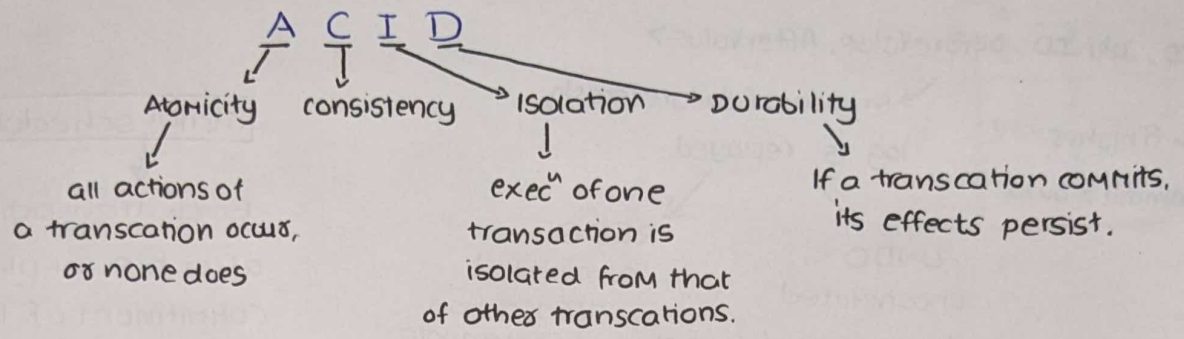
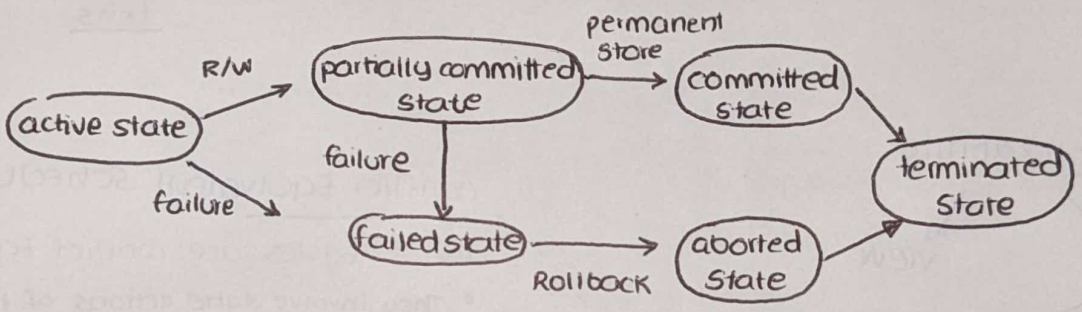
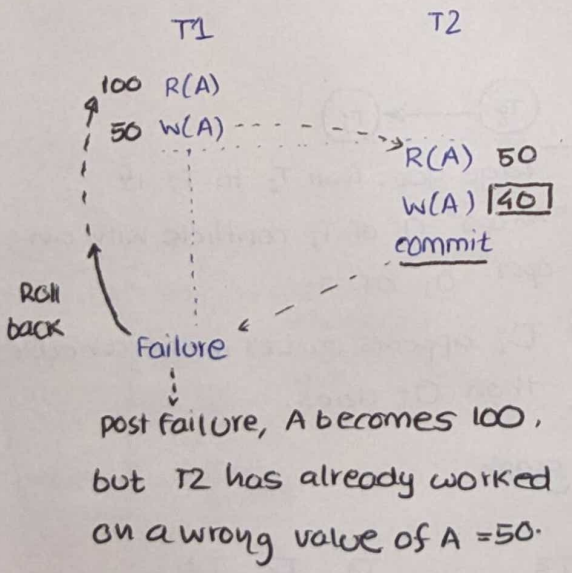
Transaction correctness criteria:





Concurrency Problems

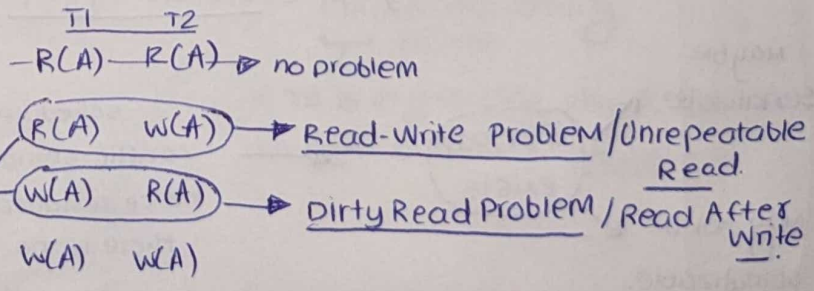
Dirty Read / Read After Write

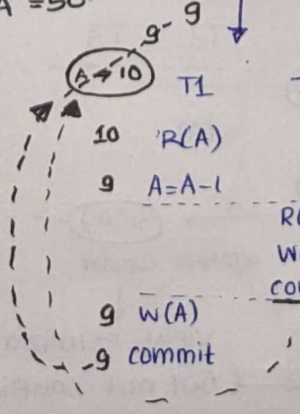


post failure, A becomes 100, but T2 has already worked on a wrong value of A = 50.

Lost Update

One commit over writes the changes made by some prev. commits.





↳ 2 books were issued, one each in T1 & T2, but A should have been 8, not 9.

Write-ahead log.

<txnID, obj.ID, beforeValue, AfterValue>

- when a txn finishes, it writes <commit> onto the log
- in case of failure/crash, log is replayed.
  - UNDO uncommitted transaction statements
  - REDO committed transaction statements.

serial schedules

Each transaction occurs after the completion/commitment of the prev. txn. No interleaving of txns.

Serializability

- conflict
- view

conflict Equivalent schedules

Two schedules are conflict Equivalent iff:
- They involve same actions of the same txns.
- Every pair of conflicting actions is ordered the same way.

schedule S is **conflict serializable** iff
- S is conflict equivalent to some serial schedule
- Able to transform S into a serial sched. by swapping consecutive non-conflicting operations of different transactions.

another way to check conflict serializability

Make a directed graph (dependency graph)

Edge goes from T_i to T_j if
- An opr^n O_i of T_i conflicts with an oper^n O_j of T_j
- O_i appears earlier in the schedule than O_j does.

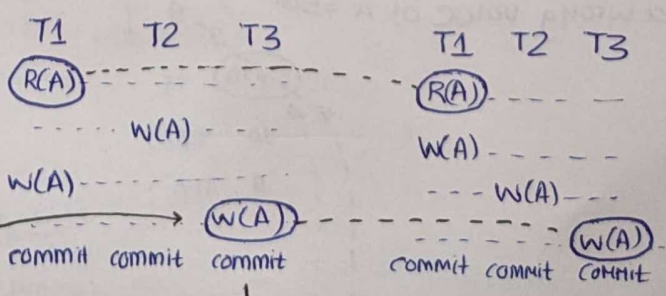
schedule is conflic serializable (and hence serializable) if there is no loop in the graph.

if a loop exists
- Maybe serializable
- May not be serializable.

this is why

view serializability

- If T1 reads initial value of A in S1, then T1 reads initial value of A in S2.
- If T1 ~~reads~~ writes final value of A in S1, then T1 writes the final value of A in S2.
- ★ T1 reads value of A written by T2 in S1, then T1 does the same in S2.



view serializable (but not conflict serializable)

dependency graph is cyclic.

Locking based concurrency control

- Shared locks S(A)
  - (Read only)
  - can be shared, ie. can be accessed by multiple txns simultaneously.
- Exclusive locks X(A)
  - (Read + write)
  - can't be shared until one txn releases the lock.

Compatibility Matrix

T1 has \ T2 wants	S	X
S	yes	no
X	no	no

Example:

Lost update w/o locks

T1	T2
R(A)	
	R(A)
W(A)	
commit	conflict
	W(A)
	commit

drawbacks → starvation, deadlock

- may not always give serializable schedules
- may not be free from irrecoverability

With locks

T1	T2
LOCK(A)	
R(A)	
	LOCK(A) → denied because T1 has A's lock
W(A)	
UNLOCK(A)	
commit	
	R(A) → lock granted to T2 for A after T1 releases it.
	W(A)
	UNLOCK(A)
	commit

2PL (Phase locking) ★★ always ensures serializability

- growing: locks are acquired, no lock is released
- shrinking: no lock is acquired, locks are released

follow compatibility matrix

If T1 is in growing phase, then T2 can acquire a shared lock

T1	T2
X(A)	
R(A)	
W(A)	
UN(A)	S(A) → read after write (dirty read)
	R(A)
	COMMIT

rollback failure → can't roll back

drawbacks → starvation, deadlock, may not be recoverable, cascading rollback/abort

Sol^n

- Strict 2PL: basic 2PL + all exclusive locks should hold until commit/abort.
- Rigorous 2PL: basic 2PL + all X, S locks should hold until commit/abort

Timestamp ordering Protocol

$TS(T_i)$ → timestamp assigned to i^{th} transaction.

not necessarily the time at which the txn enters; a random integer value.

	10:00	10:05	10:15
	T_1	T_2	T_3
TS:	100	200	300

→ youngest

← oldest

Read Timestamp (RTS)

TS of the latest txn that performed a successful read operation.

Write Timestamp (WTS)

TS of the latest txn that performs a successful write op^n.

Basic T/O Rules

1) Transaction T_i issues a Read(A) op^n :

a) if $WTS(A) > TS(T_i) \Rightarrow$ Rollback T_i

b) otherwise, execute R(A);

set $RTS(A) = \max\{RTS(A), TS(T_i)\}$

2) Transaction T_i issues write(A) op^n :

a) if $RTS(A) > TS(T_i) \Rightarrow$ Rollback T_i

b) if $WTS(A) > TS(T_i) \Rightarrow$ Rollback T_i

c) otherwise, execute W(A) op^n;

set $WTS(A) = TS(T_i)$

Dense Index

an entry for each key value of every record.

Sparse Index

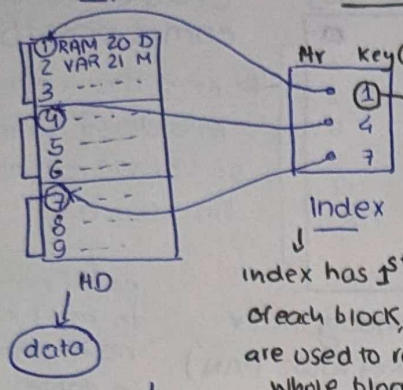
entries for only some of the key values out of all records

index → key value & ptr

single level ordered indices

- primary → key
  - 1→ 2→ 3→ 4→ 5→ 6→ (ordered)
- clustered → non key
  - 1→ 1→ 2→ 2→ 2→ 3→ 4→ 4→ (ordered)
- secondary → unordered
  - key: 1→ 5→ 3→ 2→ 4→ (unordered)
  - non key: 1→ 5→ 2→ 2→ 4→ 3→ 1→ (unordered)

primary Index → (sparse Index)



index has 1st entries of each block, which that are used to represent the whole block.

No. of entries in Index = No. of blocks in memory

search time : $\log_2 N + 1$ → searching for specific data after reaching the req. block.

disadvantages: To insert a record at its correct pos^n, move records to make space; we might change the anchor records in the data which may lead to an obsolete index, if not changed simultaneously. same for del^n

Example:

r = 30,000 records
Blocksize (B) = 1024 B
Record size (R) = 100 B

→ Blocking factor (BFR) = $\frac{B}{R} = \frac{1024}{100} = 10$ records/block

No. of blocks (b) = $\frac{r}{BFR} = \frac{30000}{10} = 3000$

using indexing:

field value size : 9B
Block ptr size : 6B
→ 15 B (size of each entry)

Blocking factor for index = $\frac{1024}{15} = 68$ entries/block

Total no. of index entries = 3000 (= no. of blocks)

∴ ~~no. of entries in index~~ = 3000 → search time

$\approx \log_2 3000 + 1 = 12 + 1$

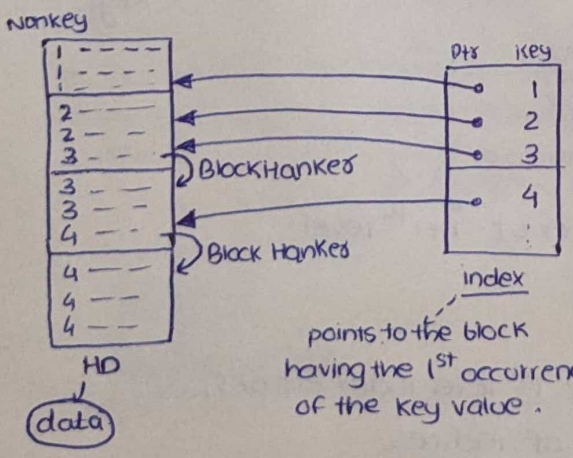
↳ no. of block access. (without indexing)

↓ no. of index blocks $b_i = \frac{3000}{68} = 45$

∴ binary search on index requires $\log_2 45 = 6$ block access

& searching a record using index = 6 + 1 = 7 block access

clustering Index → (sparse Index)



points to the block having the 1st occurrence of the key value.

→ rest is same as primary index

→ access time using index:

$\log_2 b_i + 1 + 1$

- $\log_2 b_i$: bin. search on index
- 1 : searching in the block
- 1 : the entry might be present in the subsequent blocks.

↳ disadvantages: insertion & deletion.

↓

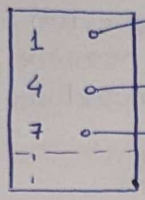
possible sol^n:

↓

reserve the whole block for each value of a clustering index.

Secondary Index (on Key value)

unordered (on Non Key value)

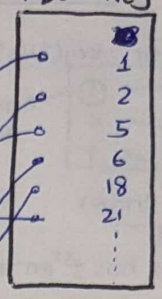


sparse index ← primary index (based on EID)

fails if the results are to be searched on the basis of "PAN"

EID	Name	PAN
1	A	40
2	B	51
3	A	63
4	C	21
5	B	5
6	E	18
7	F	1
8	A	6
9	G	2

Primary key (ordered) | non key (unordered) | candidate key (unordered)



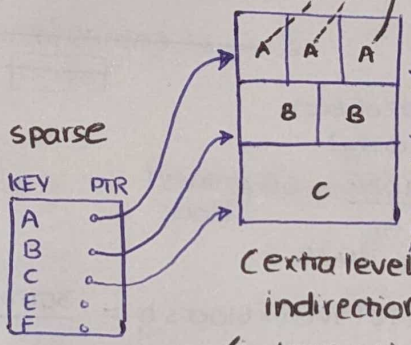
dense index (no blocks anchor entries possible)

keep the keys in ordered manner so that we can apply bin. search.

no. of entries in index = no. of entries in table.

secondary index (on key value 'PAN')

access time = $\log_2 N + 1$ (N = no. of entries (total))



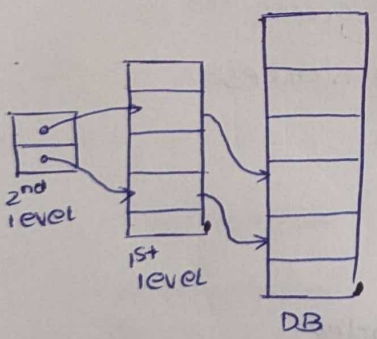
(extra level of indirection) (intermediate index)

secondary index (on Non key value 'Name')

access time = $\log_2 N + \frac{N}{\text{no. of entries in block of a particular key}} + 1$ (retrieve data)

$\log_2 N$ → # 1st index entries

Multilevel Index



- Fanout (F_o) = $\frac{B}{P+V}$ (B = block size; P = pointer size; V = key value size; P + V = 1st level index entry size)

- Number of entries in the index at $i+1^{th}$ level is $\gamma_{i+1} = \lceil \gamma_i / F_o \rceil$

- A multilevel index with γ_i 1st level index entries, will have approx. 't' levels of indices,

$$t = \lceil \log_{F_o} \gamma_i \rceil$$

Multilevel Dynamic Indexing Using B-Trees (Balanced Trees)

- p → Max no. of children a node can have
- Each node, except leaf and roots, has atleast $\lceil p/2 \rceil$ pointers (children)
- All leaf nodes are at the same level. All the pointers of the leaf nodes are NULL.