Concurrency Control

- **A database must provide a mechanism that will ensure that all possible schedules are:**
  - **Either conflict or view serializable**
  - **Are 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**
  - **Are serial schedules recoverable/cascadeless**
- **Testing a schedule for serializability *after* it has executed is a little too late!**
- **Goal – to develop concurrency control protocols that will assure serializability.**

Concurrency Control

- **Lock-Based Protocols**
- **Timestamp-Based Protocols**
- **Validation-Based Protocols**
- **Multiple Granularity**
- **Multi-Version Schemes**
- **Insert and Delete Operations**
- **Concurrency in Index Structures**

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…Contd.

- 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.

Lock-Based Protocols…Contd.

- Example of a transaction performing locking:

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

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.

- 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.

Pitfalls of Lock-Based Protocols

- Consider the partial 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)	

- Neither T_3 nor T_4 can make progress — Executing **lock-S**(B) causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X**(A) causes T_3 to wait for T_4 to release its lock on A.

- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols…contd.

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

- **Starvation** is also possible if 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.

The Two-Phase Locking Protocol

- **This is a protocol which 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 (i.e. the point where a transaction acquired its final lock).**

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

- **Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.**
- **Rigorous two-phase locking is even stricter: Here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.**

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

- 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 …contd.

- **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 trans. has any lock on *D*,
   if T_i has a **lock-S** on *D*
    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