Concurrency Control

Dr. Shrutilipi Bhattacharjee, Assistant Professor, Dept. of IT, NIT Karnataka, India

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:
  - **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

- A transaction can unlock a data item *Q* by the **unlock**(*Q*) Instruction

- Lock requests are made to concurrency-control manager by the programmer

- Transaction can proceed only after request is granted

Lock-Based Protocols

- **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 T_1 and T_2
  - Transaction T_1 transfers \$50 from account *B* to account *A*
  - Transaction T_2 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

T_1:	T_2:
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 T_1, T_2 or the order T_2, T_1, then transaction T_2 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, and 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)	
		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)		

Schedule 1

Lock-Based Protocols: Example

- Delaying unlocking till the end, T_1 becomes T_3 and T_2 becomes T_4

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)

- Hence, sequence of reads and writes as in Schedule 1 is no longer possible
- T_4 will correctly display $300

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_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Schedule 1

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

Schedule 2

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

The Two-Phase Locking Protocol

- This protocol ensures conflict-serializable schedules

- 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

- The protocol assures serializability
- It can be proved that the transactions can be serialized in the order of their **lock points**
  - That is, the point where a transaction acquired its final lock

The Two-Phase Locking Protocol

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used

- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:
  – Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - Can acquire a lock-s on item
    - Can acquire a lock-x on item
    - Can convert a lock-s to a lock-x (upgrade)
  - Second Phase:
    - Can release a lock-s
    - Can release a lock-x
    - Can convert a lock-x to a lock-s (downgrade)
- This protocol assures serializability
- But still relies on the programmer to insert the various locking instructions

Automatic Acquisition of Locks: Read

- A transaction T_i issues the standard read/write instruction, without explicit locking calls

- The operation **read**(D) is processed as:

  if T_i has a lock on D **then**

  read(D)

  else begin

  if necessary wait until no other transaction has a **lock-X** on D

  grant T_i a **lock-S** on D;

  read(D)

  end

Automatic Acquisition of Locks: Write

- The operation **write**(*D*) is processed as:

    if T_i has a **lock-X** on *D* **then**
        write(*D*)
    else begin
        if necessary wait until no other transaction has any lock on *D*,
            if T_i has a **lock-S** on **then**
                upgrade lock on *D* to **lock-X**
            else
                grant T_i a **lock-X** on *D*
                write(*D*)
    end;

- All locks are released after commit or abort

Deadlocks

- Two-phase locking *does not* ensure freedom from deadlocks

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)	

- Observe that transactions T_3 and T_4 are two phase, but, in deadlock

Starvation

- In addition to deadlocks, there is a possibility of **starvation**

- **Starvation** occurs if the concurrency control manager is badly designed, for example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item
  - The same transaction is repeatedly rolled back due to deadlocks

- Concurrency control manager can be designed to prevent starvation

Cascading Roll-back

- The potential for deadlock exists in most locking protocols
  - Deadlocks are a necessary evil
- When a deadlock occurs there is a possibility of cascading roll-backs
- Cascading roll-back is possible under two-phase locking
- In the schedule here, each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7

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)

More Two Phase Locking Protocols

- To avoid Cascading roll-back, follow a modified protocol called **strict two-phase locking**
  - A transaction must hold all its exclusive locks till it commits/aborts

- **Rigorous two-phase locking** is even stricter
  - *All* locks are held till commit/abort
  - In this protocol transactions can be serialized in the order in which they commit

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests

- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)

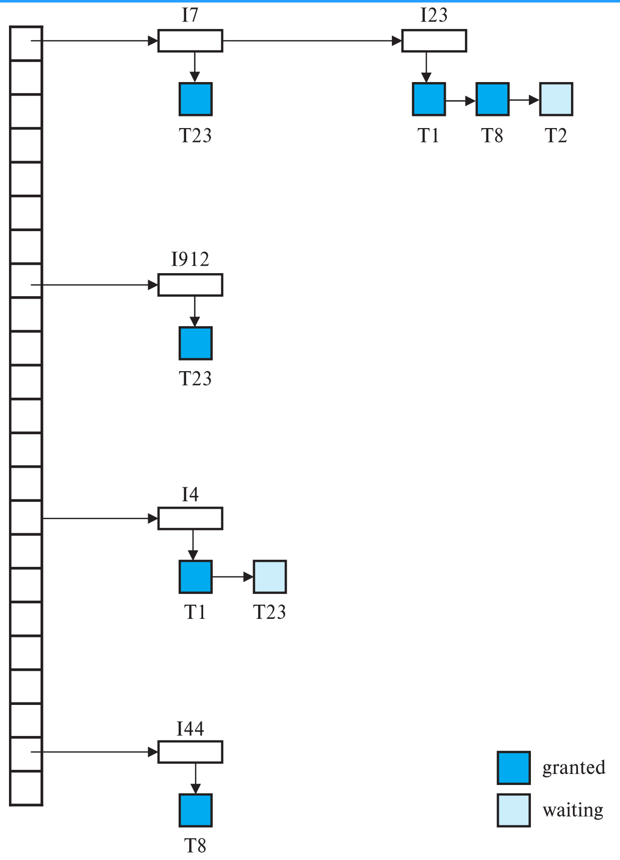
- The requesting transaction waits until its request is answered

- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests

- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table

- Dark rectangles indicate granted locks, light colored ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - Lock manager may keep a list of locks held by each transaction, to implement this efficiently



Next Lecture

Concurrency Control

Thank you for your attention...

Any question?

Contact:
Department of Information Technology, NITK Surathkal, India
6th Floor, Room: 13
Phone: +91-9477678768
E-mail: shrutilipi@nitk.edu.in