Chapter – 10: Concurrency Control

Introduction

- The process of managing simultaneous operations on the database without having them interfere with one another.
- DBMSs are required to allow simultaneous, (concurrent), multi-user access
    - multiple users within the one application accessing the same files.
    - users in different applications accessing the same files.
- Without centralised concurrency control, two types of problem can occur:
- The lost update problem.
- The so-called “ dirty-read” problem, also described as allowing a transaction to view the partial, (uncommitted), results of another transaction.

Concurrency Control in a Multi-user Environment The *'lost update'* problem - sample scenario

- Consider the relation:
- PRODUCT(prod-code,description,unit-price, quantity-onhand)
- Assume a particular record in the file is:

P45 Wheelbarrow $25.50 35

- Concurrent User 1, (in the Warehouse), has just received a shipment of twenty P45s.
- Concurrent User 2, (a salesman), has just received an order for ten P45s.
- CU1, (Concurrent User 1), and CU2 initiate transactions which each read the same P45 record. In each CWA (currentwork area), the field Q-O-H, Quantity-On-Hand, shows 35.
- CU2's transaction reduces Q-O-H by 10, and rewrites the record, (overwriting the old record), with Q-O-H of 25.
- CU1's transaction, a moment later, increments Q-O-H by 20 to 55, and rewrites the record, overwriting the existing record, (the one just written by CU2's transaction).
- Quantity-On-Hand thus shows 55, when it should show
    - 35 - 10 + 20, i.e. 45. CU2's update has been " lost".

Concurrency Control in a Multi-user Environment The *"dirty-read"* problem - similar sample scenario

- Consider again the relation:
- PRODUCT(prod-code,description,unit-price, quantity-onhand)
- Assume again a particular record in the file is:

P45 Wheelbarrow $25.50 35

- Concurrent User 1, (in the Warehouse), has just received a shipment of twenty P54s. (Note: P54s – not P45s.)
- Concurrent User 2, (a salesman), has just received an order for ten P45s.
- CU1 initiates the Warehouse update transaction, but by mistake keys in "P45", instead of " P54". The transaction reads the P45 record with Q-O-H of 35, updates it to 55, and rewrites it (in a shared memory area for example).
- A moment later, CU2's transaction reads CU1's update showing the value of Q-O-H as 55.
- Just then, CU1, realising the error, aborts the transaction, which executes rollback.
- A moment later, CU2's transaction, having decremented QO- H, rewrites it as 45.
- Quantity-On-Hand thus shows 45, when it should show 35 - 10 = 25. By viewing the uncommitted results of CU1's

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
  - Will study in Chapter 14, after studying notion of correctness of concurrent executions.

Schedules

- *Schedules* – sequences that indicate 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.

Example Schedules

- Let T_1 transfer \$50 from *A* to *B*, and T_2 transfer 10% of the balance from *A* to *B.* The following is a serial schedule (Schedule 1 in the text), 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)

Example Schedule (Cont.)

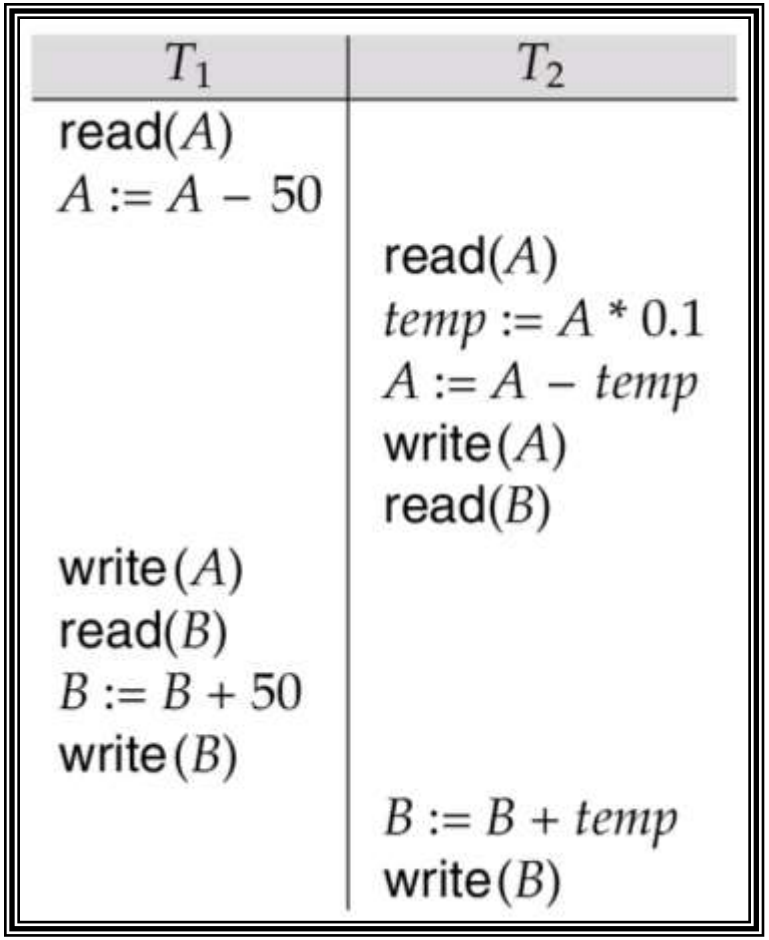
- Let T_1 and T_2 be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

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 both Schedule 1 and 3, the sum A + B is preserved.

Example Schedules (Cont.)

- The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum $A + B$.



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

Conflict Serializability

- 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 (Cont.)

- 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
- 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 >.

Conflict Serializability (Cont.)

- Schedule 3 below can be transformed into Schedule 1, 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)

View Serializability

- Let *S* and *S´* be two schedules with the same set of transactions. *S* and *S´* are **view equivalent** if the following three conditions are met:
  1. For each data item *Q,* if transaction T_i reads the initial value of *Q* in schedule *S,* then transaction T_i must, in schedule *S´*, also read the initial value of *Q.*
  2. For each data item *Q* if transaction T_i executes **read**(*Q)* in schedule *S*, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule *S´* also read the value of *Q* that was produced by transaction T_j .
  3. For each data item *Q*, the transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must perform the final **write**(*Q*) operation in schedule *S´*.

As can be seen, view equivalence is also based purely on **reads**

and **writes** alone.

View Serializability (Cont.)

- A schedule *S* is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Schedule 9 (from text) — a schedule which is view-serializable but *not* conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Every view serializable schedule that is not conflict serializable has **blind writes.**

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.

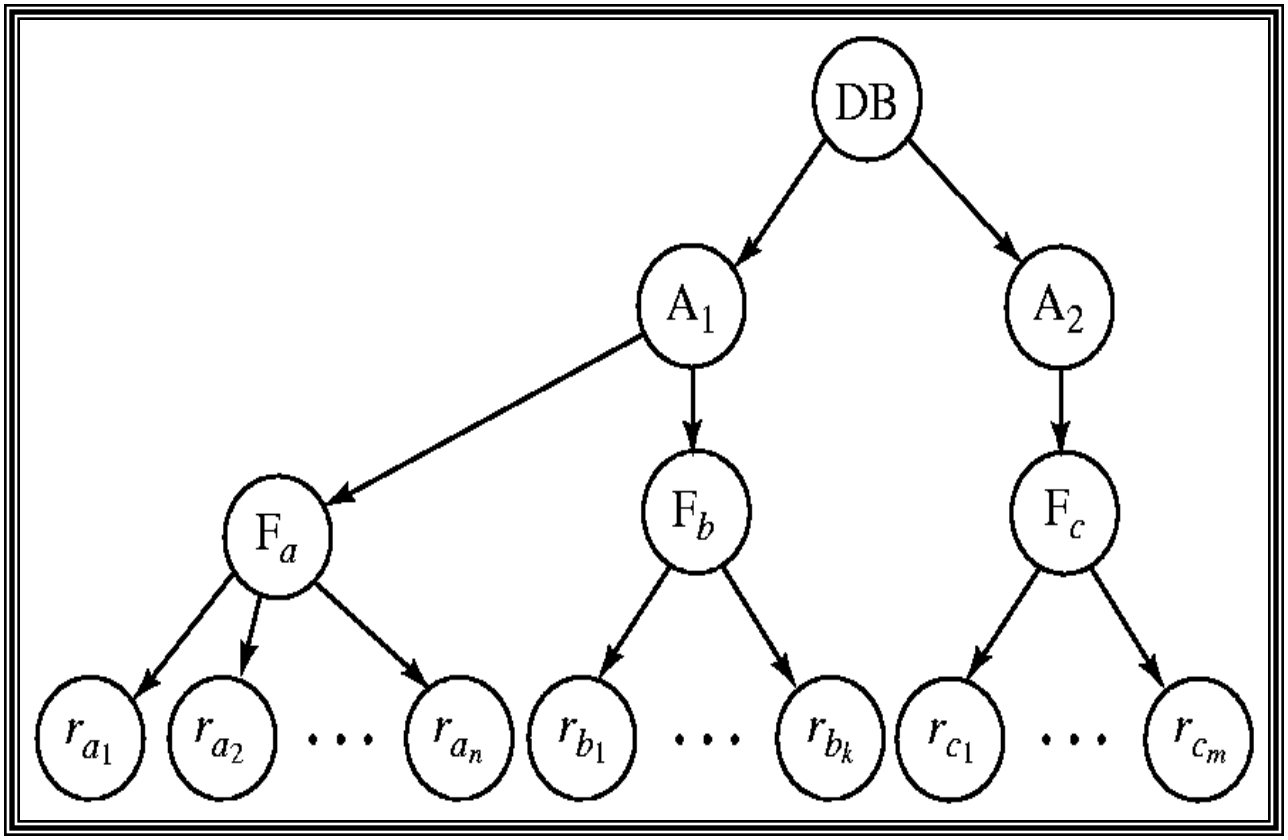
Pitfalls of Lock-Based Protocols (Cont.)

- 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.

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 highest level in the example hierarchy is the entire database.

The levels below are of type *area*, *file* and *record* in that order.

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:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. 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.
  4. 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.
  5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
  6. 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.