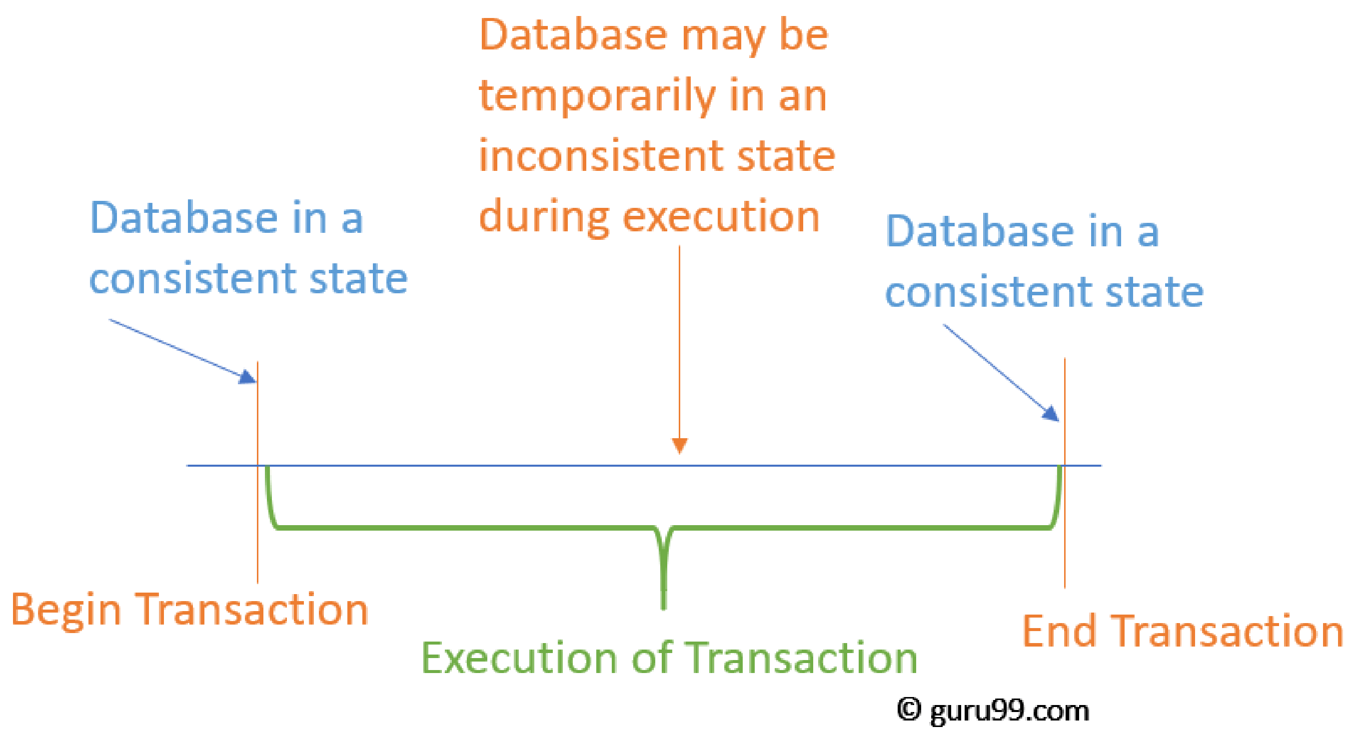
Transaction Management

Transaction Concept

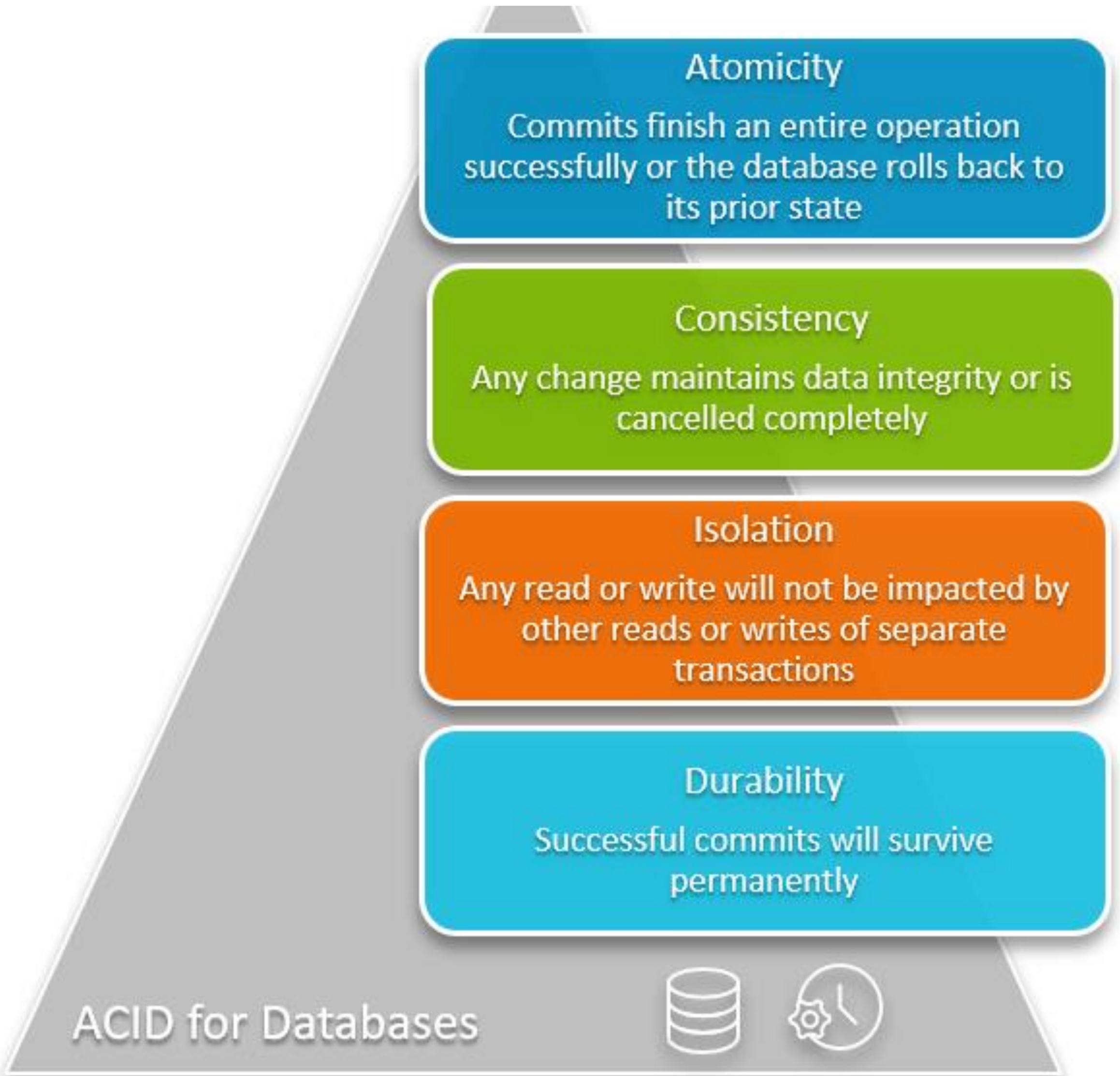
- A **transaction**
  - is a *unit* of program execution that accesses and possibly updates various data items.
  - must see a consistent database.
- Multiple transactions can execute in parallel.

- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.



- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

INTEGRITY PRESERVATION - TRANSACTION



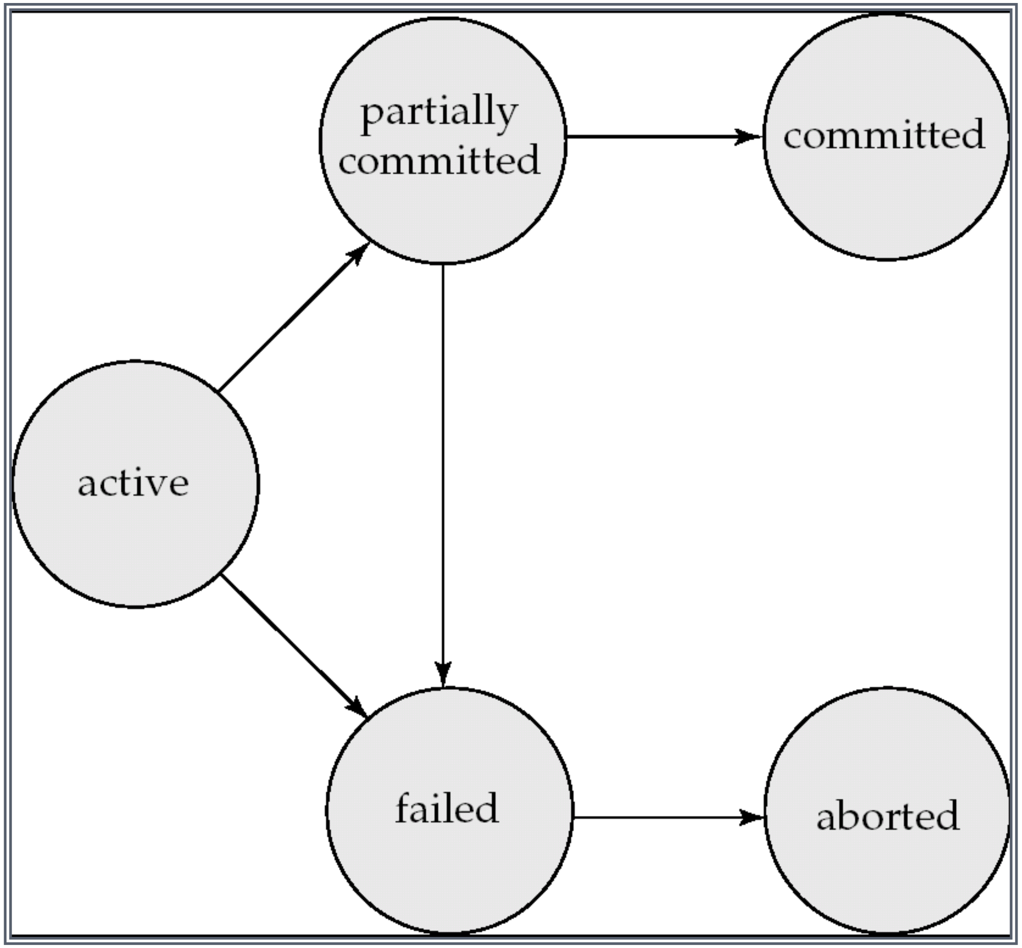
ACID Properties

To preserve the 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 run by itself 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.

Transaction States



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; can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

Example of Fund Transfer

- 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)

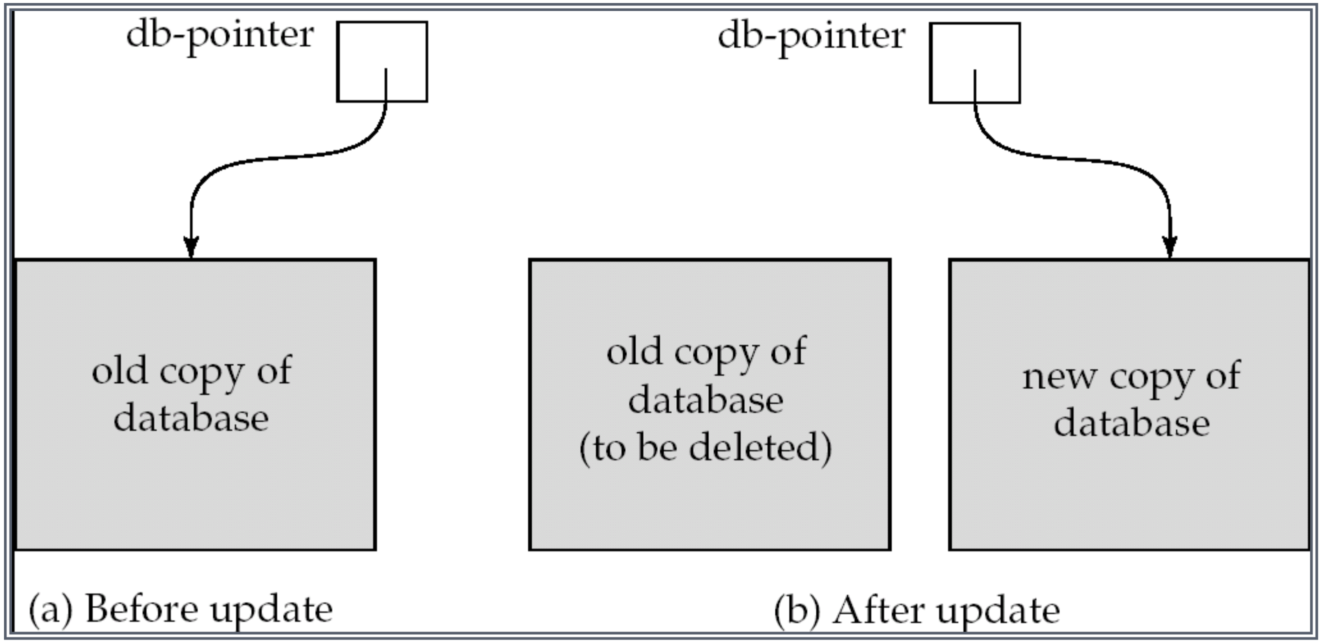
- **Atomicity requirement**
- **Consistency requirement**
- **Isolation requirement**
- **Durability requirement**

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.

Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:



- Assumes disks do not fail
- Useful for text editors, but
  - extremely inefficient for large databases
  - Does not handle concurrent transactions

CONCURRENT EXECUTIONS

- Multiple transactions are allowed to run concurrently in the system.
- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
  - **reduced average response time** for transactions

Schedules

- **Schedule** – a sequences of instructions that specify the order of concurrent transactions are executed

- COMMIT
- FAIL / ABORT

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule 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)

Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule 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 Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of *(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)

CONCURRENT EXECUTIONS OF TRANSACTIONS AND RELATED PROBLEMS

- Dirty Read Problem
- Lost Update Problem
- Unrepeatable Read Problem
- Phantom Read Problem
- Incorrect Aggregate Problem

W R Conflicts
DIRTY READ !!!

- Reading Uncommitted Data
- Temporary Update Problem

T1	T2
R(A)	
A=A-2;	
W(A)	
	R(A)
	A=A-5
	W(A)
	Commit
R(A)	
W(A)	

WW CONFLICT
LOST UPDATE – BLIND WRITE

T1	T2
R(A)	
A=A+5;	
W(A)	
	A=A+10
	W(A)
	Commit
Commit	

R W Conflicts
UNREPEATABLE READS

T1	T2
R(A)	
	R(A)
A=A-5	
W(A)	
Commit	
	R(A)

Conflicts
PHANTOM READ

T1	T2
R(A)	
	R(A)
Delete(A)	
	R(A)

CONFLICTS
INCORRECT SUMMARY

T1	T2
	sum = 0
	read_item(A)
	sum = sum + A
read_item(X)	
X = X - N	
write_item(X)	
	read_item(X)
	sum = sum + X
	read_item(Y)
	sum = sum + Y
read_item(Y)	
Y = Y + N	
write_item(Y)	

Anomalies with interleaved execution

- *R(A) R(A)*
- W(A) R(A)
- R(A) W(A)
- W(A) W(A)

- **They belong to different transactions**
- **They operate on the same data item**
- **At least one of them is a write operation**

CONFLICTING PAIRS !!!

Non – Conflict Pairs
R (A) R(A)
R(B) R(A)
W(B) R(A)
R(B) W(A)
W(A) R(B)

Conflict Pairs
$W_1(A)$ $R_2(A)$
$R_1(A)$ $W_2(A)$
$W_1(A)$ $W_2(A)$

SWAP NON -CONFLICT PAIRS

ARE S AND S^1 CONFLICT EQUIVALENT?

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

Serializability

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.

1. **Conflict Serializability**
2. **View 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

SWAP NON -CONFLICT PAIRS

Conflict Serializability

- Check if conflict serializable?

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

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

SWAP NON -CONFLICT PAIRS

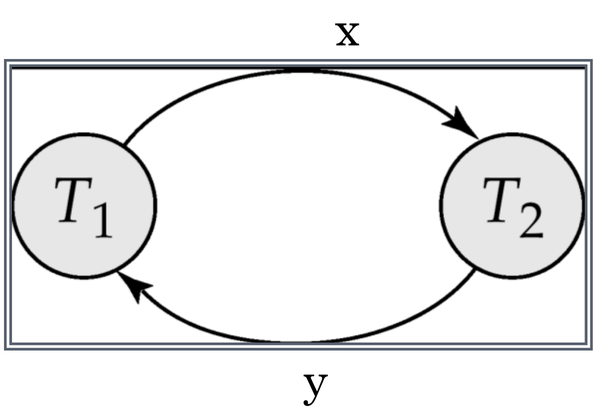
Conflict Serializability

Check if conflict serializable?

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

Testing for Serializability

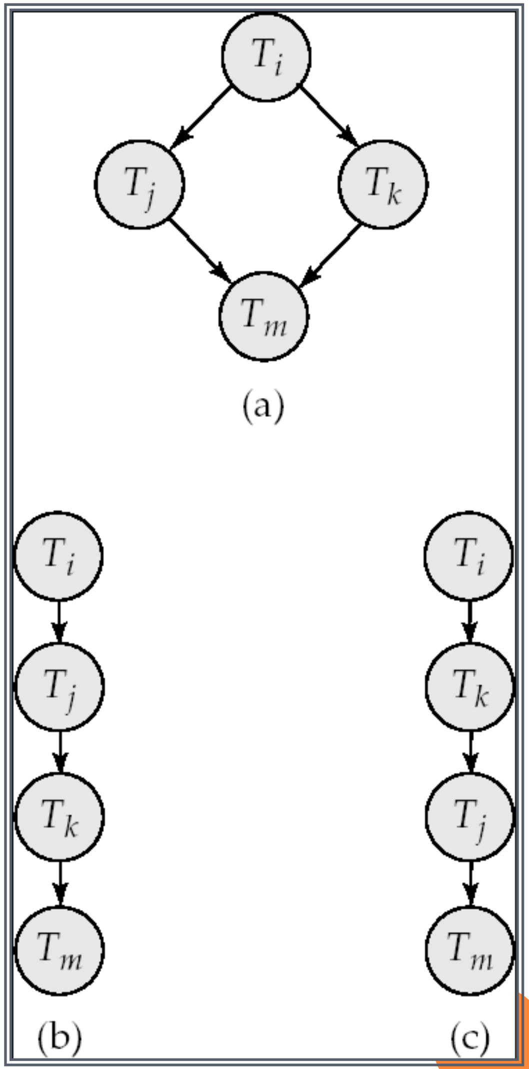
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** → a direct graph where the vertices are the transactions
- Draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- Label the arc by the item that was accessed.



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).***

Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is **acyclic.**
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.



Draw edges b/w conflict pairs

Test for Conflict Serializability !

T1	T2	T3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

Draw edges b/w conflict pairs

Test for Conflict Serializability !

T4	T5	T6
R(x)		
	W(x)	
W(x)		
		W(x)

View Serializability

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Every view serializable schedule that is not conflict serializable has **blind writes.**

LETS CHECK !

- Below is a schedule which is view-serializable but *not* conflict serializable.

T4	T5	T6
R(x)		
	W(x)	
W(x)		
		W(x)

TESTING FOR VIEW SERIALIZABILITY

- S and S' are **view equivalent** if the following three conditions are met:

1. **Initial Read :** For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.
2. **Intermediate Read:** For each data item Q if transaction T_i executes **read**(Q) in schedule S, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j .
3. **Final Write :** For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S must perform the final **write**(Q) operation in schedule S'.

View equivalence is also based purely on **reads and writes** alone.

Initial Read
Intermediate Read
Final Write

TEST FOR VIEW SERIALIZABILITY !

T1	T2
R(a)	
W(a)	
	R(a)
	W(a)
R(b)	
W(b)	
	R(b)
	W(b)

POSSIBILITIES ???

Initial Read
Intermediate Read
Final Write

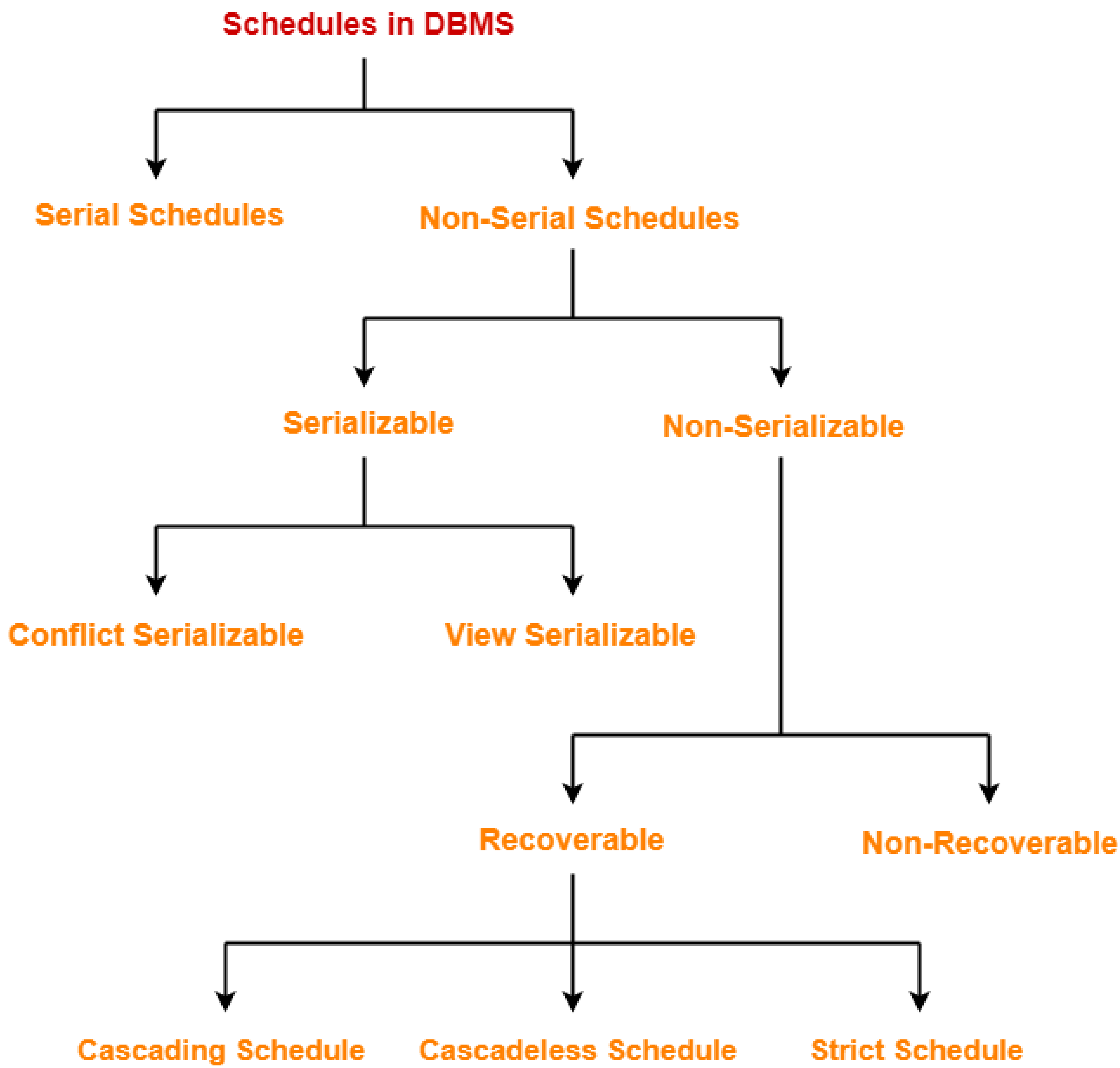
Test for VIEW Serializability !

S

T1	T2
R(a)	
W(a)	
	R(a)
	W(a)
R(b)	
W(b)	
	R(b)
	W(b)

S^1

T1	T2
R(a)	
W(a)	
R(b)	
W(b)	
	R(a)
	W(a)
	R(b)
	W(b)


Schedules in DBMS
Serial Schedules
Non-Serial Schedules
Serializable
Non-Serializable
Conflict Serializable
View Serializable
Recoverable
Non-Recoverable
Cascading Schedule
Cascadeless Schedule
Strict Schedule

SYSREM RECOVERY

- Failures
  - Transaction Failures
    - Logical error
    - System error
  - System Crash

Non- recoverable Vs Recoverable schedules

P

T1	T2
R(a)	
A=a-10	
W(a)	
	R(a)
	A=a-20
	W(a)

Q

T1	T2
R(a)	
A=a-10	
W(a)	
Commit	
	R(a)
	A=a-20
	W(a)

Recoverable Schedules

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.

CASCADING VS CASCADELESS SCHEDULE

T1	T2	T3
R(a)		
a=a+10		
W(a)		
	R(a)	
	a=a+10	
	W(a)	
		R(a)
		a=a+10
		W(a)

CASCADING VS **CASCADELESS** SCHEDULE

T1	T2	T3
R(a)		
a=a+10		
W(a)		
Commit		
	R(a)	
	a=a+10	
	W(a)	
	Commit	
		R(a)
		a=a+10
		W(a)
		Commit

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

LOG BASED RECOVERY

- **The facility to maintain or recover data if any failure may occur in the system**
- **Log == Record**

- **Deferred Database Modification**
- **Immediate Database Modification**

Fields in a log

- **Transaction identifier**
- **Data item**
- **Old value**
- **New value**

- **Types of logs**
  - **<Ti start>**
  - **<Ti commit>**
  - **<Ti abort>**

- New values are stored in the log
- Redo / No undo

DEFERRED DATABASE MODIFICATION / UPDATE

T1
R(A)
A=A+20
W(A)
R(B)
B=B+50
W(B)
Commit

T1
R(A)
A=A+20
W(A)
R(B)
B=B+50
W(B)

- New and old values are stored in the log
- Redo / Undo

IMMEDIATE DATABASE MODIFICATION / UPDATE

T1
R(A)
A=A+20
W(A)
R(B)
B=B+50
W(B)
Commit

T1
R(A)
A=A+20
W(A)
R(B)
B=B+50
W(B)

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
  - 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.

SHARED (S) – READ
EXCLUSIVE(X) – READ & WRITE

T1
R(A)

T2
R(A)
A=A-50
W(A)

Lock-Based Protocols

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

T1	T2
R(A)	
	R(A)
	W(A)

T1	T2
W(A)	
	R(A)
	W(A)

T1	T2
W(A)	
	W(A)

INTENT LOCKING

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time.

PITFALLS IN LOCK BASED PROTOCOLS

- Serialization **may not** be guaranteed
- **May not** be free from non - recoverability
- **May not** be free from Deadlocks
- **May not** be free from Starvation

Serialization may not be guaranteed

T1	T2
R(A) W(A)	
	R(A)
R(B) W(B)	

May not be free from NON- recoverability

T1	T2
R(A) W(A)	
	R(A) … …
*	

May not be free from Deadlocks

- Waiting indefinitely for resources

T1	T2
X(A)	
	X(B)
X(B)	
	X(A)

May not be free from Starvation

T1	T2	T3	T4
	S(A)		
X(A)			

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may
    - obtain locks
    - not release locks
- Phase 2: Shrinking Phase
  - transaction may
    - release locks
    - not obtain locks

Example

T2
R(A)
W(A)
R(B)
R(C)
W(D)
R(D)

Demonstration of 2PL with LOCKPOINTS

T1	T2
S(A)	
	S(A)
X(B)	
U(A)	
	X(D)
U(B)	
	U(A)
	U(D)

TYPES OF 2PL

- Strict 2-PL
- Rigorous 2-PL
- Conservative 2-PL

STRICT 2 PL

- **All Exclusive(X) locks** held by the transaction can be released until *after* the Transaction Commits.

- Helps in
  - Recoverability
  - Cascadeless

RIGOROUS 2 PL

- **All Exclusive(X) and Shared(S) locks** held by the transaction can be released until *after* the Transaction Commits.

Conservative 2PL

- Lock all items it needs then transaction starts execution
  - If any locks can not be obtained, then do not lock anything
- Difficult but deadlock free

- **Transaction concepts, properties of transactions, serializability of transactions, testing for serializability, System recovery, Two- Phase Commit protocol, Recovery and Atomicity, Log-based recovery, concurrent executions of transactions and related problems, Locking mechanism, solution to concurrency related problems, deadlock, , two-phase locking protocol, Isolation, Intent locking**