DATABASE SYSTEMS

CS – 355/CE – 373

Instructor: Maria N. Samad

November 6th, 2024

CONCURRENCY CONTROL SCHEMES

- One of the fundamental properties of a transaction is ***isolation***
- When several transactions execute concurrently in the database, the isolation property may no longer be preserved
- To ensure isolation, the system must control the interaction among the concurrent transactions
- This control is achieved through ***concurrency-control schemes***
- Cascadeless schedules overcome the temporary update problem, and ensures isolation property, as a new transaction doesn’t start its operations until the previous transaction has completed its task

CONCURRENCY CONTROL SCHEMES

- Example:
  - For the given schedule, check if it's in recoverable state or not? If not, get its cascadeless solution
- Solution:
  - On board

T_1	T_2
read(B)	
$B := B - 50$	
write(B)	
	read(B)
	read(A)
	display($B + A$)
	commit
read(A)	
$A := A + 50$	
write(A)	
commit	

CONCURRENCY CONTROL SCHEMES

- We can achieve a cascadeless schedule of the example in previous slide, but how do we make the system behave such that T_2 stops its execution until T_1 has finished execution?
  - We use ***locks***

LOCKS

- The most common method to implement concurrency control is using a ***lock***
- A lock is a mechanism which provides access privileges to a transaction on a shared resource
- While the lock is acquired, no other transaction can access that resource.
- In order to release the resource, i.e. to make it available to other transactions, a transaction has to ***unlock*** the resource.

LOCK MODES

- We present two modes in which a data item may be locked.
  - **Shared**. If a transaction T_i has obtained a ***shared-mode*** lock (***S***) on item *Q*, then T_i can read, but cannot write *Q*.
  - **Exclusive**. If a transaction T_i has obtained an ***exclusive-mode*** lock (***X***) on item *Q*, then T_i can both read and write *Q*.
- Every transaction must request to *the **concurrency-control manager***.
- The transaction can proceed with the operation only after the concurrency-control manager ***grants*** the lock to the transaction.

LOCK COMPATIBILITY

- Given a set of lock modes, we can define a compatibility function on them as follows:
- Let *A* and *B* represent arbitrary lock modes.
- Suppose that a transaction T_i requests a lock of mode *A* on item *Q* on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode *B*.
- If transaction T_i can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is ***compatible*** with mode *B*.

LOCK COMPATIBILITY

- Such a function can be represented conveniently by a matrix.
- The ***compatibility relation*** between the two modes of locking discussed in this section appears in the matrix comp below.
- An element ***comp***(*A*, *B*) of the matrix has the value true if and only if mode *A* is compatible with mode *B*.

	S	X
S	true	false
X	false	false

TRANSACTIONS WITH LOCKS

- Consider the following transactions T_1 and T_2 from the example on previous slides

- The objective is to transfer 50 units from *B* to *A*, and then display *B* + *A*.

- To maintain consistency, *B* + *A* must be the same before and after the transactions. (Let *A* = 100, *B* = 200, then *A* + *B* = 300).

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

TRANSACTIONS WITH LOCKS

- As done in the example in the earlier slides, one Interleaved/Concurrent schedule may look like the schedule given on the right

- We can introduce locks before every read/write operation

T_1	T_2
read(B)	
$B := B - 50$	
write(B)	
	read(B)
	read(A)
	display($B + A$)
	commit
read(A)	
$A := A + 50$	
write(A)	
commit	

AN INCONSISTENT SCHEDULE

- The schedule with locks look like the schedule given on the right
- However, this does not resolve the temporary update problem
- Also, despite having the acquisition of locks the following schedule results in an inconsistent state
- This is because we simply added the locks without taking inconsistency problems into consideration, like lost update or temporary update or incorrect summary problems

T_1	***T_2***	***Concurrency-Control Manager***
lock-S(B)		grant-S(B, T_1)
read(B)		
B:= B - 50		
lock-X(B)		upgrade to exclusive lock on B, i.e. grant-X (B, T_1)
write(B)	B = 150	
unlock(B)		
	lock-S(B)	grant-S(B, T_2)
	read(B)	
	unlock(B)	
	lock-S(A)	grant-S(A, T_2)
	read(A)	
	unlock(A)	
A + B = 250	display(B + A)	
	Commit	
lock-S(A)		grant-S(A, T_1)
read(A)		
A:= A + 50		
lock-X(A)		upgrade to exclusive lock on A, i.e. grant-X(A, T_1)
write(A)		
unlock(A)		
Commit		

A CONSISTENT SCHEDULE

- Update to have consistent schedule
- A simple rule is to **<u>not</u>** grant any type of lock to any transaction while the previous transaction is executing. A transaction “delays” the unlocking until it’s in a partially committed state
- Once the first transaction has written and released the lock, the second transaction may get access.
- For both transactions reading, that will not be an issue, as long as there is no write-read dependencies
- This forms a series of operations that resemble the solution of cascadeless schedules

A CONSISTENT SCHEDULE

T_1	T_2
read(B)	
$B := B - 50$	
write(B)	
	read(B)
	read(A)
	display($B + A$)
	commit
read(A)	
$A := A + 50$	
write(A)	
commit	

T_1	T_2	Concurrency-Control Manager	Instruction Queue
lock-S(B)		grant-S(B, T_1)	
read(B)			
B:= B - 50			
lock-X(B)		upgrade to exclusive lock on B, i.e. grant-X (B, T_1)	
write(B)			
	lock-S(B)	Not granted →	[read(B)]
	lock-S(A)	Not granted →	[read(B), read(A), display(B + A), commit2]
lock-S(A)		grant-S(A, T_1)	
read(A)			
A:= A + 50			
lock-X(A)		upgrade to exclusive lock on A, i.e. grant-X(A, T_1)	
write(A)			
unlock(B)			
unlock(A)			
Commit			
	lock-S(B)	grant-S(B, T_2)	
	read(B)		[read(A), display(B + A), commit2]
	lock-S(A)	grant-S(A, T_2)	
	read(A)		[display(B + A), commit2]
	display(B + A)		[commit2]
	unlock(B)		
	unlock(A)		
	Commit		[ ]

DELAYED UNLOCKING – CONSISTENT SCHEDULE

- With unlocking delayed, transactions T_3 and T_4 lead to a consistent schedule.
- It also becomes a serial schedule, giving consistent results
- However, this may cause performance issues. For example, if T_3 is being delayed due to IO operation, T_4 would be idle for long time too

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).

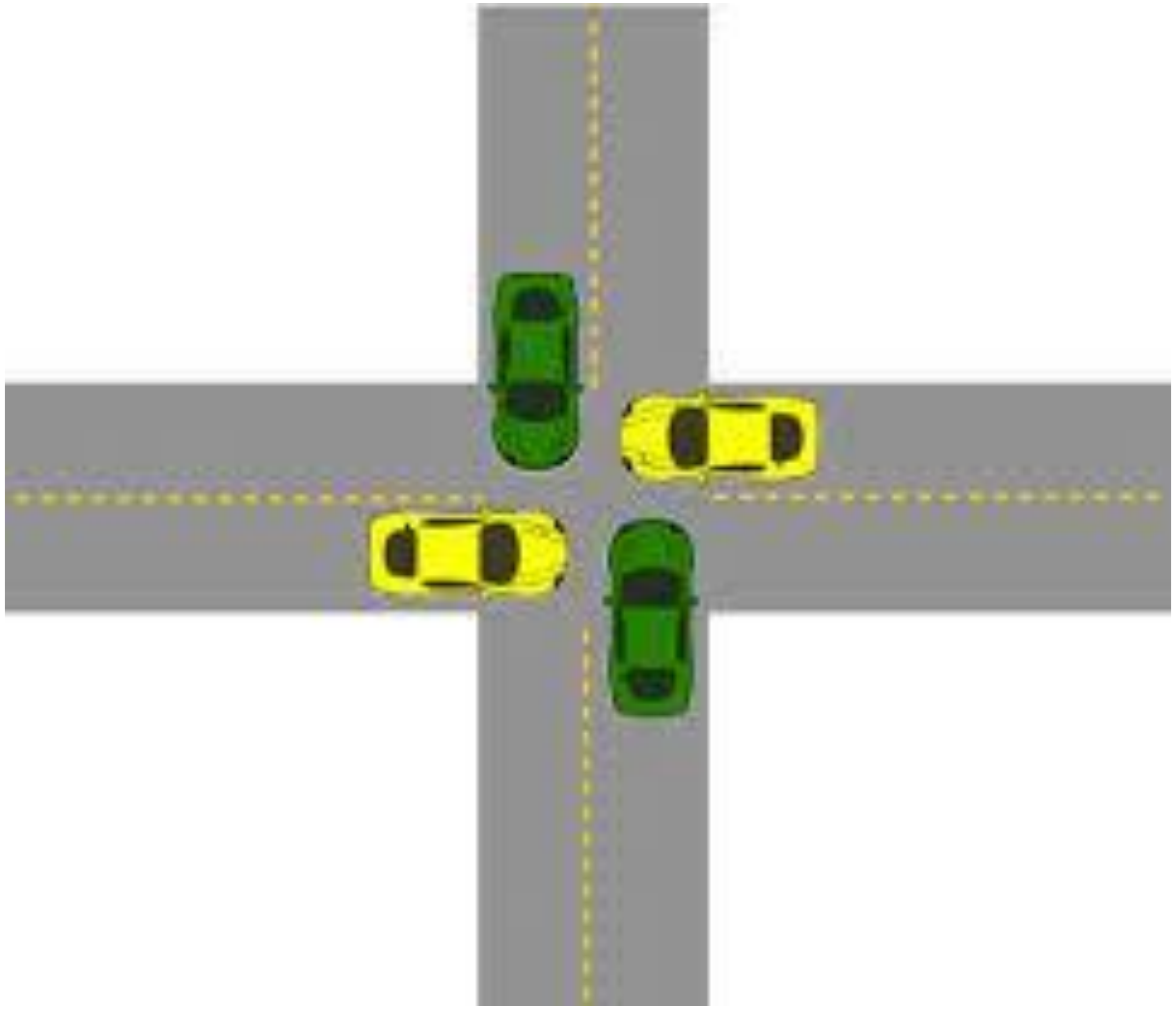
DEADLOCKS

- Unfortunately, delayed locking can also lead to an undesirable situation.
- Consider the partial schedule as shown here for T_3 and T_4.
- 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***.

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)	

Figure 18.7 Schedule 2.

DEADLOCK (TRAFFIC EXAMPLE)



DEADLOCKS (EXAMPLE)

- Example:
  - S: r1(A), w1(A), r2(B), r3(B), r3(A), r1(B), w1(B), C1, r2(A), r3(A), C2, C3
  - Using the delayed locking strategy, check if the given schedule ends in a deadlock state or not?
- Solution:
  - On board

T1	**T2**	**T3**
r(A)		
w(A)		
	r(B)	
		r(B)
		r(A)
r(B)		
w(B)		
Commit		
	r(A)	
		r(A)
	Commit	
		Commit

DEADLOCKS

- Activity Sheet

DEADLOCKS

- Activity Sheet Solution:
  - Q1) Deadlock present
  - Q2) No deadlock and resolves fine