# Module 5

# Chapter 1: Transaction Processing

## 5.0 Introduction

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:

- airline reservations
- banking
- credit card processing,
- online retail purchasing,
- Stock markets, supermarket checkouts, and many other applications

These systems require high availability and fast response time for hundreds of concurrent users. A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates.

## 5.1 Objectives

- ❖ To study transaction properties
- ❖ To study creation of schedule and maintaining schedule equivalence.
- ❖ To check whether the given schedule is serailizable or not.
- ❖ To study protocols used for locking objects
- ❖ Differentiating between 2PL and Strict 2PL

## 5.2 Introduction to Transaction Processing

### 5.2.1 Single-User versus Multiuser Systems

- ▪ One criterion for classifying a database system is according to the number of users who can use the system **concurrently**

**Single-User versus Multiuser Systems**

- ▪ A DBMS is
- **single-user**
  - at most one user at a time can use the system
  - Eg: Personal Computer System
- **multiuser**
  - many users can use the system and hence access the database concurrently
  - Eg: Airline reservation database

- Concurrent access is possible because of **Multiprogramming.** Multiprogramming can be achieved by:
  - interleaved execution
  - Parallel Processing
- **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

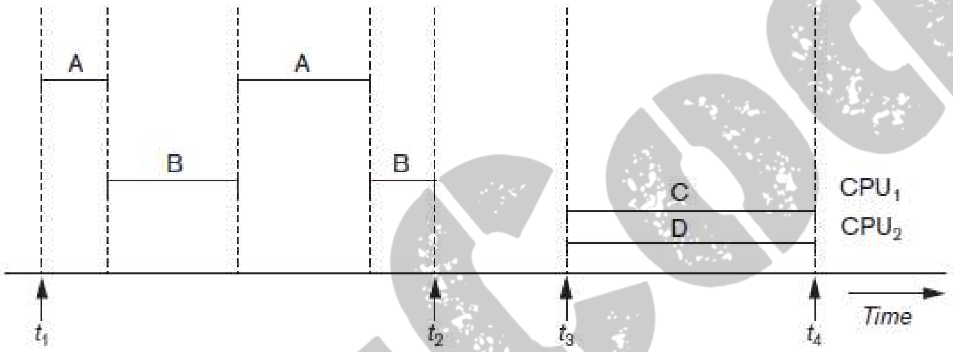


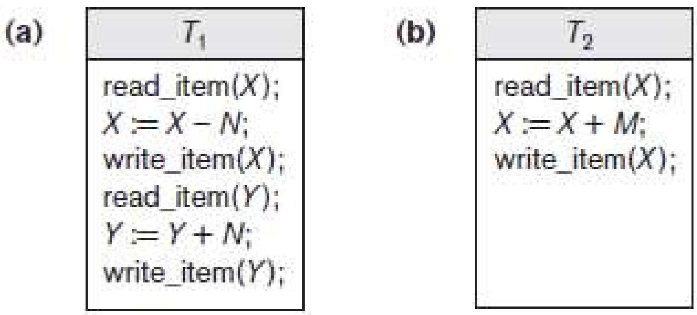
**Figure 21.1** Interleaved processing versus parallel processing of concurrent transactions.

- Figure 21.1, shows two processes, A and B, executing concurrently in an interleaved fashion
- Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk
- The CPU is switched to execute another process rather than remaining idle during I/O time
- Interleaving also prevents a long process from delaying other processes.
- If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in Figure 21.1
- Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency**
- In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

### 5.2.2 Transactions, Database Items, Read and Write Operations, and DBMS Buffers

- A Transaction an executing program that forms a logical unit of database processing
- It includes one or more DB access operations such as insertion, deletion, modification or retrieval operation.
- It can be either embedded within an application program using **begin transaction** and **end transaction** statements Or specified interactively via a high level query language such as SQL
- Transaction which do not update database are known as **read only transactions.**
- Transaction which do update database are known as **read write transactions.**
- 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
- Each data item has a unique name
- **Basic DB access operations that a transaction can include are:**
  - **read_item(X)**: Reads a DB item named X into a program variable.
  - **write_item(X)**: Writes the value of a program variable into the DB item named X
- **Executing read_item(X) include the following steps:**
  1. Find the address of the disk block that contains item X
  2. Copy the block into a buffer in main memory
  3. Copy the item X from the buffer to program variable named X.
- **Executing write_item(X) include the following steps:**
  1. Find the address of the disk block that contains item X
  2. Copy the disk block into a buffer in main memory
  3. Copy item X from program variable named X into its correct location in buffer.
  4. Store the updated disk block from buffer back to disk (either immediately or later).
- Decision of when to store a modified disk block is handled by **recovery manager** of the DBMS in cooperation with operating system.
- A DB cache includes a number of data buffers.
- When the buffers are all occupied a buffer replacement policy is used to choose one of the buffers to be replaced. EG: LRU

- A transaction includes read_item and write_item operations to access and update DB.



(a)

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

(b)

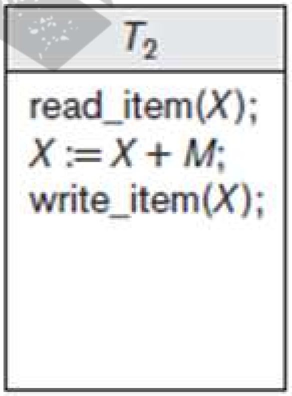
| $T_2$ |
|---|
| read_item($X$);<br>$X := X + M$;<br>write_item($X$); |

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

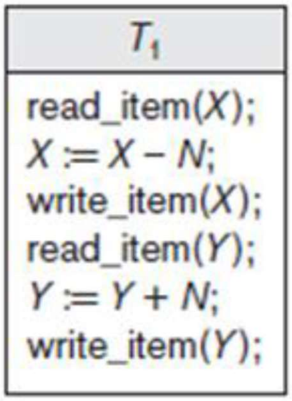
- The **read-set** of a transaction is the set of all items that the transaction reads
- The **write-set** is the set of all items that the transaction writes
- For example, the read-set of *T*1 in Figure 21.2 is {*X*, *Y*} and its write-set is also {*X*, *Y*}.

### 5.2.3 Why Concurrency Control Is Needed

- Several problems can occur when concurrent transactions execute in an uncontrolled manner
- Example:
  - We consider an Airline reservation DB
  - Each records is stored for an airline flight which includes Number of reserved seats among other information.
  - Types of problems we may encounter:
    1. The Lost Update Problem
    2. The Temporary Update (or Dirty Read) Problem
    3. The Incorrect Summary Problem
    4. The Unrepeatable Read Problem



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

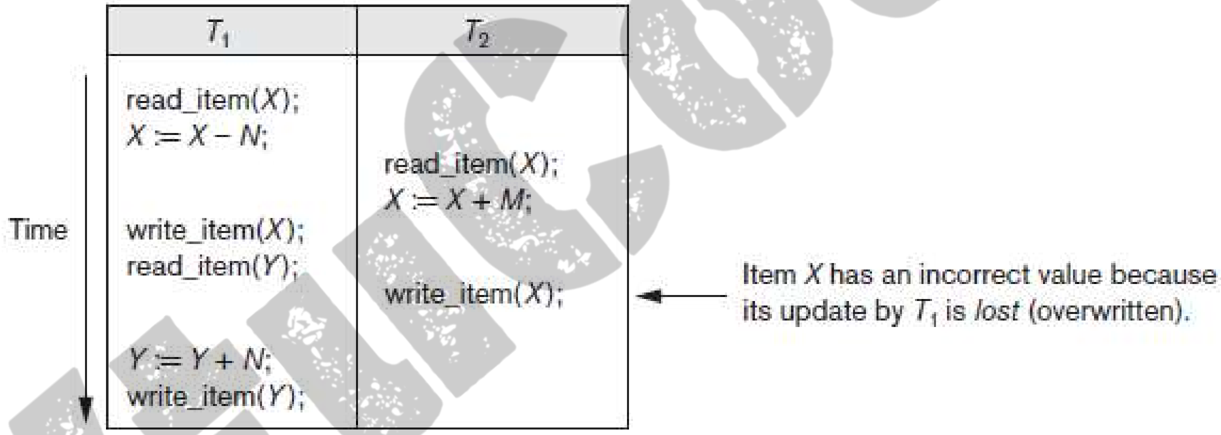


| $T_1$ |
|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$);<br>read_item($Y$);<br>$Y := Y + N$;<br>write_item($Y$); |

- Transaction T1
  - 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.
- Transaction T2
  - reserves M seats on the first flight (X)

## 1. The Lost Update Problem

- occurs when two transactions that access the same DB items have their operations interleaved in a way that makes the value of some DB item 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 below



| $T_1$ | $T_2$ |
|---|---|
| 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($Y$); | |

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

- Final value of item *X* is incorrect because *T*2 reads the value of *X before T*1 changes it in the database, and hence the updated value resulting from *T*1 is lost.
- For example:

  X = 80 at the start (there were 80 reservations on the flight)

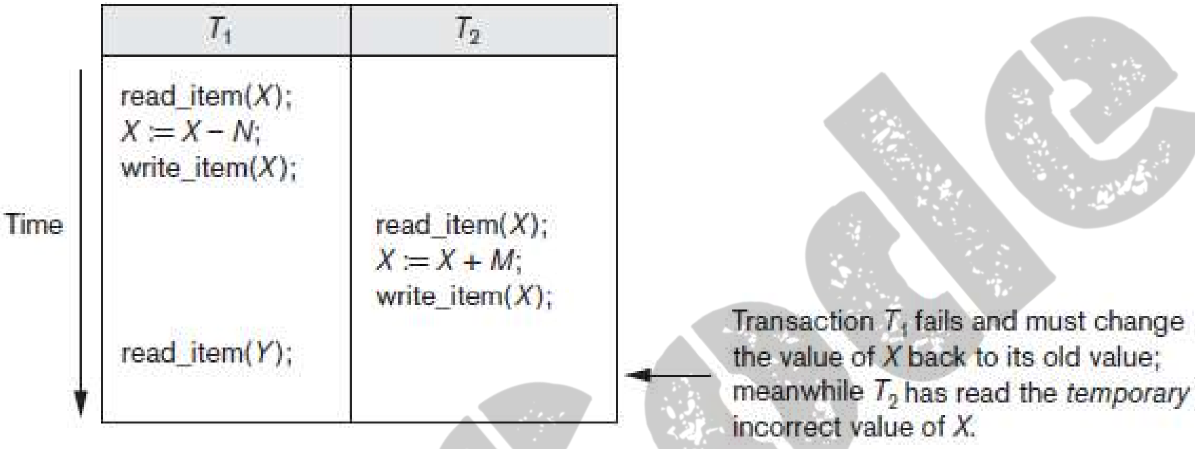
  N = 5 (T1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y)

  M = 4 (T2 reserves 4 seats on X)

  The final result should be X = 79.
- The interleaving of operations shown in Figure is X = 84 because the update in T1 that removed the five seats from X was lost.

## 2. The Temporary Update (or Dirty Read) Problem

- occurs when one transaction updates a database item and then the transaction fails for some reason
- Meanwhile the updated item is accessed by another transaction before it is changed back to its original value

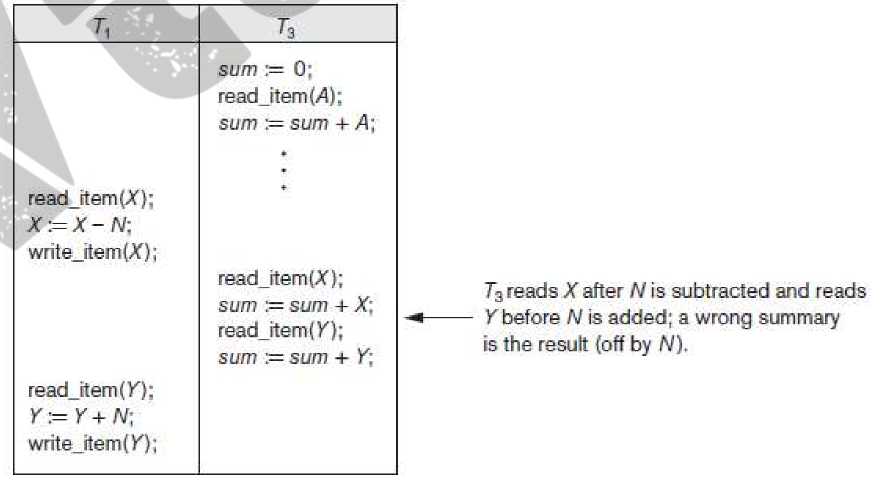


| $T_1$ | $T_2$ |
|---|---|
| read_item($X$);<br>$X := X - N$;<br>write_item($X$); | |
| | read_item($X$);<br>$X := X + M$;<br>write_item($X$); |
| read_item($Y$); | |

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 db 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.



| $T_1$ | $T_3$ |
|---|---|
| | $sum := 0$;<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 $N$).

## 4. The Unrepeatable Read Problem

- Transaction T reads the same item twice and gets different values on each read, since the item was modified by another transaction T` between the two reads.
- 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.

## 5.2.4 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 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
- 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

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
   - Also occur because of erroneous parameter values
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

**4. Concurrency control enforcement:**

- The concurrency control may decide to abort a transaction because itviolates serializability or 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.

**6. Physical problems and catastrophes:**

- refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, overwriting disks or tapes by mistake

- 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.

## 5.3 Transaction and System Concepts

### 5.3.1 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 keeps track of start of a transaction, termination, commit or aborts.
  - **BEGIN_TRANSACTION**: marks the beginning of transaction execution
  - **READ or WRITE**: specify read or write operations on the database items that are executed as part of a transaction
  - **END_TRANSACTION:** specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution
  - **COMMIT_TRANSACTION**: 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**: 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**

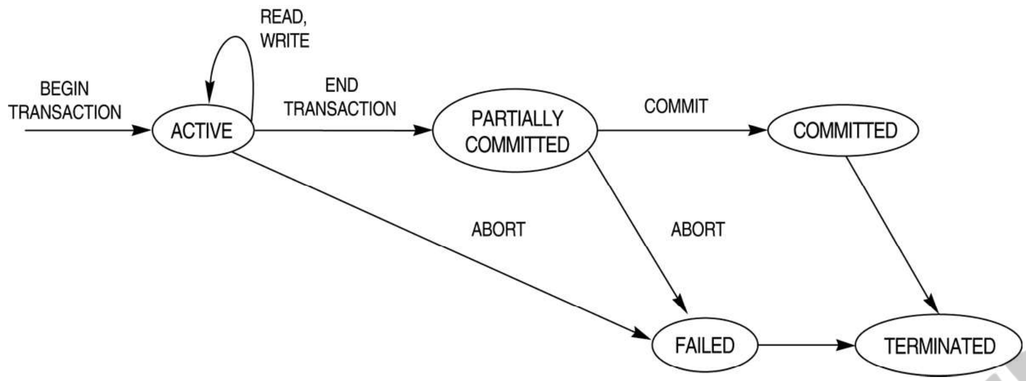


Figure: State transition diagram illustrating the states for transaction execution

- A transaction goes into **active state** immediately after it starts execution and can execute read and write operations.

- When the transaction ends it moves to **partially committed** state.
- At this end additional checks are done to see if the transaction can be committed or not. If these checks are successful the transaction is said to have reached commit point and enters **committed state.** All the changes are recorded permanently in the db.
- A transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its write operation.
- Terminated state corresponds to the transaction leaving the system. All the information about the transaction is removed from system tables.

### 5.3.2 The System Log

- **Log or Journal** keeps track of all transaction operations that affect the values of database items
- This information may be needed to permit recovery from transaction failures.
- The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure
- one (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory buffer
- When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*.

- In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures
- The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record.
- In these entries, T refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:

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

  **2. [write_item, T, X, old_value, new_value].** Indicates that transaction T has changed the value of database item X from old_value to new_value.

  **3. [read_item, T, X].** Indicates that transaction T has read the value of database item X.

  **4. [commit, T].** Indicates that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.

  **5. [abort, T].** Indicates that transaction T has been aborted.

### 5.3.3 Commit Point of a Transaction:

- **Definition a Commit Point:**
  - 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 has been recorded in the log.
  - Beyond the commit point, the transaction is said to be committed, and its effect is assumed to be permanently recorded in the database.
  - The transaction then writes an entry [commit,T] into the log.
- **Roll Back of transactions:**
  - Needed for transactions that have a [start_transaction,T] entry into the log but no commit entry [commit,T] into the log.

### 5.3.4 DBMS specific buffer Replacement policies

**Domain Separation(DS) method**

• DBMS cache is divided into separate domains, each handles one type of disk pages and replacements within each domain are handled via basic LRU page replacement.

• LRU is a **static** algorithm and does not adopts to dynamically changing loads because the number of available buffers for each domain is predetermined.

• **Group LRU** adds dynamically load balancing feature since it gives each domain a priority and selects pages from lower priority level domain first for replacement.

**Hot Set Method:**

- This is useful in queries that have to scan a set of pages repeatedly.
- The hot set method determines for each db processing algorithm the set of disk pages that will be accessed repeatedly and it does not replace them until their processing is completed.

**The DBMIN method:**

- uses a model known as QLSM (Query Locality set model), which predetermines the pattern of page references for each algorithm for a particular db operation
- Depending on the type of access method, the file characteristics, and the algorithm used the QLSM will estimate the number of main memory buffers needed for each file involved in the operation.

## 5.4 Desirable Properties of Transactions

- Transactions should possess several properties, often called the **ACID** properties

  A **Atomicity:** a transaction is an atomic unit of processing and it is either performed entirely or not at all.

  C **Consistency Preservation:** a transaction should be consistency preserving that is it must take the database from one consistent state to another.

  I **Isolation/Independence:** A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executed concurrently.

  D **Durability (or Permanency):** if a transaction changes the database and is committed, the changes must never 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.
- The preservation of ***consistency*** is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints.
- The **isolation** property is enforced by the concurrency control subsystem of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks
- **Durability** is the responsibility of recovery subsystem.

## 5.5 Characterizing Schedules Based on Recoverability

- **schedule** (or **history):** the order of execution of operations from all the various transactions
- **Schedules (Histories) of Transactions:** A schedule S of n transactions $T_1$, $T_2$,…….$T_n$ is a sequential ordering of the operations of the n transactions.
  - – The transactions are interleaved
- 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*)
- **Conflicting operations:**
  - $r_1(X)$ conflicts with $w_2(X)$ } Read write conflict
  - $r_2(X)$ conflicts with $w_1(X)$
  - $w_1(X)$ conflicts with $w_2(X)$ Write conflict
  - $r_1(X)$ do not conflicts with $r_2(X)$

**Schedules classified on recoverability:**

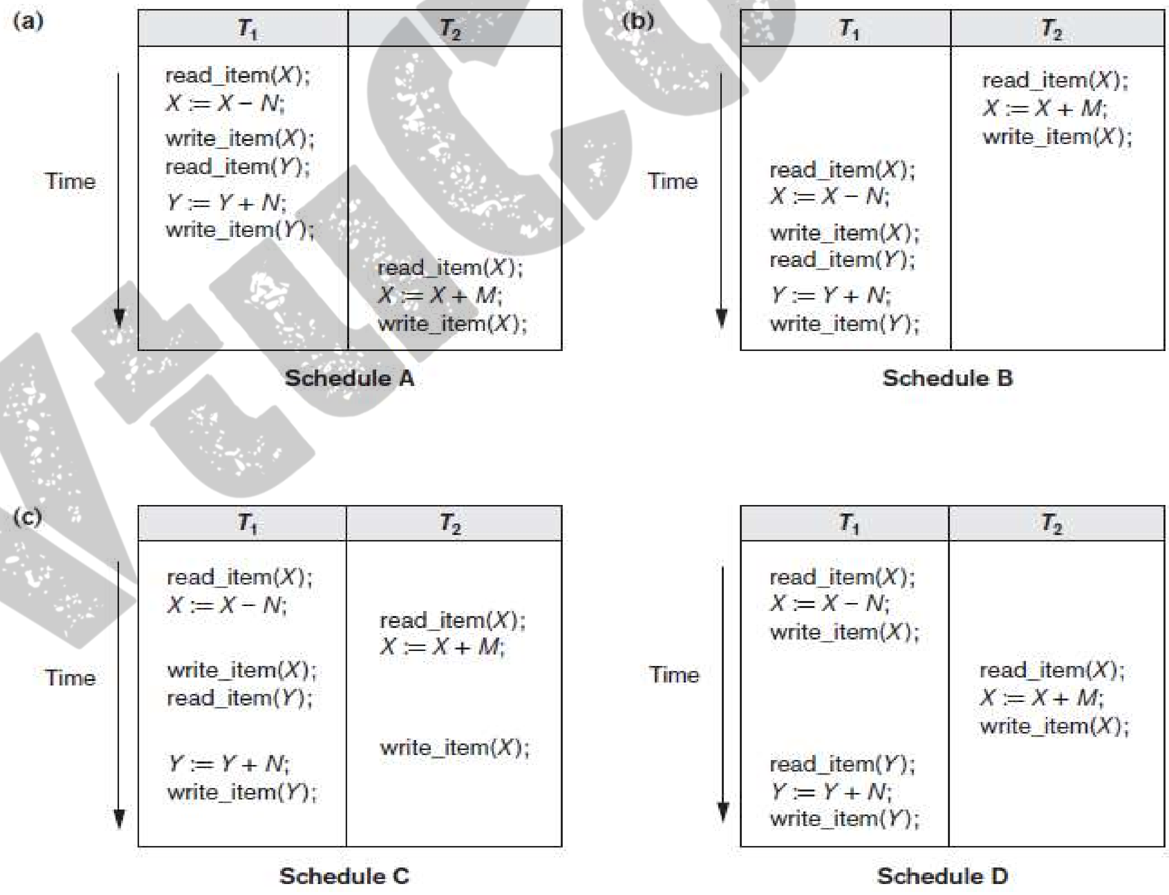
- **Recoverable schedule**:
  - – One where no transaction needs to be rolled back.
  - – A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.
  - – Example:
    - $S_c$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $c_2$; $a_1$;
    - $S_d$: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; $c_1$; $c_2$;
- **Cascadeless schedule**:
  - – One where every transaction reads only the items that are written by committed transactions.
- **Schedules requiring cascaded rollback**:
  - – A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- **Strict Schedules**:
  - – A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

## 5.6 Characterizing Schedules Based on Serializability

- schedules that are always considered to be correct when concurrent transactions are executing are known as **serializable** schedules
- Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T1 and T2 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 *T*2 (in sequence) followed by all the operations of transaction *T*1 (in sequence).

**Figure 21.5**
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.



(a)

Time ↓

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

**Schedule A**

(b)

Time ↓

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

**Schedule B**

(c)

Time ↓

| $T_1$ | $T_2$ |
|---|---|
| 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($Y$); | |

**Schedule C**

Time ↓

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

**Schedule D**

- **Serial schedule:**
  - A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
    - Otherwise, the schedule is called nonserial schedule.
- **Serializable schedule:**
  - A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
- **Result equivalent:**
  - Two schedules are called result equivalent if they produce the same final state of the database.
- **Conflict equivalent:**
  - Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- **Conflict serializable:**
  - A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.
- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

### 5.6.1 Testing conflict serializability of a Schedule S

For each transaction Ti participating in schedule S,create a node labeled Ti in the precedence graph.

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.

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.

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.

The schedule S is serializable if and only if the precedence graph has no cycles.

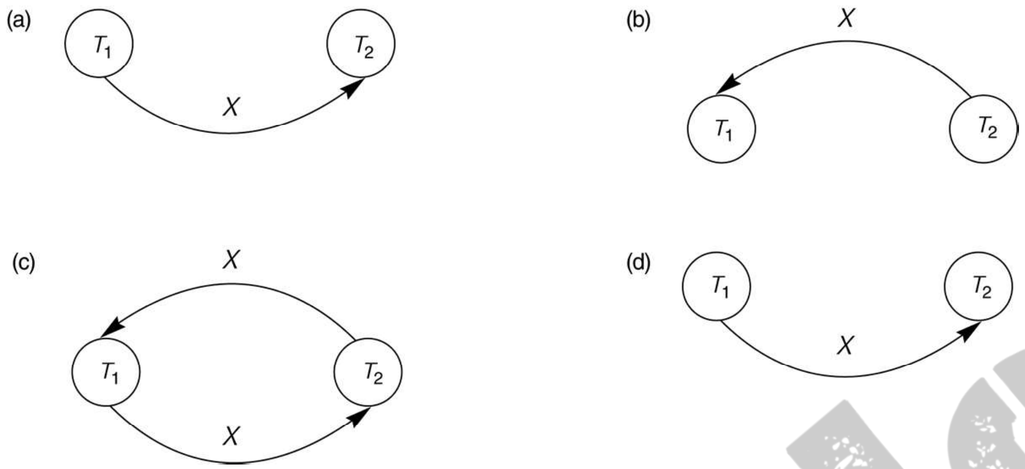


Fig: Constructing the precedence graphs for schedules *A* and *D* from fig 21.5 to test for conflict serializability.

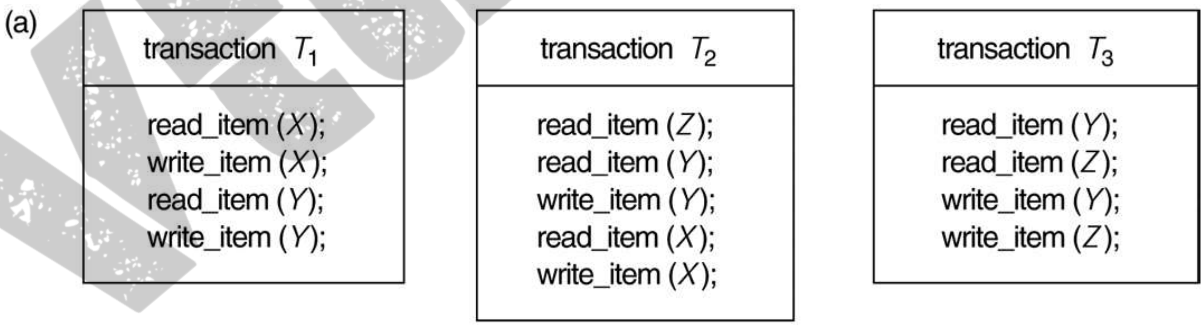
(a) Precedence graph for serial schedule *A*.

(b) Precedence graph for serial schedule *B*.

(c) Precedence graph for schedule *C* (not serializable).

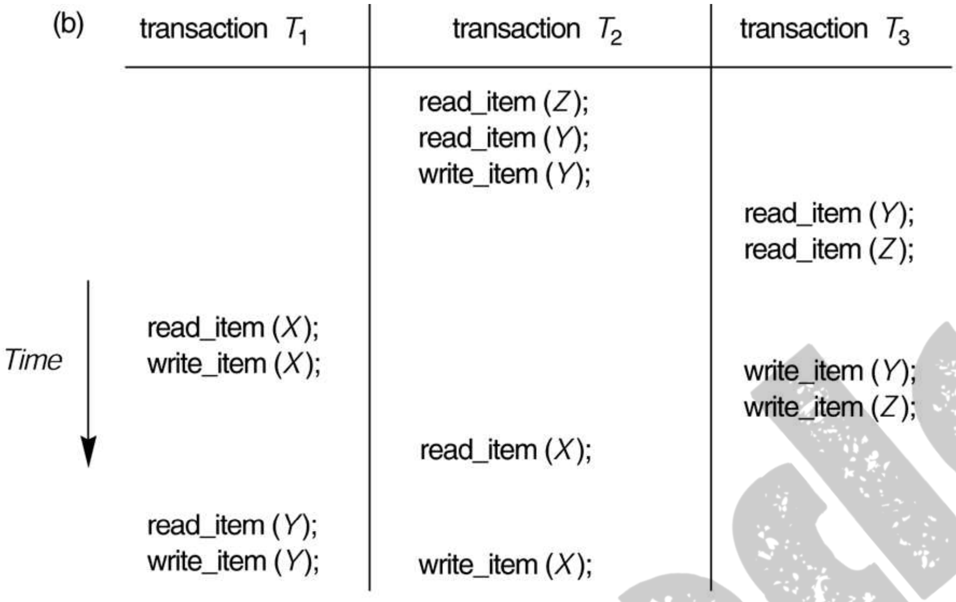
(d) Precedence graph for schedule *D* (serializable, equivalent to schedule *A*).

- Another example of serializability testing. (a) The READ and WRITE operations of three transactions $T_1$, $T_2$, and $T_3$.


(a)

| transaction $T_1$ | transaction $T_2$ | transaction $T_3$ |
|---|---|---|
| read_item (*X*);<br>write_item (*X*);<br>read_item (*Y*);<br>write_item (*Y*); | read_item (*Z*);<br>read_item (*Y*);<br>write_item (*Y*);<br>read_item (*X*);<br>write_item (*X*); | read_item (*Y*);<br>read_item (*Z*);<br>write_item (*Y*);<br>write_item (*Z*); |

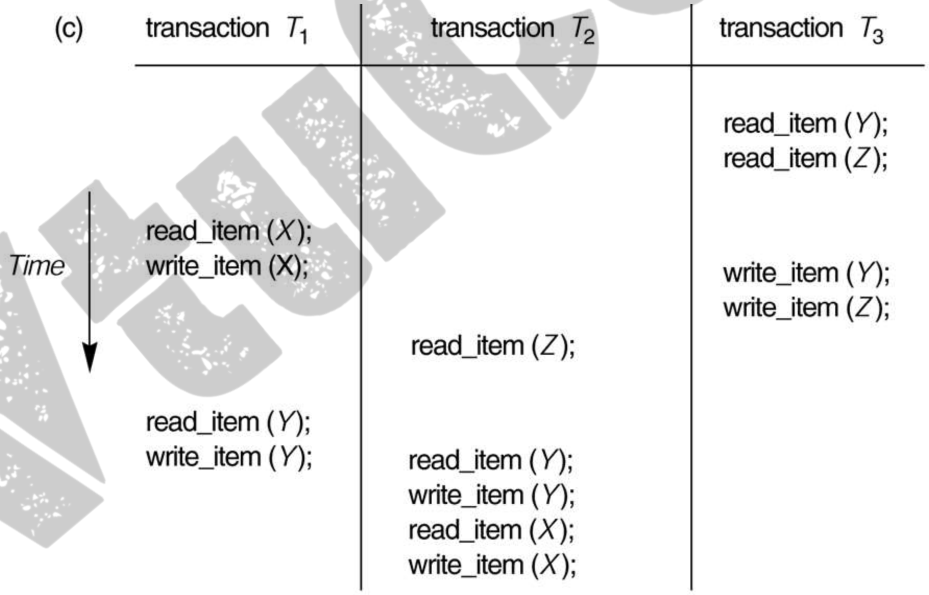
(b)



| transaction $T_1$ | transaction $T_2$ | transaction $T_3$ |
|---|---|---|
| | read_item (Z); | |
| | read_item (Y); | |
| | write_item (Y); | |
| | | read_item (Y); |
| | | read_item (Z); |
| read_item (X); | | |
| write_item (X); | | |
| | | write_item (Y); |
| | | write_item (Z); |
| | read_item (X); | |
| read_item (Y); | | |
| write_item (Y); | | |
| | write_item (X); | |

Schedule E

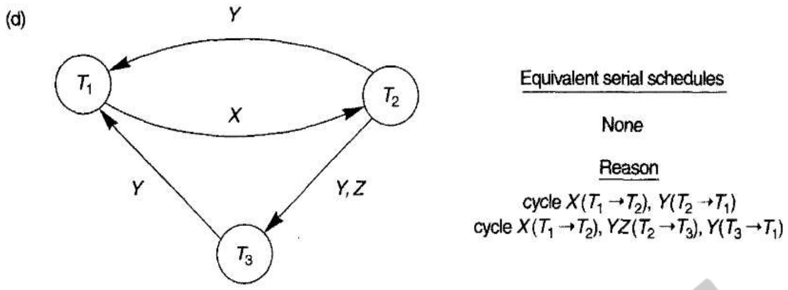
(c)



| transaction $T_1$ | transaction $T_2$ | transaction $T_3$ |
|---|---|---|
| | | read_item (Y); |
| | | read_item (Z); |
| read_item (X); | | |
| write_item (X); | | |
| | | write_item (Y); |
| | | write_item (Z); |
| | read_item (Z); | |
| read_item (Y); | | |
| write_item (Y); | | |
| | read_item (Y); | |
| | write_item (Y); | |
| | read_item (X); | |
| | write_item (X); | |

Schedule F

- Precedence graph for schedule E



Equivalent serial schedules

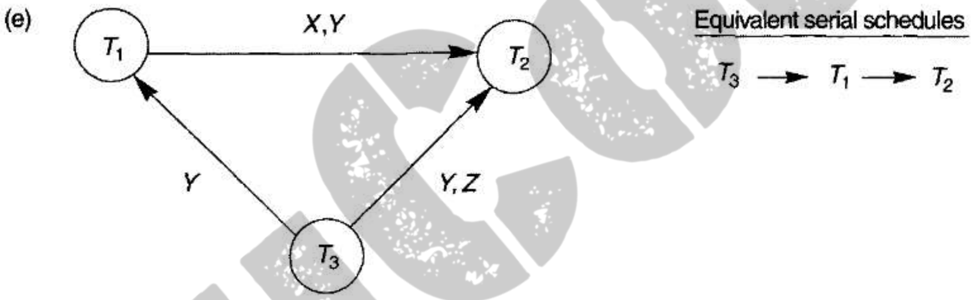
None

Reason

cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

cycle $X(T_1 \rightarrow T_2), YZ(T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

- Precedence graph for schedule F



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

## 5.7 Transaction Support in SQL

- The basic definition of an SQL transaction is, it is a logical unit of work and is guaranteed to be atomic
- A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged
- With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered
- Every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK
- Every transaction has certain characteristics attributed to it and are specified by a SET TRANSACTION statement in SQL

- The characteristics are :
  - **The access mode**
    - can be specified as READ ONLY or READ WRITE
    - The default is READ WRITE
    - A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed
    - A mode of READ ONLY, as the name implies, is simply for data retrieval.
  - **The diagnostic area size**

    - 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**

  - specified using the statement ISOLATION LEVEL <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE

  - The default isolation level is SERIALIZABLE

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

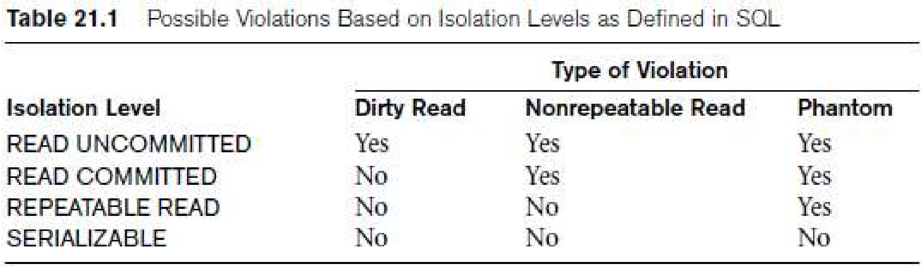
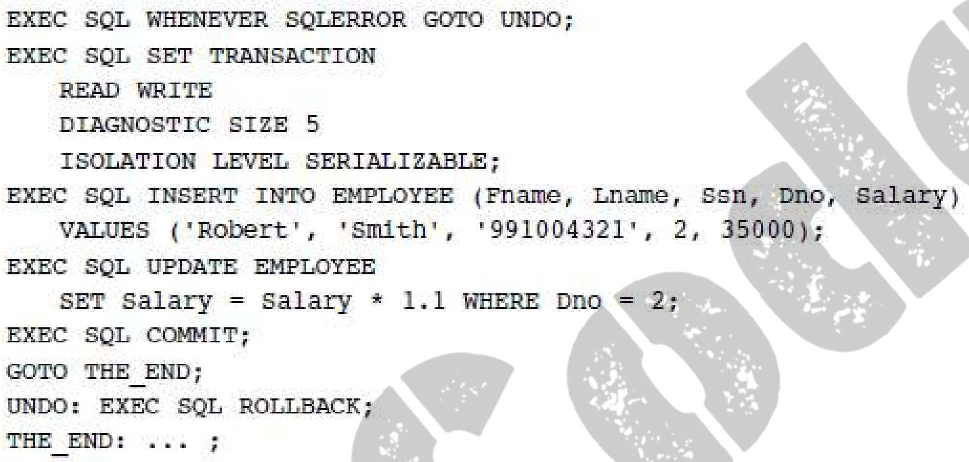


**Table 21.1** Possible Violations Based on Isolation Levels as Defined in SQL

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



```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
   READ WRITE
   DIAGNOSTIC SIZE 5
   ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
   VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
   SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

- The 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.

# Chapter 2: Concurrency Control in Databases

## 5.8 Introduction to Concurrency Control

- Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.
- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

## 5.9 Two-Phase Locking Techniques for Concurrency Control

- The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item.
- It is used for synchronizing the access by concurrent transactions to the database items.
- A transaction locks an object before using it
- When an object is locked by another transaction, the requesting transaction must wait

### 5.9.1 Types of Locks and System Lock Tables

1. **Binary Locks**

- A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0).
- If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item

- If the value of the lock on X is 0, the item can be accessed when requested, and the lock value is changed to 1
- We refer to the current value (or state) of the lock associated with item X as **lock**(**X**).
- Two operations**, lock_item** and **unlock_item**, are used with binary locking.
- A transaction requests access to an item *X* by first issuing a **lock_item(*X*)** operation
- If LOCK(*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.

```
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;
```

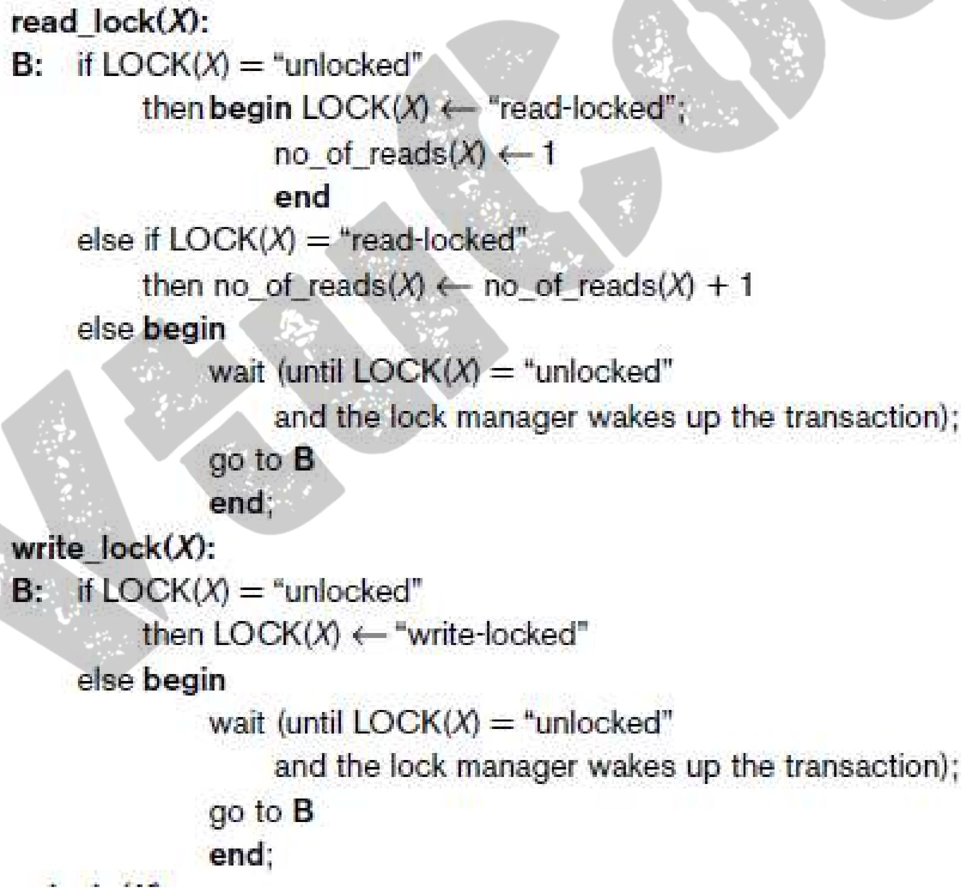
Fig: 2.1.1 Lock and unlock operations for binary licks.

- The lock_item and unlock_item operations must be implemented as indivisible units that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits
- The wait command within the lock_item(*X*) operation is usually implemented by putting the transaction in a waiting queue for item *X* until *X* is unlocked and the transaction can be granted access to it
- Other transactions that also want to access *X* are placed in the same queue.Hence, the wait command is considered to be outside the lock_item operation.
- It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item X in the database
- In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item
- 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*.
  4. A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

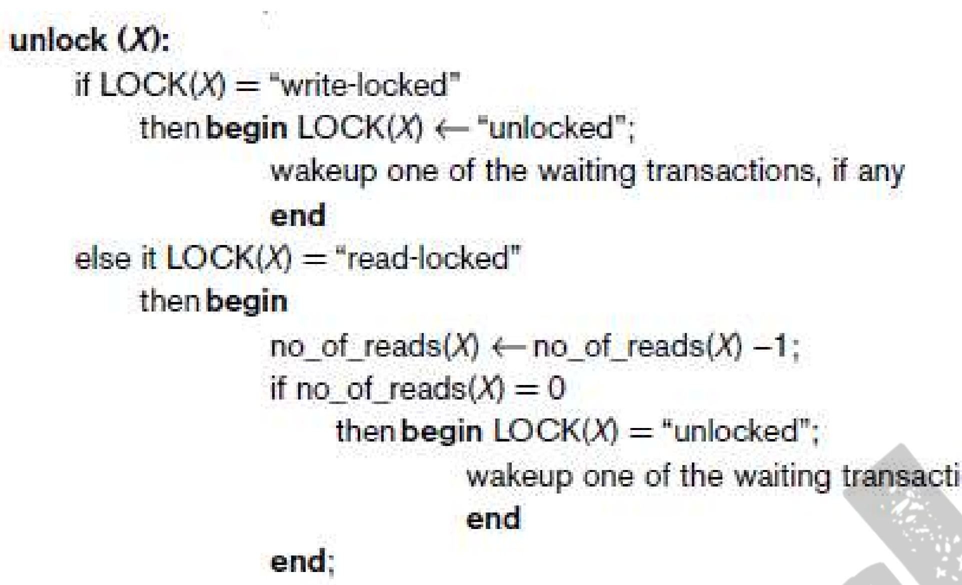
**2. Shared/Exclusive (or Read/Write) Locks**

- binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item
- should allow several transactions to access the same item X if they all access X for reading purposes only
- 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 **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
- Method to implement read/write lock is to
  - keep track of the number of transactions that hold a shared (read) lock on an item in the lock table
  - Each record in the lock table will have four fields: <Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>.
- If LOCK(*X*)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on *X*
- If LOCK(*X*)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on *X*.



```
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;
```

- 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.

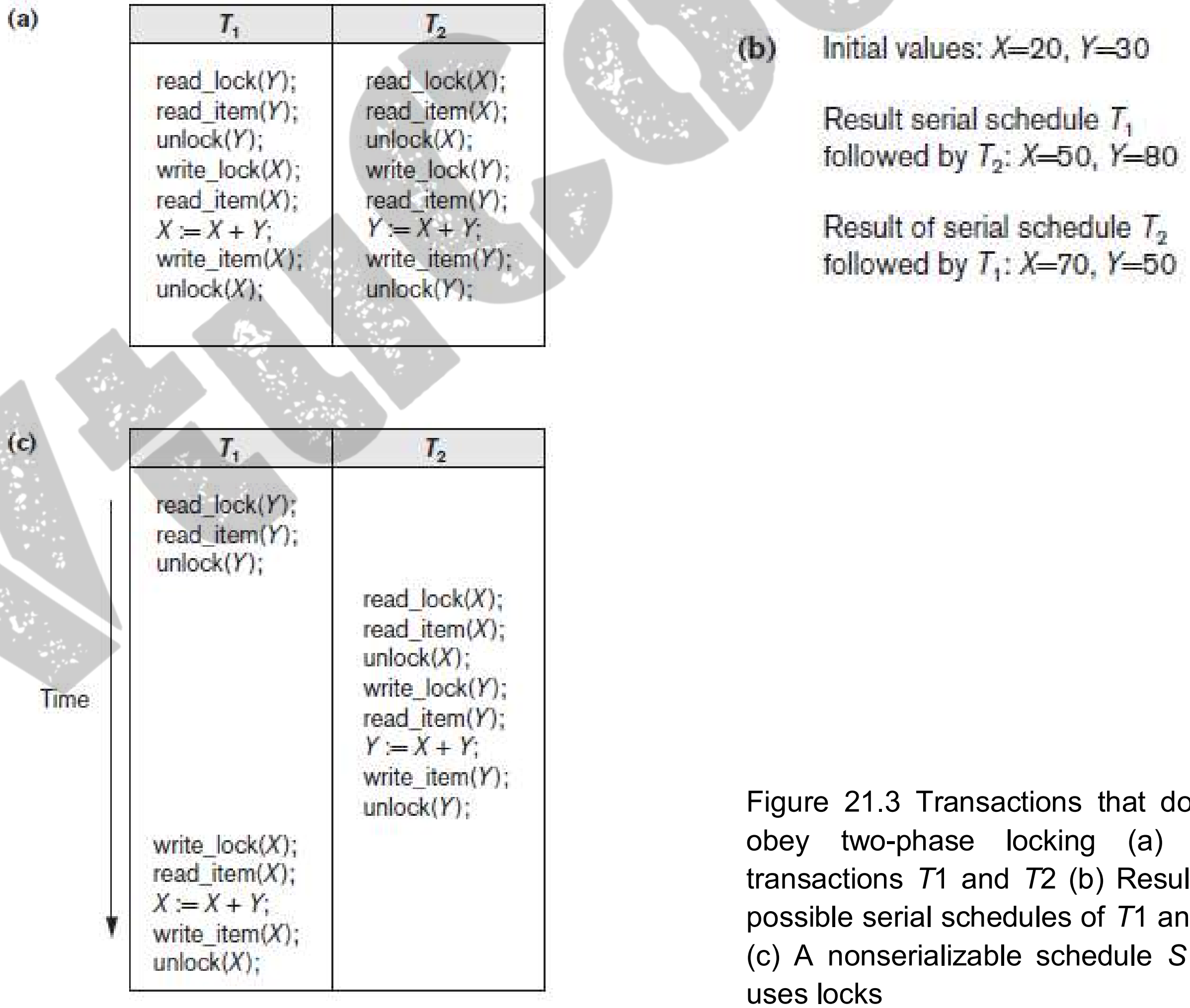
**Conversion of Locks**

- 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

## 5.9.2 Guaranteeing Serializability by Two-Phase Locking

- A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read_lock, write_lock) precede the *first* unlock operation in the transaction
- Such a transaction can be divided into two phases:
    - **Expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released
    - **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 *T*1 and *T*2 in Figure 22.3(a) do not follow the two-phase locking protocol because the write_lock(*X*) operation follows the unlock(*Y*) operation in *T*1, and similarly the write_lock(*Y*) operation follows the unlock(*X*) operation in *T*2.



(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

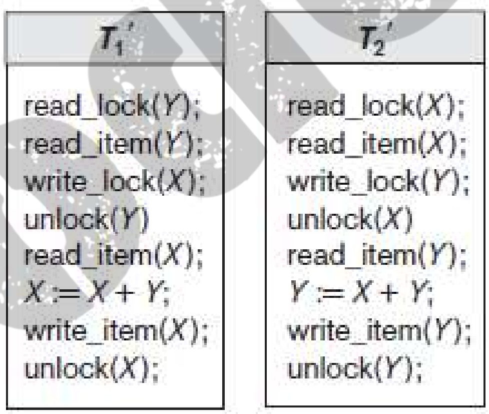
Result of serial schedule $T_2$ followed by $T_1$: $X$=70, $Y$=50

(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($X$); | |

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

- If we enforce two-phase locking, the transactions can be rewritten as *T*1’ and *T*2’ as shown in Figure 22.4.
- Now, the schedule shown in Figure 22.3(c) is not permitted for *T*1_ and *T*2_ (with their modified order of locking and unlocking operations) under the rules of locking because *T*1_ will issue its write_lock(*X*) *before* it unlocks item *Y*; consequently, when *T*2_ issues its read_lock(*X*), it is forced to wait until *T*1_ releases the lock by issuing an unlock (*X*) in the schedule.



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

**Figure 22.4**
Transactions $T_1'$ and $T_2'$, which are the same as $T_1$ and $T_2$ in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

## 5.10 Variations of Two-Phase Locking

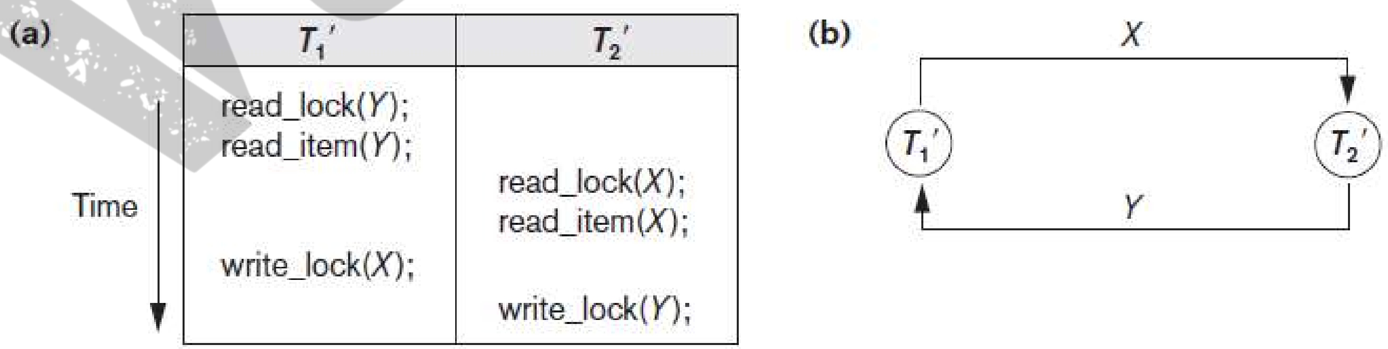
- **Basic 2PL**
  - Technique described previously
- **Conservative (static) 2PL**
  - Requires a transaction to lock all the items it accesses before the transaction begins execution by predeclaring read-set and write-set
  - Its Deadlock-free protocol

- **Strict 2PL**
  - guarantees strict schedules
  - Transaction does not release exclusive locks until after it commits or aborts
  - 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**
  - guarantees strict schedules
  - Transaction does not release any locks until after it commits or aborts
  - easier to implement than strict 2PL

## 5.11 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 22.5(a), where the two transactions *T*1' and *T*2_'are deadlocked in a partial schedule; *T*1' is in the waiting queue for *X*, which is locked by *T*2', while *T*2' is in the waiting queue for *Y*, which is locked by *T*1'. Meanwhile, neither *T*1' nor *T*2' nor any other transaction can access items *X* and *Y*



| $T_1'$ | $T_2'$ |
|---|---|
| read_lock(*Y*);<br>read_item(*Y*); | |
| | read_lock(*X*);<br>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)

## 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*. 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.
- 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
- Both approaches impractical
- 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($T_2$).
- The *older* transaction (which starts first) has the *smaller* timestamp value.
- Protocols based on a timestamp
  - Wait-die
  - 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.
- In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted.
- The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it.

- Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing.
- It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created.
- Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created.
- Another group of protocols that prevent deadlock do not require timestamps. These include the
    - • no waiting (NW) and
    - • cautious waiting (CW) algorithms
  - **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.
    - – no transaction ever waits, so no deadlock will occur
    - – this scheme can cause transactions to abort and restart needlessly
  - **cautious waiting**
    - – 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 rules are as follows:
      - If *Tj* is not blocked (not waiting for some other locked item), then *Ti* is blocked and allowed to wait; otherwise abort *Ti.*
    - – It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction.

## 5.12 Deadlock Detection.

- A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists.
- This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time.

- This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light.
- On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may be advantageous to use a deadlock prevention scheme.
- A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**.
- One node is created in the wait-for graph for each transaction that is currently executing.
- Whenever a transaction *Ti* is waiting to lock an item *X* that is currently locked by a transaction *Tj*, a directed edge (*Ti* → *Tj*) is created in the wait-for graph.
- When *Tj* releases the lock(s) on the items that *Ti* was waiting for, the directed edge is dropped from the wait-for graph.We have a state of deadlock if and only if the wait-for graph has a cycle.
- One problem with this approach is the matter of determining *when* the system should check for a deadlock.
- One possibility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead.
- Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 22.5(a).

– If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted.

– Choosing which transactions to abort is known as **victim selection**.

– The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

- **Timeouts**
  - Another simple scheme to deal with deadlock is the use of **timeouts**.
  - This method is practical because of its low overhead and simplicity.
  - In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

- **Starvation.**

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

## 5.13 Concurrency Control Based on Timestamp Ordering

guarantees serializability using transaction timestamps to order transaction execution for an equivalent serial schedule

### 5.13.1 Timestamps

- **timestamp** is a unique identifier created by the DBMS to identify a transaction.
- Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*.
- We will refer to the timestamp of transaction *T* as **TS(*T*)**.
- Concurrency control techniques based on timestamp ordering do not use locks;hence, *deadlocks cannot occur*.
- Timestamps can be generated in several ways.
  - One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3,

... in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.

- Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

### 5.13.2 The Timestamp Ordering Algorithm

- The idea for this scheme is to order the transactions based on their timestamps.
- A schedule in which the transactions participate is then serializable, and the only equivalent serial schedule permitted has the transactions in order of their timestamp values. This is called **timestamp ordering (TO**).
- The algorithm must ensure that, for each item accessed by *conflicting Operations* in the schedule, the order in which the item is accessed does not violate the 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.

### Basic Timestamp Ordering (TO).

- Whenever some transaction *T* tries to issue a read_item(*X*) or a write_item(*X*) operation, the **basic TO** algorithm compares the timestamp of *T* with read_TS(*X*) and write_TS(*X*) to ensure that the timestamp order of transaction execution is not violated.
- If this order is violated, then transaction *T* is aborted and resubmitted to the system as a new transaction with a *new timestamp*.
- If *T* is aborted and rolled back, any transaction *T*1 that may have used a value written by *T* must also be rolled back.

- Similarly, any transaction *T*2 that may have used a value written by *T*1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable.
- An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict.
- **The basic TO algorithm :**
    - 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 is checked:
    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 is checked:
    a. If write_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation. This should be done because some younger transaction with timestamp greater than TS(*T*)—and hence *after T* in the timestamp ordering—has already written the value of item *X* before *T* had a chance to read *X*.
    b. If write_TS(*X*) ≤ TS(*T*), then execute the read_item(*X*) operation of *T* and set read_TS(*X*) to the *larger* of TS(*T*) and the current read_TS(*X*).

    - Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*

**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 *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).

  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*).

## 5.14 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
- An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items

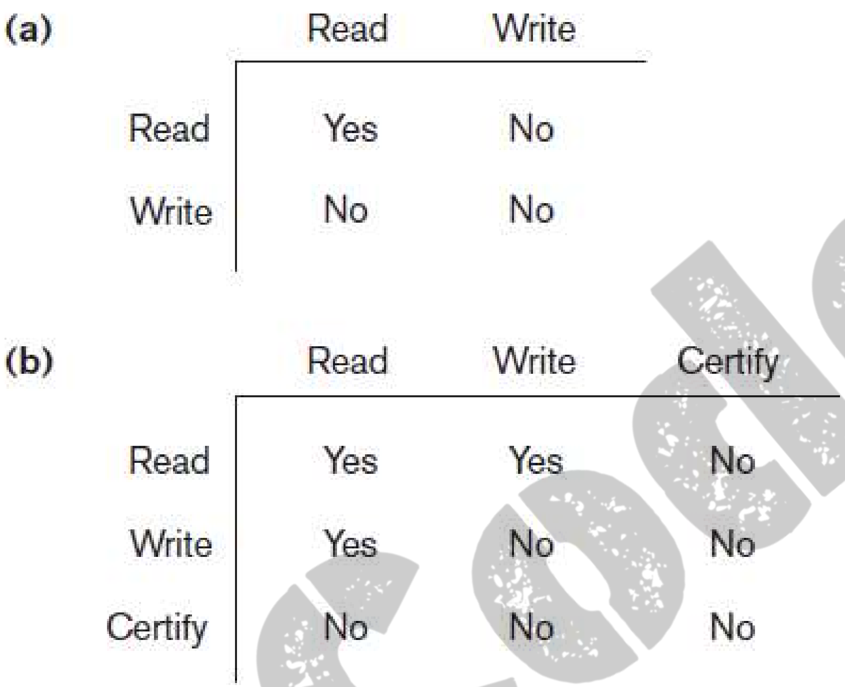
### 5.14.1 Multiversion Technique Based on Timestamp Ordering

- In this method, several versions *X*1, *X*2, ..., *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*).** The **read timestamp** of *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 *T* is 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 *X* that is also *less than or equal to* TS(*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*).

### 5.14.2 Multiversion Two-Phase Locking Using Certify Locks

- In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and *certify*
- Hence, the state of LOCK(*X*) for an item *X* can be one of read-locked, writelocked, certify-locked, or unlocked
- We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a)
- An entry of *Yes* means that if a transaction *T* holds the type of lock specified in the column header on item *X* and if transaction *T*_ requests the type of lock specified in

the row header on the same item *X*, then *T_ can obtain the lock* because the locking modes are compatible



(a)

|  | Read | Write |
|---|---|---|
| Read | Yes | No |
| Write | No | No |

(b)

|  | Read | Write | Certify |
|---|---|---|---|
| Read | Yes | Yes | No |
| Write | Yes | No | No |
| Certify | No | No | No |

**Figure 22.6:** Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

- On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so *T' must wait* until *T releases* the lock
- The idea behind multiversion 2PL is to allow other transactions *T'* to read an item *X* while a single transaction *T* holds a write lock on *X*
- This is accomplished by allowing *two versions* for each item *X*; one version must always have been written by some committed transaction
- The second version *X'* is created when a transaction *T* acquires a write lock on the item

## 5.15Validation (Optimistic) Concurrency Control Techniques

- In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing
- In this scheme, updates in the transaction are *not* applied directly to the database items until the transaction reaches its end

- During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction
- At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability.
- There are three phases for this concurrency control protocol:
  1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
  2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
  3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.
- The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached
- The techniques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.
- The validation phase for *Ti* checks that, for *each* such transaction *Tj* that is either committed or is in its validation phase, *one* of the following conditions holds:
  1. Transaction *Tj* completes its write phase before *Ti* starts its read phase.
  2. *Ti* starts its write phase after *Tj* completes its write phase, and the read_set of *Ti* has no items in common with the write_set of *Tj*.
  3. Both the read_set and write_set of *Ti* have no items in common with the write_set of *Tj*, and *Tj* completes its read phase before *Ti* completes its read phase.

## 5.16 Granularity of Data Items and Multiple Granularity Locking

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

■ A database record

■ A field value of a database record

■ A disk block

■ A whole file

  - The whole database
- The granularity can affect the performance of concurrency control and recovery

### 5.16.1 Granularity Level Considerations for Locking

- The size of data items is often called the **data item granularity**.
- *Fine granularity* refers to small item sizes, whereas *coarse granularity* refers to large item sizes
- The larger the data item size is, the lower the degree of concurrency permitted.
- For example, if the data item size is a disk block, a transaction *T* that needs to lock a record *B* must lock the whole disk block *X* that contains *B* because a lock is associated with the whole data item (block). Now, if another transaction *S* wants to lock a different record *C* that happens to reside in the same block *X* in a conflicting lock mode, it is forced to wait. If the data item size was a single record, transaction *S* would be able to proceed, because it would be locking a different data item (record).
- The smaller the data item size is, the more the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead
- The best item size *depends on the types of transactions involved*.
- If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record
- On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items

### 5.16.2 Multiple Granularity Level Locking

- Since the best granularity size depends on the given transaction, it seems appropriate that a database system should support multiple levels of granularity, where the granularity level can be different for various mixes of transactions
- Figure 22.7 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records.
- This can be used to illustrate a **multiple granularity level** 2PL protocol, where a lock can be requested at any level

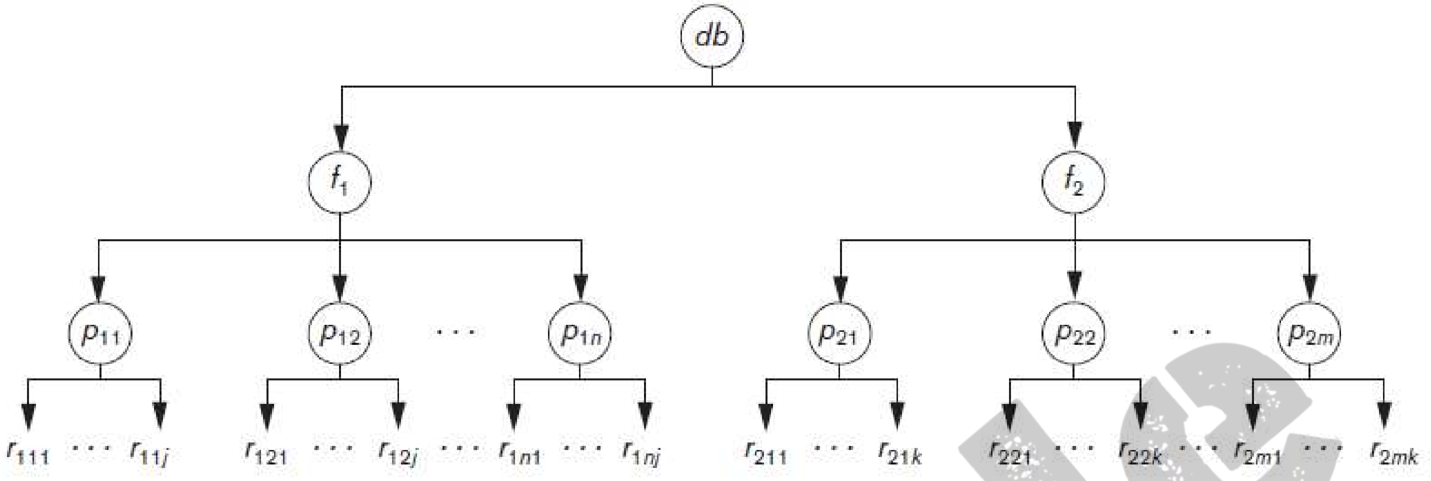
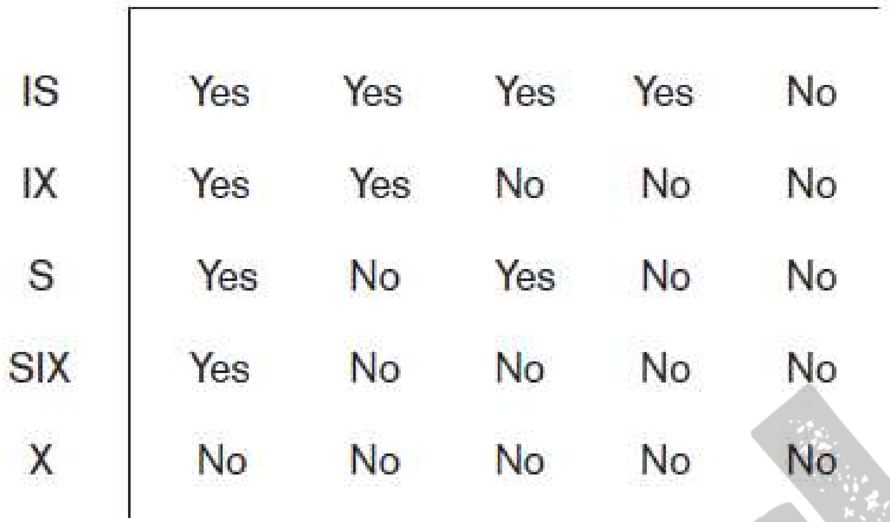


Figure 22.7 A granularity hierarchy for illustrating multiple granularity level locking

- To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed
- The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node’s descendants.
- There are three types of intention locks:
  1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
  2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).
  3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).
- The compatibility table of the three intention locks, and the shared and exclusive locks, is shown in Figure 22.8.



| | | | | | |
|---|---|---|---|---|---|
| IS | Yes | Yes | Yes | Yes | No |
| IX | Yes | Yes | No | No | No |
| S | Yes | No | Yes | No | No |
| SIX | Yes | No | No | No | No |
| X | No | No | No | No | No |

**Figure 22.8:** Lock compatibility matrix for multiple granularity locking.

- The **multiple granularity locking (MGL)** protocol consists of the following rules:
    1. The lock compatibility (based on Figure 22.8) must be adhered to.
    2. The root of the tree must be locked first, in any mode.
    3. A node *N* can be locked by a transaction *T* in S or IS mode only if the parent node *N* is already locked by transaction *T* in either IS or IX mode.
    4. A node *N* can be locked by a transaction *T* in X, IX, or SIX mode only if the parent of node *N* is already locked by transaction *T* in either IX or SIX mode.
    5. A transaction *T* can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
    6. A transaction *T* can unlock a node, *N*, only if none of the children of node *N* are currently locked by *T*.
- The multiple granularity level protocol is especially suited when processing a mix of transactions that include

  (1) short transactions that access only a few items (records or fields) and

  (2) long transactions that access entire files.