CS322: Database Systems

Concurrency Control:

Lock-Based Protocol

Dr. Manas Khatua
Assistant Professor
Dept. of CSE
IIT Jodhpur
E-mail: manaskhatua@iitj.ac.in

Concurrency Control

- The fundamental properties of a transaction is isolation.
- When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved.
- The system must control the interaction among the concurrent transactions to ensure the isolation.
- This control is achieved through one of a variety of mechanisms called *concurrency control* schemes.
- Using **concurrency control protocols** (sets of rules) the serializability is ensured.
- There are a variety of concurrency-control techniques
  - Lock-Based Protocols
  - Timestamp-Based Protocols
  - Validation-Based Protocols
- No one scheme is clearly the best; each one has advantages.

Lock-Based Protocol

Lock-Based Protocols

- What is Lock?
  - A lock is a variable associated with a data item
  - It describes the status of the item w.r.t. possible operations that can be applied to it.

  - A lock is a mechanism
  - It controls concurrent access to a data item

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

- Several types of locks are used in concurrency control.
  - Binary lock
  - Shared/exclusive lock (or, read/write lock)

Binary Lock

- A **binary lock** can have **two states** or values:
  - locked and unlocked (or 1 and 0, for simplicity).

- A distinct lock is associated with each database item *X*.

- If the value of the lock on *X* is 1, item *X* *cannot be accessed* by a database operation that requests the item.

- If the value of the lock on *X* is 0, the item can be accessed when requested, and the lock value is changed to 1.

Lock & Unlock operations in Binary lock

- **lock_item(*X*):**

```
B: if LOCK(X) = 0                    (* item is unlocked *)
        then LOCK(X) ←1              (* lock the item *)
   else
       begin
        wait (until LOCK(X) = 0
               and the lock manager wakes up the transaction);
        go to B
       end;
```

- **unlock_item(*X*):**

```
        LOCK(X) ← 0;                 (* unlock the item *)
        if any transactions are waiting
               then wakeup one of the waiting transactions;
```

- Hence, a binary lock enforces **mutual exclusion** on the data item

Binary Lock (Cont...)

- It is quite simple to implement a binary lock

- each lock can be a record with three fields:
  - <Data_item_name, LOCK_variable, Locking_transaction>
  - plus a queue for transactions that are waiting to access the item.

- The system needs to maintain *only these records for the items that are currently locked* in a **lock table**, which could be organized as a hash file on the item name

- The DBMS has a **lock manager subsystem** to keep track of and control access to locks.

- In binary locking, every transaction must obey the following **rules**

  1) A transaction T must issue the operation *lock_item(X)* before any *read_item(X)* or *write_item(X)* operations are performed in T.
  2) A transaction T must issue the operation *unlock_item(X)* after all *read_item(X)* and *write_item(X)* operations are completed in T.

  3) A transaction T will not issue a *lock_item(X)* operation if it already holds the lock on item X.
  4) A transaction T will not issue an *unlock_item(X)* operation unless it already holds the lock on item X.

Shared/Exclusive Lock

- Binary locking scheme is **too restrictive** for database items because at most one transaction can hold a lock on a given item
- We should allow several transactions to access the same item *X* if they all access *X* for *reading purposes only*.
  - Solution: multiple-mode lock (e.g., shared/exclusive lock)
- Data items can be locked in two modes in shared/exclusive lock:
  - exclusive (X) mode. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  - shared (S) mode. Data item can only be read. S-lock is requested using **lock-S** instruction.
- there are three operations:
  - read_lock(*X*) OR, lock-S(X) : shared mode
  - write_lock(*X*) OR, lock-X(X) : exclusive mode
  - unlock(*X*)
- Lock requests are made to the concurrency-control manager by the programmer.
- Transaction can proceed only after request is granted.

Cont…

- **Rules** for the shared/exclusive locking scheme

1. A transaction *T* must issue the operation read_lock(*X*) or write_lock(*X*) before any read_item(*X*) operation is performed in *T*.
2. A transaction *T* must issue the operation write_lock(*X*) before any write_item(*X*) operation is performed in *T*.
3. A transaction *T* must issue the operation unlock(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T*.
4. A transaction *T* will not issue a read_lock(*X*) operation if other transaction already holds a write (exclusive) lock on item *X*.
5. A transaction *T* will not issue a write_lock(*X*) operation if other transaction already holds a read (shared) lock or write (exclusive) lock on item *X*.
6. A transaction *T* will not issue an unlock(*X*) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item.
All other combinations are not allowed.

Cont…

- Each record in the lock table will have four fields:
  - <Data_item_name, LOCK_variable, No_of_reads, Locking_transaction(s)>

- Example of a transaction performing locking:

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

Figure 15.2 Transaction T_1.

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Figure 15.3 Transaction T_2.

Shortcomings of Read-Write lock

- Locking as above **is not sufficient** to guarantee conflict serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- The schedule shows an inconsistent state.
- The reason for this mistake is that the transaction *T*1 unlocked data item *B* too early, as a result of which *T*2 saw an inconsistent state.

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 15.4 Schedule 1.

Naïve Solution

- unlocking is delayed to the end of the transaction

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

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 15.7 Schedule 2.

- Delayed unlocking can lead to an undesirable situation (e.g., deadlock)
- We have arrived at a state where neither of these transactions can ever proceed with its normal execution.
- This situation is called deadlock.

Deadlock and Starvation

- If we do not use locking, or if we unlock data items too soon after reading or writing them, we may get inconsistent states.
- On the other hand, if we do not unlock a data item before requesting a lock on another data item, deadlocks may occur.
- It is possible that there is a sequence of transactions that each requests a lock-S() on the data item, and each transaction releases the lock a short while after it is granted.
- In between, if any transaction requests for lock-X() but never gets the exclusive-mode lock on the data item, then the transaction may never make progress, and is said to be starved.
- Example: T1 (read A), T'(write A), T2(read A) ….., Tn(read A)

Lock Conversions

- A transaction that already holds a lock on item *X* is allowed under certain conditions to convert the lock from one locked state to another.

- Type of Conversion:
  - Upgrade
  - Downgrade

- For example, it is possible for a transaction *T* to issue a lock-S(*A*) and then later to upgrade the lock by issuing a lock-X(*A*) operation

- It is also possible for a transaction *T* to issue a lock-X(*A*) and then later to downgrade the lock by issuing a lock-S(*A*) operation.

Obtain Conflict-Serializable Schedule

Two-Phase Locking Protocol

- Two-Phase Locking protocol ensures conflict-serializable schedules.
- A transaction is said to follow the two-phase locking protocol if *all* locking operations (lock-S, lock-X) precede the *first* unlock operation in the transaction
- It can be divided into two phases:
  - Phase 1: Growing Phase
    - Transaction may obtain locks
    - Transaction may not release locks
  - Phase 2: Shrinking Phase
    - Transaction may release locks
    - Transaction may not obtain locks
- Initially, a transaction is in the growing phase.
- The protocol assures 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).

Cont…

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

- Transactions *T1* and *T2* are not two-phase
- Transactions *T*3 and *T*4 are two-phase

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

Growing Phase

Shrinking Phase

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Cont…

- Another Example:
  - If we enforce two-phase locking, the transactions (T_1 and T_2) can be rewritten as T_1' and T_2'

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

T_1'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'

read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

Cont…

T_1'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'

read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

Equivalent Concurrent Schedule which is Inconsistent

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);	

- because T_1' will issue its write_lock(X) *before* it unlocks item Y; consequently, when T_2' issues its read_lock(X), it is forced to wait until T_1' releases the lock by issuing an unlock (X) in the schedule.

Conflict-Serializable Schedule

- Lock point: 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 a serializability ordering for the transactions.

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

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 15.8 Partial schedule under two-phase locking.

Shortcomings of Two-Phase Locking

- does *not* ensure freedom from 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)	

Figure 15.7 Schedule 2.

- Cascading rollback may occur under two-phase locking

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)

- failure of *T5* after the read(A) step of *T7* leads to cascading rollback of *T6* and *T7*

Variations of Two-phase Locking

- Strict two-phase locking:
  - Cascading rollbacks can be avoided by this version
  - 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.

- Rigorous two-phase locking:
  - transactions can be serialized in the order in which they commit
  - Cascading rollbacks can be avoided
  - This protocol requires that
    - all locks be held until the transaction commits.

Lock Conversion in Two-phase Locking

- If lock conversion is allowed, then
  - upgrading of locks (from lock-S to lock-X) must be done in the growing phase,
  - downgrading of locks (from lock-X to lock-S) must be done in the shrinking phase

- If we employ the two-phase locking protocol, then *T*8 must lock *a*1 in exclusive mode.

- However, if *T*8 could initially lock *a*1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency,

T_8: read(a_1);
read(a_2);
. . .
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

Cont…

T_8: read(a_1);
read(a_2);
. . .
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

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)	

Figure 15.9 Incomplete schedule with a lock conversion.

- Transactions *T*8 and *T*9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule of Figure 15.9.

Lock Generation Method

- A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction
  - When a transaction T_i issues a *read(Q)* operation, the system issues a *lock-S(Q)* instruction followed by the *read(Q)* instruction.
  - When T_i issues a *write(Q)* operation, the system checks to see whether T_i already holds a *lock-X(Q)*.
    - If it does, then the system issues an *upgrade(Q)* instruction, followed by the *write(Q)* instruction.
    - Otherwise, the system issues a *lock-X(Q)* instruction, followed by the *write(Q)* instruction.
  - All locks obtained by a transaction are unlocked after that transaction commits or aborts.

Summary (of Two-Phase Locking)

- Two-phase locking (with lock conversion) generates conflict-serializable schedules, and transactions can be serialized by their lock points.
- if exclusive locks are held until the end of the transaction, the schedules are cascadeless.
- Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.
- Note: for a set of transactions, there may be conflict-serializable schedules that cannot be obtained through the two-phase locking protocol.
- to obtain conflict-serializable schedules through **non-two-phase locking** protocols, we need
  - either to have additional information about the transactions
  - or to impose some structure or ordering on the set of data items in the database.

Graph-Based Protocol

- 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.

- Example of prior knowledge: partial ordering
  - Let, a partial ordering → on the set **D** = $\{d_1, d_2, \ldots, d_n\}$ of n data items.
  - If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before d_j.
  - partial ordering implies that the set D may now be viewed as a directed acyclic graph, called a database graph

- simple protocol using partial ordering: tree protocol or tree-locking protocol
  - restricted to employ only exclusive locks (lock-X)
  - Each transaction T_i can lock a data item at most once
  - Follow partial ordering
  - Data items may be unlocked at any time.
  - unlocked item cannot subsequently be relocked by same transaction .

Cont…

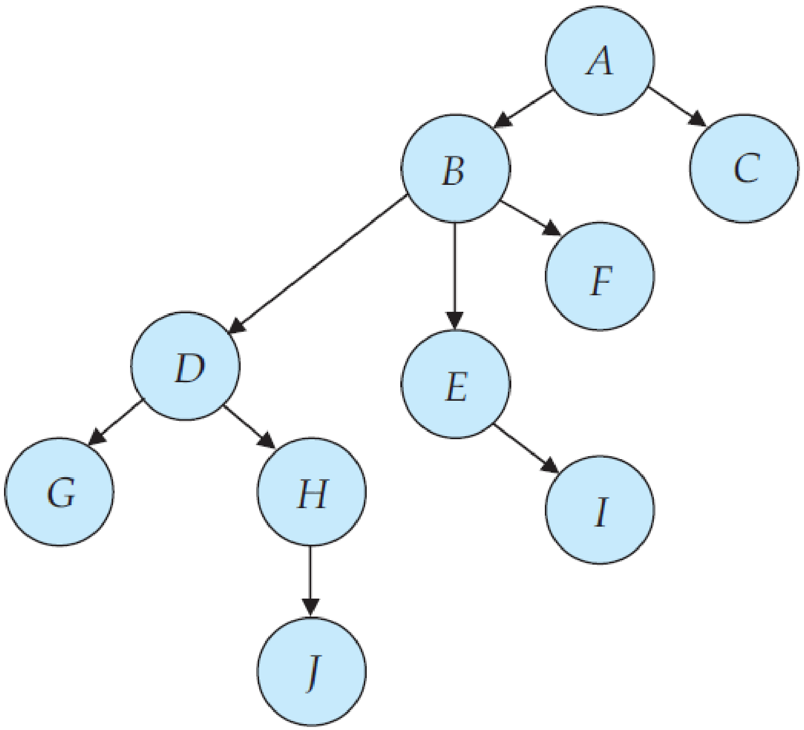

Figure 15.11 Tree-structured database graph.

- Let four transactions follow the tree protocol on this graph.
- T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
- T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
- T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
- T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).
- What is conflict-serializable schedule corresponding to the above transactions?

Cont…

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 15.12 Serializable schedule under the tree protocol.

- The tree protocol ensures conflict serializable, and freedom from deadlock
- The tree protocol does not ensure recoverability and cascadelessness.
- To ensure recoverability and cascadelessness:
  - do not release the exclusive locks until the end of the transaction
  - This approach reduces concurrency
  - Alternative approach: commit dependency

Commit Dependency

- Whenever a transaction T_i performs a read of an uncommitted data item, we record a *commit dependency* of T_i on the transaction that performed the last write to the data item.

- Transaction T_i 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, T_i must also be aborted.

- It improves concurrency than delayed unlock (i.e. at the end of transaction)
- But, it ensures only recoverability but not cascadelessness

Two-phase v/s Tree Locking Protocol

- Advantages of tree-locking over two-phase locking
  - It is deadlock-free,
  - so no rollbacks are required.

  - Unlocking may occur earlier,
  - which may lead to shorter waiting times, and to an increase in concurrency.

- Disadvantages of tree-locking compared to two-phase locking
  - A transaction may have to lock data items that it does not access
    - Example, a transaction that needs to access data items A and J in the database graph (Fig. 15.11) must lock not only A and J, but also data items B, D, and H.
  - So, increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency.

  - Without prior knowledge of what data items will need to be locked, transactions will have to lock the root of the tree !
    - can reduce concurrency greatly

Handling Deadlock and Starvation

Deadlock Handling

- What is deadlock?
  - 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.

- Remedy to deadlock:
  - rolling back some of the transactions involved in the deadlock
  - Rollback of a transaction may be partial i.e. rolled back to the point where it obtained a lock whose release resolves the deadlock.

- Two principal methods for dealing with the deadlock
  - deadlock prevention
    - it ensure that the system will never enter a deadlock state
  - deadlock detection and recovery
    - allow the system to enter a deadlock state, and then try to detect and recover

Deadlock Prevention

- Two approaches to deadlock prevention
  1) ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.

Scheme 1:

- each transaction locks all its data items before it begins execution;
- either all are locked in one step or none are locked

Disadvantages:

- it is often hard to predict, before the transaction begins, what data items need to be locked;
- data-item utilization may be very low, since many of the data items may be locked but unused for a long time

Scheme 2:

- impose an ordering of all data items;
- transaction lock data items only in a sequence consistent with the ordering

Disadvantages:

- it is often hard to predict, before the transaction begins, what ordering is needed

Cont…

2) performs transaction rollback, instead of waiting for a lock, whenever the wait could potentially result in a deadlock.
   - use preemption and transaction rollbacks

- In preemption, when a transaction T_j requests a lock that 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, we assign a unique timestamp 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 schemes using timestamps:
  - Scheme 1: wait–die scheme is a nonpreemptive technique
    - When transaction 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 (that is, T_i is older than T_j). Otherwise, T_i is rolled back (dies).
  - Scheme 2: wound–wait scheme is a preemptive technique:
    - When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp larger than that of T_j (that is, T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is wounded by T_i).
  - Disadvantages of Scheme 1 & 2: unnecessary rollbacks may occur.

Deadlock Detection

- Deadlocks can be described precisely in terms of a directed graph called a wait-for graph
- The set of vertices consists of all the transactions in the system
- Each edges is an ordered pair *Ti* → *Tj*.
- *Ti* → *Tj* implies that transaction *Ti* is waiting for transaction *Tj* to release a data item that it needs
- An edge is inserted and removed dynamically when a request for an item comes from a transaction

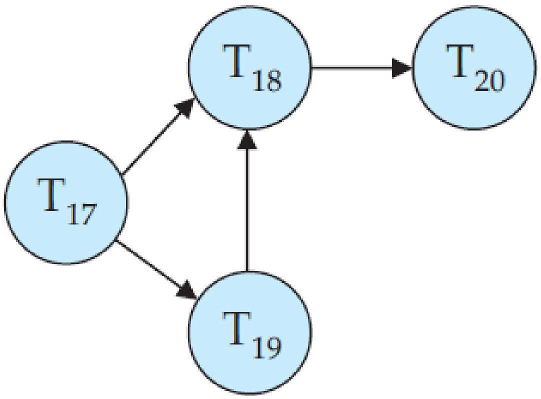

Figure 15.13 Wait-for graph with no cycle.

A deadlock exists in the system if and only if the wait-for graph contains a cycle.

Deadlock Recovery

- When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock
- common solution is to roll back one or more transactions to break the deadlock
- Three actions need to be taken:
  - Selection of a victim: determine which transaction (or transactions) to roll back to break the deadlock
  - Rollback: Once we have decided that particular transaction, we must determine how far this transaction should be rolled back. (either do total rollback or partial rollback)
  - Starvation: it may happen that the same transaction is always picked as a victim. We should have a maximum number of

Starvation Handling

- In lock-based protocol, we can avoid starvation of transactions by granting locks in the following manner:
  - When a transaction T_i requests a lock on a data item *Q* in a particular mode *M (either shared or exclusive)*, 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 T_i .

Thanks!