Unit 7

Transaction Management

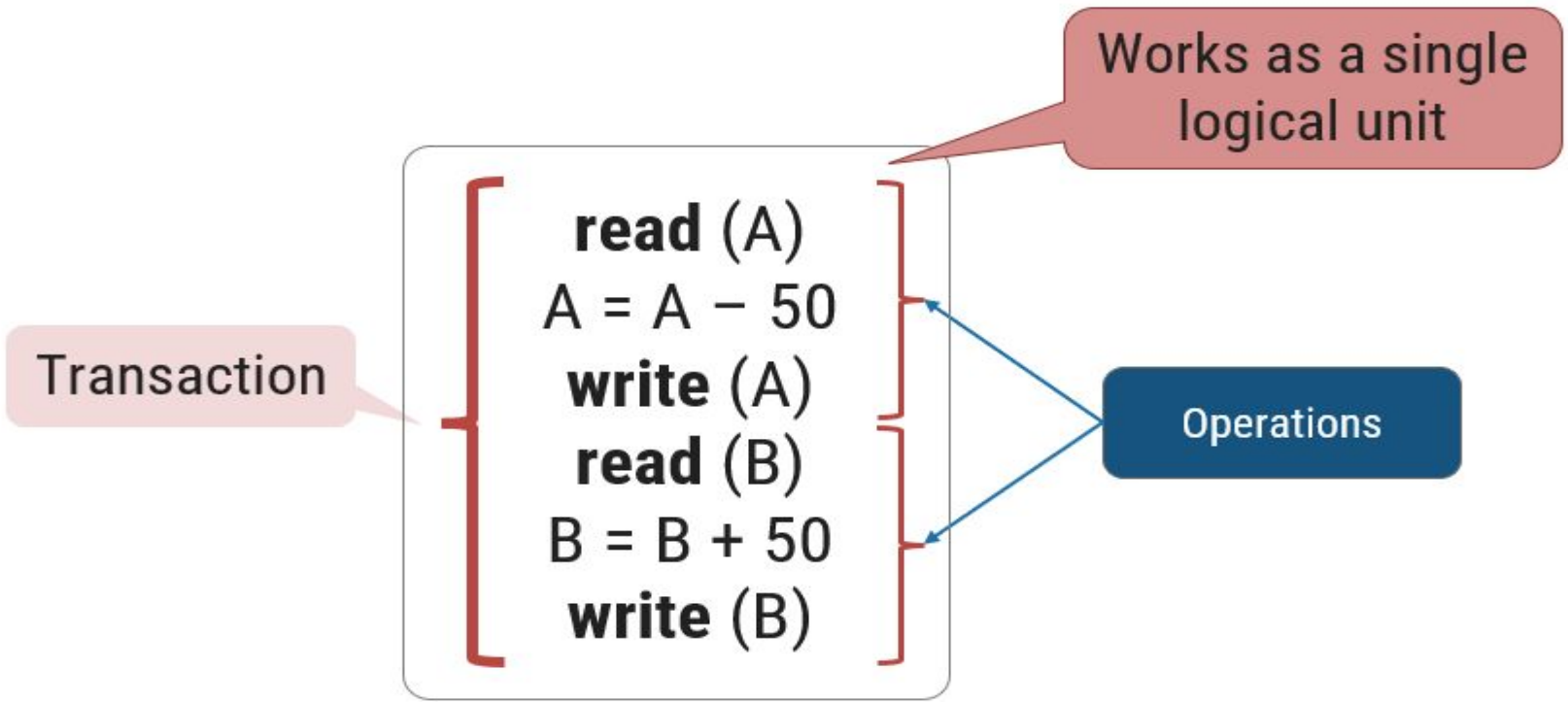
Index

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction

- A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - ★ Failures of various kinds, such as hardware failures and system crashes
  - ★ Concurrent execution of multiple transactions

- A transaction is a **sequence of operations performed as a single logical unit of work**.
- A transaction is a **logical unit of work that contains one or more SQL statements**.
- Example of transaction: Want to transfer Rs. 50 from Account-A to Account-B



ACID Properties

To preserve integrity of data, the database system must ensure:

- **Atomicity**. Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation**. Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - ★ That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

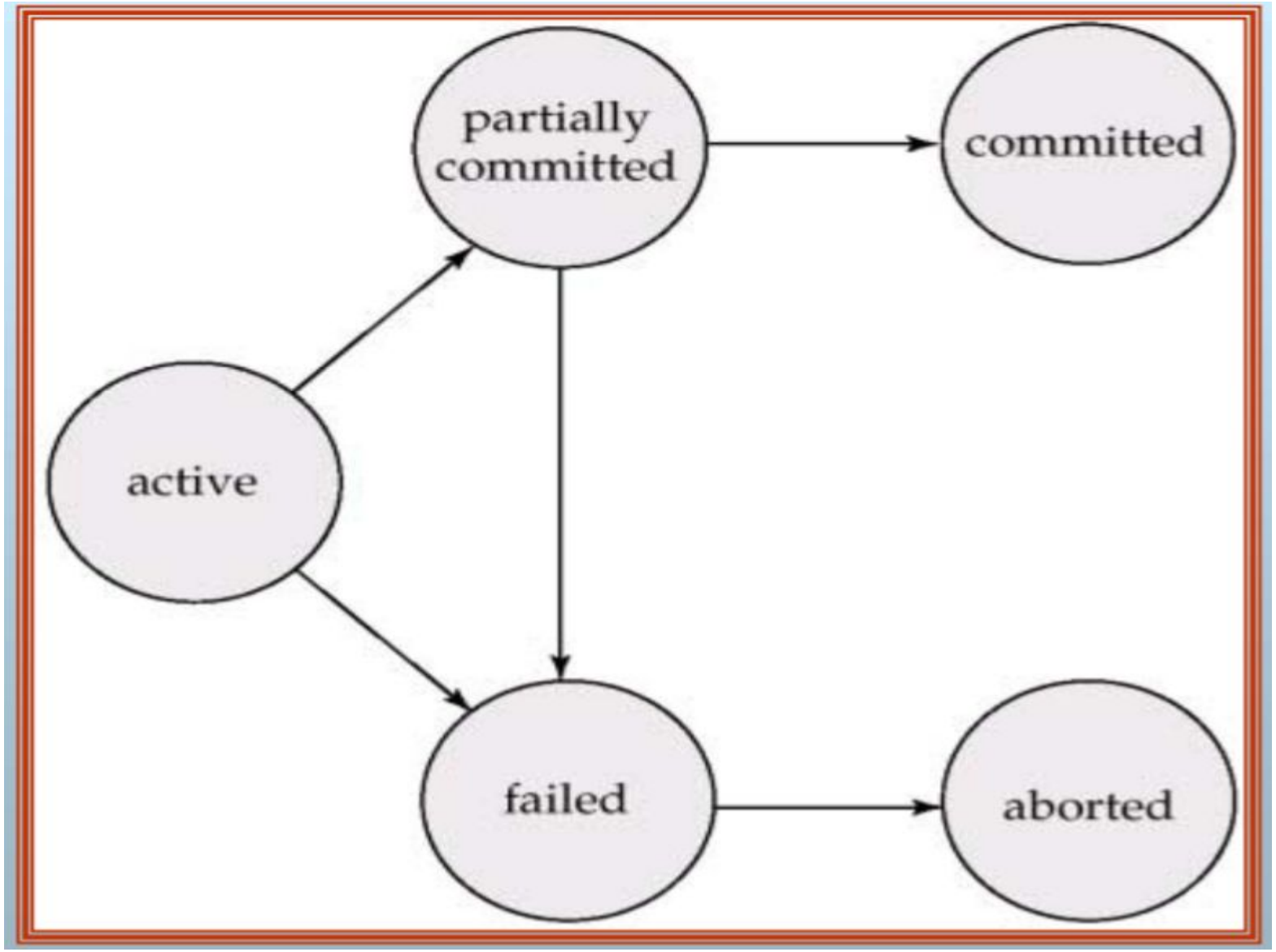
Fund Transfer Example

- Transaction to transfer $50 from account *A* to account *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.
- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).
  Can be ensured trivially by running transactions ***serially***, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Transaction State

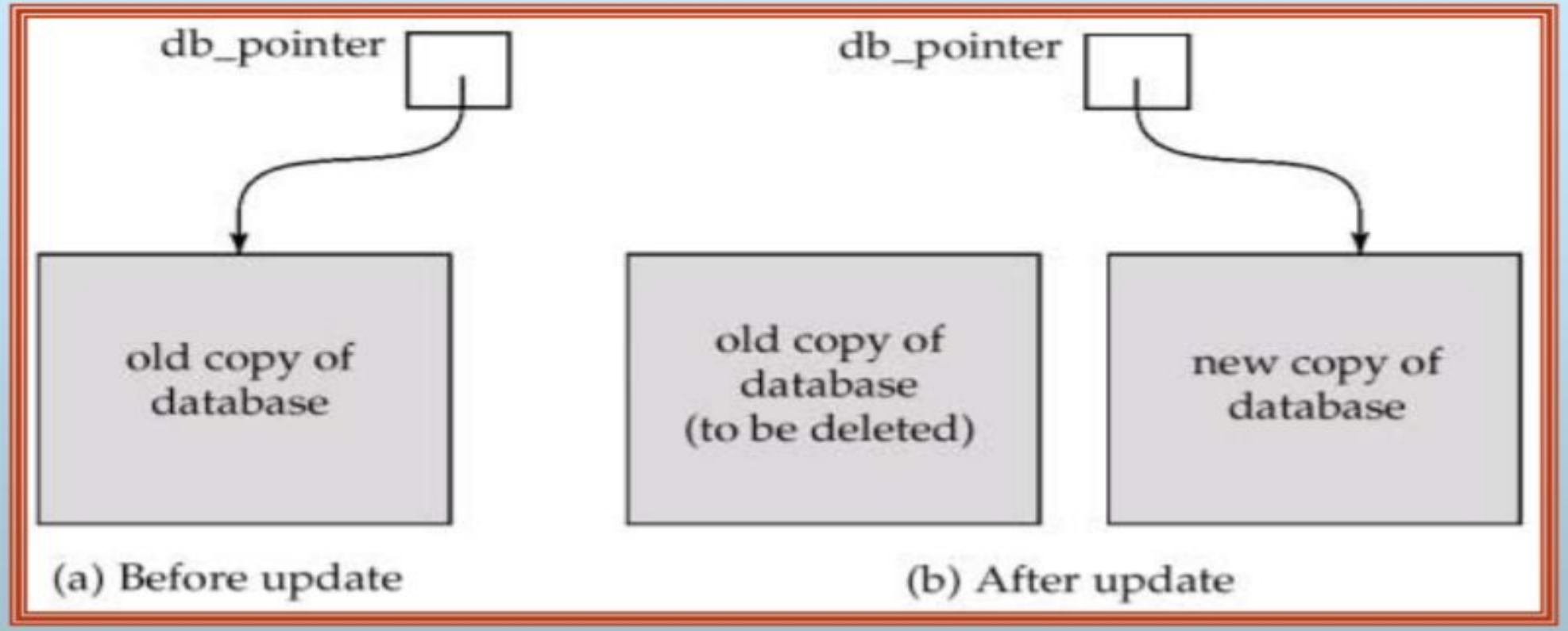
- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction – only if no internal logical error
  - kill the transaction
- **Committed,** after *successful completion*.


partially committed
committed
active
failed
aborted

Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.
- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

The shadow-database scheme:



- Assumes disks to not fail
- Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the *entire* database. Will see better schemes in Chapter 17

Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

Example

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which T_1 is followed by T_2.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

T2 followed by T1

- Let T_1 and T_2 be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In both Schedule 1 and 3, the sum A + B is preserved.

- The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum $A + B$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- A schedule is serializable if it is **equivalent to a serial schedule**.
- In **serial schedules**, only **one transaction is allowed to execute at a time** i.e. **no concurrency is allowed**.
- Whereas in **serializable schedules**, **multiple transactions can execute simultaneously** i.e. **concurrency is allowed**.
- Types (forms) of serializability
  - Conflict serializability
  - View serializability

Conflicting Instruction

- Let I_i and I_j be two instructions of transactions T_i and T_j respectively.

1. I_i = read(Q), I_j = read(Q)
   I_i and I_j don't conflict

2. I_i = read(Q), I_j = write(Q)
   I_i and I_j conflict

3. I_i = write(Q), I_j = read(Q)
   I_i and I_j conflict

4. I_i = write(Q), I_j = write(Q)
   I_i and I_j conflict

T_i	T_j
read (Q)	
	read (Q)

T_i	T_j
	read (Q)
read (Q)	

T_i	T_j
read (Q)	
	write(Q)

T_i	T_j
	write(Q)
read (Q)	

T_i	T_j
write(Q)	
	read (Q)

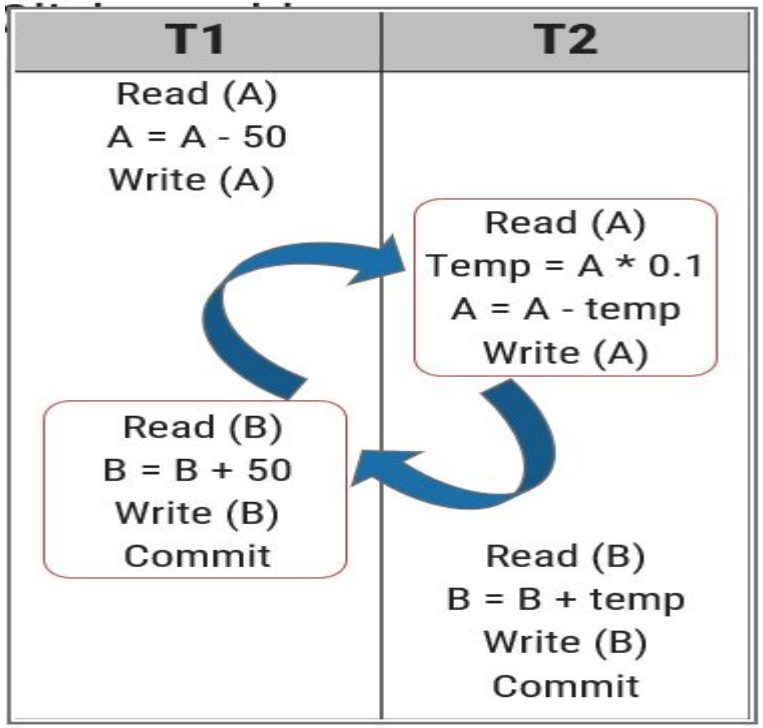
T_i	T_j
	read (Q)
write(Q)	

T_i	T_j
write(Q)	
	write(Q)

T_i	T_j
	write(Q)
write(Q)	

Conflict serializable

- If a given schedule can be **converted into a serial schedule by swapping its non-conflicting operations**, then it is called as a conflict serializable schedule.

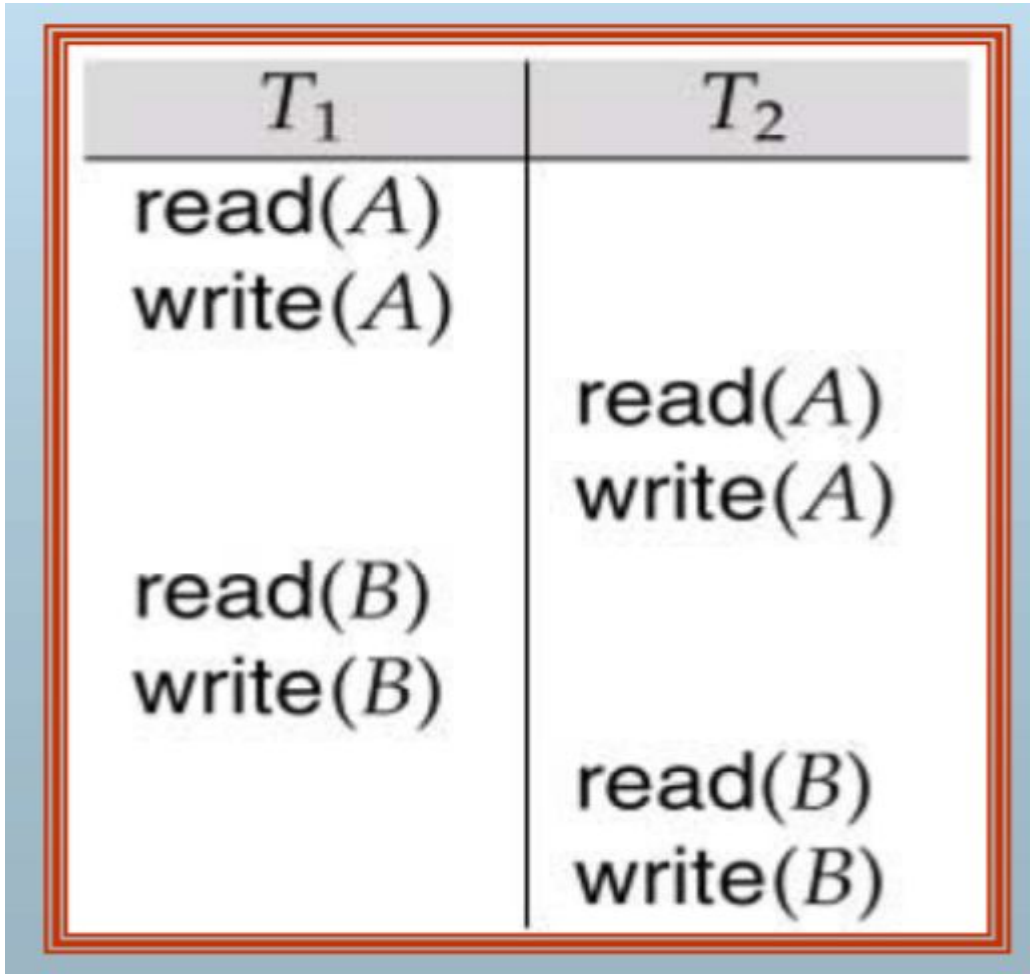


T1	T2
Read (A) A = A - 50 Write (A) Read (B) B = B + 50 Write (B) Commit	Read (A) Temp = A * 0.1 A = A - temp Write (A) Read (B) B = B + temp Write (B) Commit

Check Conflict serializable or not?

T1	T2
Read (A)	Write (A)
Read (A)	

Check Conflict serializable or not?



T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

View Serializability

- Let S1 and S2 be two schedules with the same set of transactions. S1 and S2 are view equivalent if the following three conditions are satisfied, for each data item Q
  - Initial Read
  - Updated Read
  - Final Write
- If a schedule is view equivalent to its serial schedule then the given schedule is said to be view serializable.

- A schedule *S* is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Schedule 9 (from text) — a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Every view serializable schedule that is not conflict serializable has **blind writes.**

Initial Read

- If in **schedule S1, transaction Ti reads the initial value of Q**, then in **schedule S2 also transaction Ti must read the initial value of Q**.

S1	
T1	T2
Read (A)	
	Write (A)

S3	
T1	T2
	Read (A)
Write (A)	

S2	
T1	T2
	Write (A)
Read (A)	

- Above two schedules **S1 and S3 are not view equivalent** because **initial read operation in S1 is done by T1 and in S3 it is done by T2**.
- Above two schedules **S1 and S2 are view equivalent** because **initial read operation in S1 is done by T1 and in S2 it is also done by T1**.

Updated Read

- If in **schedule S1 transaction Ti executes read(Q), and that value was produced by transaction Tj** (if any), then in **schedule S2 also transaction Ti must read the value of Q that was produced by transaction Tj**.

S1		
T1	T2	T3
Write (A)		
	Write (A)	
		Read (A)

S3		
T1	T2	T3
	Write (A)	
Write (A)		
		Read (A)

S2		
T1	T2	T3
	Write (A)	
Write (A)		
		Read (A)

- Above two schedules **S1 and S3 are not view equal** because, **in S1, T3 is reading A that is updated by T2 and in S3, T3 is reading A which is updated by T1**.
- Above two schedules **S1 and S2 are view equal** because, **in S1, T3 is reading A that is updated by T2 and in S2 also, T3 is reading A which is updated by T2**.

Final Write

- If **Ti performs the final write on the data value in S1, then it also performs the final write on the data value in S2**.

S1		
T1	**T2**	**T3**
Write (A)		
	Read (A)	
		Write (A)

S3		
T1	**T2**	**T3**
	Write (A)	
Write (A)		
		Read (A)

S2		
T1	**T2**	**T3**
	Read (A)	
Write (A)		
		Write (A)

- Above two schedules **S1 and S3 are not view equal** because **final write operation in S1 is done by T3 and in S3 final write operation is also done by T1**.
- Above two schedules **S1 and S2 are view equal** because **final write operation in S1 is done by T3 and in S2 also the final write operation is also done by T3**.

View serializability example (Initial Update)

Non-Serial Schedule (S1)	
T1	**T2**
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

Serial Schedule (S2)	
T1	**T2**
Read (A)	
Write (A)	
Read (B)	
Write (B)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)

- In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.
- In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.
- The initial read condition is satisfied for both the schedules.

Update Read

Non-Serial Schedule (S1)	
T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

Serial Schedule (S2)	
T1	T2
Read (A)	
Write (A)	
Read (B)	
Write (B)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)

- In schedule S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.
- In schedule S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.
- The updated read condition is also satisfied for both the schedules.

Final write

Non-Serial Schedule (S1)	
T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

Serial Schedule (S2)	
T1	T2
Read (A)	
Write (A)	
Read (B)	
Write (B)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)

- In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.
- In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.
- The final write condition is also satisfied for both the schedules.

View serializable example

Non-Serial Schedule (S1)	
T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
Read (B)	
Write (B)	
	Read (B)
	Write (B)

Serial Schedule (S2)	
T1	T2
Read (A)	
Write (A)	
Read (B)	
Write (B)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)

- Since **all the three conditions** that checks whether the two schedules are view equivalent **are satisfied** in this example, which means **S1 and S2 are view equivalent**.
- Also, as we know that the **schedule S2 is the serial schedule of S1**, thus we can say that the **schedule S1 is view serializable schedule**.

Recoverability

- **Recoverable schedule** : It means one where for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti the commit operation of Ti appears before the commit operation of Tj.

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- **Cascadeless schedule :** It means one where for each pair of tranactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Lock Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

- Example of a transaction performing locking:

T_2: **lock-S**(*A*);
read (*A*);
unlock(*A*);
lock-S(*B*);
read (*B*);
unlock(*B*);
display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfall of lock based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - ★ To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - ★ A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - ★ The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

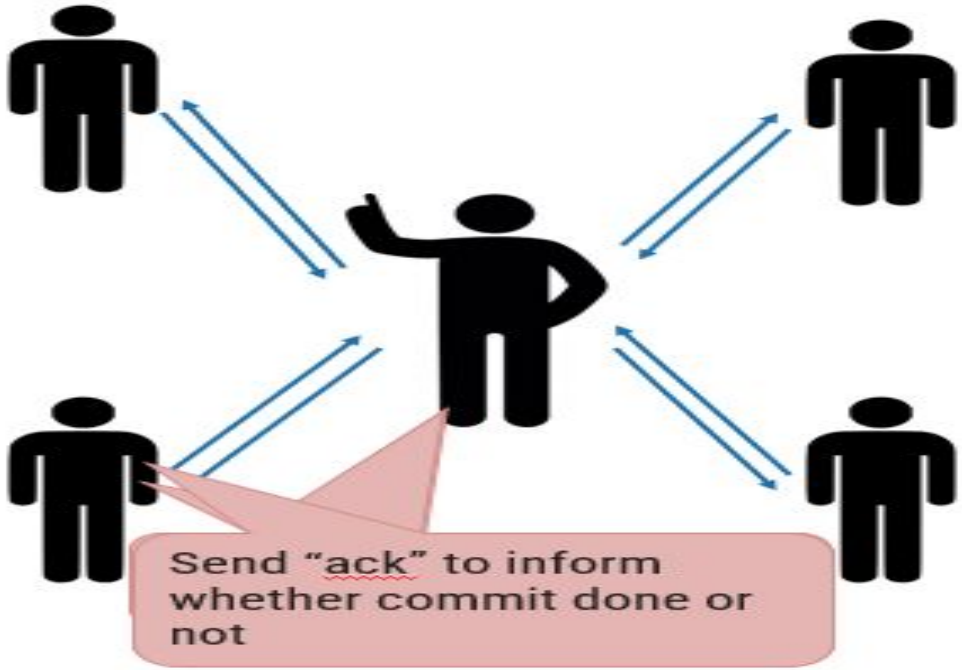
Two phase locking protocol

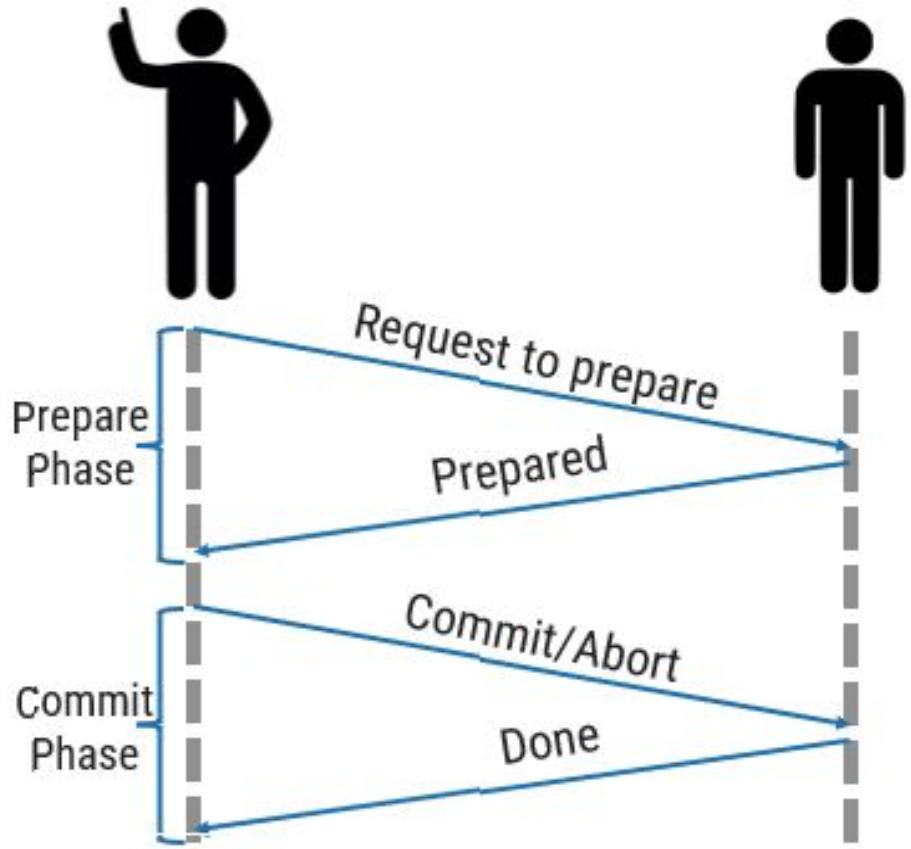
- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - ★ transaction may obtain locks
  - ★ transaction may not release locks
- Phase 2: Shrinking Phase
  - ★ transaction may release locks
  - ★ transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Two phase commit protocols

- Two phase commit protocol **ensures that all participants perform the same action (either to commit or to rollback a transaction)**.
- It is designed to **ensure that either all the databases are updated or none** of them, so that the databases remain synchronized.
- In two phase commit protocol there is one node which is act as a coordinator or controlling site and all other participating node are known as cohorts or participant or slave.
- **Coordinator** (controlling site) – the component that coordinates with all the participants.
- **Cohorts** (Participants/Slaves) – each individual node except coordinator are participant.
- As the name suggests, the two phase commit protocol involves two phases.
  - Commit request phase OR Prepare phase
  - Commit/Abort phase


Send “ack” to inform whether commit done or not


Request to prepare
Prepare Phase
Prepared
Commit/Abort
Commit Phase
Done

Two phase commit protocol Commit Request Phase (Obtaining Decision)

- Commit Request Phase (Obtaining Decision)
  - **After each slave has locally completed its transaction**, it **sends a "DONE"** message to the controlling site.
  - When the **controlling site has received "DONE" message from all slaves**, it **sends a "Prepare"** (prepare to commit) message to the slaves.
  - The **slaves vote** on whether they **still want to commit or not**.
  - If a **slave wants to commit**, it **sends a "Ready" message**.
  - A **slave that does not want to commit sends a "Not Ready" message**.
  - This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Two phase commit protocol Commit Phase (Performing Decision)

- Commit Phase (Performing Decision)
  - After the controlling site has **received "Ready" message from all the slaves**:
  - The **controlling site sends a "Global Commit"** message to the slaves.
  - The **slaves commit** the transaction and **send a "Commit ACK"** message to the controlling site.
  - When the **controlling site receives "Commit ACK"** message from all the slaves, it **considers the transaction as committed**.
- Commit Phase (Performing Decision)
  - After the controlling site **has received the first "Not Ready" message from any slave**:
  - The **controlling site sends a "Global Abort"** message to the slaves.
  - The **slaves abort** the transaction and **send a "Abort ACK"** message to the controlling site.
  - When the **controlling site receives "Abort ACK"** message from all the slaves, it **considers the transaction as aborted**.

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Rule

- Only Lock-ex
- First lock can be acquired on any data item
- Subsequent locks are allowed only if the parent is locked.
- Unlock at any point
- Each data item can be accessed at most once
- Relocking by same transaction is not allowed

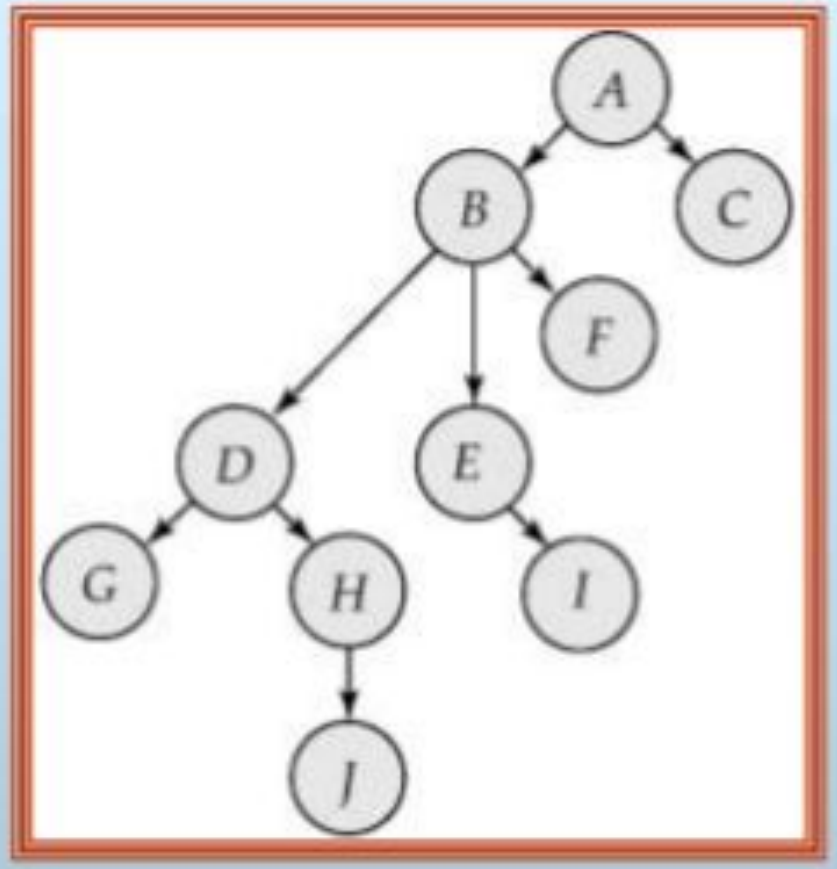
Advantages

- Deadlock free
- No rollback
- Shorter waiting time

Disadvantages

- Prior knowledge of data access
- Unnecessary locks

Tree Protocol



T1	T2	T3	T4
Lock-x(B)			
	Lock-x(D)		
	Lock-x(H)		
	Unlock(D)		
Lock-x(E)			
Lock-x(D)			
Unlock(B)			
Unlock(E)			
		Lock-x(B)	
		Lock-x(E)	
	Unlock(H)		
Lock-x(G)			
Unlock(D)			
			Lock-x(D)
			Lock-x(H)
			Unlock(D)
			Unlock(H)
		Unlock(E)	

Time Stamp based Protocol

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction T_i issues a **read**(Q)

1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten. Hence, the **read** operation is rejected, and T_i is rolled back.
2. If TS(T_i)≥ **W**-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

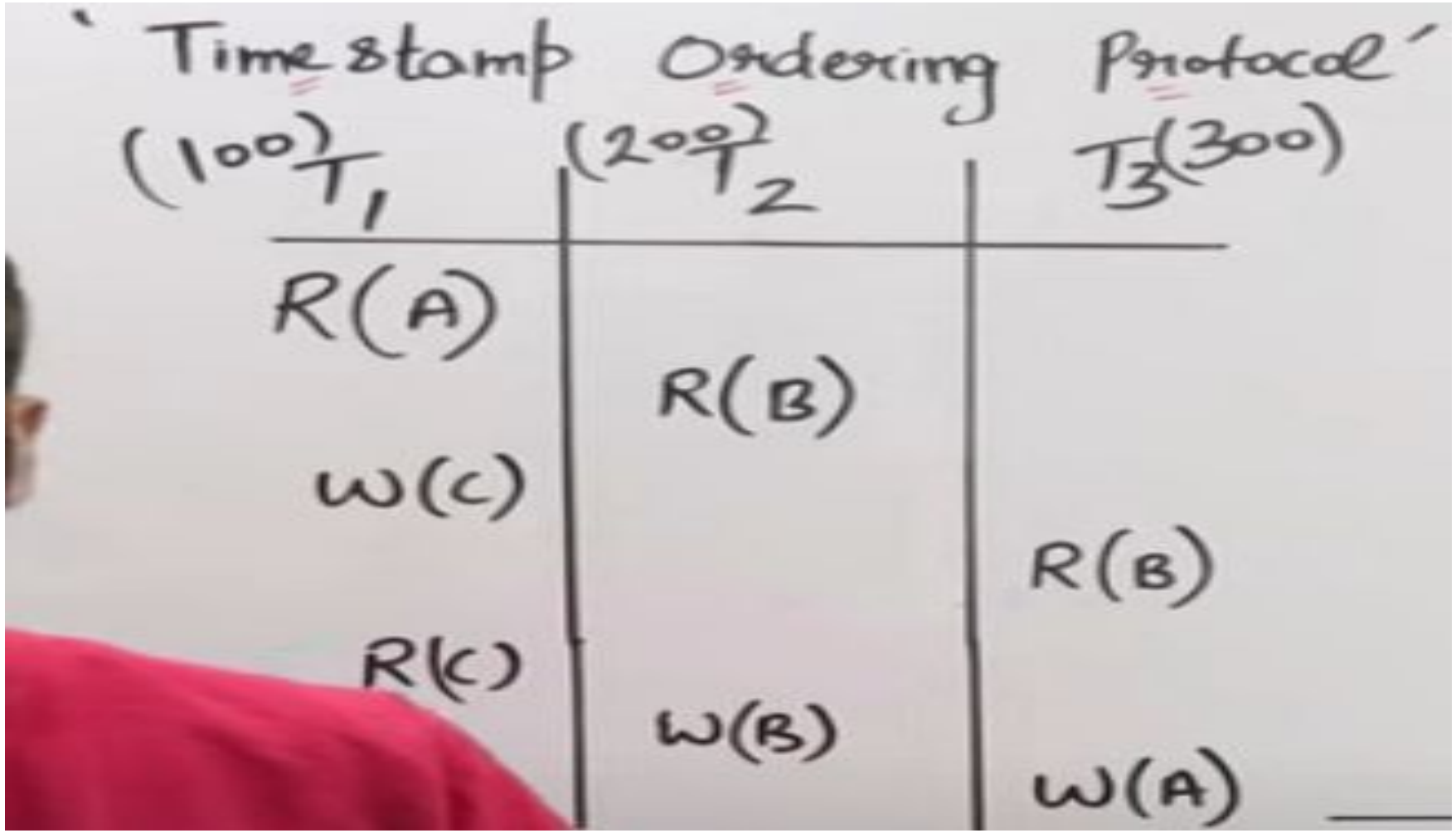
- Suppose that transaction T_i issues **write**(Q).
- If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and T_i is rolled back.
- If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected, and T_i is rolled back.
- Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

* Rules:

1) Transaction T_i issues a Read(A) operation
   a) if $WTS(A) > TS(T_i)$, Rollback T_i
   b) Otherwise execute R(A) operation
      Set $RTS(A) = Max\{RTS(A), TS(T_i)\}$
      200, 100, 300

2) Transaction T_i issues Write(A) operation
   a) if $RTS(A) > TS(T_i)$ then Rollback T_i
   b) if $WTS(A) > TS(T_i)$ then Rollback T_i
   c) otherwise execute Write(A) operation
      Set $WTS(A) = TS(T_i)$

Example


Timestamp Ordering Protocol
(100) T1
(200) T2
T3 (300)
R(A)
R(B)
W(C)
R(B)
R(C)
W(B)
W(A)

Database recovery

- There are many situations in which a transaction may not reach a commit or abort point.
  - Operating system crash
  - DBMS crash
  - System might lose power (power failure)
  - Disk may fail or other hardware may fail (disk/hardware failure)
  - Human error
- In any of above situations, data in the database may become inconsistent or lost.
- For example, if a transaction has completed 30 out of 40 write instructions to the database when the DBMS crashes, then the database may be in an inconsistent state as only part of the transaction's work was completed.
- Database recovery is the **process of restoring the database and the data to a consistent state**.
- This may include **restoring lost data up to the point of the event** (e.g. system crash).

Log based recovery method

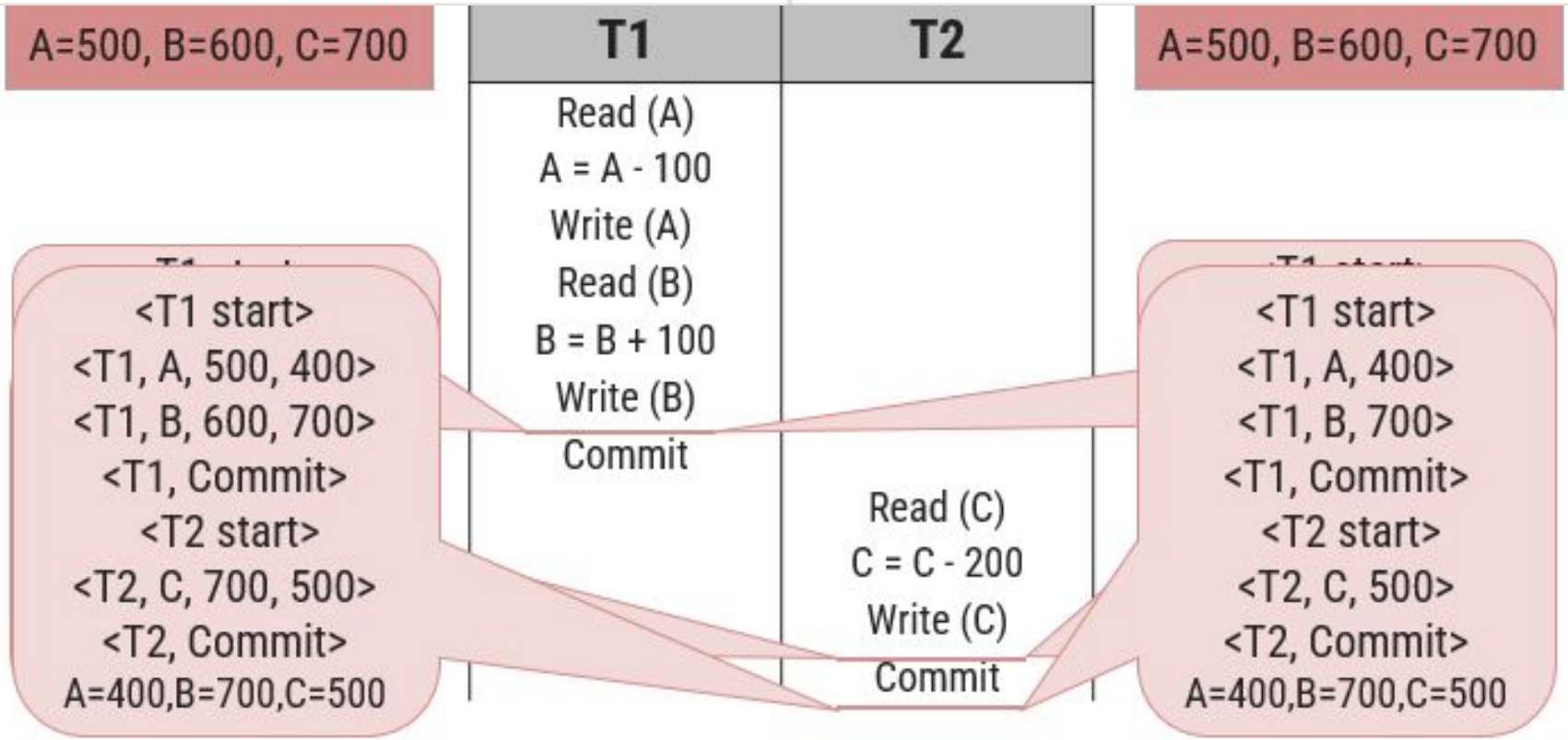
- The log is a **sequence of log records, which maintains information about update activities on the database**.
- A log is **kept on stable storage (i.e HDD)**.
- Log contains
  - Start of transaction
  - Transaction-id
  - Record-id
  - Type of operation (insert, update, delete)
  - Old value, new value
  - End of transaction that is committed or aborted.

Log based recovery method

- When transaction **Ti starts**, it registers itself by writing a record **<Ti start>** to the log.
- Before **Ti executes write(X)**, a log record **<Ti, X, V1, V2>** is written, where V1 is the value of X before the write (the old value), and V2 is the value to be written to X (the new value).
- When **Ti finishes it last statement**, the log record **<Ti commit>** is written.
- **Undo** of a log record **<Ti, X, V1, V2>** writes the old value V1 to X
- **Redo** of a log record **<Ti, X, V1, V2>** writes the new value V2 to X
- Types of log based recovery method
  - Immediate database modification
  - Deferred database modification

Immediate v/s Deferred database modification

Immediate database modification	Deferred database modification
Updates (changes) to the database are **applied immediately** as they occur without waiting to reach to the commit point.	**Updates (changes)** to the database are **deferred (postponed)** until the transaction commits.

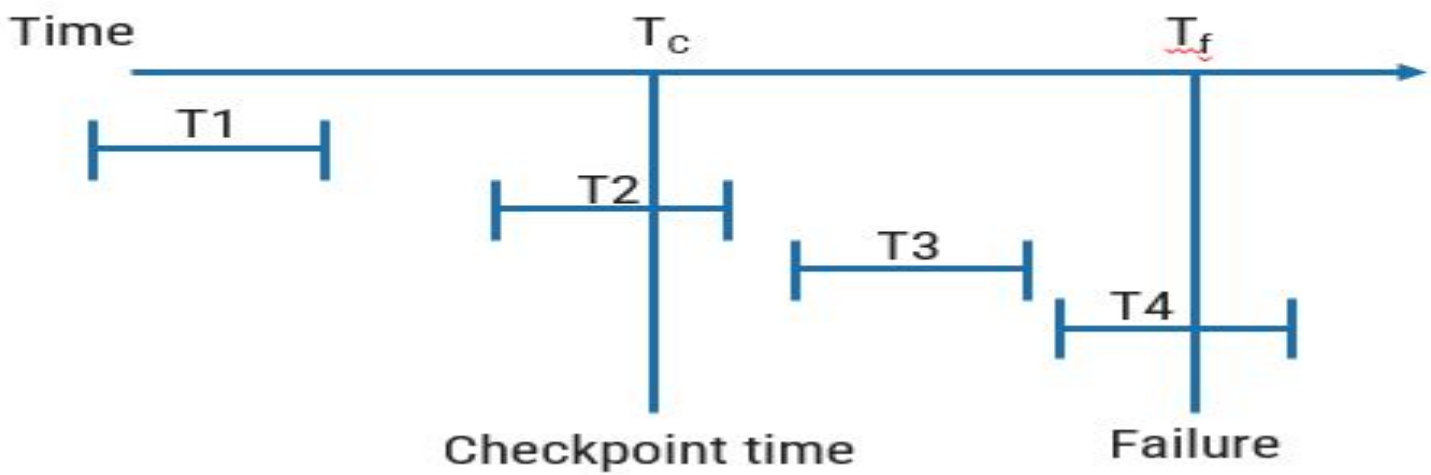


Immediate database modification	Deferred database modification
Updates (changes) to the database are **applied immediately** as they occur without waiting to reach to the commit point.	**Updates (changes)** to the database are **deferred (postponed)** until the transaction commits.
If **transaction is not committed**, then we **need to do undo** operation and **restart the transaction** again.	If **transaction is not committed**, then **no need to do any undo** operations. Just **restart the transaction**.
If **transaction is committed**, then **no need to do redo** the updates of the transaction.	If **transaction is committed**, then we **need to do redo** the updates of the transaction.
Undo and Redo both operations are performed.	**Only Redo operation is performed**.

Problems with Deferred & Immediate Updates (Checkpoint)

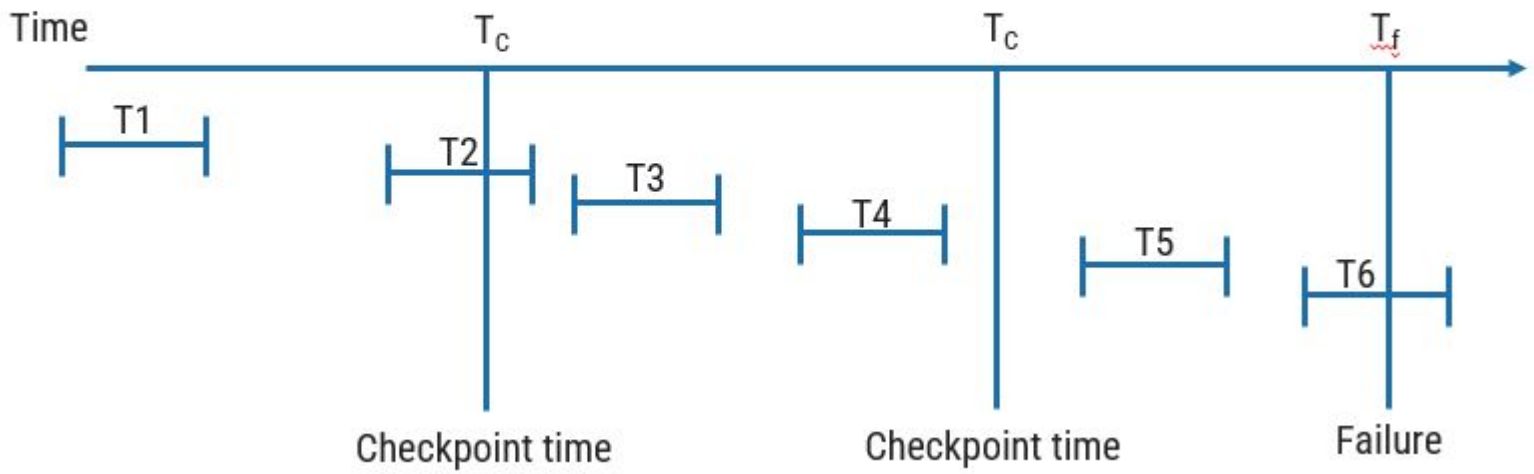
- Searching the entire log is time consuming.
  - Immediate database modification
    - When transaction fail log file is used to undo the updates of transaction.
  - Deferred database modification
    - When transaction commits log file is used to redo the updates of transaction.
- **To reduce the searching time** of entire log we can use **check point**.
- It is a **point** which specifies that **any operations executed before it are done correctly and stored safely** (updated safely in database).
- At this point, all the **buffers are force-fully written to the secondary storage** (database).
- Checkpoints are scheduled at predetermined time intervals.
- It is used to limit:
  - Size of transaction log file
  - Amount of searching

How the checkpoint works when failure occurs



- At failure time:
  - **Ignore the transaction T1** as it has already been committed before checkpoint.
  - **Redo transaction T2 and T3** as they are active after checkpoint and are committed before failure.
  - **Undo transaction T4** as it is active after checkpoint and has not committed.

How the checkpoint works when failure occurs



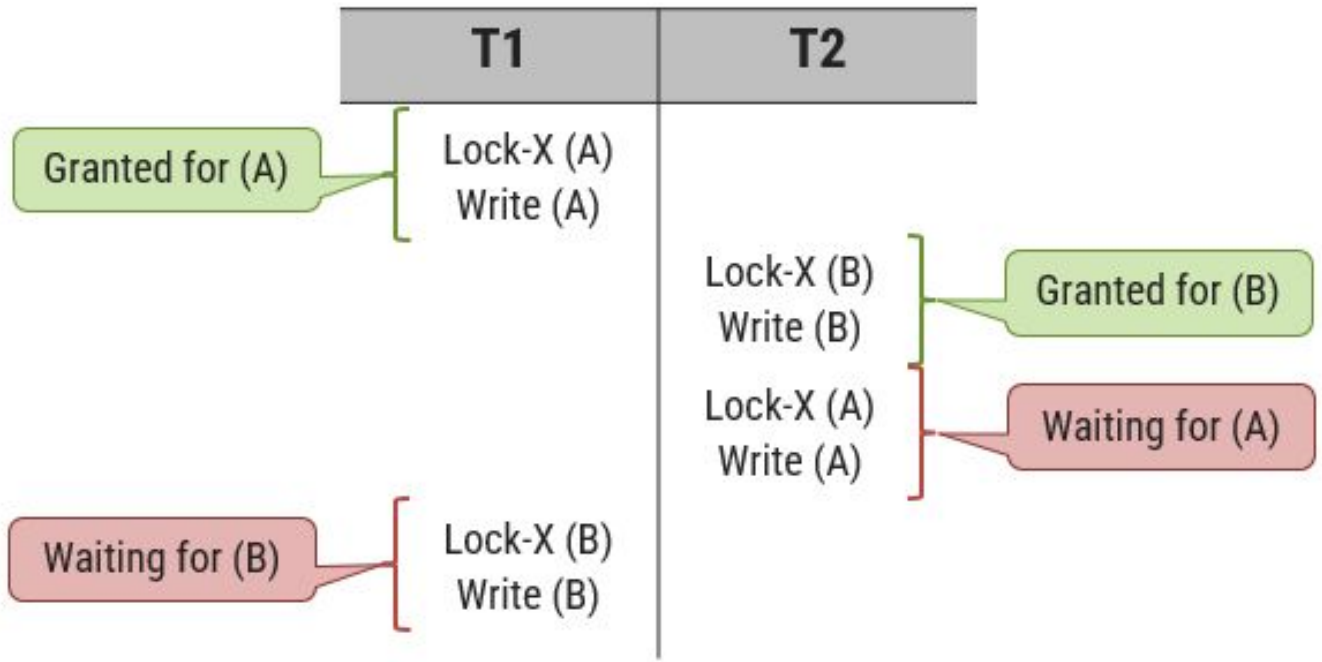
Exercise Give the name of transactions which has already been committed before checkpoint.

Exercise Give the name of transactions which will perform Redo operation.

Exercise Give the name of transactions which will perform Undo operation.

Deadlock

- Consider the following two transactions:



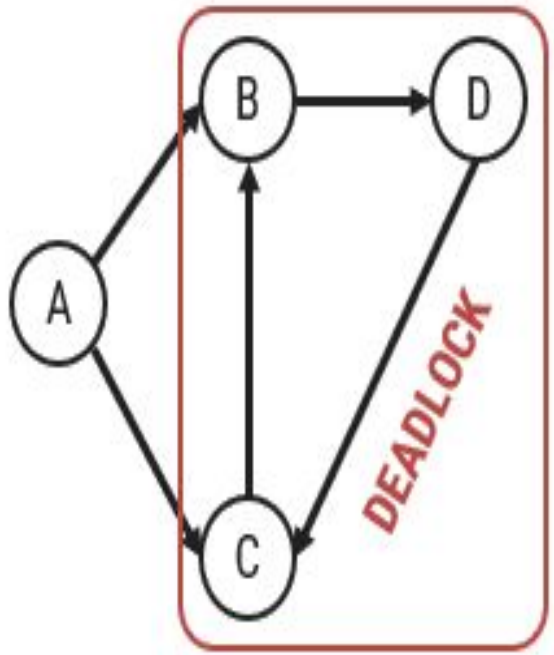
- A deadlock is a **situation in which two or more transactions are waiting for one another to give up locks**.

Deadlock detection

- A simple way to detect deadlock is with the help of **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 from Ti to Tj (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**.
- Then **each transaction involved in the cycle is said to be deadlocked**.

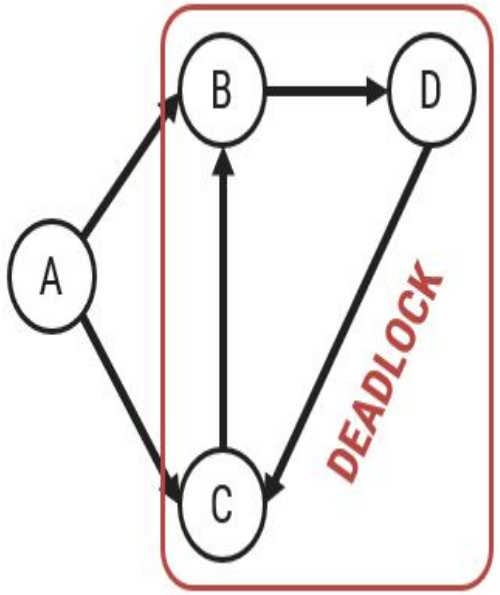
Deadlock detection

- Transaction **A is waiting for** transactions **B and C**.
- Transactions **C is waiting** for transaction **B**.
- Transaction **B is waiting** for transaction **D**.
- This wait-for graph has **no cycle**, so there is **no deadlock state**.
- Suppose now that transaction **D is requesting an item held by C**. Then the **edge D → C is added to the wait-for graph**.
- Now this **graph contains the cycle**.
- B → D → C → B
- It means that **transactions B, D and C are all deadlocked**.



Deadlock recovery

- When a deadlock is detected, the system must recover from the deadlock.
- The most common **solution is to roll back one or more transactions to break the deadlock**.
- Choosing which transaction to abort is known as **victim selection**.
- In this wait-for graph transactions B, D and C are deadlocked.
- In order to remove deadlock one of the transaction out of these three (B, D, C) transactions must be roll backed.
- We should **rollback those transactions that will incur the minimum cost**.
- When a deadlock is detected, the choice of which transaction to abort can be made using following criteria:
  - The transaction which have the fewest locks
  - The transaction that has done the least work
  - The transaction that is farthest from completion



Deadlock prevention

- A protocols **ensure that the system will never enter into a deadlock state**.
- Some prevention strategies :
  - Require that **each transaction locks all its data items before it begins execution** (pre-declaration).
  - Impose **partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial**.

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

1. Wait-die scheme – non-preemptive
   - If an **older transaction is requesting a resource** which is held by younger transaction, then **older transaction is allowed to wait** for it till it is available.
   - If an **younger transaction is requesting a resource** which is held by older transaction, then **younger transaction is killed**.
2. Wound-wait scheme – preemptive
   - If an **older transaction is requesting a resource** which is held by younger transaction, then **older transaction forces younger transaction to kill** the transaction and release the resource.
   - If an **younger transaction is requesting a resource** which is held by older transaction, then **younger transaction is allowed to wait** till older transaction will releases it.

	Wait-die	Wound-wait
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.

3. Timeout-Based Schemes
   - A **transaction waits for a lock only for a specified amount of time**. **After that, the wait times out and the transaction is rolled back**. So deadlocks never occur.
   - **Simple to implement; but difficult to determine good value of the timeout interval**.

Asked Question in GTU

1. Write a note on two phase locking protocol.
2. Explain ACID properties of transaction with suitable example.
3. What is log based recovery? Explain immediate database modification technique for databa recovery. OR Define Failure. Write a note on log based recovery.
4. State differences between conflict serializability and view serializability.
5. Explain two-phase commit protocol.
6. Define transaction. Explain various states of transaction with suitable diagram.
7. Write differences between shared lock and exclusive lock.
8. Explain deadlock with suitable example.
9. What is locking? Define each types of locking.
10. Define wait-Die & wound-wait.