Chapter 3

Concurrency Control Techniques

Outline

- Introduction
- Concurrency control techniques
  - Locking techniques
  - Timestamp ordering techniques
  - Multi version concurrency control techniques

Introduction

•**Concurrency control** is the activity coordinating concurrent accesses to a database in a multi-user database management system (DBMS).

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.

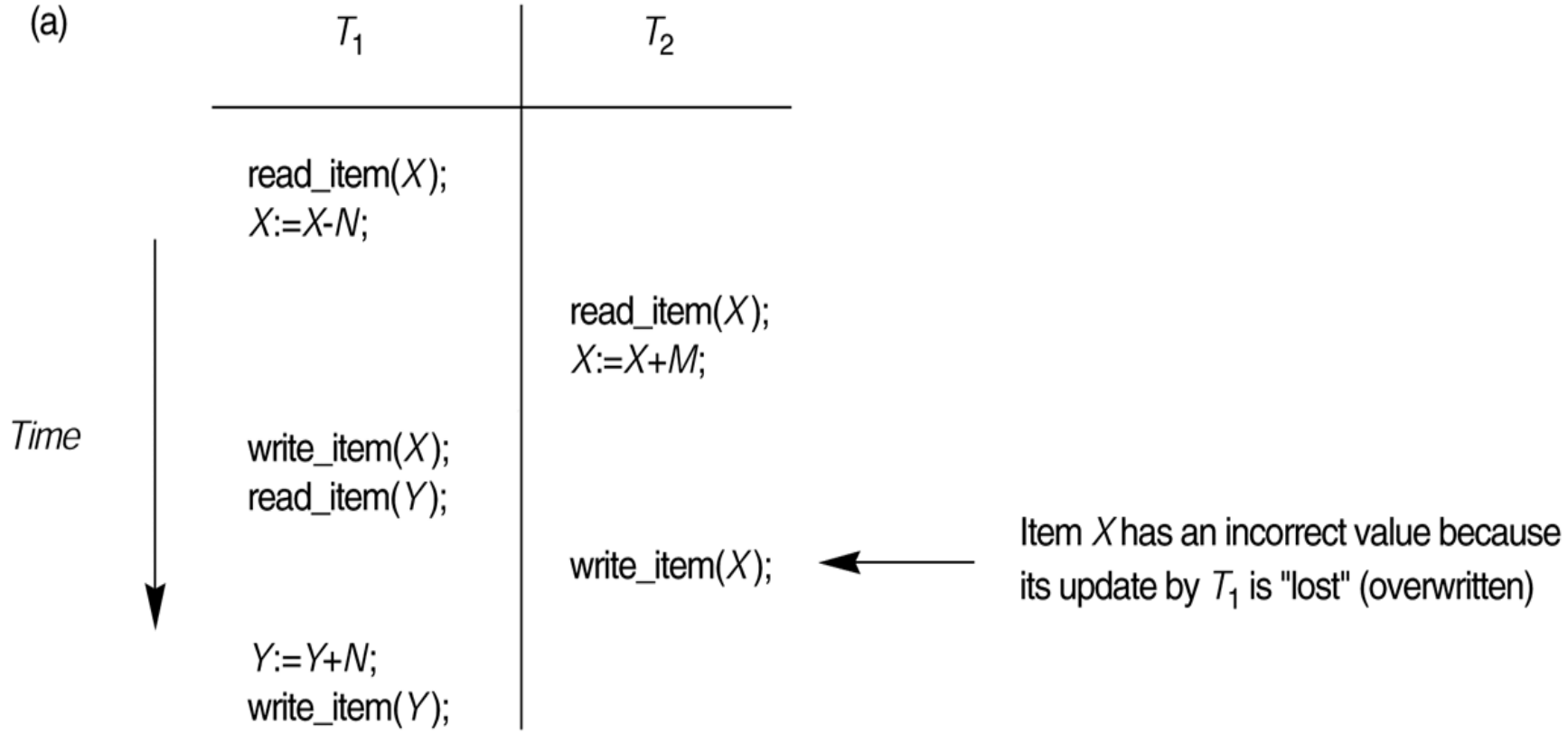
Cont’d

- During concurrent access to the data item three problems will occur
  1. The lost update problem
  2. The temporary update (dirty read) problem
  3. Incorrect summary problem

The Lost Update

- The lost update problem occurs when 2 concurrent transactions try to read and update the same data
- In a schedule, if update performed by transaction T1 on data item ‘X’ gets overwritten by the update performed by transaction T2 on same data item ‘X’, then we say that update of T1 is lost to the update of T2.

Example of Lost update problem



For example, if X = 80 at the start, N = 5 and M = 4 the final result should be X = 79; but in the interleaving of operations, it is X = 84 because the update in T_1that removed 5 from X was *lost.*

Temporary Update or Dirty Read

- **This problem occurs when one** transaction updates a database item and then the transaction fails for some reason Meanwhile, the updated item is accessed (read) by another transaction before it is changed back (or rolled back) to its original value
- Occurs when one transaction can see intermediate results of another transaction before it has committed.

Cont'd

- **Problem avoided by preventing T_3 from reading bal_x until after T_4 commits or aborts.**

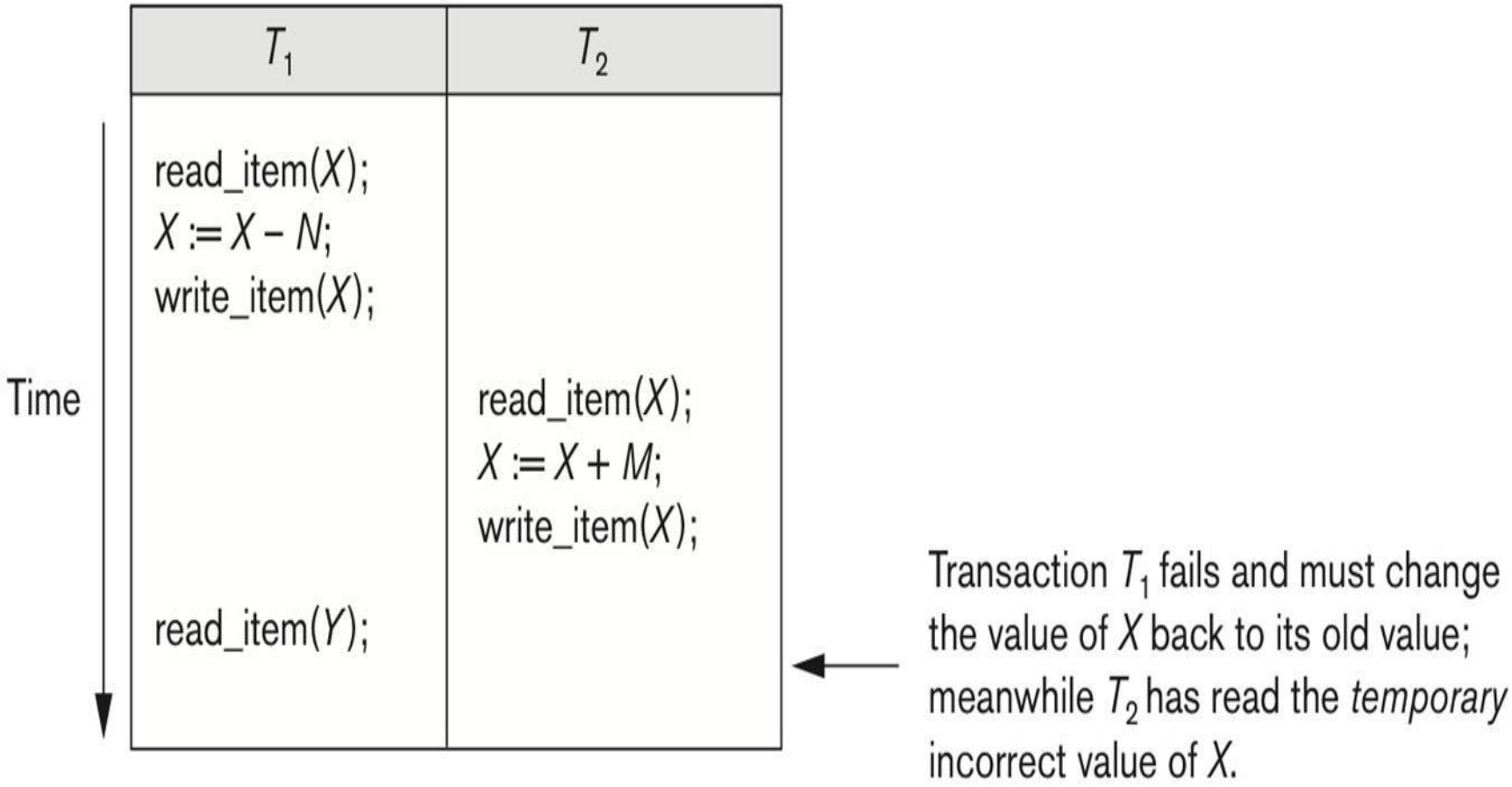
Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	⋮	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

Example:

T_4 updates bal_x to 200Birr but it aborts, so bal_x should be back at original value of 100Birr.

T_3 has read new value of bal_x (200Birr) and uses value as basis of 10Birr reduction, giving a new balance of 190Birr, instead of 90Birr.

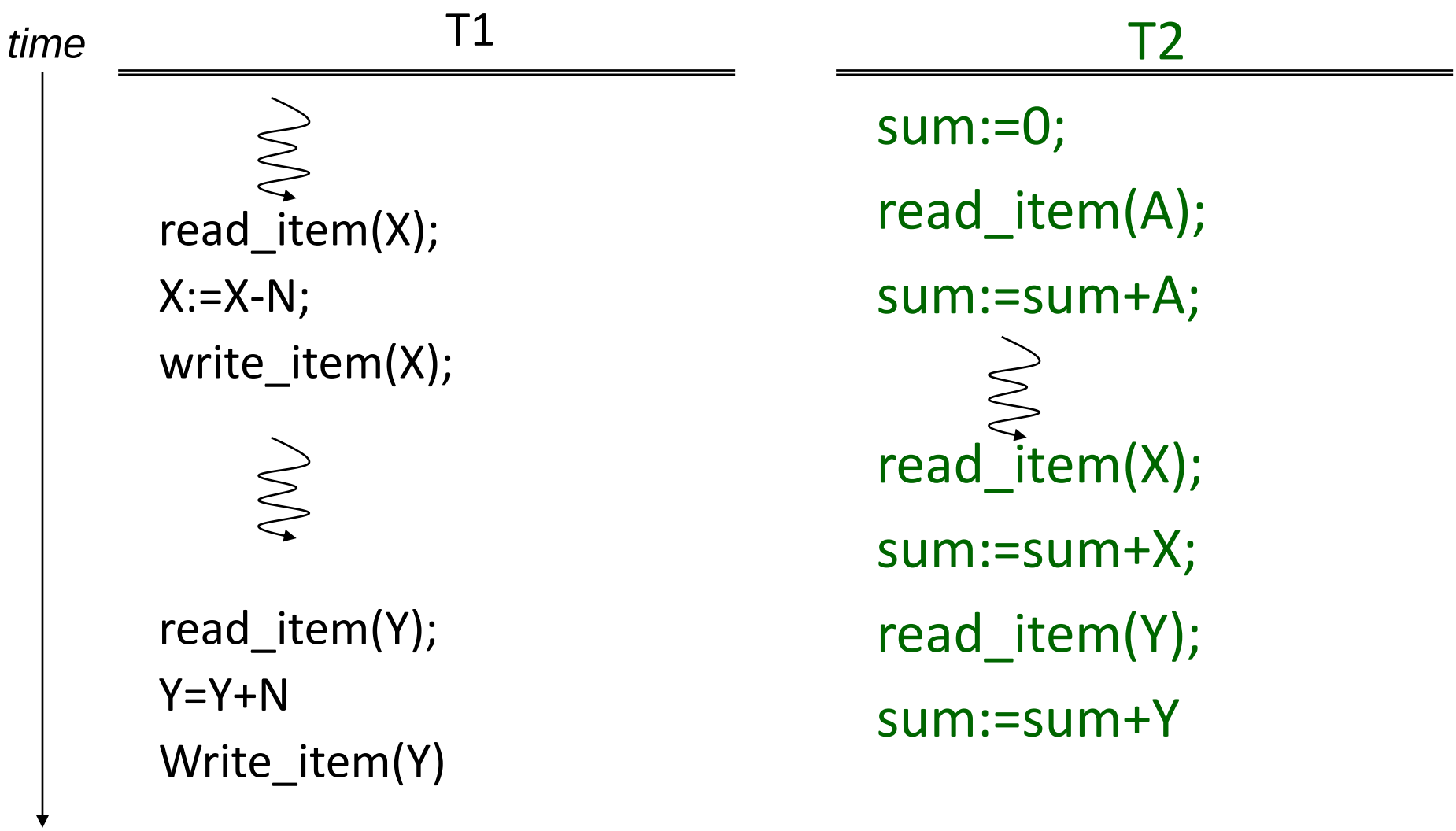
Cont'd


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

The Incorrect Summary

- If one transaction is calculating an aggregate summary function on a number of database items while
- other transactions are updating some of these items the aggregate function may calculate some values before they are updated and others after they are update
- Occurs when one transaction reads several values but second transaction updates some of them during execution of first.
- This causes a inconsistent database state.

Cont'd


time
T1
T2
read_item(X);
X:=X-N;
write_item(X);
read_item(Y);
Y=Y+N
Write_item(Y)
sum:=0;
read_item(A);
sum:=sum+A;
read_item(X);
sum:=sum+X;
read_item(Y);
sum:=sum+Y

Types of Failures.

- Failures are generally classified as transaction, system, and media failures.
- A computer failure (system crash).
  - ✓ A hardware, software, or network error occurs
- A transaction or system error.
  - ✓ Some operation in the transaction may cause it to fail, such as integer overflow or division by zero
- Local errors or exception conditions detected by the transaction.
  - ✓ certain conditions may occur that necessitate cancellation of the transaction data for Txn missing, insufficient account balance

Concurrency Control Techniques

Categories of concurrency techniques

1. Locking techniques
2. Timestamp ordering techniques
3. Multi-version concurrency control techniques
4. Optimistic concurrency control techniques

Concurrency Control Using Locks

- **Lock**
  - A variable associated with a data item
  - Describes status of the data item with respect to operations that can be performed on it
- Types of locks
  - **Binary locks**
  - **Shared/Exclusive locks or Read/Write lock**

Cont'd

- **Locking** is an operation which secures
  - Permission to read
  - Permission to write a data item for a Txn
  - Ex. Lock(X). data item X is locked on behalf of Txn
- **Unlocking** is an operation which removes these permissions from the data item
  - Ex. Unlock(X). data item X is made available for all other Txns
  - Lock and Unlock are atomic operations

Binary Locks

- Can have Two states (values)
  - Locked (1)
    - Item cannot be accessed
  - Unlocked (0)
    - Item can be accessed when requested

Shared/exclusive or read/write locks

- Read operations on the same item are not conflicting
- Must have exclusive lock to write
- Three locking operations
  - read_lock(X)
  - write_lock(X)
  - unlock(X)

Locking Rules

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

1. T must issue read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T
2. T must issue write_lock(X) before any write_item(X) operation is performed in T
3. T must issue unlock(X) after all read_item(X) and write_item(X) operations are completed in T
4. T will not issue a read_lock(X) if it already holds a read lock or write lock on X (may be relaxed)
5. T will not issue a write_lock(X) if it already holds a read lock or write lock on X (may be relaxed)
6. T will not issue unlock (X) request unless it holds a read lock write lock on X

Cont’d

- These rules can be enforced by the lock manager module of the DBMS.
- Between the lock_item(X) and unlock_item(X) operations in transaction T, T is said to hold the lock on item X.
- At most one transaction can hold the lock on a particular item.
- Thus no two transactions can access the same item concurrently.

Conversion of Locks

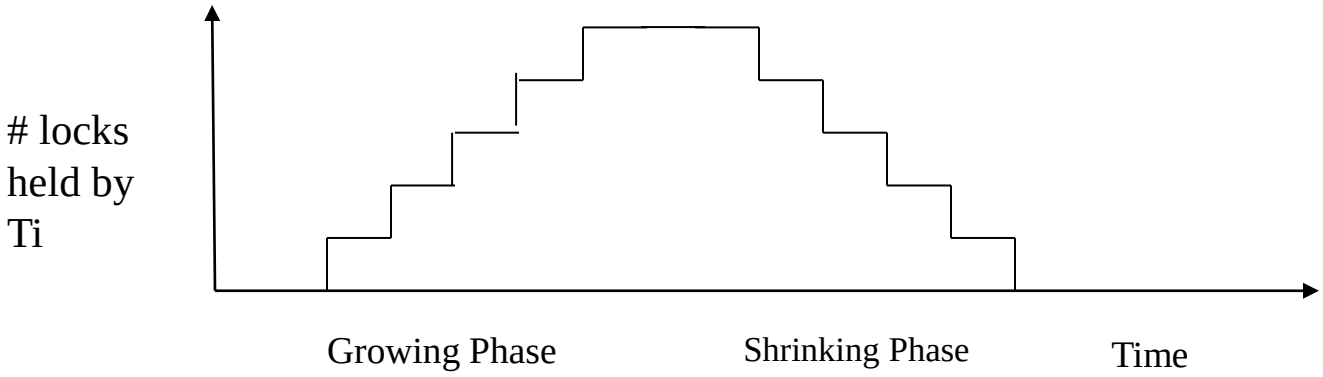
- A transaction T that holds a lock on item X, under certain conditions, is allowed to **convert** the lock from one **state** to **another**
  - Upgrade
    - Issue a read_lock operation then a write_lock operation
    - Convert read_lock to write_lock
  - Downgrade
    - Issue a write_lock operation then a read_lock operation
    - Convert write_lock to read_lock

Two-Phase Locking (2PL) Protocol

- Transaction is said to follow the two-phase-locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation

Cont'd

- Every transaction can be divided into Two Phases: Locking (Growing) & Unlocking (Shrinking)
  - **Locking (Growing) Phase:**
    - A transaction applies locks (read or write) on desired data items one at a time.
    - acquires all locks but cannot release any locks.
  - **Unlocking (Shrinking) Phase:**
    - A transaction unlocks its locked data items one at a time.
    - Releases locks but cannot acquire any new locks.
- **Requirement:**
  - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.



Cont'd

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);

- Transactions T1 and T2 **do not follow the two-phase locking protocol** because the write_lock(X) operation follows the unlock(Y) operation in T1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T2.

2PL Example

T_1'

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

T_2'

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

- **Both T1' and T2' follow the 2PL protocol**
- **Any schedule including T1' and T2' is guaranteed to be serializable**
- **Limits the amount of concurrency**

Cont’d

1. **Basic 2PL**
   All lock operations before the first unlock operation
2. **Conservative 2PL** or static 2PL
   Lock all the items it accesses **before** the transaction begins execution.
   - Deadlock free protocol
   - Read-set and write-set of the transaction should be known

Cont’d

3. Strict 2PL

No exclusive lock will be unlocked until the transaction **commits** or aborts

- Strict 2PL guarantees strict schedules

4. Rigorous 2PL

No lock will be **unlocked** until the transaction **commits** or aborts

Dealing with Deadlock and Starvation

- **Deadlock**
  - It is a state that may result when two or more transaction are each waiting for locks held by the other to be released

  Example :

T1	**T2**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
	write_lock (Y);

  - ✓ T1 is in the waiting queue for X which is locked by T2
  - ✓ T2 is on the waiting queue for Y which is locked by T1
  - ✓ No transaction can continue until the other transaction completes
  - ✓ T1 and T2 did follow two-phase policy but they are deadlock
- ✓ So the DBMS must either prevent or detect and resolve such deadlock situations

Cont’d

- **Starvation**
  - Occurs if a transaction cannot proceed for an indefinite period of time while other transactions continue normally.
  - **Unfair waiting scheme**
  - Solution: first-come-first-served queue
- There are possible solutions : Deadlock prevention, deadlock detection and avoidance

Cont’d

- **Deadlock prevention protocols**
  - **Conservative** 2PL, lock all needed items in advance
  - **Ordering** all items in the database
    - Transaction that needs several items will lock them in that order
- Possible actions if a transaction is involved in a possible deadlock situation
  - Block and wait
  - Abort and restart
  - Preempt and abort another transaction

Cont’d

- Two schemes that prevent deadlock and do not require timestamps
  - **No waiting algorithm**
    - If transaction unable to obtain a lock, immediately aborted and restarted after a certain time delay
  - **Cautious waiting** algorithm
    - If a transaction holding the lock is **not waiting** for another item to be locked then allow T to wait otherwise **abort** and **restart** T
    - **Deadlock-free**

Deadlock detection and timeouts

- **Victim selection**
  - Deciding which transaction to abort in case of deadlock
- **Timeouts**
  - If system waits longer than a predefined time, it aborts the transaction

Concurrency Control Based on Timestamp Ordering

- **Timestamp**
  - Unique **identifier assigned** by the DBMS to identify a transaction
  - Assigned in the order submitted
  - Transaction **start time**
- Concurrency control techniques based on **timestamps do not use locks**
  - Deadlocks cannot occur

Cont’d

- Generating timestamps
  - Counter incremented each time its value is assigned to a transaction
  - Current date/time value of the system clock
    - Ensure no two timestamps are generated during the same tick of the clock

Cont'd

- Timestamp ordering (TO)
  - Allows **interleaving** of transaction operations
  - Must ensure timestamp order is followed for **each pair** of **conflicting operations**
- Each database item assigned two timestamp values
  - **read_TS(X)**
    - The largest timestamp among all the timestamps of transactions that have successfully **read item X**
  - **write_TS(X)**
    - The largest timestamp among all the timestamps of transactions that have successfully **written item X**

Cont’d

- Timestamp Ordering (TO) algorithms
  - **Basic timestamp ordering**
  - **Strict timestamp ordering**
  - **Thomas’s write rule**

Multi version Concurrency Control Techniques

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction.
- This algorithm uses the concept of view serilazabilty than conflict serialiazabilty
- Multiversion currency control scheme types
  - Based on **timestamp** ordering
  - Based on **two-phase** locking

Cont'd

Multi version technique based on timestamp ordering

- Assume X1, X2, …, Xn are the version of a data item X created by a write operation of transactions.
- With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.)
  - **read_TS(Xi)**: The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.
  - **write_TS(Xi)**: The write timestamp of Xi that wrote the value of version Xi.
- A new version of Xi is created only by a write operation.

Cont'd

Multi version Two-Phase Locking Using Certify Lock

- Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T.
- This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction.
- This means a write operation always creates a new version of X.

Steps

1. X is the committed version of a data item.
2. T creates a second version X' after obtaining a write lock on X.
3. Other transactions continue to read X.
4. T is ready to commit so it obtains a certify lock on X'.
5. The committed version X becomes X'.
6. T releases its certify lock on X', which is X now.

Validation (Optimistic) Concurrency Control Techniques

- This technique allow transaction to proceed asynchronously and only at the time of commit, serializability is checked &
- transactions are aborted in case of non-serializable schedules.
- Good if there is little interference among transaction
- It has three phases:
  - ✓ **Read,**
  - ✓ **Validation , and**
  - ✓ **Write phase**

Cont'd

i. **Read phase:**
   - A transaction can read values of committed data items.
   - However, updates are applied only to local copies (versions) of the data items (in database cache).

ii. Validation phase:

- If the transaction Ti decides that it wants to commit, the DBMS checks whether the transaction could possibly have conflicted with any other concurrently executing transaction

iii **Write phase**:

- On a successful validation, transactions' updates are applied to the database; otherwise, transactions are restarted.