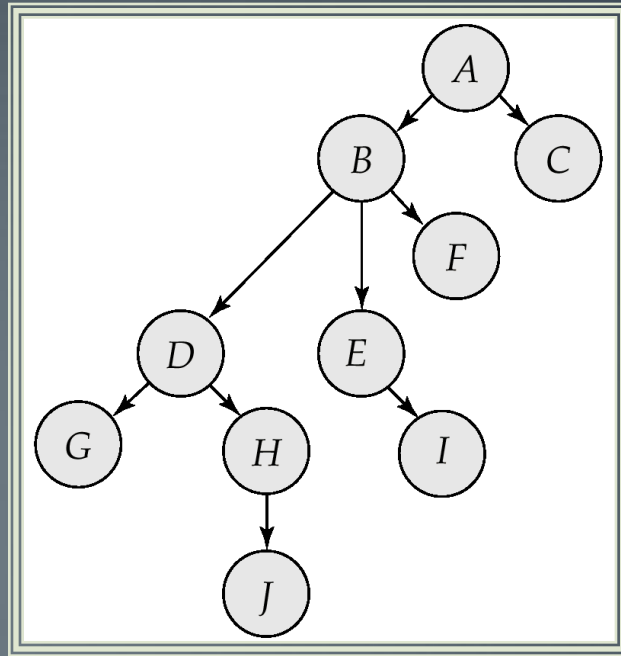


Lecture –  
Concurrency Control  
(Graph Based  
Protocol)

# Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering  $\rightarrow$  on the set  $\mathbf{D} = \{d_1, d_2, \dots, d_n\}$  of all data items.
  - If  $d_i \rightarrow d_j$  then any transaction accessing both  $d_i$  and  $d_j$  must access  $d_i$  before accessing  $d_j$ .
  - Implies that the set  $\mathbf{D}$  may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

# Tree Protocol



- Only exclusive locks are allowed.
- The first lock by  $T_i$  may be on any data item. Subsequently, a data  $Q$  can be locked by  $T_i$  only if the parent of  $Q$  is currently locked by  $T_i$ .
- Data items may be unlocked at any time.

# Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
  - the abort of a transaction can still lead to cascading rollbacks.  
(this correction has to be made in the book also.)
- However, in the tree-locking protocol, a transaction may have to lock data items that it does not access.
  - increased locking overhead, and additional waiting time
  - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under tree protocol, and vice versa.

# Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has time-stamp  $TS(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $TS(T_j)$  such that  $TS(T_i) < TS(T_j)$ .
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data  $Q$  two timestamp values:
  - **W-timestamp**( $Q$ ) is the largest time-stamp of any transaction that executed **write**( $Q$ ) successfully.
  - **R-timestamp**( $Q$ ) is the largest time-stamp of any transaction that executed **read**( $Q$ ) successfully.

# Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction  $T_i$  issues a **read**( $Q$ )
  1. If  $TS(T_i) \leq \mathbf{W}$ -timestamp( $Q$ ), then  $T_i$  needs to read a value of  $Q$  that was already overwritten. Hence, the **read** operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) \geq \mathbf{W}$ -timestamp( $Q$ ), then the **read** operation is executed, and  $\mathbf{R}$ -timestamp( $Q$ ) is set to the maximum of  $\mathbf{R}$ -timestamp( $Q$ ) and  $TS(T_i)$ .

# Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues **write**( $Q$ ).
- If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and  $T_i$  is rolled back.
- If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $Q$ . Hence, this **write** operation is rejected, and  $T_i$  is rolled back.
- Otherwise, the **write** operation is executed, and  $W\text{-timestamp}(Q)$  is set to  $TS(T_i)$ .

# Example Use of the Protocol

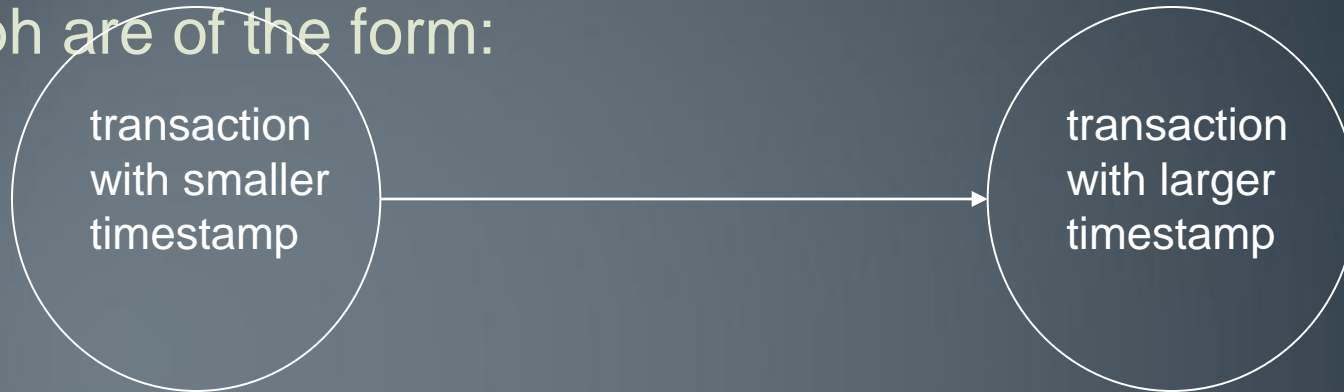
A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read(Y)	read(Y)	write(Y) write(Z)		read(X)
read(X)	read(X) abort	write(Z) abort		read(Z)
				write(Y) write(Z)



# Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

- Timestamp protocol ensures freedom from deadlock as no transaction ever waits.

- But the schedule may not be cascade-free, and may not even be recoverable.

# Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:

- Suppose  $T_i$  aborts, but  $T_j$  has read a data item written by  $T_i$
- Then  $T_j$  must abort; if  $T_j$  had been allowed to commit earlier, the schedule is not recoverable.
- Further, any transaction that has read a data item written by  $T_j$  must abort
- This can lead to cascading rollback --- that is, a chain of rollbacks

- Solution:

- A transaction is structured such that its writes are all performed at the end of its processing
- All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written
- A transaction that aborts is restarted with a new timestamp

# Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- When  $T_i$  attempts to write data item  $Q$ , if  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $\{Q\}$ . Hence, rather than rolling back  $T_i$  as the timestamp ordering protocol would have done, this **{write}** operation can be ignored.
- Otherwise this protocol is the same as the timestamp ordering protocol.
- Thomas' Write Rule allows greater potential concurrency. Unlike previous protocols, it allows some view-serializable schedules that are not conflict-serializable.

# Validation-Based Protocol

- Execution of transaction  $T_i$  is done in three phases.
  1. **Read and execution phase:** Transaction  $T_i$  writes only to temporary local variables
  2. **Validation phase:** Transaction  $T_i$  performs a "validation test" to determine if local variables can be written without violating serializability.
  3. **Write phase:** If  $T_i$  is validated, the updates are applied to the database; otherwise,  $T_i$  is rolled back.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation

# Validation-Based Protocol (Cont.)

- Each transaction  $T_i$  has 3 timestamps

- **Start**( $T_i$ ) : the time when  $T_i$  started its execution

- **Validation**( $T_i$ ): the time when  $T_i$  entered its validation phase

- **Finish**( $T_i$ ) : the time when  $T_i$  finished its write phase

- Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus  $TS(T_i)$  is given the value of **Validation**( $T_i$ ).

- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low. That is because the serializability order is not pre-decided and relatively less transactions will have to be rolled back.

# Validation Test for Transaction $T_j$

If for all  $T_i$  with  $TS(T_i) < TS(T_j)$  either one of the following condition holds:

- **finish**( $T_i$ ) < **start**( $T_j$ )
- **start**( $T_j$ ) < **finish**( $T_i$ ) < **validation**( $T_j$ ) and the set of data items written by  $T_i$  does not intersect with the set of data items read by  $T_j$ .

then validation succeeds and  $T_j$  can be committed. Otherwise, validation fails and  $T_j$  is aborted.

*Justification:* Either first condition is satisfied, and there is no overlapped execution, or second condition is satisfied and

1. the writes of  $T_j$  do not affect reads of  $T_i$  since they occur after  $T_i$  has finished its reads.
2. the writes of  $T_i$  do not affect reads of  $T_j$  since  $T_j$  does not read any item written by  $T_i$ .

# Schedule Produced by Validation

- Example of schedule produced using validation

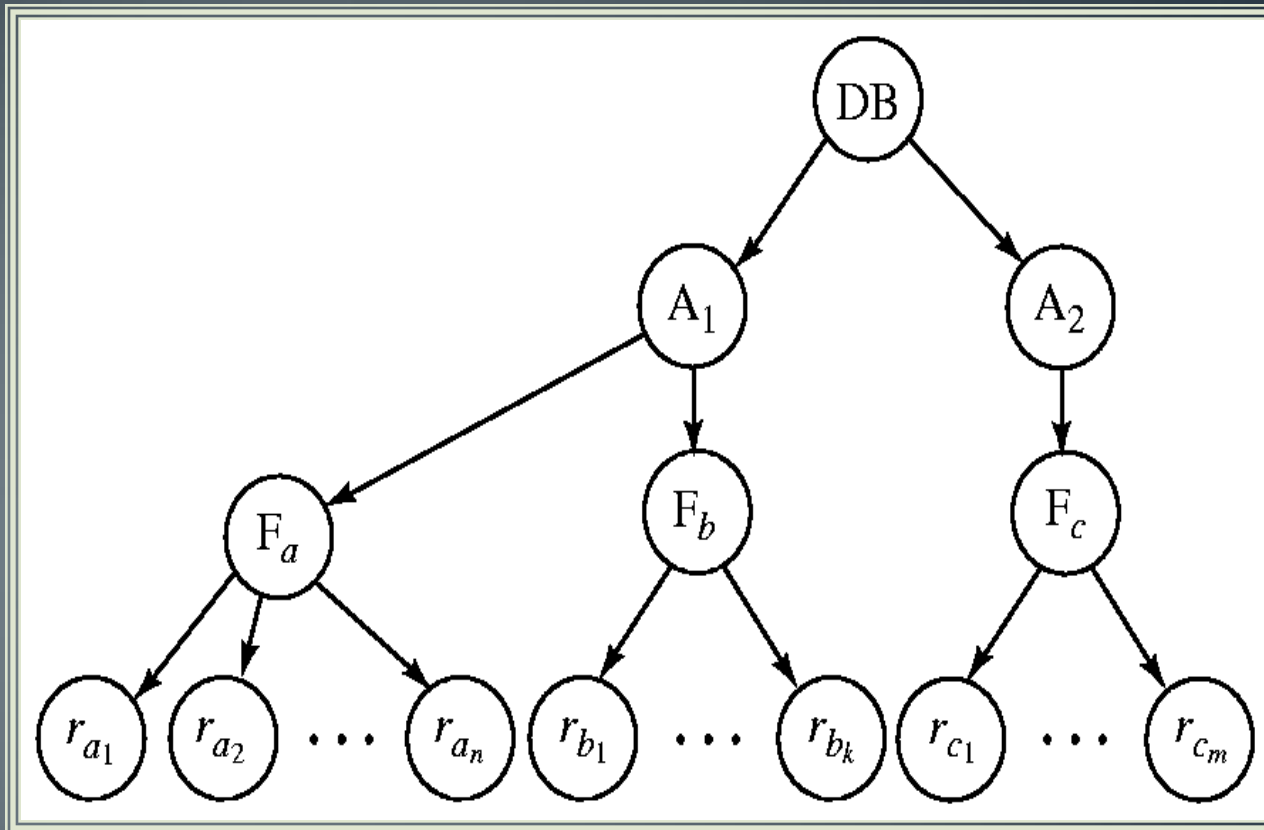
$T_{14}$	$T_{15}$
<b>read(B)</b>	<b>read(B)</b>
	<i>B:- B-50</i>
	<b>read(A)</b>
	<i>A:- A+50</i>
<b>read(A)</b>	
<i>(validate)</i>	
<b>display (A+B)</b>	<i>(validate)</i>
	<b>write (B)</b>
	<b>write (A)</b>

# 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.