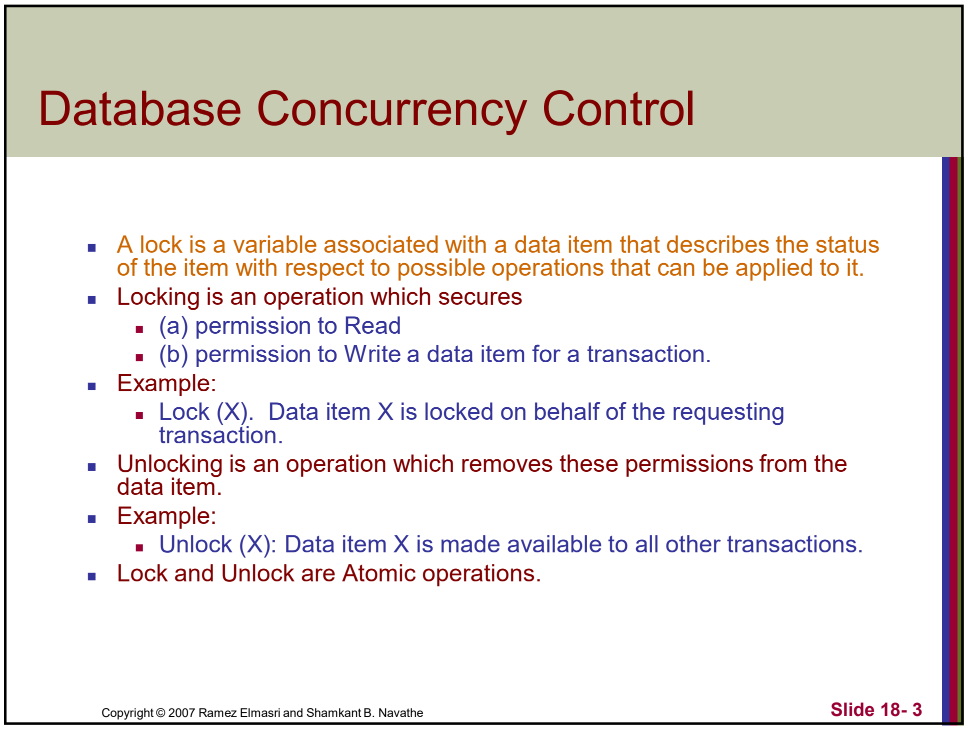
Concurrency Control Techniques

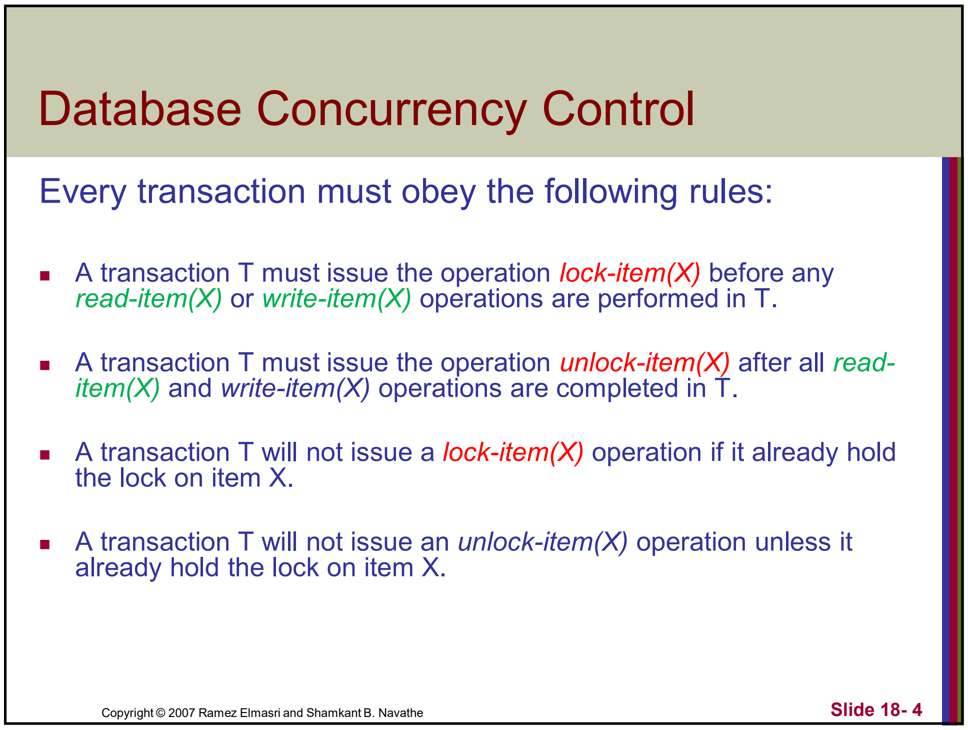
Database Concurrency Control

- 1 Purpose of Concurrency Control
  - To enforce Isolation (through mutual exclusion) among conflicting transactions.
    - Isolation: A transaction should not make its updates visible to other transactions until it is committed
  - To preserve database consistency through consistency preserving execution of transactions.
  - To resolve read-write and write-write conflicts.
- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get A and if the other transaction is rolled-back or waits.



Database Concurrency Control

- A lock is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked on behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.



Database Concurrency Control

Every transaction must obey the following rules:

- A transaction T must issue the operation *lock-item(X)* before any *read-item(X)* or *write-item(X)* operations are performed in T.
- A transaction T must issue the operation *unlock-item(X)* after all *read-item(X)* and *write-item(X)* operations are completed in T.
- A transaction T will not issue a *lock-item(X)* operation if it already hold the lock on item X.
- A transaction T will not issue an *unlock-item(X)* operation unless it already hold the lock on item X.

Database Concurrency Control

- Types of Locks:
  - (a) Binary Locks (b) shared (read) (c) exclusive (write).
- Binary Lock
  - Can have two states: Locked or Unlocked
  - A distinct lock is associated with each database item
- Shared (Read) lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive (Write) lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Database Concurrency Control

- Lock Manager:
  - A subsystem to keep track of and control access to locks
  - Managing locks on data items.
- Lock table:
  - Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Database Concurrency Control

- Database requires that all transactions should be well-formed. A transaction is well-formed if:
  - It must lock the data item before it reads or writes to it.
  - It must not lock an already locked data item and it must not try to unlock a free data item.

Database Concurrency Control

Lock and Unlock operations for Binary Locks

- The following code performs the lock operation:

```
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);
    goto B
    end;
```

Database Concurrency Control

- The following code performs the unlock operation:

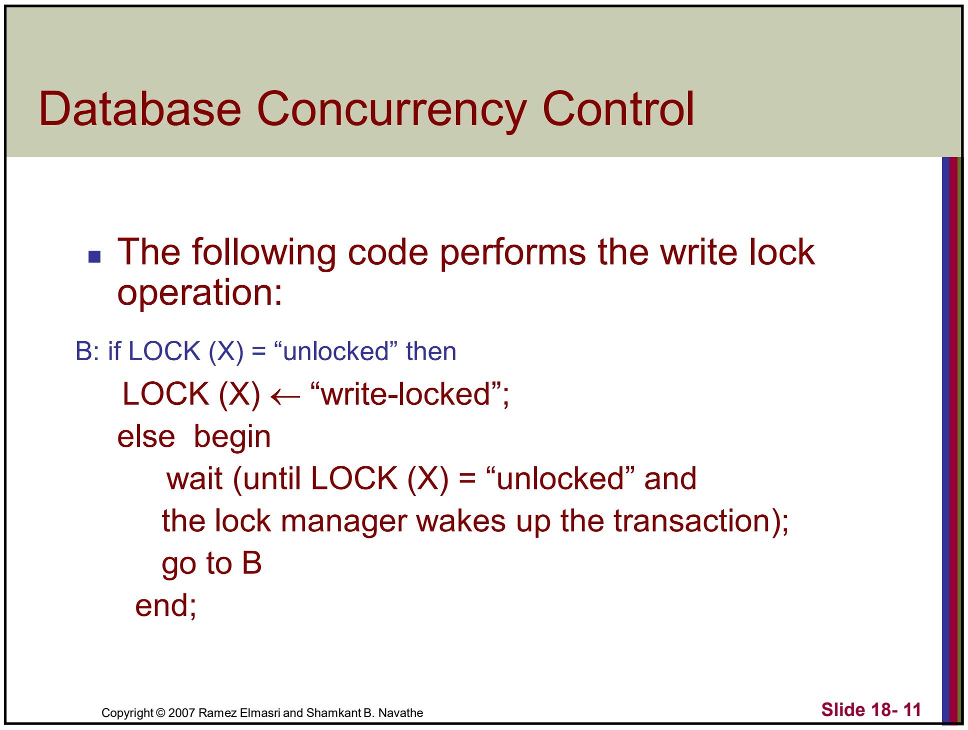
```
LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
  wake up one of the waiting transactions;
```

Database Concurrency Control

Operations for Read/Write Locks (Shared/Exclusive)

- The following code performs the read operation:

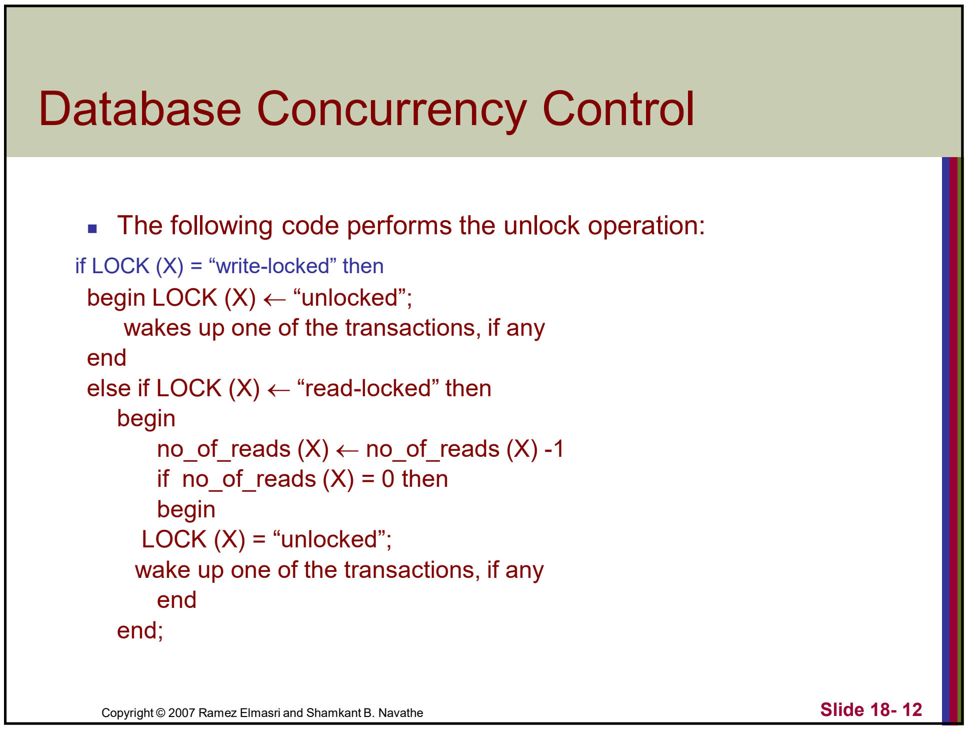
```
B: if LOCK (X) = "unlocked" then
 begin LOCK (X) ← "read-locked";
    no_of_reads (X) ← 1;
 end
 else if LOCK (X) ← "read-locked" then
        no_of_reads (X) ← no_of_reads (X) +1
      else begin wait (until LOCK (X) = "unlocked" and
         the lock manager wakes up the transaction);
         go to B
      end;
```



Database Concurrency Control

- The following code performs the write lock operation:

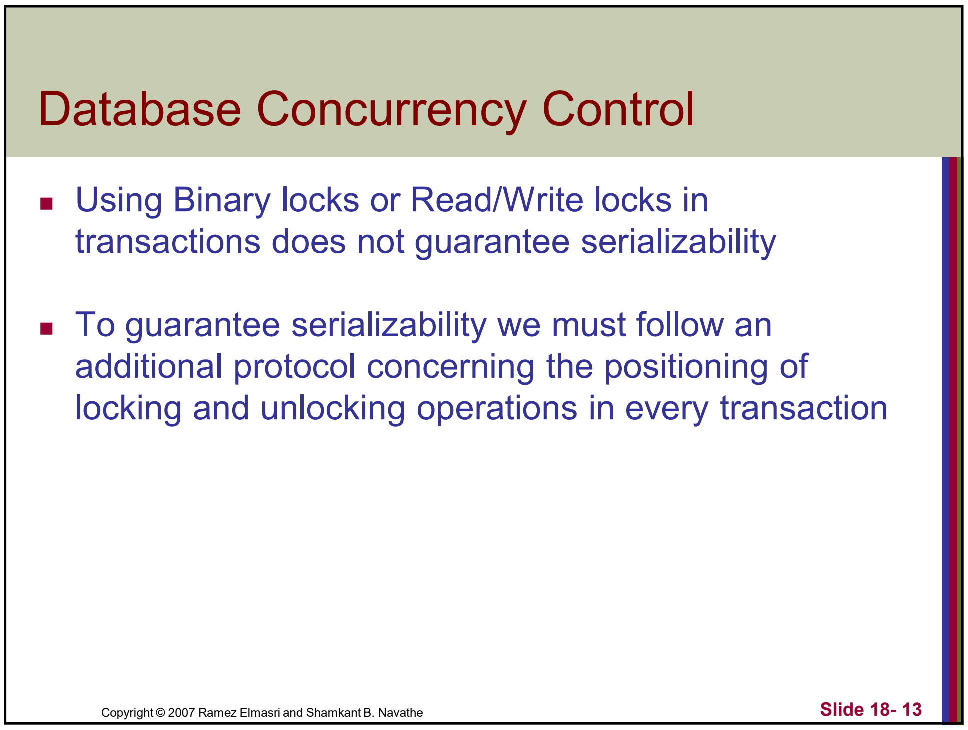
```
B: if LOCK (X) = “unlocked” then
     LOCK (X) ← “write-locked”;
    else  begin
        wait (until LOCK (X) = “unlocked” and
        the lock manager wakes up the transaction);
        go to B
      end;
```



Database Concurrency Control

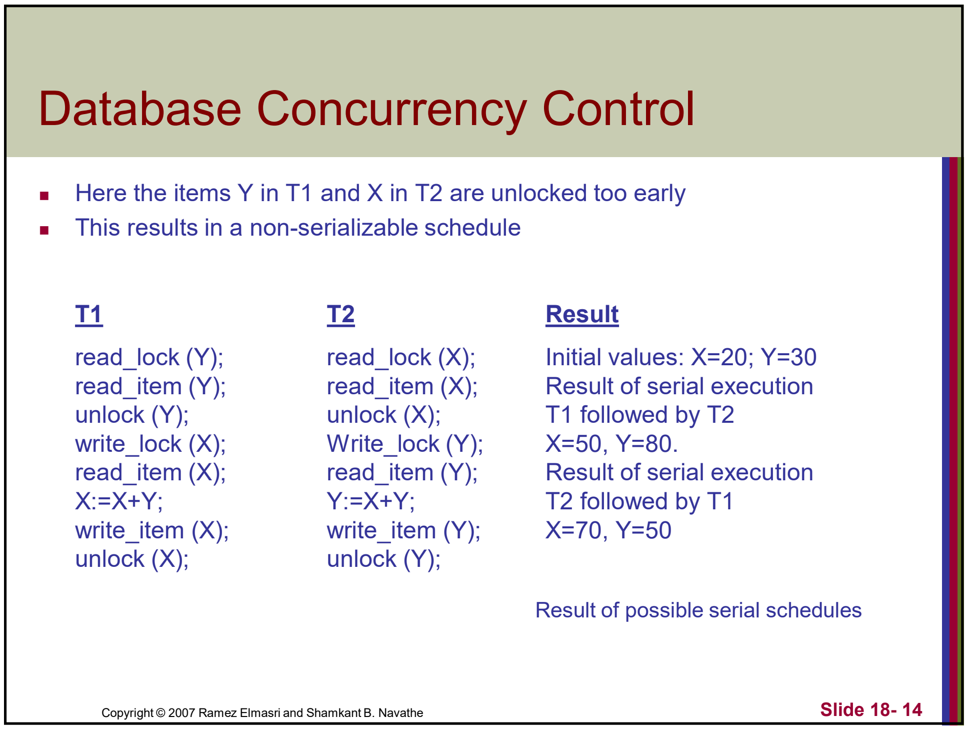
- The following code performs the unlock operation:

```
if LOCK (X) = “write-locked” then
 begin LOCK (X) ← “unlocked”;
      wakes up one of the transactions, if any
 end
 else if 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”;
      wake up one of the transactions, if any
        end
    end;
```



Database Concurrency Control

- Using Binary locks or Read/Write locks in transactions does not guarantee serializability
- To guarantee serializability we must follow an additional protocol concerning the positioning of locking and unlocking operations in every transaction

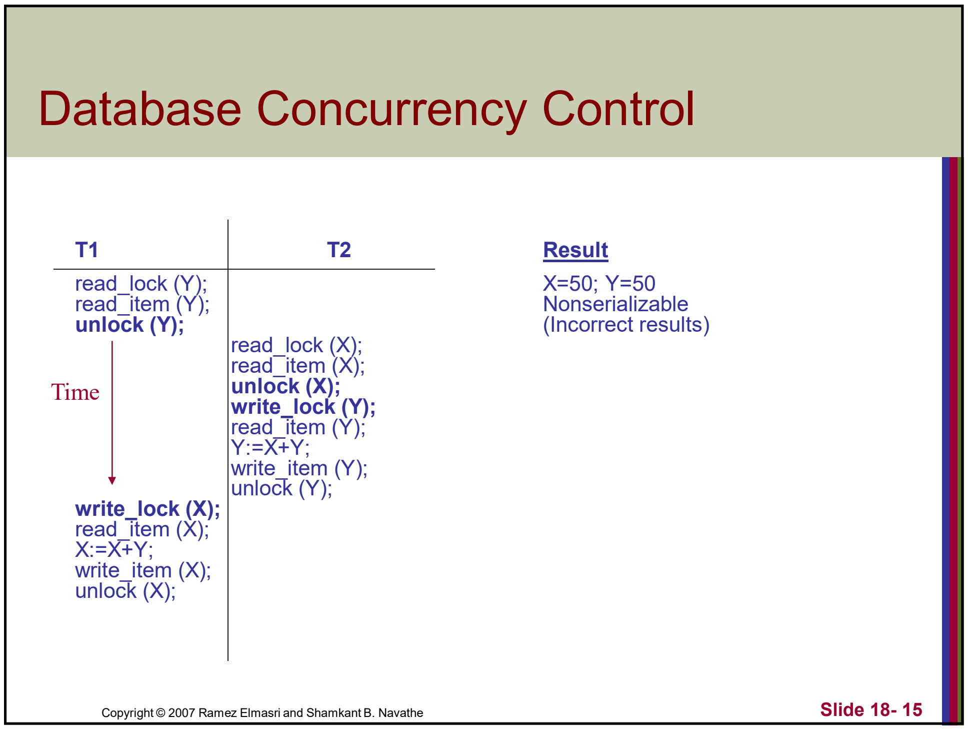


Database Concurrency Control

- Here the items Y in T1 and X in T2 are unlocked too early
- This results in a non-serializable schedule

T1	**T2**	**Result**
read_lock (Y);	read_lock (X);	Initial values: X=20; Y=30
read_item (Y);	read_item (X);	Result of serial execution
unlock (Y);	unlock (X);	T1 followed by T2
write_lock (X);	Write_lock (Y);	X=50, Y=80.
read_item (X);	read_item (Y);	Result of serial execution
X:=X+Y;	Y:=X+Y;	T2 followed by T1
write_item (X);	write_item (Y);	X=70, Y=50
unlock (X);	unlock (Y);	

Result of possible serial schedules



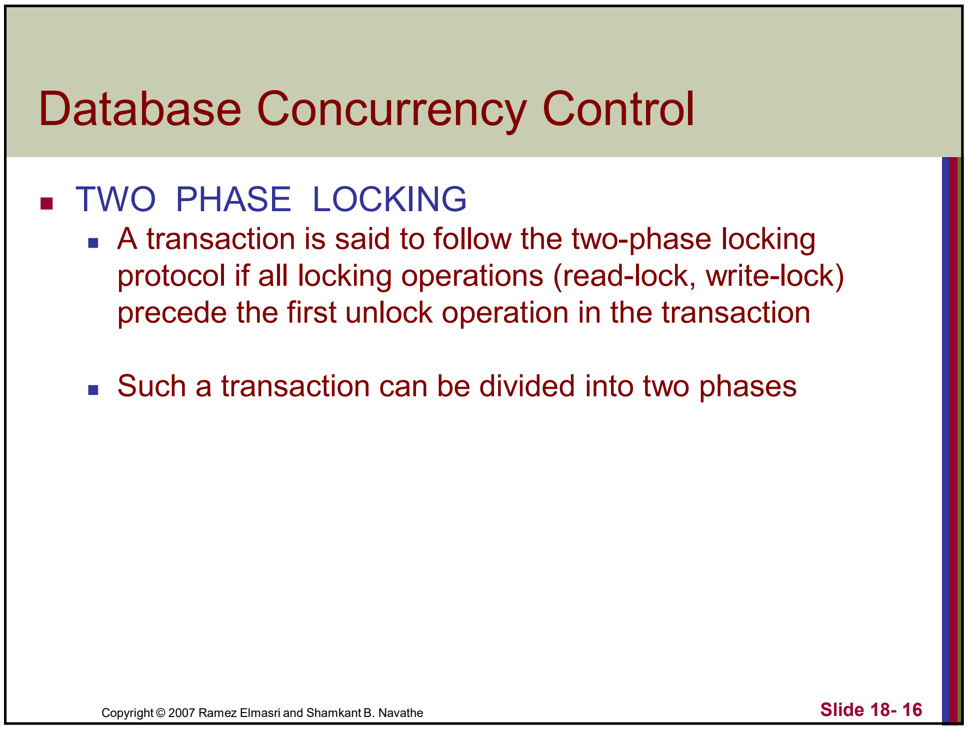
Database Concurrency Control

T1	T2
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);	

Time

Result

X=50; Y=50
Nonserializable
(Incorrect results)



Database Concurrency Control

- TWO PHASE LOCKING
  - A transaction is said to follow the two-phase locking protocol if all locking operations (read-lock, write-lock) precede the first unlock operation in the transaction
  - Such a transaction can be divided into two phases

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

- Two Phases:
  - (a) Locking (Growing)
  - (b) Unlocking (Shrinking).
- **Locking (Growing) Phase:**
  - A transaction applies locks (read or write) on desired data items one at a time.
  - New locks acquired, but none can be released
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time.
  - No new locks can be acquired
- **Requirement:**
  - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Database Concurrency Control

Transactions which follow 2-phase locking protocol

- Order of locking and unlocking modified
- T1’ will issue its writelock(X) before it unlocks item Y
- Consequently, when T2’ issues its readlock(X), it is forced to wait until T1’ releases the lock by issuing an unlock(X) in the schedule

<u>**T’1**</u>	<u>**T’2**</u>
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);

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Database Concurrency Control

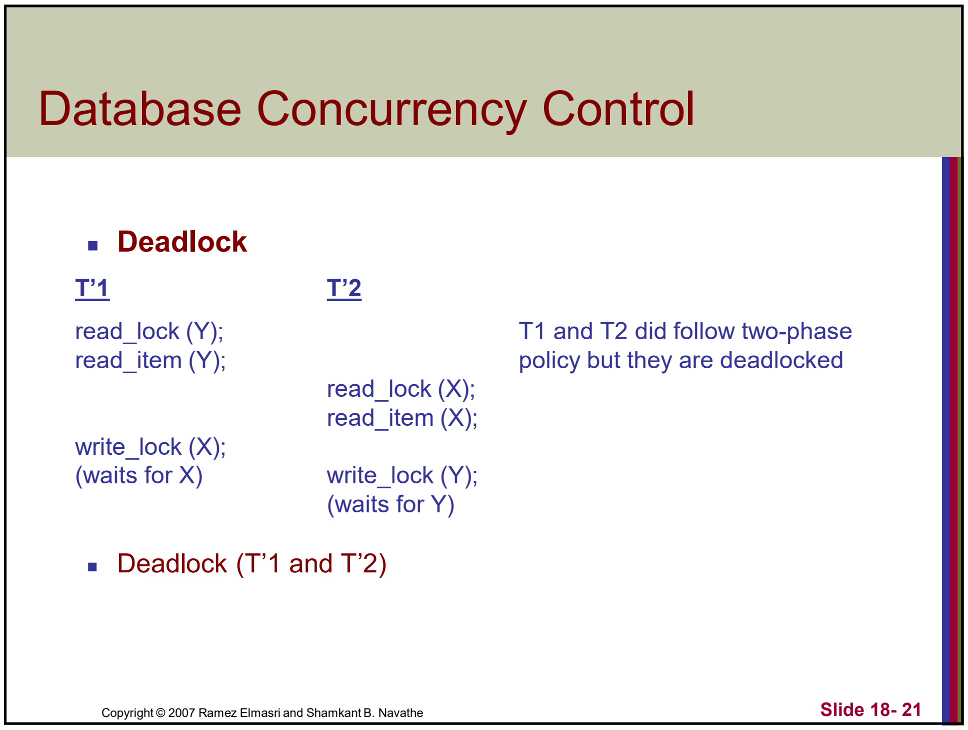
Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
  - (a) **Basic**
  - (b) **Conservative**
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Strict**:
  - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Database Concurrency Control

Deadlocks

- Occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T' in the set
- Each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item



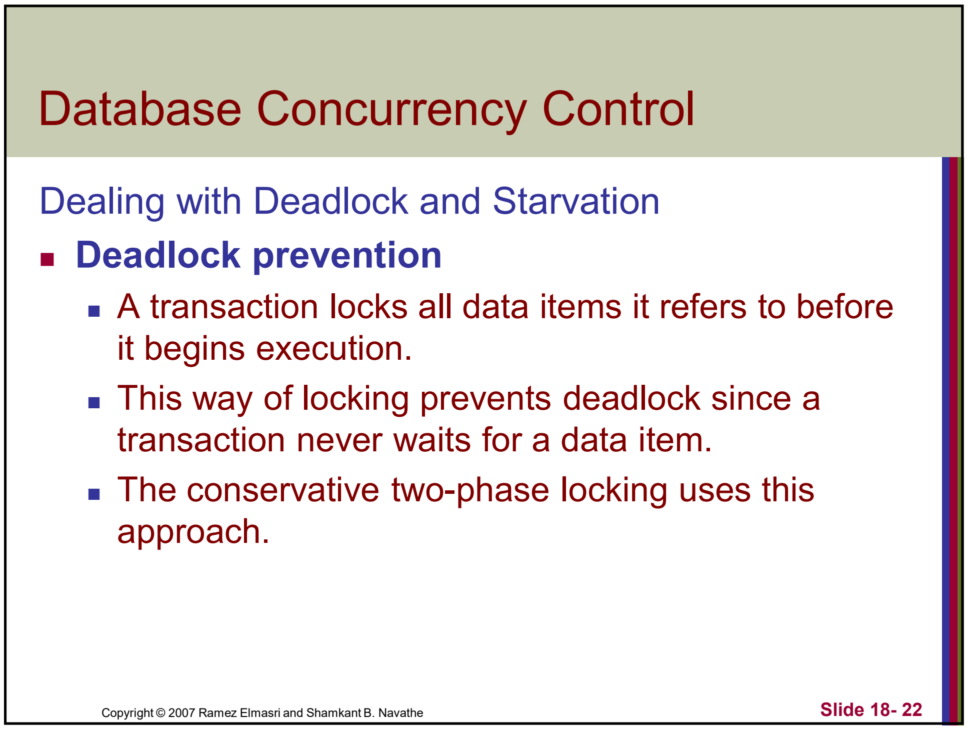
Database Concurrency Control

- **Deadlock**

T'1	T'2
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

T1 and T2 did follow two-phase policy but they are deadlocked

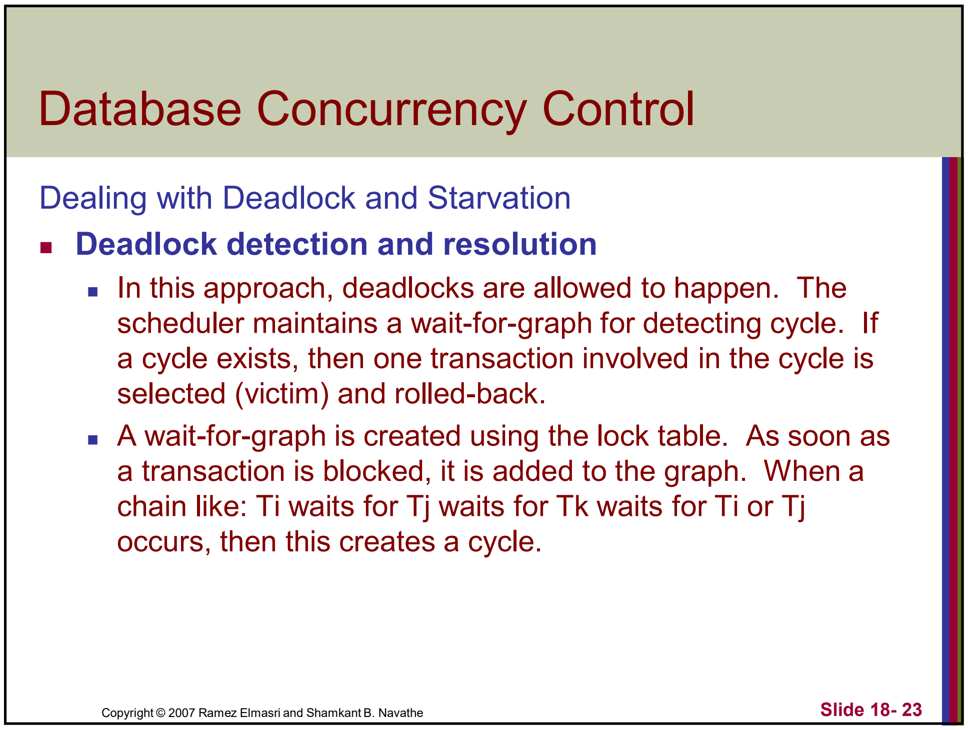
- Deadlock (T'1 and T'2)



Database Concurrency Control

Dealing with Deadlock and Starvation

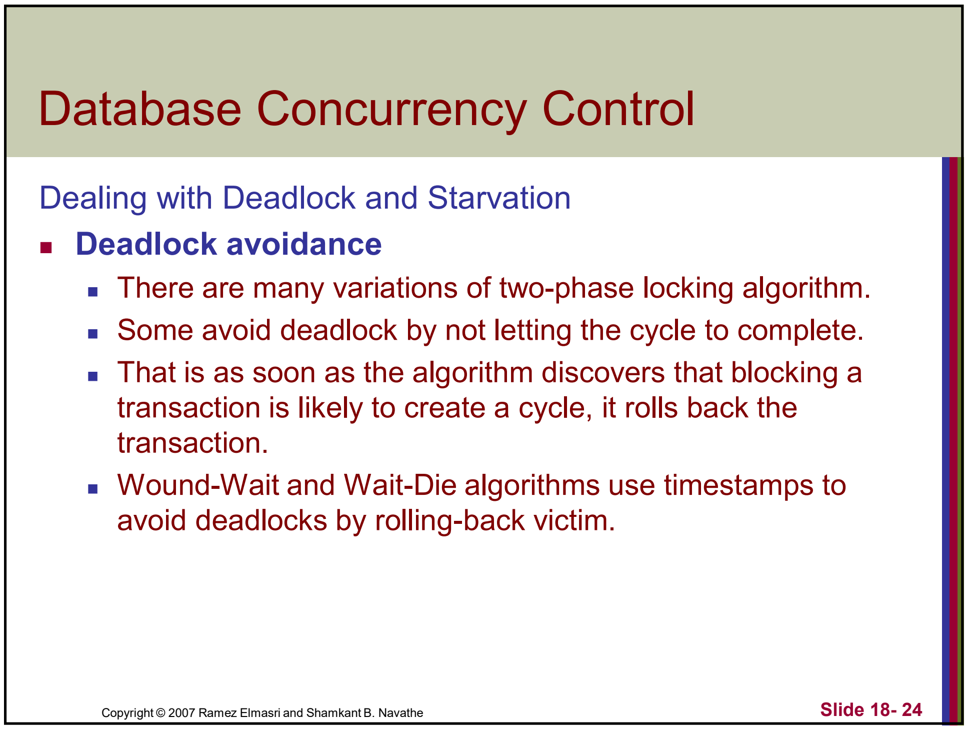
- **Deadlock prevention**
  - A transaction locks all data items it refers to before it begins execution.
  - This way of locking prevents deadlock since a transaction never waits for a data item.
  - The conservative two-phase locking uses this approach.



Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock detection and resolution**
  - In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
  - A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: Ti waits for Tj waits for Tk waits for Ti or Tj occurs, then this creates a cycle.



Database Concurrency Control

Dealing with Deadlock and Starvation

- **Deadlock avoidance**
  - There are many variations of two-phase locking algorithm.
  - Some avoid deadlock by not letting the cycle to complete.
  - That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
  - Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

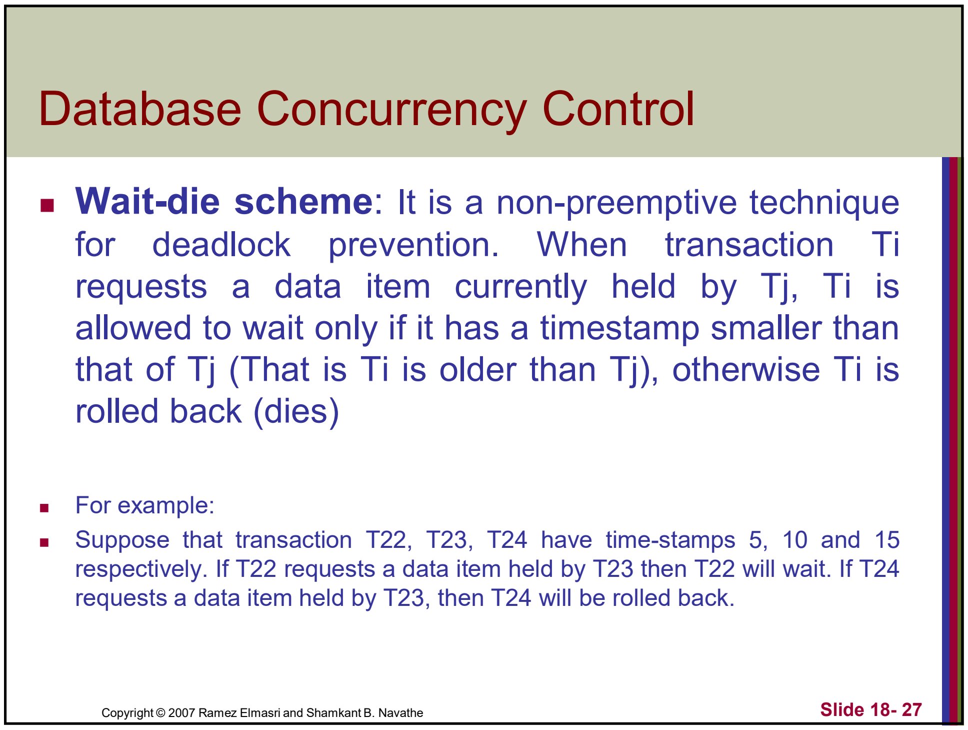
Database Concurrency Control

Dealing with Deadlock and Starvation

- **Wait-Die**
  - If TS(Ti) <TS(Tj) (Ti is older) Ti is allowed to wait; Otherwise abort Ti and restart it later with the same timestamp
- **Wound-wait**
  - If TS(Ti) <TS(Tj) (Ti is older), then abort Tj and restart it later; Otherwise Ti is allowed to wait

Database Concurrency Control

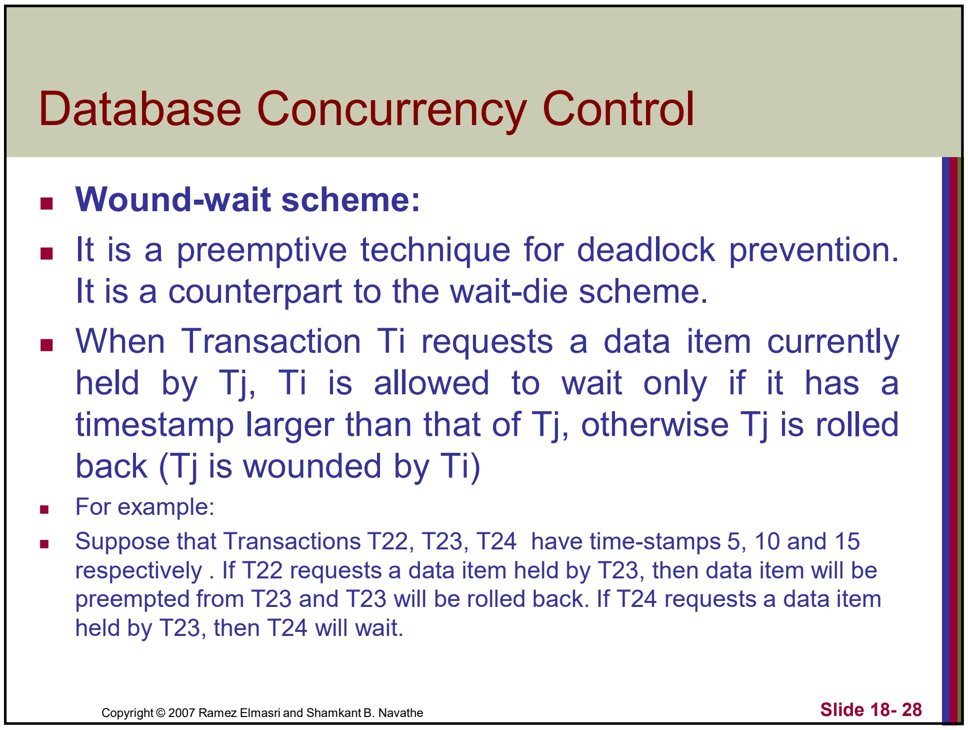
- In Wait-Die, an older transaction is allowed to wait on a younger transaction, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted
- In Wound-wait, a younger transaction is allowed to wait on an older transaction, whereas an older transaction requesting an item held by a younger transaction preempts the younger transaction by aborting it



Database Concurrency Control

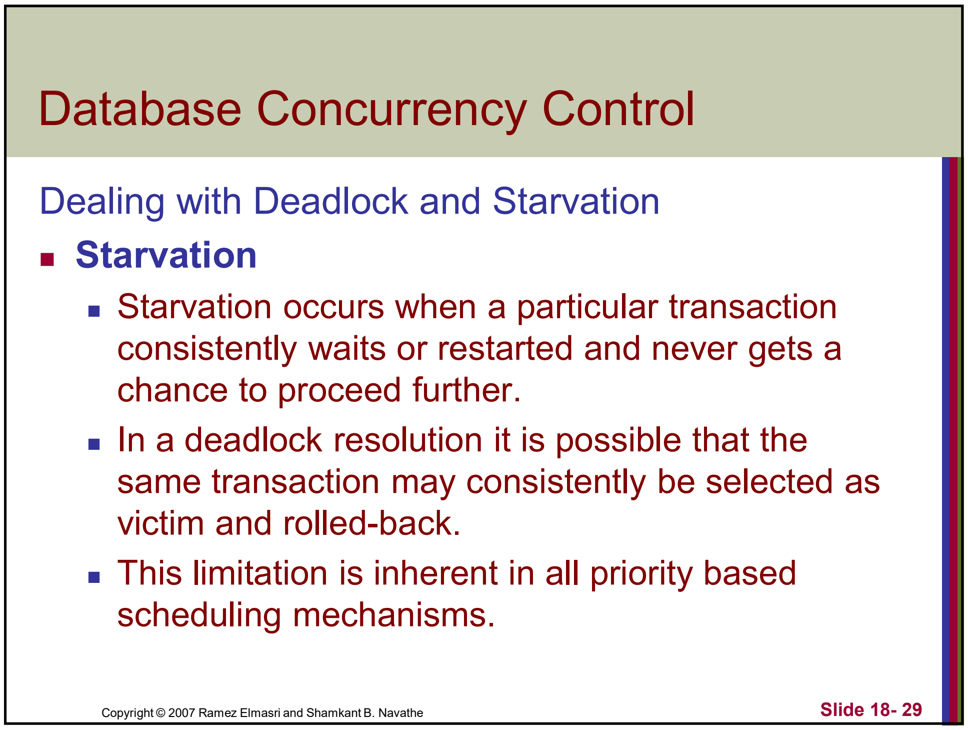
- **Wait-die scheme**: It is a non-preemptive technique for deadlock prevention. When transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than that of Tj (That is Ti is older than Tj), otherwise Ti is rolled back (dies)

- For example:
- Suppose that transaction T22, T23, T24 have time-stamps 5, 10 and 15 respectively. If T22 requests a data item held by T23 then T22 will wait. If T24 requests a data item held by T23, then T24 will be rolled back.



Database Concurrency Control

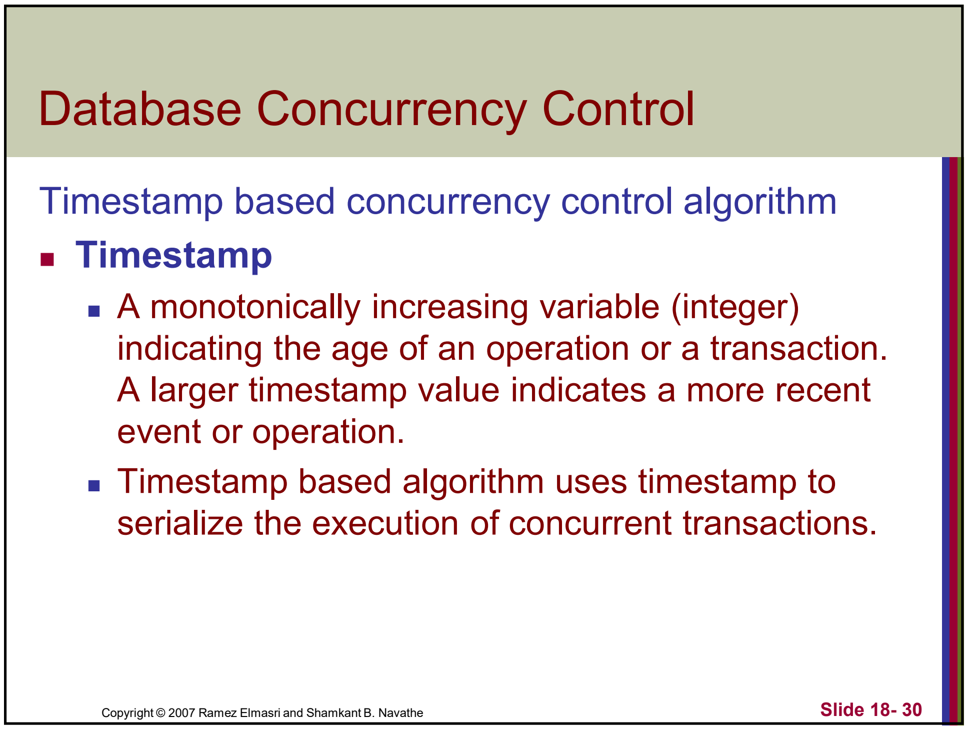
- **Wait-wait scheme:**
- It is a preemptive technique for deadlock prevention. It is a counterpart to the wait-die scheme.
- When Transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp larger than that of Tj, otherwise Tj is rolled back (Tj is wounded by Ti)
- For example:
- Suppose that Transactions T22, T23, T24 have time-stamps 5, 10 and 15 respectively . If T22 requests a data item held by T23, then data item will be preempted from T23 and T23 will be rolled back. If T24 requests a data item held by T23, then T24 will wait.



Database Concurrency Control

Dealing with Deadlock and Starvation

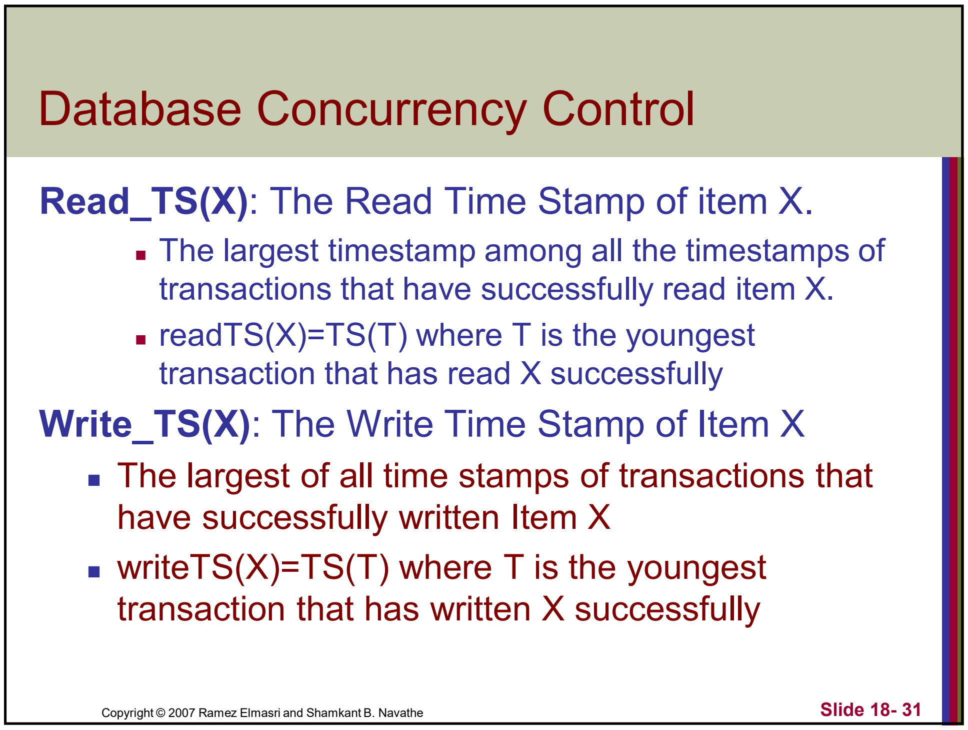
- **Starvation**
  - Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
  - In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
  - This limitation is inherent in all priority based scheduling mechanisms.



Database Concurrency Control

Timestamp based concurrency control algorithm

- **Timestamp**
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
  - Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.



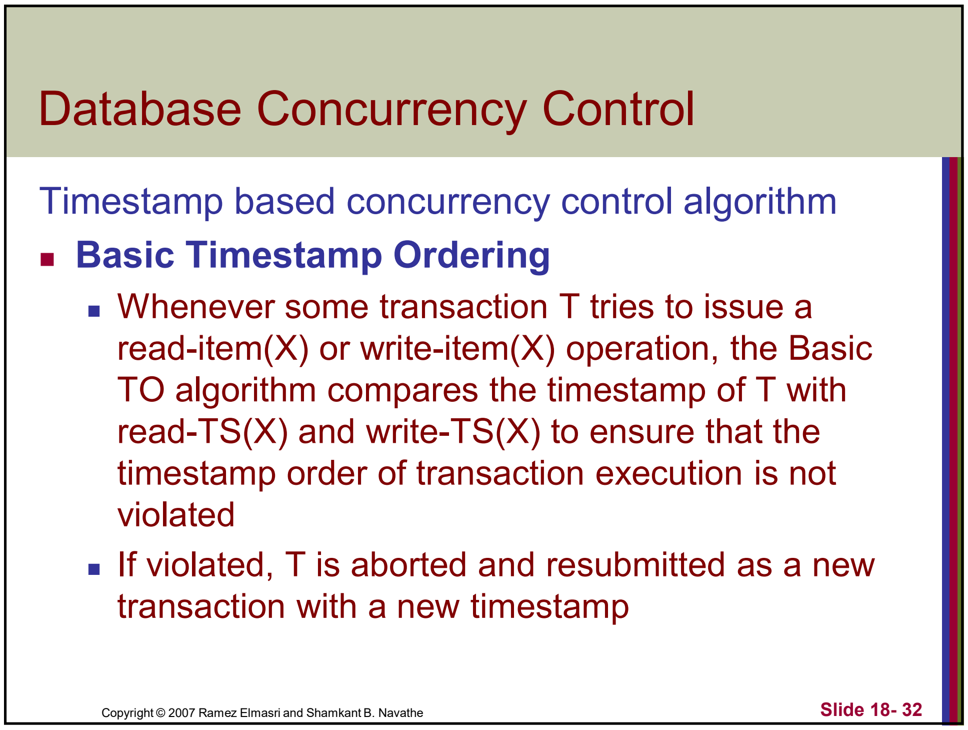
Database Concurrency Control

Read_TS(X): The Read Time Stamp of item X.

- The largest timestamp among all the timestamps of transactions that have successfully read item X.
- readTS(X)=TS(T) where T is the youngest transaction that has read X successfully

Write_TS(X): The Write Time Stamp of Item X

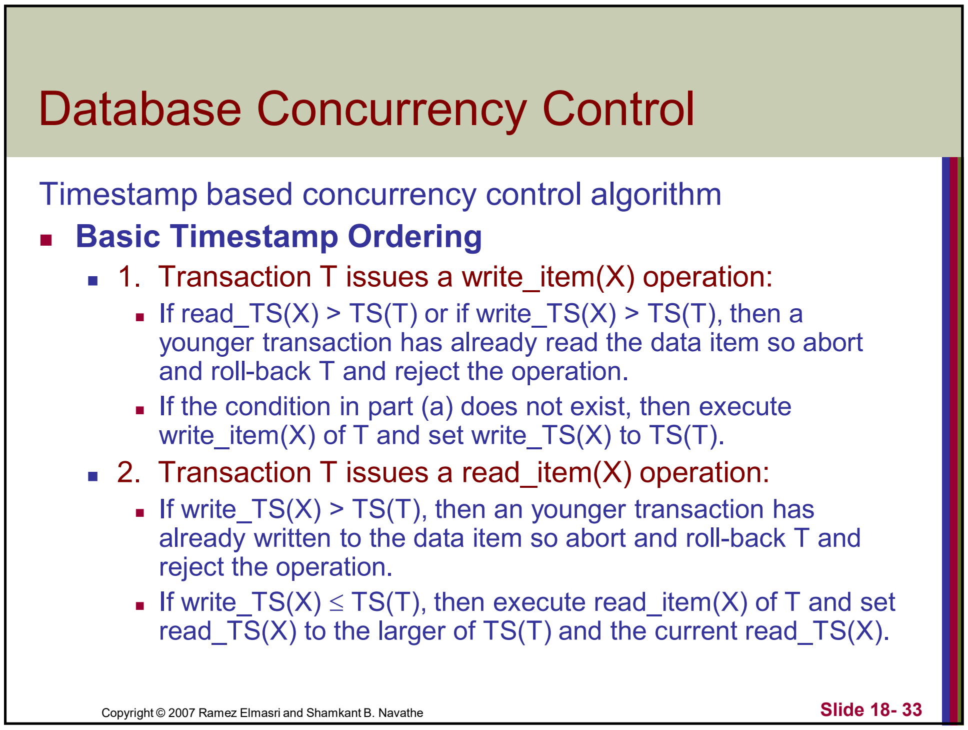
- The largest of all time stamps of transactions that have successfully written Item X
- writeTS(X)=TS(T) where T is the youngest transaction that has written X successfully



Database Concurrency Control

Timestamp based concurrency control algorithm

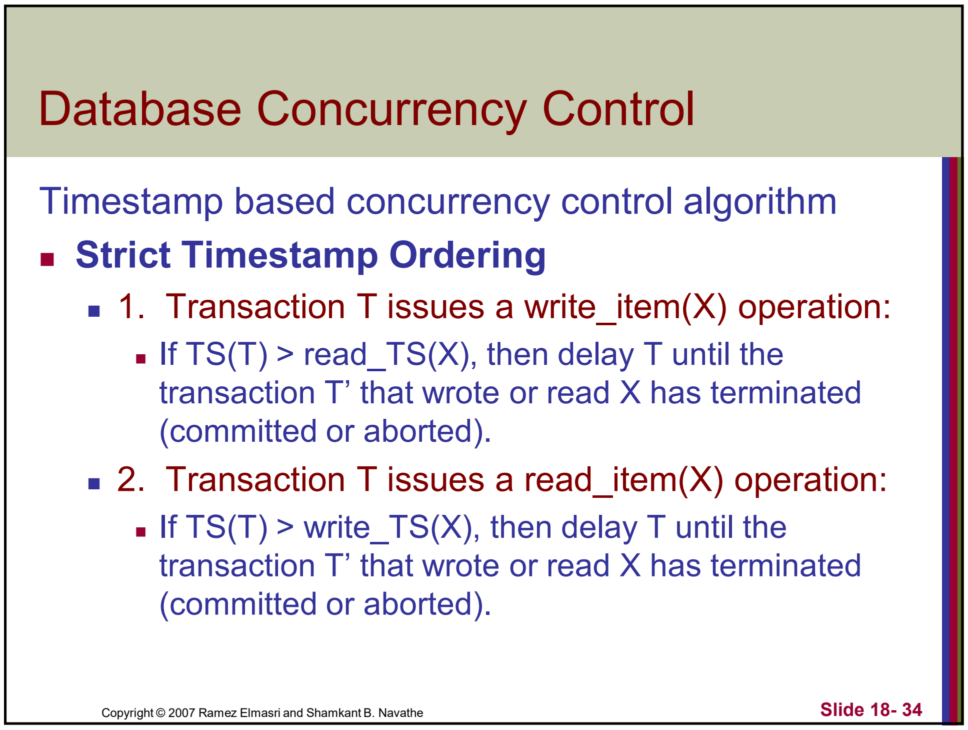
- **Basic Timestamp Ordering**
  - Whenever some transaction T tries to issue a read-item(X) or write-item(X) operation, the Basic TO algorithm compares the timestamp of T with read-TS(X) and write-TS(X) to ensure that the timestamp order of transaction execution is not violated
  - If violated, T is aborted and resubmitted as a new transaction with a new timestamp



Database Concurrency Control

Timestamp based concurrency control algorithm

- **Basic Timestamp Ordering**
  - 1. Transaction T issues a write_item(X) operation:
    - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then a younger transaction has already read the data item so abort and roll-back T and reject the operation.
    - If the condition in part (a) does not exist, then execute write_item(X) of T and set write_TS(X) to TS(T).
  - 2. Transaction T issues a read_item(X) operation:
    - If write_TS(X) > TS(T), then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
    - If write_TS(X) ≤ TS(T), then execute read_item(X) of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).



Database Concurrency Control

Timestamp based concurrency control algorithm

- **Strict Timestamp Ordering**
  - 1. Transaction T issues a write_item(X) operation:
    - If TS(T) > read_TS(X), then delay T until the transaction T’ that wrote or read X has terminated (committed or aborted).
  - 2. Transaction T issues a read_item(X) operation:
    - If TS(T) > write_TS(X), then delay T until the transaction T’ that wrote or read X has terminated (committed or aborted).

Database Concurrency Control

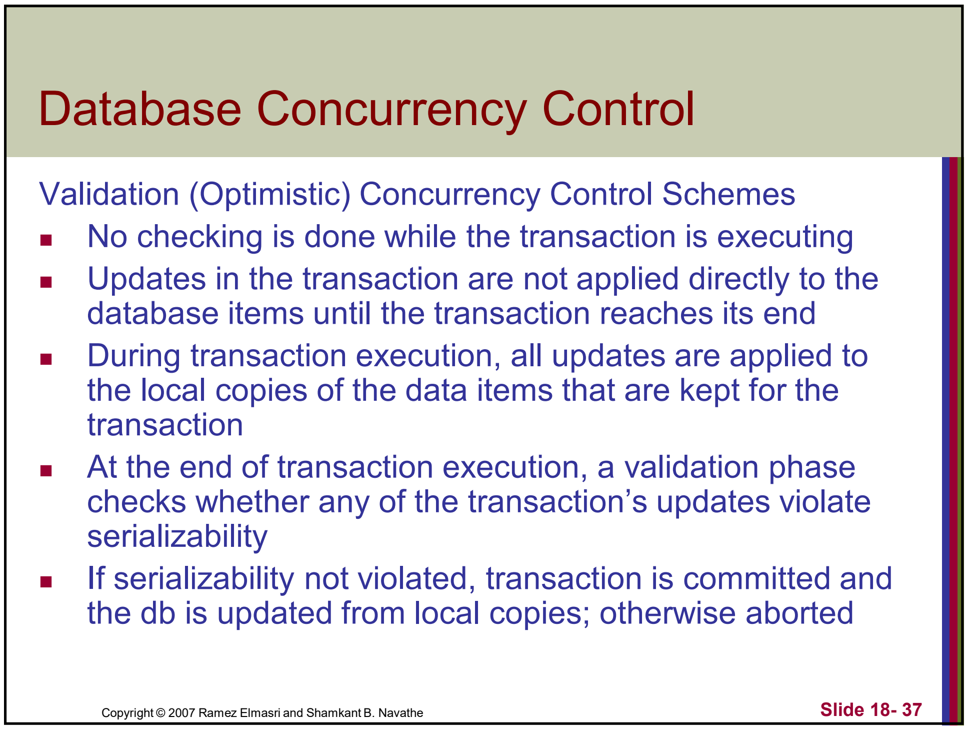
Multiversion technique based on timestamp ordering

- This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction.
  - Thus unlike other mechanisms a read operation in this mechanism is never rejected.
- Side effects: Significantly more storage (RAM and disk) is required to maintain multiple versions.

Database Concurrency Control

Multiversion technique based on timestamp ordering

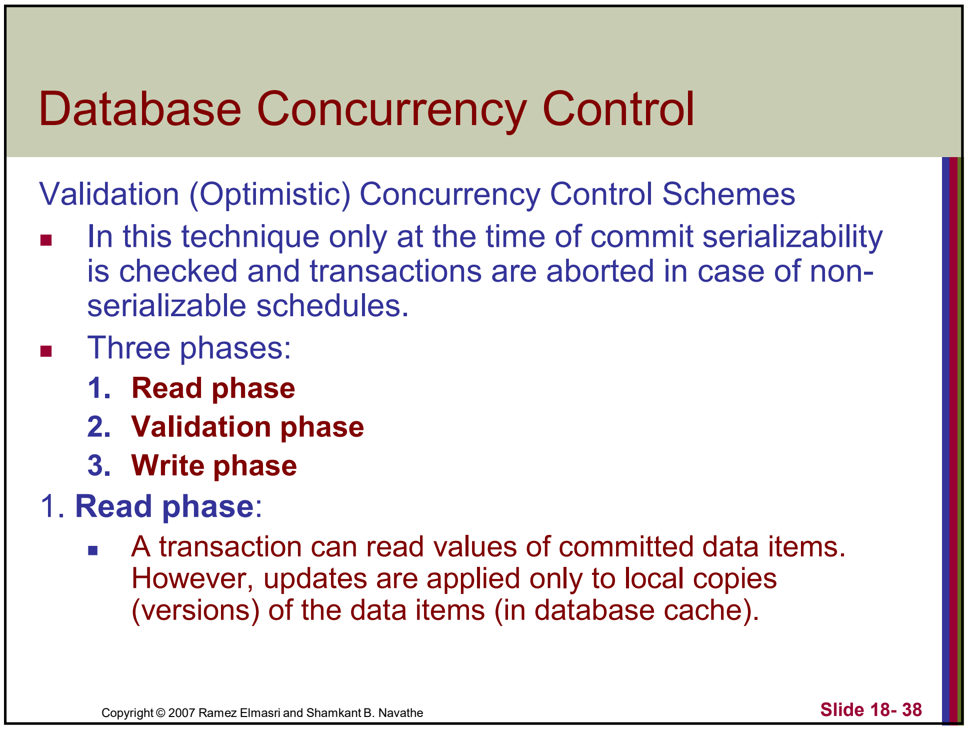
- Assume X1, X2, …, Xn are the version of a data item X created by a write operation of transactions. With each Xi a read_TS (read timestamp) and a write_TS (write timestamp) are associated.
- **read_TS(Xi)**: The read timestamp of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.
- **write_TS(Xi)**: The write timestamp of Xi that wrote the value of version Xi.
- A new version of Xi is created only by a write operation.



Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

- No checking is done while the transaction is executing
- Updates in the transaction are not applied directly to the database items until the transaction reaches its end
- During transaction execution, all updates are applied to the local copies of the data items that are kept for the transaction
- At the end of transaction execution, a validation phase checks whether any of the transaction's updates violate serializability
- If serializability not violated, transaction is committed and the db is updated from local copies; otherwise aborted

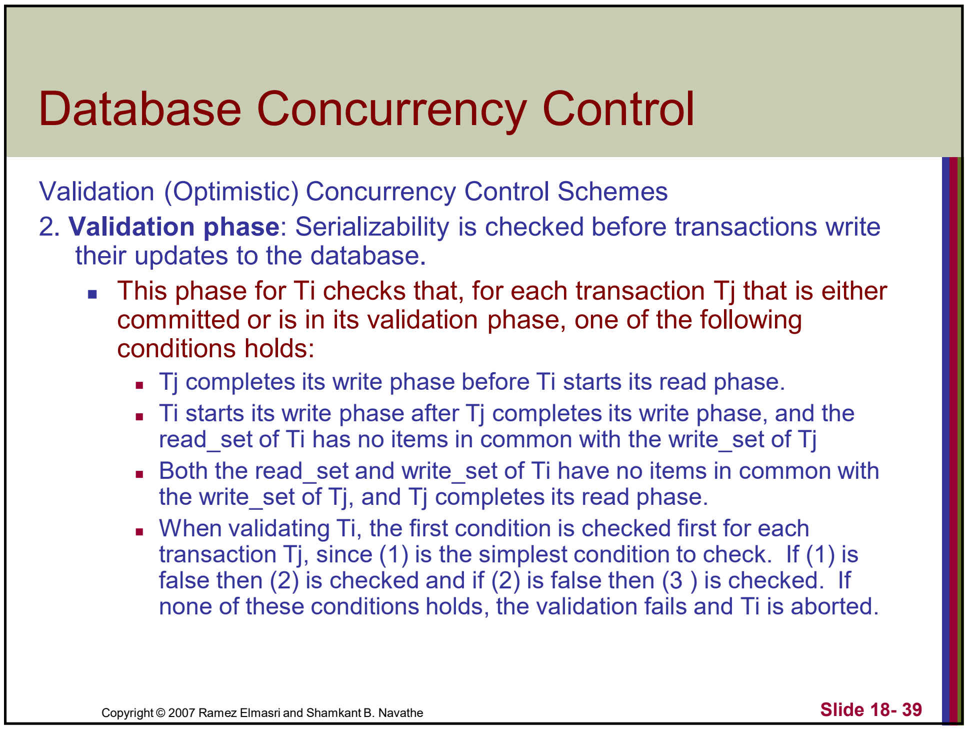


Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

- In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.
- Three phases:
  1. **Read phase**
  2. **Validation phase**
  3. **Write phase**

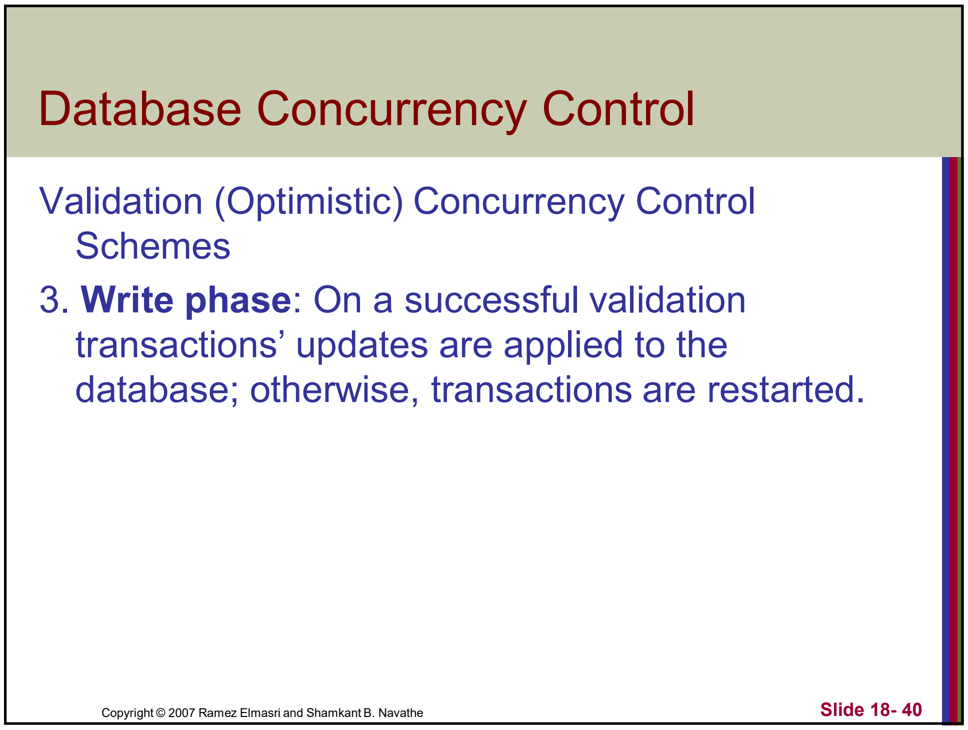
1. **Read phase**:
   - A transaction can read values of committed data items. However, updates are applied only to local copies (versions) of the data items (in database cache).



Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

2. **Validation phase**: Serializability is checked before transactions write their updates to the database.
   - This phase for Ti checks that, for each transaction Tj that is either committed or is in its validation phase, one of the following conditions holds:
     - Tj completes its write phase before Ti starts its read phase.
     - Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj
     - Both the read_set and write_set of Ti have no items in common with the write_set of Tj, and Tj completes its read phase.
     - When validating Ti, the first condition is checked first for each transaction Tj, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and Ti is aborted.



Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

3. **Write phase**: On a successful validation transactions’ updates are applied to the database; otherwise, transactions are restarted.

Database Concurrency Control

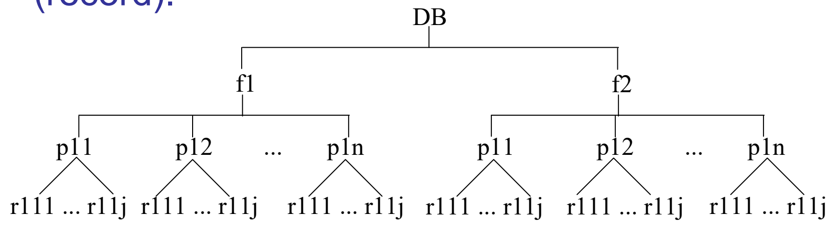
Granularity of data items and Multiple Granularity Locking

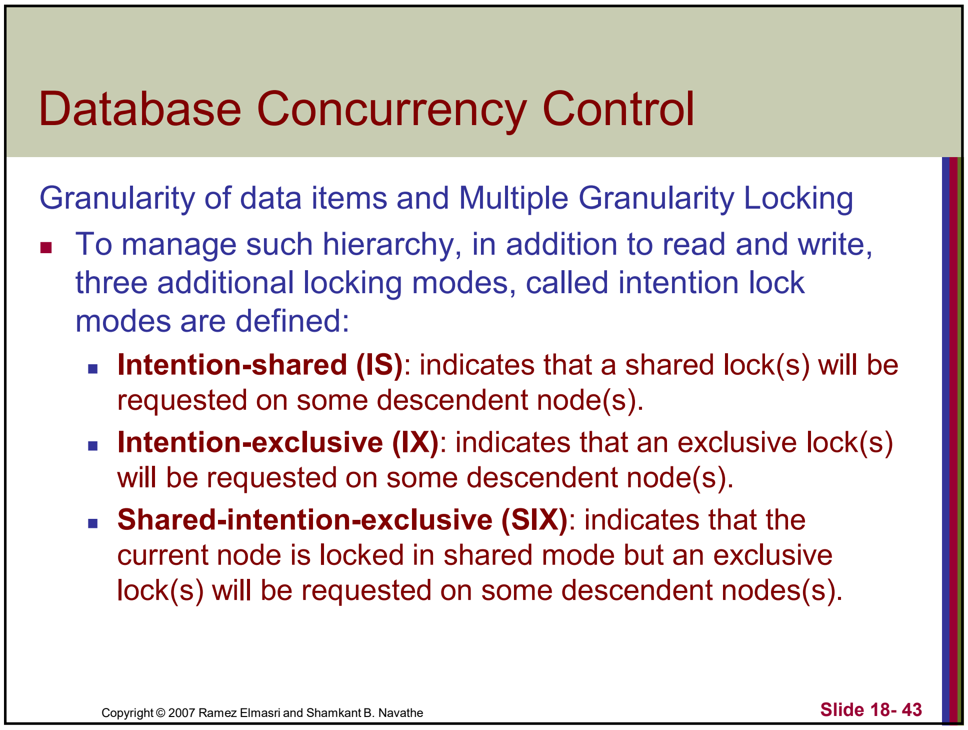
- A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation).
- Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity.
- Example of data item granularity:
  1. A field of a database record (an attribute of a tuple)
  2. A database record (a tuple or a relation)
  3. A disk block
  4. An entire file
  5. The entire database

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).

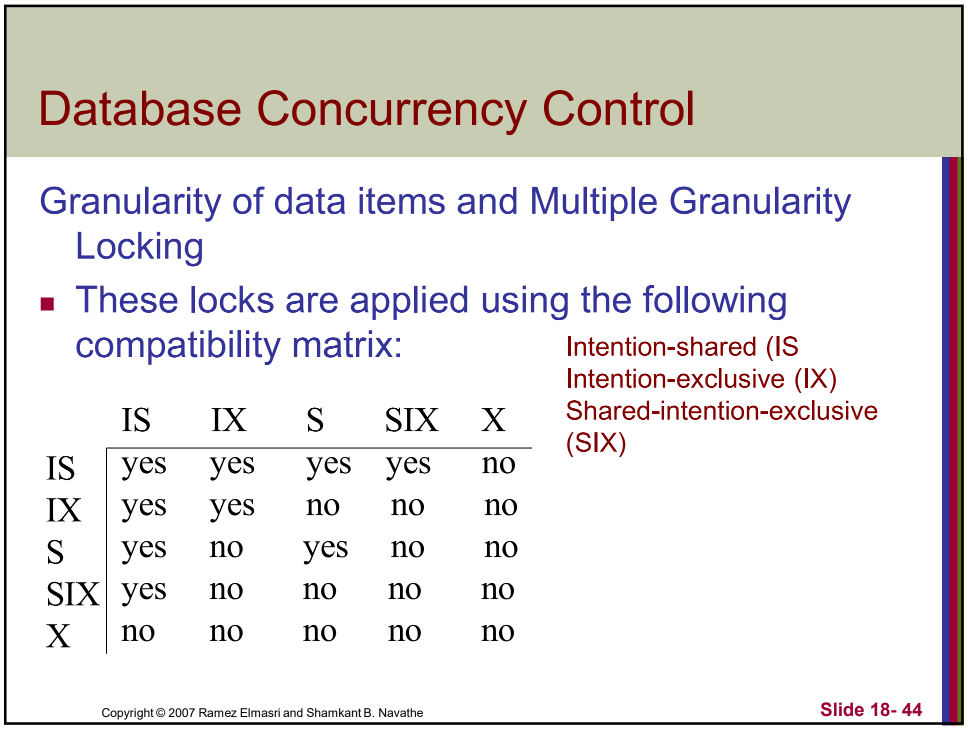




Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:
  - **Intention-shared (IS)**: indicates that a shared lock(s) will be requested on some descendent node(s).
  - **Intention-exclusive (IX)**: indicates that an exclusive lock(s) will be requested on some descendent node(s).
  - **Shared-intention-exclusive (SIX)**: indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).



Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

- These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Intention-shared (IS
Intention-exclusive (IX)
Shared-intention-exclusive (SIX)

Database Concurrency Control

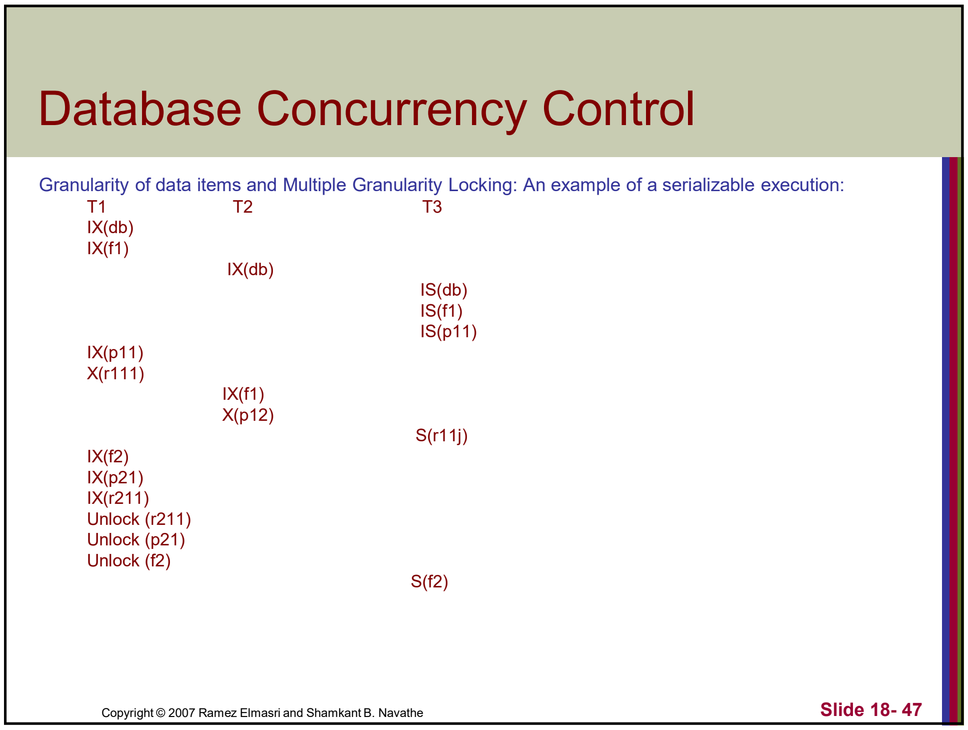
Granularity of data items and Multiple Granularity Locking

- The set of rules which must be followed for producing serializable schedule are
  1. The lock compatibility must adhered to.
  2. The root of the tree must be locked first, in any mode..
  3. A node N can be locked by a transaction T in S or IX mode only if the parent node is already locked by T in either IS or IX mode.
  4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
  5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
  6. T can unlock a node, N, only if none of the children of N are currently locked by T.

Database Concurrency Control

Example:

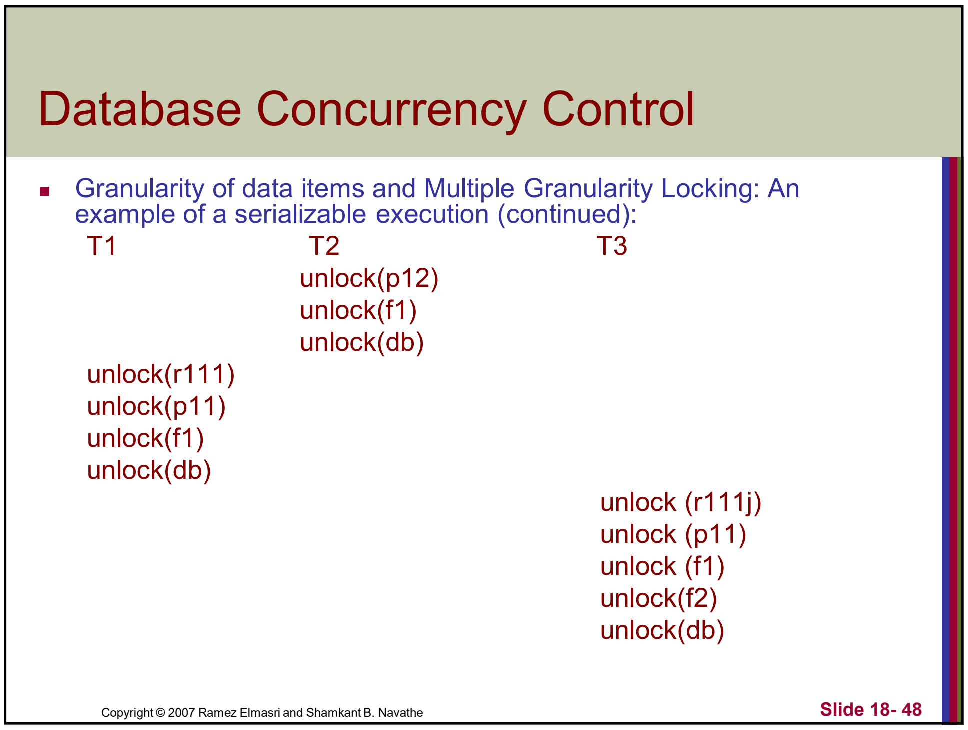
3 Transactions
1. T1 wants to update record r111 and record r211
2. T2 wants to update all records on page p12
3. T3 wants to read record r11j and the entire f2 file



Database Concurrency Control

Granularity of data items and Multiple Granularity Locking: An example of a serializable execution:

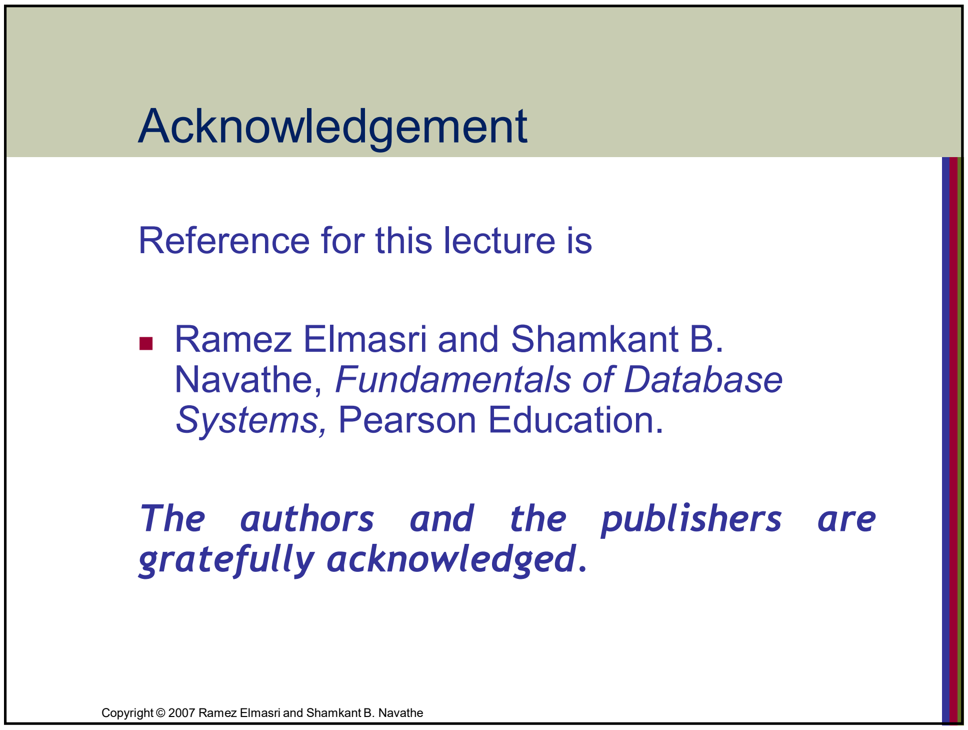
T1	T2	T3
IX(db)		
IX(f1)		
	IX(db)	
		IS(db)
		IS(f1)
		IS(p11)
IX(p11)		
X(r111)		
	IX(f1)	
	X(p12)	
		S(r11j)
IX(f2)		
IX(p21)		
IX(r211)		
Unlock (r211)		
Unlock (p21)		
Unlock (f2)		
		S(f2)



Database Concurrency Control

- Granularity of data items and Multiple Granularity Locking: An example of a serializable execution (continued):

T1	T2	T3
	unlock(p12)	
	unlock(f1)	
	unlock(db)	
unlock(r111)		
unlock(p11)		
unlock(f1)		
unlock(db)		
		unlock (r111j)
		unlock (p11)
		unlock (f1)
		unlock(f2)
		unlock(db)



Acknowledgement

Reference for this lecture is

- Ramez Elmasri and Shamkant B. Navathe, *Fundamentals of Database Systems,* Pearson Education.

The authors and the publishers are gratefully acknowledged.