# Transaction Processing

Unit-7

What is transaction?

- 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**

Works as a single logical unit

Transaction

read (A)
A = A − 50
write (A)
read (B)
B = B + 50
write (B)

Operations
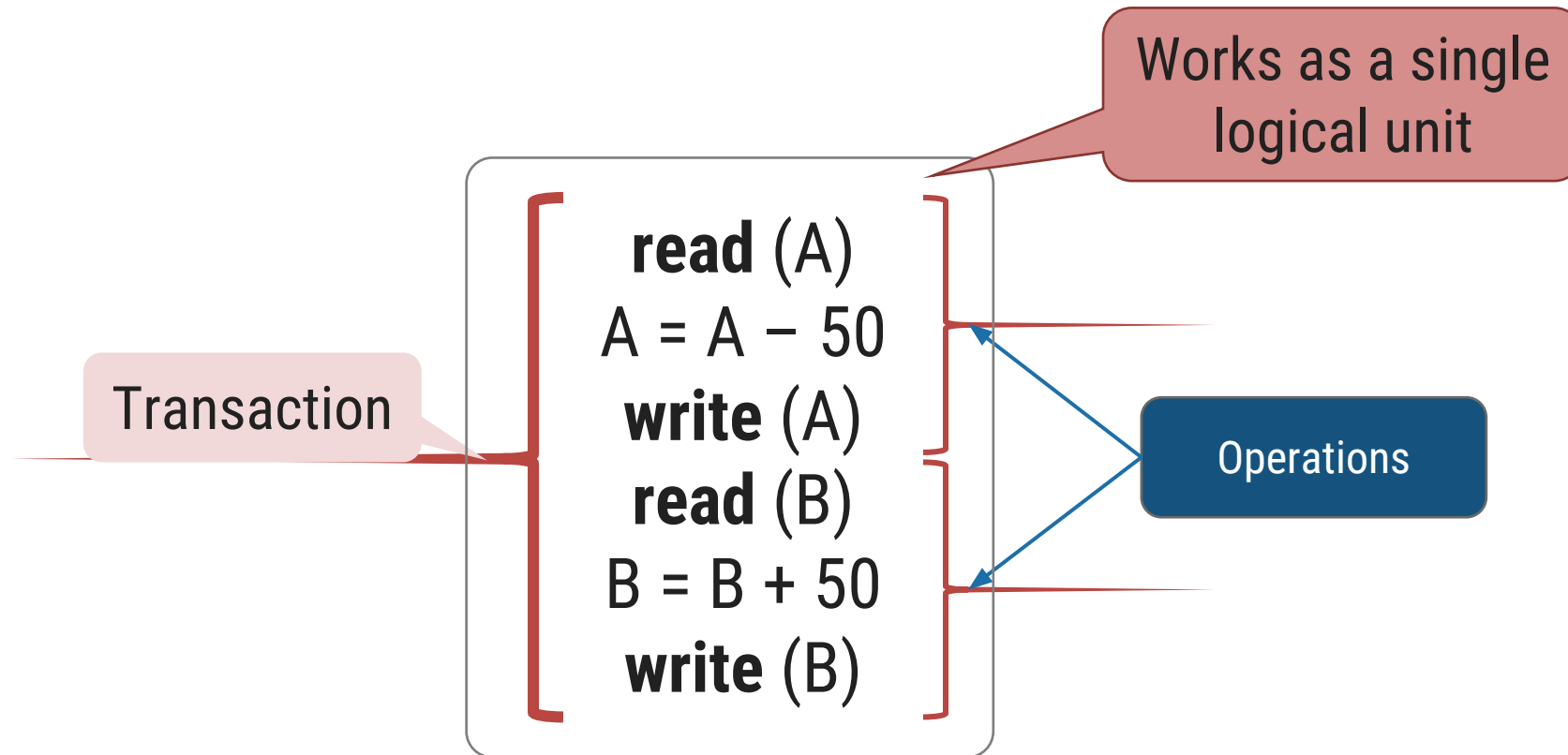
ACID properties of transaction

- **A**tomicity (Either transaction execute 0% or 100%)

- **C**onsistency (Database must remain in a consistent state after any transaction)

- **I**solation (Intermediate transaction results must be hidden from other concurrently executed transactions)

- **D**urability (Once a transaction completed successfully, the changes it has made into the database should be permanent)

ACID properties of transaction (Atomicity)

- This property states that a **transaction must be treated as an atomic unit**, that is, **either all of its operations are executed or none**.

- **Either transaction execute 0% or 100%**.

- For example, consider a transaction to transfer Rs. 50 from account A to account B.

- In this transaction, if Rs. 50 is deducted from account A then it must be added to account B.

0%

read (A)

A = A − 50

write (A)   FAIL

read (B)

B = B + 50

write (B)

100%
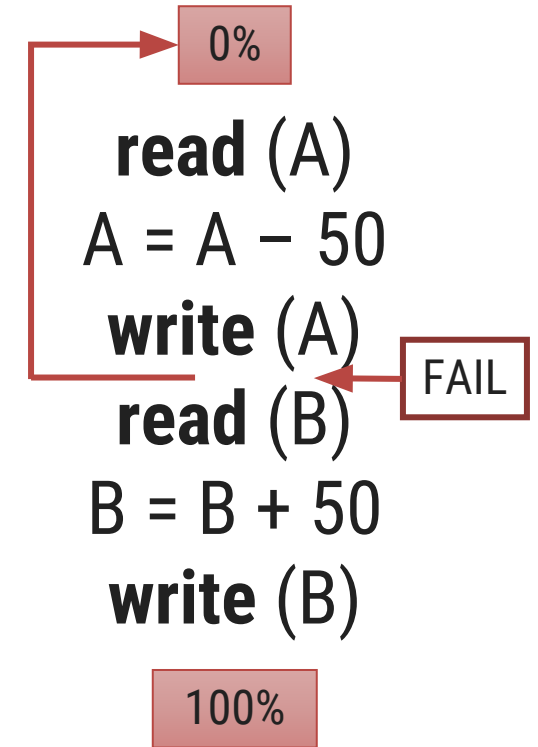
ACID properties of transaction (Consistency)

- The **database must remain in a consistent state** after any transaction.

- If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

- In our example, total of A and B must remain same before and after the execution of transaction.

A=500, B=500
A+B=1000

read (A)
A = A − 50
write (A)
read (B)
B = B + 50
write (B)

A=450, B=550
A+B=1000

ACID properties of transaction (Isolation)

- **Changes occurring in a particular transaction will not be visible to any other transaction until it has been committed**.

- **Intermediate transaction results must be hidden** from other concurrently executed transactions.

- In our example once our transaction starts from first step (step 1) its result should not be access by any other transaction until last step (step 6) is completed.

Start Transaction

read (A)

A = A − 50

write (A)

read (B)

B = B + 50

write (B)
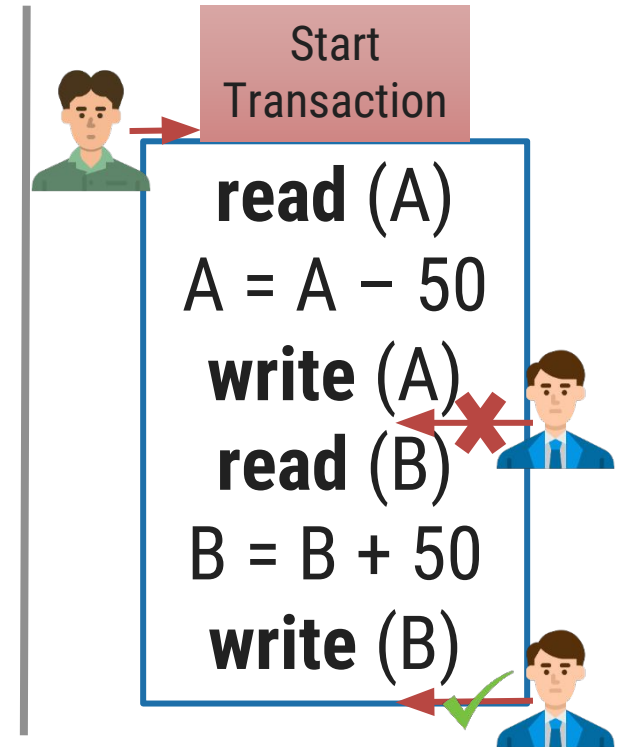
ACID properties of transaction (Durability)

- After a transaction completes successfully, the **changes it has made to the database persist (permanent)**, even if there are system failures.

- Once our transaction completed up to last step (step 6) its result must be stored permanently. It should not be removed if system fails.

A=500, B=500

read (A)
A = A − 50
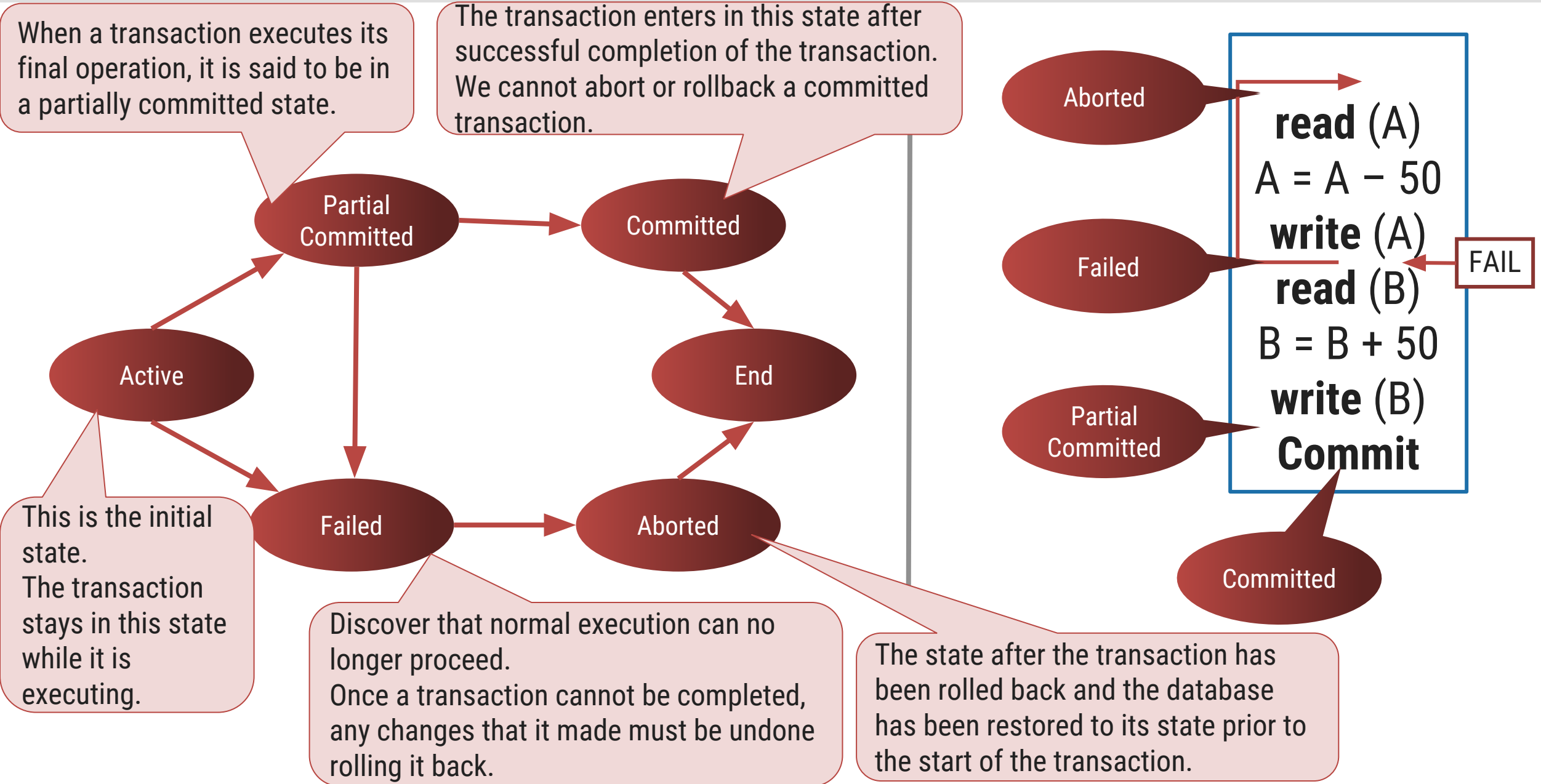**write** (A)
read (B)
B = B + 50
write (B)

A=450, B=550

These values must be stored permanently in the database

Transaction State Diagram \ State Transition Diagram

When a transaction executes its final operation, it is said to be in a partially committed state.

The transaction enters in this state after successful completion of the transaction. We cannot abort or rollback a committed transaction.

This is the initial state.
The transaction stays in this state while it is executing.

Discover that normal execution can no longer proceed.
Once a transaction cannot be completed, any changes that it made must be undone rolling it back.

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

Aborted

Failed

Partial Committed

Committed

read (A)
A = A − 50
write (A)
read (B)
B = B + 50
write (B)
Commit

FAIL

Partial Committed

Active

Committed

End

Failed

Aborted

Transaction State Diagram \ State Transition Diagram

- Active
  - This is the **initial state**.
  - The transaction **stays in this state while it is executing**.
- Partial Committed
  - When a transaction **executes its final operation/ instruction**, it is said to be in a partially committed state.
- Failed
  - Discover that **normal execution can no longer proceed**.
  - Once a transaction **cannot be completed**, any **changes that it made must be undone rolling it back**.
- Committed
  - The transaction enters in this state **after successful completion of the transaction** (after committing transaction).
  - 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 its state prior to the start of the transaction**.

What is schedule?

- A schedule is a **process of grouping the transactions** into one and **executing them in a predefined order**.
- A schedule is the **chronological (sequential) order in which instructions are executed** in a system.
- A schedule is required in a database because when some transactions execute in parallel, they may affect the result of the transaction.
- Means if one transaction is updating the values which the other transaction is accessing, then the order of these two transactions will change the result of another transaction.
- Hence a schedule is created to execute the transactions.

Example of schedule

Schedule		Schedule Execution
T1	**T2**	**A=B=1000**
Read (A)		Read (1000)
A = A - 50		A = 1000 - 50
Write (A)		Write (950)
Read (B)		Read (1000)
B = B + 50		B = 1000 + 50
Write (B)		Write (1050)
Commit		Commit
	Read (A)	Read (950)
	temp = A * 0.1	temp = 950 * 0.1
	A = A - temp	A = 950 - 95
	Write (A)	Write (855)
	Read (B)	Read (1050)
	B = B + temp	B = 1050 + 95
	Write (B)	Write (1145)
	Commit	Commit

Example of schedule

Schedule		Schedule Execution
T1	**T2**	**A=B=1000**
Read (A)		Read (1000)
Temp = A * 0.1		Temp = 1000 * 0.1
A = A - temp		A = 1000 - 100
Write (A)		Write (900)
Read (B)		Read (1000)
B = B + temp		B = 1000 + 100
Write (B)		Write (1100)
Commit		Commit
	Read (A)	Read (900)
	A = A - 50	A = 900 - 50
	Write (A)	Write (850)
	Read (B)	Read (1100)
	B = B + 50	B = 1100 + 50
	Write (B)	Write (1150)
	Commit	Commit

Serial schedule

 A serial schedule is a schedule in which **no transaction starts until a running transaction has ended**.

 A serial schedule is a schedule in which **one transaction is executed completely before starting another transaction**.

 Transactions are executed one after the other.

 This type of schedule is called a serial schedule, as transactions are executed in a serial manner.

Example of Serial Schedule

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

Non-serial Schedule (Interleaved Schedule)

- Schedule that **interleave the execution of different transactions**.

- Means **second transaction is started before the first one could end** and execution can switch between the transactions back and forth.

- It contains many possible orders in which the system can execute the individual operations of the transactions.

Example of Non-serial Schedule (Interleaved Schedule)

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

Equivalent Schedule

- If two schedules **produce the same result after execution**, they are said to be equivalent schedule.

- They may yield the same result for some value and different results for another set of values.

- That's why this equivalence is not generally considered significant.

Equivalent Schedule

Schedule-1 (A=B=1000)		Both schedules are equivalent — In both schedules the sum "A + B" is preserved.	Schedule-2 (A=B=1000)	
T1	**T2**		**T1**	**T2**
Read (A) A = A - 50 Write (A)			Read (A) A = A - 50 Write (A) Read (B) B = B + temp Write (B) Commit	
	Read (A) temp = A * 0.1 A = A - temp 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 + 50 Write (B) Commit
	Read (B) B = B + temp Write (B) Commit			

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 instructions

❑ Let l_i and l_j be two instructions of transactions T_i and T_j respectively.

1. $l_i = read(Q)$, $l_j = read(Q)$
   l_i and l_j don't conflict

T_i	T_j
read (Q)	
	read (Q)

T_i	T_j
	read (Q)
read (Q)	

2. $l_i = read(Q)$, $l_j = write(Q)$
   l_i and l_j conflict

T_i	T_j
read (Q)	
	write(Q)

T_i	T_j
	write(Q)
read (Q)	

3. $l_i = write(Q)$, $l_j = read(Q)$
   l_i and l_j conflict

T_i	T_j
write(Q)	
	read (Q)

T_i	T_j
	read (Q)
write(Q)	

4. $l_i = write(Q)$, $l_j = write(Q)$
   l_i and l_j conflict

T_i	T_j
write(Q)	
	write(Q)
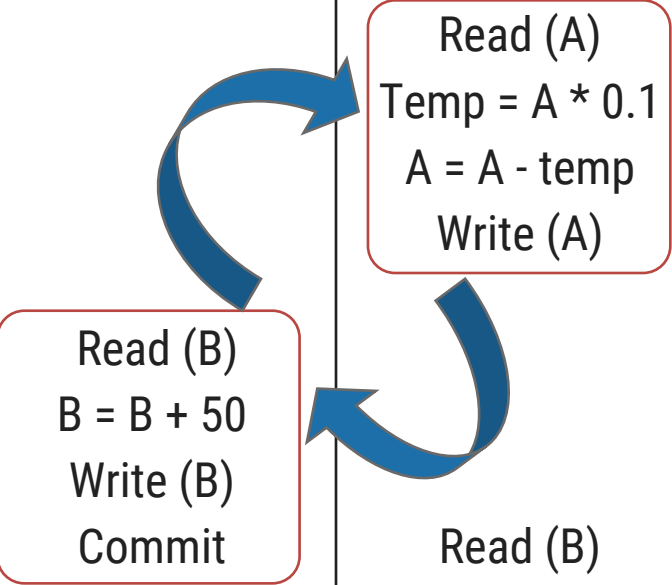
T_i	T_j
	write(Q)
write(Q)	

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 serializable schedule.

Conflict serializability (Example)

T1	T2
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)	
Commit	
	Read (B)
	B = B + temp
	Write (B)
	Commit

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

Conflict serializability (Example)

- Example of a **schedule that is not conflict serializable**:

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

- We are **unable to swap instructions** in the above schedule to obtain either the serial schedule <T1, T2>, or the serial schedule <T2, T1>.

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.

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)	
		Read (A)
Write (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 serializable example

- If a schedule is view equivalent to its serial schedule then the given schedule is said to be view serializable.

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)

- **S2 is the serial schedule of S1**. If we can **prove that they are view equivalent** then we can says that **given schedule S1 is view serializable**.

View serializable example (Initial Read)

Non-Serial Schedule (S1)		Serial Schedule (S2)	
T1	**T2**	**T1**	**T2**
Read (A)		Read (A)	
Write (A)		Write (A)	
	Read (A)	Read (B)	
	Write (A)	Write (B)	
Read (B)			Read (A)
Write (B)			Write (A)
	Read (B)		Read (B)
	Write (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.

View serializable example (Updated Read)

Non-Serial Schedule (S1)		Serial Schedule (S2)	
T1	**T2**	**T1**	**T2**
Read (A)		Read (A)	
Write (A)		Write (A)	
	Read (A)	Read (B)	
	Write (A)	Write (B)	
Read (B)			Read (A)
Write (B)			Write (A)
	Read (B)		Read (B)
	Write (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.

View serializable example (Final Write)

Non-Serial Schedule (S1)		Serial Schedule (S2)	
T1	**T2**	**T1**	**T2**
Read (A)		Read (A)	
Write (A)		Write (A)	
	Read (A)	Read (B)	
	Write (A)	Write (B)	
Read (B)			Read (A)
Write (B)			Write (A)
	Read (B)		Read (B)
	Write (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)		Serial Schedule (S2)	
T1	**T2**	**T1**	**T2**
Read (A)		Read (A)	
Write (A)		Write (A)	
	Read (A)	Read (B)	
	Write (A)	Write (B)	
Read (B)			Read (A)
Write (B)			Write (A)
	Read (B)		Read (B)
	Write (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**.

Two phase commit protocol

- 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

Two phase commit protocol
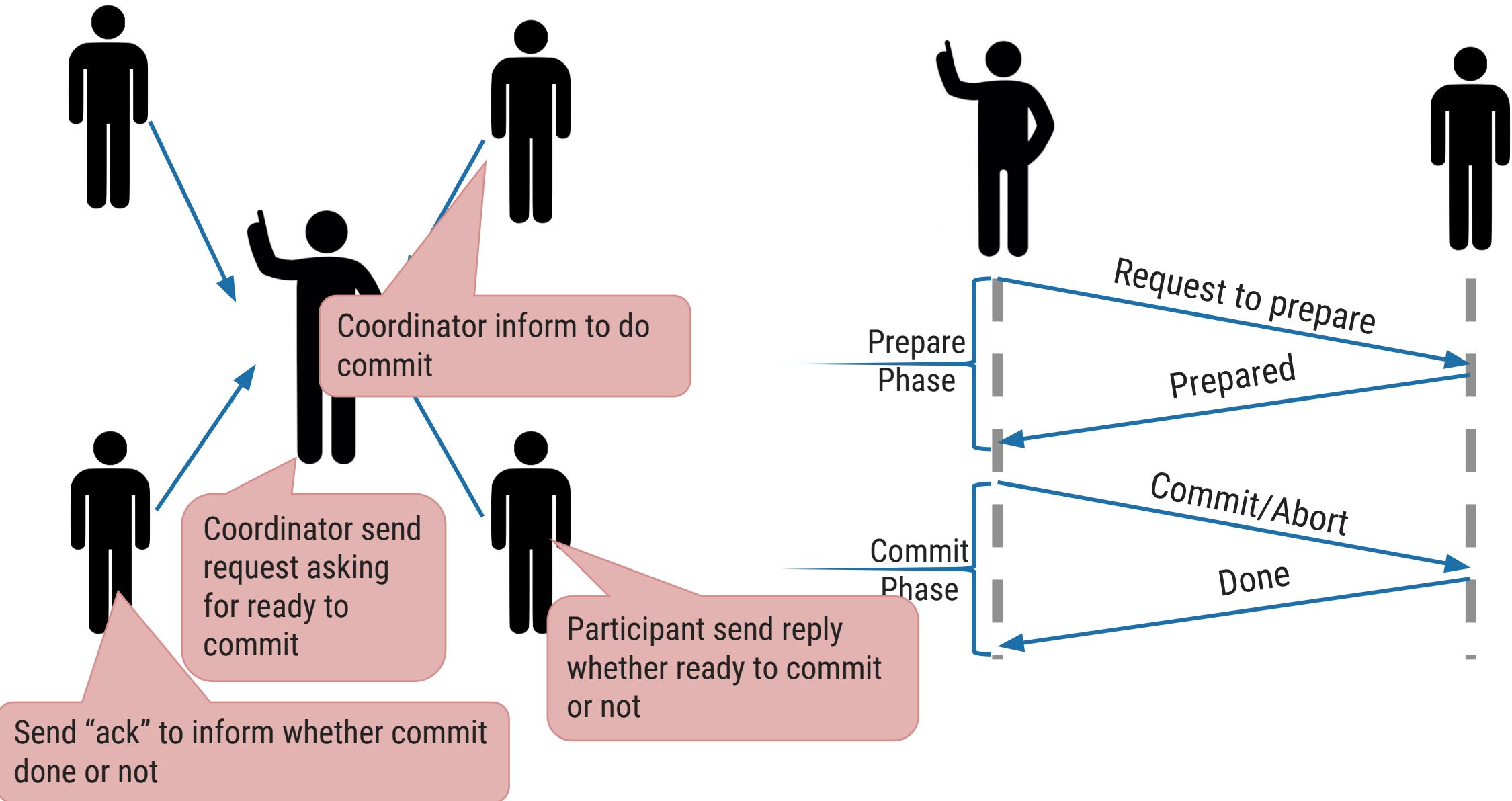
Two phase commit protocol Commit Request Phase (Obtaining Decision)

 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**.

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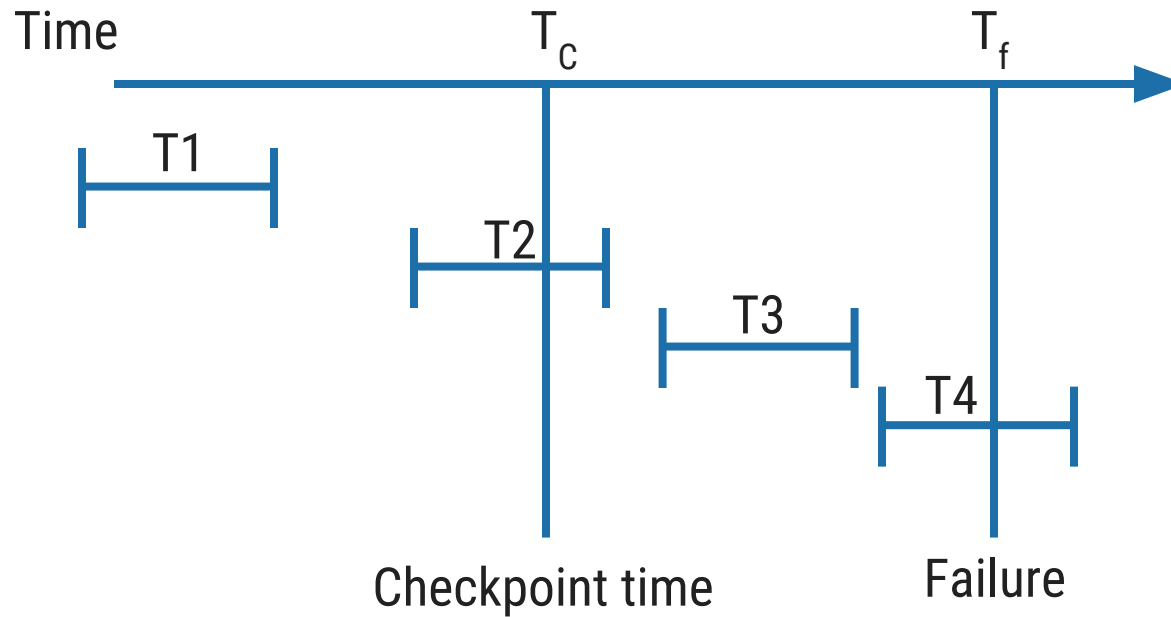
- At failure time:
  - **Ignore the transaction T1** as it has already been committed before checkpoint.
  - **Redo transaction T2 and T3** as they are active after checkpoint and are committed before failure.
  - **Undo transaction T4** as it is active after checkpoint and has not committed.

Page table structure

Page Table
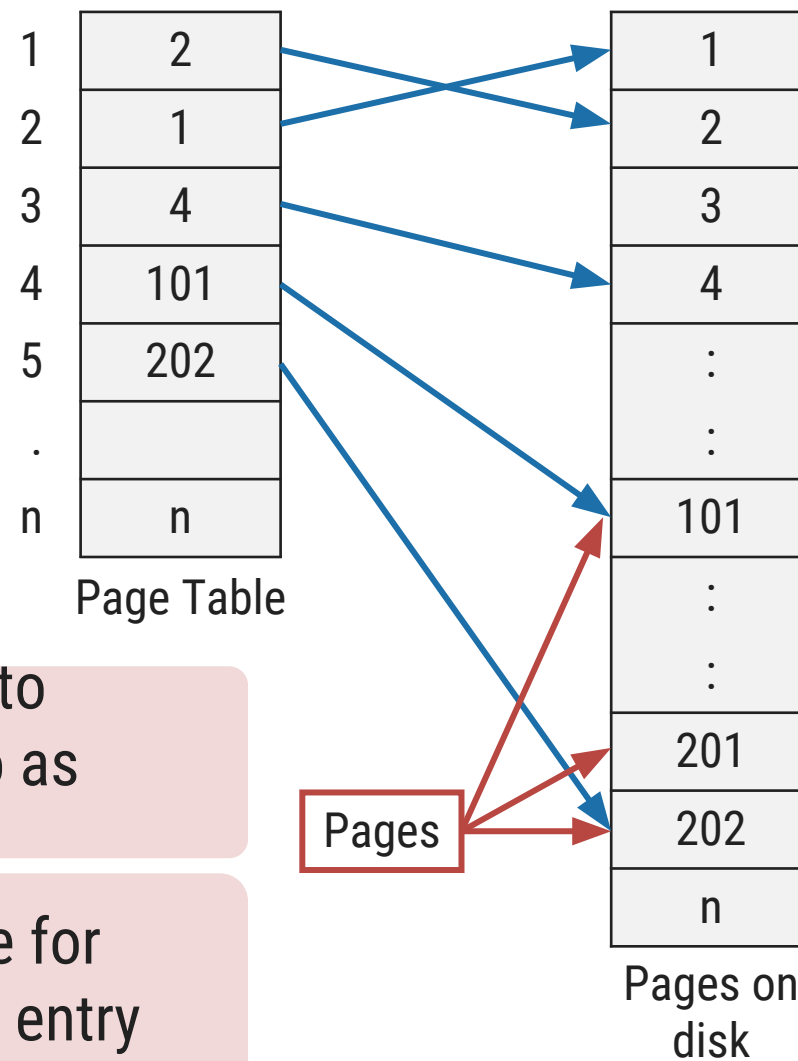
Pages on disk

The database is partitioned into fixed-length blocks referred to as **PAGES**.

Page table has n entries – one for each database page and each entry contain pointer to a page on disk.
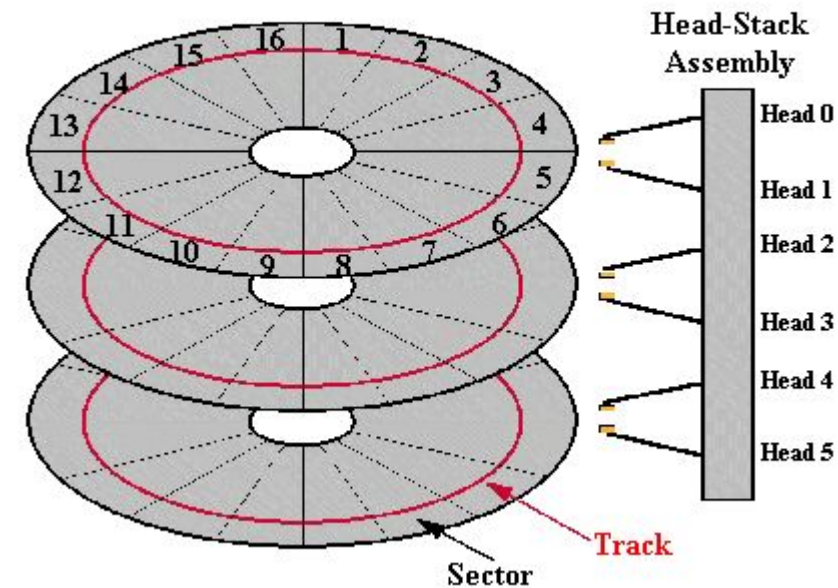
Shadow paging technique

- Shadow paging is an alternative to log-based recovery.

- This scheme is **useful if transactions execute serially**.

- It **maintain two page** tables during the lifetime of a transaction
    - current page table
    - shadow page table

- **Shadow page table** is **stored on non-volatile storage**.

- When a **transaction starts, both the page tables are identical**. Only **current page table is updated for data item accesses (changed) during execution of the transaction**.

- **Shadow page table is never modified** during execution of transaction.

Shadow paging technique
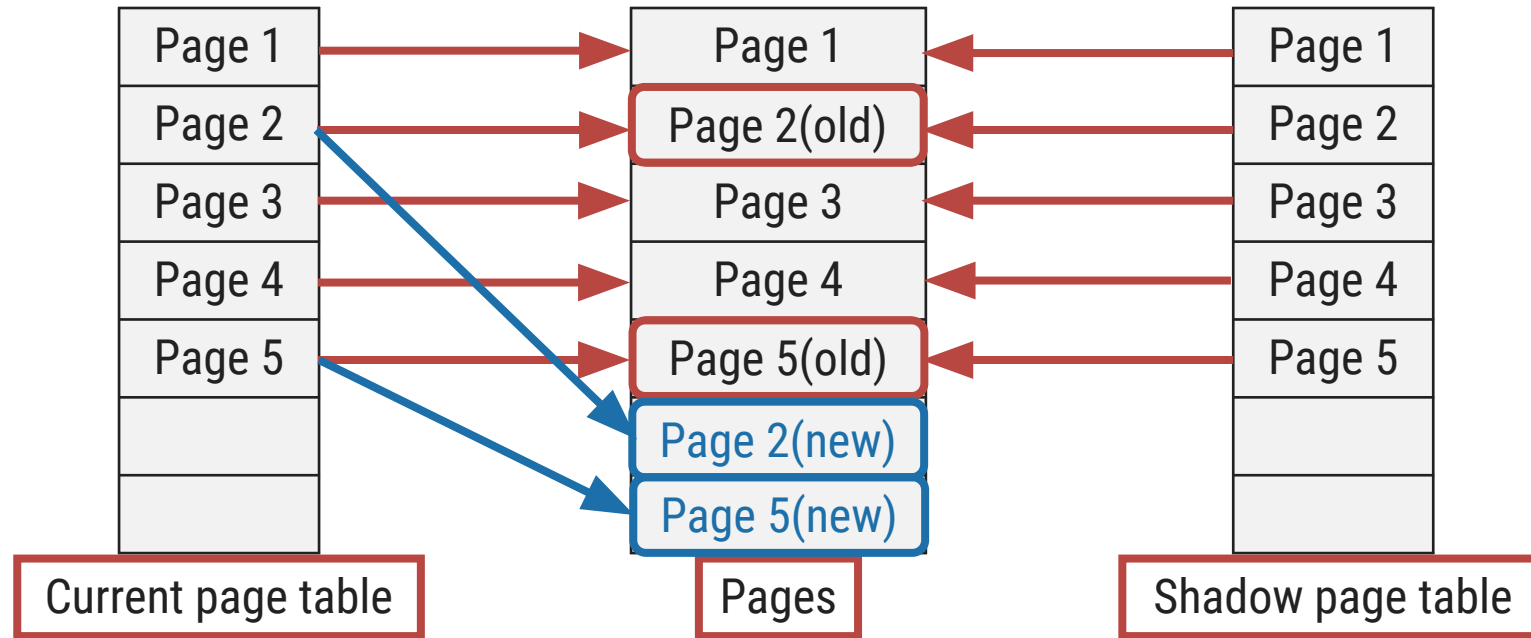
- Whenever any page is updated first time
  1. A copy of this page is made onto an unused page
  2. The current page table is then made to point to the copy
  3. The update is performed on the copy

☐ Two pages - **page 2 & 5 - are affected by a transaction and copied to new physical pages**. The **current page table points to these pages**.

☐ The **shadow page table continues to point to old pages which are not changed by the transaction**. So, this table and pages are used for undoing the transaction.

Shadow paging technique

- When **transaction start, both the page tables are identical**.
- The **shadow page table is never changed** over the duration of the transaction.
- The **current page table will be changed when a transaction performs a write operation**.
- All **input and output operations use the current page table**.
- Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page
  - The current page table is then made to point to the copy
  - The update is performed on the copy
- When the **transaction completes,** all the **modifications which are done by transaction which are present in current page table are transferred to shadow page table**.
- When the **transaction fails,** the **shadow page table are transferred to current page table**.

What is concurrency?

- Concurrency is the **ability of a database to allow multiple (more than one) users to access data at the same time**.

- Three problems due to concurrency
    - Lost update problem
    - Dirty read problem
    - Incorrect retrieval problem

Lost update problem

☐ This problem indicate that if **two transactions T1 and T2 both read the same data and update it then effect of first update will be overwritten by the second update**.

☐ How to **avoid**: A transaction **T2 must not update the data item (X) until the transaction T1 can commit** data item (X).

X=100

T1	Time	T2
---	T0	---
Read X	T1	---
---	T2	Read X
Update X X=75	T3	---
---	T4	Update X X=50
---	T5	---

Dirty read problem

- The dirty read arises when **one transaction update some item and then fails** due to some reason. This **updated item is retrieved by another transaction before it is changed back to the original value**.

- How to **avoid**: A transaction **T1 must not read the data item (X) until the transaction T2 can commit** data item (X).

X=100

T1	Time	T2
---	T0	---
---	T1	Update X X=50
Read X	T2	---
---	T3	Rollback
---	T4	---

Incorrect retrieval problem

- The inconsistent retrieval problem arises when **one transaction retrieves data to use in some operation but before it can use this data another transaction updates that data and commits**.

- Through this change will be hidden from first transaction and it will continue to use previous retrieved data. This problem is also known as inconsistent analysis problem.

- How to **avoid**: A **transaction T2 must not read or update data item (X) until the transaction T1 can commit** data item (X).

Balance (A=200, B=250, C=150)		
T1	**Time**	**T2**
Read (A) **Sum ← 200**	T1	---
Read (B) **Sum ← Sum + 250 = 450**	T2	---
---	T3	Read (C)
---	T4	Update (C) 150 → 150 – 50 = 100
---	T5	Read (A)
---	T6	Update (A) 200 → 200 + 50 = 250
---	T7	COMMIT
Read (C) **Sum ←Sum + 100 = 550**	T8	---

What is lock?
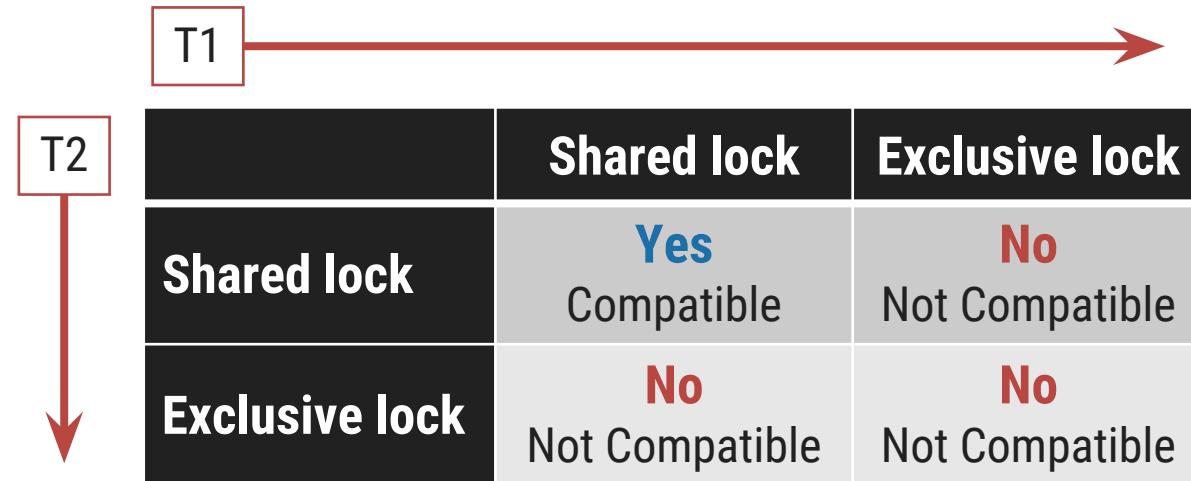
- A lock is a **variable associated with data item to control concurrent access to that data item**.

Locking is a strategy that is used to prevent such concurrent access of data.

Lock variable

1

Database
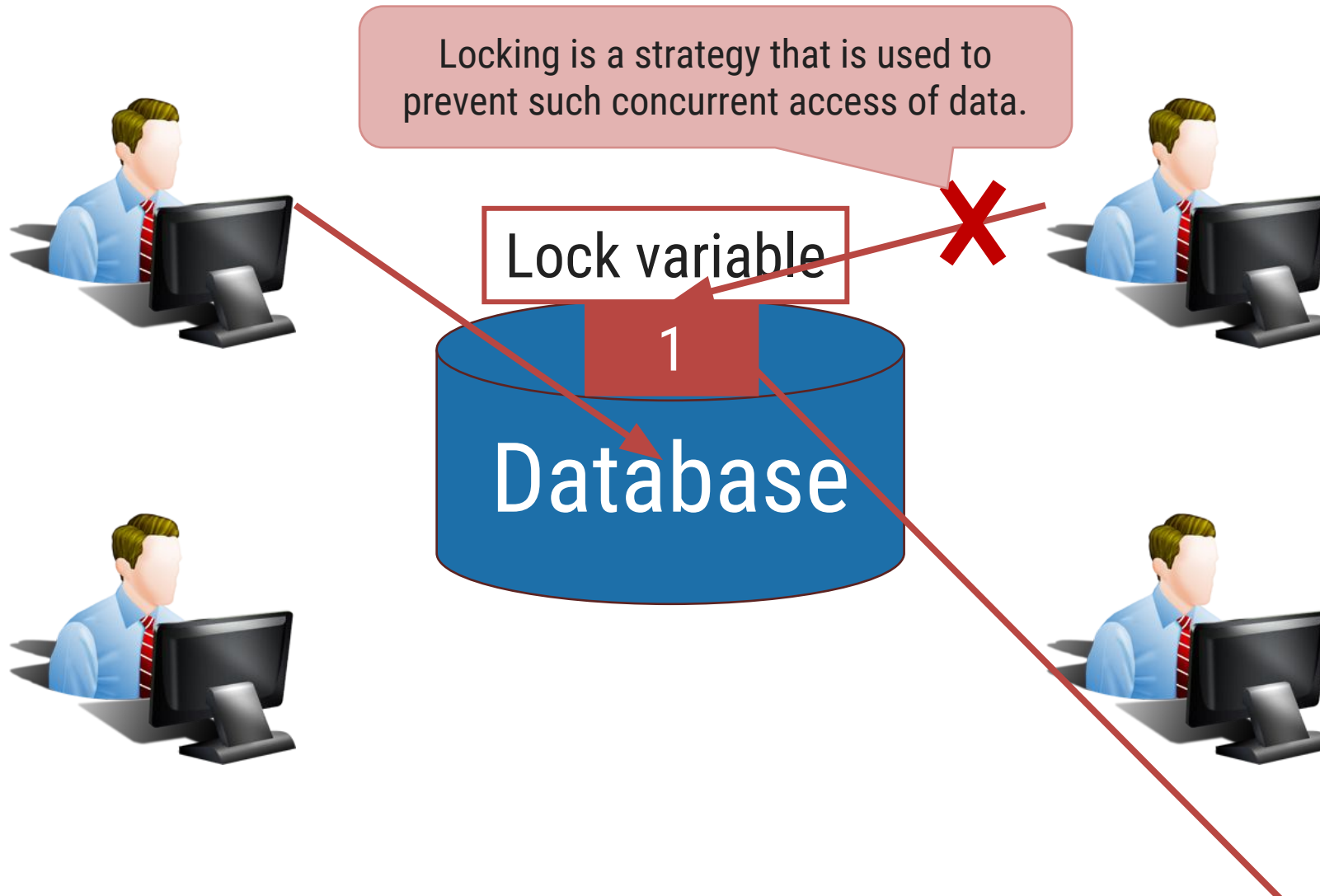
Lock based protocol

- Data items can be locked in two modes :
  - **Shared (S) mode**: When we take this lock **we can just read the item but cannot write**.
  - **Exclusive (X) mode**: When we take this lock **we can read as well as write the item**.
- Lock-compatibility matrix

	Shared lock	Exclusive lock
Shared lock	Yes Compatible	No Not Compatible
Exclusive lock	No Not Compatible	No Not Compatible

- 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**.
- 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.
- **Any number of transactions can hold shared locks** on an item, but **if any transaction holds an exclusive on the item no other transaction can hold any lock** on the item.

Lock based protocol

☐ This locking protocol divides transaction execution phase into three parts:

1. When transaction starts executing, **create a list of data items on which they need locks** and **requests the system for all the locks it needs**.

2. Where the **transaction acquires all locks** and **no other lock is required**. **Transaction keeps executing its operation**.

3. As soon as the **transaction releases its first lock, the third phase starts**. In this phase a **transaction cannot demand for any lock but only releases the acquired locks**.
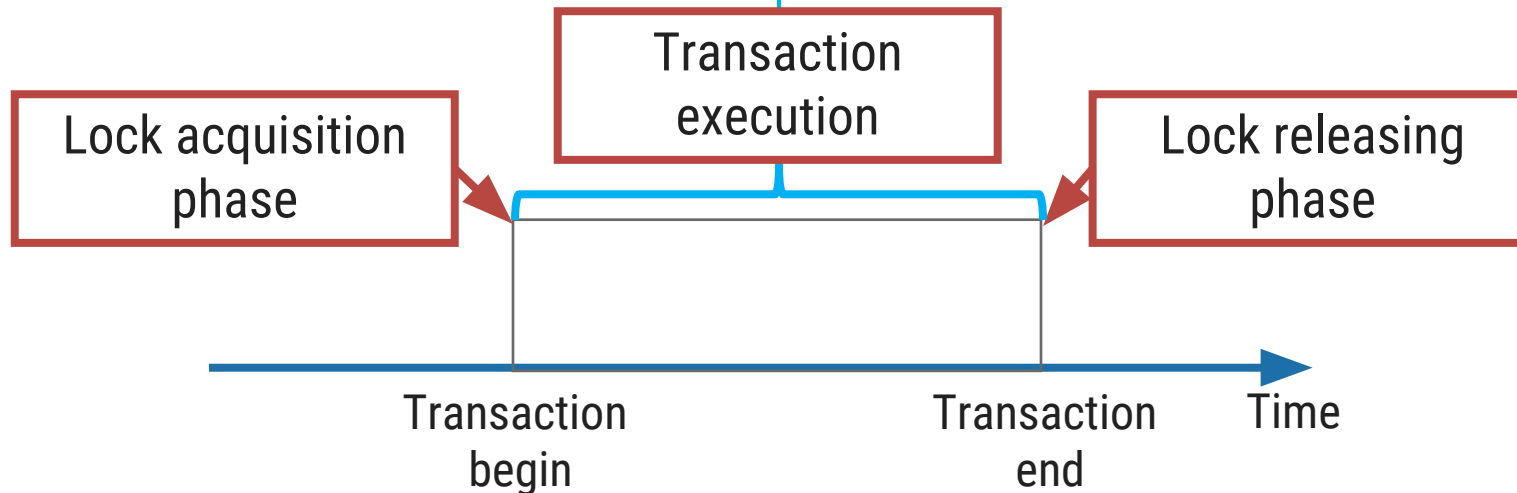
Two phase locking protocol

- This protocol works in two phases,

1. Growing Phase
   - In this phase a **transaction obtains locks**, but **can not release any lock**.
   - When a transaction takes the final lock is called lock point.

2. Shrinking Phase
   - In this phase a **transaction can release locks**, but **can not obtain any lock**.
   - The **transaction enters the shrinking phase as soon as it releases the first lock** after crossing the Lock Point.
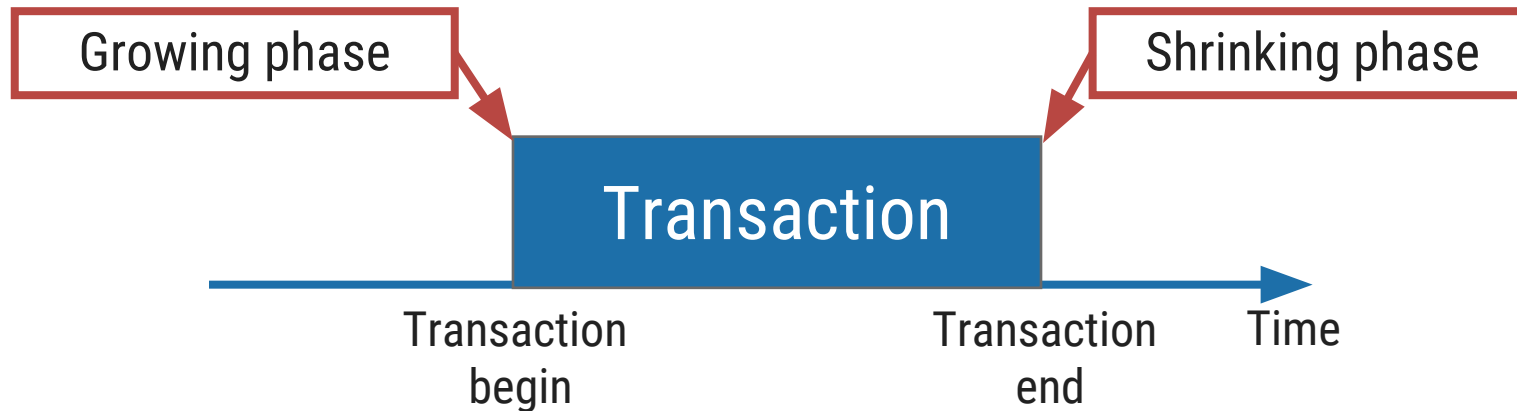
Strict two phase locking protocol V/S Rigorous two phase locking protocol

- Strict two phase locking protocol
  - In this protocol, a **transaction may release all the shared locks after the Lock Point has been reached**, but **it cannot release any of the exclusive locks until the transaction commits or aborts**.
  - It **ensures that if data is being modified by one transaction, then other transaction cannot read it until first transaction commits**.
  - This protocol **solves dirty read problem**.

- Rigorous two phase locking protocol
  - In this protocol, a **transaction is not allowed to release any lock (either shared or exclusive) until it commits**.
  - This means that **until the transaction commits, other transaction can not acquire even a shared lock on a data item on which the uncommitted transaction has a shared lock**.
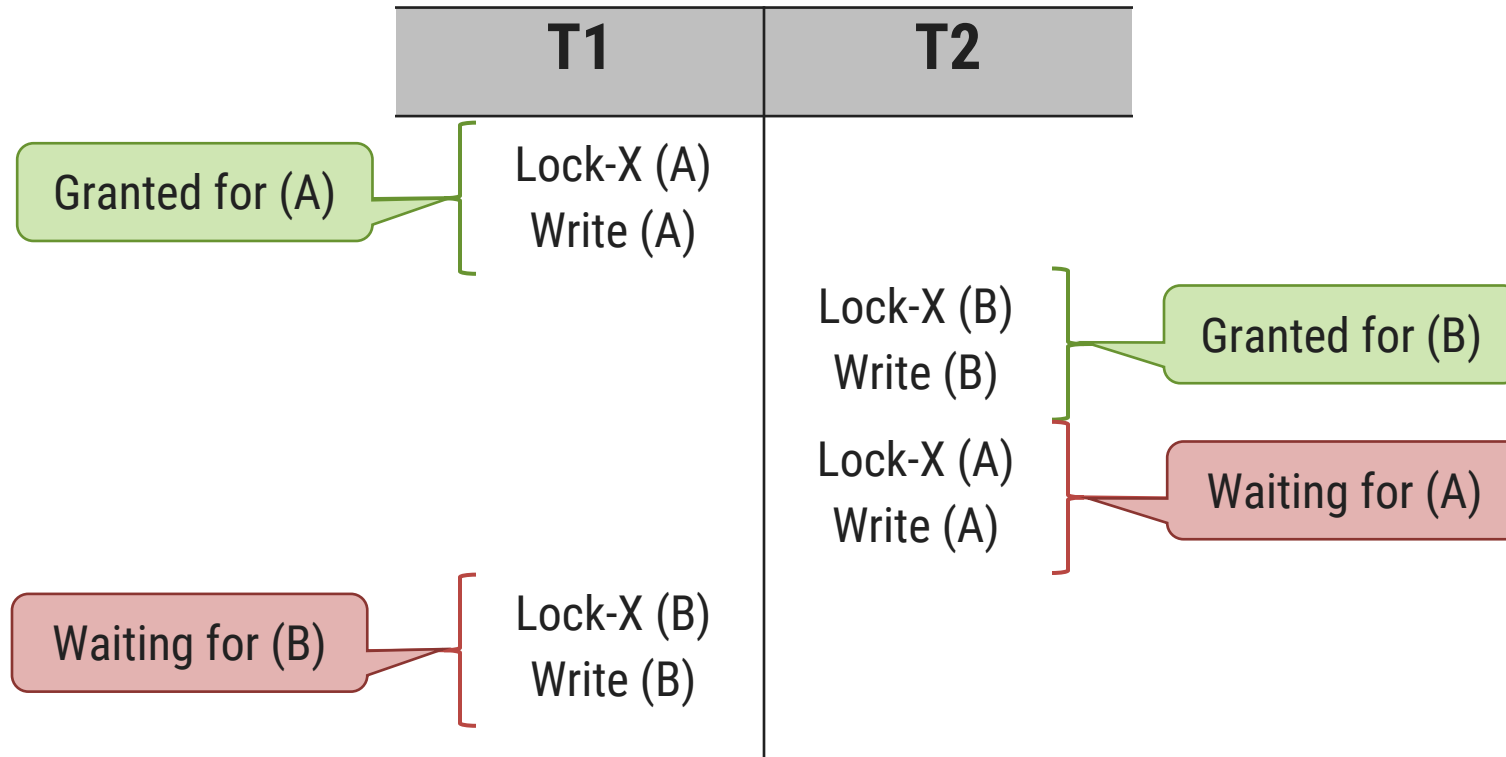
Time stamp based protocol

- This protocol **uses either system time or logical counter** to be used as a time-stamp.

- Every **transaction has a time-stamp** associated with it and the **ordering is determined by the age of the transaction**.

- A transaction 'T1' created at 0002 clock time would be older than all other transaction, which come after it.

- For example, any transaction 'T2' entering the system at 0004 is two seconds younger than transaction 'T1' and priority is given to the older one.

- In addition, **every data item is given the latest read and write time-stamp**. This lets the system know, when last read and write operations was made on the data item.

- This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

    - Time-stamp of Transaction Ti is denoted as TS(Ti).

    - Read time-stamp of data-item X is denoted by R-timestamp(X).

    - Write time-stamp of data-item X is denoted by W-timestamp(X).

Time stamp ordering protocol

☐ This is the responsibility of the protocol system that the conflicting pair of tasks should be executed according to the timestamp values of the transactions.

- ☐ Time-stamp of Transaction Ti is denoted as TS(Ti).
- ☐ Read time-stamp of data-item X is denoted by R-timestamp(X).
- ☐ Write time-stamp of data-item X is denoted by W-timestamp(X).

☐ Timestamp ordering protocol works as follows:

- ☐ If a transaction Ti issues read(X) operation:
  - ▪ If TS(Ti) < W-timestamp(X)
    - • Operation rejected.
  - ▪ If TS(Ti) >= W-timestamp(X)
    - • Operation executed.
- ☐ If a transaction Ti issues write(X) operation:
  - ▪ If TS(Ti) < R-timestamp(X)
    - • Operation rejected.
  - ▪ If TS(Ti) < W-timestamp(X)
    - • Operation rejected and Ti rolled back.
  - ▪ Otherwise, operation executed.

What is deadlock?

- Consider the following two transactions:

| T1	T2
Lock-X (A) Write (A) — **Granted for (A)**	
	Lock-X (B) Write (B) — **Granted for (B)**
	Lock-X (A) Write (A) — **Waiting for (A)**
Waiting for (B) — Lock-X (B) Write (B)	

- 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**.
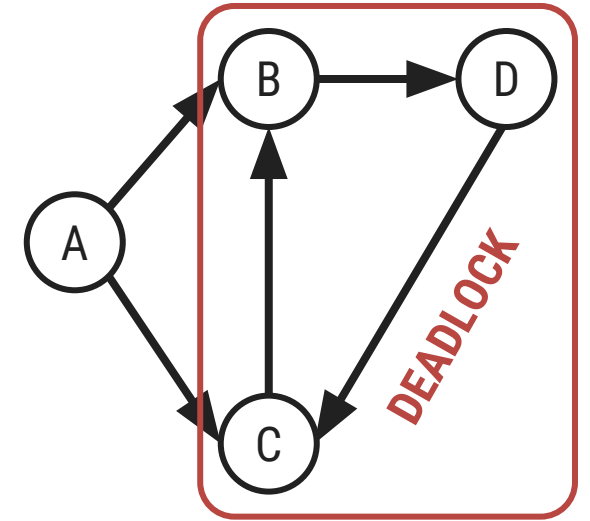
Deadlock recovery

- 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
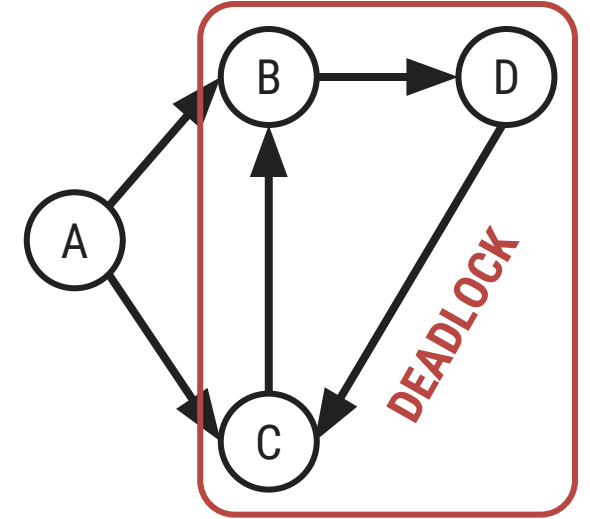
Deadlock prevention

- 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**.

Deadlock prevention

 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**

Deadlock prevention

 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.

Questions asked 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 database 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.

Thank You...!!