Database Systems (CSF212)

BITS Pilani
Hyderabad Campus

Prof.R.Gururaj
CS&IS Dept.

Concurrency Control (Ch.21 of T1)

- Problems with concurrency
- Lock-Based Protocols
- Deadlock Condition and prevention
- Deadlock detection.
- Timestamp-based Protocols

Why Concurrency control is needed?

- **The Lost Update Problem**
  This occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database item incorrect.
- **The Temporary Update (or Dirty Read) Problem**
  This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4). The updated item is accessed by another transaction before it is changed back to its original value.
- **The Incorrect Summary Problem**
  If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

Problems with Concurrency

Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(a)

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(b)

Time

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)

T_1	T_3
	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

- In a DBMS multiple transactions are executed concurrently.

- If the transactions are executed concurrently then the resources can be utilized more efficiently hence more throughput is achieved.

- Here, for transactions we consider data items as resources because transactions process data by accessing them.

- When multiple transactions access data elements in a concurrent way, this may destroy the consistency of the database.

Implementing Serializabilty

One way to ensure *serializability* is to allow the transactions to access the data items in a mutually exclusive manner.

This is to make sure that when one transaction access a data item no other transaction can modify that data item.

The following techniques implement mutual exclusion and control concurrency.

1. *Lock-based protocols*
2. *Timestamp-based protocols*

S_1

T_1	T_2
lock(A)	
R(A)	
	lock(B)
	R(B)
	B=B+20
A=A+40	
	Lock(C)
	R(C)
	W(B)
	Lock(A)
	R(A)
	Release(B)
	W(C)
	Release(C)
W(A)	
Release(A)	
	A=A+400
	W(A)
	Release(A)

S_2

Implementing serializability using locks

T_1	T_2
lock(A)	
R(A)	
	lock(B)
	R(B)
	B=B+20
A=A+40	
	lock(C)
	R(C)
	W(B)
W(A)	lock
Release(A)	
	lock(A)
	R(A)
	Release(B)
	W(C)
	Release(C)
	A=A+400
	W(A)
	Release(A)

1. ***Concurrency Control Using* Locks**: A data item may be locked in various modes.

A *lock* is a variable associated with a data item.

i) ***Shared*** (denoted by *S*): if a transaction obtains a shared mode lock on a data item Q, it can read Q but not modify Q.

(ii) ***Exclusive*** (denoted by *X*): if this lock is obtained, a transaction can read or write the data item.

	S	X
S	true	false
X	false	false

Lock compatibility matrix

- 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

Two-Phase Locking (2PL) 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.
- Two-phase locking *does not* ensure freedom from deadlocks

- **Basic 2PL**: explained earlier. It is deadlock prone.
- **Conservative 2PL:** a transaction requires to lock all its data items before the transaction begins. (not the case with Basic 2PL). Hence deadlock free.
- **Strict 2PL:** a transaction will not release any of its Exclusive lock before it commits or aborts. Hence always recoverable. Most popular.
- **Rigorous 2PL:** a transaction will not release any of its lock (S or X) before it commits or aborts. Hence always recoverable.

Deadlock

- 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.

Deadlock Handling

- Consider the following two transactions:

  T_1: write (A) T_2: write(B)

  write(B) write(A)

- Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
    - Require that each transaction locks all its data items before it begins execution.

Ex: predeclaration- conservative 2PL.

Schemes for Deadlock Resolution

- Following schemes use transaction timestamps/priority for the sake of resolving deadlock.
- **wait-die** scheme — non-preemptive
  - Older/high-priority transaction may wait for younger/low-priority one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - Older/high-priority transaction *wounds* (forces rollback) of younger/low-priority transaction instead of waiting for it. Younger transactions may wait for older ones.

Deadlock Detection

innovate achieve lead

Wait-for Graph

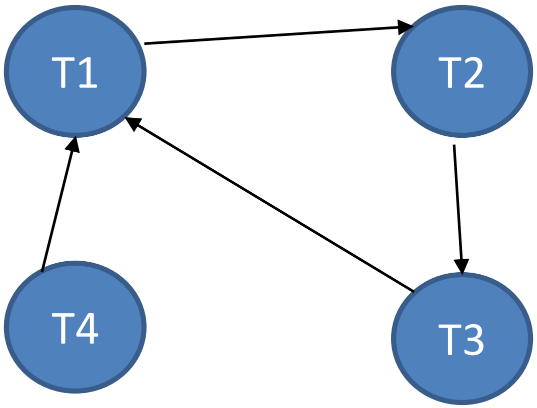
Deadlock condition can be determined by a *wait-for* graph.

All transactions of the schedule become vertices.

And we have an edge between two transactions T_i and T_j. if T_i is waiting for T_j to release a lock on a data item.
If the graph has a cycle then we can say that the schedule will result in a deadlock.

T_1	T_2	T_3	T_4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

S(A) means transaction locks A in share mode
R(A) – transaction reads A
X(C) – Transaction locks C in X-mode
W(B) – transaction write B



Held by & mode	Data Item	1	2	3
T1-S	A	T3-X		
T2-X	B	T1-s	T4-X	
T3-S	C	T2-X		

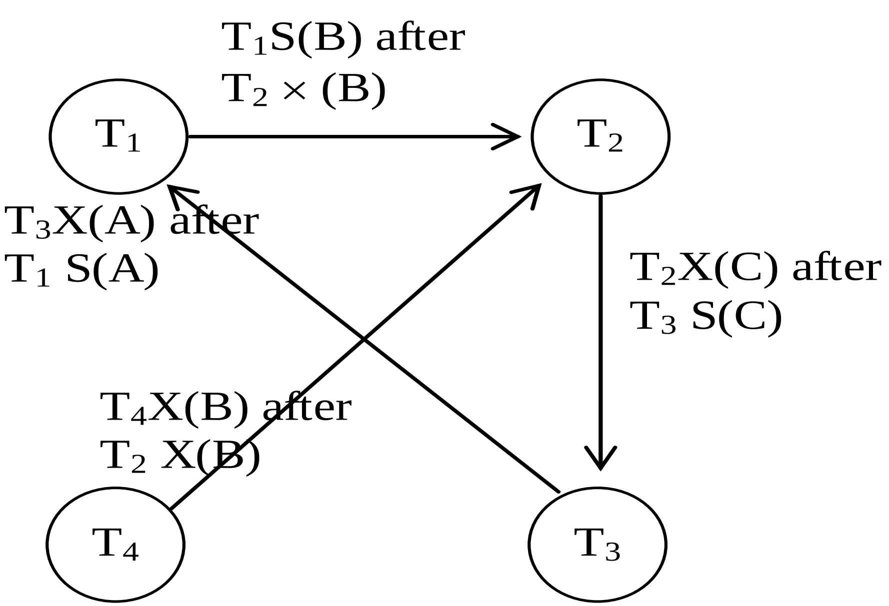
T_1	T_2	T_3	T_4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

S(A) means transaction locks A in share mode

R(A) – transaction reads A

X(C) – Transaction locks C in X-mode

W(B) – transaction write B



In the above graph there exists a cycle hence this schedule leads to deadlock.
If a transaction T_i requests a lock and transaction T_j holds a conflicting lock.

Timestamp-based Concurrency Control

Maintaining the ordering between every pair of conflicting transactions is significant.

If we select the ordering in advance, we can achieve serializability. Time-stamping is a method to fix the ordering.

Each transaction is assigned a unique fixed timestamp.
If $TS(T_i) < TS(T_j)$, this implies that T_i should be executed before T_j.

The time-stamps determine the serializability order.
Each data item is associated with two timestamp values.

W-timestamp(Q) – represents the largest timestamp of any transaction that successfully executes Write(Q).

R-timestamp(Q) - which denotes the largest time stamp of any transaction that successfully executed Read(Q).

These values are updated whenever read(Q) or write(Q) are executed.

Basic Timestamp Ordering (TO) protocol

This protocol operates as follows:

RTS 139

WTS RTS

i)*Suppose Transaction T_i* issues read(Q) 128 121

If $TS(T_i) <$ W-stamp(Q), then it implies that T_i need to read Q which was already overwritten.

Hence read operation is rejected and T_i is rolled back.

If $TS(T_i) \geq$ W-timestamp (Q) then read operation is executed.

Wts 120 RTs 140

100

ii)*Suppose T_i issues write (Q)*

If $TS(T_i)$ < R-timestamp(Q) it implies that the value of Q being produced by T_i had to be written long back.
Hence reject T_i & roll back.
If $TS(T_i)$ < W-timestamp(Q), T_i is attempting to write some absolute value of Q.
Hence reject T_i & roll back.
Otherwise write operation is executed.

Note: It is guaranteed to be conflict serializable. But may not be recoverable.

Data item A :

Transaction →

TS : 450
Requesting : Read(A)

	Case 1	Case 2	Case 3	Case 4
RTS :	300	500	500	300
WTS :	320	400	480	490

TS : 450
Requesting W(A)

	Case 1	Case 2	Case 3	Case 4
RTS	300	500	500	300
WTS	320	400	480	490

Strict Timestamp Ordering (TO) protocol

This protocol operates as follows:

i)*<u>Suppose the transaction T_i</u>* issues read(Q)

If TS(T_i) < W-stamp(Q), then it implies that T_i need to read Q which was already overwritten.

Hence read operation is rejected and T_i is rolled back.

If TS(T_i) ≥ W-timestamp (Q), (latest write by T_j) then read operation is allowed, but T_i waits till the T_j commits/aborts. This makes it recoverable.

We need to simulate locks but this will not cause deadlock.

ii) *Suppose T_i issues write (Q)*

If $TS(T_i)$ < R-timestamp(Q) it implies that the value of Q being produced by T_i had to be written long back.

Hence reject T_i & roll back.

If $TS(T_i)$ < W-timestamp(Q), T_i is attempting to write some absolute value of Q.

Hence reject T_i & roll back.

Otherwise write operation write operation by T_i is allowed, but T_i waits till T' commits/aborts. .

Thomas's Write Rule

Suppose T_i issues write (Q)

Rule 1: If $TS(T_i)$ < R-timestamp(Q) - reject T_i & roll back.

Rule 2: If $TS(T_i)$ < W-timestamp(Q), ignore write operation by T_i do not rollback, continue with processing. This results in rejecting fewer write operations.

Any conflict arising from this will be dealt by Rule-1.

If none of the above happen then perform write and update WTS to T_i

Granularity of Data items

- Database level
- File level
- Block level
- Record level
- Field level

Ex:

Look at the following partial schedule for the concurrent transactions T1, T2 and T3. The data items are A, B and C.

Schedule : T1_XL(A); T1_R(A); T1_Release_XL(A); T2_XL(A); T2_R(A); T3_XL(C); T3_W(C); T3_SL(A); T2_W(A); T3_R(A); T3_Release_XL(C); T1_XL(B); T2_XL(B); T1_R(B); T1_W(B); T1_XL(C); //total 16 operations are there in this schedule

Here, T1_XL(A) - means that the transaction T1 locks data item A in exclusive mode

T1_SL(A) - means that the transaction T1 locks data item A in shared mode

T1_R(A) – means that the transaction T1 reads data item A

T1_Release XL(A) – means that the transaction T1 releases exclusive lock on data item (A)

T1_W(A) - means that the transaction T1 writes data item A

For the above partial schedule, draw the ***wait-for graph*** and find if it results leads to deadlock situation.(Give your reasoning for the answer)

T1	T2	T3
XL(A)		
R(A)		
Rel(A)		
	XL(A)	
	R(A)	
		XL(C)
		W(C)
		SL(A)
	W(A)	
		R(A)
		Rel(C)
XL(B)		
	XL(B)	
R(B)		
W(B)		
XL(C)		

Held by & mode	Data Item			
	A			
	B			
	C			

Look at the following ***partial schedule*** with three transactions T1, T2, and T3.
Now, draw a *wait-for* graph and determine if this leads to deadlock.
Give your reasons in few sentences.

T1	T2	T3
		SL(A)
		R (A)
		Rel(A)
XL(B)		
W(B)		
		SLB)
		R(B)
		Rel(B)
	SLC)	
	R(C)	
XLC)		
		XL(B)
		R(B)
SL(A)		

Summary

- ✓ *Concepts related to Concurrency Control*
- ✓ *Approaches for Implementing Serializability*
- ✓ *How lock-based protocols work*
- ✓ *Detecting the Deadlock condition and prevention*
- ✓ *Two-phase locking protocol*
- ✓ *How timestamp-based protocol works*