Jatin

Concurrency Control

- Two Phase locking
- Snapshot isolation (Time Stamp Based Protocol)

Lock Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner. The most common method used to implement this requirement is to allow transaction to access a data item only if it is currently holding a lock on that item.

Locks (Binary Locks).

There are various modes in which data item may be locked.

→ Shared mode lock (S), T_i can read but cannot write.

→ Exclusive mode Lock (X): T_i can both Read & write.

	S	X
S	Tove	False
X	fals	false

Lock Compatibility Matrix

T_1 :
```
lock-X(B)
R(B);
B := B - 50;
W(B);
Unlock(B)
Lock-X(A);
R(A)
A := A + 50;
W(A)
Unlock(A);
```

T_2 :
```
lock-S(A)
R(A);
Unlock(A);
lock-S(B);
R(B);
unlock(B)
display(A+B);
```

display sum of A+B

Transfer Balance

Now if executed serially T_1, T_2, or T_2, T_1 the sum display is consistent. But if executed concurrently the according following Schedule 1.

A = 100
B = 200

T_1	T_2	
lock-X(B)		grant-X(B, T_1)
R(B)		
B := B-50;		
W(B)		
Unlock(B)		
	lock-S(A)	grant-S(A, T_2)
	R(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	R(B)	
	unlock(B)	
	display(A+B)	= 250
lock-X(A)		
		grant-X(A, T_1)
R(A),		
A := A + 50		
W(A)		
unlock(A)		

now this Schedule is giving incorrect sum of A+B which occurs due to early release of lock and this Operation can be delayed. as per the modified Transaction T_1'

```
lock-X(B);
R(B);
B:=B-50;
W(B)
lock-X(A);
R(A);
A:=A+50;
W(A);
unlock(B);
unlock(A);
```

Problems

inconsistent, Deadlock
Not guaranteed Serializability

Now T_2 will print correct result of A+B, because unlocking is delayed.

Deadlock

T_3	T_4
lock-X(B)	
R(B)	
B:=B-50	
W(B)	
	lock-S(A)
	R(A)
	lock-S(B)
lock-X(A)	

Consider the partial Schedule consisting of T_3, T_4 T_4 is requesting Shared lock on B which is held by T_3. T_3 is requesting exclusive lock on A which is held by T_4 and causes Deadlock. It can overcome by rollback.

It is desirable over inconsistent state of DB. If locking don't use or unlocking is done earlier, then DB leads to inconsistent state.

Locking Protocol

It indicates when a transaction may lock and u each of the data items. locking protocol restricts the number of possible schedules.

Granting of Locks

When data item is not locked by any transaction, the locks can be granted. However care must be taken while granting Locks. Consider a scenario, T_2 has shared mode lock on B, & T_1 is requesting exclusive mode lock on B. T_1 has to wait meanwhile T_3 may request shared lock on B and since it is compatible, therefore it is granted. Now T_1 has to further wait for T_3. This leads to starvation. It can be prevented by following ways: -

(1) There is no other transaction holding a lock on Q in a mode that conflicts with Mode

(2) There is no other transaction that is waiting for a block on data item and that mode its lock request before T_1.

Two Phase Locking Protocol

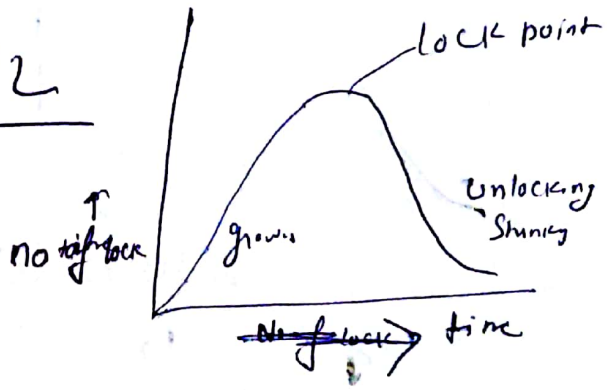
- ensures serializability by requiring that each transaction issue lock and unlock request in two phase.

Growing phase : Transaction may obtain locks, but may not release any lock.

Shrinking phase : A transaction may release locks, but may not obtain any new locks.

```
T1 : lock-x(B);
     R(B);
     B := B - 50;
     W(B);
     lock-x(A);
     R(A);
     A := A + 50;
     W(A);
     unlock(B);
     unlock(A);
```

2 PL



The point in the schedule when the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction. Now if transaction are ordered according to their lock points, then it ensures serializability for the transaction.

T_1	T_2
Lx(A)	
w(A)	w(A)
	w(B)
L-x(B)	
w(B)	

T_1	T_2	T_3
~~lock s(A)~~	L-S(A)	L S(A)
	L-X(B)	
	U(A)	L X(C)
	U(B)	
L-S(B)		
L-X(A)		
U(A)		
U(B)		

locking point

T_2 T_3 T_1

→ Two phase Locking does not ensure freedom from deadlock.

T_3	T_4
L- X(B)	
R(B)	
B = B-50	
W(B)	
	Lock-S(A)
	R(A)
	L-S(B)
L-X(A)	

Growing phase

Deadlocked.

→ Cascading Rollback may occur under two phase locking. but it is recoverable.

T_1	T_2	T_3
L-X(A)		
R(A)		
L-S(B)		
R(B)		
W(A)		
UnLock(A)		
	Low-X(A)	
	F(A)	
	W(A)	
	U(A)	
		L-S(A)
		R(A)

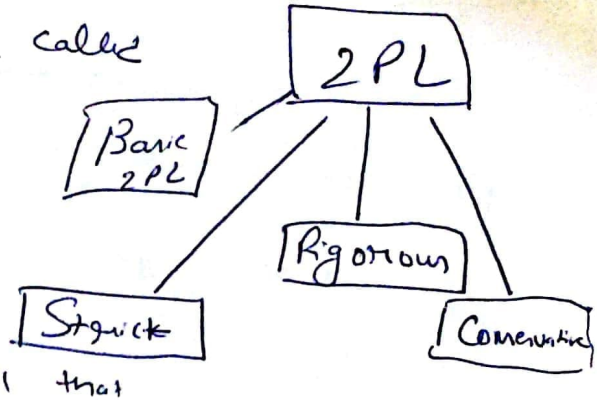
failure

Now failure of T_1 may lead to the Cascading Rollback of T_2 and T_3 as well.

2PL

Cascading Rollback can be avoided by a modification of 2PL called Strict two phase Locking.

This protocol follows 2PL as well as all exclusive-mode locks taken by a transaction be held until that transaction commits, preventing any other transaction reading that data.



, deadlock may occur, Avoid cascading Rollback.

Rigorous 2PL

It requires that all locks be held until the transaction commits. Transaction can be serialized in the order in which they commit. Avoid cascading Rollback, deadlock may occur

Conservative

It requires a transaction to lock all the items it accesses before the transaction begins execution, by predeclaring its read-set and write-set. Read-set is the set of all items that the transaction reads and similarly write-set is the set of item that ~~all~~ ~~the~~ it writes. If any of the predeclared item is not available for locking, then transaction does not hold any lock, instead it wait till all item are available. It is not practically possible. Resource utilization is less.

No cascading Rollback, deadlock free

Lock conversions

It is refinement over basic 2PL, in which lock conversions are allowed. May be upgrade by conversion from shared to exclusive, downgrade by conversion from exclusive to shared. upgrading allowed only in growing phase and downgrading allowed only in shrinking phase.

T_8	T_9
L-S(a_1)	
	L-S(a_1)
L-S(a_2)	
	to S(a_2)
L-S(a_3)	
L-S(a_4)	
	unL(a_1)
	unL(a_2)
L-S(a_n)	
upgrade(a_1)	

It also ensures conflict serializability schedules, transactions can be made serializable by their Lock points.

- All the locks held by transaction are released after commit / abort

→ R(Q) issues L-S(Q) followed by R(Q)

→ W(Q) issues L-X(Q) followed by W(Q), if lock already taken by Transaction on Q as Shared lock then upgrade(Q) is used.

Deadlock

A System is in deadlock state if every transaction in the system is waiting for another transaction in the Set.

$(T_0\ T_1 \ldots T_n)$ T_0 is waiting for data item held by T_1, and T_1 is waiting for T_2 and T_n is waiting for T_0.

One solution is to rollback some transaction fully or Partial.

Two methods for dealing with deadlock are

-1 Deadlock Prevention

-2 Deadlock Detection / Deadlock Recovery.

1 **Deadlock Prevention**

No Cyclic Wait

→ use total ordering of data items along with 2PL

→ Locks must be requested in the right order

Acquire all the locks together

→ It is hard to predict, what item to be locked before execution

→ Utilization is very low

Preemption and Rollbacks

If T_j request a resource held by T_i, the lock granted to T_j by preemption and rolling back of T_i. To control preemption, a unique timestamp is assigned to each transaction when it begins. This timestamp is used to decide whether a transaction should wait or rollback

Two deadlock prevention schemes using timestamp have been proposed

(i) Wait-die

It is non-preemptive technique. When T_i requests a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j, otherwise T_i is rolled back.

for eg T_1, T_2, T_3
TS —: 5 10 15
old new

If T_1 requ data item held by T_2, then T_1 will wait

If T_3 request data item held by T_2, then T_3 will be roll back.

(ii) Woond-wait

It is preemptive technique, counterpart of wait-die. When T_i request data item held by T_j, T_i is allowed to wait only if it has a timestamp larger than that of T_j (T_i is young than T_j), otherwise T_j is rollback.

for eg T_1 T_2 T_3
TS:- 5: 10 15

If T_1 request data item held by T_2 then data item will be preempted from T_2 & T_2 will be roll back. [If T_3 request from T_2 then, T_3 will wait.]

Problems with these schemes

→ unnecessary Rollbacks

Lock Timeout Technique

In this scheme, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to be time out, and rolls itself back and restarts. If there was deadlock, one or more transaction stucked in deadlock will time out & roll back and allowing the others to proceed.

→ Easy to implement
→ Work well with Short transactions
→ Hard to decide how long a transaction must wait before time out.
→ Long wait is unnecessary once a deadlock occurs, & short wait results in rollback even there is no deadlock.
→ Resource wastage
→ Starvation.

Deadlock Detection & Recovery

If prevention is not used, then detection & recovery scheme must be used. It is invoked periodically to check for deadlock. If deadlock occur, then system must attempt to recover from deadlock.

→ Maintain information about current allocation, outstanding requests.

→ Use detection technique that uses this information

→ Recover from the deadlock.

Detection Algorithm

Wait-for-Graph

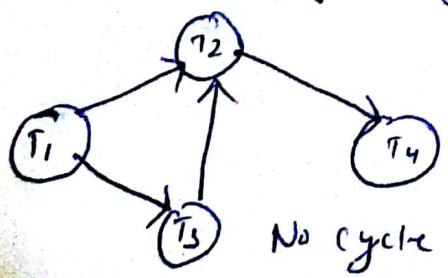
$G = (V, E)$

V: Set of Transaction

E: Set of ordered pair $T_i \rightarrow T_j$: T_i is waiting for T_j to release data item held by T_j.

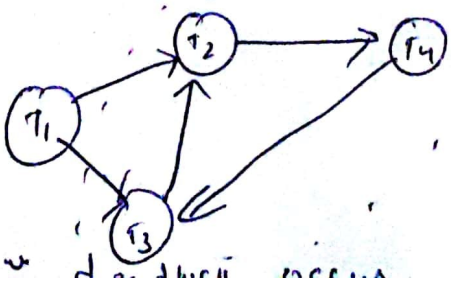
Edge is removed when data item is released by T_j.

If cycle exist, then deadlock occurs. System maintain WFG, and periodically invoke an algorithm that searches for a cycle in the graph.



No cycle

Now T_4 requests T_3



Now

Challenges of Detection Algo

→ How often does a deadlock occur

→ How many transaction will be effected by deadlock

Recovery from Deadlock

1 Selection of victim

We must determine which transaction to roll back to break the deadlock. Rollback only those transaction which incur minimum cost.

→ How long transaction has computed

→ How many data items has used

→ how many data items needed

→ How many transaction will be involved in the Rollback.

2. Roll back

↳ once victim is decided. Simple solution is total roll back. Abort the transaction and then restart it.

Partial Rollback :→ Undo till the lock causing deadlock.

3. Starvation : It may happen that same transaction becomes victim based on cost factor. So transaction must be picked up as victim only a finite number of times.

me Stamp Based Protocols (Deadlock Preventive Protocols) ⑧

Earlier schemes utilize scheme of ordering transaction based every conflicting pair of transactions at execution time. This technique utilizes an ordering among transactions based on timestamp-ordering.

Associate each transaction T_i with a unique fixed timestamp denoted by $TS(T_i)$.

If $TS(T_i) < TS(T_j)$ then T_j is new.

Timestamp may be implemented by two methods

- ↳ System Clock: Timestamp value is equal to value of clock
- ↳ Logical Counter: A counter value is incremented after a new timestamp has been assigned.

The timestamps of transaction determines the serializability order. Thus if, $TS(T_i) < TS(T_j)$ then the system must ensure that the produced schedule is equivalent to a serial schedule.

Assign Two TS values with each data item Q:

W-timestamp(Q): largest timestamp of any transaction that executed write(Q) successfully

R-timestamp(Q): largest timestamp of any transaction that executed Read(Q) successfully.

Timestamp Ordering Protocol

It ensures that any conflicting read and write operations are executed in timestamp order. It follows the following operations.

T_i issues Read(Q)

(a) If $TS(T_i) < W\text{-timestamp}(Q)$ then T_i needs to read a value of Q that was already overwritten, hence Read operation is rejected and T_i is rolled back

(b) If $TS(T_i) \geq W\text{-timestamp}$, then read is executed and R-timestamp is set to maximum of R-timestamp(Q) and $TS(T_i)$.

T_i issues Write(Q)

(a) If $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously and the system assumed that value would never be produced. Hence system rejects the write operation & rollback.

(b) If $TS(T_i) < W\text{-timestamp}(Q)$ then T_i is attempting to write an obsolete value of Q, hence system rejects write operation and rollback.

(c) Otherwise, System executes the write operation & sets W-timestamp(Q) to $TS(T_i)$

T_1

R(B)
R(A)
Display (A+B)

T_2

R(B);
B := B − 50;
w(B);
R(A);
A := A + 50;
w(A)
display (A+B);

T_1	T_2
R(B)	
	R(B)
	B : B − 50
	w (B)
R(A)	
	R(A)
display (A+B)	
	A : A + 50
	w(A)
	display (A+B)

This protocol can generate schedules that are not recoverable. However it can be extended to make the schedules recoverable in one of several ways.

→ Recovery / cascadelessness can be ensured by performing all writes together at the end of the transaction.

→ Recovery / cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commit.

→

Thomas Write Rule

T_1	T_2
R(Q)	
	W(Q)
W(Q)	

$TS(T_1) < TS(T_2)$

R(Q) of T_1 succeed and W(Q) of T_2 succeed.

When W(Q) of T_1 attempts to operate, then

$TS(T_1) <$ W-timestamp(Q) Thus it is rejected and T_1 is rolled back.

Although T_2 has already wrote Q, and roll back of T_1 is unneccesary and the value T_1 is going to write in Q is never Read. So any transaction T_i whose $TS(T_i) < TS(T_2)$ that attempts to Read (Q) will be rolled back.

And any transaction T_j whose $TS(T_j) > TS(T_2)$ must Read the value of Q.

So obsolete write operation can be ignored under certain circumstances - only 2nd rule of write is modified.

If $TS(T_i) <$ W-timestamp(Q) then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.