

Concurrency Control

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
 - Conflict serializable
 - Recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur
- Testing a schedule for serializability *after* it has executed is a little too late!
 - Tests for serializability help us understand why a concurrency control protocol is correct
- **Goal** – to develop concurrency control protocols that will assure serializability



Concurrency Control

- 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
 - Should a transaction hold a lock on the whole database
 - ▶ Would lead to strictly serial schedules – very poor performance
- The most common method used to implement locking requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item



Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
 1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
 2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.



Lock-Based Protocols (Cont.)

■ Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
 - But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted
- Transaction T_i may unlock a data item that it had locked at some earlier point
- Note that a transaction must hold a lock on a data item as long as it accesses that item
- Moreover, it is not necessarily desirable for a transaction to unlock a data item immediately after its final access of that data item, since serializability may not be ensured



Lock-Based Protocols: Example

- 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 $A + B$
 - Suppose that the values of accounts A and B are \$100 and \$200, respectively

$T1$:	$T2$:
lock-X(B);	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);	

- If these transactions are executed serially, either as $T1, T2$ or the order $T2, T1$, then transaction $T2$ will display the value \$300



Lock-Based Protocols: Example

- If, however, these transactions are executed concurrently, then schedule 1 is possible
- In this case, transaction T_2 displays \$250, which is incorrect. 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
- Suppose we delay unlocking till the end

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($A + B$)
---	---

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)	
	read(A)	grant-S(A, T_2)
	unlock(A)	
	lock-S(B)	
	read(B)	grant-S(B, T_2)
	unlock(B)	
	display($A + B$)	
lock-X(A)		
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		grant-X(A, T_1)

Schedule 1



Lock-Based Protocols: Example

- Delaying unlocking till the end, T1 becomes T3 and T2 becomes T4

T3:	T4:
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)	

- Hence, sequence of reads and writes as in Schedule 1 is no longer possible
- T4 will correctly display \$300



Lock-Based Protocols: Example

- Given, T_3 and T_4 , consider Schedule 2 (partial)
- Since T_3 is holding an exclusive mode lock on B and T_4 is requesting a shared-mode lock on B , T_4 is waiting for T_3 to unlock B
- Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A , T_3 is waiting for T_4 to unlock A
- Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution
- This situation is called **deadlock**
- When deadlock occurs, the system must roll back one of the two transactions.
- Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked
- These data items are then available to the other transaction, which can continue with its execution

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-X(A)	lock-S(A) read(A) lock-S(B)



Lock-Based Protocols

- 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
- Deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states
- Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back transactions, whereas inconsistent states may lead to real-world problems that cannot be handled by the database system
- 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
- The set of all such schedules is a proper subset of all possible serializable schedules
- We present locking protocols that allow only conflict-serializable schedules, and thereby ensure isolation

