KOREA UNIVERSITY
DATAX LAB

Chapter 18 :

Concurrency Control

Outline

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
    - Snapshot Isolation

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.
  - Until the lock is granted, the lock-requesting transaction must wait !!

Lock-Based Protocols (Cont.)

- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- 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

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)		

- **Notice**: Locking like this is not sufficient to guarantee serializability
  - This schedule is not serializable (why?)
- 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.

'*grant*' operations will be omitted in rest of chapter

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)	

4 conditions for deadlock

Mutual Exclusion
Hold & wait
Circular wait
Non-preemption

- 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.
- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.

Starvation

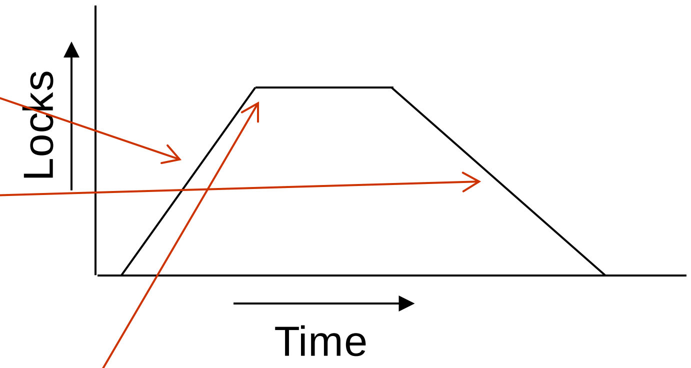
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.
- Concurrency control manager can be designed to prevent starvation.

The Two-Phase Locking (2PL) Protocol

- A protocol which ensures *conflict-serializable* schedules.

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.

- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - 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).



The Two-Phase Locking Protocol (Cont.)

- Two-phase locking is *not a necessary condition* for conflict serializability
  - There are conflict serializable schedules that cannot be obtained if the two-phase locking protocol is used.
- Two-phase locking *does not* ensure freedom from deadlocks
- Extensions to basic two-phase locking needed to ensure the freedom from *cascading rollbacks*
  - **Strict two-phase locking:** a transaction must hold all *exclusive* locks until it commits/aborts.
    - Avoids cascading roll-backs (and thus ensures recoverability)
  - **Rigorous two-phase locking**: a transaction must hold *all* locks till commit/abort.
    - 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*

Lock Conversion

- Upgrade: convert a shared lock into an exclusive lock
- Downgrade: convert an exclusive lock into a shared lock

T_8: read(a_1);
read(a_2);
. . .
read(a_n);
write(a_1).

T_9: read(a_1);
read(a_2);
display($a_1 + a_2$).

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

T_8 initially locks a_1 in a shared mode, and at the end of the transaction, upgrades it into an exclusive lock.
More concurrency can be achieved by allowing T_9's reads during the period of T_8's reads.

2PL with 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 also ensures *conflict 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 T_i has a lock on *D* **then**

  read(*D*) ;

  else begin

  if necessary wait until no other

  transaction has a **lock-X** on *D* ;

  grant T_i a **lock-S** on *D* ;

  read(*D*) ;

  end ;

Automatic Acquisition of Locks (Cont.)

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

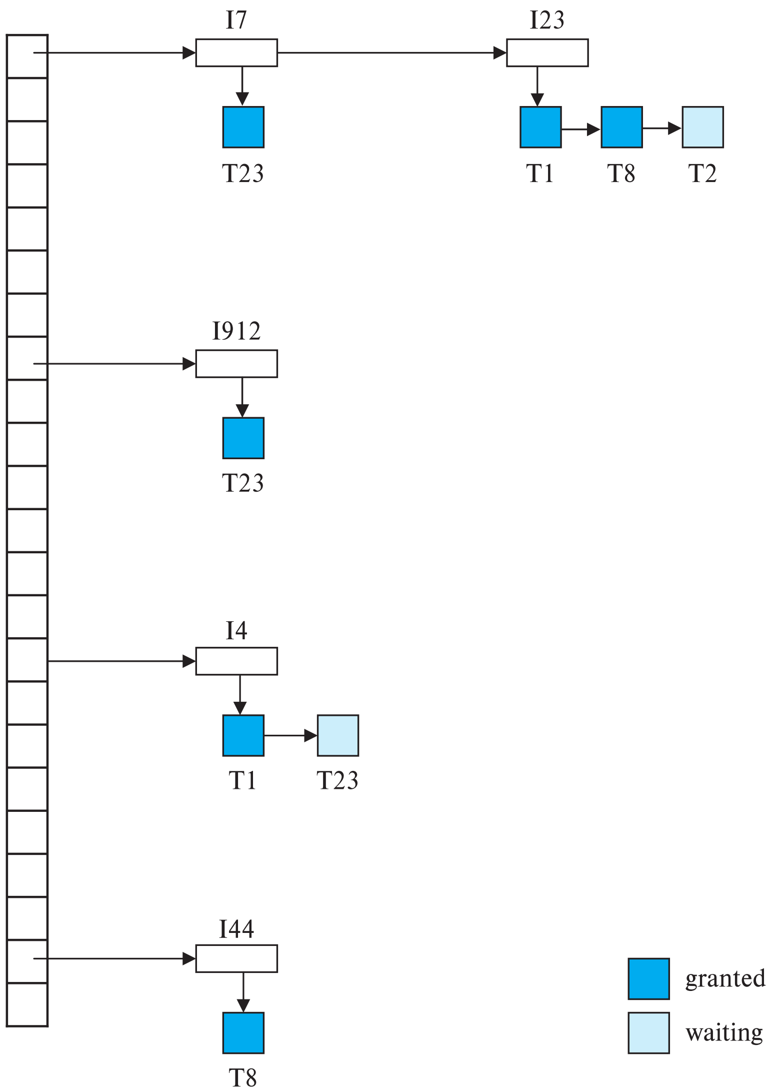
> **if** T_i has a **lock-X** on D **then**
>   write(D) ;
> **else begin**
>   **if** necessary, wait until no other trans. has any lock on D ;
>   **if** T_i has a **lock-S** on D **then**
>   **upgrade** lock on D to **lock-X** ;
>   **else**
>   grant T_i a **lock-X** on D ;
>   write(D) ;
>   **end**;

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

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

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

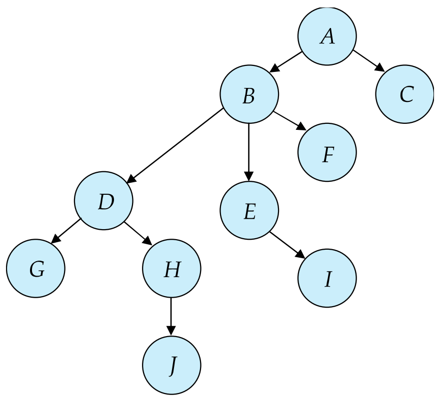
Graph-Based Protocols

- Graph-based protocols are locking protocols ensuring conflict serializability, but not two phase.
  - Requiring additional information e.g. a prior knowledge about the order in which data items are accessed

- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ 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* (or *tree-based locking protocol*) is a simple kind of graph-based protocols.

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



T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

A serializable schedule under tree protocol based on the left

Graph-Based Protocols (Cont.)

- The *tree protocol* ensures the conflict serializability; schedules not possible under two-phase locking are possible under the tree protocol, and vice versa.
- **Pros**
  - 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 due to deadlock handling
- **Cons**
  - Protocol does not guarantee cascade freedom (due to early unlocks)
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access (i.e., ancestors);
    - e.g. a transaction accessing A and J need lock B, D and H in the previous figure .
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency

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)	

4 conditions for deadlock

Mutual Exclusion
Hold & wait
Circular wait
Non-preemption

Deadlock Handling

- Deadlock prevention/avoidance protocols ensure that the system will never enter into a deadlock state.
  - Deadlock Prevention - *Statically & structurally*
  - Deadlock Avoidance – *Dynamically*
    - More concurrency and resource utilization

- ***Deadlock prevention*** strategies:
  - [**Pre-declaration**] Require that each transaction locks all its data items before it begins execution
  - [**Resource ordering**] 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

Deadlock Handling (Cont.)

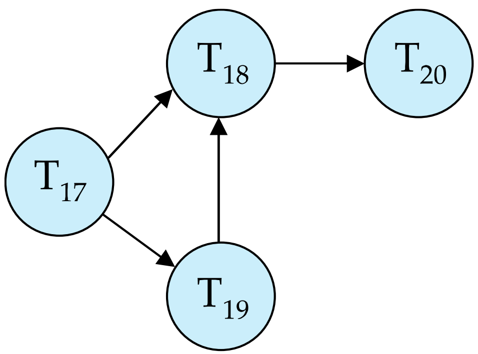
- **Deadlock avoidance** strategies:
- [**wait-die** scheme] — non-preemptive
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- [**wound-wait**] scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it.
  - Younger transactions may wait for older ones.
  - (may be) Fewer rollbacks than *wait-die* scheme.
- In both schemes, the rolled back transactions are restarted with its original timestamp.
  - Ensures that older transactions have precedence over newer ones, and starvation is thus avoided.

Deadlock Handling (Cont.)

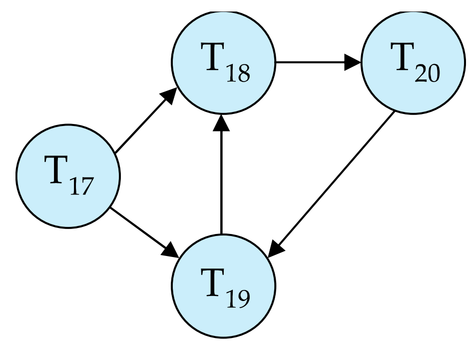
- **Timeout-Based Schemes**:
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Ensures that deadlocks get resolved by timeout if they occur
  - Simple to implement
  - But may roll back transaction unnecessarily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

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

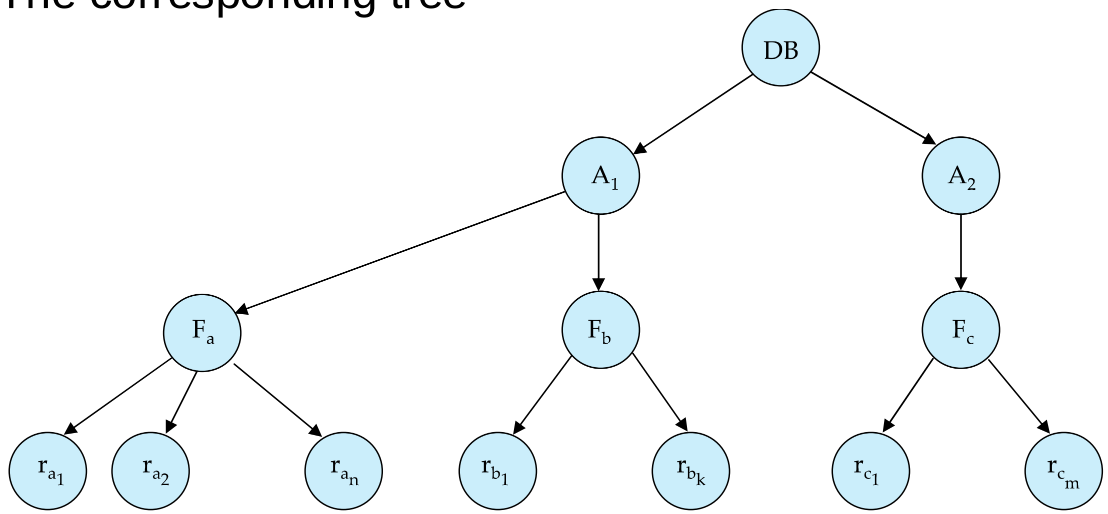
- When deadlock is detected :
  - Some transaction will have to rolled back (made a **victim**) to break deadlock cycle.
    - Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - **Total rollback**: Abort the transaction and 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
  - One solution: oldest transaction in the deadlock set is never chosen as victim

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*
- The corresponding tree



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
  - Timestamp could be based on a logical counter or system clock
  - Newer transactions have timestamps strictly greater than earlier ones
- Timestamp-based protocols manage concurrent execution such that
  time-stamp order = serializability order
- Several alternative protocols based on timestamps

Timestamp-Ordering Protocol

The (basic) **timestamp ordering (BTO) 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(T_i) $\geq$ **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 BTO

- Is this schedule valid under BTO?

  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

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($A + B$)

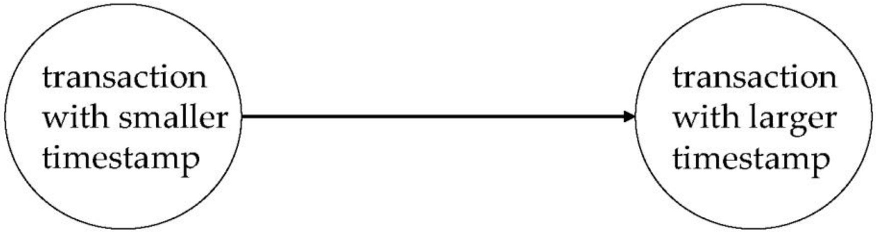
- How about this one ?

  Assume R-TS(Q)=W-TS(Q)=0

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

Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees **conflict 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 all writes are 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 (only) recoverability

Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances.
- Suppose 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} → T_i must be rolled back in BTO (basic timestamp ordering)
  - But rather than rolling back T_i, 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.
    - *E.g. The bottom schedule in Slide#35*
      (Note: This is a part of the schedule in Slide#23 of Ch17.)

Validation-Based Protocol

- Idea: “Can we use commit time as serialization order?”
- To do so:
  - Keep track of data items read/written by transaction
  - Postpone writes to end of transaction
  - **Validation** performed at commit time, detect any out-of-serialization order reads/writes
- Also called as **optimistic concurrency control** since transaction executes fully in the hope that all will go well during validation
  - In cases where most transactions are read-only, the rate of conflicts among transactions may be low

Validation-Based Protocol

- Execution of transaction T_i is done in (two or) 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. (Read-only transactions omit this phase.)
- We assume for simplicity that
  - The validation and write phases occur together, atomically and serially
  - Only one transaction executes validation/write at a time.
- The three phases of concurrently executing transactions can be interleaved, but each transaction must go through the three phases in that order.

- *Typically implemented with timestamps and multiversions*

Multiversion Concurrency Control

Multiversion Schemes

- Key ideas:
  - Each successful **write** results in the creation of a new version of the data item written.
  - Use timestamps to label versions.
  - When a **read**(*Q*) operation is issued, select an appropriate version of *Q* based on the timestamp of the transaction issuing the read request, and return the value of the selected version.
- **read**s never have to wait as an appropriate version is returned immediately.

- Multiversion schemes keep old versions of data item to increase concurrency. Several variants:
  - **Multiversion Timestamp Ordering**
  - **Multiversion Two-Phase Locking**
  - **Snapshot isolation**

Snapshot Isolation

- Motivation
  - Decision support queries that read large amounts of data have concurrency conflicts with OLTP transactions that update a few rows
  - Poor performance results

☞ Give snapshot of database state to every transaction
(although not fully serializable)

- Reads are performed on snapshot (similar to ***Read Committed***)
- Writes are performed locally
  - In validation phase, check the only *write sets* of other concurrent transactions; If conflicts, the first committer (or first writer) wins. The losers are aborted.
- Problem: variety of anomalies such as *write skews* can result
  - Solution: *Serializable Snapshot Isolation* (SSI)

Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - Takes snapshot of committed data at start
  - Always reads/modifies data in its own snapshot
  - Updates of concurrent transactions are not visible to T1
  - Writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		Start W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Snapshot Read

- Concurrent updates invisible to snapshot read

$X_0 = 100,\ Y_0 = 0$

$r_1(X_0, 100)$	
$r_1(Y_0, 0)$	
	$r_2(Y_0, 0)$
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(Y_1, 50)$	
$r_1(X_0, 100)$ (update by T_2 not seen)	
$r_1(Y_1, 50)$ (can see its own updates)	
	$r_2(Y_0, 0)$ (update by T_1 not seen)

$X_2 = 50,\ Y_1 = 50$

Snapshot Write: First Committer Wins

$X_0 = 100$

$r_1(X_0, 100)$	
	$r_2(X_0, 100)$
	$w_2(X_2, 50)$
$w_1(X_1, 150)$	
$commit_1$	
	$commit_2$ (Serialization Error T_2 is rolled back)

$X_1 = 150$

Snapshot Write: First Updater Wins

- (A variant of "First Committer Wins") "**First-updater-wins**" uses locking mechanism to *only* updates
  - A transaction t_i requests write locks when attempting a data item
  - If no other concurrent transactions hold the (write) lock, do the following steps:
    - If the target data is updated by any concurrent transactions, then t_i aborts.
    - Otherwise, t_i proceeds its execution (and possibly commits).
  - If some other concurrent t_j transaction already holds the (write) lock, t_i cannot proceed and do the followings:
    - If t_j aborts, the the lock is released, and t_i can obtain the lock and do the followings:
      - If the target data is updated by any concurrent transactions, then t_i aborts.
      - Otherwise, t_i proceeds its execution.
    - If t_j commits, then t_i must abort.

      (e.g. T_1 must abort after t_2's commit n the previous slide.)
  - Locks are released when the transaction commits or aborts.
- *Adopted by most commercial DBMS's*

Benefits of SI

- Reads are *never* blocked, and also don't block other transactions' activities
  → Longer analytic transactions do not interfere with shorter update transactions.
- Performance similar to **Read Committed**
- Avoids several anomalies
  - No dirty read, i.e. no read of uncommitted data
  - No lost update
    - i.e., update made by a transaction is overwritten by another transaction that did not see the update)
  - No non-repeatable read
    - i.e., if read is executed again, it will see the same value
- Problems with SI
  - SI does not always give serializable executions
    - Serializable: among two concurrent transactions, one sees the effects of the other
    - In SI: neither sees the effects of the other (e.g. *write skew*)

SI is NOT serializable !

- [Write Skew] Each of a pair of transactions has read a data item that is written by the other, but the set of data items written by two transactions do not have any data in common.
  - Example
    - Initially A = 3 and B = 17
    - Serial execution: A = ??, B = ??
- SI breaks serializability when transactions modify different items, each based on a previous state of the item the other modified
  - Not very common in practice
    - E.g., the TPC-C benchmark runs correctly under SI
  - When transactions conflict due to modifying different data, there is usually also a shared item they both modify, so SI will abort one of them
  - But problems do occur. Application developers should be careful about write skews

T_i	T_j
read(A)	
read(B)	
	read(A)
	read(B)
A=B	
	B=A
write(A)	
	write(B)

Notice: The 3 anomalies (dirty reads, non-repeatable reads and phantom reads) defined in ANSI SQL92 based on the classical serializability definition are not enough to address modern serialization anomalies in a comprehensive manner.

Serializable Snapshot Isolation

- **Serializable snapshot isolation (SSI)**: extension of snapshot isolation that ensures serializability
  - Snapshot isolation tracks write-write conflicts, but does not track read-write conflicts.

  → SSI tracks read-write conflicts in addition to write-write conflicts

- Implemented in many commercial and open-source DBMSs including Oracle, PostgreSQL and SQL Server
  - PostgreSQL (>= version 9.1) has supported SSI.
    - PSQL is the first DBMS adopting SSI.
    - Since PSQL adopts the *First-Updater-Wins* policy (reducing the rollbacks caused from *w-w* conflicts), writes are processed via locking although not 2PL; w-locks are held until the end of transaction, and thus deadlocks are possible.
  - Oracle supports only SI
    - Programmers have to handle possible serialization anomalies appropriately
      e.g. select ~ for update

End of Chapter 18