Database Systems
Lecture20 – Chapter 18: Concurrency Control

SUNG KYUN KWAN UNIVERSITY

Beomseok Nam (남범석)

bnam@skku.edu

Timestamp Based Concurrency Control

Timestamp-Based Protocols

- Each transaction T_i has a unique timestamp TS(T_i).
  - Timestamp is assigned when a transaction starts
  - Newer transactions have timestamps 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

  timestamp order = serializability order

Timestamp-Ordering Protocol

The **timestamp ordering (TSO) protocol**

- Maintains two timestamp values for each data *Q* :
  - **W-timestamp**(*Q*) is the largest timestamp of any transaction that executed **write**(*Q*) successfully.
  - **R-timestamp**(*Q*) is the largest timestamp 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),
   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) ≥ **W**-timestamp(Q),
   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),
   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),
   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?
  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, where initially R-TS(Q)=W-TS(Q)=0

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

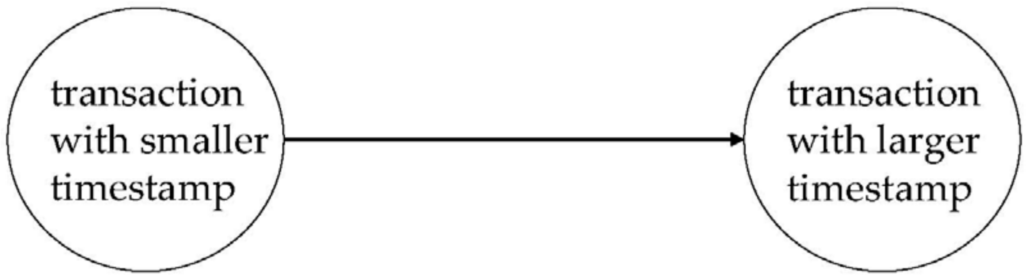
Another Example Under TSO

- 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)

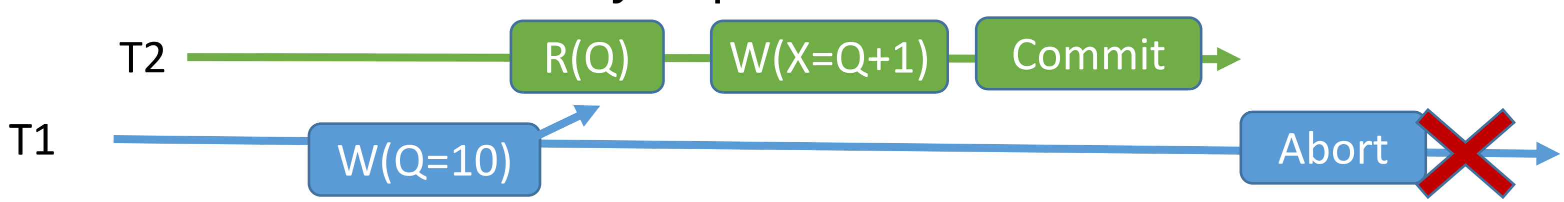
Correctness of Timestamp-Ordering Protocol

- 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**.
  - A schedule is recoverable if transactions commit only after all transactions that they depend on commit.



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
  - → Obsolete **write** operations are ignored
- 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.

Validation-Based Protocol

- Idea: can we use **commit time as serialization order**?
- To do so:
  - Postpone writes to end of transaction
  - Keep track of data items read/written by 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

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.

- We assume for simplicity that the validation and write phase occur together, atomically and serially
  - I.e., only one transaction executes validation/write at a time.
  - But, the three phases of concurrently executing transactions can be interleaved.

Validation-Based Protocol (Cont.)

- Each transaction T_i has 3 timestamps
  - **StartTS**(T_i) : the time when T_i started its execution
  - **ValidationTS**(T_i): the time when T_i entered its validation phase
  - **FinishTS**(T_i) : the time when T_i finished its write phase
- Validation tests use above timestamps and read/write sets to ensure that serializability order is determined by validation time
  - Thus, TS(T_i) = ValidationTS(T_i)
- Validation-based protocol has been found to give greater degree of concurrency than locking/TSO if probability of conflicts is low.

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:
  - **finishTS**(T_i) < **startTS**(T_j)
    - execution is not concurrent
  - **startTS**(T_j) < **finishTS**(T_i) < **validationTS**(T_j) **and** T_j does not read any data item written by T_i ,
    - there is no dependency

  then validation succeeds and T_j can be committed.

- Otherwise, validation fails and T_j is aborted.

Schedule Produced by Validation

- Example of schedule produced using validation

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

startTS(T_{26}) < **finishTS**(T_{25}) < **validationTS**(T_{26})