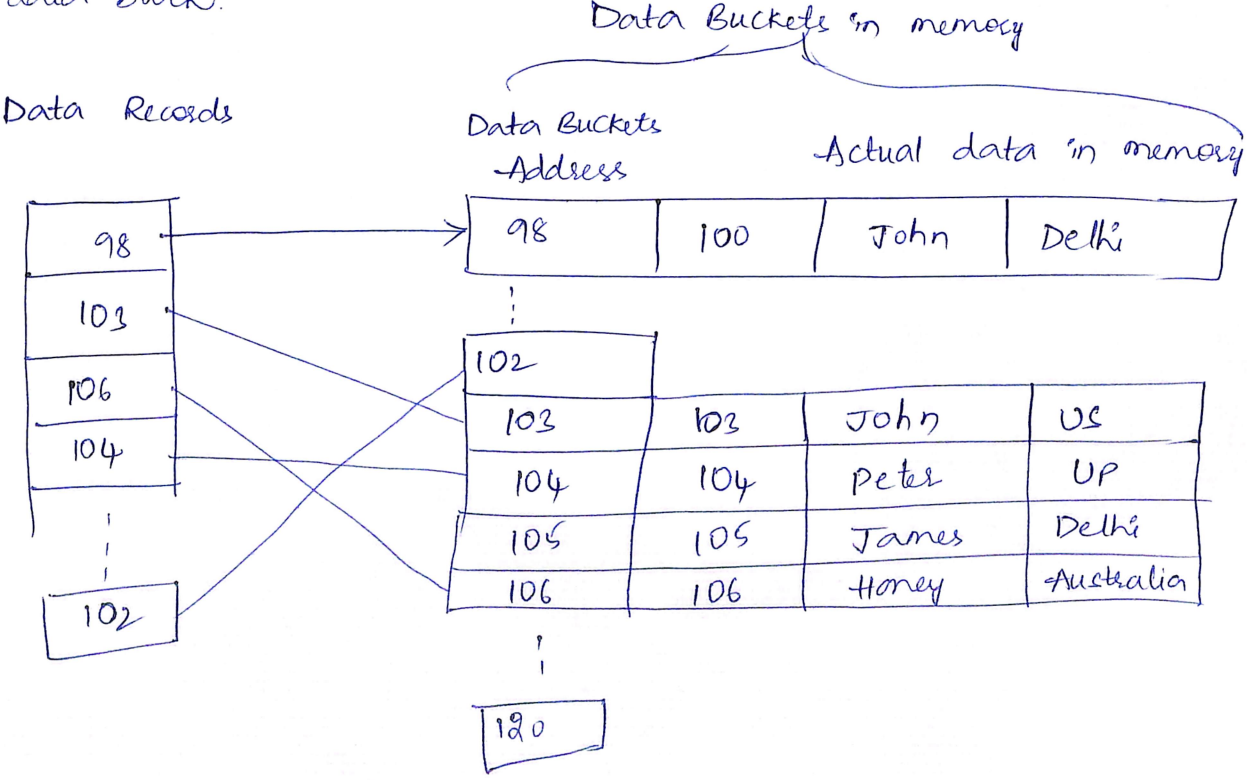
UNIT-IV

Hash based Indexing

* ~~Static~~

* **Hashing:**

- In a huge DB structure, it is not so efficient to search all the index values & reach the desired data. Hashing technique is used to calculate the direct location of data record on the disk without using index structure.
- Data is stored at the data blocks whose address is generated by using the hashing function. The memory location where these records are stored is known as data blocks.
- A hash function can chose any of the column value to generate the address.
- Hash function uses a primary key to generate the address of data block.
- " " is a simple mathematical fun to any complex mathematical fun.
- we can consider PK itself as the address of the data block.
- That means each row whose address will be the same as a PK stored in the data block.

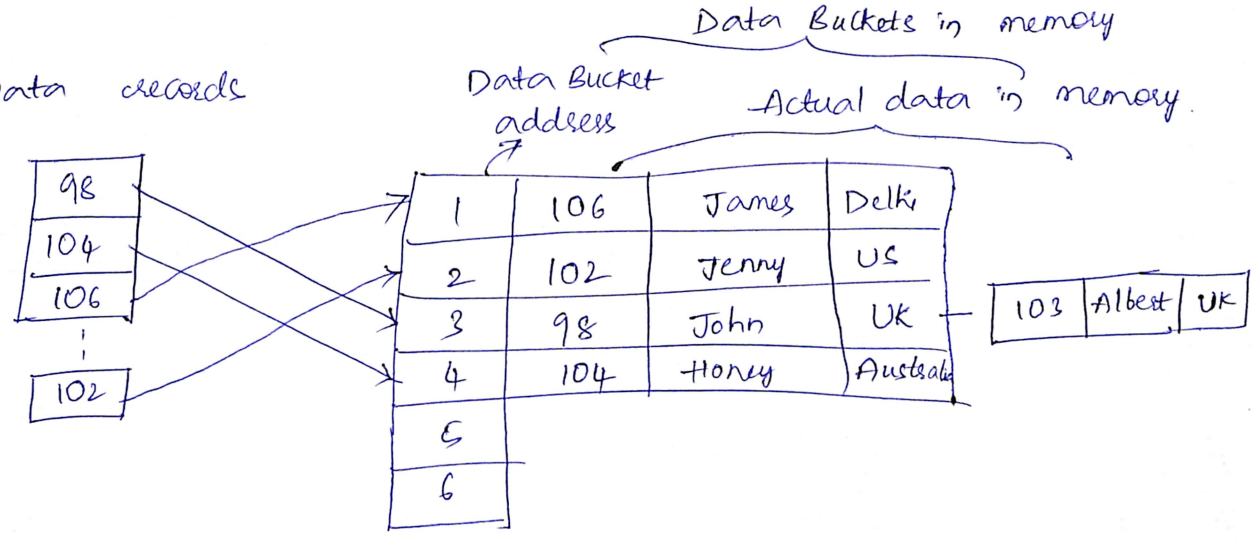


Here, block address is same as PK value. This hash function can also be

a simple mathematical fun^n like exponential, mod, cos, sin, ...

- Suppose we have mod(5) hash fun^n to determine address of the data block. It applies mod(5) hash fun^n on PK & generates 3, 3, 1, 4, 2, ... records are stored in those data block addresses.

Eg:



* Hashing Types: (i) Static Hashing (ii). Dynamic Hashing

- Bucket: A hash file stores data in bucket format. Bucket is considered as a unit of storage. A bucket typically stores one complete disk block, which in turn can store one or more records.

- Hash Function:

" , h, is a mapping fun^n that maps all the set of search keys K to the address where actual records are placed. It is a fun^n from search keys to bucket addresses.

(i). Static Hashing:

In " " , the resultant data bucket address will always be the same. that means, For example if we generate an address for Emp_id = 103 using hash fun^n mod(5) then it will give the bucket address as 3 always. Here, no change in the bucket address.

- In static hashing, the no of data buckets in memory remains constant throughout.

Operations:

(i). Searching: When a record needs to be searched, then the same hash fun retrieves the address of the bucket where the data is stored.

(ii). Insert: When a new record is inserted into the table, then we will generate an address for a new record based on the hash key & record is stored in that location.

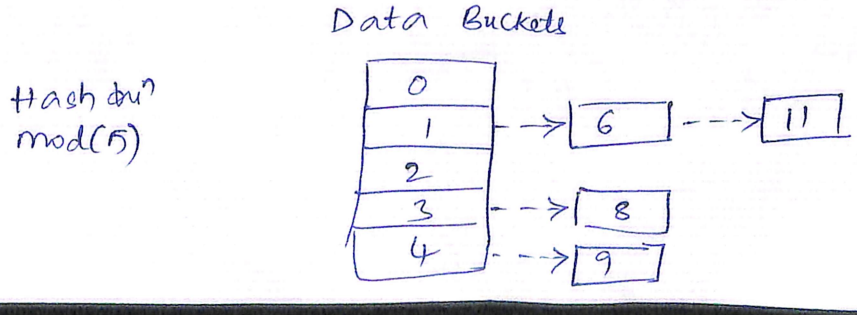
(iii). Delete: We will first fetch the record which is supposed to be deleted. Then we will delete the records for that address in memory.

(iv). Update: To update a record, we will first search it using hash function, then data record will be updated.

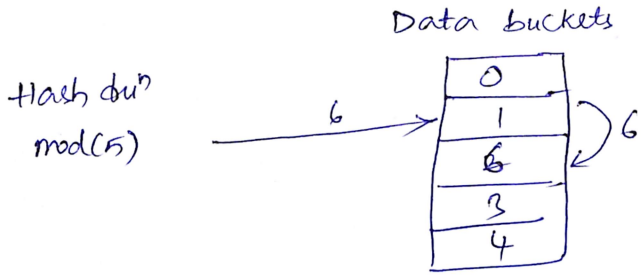
- If we want to insert some new record into the file but the address of a data bucket generated by the hash fun is not empty or data already exists in that address. This situation is known as bucket Overflow. To overcome bucket overflow, we have linear probing, Overflow Chaining.

- Bucket overflow, we can also call it as collision.

- Overflow Chaining: When buckets are full, a new bucket is allocated for the same hash result & is linked after the previous one. It is called closed hashing.



Linear probing: when a hash fun^{n} generates an address at which data is already stored, the next free bucket is allocated to it. It is called as Open Hashing.



(ii). Dynamic Hashing:

- To overcome the problems of static hashing like bucket overflow. we have dynamic hashing (or) Extendible hashing.
- In this method, data buckets grow or shrink as the records increases or decreases. It makes hashing dynamic ie., It allows insertion or deletion without resulting in poor performance.

Searching:

(i) calculate the hash address of the key

(ii). Check how many bits are used in the directory & these bits are 'i'.

(iii). Take the least significant 'i' bits of the hash address. This gives index of the ~~dictionary~~ directory.

(iv). Using the index go to directory & find bucket address where the record might be.

Insertion:

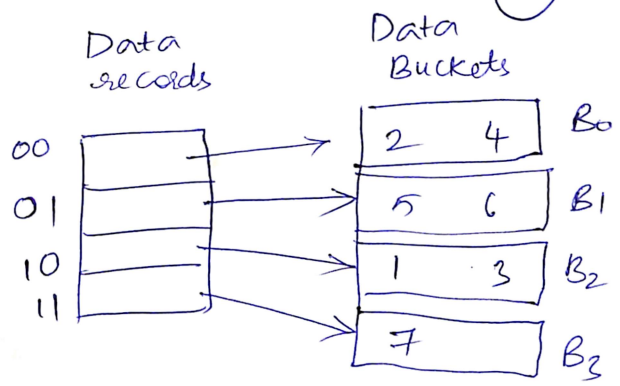
(i). Follow same procedure for retrieval, ending up in some bucket.

(ii). If there is still space in that bucket then place record in it.

(iii). If bucket is full, then we will split the bucket & redistribute the records.

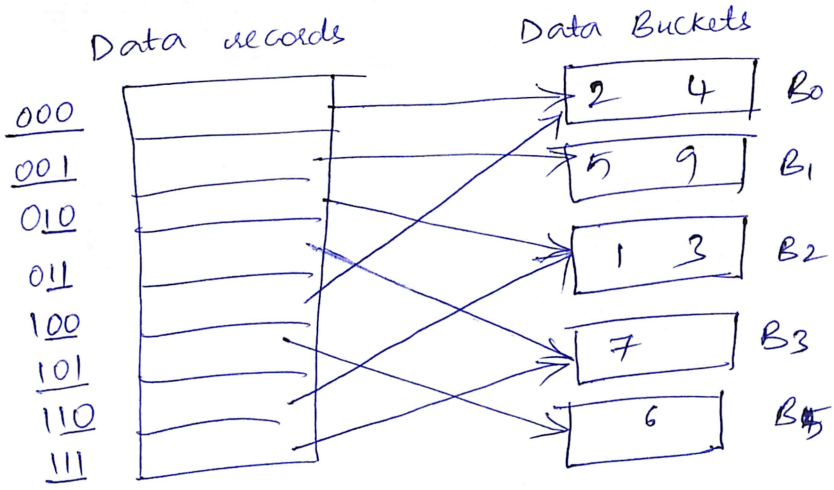
Eg:

Key	hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111



Here, last two bits need to be considered

- Suppose, if we want to insert 9 with hash address 10001 Here 9 must be stored in B_1 as it is ending with 01 But B_1 is full. So it will get split. But which element needs to be removed from B_1. We need to consider last 3 digits.



Advantages of dynamic hashing:

- performance doesn't decrease as the data grows in the system. It simply increases the size of memory to accommodate the data.
- Memory is well utilized as it grows & shrinks with the data.
- Good for dynamic DB

Disadvantages of dynamic hashing:

- If data size increases then bucket size also increased. Theses addresses of data will be maintained in the bucket address table Because data ad. will keep on changing as bucket grow & shrink.
- Bucket Overflow situation will also occur. But it take little time to reach than static.

* Transaction:

" is set of logically related operations. It contains a group of tasks A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Eg: Suppose bank Employee transfers 500/- from A's account to B's account

A's account

```
Open_Account(A)
old_balance = A.balance
New_balance = Old_balance - 500
A.balance = New_balance
close_Account(A)
```

B's account

```
Open_Account(B)
Old_balance = B.balance
New_balance = Old_balance + 500
B.balance = New_balance
Close_Account(B)
```

Operations of Transaction:

(1). Read(X) used to read the value of X from DB & stores it in buffer in main memory.

(2) Write: used to write the value back to DB from buffer.

* ACID properties:

A = Atomicity — The entire transaction takes place at once or doesn't happen at all.
C = Consistency — DB must be consistent before & after transaction.
I = Isolation — Multiple transactions occur independently without interference.
D = Durability — changes of successful transaction occurs even if system failure occurs.

(i). Atomicity: It any operation is performed on the data, either it should be performed or executed completely or should not be executed at all.

It involves the following two operations:

- Abort: If transaction aborts, changes made to DB are not visible.
- Commit: " " commits, changes made are visible.

Consider the following transaction T consisting of T_1 & T_2, transferring 100/- from X to Y.

	T_1	T_2
Before	X : 500	Y : 200
	Read(X)	Read(Y)
	X := X-100	Y := Y+100
	write(X)	write(Y)
After	X: 400	Y: 300

Here, If transaction fails after completion of T_1 (after write(x)) but before completion of T_2 (before write (y)), then amount has been deducted from X but not added to Y. This results in an inconsistent DB. The transaction must be executed in its entirety in order to ensure the correctness of DB.

(ii) Consistency:

Integrity constraints must be maintained so that the DB is consistent before & after transaction. It refers to the correctness of DB.

In the above example, total amount before and after transaction must be maintained.

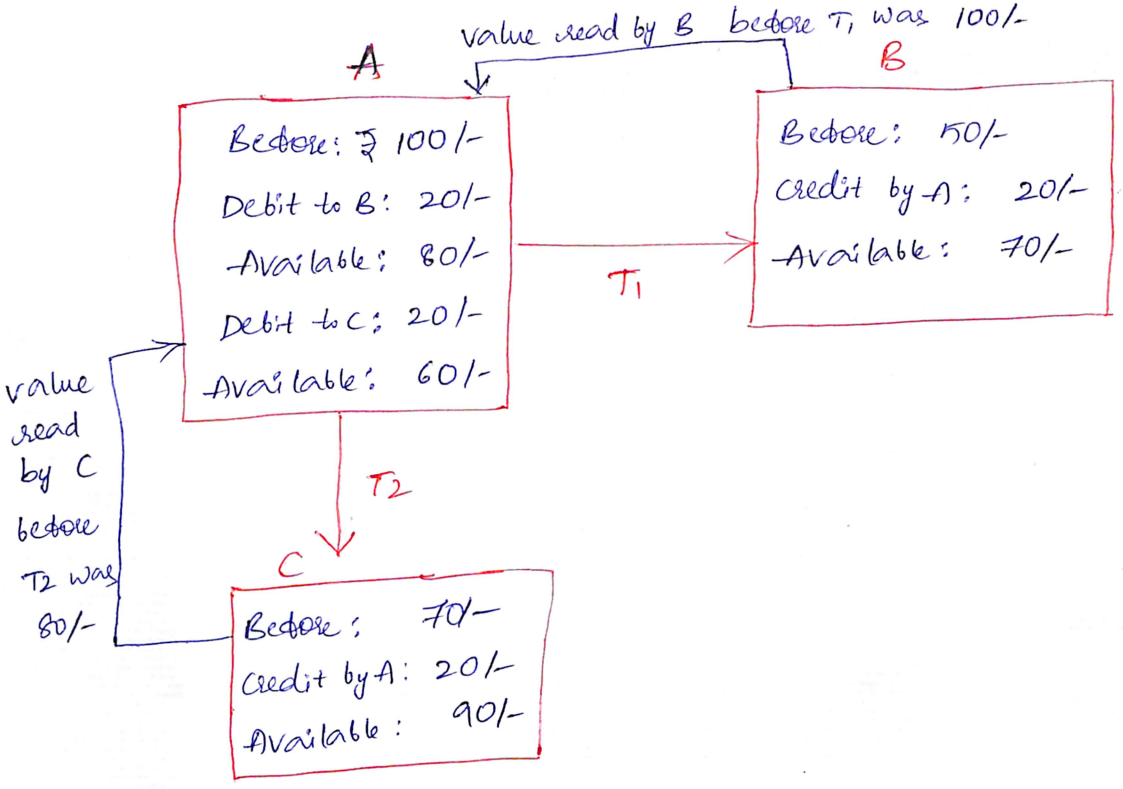
Before $T_1 + T_2 = 500 + 200 = 700$

After : $400 + 300 = 700$

So, DB is consistent. Inconsistency occurs when either of transaction fails

(iii). Isolation:

" is a property of DB where no data should affect the other one & may occur concurrently. i.e, multiple transactions can occur concurrently without leading to the inconsistency of DB. Transactions occur independently without interference.



Here, T_1 & T_2 are not affecting each other. So, transaction is in isolation.

(iv). **Durability:**

" ensures that once transaction has completed execution, the modification or updates to DB are stored in & written to disk & they persist even if a system failure occurs. These updates become permanent & stored in non volatile memory.

If anything lost, it will be rectified by recovery manager.

Atomicity — Transaction manager

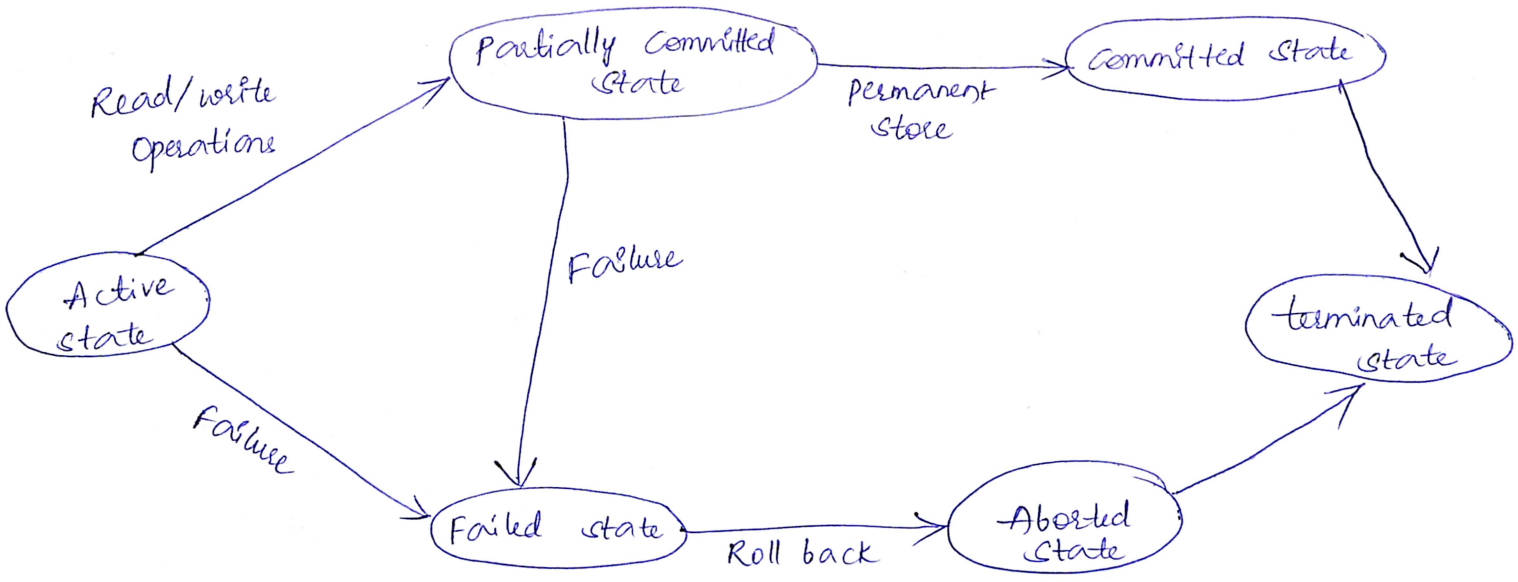
Consistency — Application programmer

Isolation — Concurrency control manager

Durability — Recovery manager.

* **States of transaction:**

States through which a transaction goes during its lifetime. State will tell the current state of transaction.



(i). Active state:

- First state of every transaction. In this state, transaction is being executed
- If all read & write operations are performed without any errors then it goes to the "partially committed state". If transaction fails, it goes to the "failed state".

(ii). partially committed state:

- After completion of all read & write operation the changes are made in main memory or local buffers. If changes are made permanent on the DB then the state will change to "committed state" & incase of failure it will go to failed state.
- Final statement of transaction is executed.

(iii). Committed state:

- It is the state when the changes are made permanent on the DB & the transaction is complete & therefore terminated in the "terminated state".

(iv). Failed state:

- When any instruction of transaction fails, it goes to the 'failed state'.
- If failure occurs in making a permanent change of data on DB.

(v). ~~Aborted~~ state: terminated:

- If there isn't any rollback or transaction comes from "committed state"

then the system is consistent & ready for new transaction & old transaction is terminated. When transaction finished, terminated.

(vi). ~~terminated~~ ~~state~~: ~~Aborted~~ state:

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

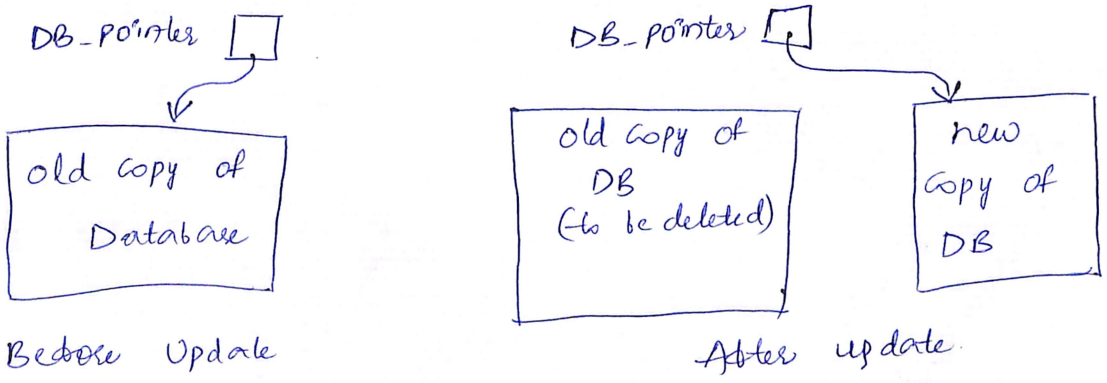
* Implementation of Atomicity & Durability:

Durability & Atomicity can be ensured by using recovery manager

- we can achieve atomicity using shadow copying technique.
- " " " Durability using logs.

i). Shadow copying technique:

- maintaining a shadow copy of original DB & redirecting all changes to the DB as a result of any transaction after committing transaction.



- DB is simply a file on disk. A pointer called DB_Pointer is maintained on disk. It points to the current copy of the DB.
- In shadow copy technique, a transaction that wants to update the DB - first creates complete copy of DB. All updates done on the new

DB copy, leaving the original copy, the shadow copy. If at any point the transaction has to be aborted, the system detects the new copy. The old copy of DB has not been affected.

- If transaction completed, It is committed as follows
  (i). OS asks to make sure that all pages of new copy of DB have been written out to disk.
  (ii). After that, the DB system updates the pointer to point to the new copy of DB. The new copy then becomes the current copy of DB. The old copy deleted.
- If transaction failure occurs at any time before db_pointer is updated, old contents are not affected. We can abort the transaction by deleting new copy of DB.
- If system fails at any time before updated db_pointer written to disk. Then, when system restarts, it will read db_pointer & will see original contents of DB. Once file is written to disk, its contents will not be damaged even if there is system failure.

(i). Logs:

- Logs keep track of actions carried out by transactions which can be used for the recovery of DB incase of failure. Log files will be stored on stable storage devices.
- When transaction begins its execution it is recorded in log file $<T_n, \text{start}>$
- When " performs an operation It is recorded in log as follows
  $<T_n, X, V_1, V_2>$
- When transaction finishes its execution $<T_n, \text{Commit}>$

* Concurrent Executions:

- Executing a set of transactions simultaneously in a preemptive & time shared method.
- Concurrency control is the management procedure that is required for controlling concurrent execution of the operations that take place on a DB.
- Concurrency control deals with interleaved execution of more than one transaction
- In a multi user system, multiple users can access & use the same DB at one time, known as concurrent execution of the DB. It means that the same DB is executed simultaneously on a multi user system by different users.
- While working on the DB transactions, there occurs the requirement of using the DB by multiple users for performing different operations
- Simultaneous execution should be in interleaved manner & no operation should affect the other executing operations, thus maintaining consistency of the DB.

problems with concurrent execution:

(i). Lost update problems (W-W conflict):

This problem occurs when two different DB transactions perform the read/write operations on the same DB items in an interleaved manner that makes the values of the items incorrect hence making the DB inconsistent.

Ex: Two transactions T_x & T_y are performed on the same account A where balance of A is 300/-

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

- At time t_1, Tx reads the value of account A ie., 300/-
- t_2, Tx deducts 50/- from A ie., 250/- (not updated / write).
- t_3, Ty reads value of A ie., 300/- because Tx didn't update value yet.
- t_4, Ty adds 100/- to A ie., 400/- (not updated / write)
- t_6, Tx writes value of A ie., 250/-
- t_7, Ty writes value of A ie., 400/-. Here, the value written by Tx is lost ie., 250/- lost. Hence data becomes incorrect & DB inconsistent

Dirty Read problems (W-R Conflict):

" " " occurs when one transaction updates an item of the DB & somehow the transaction fails & before the data gets rollback, the updated DB item is accessed by another transaction. These comes the Read-write Conflict b/w both transactions.

Eg: two transactions tx and ty performing read/write operations on account A & balance of 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	–

- At time t_1, tx reads value of account A i.e., 300/-
- " " t_2, tx adds 50/- to " A ie., 350/- (not updated)
- t_3, updated in A ie., 350/-
- t_4, ty reads account A ie., 350/-
- t_5, tx rollbacks due to server problem & value changes back to 300/- (initial value)
- But value for account A remains 350/- as ty committed

Unrepeatable Read problem (W-R Conflict):

Also called as Inconsistent Retrievals problem that occurs when in a transaction, two different values are read for the same DB item.

Eg:

Initial A = 300/-

Time	t_x	t_y
t_1	Read(A)	–
t_2	–	Read(A)
t_3	–	A = A+100
t_4	–	write (A)
t_5	Read(A)	–

- At time t_1, t_x reads value from account A ie., 300/-
- t_2, t_y reads A ie, 300/-
- t_3, t_y adds 100/- to A so 400/- (not updated)
- t_4, t_y updated value of A, 400/-
- t_5, t_x reads value of A, 400/-
- that means, In t_x, it reads two different values of account of A ie, 300/- initially after it reads 400/-. This Un repeatable read problem.

* Serializability:

A transaction is said to be serializable if it is equivalent to serial schedule. It leaves database in a consistent state.

- A serial schedule is always serializable because, a transaction starts when the other transaction finished execution.
- Two types of Serializability (i) Conflict (ii) View.

(i). Conflict Serializability Schedule:

- A schedule is called " " if after swapping of non conflicting operations, it can transform into a serial schedule.

- The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.
- Two operations become conflicting if all conditions satisfy below:
  (i) Both belong to seperate transactions
  (ii) They have the same data item
  (iii) They contain atleast one write operation.

Eg: Swapping is possible only if S_1 & S_2 are logically equal.

Schedule S_1

T_1	T_2
Read(A)	
	Read(A)

swapped →

Schedule S_2

T_1	T_2
	Read(A)
Read(A)	

Here, $S_1 = S_2$ means it is non-conflict.

Eg:

Schedule S_1

T_1	T_2
Read(A)	
	write(A)

swapped →

Schedule S_2

T_1	T_2
	write(A)
Read(A)	

Here, $S_1 \neq S_2$. It is conflict

- In conflict Equivalent, one can be transformed to another by swapping non conflicting operations

Two schedules are said to be conflict equivalent if and only if,
  (i) They contain the same set of transaction
  (ii) If each pair of conflict operations are ordered in the same way.

Eg: Non serial schedule

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

Serial schedule

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

S_2 is serial schedule because all operations of T_1 are performed before starting any operation of T2. S_1 can be transformed into a serial schedule by swapping non conflicting operations of S_1.

After swapping of non conflict operations, S_1 becomes

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

S_1 is conflict serializable.

(ii) View Serializability:

- If a schedule is view equivalent to a serial schedule, ⇒ view serializa
- If a schedule is conflict serializable, then it will be view seri
- the view serializable which does not conflict serializable contains blind writes.

Two schedules S_1 & S_2 are said to be view equivalent if they satisfy following conditions.

Initial Read:

- Initial read of both schedules must be the same.

Updated Read: In S_1, if T_i is reading A which is updated by T_j then in S_2 also, T_i should read A which is updated by T_j.

* Recoverability:

- Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we have to rollback those transactions.

T_1	T_1's buffer Space	T_2	T_2's buffer Space	Database
				A = 6500
Read (A)	A = 6500			A = 6500
				A = 6500
A = A - 500	A = 6000			A = 6000
write (A)	A = 6000			A = 6000
		Read (A)	A = 6000	A = 6000
		A = A + 1000	A = 7000	A = 6000
		write (A)	A = 7000	A = 7000
		Commit;		
Failure point				
commit				
		commit;		

- Here, we have two transactions. T_1 reads & writes the value of A & that value is read & written by T_2. T_2 commits. Due to failure, we have to rollback T_1. T_2 also should roll back because it reads the value written by T_1 but T_2 can't be rollback because it is already committed. This is irrecoverable schedule.

- If we have commit in T_2, after T_1's commit then, we can roll back both T_1, T_2 due to failure in T_1. It is recoverable with cascade rollback.

- Recoverable Cascading rollback: The schedule will be recoverable with Cascading rollback if T_j reads the updated value of T_i. Commit of T_j is delayed till Commit of T_i.

T_1	T_1's buffer space	T_2	T_2's buffer space	Database.
				A = 6500
Read (A)	A = 6500			A = 6500
A = A - 500	A = 6000			A = 6000
write (A)	A = 6000			A = 6000
Commit;				
		Read (A);	A = 6000	A = 6000
		A = A + 1000	A = 7000	
		write (A)	A = 7000	A = 7000
		Commit;		

- T_1 reads & write A & commits & that value is read & written by T_2. This is Cascadeless recoverable schedule.

* Lock based protocol:

- Any transaction can't read or write data until it acquires an appropriate lock on it. There are 2 types of lock

(1) Shared lock:

- It is also known as Read only lock. In shared lock, the data item can only read by the transaction.
- It can be shared b/w the transactions because when the transaction holds a lock, then it can't update the data on data item.

(ii). Exclusive lock:

- The data item can be both reads as well as written by transaction.
- This lock is exclusive & in this lock, multiple transactions donot modify the same data simultaneously.

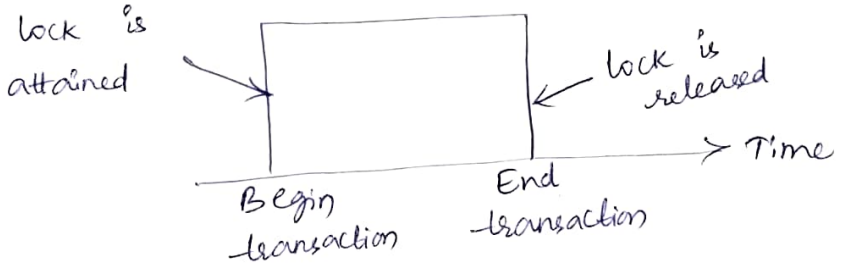
types of lock protocols:

(i). Simplistic lock protocol:

- It is the simplest way of locking the data while transaction.
- It allow all the transactions to get the lock on the data before insert or update or delete on it. It will unlock the data item after completing the transaction.

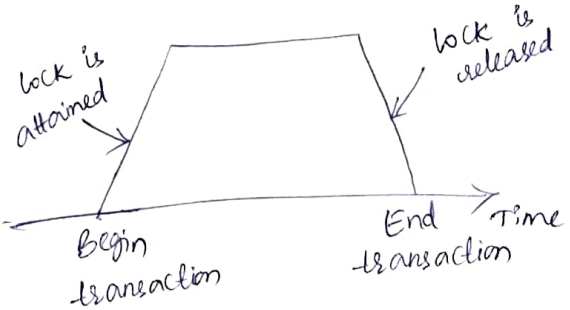
(ii). pre-claiming lock protocol:

- Before initiating the transaction, it requests DBMS for the lock.
- If locks are granted then this protocol allows the transaction to begin. when the transaction is completed then it releases the lock.
- If all the locks are not granted then it allows the transaction to rolls back & wait until all locks are granted.



(iii). Two-phase locking protocol:

- It divides the execution phase into 3 parts
- In first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In second part, transaction acquires all the locks.
- In third part, transaction can't demand any new locks. It only releases the acquired locks.



Growing phase:

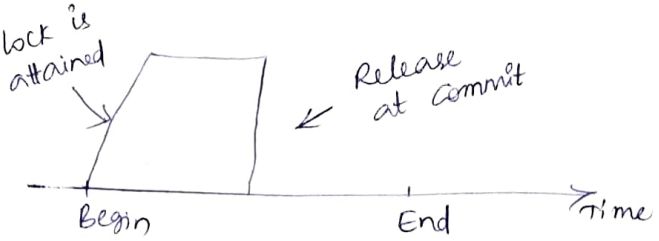
- In " ", a new lock on the data item may be acquired by the transaction. But none can be released.

Shrinking phase:

- existing lock held by the transaction will be released, but no new locks can be acquired.

(iv). Strict Two-phase locking protocol (Strict-2PL):

- In first phase, after acquiring all the locks, transaction continues to execute normally.
- Strict 2PL doesn't release the lock after using it. So, it does not have shrinking phase of lock release.
- strict 2PL waits for the whole transaction to commit



* Time stamp based protocols:

- " " " " is used to order the transactions based on (ascending) their timestamps.
- priority of the older transaction is higher that's why it executes first.
- To determine the timestamp of the transaction, it uses system time or logical counter.
- TS(Ti) denotes the timestamp of transaction Ti.

  R_TS(x) denotes the Read time stamp of data item x.

  W_TS(x) denotes the write time stamp of data item x.

(i). Whenever Ti issues a read(x) operation:

- If W_TS(x) > TS(Ti) then the operation is rejected.
- If W_TS(x) <= TS(Ti) " " " " executed.
- Timestamps of all data items are updated.

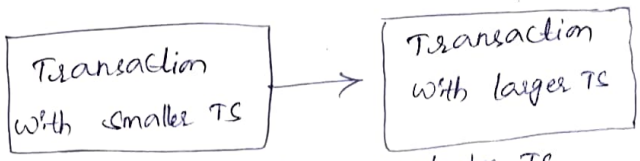
(ii). Whenever Ti issues a write(x) operation:

- If TS(Ti) < R_TS(x) then the operation is rejected.
- If TS(Ti) < W_TS(x) " " " & Ti is rolled back Otherwise / or else operation is executed, Sets W_TS(x) to TS(Ti).

Advantages

(i). It ensures serializability

(ii). It ensures freedom from deadlock that means no transaction ever waits



precedence graph for TS.

Disadvantage:

(i). Schedule may not be recoverable & may not even be cascade-free.

* Validation based protocols:

- " " , also called as Optimistic concurrency control technique.

The transaction is executed in following three phases.

(i) Read phase: The transaction T is read & executed. It is used to read the value of various data items & stores them in temporary local variables. It can perform all write operations on temporary variables without an update to the actual DB.

(ii) validation phase: The temporary variable value will be validated against the actual data to see if it violates the serializability.

(iii) write phase: If validation of the transaction is validated, then the temporary results are written to the DB or System otherwise the transaction is rolled back.

- Each phase has the following different timestamps:

start(T_i): It contains the time when T_i started its execution.

validation(T_i): " " " , T_i finishes its read phase & starts its validation phase.

Finish(T_i): It contains the time when T_i finishes its write phase.

- validation phase will check the following:

(i) Finish(T_j) < start(T_i)

T_j finishes its execution means completes its write phase before T_i started its execution (read phase). Then the serializability maintained.

(ii) T_i begins its write phase after T_j completes its write phase, & the read_set of T_i should be disjoint with write_set of T_j.

(iii) T_j completes its read phase before T_i completes its read & read_set & write_set of T_i are disjoint with the write_set of T_j

Advantages:

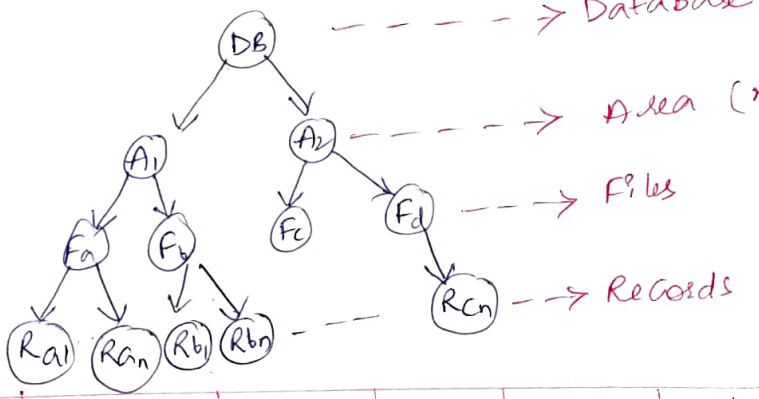
- Avoids cascading rollbacks
- Avoids deadlock

Disadvantage:

- Starvation:
  ↳ when a transaction has to wait for an indefinite period of time to acquire lock.

* Multiple granularity:

- Granularity is the size of data item allowed to lock.
- Multiple granularity is defined as hierarchically breaking up the DB into blocks which can be locked.
- It enhances concurrency & reduces lock overhead.
- It maintains the track of what to lock & how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. this type of hierarchy graphically represented as a tree.



	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

IS – Intention Shared
IX – Intention Exclusive
S – Shared
SIX – Shared & Intention Exclusive
X – Exclusive

Intention shared (IS): It contains explicit locking at a lower level of tree but only with shared locks.

Intension - Exclusive (IX): " " at lower level with exclusive or shared locks.

Shared & Intention exclusive (SIX): Here, the node is locked in shared mode, some node is locked in exclusive mode by same transaction.

- Multiple granularity uses intention lock modes to ensure serializability. It requires that a transaction T_i that attempts to lock a node must follow these protocols:

(i) T_i must follow lock compatibility matrix

(ii) T_i must lock the root first & it can lock any node.

(iii) T_i can lock any node in S or IS only if T_i currently has the parent of the node locked in either IX or IS mode.

(iv) T_i can lock a node in X, SIX or IX mode, if T_i currently has the parent of the node locked in either IX or SIX modes.

(v) T_i can lock a node only if T_i has not previously unlocked any node.

(vi) T_i can unlock a node only if T_i currently has none of the children of the node locked.

Eg: T_1 transaction reads record Ra_2 in file Fa. Then, T_2 needs to lock DB, area A_1, Fa in 'IS' mode & finally lock Ra_2 in 'S' mode.