DBMS:Database Management System

Transaction

Marwadi University

Department of ICT

Unit no : 5
Transaction

Prof. Harikesh Chauhan

- Transaction concepts
- ACID
- Transaction Life Cycle
- Serializability
- Recovery
- Concurrency Control
- Locking mechanism
- Deadlock

Marwadi University

Department of ICT

Unit no : 5
Transaction

Transaction

- Anything under execution can be known as Transaction.
- A transaction is a sequence of operations performed as a single logical unit of work that contains one or more than one SQL statements.

1. Read(A);
2. A := A – 50;
3. Write(A);
4. Read(B);
5. B := B + 50;
6. Write(B);

Operations

Work as a single Unit

Transactions

Transaction (ACID Properties)

- Atomicity **(Either transaction is Complete or No Transaction)**
- Consistency **(consistent data must be available whenever the database is accessed after any transaction)**
- Isolation **(Transactions details of one transaction must be hidden from other transactions)**
- Durability **(All changes made by the transaction must remain permanent through out the database)**

Transaction (ACID Properties)

Atomicity

- This property states that, in a transaction either all of its operations are executed or none.
- Either transaction execute 0% or 100%.
- For example, consider a transaction to transfer Rs. 1000 from account A to account B.
- So, if Rs. 1000 is deducted from account A, then it must be added to account B else in case of error, it must be added back to Account A.

Transaction (ACID Properties)

Consistency

- Consistent data/values must be available after any transaction.
- If the database was in a consistent state before the execution of a transaction; then it must remain consistent after the execution of the transaction is completed.
- If the amount has been debited from account A and has been added to account B, then both the accounts must show the amount after the transaction has been performed.
- For e.g. Initial amount in account A = 5000 and account B = 3000. Amount to be transferred from A to B = 1000.
- Thus, after the transaction is completed, amount in Account A should be 5000 and in account B should be 5000.

Transaction (ACID Properties)

Isolation

- Transactions details of one transaction must be hidden from other transactions until the transaction is committed.
- Intermediate transaction results must be hidden from other concurrently executed transactions.
- For e.g. if account A start transferring amount to account B, then no other transaction should be allowed to access account A until the current transaction is fully completed.

Transaction (ACID Properties)

Durability

- All changes made by the transaction must remain permanent through out the database.
- After a transaction completes successfully, the changes to the database must be permanent, even if system fails.
- For e.g. Initial amount in account A = 5000 and account B = 3000. Amount to be transferred from A to B = 1000.
- Thus, after the transaction is completed, amount in Account A should be 5000 and in account B should be 5000.
- This change should remain permanent until some other transaction changes any account's amount.

Transaction (State Diagram or Life Cycle)

Active

- ✔ This is the initial state.
- ✔ The transaction remains in this state while execution.

Partial Committed

- ✔ It is a state where a transaction actually enter in its last operation.

Failed

- ✔ An execution cannot be performed due to errors.
- ✔ Once a transaction cannot be completed, any changes that it made must be undone rolling it back.

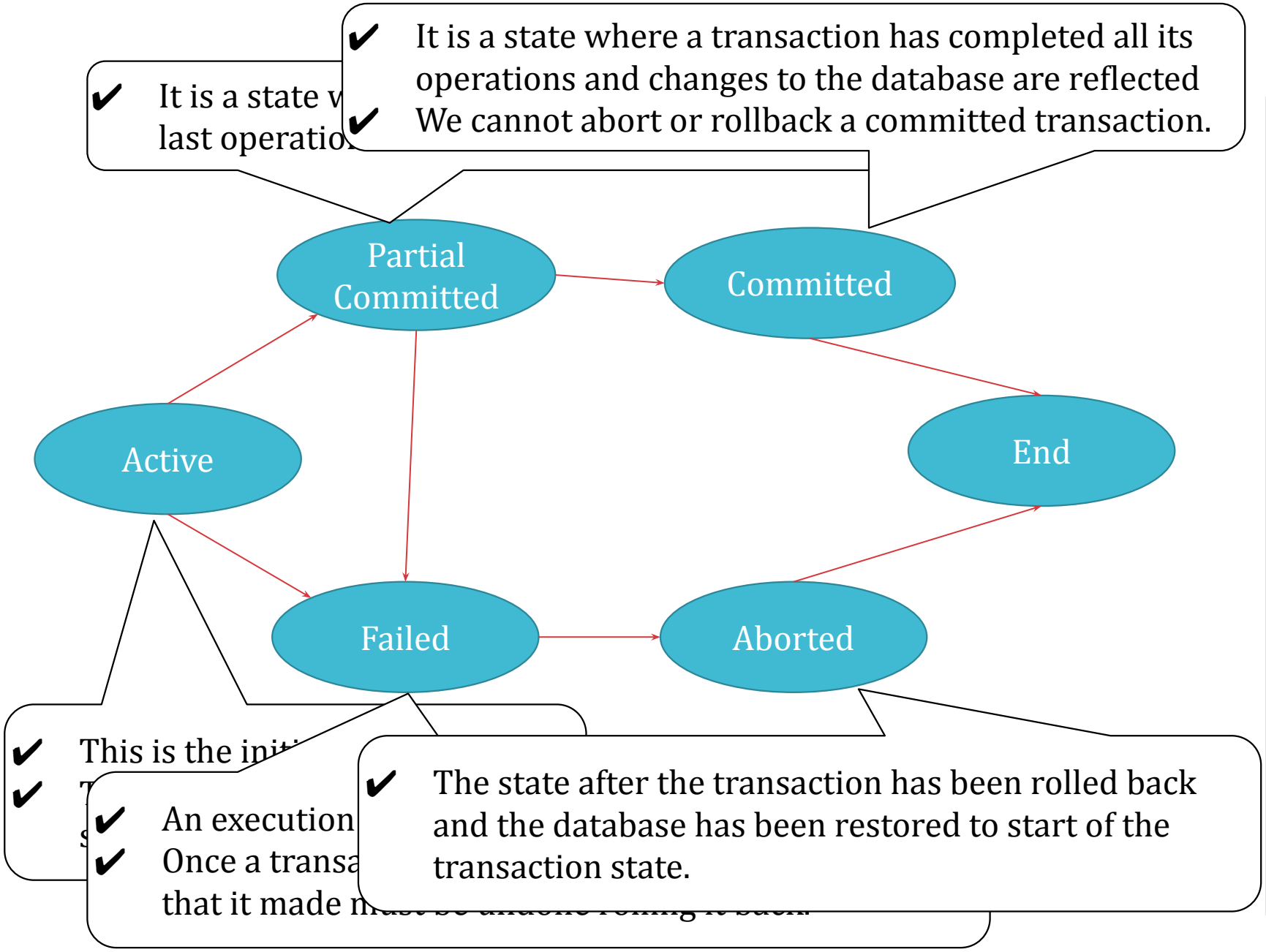
Committed

- ✔ It is a state where a transaction has completed all its operations and changes to the database are reflected
- ✔ We cannot abort or rollback a committed transaction.

Aborted

- ✔ The state after the transaction has been rolled back and the database has been restored to start of the transaction state.

Transaction (State Diagram or Life Cycle)



Transaction (Operations)

1. Read (X)

- This operation is mainly used for accessing the data from the database.
- It transfers the data item accessed from the database to the local buffer/memory.

2. Write (X)

- This operation is mainly used for updating/modifying the data in the database.
- It updates the data of the database as a result of any transaction been executed.

Transaction (Schedule)

- A schedule is a process of grouping and ordering the operation of transactions.
- A schedule is a sequence in which all the operations are executed.
- A schedule is required in a database because when some transactions execute in parallel, they may affect the result of the transaction.
- If a transaction is updating the data in the database and other transaction at the same time is accessing the same data, then the result of the database is fully dependent on the sequence of these transactions.

Transaction (Schedule 1-Example)

T1	T2
Read (A) A = A - 1000 Write (A) Read (B) B = B + 1000 Write (B) Commit	
	Read (A) T = A * 0.7 A = A - T Write (A) Read (B) B = B + T Write (B) Commit

Transaction (Schedule2 - Example)

Transaction (Serial Schedule)

- A serial schedule is one in which no transaction starts until an active transaction is complete or commit.
- Transactions are executed one by one in a sequence.
- This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

Transaction (Serial Schedule Example)

Transaction (Interleaved Schedule)

- Schedule that interferes during the execution of different transactions.
- It means that any other transaction starts before the active transaction commits.
- Execution can switch between the transactions in any of the schedules.

Transaction (Interleaved Schedule Example)

Transaction (Serializability)

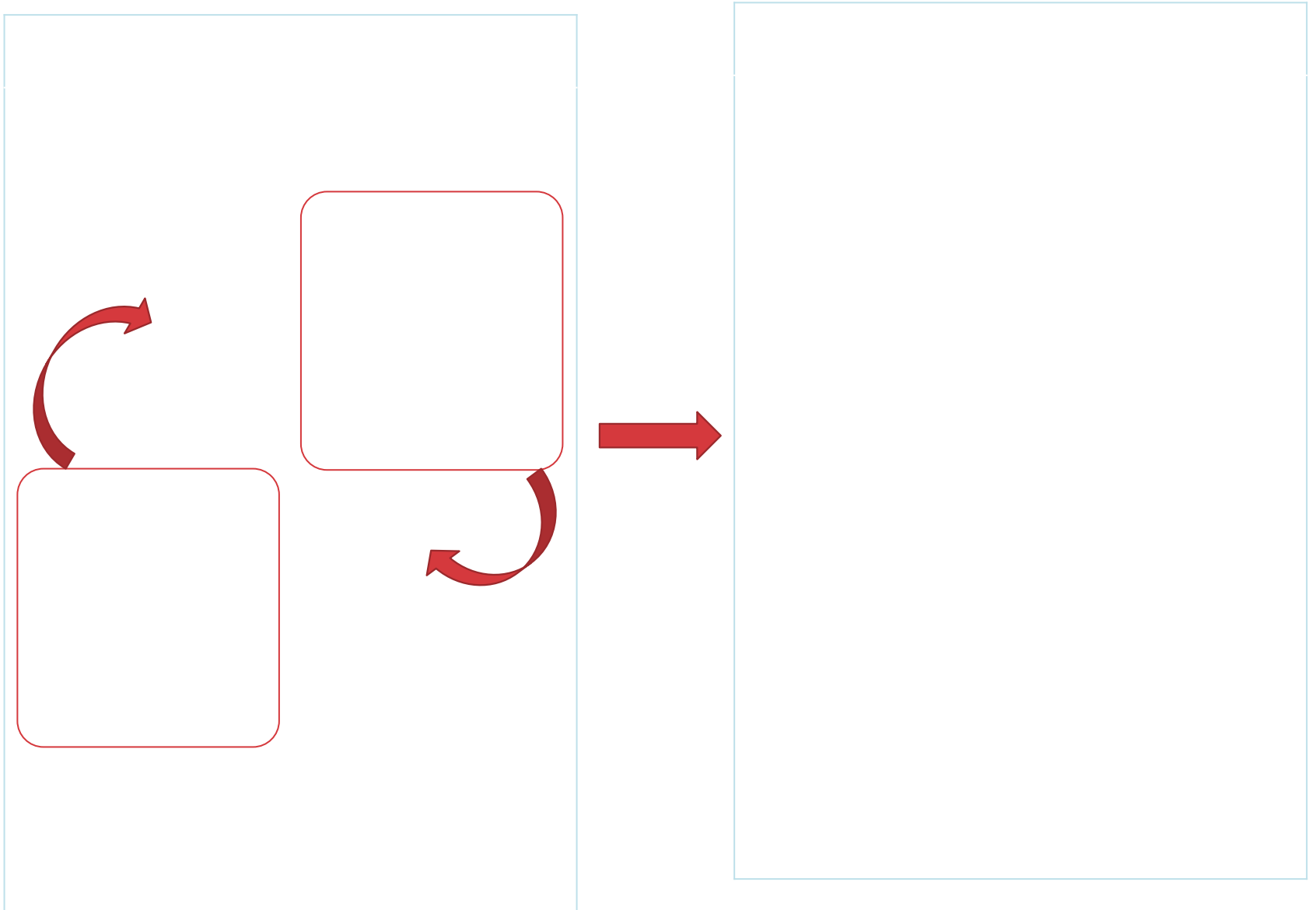
- Each transaction preserves database consistency
- A schedule is serializable if it is equivalent to a serial schedule.
- In serial schedules, execution of one transaction is allowed at a time i.e. no concurrency is allowed.
- In serializable schedules, execution of more than one transactions are allowed at a same time i.e. concurrency is allowed.
- Types (forms) of serializability
  - ✔ Conflict serializability
  - ✔ View serializability

Transaction (Conflicting Operations)

- Let o1 and o2 be the two operations of transactions T1 and T2 respectively.
- o1 = read(T), o2 = read(T) => o1 and o2 don't conflict
- o1 = read(T), o2 = write(T) => o1 and o2 conflict
- o1 = write(T), o2 = read(T) => o1 and o2 conflict
- o1 = write(T), o2 = write(T) => o1 and o2 conflict
- Operations o1 and o2 conflict if there exists at least one of these operations o1 or o2 wrote **T**.
- If both the transactions access different data item or both the transactions read the same data item then they are not conflict.

Transaction (Conflict Serializability)

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



Conflict Serializability

- If a schedule S can be transformed into a schedule S´ by a series of swaps of non-conflicting instructions, we say that S and S´ are **conflict equivalent.**
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

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

Schedule 3

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

Schedule 6

Transaction (View Serializability)

- Let there be two schedules with the same set of transactions. Both the schedules are said to be view serialized, if the following three conditions are satisfied, for each data item **T:**
  - ✔ Initial Read
  - ✔ Updated Read
  - ✔ Final Write

Transaction (View Serializability)

Initial Read

- If in schedule R1, transaction T1 reads the initial value of **T**, then in schedule R2 also transaction T1 must read the initial value of **T**.

R1

T1 : Read (T)
T2 : Write (T)

R2

T1 : Write (T)
T2 : Read (T)

R3

T1 : Read (T)
T2 : Read (T)

Schedules R1 and R2 are not view equivalent because initial read operation in R1 is done by T1 and in R2 it is done by T2

Schedules R1 and R3 are view equivalent because initial read operation in R1 is done by T1 and in R3 it is also done by T1

Transaction (View Serializability)

Update Read

- If in schedule R1 transaction T1 executes read(T), and that value was produced by transaction T2 (if any), then in schedule R2 also transaction T1 must read the value of **T** that was produced by transaction T2.

R1

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

R2

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

Schedules R1 and R2 are not view equivalent because; in R1, T3 reads the data item updated by T2 and in R2, T3 reads the data item updated by T1.

Transaction (View Serializability)

Update Read

- If in schedule R1 transaction T1 executes read(T), and that value was produced by transaction T2 (if any), then in schedule R2 also transaction T1 must read the value of **T** that was produced by transaction T2.

R1

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

R2

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

Schedules R1 and R2 are view equivalent because; in R1, T3 reads the data item updated by T2 and in R2, T3 reads the data item updated by T2.

Transaction (View Serializability)

Final Write

- If in schedule R1, transaction T1 performs final write operation on some data item **T**, then in schedule R2, transaction T1 also performs final write operation on same data item **T**.

R1

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

R2

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

Schedules R1 and R2 are view equivalent because; in R1 and R2, final write on data item A; is performed by transaction T3.

Recoverability

- Need to address the effect of transaction failures on concurrently running transactions.
- **Recoverable schedule** — if a transaction Tj reads a data items previously written by a transaction Ti , the commit operation of Ti appears before the commit operation of Tj.
- The following schedule (Schedule 11) is not recoverable if T9 commits immediately after the read

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

- If T8 should abort, T9 would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable

Recoverability

- **Cascading rollback –** a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

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

- If T10 fails, T11 and T12 must also be rolled back.
- Can lead to the undoing of a significant amount of work

Recoverability

- **Cascadeless schedules** — cascading rollbacks cannot occur; 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 read operation of Tj.
- Every cascadeless schedule is also recoverable.
- It is desirable to restrict the schedules to those that are cascadeless

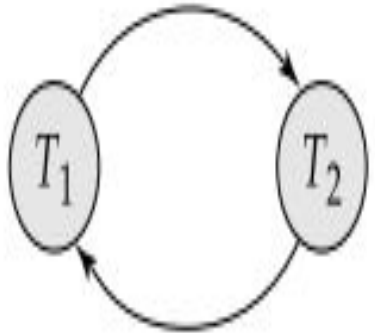
Testing for Serializability

- Consider a schedule S. We construct a directed graph, called a precedence graph, from S.
- This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule.
- The set of edges consists of all edges Ti → Tj for which one of three conditions holds:
- 1. Ti executes write(Q) before Tj executes read(Q).
- 2. Ti executes read(Q) before Tj executes write(Q).
- 3. Ti executes write(Q) before Tj executes write(Q).

Testing for Serializability

- If an edge Ti → Tj exists in the precedence graph, then, in any serial schedule S equivalent to S', Ti must appear before Tj .
- For example, the precedence graph for schedule 1 contains the single edge T1 → T2 , since all the instructions of T1 are executed before the first instruction of T2 is executed.
- Similarly, the precedence graph for schedule 2 with the single edge T2 → T1 , since all the instructions of T2 are executed before the first instruction of T1 is executed.
- The precedence graph for schedule 5 is in next slide. It contains the edge T1 → T2 , because T1 executes read(A) before T2 executes write(A). It also contains the edge T2 → T1 , because T2 executes read(B) before T1 executes write(B).
- If the precedence graph for S has a cycle, then schedule S is not conflict serializable.
- If the graph contains no cycles, then the schedule S is conflict serializable.

Testing for Serializability

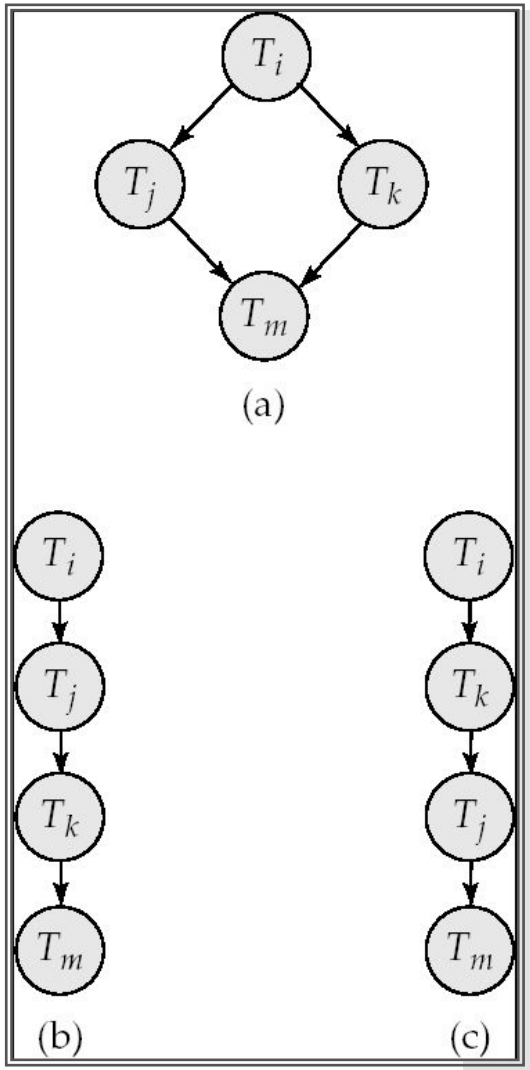


T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

schedule 5

Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order n + e where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a topological sorting of the graph.
  - This is a linear order consistent with the partial order of the graph.

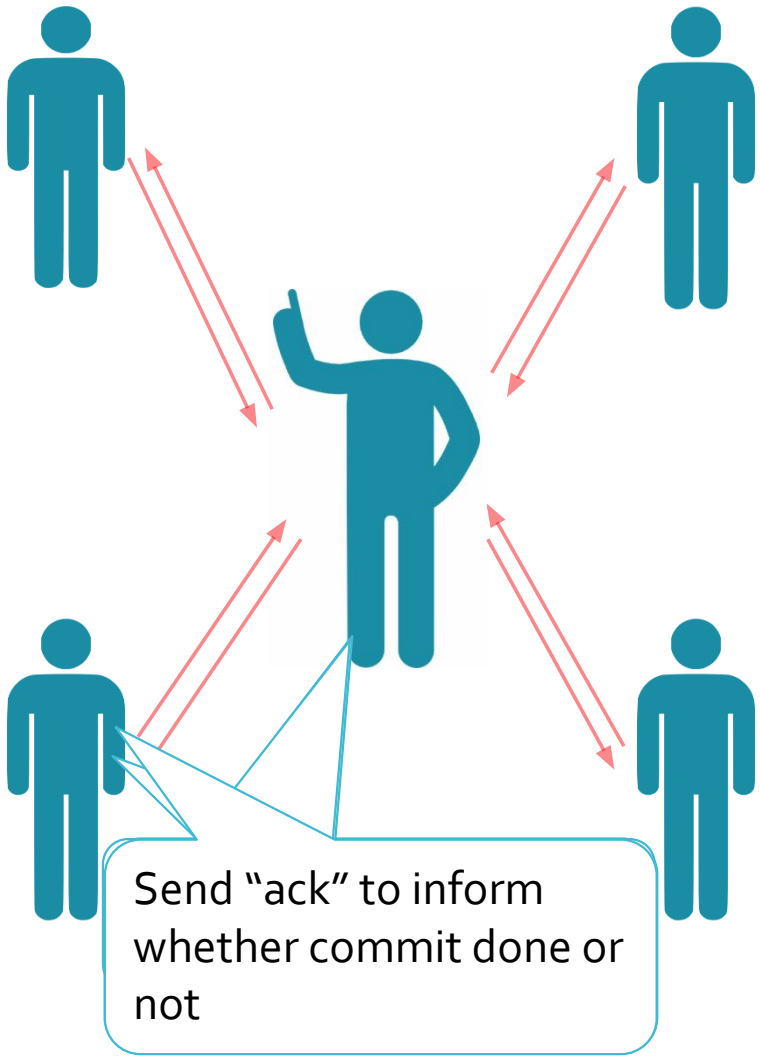


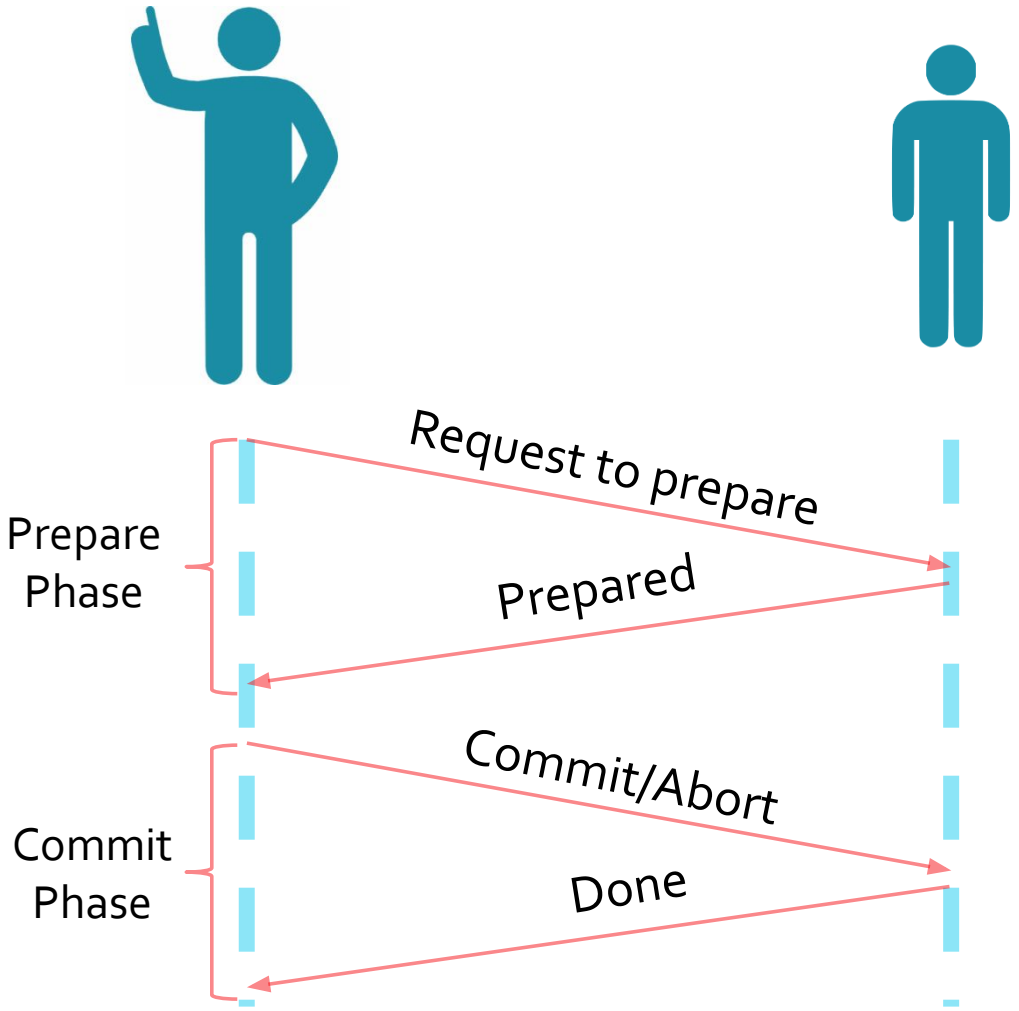
Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of NP-complete problems.
  - Thus existence of an efficient algorithm is extremely unlikely.
- However practical algorithms that just check some sufficient conditions for view serializability can still be used.

Transaction (Two Phase Commit Protocol)

- Two phase commit protocol ensures that all the participating nodes perform the same task of transaction.
- It is required to make sure either all the databases are updated or none of them are updated so as to main database synchronization.
- In two phase commit protocol there is one node which acts as a Coordinator or a Controlling Node and all other participating nodes are known as Cohorts or Participant or Slave.
- It involves 2 phases:
  - ✔ Prepare Phase (Preparing to Commit transaction)
  - ✔ Commit/Abort Phase (Final Response)


Send “ack” to inform whether commit done or not


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

Transaction (Two Phase Commit Protocol)

Prepare Phase

- Once each slave completes its transaction, it sends a "DONE" message to the Coordinator.
- When the Coordinator receives "DONE" message from all the slaves, it sends a "Prepare" (prepare to commit) message to the slaves.
- If a slave wants to commit, it sends a "Ready" message.
- If a slave do not want to commit, it sends a "Not Ready" message.

Transaction (Two Phase Commit Protocol)

Commit Phase

- When the Coordinator receives “Ready” message from all the slaves, it sends a “Global Commit” message to all the slaves
- The slaves commit the transaction and send “Commit Ack” message back to the Coordinator.
- Once a Coordinator receives “Commit Ack” message from all the slaves, then the transaction is considered to be Committed or Completed.

Transaction (Two Phase Commit Protocol)

Commit Phase

- When the Coordinator receives “Not Ready” message from all the slaves, it sends a “Global Abort” message to all the slaves
- The slaves abort the transaction and send “Abort Ack” message back to the Coordinator.
- Once a Coordinator receives “Abort Ack” message from all the slaves, then the transaction is considered to be Aborted or Error.

Transaction (Database Recovery)

- There are many scenarios where a transaction could not reach commit or abort state. Some of them are:
  - ✔ Disk Failure
  - ✔ DBMS Crash
  - ✔ OS Failure
  - ✔ Power Failure
- This may lead to Data Inconsistency or Data Loss.
- For e.g., a DBMS crashes after about 20 out of 25 operations were executed successfully. This may lead to Inconsistent Database state.
- Database Recovery is the process of restoring a database and its data to the consistent state.
- It helps to recover the data up to the state of crash or failure.

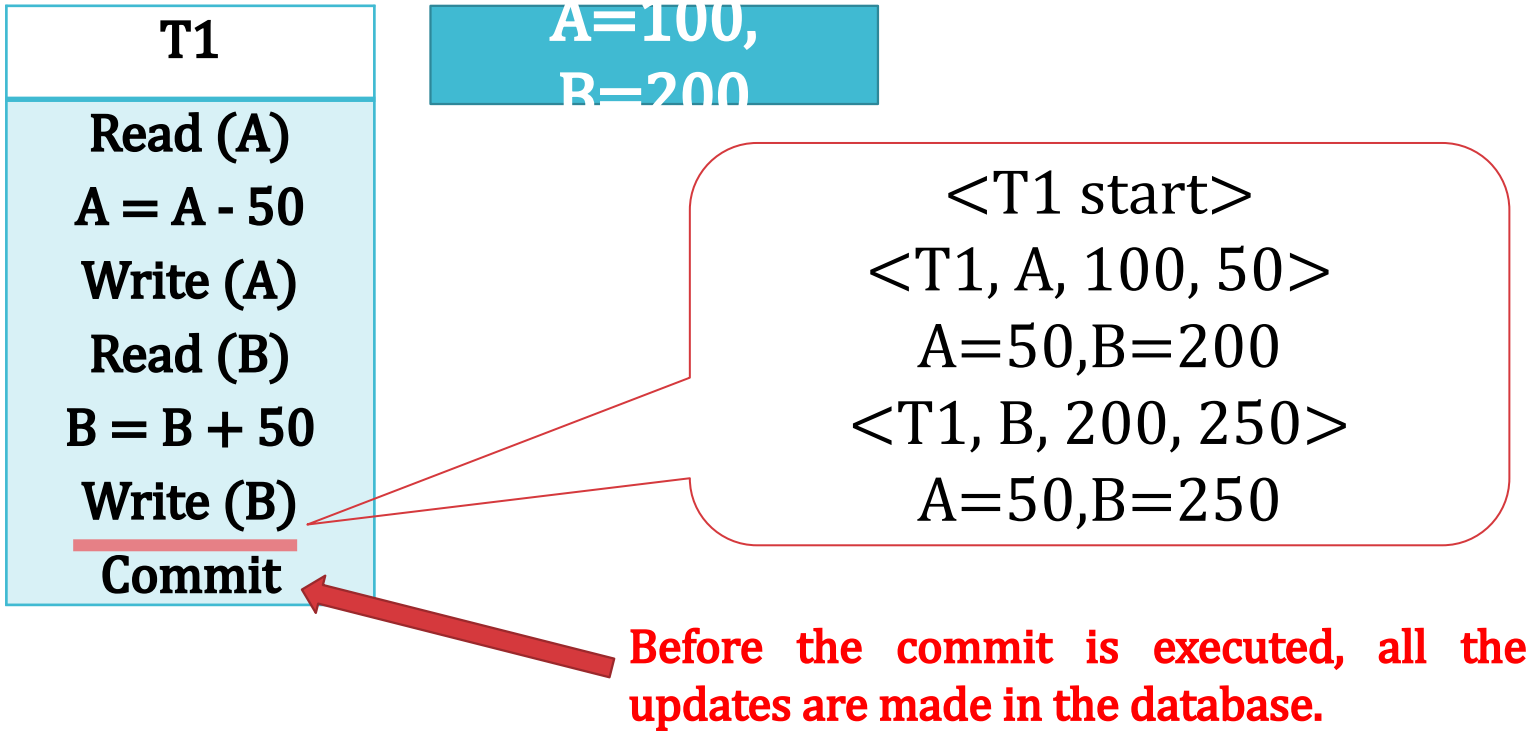
Transaction (Log Based Recovery)

- The log is a sequence of records, which maintain information about all the activities done by each operation on the database.
- A log is kept on stable storage (i.e. HDD) and it contains:
  - ✔ Start of transaction
  - ✔ Transaction-id
  - ✔ Record-id
  - ✔ Type of operation (insert, update, delete)
  - ✔ Old value, new value
  - ✔ Status of transaction that is committed or aborted.
- For e.g., when transaction T1 starts, it records <T1 Start> in the log.
- When T1 is about to execute Write Operation, just before that, it records new log as <T1,X,D1,D2>, where X is a data item , D1 is the value of X before transaction and D2 is the new value to be updated for X.
- If the transaction is completed, then it records <T1 Commit> and in case of failure, it records <T1 Abort>.

Transaction (Types of Log Based Recovery)

Immediate Database Modification

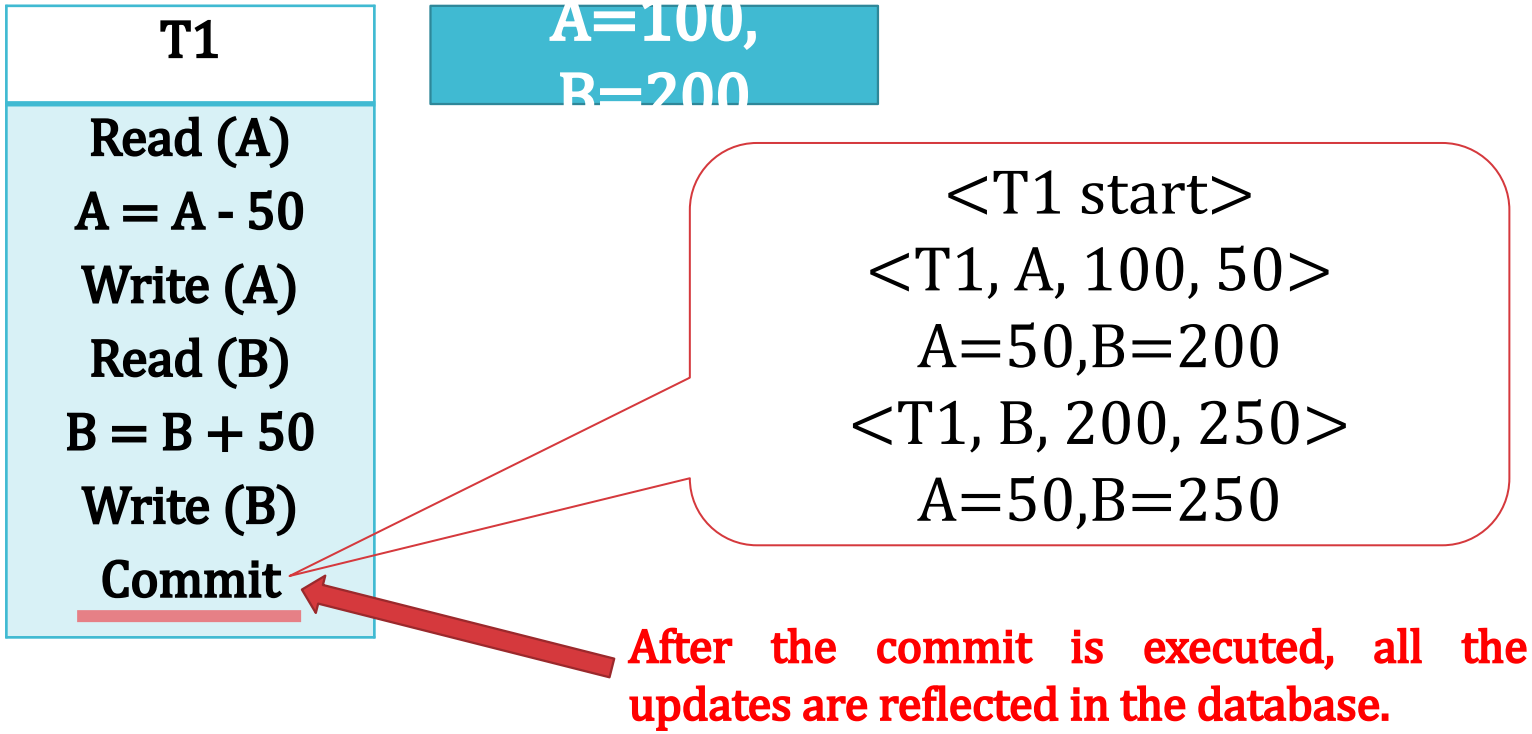
- Changes to the database are executed immediately as they happen without waiting for commit command.
- If the transaction is not committed, then the complete operation needs to be undone and transaction is restarted.
- If the transaction is committed, then changes made will be permanent.



Transaction (Types of Log Based Recovery)

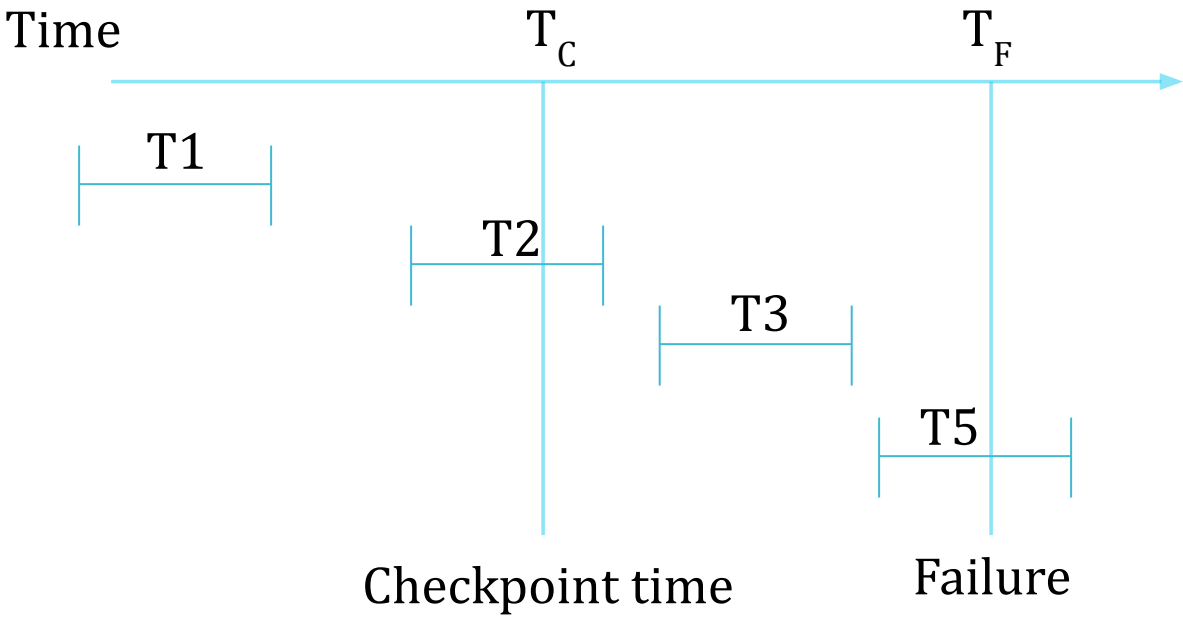
Deferred Database Modification

- Changes to the database are executed only after the commit command is executed.
- If the transaction is not committed, then the changes wont be reflected in database and only transaction is restarted.
- If the transaction is committed, then all the changes are reflected and will be permanent.



Transaction (Checkpoints)

- It may be possible that Log Based Recovery consumes more time while recovering the data using Logs. Thus, to reduce the time consumption, we can use Checkpoints.
- Checkpoint is a point which describes that any operation of the transaction done before that point are executed correctly and stored safely in the database.



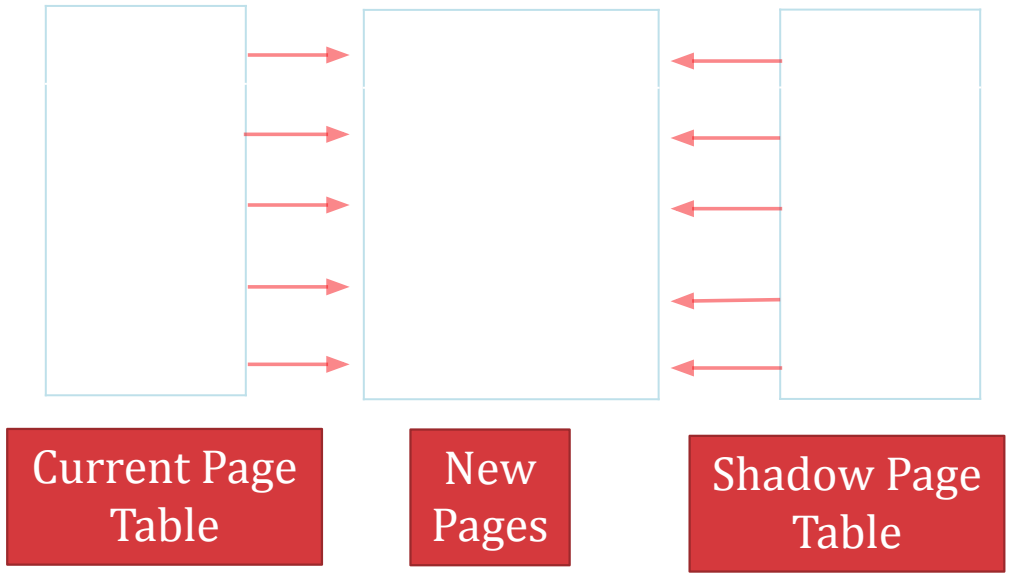
Working of Checkpoints in case of Failure

- ✔ **T1 will be ignored as it is committed before checkpoint.**
- ✔ **T2 and T3 will restart again as they were active even after checkpoint, but committed before Failure.**
- ✔ **T5 will be completely ignored as it was active even after the checkpoint and has not committed.**

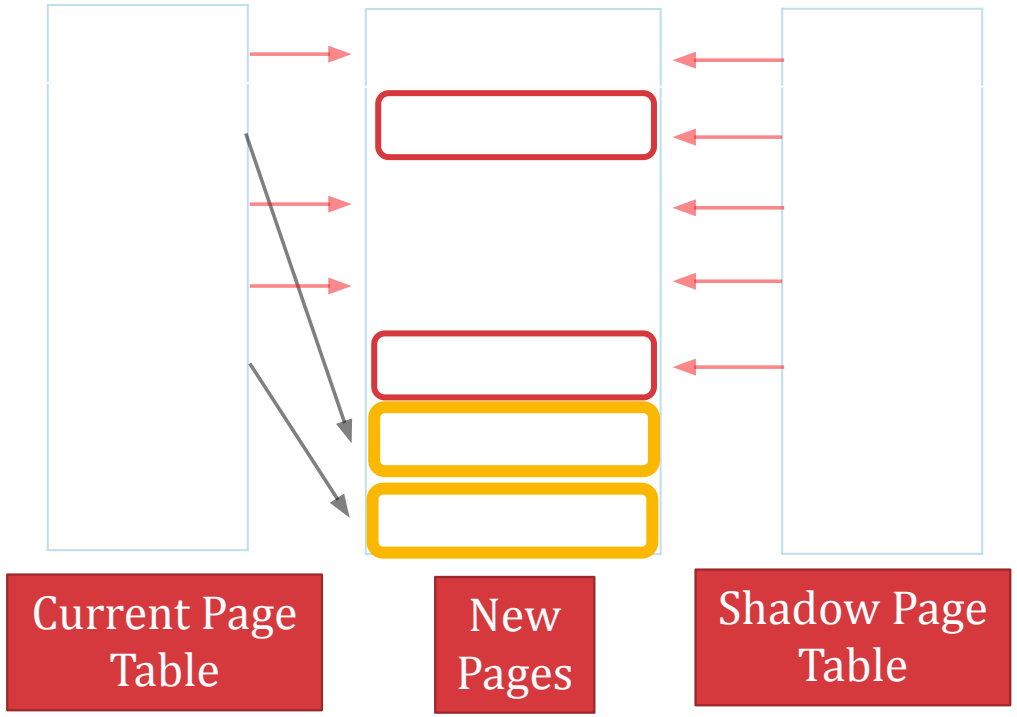
Transaction (Shadow Paging)

- Shadow Paging is an another method of database recovery and it is very useful in case of serial transactions.
- The database is fragmented into fixed sized blocks referred as **PAGES**. Each pages are stored in a **Page Table**.
- Shadow Paging maintains 2 Page Tables:
  - ✔ Current Page Table
  - ✔ Shadow Page Table
- When the transaction starts, both the tables are identical.
- During transaction, only Current Page Table changes, Shadow Page Table never changes.
- All the operations are performed on Current Page Table.
- When a database page is updated:
  - ✔ A New Copy of the page is created
  - ✔ Current Page Table points to the New Copy.
  - ✔ Then, the updates are performed

Transaction (Shadow Paging)



- ✔ When the transaction starts, both Current Page Table and Shadow Page Table points to Same Pages in New Pages



- ✔ When the transaction updates Page 2 and Page 5, then a new copy of the Page 2 and 5 is created in New Pages.
- ✔ Current Page Table points to New Copy and Shadow Page Table continues pointing Old Copy
- ✔ Thus, in case of Failure, Shadow Page Table can be used to recover the data as it keeps pointing Old Pages.

Concurrency Control

- Concurrency control is the procedure for managing simultaneous operations without conflicting with each other.
- Concurrency control is used to address conflicts which occur with a multi-user system who have access to perform READ and WRITE operation in database.
- 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.
- Resolve read-write and write-write conflicts.
- Apply isolation through mutual exclusion between conflicting transactions.
- Concurrency control helps to ensure serializability.

Concurrency Control

- Allowing Concurrent execution of operations may lead to following problems:
  - ✔ **Lost Update:** if two transactions T1 and T2 both read the same data and then update it; in this case, first update will be overwritten by the second update.
  - ✔ **Dirty Read:** when one transaction update some item and then fails. Moreover, this updated item is accessed by another transaction before it is rolled back to its initial value.
  - ✔ **Incorrect Retrieval:** when one transaction accesses data to use it in other operation; but before it can use, another transaction updates that data and commits.

Concurrency Control

- A lock is a variable associated with data item to control concurrent access to that data item.
- Data items can be locked in 2 ways:
  - ✔ **Exclusive (X) mode**: Data item can be both read as well as write. X-lock is requested using **lock-X** instruction.
  - ✔ **Shared (S) mode**: Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock-compatibility matrix:

T1 ----------

T2----

	S	X
S	true	false
X	false	false

- Transaction can proceed only after request is granted.

Concurrency Control (Lock Based Protocol)

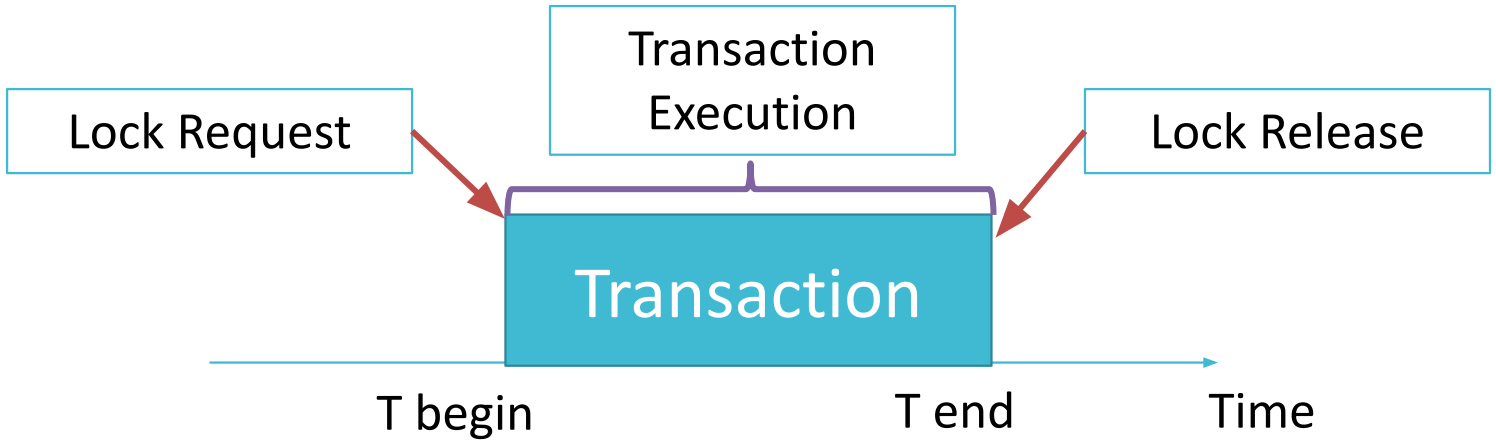
- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - ✔ but if any transaction holds an exclusive on the item no other transaction may hold any lock 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.

T2: lock-S(A);
read (A);
unlock(A);
lock-S(B);
read (B);
unlock(B);
display(A+B)

- **Example of Transactions using Locking.**
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks

Concurrency Control (Lock Based Protocol)

- The execution phase of transaction can be described as following:
  - ✔ When execution of transaction starts, create a list of data items and type of lock, it needs and request for that lock.
  - ✔ When all the locks are granted, transaction continues execution of its operation.
  - ✔ As soon as the transaction releases its first lock, it cannot demand for any lock; but can only release the acquired locks.



Concurrency Control (Pitfalls of Lock Based Protocol)

- Consider the following 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 T3 nor T5 can make progress. lock-X(B) causes T5 to wait for T3 to release its lock on B, while lock-X(A) causes T3 to wait for T5 to release its lock on A.
- Such a situation is called a **Deadlock**.
- To handle a deadlock, one of T3 or T5 must be rolled back and its locks must be released.

Concurrency Control (Pitfalls of Lock Based Protocol)

- 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.
- Starvation can be managed in following way:
  - ✔ There is no other transaction holding a lock that arises a certain type of conflicts.
  - ✔ There is no other transaction who requested lock before any other transaction and is still waiting to acquire lock.

Concurrency Control (Two Phase Lock Protocol)

- The Two Phase Lock protocol has 2 phases:
  - ✔ **Growing Phase:**
    - transaction may obtain locks
    - transaction may not release locks
  - ✔ **Shrinking Phase:**
    - transaction may release locks
    - transaction may not obtain locks
- It ensures serialized transactions based on lock points (the point where a transaction acquired its final lock).

Concurrency Control (Two Phase Lock Protocol – Lock Conversions)

- We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock.
- Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Concurrency Control (Two Phase Lock Protocol – Lock Conversions)

- The Two Phase Lock Conversions are as follows:
  - ✔ **Growing Phase:**
    - transaction receive Lock - S
    - transaction receive Lock - X
    - transaction convert Lock - S to Lock - X(upgrade)
  - ✔ **Shrinking Phase:**
    - transaction releases Lock - S
    - transaction releases Lock - X
    - transaction convert Lock - X to Lock – S (downgrade)
- It ensures serialized transactions.

Concurrency Control (Two Phase Lock Protocol)

- The Two Phase Lock can be of 2 types:

✔ **Strict two phase locking protocol:**

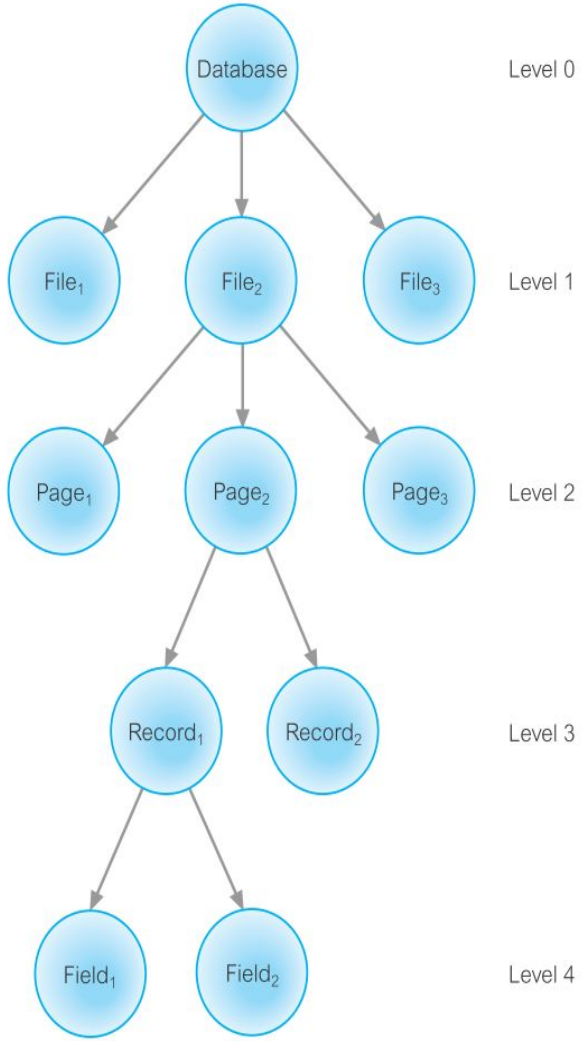
- A transaction may release all the shared locks after the Lock Point, but cannot release any of the exclusive locks until the transaction commits or aborts.
- It ensures that if the data is being modified by one transaction, then other transaction cannot read it until first transaction commits.

✔ **Rigorous two phase locking protocol:**

- A transaction is not allowed to release any lock (either shared or exclusive) until it commits.
- Until the transaction commits, other transaction can not acquire even a shared lock on a data item which is already been used by active transaction.

Concurrency Control (Multiple Granularity)

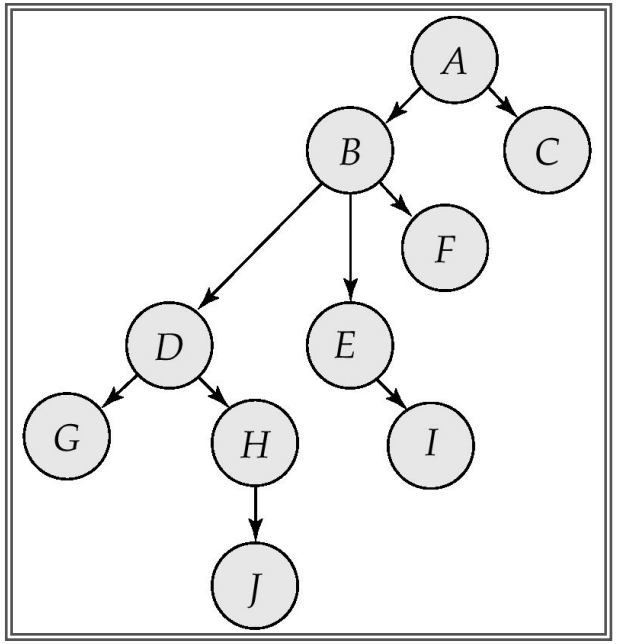
- Data Items are represented in a tree like structure. It has a parent as well as child.
- If a lock is acquired on Parent node, then automatically, all its children are locked.
- If any transaction requests lock on any child, then compiler checks whether a Lock on any parent has not been given.
- If Page2 has been locked by any transaction, then all its children i.e. Record1, Record 2, Field1 and Field2 will be locked.
- At the same time, if other transaction request lock on Record2, then lock will not be granted as its Parent, Page 2 has been locked by some other transaction.



Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering → on the set D = {d1, d2 ,..., dh} of all data items.
  - If di → dj then any transaction accessing both di and dj must access di before accessing dj.
  - 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.

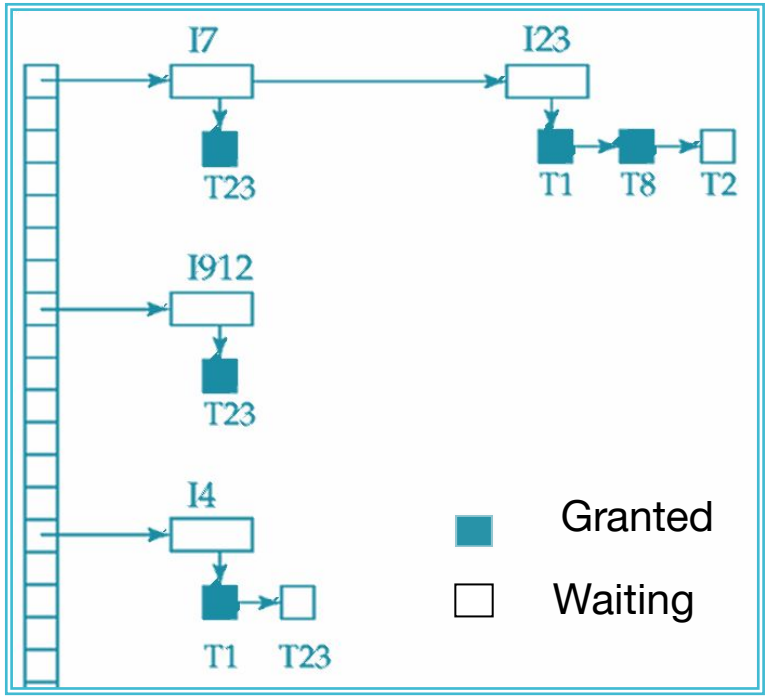
Graph-Based Protocols



- Only exclusive locks are allowed.
- The first lock by Ti may be on any data item. Subsequently, a data Q can be locked by Ti only if the parent of Q is currently locked by Ti.
- Data items may be unlocked at any time.
- A data item that has been locked and unlocked by Ti cannot subsequently be relocked by Ti

Concurrency Control (Implementing Locking)

- Transactions send lock and unlock requests to **Lock Manager.** All transactions has to wait for lock manager's response.
- Lock manager responds to the request with either "Lock Granted" or "Wait" message.
- Lock manager keeps all the records of lock grants and pending requests in a **Lock Table.**



- ✔ Dark rectangles indicate granted locks, white ones indicate waiting requests.
- ✔ It has the record of type of locks issued on the data items.
- ✔ The new requests are listed in the queue from the bottom of the table.
- ✔ If a transaction releases a lock, it will check which transaction can be provided with that lock.

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - **intention-shared (IS):** indicates explicit locking at a lower level of the tree but only with shared locks.
  - **intention-exclusive (IX):** indicates explicit locking at a lower level with exclusive or shared locks
  - **shared and intention-exclusive (SIX):** the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

The compatibility matrix for all lock modes is:

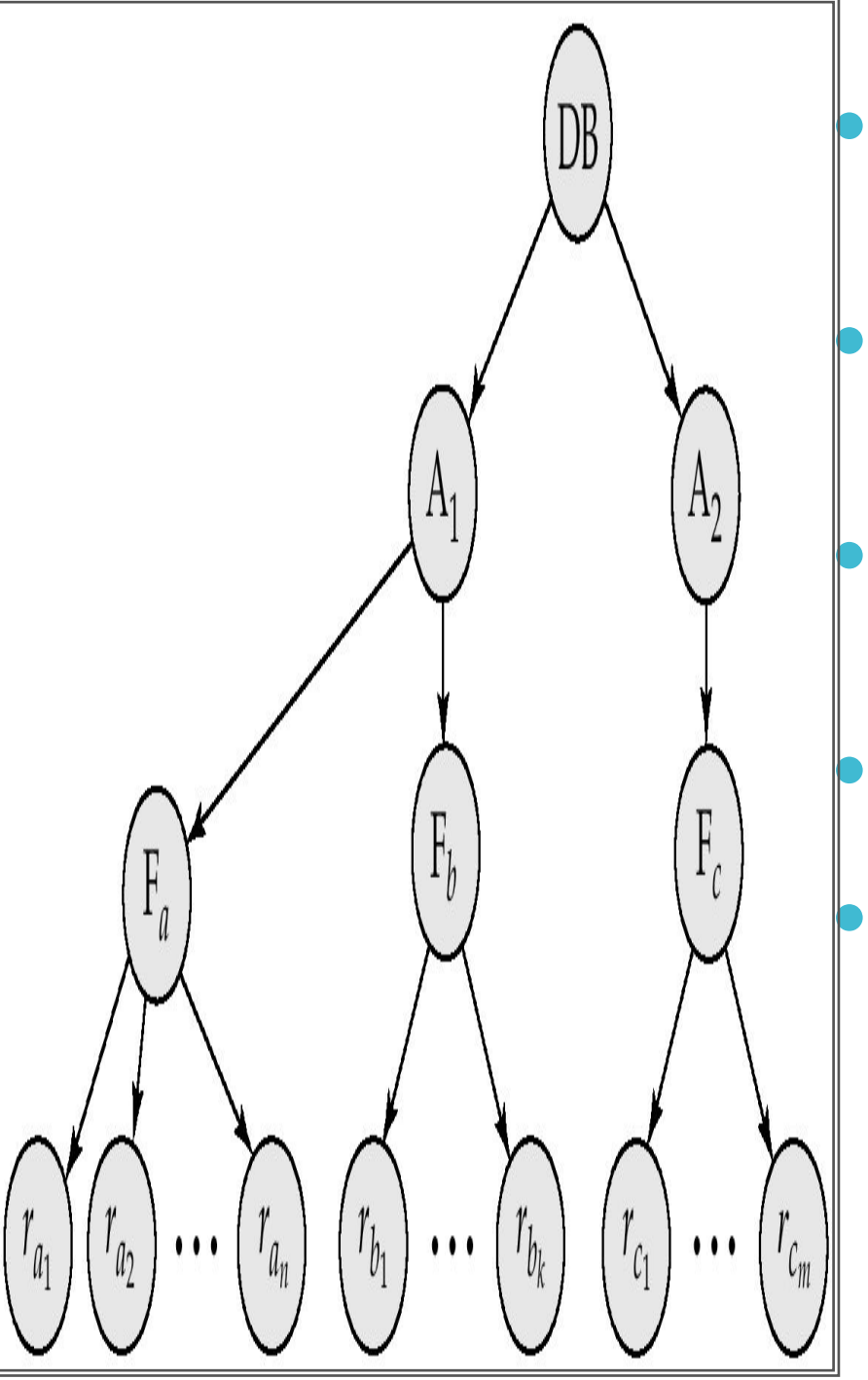
	IS	IX	S	S IX	X
IS	✔	✔	✔	✔	×
IX	✔	✔	×	×	×
S	✔	×	✔	×	×
S IX	✔	×	×	×	×
X	×	×	×	×	×

Multiple Granularity Locking Scheme

Transaction Ti can lock a node Q, using the following rules:

- The lock compatibility matrix must be observed.
- The root of the tree must be locked first, and may be locked in any mode.
- A node Q can be locked by Ti in S or IS mode only if the parent of Q is currently locked by Ti in either IX or IS mode.
- A node Q can be locked by Ti in X, SIX, or IX mode only if the parent of Q is currently locked by Ti in either IX or SIX mode.
- Ti can lock a node only if it has not previously unlocked any node (that is, Ti is two-phase).
- Ti can unlock a node Q only if none of the children of Q are currently locked by Ti.

- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.



- Suppose that transaction T18 reads record ra2 in file Fa . Then, T18 needs to lock the database, area A1 , and Fa in IS mode (and in that order), and finally to lock ra2 in S mode.
- Suppose that transaction T19 modifies record ra9 in file Fa . Then, T19 needs to lock the database, area A1 , and file Fa in IX mode, and finally to lock ra9 in X mode.
- Suppose that transaction T20 reads all the records in file Fa . Then, T20 needs to lock the database and area A1 (in that order) in IS mode, and finally to lock Fa in S mode.
- Suppose that transaction T21 reads the entire database. It can do so after locking the database in S mode.
- We note that transactions T18 , T20 , and T21 can access the database concurrently. Transaction T19 can execute concurrently with T18 , but not with either T20 or T21 .

Concurrency Control (Time Stamp Based Protocol)

- Each transaction is issued a timestamp when it enters the system. An old transaction T1 has time-stamp TS(T1) and new transaction T2 is assigned time-stamp TS(T2), then following condition satisfies: TS(T1) < TS(T2). For e.g. TS(T1) = 0002 and TS(T2) = 0005, then TS(T1) < TS(T2).
- Transaction with older timestamp are give priority for execution.
- The protocol manages concurrent execution such that the **read** and **write** operations are executed in timestamp order.
- The protocol maintains for each data Q two timestamp values:
  - **W-timestamp(Q):** largest time-stamp of any transaction that executed write(Q) successfully.
  - **R-timestamp(Q):** largest time-stamp of any transaction that executed read(Q) successfully.

Concurrency Control (Time Stamp Based Protocol)

- Suppose a transaction T1 issues a **read(Q).**
  - ✔ If **TS(T1)** <= **W-timestamp(Q)**, then the read operation is rejected as **write** operation is in execution. For e.g. TS(T1) = 0002, W-timestamp(Q) = 0008
  - ✔ If **TS(T1)** >= **W-timestamp(Q)**, then the read operation is executed as it is assumed that **write** operation is committed. For e.g. TS(T1) = 0005, W-timestamp(Q) = 0003
- Suppose a transaction T1 issues a **write(Q).**
  - ✔ If **TS(T1)** <= **R-timestamp(Q)**, then the write operation is rejected as **read** operation is in execution. For e.g. TS(T1) = 0002, R-timestamp(Q) = 0008
  - ✔ If **TS(T1)** >= **R-timestamp(Q)**, then the write operation is executed as it is assumed that **read** operation is complete. For e.g. TS(T1) = 0005, R-timestamp(Q) = 0003

Deadlock

- A deadlock is a situation in which one transaction is waiting for other transaction to release the resources.
- Let there be a set of transaction {T1, T2, T3} such that T1 is waiting for a data item that T2 holds, and T2 is waiting for a data item that T3 holds, T3 is waiting for a data item that T1 holds.
- Here, each transaction is waiting for some other. Hence, no transaction will work smoothly.
- The only solution to this situation is transaction being rolled back to its initial state every time and come again to check the availability of resources.

Deadlock Detection

- A Deadlock Condition arise only if all these four conditions are true together.

✔ **Mutual Exclusion:** One Process to work at one time. If any other process requests the same resource, it will wait till the first release it.

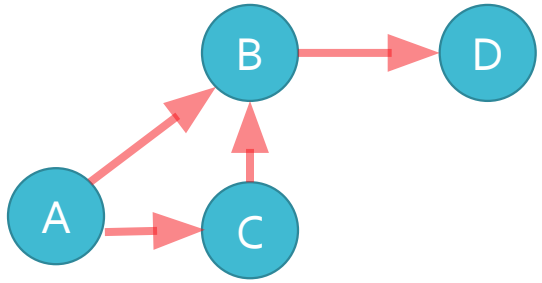
✔ **Hold and Wait:** One process holds one resource and waiting for other resource that is held by other process.

✔ **Non-Preemption:** Process can release resources only when it has completed its task. No force to acquire resource

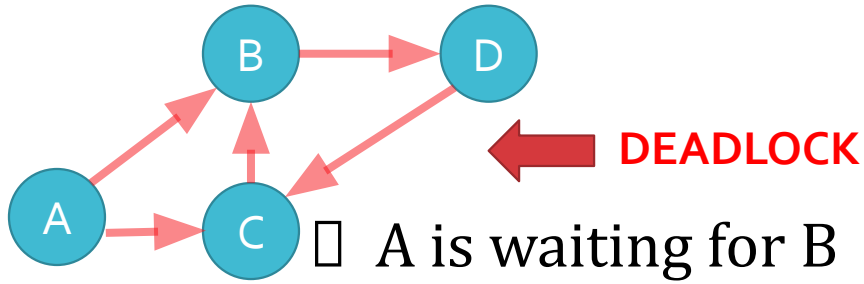
✔ **Circular Wait:** One Process waiting for other, Other waiting for another and thus creates the cycle.

Deadlock Detection

- The most simpler way to detect deadlock is to use **Wait-for Graph.**
- Each Transaction is represented by a node. When a Transaction Ti is waiting for the resource held by Tj, a **directed edge** is drawn from Ti to Tj (Ti -----□ Tj).
- If the Wait-for graph has any cycle, then we can say there is a **Deadlock** and all the transactions in the cycle are also said to be deadlocked.



- A is waiting for B and C
- C is waiting for B
- B is waiting for D
- Still, there is no cycle, hence it is not **Deadlock.**



- A is waiting for B and C
- B is waiting for D
- D is waiting for C
- C is waiting for B
- There is a cycle B – D – C - B, It is a **Deadlock.** for B, C, and D.

Deadlock Recovery

- The most simpler way to recover from deadlock is to **Roll Back** any of the transactions.
- The transaction which incurs minimum loss is rolled back and it is known as **Victim.**
- The decision to roll back a transaction is made on following criteria:
  - ✔ The transaction which have minimum locks
  - ✔ The transaction which has executed less work
  - ✔ The transaction which is very far from completion

Deadlock Prevention

- A Deadlock can be prevented using:

✔ **Wait-Die Approach:**

- If an older transaction is requesting a resource which is held by younger transaction, then older transaction waits.
- If an younger transaction is requesting a resource which is held by older transaction, then younger transaction is killed and rolled back.

Wait-Die	
O needs a resource held by Y	O Waits
Y needs a resource held by O	Y Dies

Deadlock Prevention

- A Deadlock can be prevented using:

✔ **Wound-Wait Approach:**

- 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 releases the resource.
- If an younger transaction is requesting a resource which is held by older transaction, then younger transaction waits till older transaction releases resource.

Wound-Wait	
O needs a resource held by Y	Y Hurts / Dies
Y needs a resource held by O	Y Waits

Deadlock Prevention

- A Deadlock can be prevented using:

✔ **Timeout-Based Approach:**

- A transaction waits for a lock only for a specified amount of time. After that, the transaction is rolled back.
- So deadlock never occurs.

Thanks

Marwadi University