Chapter 10(contd..): Schedules and Serializability
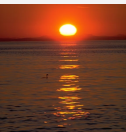# Concepts of locking for concurrency control
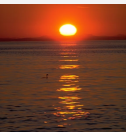
Database System Concepts, 5th Ed.

Chapter 10(contd..): Schedules and Serializability
# Concurrency Control

- Concurrent Executions
- Schedules
- Serializability
- Lock-Based Protocols
- Multiple Granularity

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement
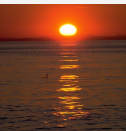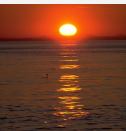
Schedule 1

- Let T_1 transfer $50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.*

- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
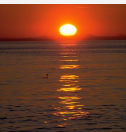
Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of $(A + B)$.

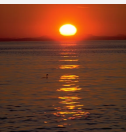| T_1	T_2
read(A) $A := A - 50$	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B)
write(A) read(B) $B := B + 50$ write(B)	
	$B := B + temp$ write(B)

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.

- Thus serial execution of a set of transactions preserves database consistency.

- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**

2. **view serializability**

- *Simplified view of transactions*

  - We ignore operations other than **read** and **write** instructions

  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.

  - Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.

    1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
    2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
    3. I_i = **write**(Q), I_j = **read**(Q).   They conflict
    4. I_i = **write**(Q), I_j = **write**(Q). They conflict

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.

    - If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule $S´$ by a series of swaps of non-conflicting instructions, we say that S and $S´$ are **conflict equivalent**.

- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

■ Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.

● Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializability (Cont.)

■ Example of a schedule that is not conflict serializable:

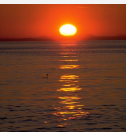| T_3	T_4
read(Q)	
	write(Q)
write(Q)	

■ We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are

    - either conflict or view serializable, and

    - 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 vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .

- Concurrency control protocols generally do not examine the precedence graph as it is being created

  - Instead a protocol imposes a discipline that avoids nonseralizable schedules.

  - We study such protocols in Chapter 16.

- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.

- Tests for serializability help us understand why a concurrency control protocol is correct.
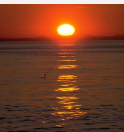
Chapter 10(contd..) : Concurrency Control

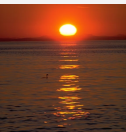**Jan 26, 2014**

Database System Concepts, 5th Ed.

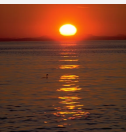# Chapter 10(contd..): Concurrency Control

- Lock-Based Protocols
- Multiple Granularity

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.
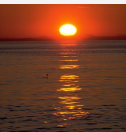
Lock-Based Protocols (Cont.)

■ 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.
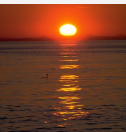
Lock-Based Protocols (Cont.)

- 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.
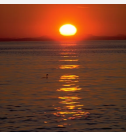
Pitfalls of Lock-Based Protocols

- Consider the partial schedule

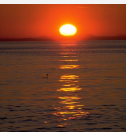| T_3	T_4
lock-x(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-s(A)
	read(A)
	lock-s(B)
lock-x(A)	

- 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 (Cont.)

■ 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).
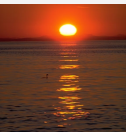
The Two-Phase Locking Protocol (Cont.)

- 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.

The Two-Phase Locking Protocol (Cont.)

■ 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:
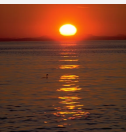
Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.
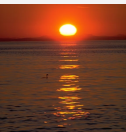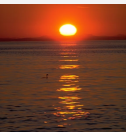
Lock Conversions

■ 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.

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
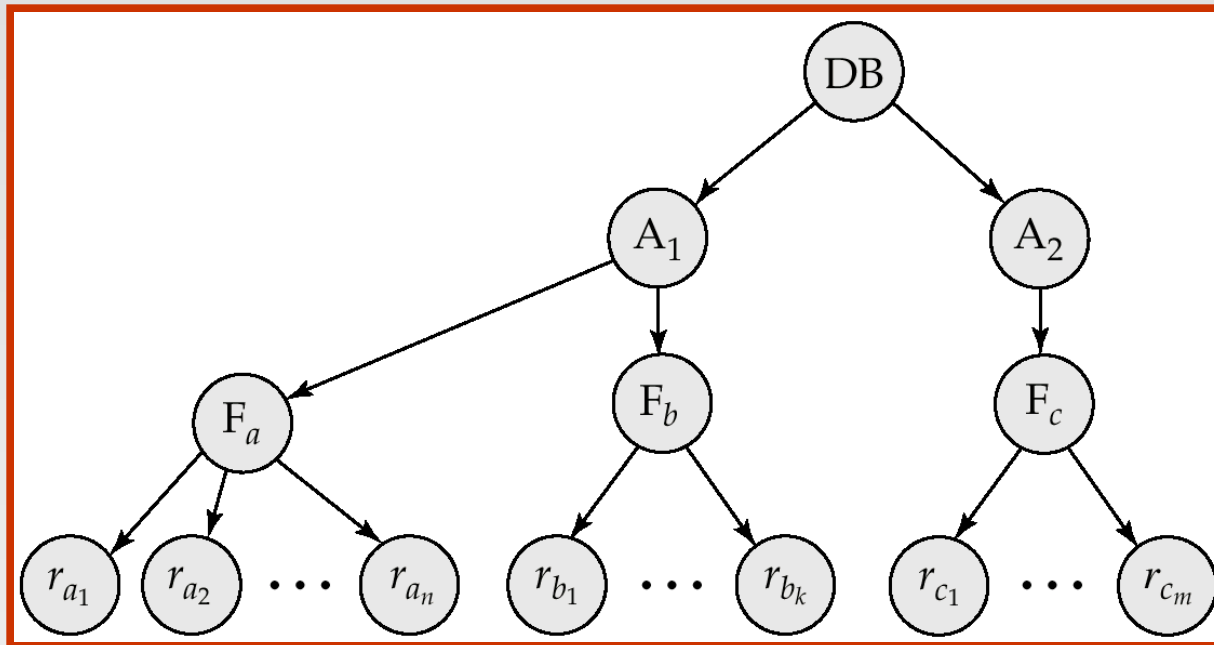
Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones

- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)

- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendents in the same mode.

- Granularity of locking (level in tree where locking is done):

  - **fine granularity** (lower in tree): high concurrency, high locking overhead

  - **coarse granularity** (higher in tree): low locking overhead, low concurrency
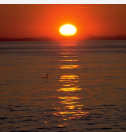
Example of Granularity Hierarchy

The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

  - *intention-shared* (IS): indicates explicit locking at a lower level of the tree but only with shared locks.

  - *intention-exclusive* (IX): indicates explicit locking at a lower level with exclusive or shared locks

  - *shared and intention-exclusive* (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.
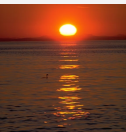
Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	✕
IX	✓	✓	✕	✕	✕
S	✓	✕	✓	✕	✕
S IX	✓	✕	✕	✕	✕
X	✕	✕	✕	✕	✕

Multiple Granularity Locking Scheme

■ Transaction T_i can lock a node Q, using the following rules:

- The lock compatibility matrix must be observed.

- The root of the tree must be locked first, and may be locked in any mode.

- A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.

- A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.

- T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).

- T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

■ Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.