PPT for Session No.: 4th, 5th and 6th of Unit-V

Prepared by: Dr. Pragnyaban Mishra
Professor-CSE, GITAM-Visakhapatnam

Course Code: CSEN2061

Course Name: DATABASE MANAGEMENT SYSTEMS

Branch: CSE, 3rd Yr

Semester: Even (2024-25)

Session No: 4th / Unit-V
Session Topic: Concurrency control schemes in DBMS: Lock based

Session No: 5th / Unit-V
Session Topic: Explanation about 2PL, Strict 2PL

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items.

A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations

that can be applied to it. Generally, there is one lock for each data item in the database.

Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

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

Two operations, lock_item and unlock_item, are used with binary locking. A transaction

requests access to an item *X* by first issuing a **lock_item(*X*)** operation. If LOCK(*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;
```

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

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

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

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

```
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. 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.

Conservative, Strict, and Rigorous Two-Phase Locking. There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**.

A variation known as **conservative 2PL** (or **static 2PL**)
requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. the **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a *deadlock-free protocol.*

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. 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

Example of 2PL Protocol

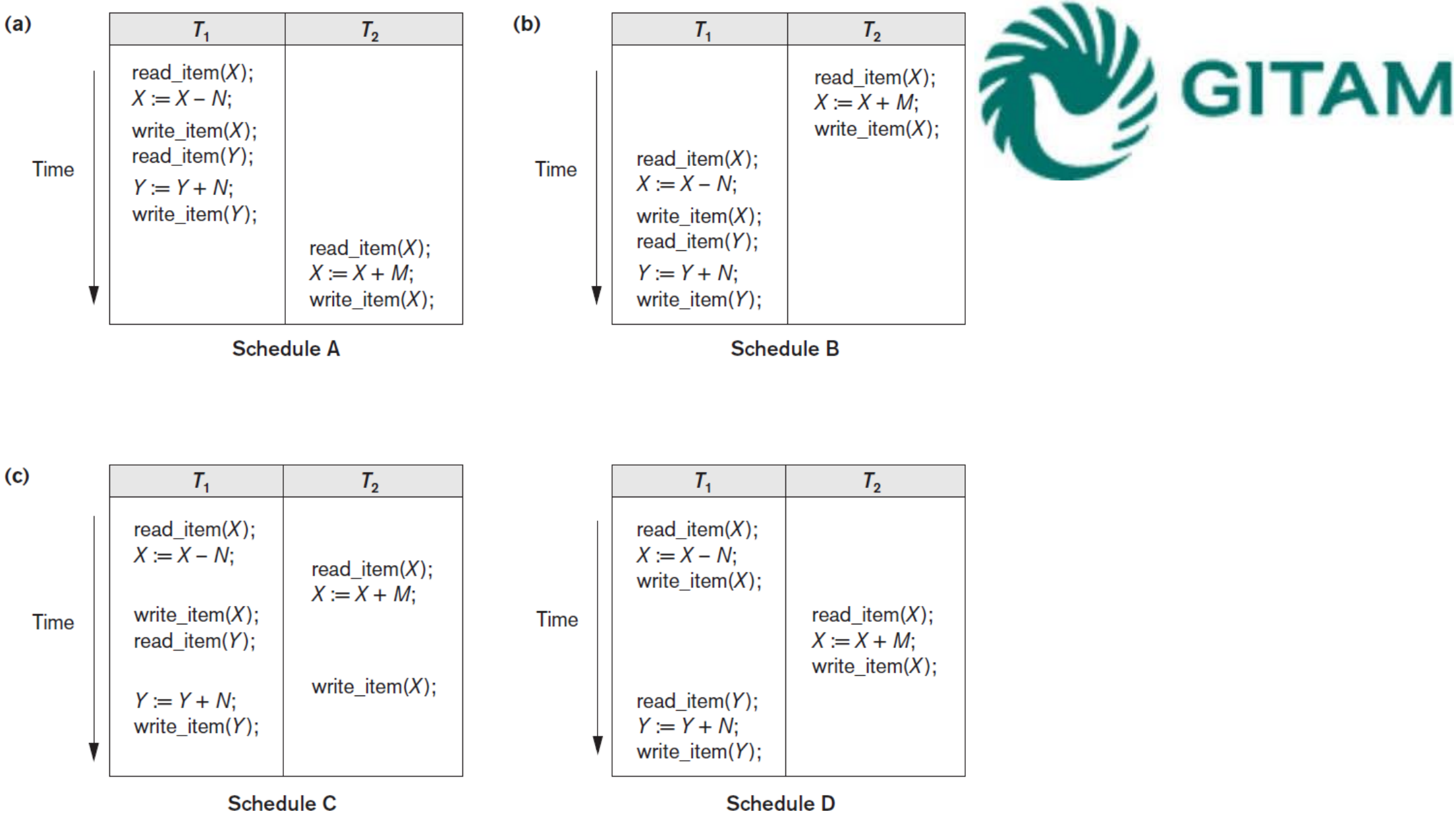
T_1	T_2

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

Session No: 6th / Unit-V

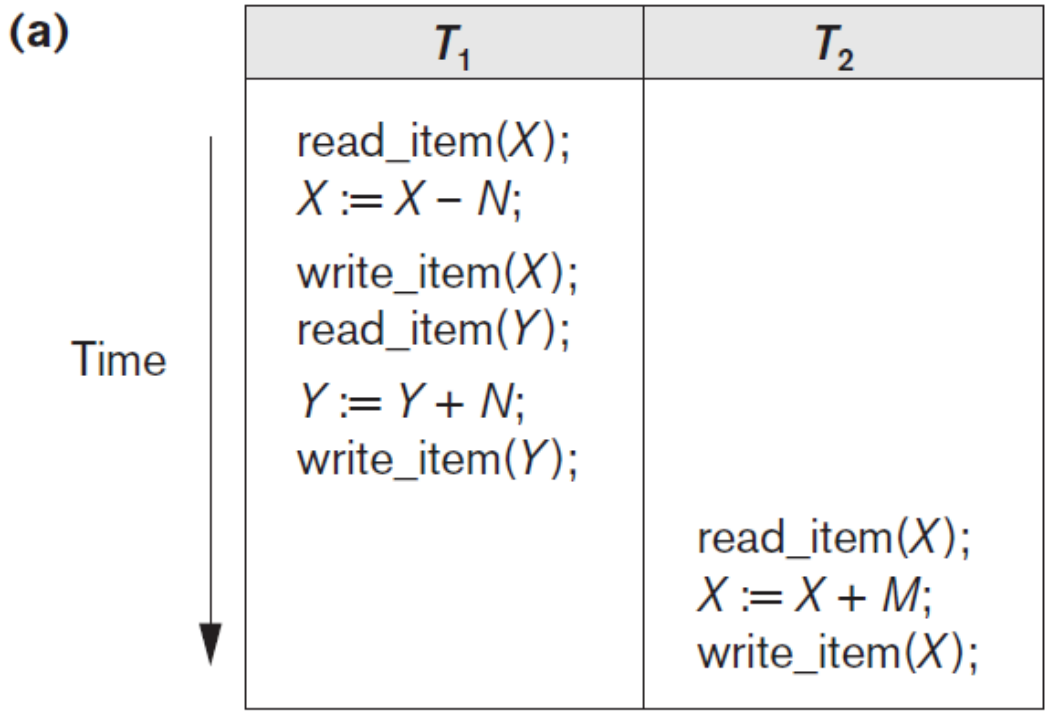
Session Topic: Introduction to Serializability: View Serializability; Conflict Serializability; Recoverability

Formally, a schedule *S* is **serial** if, for every transaction *T* participating in the schedule, all the operations of *T* are executed consecutively in the schedule; otherwise, the schedule is called **nonserial**. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that *every serial schedule is considered correct*.



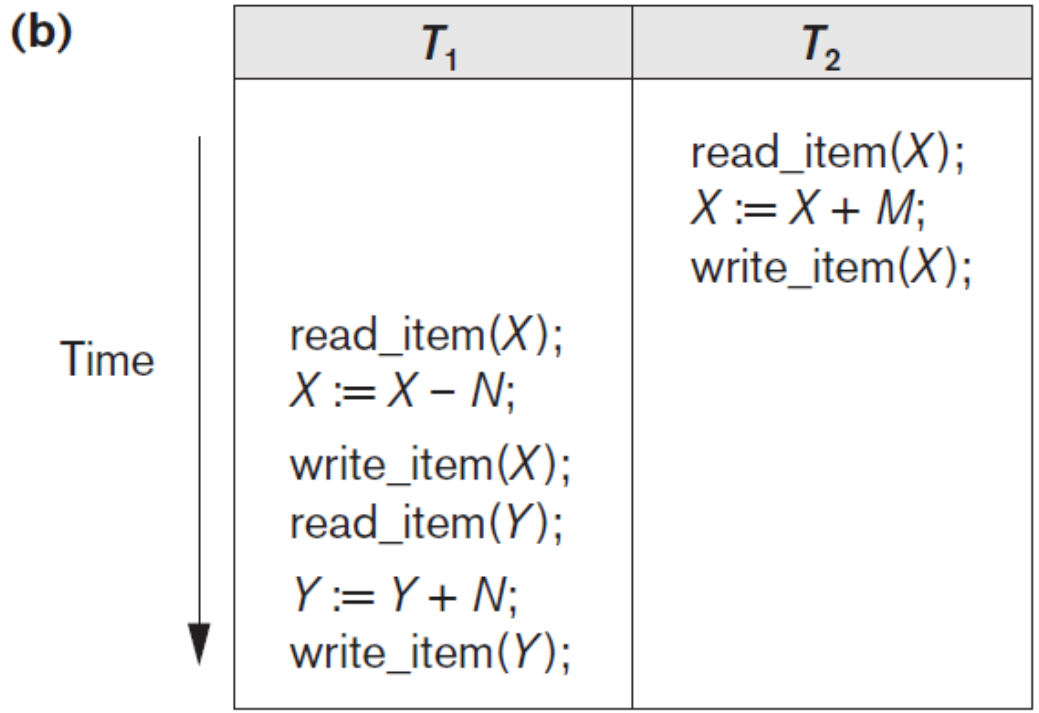
(a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.

A **schedule** (or **history**) *S* of *n* transactions *T*1, *T*2, … , *Tn* is an ordering of the operations of the transactions. Operations from different transactions can be interleaved in the schedule *S*.



Schedule A

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



Schedule B

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

Conflicting Operations in a Schedule:-

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

- They belong to different transactions
- They access the same item X
- At least one of the operations is a write_item(X).

Type of Conflicts:-

read-write conflict, write-write conflict

A schedule *S* of *n* transactions
is **serializable** if it is *equivalent to some serial schedule* of the same *n* transactions.

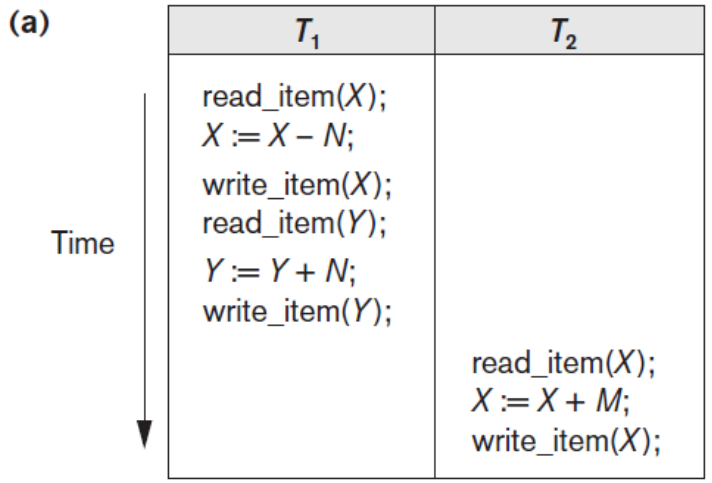
Conflict Equivalence of Two Schedules. Two schedules are said to be **conflict equivalent** if the relative order of any two *conflicting operations* is the same in both
schedules.

Algorithm 20.1. Testing Conflict Serializability of a Schedule S

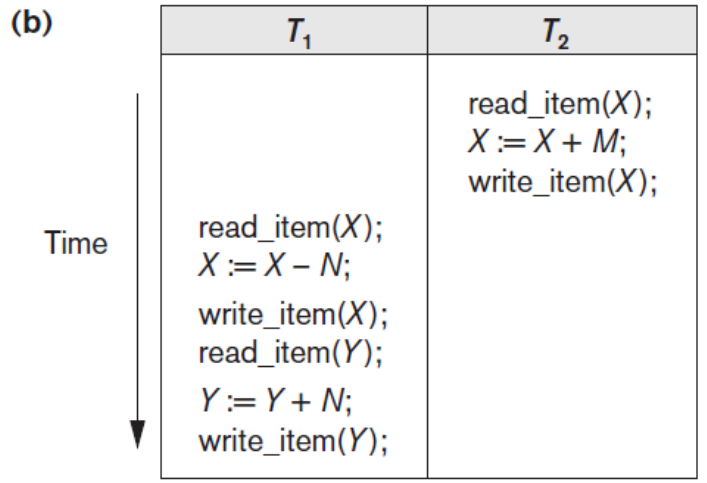
1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge $(T_i \rightarrow T_j)$ in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

The precedence graph is constructed as described in Algorithm 20.1. If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable. A **cycle** in a directed graph is a **sequence of edges** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_p), \ldots, (T_i \rightarrow T_j))$ with the property that the starting node of each edge—except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node).

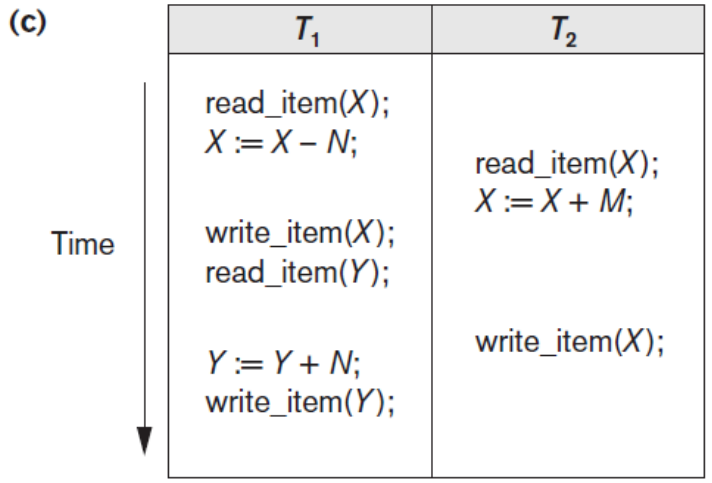
Check Conflict Serializable using Precedency Graph



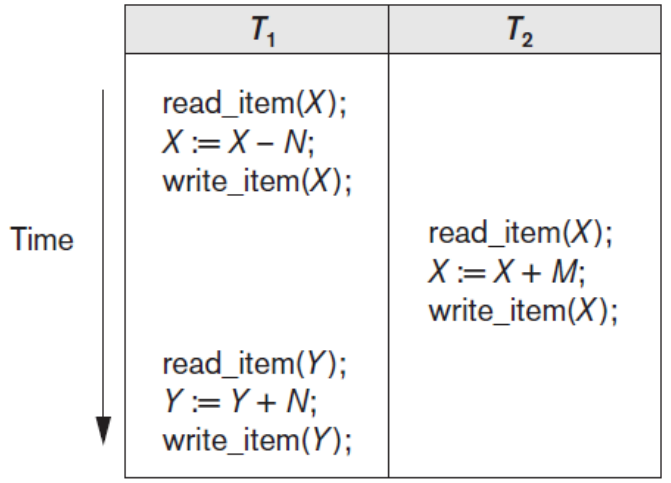
Schedule A



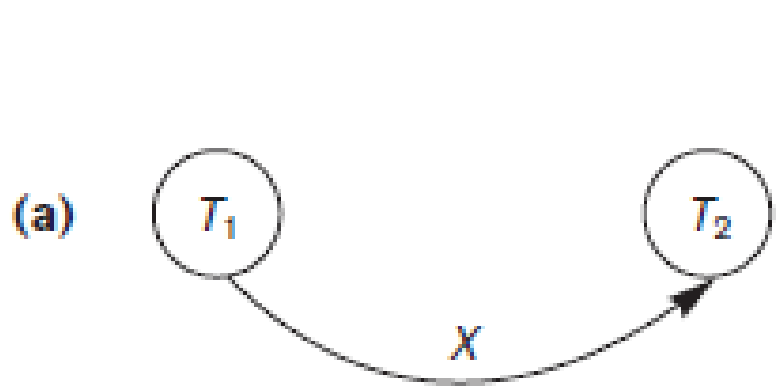
Schedule B

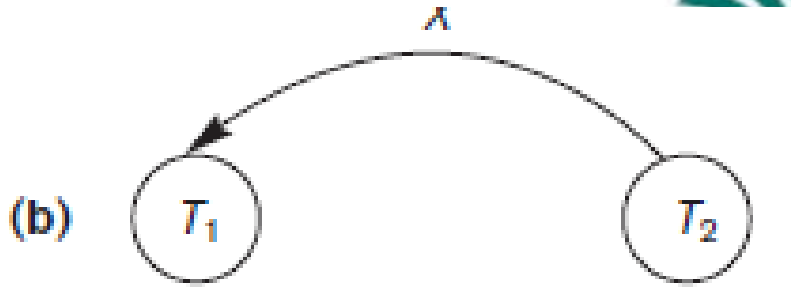


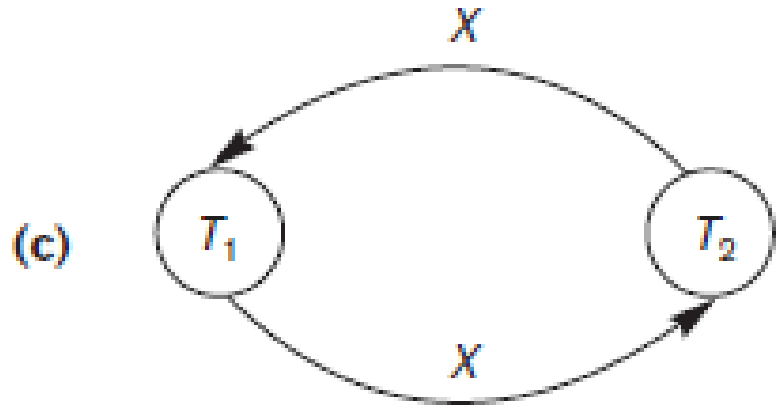
Schedule C

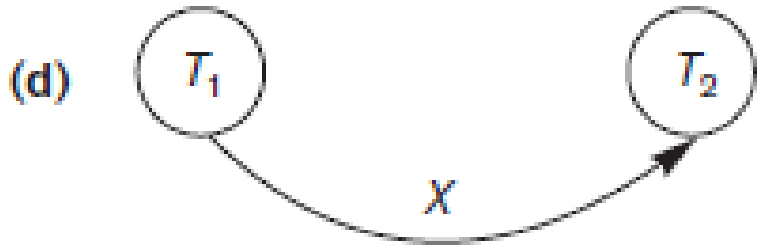


Schedule D


(a)
T_1
T_2
X


(b)
T_1
T_2


(c)
X
T_1
T_2
X


(d)
T_1
T_2
X

called *view serializability*. Two schedules S and S' are said to be **view equivalent** if the following three conditions hold:

1. The same set of transactions participates in S and S', and S and S' include the same operations of those transactions.
2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S'.
3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S'.

A schedule *S* is said to be view serializable if it is view equivalent to a serial schedule.

Session No: 6th / Unit-V

Session Topic: Recoverability

A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS. The condition for a **recoverable schedule** is as follows:

- A schedule *S* is recoverable if no transaction *T* in *S* commits until all transactions *T′* that have written some item *X* that *T* reads have committed.

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

Log based Recovery in DBMS

Log and log records

1.Transaction identifier: Unique Identifier of the transaction that performed the write operation.
2.Data item: Unique identifier of the data item written.
3.Old value: Value of data item prior to write.
4.New value: Value of data item after write operation.

Other types of log records are:
1.<Ti start>: It contains information about when a transaction Ti starts.
2.<Ti commit>: It contains information about when a transaction Ti commits.
3.<Ti abort>: It contains information about when a transaction Ti aborts.

Undo and Redo Operations

Because all database modifications must be preceded by the creation of a log record, the system has available both the old value prior to the modification of the data item and new value that is to be written for data item. This allows system to perform redo and undo operations as appropriate:

1.Undo: using a log record sets the data item specified in log record to old value.

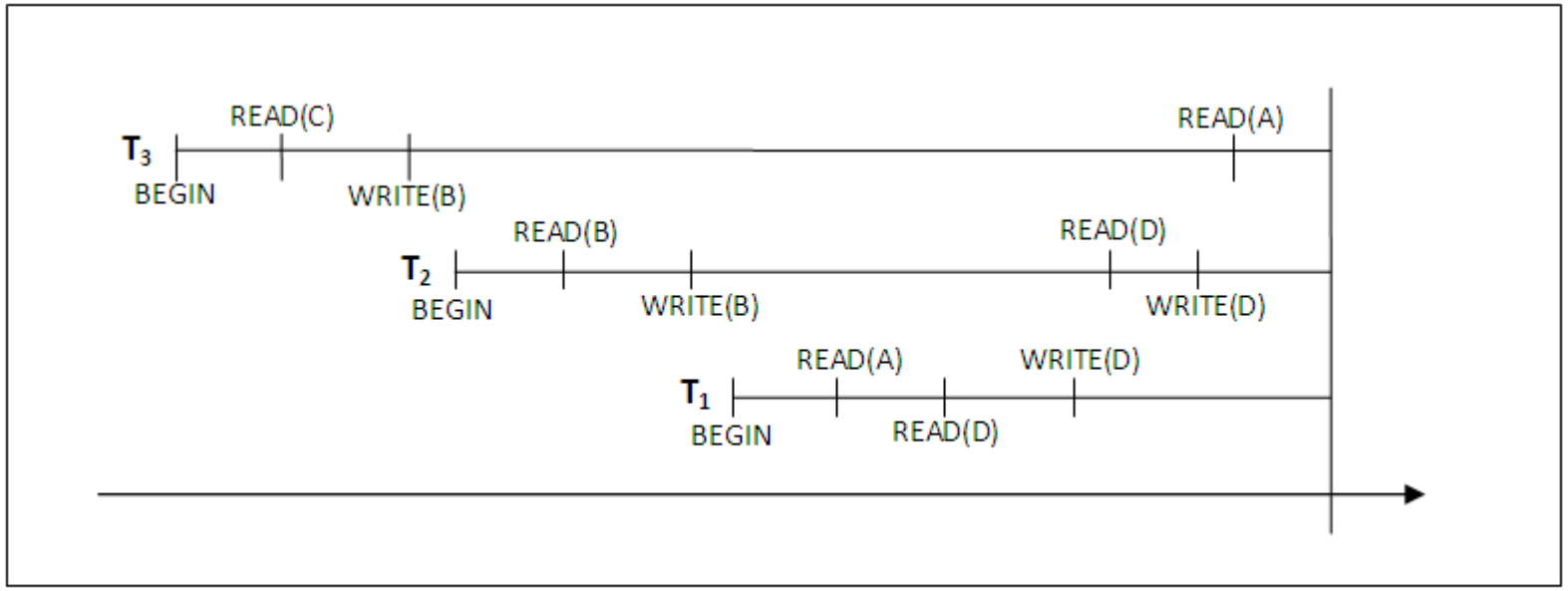
2.Redo: using a log record sets the data item specified in log record to new value.

Types of Recovery Techniques in DBMS

- Rollback/Undo Recovery Technique
- Commit/Redo Recovery Technique
- CheckPoint Recovery Technique

Transaction rollback

T_1	T_2	T_3
Read_item(A);	read_item(B);	Read_item(C);
Read_item(D);	write_item(B);	Write_item(B);
Write_item(D);	read_item(D);	Read_item(A);
	write_item(D);	Write_item(A);



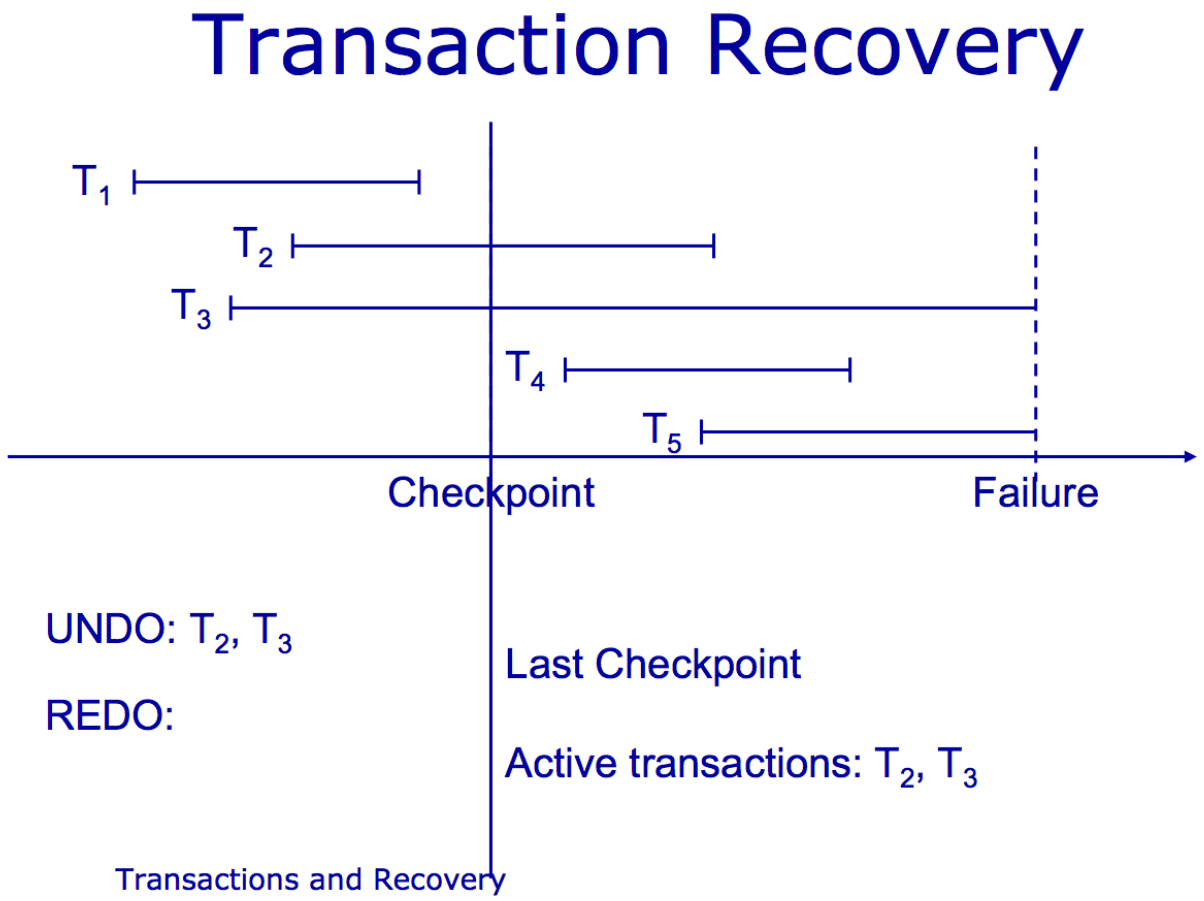
Log	A	B	C	D
	30	15	40	20
start_transaction(T_3)				
read_item(T_3, C)				
*write_item(T_3, B, 15, 12)		12		
start_transaction(T_2)				
read_item(T_2, B)				
**write_item(T_2, B, 12, 18)		18		
start_transaction(T_1)				
read_item(T_1, A)				
read_item(T_1, D)				
write_item(T_1, D, 20, 25)				25
read_item(T_2, D)				
**write_item(T_2, D, 25, 26)				26
read_item(T_3, A)				
System crash				

* T_3 is rolled back because it did not reach its commit point when system crash happens.

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

The rest of the write entries in the log are redone.

GITAM



The transactions T3 and T5 are in the UNDO queue. (checked either commit statement is executed prior to the failure or not
The transactions T2 and T4 are in the REDO queue.
T1 is completed before check point so free from undo and redo operation.

Example of Recovery

:
<T_0 **start**>
<T_0, *A*, 0, 10>
<T_0 **commit**>
<T_1 **start**>
<T_1, *B*, 0, 10>
<T_2 **start**>
<T_2, *C*, 0, 10>
<T_2, *C*, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, *A*, 10, 20>
<T_3, *D*, 0, 10>
<T_3 **commit**>

Redo-list{T3}
Undo-list{T1, T2}

Undo:
Set C to 10
Set C to 0
Set B to 0

Redo:
Set A to 20
Set D to 10

DB	A	B	C	D
Initial	0	0	0	0
At crash	20	10	20	10
After rec.	20	0	0	10