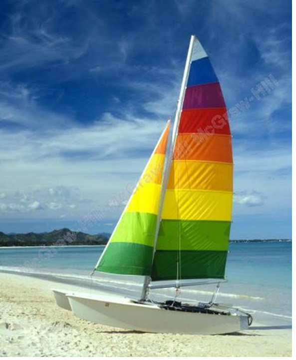

Chapter 15 : Concurrency Control

Edited by Radhika Sukapuram

Database System Concepts, 6th Ed.

Concurrency Control

- Management of concurrently executing transactions
- Ignoring failures

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 the concurrency-control manager by the programmer. 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.

Lock-Based Protocols (Cont.)

- 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.

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** (i.e., the point where a transaction acquired its final lock). Proof ?

Lock Conversions

- T8: r8(a),r8(b),w8(a)
- T9: r9(a),r9(b), display9(a+b)
  - T8 will lock a in X mode. T9 cannot read a
- 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 relies on the programmer to insert the various locking instructions.

Deadlocks

- 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.

Deadlocks (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks.
  - T_3 and T_4 are in 2PL, but deadlocked
- 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.
  - When a transaction T_i requests for a lock on data item Q
    - , a lock is granted if no other transaction is waiting for a lock on Q that made its lock request before T_i

Deadlocks (Cont.)

- 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.

Strict 2PL

T5	T6	T7
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)

T5 aborts

- Cascading roll-back is possible under two-phase locking.
  - To avoid this, follow a modified protocol called **strict two-phase locking** -- a transaction must hold all its exclusive locks till it commits/aborts.
  - Assume T1(a) releases its s-lock(a) immediately after r1(a), S2 follows strict 2PL:
    - S2: r1(a) r2(a) w2(a) commit2 commit1

Rigorous 2PL

- **Rigorous two-phase locking** is even stricter.
  - *all* locks are held till commit/abort.
  - transactions can be serialized in the order in which they commit.
  - S3: r1(a) r2(a) commit1 w2(a) commit2