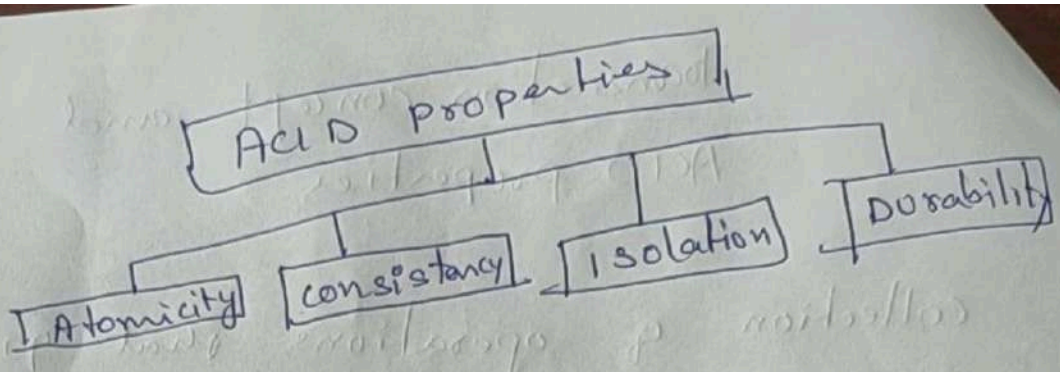
Transaction concept and ACID properties

collection of operations that form a single logical unit of work are called transactions.

A transaction is a unit of program execution that accesses and possibly updates various data items.
The transaction consist of all operations executed between the begin transaction and end transaction.

To ensure integrity of the data, the database system maintain the following properties of the transaction. These properties are called ACID properties.

(1)

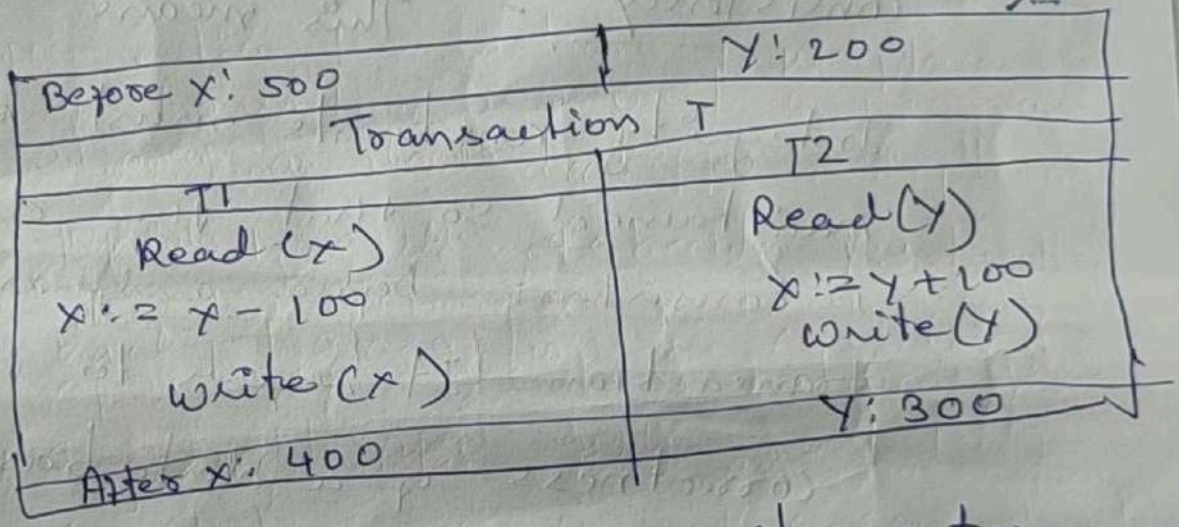


(1) Atomicity :- This means that either the entire transaction take place at once or doesn't happen at all. Transaction do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

- Abort :- If a transaction aborts, changes made to the database are not visible.
- commit :- If a transaction commits, changes made are visible.

Atomicity is also known as the "All or nothing rule"

Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account x to account y



If the transaction fails after completion of T1 but before completion of T2. say after write(X) and 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 :- This means that Integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database.

Referring the example above,

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.

Isolation :- Even though multiple ~~can occur~~ occur, the transaction 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.

This property ensures that the execution of transactions concurrently will result in a state that is

equivalent to a state achieved there were executed serially in some order.

let x=500, y=500

consider two transaction T_1 and T_2.

T_1	T_2
Read(x) x := x * 100 write(x) Read(y) y := y - 50 write(y)	Read(x) Read(y) z := x + y write(z)

Suppose T_2 has been executed till Read(y) and then T_2 starts. As a result, interleaving of operations take place due to which T_2 reads the correct value of x but the incorrect

Value of y and sum computed by

T_2 (x+y = 50,000 + 500 = 50,500)

is thus not consistent with the sum at end of the transaction.

T (x+y = 5,000 + 450 = 50,450)

This results in database inconsistency due to a loss of 50 units. Hence transaction must take place in isolation and changes should be visible only after they have been made to the main memory.

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 may 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,

Concurrent Executions

concurrent execution refers to the simultaneous execution of more than one transaction.

There are two good reasons for allowing concurrency

(1) Improved throughput and resource utilization! - A transaction may contain I/O activity or cpu activity. The cpu and I/O activity in computer system can operate in parallel.

Read or write operations of one transaction is in progress on one disk, another transaction can be running in the cpu, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the

throughput of the system. correspondingly the processor and disk utilization also increases.

(b) Reduced waiting time :-

If the transactions are operating on different parts of the database it is better to let them run concurrently, sharing the cpu cycles and disk access among them.

Here schedule 1 - a serial schedule in which T_1 is followed by T_2

A=1000
B=2000

T_1	T_2
read(A)	
A := A-50	
950 write(A)	
read(B)	
B := B+50	read(A)
2050 write B	temp := A * 0.1

T_1	T_2	
	A := A - temp	A=855
	write (A)	
	read (B)	2050
	B := B + temp	2145
	write (B)	

So final A=855 B = 2,145

Suppose the current values of account A and B are 1000 and 2000 respectively. Here Transaction T_1 tranfers 50 Rs from A to account B. And Transaction T_2 transfers 10% of balance from account A to account B.

The final values of accounts A and B after execution of T_1 and T_2 transactions in order are 855 and 2145 respectively.

sum $855 + 2145 = 3000$ is preserved after the execution of both transaction.

Similarly if the transactions are executed one after time in order T_2 followed by t_1 then again expected, the sum $A + B$ is preserved.

(" C schedules - a serial schedule in which T2 after T1)

T_1	T_2
	read (A)
	temp = A * 0.1
	A := A − temp
	write (A)
	Read (B)
	B := B + temp
	write (B)
read (A)	
A := A − 50	
write (A)	
read (B)	
B := B + 50	
write (B)	

schedule 3 - a concurrent schedule equivalent to schedule 1

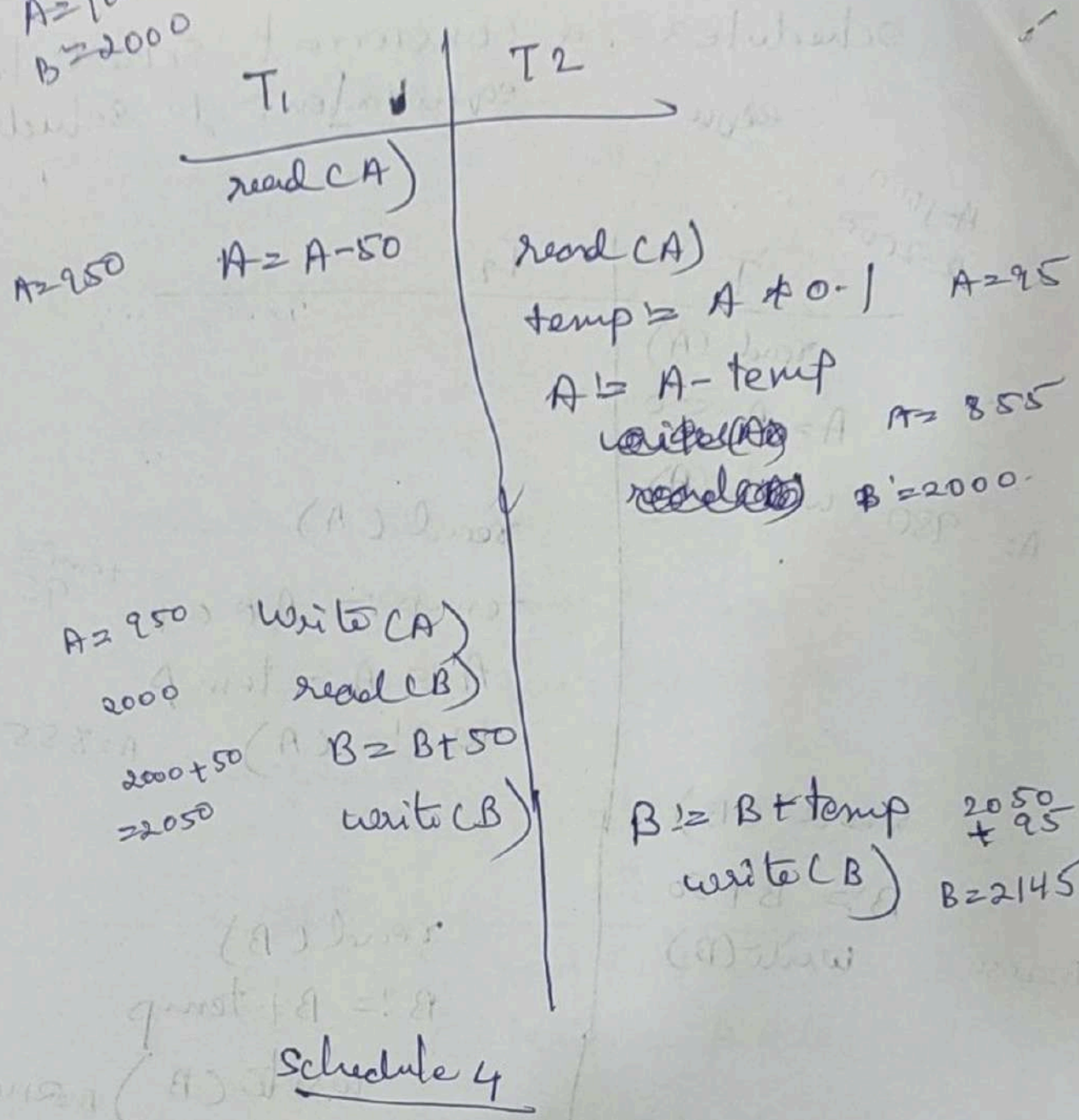
A=1000
B=2000

T_1	T_2
read (A)	
A = A - 50	
write (A) A=950	
	read (A)
	temp := A * 0.1 temp=95
	A := A - temp
	write (A) A=855
read (B)	
B := B + 50	
write (B) B=2050	
	read (B)
	B := B + temp
	write (B) B=2145

schedule 3 a concurrent schedule equivalent to schedul.

855 + 2145 = 3000

A=1000
B=2000



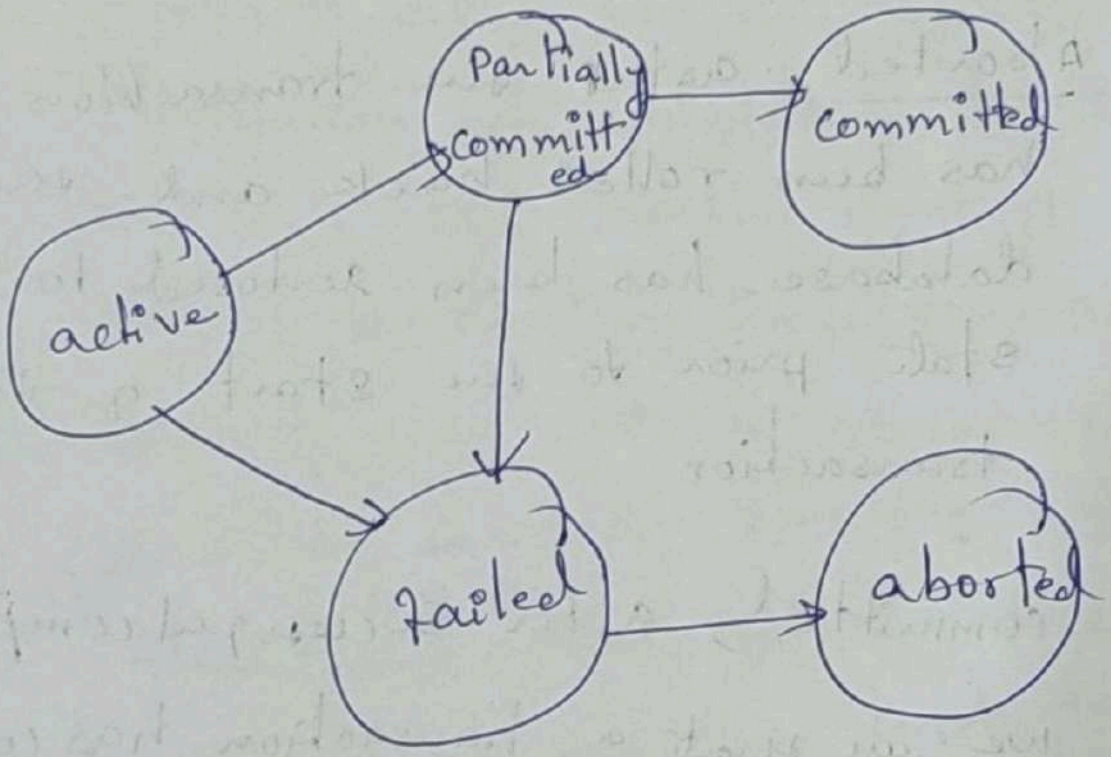
Schedule 4

After the execution of this schedule we arrive at a state when the final accounts A and B are 950 and 2145 respectively. So final state is an inconsistent sate.

950 + 2145 = 3095 inconsistent state.

Transaction state

state diagram of a transaction.



A transaction must be in one of the following states:

(1) Active :- the initial state; the transaction stays in this state while it is executing

(2) Partially committed; after the final statement has been executed

(3) Failed :- after the discovery that normal execution can no longer proceed.

(4) Aborted, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction

(5) committed, after successful completion we say that a transaction has comitted only if it has entered the committed state. Similarly, we say that a transaction has aborted only if It has entered the aborted state. A transaction is said to have terminated if it has either committed or aborted.

A transaction starts in the active state. when it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure occurs then transaction ~~may~~ ~~leads to~~ may not be successfully complete.

The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. when the last of this

information is written out, the transaction enters the committed state

Implementation of Atomicity and Durability

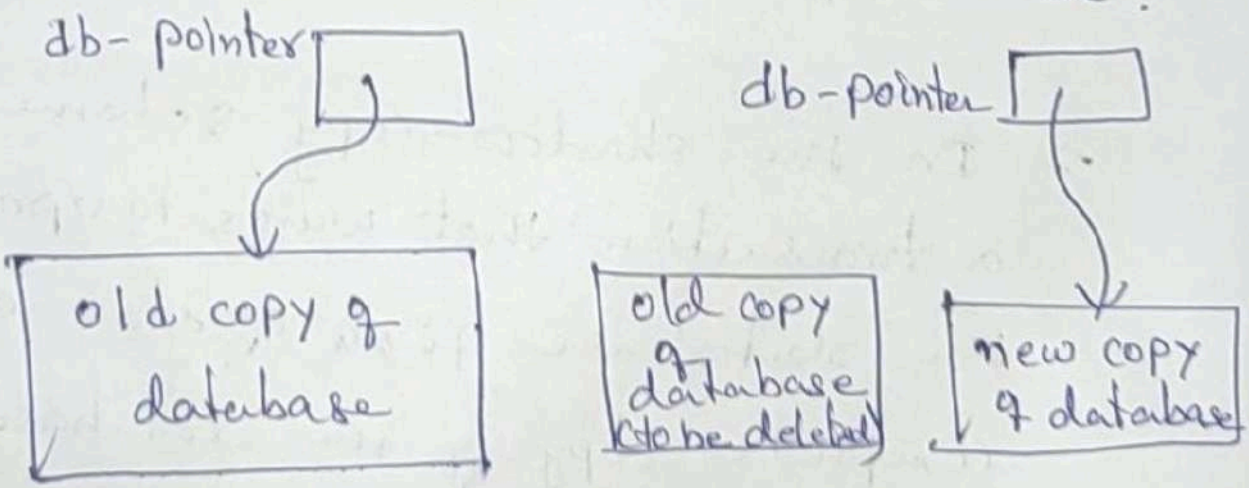
→ The recovery-management component of a database system can support atomicity and durability by a variety of schemes.

shadowcopy
shadow copy scheme is the simplest scheme.

→ In the shadow-copy scheme, a transaction that wants to update the database first creates a complete copy of the database. All updates are done on the new database copy, leaving the original copy, the shadow copy, untouched. If at any

Point the transaction has to be aborted, the system deletes the new copy. The old copy of the database has not been affected.

The scheme also assumes that the database is simply a file on disk. A pointer called db-pointer is maintained on disk. It points to the current copy of the database.



(a) Before update (b) After update

Shadow-copy technique for atomicity and durability.

If the transaction completes, it is committed as follows.

(a) First, the operating system is asked to make sure that all pages of the new copy of the database have been written out to disk.

(b) After the operating system has written all the pages to disk, the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the current copy of the database. The old copy of the database is then deleted.

Implementation of Isolation

A DBM's isolation feature ensures that several transactions can take place simultaneously and that no data from one database should have an impact on another. In other words, the process on the second state of the database will start after the operation on the first state is finished.

Implementation of isolation typically involves concurrency control mechanisms. Here are common mechanisms used

(1) Locking Mechanisms

Locking ensures exclusive access to a Data item for a transaction. This means that while one transaction holds a lock on a data item, no other transaction can access that item.

→ Shared Lock (S-Lock): Allows a transaction to read an item but not write to it.

→ Exclusive Lock (X Lock):

Allows a transaction to read and write an item. No other transaction can read or write until the lock is released.

→ Two phase Locking (2PL):

This protocol ensures that a transaction acquires all the locks before it releases any. This results in a growing phase (acquiring Locks and not releasing any) and a shrinking phase (releasing locks and not acquiring any).

2- Timestamp-based protocols

Every transaction is assigned a unique timestamp when it starts. This timestamp determines the order of transactions. Transactions can only acces the database if they respect the timestamp oder, ensuring old transactions get priority.

Serializability

The database Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database. because all non-serial schedules are not correct, some may lead to inconsistency of database.

There are two types of serializability

1. conflict serializability.
2. view. Serializability

conflict serializability :-

In a given non-serial schedule can be converted into a serial schedule by swapping its

non conflicting operations then it is called as conflict serializable schedules.

→ while swapping the order of execution of two conflicting operations cannot be changed because if they are applied in different order, they can have different effect on the database or on the other transactions in the schedule.

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

Conflict Serializability

Let us consider a schedule S in which there are two consecutive instructions I_i and I_j of transactions T_i and T_j respectively ($i \neq j$).

(1) If I_i and I_j refer to different data items, then we can swap I_i and I_j without affecting the results of any Instruction in the schedule.

(2) But if I_i and I_j refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions there are four cases that we need to consider.

(a) I_i = read(Q) and I_j = read(Q)

The order of I_i and I_j does not

matter, since the same value of Q is read by T_i and T_j regardless of the order.

(b) I_j = read(Q), I_j = write(Q). If I_i comes before I_j, then T_i does not read the value of Q that is written by T_j in instruction I_j. If I_j comes before I_i, then T_i reads the value of Q that is written by T_j. Thus the order of I_i and I_j matters.

(c) I_i = write(Q), I_j = read(Q)

The order of I_i and I_j matters for reasons similar to those of the previous case.

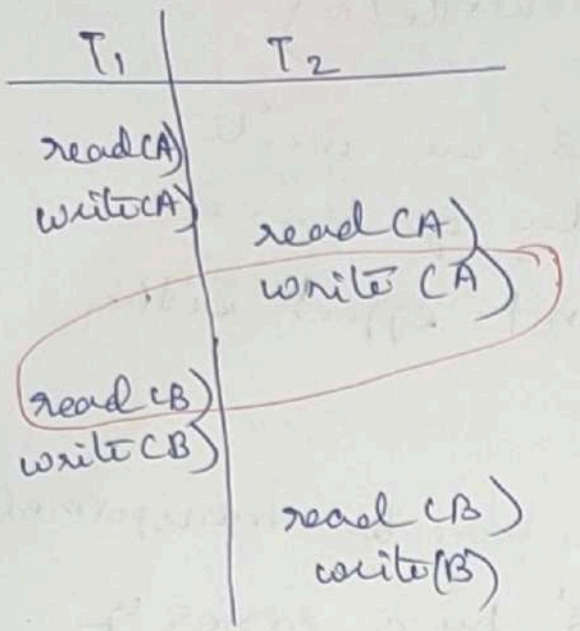
d) I_i = write (a), I_j = write (a)

Since both instructions are write operations, the order of these instructions does not effect either T_i or T_j;

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 <u>conflict equivalent</u>.

The concept of <u>conflict equivalence</u> leads to the concept of conflict serializability we say that a given schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

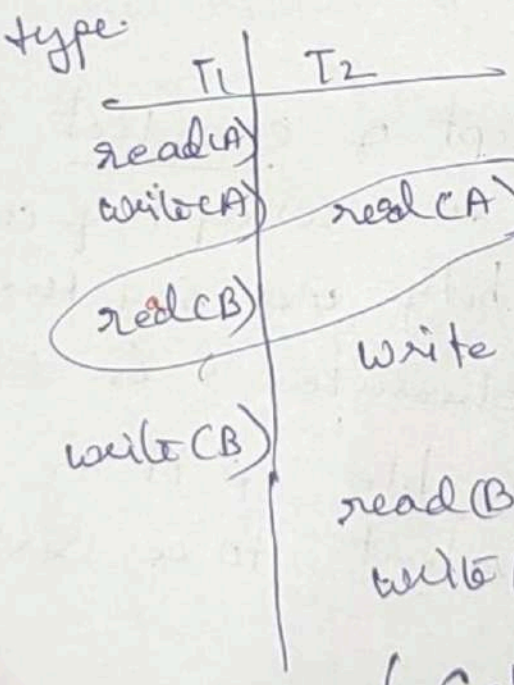
S1



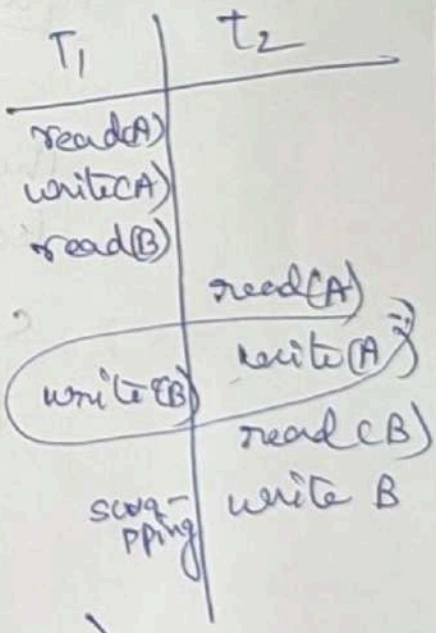
Swapping

(Schedule 1)

Here write(A) of T_2 and read(B) of T1 we can swap as there is no conflict because of different data type.



swapping



(Schedule 2)

So schedule (1) and schedule 2 both produce the same final system state.

So we say that schedule 1 and schedule 2 are conflict equivalent

S2

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

schedule-3

Now this schedule 3 is conflict serializable.

View Serializability

Consider two seechedules s and s' where the same set of transactions participates in both schedules. The schedules s and s' are said to be view equivalent if

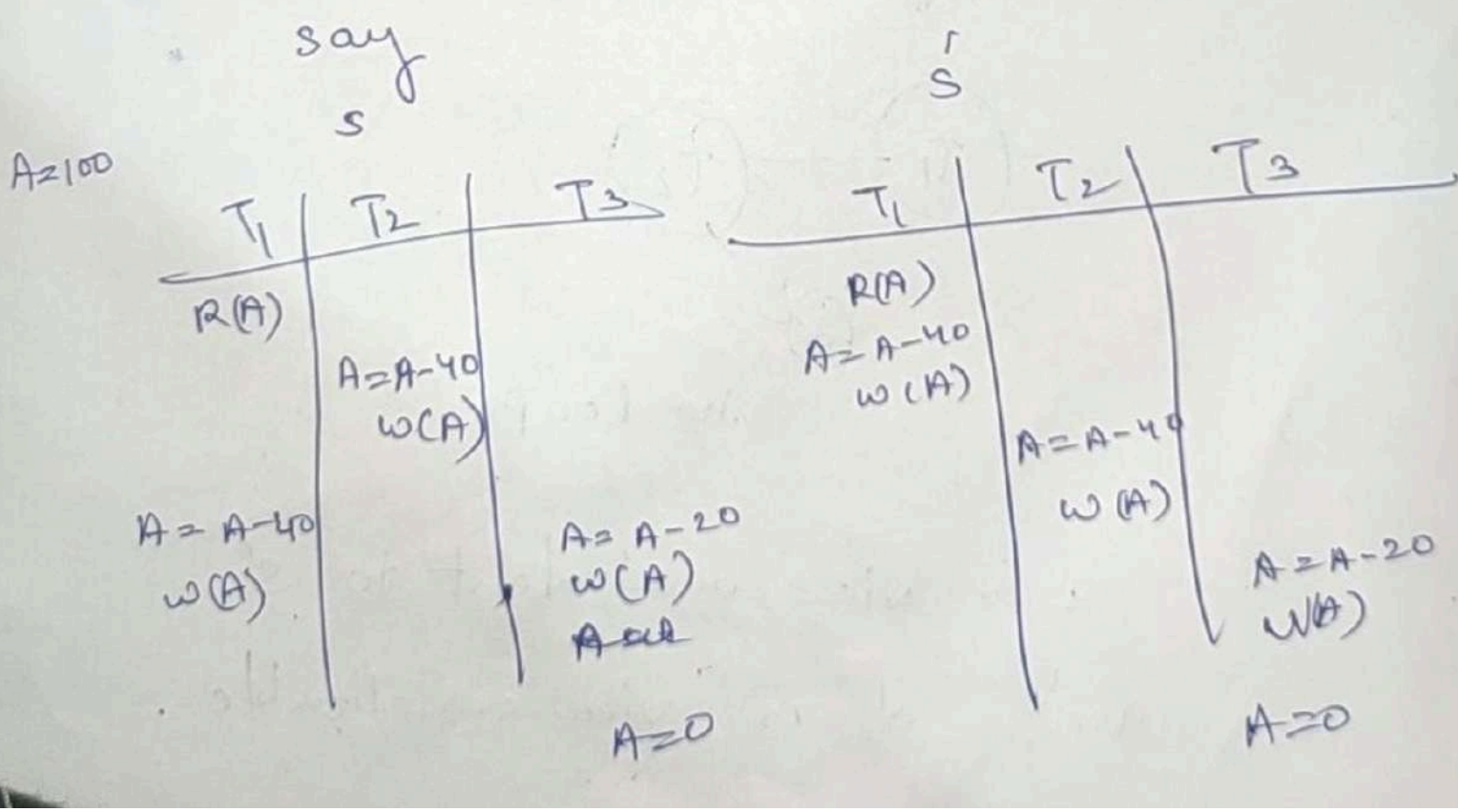
three conditions are met

(1) For each data item Q, if transactions T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S' also read the initial value of Q.

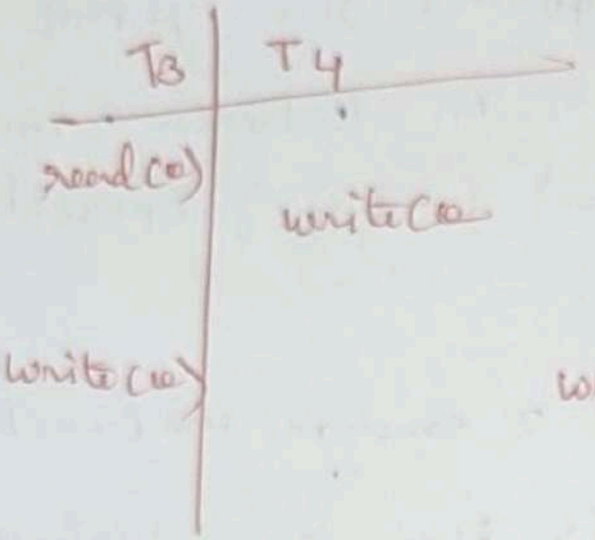
(2) For each data item Q, if transaction T_i executes read(Q) in schedule S, and if that was produced by a write(Q) operation executed by transaction T_j, then the read(Q) operation of transaction T_i must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of transaction T_j.

For each data item Q, the transaction (if any) that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'.

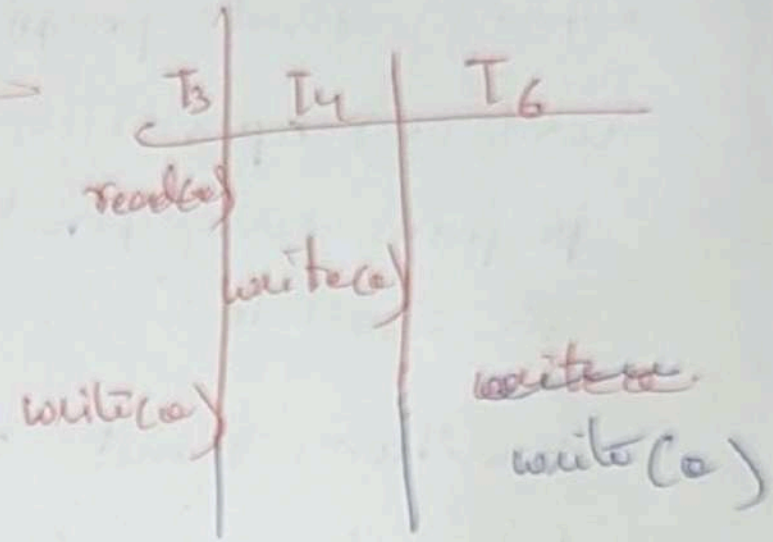
Condition 1 and 2 ensure that each transaction reads the same values in both schedules. and therefore performs the same computation. condition 3, coupled with conditions 1 and 2 ensures that both schedules result in the same final system state.



Schedule1

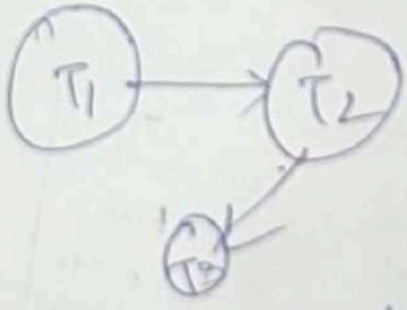


Schedule2



Here schedule 2 is view serializable

so we say schedule S' is view serializable



No Loop

S is view equivalent to S'

and S' is view serializable.

Testing for Serializability

when designing concurrency control, we must show that schedules generated by the scheme are serializable.

A simple and efficient method for determining conflict serializability of a schedule. we go for precedence graph. The graph consist of a pair $G = (V, E)$ where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consist of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

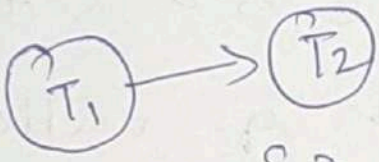
(1) T_i executes write(Q) before T_j executes read(Q).

2. T_i executes read(a) before T_j executes write(a).

(3) T_i executes write(a) before T_j executes write(a).

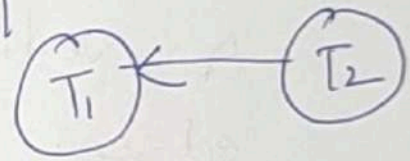
S1

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)



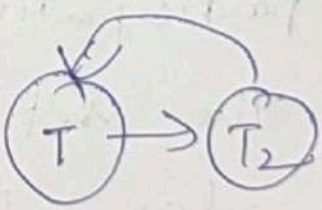
S2

T_1	T_2
	R(A)
	W(A)
R(A)	
W(A)	



S3

T_1	T_2
R(A)	
	R(A)
	W(A)
W(A)	



If the precedence graph for S3 has a cycle then schedule S is not conflict serializable.

If the graph contains no cycle, then the schedule S is conflict serializable.

Concurrency control

Concurrency control is a crucial database management system (DBMS) component. It manages simultaneous operations without them conflicting with each other. The primary aim is maintaining consistency, integrity and isolation when multiple users or applications access the database simultaneously.

Problems with concurrent Execution

In a database transaction, the two main operations are Read and write operations. So there is a need to manage these two operations in the concurrent execution.

Problem ① Lost update problems

(W - W conflict)

The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

For example

Consider the below diagram where two transactions T_x and T_y are performed on the same account A where the balance of account A is 300

let :
A = 300

Time	T_x	T_y
t_1	Read(A)	–
t_2	A = A-50	
t_3	–	Read(A)
t_4	–	A = A+100
t_5	–	–
t_6	write(A)	–
t_7		write(A)

→ At time t_1 transaction T_x reads the value of account A i.e 300

→ At time t_2 transaction T_x deducts 50 Rs from account A that becomes 250 Rs

→ Alternately at time time t_3, Transaction T_y reads the value of account A that will be Rs 300

only because Tx didn't update the va
yet.

→ At time t_4 transaction T_y adds Rs 100 to account A that becomes 400.

→ At time t_6, transaction Tx writes the values of account A that will be updated as 250 Rs only as Ty didn't update yet.

→ Similarly at time t_7, transaction Ty writes the values of account A, so it will write as done at time t_4 that will be $400.
It means the value written by Tx is lost is Rs 250 is lost.
Hence data becomes incorrect, and database sets to inconsistent.

(2) Dirty Read problems (W-R conflict)

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rolled back, the updated database item is accessed by accessed by another transaction. Then comes write-Read conflict between both transactions.

Food example.

Consider two transactions T_x and T_y in the below diagram performing read/write operations on account A where the available balance is account A is 300

Time	T_x	T_y
t_1	Read (A)	–
t_2	A = A + 50	
t_3	write (A)	
t_4	–	Read (A)
t_5	Server down Roll back	–

→ At time t_1, transaction T_x reads the value of account A, ie 300

→ At time t_2, transaction T_x adds 50 to account A that becomes 350

→ At time t_3 transaction T_x writes the updated value in account A ie 350 Rs

→ Then at time t_4, transaction T_y reads account A that will be read as 350 Rs

4

→ Then at time t_5, transaction T_y rollbacks due to server problem, and the value changes back to 300 (as initially).

→ But the value for account A remains 350 Rs for transaction T_y as committed. which is the dirty read and therefore known as Dirty Read problem

(3) Unrepeatable Read problem (RW-R conflict)

- Also known as inconsistent Retrieval problem that occurs when in a transaction, two different values are read for the same database item.

Consider two transaction T_x and T_y performing the read/write

operations on account A, having an available balance = 300 Rs

Time	T_x	T_y
t_1	Read(A)	—
t_2	—	Read(A)
t_3	—	A = A + 100
t_4	—	write(A)
t_5	Read(A)	—

→ At the time t_1, transaction T_x reads the value from account A ie 300 Rs

→ At the time t_2, transaction T_y reads the value the value from account A, ie 300.

→ At the time t_3, Transaction T_y updates the value of account A by adding 100 Rs to the available

balance and then it becomes 400

→ At time t_4 transaction T_y writes the updated value i Rs 400

→ After that, at time t_5, transaction T_y reads the available value of account A, and that will be read as 400.

→ It means that within the same transaction T_x, it reads two different values of account A, i 300 initially, and after updation made by transaction t_y, it reads 400 Rs. It is an unrepeatable read and is therefore known as the unrepeatable read problem.