Transaction

A → Rs 50 → B

Begin Txn

Read(A);
A = A - 50;
write(A);

Atomicity

Transaction

Read(B);
B = B + 50;
write(B);

End Txn

Problems:

1) poweroff, system crashed, . . .

2) concurrency / concurrent users

Rs 2500, 5 tickets

4 → available

2 → available

Transaction:

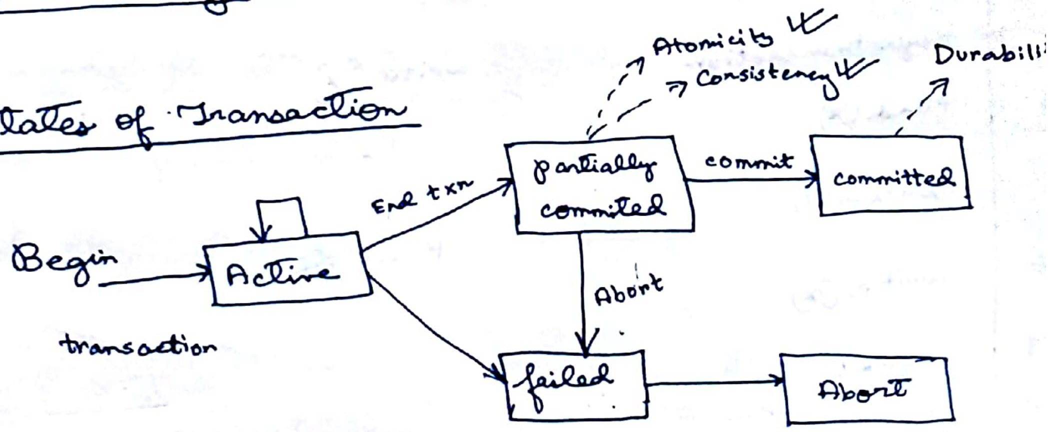
A set of operations grouped together.

1. Atomicity
2. Consistency
3. Isolation
4. Durable

Acid Properties

- Serializability

States of Transaction



Lost Update Problem

	T_1	T_2
t_1		begin transaction
t_2	begin txn	read(x)
t_3	read(x)	$x = x + 100;$
t_4	$x = x - 10;$	write(x);
t_5	write(x)	commit
t_6	commit	

2) Uncommitted Dependency Problem (~~Wrong~~ Dirty Read Problem)

	T_1	T_2
t_1		begin txn
t_2		read(x)
t_3		$x = x + 100$
t_4	begin transaction	write(x)
t_5	read(x)	
t_6	$x = x - 10$	Rollback
t_7	write(x)	
t_8	commit	

③ Inconsistent Analysis Problem

	T1	T2
t_1		
t_2	begin txn	begin txn
t_3	read(x)	sum=0
t_4	x=x-10	read(x)
t_5	write(x)	sum = sum + x
t_6	read(z)	read(y)
t_7	z=z+10	sum = sum + y
t_8	write(z)	
t_9	commit	
t_{10}		read(z)
t_{11}		sum = sum + z
		commit

Schedule: A sequence of operation by a set of transaction that preserves the order of the operations in each of the individual transaction.

Schedule S1

Time	T_7	T_8
t_1	begin txn	
t_2	R(x)	
t_3	W(x)	
t_4		begin Txn
t_5		R(x)
t_6		W(x)
t_7	R(y)	
t_8	W(y)	
t_9	commit	
t_{10}		R(y)
t_{11}		W(y)
t_{12}		commit

Schedule S2

time	T_7	T_8
t_1	begin Txn	
t_2	R(x)	
t_3	W(x)	
t_4		begin Txn
t_5		R(z)
t_6	R(Y)	
t_7		W(x)
t_8	W(Y)	
t_9	commit	
t_{10}		R(Y)
t_{11}		W(Y)
t_{12}		commit

Schedule S3

T_7	T_8
begin Txn	
R(x)	
W(x)	
R(Y)	
W(z)	
commit	
	begin Txn
	R(x)
	W(x)
	R(Y)
	W(Y)
	commit

- Schedules S1, S2 and S3 are <u>equivalent</u>.

<u>Serial Schedule:</u> A schedule where the operations of each transaction are executed consecutively without any interleaved operation from other transaction.

<u>Non-Serial Schedule:</u> Interleaved operations are present

- <u>Serializability</u> — Definition

S1 and S3 are serializable schedules.

Scanned by CamScanner

① T_1 reads x, T_2 also reads x okay. — no conflict by changing order of execution

② T_1 reads x, T_2 reads $y (\neq x)$ — no conflict by changing order of execution

③ T_1 writes x, T reads x, or vice versa — order matters

S3

T_1	T_2
Begin	
R(x)	
W(x)	
R(y)	
W(y)	
commit	
	R(x)
	W(x)
	R(y)
	W(y)
	commit

S1

Time	T_1	T_2
t_1	Begin	
t_2	R(x)	
t_3	W(x)	
t_4		Begin
t_5		R(x)
t_6		W(x)
t_7	R(y)	
t_8	W(y)	
t_9	commit	
t_{10}		R(y)
t_{11}		W(y)
t_{12}		commit

S2

T_1	T_2
Begin	
R(x)	
W(x)	
	Begin
	R(x)
R(y)	
	W(x)
W(y)	
commit	
	R(y)
	W(y)
	commit

S1, S2 and S3 are equivalent.

These schedules are called <u>conflict serializable</u>.

Non-conflict serializable

Time	T_3	T_4
	Begin	
	R(x)	
	x=x+10	
	W(x)	
		Begin
		R(x)
		x=x+1.1
		W(x)
		R(y)
		y=y+1.1
		W(y)
		commit
	R(y)	
	y=y-100	
	W(y)	
	(commit)	
	Rollback	

read of
mit(suppose),
e to some error

Precedence Graph

Navathe/Korth

1) Create a node for each transaction
2) Create a directed edge from T_i to T_j, if T_j reads an item written by T_i.
3) T_j writes a value into an item, read by T_i, draw an edge for T_i to T_j.

4) T_j writes an item, already written by T_i draw an ~~item written~~ edge from T_i to T_j

Precedence graph of S3

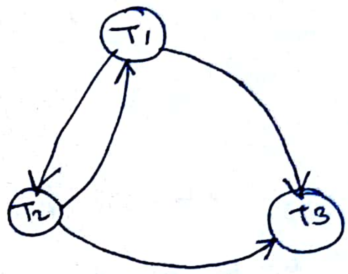
for S2

for S1

- If a cycle is present in the precedence graph, then the graph may not be conflict serializable.

~~Schedule~~ Schedule ~~S1~~ S4

Time	T_1	T_2	T_3
t_1	Begin		
t_2	R(x)		
t_3		Begin	
t_4		W(x)	
t_5		commit	
t_6	W(x)		
t_7	commit		
t_8			Begin
t_9			W(x)
t_{10}			commit
t_{11}			
t_{12}			



∴ non-conflict serializable

Two schedules S_1, S_2, each consisting of $T_1, T_2 \dots T_n$ transactions and they satisfy the following properties —

1) T_i reads x ~~in~~ in S_1 and S_2

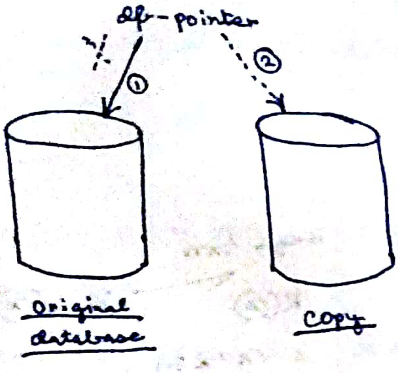
2) T_i ~~read~~ reads x, which has been written by T_j in $S1$ and $S2$

3) For each data item x written by T_i (last operation) in S_1, then in $S2$ also the last write operation is performed by T_i

- ~~It~~ No ~~and~~ matter how much we successfully rearrange the schedule ~~by~~ $S3$, it always satisfies the above ~~to~~ 3 properties.
- If a schedule satisfies these 3 properties it is called View Serializable.

- Any ~~conflict~~ conflict serializable schedule is ~~conflict~~ view serializable, but not the other way round.

ecoverability



Disadvantages

1) Expensive
2) Space consumption more

— Lost Update
— Uncommitted dependency
— Inconsistent analysis

Locking Mechanism

— shared lock (more than 1 transaction can read a data item)
— exclusive lock
(No other transaction can read or write a data item)

• Locking can be on a particular field and also on an ~~and~~ entire database.

Working Principle:

1) A transaction wants to access 'x'.
  — T requests a shared lock.
  — If x is been placed in a shared lock by some other transaction, request is approved.

2) If x is been exclusively locked by another transaction, then T has to wait.

• This locking is time consuming. Hence upgrading and downgrading of locking is necessary.

Two phase locking (2 PL)

$x = 100, y = 100$

	T_9	T_{10}
t_1	begin tran ← X_L(x)	
t_2	R(x)	
t_3	$x = x + 100$	
t_4	W(x)	
t_5	R_X_L(x)	begin tran
t_6		R(x) ← X_L(x)
t_7		$x = x * 1.1$
		W(x)
t_8		R(y)
t_9		$y = y * 1.1$
t_{10}		W(y)
t_{11}	R(y)	commit
t_{12}	$y = y - 100$	
t_{13}	W(y)	
t_{14}	commit	

$x = 220, \; y = 340$

$x = 100, y = 100$

T_9	T_{10}
X_L(x)	
R(x)	
$x = x + 100$	
W(x)	
R_X_L(x)	
	X_L(x)
	R(x)
	$x = x * 1.1$
	W(x)
	R_X_L(x)
	X_L(y)
	⋮
	R_X_L(y)

Scanned by CamScanner

If we perform T_9 and T_{10} serially,

T_9 first, then $T_{10} \longrightarrow x = 220, y = 340$

T_{10} first, then $T_9 \longrightarrow x = 210, 340$

$\therefore$ inconsistency still occurs here.

Solutions:

- Once you put a lock, do not unlock until the end of the transaction.
- Once we lock one data item, we can keep on locking other data items without unlocking the previous locks. But if we unlock the lock on a data item, we cannot further perform any locks.

~~→~~ A transaction follows the two phase locking protocol if —

all locking ~~the~~ operations precede the first unlock operation in the transaction.

2-phase locking:
- — Growing Phase: You keep on locking data items
- — Shrinking Phase: You keep on unlocking one by one

Scanned by CamScanner

~~No lost~~

Example 1:

Lost Update

	T_1	T_2
t_1		Begin
t_2	Begin	W_L(x)
t_3	W_L(x)	R(x)
t_4	wait	$x=x+100$
t_5	wait	W(x)
t_6	wait	commit / unlock(x)
t_7	R(x)	
t_8	$x=x-10$,	
t_9	W(x)	
t_{10}	commit / unlock(x)	

— No Lost Update

— May lead to deadlock (Problem)

Example 2:

Dirty Read/ Uncommitted Dependency Problem

In T_2, if we perform rollback instead of commit, ie. T_2 becomes —

T_2

Begin

W_L(x)

R(x)

$x = x+100$

W(x)

rollback / unlock(x).

Here since we do not ~~s~~ commit, hence unlock(x) is not performed.

Hence, T_1 cannot lock x and read item x.

∴ there is no problem of ~~dirty~~ dirty read.

Here also 2 phase locking provides a solution.

Cascading Rollback

t_a	t_b	t_c
W-L(x)		
R(x)		
R-L(y)		
R(y)		
x = x + y		
W(x)		
unlock(x)		
	W-L(x)	
	R(x)	
	x = x + 100	
Rollback	W(x)	
	unlock(x)	
		R(x)
	Rollback	
		Rollback

If there are many txns with a lock on one item, then if the 1st txn rolls backs, all the others must roll back too.

- **Deferred Update**: Unless there is a commit, the exact values won't be updated
- **Immediate Update**: As soon as we update, the database along with log file is updated.

Indices, Storage, B, B^+ Trees

- Sequential
- Random Access
- Hashing
  - bucket

    $h(x) = 1, \ 1 \leq x \leq 100$

    $= 2, \ 101 \leq x \leq 200$

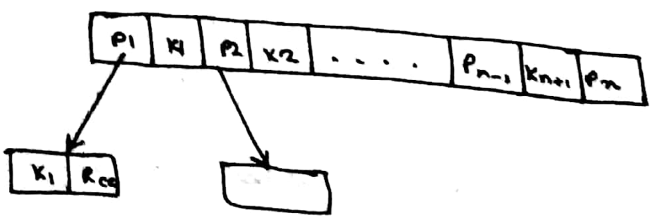
- Storing

  Retrieving is faster

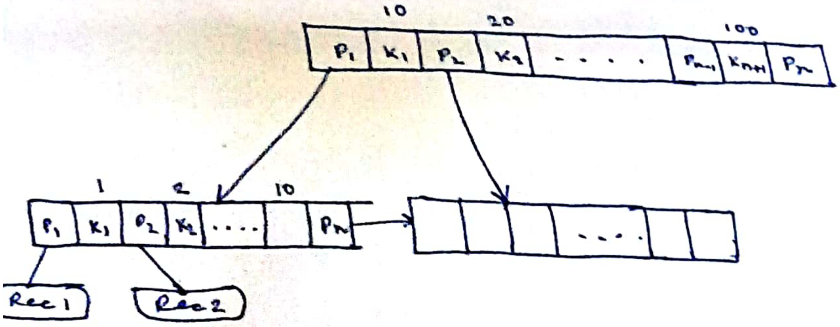
- DBMS always creates a <u>primary index</u> on a table.
- If we require to fetch a large amount of data from the database using some attribute which is not the primary key, we can use <u>secondary index</u>.

  The attribute on which secondary indexing is created need not be necessarily unique.

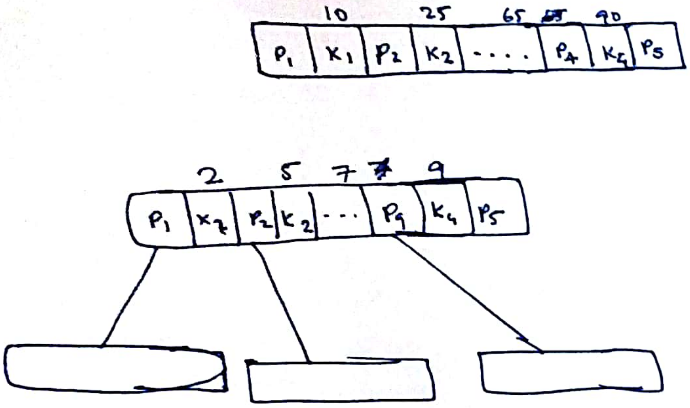
  <u>B^+ Trees</u>



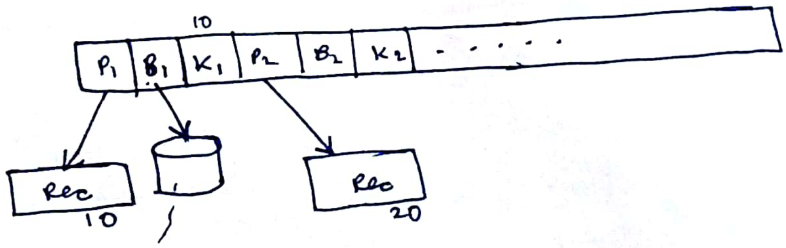
Example 1:



Example 2:



B Tree



Bucket storing all the records less than K1 (in this case 10)