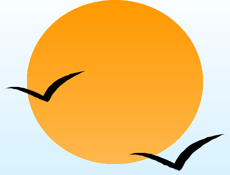
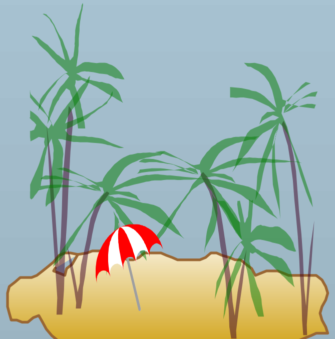


Transactions Processing

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability

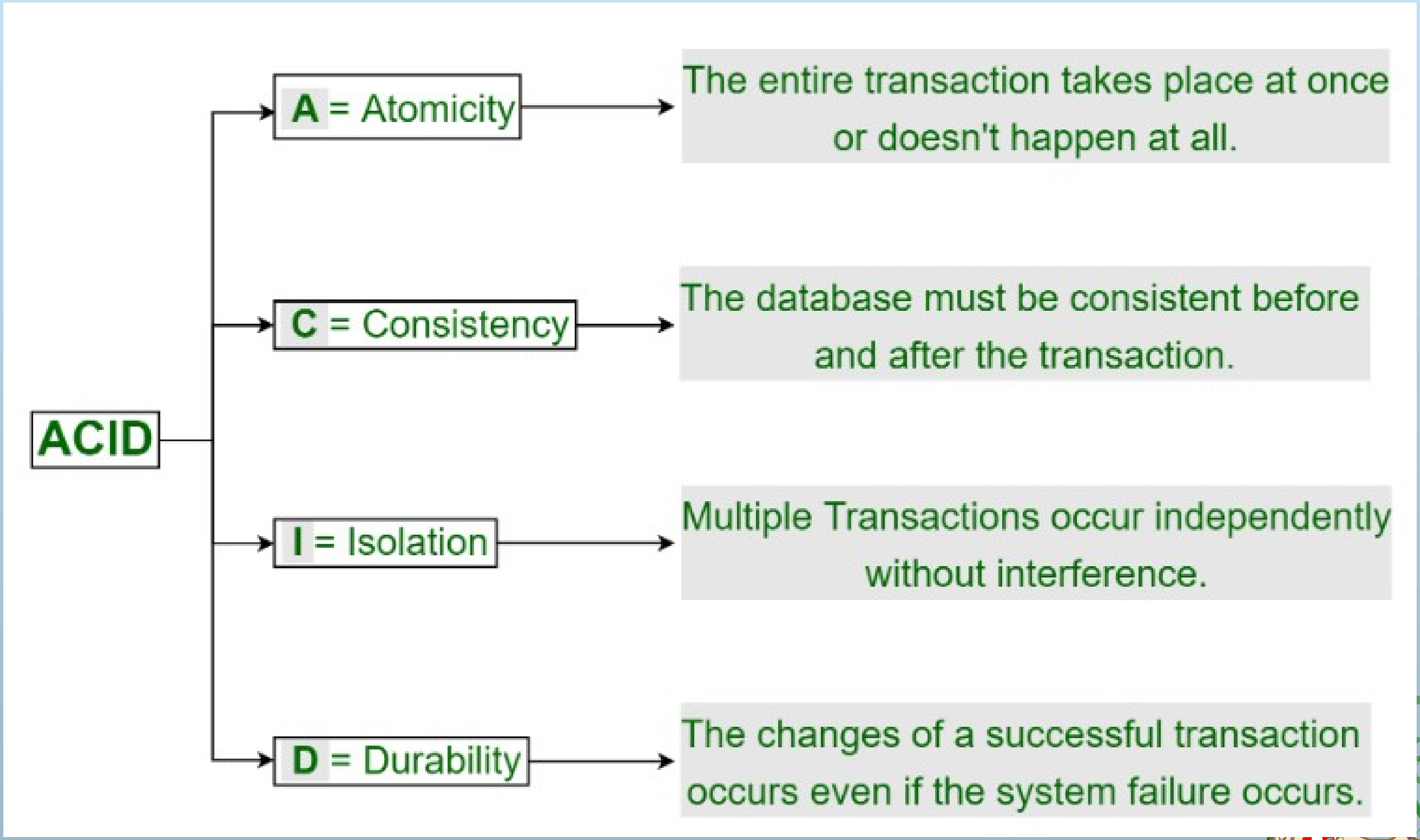


Transaction Concept

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

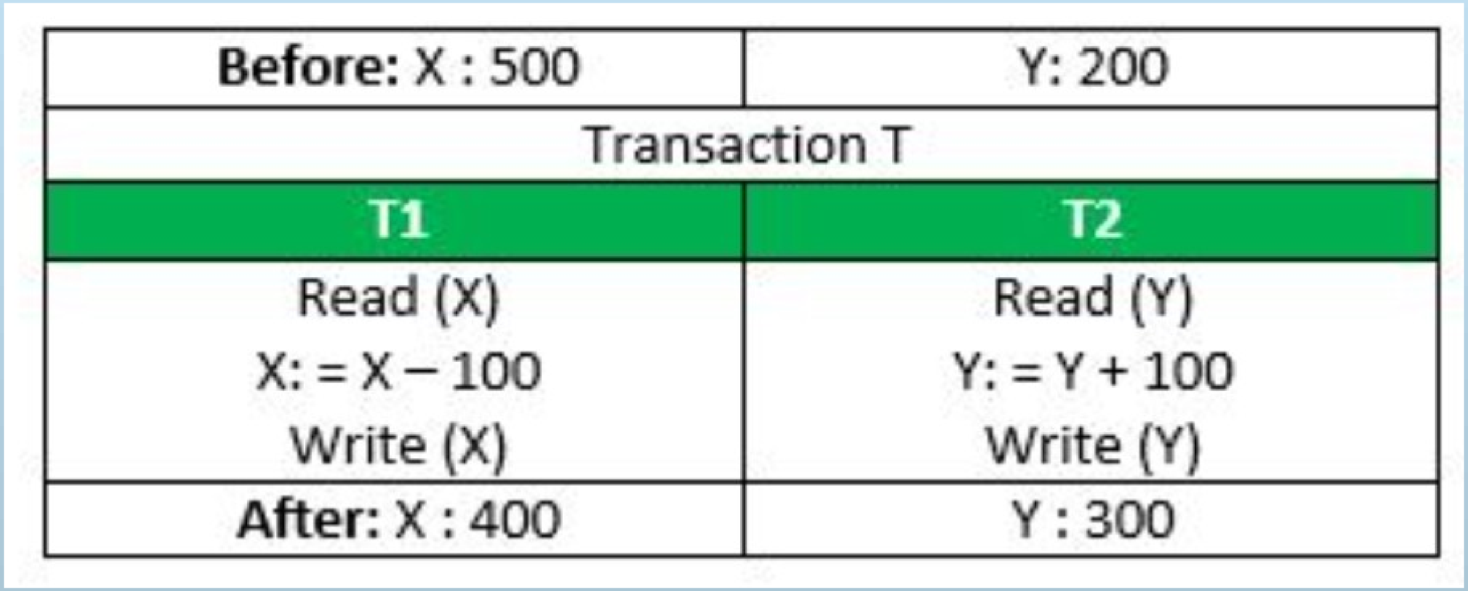
ACID Properties

To preserve integrity of data, the database system must ensure ACID Properties


ACID
A = Atomicity
The entire transaction takes place at once or doesn't happen at all.
C = Consistency
The database must be consistent before and after the transaction.
I = Isolation
Multiple Transactions occur independently without interference.
D = Durability
The changes of a successful transaction occurs even if the system failure occurs.

Example of Fund Transfer

- Transaction to transfer $100 from account X to account Y:



Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X – 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

Example of Fund Transfer

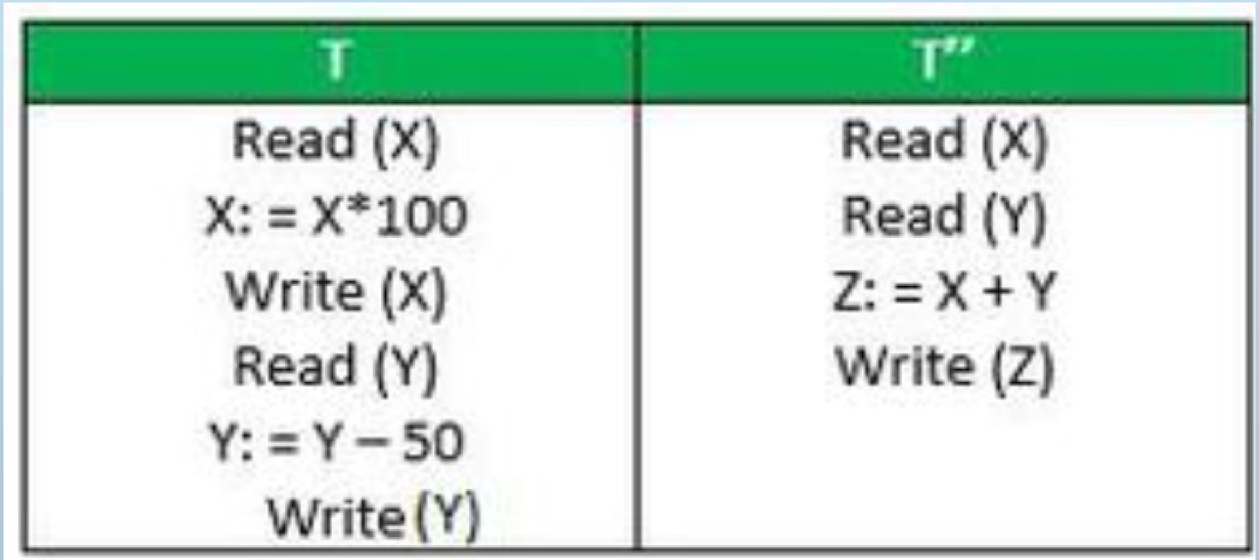
- **Atomicity:** If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state.

- **Consistency:** The total amount before and after the transaction must be maintained.
  Total before T occurs = 500 + 200 = 700.
  Total after T occurs = 400 + 300 = 700.
  Therefore, the database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result, T is incomplete.

- **Durability:** This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

Example of Fund Transfer

- **Isolation:** Let **X**= 500, **Y** = 500.
  Consider two transactions **T** and **T”.**

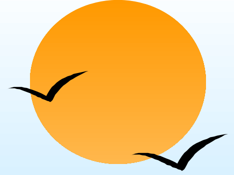


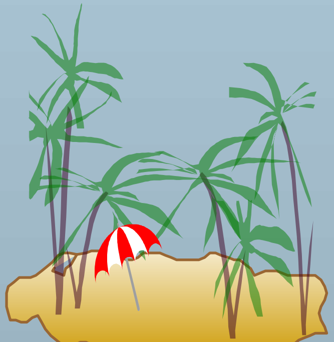
T	T”
Read (X)	Read (X)
X: = X*100	Read (Y)
Write (X)	Z: = X + Y
Read (Y)	Write (Z)
Y: = Y – 50	
Write (Y)	

- Suppose **T** has been executed till **Read (Y)** and then **T”** starts. As a result, interleaving of operations takes place due to which **T”** reads the correct value of **X** but the incorrect value of **Y** and sum computed by
  T”: (X+Y = 50, 000+500=50, 500)
  is thus not consistent with the sum at end of the transaction:
  T: (X+Y = 50, 000 + 450 = 50, 450).
  This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

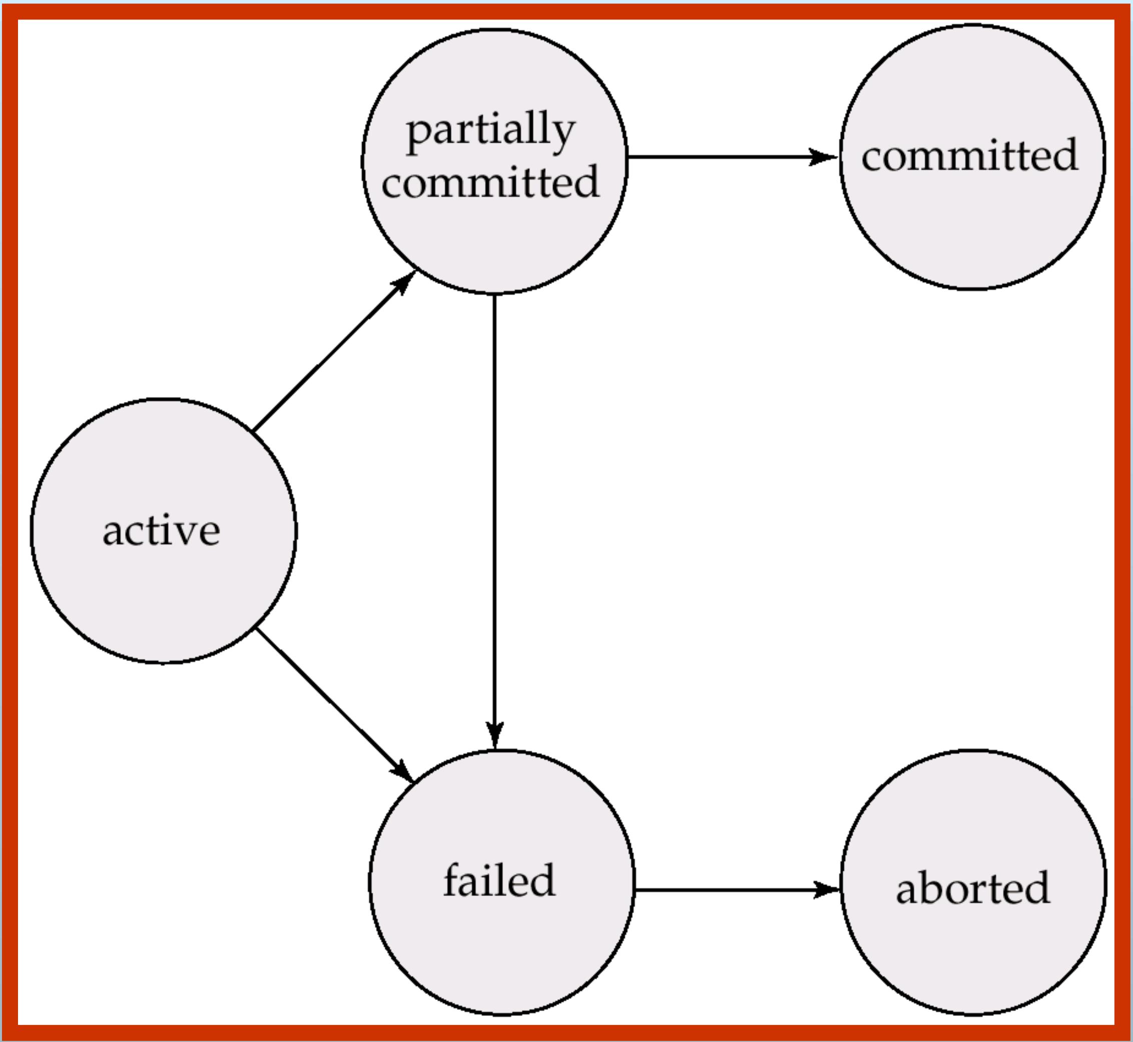
Transaction State

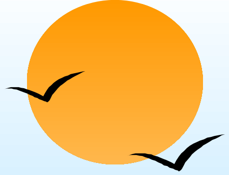
- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the transaction has been rolled back and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





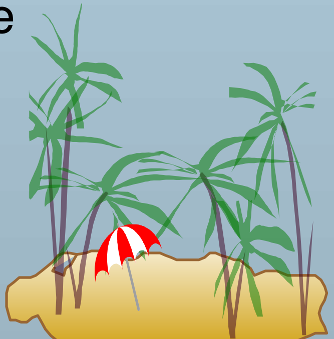
Transaction State (Cont.)





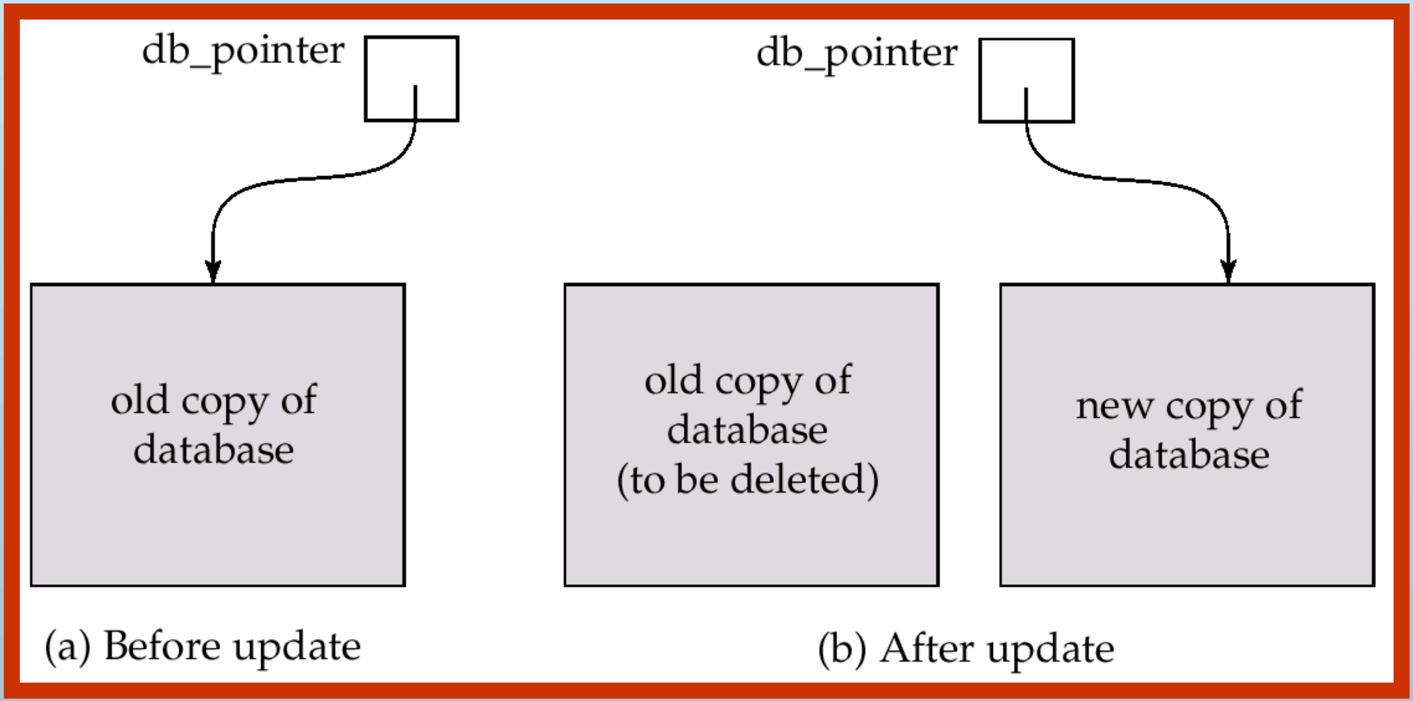
Implementation of Atomicity and Durability

- 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:



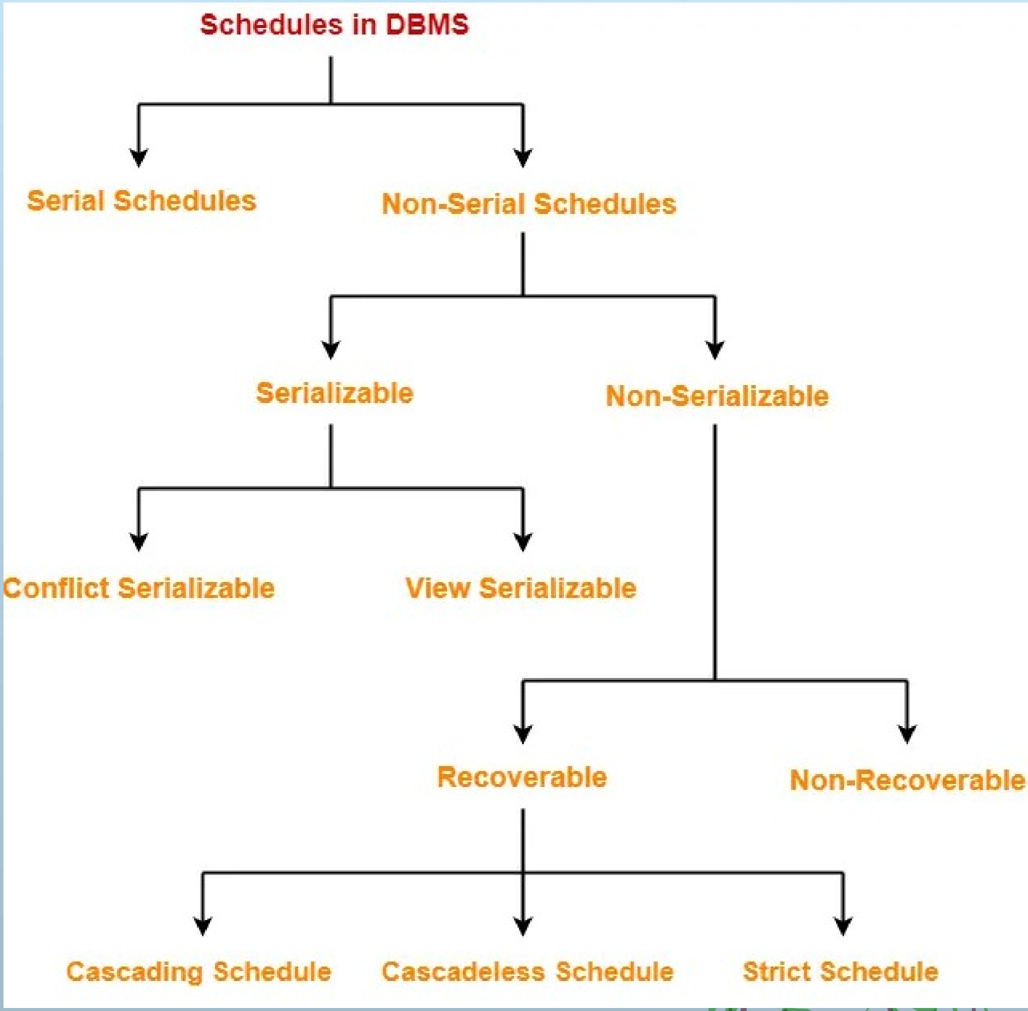
- Assumes disks to not fail
- Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the *entire* database.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

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



Example Schedules

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

Example Schedule (Cont.)

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

Example Schedules (Cont.)

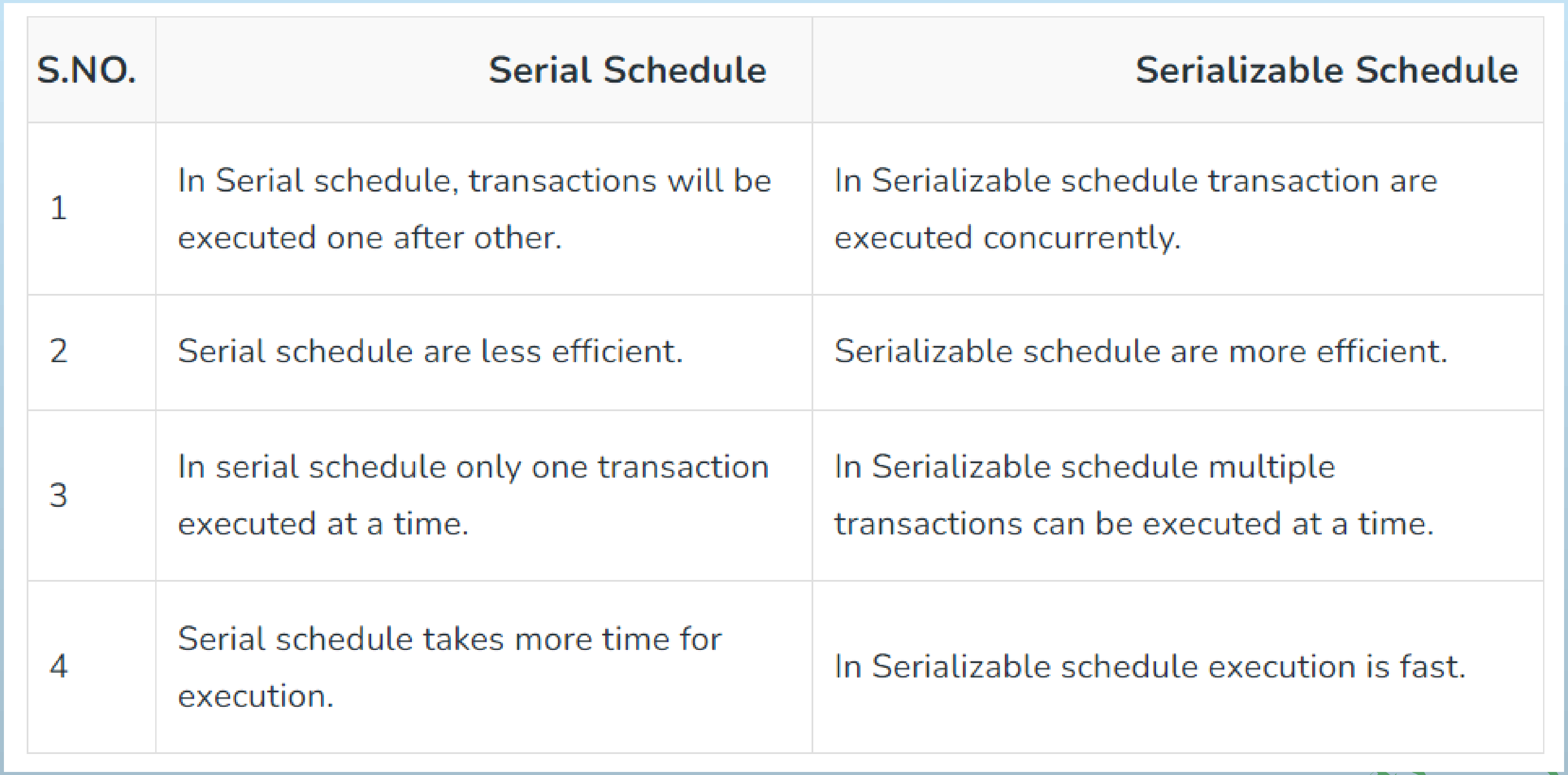
- The following concurrent schedule does not preserve the value of the sum *A* + *B*.

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

Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability
- We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Serial Vs Serializable



S.NO.	Serial Schedule	Serializable Schedule
1	In Serial schedule, transactions will be executed one after other.	In Serializable schedule transaction are executed concurrently.
2	Serial schedule are less efficient.	Serializable schedule are more efficient.
3	In serial schedule only one transaction executed at a time.	In Serializable schedule multiple transactions can be executed at a time.
4	Serial schedule takes more time for execution.	In Serializable schedule execution is fast.

Conflict Serializability

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them. If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.
- A pair of operations are said to be conflicting operations if they follow the set of conditions given below:
  - Each operation is a part of different transactions.
  - Both operations get performed on the same data item.
  - One of the performed must be a write operation.

Conflict Serializability (Cont.)

- 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
- A conflict serializable schedule is one that can be converted from a non-serial schedule to a serial schedule by swapping its non-conflicting operations.
- Example of a schedule that is not conflict serializable:

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

We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Checking Whether a Schedule is Conflict Serializable Or Not

A non-serial schedule gets checked for conflict serializability using the following steps:

1. Figure out all the conflicting operations and enlist them.

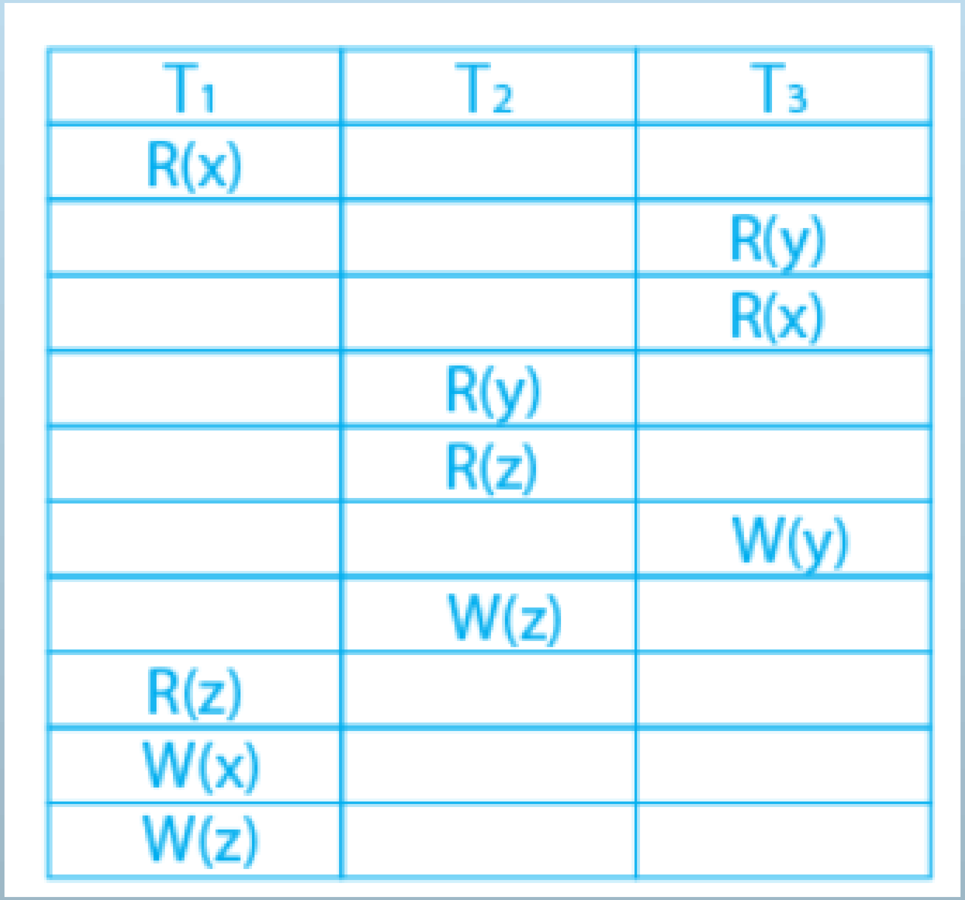
2. Create a precedence graph. For every transaction in the schedule, draw a node in the precedence graph.

3. If Xi(A) and Yj(A) represent a conflict pair, then draw an edge from Ti to Tj for each conflict pair. The precedence graph ensures that Ti gets executed before Tj.

4. The next step involves checking for any cycle formed in the graph. **A schedule is a conflict serializable if**

Example

In this example we will check schedule is conflict serializable or not.



T_1	T_2	T_3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

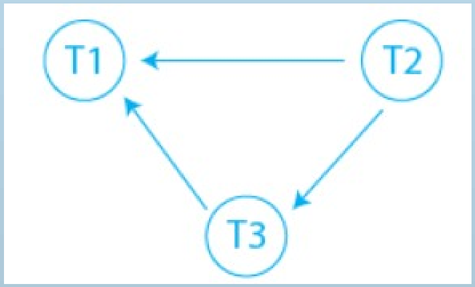
Example contd..

Step 1: We will find and list all of the operations that are in conflict.

Conflict Pair

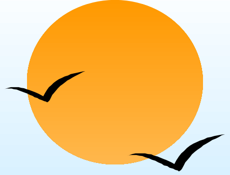
1. Read [3] (x) – Write [1] (x)
2. Read [2] (y) – Write [3] (y)
3. Read [2] (z) – Write [1] (z)
4. Write [2] (z) – Read [1] (z)
5. Write [2] (z) – Write [1] (z)

Step 2: We will create a precedence graph



Step 3: We will draw an edge from Ti to Tj for each conflict pair.

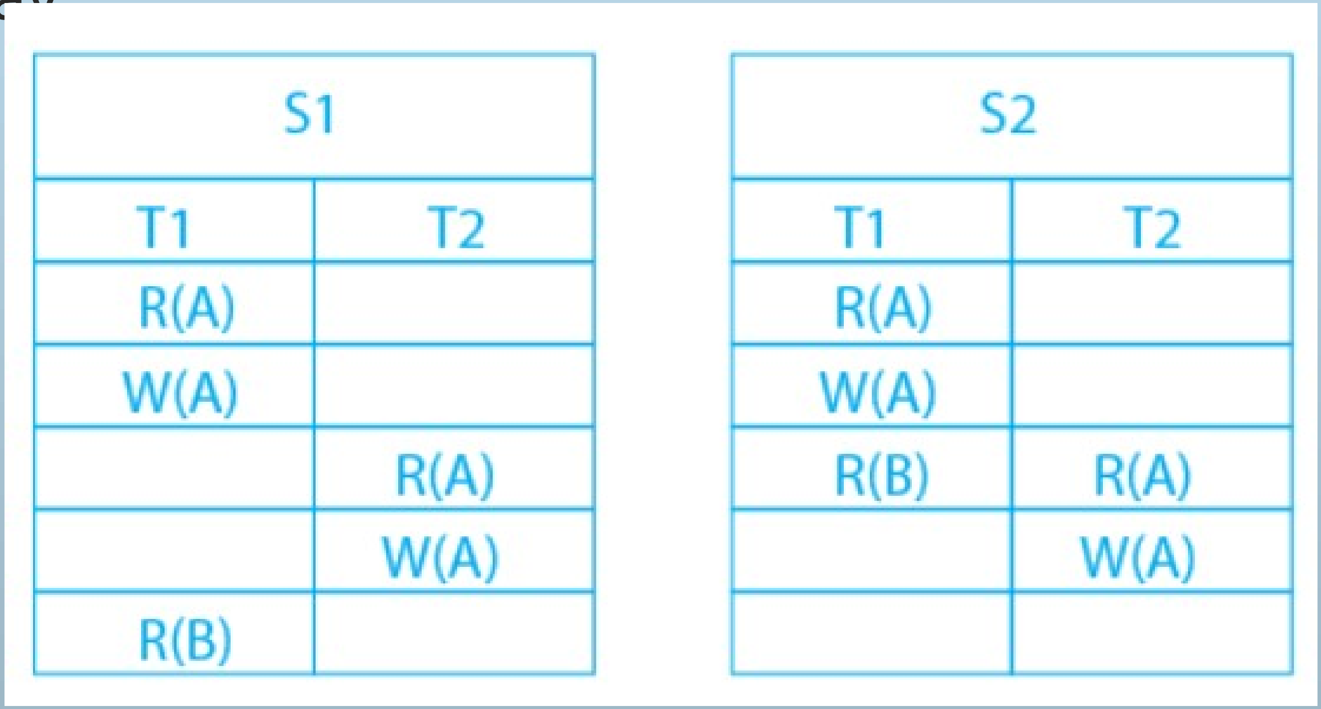
Step 4: Now we will check the graph to see if any cycles or loops have formed. Here in this example, we can not find out any cycles or loops. So this schedule is **conflict serializable.**



Conflict Equivalent Example

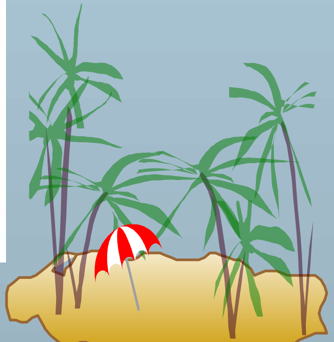
Two schedules are said to be conflict equivalent if and only if they satisfy the following conditions:

1. They both have the same transaction set.
2. If each conflict operation pair is ordered in the same way



S1	
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

S2	
T1	T2
R(A)	
W(A)	
R(B)	R(A)
	W(A)



Example contd..

Step 1: We will make a list of all conflict pairs and non-conflict pairs.

Conflict pairs

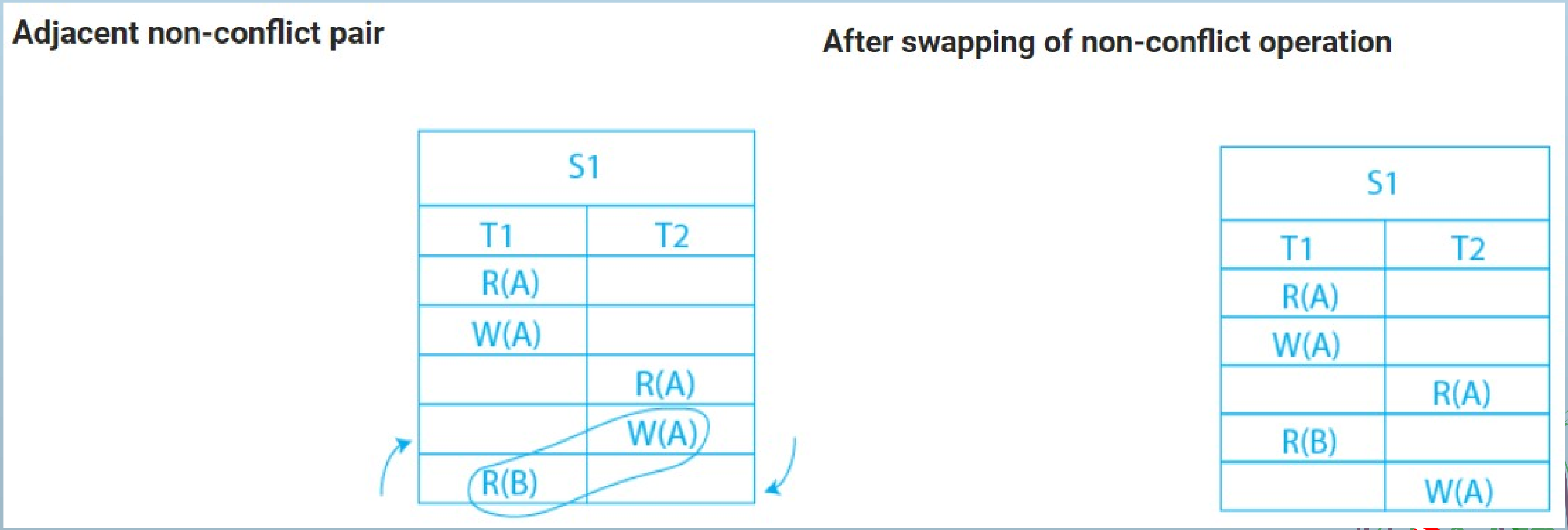
R(A)	W(A)
W(A)	R(A)
W(A)	W(A)

Non-conflict pairs

R(A)	R(A)
R(B)	R(A)
W(B)	R(A)
R(B)	W(A)
W(A)	W(B)

Example contd..

Step 2: We will find the element in S1 which has no similar sequence order to S2. In this example, R(B) element in transaction 1 has no similar sequence.
So we will swap the adjacent non-conflict pair R(B)-W(A).



Adjacent non-conflict pair

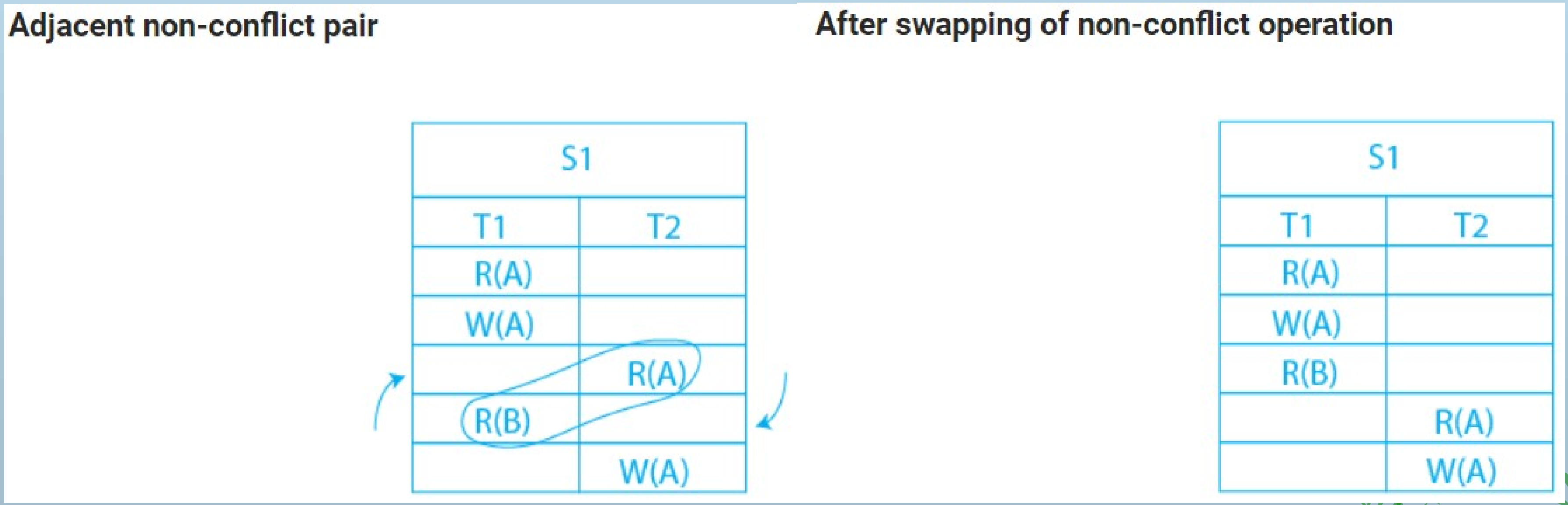
S1	
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

After swapping of non-conflict operation

S1	
T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)

Example contd..

Step 3: Here still we have R(B) element which has no similar order to S2. So we will swap the adjacent non-conflict pair R(B)-R(A).



Adjacent non-conflict pair

S1	
T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)

After swapping of non-conflict operation

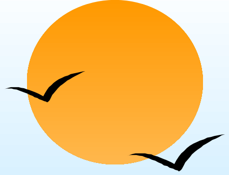
S1	
T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

Now, S1 becomes conflict serializable.

View Serializability

- If a schedule is **view equivalent** to its serial schedule then the given schedule is said to be **View Serializable**.
- Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions:
- 1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

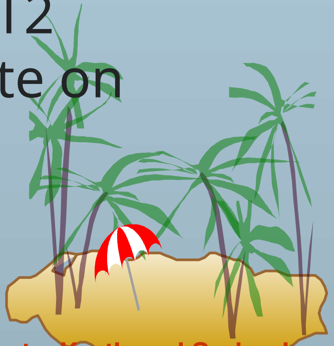
Read vs Initial Read: You may be confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read.



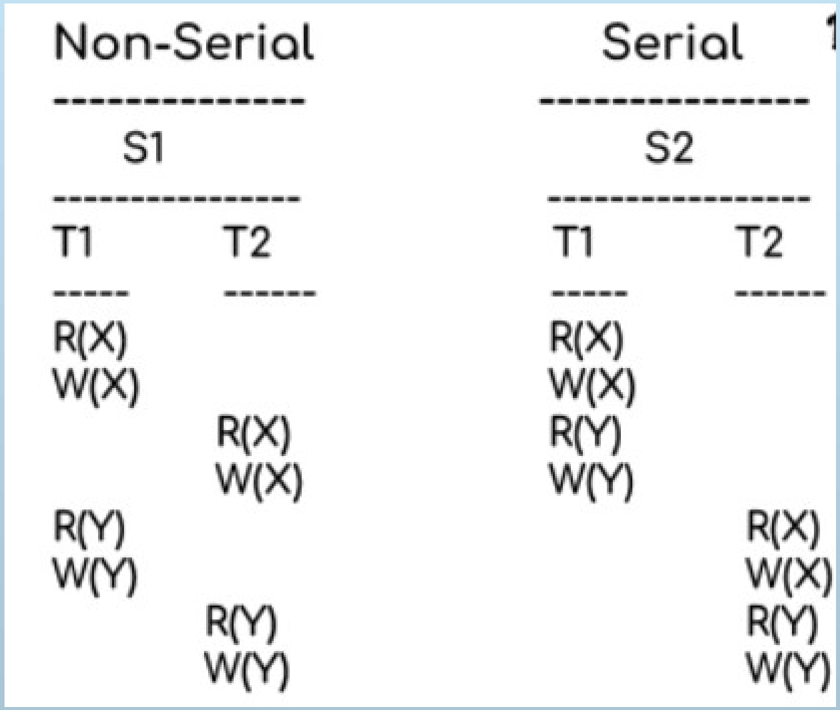
View Serializability contd..

- 2. **Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

- 3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.



View Serializability Example



```
Non-Serial                    Serial
--------------              ---------------
    S1                          S2
---------------              ---------------
T1        T2                 T1         T2
-----     ------             -----      ------
R(X)                         R(X)
W(X)                         W(X)
          R(X)               R(Y)
          W(X)               W(Y)
R(Y)                                    R(X)
W(Y)                                    W(X)
          R(Y)                          R(Y)
          W(Y)                          W(Y)
```

Initial Read
In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.

Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.

We checked for both data items X & Y and the **initial read** condition is satisfied in S1 & S2.

View Serializability Example

- **Final Write**
- 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.
- Lets check for Y. 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.
- We checked for both data items X & Y and the **final write** condition is satisfied in S1 & S2.
- **Update Read**
- In 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 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 update read condition is also satisfied for both the schedules.
- **Result:** 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.

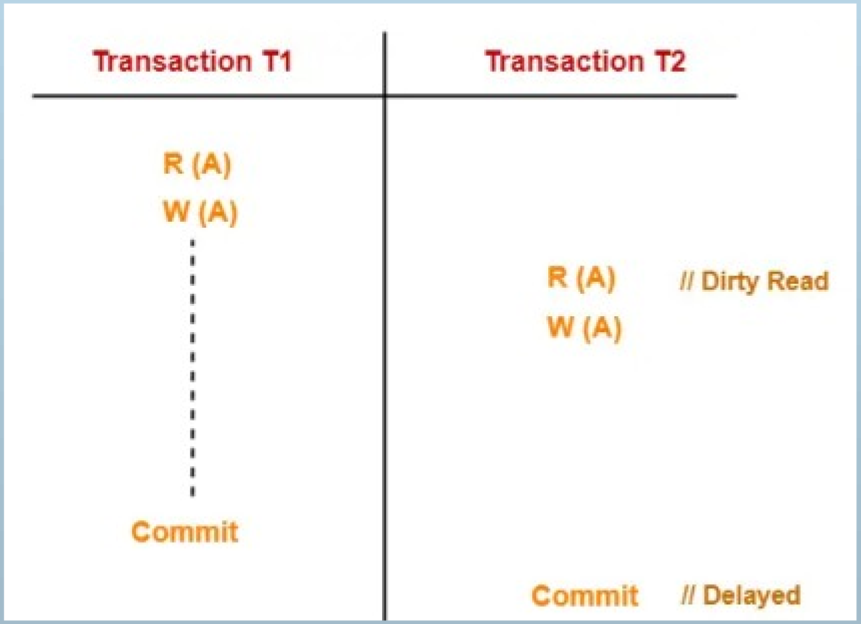
View Serializability Thumb Rules

- All conflict serializable schedules are view serializable.
- All view serializable schedules may or may not be conflict serializable.
- Every view serializable schedule that is not conflict serializable has **blind writes.**
- (Writing without reading is called as a blind write).

Recoverability

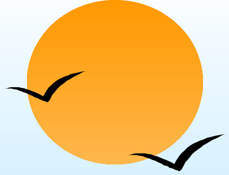
Recoverability is a property of database systems that ensures that, in the event of a failure or error, the system can recover the database to a consistent state.

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



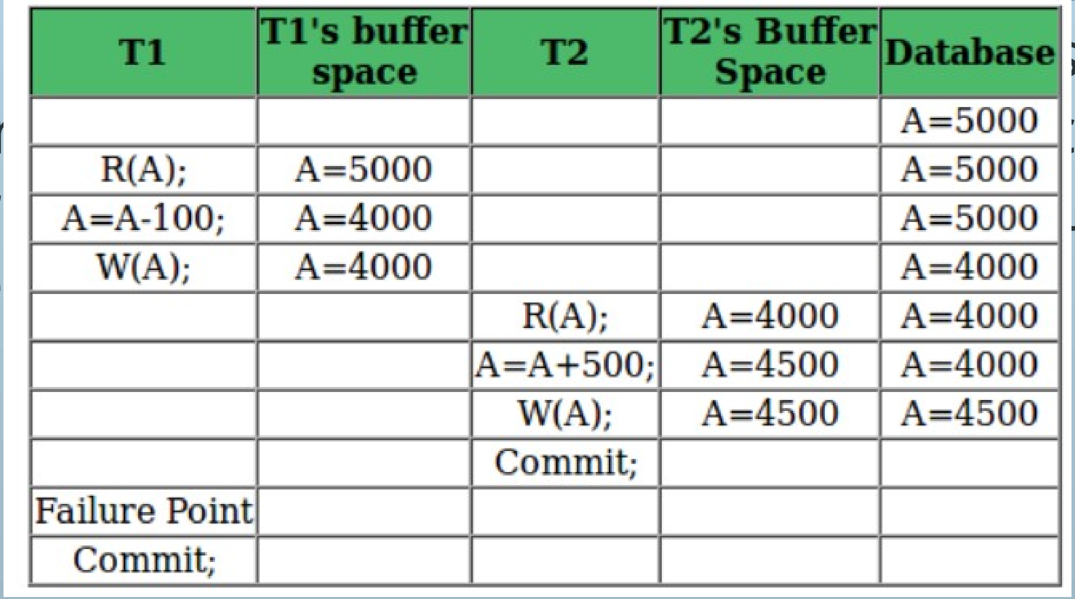
Transaction T1	Transaction T2
R (A)	
W (A)	
	R (A) // Dirty Read
	W (A)
Commit	
	Commit // Delayed

This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

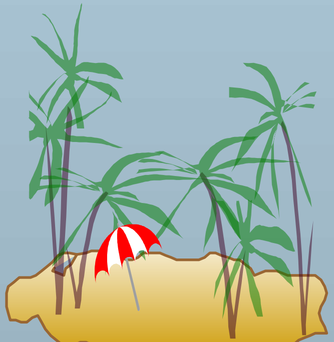


Recoverability contd..

- **Irrecoverable schedule** — When T_j is reading the value updated by T_i and T_j is committed before committing of T_i, the schedule will be irrecoverable.
- The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So … read the value wr… …ked. But we have alr… …ule is irrecove…

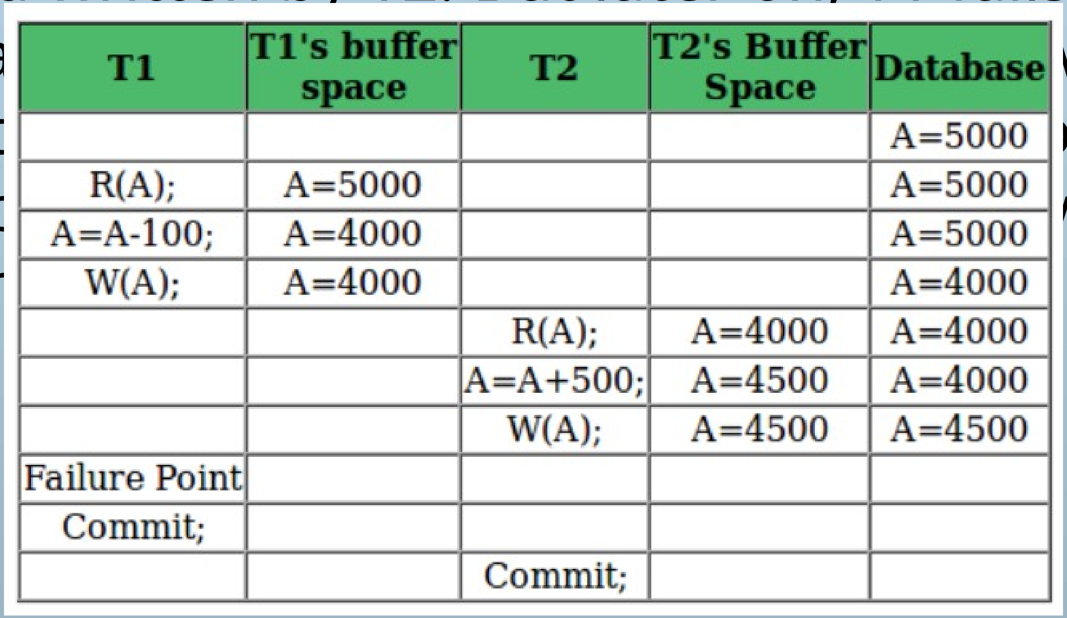


T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		
Failure Point				
Commit;				



Recoverability contd..

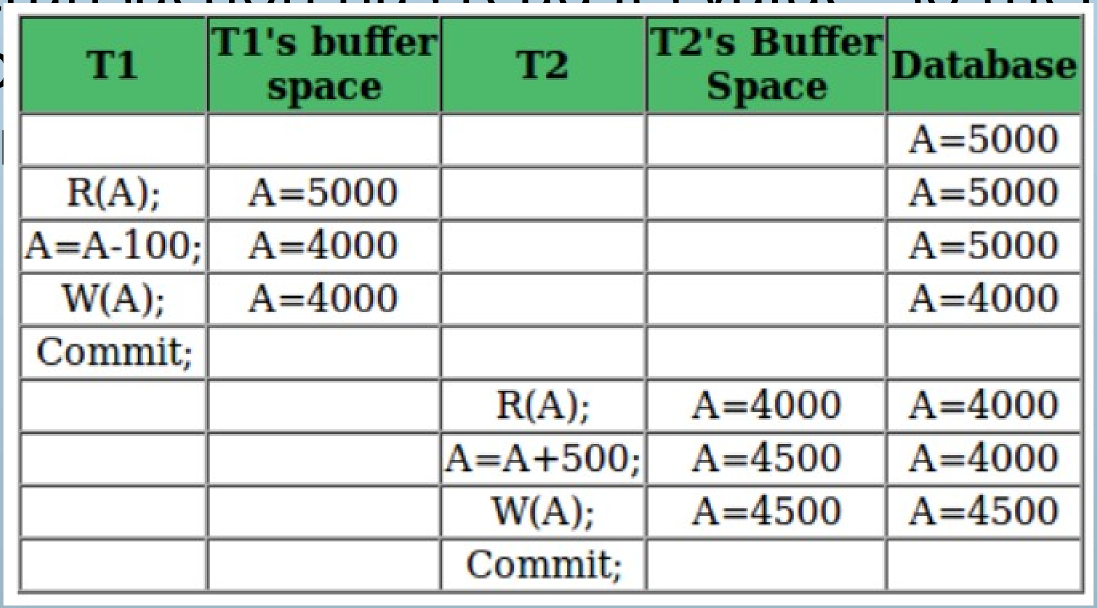
- **Recoverable with Cascading Rollback**— If Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.
- The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rollba written by T1, it shoul ommitted, we can rollb with cascadi



T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
Failure Point				
Commit;				
		Commit;		

Recoverability contd..

- **Cascade less Recoverable Rollback**— If Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.
- The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollb scadeless recove



T1	T1's buffer space	T2	T2's Buffer Space	Database
				A=5000
R(A);	A=5000			A=5000
A=A-100;	A=4000			A=5000
W(A);	A=4000			A=4000
Commit;				
		R(A);	A=4000	A=4000
		A=A+500;	A=4500	A=4000
		W(A);	A=4500	A=4500
		Commit;		

Concurrency Control

- Lock-Based Protocols
- Deadlock Handling

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :

  1. *exclusive (X) mode*.

     Data item can be both read as well as written. This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously. X-lock is requested using **lock-X** instruction.

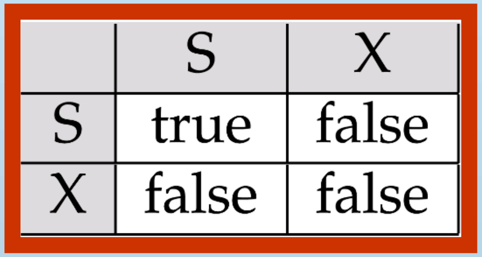
  2. *shared (S) mode*.

     Data item can only be read. It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item. S-lock is requested using **lock-S** instruction.

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix



	S	X
S	true	false
X	false	false

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

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

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

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

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

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

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

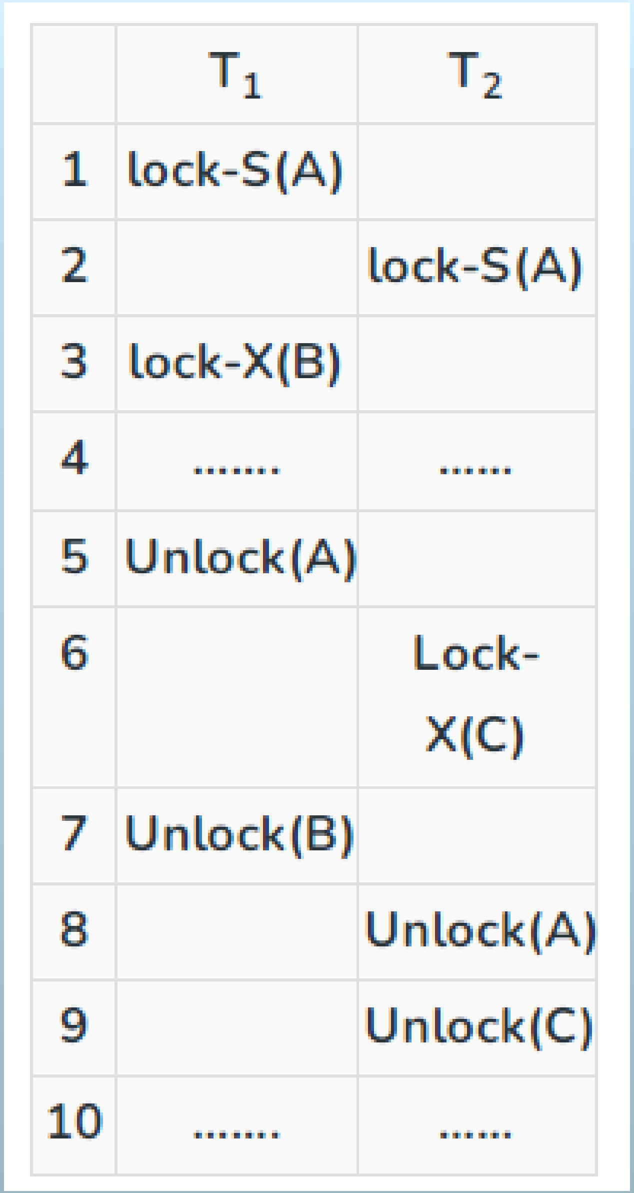
Pitfalls of Lock-Based Protocols (Cont.)

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

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - New locks on data items may be acquired but none can be released.
- Phase 2: Shrinking Phase
  - Existing locks may be released but no new locks can be acquired.
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock).
- Note – If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

The Two-Phase Locking Protocol Example



	T_1	T_2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4	....	......
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	....	......

Transaction T_1:

- The growing Phase is from steps 1-3.
- The shrinking Phase is from steps 5-7.
- Lock Point at 3

Transaction T_2:

- The growing Phase is from steps 2-6.
- The shrinking Phase is from steps 8-9.
- Lock Point at 6

Strict Two-Phase Locking Protocol

- The transaction can release the shared lock after the lock point.
- The transaction can not release any exclusive lock until the transaction commits.
- In strict two-phase locking protocol, if one transaction rollback then the other transaction should also have to roll back. The transactions are dependent on each other. This is called **Cascading schedule**.

Rigorous Two-Phase Locking Protocol

- The transaction cannot release either of the locks, i.e., neither shared lock nor exclusive lock.
- Serailizability is guaranteed in a Rigorous two-phase locking protocol.
- Deadlock is not guaranteed in rigorous two-phase locking protocol.

Conservative Two-Phase Locking Protocol

- The transaction must lock all the data items it requires in the transaction before the transaction begins.
- If any of the data items are not available for locking before execution of the lock, then no data items are locked.
- The read-and-write data items need to know before the transaction begins. This is not possible normally.
- Conservative two-phase locking protocol is **deadlock-free**.
- Conservative two-phase locking protocol does not ensure a strict schedule.

Timestamp based Locking Protocol

- It uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions.
- The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.
- The older transaction is always given priority in this method.
- It uses system time to determine the time stamp of the transaction. This is the most commonly used concurrency protocol.
- Lock-based protocols help you to manage the order between the conflicting transactions when they will execute. Timestamp-based protocols manage conflicts as soon as an operation is created.

Deadlock Handling

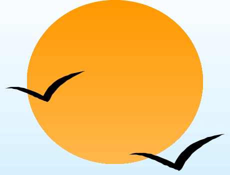
- Consider the following two transactions:

  T_1: write (*X*) write(*Y*)

  T_2: write(*Y*) write(*X*)

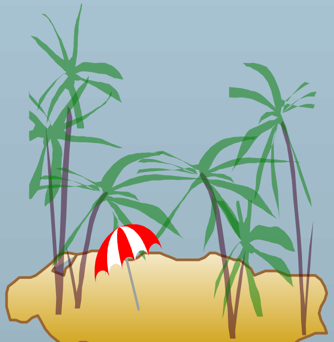
- Schedule with deadlock

T_1	T_2
lock-X on *X*	
write (*X*)	
	lock-X on *Y*
	write (Y)
	wait for **lock-X** on *X*
wait for **lock-X** on *Y*	



Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** 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 (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).



More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - When a transaction requests a resource that is already locked by some other transaction, then the DBMS checks the timestamp of both the transactions and makes the older transaction wait until that resource is available for execution.
- **wound-wait** scheme — preemptive
  - When an older transaction demands a resource that is already locked by a younger transaction (a transaction that is initiated later), the younger transaction is forced to kill/stop its processing and release the locked resource for the older transaction's own execution.
  - The younger transaction is now restarted with a one-minute delay, but the timestamp remains the same.
  - If a younger transaction requests a resource held by an older one, the younger transaction is made to wait until the older one releases the resource.

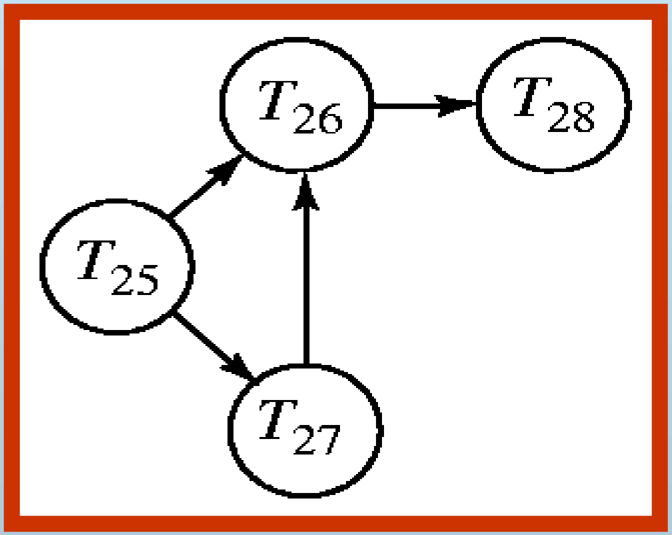
Deadlock prevention (Cont.)

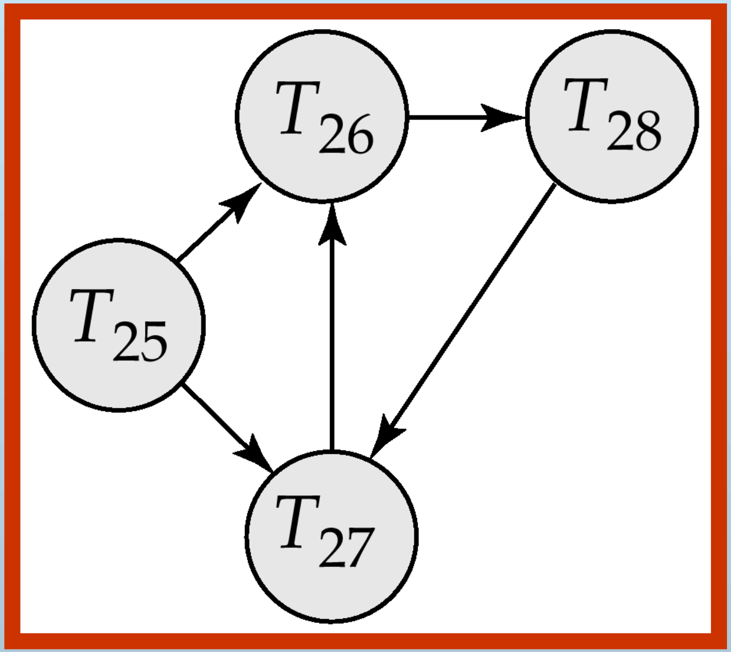
- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- Timeout-Based Schemes :
  - a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - thus deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge T_i T_j is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)





Wait-for graph without a cycle

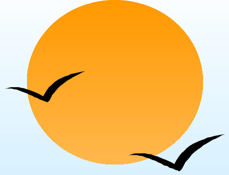
Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

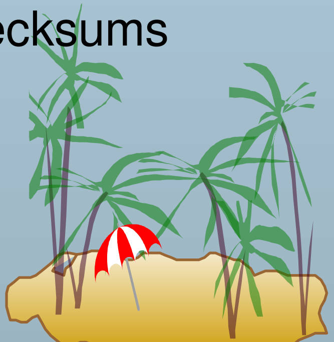
Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging



Failure Classification

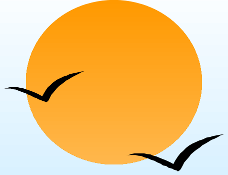
- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures



Recovery Algorithms

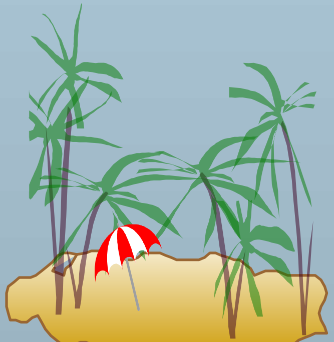
- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures

- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability



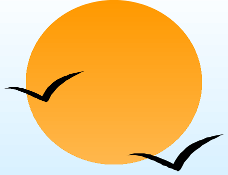
Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction T_i that transfers $50 from account *A* to account *B*; goal is either to perform all database modifications made by T_i or none at all.
- Several output operations may be required for T_i (to output *A* and *B*). A failure may occur after one of these modifications have been made but before all of them are made.



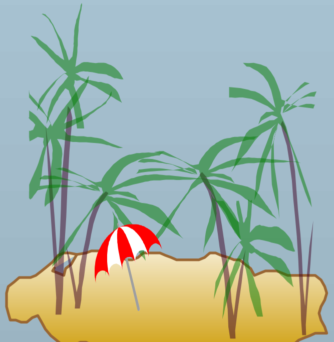
Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.



Log-Based Recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a
  $<T_i$ **start**$>$log record
- *Before* T_i executes **write**(X), a log record $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record $<T_i$ **commit**$>$ is written.
- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification



Deferred Database Modification

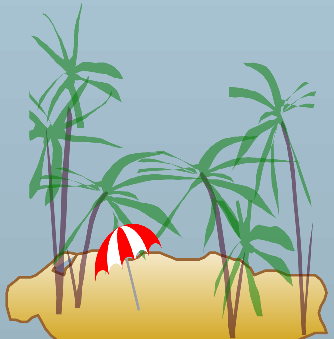
- Whenever any transaction is executed, the updates are not made immediately to the database. They are first recorded on the log file and then those changes are applied once the commit is done.
- Assume that transactions execute serially
- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(*X*) operation results in a log record <T_i, *X*, *V*> being written, where *V* is the new value for *X*
  - Note: old value is not needed for this scheme
- The write is not performed on *X* at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be redone if and only if both <T_i **start**> and<T_i**commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- example transactions T_0 and T_1 (T_0 executes before T_1):

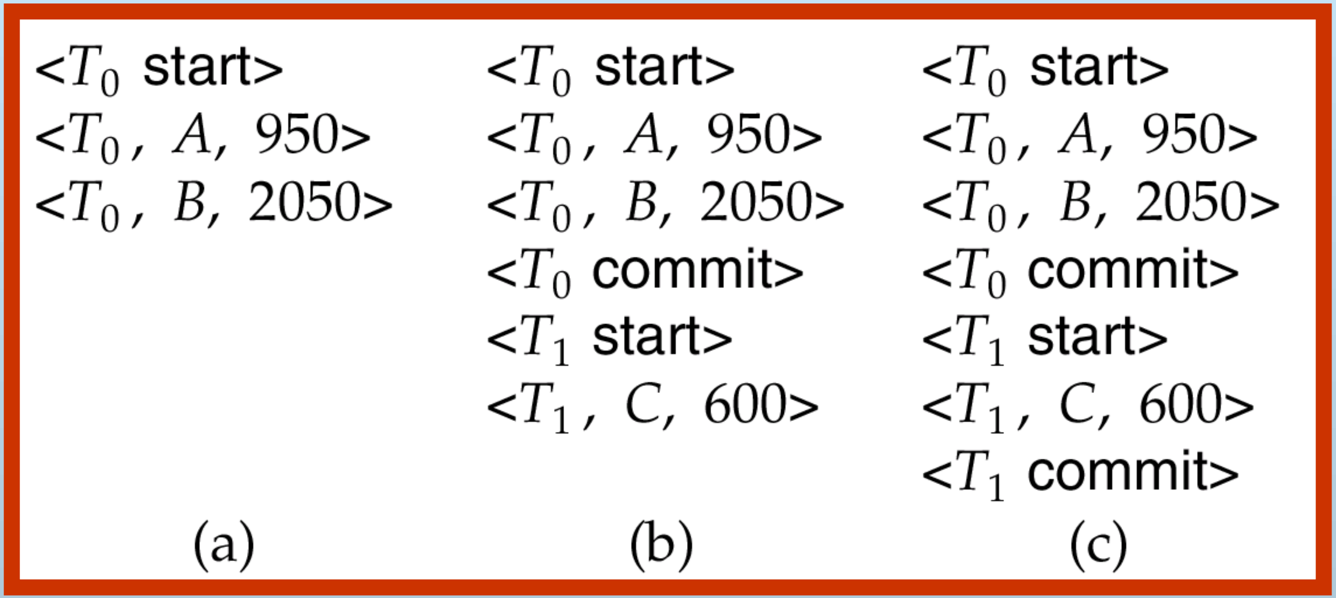
T_0: **read** (*A*)
A: - A - 50
Write (*A*)
read (*B*)
B:- B + 50
write (*B*)

T_1 : **read** (*C*)
C:- C- 100
write (*C*)



Deferred Database Modification (Cont.)

- Below we show the log as it appears at three instances of time.



<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 950>	<T_0, A, 950>	<T_0, A, 950>
<T_0, B, 2050>	<T_0, B, 2050>	<T_0, B, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 600>	<T_1, C, 600>
		<T_1 commit>
(a)	(b)	(c)

- If log on stable storage at time of crash is as in case:

  (a) No redo actions need to be taken

  (b) redo(T_0) must be performed since <T_0 **commi**t> is present

  (c) **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i commit> are present

Immediate Database Modification

- Whenever any transaction is executed, the updates are made directly to the database and the log file is also maintained which contains both old and new values.
- Once the commit is done, all the changes get stored permanently in the database, and records in the log file are thus discarded.
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification Example

Log	**Write**	**Output**
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_o, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**> x_1		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

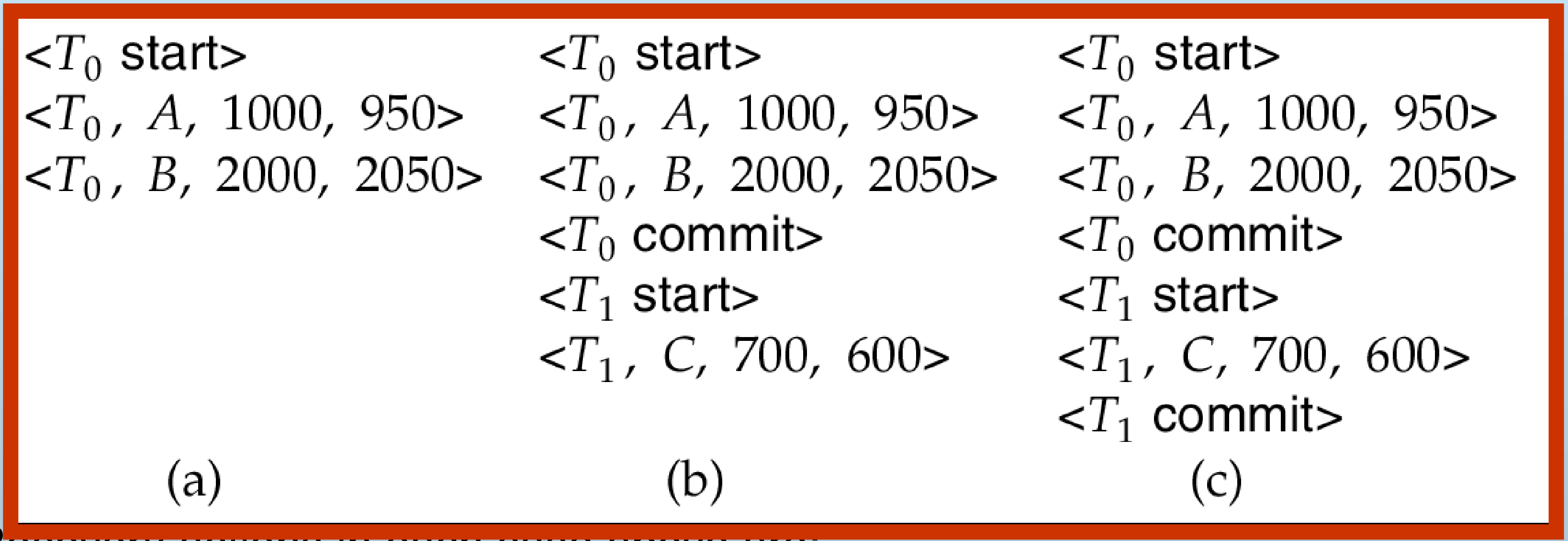
- Note: B_X denotes block containing X.

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
  - Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.
- Undo operations are performed first, then redo operations.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.



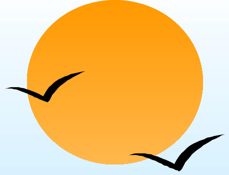
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

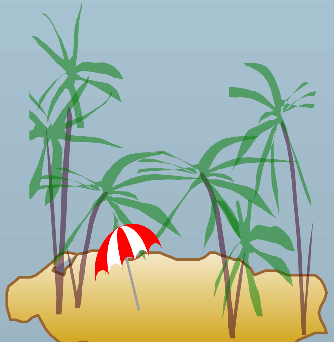
(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

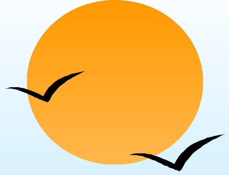


Checkpoints

- Problems in recovery procedure are as follows:
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already output their updates to the database.

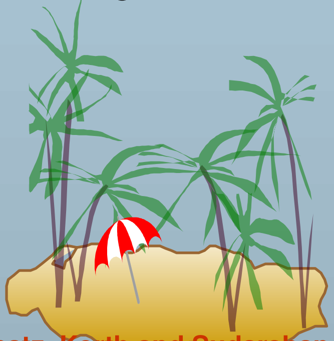
- Streamline recovery procedure by periodically performing **checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**> onto stable storage.



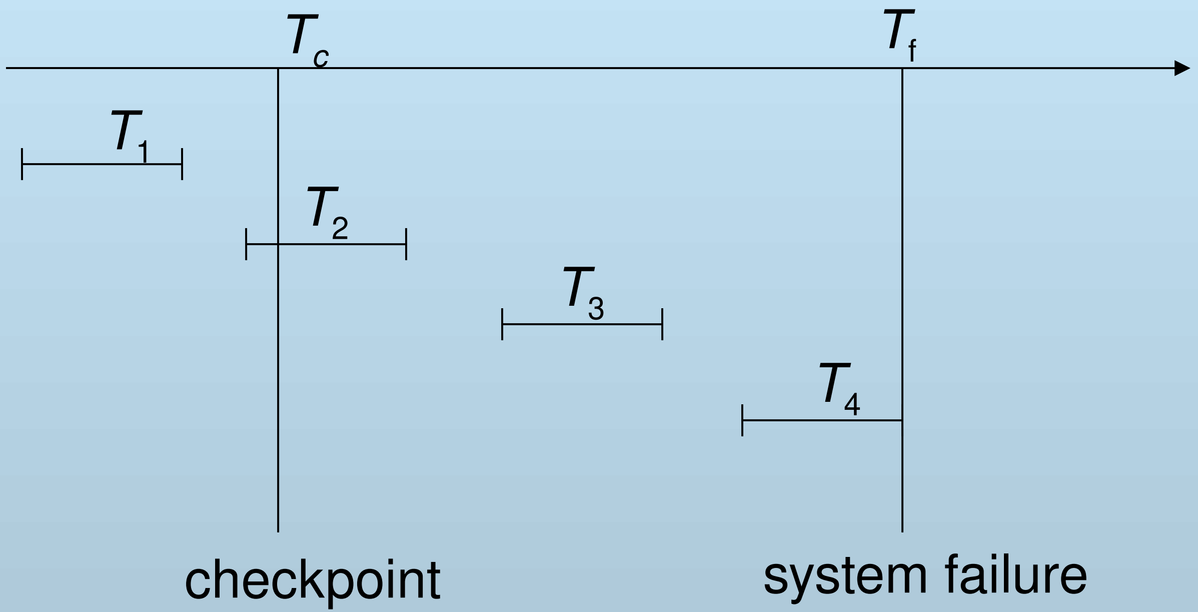


Checkpoints (Cont.)

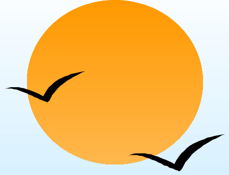
- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record
  2. Continue scanning backwards till a record $<T_i$ **start**> is found.
  3. Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no $<T_i$ **commit**>, execute **undo**(T_i). (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ **commit**>, execute **redo**(T_i).



Example of Checkpoints

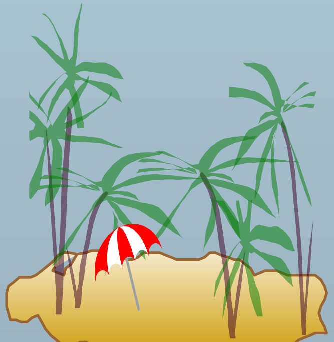


- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

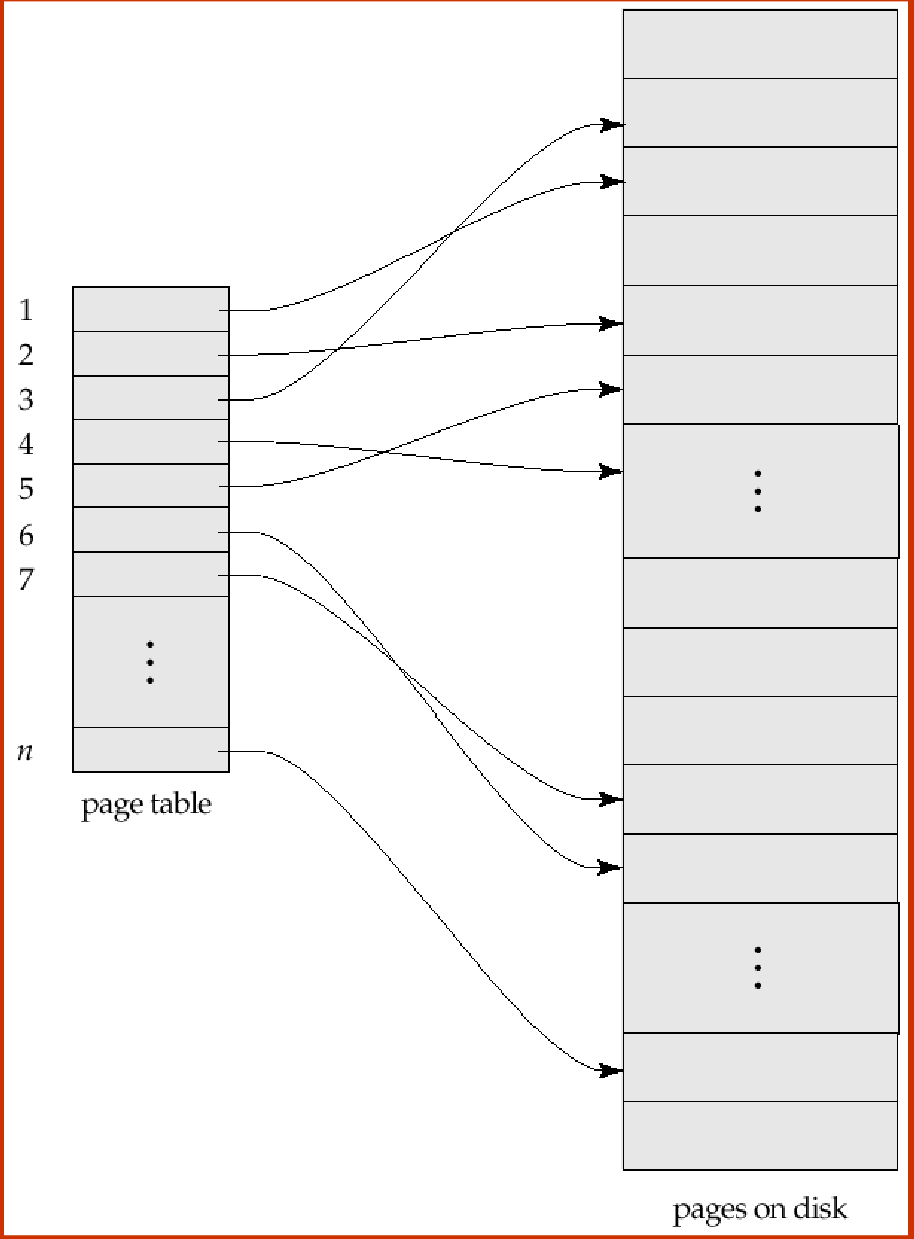


Shadow Paging

- **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction – the **current page table**, and the **shadow page table**
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are identical. The current table is updated for each write operation.
- 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

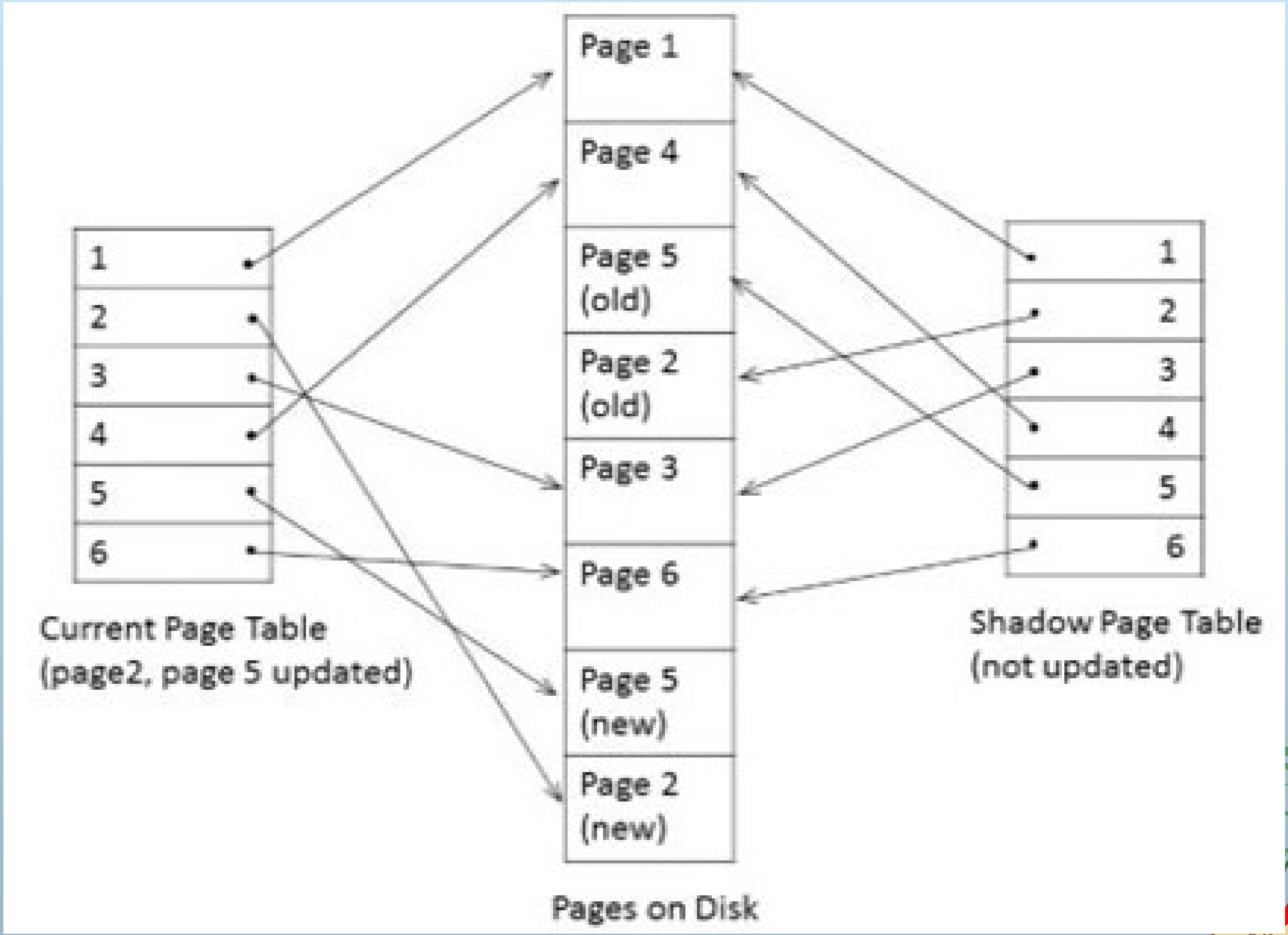


Sample Page Table



Example of Shadow Paging

Shadow and current page tables after write to page 2 and 5



Shadow Paging (Cont.)

- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table to be the new shadow page table, as follows:
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
     - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected).

Shadow Paging (Cont.)

- Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery is trivial
- Disadvantages :
  - Copying the entire page table is very expensive
    - Can be reduced by using a page table structured like a B^{+}-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
  - Commit overhead is high even with above extension
    - Need to flush every updated page, and page table
  - Data gets fragmented (related pages get separated on disk)
  - After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
  - Hard to extend algorithm to allow transactions to run concurrently
    - Easier to extend log based schemes