IF3140 – Sistem Basis Data

Concurrency Control:

- Graph-based protocols
- Deadlock handling
- Multiple granularity
- Timestamp-Based Protocols

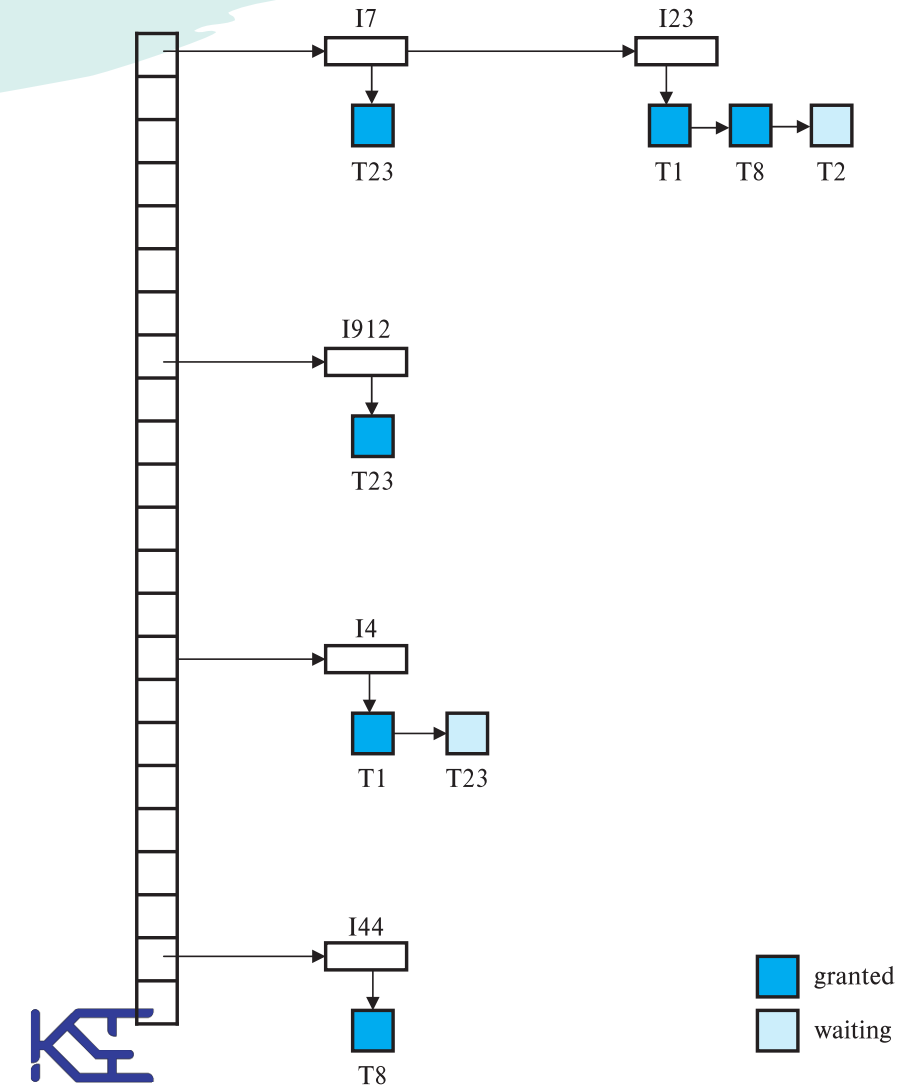
KNOWLEDGE & SOFTWARE ENGINEERING

Implementation of Locking

- A **lock manager** can be implemented as a separate process
- Transactions can send lock and unlock requests as messages
- 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 an in-memory data-structure called a **lock table** to record granted locks and pending requests

KNOWLEDGE & SOFTWARE ENGINEERING

Lock Table



- Dark rectangles indicate granted locks, light colored ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

KNOWLEDGE & SOFTWARE ENGINEERING

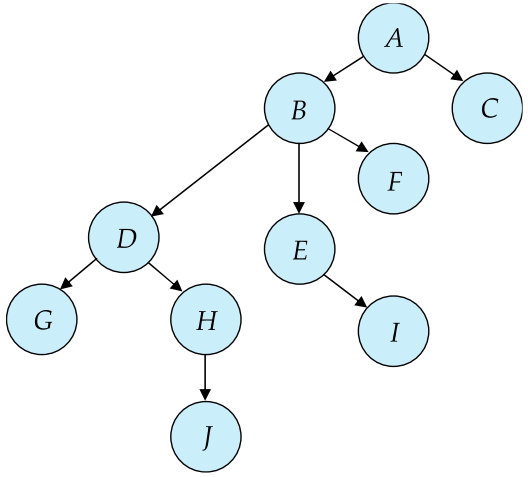
Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, ..., d_h\}$ 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 **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

KNOWLEDGE & SOFTWARE ENGINEERING

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.
- A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



KNOWLEDGE & SOFTWARE ENGINEERING

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
- Drawbacks
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency
- Schedules not possible under two-phase locking are possible under the tree protocol, and vice versa.

KNOWLEDGE & SOFTWARE ENGINEERING

Deadlock Handling

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.

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)	

Deadlock Handling

↳ setiap transaksi hrs menyebutkan duluhnya apa
trus nanti di atur biar gak deadlock
tpi problemnya cuirnya jedet gak tau transaksinya butuh apa aja

- ***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 (pre-declaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger 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 a lock
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transactions is restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

KNOWLEDGE & SOFTWARE ENGINEERING

Deadlock prevention (Cont.)

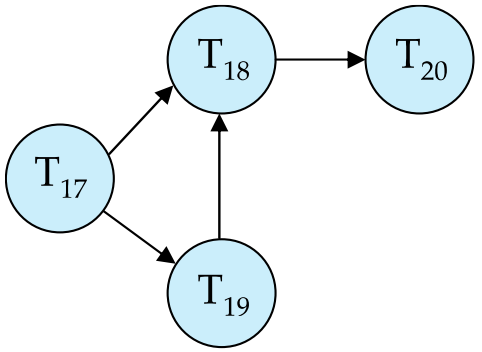
deadlock dtogak tetap
rollback

- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily in absence of deadlock
    - Difficult to determine good value of the timeout interval.
  - Starvation is also possible

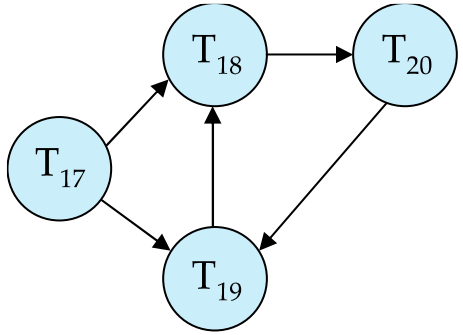
KNOWLEDGE & SOFTWARE ENGINEERING

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and then restart it.
    - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim

Exercise 9.4 – Deadlock Prevention

Instructions from T1, T2, and T3 arrive in the following order (the same as Exercise 9.3).

R1(A); R2(A); R3(B); W1(A); R2(C); R2(B); C3; W2(B); C2; W1(C); C1;

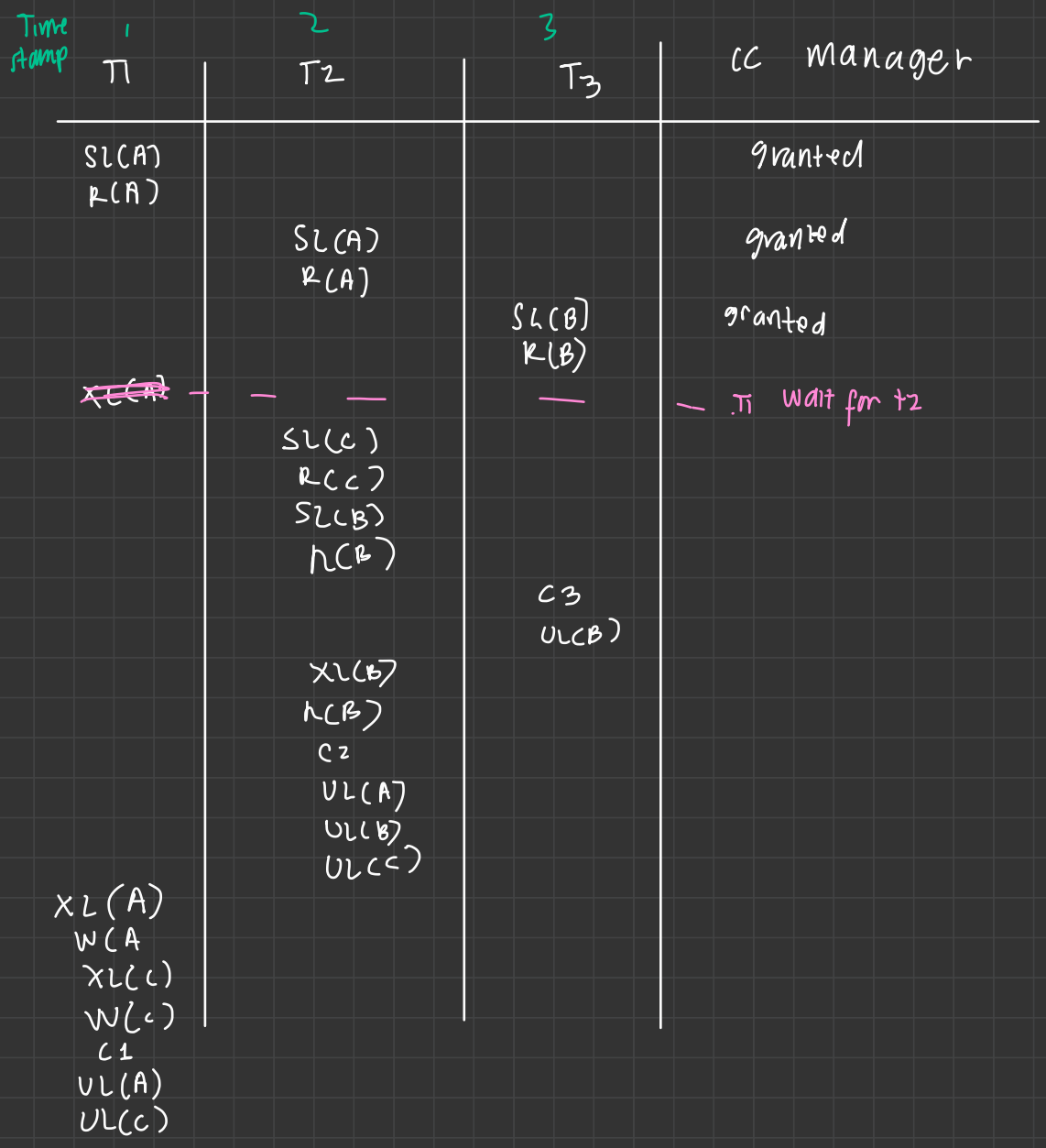
What is the final *schedule* if the *2-phase locking with automatic acquisition of locks* with

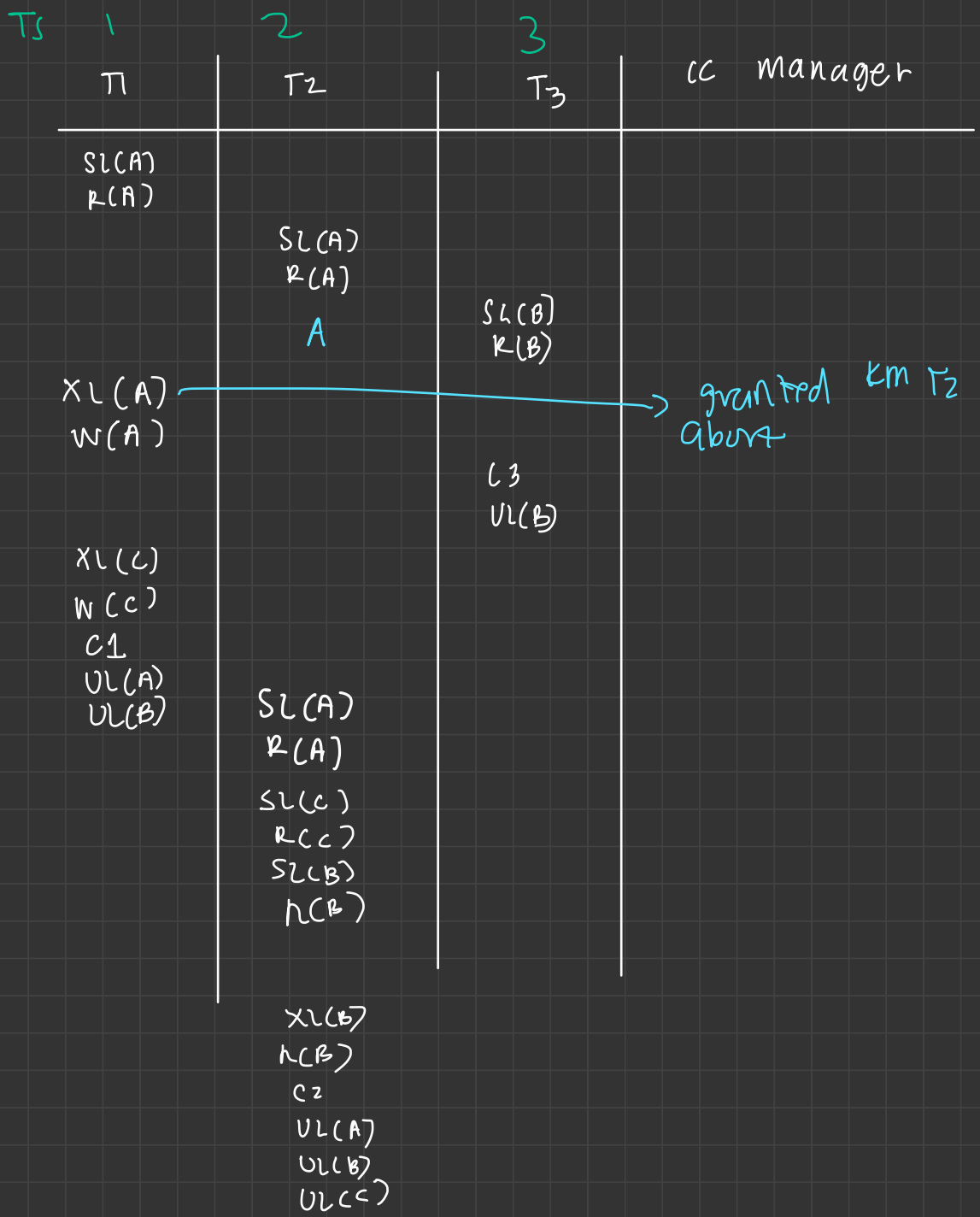
a. wait-die deadlock prevention scheme
b. wound-wait deadlock prevention scheme

is implemented by CC Manager?
Assume that Timestamp (T1,T2,T3)=(1,2,3)

KNOWLEDGE & SOFTWARE ENGINEERING

Solution


Time stamp
1
2
3
T1
T2
T3
CC manager
SL(A)
R(A)
granted
SL(A)
R(A)
granted
SL(B)
R(B)
granted
XL(A)
T1 wait for T2
SL(C)
R(C)
SL(B)
R(B)
C3
UL(B)
XL(B)
R(B)
C2
UL(A)
UL(B)
UL(C)
XL(A)
W(A
XL(C)
W(C)
C1
UL(A)
UL(C)
→ wait and die


T5
1
2
3
T1
T2
T3
CC manager
SL(A)
R(A)
SL(A)
R(A)
A
SL(B)
R(B)
XL(A)
W(A)
granted
about
to T2
C3
UL(B)
XL(C)
W(C)
C1
UL(A)
UL(B)
SL(A)
R(A)
SL(C)
R(C)
SL(B)
R(B)
XL(B)
R(B)
C2
UL(A)
UL(B)
UL(C)

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 descendants 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

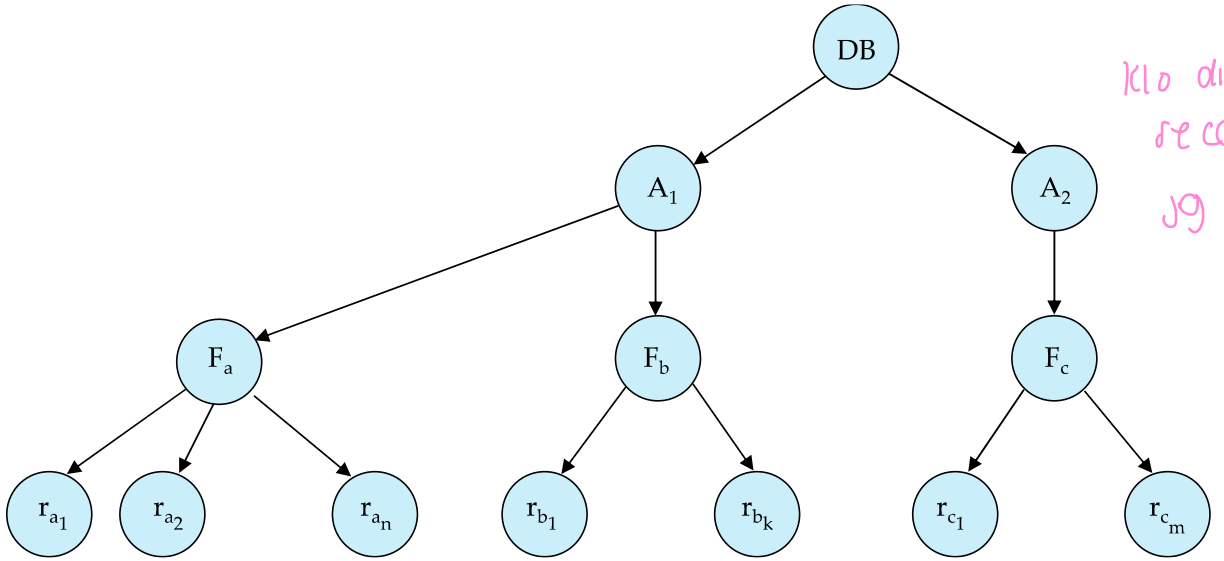
KNOWLEDGE & SOFTWARE ENGINEERING

Example of Granularity Hierarchy

↳ data punyo level x kita bisa locking

The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*



klo di lock di satu level secara implisit level bawah yg di lock

Intention Lock Modes

disini dibaca dibuat

- 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.

KNOWLEDGE & SOFTWARE ENGINEERING

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

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.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

KNOWLEDGE & SOFTWARE ENGINEERING

Timestamp Based Concurrency Control

Timestamp-Based Protocols

- Each transaction T_i is issued a timestamp TS(T_i) when it enters the system.
  - Each transaction has a *unique* timestamp
  - Newer transactions have timestamps strictly greater than earlier ones
  - Timestamp could be based on a logical counter
    - Real time may not be unique
    - Can use (wall-clock time, logical counter) to ensure
- Timestamp-based protocols manage concurrent execution such that

  time-stamp order = serializability order
- Several alternative protocols based on timestamps

KNOWLEDGE & SOFTWARE ENGINEERING

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) 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. → siapa yg terakhir sukses nge-write
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully. → siapa yg terakhir sukses read
- Imposes rules on read and write operations to ensure that
  - Any conflicting operations are executed in timestamp order
  - Out of order operations cause transaction rollback

KNOWLEDGE & SOFTWARE ENGINEERING

Timestamp-Based Protocols (Cont.)

- Suppose a transaction T_i issues a **read**(Q)

  1. If TS(T_i) < **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(Ti) ≥ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to

     max(R-timestamp(Q), TS(T_i)).

Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < R-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.
  2. If TS(T_i) < W-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.
  3. Otherwise, the **write** operation is executed, and W-timestamp(Q) is set to TS(T_i).

KNOWLEDGE & SOFTWARE ENGINEERING

Example of Schedule Under TSO

- Is this schedule valid under TSO?

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Assume that initially:
R-TS(A) = W-TS(A) = 0
R-TS(B) = W-TS(B) = 0
Assume TS(T_{25}) = 25 and
TS(T_{26}) = 26

- How about this one, where initially
R-TS(Q)=W-TS(Q)=0

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

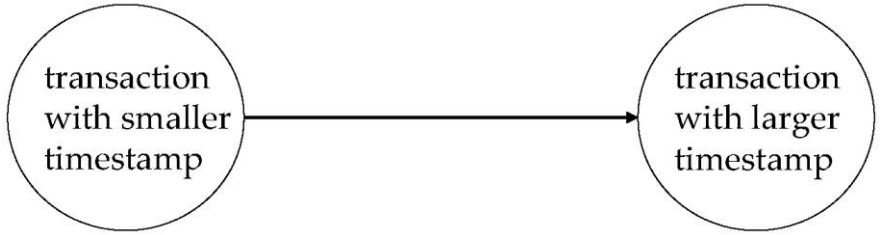
Another Example Under TSO

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5, with all R-TS and W-TS = 0 initially

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

KNOWLEDGE & SOFTWARE ENGINEERING

Recoverability and Cascade Freedom

- Solution 1:
  - 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
- Solution 2:
  - Limited form of locking: wait for data to be committed before reading it
- Solution 3:
  - Use commit dependencies to ensure recoverability

KNOWLEDGE & SOFTWARE ENGINEERING

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-timestamp(Q), then T_i is attempting to write an obsolete value of {Q}.
  - 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.
  - Allows some view-serializable schedules that are not conflict-serializable.

KNOWLEDGE & SOFTWARE ENGINEERING

Exercise 9.5

Instructions from T1, T2, and T3 arrive in the following order.

R1(A); R2(A); R3(B); R2(C); R2(B); W1(A); W1(C); C1; C3; W2(B); C2;

What is the final *schedule* if the *timestamp ordering protocol* is implemented by CC Manager?

Assume that Timestamp (T1,T2,T3)=(1,2,3)

KNOWLEDGE & SOFTWARE ENGINEERING

Solution

KNOWLEDGE & SOFTWARE ENGINEERING