



## Module 7c: Atomicity

- Atomic Transactions
- Log-based Recovery
- Checkpoints
- Concurrent Transactions
- Serializability
- Locking Protocols



## Atomic Transactions

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
- Related to field of database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
  - Here we are concerned with changes to stable storage – disk
  - Transaction is series of read and write operations
  - Terminated by commit (transaction successful) or abort (transaction failed) operation
  - Aborted transaction must be rolled back to undo any changes it performed





## Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
  - Example: main memory, cache
- Nonvolatile storage – Information usually survives crashes
  - Example: disk and tape
- Stable storage – Information never lost
  - Not actually possible, so approximated via replication or RAID to devices with independent failure modes

Goal is to assure transaction atomicity where failures cause loss of information on volatile storage



## Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common is write-ahead logging
  - Log on stable storage, each log record describes single transaction write operation, including
    - ▶ Transaction name
    - ▶ Data item name
    - ▶ Old value
    - ▶ New value
  - $\langle T_i \text{ starts} \rangle$  written to log when transaction  $T_i$  starts
  - $\langle T_i \text{ commits} \rangle$  written when  $T_i$  commits
- Log entry must reach stable storage before operation on data occurs





## Log-Based Recovery Algorithm

- Using the log, system can handle any volatile memory errors
  - Undo( $T_i$ ) restores value of all data updated by  $T_i$
  - Redo( $T_i$ ) sets values of all data in transaction  $T_i$  to new values
- Undo( $T_i$ ) and redo( $T_i$ ) must be idempotent
  - Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
  - If log contains  $\langle T_i \text{ starts} \rangle$  without  $\langle T_i \text{ commits} \rangle$ , undo( $T_i$ )
  - If log contains  $\langle T_i \text{ starts} \rangle$  and  $\langle T_i \text{ commits} \rangle$ , redo( $T_i$ )



## Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
  1. Output all log records currently in volatile storage to stable storage
  2. Output all modified data from volatile to stable storage
  3. Output a log record  $\langle \text{checkpoint} \rangle$  to the log on stable storage
- Now recovery only includes  $T_i$ , such that  $T_i$  started executing before the most recent checkpoint, and all transactions after  $T_i$   
All other transactions already on stable storage





## Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
  - Inefficient, too restrictive
- Concurrency-control algorithms provide serializability



## Serializability

- Consider two data items A and B
- Consider Transactions  $T_0$  and  $T_1$
- Execute  $T_0, T_1$  atomically
- Execution sequence called schedule
- Atomically executed transaction order called serial schedule
- For N transactions, there are  $N!$  valid serial schedules





## Schedule 1: $T_0$ then $T_1$

$T_0$	$T_1$
read( $A$ )	
write( $A$ )	
read( $B$ )	
write( $B$ )	
	read( $A$ )
	write( $A$ )
	read( $B$ )
	write( $B$ )



## Nonserial Schedule

- Nonserial schedule allows overlapped execute
  - Resulting execution not necessarily incorrect
- Consider schedule  $S$ , operations  $O_i, O_j$ 
  - Conflict if access same data item, with at least one write
- If  $O_i, O_j$  consecutive and operations of different transactions &  $O_i$  and  $O_j$  don't conflict
  - Then  $S'$  with swapped order  $O_j O_i$  equivalent to  $S$
- If  $S$  can become  $S'$  via swapping nonconflicting operations
  - $S$  is conflict serializable





## Schedule 2: Concurrent Serializable Schedule

$T_0$	$T_1$
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)



## Locking Protocol

- Ensure serializability by associating lock with each data item
  - Follow locking protocol for access control
- Locks
  - Shared –  $T_i$  has shared-mode lock (S) on item Q,  $T_i$  can read Q but not write Q
  - Exclusive –  $T_i$  has exclusive-mode lock (X) on Q,  $T_i$  can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
  - Similar to readers-writers algorithm





## Two-phase Locking Protocol

- Generally ensures conflict serializability
- Each transaction issues lock and unlock requests in two phases
  - Growing – obtaining locks
  - Shrinking – releasing locks
- Does not prevent deadlock



## Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction  $T_i$  associated with timestamp  $TS(T_i)$  before  $T_i$  starts
  - $TS(T_i) < TS(T_j)$  if  $T_i$  entered system before  $T_j$
  - TS can be generated from system clock or as logical counter incremented at each entry of transaction
- Timestamps determine serializability order
  - If  $TS(T_i) < TS(T_j)$ , system must ensure produced schedule equivalent to serial schedule where  $T_i$  appears before  $T_j$





## Timestamp-based Protocol Implementation

- Data item Q gets two timestamps
  - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully
  - R-timestamp(Q) – largest timestamp of successful read(Q)
  - Updated whenever read(Q) or write(Q) executed
- Timestamp-ordering protocol assures any conflicting read and write executed in timestamp order
- Suppose  $T_i$  executes read(Q)
  - If  $TS(T_i) < W\text{-timestamp}(Q)$ ,  $T_i$  needs to read value of Q that was already overwritten
    - ▶ read operation rejected and  $T_i$  rolled back
  - If  $TS(T_i) = W\text{-timestamp}(Q)$ 
    - ▶ read executed, R-timestamp(Q) set to  $\max(R\text{-timestamp}(Q), TS(T_i))$



## Timestamp-ordering Protocol

- Suppose  $T_i$  executes write(Q)
  - If  $TS(T_i) < R\text{-timestamp}(Q)$ , value Q produced by  $T_i$  was needed previously and  $T_i$  assumed it would never be produced
    - ▶ Write operation rejected,  $T_i$  rolled back
  - If  $TS(T_i) < W\text{-timestamp}(Q)$ ,  $T_i$  attempting to write obsolete value of Q
    - ▶ Write operation rejected and  $T_i$  rolled back
  - Otherwise, write executed
- Any rolled back transaction  $T_i$  is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock





## Schedule Possible Under Timestamp Protocol

$T_2$	$T_3$
read( $B$ )	read( $B$ ) write( $B$ )
read( $A$ )	read( $A$ ) write( $A$ )

