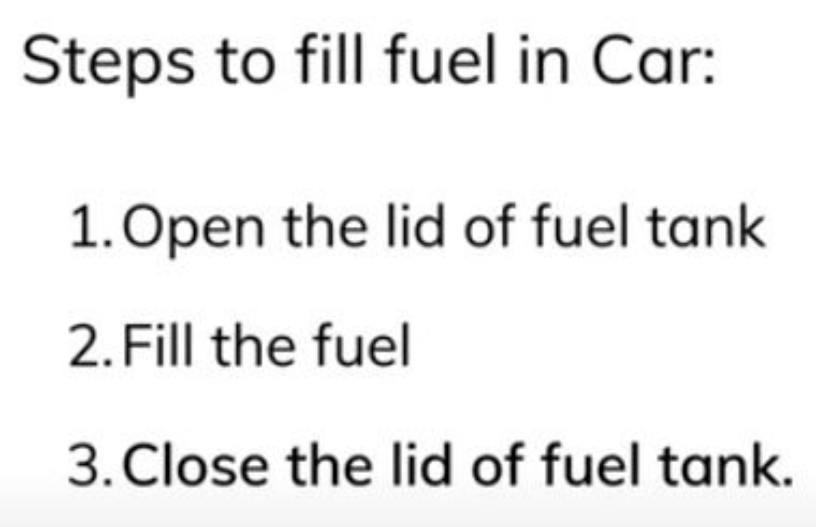
Concurrency Control

Lock-based protocol

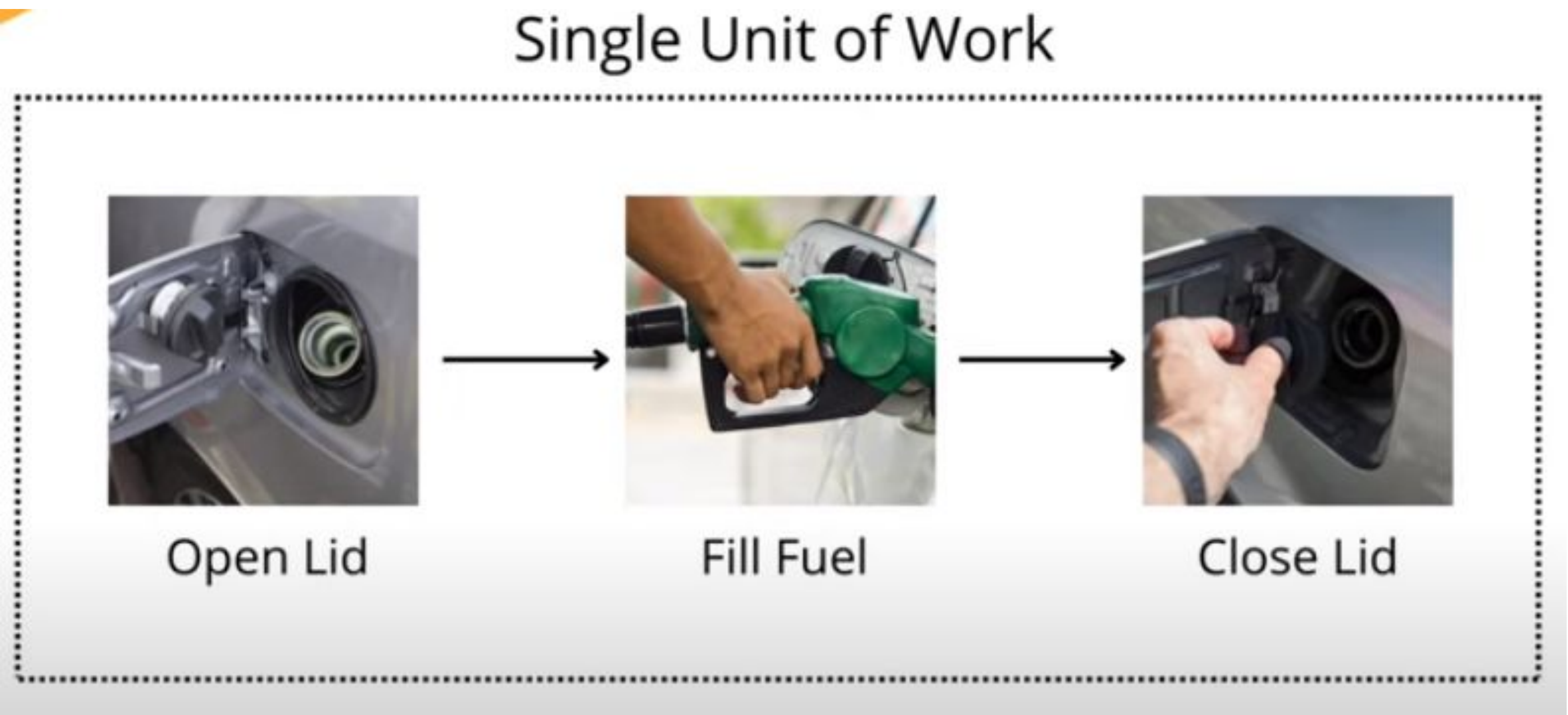
Transaction

- The transaction is a **set of logically related operation**. It contains a **group of tasks**.
- A transaction is a **single logical unit of work** which accesses and **possibly modifies** the **contents of a database**.
- Transactions access data using **read and write operations**.
- In order to **maintain consistency** in a database, **before and after** the transaction, **certain properties are followed**.



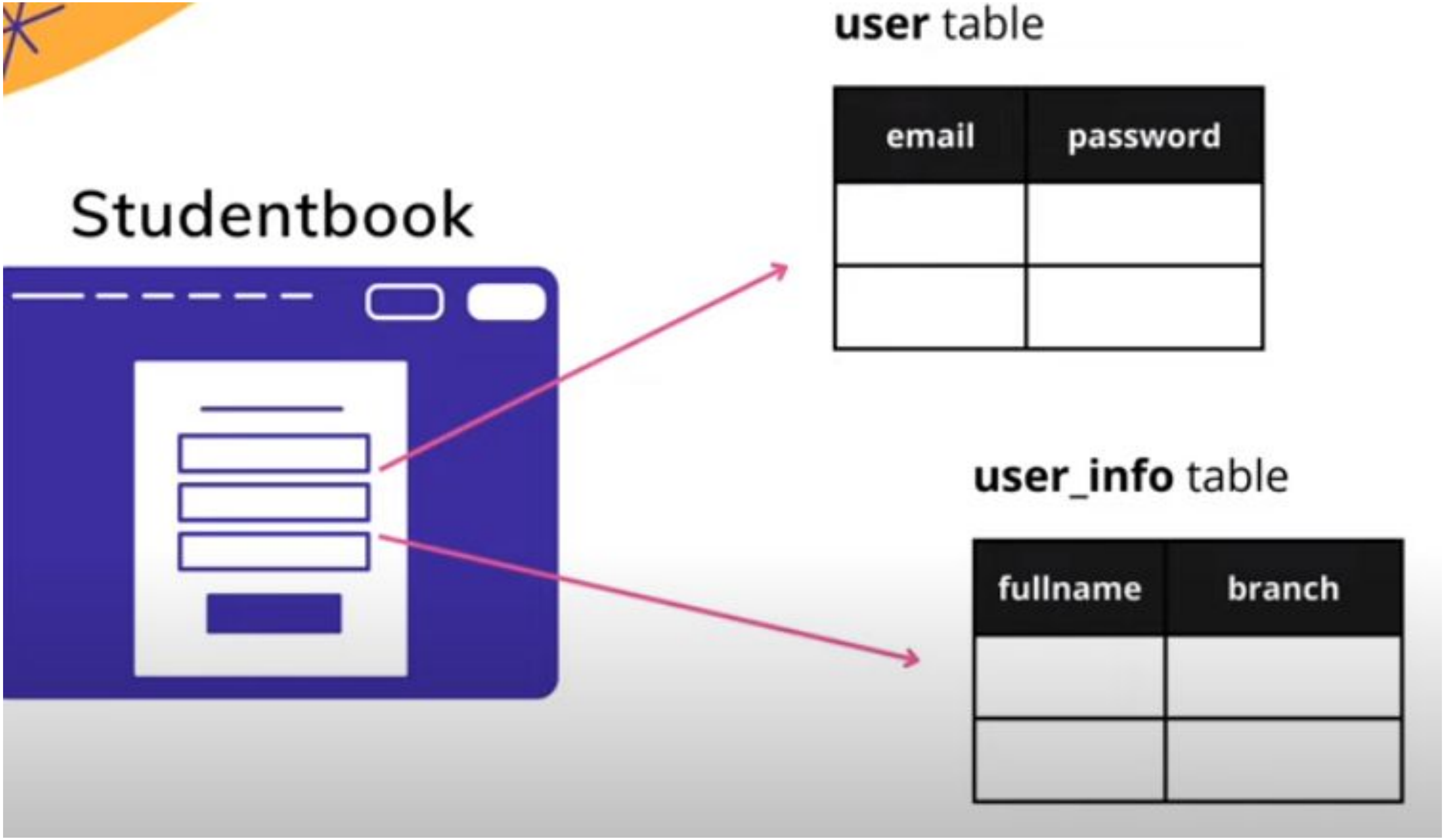
Steps to fill fuel in Car:

1. Open the lid of fuel tank
2. Fill the fuel
3. Close the lid of fuel tank.



Single Unit of Work

Can’t miss any of the single step



Studentbook

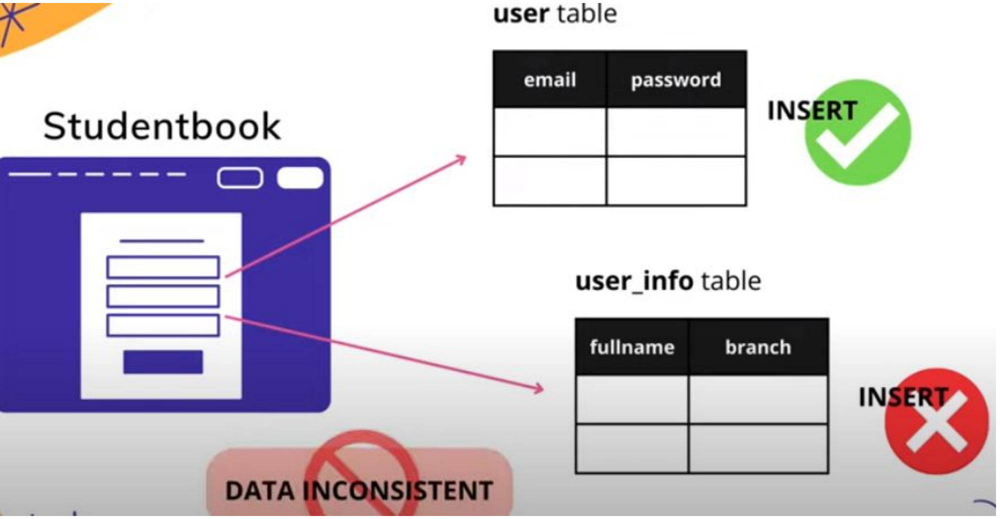
user table

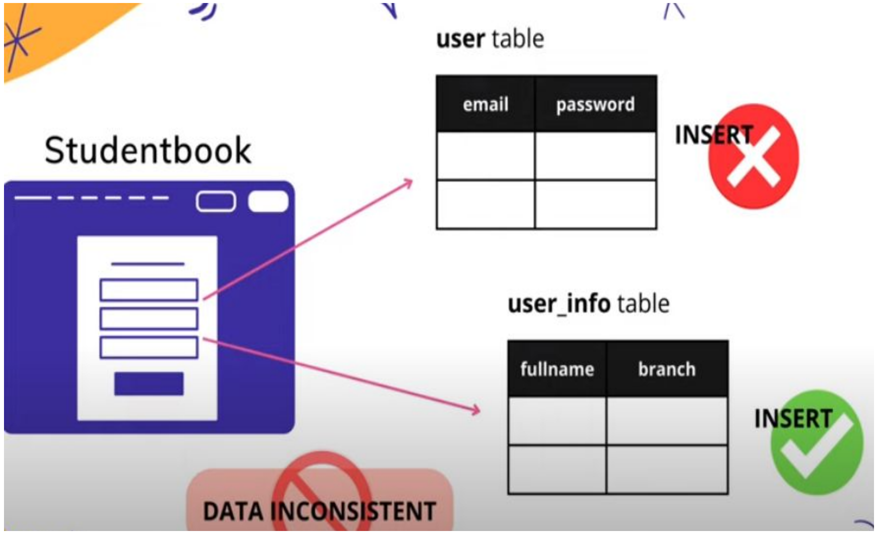
email	password

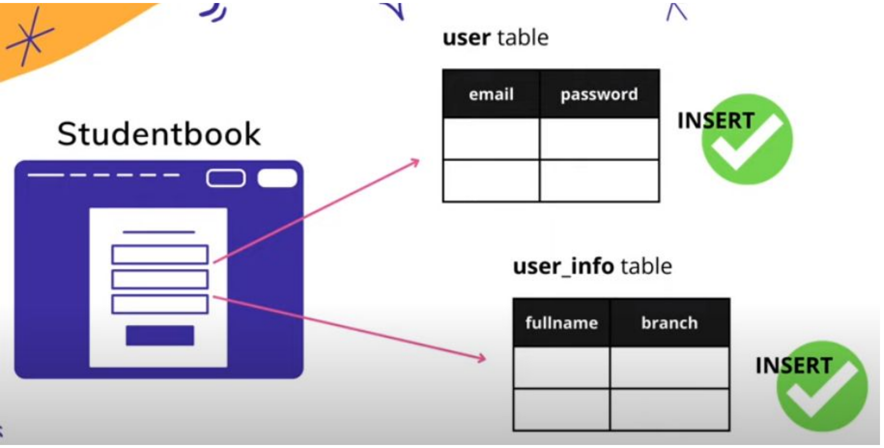
user_info table

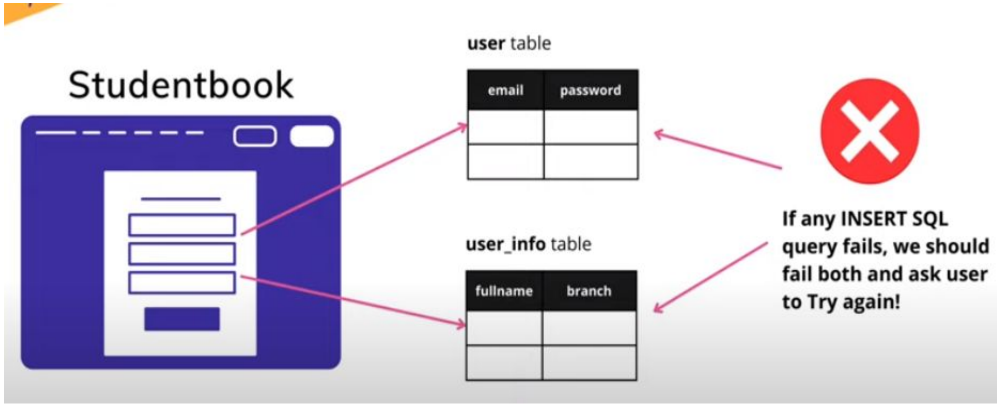
fullname	branch

- Both the queries to run successfully for successful transaction

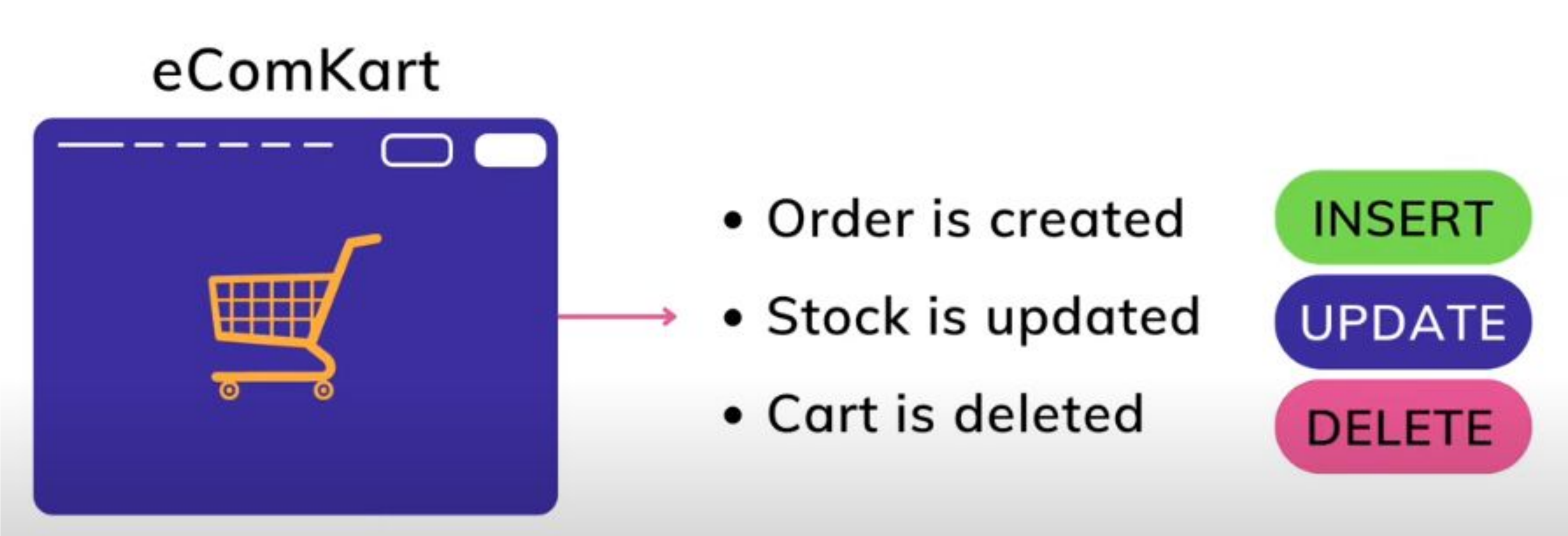






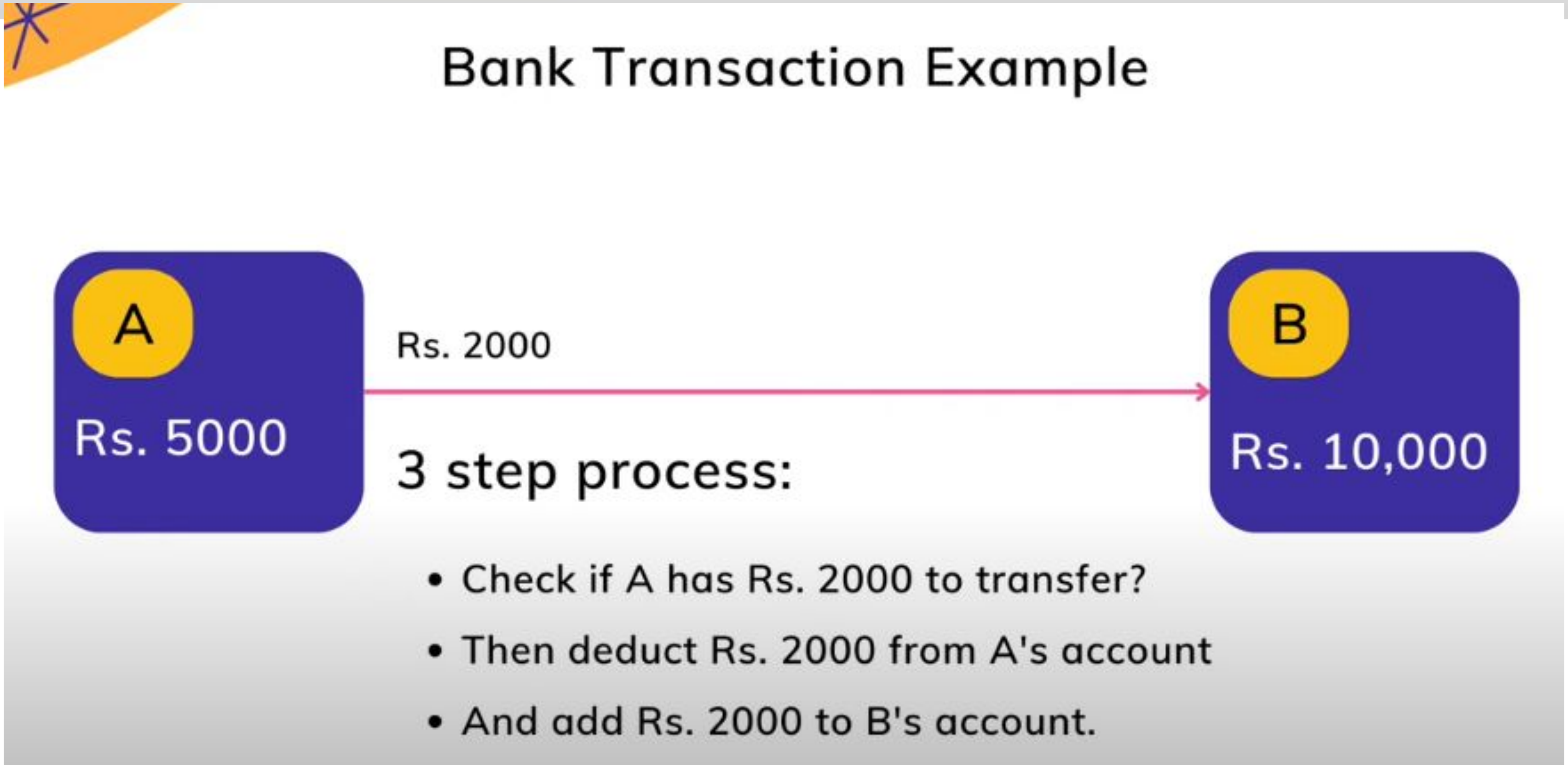


If any INSERT SQL query fails, we should fail both and ask user to Try again!



eComKart

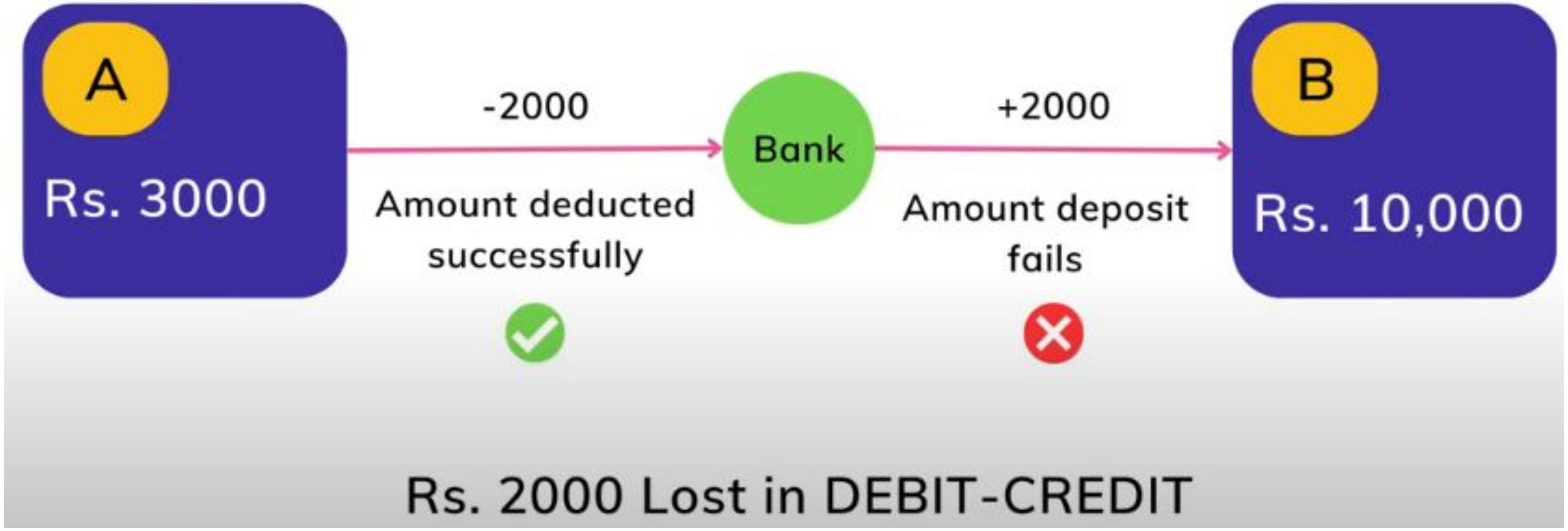
- Order is created — INSERT
- Stock is updated — UPDATE
- Cart is deleted — DELETE



Bank Transaction Example

3 step process:

- Check if A has Rs. 2000 to transfer?
- Then deduct Rs. 2000 from A's account
- And add Rs. 2000 to B's account.



A

Rs. 3000

-2000

Amount deducted successfully

✓

Bank

+2000

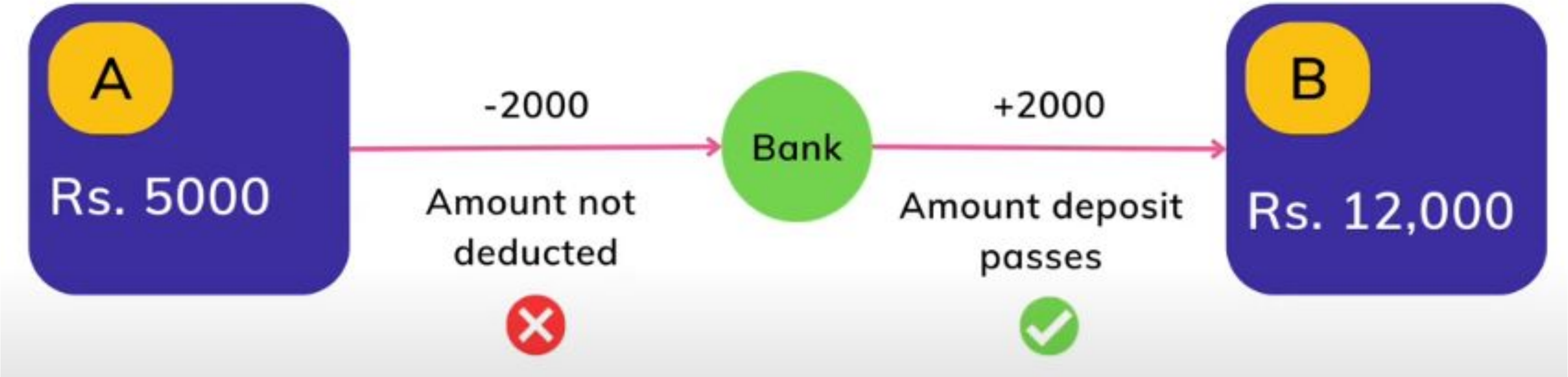
Amount deposit fails

✗

B

Rs. 10,000

Rs. 2000 Lost in DEBIT-CREDIT



A

Rs. 5000

-2000

Amount not deducted

✗

Bank

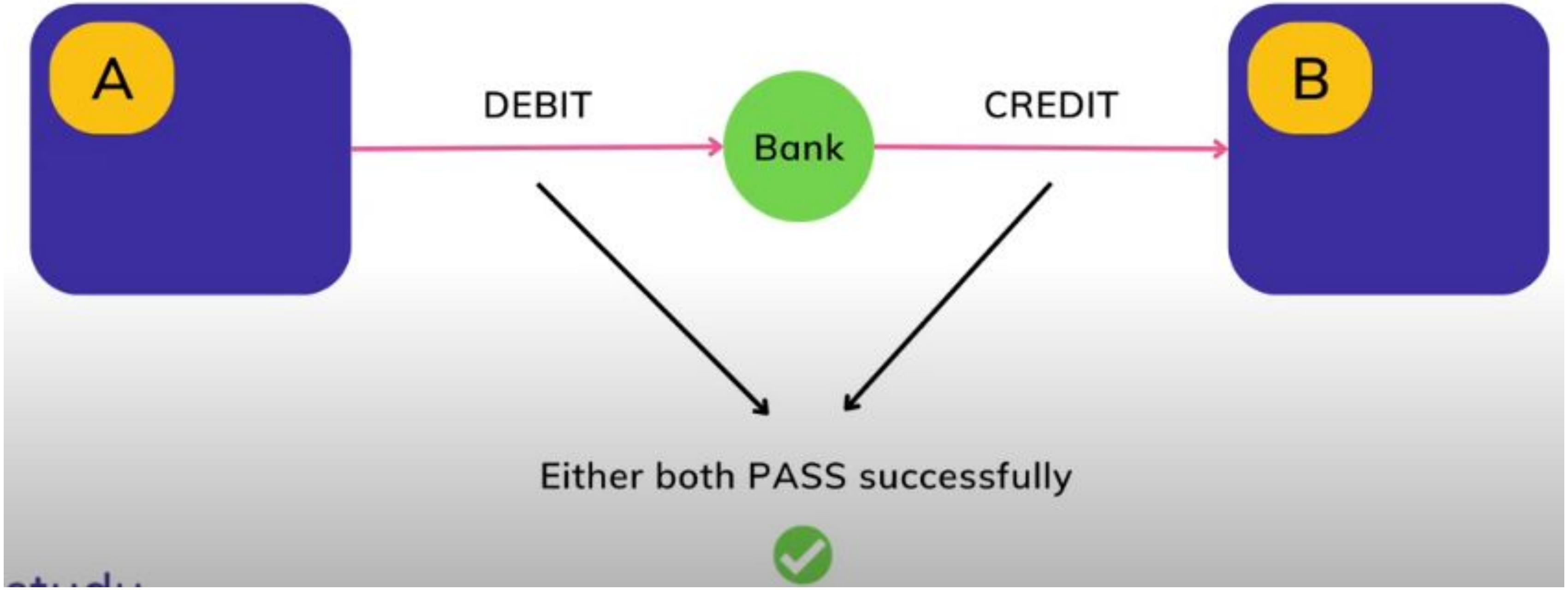
+2000

Amount deposit passes

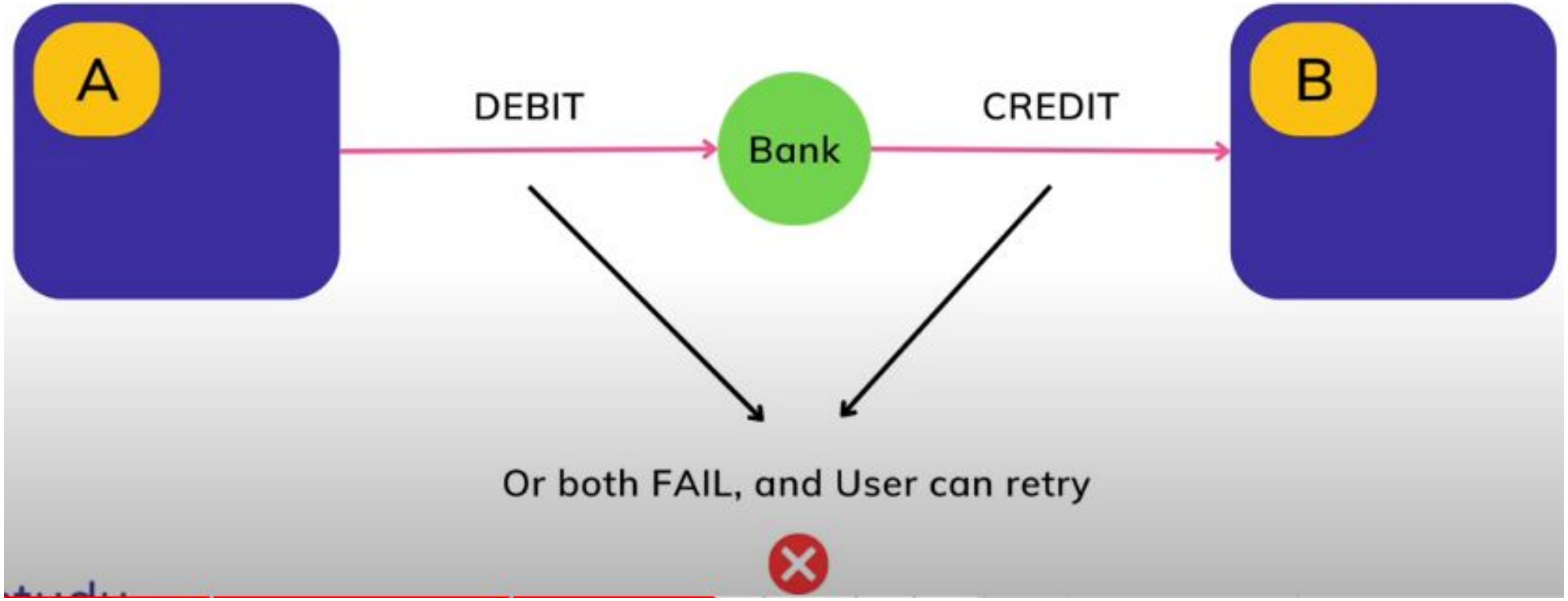
✓

B

Rs. 12,000



Either both PASS successfully

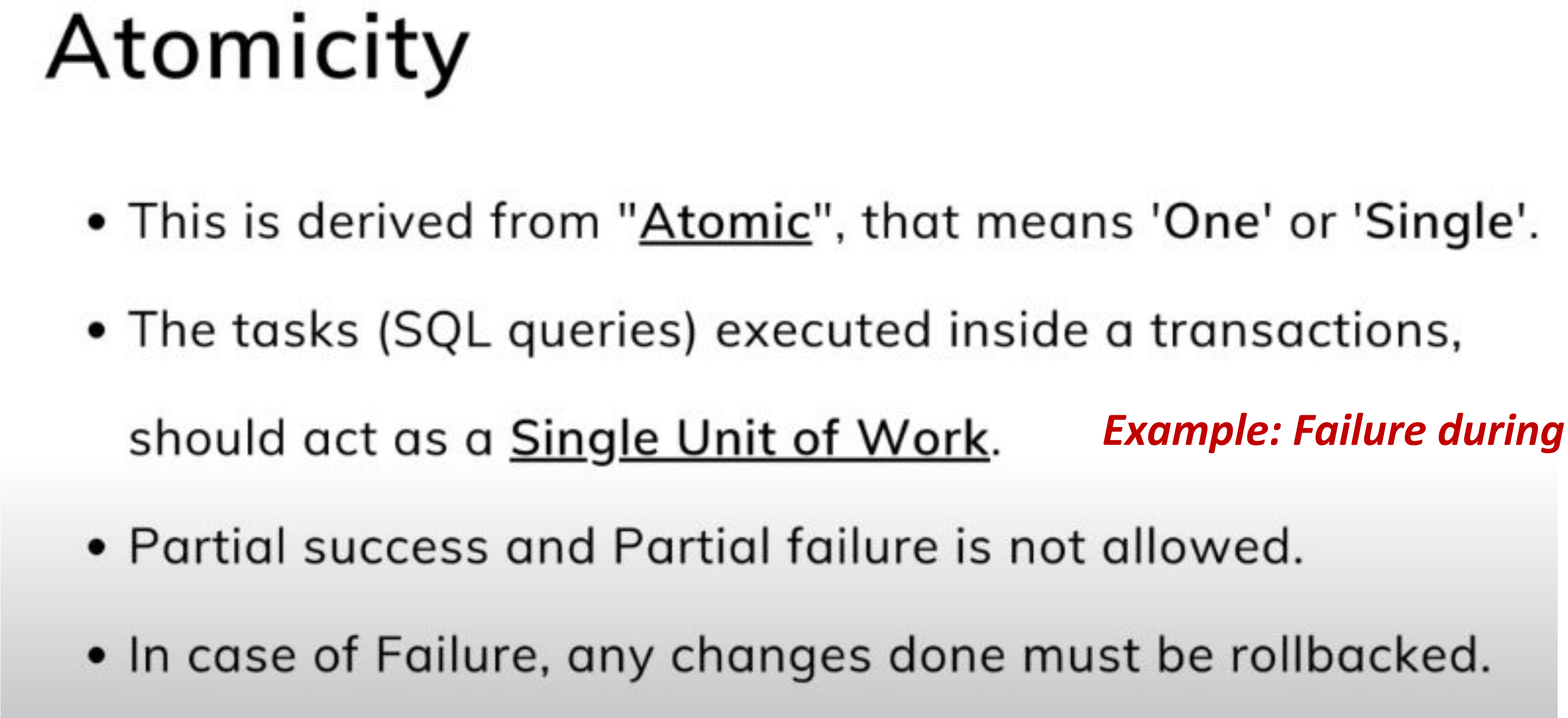


Or both FAIL, and User can retry

Database Transaction or Tx

When in a Database, we have to run multiple SQL queries, that affects the state of data in our database through INSERT, UPDATE or DELETE queries. We must combine them into a single unit of work, so that all of them pass together, or fail together.

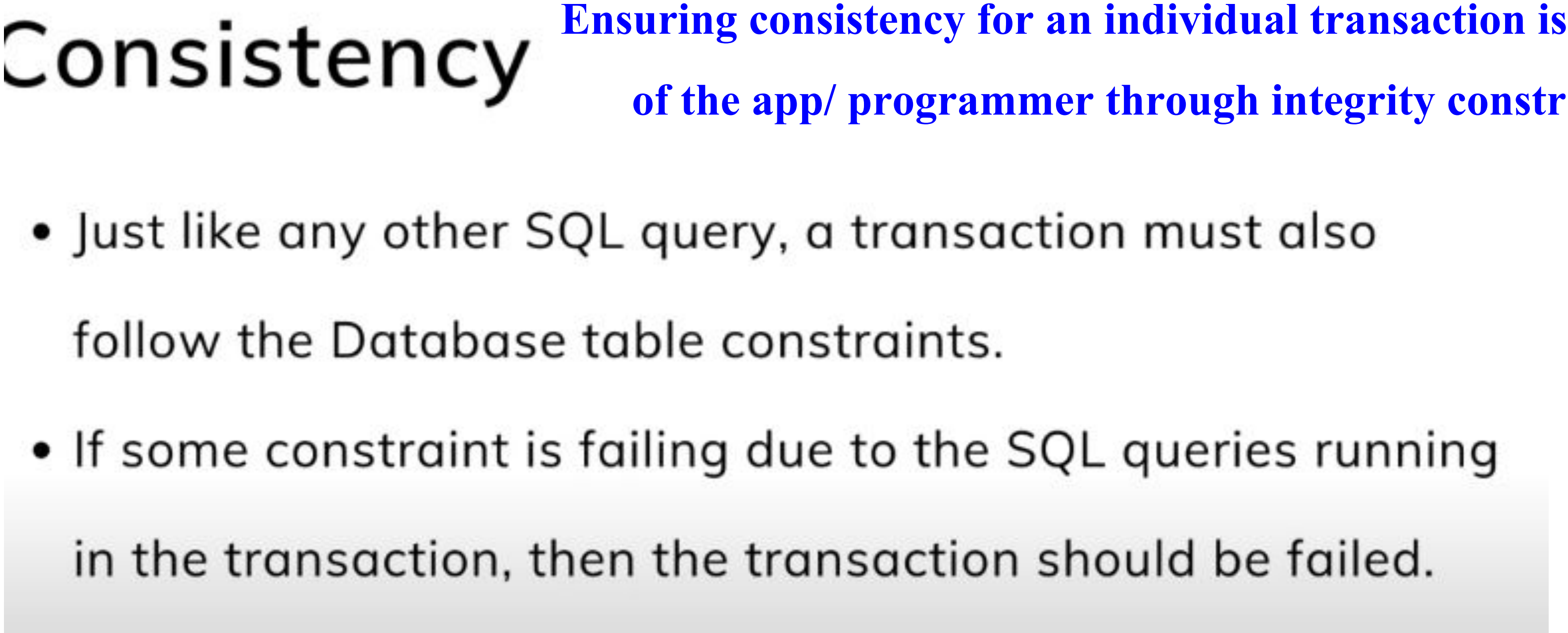
A	C	I	D
Atomicity	Consistency	Isolation	Durability



Atomicity

- This is derived from "Atomic", that means 'One' or 'Single'.
- The tasks (SQL queries) executed inside a transactions, should act as a Single Unit of Work.
- Partial success and Partial failure is not allowed.
- In case of Failure, any changes done must be rollbacked.

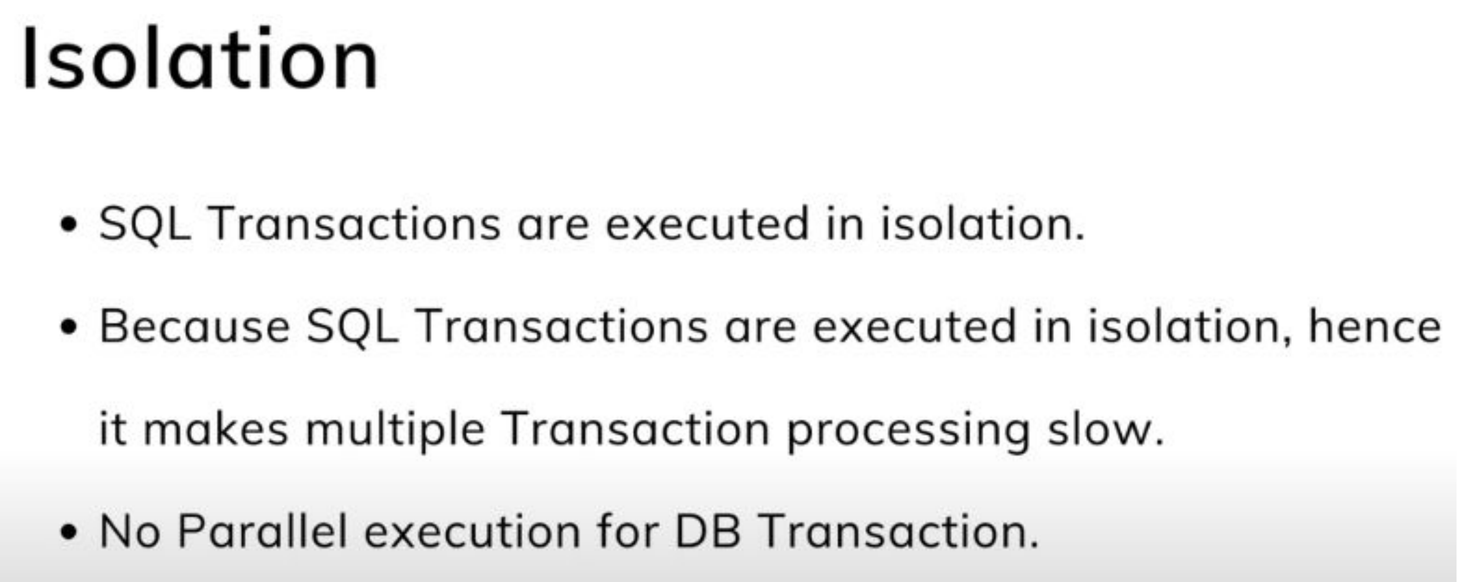
Example: Failure during A -> B



Consistency

Ensuring consistency for an individual transaction is the responsibili of the app/ programmer through integrity constraint

- Just like any other SQL query, a transaction must also follow the Database table constraints.
- If some constraint is failing due to the SQL queries running in the transaction, then the transaction should be failed.



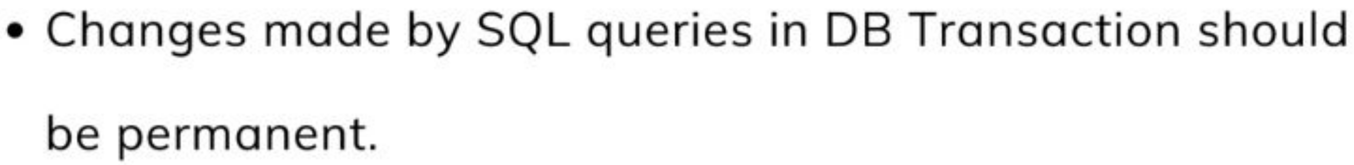
Isolation

- SQL Transactions are executed in isolation.
- Because SQL Transactions are executed in isolation, hence it makes multiple Transaction processing slow.
- No Parallel execution for DB Transaction.

Example :
T1 : transferring fund 500 from A to B account.
Deducting 500 from A, adding 500 to B.
T2: Reading A and B for amount to be known
Have to isolate T1 from T2 for successful completion of the transferring amount and reading the amount status from both the account.
T1 need to be completed first and then T2

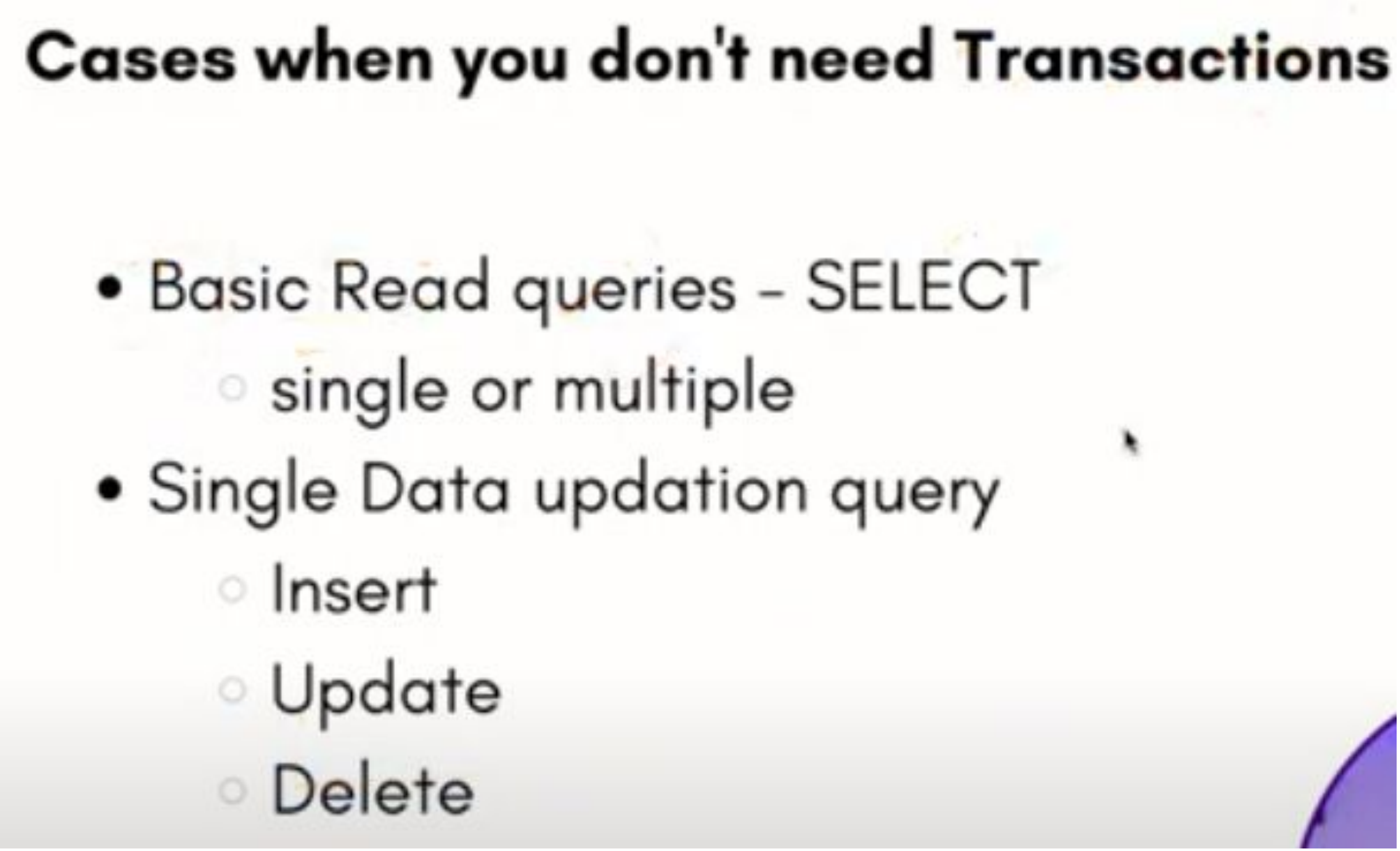
Durability

- **Updates carried out by the transaction have been written to disk before trans/. Completed.**
- **Info/ about the update need to written to disk.(useful when restarted after failure**



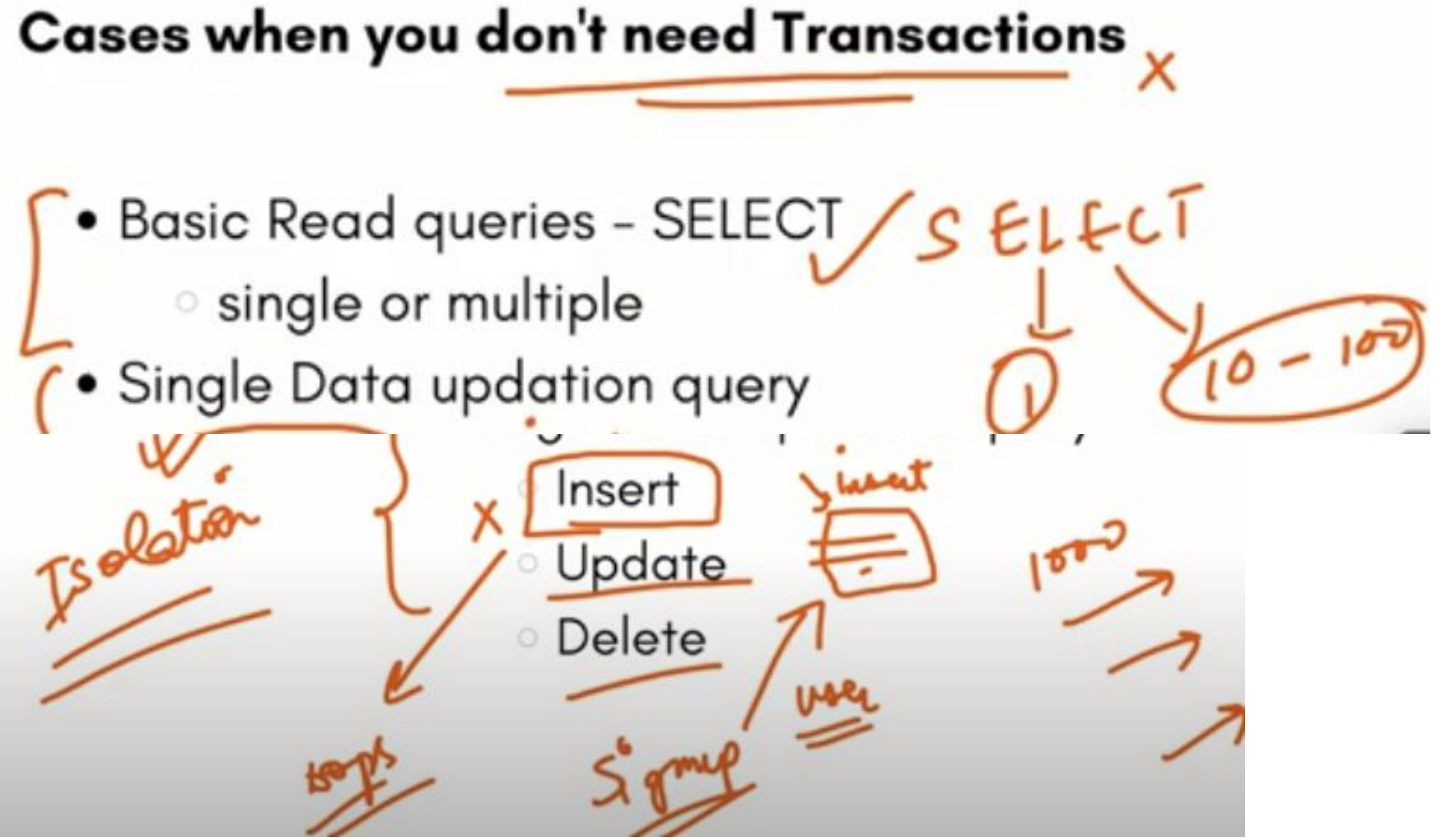
- Changes made by SQL queries in DB Transaction should be permanent.

Not in all cases your applications needs standards of transaction



Cases when you don't need Transactions

- Basic Read queries - SELECT
  - single or multiple
- Single Data updation query
  - Insert
  - Update
  - Delete



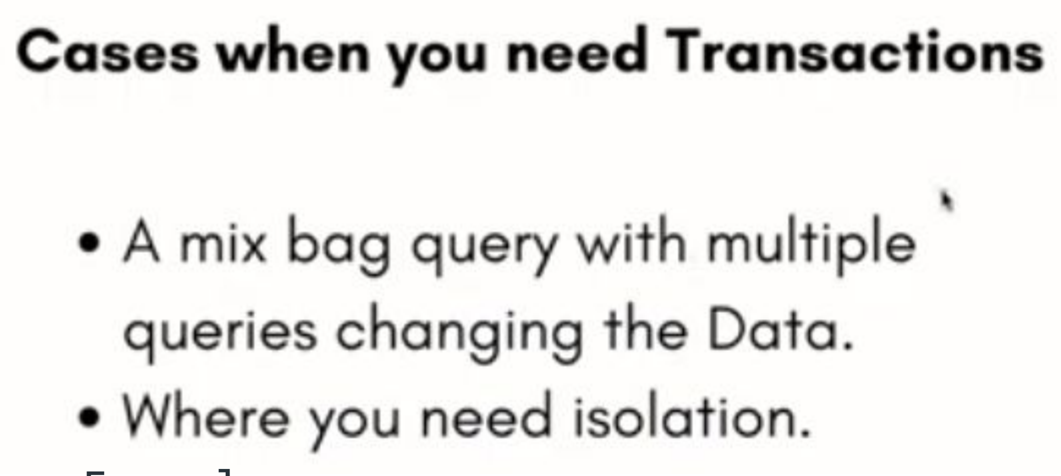
Cases when you don't need Transactions

- Basic Read queries - SELECT
  - single or multiple
- Single Data updation query
  - Insert
  - Update
  - Delete

- but this cannot be true in all cases.

Eg. Admin, user and product entity.

- If admin tells after 12'clock 30% discount on the particular product will be updated. So user will be accessing the products to know the discount and buy the product.
- so here it is of single update but any way needs to follow the stds of trans/ for consistent information to the user.



Cases when you need Transactions

- A mix bag query with multiple queries changing the Data.
- Where you need isolation.

```
Example:
1.Read(A)  -- 500     // Accessed from RAM.
2.A = A-50            // Deducting 50₹ from A.
3.Write(A)  --450     // Updated in RAM.
4.Read(B)  -- 800     // Accessed from RAM.
5.B=B+50          // 50₹ is added to B's Account.
6.Write(B)  --850
```

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

Write(X): Write operation is used to write the value back to the database from the buffer.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then A's value will remain 500 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

Rollback: It is used to undo the work done.

States of Transaction

A transaction must be in one of the following states:

- **Active State –**

  When the instructions of the **transaction are running** then the transaction is in active state.
- If all the 'read **and write' operations are performed without any error** then it goes to the "partially committed state"; if **any instruction fails**, it goes to the "failed state".

- **Partially Committed –**

  After completion of all the read and write operation the **changes are made in main memory or local buffer**.
- If the **changes are made permanent on the DataBase** then the state will change to "committed state" and in case of failure it will go to the "failed state".

- **Failed State –**

  When any instruction of the transaction fails, it goes to the "failed state" or if failure occurs in making a permanent change of data on Data Base.

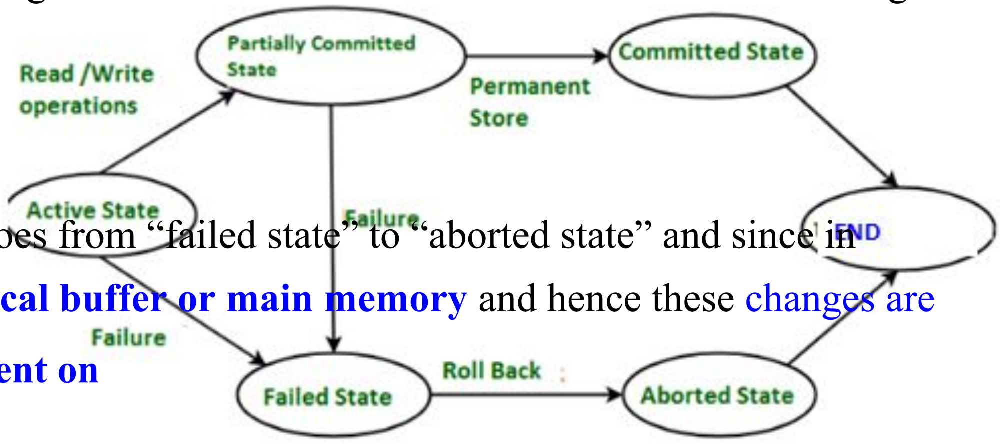
- **Aborted State –**

  After having any type of failure the transaction goes from "failed state" to "aborted state" and since in previous states, the **changes are only made to local buffer or main memory** and hence these changes are deleted or rolled-back.

- **Committed State –**

  It is the state when the **changes are made permanent on the Data Base** and the transaction is complete and therefore terminated in the "terminated state".

-



Transaction States in DBMS

Schedule

- A **series of operation** from one transaction to another transaction is known as schedule.
- a sequence of operations **in the chronological order** in which such things are intended to take place
- It is **used to preserve the order** of the operation in each of the individual transaction.

•Transaction Isolation: **Transaction processing system** allow **multiple transaction to run concurrently**.

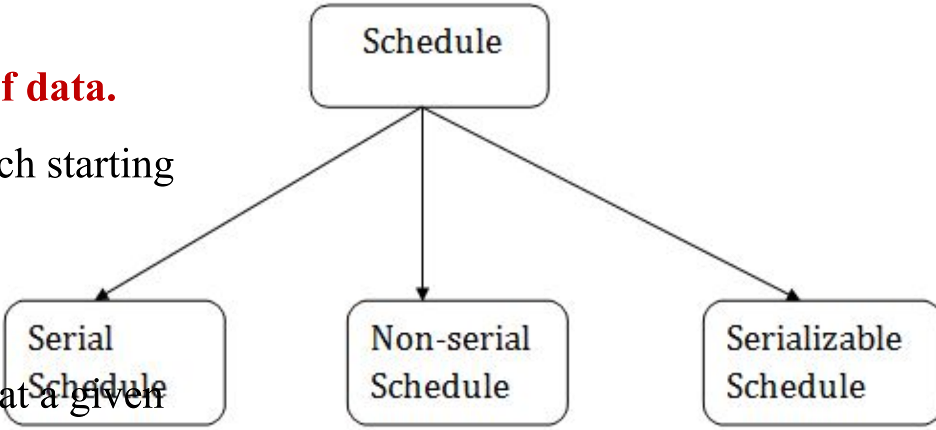
•But cause **several complications** with **consistency of data.**

Easier method: **make it serial**: one at a time and each starting only after the previous one is completed.

2 good reasons for concurrency:

•**Improved throughput** (no.of transactions executed at a given amount of time)and resource utilization (processor and disk – ie spend less time idle)

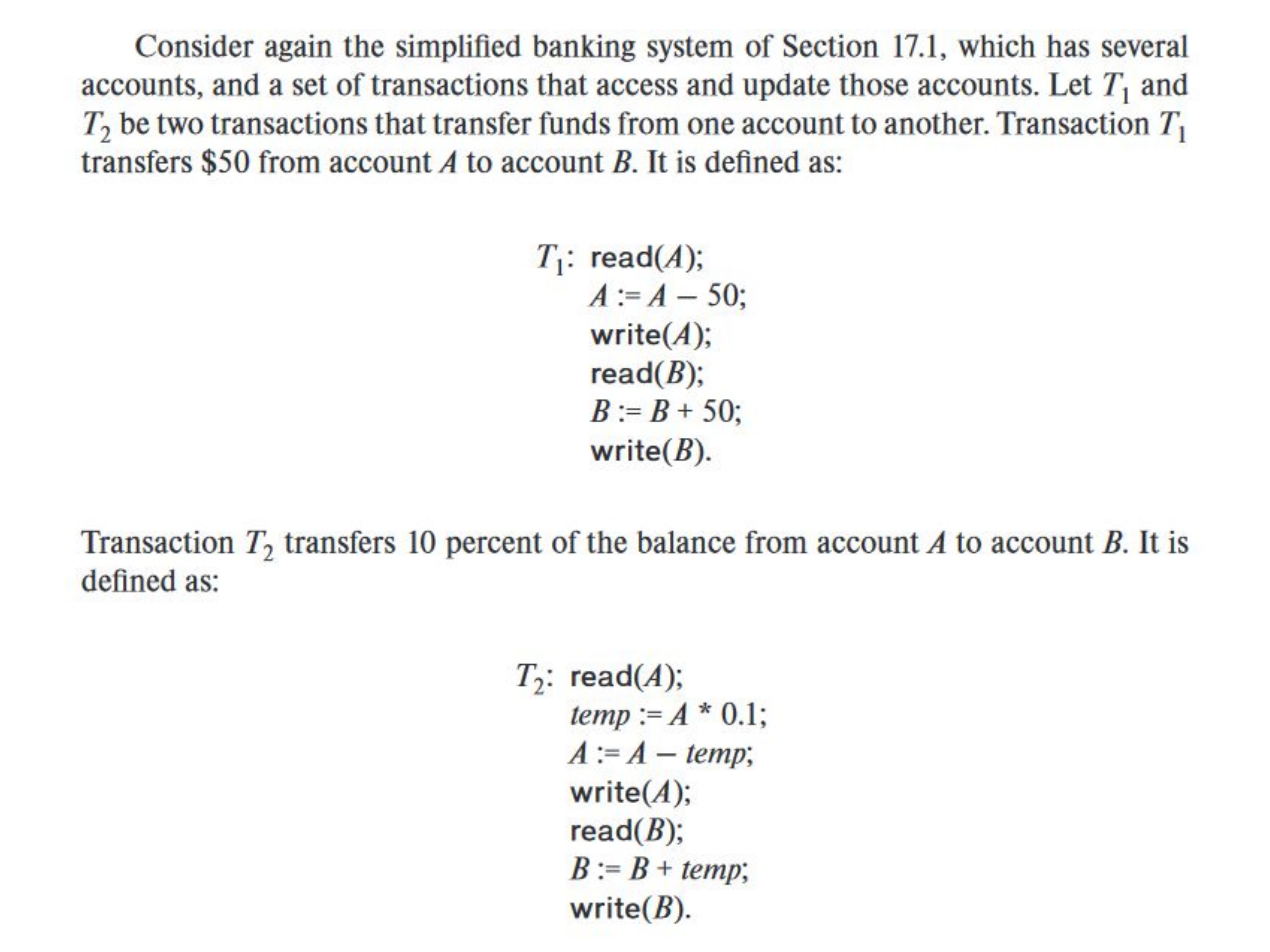
•**Reduced waiting time**. (short and long transactions. Serial: short needs to wait for long to complete, so delays. Concurrency reduce the delays.)



concurrent

Concurrent to be serial

Note:
Chronological order:
an arrangement of events in the **order of their happening** or based on the **time** they have occurred.



Consider again the simplified banking system of Section 17.1, which has several accounts, and a set of transactions that access and update those accounts. Let T_1 and T_2 be two transactions that transfer funds from one account to another. Transaction T_1 transfers $50 from account *A* to account *B*. It is defined as:

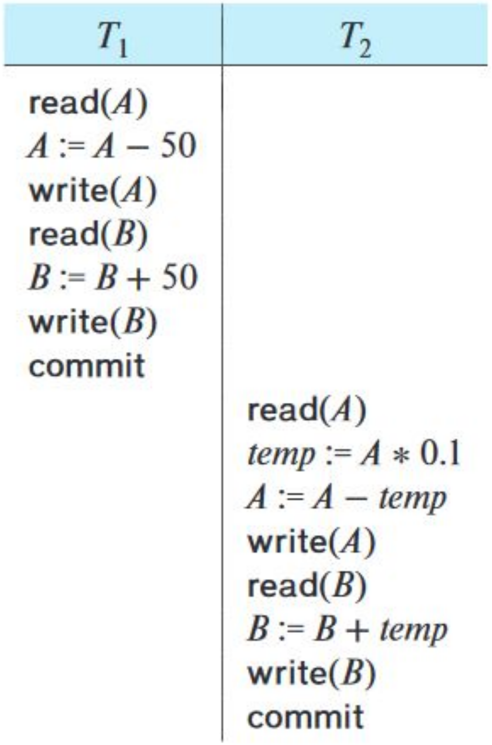
```
$T_1$: read($A$);
       $A := A - 50$;
       write($A$);
       read($B$);
       $B := B + 50$;
       write($B$).
```

Transaction T_2 transfers 10 percent of the balance from account *A* to account *B*. It is defined as:

```
$T_2$: read($A$);
       $temp := A * 0.1$;
       $A := A - temp$;
       write($A$);
       read($B$);
       $B := B + temp$;
       write($B$).
```

Assume ,
A=1000
B=2000

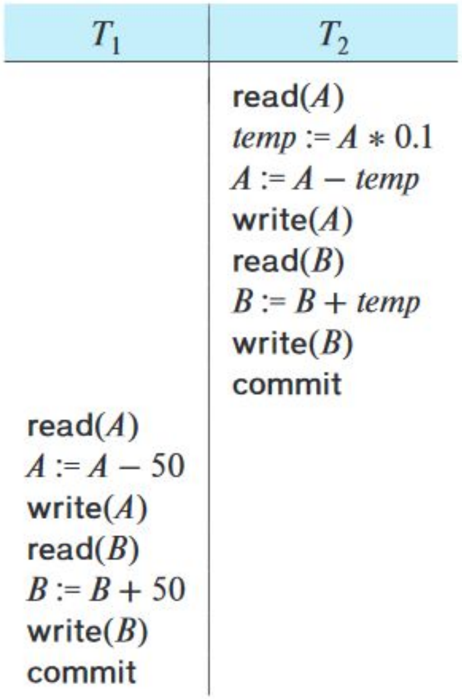
Total amount in account =3000



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

Figure 17.2 Schedule 1—a serial schedule in which T_1 is followed by T_2.

When DB sys/. Executes several trans. Concurrently, no need to be serial



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

Figure 17.3 Schedule 2—a serial schedule in which T_2 is followed by T_1.

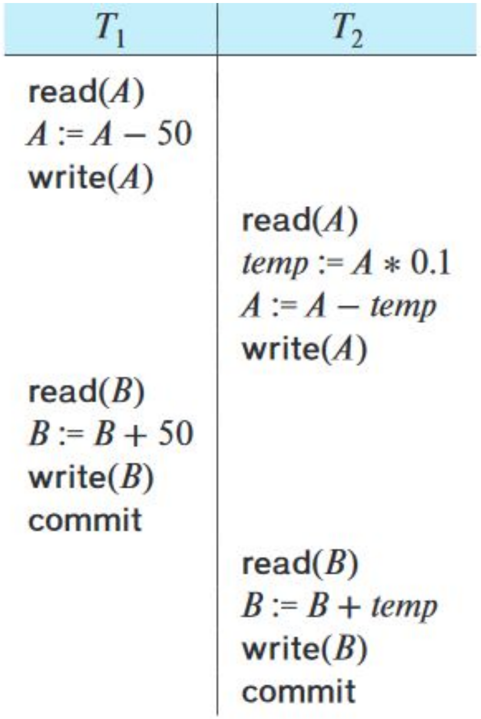
Final value of A and B after schedule 1.
A=855
B=2145
Total amount of money in the account A+B is preserved.

This execution sequence of transaction is called schedules. Preserve the order in which the instructions appears

Final value of A and B after schedule 2.
A=850
B=2150
Total amount of money in the account A+B is preserved.

- When the **database system** executes **several transactions concurrently**, the corresponding schedule no longer needs to be serial.
- If two transactions are running concurrently, the **OS may execute one transaction for a little while**, then perform a **context switch**, **execute the second transaction for some time**, and then **switch back to the first** transaction for some time, and **so on.**
- With **multiple transactions**, the **CPU time is shared** among **all the transactions.**

- it is **not possible to predict exactly - how many instructions** of a transaction will be **executed before the CPU switches** to another transaction.



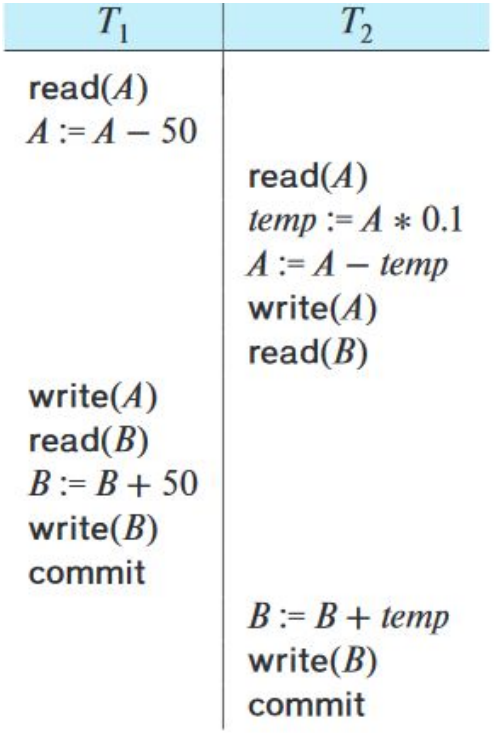
T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

igure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

- One possible schedule - Figure 17.4 arrive at the same state – as executed serially.

- But ,not all concurrent executions result in a correct state.

- 17.5 - arrive at a state where the final values of accounts A and B are $950 and $2100, respectively.
- This final state is an inconsistent state, since we have gained $50 in the process of the concurrent execution.
- Indeed, the sum A + B is not preserved



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)	
commit	
	$B := B + temp$
	write(B)
	commit

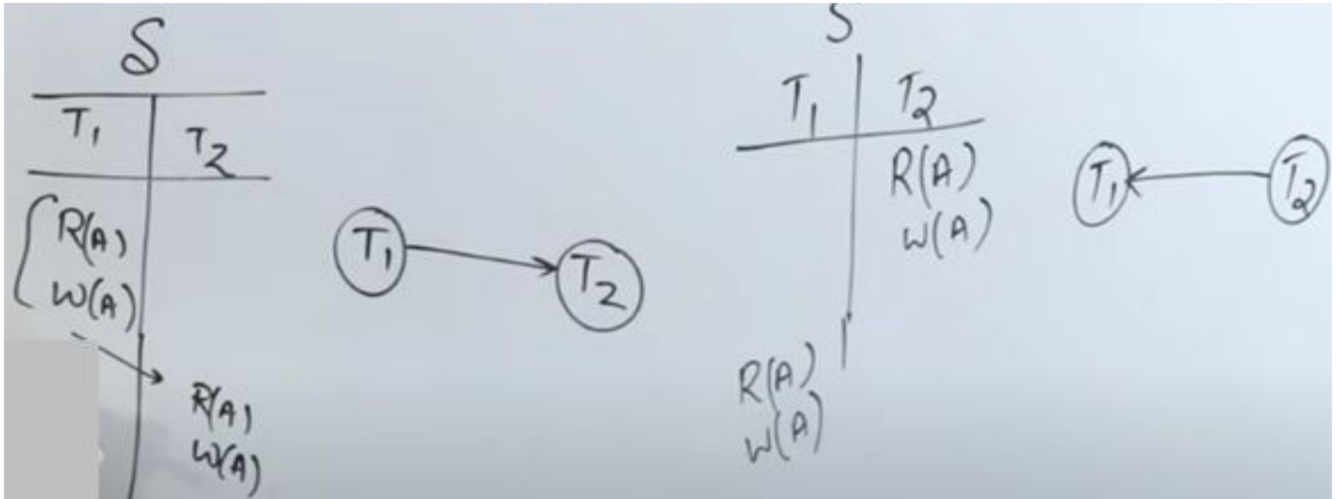
Figure 17.5 Schedule 4—a concurrent schedule resulting in an inconsistent state.

- If control of concurrent execution is **left entirely** to the **operating system**, ***many possible schedules***, including ***ones that leave the database*** in an ***inconsistent state***, such as the one just described, are possible.
- It is the **job of the database system to ensure** that **any schedule** that is executed will **leave the database in a consistent state**.
- **The concurrency-control component of the database system** carries out this task.

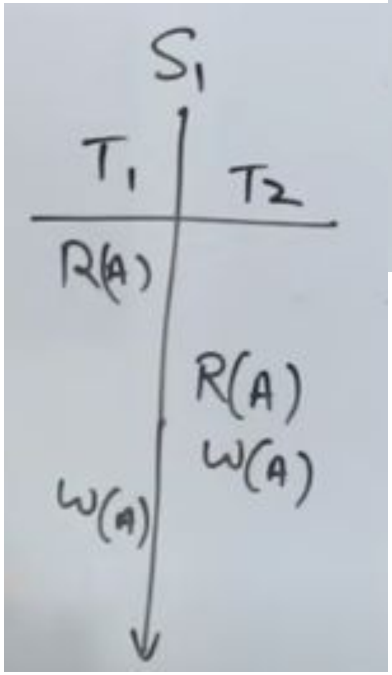
Serializable:

Making concurrent transactions to be done serially.

- Before we can consider how the concurrency-control component of the database system can ensure serializability, we consider **how to determine when a schedule is serializable**

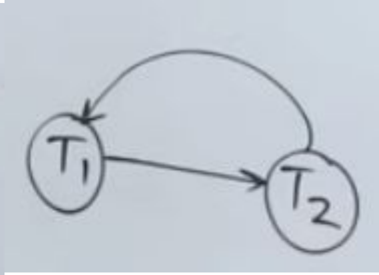


- **Both schedule are serial and no interleaving.**
- **Serial schedule cannot be serializable.**



S1

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



Loop gets created

To **check whether given parallel schedule be serializable** or not.

Problems:

Given parallel schedule and convert to serial.

To convert this into serial :two ways

- T1 -> T2
- T2 -> T1

- Does **any equivalent serial schedule exist** for this **parallel schedule**. That is called as **serializable**

T1 -> T2

T1	T2

R(A)
W(A)	

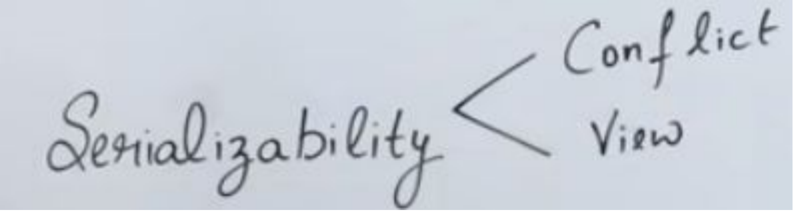
	R(A)
	W(A)

T2 -> T1

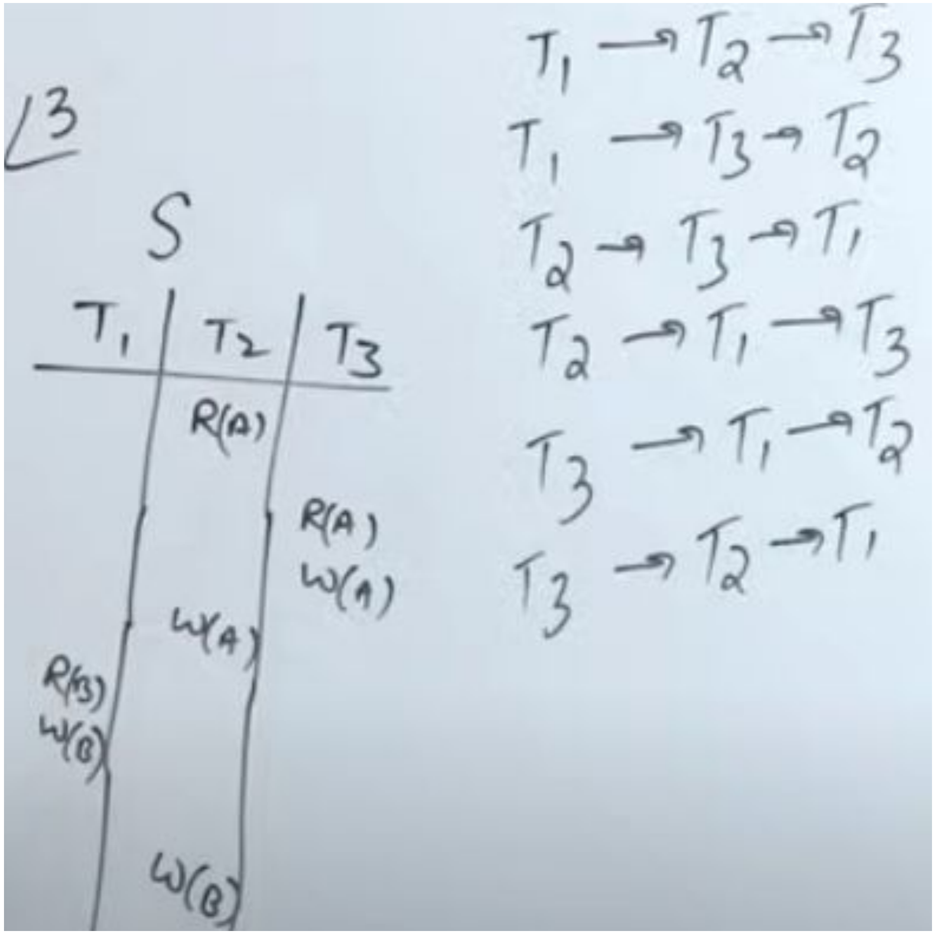
T1	T2

	R(A)
	W(A)

R(A)
W(A)	



- If three transactions are given and asked to find whether this is serializable or not.
- For that take factorial of no.of transactions (n!) ie here no.of trsansactions n=3 T1,T2,T3, 3!=6 . 6 as below



⌊3

S

T_1	T_2	T_3
	R(A)	
		R(A)
		W(A)
	W(A)	
R(B)		
W(B)		
	W(B)	

$T_1 \to T_2 \to T_3$

$T_1 \to T_3 \to T_2$

$T_2 \to T_3 \to T_1$

$T_2 \to T_1 \to T_3$

$T_3 \to T_1 \to T_2$

$T_3 \to T_2 \to T_1$

While converting this schedule into serial and can able to **bring out any one of this order** from the 6 possibilities, then you can it is serializable.

ie equivalent to any one of the 6

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

Figure 17.6 Schedule 3—showing only the read and write instructions.

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

Figure 17.7 Schedule 5—schedule 3 after swapping of a pair of instructions.

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

Figure 17.8 Schedule 6—a serial schedule that is equivalent to schedule 3.

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

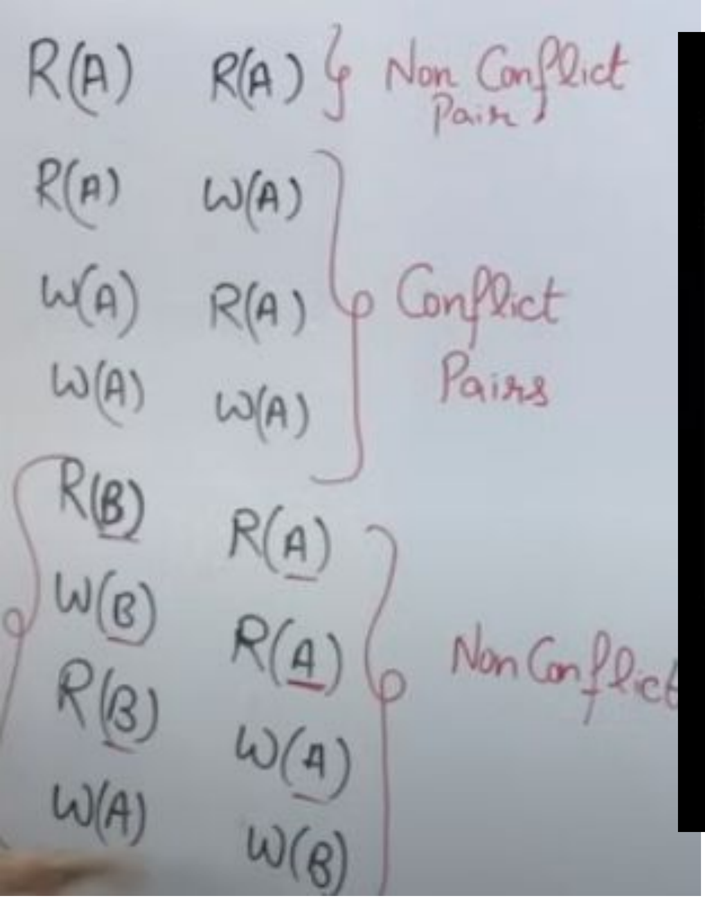
Figure 17.9 Schedule 7.

Conflict equivalence

- Two schedules are said to be conflict equivalent if the **order of two conflicting operations is the same** in both the schedules.

Conditions to be satisfied for Conflicting operation:

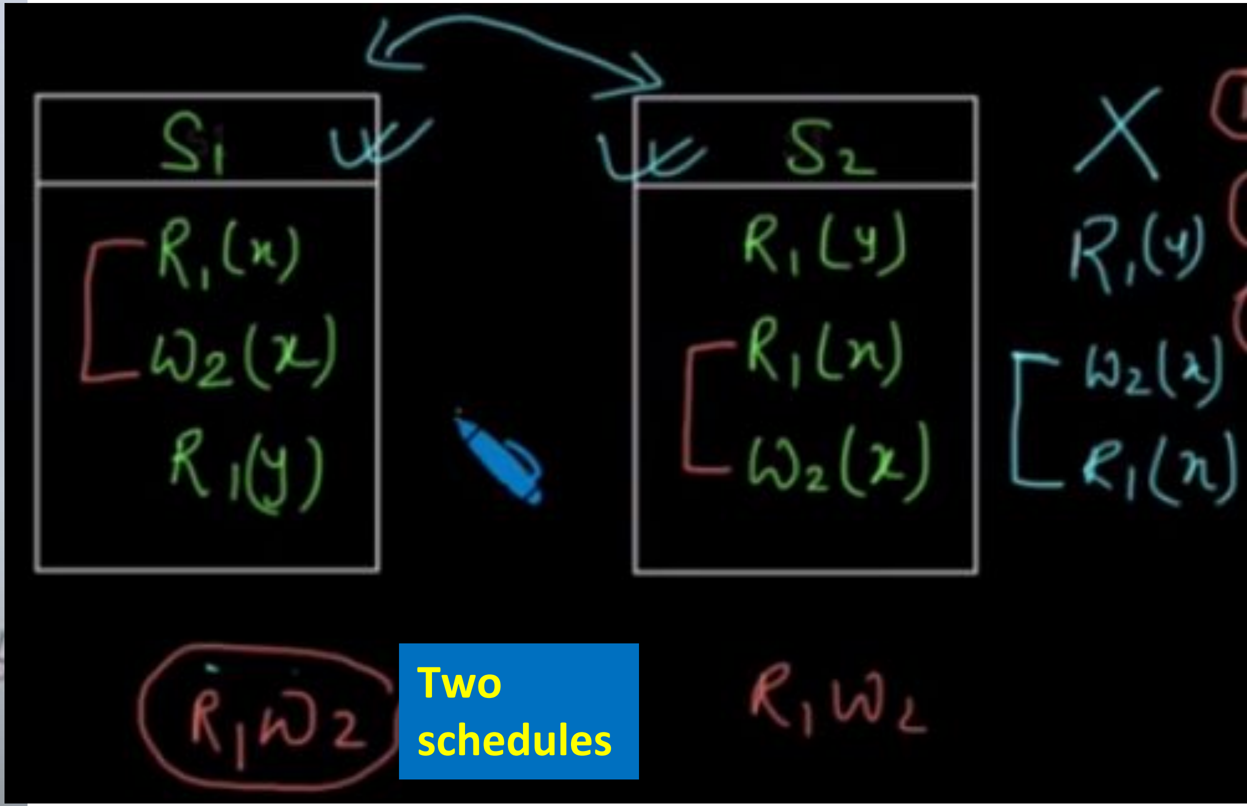
- Operations belongs to **two different transactions**.
- They should be working or **acting over same database item** or variable.
- **Atleast one** of the operation should be **write**.



R(A) R(A) } Non Conflict Pair

R(A) W(A)
W(A) R(A) } Conflict Pairs
W(A) W(A)

R(B) R(A)
W(B) R(A) } Non Conflict
R(B) W(A)
W(A) W(B)

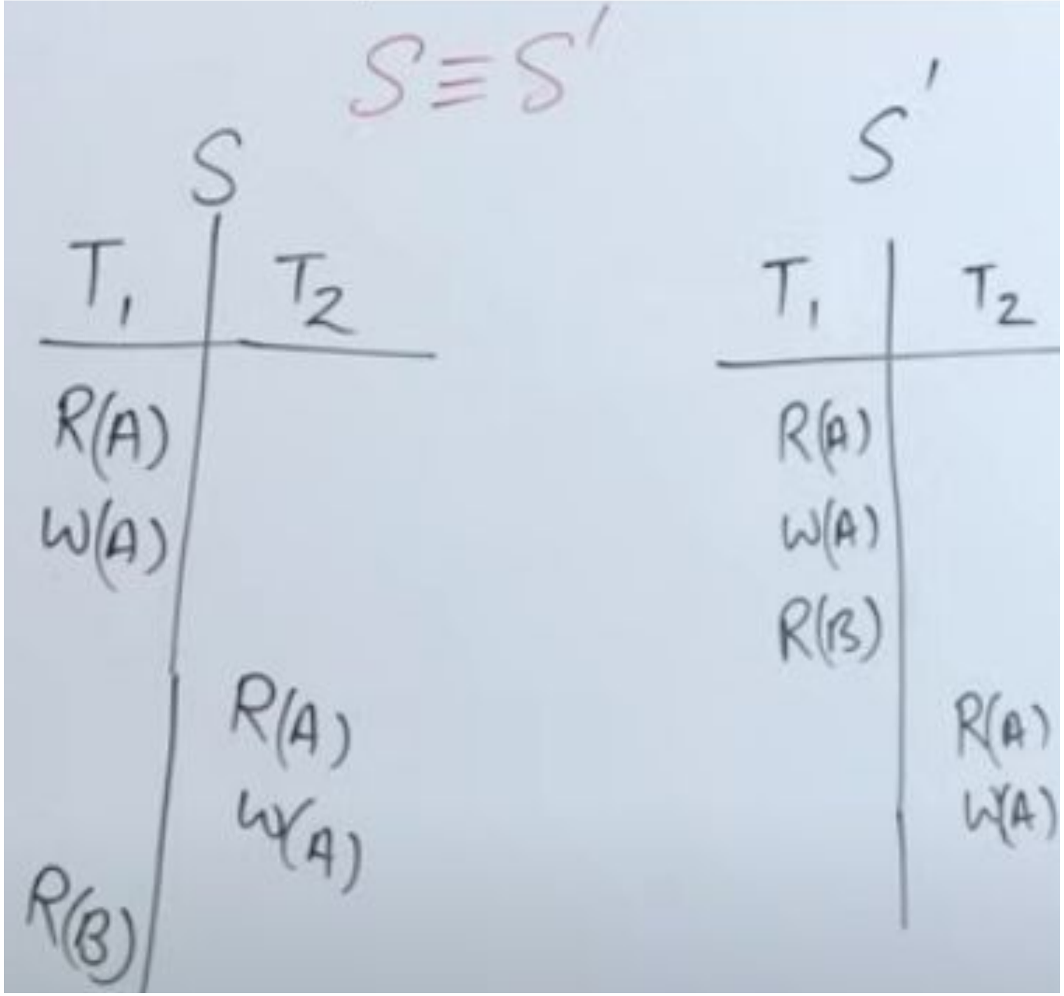


S_1	S_2	X
$R_1(x)$	$R_1(y)$	$R_1(y)$
$W_2(x)$	$R_1(x)$	$W_2(x)$
$R_1(y)$	$W_2(x)$	$R_1(x)$

R_1W_2 R_1W_2

Two schedules

Check whether they are equivalent or not



$S \equiv S'$

S

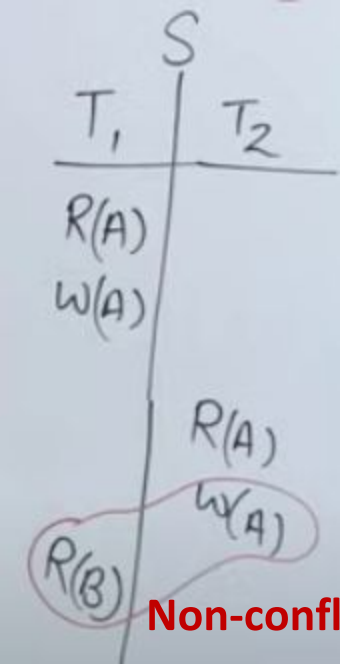
T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

S'

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

Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations

1. R(A) and W(A) in both S ans S' is same position. But R(B) changed.
2. Check adjacent non-conflict pairs.
3. If the pairs are non-conflict, then swap the position.
4. Again check for adjacent non-conflict pair.

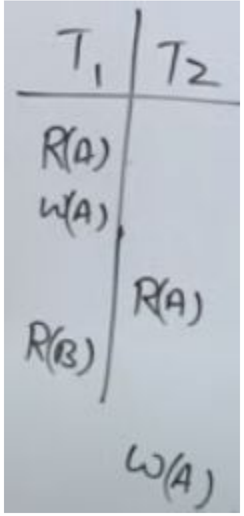


S

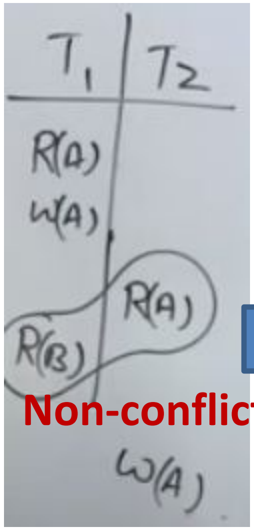
T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

Non-conflict pairs

swap



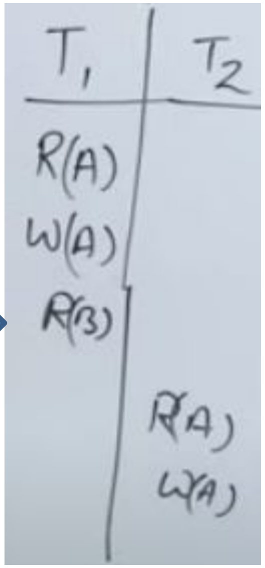
T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)



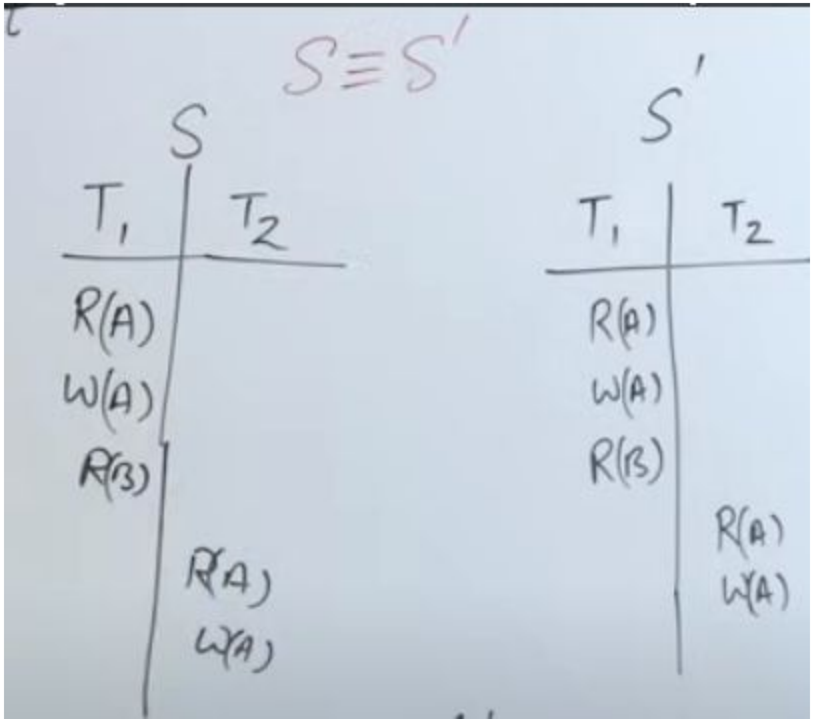
T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)

Non-conflict pairs

Swap again



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



S≡S'

S

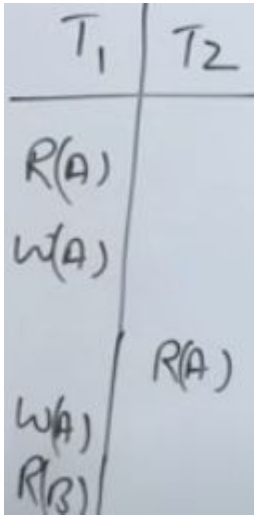
T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

S'

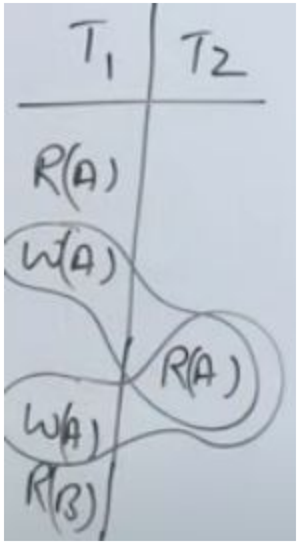
T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

Now S and S' are same ie they both are conflict equivalent schedule

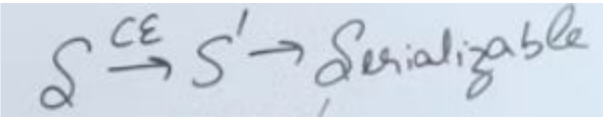
- Check the equivalent schedule of this given schedule. Here no swapping(no change in positions) is needed between the adjacents bex already it is conflicting pairs



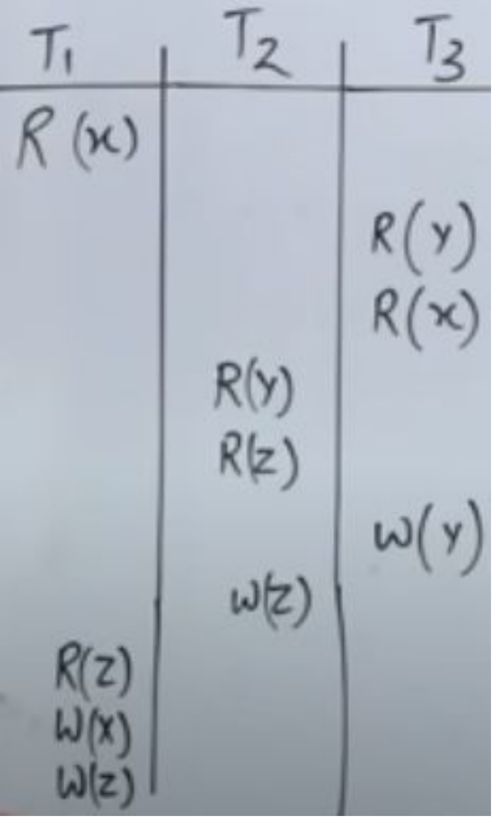
T1	T2
R(A)	
W(A)	
	R(A)
W(A)	
R(B)	



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



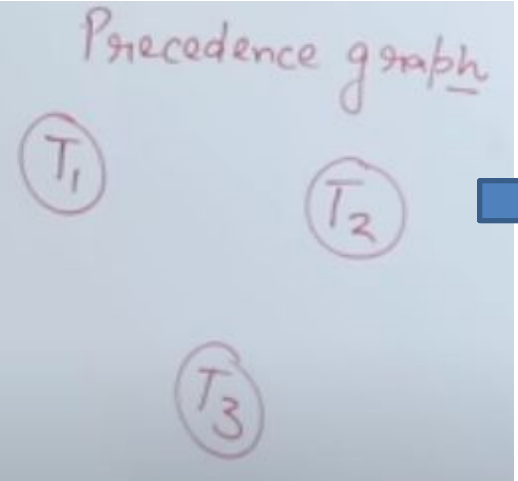
$S \xrightarrow{CE} S' \rightarrow$ Serializable

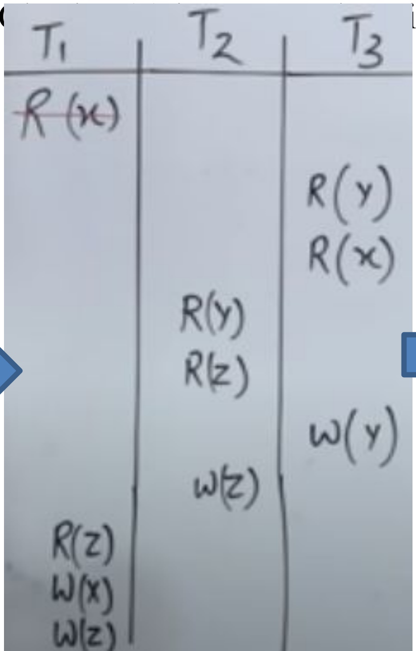


T1	T2	T3
R(x)		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		

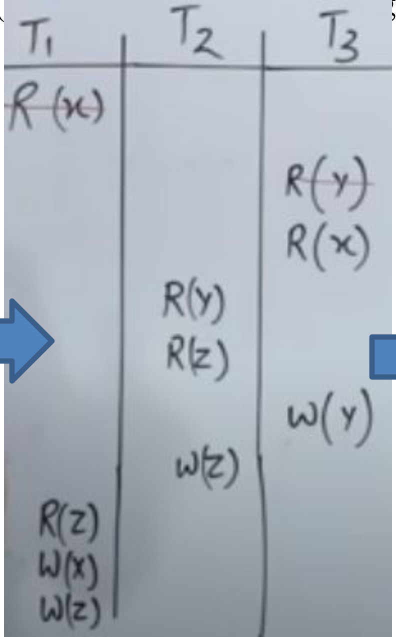
Conflict serializability

- Check Whether given schedule is conflict serializable or not?
- How to check it? – first draw precedence graph.(edges and vertices/nodes, vertices -transactions, edges- flow of conflict pairs)
- How to draw the edges? Check conflict pairs in other transactions and draw the edges.
- What is conflict pairs? R-W, W-R,W-W
- To check, start from T1 having R(x) as first operation and check with other two transactions for W(x) ie conflict pair, but there is no W(x) in T2 and T3. so cancel R(x) from T1.
- Next check R(y) from T3, no W(Y) in T1 and T2, so cancel it.
- C [illegible] is W([illegible] in T1, then draw the edges from t3 to T1

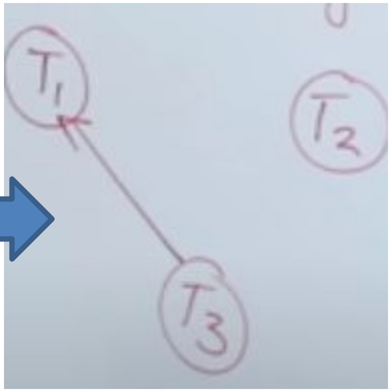


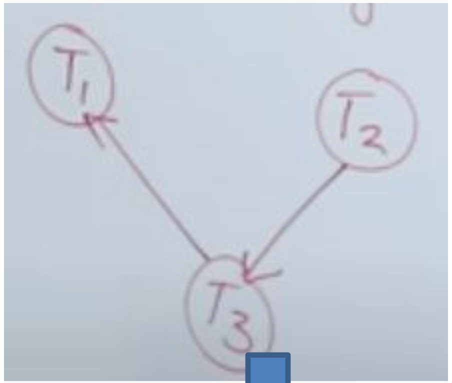


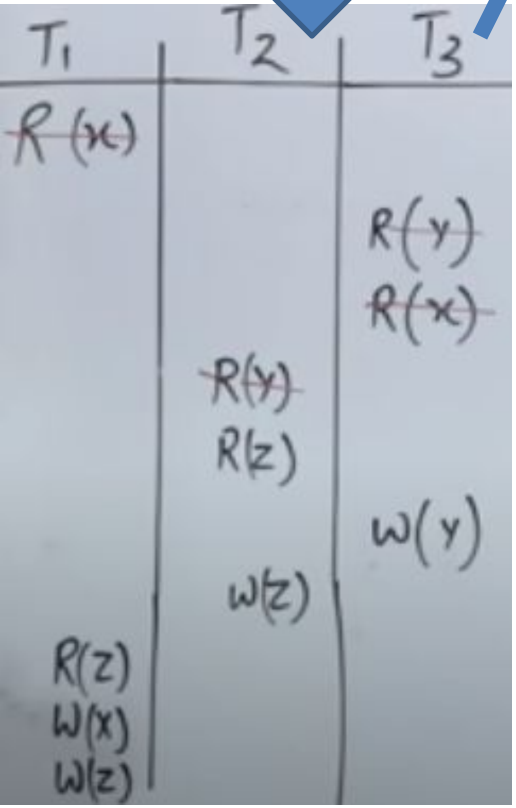
T1	T2	T3
~~R(x)~~		
		R(y)
		R(x)
	R(y)	
	R(z)	
		W(y)
	W(z)	
R(z)		
W(x)		
W(z)		



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

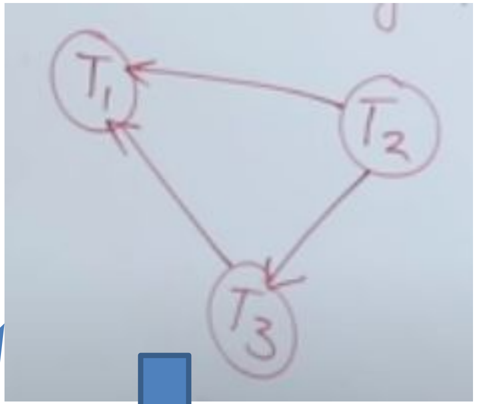


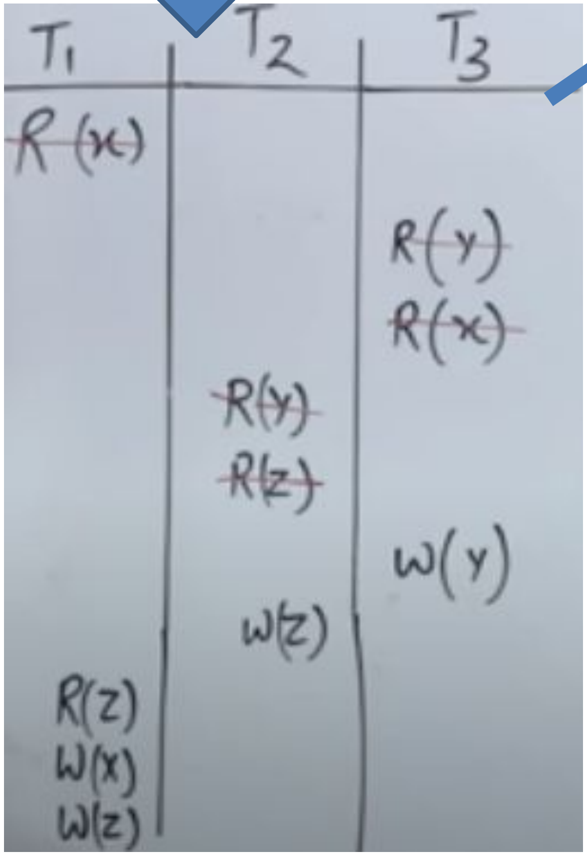




T1	T2	T3
~~R(x)~~		
		~~R(y)~~
		~~R(x)~~
	~~R(y)~~	
	R(z)	
		w(y)
	w(z)	

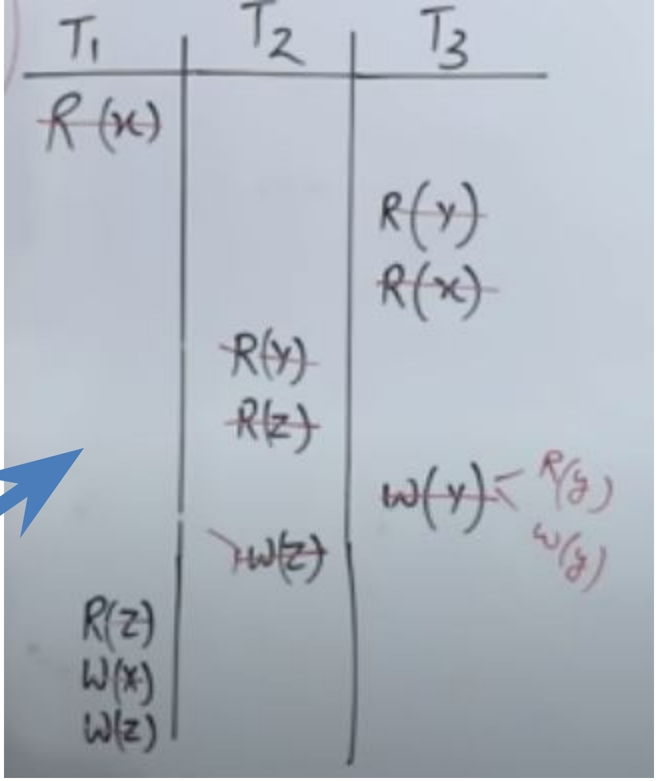
R(z)
W(x)
W(z)		





T1	T2	T3
~~R(x)~~		
		~~R(y)~~
		~~R(x)~~
	~~R(y)~~	
	~~R(z)~~	
		w(y)
	w(z)	

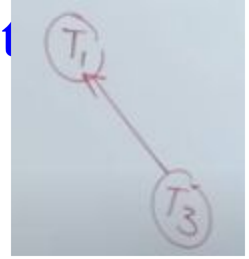
R(z)
W(x)
W(z)		



T1	T2	T3
~~R(x)~~		
		~~R(y)~~
		~~R(x)~~
	~~R(y)~~	
	~~R(z)~~	
		~~w(y)~~ ← R(y) w(y)
	~~w(z)~~	

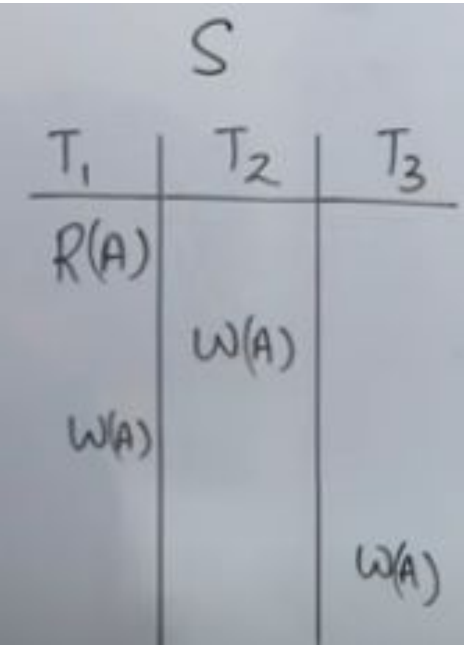
R(z)
W(x)
W(z)		

- To serialize this 3 transaction, find 3!, so 6 possibilities are there as:
- T1->T2->T3
- T1->T3->T2
- T2->T1->T3
- **T2->T3->T1**
- T3->T1->T2
- T3->T2->T1
- Among this graph is equivalent to which serial transaction?
- To check this, go to the graph for **Indegree =0 ,means find the vertices whose indegree is zero (no edges is towards that**
- **In this example T2 indegree is 0, so first remove it.**
- **After T3 has indegree 0, so remove it. (T2->T3)**
- **After removing T3 only one node T1 will be there, and its indegree is now 0, so remove it. (T2->T3->T1)**



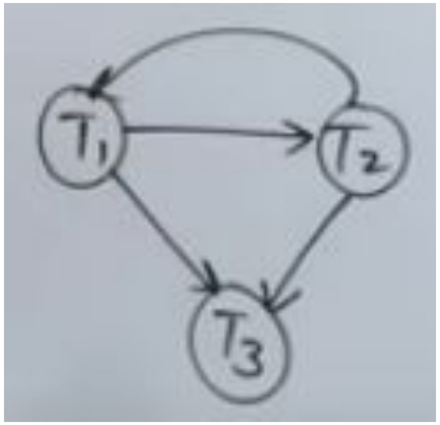
View serializability

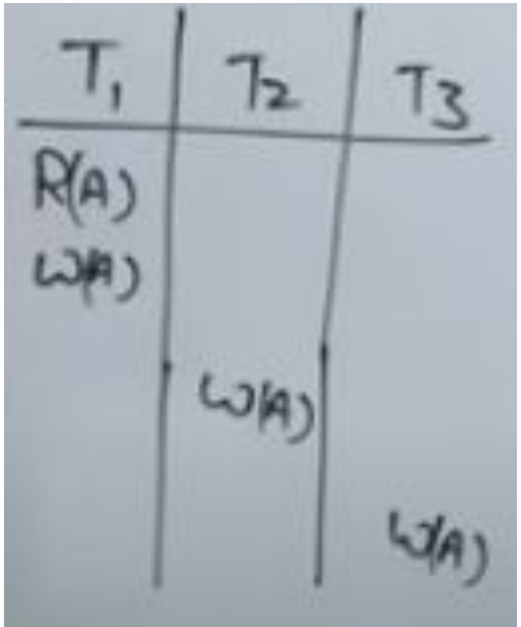
- Check whether this is conflict serializable or not?
- It is **not a conflict seriablizable(ie. It is non-conflict serializable)**, because loop exist between T1 and T2.**But can't answer for whether it is serial.**
- **So make use of second method called view serializable method**
- **Little bit rearrange the schedule as given in 3rd diagram.check whether first and third are matching each other**



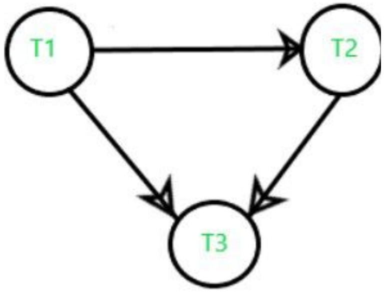
S

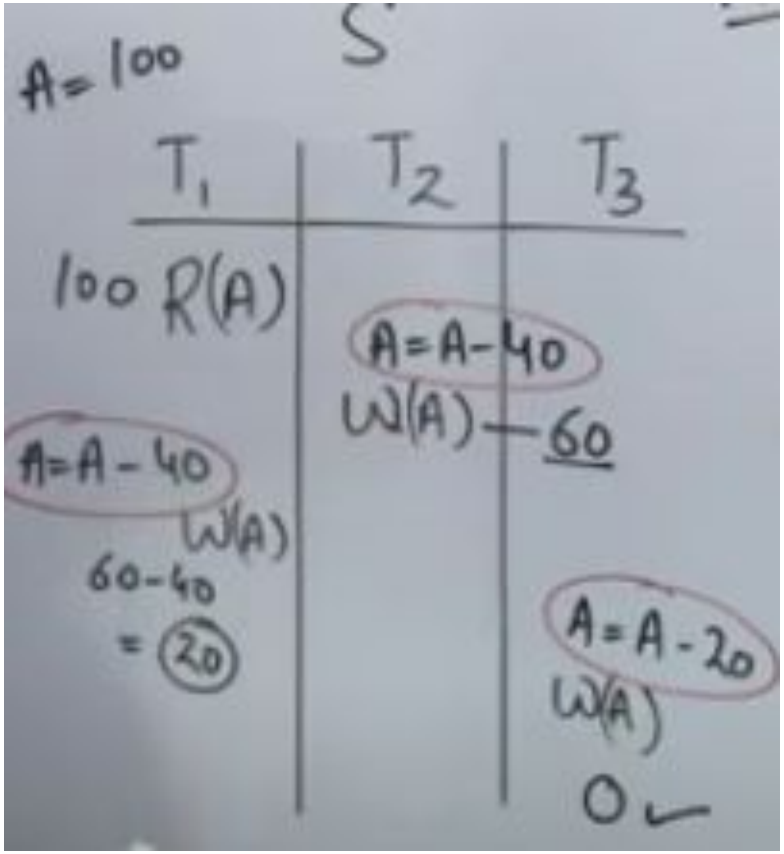
T_1	T_2	T_3
R(A)		
	W(A)	
W(A)		
		W(A)





T_1	T_2	T_3
R(A)		
W(A)		
	W(A)	
		W(A)

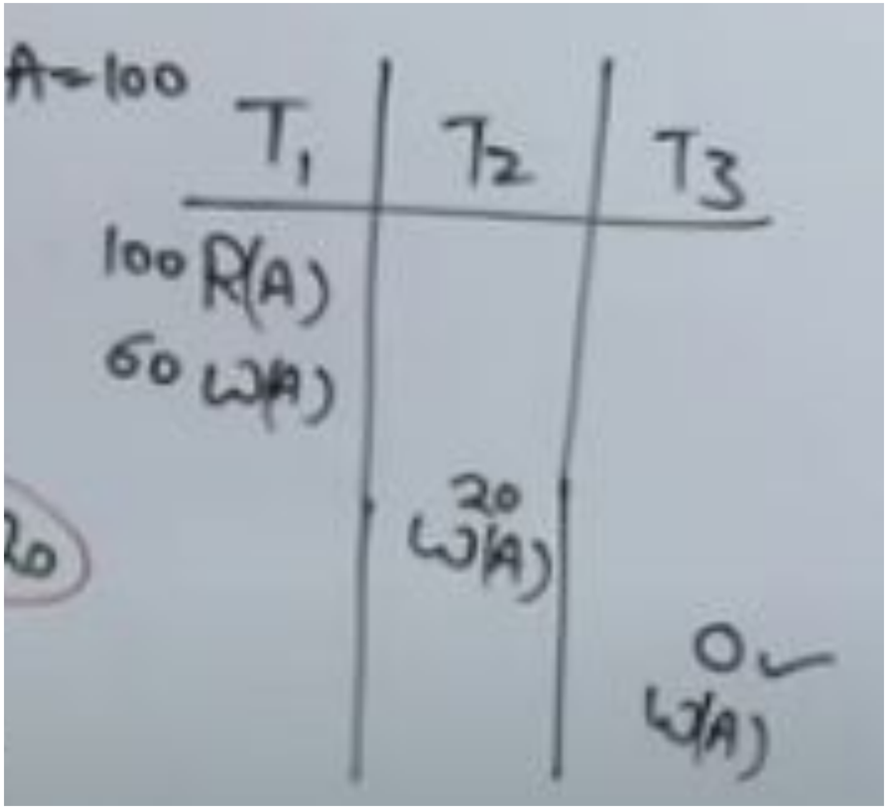




T1	T2	T3
100 R(A)		
	A=A-40 W(A) — 60	

A=A-40
W(A)		

		A=A-20
		W(A)
		0 ✓



T1	T2	T3
100 R(A) 60 W(A)		
	20 W(A)	
		0 ✓ W(A)

- They both are view equivalent and not conflict equivalent.
- Here conflict serializable says it is not serializable but view says it is serializable.
- If **serializable is not achieved by conflict** then **go for view serializable**

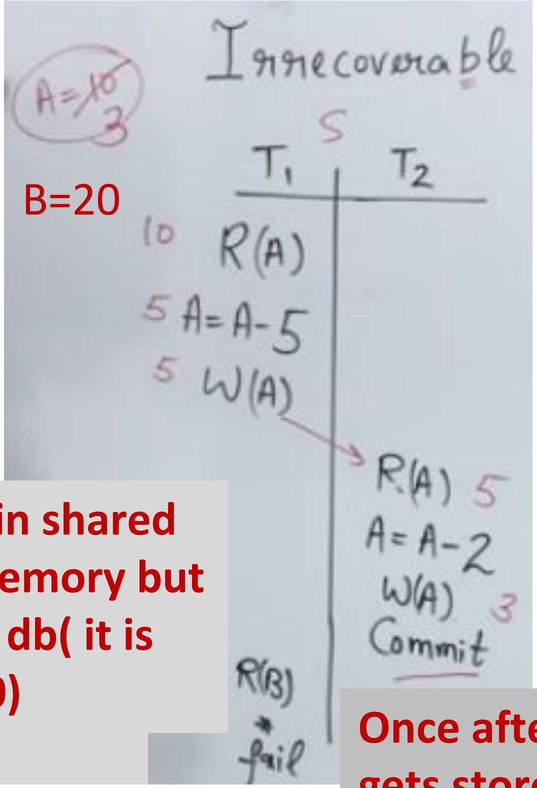
Conflict Serializability in DBMS checks if a non-serial schedule is conflict serializable or not. View Serializability checks if a schedule is view serializable or not. If a schedule is a view equivalent to a Serial Schedule, it is said to be view serializable.

Need of View-Serializability

- There may be **some schedules that are not Conflict-Serializable - but still gives a consistent** result because the concept of Conflict-Serializability becomes limited when the **Precedence Graph** of a schedule contains a **loop/cycle.**
- In such a case we cannot predict whether a schedule would be consistent or inconsistent.
- As per the concept of Conflict-Serializability, We can say that a schedule is Conflict-Serializable (means serial and consistent) if its corresponding precedence graph does not have any loop/cycle.
- But, what if a schedule's precedence graph contains a cycle/loop and is giving consistent result/accurate results as a conflict serializable schedule is giving?
- So, to address such cases we brought the concept of **View-Serializability** because we did not want to confine the concept of serializability only to Conflict-Serializability.

Recoverable Schedules

- Always tow questions: serializability and recoverrability
- Situation where we recover the data (old database state) after some failure of the schedule.
- From the given schedule, find whether it is recoverable or irrecoverable? (ie whether the schedule is having the ability of getting recovered or not?

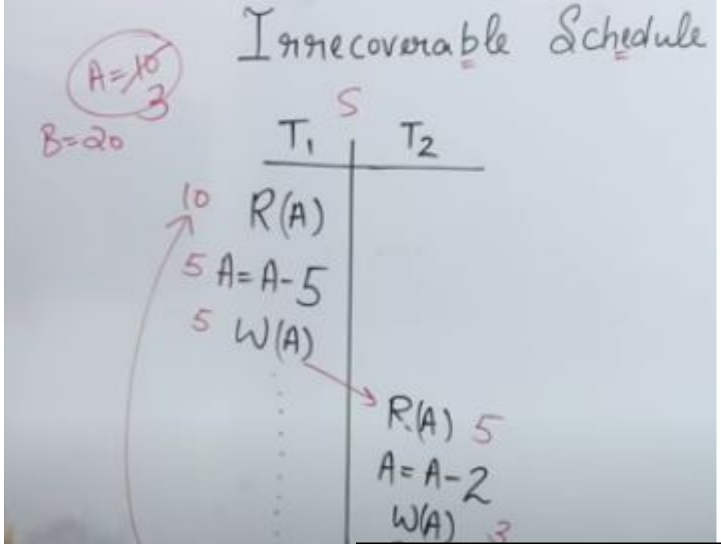


T1	T2
R(A)	
A=A-5	
W(A)	
	R(A)
	A=A-2
	W(A)
	Commit
R(B)	
fail	

B=20

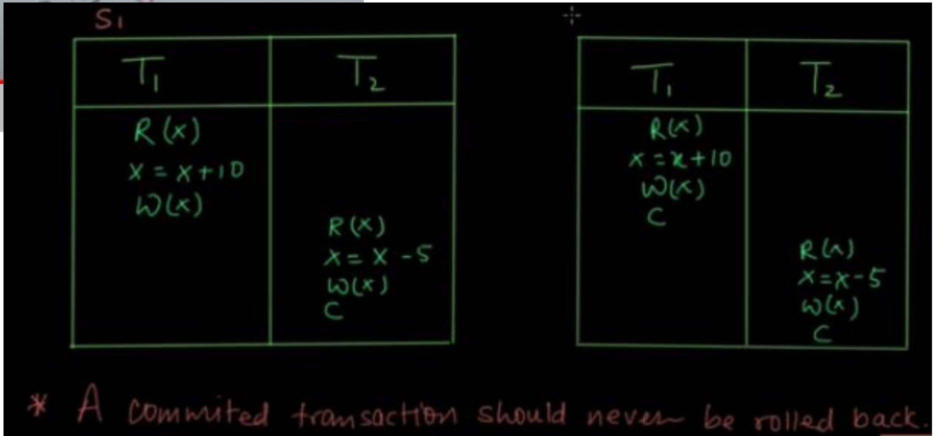
5 in shared memory but in db(it is 10)

Once after committed, it gets stored or reflected in db (ie durability)



T1	T2
R(A)	
A=A-5	
W(A)	
	R(A)
	A=A-2
	W(A)
R(B)	

If T1 failed after R(B) due to h/w or s/w failure, then as per atomicity, back to first ie rollbacked to start of T1, so everything lost. So it is irrecoverable


S1

T1	T2

R(x)
x = x+10
W(x)	

	R(x)
	x = x -5
	W(x)
	C

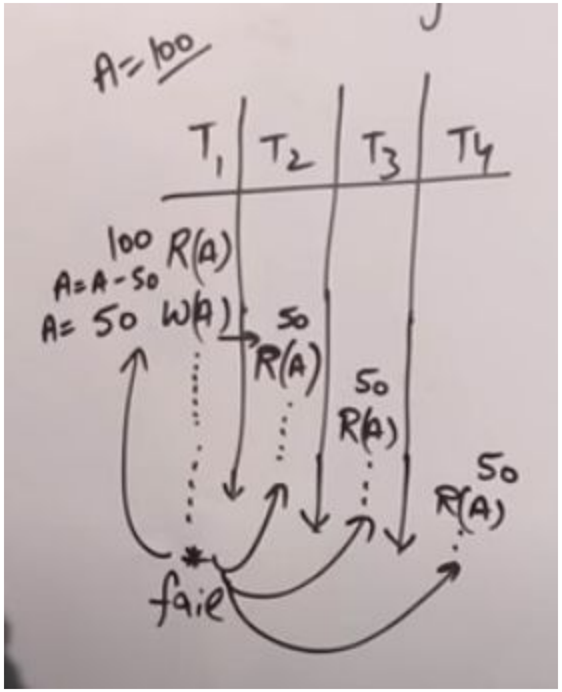
T1	T2

R(x)
x = x+10
W(x)
C	

	R(x)
	x = x-5
	W(x)
	C

* A commited transaction should never be rolled back.

Cascading and cascadless schedule



- This is dirty write- read. To avoid this perform rollback

Concurrency Control

- In a multiprogramming environment where **multiple transactions can be executed simultaneously**, it is highly important to control the concurrency of transactions.
- To ensure the system must control theinteraction among the concurrent transactions – through a mechanism concurrency control
- We have **concurrency control protocols** to ensure atomicity, isolation, and serializability of concurrent transactions.
- Concurrency control protocols can be broadly divided into two categories –
  - **Lock based protocols**
  - **Time stamp based protocols**

Lock-Based Protocols

- One way to ensure isolation is to require that **data items be accessed in a mutually exclusive** manner; that is, **while one transaction is accessing a data item**, **no other transaction can modify** that data item.
- The most common method - **holding a lock** on that item.

Locks:

two modes:

1. Shared.
2. Exclusive.

Shared lock: (lock-S)

- It is also known as a **Read-only lock**. In a shared lock, the **data item can only read by the transaction**.
- 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.
- Whenever a shared lock is used on a database, it can be **read by several users**, but these users who are reading the information or the data items **will not have permission to edit it** or make any changes to the data items.
- If a transaction **Ti has obtained a shared-mode lock** (denoted by S) **on item Q**, then **Ti can read, but cannot write, Q.**

Exclusive lock: (lock-X)

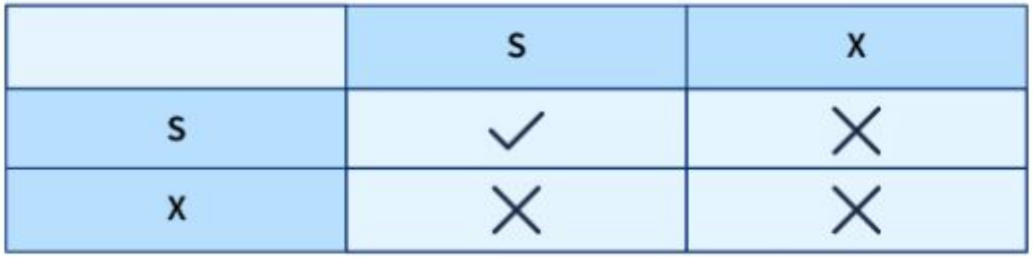
- In the exclusive lock or **write lock**, the data **item can be both reads as well as written by the transaction**.
- This lock is exclusive, and in this lock, **multiple transactions do not modify the same data simultaneously**.
- After **finishing the 'write' step**, transactions can **unlock the data item.**
- If a transaction **Ti has obtained an exclusive-mode lock** (denoted by X) **on item Q**, then **Ti can both read and write Q.**

Lock Compatibility Matrix

- **Shared Lock** can be **held by any amount of transactions.**
- On the other hand, an **Exclusive Lock** can only be **held by one transaction** in DBMS.
- this is because a **shared lock only reads** data but does not perform any other activities, whereas an **exclusive lock performs read as well as writing activities**.

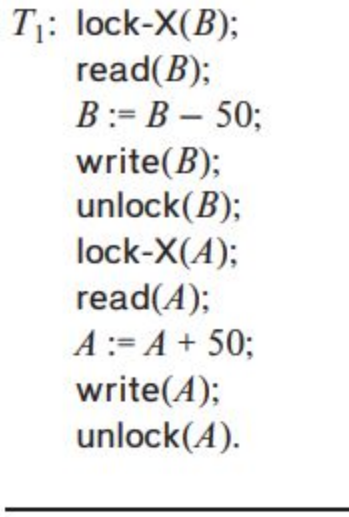
Example:

- when two transactions are involved, and **both of these transactions seek to read a specific data** item, the transaction is authorized, and **no conflict occurs**; but, in a situation **when one transaction intends to write the data item** and **another transaction attempts to read or write** simultaneously, the **interaction is rejected**.
-



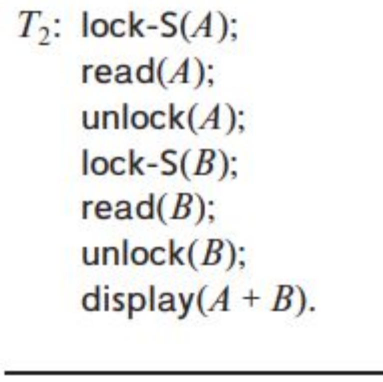
	S	X
S	✓	✗
X	✗	✗

- Suppose that the values of accounts A and B are $100 and $200, respectively.
- If these **two transactions are executed serially**,
  - either in the order T1, T2
  - or the order T2, T1,
  - then transaction T2 will display the value $300.
- If, however, these transactions **are executed concurrently**, then schedule 1, in Figure 18.4 (next slide), is possible.
- In this case, transaction T2 displays $250, which is incorrect. The reason for this mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state



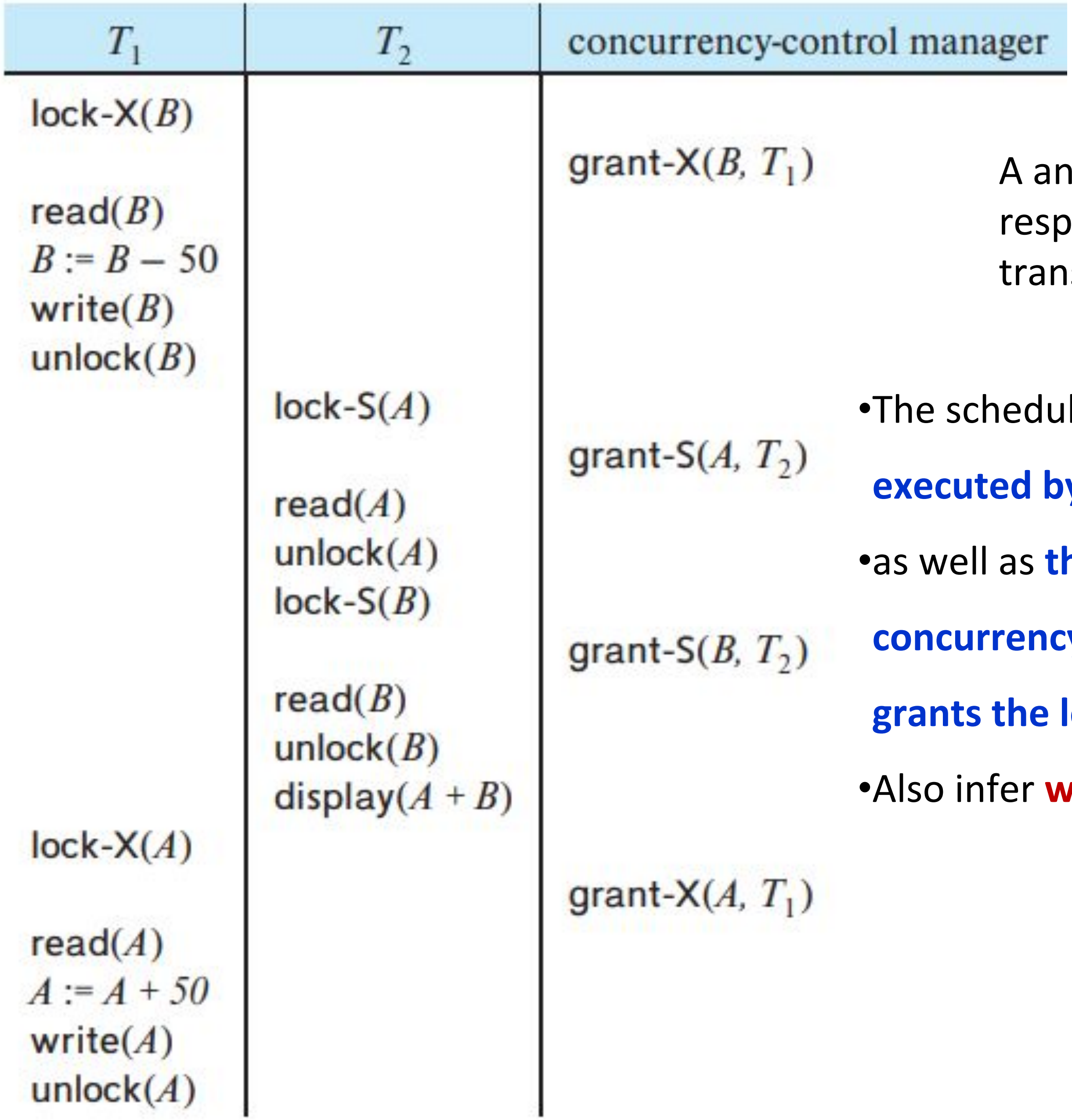
```
T1: lock-X(B);
    read(B);
    B := B – 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Transaction T_1.



```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Transaction T_2.



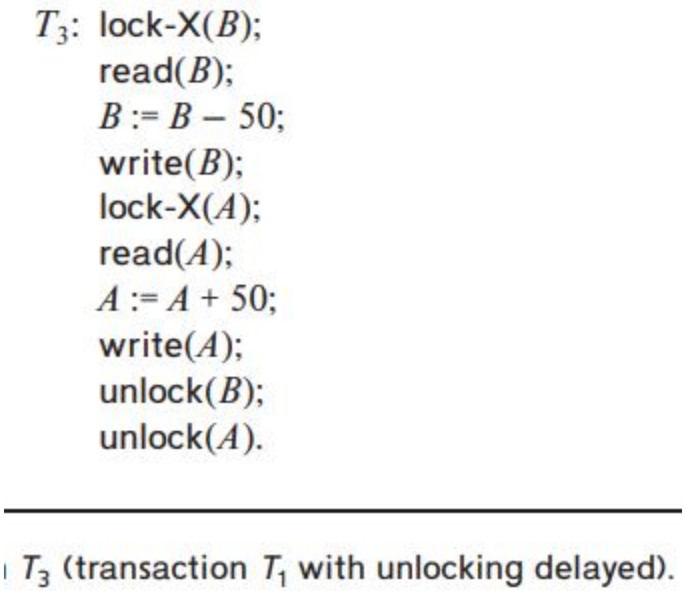
T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Fig. schedule 1

A and B are \$100 and \$200, respectively. ,
transaction T2 displays \$250

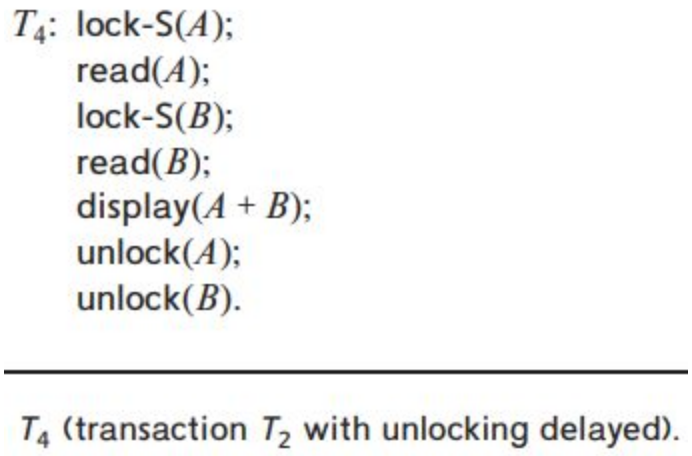
- The schedule shows the **actions executed by the transactions**,
- as well as **the points at which the concurrency-control manager grants the locks.**
- Also infer **when locks are granted.**

- Suppose now that unlocking is delayed to the end of the transaction.
- Transaction T3 corresponds to T1 with unlocking delayed (Figure 18.5).
- Transaction T4 corresponds to T2 with unlocking delayed (Figure 18.6).



```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

T_3 (transaction T_1 with unlocking delayed).



```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

T_4 (transaction T_2 with unlocking delayed).

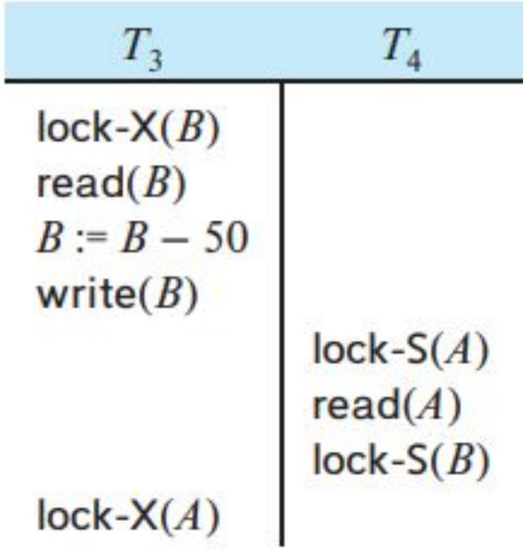
You should verify that the sequence of reads and writes in schedule 1, which lead to an incorrect total of $250 being displayed, is **no longer possible with *T3 and T4***.

deadlock

- Unfortunately, **locking can lead to an undesirable situation**.

Consider the partial schedule of Figure for T3 and T4.

partial schedule of Figure for T3 and T4.



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)	

Figure 18.7 Schedule 2.

- T3 - holding an **exclusive-mode lock on B**
- T4 - **requesting a shared-mode** lock on B
- T4 is waiting for T3 to unlock B.

Similarly,

- T4 - holding a **shared-mode lock on A**
- T3 - **requesting an exclusivemode lock** on A
- T3 is waiting for T4 to unlock A.
- Thus, we have arrived at a state where **neither of these transactions can ever proceed with its normal execution**. This situation is called **deadlock.**

Granting of Locks

- When a **transaction requests a lock on a data item in a particular mode**, and **no other transaction has a lock on the same data item in a conflicting mode**, **the lock can be granted**.

However, care must be taken to avoid the following scenario.

- Suppose a **transaction T2 has** a **shared-mode lock** on a data item, and another **transaction T1 requests** an **exclusive-mode lock** on the data item.
- **T1 has to wait for T2** to **release the sharedmode lock**.
- Meanwhile, a transaction **T3 may request** a **shared-mode lock** on the same data item.
- The **lock request is compatible with the lock granted to T2**, so **T3 may** be **granted the shared-mode lock**.
- At this point **T2 may release** the lock, but **still T1 has to wait for T3 to finish**.
- But again, there may be a **new transaction T4** that **requests a shared-mode lock** on the same data item, and is **granted the lock before T3 releases** it.
- but **T1 never gets the exclusive-mode lock** on the data item. The transaction T1 may never make progress, and is said to be **starved**.

□ **T2 – S(A)**

□ **T1 -> X(A) (need to wait)**

□ **T3 -> S(A) (compatible with T2) ,so T3- S(A)**

□ **T2 ---released the shared lock on A**

□ **T1 --- need to wait for T3 ‘s completion.**

□ **T4 -> S(A) (compatible with T3) ,so T4 - S(A)**

avoid starvation

- We can avoid starvation of transactions by granting locks in the following manner:
- When a transaction Ti requests a lock on a data item Q in a particular mode M, the **concurrency-control manager grants the lock** provided that:
  - There is no other transaction **holding a lock on Q in a mode** that **conflicts with M**.
  - There is no other transaction that is **waiting for a lock on Q** and that **made its lock request before Ti** .

locking protocol

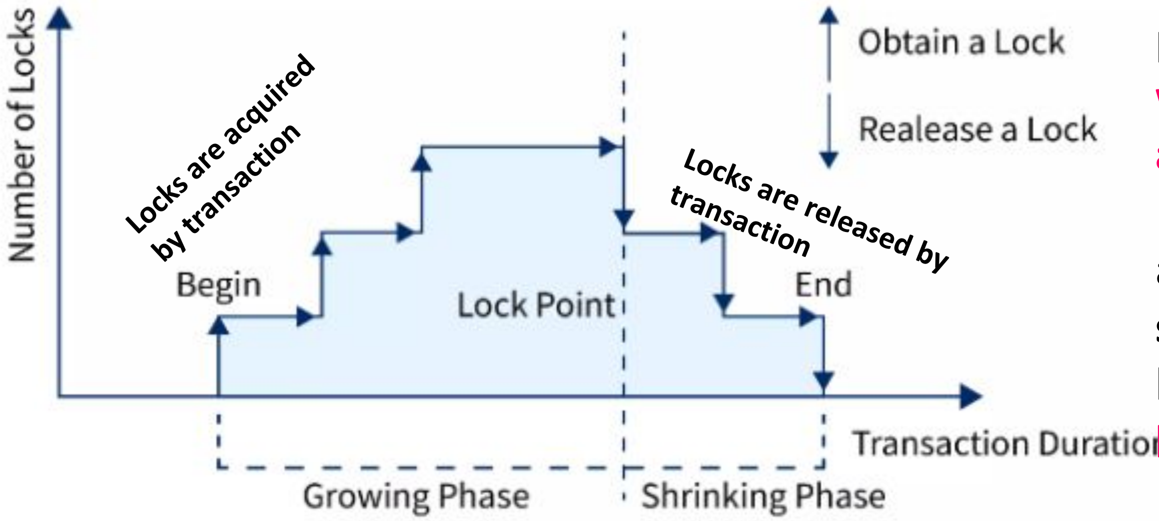
- require that **each transaction in the system follow a set of rules**, called a **locking protocol**, **indicating when a transaction may lock and unlock** each of the data items.
- Locking protocols restrict the number of possible schedules.
- present several locking protocols that allow only **conflict-serializable** schedules, and thereby **ensure isolation**

Two-Phase Locking Protocol

- One protocol that **ensures conflict serializability** is the two-phase locking protocol.
- This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase**. A transaction **may obtain locks**, but **may not release any lock**.
2. **Shrinking phase.** A transaction may **release locks**, but **may not obtain any new locks**

The point in the schedule where the **transaction has obtained its final lock** (the end of its growing phase) is called the lock point of the transaction.



Two-phase locking has two phases, one is **growing**, **where all the locks are being acquired** by the transaction;

and the second phase is **shrinking**, where **the locks** held by **the transaction are being released**.

- **Serial Schedule:** The serial schedule is **a type of Schedule where one transaction is executed completely before starting another transaction**.
- **Non-serial Schedule**: In a Non-serial schedule, **multiple transactions execute concurrently/simultaneously.**

- **Conflict Serializable:** A schedule is called conflict serializable **if it can be transformed into a serial schedule** by swapping non-conflicting operations.

- **Conflicting operations:** Two operations are said to be conflicting if all conditions satisfy:

❑ They **belong to different transactions.**

❑ They **operate on the same data item.**

❑ **At Least one** of them **is a write operation.**

- **Conflicting** operations pair **(R_1(A), W_2(A))** because they belong to two different transactions on same data item A and one of them is write operation.
- Similarly, **(W_1(A), W_2(A)) and (W_1(A), R_2(A))** pairs are also **conflicting**.
- On the other hand, **(R_1(A), W_2(B))** pair is **non-conflicting** because they operate on different data item.
- Similarly, **((W_1(A), W_2(B))** pair is **non-conflicting.**

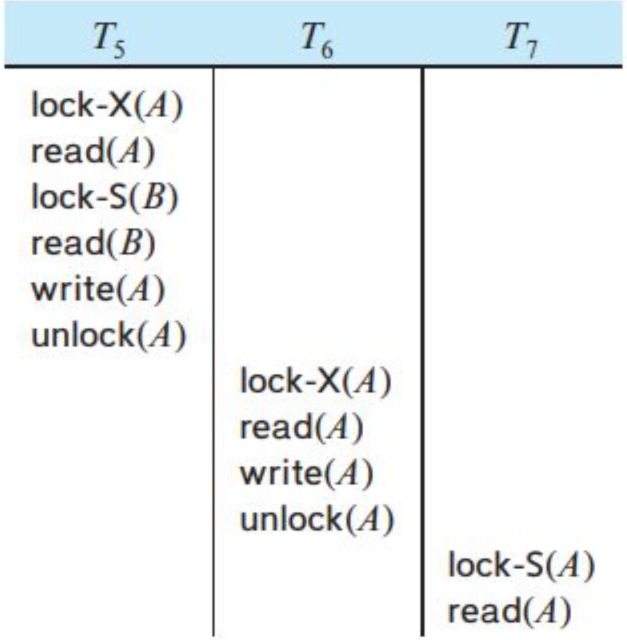
- Two-phase locking **does not ensure freedom from deadlock**.

- Observe that transactions T3 and T4 are two phase, but, in schedule 2 (Figure 18.7), they are deadlocked.

- in **addition to being serializable**, schedules **should be cascadeless.**

- Each transaction observes the two-phase locking protocol, but the failure of T5 after the read(A) step of T7 leads to cascading rollback of T6 and T7.

- **Cascading rollback may occur under two-phase locking.**



T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

18.8 Partial schedule under two-phase locking.

Cascading rollback

- **when a transaction (T1) causes a failure and a rollback must be performed**.
- **Other transactions dependent on T1's actions** must **also be rollbacked** due to T1's failure, thus causing a cascading effect
- **A single transaction failure leads to a series of transaction rollbacks** is called Cascading rollback.

- Cascading rollbacks can be **avoided by a modification of two-phase locking** called the **strict two-phase locking protocol**.

- This protocol **requires not only that locking be two phase**, but also that all **exclusive-mode locks** taken by a transaction **be held until that transaction commits.**

- Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that **all locks (both S and X) be held until the transaction commits.**

refinement of the basic two-phase locking protocol,

- which **allows lock conversions**.
- a mechanism for **upgrading** a **shared lock to an exclusive lock**, and **downgrading** an **exclusive lock to a shared lock**.
- We denote conversion from shared to exclusive modes by upgrade, and from exclusive to shared by downgrade.
- Lock **conversion cannot** be allowed arbitrarily (**randomly**).
- Rather, **upgrading** can ***take place in only the growing phase***, whereas **downgrading** can **take place in only the shrinking phase**.

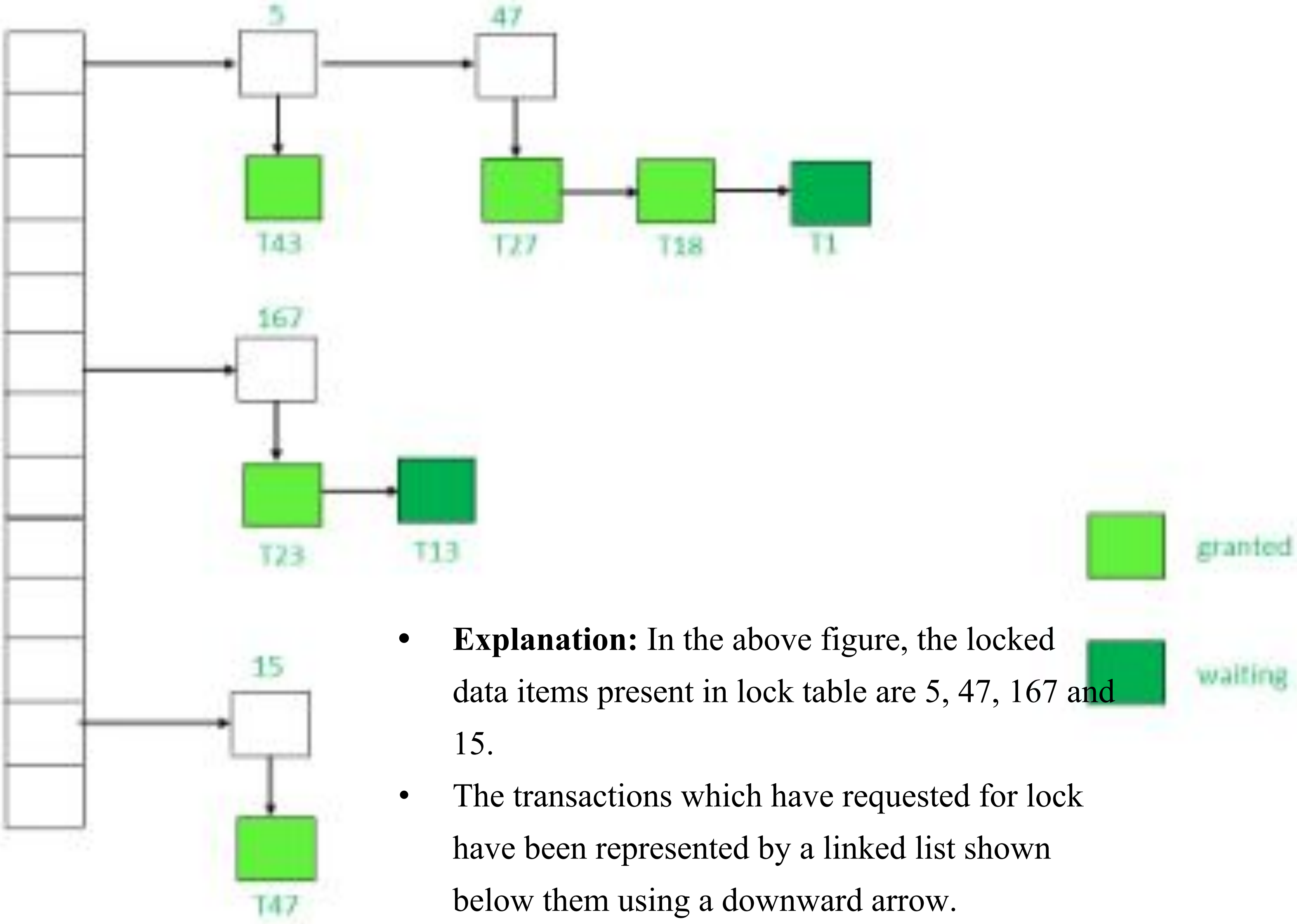
Implementation of Locking

- **Locking protocols** are used in database management systems **as a means of concurrency control.**
- **Multiple transactions may request a lock on a data item** simultaneously.
- Hence, we require a ***mechanism to manage the locking requests*** made by transactions. Such a mechanism is called as **Lock Manager**.
- It **relies on the process of message passing** where transactions and lock manager **exchange messages to handle the locking and unlocking** of data items.

Data structure used in Lock Manager –

The data structure required for implementation of locking is called as **Lock table**.

- It is a hash table where name of data items are used as hashing index.
- Each locked data item has a linked list associated with it.
- Every node in the linked list represents the transaction which requested for lock, mode of lock requested (mutual/exclusive) and current status of the request (granted/waiting).
- Every new lock request for the data item will be added in the end of linked list as a new node.
- Collisions in hash table are handled by technique of separate chaining.



- **Explanation:** In the above figure, the locked data items present in lock table are 5, 47, 167 and 15.
- The transactions which have requested for lock have been represented by a linked list shown below them using a downward arrow.
- Each node in linked list has the name of transaction which has requested the data item like T33, T1, T27 etc.
- The colour of node represents the status i.e.

Working of Lock Manager –

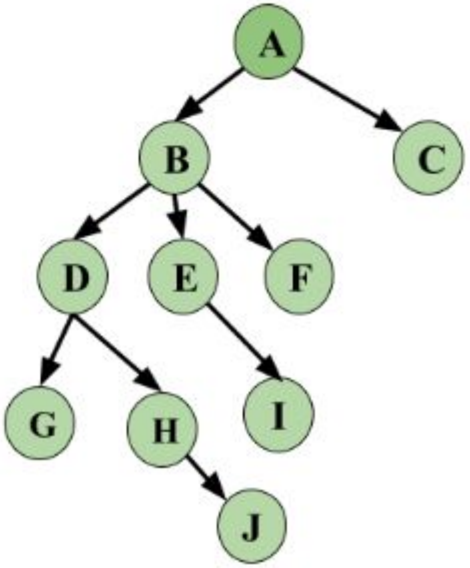
- Initially the lock table is empty as no data item is locked.
- Whenever lock manager receives a **lock request** from a transaction T_i on a particular data item Q_i following cases may arise:
  - If Q_i is not already locked, a **linked list will be created** and **lock will be granted** to the requesting transaction T_i.
  - If the data item is already locked, a new node will be added at the end of its linked list containing the information about request made by T_i.
- If the lock mode requested by T_i is **compatible** with lock mode of transaction currently having the lock, T_i will acquire the lock too and status will be changed to 'granted'. Else, status of T_i's lock will be 'waiting'.
- If a transaction T_i wants to **unlock the data item** it is currently holding, it will send an unlock request to the lock manager. The lock manager **will delete T_i's node** from this linked list. **Lock will be granted to the next transaction** in the **list**.
- Sometimes transaction T_i may have to be aborted. In such a case all the waiting request made by T_i will be deleted from the linked lists present in lock table. Once abortion is complete, locks held by T_i will also be released.

Graph Based Concurrency Control Protocol

- another way of implementing Lock Based Protocols.
- *Tree Based Protocols is a simple implementation of Graph Based Protocol.*
- A **prerequisite** of this protocol is that **we know the order to access a Database Item.**
- For this we implement a **Partial Ordering** on a set of the **Database Items (D)** $\{d_1, d_2, d_3, \ldots, d_n\}$. The protocol following the implementation of Partial Ordering is stated as-
- If $d_i \rightarrow d_j$ then any transaction **accessing both d_i and d_j** must access d_i before accessing d_j.
- Implies that the set **D** may now be viewed as a **directed acyclic graph** (DAG), called a *database graph*.

Tree Based Protocol –

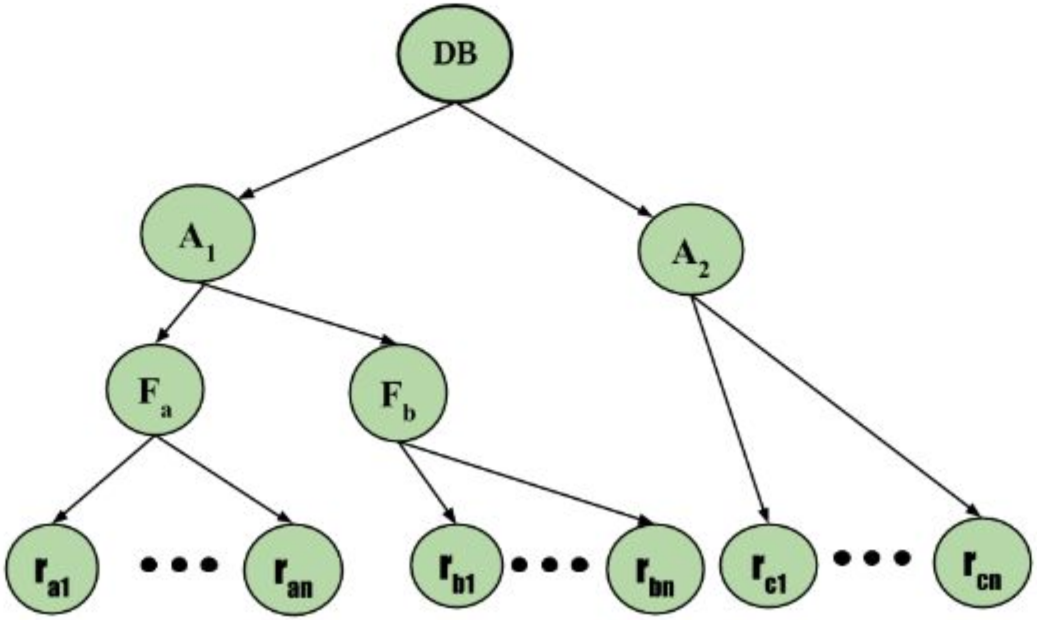
- Partial Order on Database items determines a **tree like structure**.
- **Only Exclusive Locks** are allowed.
- The first lock by T_i may be on any data item. Subsequently, **a data Q can be locked by T_i** only if the parent of Q is currently locked by T_i.
- Data items can be **unlocked at any time**.



Multiple Granularity

- **Granularity –** It is the size of the data item allowed to lock.
- Now *Multiple Granularity* means hierarchically **breaking up the database into blocks** that can **be locked** and can be tracked needs what needs to lock and in what fashion.
- Such a hierarchy can be **represented graphically as a tree.**
- the levels starting from the top level are:

- database
- area
- file
- record



- it shall use shared and exclusive lock modes.
- When a transaction locks a node, in either shared or exclusive mode, the transaction also **implicitly locks all the descendants** of that node in the **same lock mode**.
- For example, if transaction T_i gets an explicit lock on file F_c in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file.
- It does not need to lock the individual records of F_c explicitly.

introduce a new lock mode, called *Intention lock mode*.

Intention Mode Lock –

In addition to **S** and **X** lock modes, there are three additional lock modes with multiple granularities:

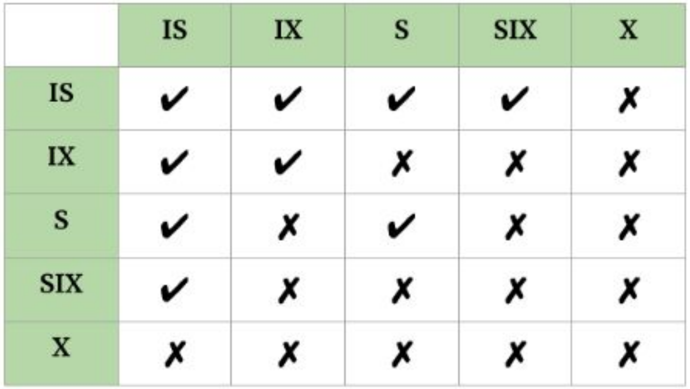
- **Intention-Shared (IS):** explicit locking at a lower level of the tree but only with **shared locks.**
- **Intention-Exclusive (IX):** explicit locking at a lower level with **exclusive or shared locks**.
- **Shared & Intention-Exclusive (SIX):** the **subtree rooted by that node** is **locked explicitly in shared mode** and **explicit locking** is being done at a **lower level with exclusive mode** locks.

Compatibility matrix

- It requires that a transaction T_i that attempts to lock a node must follow these protocols:

- Transaction T_i must follow the lock-compatibility matrix.
- Transaction T_i must lock the root of the tree first, and it can lock it in any mode.
- Transaction T_i can lock a node in S or IS mode only if T_i currently has the parent of the node-locked in either IX or IS mode.
- Transaction T_i can lock a node in X, SIX, or IX mode only if T_i currently has the parent of the node-locked in either IX or SIX modes.
- Transaction T_i can lock a node only if T_i has not previously unlocked any node (i.e., T_i is two-phase).
- Transaction T_i can unlock a node only if T_i currently has none of the children of the node-locked.

•Observe that the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf to-root) order.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

IS : Intention Shared
IX : Intention Exclusive
S : Shared

X : Exclusive
SIX : Shared & Intention Exclusive

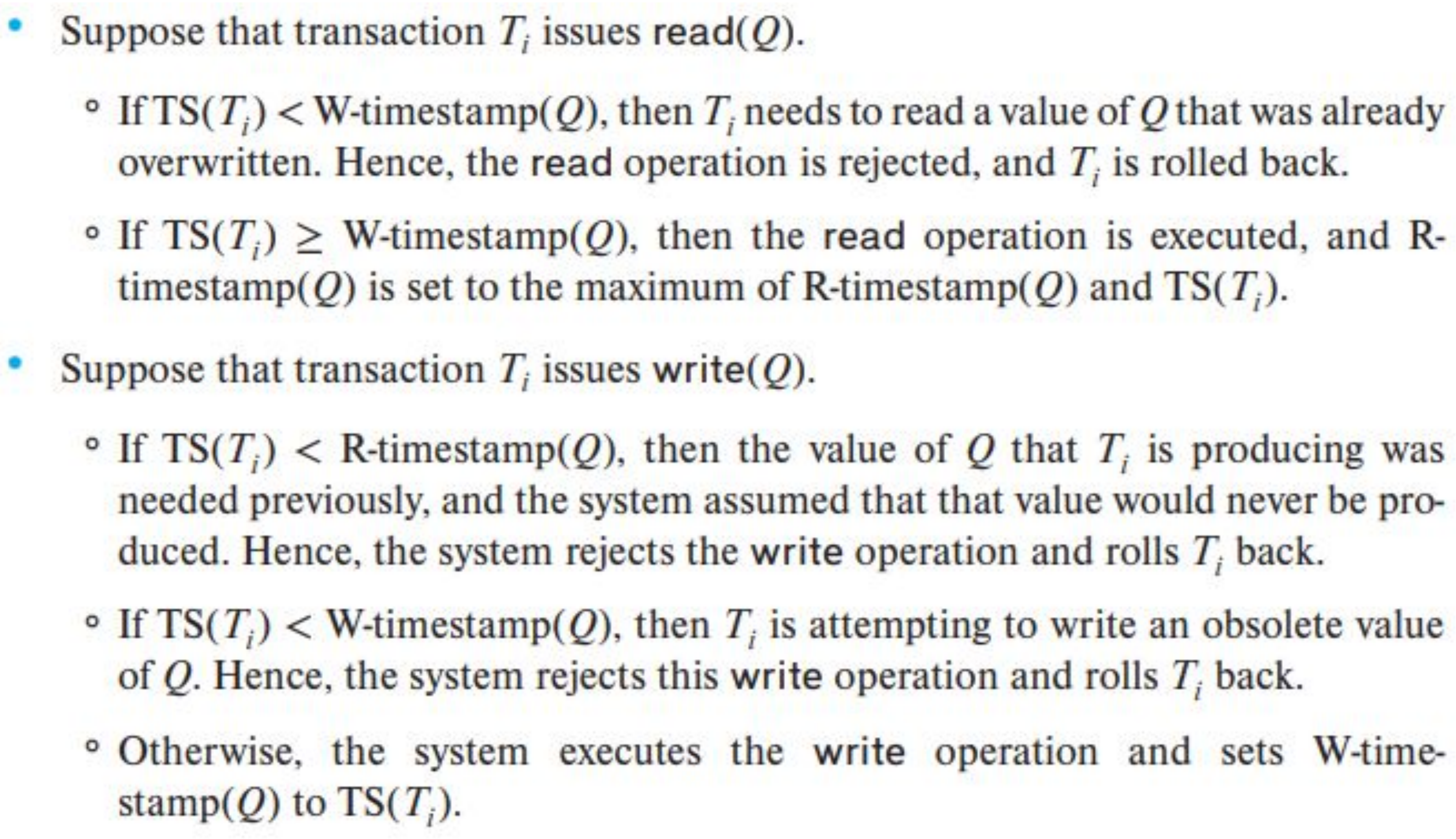
- Say transaction **T_1 reads record R_{a2} in file F_a**. Then, T_1 needs to lock the database, area **A_1, and F_a in IS mode** (and in that order), and finally to lock **R_{a2} in S mode**.
- Say transaction **T_2 modifies record R_{a9} in file F_a** . Then, T_2 needs to lock the database, **area A_1, and file F_a** (and in that order) in **IX mode**, and at last to lock **R_{a9} in X mode.**
- Say transaction **T_3 reads all the records in file F_a**. Then, T_3 needs to lock the **database and area A_1** (and in that order) in **IS mode**, and at last to lock **F_a in S mode.**
- Say transaction **T_4 reads the entire database**. It can do so after locking the **database in S mode.**

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is **used to order the transactions based on their Timestamps.** The order of transaction is nothing but the **ascending order** of the transaction creation.
- A **timestamp is a unique identifier** that is being created by the DBMS when a transaction enters into the system.
- The **priority of the older transaction is higher** that's why it executes first.
- To **determine the timestamp** of the transaction, this protocol uses **system time or logical counter.**
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.

- Let's assume there are two transactions T1 and T2. Suppose the transaction **T1** has entered the system at **007 times** and transaction **T2** has entered the system at **009 times. T1 has the higher priority**, so it executes first as it is entered the system first.
- The timestamp ordering protocol **also maintains the timestamp of last 'read' and 'write' operation on a data.**
- For example, let's say an old transaction T1 timestamp is TS(T1) and a new transaction T2 enters into the system, timestamp assigned to T2 is TS(T2). Here TS(T1) < TS(T2) so the **T1 has the higher priority because its timestamp is less than timestamp of T2**. T1 would be given the higher priority than T2. This is how timestamp based protocol maintains the **serializability order**.

- A **conflict occurs** when an **older transaction tries to read/write a value already read or written by a younger transaction**. Read or write proceeds only if the last update on that data item was carried out by an older transaction.
- Conflict is resolved by **rolling back and restarting transactions**.



- Suppose that transaction T_i issues read(Q).
  - If TS(T_i) < W-timestamp(Q), 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 TS(T_i) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).
- Suppose that transaction T_i issues write(Q).
  - If TS(T_i) < R-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls T_i back.
  - If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls T_i back.
  - Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).