Database Systems

Chapter 21

Concurrency Control Techniques

Introduction

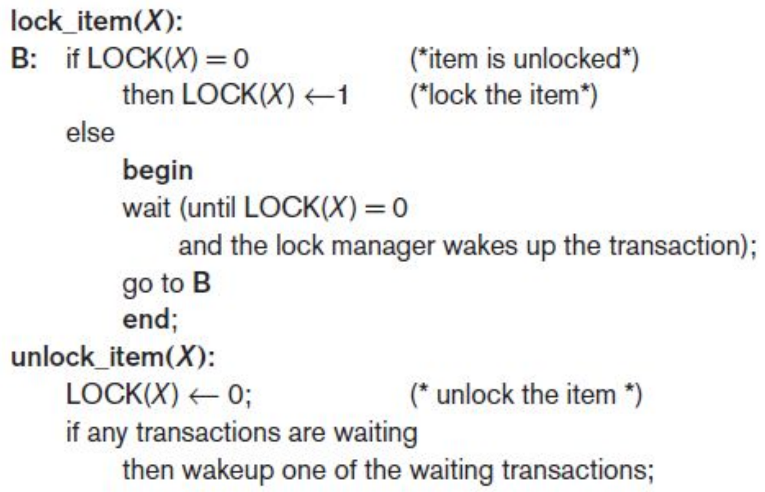
- Concurrency control protocols
  - Set of rules to guarantee serializability
- Two-phase locking protocols
  - Lock data items to prevent concurrent access
- Timestamp
  - Unique identifier for each transaction

Binary locks

- Lock
  - Variable associated with a data item describing status for operations that can be applied
  - One lock for each item in the database
- Binary locks
  - Two states (values)
    - Locked (1)
      - Item cannot be accessed
    - Unlocked (0)
      - Item can be accessed when requested

Binary Locks

- Transaction requests access by issuing a lock_item(X) operation



```
lock_item(X):
B:  if LOCK(X) = 0                  (*item is unlocked*)
        then LOCK(X) ← 1            (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                    (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Drawbacks of Binary Locks

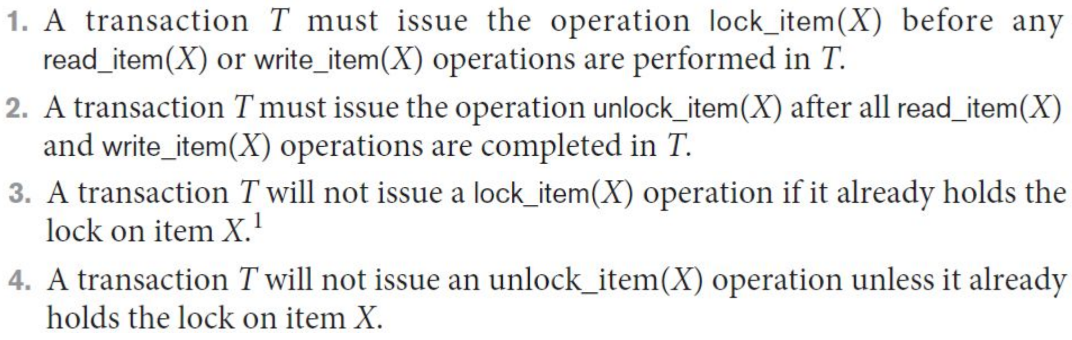
- Binary locks are highly restrictive.
- They do not even permit reading of the contents of item X. As a result, they are not used commercially.

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock table specifies items that have locks
- Lock manager subsystem
  - Keeps track of and controls access to locks
  - Rules enforced by lock manager module
- At most one transaction can hold the lock on an item at a given time
- Binary locking too restrictive for database items

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- If the simple binary locking scheme described is used, then every transaction



1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.[1]
4. A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Shared/exclusive or read/write locks or multi-mode locks
  - Read operations on the same item are not conflicting
  - Must have exclusive lock to write
  - Three locking operations
    - read_lock(X)
    - write_lock(X)
    - unlock(X)

Drawbacks of Read and write locks

May not always produce serializable schedule

May not free from irrecoverability

May not free from deadlock and starvation

May not free from Cascade Rollback

Cascading Rollback

Non Serializable Schedules

T1	T2
write_lock(A)	
read_item(A)	
Write_item(A)	
Unlock(A)	
	write_lock(A)
	read_item(A)
	Write_item(A)
write_lock(B)	Unlock(A)
read_item(B)	
Write_item(B)	
Unlock(B)	

T1	T2
write_lock(A)	
read_item(A)	
Write_item(A)	
Unlock(A)	
	write_lock(A)
	read_item(A)
	Write_item(A)
	Unlock(A)
	commit
Fail	

Cascading Rollback

T1		T2
write_lock(A)		
read_item(A)		
Write_item(A)		
Unlock(A)		
	write_lock(A)	
	read_item(A)	
	Write_item(A)	
	Unlock(A)	
		write_lock(A)
Fail		read_item(A)
		Write_item(A)
		Unlock(A)

Irrecoverability

T1	T2
write_lock(A)	
read_item(A)	
Write_item(A)	
Unlock(A)	
	write_lock(A)
	read_item(A)
	Write_item(A)
	Unlock(A)
	commit
Fail	

Deadlock

T1	T2
read_lock(A)	
read_item(A)	
	read_lock(B)
	read_item(B)
write_lock(B)	
write _item (B)	
	write_lock(A)
	write _item (A)

Starvation

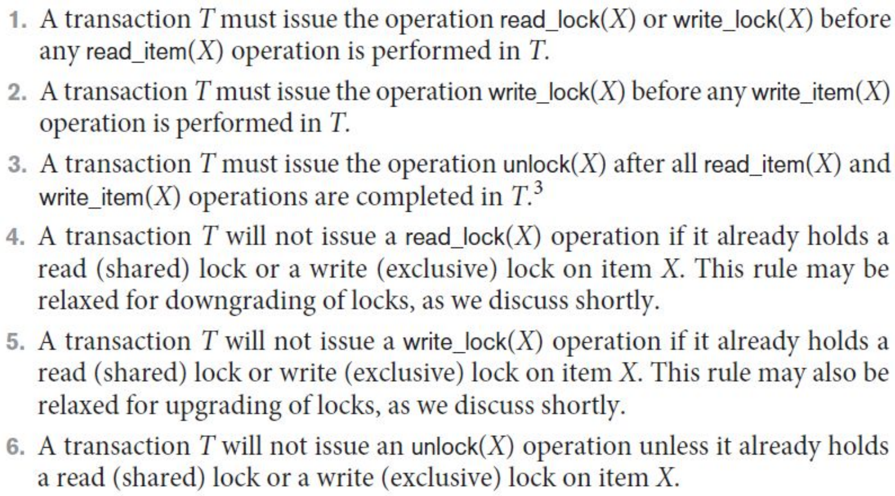
T1	T2	T3		
	read_lock(A)			
write_lock(A)	read_item(A)			
read_item(A)		read_lock(A)		
Write_item(A)		read_item(A)	read_lock(A)	
Unlock(A)			read_item(A)	read_lock(A)
				read_item(A)

Figure 21.2 Locking and unlocking operations for two-mode (read/write, or shared/exclusive) locks

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
            no_of_reads(X) ← 1
            end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else it LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- When we use the shared/exclusive locking scheme, the system must enforce the following rules:



1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.[3]
4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed for downgrading of locks, as we discuss shortly.
5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed for upgrading of locks, as we discuss shortly.
6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Lock conversion
    - Transaction that already holds a lock allowed to convert the lock from one state to another
- Upgrading
    - Issue a read_lock operation then a write_lock operation
- Downgrading
    - Issue a read_lock operation after a write_lock operation

Two-Phase Locking Techniques for Concurrency Control (cont'd.)

- Using binary read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own.
- Take an example in the next slide:

(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

(c)

Time $\downarrow$

T_1	T_2
read_lock(Y);	
read_item(Y);	
unlock(Y);	
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);
write_lock(X);	
read_item(X);	
$X := X + Y$;	
write_item(X);	
unlock(X);	

Result of schedule S:
X=50, Y=50
(nonserializable)

Figure 21.3
Transactions that do not obey two-phase locking. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

Guaranteeing Serializability by Two-Phase Locking (2PL)

- Two-phase locking protocol
  - All locking operations precede the first unlock operation in the transaction
  - Phases
    - Expanding (growing) phase
      - New locks can be acquired but none can be released
      - Lock conversion upgrades (from read-locked to write-locked) must be done during this phase
    - Shrinking phase
      - Existing locks can be released but none can be acquired
      - Downgrades (from write-locked to read-locked) must be done during this phase

Guaranteeing Serializability by Two-Phase Locking (2PL)

- If every transaction in a schedule follows the two-phase locking protocol, schedule guaranteed to be serializable
- Two-phase locking may limit the amount of concurrency that can occur in a schedule
- Some serializable schedules will be prohibited by two-phase locking protocol

Drawbacks of 2PL

May not free from irrecoverability

May not free from deadlock

May not free from starvation

May not free from cascading rollback

Upgrade / Downgrade locks :

A transaction that holds a lock on an item A is allowed under certain condition to change the lock state from one state to another.

Upgrade: A S(A) can be upgraded to X(A) if Ti is the only transaction holding the S-lock on element A.

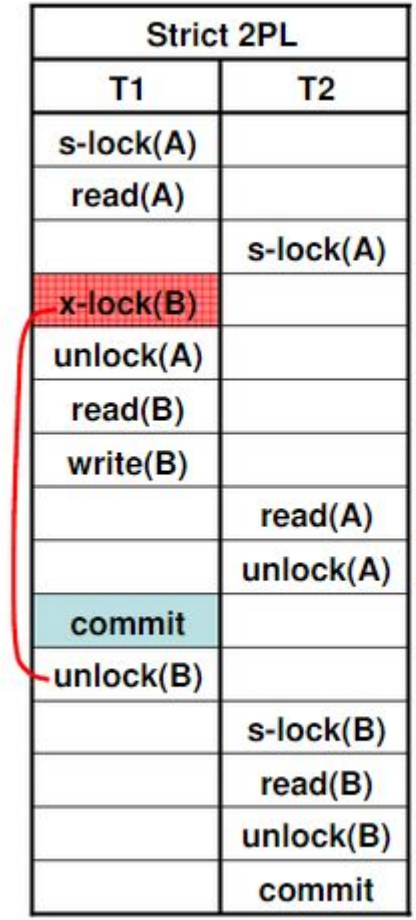
Downgrade: We may downgrade X(A) to S(A) when we feel that we no longer want to write on data-item A. As we were holding X-lock on A, we need not check any conditions.

Variations of Two-Phase Locking (2PL)

- **Basic 2PL**
  - Technique described on previous slides
- **Conservative (static) 2PL**
  - Requires a transaction to lock all the items it accesses before the transaction begins
    - Predeclare read-set and write-set
  - Deadlock-free protocol
  - difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in some situations.
- **Strict 2PL**
  - Transaction does not release exclusive locks (write locks) until after it commits or aborts. It is not deadlock-free.
  - It is always cascadeless and recoverable.

Strict 2PL

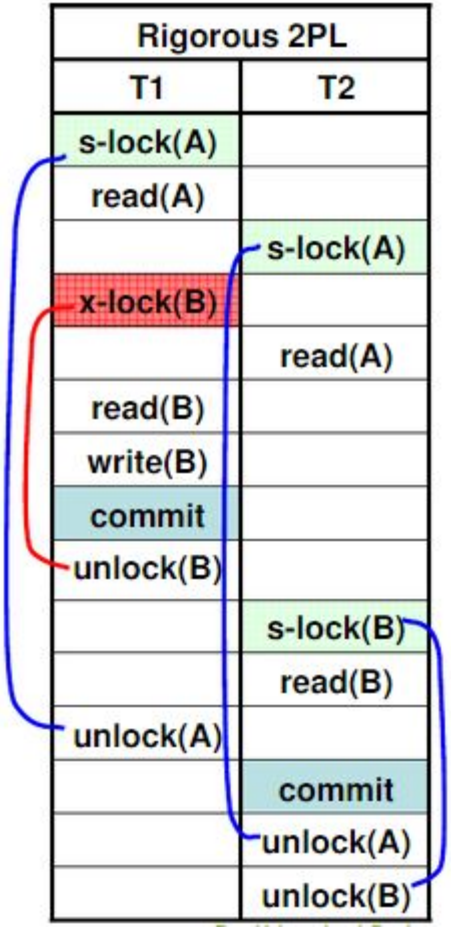
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



Strict 2PL	
T1	T2
s-lock(A)	
read(A)	
	s-lock(A)
x-lock(B)	
unlock(A)	
read(B)	
write(B)	
	read(A)
	unlock(A)
commit	
unlock(B)	
	s-lock(B)
	read(B)
	unlock(B)
	commit

Variations of Two-Phase Locking (cont'd.)

- Rigorous 2PL
  - Transaction does not release any locks (exclusive or shared) until after it commits or aborts
- Concurrency control subsystem responsible for generating read_lock and write_lock requests
- Locking generally considered to have high overhead
  - because every read or write operation is preceded by a system locking request



Rigorous 2PL	
T1	T2
s-lock(A)	
read(A)	
	s-lock(A)
x-lock(B)	
	read(A)
read(B)	
write(B)	
commit	
unlock(B)	
	s-lock(B)
	read(B)
unlock(A)	
	commit
	unlock(A)
	unlock(B)

(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Figure 21.3
Transactions that do not obey two-phase locking.
(a) Two transactions T_1 and T_2.

T_1'	T_2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y)	unlock(X)
read_item(X);	read_item(Y);
$X := X + Y$;	$Y := X + Y$;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

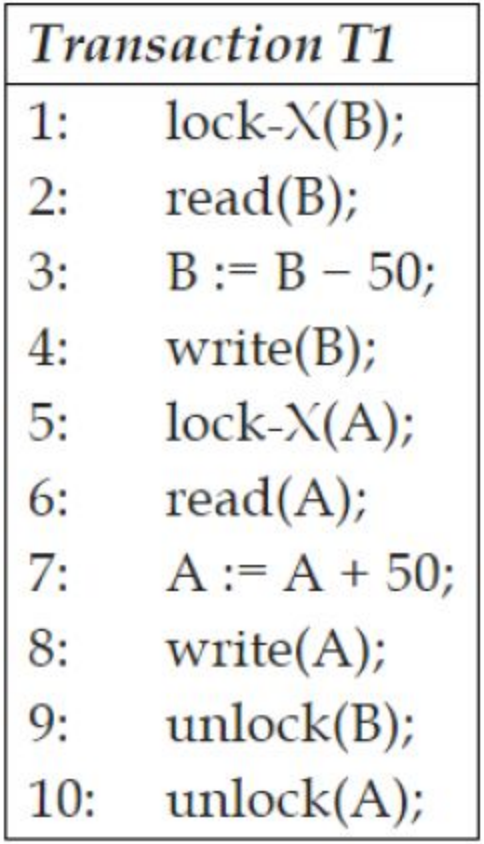
Figure 21.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 21.3 but follow the two-phase locking protocol. Note that they can produce a deadlock.

EXAMPLE(Past Paper 2020)

Add lock and unlock instructions to that particular
transaction in such a way that they observe the two-phase
locking protocol.

Transaction
R(B)
B=B-50
W(B)
R(A)
A=A+40
W(A)

SOLUTION



Transaction T1	
1:	lock-X(B);
2:	read(B);
3:	B := B − 50;
4:	write(B);
5:	lock-X(A);
6:	read(A);
7:	A := A + 50;
8:	write(A);
9:	unlock(B);
10:	unlock(A);

Dealing with Deadlock and Starvation

- Deadlock
  - Occurs when each transaction T in a set is waiting for some item locked by some other transaction *T’*
  - Both transactions stuck in a waiting queue

(a)

Time

T_1'	T_2'
read_lock(*Y*);	
read_item(*Y*);	
	read_lock(*X*);
	read_item(*X*);
write_lock(*X*);	
	write_lock(*Y*);

(b)

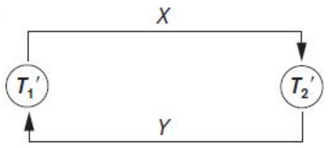
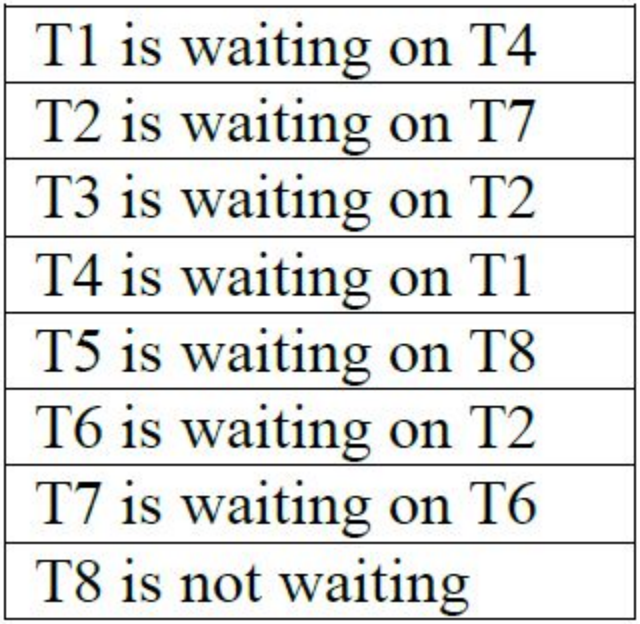


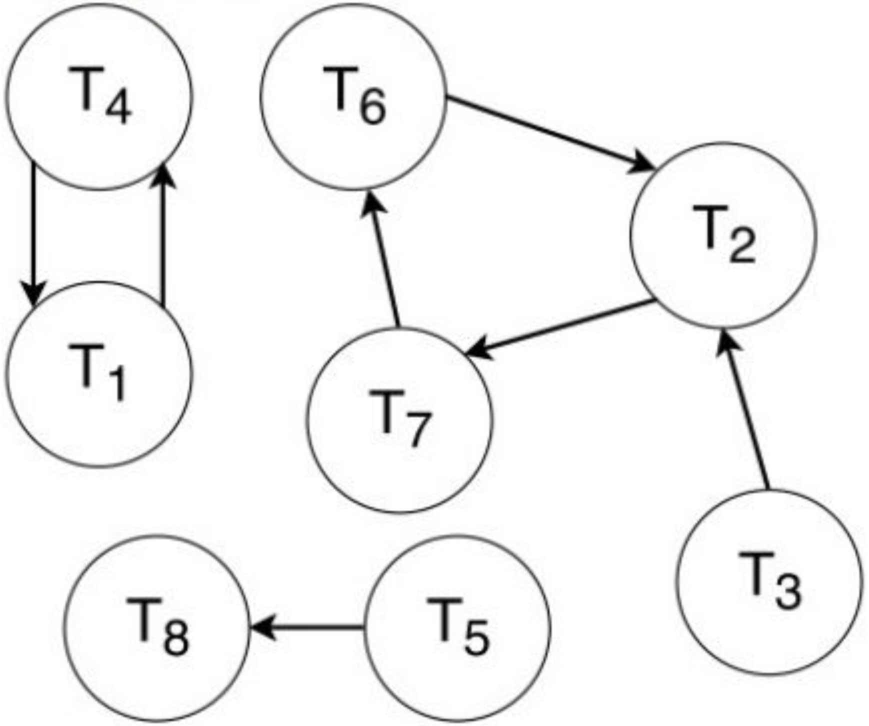
Figure 21.5 Illustrating the deadlock problem (a) A partial schedule of *T*1′ and *T*2′ that is in a state of deadlock (b) A wait-for graph (WFG) for the partial schedule in (a)

EXAMPLE(Past Paper 2020)



T1 is waiting on T4
T2 is waiting on T7
T3 is waiting on T2
T4 is waiting on T1
T5 is waiting on T8
T6 is waiting on T2
T7 is waiting on T6
T8 is not waiting

T1, T2, T4, T6, and T7.



Dealing with Deadlock and Starvation (cont'd.)

- Deadlock prevention protocols
  - Every transaction locks all items it needs in advance
  - Ordering all items in the database
    - Transaction that needs several items will lock them in that order
  - Both approaches impractical

Dealing with Deadlock and Starvation (cont'd.)

- A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation:
- these techniques use the concept of **transaction timestamp** TS(*T*′), which is a unique identifier assigned to each transaction.
- if transaction T_1 starts before transaction T_2,

  then $TS(T_1) < TS(T_2)$.

NOTE: *older* transaction (which starts first) has the smaller timestamp value

Dealing with Deadlock and Starvation (cont'd.)

- 2 Protocols based on a timestamp
- Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j

Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

Check if **TS(Ti) < TS(Tj) - If Ti** is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.

Check if **TS(Ti) < TS(Tj)** - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

Wound wait scheme

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.

Other Methods for Deadlock Prevention

Wait – Die	Wound -Wait
It is based on a non-preemptive technique.	It is based on a preemptive technique.
In this, older transactions must wait for the younger one to release its data items.	In this, older transactions never wait for younger transactions.
The number of aborts and rollback is higher in these techniques.	In this, the number of aborts and rollback is lesser.

No Waiting (NW)

Another group of protocols that prevent deadlock do not require timestamps.

These include the no waiting (NW) and cautious waiting (CW) algorithms.

In the no waiting algorithm, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.

In this case, no transaction ever waits, so no deadlock will occur.

However, this scheme can cause transactions to abort and restart needlessly.

Cautious Waiting

The cautious waiting algorithm was proposed to try to reduce the number of needless aborts/restarts.

Suppose that transaction Ti tries to lock an item X but is not able to do so because X is locked by some other transaction Tj with

a conflicting lock. The cautious waiting rule is as follows:

■ Cautious waiting. If Tj is not blocked (not waiting for some other locked item), then Ti is blocked and allowed to wait; otherwise abort Ti.

Dealing with Deadlock and Starvation (cont'd.)

- No waiting algorithm
  - If transaction unable to obtain a lock, immediately aborted and restarted later
- Deadlock detection
  - System checks to see if a state of deadlock exists
  - Wait-for graph (a state of deadlock if and only if the wait-for graph has a cycle)

Dealing with Deadlock and Starvation (cont'd.)

- **Victim selection**
  - Deciding which transaction to abort in case of deadlock (avoid selecting transactions that have been running for a long time)
- **Timeouts**
  - If system waits longer than a predefined time, it aborts the transaction
- **Starvation**
  - Occurs if a transaction cannot proceed for an indefinite period of time while other transactions continue normally
  - Solution: first-come-first-served queue

Concurrency Control Based on Timestamp Ordering

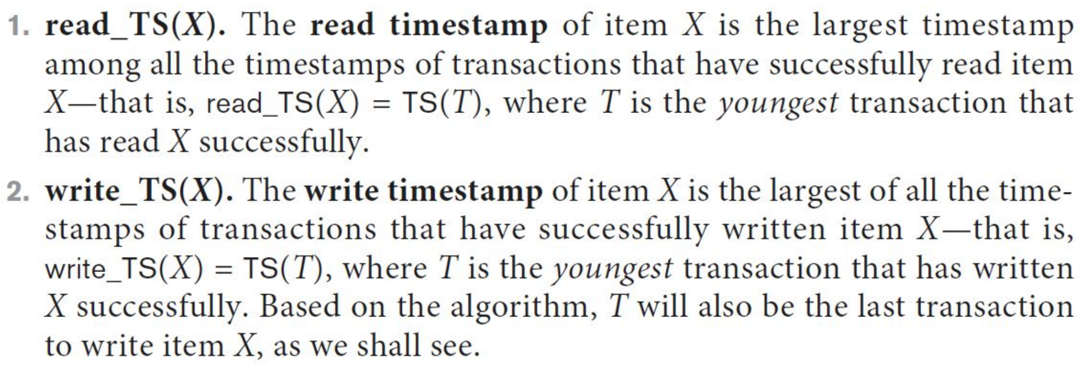
- Timestamp
  - Unique identifier assigned by the DBMS to identify a transaction
  - Assigned in the order submitted
  - Transaction start time
- Concurrency control techniques based on timestamps do not use locks, so
  - Deadlocks cannot occur

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Generating timestamps
  - Counter incremented each time its value is assigned to a transaction
  - Current date/time value of the system clock
    - Ensure no two timestamps are generated during the same tick of the clock
- General approach
  - Enforce equivalent serial order on the transactions based on their timestamps

Concurrency Control Based on Timestamp Ordering (cont'd.)

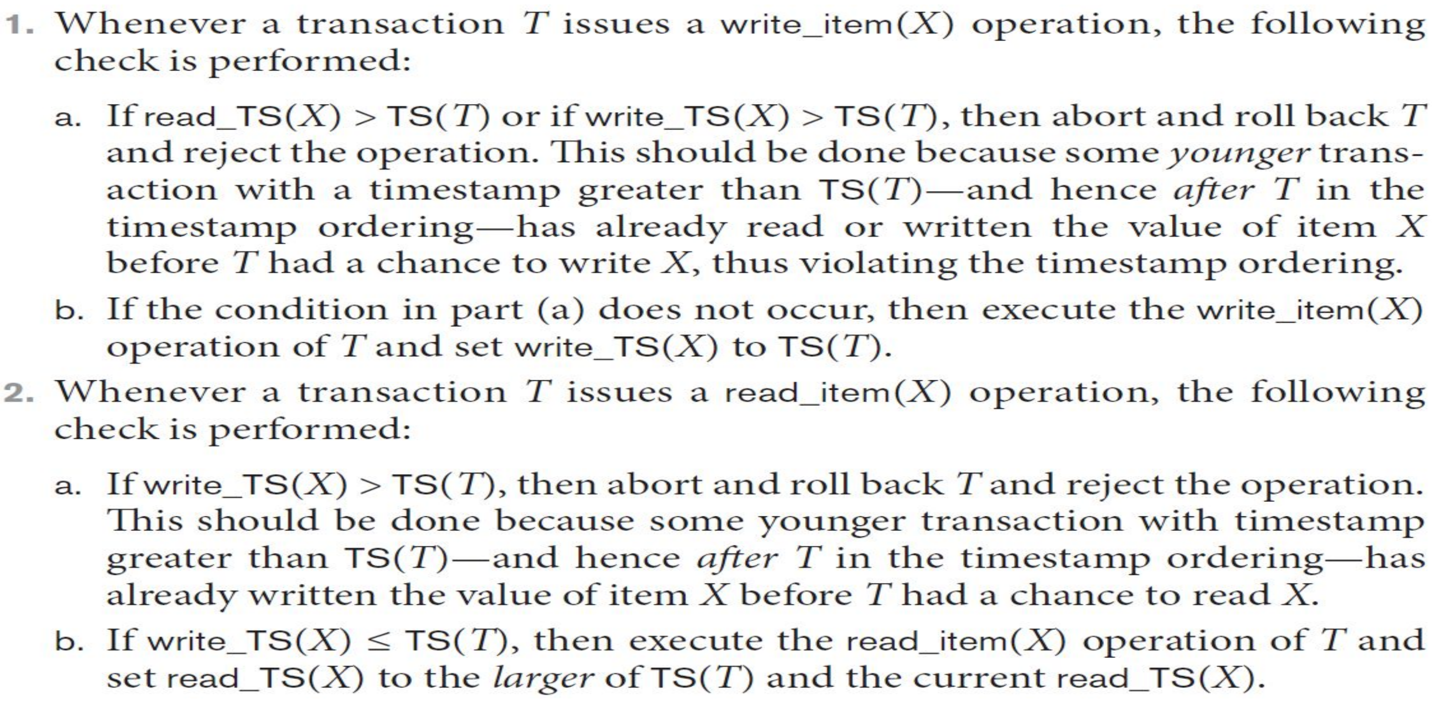
- Timestamp ordering (TO)
  - Allows interleaving of transaction operations
  - Must ensure timestamp order is followed for each pair of conflicting operations
- Each database item assigned two timestamp values 1) read_TS(X) and 2) write_TS(X)



1. **read_TS(*X*).** The **read timestamp** of item *X* is the largest timestamp among all the timestamps of transactions that have successfully read item *X*—that is, read_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has read *X* successfully.
2. **write_TS(*X*).** The **write timestamp** of item *X* is the largest of all the timestamps of transactions that have successfully written item *X*—that is, write_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has written *X* successfully. Based on the algorithm, *T* will also be the last transaction to write item *X*, as we shall see.

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Basic TO algorithm



1. Whenever a transaction T issues a write_item(X) operation, the following check is performed:

   a. If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.

   b. If the condition in part (a) does not occur, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2. Whenever a transaction T issues a read_item(X) operation, the following check is performed:

   a. If write_TS(X) > TS(T), then abort and roll back T and reject the operation. This should be done because some younger transaction with timestamp greater than TS(T)—and hence *after* T in the timestamp ordering—has already written the value of item X before T had a chance to read X.

   b. If write_TS(X) ≤ TS(T), then execute the read_item(X) operation of T and set read_TS(X) to the *larger* of TS(T) and the current read_TS(X).

RULES

- Transaction Ti issues a Read(item) operation.
  - if WTS(item)>Ts(Ti) then Rollback Ti
  - Else excecute R(A) operation.
    - Set RTS(item)=Max(RTS(item), Ts(Ti))
- Transaction Ti issues Write(item) operation
  - If RTS(item)>Ts(Ti) then Rollback Ti
  - If WTS(item)>Ts(Ti) then Rollback Ti
  - ELSE execute write operation
    - SET WTS(Item)=Ts(Ti)

EXAMPLE

T1(100)	T2(200)	T3(300)
R(A)		
	R(B)	
W(C)		
		R(B)
R(C)		
	W(B)	
		W(A)

	A	B	C
RTS(Ti)	0	0	0
WTS(Ti)	0	0	0

	A	B	C
RTS(Ti)	0 100	0 200 300	0 100
WTS(Ti)	0 300	0	0 100

WHICH TRANSACTION WILL BE ROLLED BACK? T2

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Strict TO algorithm
  - Ensures schedules are both strict and conflict serializable

Concurrency Control Based on Timestamp Ordering (cont'd.)

- Thomas's write rule
  - Modification of basic TO algorithm
  - Does not enforce conflict serializability
  - Rejects fewer write operations by modifying checks for write_item(X) operation

Thomas write rule

- If TS(T) < R_TS(X) then transaction T is aborted and rolled back, and operation is rejected.
- If TS(T) < W_TS(X) then don't execute the W_item(X) operation of the transaction and continue processing.
- If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set W_TS(X) to TS(T).