Concurrency Control

The system must control the interaction among the concurrent executing transactions. this control is achieved through one of Concurrency - Control Schemes

Lock-Based protocols:

→ one way to ensure Serializability is to require that data items be accessed in a mutually exclusive manner

↓

on T is accessing a data item, no other T can modify that data item.

→ allow a T to access a data item only if it is currently holding a lock on that item.

Locks:

Data items can be locked in two modes:

① Shared (X) mode: Data item can only be read.

If a Transaction T_i has obtained a shared-mode lock on item Q, then T_i can read, but cannot write, Q.

② Exclusive (X) mode: Data item can be both read as well as written. If a T_i has obtained a exclusive-mode lock on item Q, then T_i can both read and write Q.

→ we require that every transaction request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q.

→ the Transaction makes the request to the Concurrency-control mgr.

→ the T can proceed with the execution only after

→ Compatability function:

A & B two modes of lock

	S	X
S	true	f
X	F	F

Fig: Lock-Compatibility matrix group

→ Ti request a lock of mode A on item on, on which Tj currently holds mode B.

→ If Ti can be granted a lock on immediately, even the presence of the mode B lock, then mode A is Compatable with mode B.

In the above matrix, shared mode is compatible with shared mode.

⇒ Several shared-mode locks can be held simultaneously on a same data item.

⇒ A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

T1 : lock-x(B);
read(B);
B := B-50;
write(B);
unlock(B);
lock-x(A);
read(A);
A := A+50;
write(A);
unlock(A)

T2 : lock-s(A);
read(A);
unlock(A);
lock-s(B);
read(B);
unlock(B);
display(A+B)

T_1	T_2	concurrency-control mgr
lock-x(B)		grant-x(B, T_1)
read(B)		
B := B-50		
write(B)		
unlock(B)		
	lock-s(A)	grant-s(A, T_2)
	read(A)	
	unlock(A)	
	lock-s(B)	grant-s(B, T_2)
	read(B)	
	unlock(B)	
	display(A+B)	
lock-x(A)		grant-x(A, T_1)
read(A)		

To access a data item, T_i must first lock that item. If the data item is already locked by T_j in incompatible mode, the concurrency-control will not grant the lock until all incompatible locks held by other transactions have been released.

→ a 'T' must hold a lock on a data item as long as it accesses that item.

→ for a 'T', to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

A = 100$ B = 200$.

o/p of T_2 = 300$

In the above case T_2 displays 250$ which is incorrect.

T_3	T_4
lock-x(B);	lock-s(A);
read(B);	read(A);
B := B-50;	lock-s(B);
write(B);	read(B);
lock-x(A);	display(A+B);
read(A);	unlock(A);
A := A+50;	unlock(B);
write(A);	
unlock(B);	
unlock(A)	

db is correct with $T_3 \leftarrow T_4$.

→ executing lock-s(B) causes T_4 to wait for T_3 to release its lock on B, while executing lock-x(A) causes T_3 to wait for T_4 to release its lock on A.

→ neither of these transactions can ever proceed with its normal execution.

This situation is called a deadlock.

T_3	T_4
lock-x(B)	
read(B);	
B := B-50	
write(B);	lock-s(A)
	read(A)
	lock-s(B).
(lock-x(A))	

fig: which leads to deadlock.

→ To handle deadlock

A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.

→ Locking protocols restrict the set of possible schedules.

Starvation:

T_2 → has a shared mode lock on data item

T_1 requests a x-mode lock on that data item

T_1 has to wait until T_2 releases the lock.,

In the meanwhile, T_3 may request a S-mode lock on same data item.. T_2 grants the S-mode lock.

T_2 may release the lock, but still T_1 has to wait for T_3 to finish

→ T_1 never gets an X-mode lock on the data item

So T_1 may never make progress, is said to be starved.

The Two-phase locking protocol:

→ This is a protocol which ensures conflict-serializable schedules.

phase 1: Growing phase:

- transaction may obtain locks,
- " may not release any lock.

phase 2: shrinking phase:

- transaction may release locks.
- " may not obtain any new locks

T_3 & T_4 are two phase, T_1 & T_2 are not two phase

→ This protocol ensures conflict serializability.

It can be proved that the transactions can be serialized in the order of their lock points (i.e. the point where a transaction acquired its final lock)

Two phase locking does not ensure freedom from deadlock. Cascading roll backs is possible under two-phase locking. To avoid this, follow a modified protocol called Strict two-phase locking protocol: a Transaction locking be two phase, & must hold all its exclusive locks till it commits / abort.

Rigorous two-phase locking protocol: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

```
T8 : read (a1);          T9 : read (a1)
     read (a2);               read (a2)
     ...                      display(a1+a2)
     read(an);
     write(a1).
```

~~Schedule 2 to 2 two phase~~

Two-phase locking with lock conversions:

→ Convert a lock-S to a lock-X is upgrading

→ Convert a lock-X to a lock-S is downgrading.

→ upgrading takes place in only the growing phase

→ Downgrading takes place in only the shrinking phase.

→ This protocol ensures serializability. The schedules are cascadeless.

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	
write(a)	
unlock...	

Automatic acquisition of locks.

→ when a transaction T_i issues a read(Q) operation, the system issues a lock-S(Q) inst/.

→ when T_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q, if it does, then the system issues an upgrade(Q) inst/, followed by the write(Q) inst/.., Otherwise, the system issues a lock-X(Q) inst/

→ All locks obtained by a transaction are unlocked after Commits & aborts.

Graph-Based protocols:

→ It is an alternative to two-phase locking..

→ Impose a partial ordering → on the set $D=\{d_1, d_2, \ldots d_n\}$ of all data items.

→ If $d_i \to d_j$ then any transaction accessing both d_i & d_j, must access d_i before accessing d_j.

→ partial ordering implies that the set D may now be viewed as a directed acyclic graph, called a database graph.

→ Tree-protocol: is a simple kind of graph protocol, which employ only exclusive locks

→ only exclusive locks are allowed.

→ each T_i must lock a data item at most once.

→ Rules are.

① The first lock by T_i may be on any data item

② subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.

③ Data items may be unlocked at any time.

④ A data item that has been locked

Time-stamp based protocols:

Time stamps: Time stamp is a unique identifier created by DBMS to identify a transaction

With each transaction T_i in the system, a unique fixed timestamp is assigned as $TS(T_i)$ → before the transaction T_i starts execution.

If T_i has been assigned timestamp $TS(T_i)$ and a new transaction T_j enters the system, then $TS(T_i) < TS(T_j)$.

→ use the value of system clock as the timestamp.

Time stamp of T_i is equal to the value of clock when the transaction enters the system.

If $TS(T_i) < TS(T_j)$, then the produced schedule is equivalent to a serial shedule in which T_i appears before T_j

for each data item Q, two time stamp values are assigned.

→ **W-timestamp(Q)** : denotes the largest (latest) timestamp of any transaction that executed write(Q) successfully.

→ **R-timestamp(Q)**: denotes the largest (latest) timestamp of any transaction that executed Read(Q) successfully.

→ these timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

The time-stamp-ordering protocol:

→ It ensures that any conflicting read and write operations are executed in timestamp order.

The protocol operates as follows

(i) Suppose T_i issues read(Q).

(a) If $TS(T_i) < W\text{-timestamp}(Q)$, then T_i needs to read

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	lock-X(D)
	unlock(H)	lock-X(E)	
lock-X(G)			
unlock(D)			
			lock-X(H)
			unlock(D)
		unlock(E)	unlock(H)
unlock(G)		unlock(B)	

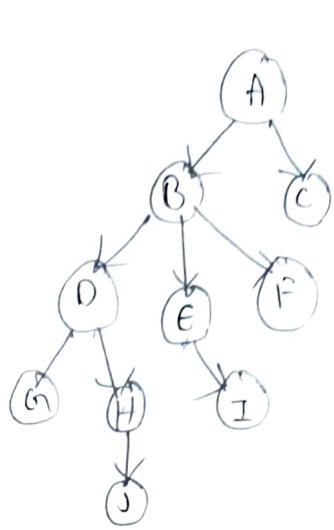

fig. Tree-structured Database graph.

fig. Serializable schedule under the tree protocol.

→ all schedules are conflict serializable schedules according to graph-based rules

→ Tree protocol does not ensure recoverability and cascadelessness

→ To ensure that, the protocol can be modified to not permit release of locks until the end of the transaction

→ holding locks improves concurrency, & ensures recoverability.

adv/ → It is a dead-lock free, so no rollbacks are required.

→ unlocking may occur earlier. So it leads to shorter waiting time, & to an increase concurrency.

disadv → A transaction may have to lock data items that it does not access. It increases waiting time, a decrease in concurrency.

→ The protocol generate schedules that are not recoverable.

T_{14}	T_{15}
read(B)	
	read(B)
	B = B - 50
	write(B)
read(A)	
	read(A)
display(A+B)	
	A = A + 50
	write(A)
	display(A+B)

To ensure recoverability

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

→ It can be guaranteed by using a limited form of locking.

Thomas' write Rule:

→ modification to the TS-Or protocol, that allows greater concurrency

if $TS(T_{16}) < TS(T_{17})$

So above schedule is not true in TS-OP

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

The modification to the TS-Ordering protocol, called Thomas' write rule. If T_i issues a write(Q).

(1) If $TS(T_i) < R\text{-timestamp}(Q)$, the system rejects write operation & T_i Rolled back.

(2) If $TS(T_i) < W\text{-timestamp}(Q)$, Hence this write operation can be ignored.

(3) otherwise, the system executes the write operation & $W\text{-TS}(Q) = TS(T_i)$

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

(2) Suppose T_i issues write (Q).

(a) If $TS(T_i) < R\text{-Timestamp}(Q)$, the value of Q that T_i produced was previously needed & system rejects write operation and T_i rolls back.

(b) If $TS(T_i) < W\text{-timestamp}(Q)$.
System rejects this write operation & T_i rolls back.

(c) Otherwise, the system executes the write operation and set $W\text{-TS}(Q)$ to $TS(T_i)$.

If a Transaction T_i is rolled back, the system assigns a new timestamp and restarts it's.

T_{14}: read(B);
read(A);
display(A+B);

T_{15}: read(B)
B := B - 50;
write(B);
read(A);
A := A + 50;
write(A);
display(A+B).

$TS(T_{14}) < TS(T_{15})$

& the schedule is possible under the time-stamp protocol.

Some schedules possible under the two-phase locking protocol but are not possible under the timestamp protocol.

→ This TS-Ordering protocol ensures Conflict Serializability because Conflicting operations are processed in timestamp order.

→ It ensures freedom from deadlock. So no transaction ever waits.

→ But there is a possibility of starvation.

Validation-Based protocols:

If many of the T's are read-only transactions, the rate of conflicts among transactions may be low. without supervision of concurrency-control mgr, it leads db is inconsistent state. But under the supervision, it may impose overhead. to impose less overhead, you can use these schemes

T_i executes in two or three different phases.

① Readphase. During this phase, the system executes T_i. It reads data items & stores in local variables. It performs write operations on temporary local variables, without updates of the actual db.

② Validation phase:

T_i performs a validation test to determine whether it can copy to the db the temporary local variables that had the results of write operations without causing a violation of serializability.

③ write phase: If T_i succeeds in validation, then the system applies the actual updates to the db. otherwise the system rolls back.

Three different time stamps with T_i

① Start(T_i), the time when T_i started its execution.

② Validation(T_i), the time when T_i finished its read phase and started its validation phase.

③ Finish(T_i), the time when T_i finished its write phase

if $TS(T_j) < TS(T_k)$, then any produced schedule must be equivalent to a serial schedule.

But we have choose validation(T_i) rather than start(T_i)

The validation test for transaction T_j requires that, for all transaction T_i with $TS(T_i) < TS(T_j)$, one of the following two conditions must hold.

① Finish(T_i) < start(T_j), T_i completes its execution before T_j started,

② The set of data items written by T_i does not interact with the set of data items read by T_j, and T_i completes its write phase before T_j starts its validation phase.

i.e. start(T_j) < Finish(T_i) < validation(T_j)

writes of T_i & T_j does not overlap.

validation phase succeeds.

This schedule is serializable

→ validation ~~automatically~~ avoids cascading roll backs.

→ possibility of starvation.

→ validation scheme is called the optimistic concurrency-control scheme, they will be able to finish execution & validate & locking & time stamp are pessimistic, in that they are ~~force~~ wait & a roll back whenever a conflict is detected, ...

T_{14}	T_{15}
read(B)	
	read(B)
	B = B - 50
	read(A)
	A = A + 50
read(A)	
<validate>	
display(A+B)	
	<validate>
	write(B)
	write(B)

Deadlock handling:

→ A system is in deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set.

→ There exists a set of waiting transactions $\{T_0, T_1, \ldots T_n\}$ such that T_0 is waiting for data item that T_1 holds, & T_1 is waiting for data item that T_2 holds, and ... and T_{n-1} is waiting for " " T_n holds, and T_n is waiting for " " T_0 holds, none of the transactions can make progress in such a situation.

→ Rolling back some of T's is the remedy.

→ Deadlock prevention protocol to ensure that the system will never enter a deadlock state.

→ Deadlock Detection and deadlock recovery scheme:

We can allow the system to enter a deadlock state, and then try to recover.

Deadlock prevention:

1st approach → ensures that no cyclic waits can occur by ordering the requests for locks, & requiring all locks to be acquired together.

→ Perform transaction rollback instead of waiting for a lock.

1st approach → Each transaction locks all its data items before it begins execution.

disadv → 1) it is often hard to predict, before the transa. begins, what data items need to be locked

2) data-item utilization

preemption and transaction rollbacks.

In preemption, when a transaction T_2 requests a lock that transaction T_1 holds, the lock granted to T_1 may be preempted by rolling back of T_1, and granting of the lock to T_2.

→ timestamp is assigned to each transaction.

two different deadlock prevention schemes using TS's.

1) **wait-die:** is a non-preemptive technique.

when transl. 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 (dies).

T_{22} requests a data item held by T_{23}. T_{22} : 5 T_{23} : 10 T_{24} : 15

T_{22} has to wait.

T_{24} requests a data item held by T_{23}.

T_{24} will be rolled back.

2) **wound-wait:** is a preemptive technique.

when T_i requests a data item currently held by T_j, and T_i is allowed to wait only if it has a timestamp larger than that of T_j. otherwise, T_j is rolled back & preempted.

→ whenever the system rolls back T's, it is impl. to ensure that there is no starvation.. i.e. no transaction gets rolled back repeatedly. Both schemes avoids starvation.

→ each node in the tree can be locked individually

→ when a 'T' locks a node in 'S' & 'X' mode, all the descendents are implicitly locked in the same mode

→ Ti gets an explicit lock one file F_c in exclusive mode, then it has an implicit lock in exclusive mode on all the records belonging to that file. no need to lock the individual records of F_c explicitly.

→ intention lock modes

↓

If a node is locked in an intention mode, explicit locking is being done at a lower levels of the tree.

→ Intention locks are put on all the ancestors of a node before that node is locked explicitly.

→

Multiple Granularity

→ grouping of several data items, & treat them as one individual unit. If T_i needs to access the entire db, & a locking protocol is used then T_i must lock each item in the db.

→ It is better in a single lock request to lock the entire db.

→ allow the system to define multiple levels of granularity, allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within one larger ones. Such a hierarchy can be represented graphically as a tree.

→ nonleaf-node represents the data associated with its descendants.

→ each node is an independent data item.

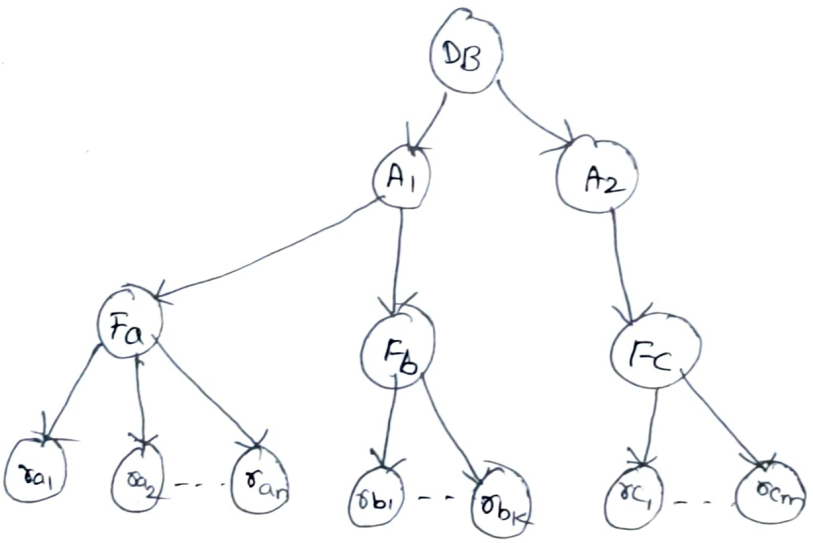

fig: Granularity hierarchy.

→ It consists of four levels of nodes.

→ highest level represents the entire db

→ below it are nodes of type area (db consists of exactly these areas)

→ file - each area contains exactly those files that are its child nodes

→ each file has nodes of type record

Deadlock Detection and Recovery:

The system must

→ Maintain infol. about the current allocation of data items to transactions, as well as any outstanding data item requests.

→ provide an algol. that uses this infol. to determine whether the system has entered a deadlock state.

→ Recover from the deadlock when the detection algol. determines that a deadlock exists

Deadlock Detection:

Deadlocks can be described in terms of directed graph called a wait-for-graph.

When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for-graph.

→ A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for-graph and periodically to invoke an algol. that searches for a cycle in the graph.

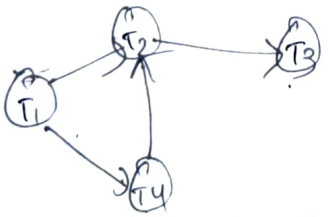

fig: wait-for-graph with no cycle

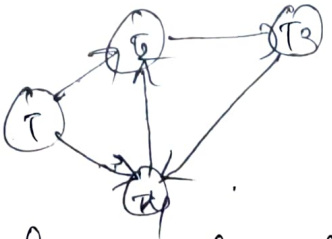

fig: wait-for-graph with a cycle

Reco

Recovery from Deadlock:

→ when a detection algo/. determines that a deadlock the system must recover from the deadlock.

i.e roll back one & more transaction to break the deadlock

1) Selection of a victim: we must determine which transaction to roll back to break the deadlock

2) Roll back. once we have decided that a particular trans/ must be rolled back, we must determine how far this trans/ should be rolled back.

1) total roll back. — i.e. Abort the trans/. & restart it.
2) partial roll back — which looks the selected trans/. needy to release in order to break the deadlock

3) Starvation. — it may happen that the same transa/. is always picked as a victim. then this trans/. never completes its task. so there is a starvation.

we must ensure that a transaction can be picked as a victim only a finite no. of times.

Deferred Database modification:

→ It records all modifications to the log, but deferring the execution of all write operations of a transaction until after partially commits.

→ Transactions execute serially.

→ when a Transaction partially commits, the info. on the log associated with the transaction is used in executing the deferred writes.

→ Transaction starts by writing $\langle T_i \text{ start} \rangle$ record to log

→ A write(X) operation results in a log record $\langle T_i, X, V \rangle$ being written, V is new value
the write is not performed on X at this time, it is delayed

→ when T_i partially commits, $\langle T_i \text{ commit} \rangle$ is written to the log.

→ finally, the log records are read and used to actually execute the previously deferred writes.

T_0 : read(A)
A := A-50
write(A)
read(B)
B := B+50
write(B)

T_1 : read(C)
C := C-100
write(C)

T_0 followed by T_1

→ During recovery after a crash, a transaction needs to be redone if and only if both $\langle T_i \text{ start} \rangle$ and $\langle T_i \text{ commit} \rangle$ are there in the log.

→ Redoing a transaction T_i (redo T_i) - sets the value of all data items updated by the transaction to the new values

Log-Based Recovery:

→ log is used for recording database modifications.

→ the log is a sequence of log records, recording all the update activities in the db.

→ An update log record describes a single database write.

It has 5 fields.

(1) Transaction identifier: is the unique identifier of the transaction that performed the write operation.

(2) Data-item identifier: " " " " data item written

(3) Old value: is the value of the data item prior to the write

(4) New value: is the value that the data item will have after the write.

Various types of log records are.

$\langle T_i \text{ start} \rangle$ T_i has started.

$\langle T_i, X_j, V_1, V_2 \rangle$. T_i performed a write on data item X_j. X_j had value V_1 before the write & will have value V_2 after the write.

$\langle T_i \text{ commit} \rangle$. T_i has committed.

$\langle T_i \text{ abort} \rangle$ T_i has aborted.

→ whenever a Transaction performs a write, it is essential that the log record that write be created before the db is modified.

→ once log record exists, we o/p the modification to the db.

→ undo a modification that has already been o/p to the db.

→ log records are written directly to stable storage.

→ log contains a complete record of all db activity.

Two approaches using logs.

→ Deferred database modification