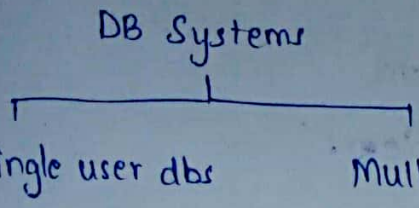
FOUNDATIONS OF DATABASE

TRANSACTION PROCESSING

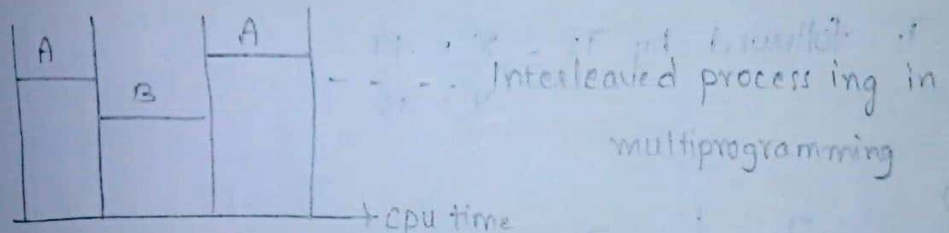
Introduction to Transaction processing:

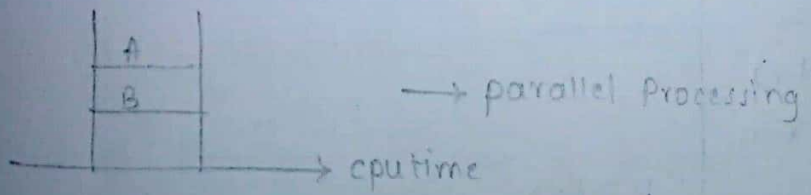


In an application, if dbs are allowed to insert, update the data bases only for single user - Single user dbs

Multi user dbs - Railway reservation, Banking system etc

multiuser dbs which allows to access multiple users concurrently are known as Online Transaction Processing Systems





Only a parallel processor can do pasallel processing.

A logical, atomic unit of database procesing requesting for to execute insert, delete, update operations is called Transaction.

Transaction accesses the data which is organized into DB items. These items may be row, block, disk, attribute

value

Need of Concurrency Control: To avoid following problems

ⓐ Last update problem
ⓑ Dirty read problem
ⓒ Incorrect summary problem
ⓓ Unrepeatable read problem.

T_1	T_2
Read_item(x)	Read_item(x)
x := x - N	x = x + m
Write_item(x)	Write_item(x)
Read_item(Y)	
Y = Y + N	
Write_item(Y)	

x = No. of tickets for flight x = 50
Y = No. of tickets for flight Y = 40
N = 5 m = 4

T_1 followed by T_2 — x = 49
Y = 45

ⓐ →

Fig@

T_1	T_2
R_I(x)	
x = x - n	
	R_I(x)
	x = x + m
W_I(x)	
R_I(Y)	
Y = Y + n	
	W_I(x)
W_I(Y)	

x = 54 Y = 45

Scanned with CamScanner

(b) →

fig (b)

T_1	T_2
R1(x)	
x = x - n	
W-1(x)	
abort	R-I(x)
	x = x + m
	W-I(x)

x = 49
y = 40

(c) →

T_1	T_3
	sum = 0
	RI(N)
	sum = sum + n
Ri(x)	
x = x - n	
wi(x)	
	R-I(x)
	sum = sum + x
	R-I(y)
	sum = sum + y
Ri(y)	
y = y + n	
wI(y)	

Need of Recovery: In case of failures

(a) System crash
(b) Local errors of exceptions
(c) Disk failures
(d) Physical catastropic failure

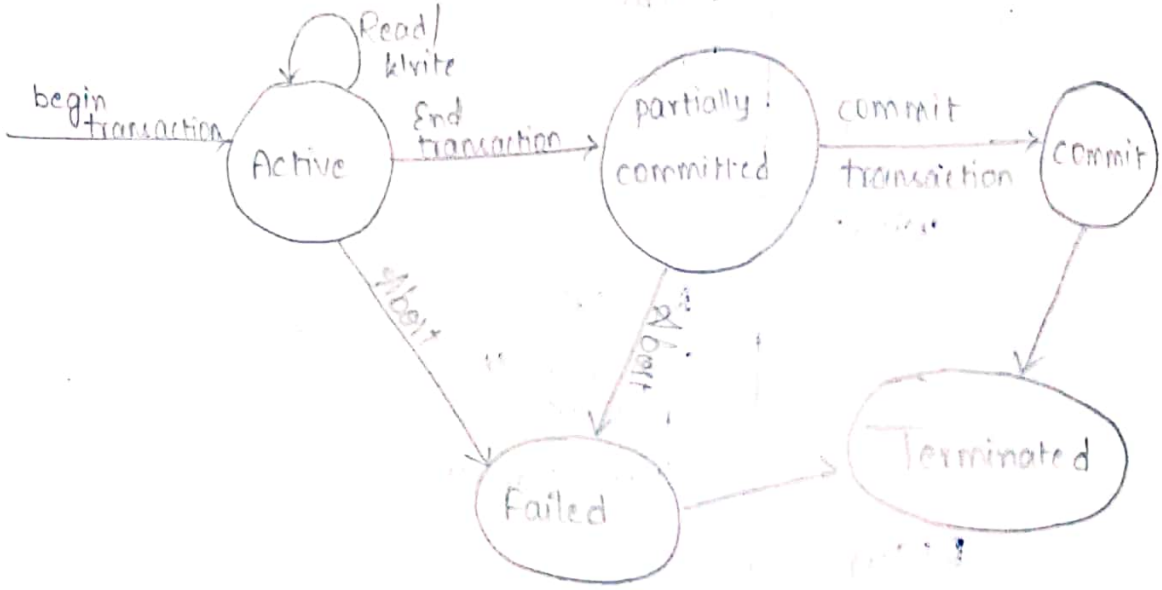
Scanned with CamScanner

Transaction and System Concepts:

Each transaction contains

1. Begin transaction
2. Read or write
3. End transaction
4. commit transaction
5. Abort

States of Transaction:



A System log contains info about transaction. System log is created in disk.

Desirable Properties of a transaction:

1. Atomicity – ensured by transaction recovery subsystem
2. Consistency Preservation - concurrency control subsystem
3. Isolation
4. Durability or Permanancy

Scanned with CamScanner

Schedules:

A schedule S of n transactions $T_1, T_2, T_3 \ldots T_n$ is ordering of operations

$S_a = r_1(x); r_2(x); w_1(x), r_1(y); w_2(x); w_1(y);$

$S_b = r_1(x); w_1(x); r_2(x); w_2(x); a_1;$

Conflict Vs non-conflict operations:

(a) $r_1(x); w_2(x)$ - conflict (c) operation

(b) $w_1(x); w_2(x)$ - conflict (c) operation

(c) $r_2(x); w_1(y)$ - non-conflict (nc) operation

Schedule - Introduction:

read-write conflict

for ex: $r_1(x); w_2(x)$

interchange in S_a

$S_{a_1} = r_2(x); r_1(y); w_2(x); r_1(x); w_1(x); w_1(y)$

If The result from S_a and result from S_{a_1} is different then it is called as read-write conflict.

Write-write conflict:

If we interchange 2 write operations such that results are different from both schedules then it is write-write conflict.

Complete schedule S of n ($T_1, \ldots T_n$) concurrent transactions must satisfy:

1. Operations of all transactions should be represented

2. Order of operations in n transactions must be retained

3. Ord If we take any pair of conflicting operations their order should be same in schedule.

Characterization of Schedule:

1. Characterizing schedules based on Recoverability:
   - (a) Recovarable Schedule
   - (b) Non-Recovarable
   - (c) Cascadeless
   - (d) Strict

(a) → Transaction T_j reads from written (b) transaction T_i then T_j is not allowed to commit the operation unit T_i commits the operation. If this condition satisfied then it is Recovarable. Otherwise Non Recovarable

Ex: (c)

T_1	T_2
$r_1(x)$	
$x = x - n$	
$w_1(x)$	
	$r_2(x)$
	$x = x + m$
$r_1(y)$	
$y = y + n$	
$w_1(y)$	
commit	
	$w_2(x)$
	commit

- $S_c = r_1(x); w_1(x); r_2(x); r_1(y); w_1(y); c_1; w_2(x); c_2;$

(c) → Cascading roll back is involved in case of transaction updating a value which was read from other transaction which updated that value and before comitting if it entered into abort state.

T_j is reading some value which was written by T_i is not allowed to read from T_i until it commits.

Cascadeless schedule

T_1	T_2
$r_1(x)$	
$x = x - n$	
$w_1(x)$	
	$r_2(x)$
	$x = x + m$
	$w_2(x)$
$r_1(y)$	
Abort	
↓	

(d) → Read & write operations are strictly prohibited until T_i commits.

2. Characterizing schedules based on Serializabity:

(a) Serial schedule
(b) Non-Serial schedule
(c) Serializable schedule
(d) Result-Equivalence schedule
(e) Conflict-serializable equivalance
(f) View-serializable schedule

fig: NS

T_1	T_2
$r_1(x)$	
$x = x - 0$	
$w_1(x)$	
	$r_2(x)$
	$x = x + m$
	$w_2(x)$
$r_1(y)$	
$y = y + n$	
$w_1(y)$	

$x = 49$ ✓

$y = 45$ ✓

99
41
99
98 x
108 9

$S_{NS} = r_1(x); w_1(x); r_2(x); w_2(x); r_1(y); w_1(y);$

But based on result serializability is not determined

(d) → Two schedules S_0 and S' are called r-e schedules if result produced by transactions are same.

S_1 : $r_1(x)$; $x = x * 1.1$; $w_1(x)$

T_1
$r_1(x)$
$x = x * 1.1$
$w_1(x)$

$x = 100$

T_2
$r_2(x)$
$x = x + 10$
$w_2(x)$

S_2 : $r_2(x)$; $x = x + 10$; $w_2(x)$

Here, S_1 & S_2 are r-e only if $x = 100$ but not r-e if $x \neq 100$

So we ~~not~~ should not conclude by results.

(e) → NS is CS if

Algorithm to check Serializability

Construct precedence graph based on

(a) for each T_i in S create a node

(b) for each pair T_i & T_j if T_j executes write_item after T_i performs read_item(x) create a edge from $T_i \rightarrow T_j$

(c) If T_j performs read after T_i perform write,

Scanned with CamScanner

create edge

(d) If T_j performs write after T_i perform write, create edge

If the graph doesnot contain cycle then it is serializable.

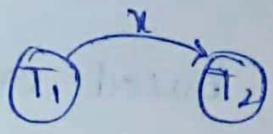
Consider Figs, Fig a and Fig @

T_1	T_2
$r_i(x)$	
$x = x - n$	
$w_i(x)$	
	$r_i(x)$
	$x = x + m$
	$w_i(x)$
$r_i(y)$	
$y = y + n$	
$w_i(x)$	

For x:

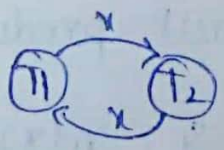
case 1: $T_i = T_1$

$T_j = T_2$

so 

case 2: $T_i = T_2$

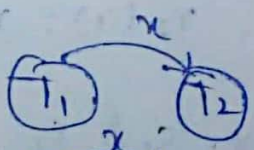
$T_j = T_1$

so 

For y

x

Fig @ → ∴ Non-serializable

for fig (d) case 1: $T_i = T_1$ $T_j = T_2$ 

case 2: $T_i = T_2$ $T_j = T_1$ 

∴ fig d is serializable or conflict-serializable.

(4)

Concurrency Control Techniques in Transaction Processing

Types:

1. Lock Based Concurrency control Technique
   - (a) Binary locking scheme
   - (b) Read/Write or shared/exclusive locking scheme
   - (c) Two phase locking scheme (2PL)
     - i, Basic 2PL
     - ii, conservation
     - iii, Strict
     - iv, Rigorous

(a) this will exhibit limited concurrency

(b) may not exhibit serializability

(c) Always exhibit serializability

But all these suffer from deadlock & starvation

Lock: A variable which is associated with each database item which will specify that database item is accessed by transaction or not.

(a) lock(x) ← 0 or 1

0 - no transaction is accessing x

1 - already it is accessed by a transaction

There are two states lock() & unlock()

Scanned with CamScanner

→ Binary

@ lock(x): Ti issued a lock(x);

if lock(x) == 0 then

B: lock(x) ← 1

// Ti placed lock

else

begin

Transaction Ti is appended to queue;

Transaction manager (TM) wakes up waiting transactions frequently by going to label B)

end (if)

goto B

end.

Ti issued unlock(x)

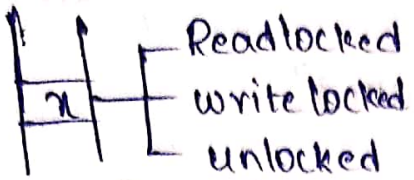
lock (x) ← 0

begin

TM wakes up waiting transaction in queue

end;

Shared / exclusive locking scheme:



Transaction T

T issues Read_lock to DB item

```
begin
B: if lock(x) = "unlocked" then
       lock(x) ← "Read_locked"
       No.of reads := 1;
   else if lock(x) = "Read_locked" then
       No.of reads(x) ← No.of reads + 1;
   else (if lock(x) = "Write_locked")
       wait (until lock(x) = unlocked)
       TM wakes up waiting transaction
   Goto B;
end
```

T issues write_lock

```
B: if lock(x) = "unlocked" then
       lock(x) ← "writelocked"
   else
       wait (until lock(x) = "unlocked")
       TM wakes up waiting transaction
       goto B;
```

T issues unlock:

```
B: if lock(x) = "write-locked" then
     lock(x) ← "unlocked"
   else if lock(x) = "read-locked" then
     if No.of reads = = 0 then
       lock(x) ← "unlocked"
     else
       No.of reads ← No.of reads - 1
   goto B;
   end;
```

$T_1 \leftarrow x := x+y$ $x = 20$

$T_2 \leftarrow y := x+y$ $y = 30$

T_1	T_2
Readlock(y)	Readlock(x)
read item(y)	Read item(x)
unlock(y)	unlock(x)
writelock(x)	writelock(y)
readitem(x)	readitem(y)
x := x+y	y = x+y
write item(x)	write item(y)
unlock(x)	unlock(y)

T_1 followed by T_2 ($T_1 \rightarrow T_2$)

$x = 50$

$y = 80$

T_2 followed T_1 ($T_2 \rightarrow T_1$)

$x = 70$

$y = 50$

Interleaved fashion

T_1	T_2
R_l(y)	
R_i(y)	
un(y)	
	R_l(x)
	R_i(x)
	un(x)
w_l(x)	
r_i(x)	
x = x+y	
w_i(x)	
un(x)	
	w_l(y)
	r_i(y)
	y = x+y
	w_i(y)
	un(y)

After execution:

$x = 50$

$y = 50$

This scheme does not obey serializability always.

Two phase locking scheme (2PL) or variants

ⓐ Basic 2PL

ⓑ conservative 2PL

ⓒ strict 2PL

ⓓ Rigorous 2PL

2PL always ensures serializability.

Two phases — 1. Drowning - can acquire new locks
2. Shrinking - can release locks

ⓐ → All lock operations will precede the first unlock operation

	T_1	T_2
Drowing phase	R_L(y)	R_L(x)
	R_i(y)	R_i(x)
	W_L(x)	W_L(y)
shrinking phase	unlock(y)	unlock(x)
	r_i(x)	R_i(y)
	x = x+y	y = x+y
	W_i(x)	W_i(y)
	un(x)	un(y)

→

T_1	T_2
R_L(y)	
R_i(y)	
W_L(x)	
	W_L(y)
unlock(y)	
R_I(x)	
	R_I(y)
	R_L(x)
x = x+y	
Wi(x)	
un(x)	
	R_i(x)
	y = x+y
	W_i(y)
	un(y)

After execution it satisfies serializability

(b) → Apply all locks on desired dB, items. If not able lock then abort that transaction & restarted later.

T_1	T_2
R_l(y)	
W_l(x)	
R_i(y)	
un(y)	
R_i(x)	
x=x+y	
w_i(x)	
un(x)	

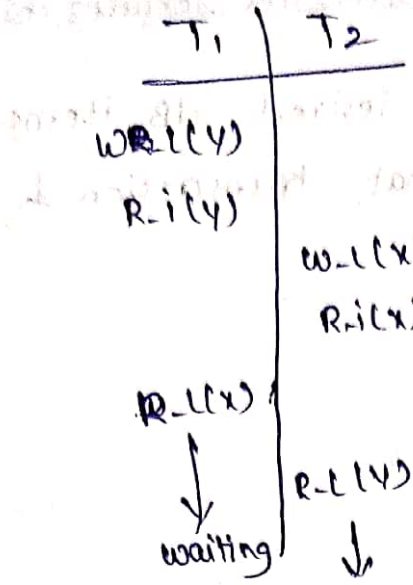
Drawback: limited concurrency

Strict 2PL : (locks are applied only) After commit only write_locks on dB items are unlocked

Rigorous 2PL: Until commit is issued any lock (read/write) unlock can't be done

Problems with lock based concurrency control techniques:

1. Deadlock - Deadlock detection - is by construction of precedence graph.
2. Starvation.



T1 & T2 are in deadlock situation

Deadlock prevention protocols

1. wait-die approach → based on transaction timestamp value.

Time stamp-unique identifier which holds the time at which transaction is stated

submission is at a time but TS value differ in nanoseconds

Ts(t3) → value1
Ts(t2) → value2
Ts(t1) → value3

If Ti & Tj are in deadlock situation and if Ts(Ti) > Ts(Tj) then Tj is youngee and Ti is allowed to wait then abort Tj.

2. wound-wait approach:

If $Ts(Ti) > Ts(Tj)$ then Ti is older

Ti wounds Tj and Ti will allow to wait

Ti wounds Tj → forcefully suspended and by resubmitting with new Ts.

3. No-waiting: If any transaction is waiting, without checking that the another transaction

4. Cautitious waiting: if Ti wants a lock on dB item currently another Tj holds a lock then Ti will wait until Tj unlocks and commits.

(2) → It is a problem which may encounter if there is indefinite waiting

Starvation problem is solved by FCFS.

Time stamp based Concurrency control techniques:

(Time stamp value, Ti's are allowed to access)

Time stamp value, each Ti associates Ts(Ti)

Time stamp ordering is prepared in which Ti's are allowed to access items by satisfying serializability.

$T_1 \rightarrow x$ oly

$T_2 - y \leftarrow m$

$T_3 \leftarrow x \leftarrow p$

$T_4 - y \leftarrow q$

T_1 T_2 T_3 T_4 starts at a time

Read_TS(T) - transaction which has read recently that transactions timestamp

Write_TS(T) - transaction which has updated its value recently that transactions timestamp

Based on Read_TS & write_TS there are 3 protocols

1. Basic TO (Time stamp ordering)
2. Strict TO
3. Thomas write rule

① → If a transaction T_i issues write_item(x)

@ If read_ts(x) $\geq$ ts(t_i) or

write_ts(x) $\geq$ Ts(t_i) then

T_i aborte and restart with new time stamp.

ⓑ If rule @ fails T_i will issues write_item

If a transaction T_i issues read_item(x)

@ If write_ts(x) > ts(t_i) then

t_i abort and restarted with new stamp

ⓑ If rule @ fails T_i will issue read_item

② → If a transaction T_i issues read item(x) or write item

@ If $ts(t) > write\_ts(x)$

Then t is youngee transaction t is delayed until write_ts(x) is commited.

③ → If a T issues write item(x)

@ If $write\_ts(x) > Ts(t)$

then abort T_i

$read\_item(x) > Ts(t)$

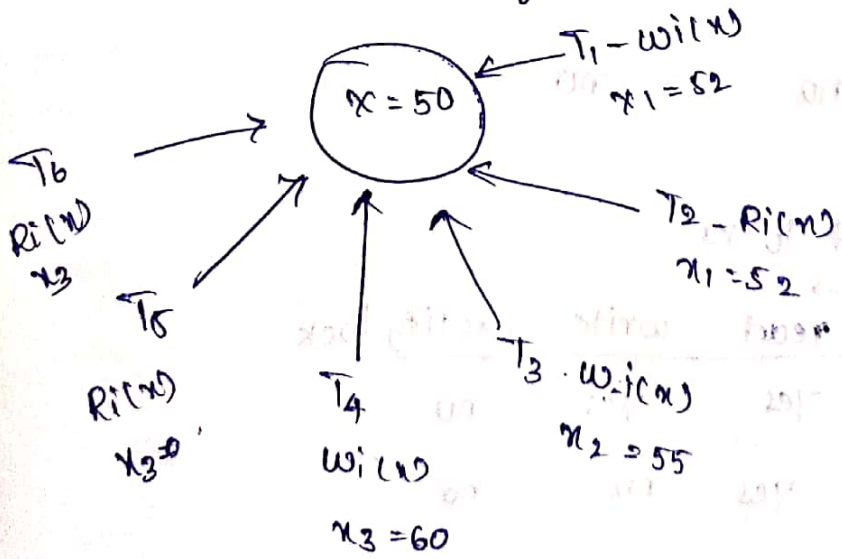
T_i is allowed to read.

(b) If rule @ fails T_i will issue write

Multiversion based concurrency control technique:

If x is DB item, x is available in $x_1, x_2 \dots x_n$

after transaction utilizes



Versions

$x = 50$

$x_1 = 52$

$x_2 = 55$

$x_3 = 50$

} Version

Read_ts(x_i) — $Ts(T_6)$

Write_ts(x_i) — $Ts(T_4)$

① Multiversion concurrency control based on timestamp

② Multiversion concurrency control based on usage of certified lock.

① → If T_i issues $w_i(x)$ and version i of x has highest write_ts(x_i) also are equal to or less than TS(T), read_ts(x_i) is greather than TS(t) then abort T and submit later.

If T issues read$_i$(x) and version i of x has highest write_ts(x_i) then return value of x_i to T

② → Lock compatibility table for shared/exclusive locking schem

T_i ↓ / T_j →	Read	write
Read	yes	no
write	no	no

for certified locks

T_i ↓ / T_j →	read	write	certify lock
read	yes	yes	no
write	yes	no	no
certify	no	no	no

Validation based concurrency control Technique:

or optimistic CC7

3 phase

(a) Read Phase
(b) Validation phase
(c) write phase

(a) → allows to read dB items as many as they want

(b) → If transaction t_i checks that for each T_j i.e either committed or in its validation phase doing concurrent execution following one of the condition must hold

1) T_j completes write phase before T_i starts read phase
2) T_i starts write phase after T_j completes write phase
3) Read set & write set of T_i & T_j have no common data items