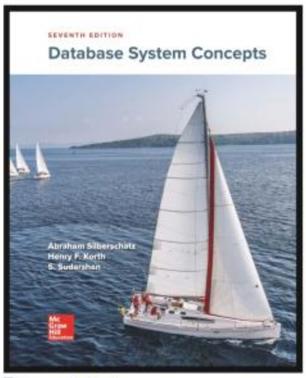
Concurrency Control Pt. 1 & 2

W09-1-Concurrency...

IF3140 – Sistem Basis Data
Concurrency Control:
- Introduction
- Lock-based Protocol



Sumber

Silberschatz, Korth, Sudarshan: "Database System Concepts", 7th Edition
- Chapter 18: Concurrency Control

Objectives

Students are able to:

- Explain the effect of different isolation levels on the concurrency control mechanisms
- Choose the proper isolation level for implementing a specified transaction protocol

---

Outline

mendapat lock
untuk item data yang ingin digunakan

- Lock-Based Protocols
  - 2-phase locking
  - Graph-based protocols
  - Deadlock handling
  - Multiple granularity
- Timestamp-Based Protocols

setiap transaksi diberi timestamp, urutan timestamp diatur sedemikian rupa sehingga ekuivalen dengan schedule serial

- Validation-Based Protocols → validasi di akhir sebelum commit
- Insert and delete operations
- Multiversion Schemes
  - MV Timestamp ordering

mendapat lock
untuk item data yang ingin digunakan

4

Outline

- Lock-Based Protocols
  - 2-phase locking
  - Graph-based protocols
  - Deadlock handling
  - Multiple granularity
- Timestamp-Based Protocols
- Validation-Based Protocols
- Insert and delete operations
- Multiversion Schemes
  - MV Timestamp ordering
  - MV 2-phase locking
  - Snapshot isolation
- Weak levels of consistency

setiap transaksi diberi timestamp, urutan timestamp diatur sedemikian rupa sehingga ekuivalen dengan schedule serial

→ validasi di akhir sebelum commit

5

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. **exclusive** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. **shared** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

karena bisa dipakai oleh >1 transaksi

6

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

hanya shared lock yang bisa kompatibel dengan banyak transaksi

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

Schedule With Lock Grants

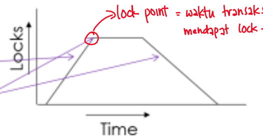
- Grants omitted in rest of chapter
  - Assume grant happens just before the next instruction following lock request
- This schedule is not serializable (why?) karena T_2 membaca nilai A awal, tapi nilai B sudah diubah T_1 <tidak bisa dikatakan T_1 / T_2 dulu>
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- Locking protocols enforce serializability by restricting the set of possible schedules.

atasi dengan locking protocol

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

The Two-Phase Locking Protocol (2PL)

- A protocol which ensures conflict-serializable schedules. growing → jumlah lock ++
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase → hanya boleh lepas lock
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).



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)	

dipegang T_3

dipegang T_4

- 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 (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.

    ada transaksi yang terus menunggu
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure recoverability or freedom from cascading roll-back
  - **Strict two-phase locking:** a transaction must hold all its exclusive locks till it commits/aborts. → pegang X-lock sampai transaksi commit supaya data yang diubah oleh transaksi saat ini hanya bisa digunakan oleh transaksi lain setelah commit
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort. → pegang SEMUA lock sampai transaksi commit
    - Transactions can be serialized in the order in which they commit.
- Most databases implement rigorous two-phase locking, *but refer to it as simply two-phase locking*

The Two-Phase Locking Protocol (Cont.)

banyak kasus conflict serializable yang melanggar "tidak boleh lock setelah unlock"

- Two-phase locking is not a necessary condition for serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.
- In the absence of extra information (e.g., ordering of access to data), two-phase locking is necessary for conflict serializability *in the following sense*:
  - *Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.*

kalau 2PL pasti conflict serializable

T_1	T_2
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
unlock(B)	
	lock-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display($A + B$)
lock-X(A)	
read(A)	
$A := A + 50$	
write(A)	
unlock(A)	

Locking Protocols

- Given a locking protocol (such as 2PL)
  - A schedule S is **legal** under a locking protocol if it can be generated by a set of transactions that follow the protocol
  - A protocol **ensures** serializability if all legal schedules under that protocol are serializable

Exercise 9.1

Consider the following two transactions:

T1:	**read (A)**	**T2:**	**read (B)**
	read (B)		**read (A)**
	B := B + 0.1*A		**A := A – 0.05*B**
	write (B)		**write (A)**

untuk T1,
A → baca
B → baca + tulis
untuk T2,
A → baca + tulis
B → baca

a) Add lock and unlock instructions to both transactions so that they follow the two-phase locking protocol.

untuk T1,
lock-S (A)
lock-X (B)
— — — — growing phase
read (A)
read (B)
B := B + 0,1A
write (B)
— — — — shrinking phase
unlock (B)
unlock (A)

untuk T2,
lock-S (B)
lock-X (A)
— — — — growing phase
read (B)
read (A)
A := A - 0,05 (B)
write (A)
— — — — shrinking phase
unlock (A)
unlock (B)

b) Can the execution of these two transactions result in a deadlock?

bisa, ketika T1 memegang S-lock (A), T2 memegang S-lock (B), kemudian T1 mau X-lock (B), namun T2 mau X-lock (A)

Solution

Exercise 9.2

Is the following schedule a two-phase locking (2PL) schedule (legal under 2PL protocol)?

R1(A); R2(A); R3(B); W1(A); R2(C); R2(B); W2(B); W1(C);

T1	T2	T3
R1 (A) S-lock (A)		
	R2 (A) S-lock (A)	
		R3 (B) S-lock (B)
W1 (A) X-lock (A)		
	R2 (C) S-lock (C)	
	R2 (B)	
	W2 (B)	
W1 (C)		

Solution

Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can **convert** a lock-S to a lock-X (**upgrade**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

```
if Ti has a lock on D
  then
      read(D)
  else begin
        if necessary wait until no other
           transaction has a lock-X on D
        grant Ti a lock-S on D;
        read(D)
      end
```


Automatic Acquisition of Locks (Cont.)

- The operation **write**(*D*) is processed as:

```
if Ti has a lock-X on D
  then
   write(D)
  else begin
    if necessary wait until no other trans. has any lock on D,
    if Ti has a lock-S on D
       then
         upgrade lock on D to lock-X
      else
         grant Ti a lock-X on D
      write(D)
   end;
```

- **All locks are released after commit or abort**

Exercise 9.3

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

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* is implemented by CC Manager?

Solution

final schedule
$SL_1(A)$; $R1(A)$
$SL_2(A)$, $R2(A)$
$SL_3(B)$, $R3(B)$
waiting : $W1(A)$
$SL_2(C)$
$SL_2(B)$
C_3, $UL_3(B)$
$XL_2(B)$; $W_2(B)$
C_2 ; $UL_2(A)$, $UL_2(B)$; $UL_2(C)$
$XL_1(A)$, $W_1(A)$
$XL_1(C)$, $W_1(C)$
C_1, $UL_1(A)$, $UL_1(C)$

W09-2-Concurrency...

IF3140 – Sistem Basis Data

Concurrency Control:

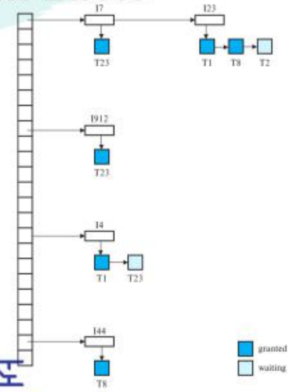
- Graph-based protocols
- Deadlock handling
- Multiple granularity
- Timestamp-Based Protocols

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

Solution

Lock Table



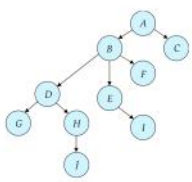
- Dark rectangles indicate granted locks, light colored ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests 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

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.

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



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.

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

- ***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).

9

More Deadlock Prevention Strategies

- **wait-die** scheme — non-preemptive   tidak bisa di-interrupt
  - 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

  transaksi lama menunggu transaksi baru

- **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.

  transaksi lama bisa "wound" transaksi baru → transaksi baru menunggu transaksi lama

- 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.

10

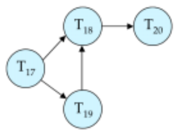
Deadlock prevention (Cont.)

- **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

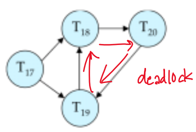
11

Deadlock Detection

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \to T_j$. : 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 <u>*2-phase locking with automatic acquisition of locks*</u> 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)

Solution

(a) $SL_1(A)$, $R1(A)$
$SL_2(A)$, $R2(A)$
$SL_3(B)$, $R3(B)$
waiting = $W1(A)$
$SL_2(C)$, $R_2(C)$
$SL_2(B)$, $R_2(B)$
$C3$, $UL_3(B)$
$XL_2(B)$; $W_2(B)$
$C2$, $UL_2(A)$, $UL_2(B)$
$XL_1(A)$, $W_1(A)$
$XL_1(C)$, $W_1(C)$
C_1, $UL_1(A)$, $UL_1(B)$; $UL_1(C)$

(b) $SL_1(A)$, $R_1(A)$
$SL_2(A)$; $R_2(A)$
$SL_3(B)$, $R_3(B)$
$T_1 < T_2 \rightarrow$ wound T_2
rollback T_2, $UL_2(A)$ $\longrightarrow$ sisa T_2 diabaikan
$XL_1(A)$, $W_1(A)$
C_3, $UL_3(B)$
$XL_1(C)$; $W_1(C)$
C_1, $UL_1(A)$; $UL_1(C)$

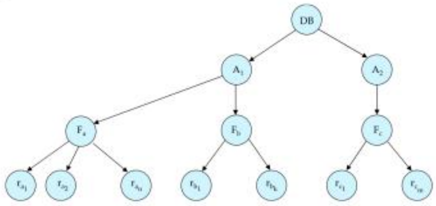
Multiple Granularity

- 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

Example of Granularity Hierarchy

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

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



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	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

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

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.
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully.
- 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

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).

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)	

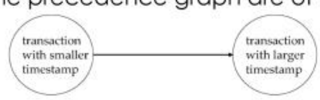
Another Example Under TSO

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.

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

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.

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 <u>*timestamp ordering protocol*</u> is implemented by CC Manager?

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

Solution

LATIHAN

1. R1(X), W2 (X), W2(Y), W3(Y), W1(X), C1, C2, C3

T1	T2	T3	CC Manager
R1(X)			XL1(X), R1 (X)
	W2(X)		wait for XL (X), queue. W2(X)
	W2(Y)		queue W2(X), W2(Y)
		W3(Y)	XL3(Y), W3 (Y)
W1(X)			W1 (X)
C1			UL(X), C1
	W2 (X)		XL2(X), W2(X)
	W2(Y)		wait for XL(Y), queue W2(Y)
	C2		queue. W2(X), C2
		C3	UL(Y), C3
	W2(Y)		XL2(Y); W2(Y)
	C2		UL(X); UL(Y), C2

2. (1) konkuren deadlock

T1	T2	CC Manager

$R_1(E)$		$SL_1(E)$, $R_1(E)$
	$R_2(G)$	$SL_2(G)$, $R_2(G)$
$R_1(F)$		$XL_1(F)$, $R_1(F)$
	$R_2(F)$	wait for $XL(F)$, queue $R_2(F)$
$R_1(G)$		wait for $XL(G)$, queue $R_1(G)$

$$T_1 \rightleftarrows T_2$$

(2) deadlock recovery

↳ rollback salah satu transaksi
selesaikan transaksi lain,
restart transaksi yang di-rollback

(3) - partial ordering

- wait-die scheme
  ↳ saat T_2 meminta $XL(F)$, karena $TS(T_1) < TS(T_2)$
  T_2 harus rollback
- wound-wait scheme
  ↳ saat T_1 meminta $XL(G)$, T_1 bisa "melukai" T_2
  sehingga T_2 rollback melepas lock G