Unit - 4

Transaction Concept / Management.

Contents :-

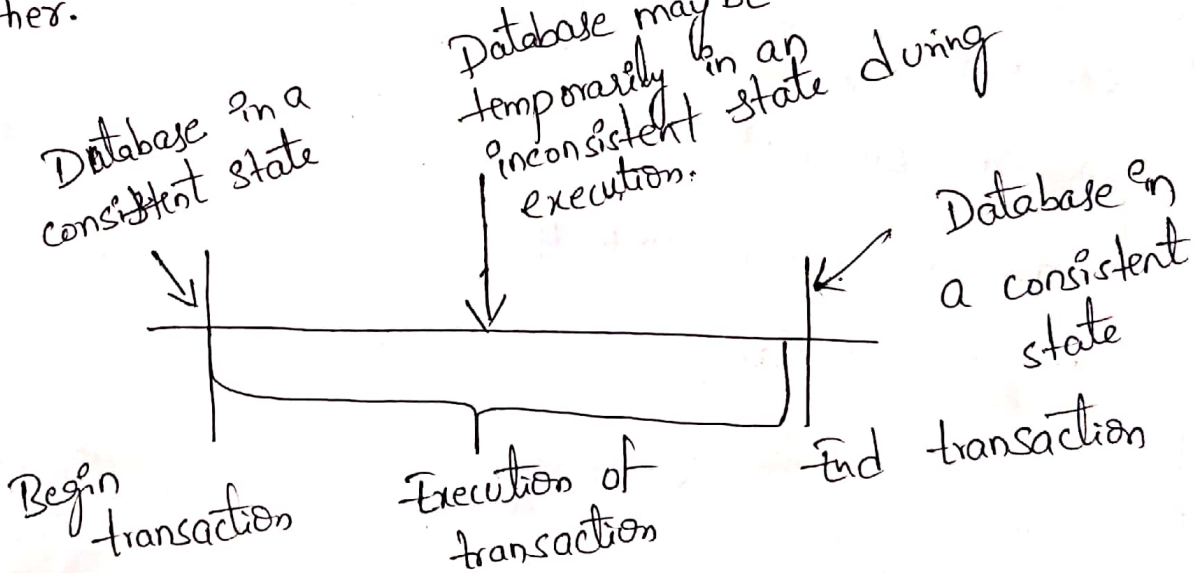
- Transaction Concept
- Transaction state
- Implementation of Atomicity and Durability.
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Testing for Serializability
- Lock based protocols
- Timestamp Based Protocols
- Validation based Protocols
- Multiple Granularity
- Recovery & Atomicity
- Log-Based Recovery
- Recovery with concurrent executions.

Transaction Concept :-

→ A transaction is a unit of a program execution that accesses and possibly modies various data objects (tuples, relations)

→ All types of database access operation which are held between the begining and end transaction statements are considered as a single logical transaction in DBMS.

→ During the transaction the database is inconsistent. only once the database is committed the state is changed from one consistent state to another.



→ DBMS can be mainly used to deal with

- Failures of various kinds, such as hardware failures and system clashes
- Concurrent execution of multiple transactions.

→ The main operations in a transaction are

1. Read operation
2. Write operation.

1. Read operation :-

→ Read operation reads the data from the database and then stores it in the buffers in main memory

Eg:- Read(A) instruction will read the value of A from the database and will store it in the buffer in main memory.

2. Write operation :-

→ Write operation writes the updated data value back to the database from the buffer.

Eg:- Write(A) will write the updated value of A from the buffer to the database.

ACID Properties:- ACID Properties are used for maintaining the data integrity of database during transaction Processing. ACID in DBMS stands for Atomicity, Consistency, Isolation, and durability.

Atomicity:- A transaction is a single unit of operation, you either execute it entirely or do not execute it at all. There cannot be partial execution

Consistency:- Once the transaction is executed, it should move from one consistent state to another.

Isolation:- Transaction should be executed in isolation from other transactions (no locks). During concurrent transaction execution, intermediate transaction results from simultaneously executed transactions should not be made available each other.

Durability:- After successful transaction, the changes in the database should persist. Even in the case of system failures.

Example:-

Transaction 1: Begin. X=X+50, Y=Y-50
END

Transaction 2: Begin X=1.1*X, Y=1.1*Y
END

→ Transaction 1 is transferring $50 from account X to account Y.

→ Transaction 2 is crediting each account with a 10% interest payment.

If the both transactions are submitted together, there is no guarantee that the transaction 1 will execute before transaction 2 or vice versa.

→ Irrespective of the order, the result must be as if the transactions take place serially one after the other.

Transaction States :-

States through which a transaction goes during its lifetime. These are the states which tell about the current state of the transaction and also tell how we will further do processing we will do on the transactions.

→ These are different types of transaction states.

1. Active state
2. Partially committed state
3. Failed state
4. Committed state

5. Aborted state.

Active :- When the instructions of the transactions 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".

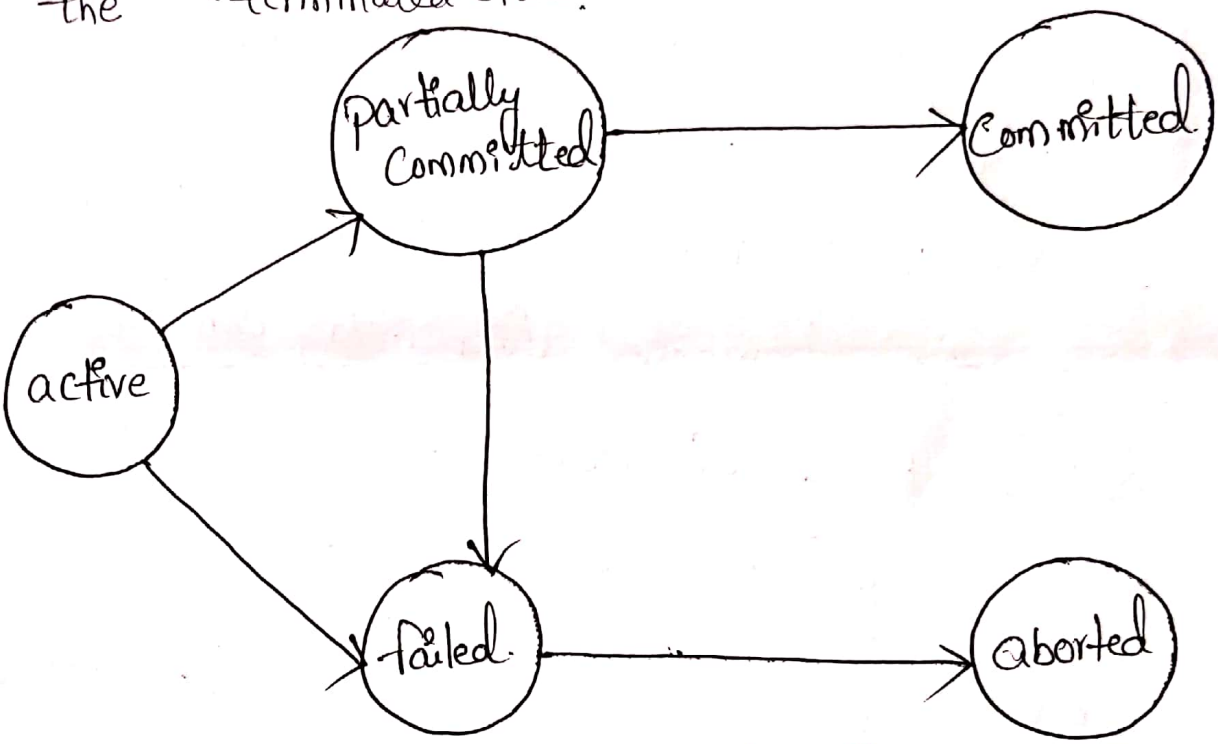
Partially committed :- After completion of all the read and write operation the changes are made in main memory & local buffer. If the changes are made permanent on the data Base then the state will change to "committed state" and in case of failure it will go to the "failed state".

Failed :- When any ~~transaction~~ instruction of the transaction fails, it goes to the "failed state" & 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 & main memory and hence these changes are deleted & 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 terminted in the "terminated state".



Implementation of Atomicity and Durability

The recovery-management component of a database system implements the support for atomicity and durability.

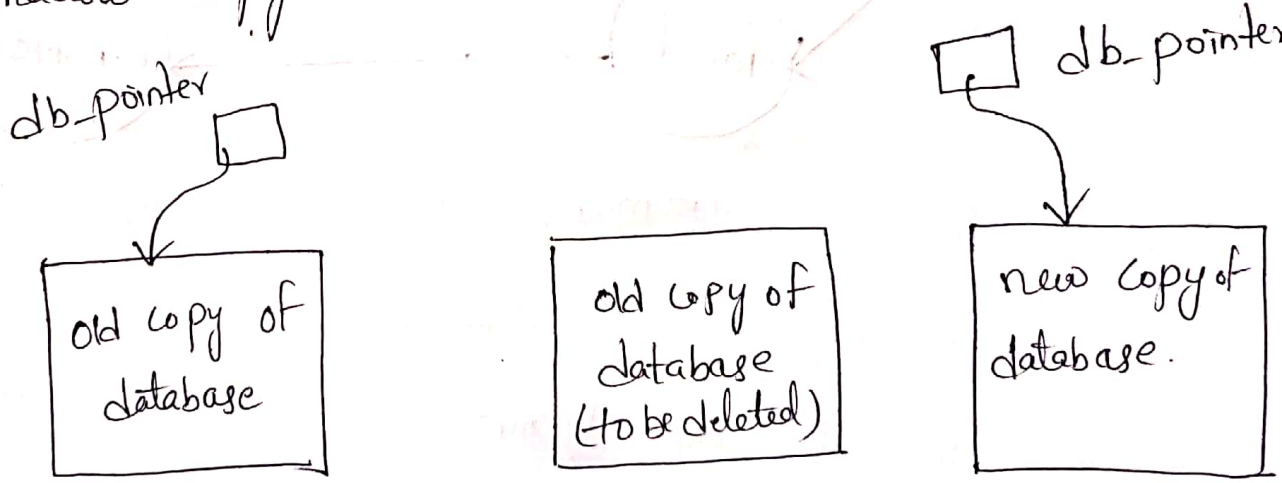
→ the shadow-database scheme :-

→ assume that only one transaction is active at a time

→ a pointer called db-pointer always points to the current consistent copy of the database.

→ all updates are made on a shadow copy of the database, and db-pointer is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.

→ In case transaction fails, old consistent copy pointed to by db-pointer can be used, and the shadow copy can be deleted.



(a) Before update

(b) After update

The shadow-database scheme

Concurrent Executions :-

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

increased processor and disk utilization leading to better transaction throughput: One transaction can be using the CPU while another is reading from or writting to the disk.

reduced average response time for transactions. short transactions need not wait behind long ones.

Problems with concurrent execution :-

In a database transaction, the two main operations were Read and Write operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So the following problems occur with the concurrent execution of the operations.

Problem 1: lost update Problems (W-W conflict)

Problem 2: Dirty Read Problems (W-R conflict)

Problem 3: Unrepeatable Read Problem (W-R conflict)

Lost update Problems :- the problems occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner that makes the values of the items incorrect hence making the database inconsistent.

For example :- Consider the below diagram where two transactions Tx and Ty, were performed on the same account A where the balance of account A is $300

Time	Tx	Ty
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)

Lost update problem.

→ At time t_1, transaction Tx reads the value of account A, $300 (only read)

→ At time t_2, transaction Tx deducts $50 from account A that becomes $250 (only deducted and not updated/write)

→ Alternatively, at time t_3, transaction Ty adds $100 to account A that becomes $400 (only added but not updated/write)

→ At time t_4, transaction Ty adds $100 to account A that becomes $400 (only added but not updated/write)

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

→ Similarly, at time t_7, transaction t_8 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, i.e $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

Data 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 rollback, the updated database item is accessed by another transaction. There comes the read-write conflict between both transactions.

For example:- Consider two transactions Tx and Ty in the below diagram performing read/write operations on account A where the available balance in account A is $300.

Time	Tx	Ty
t_1	Read (A)	–
t_2	A = A+50	–
t_3	Write (A)	–
t_4	–	Read (A)
t_5	Server down rollback	–

Dirty read problem.

→ At time t_1, transaction Tx reads the value of account A, i.e, $300.

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

→ At time t_3, transaction Tx writes the updated value in account A, i.e $350

→ Then at time t_5, Transaction Tx rollbacks due to serves problem, and the value changes back to $300 (as initially)

→ But the value for account A remains $350 for transaction Ty as committed, which is the dirty read and therefore known as the dirty read problem.

Unrepeatable Read problem:-

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

for example:- Consider two transactions, Tx and Ty performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	Tx	Ty
t_1	Read(A)	—
t_2	—	Read(A)
t_3	—	A = A+100
t_4	—	Write(A)
t_5	Read(A)	—

Unrepeatable read problem.

→ At time t_1, transaction T_x reads the value from account A, i.e $ 300

→ At time t_2, transaction T_y reads the value from account A. i.e $ 300

→ At time t_3, transaction T_y updates the value of account A by adding $100 to the available balance and then it becomes $400

→ At time t_4, transaction T_y writes the updated value i.e $400

→ After that, at time t_5, transaction Tx, it reads two different values of account A, i.e. $ 300 initially and after updation made by transaction Ty, it reads $400. It is an unrepeatable read and is therefore known as the unrepeatable read problem.

Concurrency control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols :- The concurrency control protocols ensure the atomicity, consistency, isolation, durability and serializability of the concurrent execution of the database transactions. Therefore these protocols are categorized as:

- Lock based Concurrency control protocol
- Time stamp Concurrency control Protocol
- Validation Based concurrency control Protocol.

→ We will understand and discuss each protocol one by one in our next sections.

Lock-Based Protocol :-

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

→ shared lock

→ Exclusive lock

Shared lock :-

→ 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 ~~holds a lock, then~~ because when the transaction holds a lock, then it can't update the data on the data item.

Exclusive lock :-

→ In the exclusive 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.

Serializability and recoverability :-

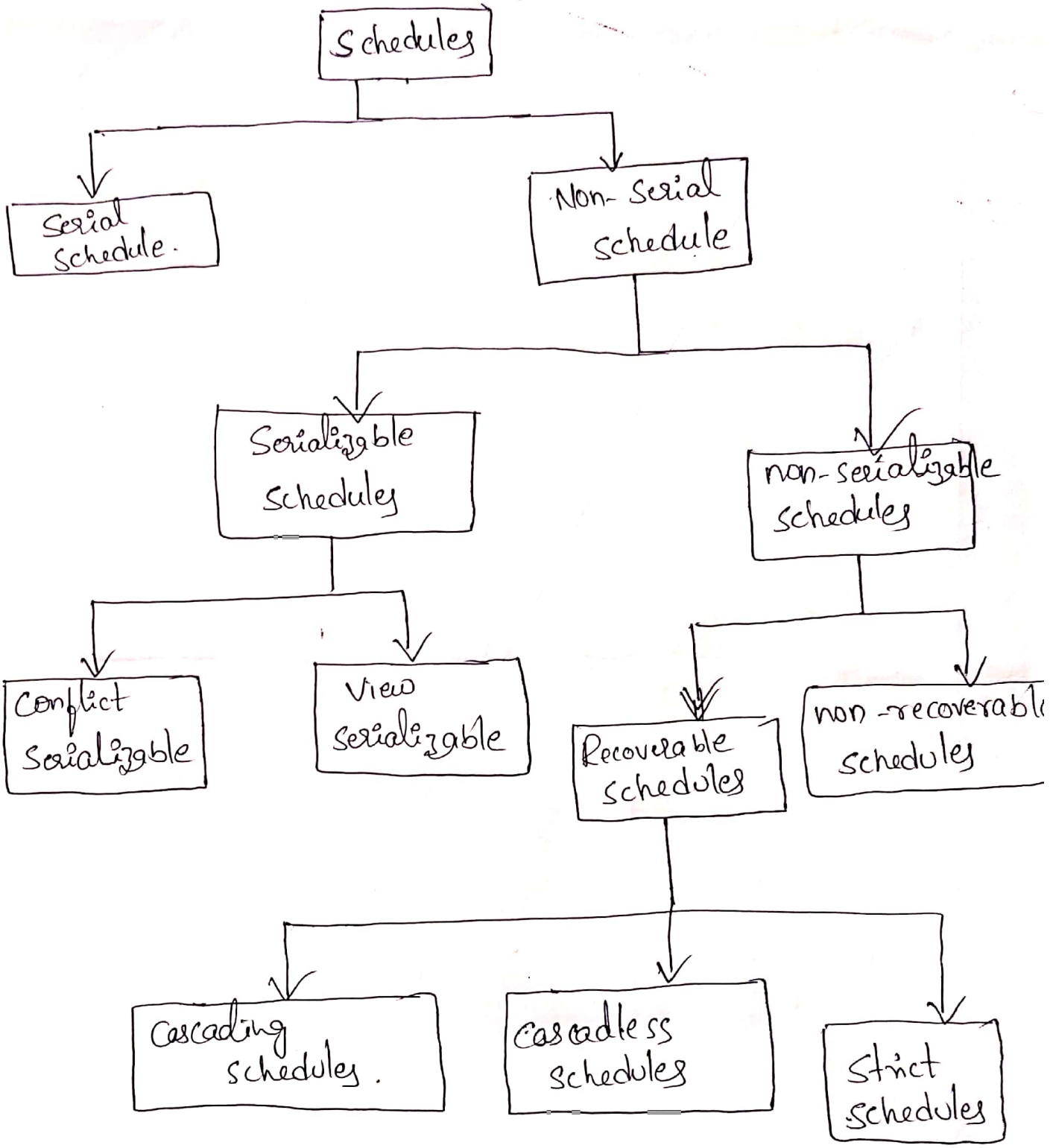
When multiple transactions are being executed by the operating system in a multiprogramming environment. there are possibilities that instructions of one transactions are interleaved with some other transaction.

Schedule :-

→ A chronological execution sequence of a transaction is called a schedule

→ A schedule can have many transactions in it, each comprising of a number of instructions / tasks.

Types of schedules in dbms :-



Serial schedule :-

A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.

→ Otherwise, the schedule is called non serial schedule.

Non-serial schedule :-

Serializable schedule :-

Serializable schedules are always considered to be correct when concurrent transactions are executing.

→ The main difference blw the serial schedule and the serializable schedule is that in serial schedule, no concurrency is allowed where as in serializable schedule, concurrency is allowed.

Eg :- Let T_1 transfer $50 from A to B, and T_2 transfer 10% of the balance from A to B.

→ A serial schedule in which T_1 is followed by T_2

Schedule 1

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

Schedule 2:-

→ A serial schedule where T_2 is followed by T_1

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

Schedule 3:-

→ Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but is equivalent to schedule 1.

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

∴ In schedules 1, 2 and 3, the sum A+B is preserved.

Schedule 4:- the following concurrent schedule does not preserve the value of (A+B)

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

There are 2 types of equivalence of schedules:

1) conflict equivalence
2). View equivalence.

Conflicting operations :-

Two operations said to be in conflict, if they satisfy all the following three conditions:

1) Both the operations should belong to different transactions
2. Both the operations are working on same data item
3. At least one of the operation is a write operation.

Eg:- I_s = read(Q), I_m = read(Q), Is and Im don't conflict

I_s = read(Q), I_m = write(Q), they conflict

I_s = write(Q), I_m = read(Q), they conflict

I_s = Write(Q), I_m = write(Q), They conflict.

Conflict equivalence / Serializability :-

We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

Example of a schedule that is not conflict serializable

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

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

Constructing precedence graph to test conflict serializability

Precedence Graph & serialization graph is used to test conflict Serializability of a schedule. It is a directed graph (V, E) consisting of a set of nodes $V = \{T_1, T_2, T_3 \ldots T_n\}$ and a set of directed edges $E = \{e_1, e_2, e_3 \ldots e_n\}$.

The algorithm can be written as:

1. Create a node T in the graph for each participating transaction in the schedule.
2. For conflicting operation read_item(x) and write_item(x) - If a transaction T_j executes a read_item(x) after T_i executes a write_item(x), draw an edge from T_i to T_j in the graph.
3. For the conflicting operation write_item(x) and read_item(x) - If a transaction T_j executes write_item(x) after T_i executes a read_item(x), draw an edge from T_i to T_j in the graph.
4. For the conflicting operation write_item(x) and write_item(x) - if a transaction T_j executes a write_item(x) after T_i executes a write_item(x), draw an edge T_i to T_j in the graph.

5. The schedule S is serializable is there no cycle in the precedence graph.

If there is no cycle in the precedence graph, it means we can construct a serial schedule S' which is conflict equivalent to the schedule S.

For example :-

Consider the schedule S:

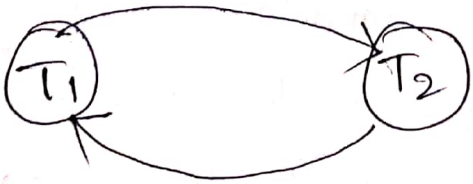
$$S: r_1(x)\ r_1(y)\ w_2(x)\ w_1(x)\ r_2(y)$$

Creating Precedence graph :-

1. Make two nodes corresponding to transaction T_1 and T_2

2. for the conflicting pair $r_1(x)\ w_2(x)$, where $r_1(x)$ happens before $w_2(x)$, draw an edge from T_1 to T_2

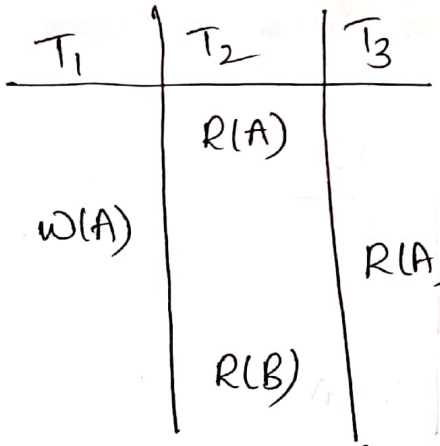
3. For the conflicting pair $w_2(x)\ w_1(x)$, where $w_2(x)$ happens before $w_1(x)$, draw an edge from T_2 to T_1



∴ Since the graph is cyclic, we can conclude that it is not conflict serializable to any schedule serial schedule.

View equivalent Serializability :— Two schedules S_1 and S_2 are said to be view-equivalent if below conditions are satisfied.

1) Initial read :— If a transaction T_i reading data item A from database in S_1 then in S_2 also T_i should read A from database.



Transaction T_2 is reading A from database.

2). Updated Read :-

If T_i is reading A which is updated by T_j in S_1 then in S_2 also T_i should read A which is updated by T_j.

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

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

Above two schedule are not view-equivalent as in $S1: T_3$ is reading A updated by T_2, in S_2 T_3 is reading A updated by T_1.

3) Final write Operation :-

If a transaction T_1 updated A at last in S1, then in S_2 also T_1 should perform final write operations.

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

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

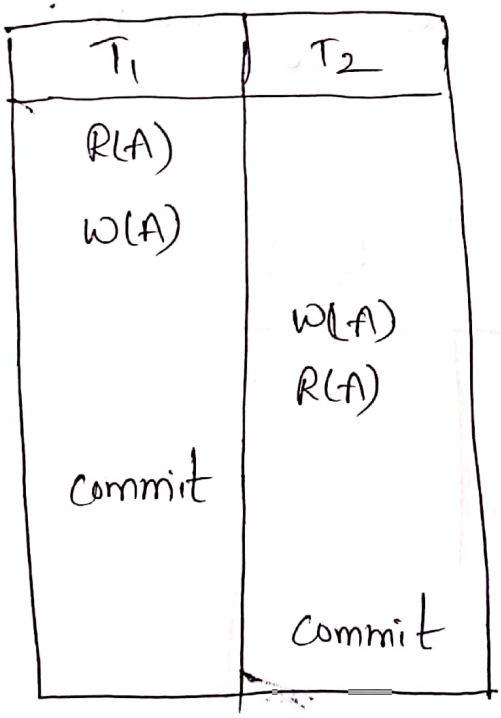
above two Schedules are not view-equivalent as final write Operation in S_1 is done by T_1 while in S_2 done by T_2.

→ View Serializability :- A Schedule is called View Serializability if it is view equal to Serial schedule.

Non-Serializable Schedules :- The non-serializable schedule is divided into two types, Recoverable and non-recoverable schedule.

* Recoverable Schedule :- Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules.

Ex:- Consider the following schedule involving two transaction T_1 and T_2



this is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct. there can be three types of recoverable schedules

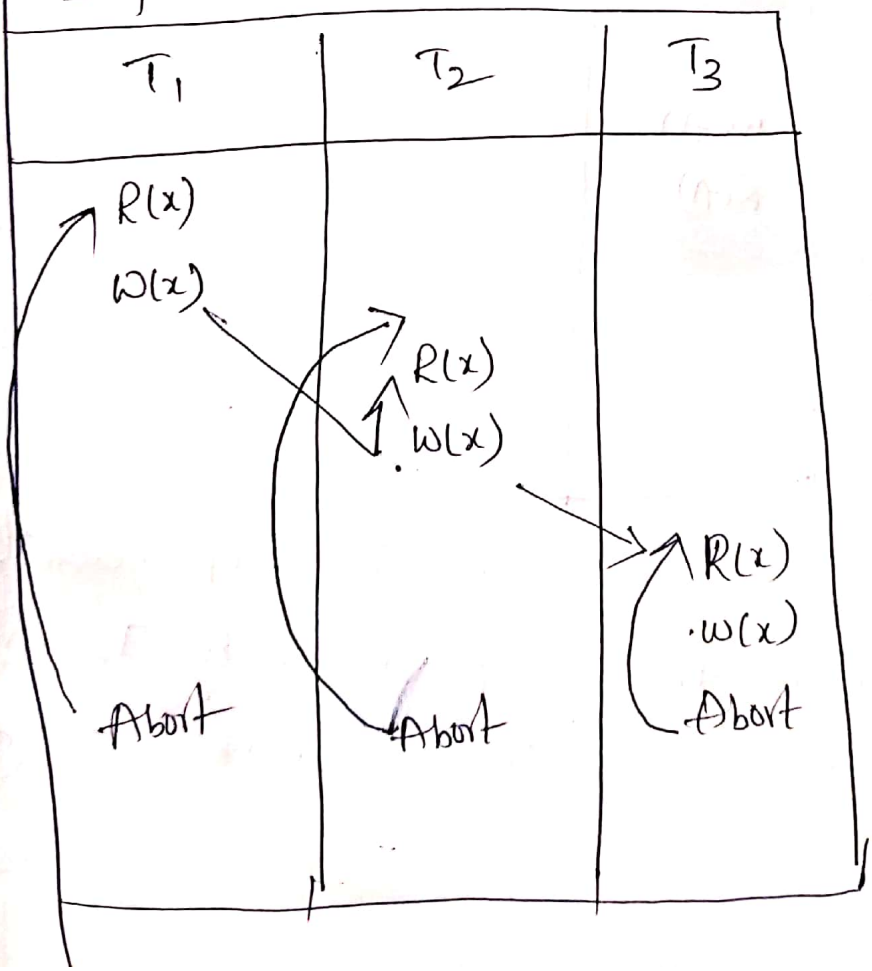
a. Cascading schedule

b. Casecadeless schedule

c. Stict schedule.

a. Cascading schedule :- also called avoids cascading aborts / rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such a scheduling is referred as cascading rollback or ~~cascading~~ cascading abort.

Example :-



b) Cascadeless Schedule :-

Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules.

→ Avoids that a single transaction abort leads to a series of transaction rollbacks.

Eg:- Consider the following schedule involving two transactions T_1 and T_2

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

This schedule is cascadeless. Since the updated value of A is read by T_2 only after the updating transaction i.e T_1 commits.

(c) Strict schedule :- A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j, then the commit or abort event of T_i also precedes that conflicting operation of T_j.

Example :- Consider the following schedule involving two transactions T_1 and T_2

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

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

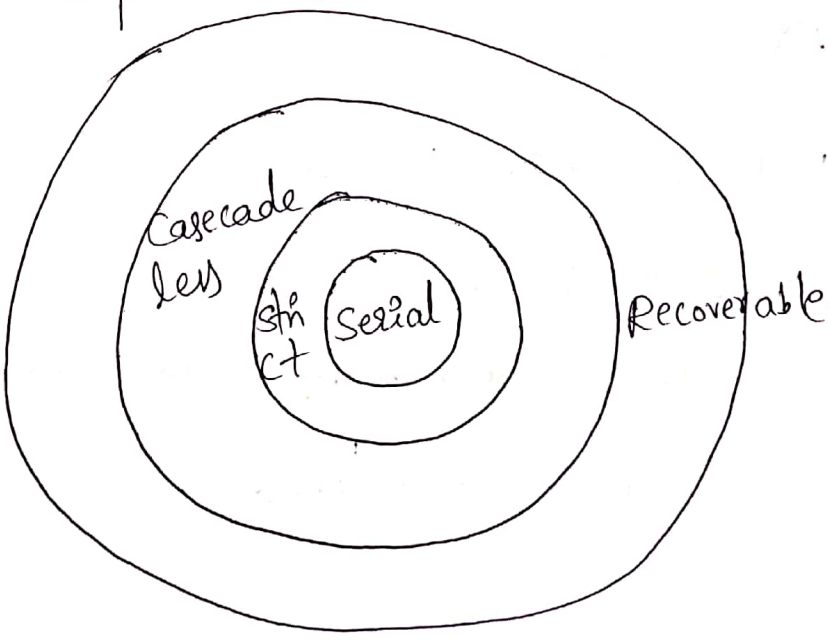
Non-recoverable schedules :-

Ex: Consider the following schedule involving two transa-ctions T_1 and T_2

T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is non-recoverable.

→ the relation between various types of schedules can be depicted as



Implementation of Isolation Levels:-

→ As we know that, in order to maintain consistency in a database, it follow ACID properties.

→ Among these 4 properties isolation determines how transaction integrity is visible to other users and systems.

→ Isolation levels define the degree to which a transaction (reads a data that has not yet been committed) must be isolated from the data modifications made by any other transaction in the database system. A transaction isolation level is defined by the following phenomena.

- → Dirty Read
- → Non Repeatable read
- → Phantom Read.

Based on these phenomena, the SQL Standard defines four isolation levels.

a) Read uncommitted :- Read uncommitted is the lowest isolation level. In this level, one transaction may read not yet committed changes made by other transaction there by allowing dirty reads. In this level, transactions are not isolated from each other.

(b) Read committed :- This isolation level guarantees that data read is committed at the moment it is read. Thus it does not allows dirty read. The transaction holds a read & write lock on the current row, and thus prevent other transactions from reading, updating & deleting it.

c) Repeatable read :- This is the most restricted isolation level. The transaction holds read locks on all rows. It references and writes locks on all rows it inserts, updates & deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read.

d) Serializable :- This is the highest isolation level A serializable execution is guaranteed to be Serializable.

→ Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing.

→ The table is given below clearly depicts the relationship between isolation levels, read phenomena and locks.

Isolation level	Dirty ready	non-repeatable ready	phantom
Read Uncommitted	May Occur	may occur	may occ
Read Committed	Don't occur	may occur	may occur
Repeatable Read	Don't occur	Don't occur	may occur
Serializable	Don't occur	Don't occur	may occ

Concurrency control protocols :-

→ Concurrency control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and maybe even cascadeless.

→ These protocols do not examine the precedence graph as it is being created; instead a protocol impose a discipline that avoids non serializable schedules

→ Different concurrency control protocols provide different advantages between the amount of concurrency they allow and the amount of overhead that they impose.

→ Database may provide a mechanism that ensures that the schedules are either conflict or view serializable and recoverable. Testing for a schedule for serializability after it has executed is obviously too late, so we need concurrency protocols that ensure serializability.

Different categories of protocols :

→ Lock Based Protocol
→ Graph Based protocol
→ Time-stamp Ordering Protocol
→ Multiple granularity protocol
→ Multi-Version protocol & Validation based protocol

Lock Based Protocol :- Lock based protocol can be used Basic 2-PL, Conservative- 2-PL, Strict 2-PL and Rigorous 2-PL.

→ A lock is a variable associated with a data item that describes a status of data item with respect to possible operation that can be applied to it.

→ They synchronize the access by concurrent transactions to the database items. Two common locks which are used and some terminology followed in this protocol.

- → Shared lock
- → Exclusive lock

Lock compatibility Matrix :-

	S	X
S	✓	×
X	×	×

→ 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(x) 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. Then the lock is granted.

Upgrade/Downgrade Locks:- A transaction that holds a lock on an item A is allowed under certain condition to change the lock state from one state to another.

Upgrade:- A S(A) can be upgraded to X(A) if T_i is the only transaction holding the S-lock on element A

Downgrade:- we may downgrade X(A) to S(A) when we feel that we no longer want to write on data-item A. As we were holding X-lock on A, we need not check any conditions. So by now we are introduced with the types of locks and how to apply them. But, wait, just by applying locks if our problems could have been avoided then life would have been so simple.

Problem with simple locking:-

Consider the partial schedule

	T_1	T_2
1.	lock-X(B)	
2.	read(B)	
3.	B := B-50	
4.	write(B)	
5.		lock-S(A)
6.		read(A)
7.		lock-S(B)
8.	lock-X(A)	
9.	.....	.....

Deadlock :- Consider the above execution phase. Now T_1 holds an exclusive lock over B and T_2 holds a shared lock over A. Consider statement 7, T_2 requests for lock on B, while in statement 8 T_1 request lock on A. This as you may notice imposes a Deadlock as non can proceed with their execution.

Starvation:- It 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 or granted on S-lock on the same item. This may avoided if the concurrency control manager is properly designed.

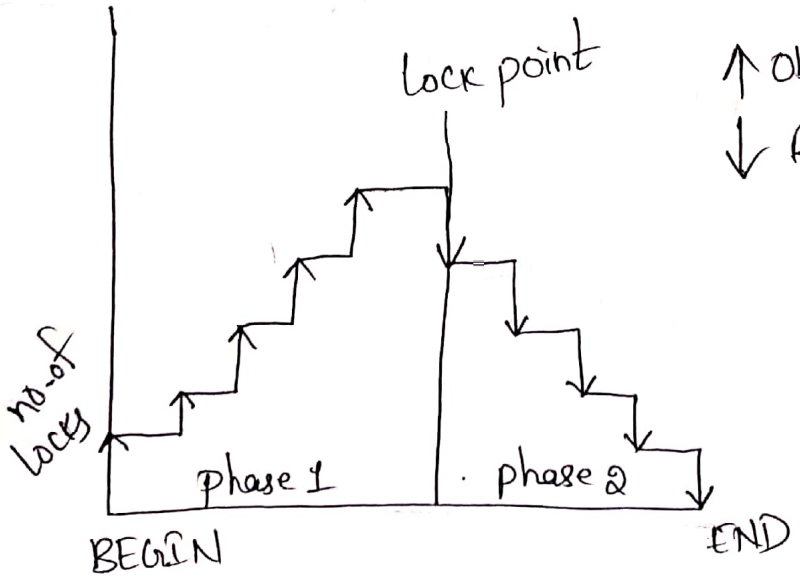
→ Locking works nicely to allow concurrent processing of transactions.

Two-phase Locking (2PL)

→ A transaction locks an object before using it.

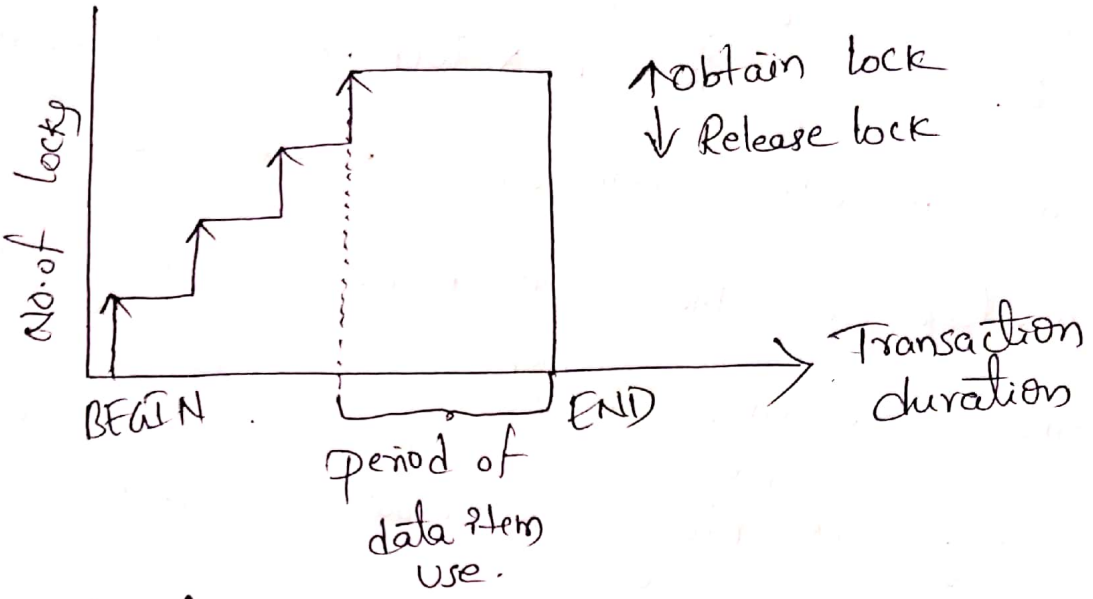
→ When an object is locked by another transaction, the requesting transaction must wait

→ When a transaction releases a lock, it may not request another lock.



Strict 2PL :-

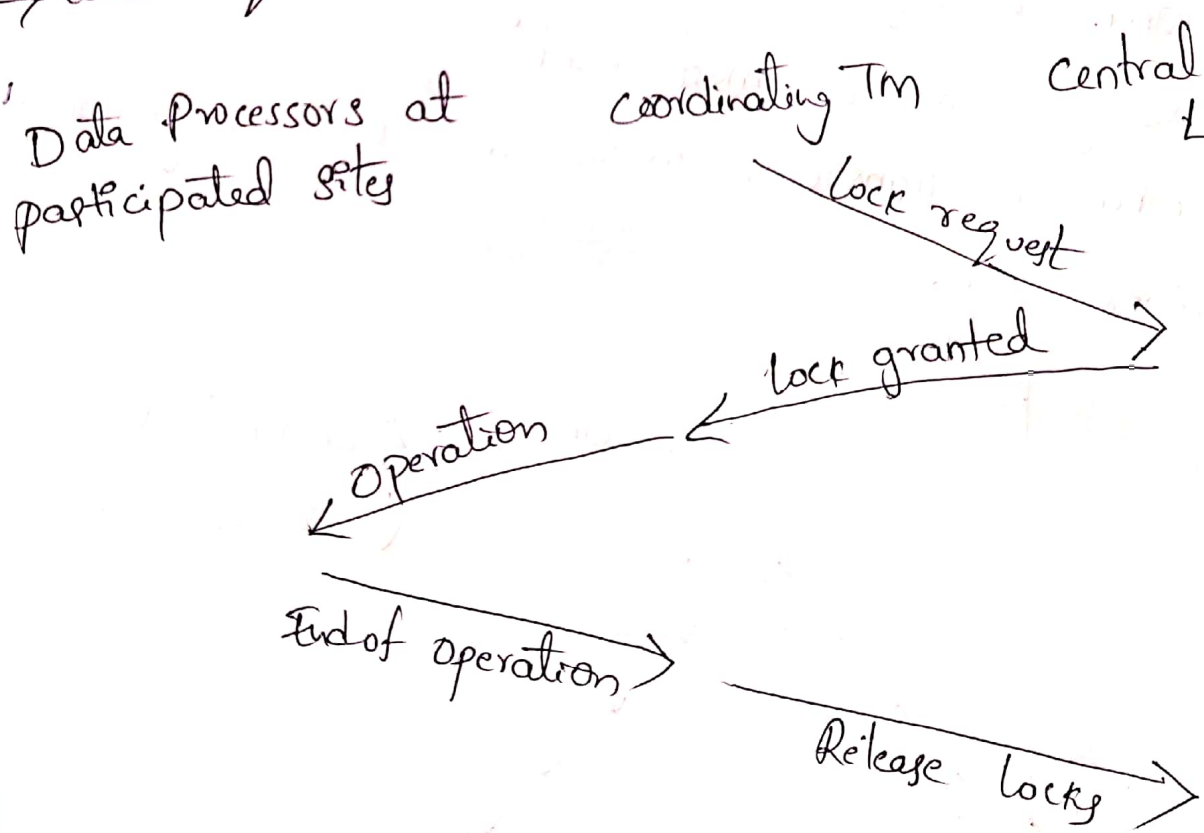
hold locks until the end



Centralized 2PL :-

→ There is only one 2PL scheduler in the distributed system.

→ lock requests are issued to the central scheduler



Time stamp based protocol :-

This is the most commonly used concurrency protocol. This protocol uses either System Time or Logical counter as a timestamp.

→ lock based protocols manage the order between the conflicting pairs among transactions at the time of execution, whereas timestamp-based protocols start working as soon as a transaction is created.

→ Every transaction has a time stamp associated with it, and the ordering is determined by the age of the transaction.

→ Every data item is given the latest read and write-timestamp. This lets the system know when the last read and write operation was performed on the data item.

→ Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

→ The time stamp ordering protocol ensures that any conflicting read and write operations are executed in time stamp order.

→ Suppose a transaction T_i issues a read(Q)

1.) If $TS(T_i) < W\text{-timestamp}(Q)$, 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

2) If $TS(T_i) \geq W\text{-timestamp}(Q)$, then the read operation is executed, and R-timestamp(Q) is set to max(R-timestamp(Q), TS(T_i))

Suppose the transaction T_i issues write(Q).

1. If $TS(T_i) < R\text{-Timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that the value would never be produced.

→ Hence, write operation is rejected. and T_i is rolled back.

2. If $TS(T_i) < W\text{-Timestamp}(Q)$, then T_i is attempting to write an obsolete value of Q.

→ Hence, the write operation is rejected, and T_i is rolled back.

3. Otherwise, the write operation is executed, and W-timestamp(Q) is set to (T_i).

→ If a transaction T_i is rolled back by the concurrency control schema as result of using distribution of either a read or write operation, the system assigns the transaction T_i a new timestamp and restarts it.

Advantages :-

→ Schedules are serializable (like 2PL protocols)

→ No waiting for transaction, thus, no deadlocks

Disadvantages :-

→ Starvation is possible if the same transaction is continually aborted and restarted.

Example use of the protocol :-

T_1	T_2	T_3	T_4	T_5
				read(x)
	read(y)			
read(y)				
		write(y)		
		write(z)		
				read(z)
	read(z)			
	abort			
read(x)				
			read(w)	
		write(w)		
		abort		
				write(y)
				write(z)

Validation Based Protocols :-

→ Execution of transaction T_i is done in three phases

1. Read and Execution phase :- Transaction T_i writes only to temporary local variables.

2. Validation phase :- Transaction T_i performs a "validation test" to determine if local variables can be written without violating serializability.

3. Write phase :- If T_i is validated, the updates are applied to the database, otherwise, T_i is rolled back

→ The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order

- Assume for simplicity that the validation and write phase occur together, atomically and serially.
- i.e. Only one transaction executes validation / write at a time.

→ Also called as Optimistic concurrency control since transaction fully in the hope that all will go well during validation.

→ Each transaction T_i has 3 timestamps

Start(T_i): the time when T_i started its execution

Validation(T_i): the time when T_i entered its validation phase.

Finish(T_i): the time when T_i finished its write phase

→ Serializability order is determined by timestamp given at validation time, to increase concurrency.

- thus TS(T_i) is given the value of validation (T_i)

→ This protocol is useful and gives greater degree of concurrency if probability of ~~conflicts~~ conflicts is low.

→ because the serializability order is not pre-decided, and relatively few transactions will have to be rolled back.

→ If for all T_i with TS(T_i) < TS(T_j) either one of the following condition holds:

→ finish(T_i) < start(T_j)

→ start(T_j) < finish(T_i) < validation(T_j) and the set of data items written by T_i does not intersect with the set of data items read by T_j

Then validation successful. and T_j can be committed.

Otherwise, validation fails. and T_j is aborted.

Schedule produced by validation :-

T_1	T_2
read(B)	
	read(B)
	B := B + 50
	read(A)
	A := A + 50
read(A)	
(validate)	
display(A+B)	
	(validate)
	write(B)
	write(A)

Multiple Granularity :-

→ Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.

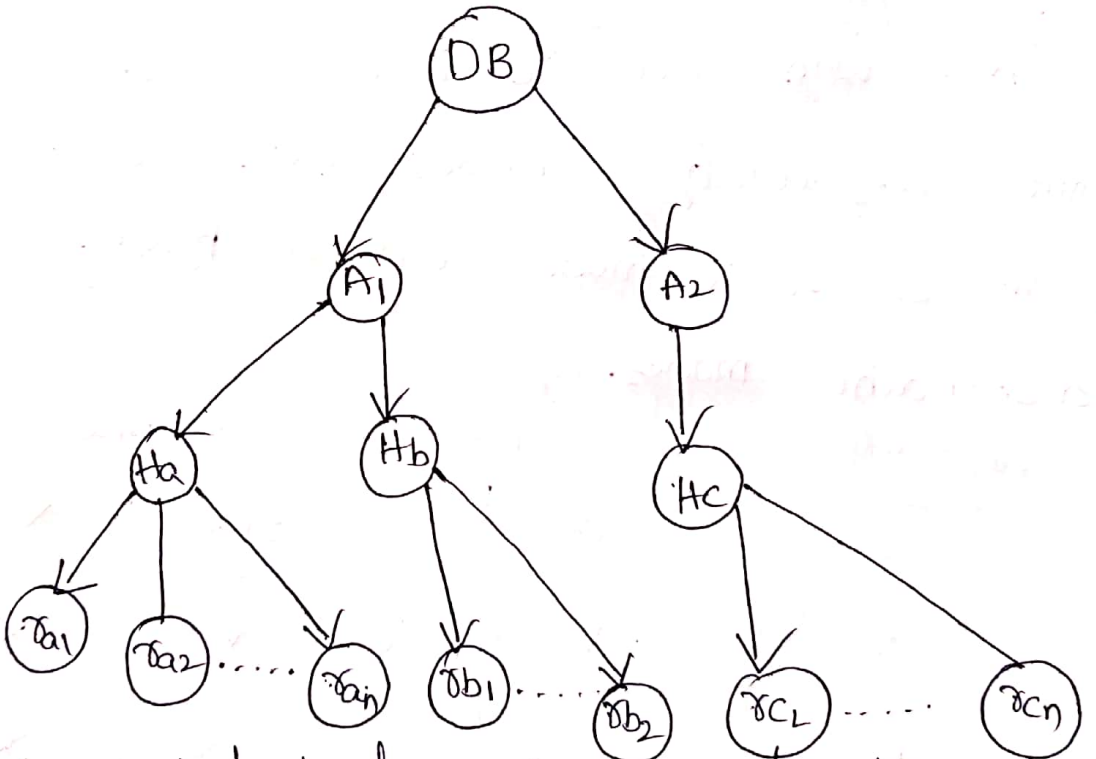
→ It can be represented graphically as a tree.

→ When a transaction locks a node in the tree explicitly, it implicitly locks all the nodes descendants in the same mode.

→ Granularity of locking (level in tree where locking is done):

fine granularity :- (lower in tree): high concurrency, high locking overhead.

coarse granularity :- (higher in tree): low locking overhead, low concurrency.



→ The highest level in the example hierarchy is the entire database. The levels below are of type area, file and record in that order.

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

intention-shared (IS):- indicates explicit locking at a lower level of the tree but only with shared locks.

intention-exclusive (IX):- indicates explicit locking at a lower level with exclusive or shared locks.

shared and 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.

→ intention locks allow a higher level node to be locked in S or X mode without having to check all descendent modes.

→ The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

Transaction T_i can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase)
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

Recovery and Atomicity :-

To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself

→ We steady two approaches :

- log-based recovery and
- shadow-paging.

→ We assume (initially) that transactions run serially; that is, one after the other.

Log-Based Recovery :-

→ The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.

→ If any operation is performed on the database then it will be recorded in the log.

→ But the process of storing the logs should be done before the actual transaction is applied in the database.

Let's assume there is a transaction to modify the city of a student. The following logs are written for this transaction.

- When the transaction is initiated, then it writes 'start' log.

  $<T_n, Start>$

- When the transaction modifies the city from 'Noida' to 'Banglore', then other log is written to the file.

  $<T_n, City, 'Noida', 'Banglore'>$

- When the transaction is finished, then it writes another log to indicate the end of the transaction.

  $<T_n, Commit>$

There are two approaches to modify the database.

1. Deferred database modification :-

→ The deferred modification technique occurs if the transaction does not modify the database until it has committed.

→ In this methods all logs are created and stored in the stable storage, and the database is updated when a transaction commits.

2. Immediate database modification :-

→ The immediate modification technique occurs if database modification occurs while the transaction is still active.

→ In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recovery using log records :- when a system is crashed then the system consult the log to find which transactions need to be undone and which need to be redone.

1. If the log contains the record $\langle T_i, start \rangle$ and $\langle T_i, commit \rangle$ or $\langle T_i, Commit \rangle$, then the transaction T_i needs to be redone.

2) If the log contains record $\langle T_n, start \rangle$ but does not contain the record either $\langle T_i, commit \rangle$ or $\langle T_i, abort \rangle$, then the transaction T_i needs to be undone.

Recovery with Concurrent transactions :-

Recovery with concurrent transactions can be done in the following four ways.

1. Interaction with concurrency control
2. Transaction rollback
3. Checkpoints
4. Restart recovery.

Interaction with concurrency control :-

On this scheme, the recovery scheme depends greatly on the concurrency control scheme that is used. So, to rollback a failed transaction we must undo the updates performed by the transaction.

Transaction rollback :-

→ In this scheme, we rollback a failed transaction by using the log.

→ The system scans the log backward a failed transaction, for every log record found in the log the system restores the data item.

Checkpoint :-

→ checkpoint is a process of saving a snapshot of the applications state so that it can restart from that point in case of failure.

→ checkpoint is a point of time at which a record is written onto the database from the buffers.

→ checkpoint shortens the recovery process.

→ when it reaches the checkpoint, the the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction. till the next checkpoint and so on.

→ The checkpoint is used to declare the point before which the DBMS was in the consistent state, and all the transactions were committed.

→ In this scheme, we used checkpoints to reduce the no. of log records that the system must scan when it recovers from a crash.

Restart Recovery :-

→ When the system recovers from a crash, it constructs two lists.

→ The undo-list consists of transactions to be undone, and the redolist consists of transaction to be redone.

→ The system constructs the two lists as follows: Initially, they are both empty. The system scans the log backward, examining each record, until it finds the first <checkpoint> record.

* Unit 4 Completed *