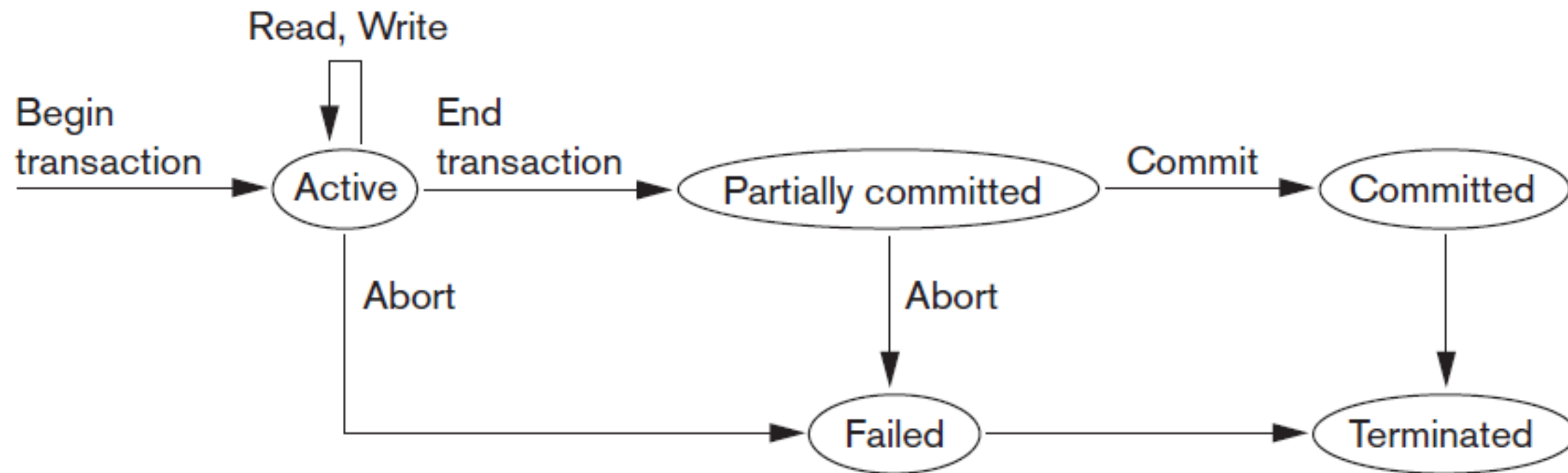
- **BEGIN_TRANSACTION.** This marks the beginning of transaction execution.
- **READ or WRITE.** These specify read or write operations on the database items that are executed as part of a transaction.
- **END_TRANSACTION.** This specifies that **READ** and **WRITE** transaction operations have ended 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 whether the transaction has to be aborted because it violates serializability (see Section 21.5) 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 21.4 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 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 Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state**.

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



Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions.

21.2.2 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 hold the last part of the log file, so that log entries are first added to the 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.

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.

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

21.2.3 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.

21.3 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 or of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints 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.

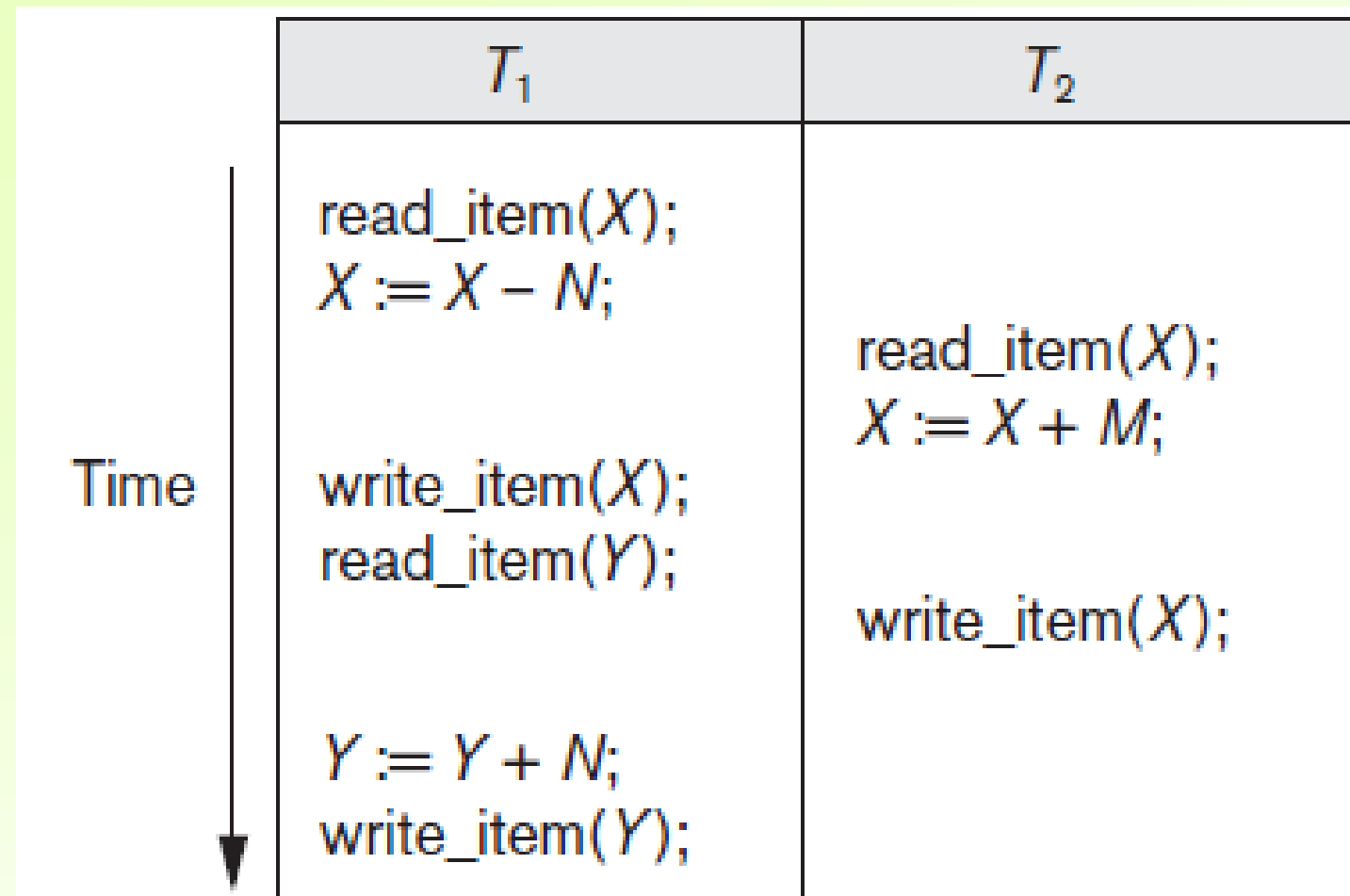
The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS.⁸ If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks (see Chapter 23)

And last, the *durability property* is the responsibility of the *recovery subsystem* of the DBMS. We will introduce how recovery protocols enforce durability and atomicity

Transaction Processing Part 3

21.4 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 21.5, we characterize schedules in terms of the interference of par-



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

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 $\text{write_item}(X)$. 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.

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

Conflic Operations

R1(x)	W2 (x)
R2(x)	W1(x)
W1(x)	W2(x)

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 order the value of X is changed by $w_2(X)$ before it is read by $r_1(X)$, whereas in the first order 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, \dots, 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, \dots, 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

21.4.2 Characterizing Schedules Based on Recoverability

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 21.3). The schedules that theoretically meet this criterion are called *recoverable schedules*; those that do not are called **nonrecoverable** and hence should not be permitted by the DBMS. The definition of

The definition of

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).

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

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

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

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

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.

A schedule is said to be cascadeless, or to avoid cascading rollback, if every transaction in the schedule reads only items that were written by committed transactions. In this case, all items read will not be discarded, so no cascading rollback will occur.

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, \dots, T_i and