

# Transaction processing and Concurrency Control:

---

## Part -I : Transaction Processing and Concurrency Control

### Definition of Transaction:

A type of computer processing in which the computer responds immediately to user requests. Each request is considered to be a transaction. Automatic teller machines for banks are an example of transaction processing.

### Basic operations on database are read and write

1. read\_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.
2. write\_item(X): Writes the value of program variable X into the database item named X.

### READ AND WRITE OPERATIONS:

- ✚ Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.

### Example:

You are working on a system for a bank. A customer goes to the ATM and instructs it to **transfer Rs. 1000 from savings to a current account**. This simple transaction requires two steps:

- Subtracting the money from the savings account balance.

Savings -1000

- Adding the money to the checking account balance.

Current + 1000

The code to create this transaction will **require two updates** to the database. For example, there will be two SQL statements: one UPDATE command to **decrease the balance in savings** and a second UPDATE command to **increase the balance in the current account**.

You have to consider what would happen if **a machine crashed between these two operations**. The money has already been subtracted from the savings account will not be added to the checking account. It is lost. You might consider performing the addition to checking first, but then the customer ends up with extra money, and the bank loses. **The point is that both changes must be made successfully.**

**Thus, a transaction is defined as a set of changes that must be made together.**

## Process of Transaction

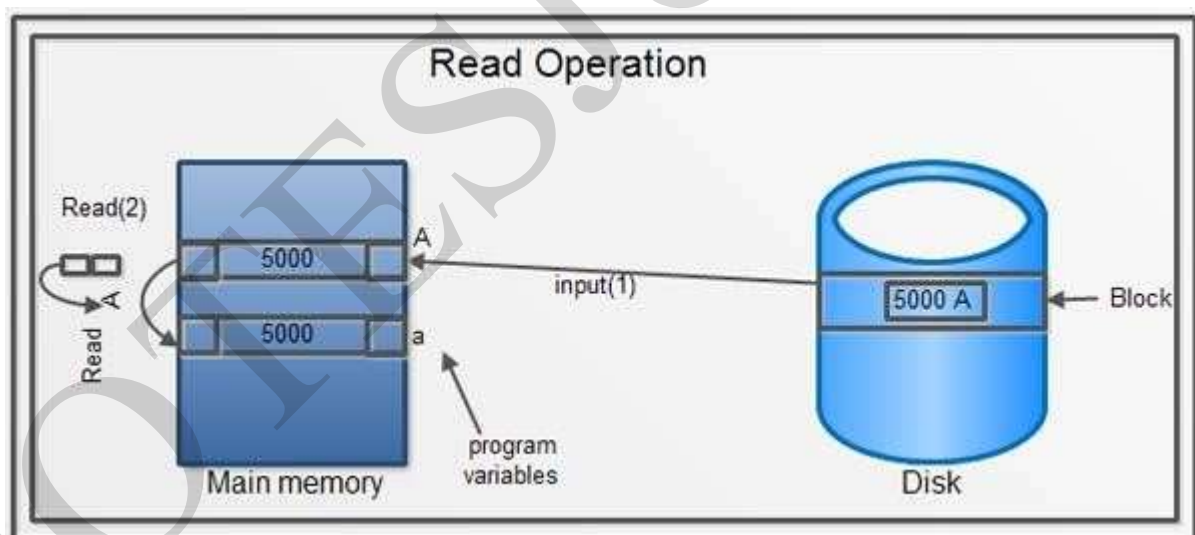
The transaction is executed as a series of reads and writes of database objects, which are explained below:

### Read Operation

`read_item(X)`

disk  $\rightarrow$  to main memory  $\rightarrow$  program variable

To read a database object, it is first brought into main memory from disk, and then its value is copied into a program variable as shown in figure.



### Write Operation

`write_item(X)`

program variable  $\rightarrow$  to main memory  $\rightarrow$  disk

To write a database object, an in-memory copy of the object is first modified and then written to disk.

## What causes a Transaction to fail

There are several reasons for a transaction to fail in the middle of execution.

1. Computer failure: a hardware, software or network error occurs in the computer system during transaction execution.
2. Transaction or system: some operations in the transaction may cause it to fail such as integer overflow or division by 0. The user may also interrupt the transaction during its execution.
3. Local errors or exception conditions detected by the transaction :during transaction execution , certain condition may occur that necessitate the cancellation of transaction.

Example: insufficient account balance in a banking database may cause a transaction to be cancelled.

4. Concurrency control enforcement: this method may decide to abort the transaction because several transactions are in a state of deadlock.
5. Disk failure: some disk blocks may lose their data because of a disk read/write head crash. This may happen during a read/ write operation of a transaction.
6. Physical problem & catastrophes: this refers to an endless list of problems that include fire, theft etc.

## Desirable ACID properties

### Atomicity (all or nothing)

1. : Either **all operations of the transaction** are reflected properly in the database, or none are.

### Consistency (No violation of integrity constraints)

2. : Execution of a transaction in isolations (that is, with no other transaction executing concurrently).

### Isolation (concurrent changes invisibles)

3. : Even though multiple transactions may execute concurrently, the system guarantees that, 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.
- Thus, each transaction is unaware of other transactions executing concurrently in the system. (Execution of transaction should not be interfered with by any other transactions executing concurrently)

### Durability (committed update persist)

4. : After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
- (The changes must not be lost because of any failure)

Or

T1 Schedule
Read (A,a)
a: =a-50;
Write ( A,a)
Read(B,b)
b: = b+50;
Write (B,b);

1. **Atomicity.** Either **all operations of the transaction** are reflected properly in the database, or none are.

State before the execution of transaction  $T_i$

The value of A = 1000

The value of B = 2000

Failure occur (ex. Hardware failure) after write (A,a) and before write (B,b)

i.e changes in A are performed but not reflected in B

Now,  $A \Rightarrow 1000 - 50 = 950$

$B \Rightarrow 2000$

We have lost Rs.50 as a result of this failure.

■ Idea behind ensuring atomicity is following:

- The database system keeps track of the old values of any data on which a transaction performs a write.
- If the transaction does not complete, the DBMS restores the old values to make it appear as though the transaction have never execute.

## 2. Consistency

The consistency requirement here is that the sum of A and B must be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction. It can. Be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction.

## 3. Isolation.

Even though multiple transactions may execute concurrently, the system guarantees that,

- 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.

- Thus, each transaction is unaware of other transactions executing concurrently in the system.
- ( **Execution of transaction should not be interfered with by any other transactions executing concurrently** )

T1	T2	Status
Read (A,a) a=a-50 Write (A,a)		value of A i.e. 1000 is copied to local variable a local variable a = 950 value of local variable a 950 is copied to a database item A
	Read (A,a) Read(B,b) display(a+b)	value of database item A 950 is copied to a value of database item B 2000 is copied to b 2950 is displayed as Sum of accounts A and B
Read(B,b) b:= b+50 Write(B,b)		value of data base item B 2000 is copied to local variable b local variable b= 2050 value of variable b 2050 is copied to database item B

## 4. Durability or permanency.

After a transaction completes successfully, the changes it has made to the database must persist, even if there are system failures.

- **These changes must not be lost because of any failure**
- ensures that, transaction has been committed, that transaction's updates do not get lost, even if there is a system failure.

## State diagram of transition/transaction

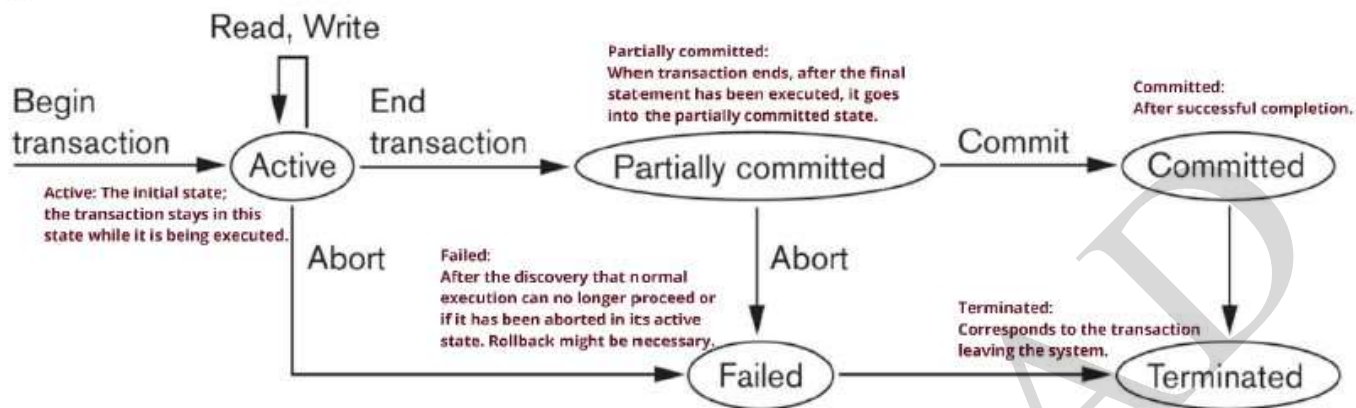


Figure 1: state diagram of transaction

- **Active**, the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed but its changes to the database have not yet been made permanent.
- **Failed**, after the discovery that normal execution can no longer proceed due to hardware/ logical errors.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of transaction.

□ The DBMS could either **kill** the transaction or **restart** the transaction.

When a transaction is to be killed or restarted?

**Kill:** because of some internal logical error, a transaction is killed

**Restarted:** when transaction is aborted because of hardware or software errors. A restarted transaction is considered a new transaction.

- **Committed**, after successful completion

overview of serializability, serializable and non serializable transactions

## Overview of serializability

When multiple transactions are being executed by the operating system in a multiprogramming environment, there are possibilities that instructions of one transaction are interleaved with some other transaction.

- **Objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference**
- **Could run transactions serially, but this limits degree of concurrency or parallelism in system**
- **Serializability identifies those executions of transactions guaranteed to ensure consistency**

### Schedule

A schedule S of n transactions is a sequential ordering of the operations of the n transactions.

A schedule maintains the order of operations within the individual transaction.

- ✚ For each transaction T if operation a is performed in T before operation b, then operation a will be performed before operation b in S.
- ✚ The operations are in the same order as they were before the transactions were interleaved

A transaction schedule is a tabular representation of how a set of transactions were executed over time

Types of schedule:

1. Complete schedule
2. Serial schedule
3. Non-serial schedule
4. Equivalent schedule
5. Serializable schedule

1. **Complete schedule:** is a schedule that must contain all actions of every transaction that appears in it. It contains either an abort or a commit for each transaction whose actions are listed in it. This type of schedule is complete.



Complete Schedule	
T <sub>1</sub>	T <sub>2</sub>
R(A)	
W(A)	
	R(B)
Commit	
	W(B)
	Abort

Complete Schedule	
T <sub>1</sub>	T <sub>2</sub>
R(A)	
	R(B)
W(A)	
	W(B)
Commit	
	Abort

### 2&3. Serial schedule & Non-serial schedule

Serial Schedule	Non-Serial Schedule
A serial schedule is a sequence of operation by a set of concurrent transaction that preserves the order of operations in each of the individual transactions.	A non-serial schedule is a schedule where the operations of a group of concurrent transactions are interleaved.
Transactions are performed in serial order.	Transactions are performed in non-serial order, but result should be same as serial.
No interference between transactions	Concurrency problem can arise here.
It does not matter which transaction is executed first, as long as every transaction is executed in its entirety from the beginning to end.	The problem we have seen earlier lost update, uncommitted data, inconsistent analysis is arise if scheduling is not proper.
EXAMPLE: If some transaction T is long, the other transaction must wait for T to complete all its operations.	EXAMPLE: In this schedule the execution of other transaction goes on without waiting the completion of T.
In case of banking transaction, there are 2 transactions – one transaction calculates the interest on the account and another transaction deposits some money into the account. Here the order of the execution is important, as the results will be different depending on whether the interest is calculated before or after the money is deposited into the account.	



Serial Schedule		Interleaved schedule	
T1	T2	T1	T2
R1(A)		R1(A)	
W1(A)		W1(A)	
R1(B)			R2(A)
R1(B)			W2(A)
C1		R1(B)	
	R2(A)	R1(B)	
	W2(A)	C1	
	R2(B)		R2(B)
	W2(B)		W2(B)
	C2		C2
schedule-1		schedule-2	

Table 1: Schedule

5. **Equivalent schedule:** 2 schedules are said to be equivalent schedule if they produce the same result on any db state.
6. **Serializable schedule:** it is defined as non-serial schedule that is equivalent to some serial execution of transactions.

T1	T2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
READ(B,t)			
t := t+100			

<b>WRITE(B,t)</b>	<b>125</b>
<b>READ(A,s)</b>	
<b>s:= s*2</b>	
<b>WRITE(A,s)</b>	<b>250</b>
<b>READ(B,s)</b>	
<b>s := s*2</b>	
<b>WRITE(B,s)</b>	<b>250</b>

<b>T1</b>	<b>T2</b>	<b>A</b>	<b>B</b>
		<b>25</b>	<b>25</b>
<b>READ(A,t)</b>			
<b>t := t+100</b>			
<b>WRITE(A,t)</b>		<b>125</b>	
	<b>READ(A,s)</b>		
	<b>s:= s*2</b>		
	<b>WRITE(A,s)</b>	<b>250</b>	
<b>READ(B,t)</b>			
<b>t := t+100</b>			
<b>WRITE(B,t)</b>			<b>125</b>
	<b>READ(B,s)</b>		
	<b>s := s*2</b>		
	<b>WRITE(B,s)</b>		<b>250</b>

**Serializable doesn't mean that schedule is a serial schedule. Being serializable means that the schedule is a correct schedule & will keep the db in consistent state**

V. IMP

**Objective of serializability** is to find nonserial schedules that allow concurrent execution without interference.

- In serializability, ordering of read/write is important:
  - (a) If two transactions only read a data item, they do not conflict and order is not important.
  - (b) If two transactions either read or write completely separate data items, they do not conflict and order is not important.
  - (c) **If one transaction writes a data item and another reads or writes same data item, order of execution is important.**

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

## Methods for TESTING serializability

### 1. CONFLICT SERIALIZABILITY

- Two operations conflict if
  - a. they are issued by different transactions,
  - b. operate on the same data item, and
  - c. one of them is a write operation
- Conflict equivalence: all conflicting operations have the same order
- Conflict serializability: S is conflict-equivalent to a serial schedule

### 2. VIEW SERIALIZABILITY

- View equivalence: if two schedules  $S_1, S_2$  cause all transactions  $T_i$  to read the same values and make the same final writes, then  $S_1$  and  $S_2$  are view-equivalent

- View serializability: S is view-equivalent to a serial schedule

## CONFLICT EQUIVALENCE & CONFLICT SERIALIZABILITY

### Conflict Serializability

Instructions  $I_i$  and  $I_j$ , of transactions  $T_i$  and  $T_j$  respectively, conflict if and only if there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ .

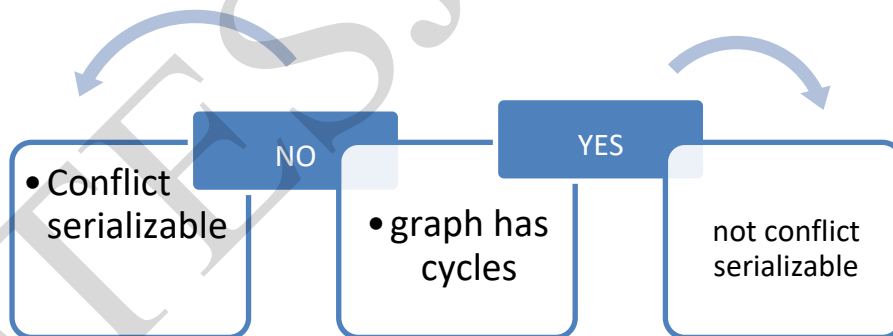
1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict.
3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ . They conflict.
4.  $I_i = \text{write}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict.

### Test for Conflict Serializability

Let  $S$  be the schedule. Get the precedence graph  $G = (V, E)$ .

$V$ - set of vertices

$E$ - set of edges



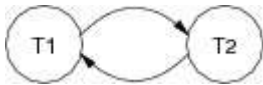
Examples :

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

Precedence graph for this schedule:



No cycles  $\Rightarrow$  serializable

T1: R(A) W(A) R(B) W(B)
T2: R(A) W(A) R(B) W(B)
Precedence graph for this schedule:

Has a cycle $\Rightarrow$ not serializable

### Exercise :

Consider this 3-transaction schedule:
T1: R(A) R(C) W(A) W(C) T2: R(B) R(A) W(B) W(A) T3: R(C) R(B) W(C) W(B)
Precedence graph for this schedule:
Result:
Consider this 3-transaction schedule:
T1: R(A) W(A) R(C) W(C) T2: R(B) W(B) R(A) W(A) T3: R(C) W(C) R(B) W(B)
Precedence graph for this schedule:
Result:

### VIEW EQUIVALENCE / SERIALIZABILITY

Let S and S1 be two schedules with the same set of transactions. S and S1 are view equivalent if the following three conditions are met:

1. 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 S1, also read the initial value of Q.
  2. 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 S1 also read the value of Q that was produced by transaction  $T_j$ .
  3. 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 S1.
- ✓ As can be seen, view equivalence is based purely on **reads** and **writes** alone.

## Part II: Concurrency Control

### Why do we need concurrent executions?

Multiple transactions are allowed to run concurrently in the system. Advantages are:

1. 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
2. Reduced waiting time for transactions: short transactions need not wait behind long ones.

### Problems resulting from concurrent execution

## Why concurrency control is needed?

To address the following problems:

- The Lost Update Problem
- The Temporary Update (or Dirty Read) Problem
- The Incorrect Summary Problem

### The Lost Update Problem

**Problem statement:** Alice & Bob want to transfer Rs.200 to Tom's account

If Alice and Bob don't make the transactions at the same time, e.g. Bob transfers first and Alice transfers next, then everything is OK.

**But in concurrent execution of transactions, any order of interleaving is possible.**

**Here is one possible interleaving of operations in the two transactions:**

	Alice's transaction		Bob's transaction	
t_balance is 1500	Read Tom's balance into t_balance <b>Read t_balance;</b>	time ↓	Read Tom's balance into t_balance <b>Read t_balance;</b>	t_balance is 1500
t_balance is 1700	Add 200 to Tom's balance <b>t_balance = t_balance + 200;</b>			
1700 is written	Write Tom's balance back to the database <b>Write t_balance;</b>		Add 100 to Tom's balance <b>t_balance = t_balance + 100;</b>	t_balance is 1600
			Write Tom's balance back to the database <b>Write t_balance;</b>	1600 is over-written

Instead of 1800, Tom will see 1600 in his account after these transactions. Alice's update is lost/over-written in this interleaving.

## The lost update problem.

Figure 2: Lost Update Problem



### The Incorrect Summary or Unrepeatable Read Problem

Alice is going to send money to Tom, so the program should first reduce the money from her account.

Alice's transaction	Bank officer's transaction
Read Alice's balance into a_balance <b>Read a_balance;</b>	Initialize sum <b>sum = 0;</b>
Deduct 200 from Alice's balance <b>a_balance = a_balance - 200;</b>	
Write Alice's balance back to the database <b>Write a_balance;</b>	
	Read Alice's balance into a_balance <b>Read a_balance;</b>
	Add Alice's balance to sum <b>sum = sum + a_balance;</b>
	Read Tom's balance into t_balance <b>Read t_balance;</b>
	Add Tom's balance to sum <b>sum = sum + t_balance;</b>
Read Tom's balance into t_balance <b>Read t_balance;</b>	
Add 200 to Tom's balance <b>t_balance = t_balance + 200;</b>	
Write Alice's balance back to the database <b>Write t_balance;</b>	

The sum transaction reads Alice's balance after 200 is subtracted, and reads Tom's balance before 200 is added. Summary is off by 200.

Figure 3: Incorrect Summary

### The Temporary Update (Dirty Read) Problem

A transaction prematurely reads a value from the DB that is later invalidated by another transaction.

Alice is going to send money to Tom, so the program should first reduce the money from her account.

around the same time, John is transferring 50SEK to Alice

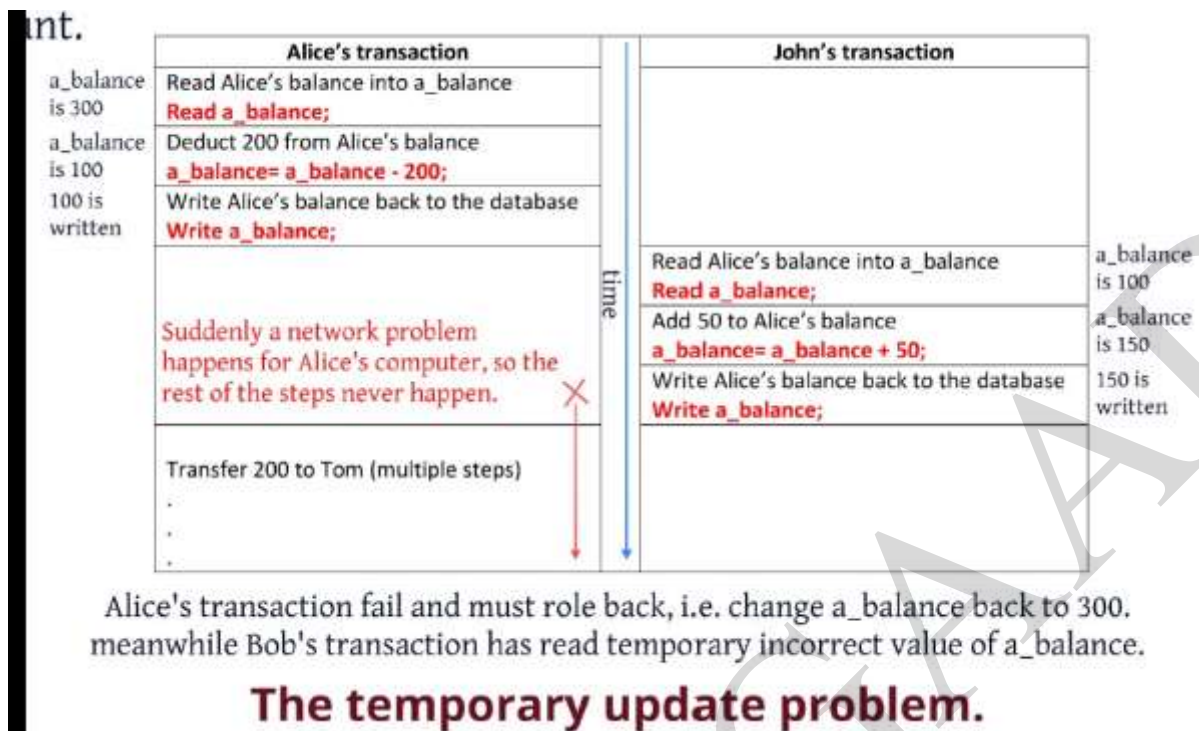


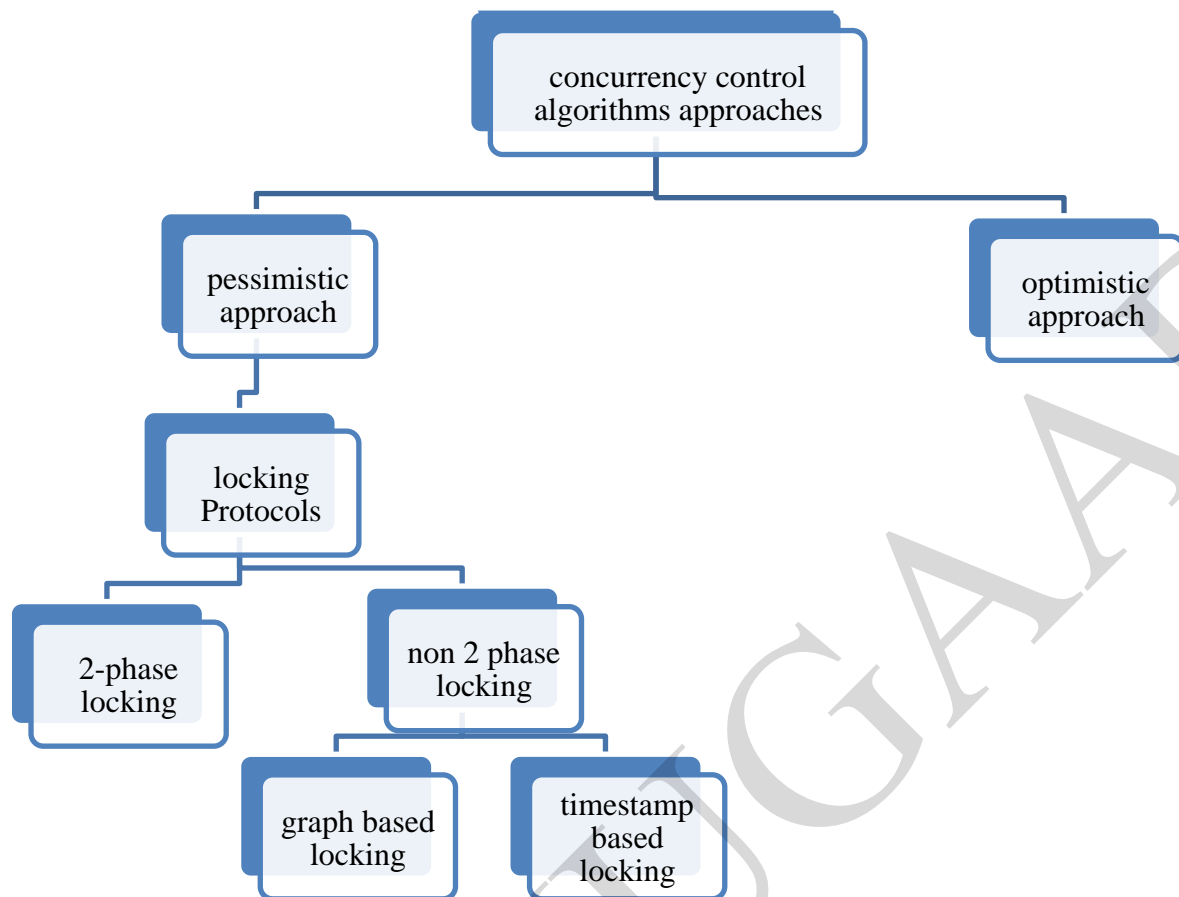
Figure 4: dirty read

Due to some reasons Alice transaction fails and must change the value of a\_balance back to its old value. But in the meanwhile John has read the "temporary" incorrect value of a\_balance .

### Part III Concurrency Control Techniques

The goal of concurrency control is to ensure that concurrent execution of transaction does not result in a loss of database consistency.

When we write concurrency control algorithms, there are 2 approaches



### **Pessimistic approach:**

An approach in which the transaction is delayed if they conflict with each other at some point of time in future is called as pessimistic approach.

### **Optimistic approach:**

It is based on the assumption that conflicts of operations(R/W) on database are rare. It is better to run the transaction to completion and to check for conflicts only before they commit.

## **Lock based protocol**

### **Overview of Locking**

Locking: A solution to problems arising due to concurrency.

**Locking of records can be used as a concurrency control technique to prevent the above mentioned problems. A transaction acquires a lock on a record if it does not want the record**

values to be changed by some other transaction during a period of time. The transaction releases the lock after this time.

Locks are of two types

1. Shared (S lock)
2. Exclusive (X Lock).

- A transaction acquires a shared (read) lock on a record when it wishes to retrieve or fetch the record.
- An exclusive (write) lock is acquired on a record when a transaction wishes to update the record. (Here update means INSERT, UPDATE or DELETE.)

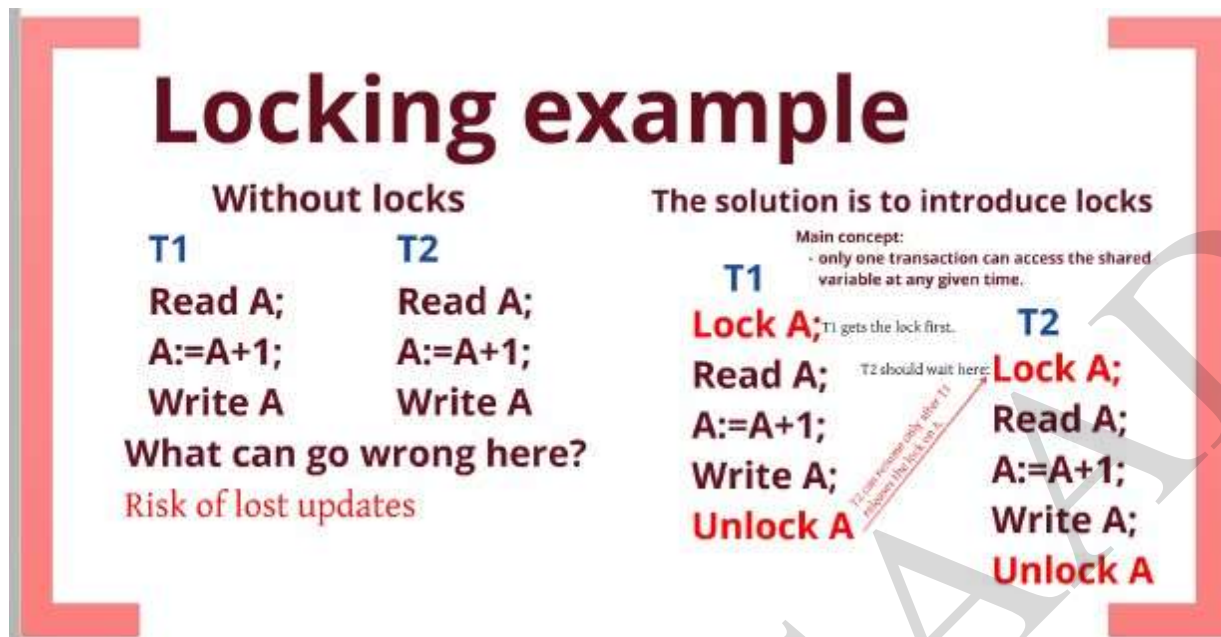
The following figure shows the Lock Compatibility matrix.

	S	X
S	true	false
X	false	false

Figure 5: Lock Compatibility matrix.

“True” means that the lock types are compatible, while “False” means that they are incompatible.

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



### Locking Protocol:

- to ensure the Serializability
- 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.

### Protocols:

1. Two-phase locking (2PL)
2. Non-two-phase locking (skip)
  - tree protocol locking / graph protocol
  - timestamp based protocol

### Two-phase locking (2PL)

2PL is a protocol that guarantees serializability.

Two Phase Locking Protocol has two phases. Growing Phase and Shrinking Phase.

1. In Growing Phase transaction obtains locks but not releasing any locks.
2. In Shrinking Phase transaction releases locks and cannot obtain any new locks.

Initially the transaction is in growing phase, that is the transaction acquires locks as needed. Once the transaction releases lock, it enters the shrinking phase and no more lock request may be issued. Upgrading of lock is not possible in shrinking phase, but it is possible in growing phase. The two phase locking protocol ensures serializability.

A transaction obey the two-phase locking protocol (2PL) iff

1. before operating on any object, the transaction first acquires a lock on that object (the locking phase)
2. after releasing a lock, the transaction never acquires any more lock (the unlocking phase)  
i.e. in any transaction, all locks must precede all unlock.

<e.g.>

T1:	T2:	T3:
LOCK A	LOCK A	LOCK B
LOCK B	LOCK C	LOCK C
UNLOCK A	UNLOCK C	UNLOCK B
UNLOCK B	UNLOCK A	LOCK A
		UNLOCK C
		UNLOCK A

T1 obey 2PL  
T2 obey 2PL  
T3 not obey 2PL

### *Pitfalls of Lock-Based Protocols*

#### 1. Deadlock

Consider the partial schedule

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

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



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

### **Deadlock**

In a multi-process system, deadlock is an unwanted situation that arises in a shared resource environment, where a process indefinitely waits for a resource that is held by another process.

For example, assume a set of transactions  $\{T_0, T_1, T_2, \dots, T_n\}$ .  $T_0$  needs a resource  $X$  to complete its task. Resource  $X$  is held by  $T_1$ , and  $T_1$  is waiting for a resource  $Y$ , which is held by  $T_2$ .  $T_2$  is waiting for resource  $Z$ , which is held by  $T_0$ . Thus, all the processes wait for each other to release resources. In this situation, none of the processes can finish their task. This situation is known as a deadlock.

Deadlocks are not healthy for a system. In case a system is stuck in a deadlock, the transactions involved in the deadlock are either rolled back or restarted.

### **Deadlock Prevention**

To prevent any deadlock situation in the system, the DBMS aggressively inspects all the operations, where transactions are about to execute. The DBMS inspects the operations and analyzes if they can create a deadlock situation. If it finds that a deadlock situation might occur, then that transaction is never allowed to be executed.

There are deadlock prevention schemes that use timestamp ordering mechanism of transactions in order to predetermine a deadlock situation.

### **Wait-Die Scheme**

In this scheme, if a transaction requests to lock a resource (data item), which is already held with a conflicting lock by another transaction, then one of the two possibilities may occur –

- If  $TS(T_i) < TS(T_j)$  – that is  $T_i$ , which is requesting a conflicting lock, is older than  $T_j$  – then  $T_i$  is allowed to wait until the data-item is available.
- If  $TS(T_i) > TS(T_j)$  – that is  $T_i$  is younger than  $T_j$  – then  $T_i$  dies.  $T_i$  is restarted later with a random delay but with the same timestamp.

This scheme allows the older transaction to wait but kills the younger one.



### Wound-Wait Scheme

In this scheme, if a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur –

- If  $TS(T_i) < TS(T_j)$ , then  $T_i$  forces  $T_j$  to be rolled back – that is  $T_i$  wounds  $T_j$ .  $T_j$  is restarted later with a random delay but with the same timestamp.
- If  $TS(T_i) > TS(T_j)$ , then  $T_i$  is forced to wait until the resource is available.

This scheme, allows the younger transaction to wait; but when an older transaction requests an item held by a younger one, the older transaction forces the younger one to abort and release the item.

In both the cases, the transaction that enters the system at a later stage is aborted.

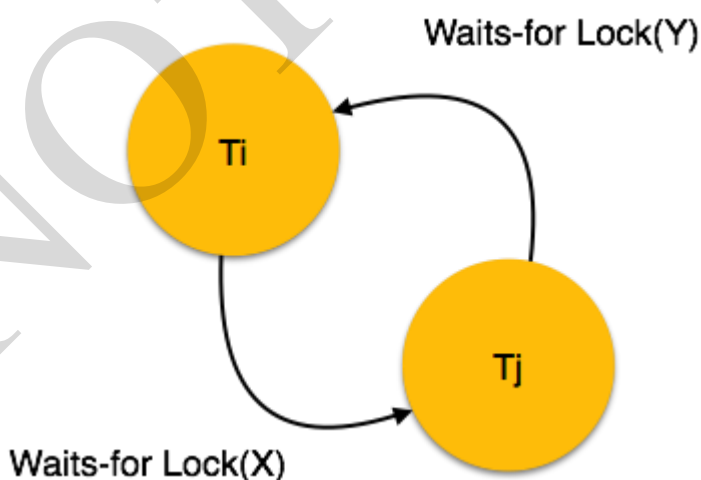
### Deadlock Avoidance

Aborting a transaction is not always a practical approach. Instead, deadlock avoidance mechanisms can be used to detect any deadlock situation in advance. Methods like "wait-for graph" are available but they are suitable for only those systems where transactions are lightweight having fewer instances of resource. In a bulky system, deadlock prevention techniques may work well.

### Wait-for Graph

This is a simple method available to track if any deadlock situation may arise. For each transaction entering into the system, a node is created. When a transaction  $T_i$  requests for a lock on an item, say  $X$ , which is held by some other transaction  $T_j$ , a directed edge is created from  $T_i$  to  $T_j$ . If  $T_j$  releases item  $X$ , the edge between them is dropped and  $T_i$  locks the data item.

The system maintains this wait-for graph for every transaction waiting for some data items held by others. The system keeps checking if there's any cycle in the graph.



Here, we can use any of the two following approaches –

- First, do not allow any request for an item, which is already locked by another transaction. This is not always feasible and may cause starvation, where a transaction indefinitely waits for a data item and can never acquire it.
- The second option is to roll back one of the transactions. It is not always feasible to roll back the younger transaction, as it may be important than the older one. With the help of some relative algorithm, a transaction is chosen, which is to be aborted. This transaction is known as the **victim** and the process is known as **victim selection**.

## 2. Cascading rollback

It is a phenomenon in which a single transaction failure leads to a series of transaction rollback.

Example:

T1	T2	T3
Read (X)		
Read(Y)		
$Z=X+Y$		
Write (Z)		
	Read(Z)	
	Write (Z)	
		Read(Z)

Here, the transaction T1, writes a value of Z that is read by transaction T2, T2 then writes a value of Z which is read by T3. Suppose now that at this point, T1 fails. So T1 must be rolled back. Since T2 depends on T1, so T2 must be rolled back. Since T3 is dependent on T2, so T3 must also be rolled back.

This is known as Cascading Rollback Effect.

### *Overcoming problems of 2PL*

#### 1. Conservative 2PL(Two Phase Locking)

In conservative 2PL there is no growing Phase, Only having shrinking Phase. It gets all the locks before the execution starts .

#### 2. Strict 2PL

Not releasing the write lock (Exclusive Lock) until commit.

#### 3. Rigorous 2PL

First commit then release the lock

### Non two phase locking protocol

These protocols are of two types:

1. Graph based/ tree protocol
2. Timestamp based

#### *Graph based protocol*

In this protocol, we must have prior knowledge about the order in which the database items will be accessed. Here, the items are arranged in a tree form.

The simplest graph based protocols is tree locking protocol which is used to empty exclusive locks and when the database is in the form of a tree of data items. In the tree locking protocol, each transaction  $T_i$  can lock a data item at most once and must observe the following rules. a)

All locks are exclusive locks

- b) The first lock by  $T_i$  can be any data item including the root node.
- c)  $T_i$  can lock a data item  $Q$  only if  $T_i$  currently locks the parent of  $Q$
- d) Data item may be unlocked at any time
- e)  $T_i$  cannot subsequently lock a data item that has been locked and unlocked by  $T_i$

A schedule with a set of transactions that uses the tree locking protocol can be shown to be serializable. The transactions need not be two phase.

#### **Advantage of tree locking control:**

- a. Compared to the two phase locking protocol, unlocking of data item is easier waiting time . So it leads to the shorter waiting times and increase in concurrency.

#### **Disadvantages of tree locking control:**

- a. A transaction may have to lock data items that it does not access descendants we have to lock its parent also. So the number of locks and associated locking overhead is high.

#### *Timestamp based protocol*

A timestamp is a unique identifier created by the DBMS to identify a transaction. Timestamp values are assigned in the order in which the transactions are submitted to the system. So a timestamp is considered as the transaction start time. With each transaction  $T_i$  in the system, a unique timestamp is assigned and it is denoted by  $TS(T_i)$ . When a new transaction  $T_j$  enters the system, then  $TS(T_i) < TS(T_j)$ , this is known as timestamp ordering scheme. To implement this scheme, each data item ( $Q$ ) is associated with two timestamp values.

1. “**W-timestamp (Q)**” denotes the largest timestamp of any transaction that executed write ( $Q$ ) successfully.

2. “**R-timestamp (Q)**” denotes the largest timestamp of any transaction that executed read (Q) successfully.

These timestamps are updated whenever a new read (Q) or write(Q) instruction is executed.

Timestamp ordering protocol

The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operation is as follows.

**A. Suppose transaction  $T_i$  issues read(Q)**

- If read TS ( $T_i$ ) (9:00am) < W-timestamp (Q) (9:10 am) then  $T_i$  needs to read a value of Q that was already overwritten. Hence, the read operation is rejected and  $T_i$  is rolled back.
- If read TS ( $T_i$ ) (9:10am)  $\geq$  W-timestamp (Q) (9:00 am), then the read operation is executed and R-timestamp (Q) is set to the maximum of R-timestamp (Q) and TS( $T_i$ ).

**B. Transaction issue a write(X)**

- If write TS (T) (9:00am) < read-timestamp(X) (9:30am), this means that a younger transaction is already using the current value of the item and it would be an error to update it now. This occurs when transaction is late in doing a write and younger transaction has already read the old value.

OR

this should be done because some younger transaction with a timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

- If TS( $T_i$ ) (9:00 am) < write-timestamp(X) (8:50 am), this means transaction T asks to write any item(X) whose value has already been written by a younger transaction i.e., T is attempting to write an absolute

Value of data item X So T should be rolled back and restarted using a later timestamp.

- Otherwise, the write operation can proceed we set write-timestamp(X)=TS( $T_i$ )

This scheme is called basic timestamp ordering and guarantees that transaction is conflict serializable and the results are equivalent to a serial schedule.

**Advantages of timestamp ordering protocol**

- 1) Conflicting operations are processed in timestamp order and therefore it ensures conflict serializability.
- 2) Since timestamps do not use locks, deadlocks cannot occur.

### Disadvantages of timestamp ordering protocol

- 1) Starvation may occur if a transaction is continually aborted and restarted.
- 2) It does not ensure recoverable schedules.

#### *Thomas's Write Rule*

A modification to the basic timestamp ordering protocol is that it relaxes conflict serializability and provides greater concurrency by rejecting absolute write operation. The extension is known as Thomas's Write Rule.

Suppose transaction T1 issues read (Q) :no change, same as Time stamp ordering protocol.

If transaction issues a write(X)

- a) If  $TS(T_i) < read\_timestamp(X)$ , this means that a younger transaction is already using the current value of the item and it would be an error to update it now. This occurs when a transaction is late in doing a write and younger transaction has already read the old value.
- b) If  $TS(T_i) < write\_timestamp(X)$ . This means that a younger transaction has already updated the value of the item and the value that the order transaction is writing must be based on the **obsolete** value of the item. In this case, write operation can safely be ignored. This is sometimes known as **ignored obsolete write rule** and allows greater concurrency.

### MULTI VERSION CONcurrency CONTROL

This concurrency control technique keeps the old values of a data item when the item is updated. These are known as multi-version concurrency control, because several versions (values) of an item are maintained. When a transaction requires access to an item, an appropriate version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an older version of the item to maintain serializability. When a transaction writes an item, it writes a new version and the old version of the item is retained. Some multi-version concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multi-version techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a temporal database, which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multi-version techniques, since older versions are already maintained.

Multi-version Technique Based on Timestamp Ordering

In this method, several versions... of each data item X are maintained. For each version, the value of version and the following two timestamps are kept:

1. Read\_TS: The read timestamp of is the largest of all the timestamps of transactions that have successfully read version.
2. Write\_TS: The write timestamp of is the timestamp of the transaction that wrote the value of version.

Whenever a transaction T is allowed to execute a write\_item(X) operation, a new version of item X is created, with both the write\_TS and the read\_TS set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version Xi, the value of read\_TS() is set to the larger of the current read\_TS() and TS(T).

To ensure serializability, the following two rules are used:

1. If transaction T issues a write\_item(X) operation, and version i of X has the highest write\_TS() of all versions of X that is also less than or equal to TS(T), and read\_TS() > TS(T), then abort and roll back transaction T; otherwise, create a new version of X with read\_TS() = write\_TS() = TS(T).
2. If transaction T issues a read\_item(X) operation, find the version i of X that has the highest write\_TS() of all versions of X that is also less than or equal to TS(T); then return the value of to transaction T, and set the value of read\_TS() to the larger of TS(T) and the current read\_TS().

As we can see in case 2, a read\_item(X) is always successful, since it finds the appropriate version to read based on the write\_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T is attempting to write a version of X that should have been read by another transaction T whose timestamp is read\_TS(); however, T has already read version Xi, which was written by the transaction with timestamp equal to write\_TS(). If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created. Notice that, if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

## VALIDATION BASED CONCURRENCY CONTROL

In optimistic concurrency control techniques, also known as validation or certification techniques, no checking is done while the transaction is executing. In this scheme, updates in the transaction are not applied directly to the database items until the transaction reaches its end. During transaction execution, all updates are applied to local copies of the data items that are kept for the transaction. At the end of transaction execution, a validation phase checks whether

any of the transaction's updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

1. **Read phase:** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase:** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase:** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later. Under these circumstances, optimistic techniques do not work well. The techniques are called "optimistic" because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.

In the validation phase for transaction  $T_i$ , the protocol checks that  $T_i$  does not interfere with any committed transactions or with any other transactions currently in their validation phase. The validation phase for  $T_i$  checks that, for each such transaction  $T_j$  that is either committed or is in its validation phase, one of the following conditions holds:

1. Transaction  $T_j$  completes its write phase before  $T_i$  starts its read phase.
2.  $T_i$  starts its write phase after  $T_j$  completes its write phase, and the `read_set` of  $T_i$  has no items in common with the `write_set` of  $T_j$ .
3. Both the `read_set` and `write_set` of  $T_i$  have no items in common with the `write_set` of  $T_j$ , and  $T_j$  completes its read phase before  $T_i$  completes its read phase.

When validating transaction  $T_i$ , the first condition is checked first for each transaction  $T_j$ , since (1) is the simplest condition to check. Only if condition (1) is false is condition (2) checked, and only if (2) is false is condition (3)—the most complex to evaluate—checked. If any one of these three conditions holds, there is no interference and  $T_i$  is validated successfully. If none of these three conditions holds, the validation of transaction  $T_i$  fails and it is aborted and restarted later because interference may have occurred.



NOTESJUGAAD

## Part IV Elementary concepts of database security

### Backup and recovery techniques



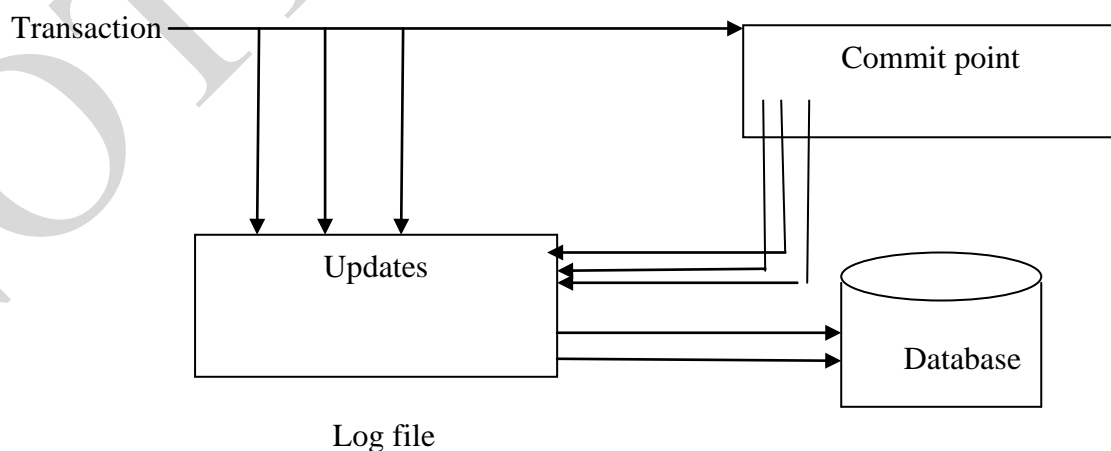
Recovery from transaction failure means that the database is restored to the most recent consistent state just before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transaction. This information is kept in system log.

Techniques used:

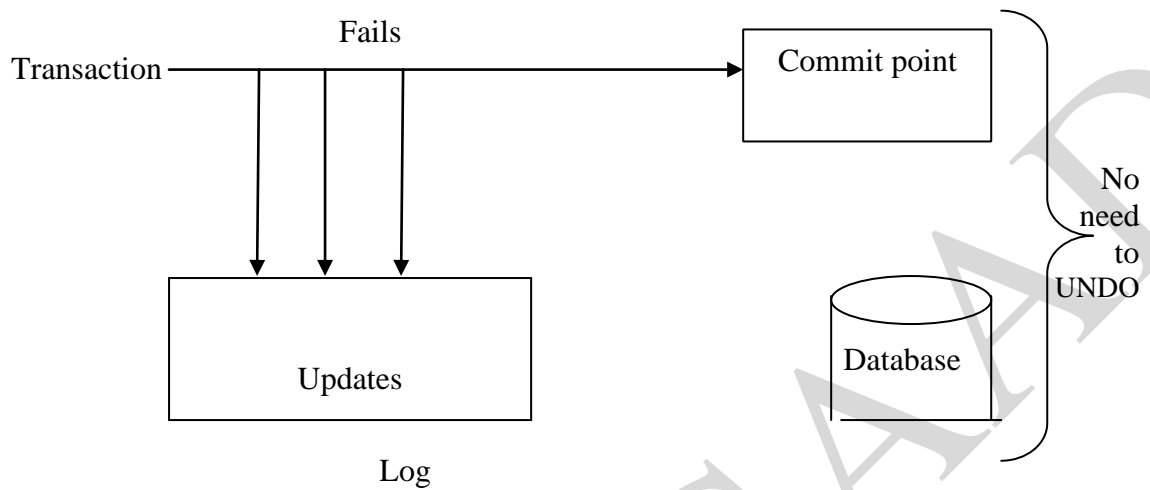
1. Deferred update( no undo/ redo algorithm)
2. Immediate update(undo/ no redo algorithm)
3. Shadow paging

### Deferred update( no undo/ redo algorithm)

These techniques defer or postpone any actual updates to the database until the transaction reaches its commit point. During transaction execution, the updates are written to the log file. After the transaction reaches its commit point, the log file is force-written to disk, then the updates are recorded in the database.



If the transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way.

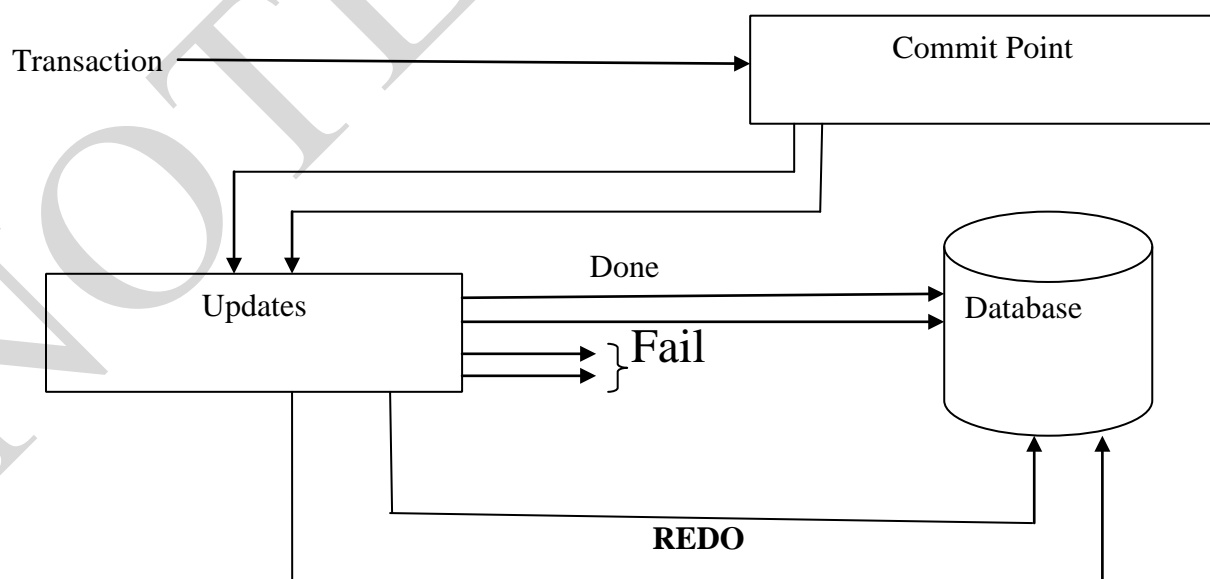


A typical deferred update protocol uses the following procedure:

- 1) A transaction cannot change the database on disk until it reaches its commit point.
- 2) A transaction does not reach its commit point until all its update operations are recorded in the log file and the log file is force-written to disk.

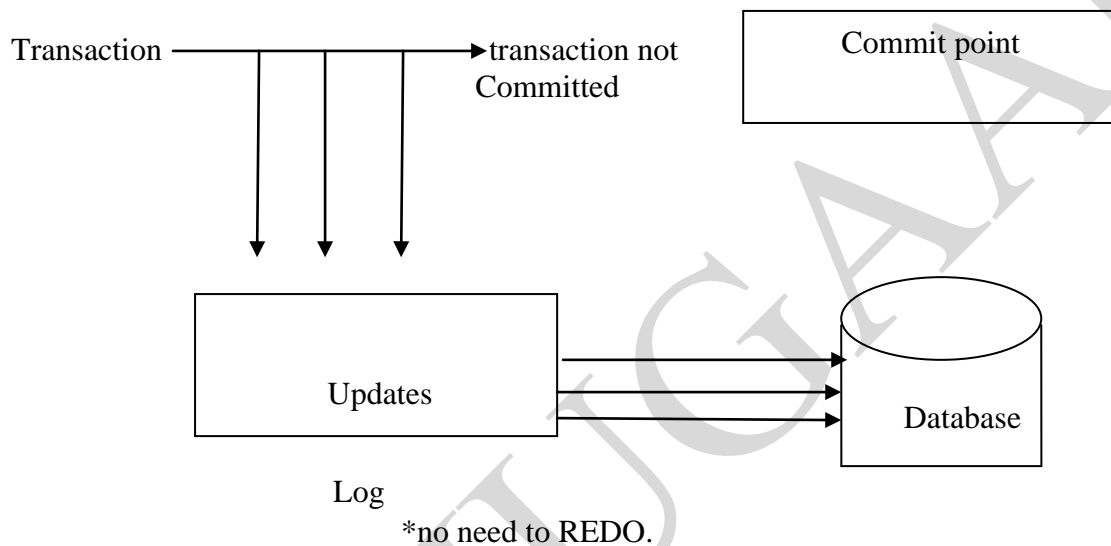
Recovery techniques based on deferred update are therefore known as NO UNDO/REDO techniques.

REDO is needed in case the system fails after a transaction commits but before all its changes are recorded on disk. In this case, the transaction operations are redone from the log file.

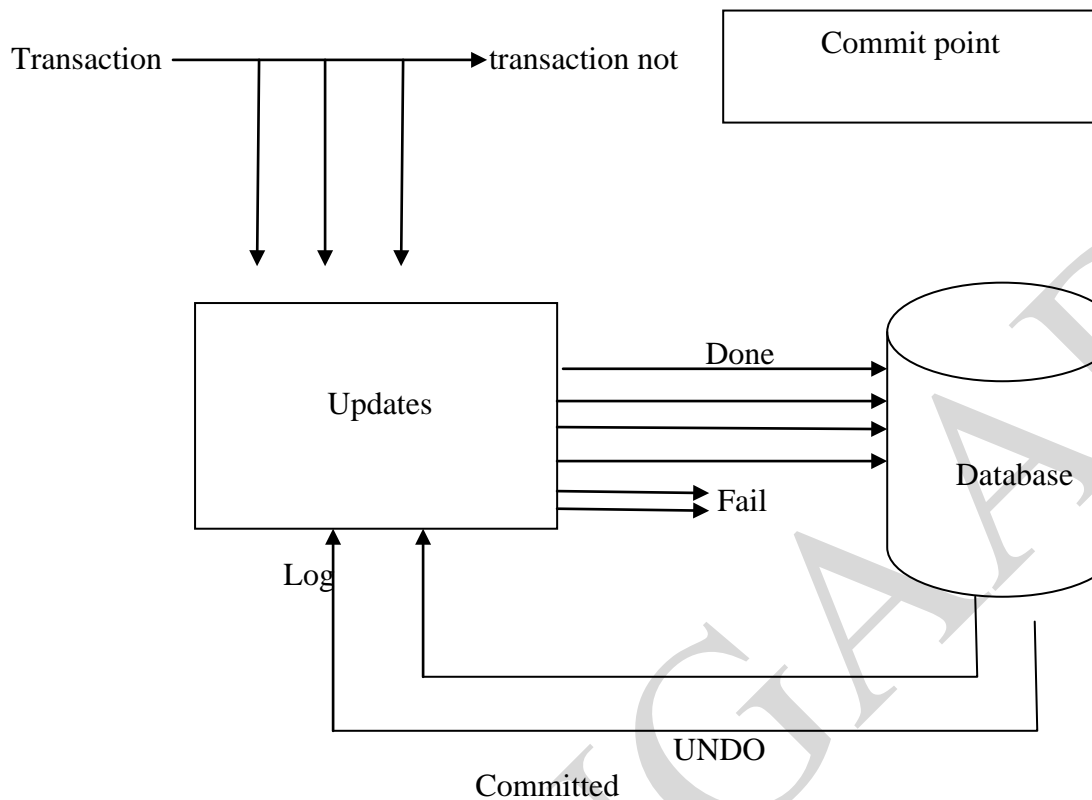


### Immediate update (undo/ no redo algorithm)

In this technique, when a transaction issues an update command the database can be updated immediately without any need to wait for transaction to reach its commit point. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk before the transaction commits, there is no need to redo any operations of committed transaction.



Provisions must be made for undoing the effect of update operation that have been applied to the database by a failed transaction. This is done by rolling back the transaction and undoing the effect of the transaction write operation.



### Shadow Paging

- This technique does not require LOG in single user environment
- In multi-user may need LOG for concurrency control method
- **Shadow paging considers**
  - The database is partitioned into fixed-length blocks referred to as **PAGES**.
  - **Page table** has n entries – one for each database page.
  - Each **contain pointer to a page on disk** (1 to 1st page on database and so on...).

The **idea is to maintain 2 pages tables** during the life of transaction.

- The current page table in main memory / volatile storage medium
- The shadow page table in non volatile storage medium

When **transaction starts**, both **page tables are identical**

- The shadow page table is never changed over the duration of the transaction.
  - The current page table may be changed when a transaction performs a write operation.
  - All input and output operations use the current page table to locate database pages on disk.

During transaction execution, all updates are performed using the current directory and the shadow directory is never modified.

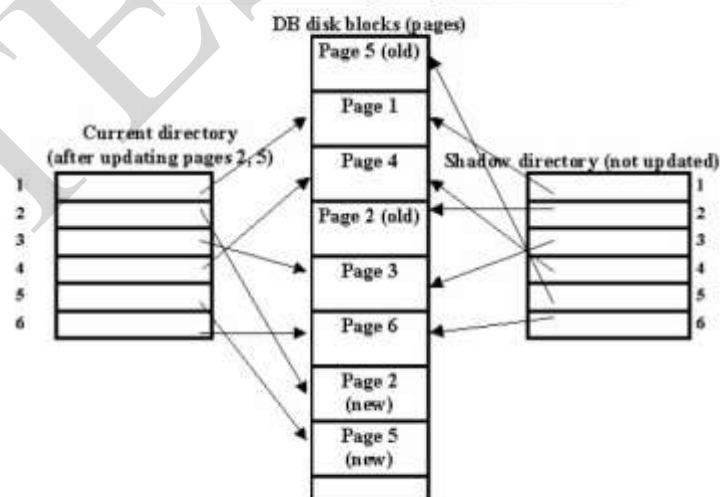
When a **write\_item operation is performed**

- A **new copy** of the modified DB page **is created** and the **old copy is not overwritten**.

Two versions, of the pages updated by the transaction, are kept.

- The new page is written elsewhere on some unused disk block.
- The current directory entry is modified to point to the new disk block.
- The shadow directory is not modified.

### Shadow Paging (cont'd)



### Advantages

- No Overhead for writing log records.
- No Undo / No Redo algorithm. □ Recovery is faster.

## Disadvantages

- Data gets fragmented or scattered.
- Hard to extend algorithm to allow transaction to run concurrently.

## Authentication and authorization

*Authentication* is the mechanism whereby systems may securely identify their users. Authentication systems provide an answer to the questions:

- Who is the user?
- Is the user really who he/she represents himself to be?

An authentication system may be as simple (and insecure) as a plain-text password challenging system. In order to verify the identity of a user, the authenticating system typically challenges the user to provide his unique information (his password, fingerprint, etc.) -- if the authenticating system can verify that the shared secret was presented correctly, the user is considered authenticated.

*Authorization*, by contrast, is the mechanism by which a system determines what level of access a particular authenticated user should have to secure resources controlled by the system. For example, a database management system might be designed so as to provide certain specified individuals with the ability to retrieve information from a database but not the ability to change data stored in the database, while giving other individuals the ability to change data. Authorization systems provide answers to the questions:

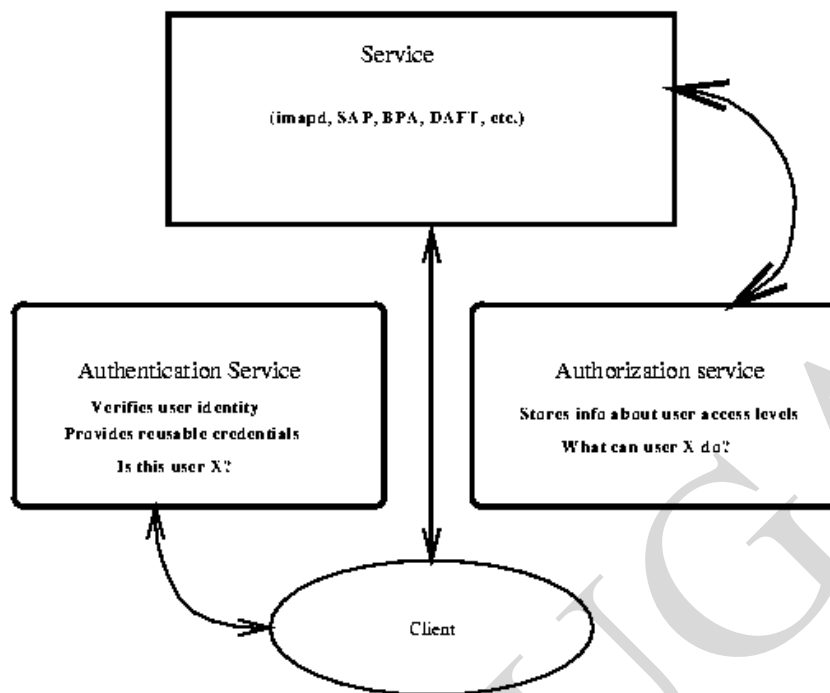
- Is user X authorized to access resource R?
- Is user X authorized to perform operation P?
- Is user X authorized to perform operation P on resource R?

Authentication and authorization are somewhat tightly-coupled mechanisms -- authorization systems depend on secure authentication systems to ensure that users are who they claim to be and thus prevent unauthorized users from gaining access to secured resources.

*Figure I*, below, graphically depicts the interactions between arbitrary authentication and authorization systems and a typical client/server application.



## Authentication vs. Authorization



In the diagram above, a user working at a client system interacts with the authentication system to prove his identity and then carries on a conversation with a server system. The server system, in turn, interacts with an authorization system to determine what rights and privileges the client's user should be granted.

**Authentication:** Prove it.  
**Authorization:** Here is what you can do.