07/8/23

UNIT-IV

Transaction Concept

Syllabus

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 Transactions.

Transaction state

Transaction is a set of operations which are logically related.

The main operations in transactions are

① Read operation

· It reads the data from the database and then stores it in buffer from Main memory.

Read (A) - It will read the value of 'A' from the database.

② write operation

It writes the updated data value back-

to the database from buffer.

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

Transaction properties

DBMS must ensure to maintain data in case of concurrent access and system failures. They are ACID properties

Atomicity (All or nothing)

A transaction is said to be atomic if a transaction always executes all its actions in one step or does not execute at all.

Consistency (No violation of integrity constraints)

A transaction must preserve the consistency of db after the execution. It refers to the correctness of db.

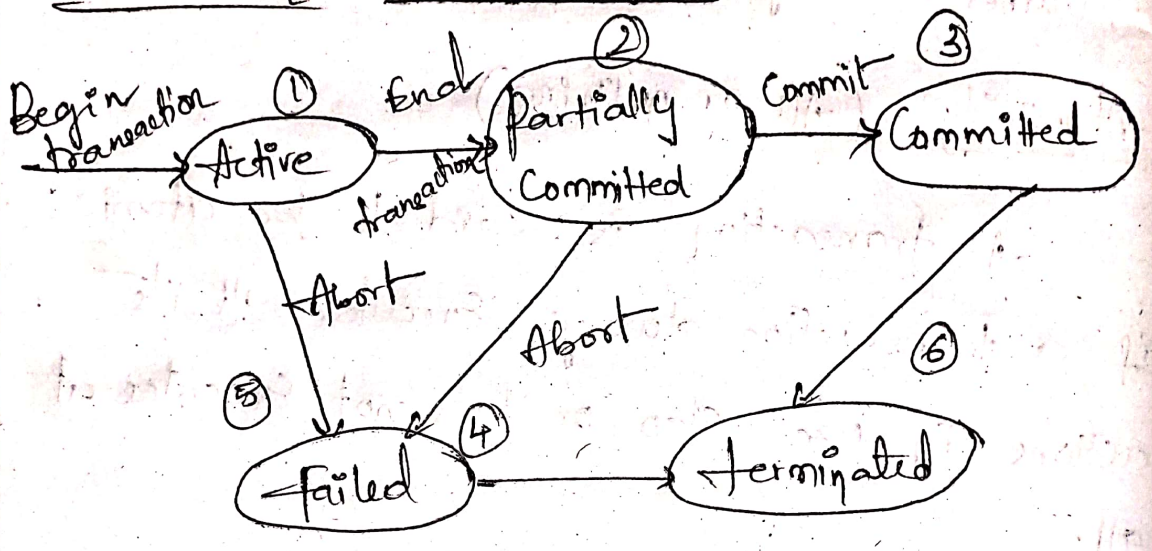
Isolation (Concurrent changes invisible)

This property ensures that multiple transactions can occur concurrently without leading to inconsistency of db. Transactions occur independently without interference.

Durability - (committed update persist)

The effect of completed or committed transactions should persist even after a crash. Once a transaction commits the system must guarantee that the result of its operations.

Transaction state Diagram



Active

A transaction enters into an active state when the execution process begins. During this state read or write operations can be performed.

Partially Committed

A transactions goes into partially committed state after the end of a transaction. Changes are stored in buffer but not updated in db.

Committed -

When the transaction is Committed to state, it has already completed its execution successfully. All of its changes are recorded to the db permanently.

Failed

A transaction considers failed when any one of the checks fails (or) if the transaction is aborted when it is in active state.

Terminated

State of transaction reaches terminated state when certain transactions are leaving the system can't be restarted.

Concurrent Execution (Multiprogramming)

Multiple transactions run concurrently. Multiple transactions causes several complications with consistency of data.

two good reasons for concurrency.

① Improved throughput and resource utilization.

Throughput is no. of transactions executed in a given amount of time.

A transaction consists of many steps. Some involve I/O activity and other involve CPU activity. The CPU and the disks in computer system can operate parallely.

The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel.

If one transaction is reading or writing data on disk another can be running in CPU. All of this increase throughput of the system (concurr.) correspondingly.

Reduced waiting time

If transaction run serially a short transaction may have to wait for a preceding long transaction to complete. If the transactions are operating on different parts of db it is better to run them concurrently.

Serializability -

When multiple transactions a running concurrently then there is a possibility that the db may be left in inconsistent state. Serializability helps us to checks that which schedule are serializable.

It means Executing transactions one after the other.

Schedule

A Sep of operations by a set of Concurent transactions that preserves the order of operations in each individual transaction.

Multiple transaction
- ↳ set of actions
  - ↳ which Schedule
    - ↳ which transaction
      - ↳ which time.

Serial schedule

A schedule where the operations of each transaction are executed consecutively without any interleared operations from other transaction

↓
No overlapping
No interleaving

Non- Serial Schedule

A schedule the operations from a set of Concurrent transactions are interleaved

Serializable Schedule

The objective of serializability is to find non-serial schedules that allow transactions to execute Concurrently without interfering one another and thereby produce a db that is produced by serial Execution.

If the result is same when it executes serial schedule and non-serial schedule then we can say that transaction is Serializable

Example

let T_1 and T_2 are two transactions.

T_1 transfers 50/- from Account 'A' to Account 'B'

```
T1: read(A);
    A:=A-50;
    write(A);
                read(B);
    B:=B+50;
    write(B);
```

T_2 transfer 10 percent of the balance from Acc 'A' to Acc 'B'.

T_2: read(A)

temp := A * 0.1

A := A - temp;

write(A);

read(B);

B := B + temp;

write(B);

Suppose two transactions are executed in the order T_1 followed by T_2.

-> It is divided into two types

Here all transactions should be arranged in a particular order, even if all the transaction is concurrent, and is not serializable, the it produce incorrect result.

① Conflict Serializability

It is one of the types of serializability which can be used to check whether a non-serial schedule is Conflict serializable or not.

Conflicting operations

Two operations are said to be in conflict, if they satisfy the below three conditions.

eg: the order table and the customer table

- One customer can have multiple orders, but each order only belongs to one customer.

⇒ Both operations should have different transactions T_1 & T_2

→ Both transactions should have the same data item (x)

- There should be atleast one write operation b/w the two operations. W(x)

eg: Swapping is possible only if S_1 and S_2 are logically equal.

① T_1 : Read (A) T_2 : Read (A)

T_1	T_2
Read(A)	
	Read(A)

→ swap

T_1	T_2
	Read(A)
Read(A)	

S_1 S_2

$S_1 = S_2$ — no-conflict

② T_1 : Read(A) T_2 - write(A)

T_1	T_2
Read(A)	
	write(A)

→

T_1	T_2
	write(A)
Read(A)	

S_1 S_2

$S_1 \neq S_2 \rightarrow$ Conflict

Conflict Equivalent

Two schedules are said to be conflict equivalent if and only if.

1. they contain the same set of transaction
2. If each pair of conflict operations are ordered in the same way.

Non-Serial Schedule

T_1	T_2
Read(A)	
write(A)	
	Read(A)
	Write(A)
Read(B)	
write(B)	
	Read(A)
	write(B)

Serial schedule

T_1	T_2
Read(A)	
write(A)	
Read(B)	
write(B)	
	Read(A)
	Write(A).
	Read(B)
	write(B)

After swapping of non-conflict operations, the schedule S_1 becomes.

T_1	T_2

→ write serial schedule Example here.

Check for conflict

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

Nodes T_1, T_2, T_3

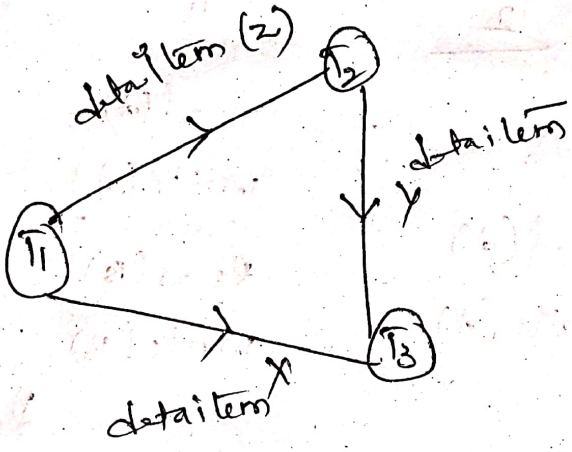
$T_1 \rightarrow R(x)$

$T_3 \rightarrow w(x)$

$T_2 \rightarrow R(y)$

$T_1 \rightarrow R(z)$

$T_2 \rightarrow w(z)$



Conflict Serializable

View Serializability

It is a process to find out the given schedule is Serializable or not.

View Equivalent

Two schedules T_1 & T_2 are said to be view equivalent if they satisfy the below conditions.

① Initial Read

Initial Read of each data item in transactions must match in both schedules.

T_1	T_2
Read(A)	
	Write(A)

S_1

T_1	T_2
	Write(A)
Read(A)	

S_2

The two schedules are equivalent because Initial read option in S_1 is done by T_1 and in S_2 it is also done by T_1.

② Updated Read

In schedule S_1 the transaction T_1 is reading data item updated by T_2.

T_1	T_2	T_3
Write(A)		
	Write(A)	
		Read(A)

S_1

T_1	T_2	T_3
	Write(A)	
Write(A)		
		Read(A)

S_2

→ Not view Equivalent, in S_1, T_3 is reading A update by T_2 and by S_2, T_3 is

reading A updated by T_1.

(3) Final write

A final write must be the same b/w both the schedules.

In S_1, if a transaction T_1 updates 'A' at least then in S_2, final write operations should also be done by T_1.

T_1	T_2	T_3	
write(A)			→ S_1
	Read(A)		
		write(A)	

T_1	T_2	T_3	
	Read(A)		→ S_2
write(A)			
		write(A)	

- Equal, final write operation in S_1 is done by T_3 and in S_2, the final write operation is also done by T_3.

eg:-

T_1	T_2	T_3	
Read(A)			→ S
	write(A)		
write(A)			
		write(A)	

→ with 3 transactions, the total no of possible schedule $3! = 6$

$S_1 - T_1\ T_2\ T_3$ $\quad$ $S_3 = T_2\ T_3\ T_1$

$S_2 - T_1\ T_3\ T_2$ $\quad$ $S_4 = T_2\ T_1\ T_3$

$S_5 - T_3\ T_1\ T_2$

$S_6 = T_3\ T_2\ T_1$

* Taking first schedule S_1:

T_1	T_2	T_3
Read (A)		
write (A)		
	write (A)	
		write (A)

Step 1: final updation

In S & S_1, there is no read except the initial Read.

Step 2: Initial Read

Condition satisfied

Step3: final write

Condition satisfied.

$T_1 \longrightarrow T_2 \longrightarrow T_3$

Recoverability of Schedule

Sometimes a transaction may not Execute Completely due to a software issue, system Crash or hardware failure.

In this Case, the failed transaction has to be roll back.

But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

① Irrecoverable schedule:

The schedule will be irrecoverable if T_j reads the updated value of T_i and T_j committed before T_i Commit.

Concurrency control protocols

① Lock-Based protocols

A lock guarantees Exclusive use of data items to a current transaction.

→ To access data items (lock acquire)

→ After completion of transaction (release lock)

lock Compatability Matrix

	S	X
S	✓	X
X	X	X

- A tran 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 ~~sha~~ no. of transactions can hold shared locks on a item, but if any tran holds an Exclusive(X) on the item no other tran may hold any lock on the item.

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 tran 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 holding 'x-lock' on A, we need not check any conditions.

Problem with Simple lock

T_1	T_2
lock-x(B)	
read(B)	
B := B-50	
write(B)	
	lock-s(A)
	read(A)
	lock-s(B)
lock-x(A)	

Deadlock - T_1 hold x(A) - over B,

T_2 holds s(B) - over A

T_2 requests for lock on B

T_1 requests for lock on A

It imposes deadlock.

Time-stamp based protocol-

It helps DBMS to identify the transactions

- It is a unique identifier. Each transaction is issued a timestamp

- They are usually assigned in order to which they are submitted to the system.

If refers as Transaction T as TS(T).

old transaction ← TS(1) < TS(2) → new transaction.

- It ensures that any conflicting read & write operations are executed in timestamp order.

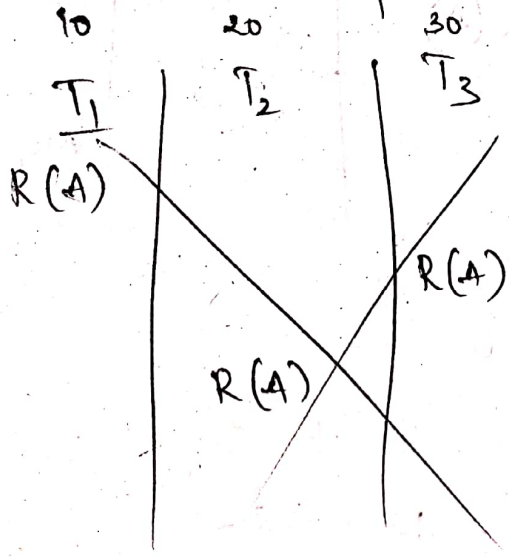
The Basic TO protocol in the following two cases. (Thomas' Write rules)

① T_1 issues Read(A) -

(i) If $Ts(T_i) < W(A)$ { T_i is an older transaction then the last tran that write the value of A } request fails

(ii) if $Ts(T_i) \geq W\_TS(A)$ { T_1 allowed to read updated value of A }

eg:



T_1 - 0 < 10

T_3 - 10 < 30

T_2 - 30 < 20 ✗

RTS(A) = 30 (largest one)

- This protocol uses system clock(or) logical clock to execute higher priority transaction first.
- As soon as transaction is created.

$T_1 \rightarrow 7:00$ $\quad TS(1) < TS(2)$

$T_2 \rightarrow 7:10$

Rules

① W - TS(x) is the largest timestamp of any tran that executes write(x) successfully.

② R - TS(x) is the largest timestamp of any tran that executes Read(x) successfully

Eg:

12	25	36	42
T_1	T_2	T_3	T_4
$R_1(A)$			
$W_1(A)$			
	$R_2(A)$		
		$R_3(A)$	
			$W_4(A)$

RTS = 7 RTS = 0

$R_1(A)$ - $T_1 = 12$ = $0 < 12$ ✓

$R_3(A)$ - $T_3 = 36$ = $12 < 36$ ✓

$R_2(A)$ - $T_2 = 25$ = $36 < 25$ ✗ ∴ stop the transaction

$RTs = 36$

WTS - write time stamp

WTS = 0

$w_1(A)$: $T_1 = 12$: $0 < 12$ ✓

$w_4(A)$: $T_4 = 12$: $12 < 42$ ✗ - stop the trans

$WTs = 42$.

② T_1 issues write (A)

(i) if $Ts(T_i) \geq w(A)$ & $Ts(T_i) \geq R(A)$ ✓

(ii) if $Ts(T_i) < R(A)$ ✗

(iii) $R(A) \leq Ts(T_i) \leq w(A)$ ✗

Validation Based protocol

It is an optimistic concurrency control techniques.

In this the transaction is executed in 3 phases

① Read phase -

The T is read & executed. Read the value of various data items and stores them in temporary local variable.

② validation phase - temporary var value will be validated against the actual data

to see its Serializability.

(3) Write phase

If the validation of trans is validated, the temporary results are written to db (or) sys.

Problem - roll back.

Start (T) - It contains the time when T starts

Validation (T) - It contains the time when T finishes its read phase & starts validation

Finish(T) - It contains the time when T finishes its write phase.

TS (Ti) = Validation (T)

↓

Serializability.

All trans's with earlier ts must have finish before transaction T started i.e finish(S) < start(T

S | R | V | W

T | R | V | W

Eg

T_1	T_2
Read (B)	
	Read (B)
	B := B - 50
	Read (A)
	A := A + 50
Read (A)	
<validate>	
display (A+B)	
	<Validate>
	write (A)
	write (B)

T_1 - validation log done before T_2

local variable

- No cascading roll back in Validation.

multiple Granularity:-

↓

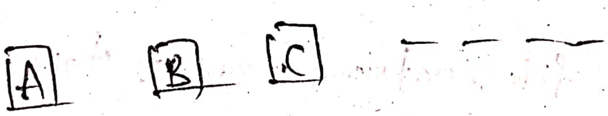
It is the size of data item allowed to lock.

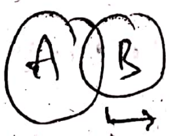
- It can be defined as heirarchically breaking up the db into blocks which can be locked.
- this protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item.

This type of hierarchy can be graphically represented as a tree.

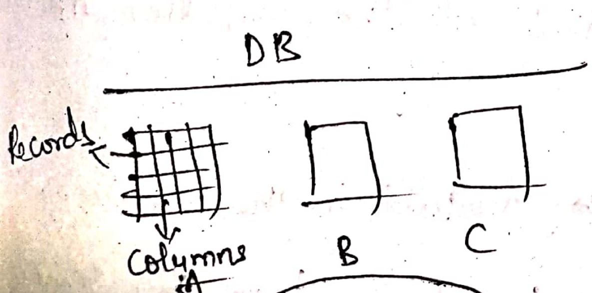
We are having to retreat each data base item as (each) individual data item.

Means there is no intersection b/w d.I.



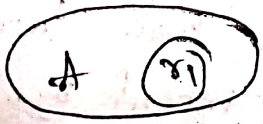


↳ But mutually Exclusive



→ tables multiple

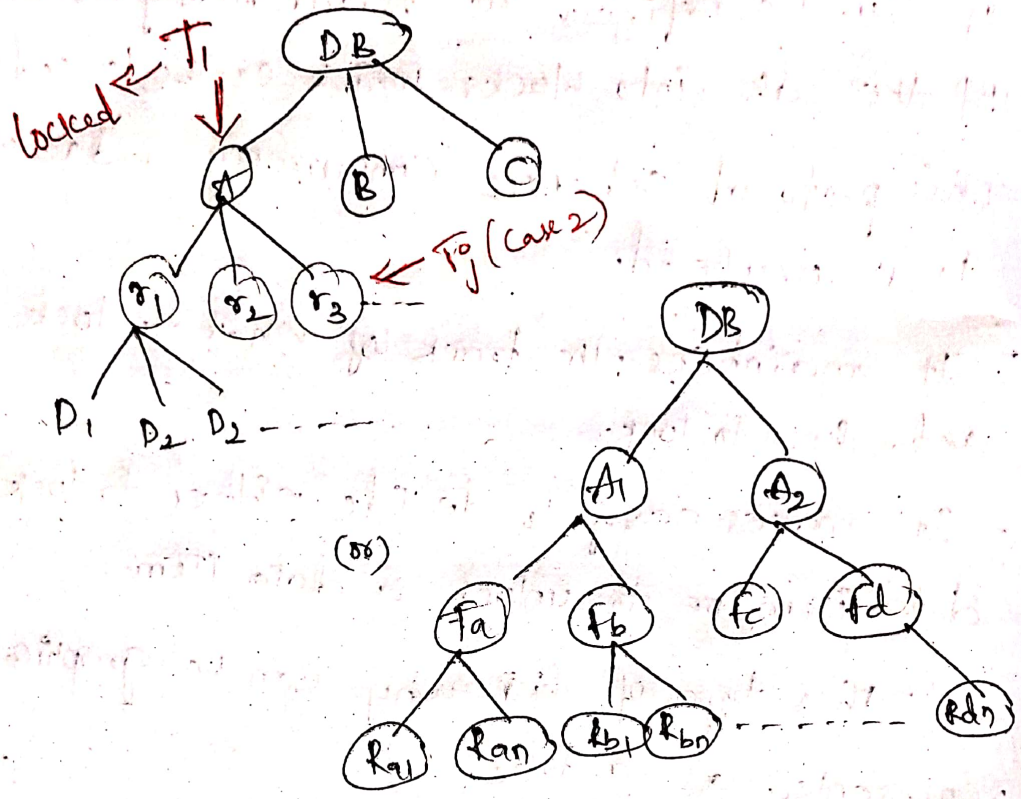
Different level of data, where as smaller levels are nested inside larger/higher levels.



- we have to use this granularity to ensure Concurrency. (To improve Efficiency)

(i) → Data items of multiple sizes

(ii) Represented in form of trees.



(1) Database - entire db

(2) Area - Represents a node of type area.

(3) file - children nodes. No file can be present in more than one area.

(4) Record

↳ Each file contains nodes known as records.

No records represent in more than one file.

(ii) when a node is locked all the children are automatically locked.

Problems faced by following these rules

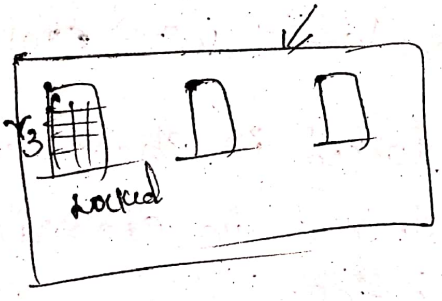
- In Emergent situation we have lock by some transaction on record r_1.

But T_i is already having lock. (T_i)
- T_i wants to acquire another lock on d_2.
No upper level in this d_2 is ~~record~~ locked.

Case-1 -

If we have to lock node d_2, we need to start from root to that node. And we need to check if any transaction will access the nodes or not from d_2.

Case 2 - we have to acquire a lock on DB. we need to check any of the decedent should not be locked by the nearer trans.



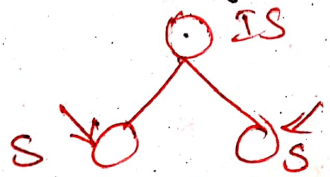
XL - Exclusive
SL - Shared.

Apart from these locks we are using 3 additional lock modes with multiple granularity.

Intention mode lock

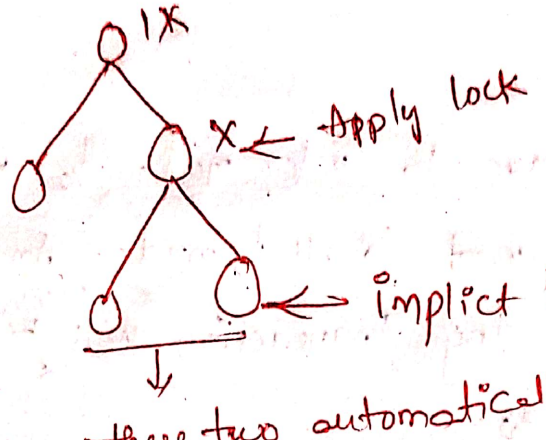
① Intention- Shared (IS) -

It contains explicit locking at a lower level of the tree but only with shared locks.



⑦ Intention Exclusive (IX)

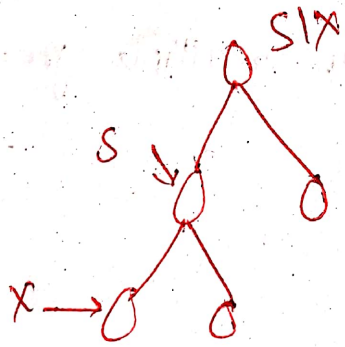
Explicit locking at lower level with exclusive or shared lock. (Atleast one exclusive lock (X))



⑧ Shared & Intension Exclusive (SIX)

Subtree rooted by that node is locked explicitly in shared mode & explicit locking is done at a lower level with exclusive mode.

The node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

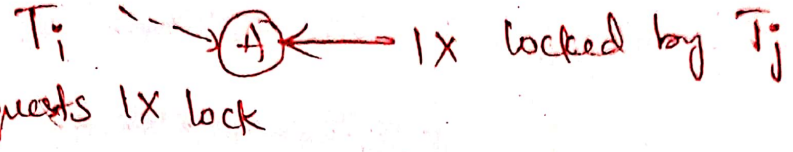


Compatability Matrix:-

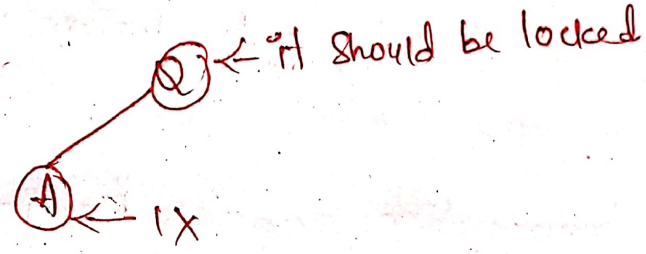
	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

If a transaction that attempts to lock a node must follow these rules-

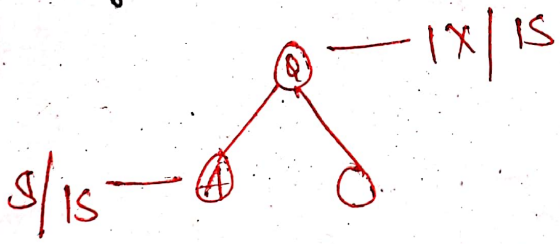
① must observe the lock compatibility matrix



② Root of the tree must be locked first and it can be locked in any mode



③ A node 'A' can be locked by T_i in S (or) IS mode only if the parent of A is currently locked by T_i in either IX or IS mode



④ If T_i has not previously unlocked any node only, then the T_i can lock a node.

⑤ If T_i currently has none of the children of the node-locked only, T_i will unlock a node.

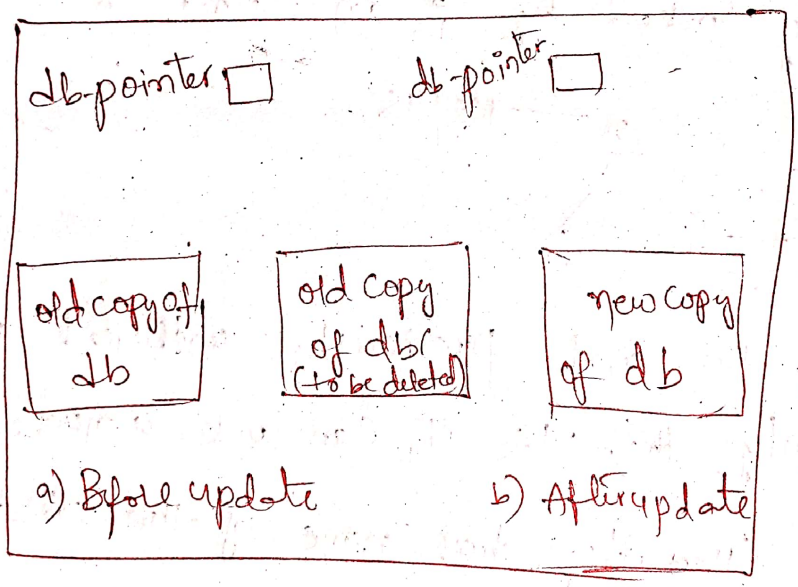
→ Implementation of Atomicity and Durability

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

eg: the shadow-database scheme

All updates are made on a shadow copy of the db.

- db pointer is made to point to the updated shadow copy after.
  - the transaction reaches partial commit and
  - All updated pages have been flushed to disk.



- It assumes that only one trans is active at a time.
- Assumes disks do not fail
- Does not handle current transactions.

Recover and Atomicity

The purpose of db recovery is to bring the db into last consistent state, which existed before the failure.

To recover from transaction failure, atomicity of transactions as a whole must be maintained - that is either all the operations are executed or none.

There are two types of techniques, which can help a DBMS, in. recovery and maintaining the atomicity of a transaction.

1. log Based Recovery
2. shadow paging

Log Based Recovery

The log is a sequence of records which maintains the records of all the update activities performed by a transaction.

It is maintained during the normal transaction processing. It contains enough information to recover from failure.

- log is the most commonly used structure for recording database.

Types of log records that are maintained in a log:

update log has following operations

① start record (Transaction identifier)

The log record $[T_i, start]$ is used to indicate that the transaction T_i has started.

② update log Record (Data item identifier)

The log record $[T_i, x, V_1, V_2]$ is used to indicate that the transaction T_i has performed an updated operation on the data item x, having old value V_1 & new value V_2.

③ Read Record - (old value (prior to write)

The log record $[T_i, x]$ is used to indicate that the transaction T_i has read the dataitem 'x' from the db.

(4) Commit record (After write)

The log record $[T_i, \text{commit}]$ is used to indicate that the transaction T_i has committed.

(5) Abord record

The log record $[T_i, \text{abort}]$ is used to indicate that the transaction T_i has aborted & needs to be rollback.

eg

	T_1	T_2	log record
A=1000	read(A)	read(C)	$[T_1, \text{start}]$
B=2000	A=A-50	C=C-100	$[T_1, A]$
C=700	write(A)	write(C)	$[T_1, A, 1000, 950]$
	read(B)		$[T_1, B]$
	B=B+50		$[T_1, B, 2000, 2050]$
	write(B)		$[T_1, \text{commit}]$
			$[T_2, \text{start}]$
			$[T_2, C]$
			$[T_2, C, 700, 600]$
			$[T_2, \text{commit}]$

② Write-ahead log strategy

- log is written before any update is made to db.
- Transaction is not allowed to modify physical db until UNDO is written to stable storage.

operations in Recovery procedure.

① UNDO - Restore old value

② REDO - updates by new value.

Techniques

① Deferred Db modification:-

↓

Put off to a later time, postpone

It ensures transaction (ensures) atomicity by recording all db modification to the log, but defering all write operations of a transaction, until the transaction partially commits (i.e. once the final action of the transaction has been executed).

- If the system crashes or if the trans aborts, then the info in the log is ignored.

eg:

A = 1000
B = 2000
C = 700

T_1	T_2
read(A)	read(C)
A = A - 50	C = C - 100
write(A)	write(C)
read(B)	
B = B + 50	
write(B)	

Case - I

$[T_1, start]$

$[T_1, A, 950]$ → new value

$[T_1, B, 2050]$

$[T_1, Commit]$

failure

db

A 950

B 2050

redo(T_1)

[both T_1, start & T_1, commit present in log]

recovery system

A 950

B 2050

Case - II

$[T_1, start]$

$[T_1, A, 950]$

$[T_1, B, 2050]$

failure

cache

A	B
~~1000~~ 950	~~2000~~ 2050

database

A	B
1000	2000

recovery system

no redate(T_1)

as no commit.

Case - III

$[T_1, start]$

$[T_1, A, 950]$

$[T_1, B, 2050]$

$[T_1, Commit]$ — redo(T_1)

$[T_2, start]$

$[T_2, C, 600]$

failure

Handling the failures, recovery scheme uses two procedures

① undo(T_i) - It restores the value of data items updated by trans T_i to the old value.

- T_i needs to undo if the log record contains T_i start but no T_i commit.

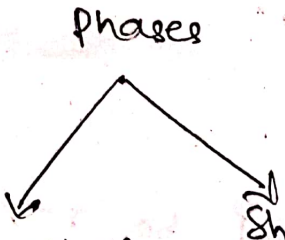
② Redo(T_i) -

It sets the value of all data items updated by trans T_i to the new value.

- T_i needs to be redo if log record contains both T_i start & T_i commit

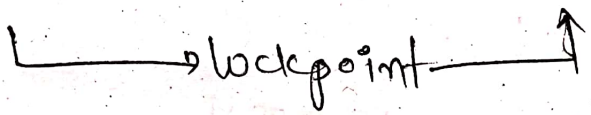
Two-phase locking protocol (2PL)

According to the rules of this protocol every trans can be divided into two phases. It requires both locks & unlocks being done.

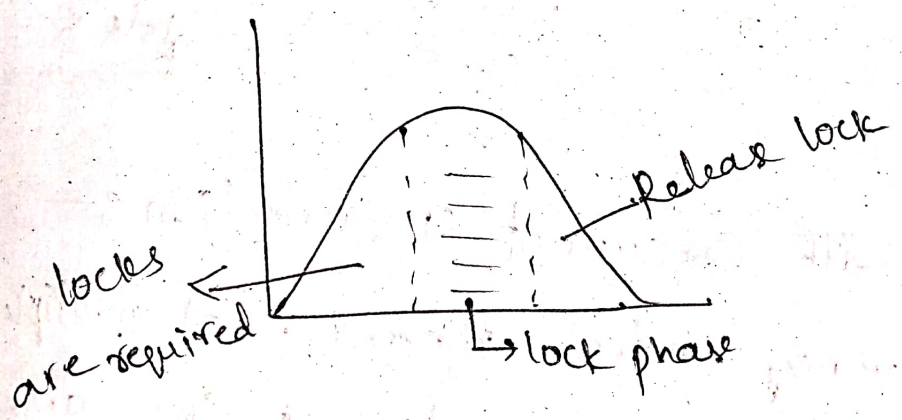


→ New locks can be acquired

→ existing locks can be released, No new lock can be acquired.



lockpoint - It is a point at which transaction has acquired final lock.



Variations of 2PL

- Acquires all lock before it starts
- Release all lock after commit
- Avoids cascading rollback

- No deadlock.

⇒ Strict 2pl -

- exclusive lock can't be released until commit.
- Helps in Cascadeless schedule
- Deadlock may occur.

⇒ Rigrous 2pl -

- Shared / Exclusive Can't be released till Commit.
- Avoid Cascading rollback
- Deadlock may occur.

Deadlock

(x, y)
lock x
(x+y)

Release lock on x

(x, y)
lock(y)
(y-x)

⇒ both the transaction are incomplete.

Recovery with Concurrent Transaction :-

Concurrency control means that multiple transactions can be executed at the same time and then the interleared logs occur. But there may be changes in trans results so maintain the order of execution

of those transactions.

During recovery, it would be very difficult for the recovery system to backtrack all the logs and then start recovering.

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

(1) Interaction with concurrency control

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

(2) 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.

(3) Checkpoints-

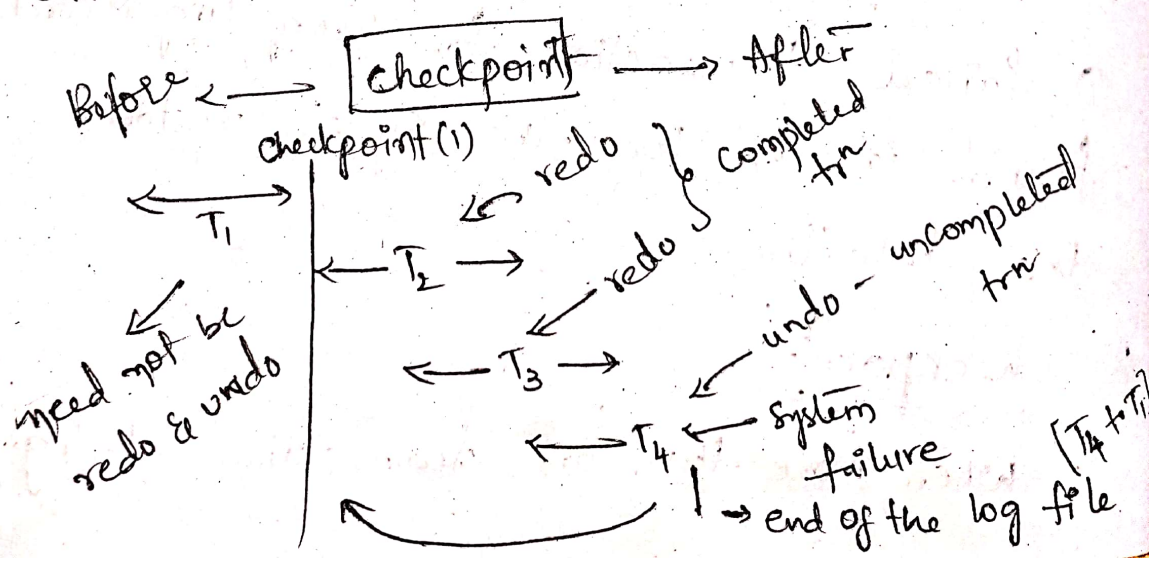
- When more than one transaction is being

executed then the logs may occur.
During recovery it would become difficult.
To make it easy we use 'checkpoint' concept.

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

- It is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- When it reaches the cp, then the trans will be updated into the db, 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 trans till the next cp & soon

It is used to declare a point before which the DBMS was in a consistent state, and all the trans were committed.



→ eg In log file T_2 & T_3 will have $<T_n, start>$ & $<T_n, commit>$ T_1 will have only $<T_n, commit>$ in log file. Hence it puts T_1, T_2, T_3 in redo list.

- The trns is put into undo state if the log is with $<T_n, start>$ but no commit or abort. In this case all the transactions are undone & logs are removed.

eg: T_4 will have $<T_n, start>$ ~~but no commit~~ It will be a failed transaction.

④ Restart Recovery.

- when the system recovers from a crash, it constructs two list.
- The undo list consists of trⁿ to be undone, the redo-list consists of trⁿ is be redone.
- The system constructs two lists. Initially they are both empty; The system scans the log backward, examining each record, until it finds the first <checkpoint> record.

ARIES -

Stands for Algorithm for recovery and Isolation exploiting Semantics.

It uses logs to record the progress of transactions and their actions which causes the change to data. It is one of the algorithm in transaction.

It is based on write ahead log protocol (WAL).

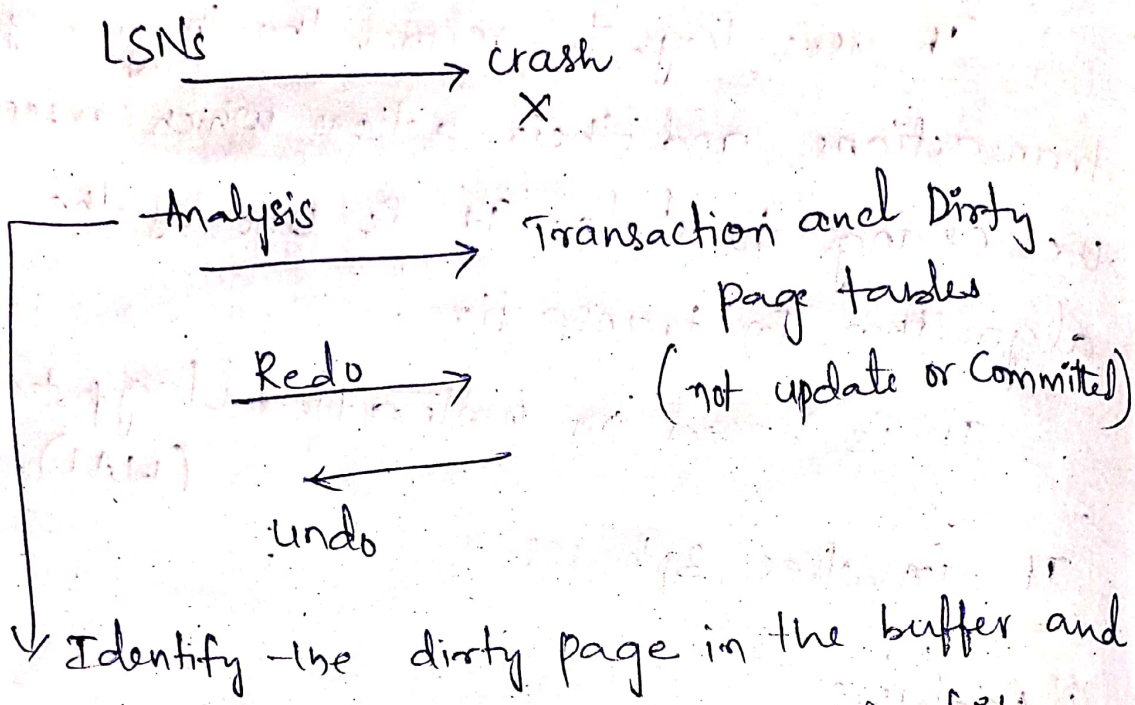
It involves 3 phases

1. Analysis
2. Redo
3. undo

Analysis - The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.

Redo - Starting at the earliest Log Sequence Number, the log is read forward and each update redone.

Undo - the log is scanned backward and updates corresponding to loser transactions are undone.



When System is restarted analysis is done and it identifies that T_1 & T_3 are active, hence restart it.

Redo - find all operations done by DBMS before crash and restore the system back to the same state before crash.

It aborts all active transactions those are still in active state at time of crash.

Undo - Scan the log backwards of active transactions in reverse.

T_4 write P_3 in undo

T_3 write P_1

T_1 write P_5