- Each transaction T_i can lock a data item at most once, and must observe the following Rules:

(1) The first Lock by T_i may be on any data item.

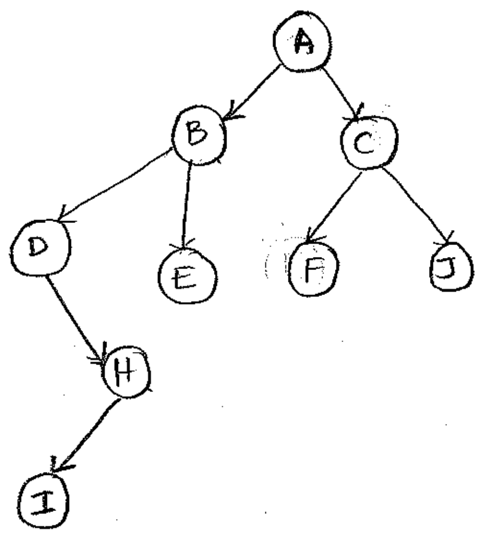
(2) Subsequently, a data item Q, can be locked by T_i only if the parent of Q is locked by T_i.

(3) Data items may be unlocked at any time

(4) A data item that has been locked and unlocked by T_i can not subsequently relocked by T_i.

→ Let $D = \{d_1, d_2, d_3, d_4, \ldots, d_n\}$ are set of dataitems,

If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j, must access d_i before accessing d_j. This implies that the set D can be viewed as Directed Acyclic graph called "Database graph (or) dependency graph."



"Tree-structured" dependency graph (or) database graph.

Advantage:-

(1) → Tree locking protocols have advantage over 2PL protocol in that, unlike 2-PL protocol, the tree protocols are "Deadlock free", so No rollbacks are required.

(2) The Unlocking may occur earlier, this leads to shorter waiting times, and to an increase in concurrency.

Disadvantage :-

In some case a transaction may have to lock data items that it does not access.

Ex:- A transaction that needs to access data items A and I in the dependency graph, it must not only Lock A and I, but also the data items B, D, H.

This additional locking resulting in increased locking overhead, this leads to additional waiting time, and potential decrease in concurrency.

∴ and Further, A is the root of the tree, if it is Locked, in transactions will lead to reduction of concurrency in great extent.

4

Time-Stamp Based Protocols:-

Time stamp ordering scheme is a method for determining the serializability order is to select an ordering in advance.

Timestamps:-

With each transaction T_i in the system, a unique fixed time stamp, denoted by $TS(T_i)$. This timestamp is assigned by database system before the transaction T_i starts execution.

∴ If a transaction T_i has been assigned timestamp $TS(T_i)$ and a new transaction T_j enters the system, then.

$$TS(T_i) < TS(T_j).$$

There are two simple methods for implementing this scheme

(1) Use the value of system clock as the timestamp;
(i.e) a transactions time stamp is equal to the value of the clock when the transaction enters the system.

(2) Use "Logical counter" that is incremented after a new time stamp has been assigned;
(i.e) a transaction time stamp is equal to the value of counter when the transaction enters the system

Ex:- When T_i starts counter = 1
T_j starts counter = 2
⋮
T_n starts counter = 6 etc

∴ whenever a new transaction starts, the counter value is incremented.

→ The time stamps of the transactions determine the serializability order.

(i.e) "$TS(T_i) < TS(T_j)$ → Indicates that T_i started before T_j

Let Q is a dataitem, with two time stamp values:

W-timestamp(Q):-

Denotes the largest time stamp of any transaction that executed write (Q) successfully.

R-timestamp(Q):-

Denotes the largest time stamp of any transaction that successfully executed "Read(Q)".

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

Time stamp ordering protocol:-

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

(1) Suppose, that a transaction T_i issues read(Q):-

(i) If $TS(T_i) < W\text{-}TS(Q)$:-

then T_i needs to read a value of Q that was already overwritten. Hence read operation rejected and T_i is rolledba[ck]

(ii) If $TS(T_i) \geq W\text{-}TS(Q)$:-

then the read operation is executed, and $R\text{-}TS(Q)$ is set to maximum of $R\text{-}TS(Q)$ and $TS(T_i)$

(2) Suppose that a transaction T_i issues write(Q):-

(a) $TS(T_i) < R\text{-}TS(Q)$:- (write is rejected & T_i is Rollback).

∴ The value of Q that T_i is producing was needed previously, and the system assumed that the value would never be produced. Hence the system rejects the write operation and T_i will be "Rollback"

∴ suppose T_j started after T_i, and T_j has written Q, then $TS(T_j) > TS(T_i)$, in that case

$\therefore$ $\boxed{R\text{-}TS(Q) = TS(T_j)}$, which is greater than $TS(T_i)$. Hence which is not conflict equivalent.

(b) $TS(T_i) < W\text{-}TS(Q)$:- Write is rejected and T_i Rollback.

T_i is attempting to write an obselete value of Q, the system rejects write and Rollback Q.

(c) Otherwise (i.e) $TS(T_i) > R\text{-}TS(Q)$ and $TS(T_i) > W\text{-}TS(Q)$:-

The system executes the write operation and sets W-TS(Q) to $TS(T_i)$.

(i.e) Some transaction $TS(T_a)$ trying to write on data item that $TS(T_a)$ is started before T_i.

So, the write is allowed and it is conflict equivalent.

Note: If T_i is rolledback by concurrency-control scheme by issuing of either read (or) write operation, the system assigns a new timestamp and restarts it.

→ Let us take two transactions T_{14} and T_{15}

T_{14}: Displays contents of A and B. and (A+B).

T_{15}: transfers 50 from Account B to A. and displays A+B.

T_{14}:	T_{15}:
read(B)	read(B)
read(A)	B=B-50
display(A+B)	write(B)
	read(A)
	A= A+50
	write(A)
	display(A+B).

schedule-S3 is under timestamp protocol, $TS(T_{14}) < TS(T_{15})$.

→ Transactions are assigned a timestamp immediately before its first instruction.

T_{14}	T_{15}
read(B)	
	read(B)
	B = B-50
	write(B)
read(A)	
	read(A)
display(A+B)	
	A = A+50
	write(A)
	display(A+B)

Schedule - S3.

→ Time stamp-ordering protocols ensures conflict serializability and ensures freedom from deadlock, since no transaction ever wait

Issues with Time stamp protocol :-

There is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.

Thomas' write Rule :-

It is a modification of time-stamp-ordering protocol that allows greater potential concurrency than TS-ordering protocol.

Let us consider - schedule S4 given below.

Schedule - S4

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

By applying rules of TS-protocols, since T_{16} starts before T_{17}. (i.e) $TS(T_{16}) < TS(T_{17})$.

→ In S4 - schedule, first read(Q) of T_{16} is done, and then write(Q) of T_{17} operation is performed.

→ When T_{16} attempts its write(Q), it finds that $TS(T_{16}) < W\text{-}TS(Q)$.

since $W\text{-}TS(Q) = TS(T_{17})$, (i.e) T_{17} has written on Q.

∴ Thus T_{16} is rejected and T_{16} must be rolledback, and it is unnecessary.

Since T_{17} has already written Q, that value that T_{16} is attempting to write, that will never need to be read.

→ Any transaction T_i with $TS(T_i) < TS(T_{17})$ that attempts read(Q) will be rolled back.

since $TS(T_i) < W\text{-}TS(Q)$.

→ Any transaction T_j with $TS(T_j) > TS(T_{17})$ must read the value of Q written by T_{17}, rather than the value written by T_{16}.

→ Hence, this leads to modify the TS-ordering protocols, in which obsolete write operations can be ignored.

Note: The protocol rules for Read operation remain Unchanged.

The Thomas write rule protocol:-

1. If $TS(T_i) < R\text{-}TS(Q)$, then the value of Q that T_i is producing was previously needed and it had been assumed that the value would never be produced. Hence, the system rejects write and T_i Roll backed.

2. $\boxed{\text{If } TS(T_i) < W\text{-}TS(Q)\text{, then } T_i \text{ is attempting to write an obsolete value of Q. Hence write operation can be IGNORED.}}$

3) Otherwise, $TS(T_i) > R\text{-}TS(Q)$ (or) $TS(T_i) > W\text{-}TS(Q)$, the system executes write operation and sets $W\text{-}TS(Q)$ to $TS(T_i)$.

→ The difference between TS-ording protocols and Thomas write rule is "only second Rule".

T_i is Rollback, if $TS(T_i) < W\text{-}TS(Q)$ in TS-ording protocols, But in Thomas write rule, those writes are ignored instead of Rollback.

Hence Thomas write rules avoids "unnecessary Rollbacks".

→ Thomas write rules makes use of view serializability, by deleting obsolete write operations from the transaction that issues them.

→ Hence, the result is a schedule, that is view equivalent to the serial schedule $<T_{16}, T_{17}>$.

Validation Based Protocols:-

→ These protocols. is an alternative scheme that imposes less overhead. These protocols. are used, in cases where a majority of transactions are readonly transactions, the conflicts among these transactions are may be low.

→ Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state.

→ Each transaction T_i executes in two (or) three different phases in its lifetime, depending on whether it is a "read-only" or an update transaction, (i.e) it involves any write operation.

→ The phases are:-

(1) Read-phase:-
During this phase, the system executes transaction T_i. It reads the values of various data items and stores them in variables (local variables) to T_i. It performs all write operations on temporary local variables, without updates on the actual database.

(2) Validation Phase:-
Transaction T_i performs validation test to determine whether it can copy to database, the temporary local variables that holds the results of write operations. without causing a violation of serializability.

(3) Write phase:-
If transaction T_i succeeds in validation, then the system applies the actual updates to database. Otherwise, the system roll backs the "T_i".

Each transaction must go through these phases in the order However, all three phases of concurrently executing transactions can be interleaved.

→ To perform the Validation Test, we need to know when the various phases of transaction T_i took place.

We associate different timestamps with transaction T_i.

(1) Start(T_i) :- the time when T_i started its execution.

(2) Validation(T_i) :- the time when T_i finished its read phase and started its validation phase.

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

→ We determine the serializability order by the timestamp-ordering technique, using the value of timestamp Validation(T_i).

Thus, $\boxed{TS(T_i) = \text{Validation}(T_i)}$ &

if, $TS(T_j) < TS(T_K)$, then any produced schedule must be equivalent to a serial schedule in which transaction T_j appears before T_K.

→ The reason for choosing Validation(T_i), rather than Start(T_i) as the time stamp of T_i is that we can expect faster responce time provided, that conflict rates among transactions are indeed Low.

..*→ The validation test for T_j requires that, for all transactions T_i with $TS(T_i) < TS(T_j)$,

One of the following conditions must hold:

(1) Finish(T_i) < Start(T_j) :-

since T_i completes its execution before T_j started, the serializability order is indeed maintained.

(2) The set of data items written by T_i does not intersect with the set of data items read by T_j and T_i complete its write phase before T_j starts its validation phase:

$$\text{start}(T_j) < \text{finish}(T_i) < \text{validation}(T_j).$$

$\therefore$ This condition ensures that the write of T_i and T_j do not overlap, since the writes of T_i do not affect the read of T_j and since T_j can not affect the read of T_i, the serializability order is indeed maintained.

→ <u>Schedule - S5</u> → with T14 and T15

T14	T15
read (B)	
	read (B)
	B = B - 50
	read (A)
	A = A + 50
read (A)	
<validate>	
display (A+B)	
	<validate>
	write (B)
	write (A).

<u>Schedule - S15</u>

Suppose that TS(T14) < TS(T15), then the validation phase succeeds in the schedule S5.

<u>Note:-</u> The writes to the actual variables are performed only after the validation phase of <u>T15</u>.

Thus, <u>T14</u> reads the old values of B and A, and this schedule is serializable.

→ The validation scheme guards against cascading rollbacks, since the actual writes takes place only after the transaction issuing the write has committed.

Issues with validation protocol :-

There is a possibility of starvation of long transaction, due to a sequence of conflicting short transactions that cause a repeated restarts of the long transactions.

→ To avoid starvation, conflicting transactions must be temporarily blocked, to enable a long transaction to finish

" This validation scheme is called Optimistic concurrency control scheme". since transactions executes "optimistically" assuming they will be able to finish execution and validate at the end.

→ In contrast, Locking and timestamp ordering are pessimistic in that they force a wait or rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

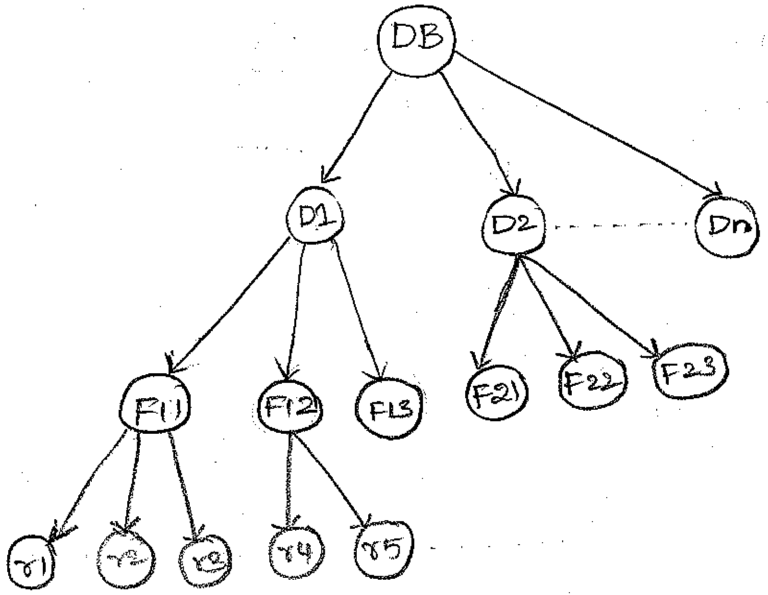
MULTIPLE GRANULARITY LOCKING:-

It is a mechanism to allow the system to define multiple levels of granularity.

The data items to be of various sizes and defining hierarchy of data granularities, where a small granularities are nested with in large granularities.

→ Such hierarchy can be represented as a tree,

Note: The multiple granularity tree is different from tree protocol.



Granularity Hierarchy

A non leaf node in multiple granularity tree represents the data associated with its descendants. In tree protocol, each node is a independent data item.

→ The core concept here is: We are Hierarchically Breaking up the database into portions which are lockable.

In the above Hierarchy:

DB represents a Database, which is highest Level,

D1, D2, ... Dn are Directories of Database.

F11, F12, F13, F21, ... are files.

r1, r2, r3, ... are records, which are in different Levels.

From the figure, the root node is "Database" and Leaf nodes are "records".

→ The multiple granularity locking is useful, where we can group several data items, and treat them as one unit.

Ex:- If Ti needs to access the access the entire database and a locking protocol is used, then Ti must lock each item in the database, clearly, executing these locks are time-consuming. It would be better if Ti could issue a single lock request to lock the entire database, which is possible with multiple granularity locking.

→ It is also possible that, in this mechanism, that if a transaction Tj needs to access only a few dataitems, it locks those dataitems, ratherthan entire database, this increases concurrency.

→ From the granularity Hierarchy figure:

- Each node can be locked individually, by using "shared" and "Exclusive lock" modes.

→ When a transaction locks a node in either shared (or) exclusive mode, the transaction also has implicitly locked all the descendants of that node in same lock mode (i.e) If you lock a directory, that means, "We locked all files and Records connected to the directory" in the mode that is locked on directory.

It doesnot need to lock all files and records explicitly.

Ex:- If we want a transaction Tj wishes to lock record r2, of file "F11", since Ti has locked "F11" explicitly, it follows that r2 is also locked implicitly. But when Tj issues a lock request for r2, r2 is not explitly Locked. This.

is a problem. Tj must traverse the tree from the root to record r2. If any node in that path is locked in an incompatible mode, then Tj must be delayed.

→ To resolve this problem multiple granularity locking mechanism provides new class of lock modes; they are called Intention lock modes.

The Intention Locks are: Intension shared, Intention Exclusive, Shared-Intention-Exclusive.

The total Locks in Multiple granularity are:

1) S: shared
2) X: Exclusive
3) IS: Intention shared
4) IX: Intention Exclusive
5) SIX: Shared and Intention Exclusive.

→ Intention Locks:-

If a node is locked in an intention mode, explicit locking is done at a lower level of the tree.

Intention Locks are put on all the ancestors of a node before that node is locked explicitly.

Thus, the transaction does not need to search the entire tree to determine whether it can lock a node successfully.

Intention shared Mode (IS):-

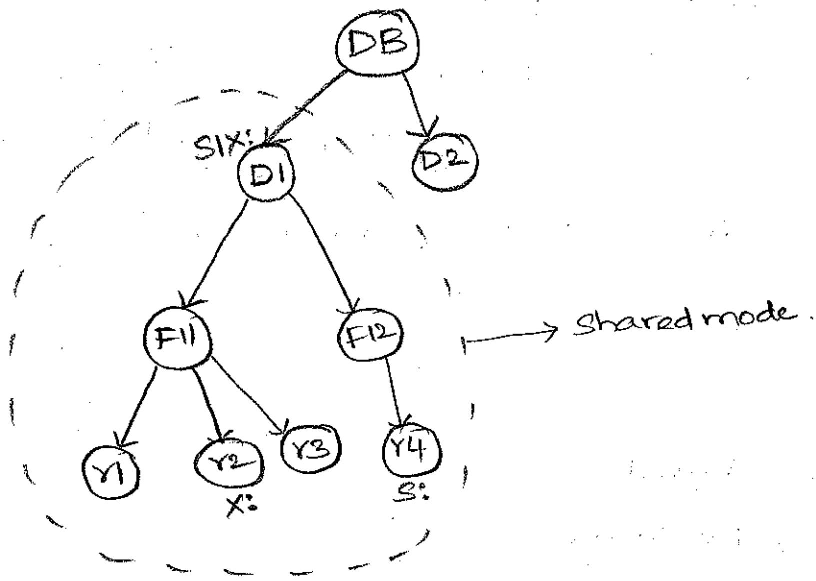
If a node is locked in IS mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks.

Intention Exclusive mode:- (IX).

If a node is locked in IX mode, then the explicit locking is being done at a lower level, with exclusive mode (or) shared mode locks.

Shared and Intention Exclusive mode (SIX) :-

If a node is locked in <u>SIX</u> mode, the subtree rooted by that node is locked explicitly in <u>shared-mode</u>, and that explicit locking is being done at a lower level with exclusive mode locks.



<u>D1</u> → SIX

<u>means</u> → D1 and its subtr
is in shared mo

Some <u>r2:</u> is in exclusiv
mode.

Lock-Compatibility Matrix (or) Compatibility Function :-

	IS	IX	S	SIX	X
IS	T	T	T	T	F
IX	T	T	F	F	F
S	T	F	T	F	F
SIX	T	F	F	F	F
X	F	F	F	F	F

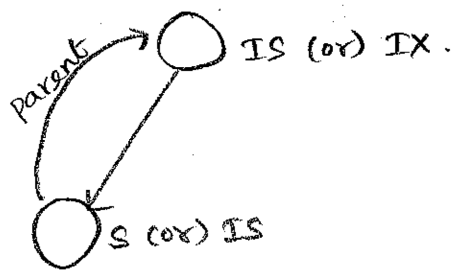
-: <u>Compatibility function</u> :-

The multiple granularity protocol locking which ensures serializability, Each transaction Ti can lock a node Q by following these rules:

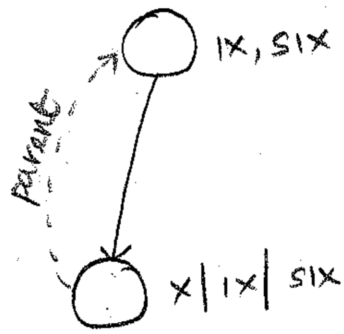
(1) It must follow Lock compatibility Function (Table).

(2) It must lock the root node of the tree first, and can lock it in any mode.

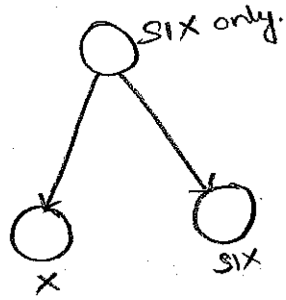
(3) It can lock a node Q in <u>S</u> or <u>IS</u> mode, only if it current has the parent of Q locked in either IX (or) IS mode.



(4) It can lock a node Q in X, SIX, IX mode only if it parent of Q locked in either IX (or) SIX mode



But



(<u>Note</u>:- For X and SIX, the parent is SIX only).

(5) It can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).

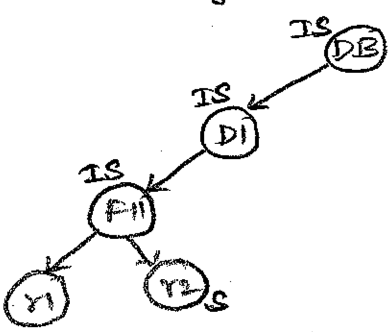
(6) T_i can unlock a node Q only if it is currently has none of the children of Q locked.

→ As we observe from the above rules:

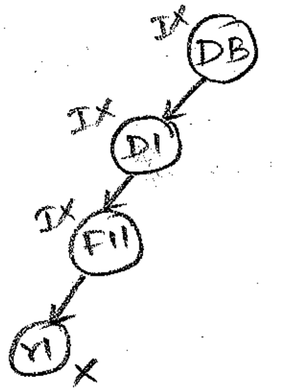
- The locks can be acquired in topdown approach (i.e) root-to-leaf order.
- The locks must be released in Bottom-up (i.e) leaf to root order.

<u>Let us look at some situations:-</u>

→ Suppose a transaction T_{18} <u>reads</u> a record <u>r2</u> in File <u>F11</u>, Then T_{18} needs to lock the database, directory, and File <u>F11</u>, in <u>IS</u> mode only, and finally lock. <u>r2</u> in <u>S</u> mode.



→ Suppose, that transaction T19 modifies a record r1 in file F11, then T19 needs to lock the database, directory D1 and file F11 in IX mode, and finally lock r1 in X mode.



→ Suppose, a transaction T20 reads all the records in F11, then T20 needs to lock database and D1 in IS mode, and finally lock F11 in S mode.

→ Suppose T21 reads the entire database, it can do so after locking the database in S mode.

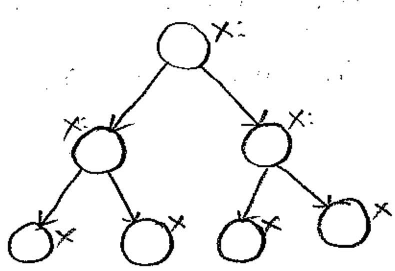
Note:- T18, T20, T21 can access the database concurrently. T19 can execute concurrently with T18, but not T20 & T21

→ This protocol enhances concurrency and reduces the lock overheads. It is particularly useful in applications that include a mix of:

(1) Short transactions that access only a few data items.

(2) Long transactions that produce reports from an entire file or set of files.

Note: Once we locked a node and its tree below exclusive mode nothing else is obviously allowed.



RECOVERY SYSTEM

Recovery From Failure :-

An integral part of a database system is a "Recovery Scheme" that can restore the database to the consistent state that existed before the failure.

- The recovery scheme must also provide High Availability, (ie) it must minimise the time for which database is not usable after a crash.

Failure Classification :-

There are various types of failures that may occur in a system.

- (1) Transaction failure :-

There are 2 types of errors that may cause a transaction to fail.

- Logical error :- The transaction can no longer continue with its normal execution because of some internal condition, such as bad input, data not found, overflow, (or) resource limit exceded.

- System error :-
The system has entered an undesirable state (For Ex:- Deadlock) as a result of which a transaction can not continue with its normal execution. The transaction, however, can be reexecuted at a later time.

(2) System Crash :-

There is a hardware malfunction, or a bug in the database software (or) the operating system, that causes the loss of the content of volatile storage and bring transaction processing to halt. The content of non-volatile storage remains intact and is not corrupted.

∴ The assumption that hardware errors and bugs in the software brings the system to halt, but do not corrupt the non-volatile storage content is known as "Fail-stop Assumption".

(3) Disk failure:-

A disk block loses its content as a result of either a disk head crash (or) failure during the data transfer operation.

To recover from this failure we copy the data on other disks, (or) archieval backups on tertiary media, such as tapes, are used to recover from the failure.

-o-

→ We must have to implement Recovery Algorithms to bring back the Lost data.

These algorithms have 2 parts.

(1) Actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.

(2) Actions taken after a failure to recover the database contents to a state that ensures database consistency, transaction atomicity, and durability.

(fe)
→ Recovery of data can be done in either prior to the failure, during the failure (or) after the failures.

Storage Structure:-

Storage types:-

(1) Volatile storage:-

Information residing in volatile storage does not usually survive the system crash.

Ex:- Main Memory and cache memory.

→ These volatile storages are very fast.

(2) Non-volatile storages:-

Information residing non-volatile storage survives system crashes.

Ex:- Secondary storage devices.— Magnetic disks, flash storage
Tertiary storage devices — optical media, Magnetic tape

(3) Stable storage:-

Information stored in stable storages never lost. Stable storages have high reliability.

Implementation of stable storages:-

To implement the stable storage, we need to replicate the needed information in several non-volatile storage media (usually disk) with independent failure modes, and to update the information in a controlled manner to ensure that failure during the data transfer does not damage the needed information.

Examples of stable storages are "RAIDS."

RAID systems guarantees that the failure of a single disk, even during data transfer will not result in loss of data.

The simplest and fastest form of RAID is the mirrored disk, which kept two copies of each block, on seperate disks.

→ RAID systems, however can not guard against data loss due to disasters such as fires (or) flooding.

→ Many systems store archieval backups of tapes off site to guard against such disasters.

→ Since, tapes can not be carried off site continually, updates since the most recent time that ~~happen~~ tapes were carried off site could be lost in such a disaster.

→ Most ~~secure~~ systems keep a copy of each block of stable storage at a remote site, writing it out over a computer network, in addition to storing the block on a local disk system.

∴ Since the blocks are output to a remote system and, hence, once an output operation complete, the output is not lost, eventhough, if there is an event of a disaster such as fire and flooding.

∴ DATA TRANSFER :-

The data is in the form of Block. The data transfer means it is block transfer only.

Block transfer between memory and disk storage can result in following situations.

(1) Successful completion :-
The transfered information arrived safely at its destination.

(2) Partial failure :-
A failure occurred in the midst of transfer and the destination block has incorrect information.

(3) Total Failure :-
The failure occurred sufficiently early during the transfer that the destination block remain intact.

4-22. If a data transfer failure occurs, the system detects it, and invokes a recovery procedure to restore the block to a consistent state.

- In order to do that, the system must maintain two physical blocks for each logical database block;

(i) In case of mirrored disks, both blocks are at the same location.

(ii) In case of "Remote Backup", one block is local and other is at a remote site.

An output operation executed as follows:

(1) Write the information onto the first physical block.

(2) When the first write complete successfully, write the same information onto second physical block.

(3) The output is completed only after the second write completes successfully.

The Recovery procedure consists of these 3 steps.

Step-1 :-

During the recovery the system examines each of physical blocks. If both are same and no detectable error exists, then no furthur action is necessary.

Step-2 :-

If the system detects error in one block, then it replaces its contents with the contents of other Block.

Step-3 :-

If both blocks contain no detectable error, but they differ in content, then the system replaces the content of first Block, with the value of the second.

* → This recovery procedure ensures that a write to stable

storage either succeeds completely (or) result in no change.

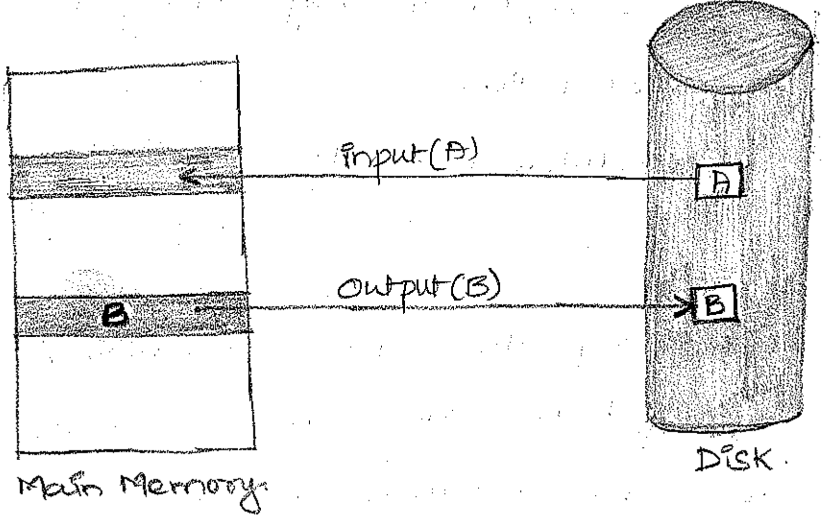
→ This requirement of comparing every corresponding pair of blocks during recovery is expensive.

→ By keeping track of block writes that are in progress, we can reduce this expensiveness. We can use small amount of non volatile storage RAM for this purpose.

DATA ACCESS:-

A Database system resides parmanently on nonvolatile storage, and is partioned into fixed-length storage units called "BLOCKS".

→ Blocks are the units of data transfer to and from disk, and may contain several data items.



⇒ A, B are data items (i.e) information

Block Storage operations

Block storage Operations:-

(1) Input(B) :- Transfers the physical Block (B) to main memo

(i.e) transactions input information from the disk to main memory

(2) Output(B) :- Transfers Buffer Block (B) to the disk, and replaces the appropriate physical Block in disk.

(i.e) Transactions output the information back onto disk.

The input and output operations are done in "Block units".

physical Block :-

The Blocks residing on the disk are referred to as "physical Blocks"

Buffer Blocks :-

The Block residing temporarily in main memory are referred to as "Buffer Blocks".

Disk Buffer :-

The area of memory where resides temporarily is called disk Buffer.

- Disk Buffer is a small amount of memory on Hard disk that resides for few time.

* → Let X is the data variable which resides in disk, on Block B.
we denote 'Bx' as the Block B, where X resides.

* → Let x_i be the local variable that is present in main memory.

When a transaction is initiated, some memory space has been created. for accessing data items, and performs updates on it.

After completion of updations, transaction is either committed (or) Aborted, after that the memory space will be removed.

The transaction (Ti) interacts with the database system by transfering the data to and from its workspace to the system Buffer.

We transfer the data by using two operations.

(1) Read(X)

(2) Write(X)

Read(X) :-

read (x) (or) read(X, x_i) Assign the value of data item X to the local variable x_i.

It executes this operation as follows:

(1) If Block B_x is not in main memory, it issues input(B_x).

(2) It assigns to x_i the value of X from the buffer Block (i.e) $x_i \leftarrow X$.

Write(X) :-

Write (x) or Write (X, x_i) assigns the value of local variable x_i to data item X in the buffer block, ($X \leftarrow x_i$).

It executes this operation as follows:

(1) If Block B_x, is not in main memory, it issues input (B_x).

(2) It assigns the value of x_i to X in Buffer B_x.

(i.e) $X \leftarrow x_i$.

Read(x) :- $x_i \leftarrow X$
write(x) :- $X \leftarrow x_i$

Note: Both read(X) and write(x) operations may require input operation. But, output operation may require specifical situations only. The output operation required only, when there are set of updates on data item; when these updates are written back to disk, then only output operation required, Otherwise there is no need for output operation.

Note → *** 1) When a transaction needs to access a data item X, for the first time, it must execute "read (X)".

*2) When all updates are done on X, after the transaction access X for the final time, it must execute "write(x)".

*. 3) Write operation is performed to reflect changes to X in database.

→ As we previously mentioned, when a failure or crash occurs, on a transaction, the recovery system performs two operations either restart (i.e. reexecute) the transaction (or) kill the transaction.

But this simple recovery procedure can also lead to a state of inconsistency. So, there are few other techniques to. recover from failures (or) crashes.

Recovery Methods:—

(1) LOG Based Recovery

(2). Recovery with Concurrent Transaction.

(3) Buffer Management.

(4) Non-volatile storage Recovery.

*** (5) Advanced Recovery Techniques — ARIES.

(1) Log-Based Recovery:-

→ "Log" is the most widely used structure for recording "database Modifications."

→ A Log record (or) sequence of log records, records all the update activities in the database.

→ An update log record describes a single database write, it has these fields:

Transaction Identifier:-

It is the unique identifier of the transaction that performed the write operation.

Data Item Identifier:-

It is the unique Id of the data item written.

It is the disk location of data item.

Old value:-

It is the value of the data item prior to the write.

New value:-

It is the value of the data item after the write operation performed.

→ Consider T_i is the transaction,

X_j is the dataitem.

V_1, V_2 are values, V_1 is before write,

V_2 is after write.

Let us look at the given statements:

<T_i start> ⇒ Transaction T_i started.

<T_i commit> ⇒ Transaction T_i has committed.

<T_i abort> ⇒ Transaction T_i has aborted.

$< T_i, X_j, V_1, V_2 >$

Transaction has performed a write on data item X_j

X_j had a value V_1 before write, and will have value V_2 after the write.

*→ The "log record" must reside in stable storage, to recover from disk failures (or) system failures.

→ Whenever a transaction performs a write, it is essential that the log record for that write be created before the database modified.

*→ Once a log record exists, we can output the modification to the database if it is desirable.

(i.e) the updated value is different from original value, we will output the modification.

UNDO :-

It revokes the modifications, that has already been output to the database.

The undo operation performed by using "old value" field in log record.

Note :- ***. A Log can keep track of set of transactions, (i.e) a log may contain complete record of all database activity. As a result, the volume of data stored in the log may become very large. Hence, we use the concept of "CHECKPOINTS" in dealing with the large log files, to maintain consistency

Log-Based Recovery Techniques :-

1) Differred Database Modifications.
2) Immediate Database Modification.

(3) Checkpointing.

(1) Deferred Database Modification :-

→ This technique ensures transaction atomicity by recording all database modifications in the log, but deffering the execution of all write operations on a transaction untill the transaction "partially commits".

→ Deffering means "to delay (or) Postpone".

(i.e) that means we will postpone the write operations at the last.

→ Because, every time a write operation is performed the updated values must be reflected on (or) send to database. If there are several write operation, this process changing database again and again can make burdensome work on system. In between transaction execution

→ Inorder to avoid this burden, we deferring (or) postponing all write operations to update the value on the database

→ Suppose, in this mean time, a crash occurs we just have to simply execute the transaction again to recover from the crash.

(i.e). it performs Redo operation to Recover from the crash.

→ Hence, all log records are written out on stable storage, before the starting of any updates.

∴ So, already these log files are in stable storage, if any failure occures, we ensure that the transaction is recoved.

Once they have written, the actual updating takes place, and the transaction enters the committed state.

→ **Example :-**

The deferred technique can only takes new values, it does not remember the old value.

Hence, in the log record it contains <Ti>, and <memory variable, and <the new value>.

(r.g) Ex:- <Ti, A, 950>.

Let's take two transactions To and T_1

To → Transfer money from A to B.

T_1 → Withdraw 100 from account c.

Initially A = 1000, B = 2000, c = 700.

Transactions		To & T_1 Database Log	Database
To :	Read (A);	<To start>	
	A:= A - 50;	<To, A, 950>	
	Write (A);	<To, B, 2050>	
	Read (B);	<To commit>	// A = 950 B = 2050
	B:= B + 50;		
	Write (B);	<T_1 start>	
T_1 :	Read (c);	<T_1, c, 600>	
	c = c - 100;	<T_1 commit>	// c = 600.
	Write (c);		

Note:- The value of A is changed in the database only after the record <To, A, 950> has been placed in the log.

→ Using the log, the system can handle any failure that result in loss of information on volatile storage. The recovery scheme uses following recovery procedure:

*. <u>Redo(T_i)</u> → sets the value of all data items updated by transaction T_i to the new values.

The set of dataitems updated by T_i and their respective new values can be found in the log.

*→ <u>Redo</u> operation is idempotent, (i.e) executing it several times is equivalent to executing it once.

→ After a failure, the recovery system consults the log to determine which transaction need to be <u>Redone</u>.

*→ If the transaction contains both <T_i start> and <T_i commit> in the log record, then only <u>Redo</u> operation will be performed.

<u>Various situations of the log:-</u>

(1) <T_0 start>
<T_0, A, 950>
<T_0, B, 2050>

The log file contains start of transaction but there is no commit, so, the '<u>Redo</u>' operation wont performed.

• Hence, there is no commit, the data is not written onto the disk.

So, we will restart the transaction.

∴ So, the values of A & B are : <u>1000</u> and <u>2000</u> respectively, in Transaction T_0.

(2) <T_0, start>
<T_0, A, 950>
<T_0, B, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 600>

The system crashes "<u>write(C)</u>" operation is executing in T_1.

T_0 has <u>start</u> and commit is there, (i.e) T_0 is fully executed. We write 950 and 2050 onto disk.

T_1 has only <start> but not <commit>, so, the value of '<u>C</u>' remains 700 only.

The log record of T_1 incomplete, and T_1 can be deleted from the log.

< T_0 start>
< T_0, A, 950>
< T_0, B, 2050>
< T_0 commit>
< T_1 start>
< T_1, C, 600>
< T_1 commit>

Here, the transactions T_0 & T_1 both has <start> and <commit>. and this stage, the system crashes and, when the system comes backup, two commit records are in the log: one for T_0 and one for T_1.

∴ The system must perform operations Redo(T_0), Redo(T_1); in the order in which their commit records appear in the log.

After the system executes these operations, the values of accounts A = 950, B = 2050, C = 600, respectively.

→ If any crash occurs during the recovery, then the steps we performed in the first recovery will be performed again,

→ ~~So~~, (i.e) Redo(T_1) will be performed again.

The result of successful second attempt at <u>redo</u> operation is same as though <u>redo</u> had successful at the first time.

(2) <u>Immediate Database Modification Technique</u> :-

<u>overview</u>:-

(1) This allows Database Modifications to be output to the database while the transaction is still in the "<u>ACTIVE state</u>"

- Data Modifications written by active transactions are called "<u>uncommitted Modifications</u>".

(2) It uses <u>Old value</u> and <u>new value</u> in the log record.

(3) It uses operation such as: UNDO(Ti) and Redo(Ti) in.

Recovery scheme of transaction.

<u>Log Record</u>:- <Ti, Data item, oldvalue, newvalue>

→ The Recovery scheme operations:

<u>UNDO(Ti)</u>:-

It restores the value of all data items updated by transaction Ti to the old value.

<u>REDO(Ti)</u>:-

It sets the value of all data items updated by transac-tion <u>Ti</u> to the new values.

The set of data items. updated by Ti and their correspon-ding old and new values are found in the log.

→ After a failure has occurred, the recovery scheme consults. the log to determine which transactions need to be <u>redo</u> and <u>undo</u>.

- Transaction Ti needs. to be <u>undone</u> if the log contain the record <Ti start>, but does not contain <Ti commit>
- Transaction Ti needs to be redone if the log contain both the record <Ti start> and <Ti commit> record.

→ Let us take the previous banking example To & T1, with the the dataitems A=1000, B=2000, C=700 as values.

<u><To start></u> written on Log, before the transaction To. starts execution.

<u><To commit></u> → when Ti partially commits, the system writes the record <To commit> to the Log

LOG	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950 B = 2050
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	C = 600
<T_1 commit>	

State of System Log and Database of T_0 and T_1.

Various situations that above log file can face at the time of crash:-

(1) <T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>

<T_0 start> is there, and <T_0 commit> is not there in the log. "T_0" must be undone. So, undo (T_0) is performed.

The crash occurs in above log, just after write(B).
The above log file describes that the crash occurs just after the T_0 values are written on the disk, but it is not committed, so, we have to undo the updates performed by T_0. (values of A = 1000, B = 2000 only).

(2) <T_0 start>
<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>
<T_0 commit>
<T_1 start>
<T_1, C, 700, 600>

- T_0 has both records <T_0 start> & <T_0 commit>, so, after the crash, the T_0 transaction performs redo operation.
- T_1 has only one record <T_1 start>, but no <T_1 commit>.

Hence, T_1 performs undo (T_1) in recovery process.

The values of A = 950, B = 2000, C = 700.

Case (3):- <T_0 Start>

<T_0, A, 1000, 950>

<T_0, B, 2000, 2050>

<T_0 Commit>

<T_1 Start>

<T_1, C, 700, 600>

<T_1 commit>

In the log record, it has <T_0 start> and <T_0 commit>. And <T_1 start> & <T_1 commit>, records.

So, when crash recovery, the system performs the recovery procedures redo (T_0) and redo (T_1).

$\boxed{\text{Final values: } A=950,\ B=2050,\ C=600}$

Note:- In "case-2", there is one undo operation and redo operation.

(i.e) undo (T_1) and redo (T_0).

(*) It is important and necessary that before we perform redo (T_0), we have to perform undo (T_1) to recover properly.

(3). CHECKPOINTS :-

Checkpoints reduces, overhead of searching the log record and unnecessary redo operations in the process of recovery to decrease the time consumption.

(i.e). after the crash (or) failure, we must consult the log to determine transactions needed to be redone, and undone.

→ We need to search the entire log for this information. The search process is time-consuming.

→ We are redoing the transactions which are already, committed, so, this is a much time-taking process for redoing all committed transactions, this will take much longer time to recovery.