Unit-7 Concurrency Control

Syllabus:

Note:- All the content are taken from Database System Concepts by Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Sixth Edition, Mc Graw Hill Publication

7.1 Lock Based Protocol

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, 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 access a data item only if it is currently holding a lock on that item.

7.1.1 Locks

- ***Shared:*** If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can read, but cannot write, Q.
- ***Exclusive:*** If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can both read and write Q.

Every transaction ***request*** a lock in an appropriate mode on data item Q. The transaction makes the request to the ***concurrency-control manager***. The transaction can proceed with the operation only after the concurrency-control manager ***grants*** the lock to the transaction. The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

Compatibility Function:

- Let A and B represent arbitrary lock modes.
- Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction Tj currently holds a lock of mode B.
- If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B.
- Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking discussed in this section appears in the matrix ***comp*** of Figure 7.1.
- An element ***comp(A, B)*** of the matrix has the value true if and only if mode A is compatible with mode B.

	S	X
S	true	false
X	false	false

Figure 7.1 Lock-compatibility matrix comp.

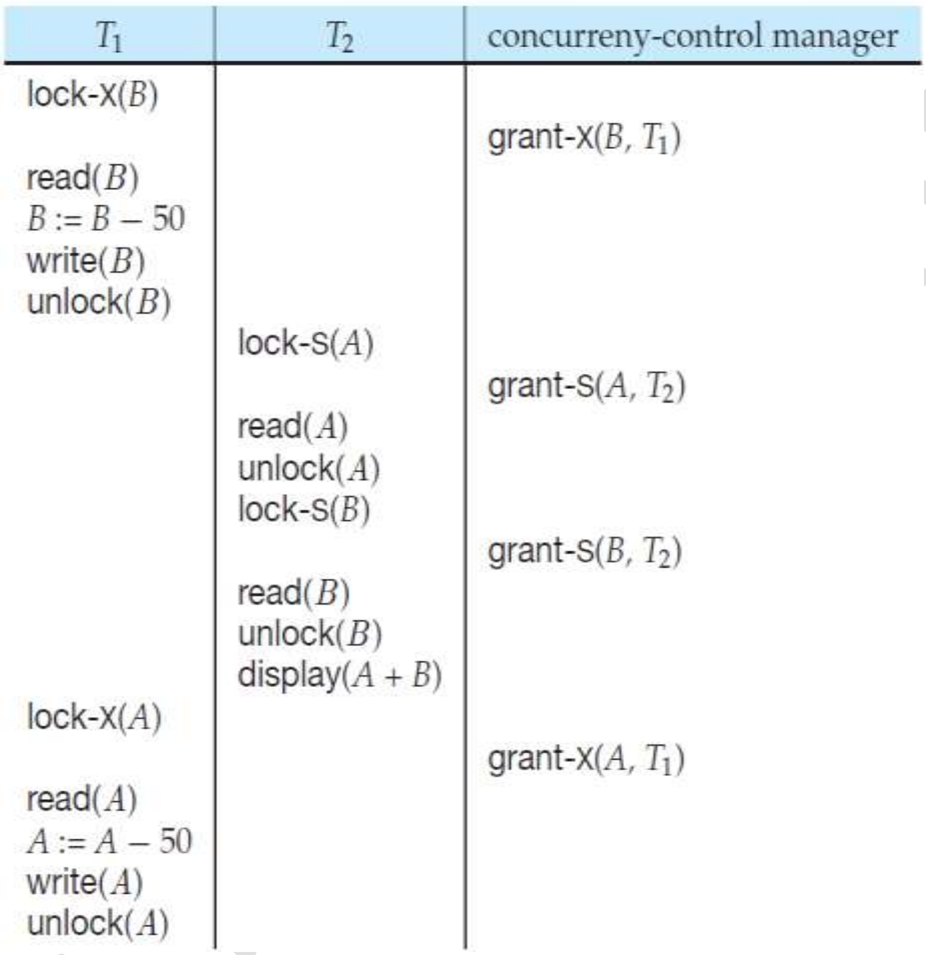
- To access a data item, transaction Ti must first lock that item.
- If the data item is already locked by another transaction in an incompatible mode, the ***concurrency control manager*** will not ***grant*** the lock until all incompatible locks held by other transactions have been released.
- Thus, Ti is made to ***wait*** until all incompatible locks held by other transactions have been released.
- Let A and B be two accounts that are accessed by transactions T1 and T2. Transaction T1 transfers $50 from account B to account A . Transaction T2 displays the total amount of money in accounts A and B—that is, the sum A + B.

```
T1: lock-X(B);                      T2: lock-S(A);
      read(B);                            read(A);
      B := B − 50;                        unlock(A);
      write(B);                            lock-S(B);
      unlock(B);                           read(B);
      lock-X(A);                           unlock(B);
      read(A);                            display(A + B).
     A := A + 50;
     write(A);
     unlock(A).
```

Figure 7.2 Transaction T1 and T2

- Suppose that the values of accounts A and B are $100 and $200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2, T1, then transaction T2 will display the value $300.

- If these transaction executing concurrently, following schedule shown in figure 7.3 is possible in which transaction T2 display $250, which is incorrect.



T_1	T_2	concurreny-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_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

Figure 7.3 Schedule 1

- **Locking Protocol:** Each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. **Locking protocols** restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules.
- **Legal Schedule:** We say that a schedule S is ***legal*** under a given locking protocol if S is a possible schedule for a set of transactions that follows the rules of the locking protocol.

- We say that a locking protocol ensures conflict serializability if and only if all legal schedules are conflict serializable.

7.1.2 Granting Lock

- When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.
- Suppose a transaction T2 has a shared-mode lock on a data item, and another transaction T1 requests an exclusive-mode lock on the data item.
- Clearly, T1 has to wait for T2 to release the shared-mode lock.
- Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock.
- At this point T2 may release the lock, but still T1 has to wait for T3 to finish.
- There may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it.
- It is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item.
- The transaction T1 may never make progress, and is said to be **starved.**
- Transaction starvation can be avoided by granting locks in the following manner:
- When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the 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 on Q and that made its lock request before Ti.
- A lock request will never get blocked by a lock request that is made later.

7.1.3 The Two-Phase Locking Protocol

- One protocol that ensures serializability is the two-phase locking protocol.
- This protocol requires that each transaction issue lock and unlock requests in two phases:
  - **Growing phase:** A transaction may obtain locks, but may not release any lock.
  - **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.
- Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and there after transaction does not acquire lock.
- Example: Following Transaction T3 and T4 are two-phase

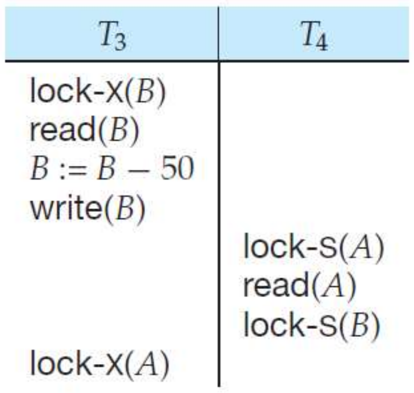
```
T3: lock-X(B);              T4: lock-S(A);
      read(B);                    read(A);
      B := B − 50;                lock-S(B);
      write(B);                   read(B);
      lock-X(A);                  display(A + B);
      read(A);                    unlock(A);
      A := A + 50;                unlock(B).
      write(A);
      unlock(B);
      unlock(A).
```

Figure 7.3 Example of two-phase locking

- Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction T3, we could move the unlock(B) instruction to just after the lock-X(A) instruction, and still retain the two-phase locking property.
- Two Phase Locking protocol ensure **Conflict Serializability:**
  - The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction.

- Transactions can be ordered according to their lock points this ordering is, in fact, a serializability ordering for the transactions.

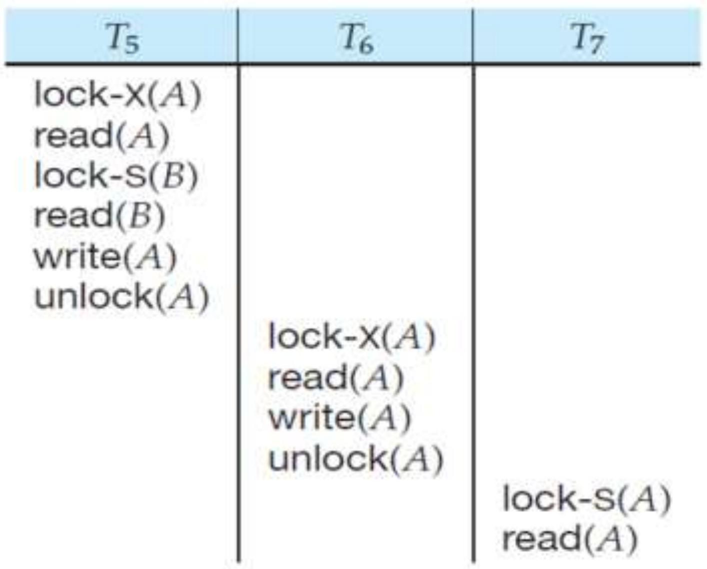
- Two Phase Locking protocol **does not give freedom from Deadlock**:
  - Consider following schedule



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)	

Figure: 7.4 Example of deadlock schedule

  - Above schedule is ensuring two phase locking but does not avoid deadlock
  - Cascading rollback may occur under two-phase locking.
  - Consider the partial schedule



T_5	T_6	T_7
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)

Figure Partial schedule under two-phase locking.

- Each transaction observes the two-phase locking protocol, but the failure of T5 after the read(A) step of T7 leads to cascading rollback of T6 and T7.

- **Strict two-phase locking protocol.**
    - Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase locking protocol.
    - This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.
    - This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits.
    - This will prevent any other transaction from reading the data.
- **Rigorous two-phase locking**
    - It requires that all locks be held until the transaction commits.
    - We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit.

7.1.4 Implementation of Locking

- **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**.
- In case of deadlock it may reply requesting **rollback message** of the transaction.
- Unlock messages require only an acknowledgment in response, but may result in a grant message to another waiting transaction.
- **Data Structure Maintained by Lock Manager:**
    - For each data item that is currently locked, it maintains a linked list of records, one for each request, in the order in which the requests arrived.
    - **Lock Table:** It uses a hash table, indexed on the name of a data item, to find the linked list (if any) for a data item; this table is called the lock table.

- Each record of linked list contain, which transaction made the request, what lock mode is requested and whether request has been granted or not.

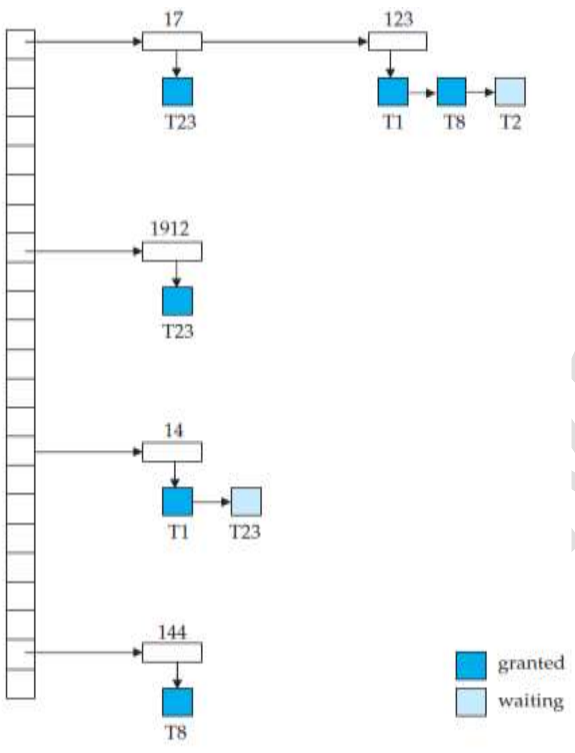


Figure Lock table.

- The table contains locks for five different data items, 14, 17, 123, 144, and 1912.
- The lock table uses overflow chaining, so 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, or are waiting for locks, for each of the data items.
- Granted locks are the rectangles filled in a darker shade, while waiting requests are the rectangles filled in a lighter shade.
- It can be seen, for example, that T23 has been granted locks on 1912 and 17, and is waiting for a lock on 14.
- The lock table should also maintain an index on transaction identifiers, so that it is possible to determine efficiently the set of locks held by a given transaction. This has been not shown in figure for simplicity.

- **Processing of request by Lock Manager:**

Step-1 When Lock Manager Receive Request Message:

  - When a lock request message arrives, 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 a lock request on a data item that is not currently locked.
  - But if the transaction requests a lock on an item on which a lock is currently held, the lock manager grants the request only if it is compatible with the locks that are currently held, and all earlier requests have been granted already. Otherwise the request has to wait.

Step-2 When Lock Manager Receive unlock Message:

  - First it deletes the record for that data item in the linked list corresponding to that transaction.
  - It check the records and see whether another transaction waiting for the same data item, if another present then it grant the request of another transaction.

- **Step-3 If transaction aborts:**
  - The lock manager deletes any waiting request made by the transaction.
  - Once the database system has taken appropriate actions to undo the transaction, it releases all locks held by the aborted transaction.
- This algorithm guarantees freedom from starvation for lock requests, since a request can never be granted while a request received earlier is waiting to be granted.

7.1.5 Implementation of Locking

- If we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database.

- There are various models that can give us the additional information, The simplest model requires that we have prior knowledge about the order in which the database items will be accessed.
- Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.
- We impose a partial ordering → on the set D = {d1, d2, . . . , dh} of all data items.
- If $di \rightarrow dj$, then any transaction accessing both di and dj must access di before accessing dj.
- The partial ordering implies that the set D may now be viewed as a directed acyclic graph, called **a database graph.**
- for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees.
- We shall present a simple protocol, called the **tree protocol**, which is restricted to employ only **exclusive locks**.
- **Tree Protocol**
  - the only lock instruction allowed is lock-X. Each transaction Ti can lock a data item at most once, and must observe the following rules:
    - The first lock by Ti may be on any data item.
    - Subsequently, a data item Q can be locked by Ti only if the parent of Q is currently locked by Ti .
    - Data items may be unlocked at any time.
    - A data item that has been locked and unlocked by Ti cannot subsequently be relocked by Ti .
  - All schedules that are legal under the tree protocol are conflict serializable.
  - consider the database graph of Figure
  - The following four transactions follow the tree protocol on this graph.We show only the lock and unlock instructions:

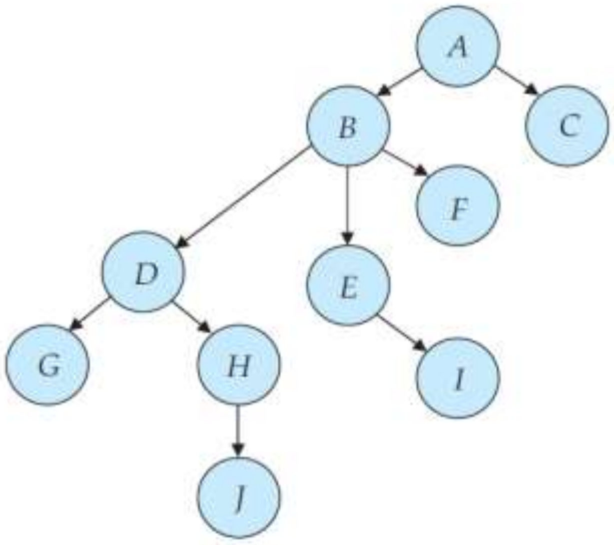
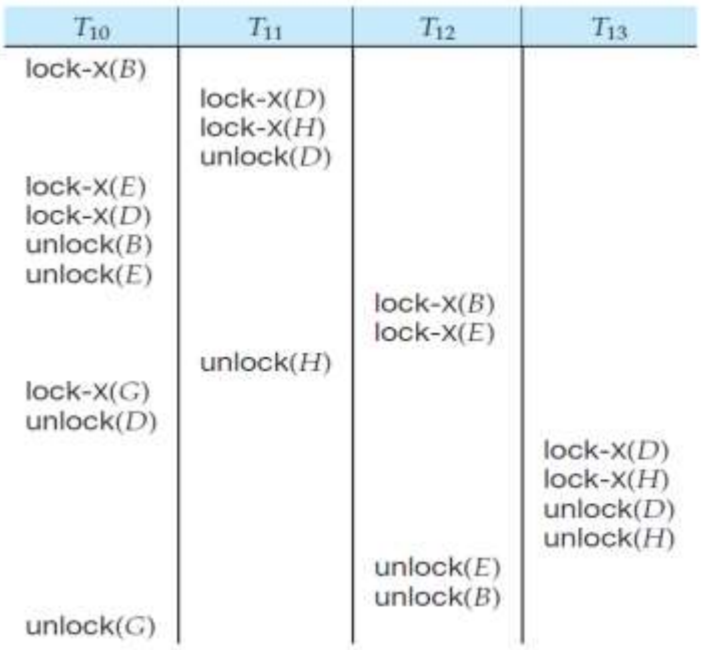


Figure Tree-structured database graph.

- T10: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
- T11: lock-X(D); lock-X(H); unlock(D); unlock(H).
- T12: lock-X(B); lock-X(E); unlock(E); unlock(B).
- T13: lock-X(D); lock-X(H); unlock(D); unlock(H).

- Following figure shows one of the schedule in which above four transaction participated



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(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

Figure Serializable schedule under the tree protocol.

- Observe that the schedule shown in above Figure is **conflict serializable**.
- It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.
- The tree protocol in Figure does not ensure recoverability and cascadelessness.
- To ensure recoverability and cascadelessness, the protocol can be modified as follows
    - Do not permit release of exclusive locks until the end of the transaction.
    - Holding exclusive locks until the end of the transaction reduces concurrency.
    - Another modification is suggested as for each data item with an uncommitted write, we record which transaction performed the last write to the data item.
    - **Example:** Whenever a transaction Ti performs a read of an uncommitted data item, we record a commit dependency of Ti on the transaction that performed the last write to the data item.
    - Transaction Ti is then not permitted to commit until the commit of all transactions on which it has a commit dependency.
    - If any of these transactions aborts, Ti must also be aborted.
- **Advantages:**
    - Over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required.
    - Unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.
- **Disadvantages:**
    - In some cases, a transaction may have to lock data items that it does not access.
    - A transaction that needs to access data items A and J in the database graph of above Figure must lock not only A and J, but also data items B, D, and H. This additional locking results in

increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency.

- Further, without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree, and that can reduce concurrency greatly.

7.2 Deadlock handling

A system is in a 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 $\{T_0, T_1, . . . , T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T1 is waiting for a data item that T_2 holds, and . . . , and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds.

There are two principal methods for dealing with the deadlock problem.

a) System Should not enter the deadlock state i.e. Deadlock Prevention

b) After system enter in deadlock state: deadlock detection and recovery

7.2.1 Deadlock Prevention

There are two approaches for deadlock prevention:

Approach One: Ensure no cyclic waits can occur by ordering the requests for locks or acquire all required lock together.

Scheme Used: Each transaction locks all its data items before it begins execution. In this either all are locked in one step or none are locked.

Disadvantages:

i) It is often hard to predict, before the transaction begins, what data items need to be locked.

ii) Data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Another variation with this approach: Total order of data items, in conjunction with two-phase locking.

Scheme: Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering.

Advantages:

i) This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution.

ii) There is no need to change the underlying concurrency-control system if two-phase locking is used.

Approach Two: Performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

Scheme: When a transaction T_j requests a lock that transaction T_i holds, the lock granted to T_i may be preempted by rolling back of T_i , and granting of the lock to T_j .

- To control the preemption, unique timestamp is assigned, based on a counter or on the system clock, to each transaction when it begins.
- The system uses these timestamps only to decide whether a transaction should wait or roll back.

Two Deadlock prevention scheme based on timestamp:

i) Wait-die

- It is a non-preemptive technique
- When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp smaller than that of Tj
- i.e, Ti is older than Tj , Ti comes in system before Tj.
- Other wise Ti is rolled back.
- This scheme allows the older transaction to "wait" but kills the younger one ("die").
- **Example:** Suppose that transaction T5, T10, T15 have time-stamps 5, 10 and 15 respectively. If T5 requests a data item held by T10 then T5 will "wait". If T15 requests a data item held by T10, then T15 will be killed ("die").

ii) Wound Wait:

- It is a preemptive technique for deadlock prevention.
- When transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp larger than that of Tj.
- Otherwise, Tj is rolled back (Tj is wounded by Ti).

- If a transaction requests to lock a resource (data item), which is already held with conflicting lock by some another transaction, one of the two possibilities may occur:
  a) **Timestamp(Ti) < Timestamp(Tj),** then Ti forces Tj to be killed – that is Ti "wounds" Tj. Tj is restarted later with a random delay but with the same timestamp(j).
  b) **Timestamp(Ti) > Timestamp(Tj),** then Ti is forced to "wait" until the resource is available.
- **Example:** Suppose that Transactions T5, T10, T15 have time-stamps 5, 10 and 15 respectively. If T5 requests a data item held by T10, then data item will be preempted from T10 and T10 will be suspended. ("wounded"). If T15 requests a data item held by T10, then T15 will "wait".

iii) Time out base approach:

- 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 time out, and it rolls itself back and restarts.
- **Advantages:**
  - Easy to implement
  - It works well if transactions are short and if long waits are likely to be due to deadlocks.
- Disadvantages:
  - It is hard to decide how long a transaction must wait before timing out.
  - Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock.
  - Starvation is also a possibility with this scheme.

7.2.2 Deadlock Detection

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

- This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges.
- The set of vertices consists of all the transactions in the system. Each element in the set E of edges is an ordered pair Ti → Tj
- When transaction Ti requests a data item currently being held by transaction Tj , then the edge Ti → Tj 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 algorithm that searches for a cycle in the graph.
- Consider the following figure 7.5

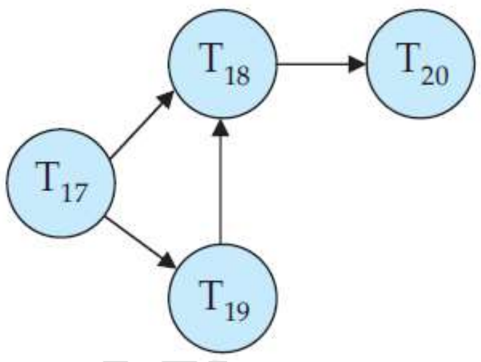


Figure 7.5 Wait for graph with no cycle

  - Transaction T17 is waiting for transactions T18 and T19.
  - Transaction T19 is waiting for transaction T18.
  - Transaction T18 is waiting for transaction T20.
- Suppose now that transaction T20 is requesting an item held by T19.
- The edge T20 → T19 is added to the wait-for graph, resulting in the new system state in Figure 7.6
- This time, the graph contains the cycle: T18 →T20 →T19 →T18 implying that transactions T18, T19, and T20 are all deadlocked.

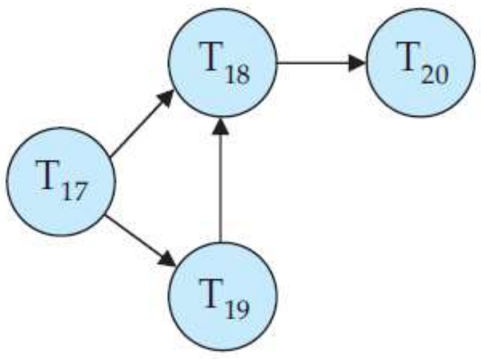


Figure 7.6 Wait for graph with a cycle

- When should we invoke the detection algorithm?
  - How often does a deadlock occur?
  - How many transactions will be affected by the deadlock?
- If deadlocks occur frequently, then the detection algorithm should be invoked more frequently

7.2.3 Deadlock Recovery

- When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock.
- The most common solution is to roll back one or more transactions to break the deadlock.

1) Selection of Victim

- Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.
- We should roll back those transactions that will incur the minimum cost.

ખર્ચ

- Many factors may determine the cost of a rollback, including:

a) How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.

b) How many data items the transaction has used.

c) How many more data items the transaction needs for it to complete.

d) How many transactions will be involved in the rollback.

2) Rollback

- Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
- **Total Rollback:** Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock.
- **Partial Rollback:** Requires the system to maintain additional information about the state of all the running transactions.
- The sequence of lock requests/grants and updates performed by the transaction needs to be recorded.
- The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock.
- The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point.

Rolling Back to a Safe Point:

If a task is causing a problem and needs to be stopped, the system needs to roll back that task to a safe point.
This safe point is where the task first locked something. So, all the changes made by that task after that point are undone.

3) Starvation:

- It may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is ***starvation***.
- We must ensure that a transaction can be picked as a victim only a (small) finite number of times.
- The most common solution is to include the number of rollbacks in the cost factor.

7.3 Multiple Granularity

In some cases it would be advantageous to group several data items to treat them as one individual synchronization unit instead of treating each individual data items separately.

Example: If a transaction Ti needs to access the entire database, and a locking protocol is used, then Ti must lock each item in the database. Executing these lock is time consuming process. It would be better if Ti could issue a single lock request to lock the entire database. If transaction Tj needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.

In order to achieve better concurrency control, separate mechanism is required to allow the system to multiple levels of **granularity**.

- This mechanism can be implemented by defining the data item in hierarchical format by representing in the form of tree.

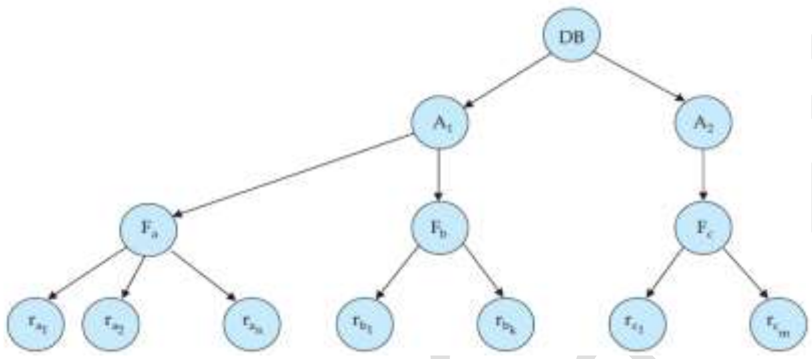


Figure 7.5 Granularity Hierarchy

- In the tree protocol, each node is an independent data item.
- A nonleaf node of the multiple-granularity tree represents the data associated with its descendants.
- Figure 7.5 shows four levels of nodes.
- The highest level represents the entire database.
- Bellow the highest level the nodes of type ***area***; the database consists of exactly these areas.
- Each area in turn has nodes of type ***file*** as its children. Each area contains exactly those ***files*** that are its child nodes. No ***file*** is in more than one area.
- Finally, each ***file*** has nodes of type ***record***. The file consists of exactly those ***records*** that are its child nodes, and no record can be present in more than one file.
- To lock the node of tree individually ***shared*** and ***exclusive*** lock modes are used.
- When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode.
- It does not need to lock the individual records explicitly.

- **Problem:** If transaction Tk wish to lock the entire database, then transaction should lock the root but transaction will not succeed if Ti lock some part of the tree. How transaction should decide to lock entire tree ?
- **Solution:** 1) Search the entire tree. This solution will not give better performance of multiple granularity.
  2)To give efficient solution new lock is introduced called intention lock mode
- **Intention lock modes:**
  - If a node is locked in an intention mode, explicit locking is done at a lower level of the tree.
  - Intention locks are put on all the ancestors of a node before that node is locked explicitly.
  - A transaction does not need to search the entire tree to determine whether it can lock a node successfully.
- **Intention-shared (IS) mode:** Explicit locking is being done at a lower level of the tree, but with only shared-mode locks.
- **Intention-exclusive (IX) mode:** Explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks.
- **Shared and Intention-Exclusive (SIX) mode:** The subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks
- Following figure shows the compatibility matrix:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Figure 7.6 Compatibility Matrix

- **Multiple-granularity locking protocol**
  - Uses lock mode to ensure serializability
  - It requires that a transaction Ti that attempts to lock a node Q must follow these rules:
    a. Transaction Ti must observe the lock-compatibility function of Figure 7.6.
    b. Transaction Ti must lock the root of the tree first, and can lock it in any mode.
    c. Transaction Ti can lock a node Q in S or IS mode only if Ti currently has the parent of Q locked in either IX or IS mode.
    d. Transaction Ti can lock a node Q in X, SIX, or IX mode only if Ti currently has the parent of Q locked in either IX or SIX mode.
    e. Transaction Ti can lock a node only if Ti has not previously unlocked any node (that is, Ti is two phase).
    f. Transaction Ti can unlock a node Q only if Ti currently has none of the children of Q locked.
  - Observe that the multiple-granularity protocol requires that locks be acquired in top-down order, whereas locks must be released in bottom-up order.
  - **Example:** Suppose that transaction T1 reads record ra_2 in file F_a . Then, T1 needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock ra_2 in S mode.
- This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of:
  - Short transactions that access only a few data items.
  - Long transactions that produce reports from an entire file or set of files.

7.4 Time Stamp-Based Protocol

Another method for determining the serializability order is to select an ordering among transactions in advance

7.4.1 Timestamps

- Database system assign unique timestamps for each transaction Ti denoted by TS(Ti)
- Timestamp is assigned by database system before the transaction Ti starts execution.
- If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj).
- Methods for implementing timestamps scheme:
    1. Use the value of the ***system clock*** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
    2. Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.
- The timestamps of the transactions determine the serializability order. Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj .
- For implementing scheme each data item Q has two timestamp
    - **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed write(Q) successfully.
    - **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.

7.4.2 Timestamp ordering protocol

- The ***timestamp-ordering protocol*** ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction Ti issues **read(Q).**

a. ***If TS(Ti) < W-timestamp(Q)***, then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is **rejected**, and Ti is **rolled back**.

b. ***If TS(Ti) ≥ W-timestamp(Q)***, then the read operation is **executed**, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

2. Suppose that transaction Ti issues **write(Q).**

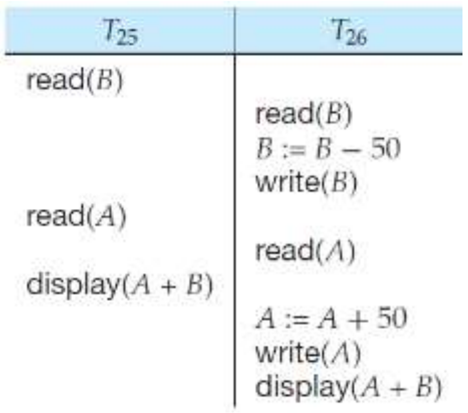
   a. **If TS(Ti) < R-timestamp(Q),** then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system **rejects** the write operation and rollsback Ti.

   b. **If TS(Ti) < W-timestamp(Q),** then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.

   c. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

- If a transaction Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.
- **Example:**
  - we consider transactions T25 and T26.
  - Transaction T25 displays the contents of accounts A and B.
  - Transaction T26 transfers $50 from account B to account A, and then displays the contents of both:

T25: read(B);
read(A);
display(A + B).
	T26: read(B);
	B := B − 50;
	write(B);
	read(A);
	A := A + 50;
	write(A);
	display(A + B).



T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Figure 7.6 Schedule under timestamp protocol

- we will assume that a transaction is assigned a timestamp immediately before its first instruction
- In schedule shown in figure 7.6 **TS(T25) < TS(T26)**, and the schedule is possible under the timestamp protocol.

- The timestamp-ordering protocol **ensures conflict serializability**. This is because conflicting operations are processed in timestamp order.
- Protocol ensures **freedom from deadlock**, since no transaction ever waits.
- There is a **possibility of starvation** of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction.
- The protocol can generate schedules that are **not recoverable.**

7.4.3 Thomas' Write Rule

This suggest the modification to the timestamp-ordering protocol that allows greater potential concurrency. Consider the following schedule shown in figure 7.7 and apply the timestamp-ordering protocol. Since T1 starts before T2, we shall assume that TS(T1) < TS(T2).

- The read(Q) operation of T1 succeeds, as does the write(Q) operation of T2.
- When T1 attempts its write(Q) operation, we find that TS(T1) < W-timestamp(Q), since W-timestamp(Q) = TS(T2).
- The write(Q) by T1 is rejected and transaction T1 must be rolled back.
- The rollback of T1 is required by the timestamp-ordering protocol, it is unnecessary.

- This observation leads to a modified version of the timestamp-ordering protocol in which obsolete ***write*** operations can be ignored under certain circumstances.
- The protocol rules for ***read*** operations remain unchanged. The protocol rules for write operations, however, are slightly different from the timestamp-ordering protocol.

T1	T2
read(Q)	
	write(Q)
write(Q)	

Figure 7.7 Schedule

- The modification to the timestamp-ordering protocol, called ***Thomas' write rule***, is this: Suppose that transaction Ti issues write(Q).
- **If TS(Ti) < R-timestamp(Q),** then the value of Q that Ti is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
- **If TS(Ti) < W-timestamp(Q)**, then Ti is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.
- **Otherwise**, the system executes the write operation and sets W-timestamp(Q) to TS(Ti).

Figure 7.7 is not conflict serializable and, is not possible under the two-phase locking protocol, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T27 would be ignored.

7.5 Validation Based Protocol

A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for monitoring the system.

The ***validation protocol*** requires that each transaction Ti executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction.

1. **Read phase:** During this phase, the system executes transaction Ti. It reads the values of the various data items and stores them in variables local to Ti. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase:** The validation test (described below) is applied to transaction Ti . This determines whether Ti is allowed to proceed to the write phase without causing a violation of serializability. If a transaction fails the validation test, the system aborts the transaction.
3. **Write phase:** If the validation test succeeds for transaction Ti, the temporary local variables that hold the results of any write operations performed by Ti are copied to the database. Read-only transactions omit this phase.

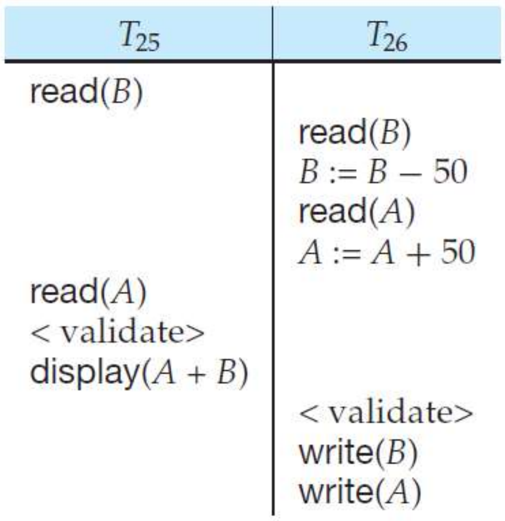
To perform the validation test, we need to know when the various phases of transactions took place. We shall, therefore, associate three different timestamps with each transaction Ti :

1. **Start(Ti):** the time when Ti started its execution.
2. **Validation(Ti):** the time when Ti finished its read phase and started its validation phase.
3. **Finish(Ti):** the time when Ti finished its write phase.

The **validation test** for transaction Ti requires that, for all transactions T_k with TS(T_k) < TS(Ti), one of the following two conditions must hold:

1. Finish(Tk) < Start(Ti). Since Tk completes its execution before Ti started, the serializability order is indeed maintained.
2. The set of data items written by Tk does not intersect with the set of data items read by Ti, and Tk completes its write phase before Ti starts its validation phase (Start(Ti) < Finish(Tk) < Validation(Ti)). This condition ensures that the writes of Tk and Ti do not overlap. Since the writes of

Tk do not affect the read of Ti , and since Ti cannot affect the read of Tk, the serializability order is indeed maintained.



T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
< validate>	
display($A + B$)	
	< validate>
	write(B)
	write(A)

Figure 7.5 Schedule produced by using validation

Consider again transactions T25 and T26. Suppose that TS(T25) < TS(T26). Then, the validation phase succeeds in the schedule shown in Figure 7.5. Note that the writes to the actual variables are performed only after the validation phase of T26. Thus, T25 reads the old values of B and A, and this schedule is serializable.

- Scheme automatically guards against cascading rollbacks since the actual writes take place only after the transaction issuing the write has committed
- Scheme is called the optimistic concurrency-control scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end

Reference:

Database System Concepts by Abraham Silberschatz, Henry F. Korth, S. Sudarshan, Sixth Edition, Mc Graw Hill Publication