Concurrency Control

Lock-based protocols:-

→ To ensure serializability is to require that data items be accened in a mutually exclusive manner, i.e while one transaction is accessing a data item, no other transaction can modify that data item.

→ The most common method used to implement this requirement is to allow a transaction to accen data item only if it is currently holding a lock on that item.

Locks:-

→ There are various modes in which data item can be locked.

(1) Shared lock:- If a transaction T_i has obtained a shared-mode lock (denoted by S), on data item Q, then T_i can read, but cannot write Q.

(2) Exclusive lock:- If a transaction T_i has obtained an exclusive-mode lock (denoted by X) on data item Q, then T_i can perform both read and write Q.

→ A tramachion requests a shared lock on data item Q by executing the lock-S(Q).

→ A tramachion request a exclusive lock through the lock-X(Q).

→ A tramachion can unlock a data item Q by unlock (Q). instruction.

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).

Tramachion T_1

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

tramachion T_2.

→ To access a data item, tramachion T_i must first lock that item.

→ If the data item is already locked by another tramachion in a incompatible mode, the concurrency control manager will not grant a lock until all incompatible locks held by another tramachion have been released.

T_1	T_2	Concurrency control Manager
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_2)
read(A) A := A+50 write(A) unlock(A)		

The above table explains how the concurrency control manager is assigned to locks and released the locks b/w two transactions T_1 & T_2.

Granting of locks:-

→ suppose a transaction T_2 has a shared-mode lock on data item,

→ Another transaction T_1 requests an exclusive mode lock on the data item.

→ T_1 has to wait for T_2 to release the shared-mode lock.

→ Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item.

→ The lock request is compatible with the lock granted to T_2, so T_3 may be granted shared-mode lock.

→ At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish.

→ Suppose T_4 new transaction requests a shared mode lock on the same data item, and it is granted the lock before T_3 is released.

→ In this process T_i never gets the chance of exclusive-mode lock on the data item.

→ This situation is called starvation.

→ To avoid starvation of transactions by granting locks in the following manner.

⇒ When a Transaction T_i requests a lock on data item Q in a particular mode M, the concurrency control Manager grants lock provided that

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

(2) There is no other transaction that is waiting for a lock Q, and that made its lock request before T_i.

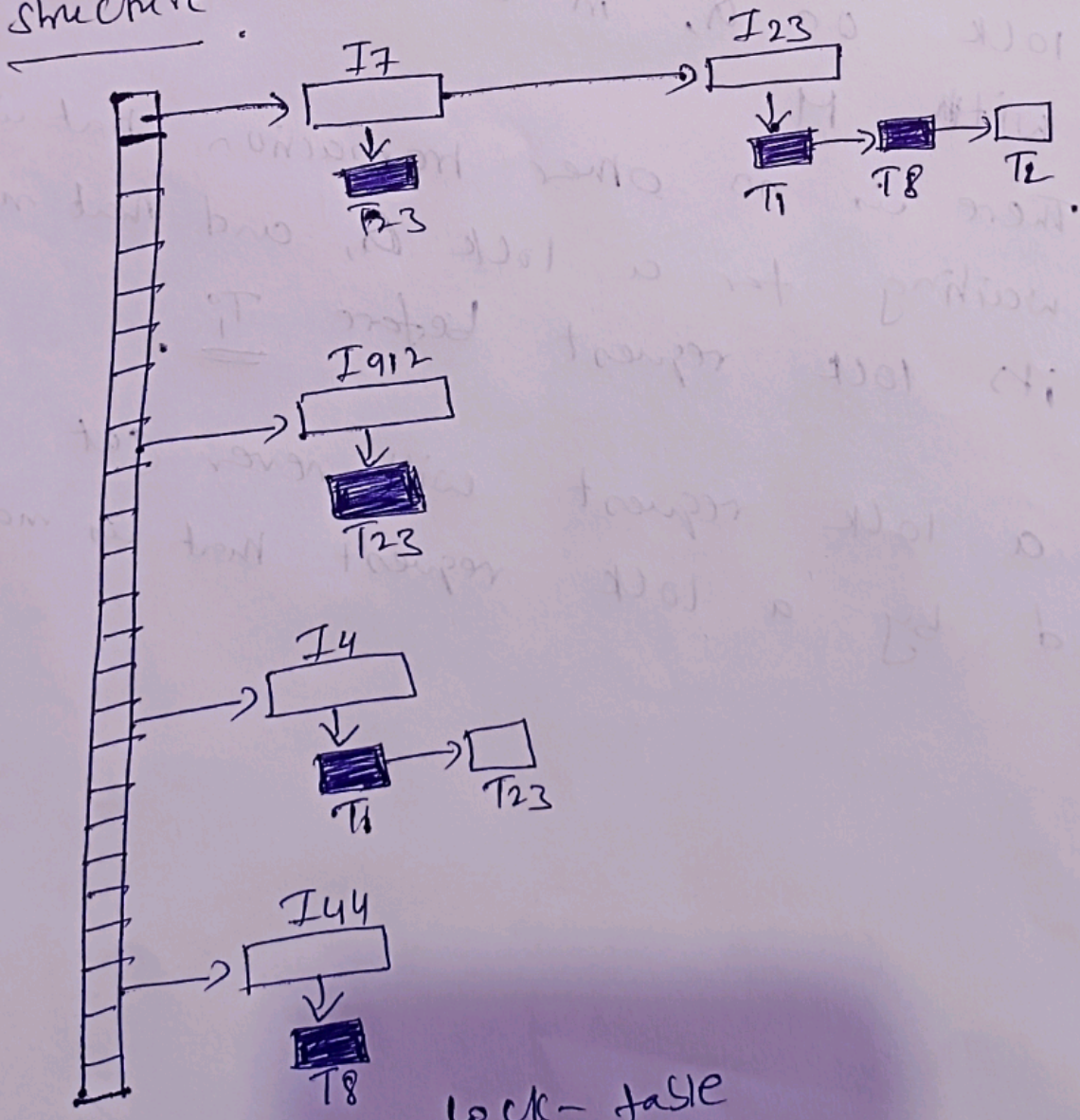
Thus, a lock request will never get blocked by a lock request that is made later.

Implementation of locking

→ A lock manager can be implemented as a process that receives messages from transactions and sends messages in reply.

→ The lock-manager process replies to lock-request messages with lock-grant messages, or with messages requesting rollback of the transaction (in case of deadlock).

→ The lock manager uses the following data structure.



lock-table

→ For each data item that it currently locked, it maintains a linked list of records.
→ one for each request, in the order in which the requests arrived.
→ It uses a hash table indexed on the name of data item.
→ To find the linked list if any for a data item.
→ This table is called the lock-table.
→ Each record of the linked list for a data item notes which transaction made the request, and what lock mode is requested.
→ The record also notes if the request has currently been granted.

In the above lock-table.
→ Table contains locks for 5 different data items.
→ I4, I7, I23, I44 and I912.
→ lock-table uses overflow chaining.
→ There is a linked-list of data items for each entry in the lock-table.
→ There is also a list of transactions that have been granted locks, waiting for locks.

→ Granted locks are filled in black rectangles

→ waiting request are the empty rectangles. □.

→ T_{23} has been granted locks on I_{9} & I_2 and I_7. and it is waiting for I_4.

Lock-Manager processes requests this way

→ When a lock request menage arives, it adds a record to the end of the linked list for the data item, if the linked list is present.

→ Otherwise it creates a new linked list containing only the record for the request.

→ It always grants the first lock request on a data item.

→ If the transaction requests a lock on a item on which a lock has already been granted, the lock manager grants the request only if it is compatible with all earlier requests.

→ If the earlier requests have been granted already.

→ otherwise the request has to wait until lock is released.

Validation based protocols:-

→ Each transaction T_i executes in two or three different phases in its lifetime, depending on whether it is read-only or an update transaction.

→ The phases are

① **Read phase:-** During this phase, the system executes transaction T_i.

→ It reads the value of the various data items and stores them in variables local to T_i.

→ It performs all write operation on temporary local variables, without updates of the actual database.

② **validation phase:-** Transaction T_i performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operation. without causing violation of serializability.

(3) Write phase:- If transaction T_i succeeds in validation step 2 done,

→ The system applies the actual updates to the database.

→ The system rolls back T_i.

→ Each transaction must go through three phases.

To perform the validation test, transaction T_i has 3 different timestamps.

(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.

The validation test for T_j requires that, for all transactions T_i with $TS(T_i) < TS(T_j)$ one of the 2 conditions must hold

① Finish(T_i) < Start(T_j).

→ Since T_i completes its execution before T_j started, the serializability order is indeed maintained.

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

Start (T_j) < Finish (T_i) < validation (T_j)