25th Sept.

Page 42

CHAPTER 10

Transaction Processing and Concurrency Control

Transaction System

What is transaction?

→ group of operations that form a single logical unit of work is called transaction.

→ a transaction is a logical unit of work that must be either entirely completed or aborted.

→ a transaction is a unit of program execution that access and possibly updates various data items.

→ a transaction is the DBMS abstract view of a user program that consists of sequence of reads and writes.

→ transaction access data using two operations
   i) read (X)
   ii) write (X)

→ Transaction processing is information processing that is derived into individual, indivisible operations called transactions.

→ each transaction must succeed or fail as a complete unit but can't remain in intermediate state.

→ Transaction processing maintains a system in as consistent state.

e.g.: T1 : read (A)
A = A - 100
write (A)
read (B)
B = B + 100
write (B)

Imp.

Properties of Transaction (ACID properties)

1) **Atomicity** - either all operations of transaction are reflected properly in database or none are.

2) **Consistency**: execution of a transaction in isolation preserves the consistency of db.

3) **Isolation**: even though multiple transactions may execute concurrently, the system guarantees that for every pair of transactions T_i & T_j for T_i either T_j finished execution before T_i started or T_j started execution after finished.

4) **Durability** - after successful completion of transaction, the changes in db persist, even if there are system failures.

Transaction State:

i) Active State:
- it is the initial state.
- transaction stays in this state while it is executing.

ii) Partially Committed:
- transaction is in P. C state after the final statement has been executed.

iii) failed:
- transaction is in failed state after normal execution can no longer proceed.

iv) Aborted:
- transaction is in aborted state after it has been rolled back & the ~~bo~~ database has been restored to its state prior to the start of transaction.

v) Committed:
- transaction is in committed state after successful completion.

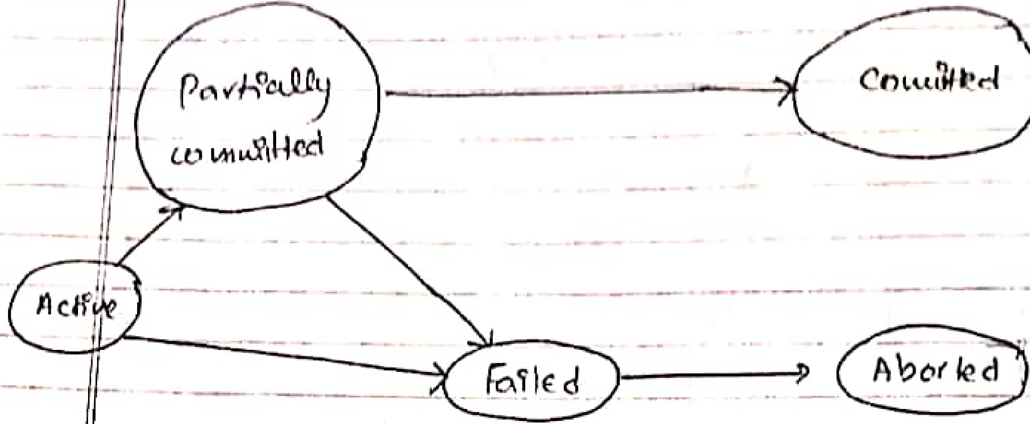


fig: state diagram of transaction.

→ a transaction has committed only if it enter committed state

↳ a transaction has aborted only if it enter aborted state.

↳ a transaction has terminated if either committed or aborted.

→ after transaction being in aborted state, system has two options either it can restart transaction.

→ Transaction processing system usually allow multiple transactions to run concurrently.

→ This cause several complications with consistency of data.

↳ though, transactions can be run serially to ensure consistency, concurrent execution of transaction has following advantages :

i) Improved throughput and resource utilization.

ii) Reduced waiting time.

→ database system control the interaction among the concurrent transactions to prevent from destroying consistency of db through variety of mechanisms called concurrency control scheme.

→ When transactions are executing concurrently in an ~~unbalanced~~ interleaved fashion, then the ~~act~~ order of execution of operations from various transactions is known as schedule.

↳ A schedule S specifies the chronological order in which the operations of concurrent transactions are executed.

↳ e.g.: Let Transaction T1 transfer Rs 100 from account A to B.

Transaction T2 transfer 20 % of balance from A to B.

let A = 1000, B = 1000

Now,

T1 :- read(A)

A = A - 1000

write(A)

read(B)

B = B + 100

write(B)

T2 : read (A)
temp = A * 0.2
A = A - temp
write (A)
read (B)
B = B + temp
write (B)

Serial Schedule .

i)

T1 .	T2
read (A)	
A = A - 100	
write (A)	
read (B)	
B = B + 100	
write (B)	
	read (A)
	temp = A * 0.2
	A = A - temp
	write (A)
	read (B)
	B = B + temp
	write (B)

ii)

T1	T2
	read (A)
	temp = A * 0.2
	A = A - temp
	write (A)
	read (B)
	B = B + temp
	write (B)
read(A)	
A = A - 100	
write (A)	
read (B)	
B = B + 100	
write (B)	

Non - Serial Schedule

i)

T1	T2
read (A)	
A = A - 100	
write (A)	
	read (A)
	temp = A × 0.2
	A = A - temp
	write (A)
read (B)	
B = B + 100	
write (B)	
	read (B)
	B = B + temp
	write (B)

ii)

T1	T2
read (A)	
A = A - 100	
	read (A) → 1000
	temp = A × 0.2 → 200
	A = A - temp → 800
	write (A) → 800
	read (B) → 1000
write (A) → 800	
read (B) → 1000	
B = B + 100 → 1100	
write (B) → 1100	B = B + temp

here, both (i) & (ii) are non serial schedule.

i) results correct state

but

ii) doesn't result correct state.

Serializability.

→ Basic assumption in each transaction preserves data consistency.

→ Thus serial execution of a set of transaction preserves database consistency.

→ Main objective of serializability is to search non serial schedules that allow transaction to execute concurrently without interfering one another transaction and produce the result of db state that could be produced by serial execution.

→ We can conclude that a non serial schedule is correct if it produce same result as serial execution.

→ a schedule is serializable if it is equivalent to a serial schedule.

Rules:

→ Ordering of read/write is important

→ if two transactions only read data item they do not conflict and order is not important.

→ if two transactions either read or

write completely separate data items, they do not conflict and order is not important.

→ if one transaction write a data item and another read or write same data item, order of execution is important

Schedule 1			Schedule 2		
T1	T2	T3	T1	T2	T3
R(a)			R(a)		
	R(b)		W(a)		
		R(c)		R(b)	
W(a)				W(b)	
	W(b)				R(c)
		W(c)			W(c)

→ Here, actions of transactions in schedule 1 are not executed as same as in schedule 2 but at the end schedule 1 gives same result as that of schedule 2.

→ Thus it is considered as serializable.

Impor

Conflict serializability.

→ Instructions I_i and I_j of transactions T_i and T_j respectively conflict if and only if there exists same item Q accessed by both I_i & I_j and at least one of these instruction wrote 'Q'.

→ If a schedule S can be transformed into a schedule 'S'' by a series of swaps of non conflicting instructions, we say that S and S' are conflict equivalent.

→ we say that a schedule S is conflict serializable if it is conflict equiv. to a serial schedule.

→

Instruction I_i	Instruction I_j	Result.
Read (Q)	Read (Q)	No conflict
Read (Q)	Write (Q)	Conflict
Write (Q)	Read (Q)	"
Write (Q)	Write (Q)	"

Eg. 1) let we have schedule A as.

T1	T2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

and schedule B as,

T1	T2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

→ Here, schedule A can be transformed into serial schedule B by series of swaps of non conflict instructions.

→ Hence, schedule A is conflict serializable.

eg.2) Let we have,

Schedule X		Schedule Y	
T1	T2	T1	T2
read (Q)		read (Q)	
	write (Q)	write (Q)	
write (Q)			write (Q)

Here, we can't transform schedule X into schedule Y by swapping non conflict instruction.

∴ Schedule X is non conflict serializable.

Impo

View serializability:

→ two schedules are said to be view equivalent if following 3 condition holds.

1) For each data item Q, if transaction T_i reads initial value of Q in schedule S, also read initial value of Q

2) For each Q, if T_i executes read (Q) in S and if value was produced by write (Q) of T_j then read (Q) of T_i must in S' also read value of Q produced by same write (Q) of T_j.

3) For each Q, the transaction that performs the final write (Q) operation in S, must perform final write (Q) operation in S'.

→ a schedule is view serializable if it is view ~~e~~ equiv. to a serial schedule.

eg: let,

Schedule A

T_1	T_2	T_3
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

Schedule B

T_1	T_2	T_3
read (Q)		
write (Q)		
	write (Q)	
		write (Q)

→ here, schedule A is view serializable ... because it is view equivalent to serial schedule B.

→ here write (Q) of ~~the~~ T2 and T3 are called blind writes because it is performed without having performed a read (Q).

→ every conflict serializable schedule is also view serializable but not vice versa.

Testing for serializability

- construct a directed graph called precedence graph from S.
- precedence graph consists of a pair $G = (V, E)$ where

V is set of vertices, it consists of all transactions in schedule.

E is set of edges.

set of edges consists of all edges $T_i \to T_j$ for which ~~one~~ of 3 cond's hold.

①

②

③

$G = (V, E)$

15

$T_i \to T_j$..

2) → If an edge $T_i \rightarrow T_j$ exists in precedence graph then in any serial schedule S' equivalent to S, T_i must appear before T_j.

3) → If the precedence graph has a cycle, then the schedule is not conflict serializable.

→ If the precedence graph has not a cycle then schedule is conflict seri

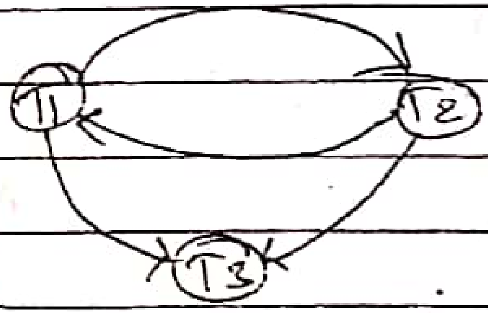
→

eg.

eg. 1)

T1	T2	T3
read (A)		
	write (A)	
write (A)		write (A)

Now,

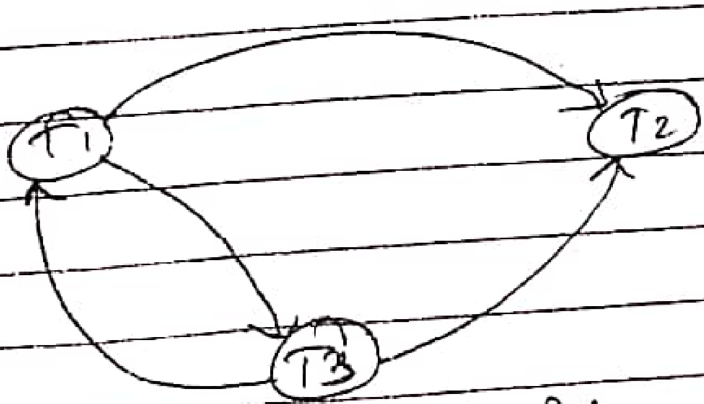


→

→ here graph contains a cycle, so above schedule is non conflict serializable.

eg. 2)

T1	T2	T3
write (A)		
	read (A)	
		write (B)
write (B)		
		write (B)
	write (A)	
		read (B)
	read (B)	



equivalent P.G.

- Here also, above schedule is non conflict serializable.

Test for following.

1. $r_1(x), w_2(x), r_1(x)$.
2. $r_1(A), r_3(A), w_1(A), r_2(A), w_3(A)$
3. $r_3(A), r_2(A), w_3(A), w_1(A), w_1(A)$
4. $r_1(A), w_3(A), w_3(A), w_1(A), r_2(A)$.

Imp

Concurrency ~~symbol~~ control.

→ ensures that database transactions are performed concurrently without violating data integrity of database.

→ different concurrency control schemes can be used to ensure the isolation property when multiple transactions are executed in parallel.

→ DBMS must guarentee that only serializable schedule are generated and that no effect of commited transaction is lost and no effect of aborted transaction remains in db. database.

→ Different methods like locking method, timestamp methods, etc. are used.

Lock based protocol

↳ a lock is a mechanism to control concurrent access to data item.

↳ Data items can be locked in two modes.

a) Exclusive mode (X)

→ Data item can be both read as well as written.

→ X-lock is requested using lock-X instruction

b) Shared mode (S)

→ Data item can only be read.

→ S-lock is requested using lock-S instruction.

→ lock requests are made to concurrency control manager.

→ Transaction can only be proceed once request is granted.

→ A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on item by other transactions.

	S	X
S	True	False
X	False	False

fig: lock compatibility matrix.

- here, shared mode is compatible only with shared mode.
- if any transaction holds an exclusive lock on item, no other transaction may hold any lock on the item.
→ If a lock can't be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released the lock is then granted.
- A transaction can unlock a data item Q by instruction unlock (Q).
- transaction must hold a lock on data item as long as it access item.

e.g:

T : lock_s (A)
read (A)
unlock (A)
lock - S (B)
read (B)
unlock (B)
display (A+B)

- A locking protocol is set of rules followed by all transactions while requesting and releasing locks.
- locking protocol restricts the set of possible schedules.
- locking protocol must ensure serializability

e.g.:

let, value of account A and B are 100 and 100 respectively.

T1 transaction transfer 20 from A to B.

T2 transaction display sum of A and B

Now, here,

T1	T2
lock – X(A)	lock – S(A)
read (A)	read (A)
A = A – 20	unlock (A)
write (A)	lock – S(B)
unlock (A)	read (B)
lock – X(B)	unlock (B)
read (B)	display (A+B)
B = B + 20	
write (B)	
unlock – X(B)	

→ If T1 and T2 are executed serially, we get correct result.

→ But concurrent execution of T1 and T2 may result incorrect value.

like

T1	T2	Concurrency Control Manager
lock – X(A)		grant – X(A, T1) ~~grant~~
read (A)		
A = A – 20		
write (A) //update		
unlock (A)		

21

T1	T2	Concurrency Control Manager
	lock - S(A)	
		grant - S(A, T2)
	read (A)	
	unlock (A)	
	lock - S(B)	
		grant - S (B, T2)
	read (B)	
	unlock (B)	
	display (A+B)	
lock - X(B)		
		grant - X (B, T1)
read (B)		
B = B + 20		
write (B)		
unlock (B)		

fig: Schedule X.

→ Assume that unlocking is delayed at the end of transaction. then ~~T1 becomes~~ .

T1 becomes

T3 : lock - X(A)
read (A)
A = A - 20
write (A)
lock - X(B)
read (B)
B = B + 20

T2 becomes

T4 : lock - S(A)
read (A)
lock - S(B)
read (B)
display (A+B)
unlock (A)
unlock (B)

write (B)
unlock (A)
unlock (B)

→ with T3 and T4, we can't get schedule X as above and for any other schedule, it produce correct value like :

T1	T2	T3	T4
lock- X(A)			~~read (B)~~
read (A)			~~display (A+B)~~
A = A − 20			~~unlock (A)~~
write (A)			~~unlock (B)~~
lock − X(B)			
read (B)			
B = B + 20			
write (B)			
unlock (A)			
	lock − S(A)		
	read (A)		
unlock (B)			
	lock − S(B)		
	read (B)		
	display (A+B)		
	unlock (A)		
	unlock (B)		

Pitfalls of lock based protocol

- consider partial schedule as

T3	T4
lock - X(B)	
read (B)	
B = B - 50	
write (B)	
	lock - S(A)
	read (A)
	lock - S(B)
lock - X(A)	

fig: Scheduler

- here, neither T3 nor T4 can make progress executing lock - S(B) cause T4 to wait for T3 to release its lock on B.
while executing lock - X(A) cause T3 to wait for T4 to release its lock on A.
- Such situation is called deadlock.
- to handle deadlock one of transactions T3 or T4 must be rolled back and its lock must be released.
- if we do not use locking, we may get inconsistent state.
- Similarly, if we do not unlock data items before requesting lock on another item,

deadlock occurs.

- However, deadlocks are necessary evil, more preferable than inconsistent states.
- the potential for deadlock exists in most locking protocol.
- Starvation is also possible,
  → if a transaction may be waiting for an X - lock on item while, a sequence of other transactions request S-lock and are ~~guarants~~ granted on same item.
  → Same transaction is repeatedly rolled back due to deadlock.

Two phase locking protocol

→ ensures serializability.

→ here, transaction issue lock and unlock requests in two phases

i) Growing phase :-
  → transaction may obtain locks.
  → transaction may not release locks.

ii) Shrinking phase:
  → transaction may release locks.
  → transaction may not obtain locks.

→ Initially, transaction is in growing phase.

→ after transaction release a lock, it enters shrinking phase.

25

e.g. T3 and T4 are two phase but not T1 and T2.

→ two phase locking does not ensure freedom from deadlocks.

→ cascading roll back is possible under two phase locking.

cascading roll back: the phenomenon in which a single transaction failure leads to series of transaction rollbacks is called cascading roll back.

e.g.:

T1	T2	T3
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)

↳ here, T2 is dependent on T1, and T3 is dependent on T2.

↳ if T1 failed, T1 must be rolled back. Similarly, as T2 is dependent on T1, T2 also be rolled back.

again as T3 is dependent on T2, T3 also must be rolled back.

T_a	T_b	T_c
lock – X(A)		
read (A)		
lock – S(B)		
read(B)		
write (A)		
unlock (A)		
	lock – X(A)	
	read (A)	
	write (A)	
	unlock (A)	
		lock – S(A)
		read (A)

fig: partial schedule X under two phase locking

- in above schedule X, if T_a fails after read(A) of T_c then there must be cascading rollback of T_b & T_c.

1) Strict two phase locking protocol

→ here, transaction must hold all its exclusive mode locks till it commits.

→ this prevents any other transaction from reading data.

→ No cascading rollback.

2) Rigorous two phase locking

→ here, all locks (S and X) are held till commits.

→ No cascading roll back.

→ here, transaction can be serialized in order in which they commit.

Deadlock Handling

↳ System is deadlocked if there is set of transactions such that every transaction in the set is waiting for another transaction in the set.

↳ let,

T1 : write (X)
write (Y)

T2 : write (Y)
write (X)

↳ here, Schedule with deadlock

T1	T2
lock - X(X)	
write (X)	
	lock - X(Y)
	write (Y)
	write (X)
write (Y)	

↳ to deal with deadlock, we can use

1) Deadlock prevention protocol

- ensure that system will never enter deadlock state.

2) Deadlock detection and recovery scheme
- try to recover system once it entered deadlock state.

Both methods may result in transaction rollback.
→ prevention is used if probability of system entering deadlock state is relatively high.
→ otherwise, detection and recovery are more efficient.

Deadlock prevention

↳ ~~dead~~ deadlock prevention protocol ensure that the system will never enter into deadlock state.

↳ some prevention strategies.

i) require that each transaction locks all data item before it begins execution.

ii) impose partial ordering of all data item.
- require that a transaction can lock data items only in order specified by partial order.

iii) Timeout based schems.
- a transaction waits for a lock only for specified amount of time.

29

- after the wait time is out, transaction is roll back.

→ Simple to implement but starvation is possible.

→ following schemes use transaction timestamp is for deadlock prevention.

i) Wait - die scheme (Non preemptive)

→ older transaction may wait for younger one to release data item.

→ younger transactions never wait for older ones; they are rolled back instead.

→ a transaction may die several times before acquiring needed data item.

2) ~~Deadlo~~ Wound - wait Scheme (pre emptive)

→ older transaction wounds (force rollback) Younger transaction instead of waiting for it.

→ younger transaction may wait for older ones.

→ may be fewer rollbacks than wait die scheme.

↳ in both schemes, rollback transaction is restarted with its original timestamp.

↳ older transaction thus have precedence over newer ones in those schemes and starvation is avoided.

Deadlock Detection and Recovery

- is used if no protocol is used to ensure deadlock freedom.
- some algo are used to check whether deadlock occurs, if deadlock occurs, then must be recovered.

Deadlock detection:

- deadlock can be described as wait for graph which consists of pair G = (V, E)
  → V is set of vertices (transaction)
  → E is set of edges, each edge is ordered pair $T_i \rightarrow T_j$
- when transaction T_i request a data item held by T_j then $T_i \rightarrow T_j$ is inserted in wait for graph.
- this edge is removed only when T_j no longer holding data item needed by T_i.
- the system is in deadlock state if and only if wait for graph has a cycle.
- the system invokes deadlock detection algorithm periodically to look for cycle.

31

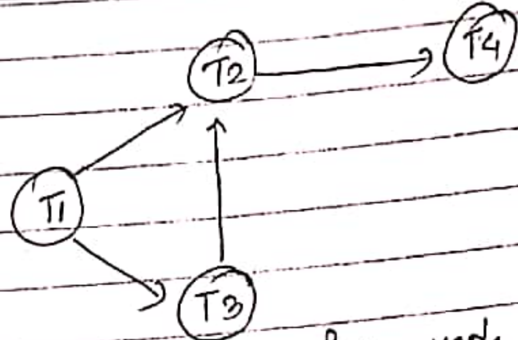


fig: wait for graph without cycle.

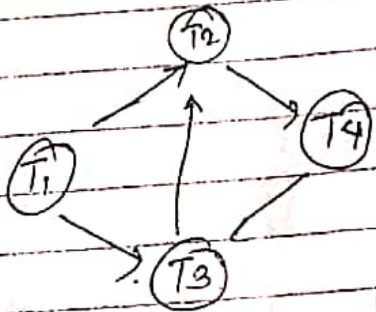


fig: wait for graph with cycle.

Deadlock Recovery

when deadlock is detected

→ Some transactions will have to roll back to break deadlock (i.e. victim selection)

→ Rollback :- determine how far to rollback transaction

a) total rollback
  - abort transaction and then restart it.

b) Partial rollback
  - rollback transaction only as far as necessary to break deadlock.
  - is more effective.

- Starvation happens if same transaction is always chosen as victim.
- must ensure that transaction can be picked as a victim only a small no. of times.