# UNIT 4
# Transaction Processing Concepts

## 4.1 Introduction to Transaction Processing

*Single-User Versus Multiuser Systems*

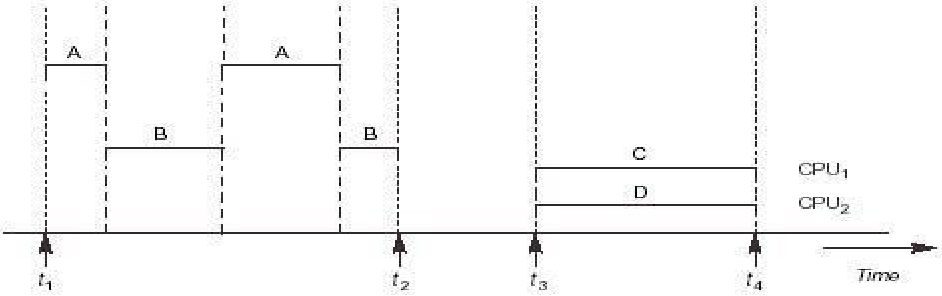
A DBMS is **single-user** id at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently.

Most DBMS are multiuser (e.g., airline reservation system).

*Multiprogramming operating systems* allow the computer to execute multiple programs (or processes) at the same time (having one CPU, concurrent execution of processes is actually interleaved).

If the computer has multiple hardware processors (CPUs), *parallel processing* of multiple processes is possible.

Figure 19.1 Interleaved processing versus parallel processing of concurrent transactions.



## 4.2 Transactions, Read and Write Operations

A *transaction* is a logical unit of database processing that includes one or more database access operations (e.g., 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.**

*Read-only transaction* - do not changes the state of a database, only retrieves data.

The basic database access operations that a transaction can include are as follows:

*read_item(X):* reads a database item *X* into a program variable *X*.
*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).
3. Copy item *X* from the buffer to the program variable named *X*.

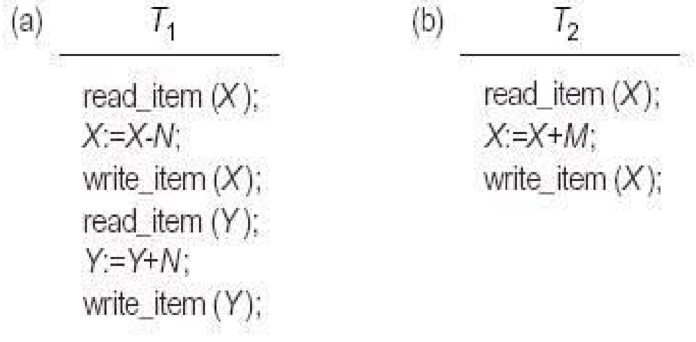
Executing a write_item(*X*) command includes the following steps:

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 buffer to the program variable named *X*.

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

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

**Figure 19.2** Two sample transactions. (a) Transaction $T_1$. (b) Transaction $T_2$.



(a) $T_1$

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

(b) $T_2$

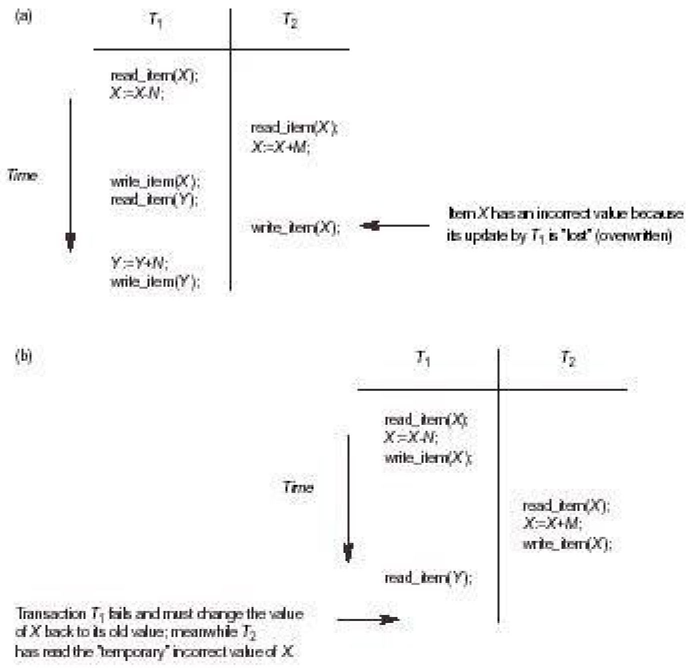
```
read_item (X);
X:=X+M;
write_item (X);
```

## 4.3 Why Concurrency Control Is Needed

**The Lost Update Problem**.

This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
Suppose that transactions T1 and T2 are submitted at approximately the same time, and suppose that their operations are interleaved then the final value of item *X* is incorrect, because T2 reads the value of *X before* T1 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;
but in the interleaving of operations, it is *X* = 84 because the update in T1 that removed the five seats from *X* was *lost.*

Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem.



(a)

| $T_1$ | $T_2$ |
|---|---|
| read_item(*X*);<br>*X*:=*X*-*N*; | |
| | read_item(*X*);<br>*X*:=*X*+*M*; |
| write_item(*X*);<br>read_item(*Y*); | |
| | write_item(*X*); |
| *Y*:=*Y*+*N*;<br>write_item(*Y*); | |

Time
Item *X* has an incorrect value because its update by $T_1$ is "lost" (overwritten)

(b)

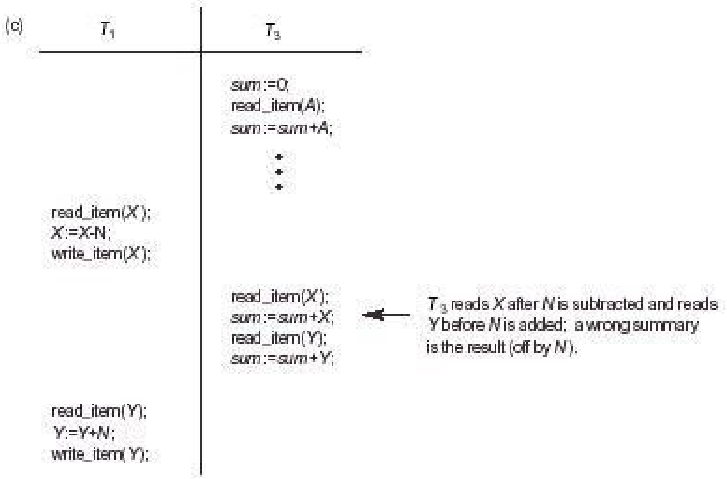
| $T_1$ | $T_2$ |
|---|---|
| read_item(*X*);<br>*X*:=*X*-*N*;<br>write_item(*X*); | |
| | read_item(*X*);<br>*X*:=*X*+*M*;<br>write_item(*X*); |
| read_item(*Y*); | |

Time
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*

**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. The updated item is accessed by another transaction before it is changed back to its original value. Figure 19.3(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.*

Figure 19.3 Some problems that occur when concurrent execution is uncontrolled. (c) The incorrect summary problem.



(c)

| $T_1$ | $T_3$ |
|---|---|
| | *sum* :=0;<br>read_item(*A*);<br>*sum* :=*sum*+*A*;<br>⋮ |
| read_item(*X* );<br>*X*:=X-N;<br>write_item(*X* ); | |
| | read_item(*X*);<br>*sum* :=*sum*+*X*;<br>read_item(*Y*);<br>*sum* :=*sum*+*Y*; |
| read_item(*Y*);<br>Y:=Y+*N*;<br>write_item( *Y*); | |

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

**The Incorrect Summary Problem.**

If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction 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 19.03(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.

Another problem that may occur is called **unrepeatable read,** where a transaction *T* reads an item twice and the item is changed by another transaction *T'* between the two reads. Hence, *T* receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer is inquiring about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation.

### 4.4 Why Recovery Is Needed

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

Types of Failures

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

1. *A computer failure (system crash):* A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.
2. *A transaction or system error:* Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error . In addition, the user may interrupt the transaction during its execution.
3. *Local errors or exception conditions detected by the transaction:* During transaction execution, certain conditions may occur that necessitate cancellation of the transaction. For example, data for the transaction may not be found. Notice that an exception condition , such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception should be programmed in the transaction itself, and hence would not be considered a failure.
4. *Concurrency control enforcement:* The concurrency control method (see Chapter 20) may decide to abort the transaction, to be restarted later, because it violates serializability (see Section 19.5) or because several transactions are in a state of deadlock.
5. *Disk failure:* Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.
6. *Physical problems and catastrophes:* This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

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

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

## 4.5 Transaction and System Concepts

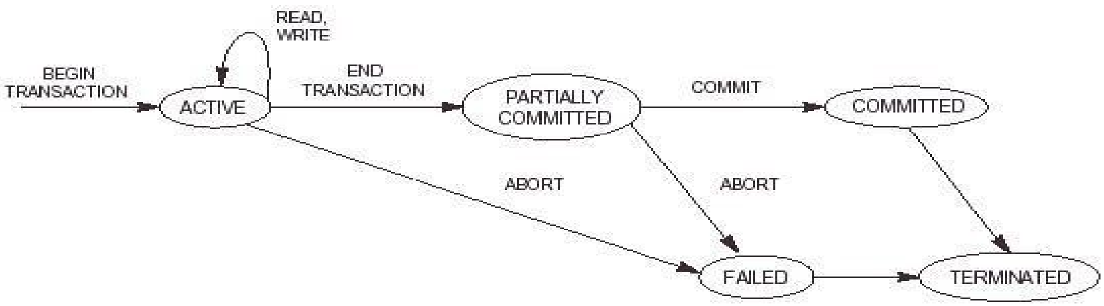
*Transaction States and Additional Operations*

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

- BEGIN_TRANSACTION: This marks the beginning of transaction execution.
- READ or WRITE: These specify read or write operations on the database items that are executed as part of a transaction.
- END_TRANSACTION: This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability (see Section 19.5) or for some other reason.
- COMMIT_TRANSACTION: This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.
- ROLLBACK (or ABORT): This signals that the transaction has *ended unsuccessfully,* so that any changes or effects that the transaction may have applied to the database must be *undone.*

Figure 19.4 shows a state transition diagram that describes how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can issue READ and WRITE operations. When the transaction ends, it moves to the **partially committed state.** At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the 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.** Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.

**Figure 19.4** State transition diagram illustrating the states for transaction execution.



## 4.6 The System Log

To be able to recover from failures that affect transactions, the system maintains a *log* to keep track of all transactions that affect the values of database items.

Log records consists of the following information (*T* refers to a unique *transaction_id*):

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

## 4.7 Desirable Properties of Transactions

Transactions should possess the following (ACID) properties:

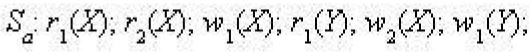
Transactions should possess several properties. These are often called the **ACID properties,** and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

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

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

## 4.8 Schedules and Recoverability

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*.



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

Similarly, the schedule for Figure 19.3(b), which we call Sb, can be written as follows, if we assume that transaction T1 aborted after its read_item(*Y*) operation:

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

Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:

1. they belong to different transactions;
2. they access the same item *X;* and
3. at least one of the operations is a write_item(*X*).

For example, in schedule $S_a$, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, ), and the operations w1(*X*) and w2(*X*). However, the operations r1(*X*) and r2(*X*) do not conflict, since they are both read operations; the operations w2(*X*) and w1(*Y*) do not conflict, because they operate on distinct data items *X* and *Y;* and the operations r1(*X*) and w1(*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:

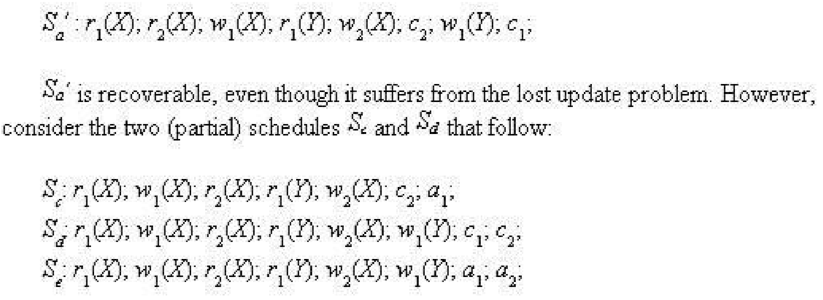
1. The operations in *S* are exactly those operations in T1, T2, ..., Tn, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction Ti, their order of appearance in *S* is the same as their order of appearance in Ti.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

## 4.9 Characterizing Schedules Based on Recoverability

Once a transaction *T* is committed, it should *never* be necessary to roll back *T.* The schedules that theoretically meet this criterion are called *recoverable schedules* and those that do not are called **non-recoverable,** and hence should not be permitted.

A schedule *S* is recoverable if no transaction *T* in *S* commits until all transactions *T'* that have written an item that *T* reads have committed. A transaction *T* **reads** from transaction *T* in a schedule *S* if some item *X* is first written by $T'$ and later read by *T.* In addition, $T'$ should not have been aborted before *T* reads item *X,* and there should be no transactions that write *X* after $T'$ writes it and before *T* reads it (unless those transactions, if any, have aborted before *T* reads *X).*

Consider the schedule $S'_a$ given below, which is the same as schedule $S_a$ except that two commit operations have been added to $S_a$ :



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

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

$S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $a_1$;
$S_d$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $c_1$; $c_2$;
$S_e$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $a_1$; $a_2$;

$S_c$ is not recoverable, because T2 reads item *X* from T1, and then T2 commits before T1 commits. If T1 aborts after the c2 operation in $S_c$, then the value of *X* that T2 read is no longer valid and T2 must be aborted *after* it had been committed, leading to a schedule that is not recoverable. For the schedule to be recoverable, the c2 operation in $S_c$ must be postponed until
after T1 commits. If T1 aborts instead of committing, then T2 should also abort as shown in Se, because the value of *X* it read is no longer valid.

In a recoverable schedule, no committed transaction ever needs to be rolled back. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed.

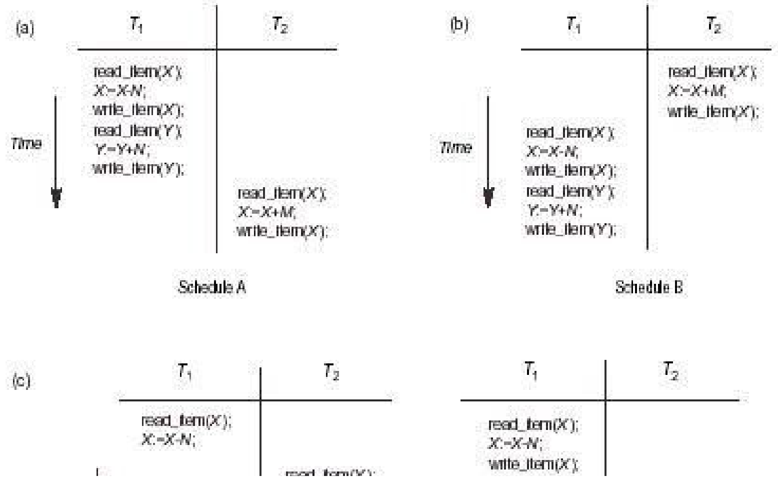
## 4.10 Serializability of Schedules – Serial and Non Serial

If no interleaving of operations is permitted, there are only two possible arrangement for transactions T1 and T2.

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

A schedule *S* is *serial* if, for every transaction *T* all the operations of *T* are executed consecutively in the schedule.
A schedule *S* of *n* transactions is *serializable* if it is equivalent to some serial schedule of the same *n* transactions.

**Figure 19.5** Examples of serial and nonserial schedules involving transactions $T_1$ and $T_2$. (a) Serial schedule A: $T_1$ followed by $T_2$. (b) Serial schedule B: $T_2$ followed by $T_1$. (c) Two nonserial schedules C and D with interleaving of operations.



## 4.11 Serializability of Schedules – Conflict Serializable

Conflict Equivalence of Two Schedules. Two schedules are said to be conflict equivalent if the relative order of any two conflicting operations is the same in both schedules.
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. If two conflicting operations are applied in different orders in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not conflict equivalent. For example, if a read and write operation occur in the order r1(X), w2(X) in schedule S1, and in the reverse order w2(X), r1(X) in schedule S2, the value read by r1(X) can be different in the two schedules. Similarly, if two write operations occur in the order w1(X), w2(X) in S1, and in the reverse order w2(X), w1(X) in S2, the next r(X) operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different

## 4.12 Testing for Serializability of a Schedule

Below given algorithm 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 ≤ k ≤ n, 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 a pair of conflicting operations exist in Tj and Tk and the conflicting operation in Tj appears in the schedule before the conflicting operation in Tk.

Algorithm. Testing Conflict Serializability of a Schedule S
1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
2. For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create an edge (Ti → Tj) in the precedence graph.

3. For each case in S where Tj executes a write_item(X) after Ti executes a read_item(X), create an edge (Ti → Tj) in the precedence graph.
4. For each case in S where Tj executes a write_item(X) after Ti executes a write_item(X), create an edge (Ti → Tj) 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.
If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable.
A cycle in a directed graph is a sequence of edges
C = ((Tj → Tk), (Tk → Tp), … , (Ti → Tj ))
with the property that the starting node of each edge— except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node).

In the precedence graph, an edge from Ti to Tj means that transaction Ti must come before transaction Tj in any serial schedule that is equivalent to S, because two conflicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an equivalent serial schedule S′ that is equivalent to S, by ordering the transactions that participate in S as follows:
Whenever an edge exists in the precedence graph from Ti to Tj , Ti must appear before Tj in the equivalent serial schedule S′. 13 Notice that the edges

(Ti → Tj ) in a precedence graph can optionally be labeled by the name(s) of the data item(s) that led to creating the edge.

Figure below shows such labels on the edges. When checking for a cycle, the labels are not relevant. In general, several serial schedules can be equivalent to S if the precedence graph for S has no cycle.
However, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable. The precedence graphs created for schedules A to D, respectively, appear in the below Figures 20.7(a) to (d).

The graph for schedule C has a cycle, so it is not serializable. The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is T1 followed by T2. The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.

# Concurrency Control Techniques

Two phase locking techniques for concurrency control, types of locks and system lock tables, Guaranteeing serializability by two-phase locking, Dealing with Deadlock and starvation, Concurrency control based on timestamp ordering.

Concurrency control protocols are a set of rules that guarantee serializability. One important set of protocols—

known as *two-phase locking protocols*— employs the technique of locking data items to prevent multiple transactions from accessing the items concurrently; Locking protocols are used in some commercial DBMSs, but they are considered to have high overhead. Another set of concurrency control protocols uses timestamps.

Another factor that affects concurrency control is the **granularity** of the data items—that is, what portion of the database a data item represents. An item can be as small as a single attribute (field) value or as large as a disk block, or even a whole file or the entire database.

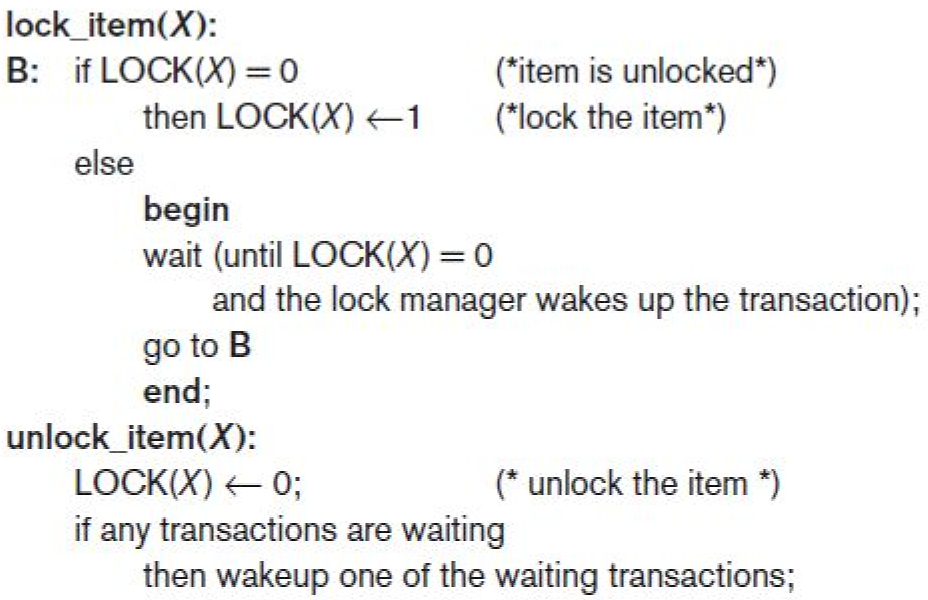
## 4.13 Two-Phase Locking Techniques for Concurrency Control

### Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking concepts gradually, first we discuss binary locks, which are simple but are also *too restrictive for database concurrency control purposes* and so are not used much.
Then we discuss *shared/exclusive* locks—also known as *read/write* locks—which provide more general locking capabilities and are used in database locking schemes.

### Binary Locks

A **binary lock** can have two **states** or **values**: locked and unlocked. Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item *X* by first issuing a **lock_item(*X*)** operation.
LOCK(*X*) = 1, the transaction is forced to wait. If LOCK(*X*) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item *X*. When the transaction is through using the item, it issues an **unlock_item(*X*)** operation, which sets LOCK(*X*) back to 0 (**unlocks** the item) so that *X* may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock_item(*X*) and unlock_item(*X*) operations is shown below.



```
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;
```

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**

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*.3

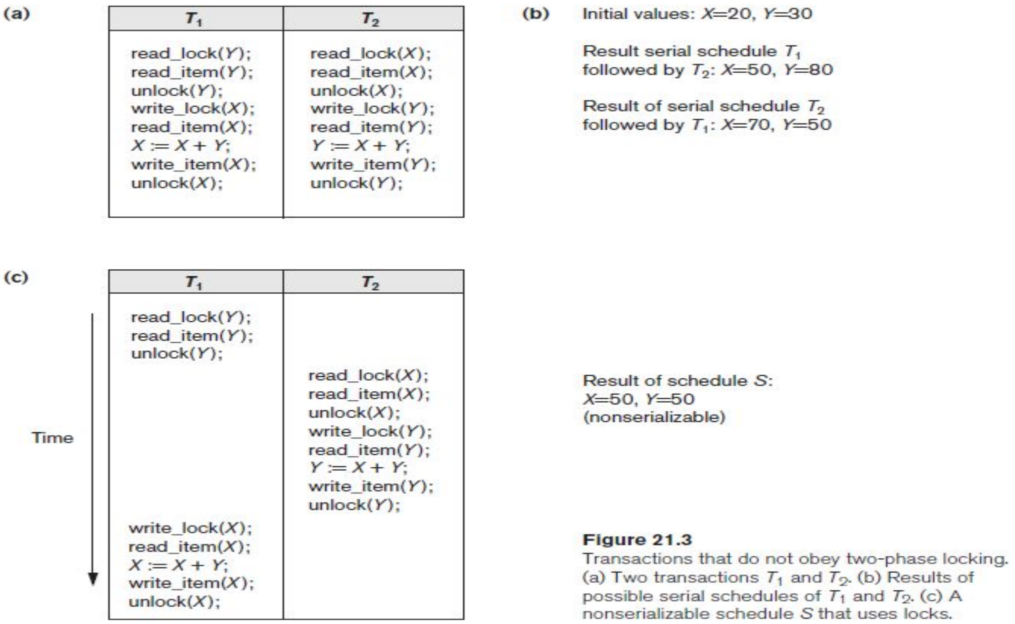
**4.** A transaction *T* will not issue a read_lock(*X*) operation if it already holds a read (shared) lock or a write (exclusive) lock on item *X*. This rule may be relaxed for downgrading of locks, as we discuss shortly.

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

**6.** A transaction *T* will not issue an unlock(*X*) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

## 4.14 Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction. Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired.



(a)

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

(b) Initial values: $X$=20, $Y$=30

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

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

(c)

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

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

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

**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.

**Deadlock Prevention Protocols** One way to prevent deadlock is to use a **deadlock prevention protocol**. One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously, this solution further limits concurrency.

A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation:
Some of these techniques use the concept of **transaction timestamp** TS(*T*'), which is a unique identifier assigned to each transaction.

The timestamps are typically based on the order in which transactions are started; hence, if transaction *T*1 starts before transaction *T*2, then TS(*T*1) < TS(*T*2).

Notice that the *older* transaction (which starts first) has the *smaller* timestamp value. Two schemes that prevent deadlock are called *wait-die* and *wound-wait*. Suppose that transaction *Ti* 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(*Ti*) < TS(*Tj*), then (*Ti* older than *Tj*) *Ti* is allowed to wait; otherwise (*Ti* younger than *Tj*) abort *Ti* (*Ti dies*) and restart it later *with the same timestamp.*

■ **Wound-wait.** If TS(*Ti*) < TS(*Tj*), then (*Ti* older than *Tj*) abort *Tj* (*Ti wounds Tj*) and restart it later *with the same timestamp;* otherwise (*Ti* younger than *Tj*) *Ti* is allowed to wait.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms.

In the **no waiting algorithm**, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly.

The **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction *Ti* tries to lock an item *X* but is not able to do so because *X* is locked by some other transaction *Tj* with a conflicting lock. The cautious waiting rule is as follows:

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

**Starvation.** Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair in that it gives priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock.

## 4.15 Concurrency Control Based on Timestamp Ordering

The use of locking, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire.

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

**The Timestamp Ordering Algorithm for Concurrency Control**
The idea for this scheme is to enforce the equivalent serial order on the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**.

In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction timestamps. The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of *conflicting operations* in the schedule, the order in which the item is accessed must follow the timestamp order. To do this, the algorithm associates with each database item *X* two timestamp (TS) values:

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

**Basic Timestamp Ordering (TO)**

Whenever some transaction *T* tries to issue a read_item(*X*) or a write_item(*X*) operation, the **basic TO** algorithm compares the timestamp of *T* with read_TS(*X*) and write_TS(*X*) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transaction *T* is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If *T* is aborted and rolled back, any transaction *T*1 that may have used a value written by *T* must also be rolled back. Similarly, any transaction *T*2 that may have used a value written by *T*1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.

The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

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

## Strict Timestamp Ordering (TO)

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

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