CANARA ENGINEERING COLLEGE

Benjanapadavu – 574219

Program: COMPUTER SCIENCE & ENGINEERING   Course code:18CS53
Course Name: DATABASE MANAGEMENT SYSTEM

MODULE 5

Notes prepared by - (Name & Designation) : Mr.LOHIT B and Mr.SANTOSH

OBJECTIVE: Demonstrate the use of concurrency and transactions in database

OUTCOME: Explain the basic issues of transaction processing and concurrency control.

Contents include:

No. of Weblinks: 3
No. of University Qs and As: 8

Structure of Notes

5. 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.

5.1. Introduction to Transaction Processing

In this section we discuss the concepts of concurrent execution of transactions and recovery from transaction failures.

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.

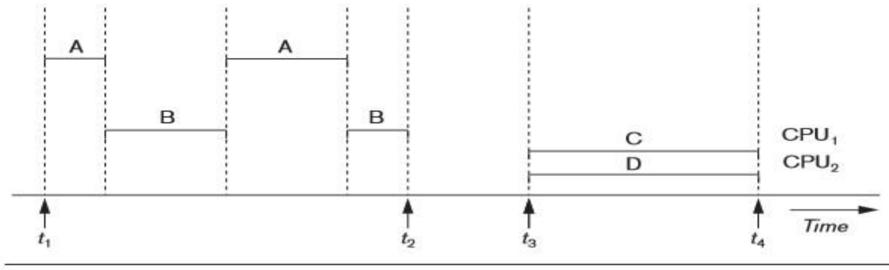


Figure 21.1 Interleaved processing versus parallel processing of concurrent transactions.

- 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 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.

5.2. 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**.
- 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 basic database access operations that a transaction can include are as follows:

1)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.

2) write_item(X). Writes the value of program variable X into the database item
named X.

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*.

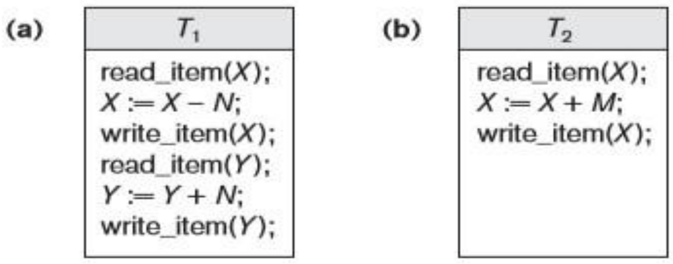
Executing a write_item(X) command includes the following steps:

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).

5.3. 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 21.2(a) shows a transaction T1 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 21.2(b) shows a simpler transaction T2 that just reserves M seats on the first flight (X) referenced in transaction T1.2


(a) T_1

```
read_item(X);
X := X − N;
write_item(X);
read_item(Y);
Y := Y + N;
write_item(Y);
```

(b) T_2

```
read_item(X);
X := X + M;
write_item(X);
```

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

Next we discuss the types of problems we may encounter with these two simple transactions if they run concurrently.

1)**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 T1 and T2 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 T2 reads the value of X before T 1 changes it in the database, and hence the updated value resulting from T1 is lost.For example,if X= 80 at the start (originally there were 80 reservations on the flight),N= 5 (T1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and M = 4 (T2 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 T1 that removed the five seats from X was lost.

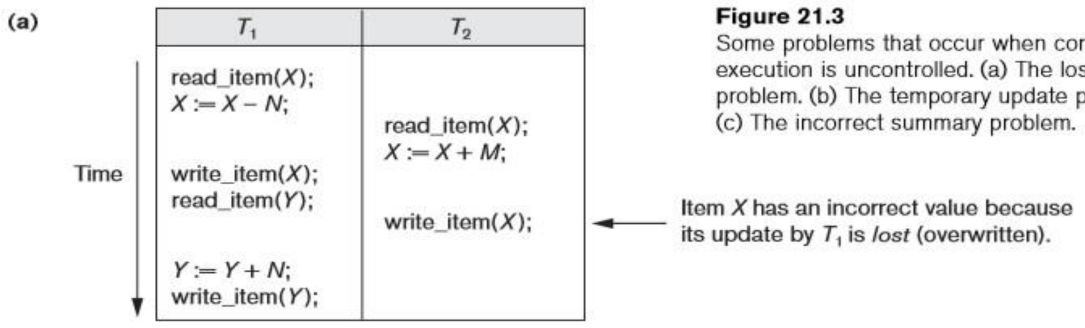
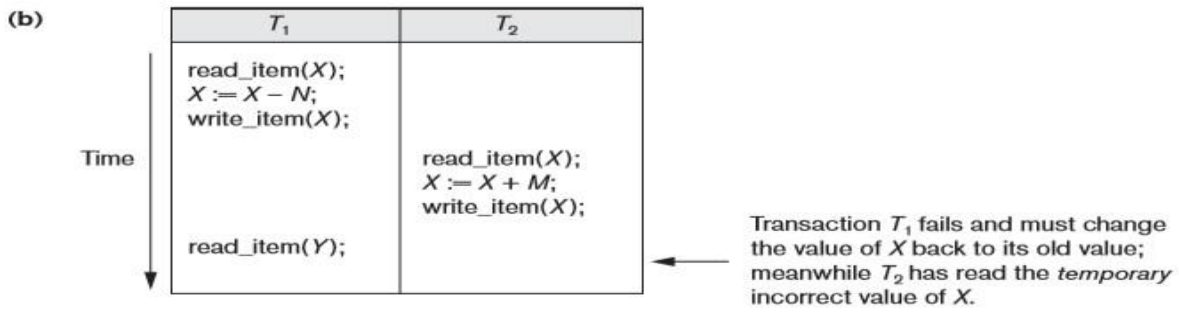


Figure 21.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

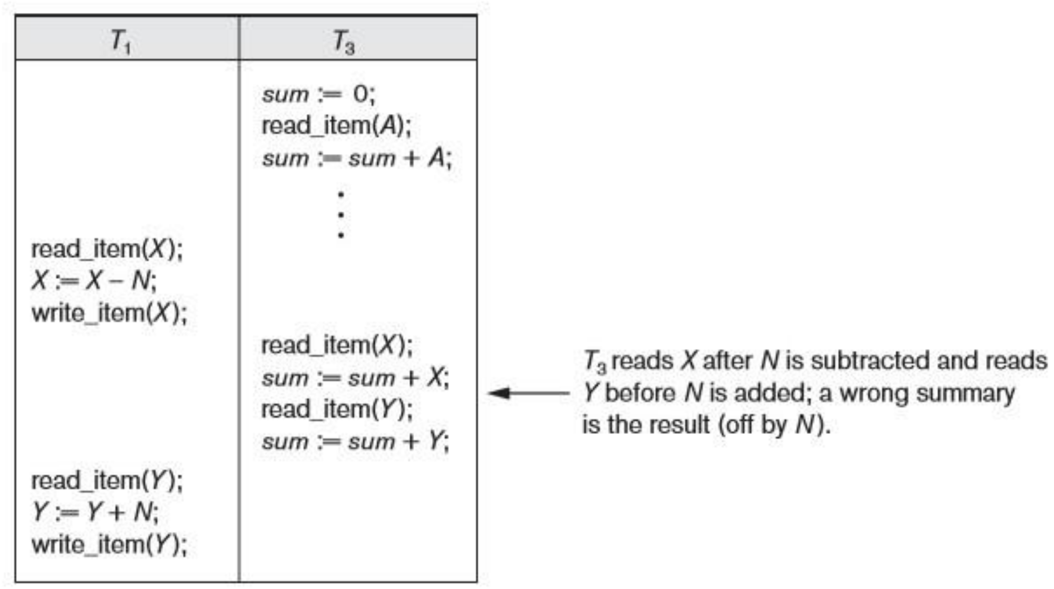
2) **The Temporary Update (or Dirty Read) Problem.** This problem occurs when one transaction updates a database item and then the transaction fails for some reason .Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure 21.3(b) shows an example where T1 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 T2 reads the temporary value of X, which will not be recorded permanently in the database because of the failure of T1.The value of item X that is read by T2 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**.


(b)
T_1
T_2
read_item(X);
$X := X - N$;
write_item(X);
Time
read_item(X);
$X := X + M$;
write_item(X);
read_item(Y);
Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

3) **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 T3 is calculating the total number of reservations on all the flights; meanwhile, transaction T1 is executing. If the interleaving of operations shown in Figure 21.3(c) occurs, the result of T3 will be off by an amount N because T3 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.

(c)



T_1	T_3

	sum := 0;
	read_item(*A*);
	sum := *sum* + *A*;
	⋮

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*).

4)**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 Tbetween 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.

5.4 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. 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, 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

4. **Concurrency control enforcement.** The concurrency control method may abort a transaction because it violates serializability or it may abort one or more transactions to resolve a state of deadlock among several transactions. 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.

5.5 Transaction and System Concepts

In this section we discuss additional concepts relevant to transaction processing.

5.5.1 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. 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. 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 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
State transition diagram illustrating the states for transaction execution.

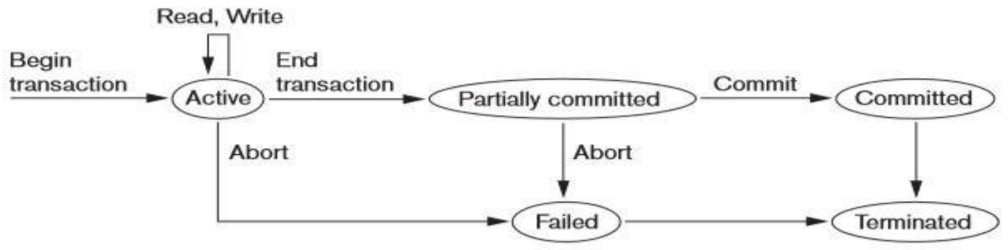


Figure 21.4 shows a state transition diagram that illustrates how a transaction moves through its execution states .A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the **partially 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). 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 **aborte**d 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.

5.5.2 The System Log

- To be able to recover from failures that affect transactions, the system maintains a **log** 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 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 the 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.

5.5.3 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

WRITEoperations in the log,so their effect on the database can be redone from the log records.

5.5.4 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 a transaction is executed 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.

- 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.

5.6 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***).***

Schedules (Histories) of Transactions

A **schedule** (or **history**) *S* of *n* transactions T1, T2, ..., Tn is an ordering of the operations of the transactions subject to the constraint that, for each transaction Ti that participates in *S*, the operations of Ti in *S* must appear in the same order in which they occur in Ti. Note, however, that operations from other transactions Tj can be interleaved with the operations of Ti in *S*. For now, consider the order of operations in *S* to be a *total ordering,* although it is possible theoretically to deal with schedules whose operations form *partial orders*.

S_a: (X) ; r2 (X) ; (X) ; (Y) ; w2 (X) ; (Y) ;

Similarly, the schedule for Figure 21.3(b), which we call Sb, can be written as follows, if we assume that transaction T1 aborted after its read_item(*Y*) operation:

***Sb:* r_1 *(X)* ; *w1* *(X)* ; r2 *(X)* ; w2 *(X)* ;(Y) ; al;**

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_c, the operations $r_i(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and wi(X), and the operations $w_i(X)$ and $w_2(X)$. But the operations ri(X) and $r_2(X)$ do not conflict, since they are both read distinct data items *X* and Y; and the operations $r_i(X)$ and $w_i(X)$ do not conflict because they belong to the same transaction.

A schedule *S* of *n* transactions T1, T2, ..., Tn, is said to be a **complete schedule** if the following conditions hold:

1. The operations in *S* are exactly those operations in T1, T2, ..., Tn, 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 Ti, their order of appearance in *S* is the same as their order of appearance in Ti.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

5.7 Characterizing Schedules Based on Recoverability

- 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).*
- If sufficient information is kept (in the log), a recovery algorithm can be devised for any recoverable schedule.

❖ The (partial) schedules S_c, and *Sb* from are both recoverable, since they satisfy the above definition. Consider the schedule *S,* 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)* ; r2 *(X) ; w1 (X) ; r1(Y) ; w2 (X) ; c2; w1 (Y) ; ci;*

S_c,' is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory.

Consider the two (partial) schedules Sy, and *Sd:*

S_c : r1 (X) ; w1 (X) ; r2 (X) ; r1(Y) ; w2 (X) ; c2; al;

***Sd* : r₁ (X) ; w1 (X) ; r2 (X) ; r1(Y) ; w2 (X) ; w₁ (Y)**

; c_1; c2; S_e : r_1 (x) ; w_1 (X) ; r2 (X) ; r1(Y) ; w2 (X) ;(Y) ; al; a2;

5, is not recoverable because *T2* reads item *X* from T_1, but *T2* commits before T_i commits. The

problem occurs if T_1 aborts after the c_2 operation in S_c; then the value of *X* that *T2* read is no longer

valid and *T2* must be aborted *after* it is committed, leading to a schedule that is *not recoverable*. For

5.8 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.**

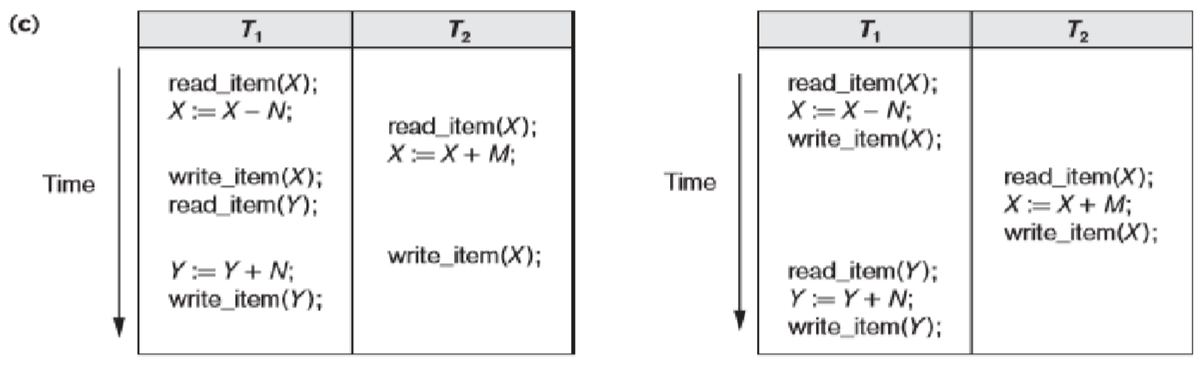
Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T1 and T2 in Figure 21.2 at approximately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

1. Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).
2. Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).

These two schedules—called **serial schedules**—are shown in Figure 21.5(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 21.5



(c)

Time

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);	

Schedule C

Time

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

Schedule D

Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figure 21.5(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: T1 and then T2 in Figure 21.5(a),and T2 and then T1 in Figure 21.5(b). Schedules C and D in Figure 21.5(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 Tare 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.

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 operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. Additionally, if some transaction T is quite long, the other transactions must wait for T to complete all its operations before starting. Hence, serial schedules are considered unacceptable in practice.

consider the schedules in Figure 21.5, 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 T1 and T2,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 21.3(a)) gives the results X= 92 and Y= 93,in which the Xvalue is erroneous,whereas schedule D gives the correct results.

Schedule C gives an erroneous result because of the lost update problem. Transaction T2 reads the value of X before it is changed by transaction T1,so only the effect of T2 on X is reflected in the database.The effect of T1 on Xislost,overwritten by T2,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 sche**dule is as follows: A schedule S of n transactions is **serializabl**e if it is equivalent to some serial schedule of the same n transactions.

There are several ways to define **schedule equivalence**. Two schedules are called **result equivalen**t 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 21.6, schedules S1 and S2 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.

Figure 21.6
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

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

The definition of **conflict equivalence** of schedules is as follows: Two schedules are said to be **conflict equivalent** if the order of any two conflicting operations is the same in both schedules. Two operations in a schedule are said to 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.

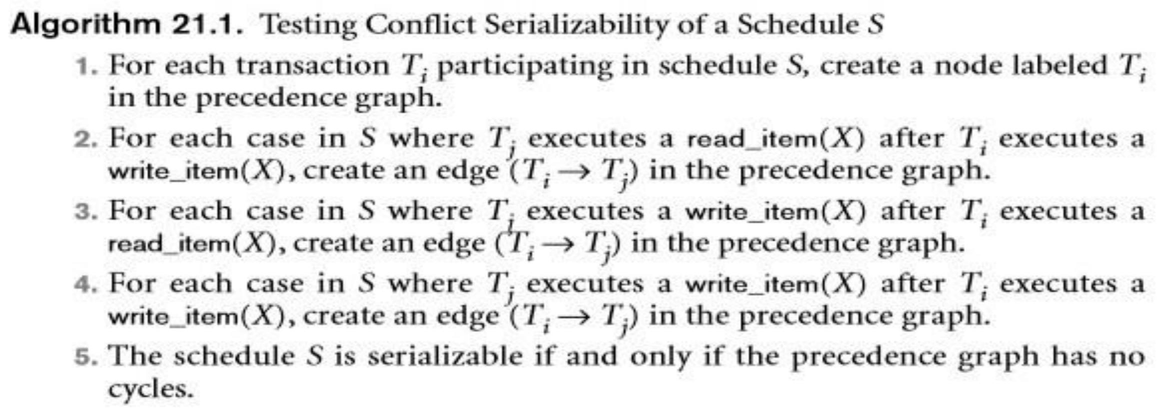
Using the notion of conflict equivalence, we define a schedule S to be **conflict serializable** if it is (conflict) equivalent to some serial schedule S. In such a case, we can reorder the nonconflicting operations in S until we form the equivalent serial schedule S.According to this definition, schedule D in Figure 21.5(c) is equivalent to the serial schedule A in Figure 21.5(a).

Schedule C in Figure 21.5(c) is not equivalent to either of the two possible serial schedules A and B, and hence is **not serializable.**

Testing for Conflict Serializability of a Schedule

There is a simple algorithm for determining whether a particular schedule is conflict serializable or not.

Algorithm 21.1 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= {T1,T2,...,Tn } and a set of directed edges E= {e1, e2,...,em }. There is one node in the graph for each transaction Ti in the schedule. Each edge ei in the graph is of the form (Tj→Tk),1 ≤j≤n,1 ≤kfn, where Tj is the starting node of ei and Tk is the ending node of ei. Such an edge from node Tj to node Tk is created by the algorithm if one of the operations in Tj appears in the schedule before some conflicting operation in Tk.



Algorithm 21.1. 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 21.1. If there is a **cycle** in the precedence graph, schedule S is **not (conflict) serializable**; if there is **no cycle**, S is **serializable**.

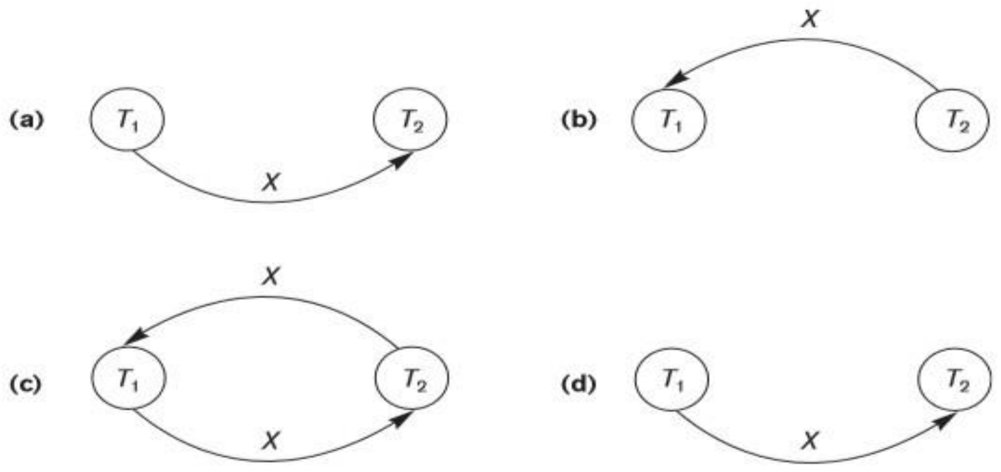


Figure 21.7
Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

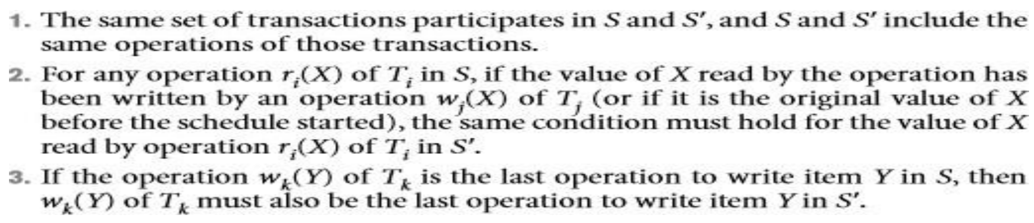
How Serializability Is Used for Concurrency Control

A serializable schedule gives the benefits of concurrent execution without giving up any correctness.In practice,it is quite difficult to test for the serializability of a schedule. The interleaving of operations from concurrent transactions—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler,which allocates resources to all processes. Factors such as system load, time of transaction submission, and priorities of processes contribute to the ordering of operations in a schedule. Hence,it is difficult to determine how the operations of a schedule will be interleaved beforehand to ensure serializability.

View Equivalence and View Serializability

view equivalence :less restrictive definition of equivalence of schedules is called **view equivalence.**

view serializability: Two schedules S and Sare 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'.

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.

A blind write is a write operation in a transaction T on an item X that is not dependent on the 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 wi(X) in Ti can be independent of its old value from the database.This is possible when blind writesare allowed,and it is illustrated by the following schedule Sg of three transactions T1: r1(X); w1(X);

T2: w2(X); and T3: w3(X):

Sg: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;

In Sg the operations w2(X) and w3(X) are blind writes, since T2 and T3 do not read the value of X. The schedule Sg is view serializable, since it is view equivalent to the serial schedule T1, T2, T3. However, Sg is not conflict serializable, since it is not conflict equivalent to any serial schedule.

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:

T1: r1(X); X := X−10; w1(X); r1(Y); Y := Y + 10; w1(Y);

T2: r2(Y); Y := Y−20; w2(Y); r2(X); X := X + 20; w2(X);

Consider the following nonserializable schedule Sh for the two transactions: **Sh: r1(X); w1(X); r2(Y); w2(Y); r1(Y); w1(Y); r2(X); w2(X);**

With the additional knowledge, or semantics, that the operations between each ri(I) and wi(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 Ti on a particular item I is not interrupted by conflicting operations. Hence, the schedule Sh is considered to be correct even though it is not serializable.

5.9 Transaction Support in SQL

- ❖ A transaction 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. But every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK.
- ❖ Transaction 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 is for data retrieval.

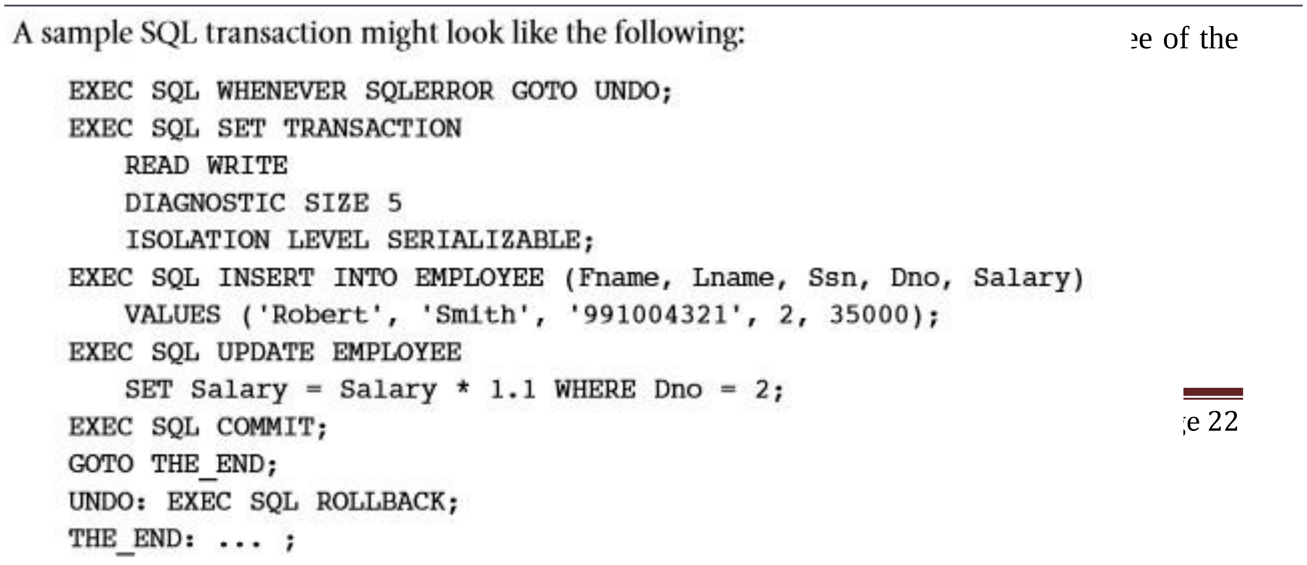
- The **diagnostic area size** option, DIAGNOSTIC SIZE *n*, specifies an integer value *n*, which indicates the number of conditions that can be held simultaneously in the diagnostic area These conditions supply feedback information (errors or exceptions) to the user or program on the *n* most recently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

+ The **isolation level** option is specified using the statement ISOLATION LEVEL <is olat ion>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE. The default isolation level is SERIALIZABLE, although some systems use READ COMMITTED as their default. The use of the term SERIALIZABLE is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms.

- 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_i may read the update of a transaction T2, which has not yet committed. If T_2 fails and is aborted, then *T1* would have read a value that does not exist and is incorrect.
2. **Nonrepeatable read:** A transaction Ti may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T_i will see a different value.

Phantoms: A transaction T1 may read a set of rows from a table based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T2 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** that a violation is possible and an entry of No indicates that it is not possible. READ UNCOMMITTED is


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: ... ;
```

e of the

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.

Concurrency Control inDatabases

In this chapter we discuss a number of concurrency control techniques that are used to ensure the non interference or isolation property of concurrently executing transactions. Most of these techniques ensure serializability of schedules.

5.10 Two-Phase Locking(2PL) 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.

Types of Locks and System Lock Tables

1. Types of Locks and System Lock Tables:

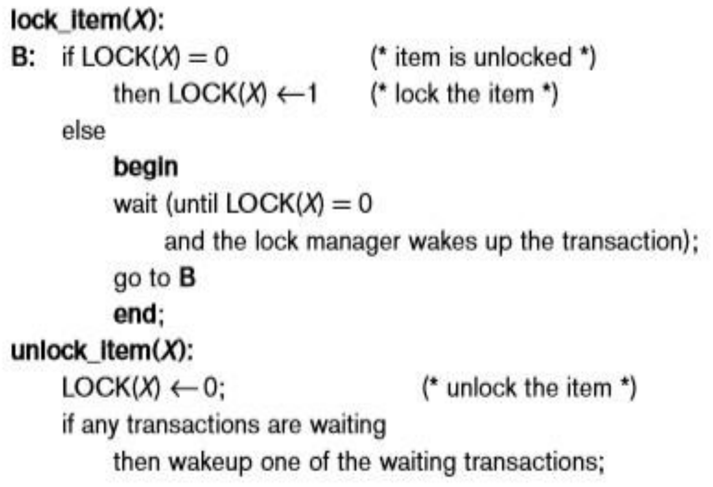
Types of locks used in concurrency control are binary locks, shared/exclusive locks—also known as read/write locks and a certify lock that improves performance of locking protocols.

Binary Locks:

- A binary lock can have two states or values: *locked* and *unlocked* (or 1 and 0).
- 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.
- The current value (or state) of the lock associated with item X is referred to as lock (X).
- Two operations, I ock_item and unlock_item, are used with binary locking.

- A transaction requests access to an item X by first issuing a lock_it em (X) operation. If LOCK (X) = 1, the transaction is forced to wait. If LOCK (X) = 0, 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.

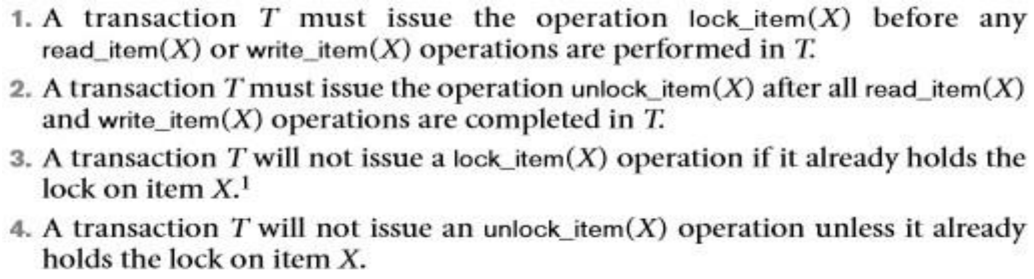
.



```
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 22.1
Lock and unlock operations for binary 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.

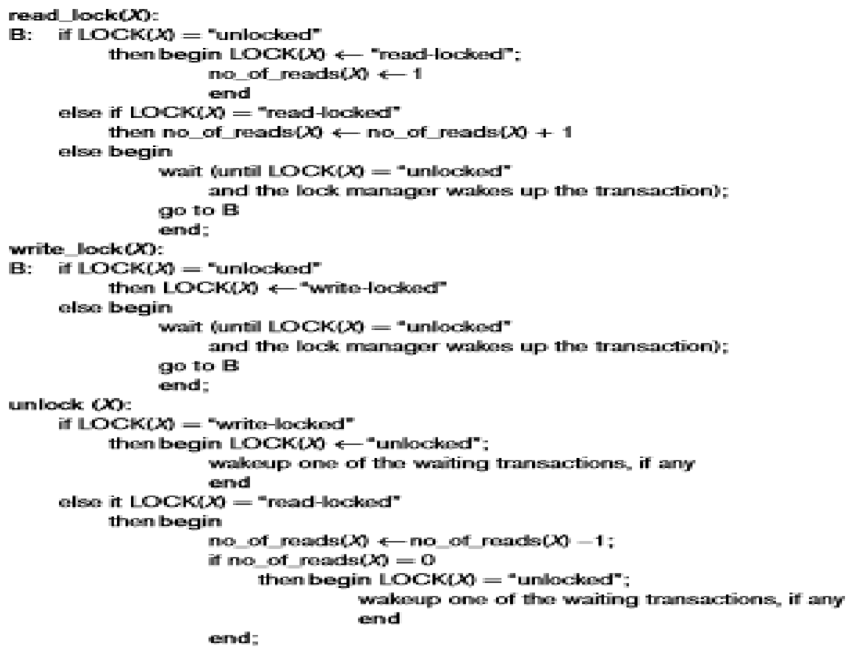
Shared/Exclusive (or Read/Write) Locks: The preceding binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item X if they all access X for reading purposes only. This is because read operations on the same item by different transactions are not conflicting. 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.

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

Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert the lock from one locked state to another**. For example, it is possible for a transaction T to issue a read_lock(X) and then later 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



```
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 22.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

5.11 Guaranteeing Serializability by Two-Phase Locking

❖ A transaction is said to follow the two-phase locking protocol if all locking operations (readlock, write_lock) precede the first unlock operation in the transaction.

❖ 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;

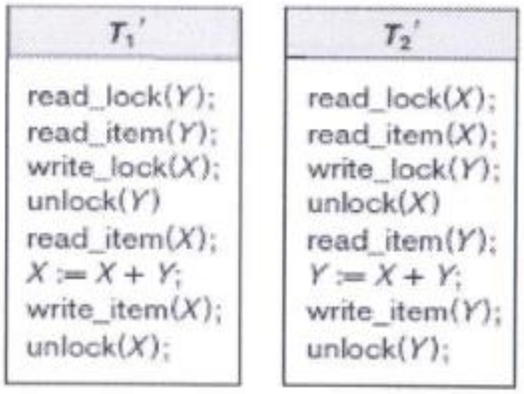
– a shrinking (second) phase, during which existing locks can be released but no new locks can

be acquired.

If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the shrinking phase.

❖ Transactions Ti and T2 in Figure 12(a) do not follow the two-phase locking protocol because the write lock (X) operation follows the unlock (Y) operation in T1, and similarly the write lock (Y) operation follows the unlock (X) operation in T2.

- ❖ If two-phase locking is enforced, the transactions can be rewritten as T_1' and T2', as shown in Figure 13. Now, the schedule shown in Figure 12(c) is not permitted for Ti' and T2' (with their modified order of locking and unlocking operations) because T_i' will issue its write_lock (X) before it unlocks item Y; consequently, when T2' issues its read_lock (X) , it is forced to wait until Ti' releases the lock by issuing unlock (X) in the schedule. However, this can lead to deadlock.



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

Variations(Types of two phase locking) of two-phase locking (2PL).

Basic, Conservative, Strict, and Rigorous Two-Phase Locking:

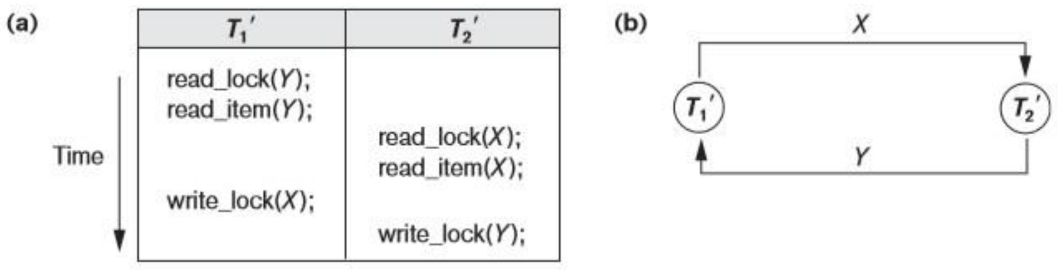
- ❖ Conservative 2PL (or static 2PL) requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring it's read-set and write-set.
- ❖ The read-set of a transaction is the set of all items that the transaction reads, and the write-set is the set of all 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. But 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.

•:• Strict 2PL guarantees strict schedules where 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.
- Rigorous 2PL 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.
- Difference between strict and rigorous 2PL: the former holds write-locks until it commits, whereas the latter holds all locks (read and write).
- Difference between conservative and rigorous 2PL: the former must lock all its items before it starts, so once the transaction starts it is in its shrinking phase; 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.

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 14(a), where the two transactions T1' and T2' are deadlocked in a partial schedule; Ti' is in the waiting queue for X, which is locked by T_2', whereas T2' is in the waiting queue for Y, which is locked by Ti'. Meanwhile, neither T1' nor T2' nor any other transaction can access items X and Y.



T_1'	T_2'
read_lock(Y); read_item(Y);	
	read_lock(X); read_item(X);
write_lock(X);	
	write_lock(Y);

Figure 22.5
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).

- 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.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.

❖ Some of the deadlock prevention 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_i starts before transaction T2, then

$TS(T_i) < TS(T2)$.

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, tries to lock an item X but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rules followed by these schemes are:

- Wait-die. If TS (T1) < TS (Ti), then (T, older than. T_j) T, is allowed to wait; otherwise (Ti younger than T_j) abort I', (T_1 dies) and restart it later with the same timestamp.
- Wound-wait. If TS (Ti) < TS (Ti) , then (T, older than T_j) abort T_1 (T, wounds T_j) and restart it

later with the same timestamp; otherwise (Ti younger than Tj) T, is allowed to wait.

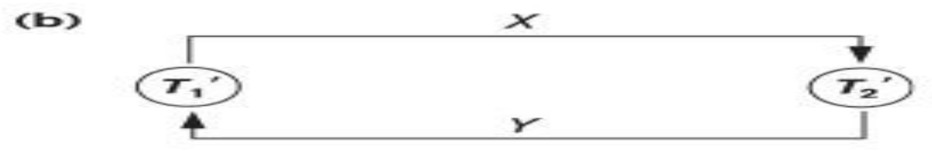
Deadlock Detection:

- In deadlock detection, the system checks if a state of deadlock actually exists. This can happen if different transactions rarely access the same items at the same time, or if transactions are short and each transaction locks only a few items, or if the transaction load is light.
- 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_r a directed edge (T, T_I) is created in the wait-for graph. When T_i releases the lock(s) on the items that
  T_i was waiting for, the directed edge is dropped from the wait-for graph. A state of deadlock occurs if and only if the wait-for graph has a cycle.

This approach has the problem of determining when the system should check for a deadlock. It can be checked 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 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 14 (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 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.**

Figure **22.5(b)(below figure)** shows the wait-for graph for the (partial) schedule shown in Figure 22.5(a).



5.12 Concurrency Control Based on Timestamp Ordering

A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule. we discuss how serializability is enforced by ordering transactions based on their timestamps.

Timestamps

A timestamp is a unique identifier created by the DBMS to identify a transaction. 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.

- ❖ Concurrency control techniques based on timestamp ordering do not use locks. Hence, deadlocks cannot occur.
- ❖ Timestamps can be generated in several ways.
  - ■ 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.

- 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.

The Timestamp Ordering Algorithm for Coneurrency Control:

- ❖ This scheme enforces 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).
- ❖ Timestamp ordering differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In "timestamp ordering, 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.

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) .

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 that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation delayed until the transaction Tthat 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 Tuntil Tis either committed or aborted. This algorithm does not cause deadlock, since T waits for Tonly if TS(T) > TS(T).

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

1. If read TS (X) > TS (T) , then abort and roll back T and reject the operation.
2. If write TS (X) > TS (T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS (T) —and hence after T in the timestamp ordering—has already written the value of X. 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) .

Multiversion Concurrency Control Techniques

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as multiversion concurrency control, because several versions (values) of an item are maintained. When a transaction requires access to an item, an appropriate version is chosen to maintain

the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an older version of the item to maintain serializability.When a transaction writes an item, it writes a new version and the old version(s) of the item are 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.

Several multiversion concurrency control schemes have been proposed. We discuss two schemes here,one based **on timestamp ordering** and the other based **on 2PL.**

Multiversion Technique Based on Timestamp Ordering

In this method, several versions X1, X2, ..., Xk of each data item X are maintained. For each version, the value of version Xi and the following two timestamps are kept:

1. read_TS(Xi).Theread timestampof Xi is the largest of all the timestamps of transactions that have successfully read version Xi.

2. write_TS(Xi). The write timestamp of Xi is the timestamp of the transaction that wrote the value of version Xi.

Whenever a transaction Tis allowed to execute a write_item(X) operation,a new version Xk+1 of item X is created, with both the write_TS(Xk+1) and the read_TS(Xk+1) set to TS(T).

Correspondingly, when a transaction T is allowed to read the value of version Xi,the value of read_TS(Xi) is set to the larger of the current read_TS(Xi) 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(Xi) of all versions of Xthat is also less than or equal toTS(T), and read_TS(Xi) > TS(T), then abort and roll back transaction T; otherwise, create a new version Xj of X with read_TS(Xj) = write_TS(Xj) = TS(T).

2. If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T); then return the value of Xi to transaction T, and set the value of read_TS(Xi) to the larger of TS(T) and the current read_TS(Xi).

Introduction to Database Recovery Protocols

In this chapter we discuss some of the techniques that can be used for database recovery from failures. This chapter presents additional concepts that are relevant to recovery protocols, and provides an overview of the various database recovery algorithms.

5.13 Recovery Concepts

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.

❖ 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 non-catastrophic failure 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 non-catastrophic 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.

- ❖ Two main policies for recovery from non-catastrophic 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. 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 database on disk. Hence, deferred update is also known as the <u>NO-UNDO/REDO</u> algorithm.
- ❖ In the immediate update techniques, the database may be updated by some operations of a transaction before the transaction reaches its commit point. 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

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.

•:• 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.

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. This can be a table of <Diskpageaddress , 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, cache. It may be necessary to replace (or flush) some of the cache buffers to make space available for the new item.

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.

- Two main strategies can be employed when flushing a modified buffer back to disk.
  - **in-place updating** writes the buffer to the same original disk location, thus overwriting the old value of any changed data items on disk. Hence, a single copy of each database disk block is maintained.
  - **Shadowing** strategy 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.

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. 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.

Two types of log entry information included for a write command: the information needed for UNDO 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 is possible, read_item entries in the log are considered to be UNDO-type entries.

❖ 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.

❖ The deferred update (NO-UNDO) recovery scheme follows a no-steal approach. However, typical database systems employ a steal/no-force (UNDO/REDO) strategy.

❖ Advantage of steal: It avoids the need for a very large buffer space to store all updated pages in memory.

Advantage of no-force: 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. Maintaining these lists makes the recovery process more efficient.

4. Checkpoints in the System Log and Fuzzy Checkpointing:

- 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 *in* or *t* are system parameters.
- Taking a checkpoint consists of the following actions:
    1. Suspend execution of transactions temporarily.
    2. Force-write all main memory buffers that have been modified to disk.
    3. Write a [checkpoint] record to the log, and force-write the log to disk.
    4. Resume executing transactions.

  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.

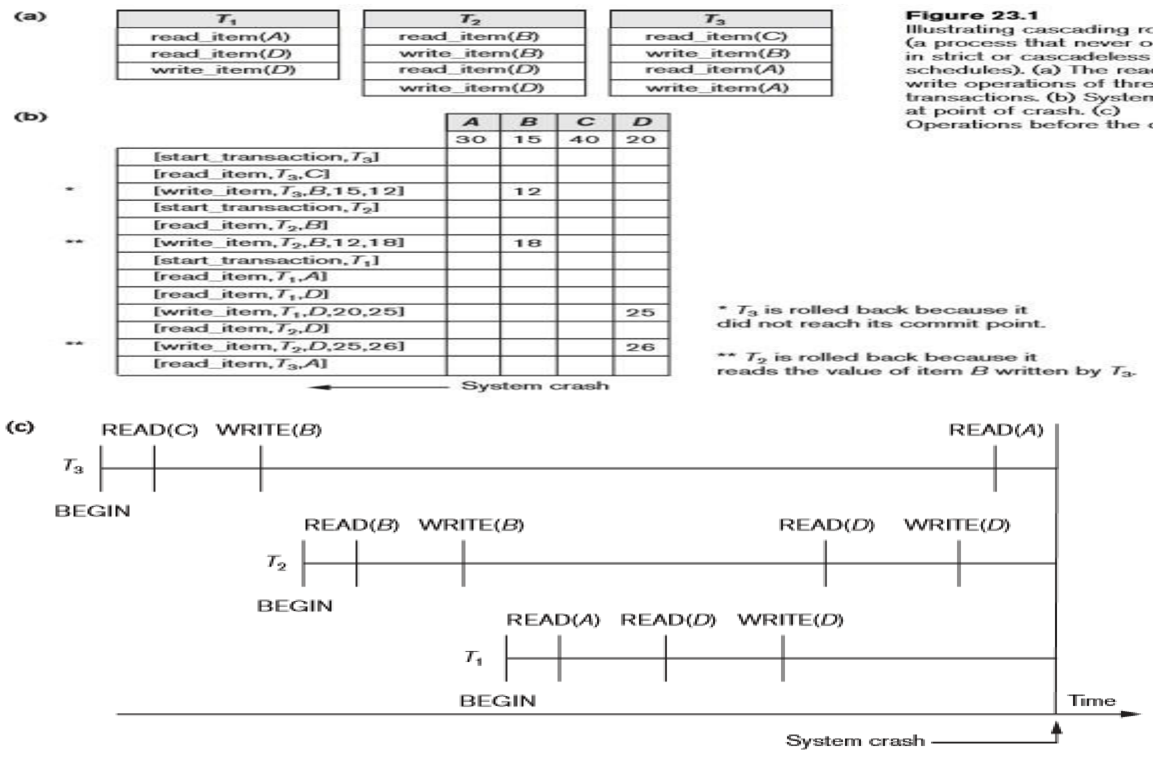
- The time needed to force-write all modified memory buffers may delay transaction processing because of step 1, which is not acceptable in practice. To overcome this, it is common to use a technique called

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. All recovery mechanisms are designed so that cascading rollback is never required.

- Figure 23 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown'in Figure 23(a).



(a)

T_1	T_2	T_3
read_item(*A*)	read_item(*B*)	read_item(*C*)
read_item(*D*)	write_item(*B*)	write_item(*B*)
write_item(*D*)	read_item(*D*)	read_item(*A*)
	write_item(*D*)	write_item(*A*)

(b)

		A	*B*	*C*	*D*
		30	15	40	20
	[start_transaction, T_3]				
	[read_item, T_3, C]				
*	[write_item, T_3, *B*, 15, 12]		12		
	[start_transaction, T_2]				
	[read_item, T_2, *B*]				
**	[write_item, T_2, *B*, 12, 18]		18		
	[start_transaction, T_1]				
	[read_item, T_1, *A*]				
	[read_item, T_1, *D*]				
	[write_item, T_1, *D*, 20, 25]				25
	[read_item, T_2, *D*]				
**	[write_item, T_2, *D*, 25, 26]				26
	[read_item, T_3, *A*]				

System crash

* T_3 is rolled back because it did not reach its commit point.

** T_2 is rolled back because it reads the value of item *B* written by T_3.

Figure 23.1
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.

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.

5.14 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.

- 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.

❖ This may simplify the recovery process, but it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held in the cache buffers until the commit point, so many cache buffers will be pinned and cannot be replaced.

❖ A typical deferred update protocol is stated 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.

The 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

Figure 23.2 illustrates a timeline for a possible schedule of executing transactions

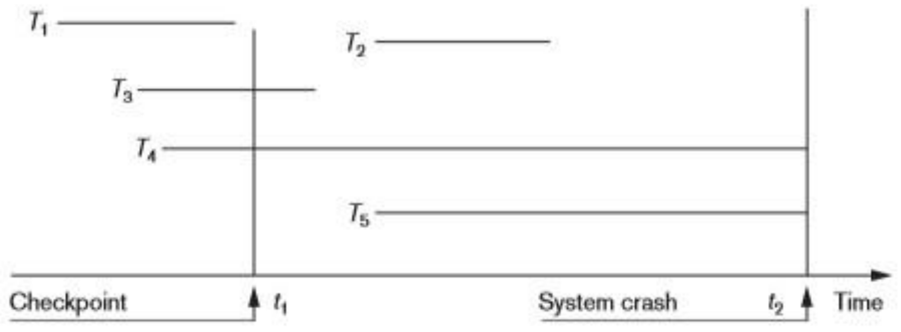


Figure 23.2
An example of a recovery timeline to illustrate the effect of checkpointing.

When the checkpoint was taken at time t1, transaction T1 had committed, whereas transactions T3 and T4 had not. Before the system crash at time t2, T3 and T2 were committed but not T4 and T5.According to the RDU_M method,there is no need to redo the write_item operations of transaction T1—or any transactions committed before the last checkpoint time t1. The write_item operations of T2 and T3 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 T4 and T5 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.

5.15 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.
- ❖ It is not a requirement that every update be 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.

❖ 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.

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 ***steaUnoforce*** strategy is applied. This is also the most complex technique, but the most commonly used in . practice.

❖ 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. A strict schedule does not allow a transaction to read or write an item unless the transaction that wrote the item has committed. But 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.

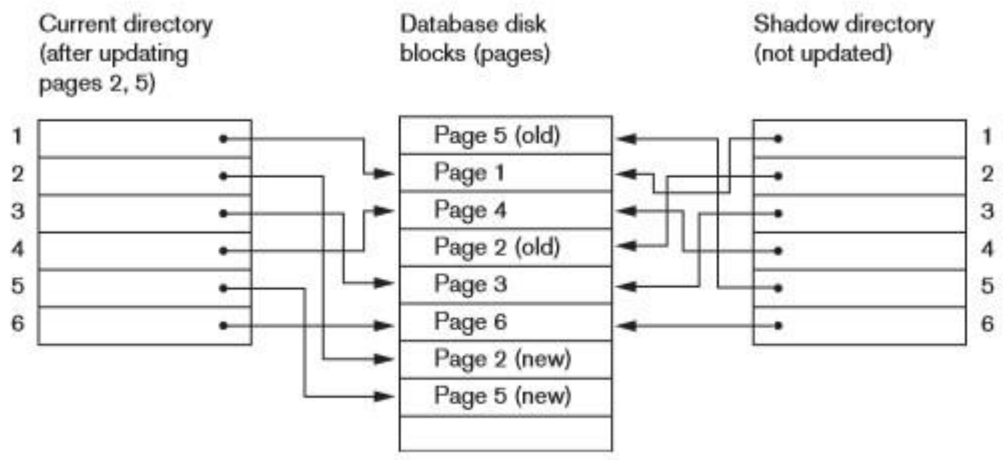
Step 3 is more efficiently done by starting from the end of the log and redoing only the last update of each item X. Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2 so that an item can be undone at most once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the

Shadow Paging(VTU QP)

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 fixedsize disk pages (or disk blocks)—say, n—for recovery purposes.A **directory** with n entries5 is constructed, where the ith entry points to the ith 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.vWhen 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 23.4 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 vers**ion by the **current directory.**

Figure 23.4
An example of shadow paging.


Current directory (after updating pages 2, 5)
Database disk blocks (pages)
Shadow directory (not updated)
1
2
3
4
5
6
Page 5 (old)
Page 1
Page 4
Page 2 (old)
Page 3
Page 6
Page 2 (new)
Page 5 (new)

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

5.15 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 business-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.