Lock manager – process that receives messages from transactions and receives replies. Responds to lock request messages with lock-grant or messages to rollback (deadlock).
Acknowledges unlock (may generate lock grant for another process)

When a lock request arrives, 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.

Conflict serializability

Two schedules are conflict equivalent if they order conflicting operations in the same order. A schedule is conflict serializable if it is conflict equivalent to a serial schedule.

Four situations to consider:

Ti =R(Q), Tj=R(Q) – no conflict since the same value of Q is read by both
Ti=R(Q), Tj=W(Q) – if Ii comes before Tj, then Ti does not read the value of Q that is written by Tj. If Tj comes before Ti, it does – thus the order matters.
Ti=W(Q), Tj=R(Q) – same as above
Ti=W(Q), Tj=W(Q) – since both are write, the order does not affect either Ti or Tj. However the NEXT R(Q) will be affected, since which ever comes last is preserved.

This can be checked with a series of swaps of non-conflicting operations.

Testing for conflict serializability

Create a precedence graph

The nodes are the transactions participating in the schedule
An edge from Ti to Tj exists if :

1) Ti executes a W(Q) before Tj executes a R(Q)
2) Ti executes a R(Q) before Tj executes a W(Q)
3) Ti executes a W(Q) before Tj executes a W(Q)

If an edge Ti -> Tj exists in the precedence graph of S, then in any serial schedule equivalent S, Ti must appear before Tj.

View serializability – less stringent than conflict equivalence.

Two schedules are said to be view equivalent if

1) For each data item Q, if Ti reads the initial value of Q in schedule S, then transaction Ti must in schedule S1 also read the initial value of Q
2) For each data item Q, if Ti executes R(Q) in schedule S and if that value was produced by a W(Q) in Tj, then the R(Q) operation of Ti must in schedule S1 also read the value of Q that was produced by the same write(Q) operation of Tj.
3) For each data item Q, the transaction (if any) that performs the final W(Q) operation in schedule S must also perform the final W(Q) operation in S1.

Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and therefore use the same values for computations. Condition 3 ensures the schedules result in the same state.

Recoverability

We can ensure that a transaction T, when committed never has to roll back by ensuring that a schedule is recoverable.

A schedule S is recoverable if no transaction T in S commits until all transactions that have written an item that T reads have committed.

Example:

T1:R(x), T2:R(x), T1:W(x), T1:R(y), T2:W(x), T2:C, T1:W(y), T1:C

Is this schedule recoverable?

Yes, because neither transaction reads data that was written by the other.

Example 2:

T1:R(x), T1:W(x), T2:R(x), T1:R(y), T2:W(x), T2:C, T1:A

Is this schedule recoverable?

No – and shouldn't be allowed. T2 reads a value written by T1 and commits before T1 commits. In this example, T1 aborts showing the problem

Cascading aborts

One of the issues associated with schedules is the issue of cascading aborts. It is desirable to have a schedule that does not allow this. Let's look at the following example:

T1:R(x)
T1:W(x)
T2:R(x)
T2:W(y)
T3:R(y)

If at this point, T1 aborts, T2 has read a value that was written by T1, so due to atomicity property, T2 must also abort. T3 has read a value written by T2, so it must abort.

This phenomenon is called cascading aborts.

Two phase locking protocol

The two phase locking protocol is guaranteed to generate conflict serializable schedules, i.e. schedules that are conflict equivalent to some serial schedule. Locks are acquired during a "growing" phase and released during a "shrinking" phase. Once a lock has been released, no other locks may be acquired.

Strict two phase locking says that no locks may be released until after the transaction commits or aborts. This ensures that cascading aborts do not occur.

Multiple granularity locks

Let's look at the situation where you want to lock groups of items rather than items themselves. This happens frequently – and keeps the protocols we have talked about to date from being successful unless you go through and lock every item in the group as an individual item. We do this by applying locks to a hierarchy. If a parent node is locked (explicit), all children of that node are considered locked as well (implicit).

To accomplish this we introduce 3 new lock types.

IS – Intention-shared lock – if this lock has been applied to a node, explicit locks are being applied lower nodes (children) but only in shared mode locking.
IX – Intention-exclusive lock – if this lock has been applied to a node, explicit locks are being applied to lower nodes (children) with both exclusive and shared mode locking.
SIX – Shared and intention-exclusive lock – if this lock has been applied to a node, 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.

Let's look at the compatability matrix associated with these locks:

	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

The multiple-granularity locking protocol – which ensures serializability – is the following. For Ti to lock node Q, it must follow these rules

1) It must observe the lock-compatibility function of the above table.
2) It must lock the root node first, locking it in any mode.
3) It can lock a node Q in S or IS mode only if it currently has the parent node of Q locked in either IX or IS mode.
4) It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.
5) It can lock a node only if it has not previously unlocked any nodes (i.e. it is 2 phase)
6) It can unlock a node Q only if it currently has not of the children of Q locked.

This protocol is useful for short transactions that only access a few data items and lock transactions that produce reports from an entire set of files.

Upgrading locks

A transaction that wants to read and write an item may first obtain a shared lock on the item for reading, then upgrade it it an exclusive lock for writing.

Example:

T1: SL(A), R(A), SL(B), R(B), XL(B), W(B), U(A), U(B)
T2: SL(A), R(A), SL(B), R(B), U(A), U(B)

```
T1              T2
-----------------------

SL(A)
R(A)
                SL(A)
                R(A)
                SL(B)
                R(B)
SL(B)
R(B)
XL(B) wait
                U(A)
                U(B)
XL(B)
W(B)
U(A)
U(B)
```

Upgrading locks can lead to deadlock,

T1	T2
SL(B)	
R(B)	
	SL(B)
	R(B)
	XL(B) wait
XL(B) wait	

This leads us to the introduction of an Update Lock

An update lock (UL(Q)) allows transaction Ti to read Q but not to write Q.
Only an update lock can be upgraded to a write lock
An update lock can be granted on Q when there are already shared locks on Q.
Once and update lock in on Q, no additional exclusive or update locks are allowed on Q.

Variation
Once an update lock is on Q, no additional locks are allowed on X (prevents starvation)

Example:

T1	T2
UL(B)	
R(B)	
	UL(B) Wait
XL(B) succeeds	

Starvation example:

T1	T2	T3	T4
SL(A)			
R(A)			
	UL(A)		
	R(A)		
	XL(A) – waits		
		SL(A)	
		R(A)	
U(A)			
			SL(A)