

More on Transactions

Kathleen Durant PhD

Lesson 10

CS3200

Outline for the day

- Review of ACID and concurrency
 - Interleaving transaction operations
- Strict Two phase locking
- Deadlock prevention and detection
- Waits-for graph vs. Precedence graph
- Granular Locking
- Concurrency without locking
 - Optimistic Concurrency Control
 - Timestamp based concurrency control

Review: ACID Properties

- **Atomicity**: either the entire set of operations happens or none of it does
- **Consistency**: the set of operations taken together should move the system for one consistent state to another consistent state.
- **Isolation**: each system perceives the system as if no other transactions were running concurrently (even though odds are there are other active transactions)
- **Durability**: results of a completed transaction must be permanent - even IF the system crashes

Review: Transaction Example

Transfer \$50 from account A to
account B

```
Read(A);  
A -= 50;  
Write(A);  
Read(B);  
B += 50;  
Write(B);
```

Transaction




ACID

- **Atomicity** - shouldn't take money from A without giving it to B
- **Consistency** - money isn't lost or gained
- **Isolation** - other queries shouldn't see A or B change until completion
- **Durability** - the money does not go back to A

Transactions

Representation of a Transaction

Application	Transaction	Time
Read(A);	Read(A);	
A -= 50;		
Write(A);	Write(A);	
Read(B);	Read(B);	
B += 50;		
Write(B);	Write(B);	

Transaction Operations:
Read, Write (Locks added later)

Building a transaction schedule

- Execution of a transaction must guarantee the ACID properties
 1. Easiest solution: run one transaction at a time (Serial execution)
 2. Chosen solution: Interleave transaction operations **safely** (Concurrent execution)
- Motivation for concurrency
 - Increase the throughput of the system

Anomalies due to interleaved execution

Schedule U	
T1	T2
Read (A)	
Write (A)	
	Read (A)
	Write (A)
	Read (B)
	Write (B)
	Commit
Read (B)	
Write (B)	
???	

- Reading uncommitted data (**WR Conflicts**)
- T2 reads value A written by T1 before T1 completed its changes
 - If T1 later aborts, T2 worked with invalid data

Anomalies due to interleaved execution

Schedule U	
T1	T2
Read (A)	
	Read (A)
	Write (A)
	Commit
Read(A)	
Write(A)	
Commit	

- Unrepeatable Reads (**RW Conflicts**)
- T1 sees two different values of A, even though it did not change A between its two reads

Anomalies due to interleaved execution

Schedule U	
T1	T2
Write (A)	
	Write (A)
	Write (B)
	Commit
Write(B)	
Commit	

- Overwriting uncommitted data (**WW Conflicts**)
- T1's B and T2's A persist, which would not happen with any serial execution
 - Either T1 determines final outcome for both variables or T2 determines final outcome

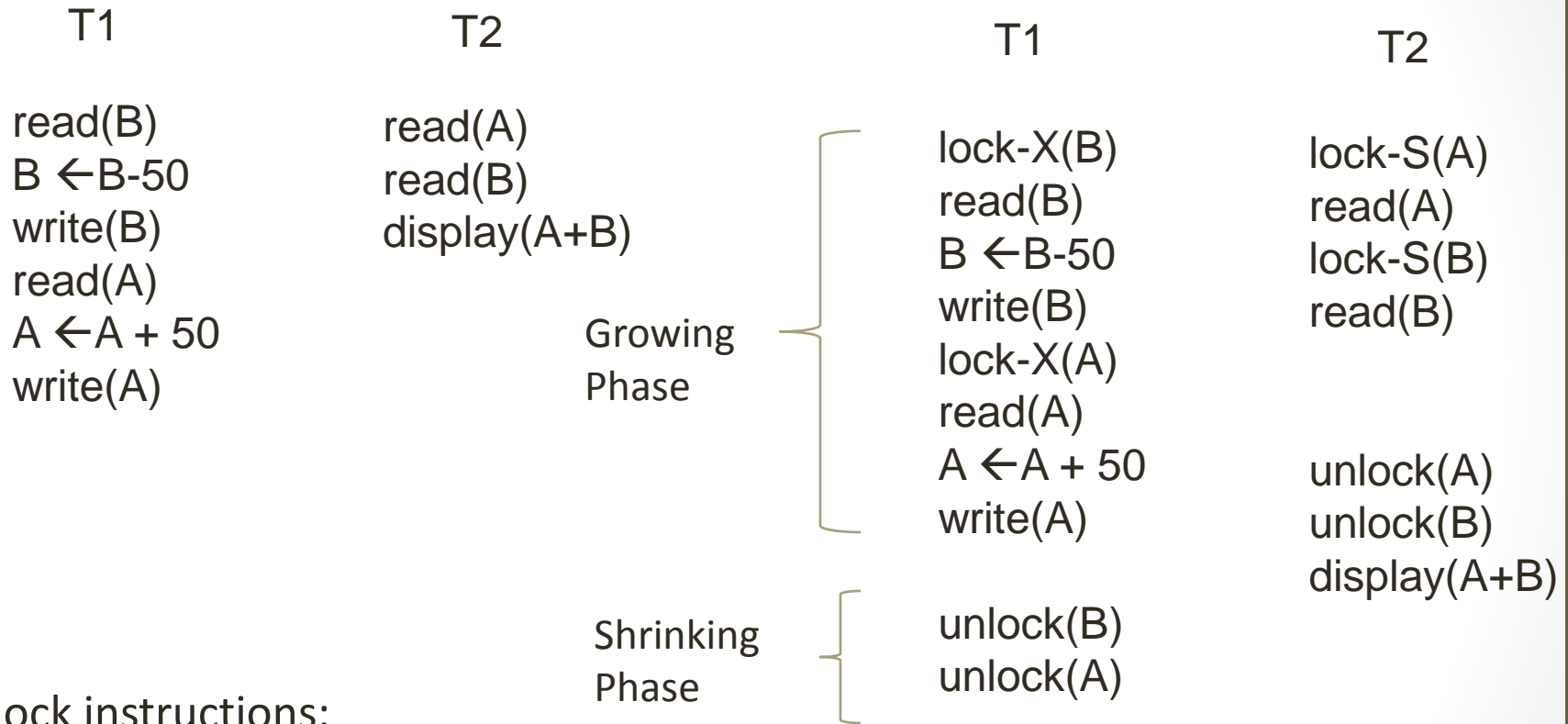
Addressing Anomalies

- Limit access to objects via locks

Strict Two phase locking (2PL)

- 2 types of locks
 - Share Lock and Exclusive Lock
 - Only 1 exclusive lock allowed on a database object
- Phase 1
 - Transaction asks for locks before access to a object
 - No release of locks in this phase
- Phase 2
 - Cannot asks for any more locks
 - Dismiss locks at the end of the transaction
- Forces transaction 2 to wait if transaction 1 is using an object that transaction 2 wants to use
- Locks managed/determined by the DBMS

Examples: Transactions with locks



Lock instructions:

- lock-S: shared lock request
- lock-X: exclusive lock request
- unlock: release previously held lock

Aborting a transaction

- If a transaction T_i is aborted
 - all its DB actions are undone
 - Transaction that have read an object last written by T_i also need to be aborted
 - **Cascading Abort**
- Avoid cascading deletes by releasing all locks at commit time
- Undo a transaction? : DBMS keeps a log
 - Records EVERY WRITE operation

The Log data

- Actions recorded in the log
 - Ti **writes** an object: log tracks old and the new value
 - Log file must go to disk before the changed data values hit the disk
 - Ti **commits/aborts** - a log record tracks this action
- Log records chained together by transaction id
 - Facilitates abortion of a transaction

Recovery algorithms: recovery from a crash

- 3 phases to ARIES recovery algorithms
- **Analysis:** scan the log **forward** (from the most recent checkpoint) to identify all transactions that were active and all dirty pages in the buffer pool
- **Redo:** Redoes all updates to dirty pages in the buffer pool
- **Undo:** Write operations of all transactions that were active during the crash are undone (works **backwards**)
 - Restore the database to values before the uncommitted transactions run

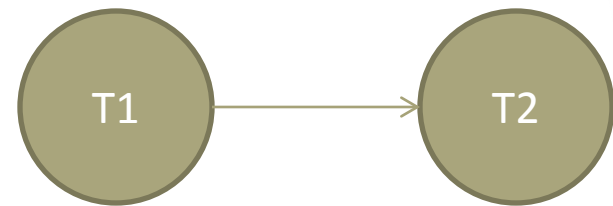
Representing scheduling dependencies

Representing conflicts among transactions

- Outcome of a schedule is dependent only on the conflicting operations
- Precedence graph
 - A node for each **committed transaction** in the schedule S
 - An arc from T_i to T_j exists if an action of T_i precedes (comes before in time) and conflicts with one of T_j 's actions
 - Precedence graphs allows us to determine if a particular transaction schedule can be serialized
 - *If you generate a cycle in the graph then the schedule is not serializable*

Example 1: Precedence graph

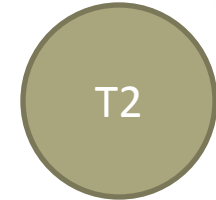
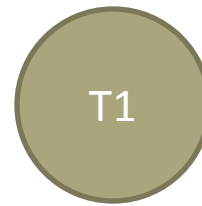
T1	T2
X(A)	
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit



Fill in the edges

Example 2: Precedence graph

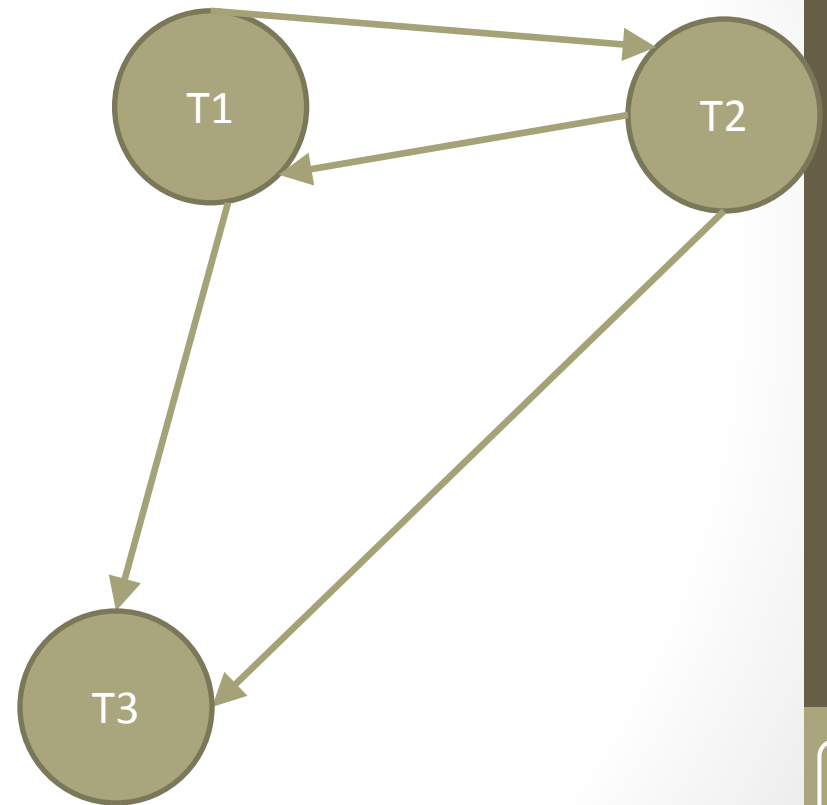
T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
	R(B)
	W(B)
	Commit
X(C)	
R(C)	
W(C)	
Commit	



Fill in the edges

Example 3: Precedence graph

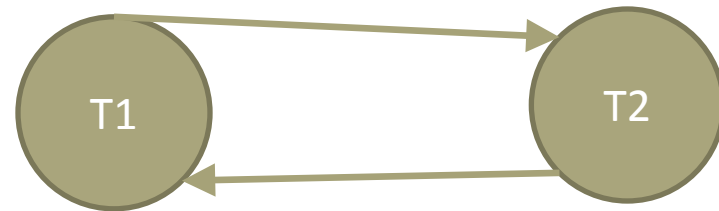
T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit



Fill in the edges

Example 4: Precedence graph

T1	T2
R(A)	
W(A)	
	Read(B)
	Write(B)
	Read(A)
	Write(A)
Read(B)	
Write(B)	



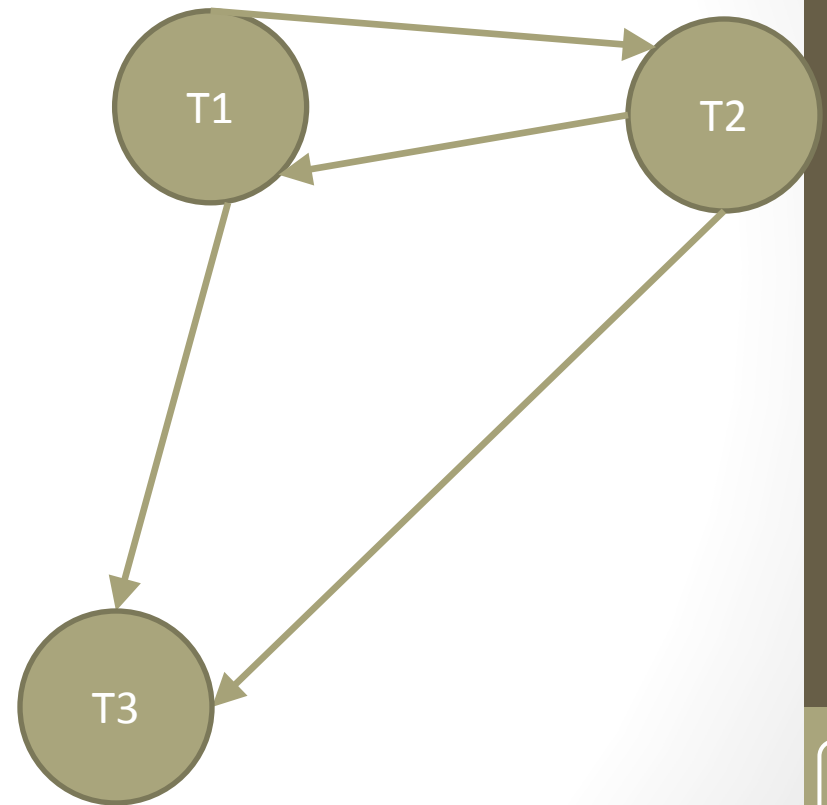
Fill in the edges

Strict 2PL Precedence Graph

- Schedule is **conflict serializable** if its precedence graph is acyclic
- Strict 2PL ensures that the precedence graph for any schedule it allows is acyclic
- Conflict serializable – equivalent to a serial schedule
- Strict Two-phase Locking (Strict 2PL) Protocol:
 - Each transaction must obtain the appropriate lock before accessing an object.
 - All locks are held by a transaction and are released when the transaction is completed

Example 3: Conflict serializable?

T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit



NO - has multiple cycles would need to abort transactions

Locking Variations

- Strict 2 Phase Locking - Strict (2PL)
 - Each transaction must obtain the appropriate lock before accessing an object.
 - All locks held by a transaction are released when the transaction is completed
- Both variations ensures an acyclic precedence graph
- 2 Phase Locking (2PL) Protocol
 - Each transaction must obtain a S(shared) lock on object before reading
 - Each transaction must obtain an X(exclusive) lock on an object before writing.
 - A transaction cannot request additional locks once it releases any locks

Specialty Subsystems

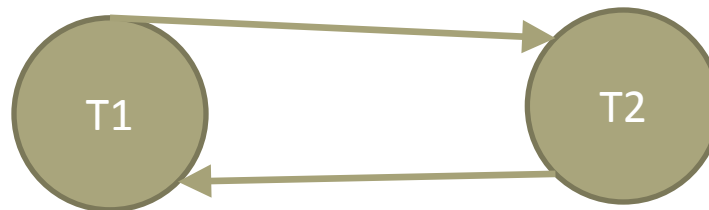
- Lock manager
 - Handles lock/unlock requests
- Transaction Manager
 - Subsystem of the DBMS responsible for controlling the execution of transactions
- Recovery Manager
 - Brings the system from a potential inconsistent state to a consistent state (After a crash)

Lock manager

- Lock table entry
 - Number of transactions currently holding a lock
 - Type of lock held (shared or exclusive)
 - Pointer to queue of lock requests
- Locking and unlocking have to be atomic processes
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other



- Two ways of dealing with deadlocks
 - Deadlock prevention
 - Deadlock detections

Deadlock Prevention

- Assign priorities to transactions based on timestamps
- Assume T_i wants a lock that T_j holds
 - 2 types of Policies
 - **Wait-Die:** If T_i has a higher priority than T_j T_i can wait for the resource otherwise T_i is aborted
 - Lower priority transactions can never wait for higher-priority transactions but higher-priority transactions can wait
 - **Wound-wait:** If T_i has higher priority than T_j ; T_j is aborted otherwise T_i is aborted
 - Higher priority transactions never wait for lower-priority transactions they wound the lower priority transaction and the lower priority transaction is forced to restart and wait until the high priority transaction finishes
- If a transactions re-starts, make sure it keeps its original timestamp

Example: deadlock policies

Wait – die (WAIT executed)

Ti	Tj
09:09:09	09:15:12
	X(B)
X(B)	
Waits for B	
	Commit
Granted B	

Wound-wait (WOUND executed)

Ti	Tj
09:09:09	09:15:12
	X(B)
X(B)	
	Abort Tj
Granted B	Abort Restart LOOP
	Restart Tj

Wait – die (DIE executed)

Tk	Tj
09:20:09	09:15:12
	X(B)
X(B)	
Abort Tk	
Restart Tk	
	Commit

Wound– wait (WAIT executed)

Tk	Tj
09:20:09	09:15:12
	X(B)
X(B)	
Waits for B	
	Commit
Granted B	

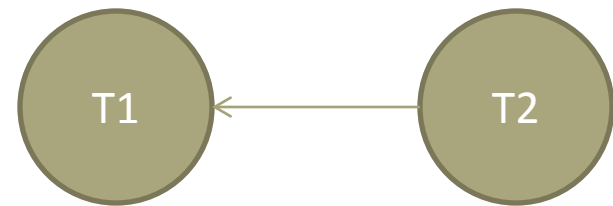
Abort
Restart
LOOP

Deadlock Detection

- Create a waits-for graph
 - Nodes in the graph are **active** transactions
 - There is an edge from T2 to T1 if T2 is waiting for T1 to release a lock
 - T1 read-locks X then T2 tries to write-lock it (RW)
 - T1 write-locks X then T2 tries to read-lock it (WR)
 - T1 write-locks X then T2 tries to write-lock (WW)
- Periodically check for cycles in the waits-for graph
 - Approach reasonable since deadlock prevalence rate is low

Example 1: Waits for graph

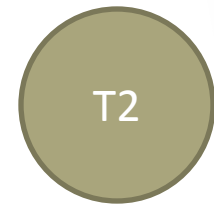
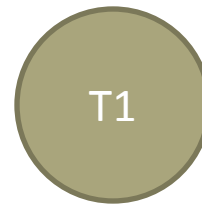
T1	T2
X(A)	
R(A)	
W(A)	
X(B)	X(A)
R(B)	
W(B)	
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)



Fill in the edges

Example 2: Waits for graph

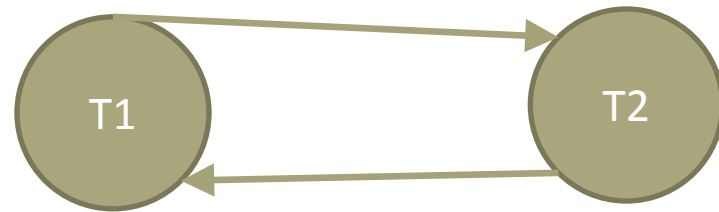
T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	X(B)
	R(B)
	W(B)
X(C)	
R(C)	
W(C)	



Fill in the edges

Example 4: Waits for graph

T1	T2
Read(A)	
Write(A)	
	Read(B)
	Write(B)
	Read(A)
	Write(A)
Read(B)	
Write(B)	



Fill in the edges

Precedence graph vs. Waits-for graph

Precedence graph

- Tracks who is allocating resources other transactions need
- Each **Committed transaction** is a vertex
- Arcs from **T1 to T2** if
 - T1 reads X before T2 writes X
 - T1 writes X before T2 reads X
 - T1 writes X before T2 writes X

Waits for graph

- Tracks who is waiting for an allocated resource
- Each **Active Transaction** is a vertex
- Arcs from **T2 to T1** if
 - T1 read-locks X then T2 tries to write-lock it
 - T1 write-locks X then T2 tries to read-lock it
 - T1 write-locks X then T2 tries to write-lock it

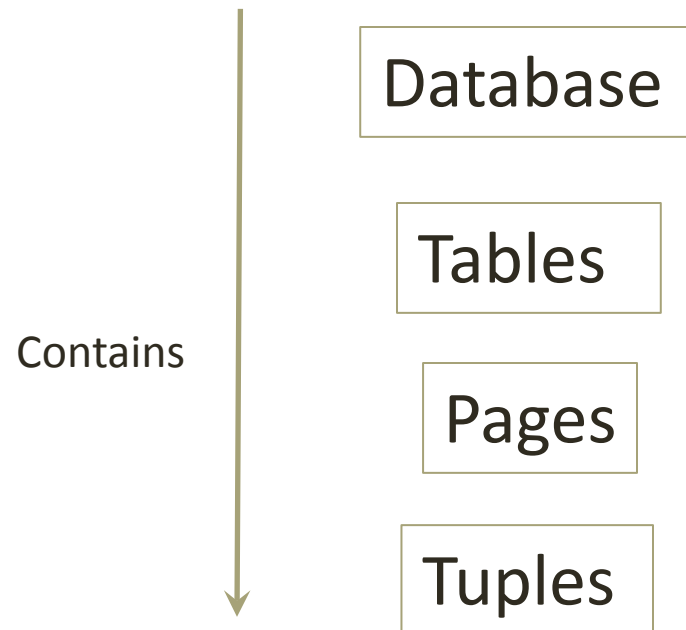
INVERTED RELATIONSHIP – SWITCH DIRECTION OF ARROWS

If both T1 and T2 Commit

then there is an edge in the opposite direction in the precedence graph for each edge in the waits for graph

Multiple-Granularity Locks

- Hard to decide what granularity to lock objects on (tuples vs. pages vs. tables)
- Should be able to lock at the appropriate level
- Data containers are hierarchical



Solution: New Lock Modes: Intent

- Allows transactions to lock at each level but with a special protocol using new 'intentions' locks.
- Before viewing an item, transaction must set intention locks on all its ancestors (higher level containers)
- For unlock, go from specific to general
- SIX model Like S & IX at the same time

	--	IS	IX	S	X
--	X	X	X	X	X
IS	X	X	X	X	
IX	X	X	X		
S	X	X		X	
X	X				

Intent Lock Algorithm

- Each transaction starts from the root of the hierarchy
- To get S or IS lock on a node, must hold IS or IX on parent node
- To get X or IX or SIX on a node, must hold IX or SIX on parent node
- Must release locks in bottom-up order
- Equivalent to directly setting locks at the leaf levels

Alternatives to Locks

- Timestamp Concurrency Control
- Optimistic Concurrency Control

Concurrency with Timestamps

- Each resource X has two timestamps
 - $RTS(X)$, the largest timestamp of any transaction that has read X
 - $WTS(X)$, the largest timestamp of any transaction that has written X
- Each transaction has a TS when it begins
 - If action a_i of transaction T_i conflicts with action a_j of transaction T_j . And $TS(T_i) < TS(T_j)$, then a_i must occur before a_j . Otherwise, restart T_j

Transaction T_i wants to read

- If $TS(T_i) > WTS(O)$
 - Allow T to read O .
 - Reset $RTS(O)$ to $\max(RTS(O), TS(T_i))$
- If $TS(T_i) < WTS(O)$ this violates timestamp order of T_i
 - Solution abort and restart T_i with a new, larger TS

Transaction T_i wants to write O

- If $TS(T_i) > RTS(O)$
 - Solution abort and restart T_i with a new, larger TS
- If $TS(T_i) < WTS(O)$ this violates timestamp order of T_i
 - Solution abort and restart T_i with a new, larger TS
- Else, allow T to write object O

Optimistic Transaction Model

- Transactions have three phases
 - READ – transaction reads from the DB and makes changes to a local copy of the data
 - VALIDATE – checks for conflicts
 - WRITE – makes local changes to the data public.

Validation phase

- Test conditions that are sufficient to ensure no conflict occurred
- Each transaction is assigned a numeric id (timestamp)
- Transaction ids assigned at the end of the READ phase. Just before validation begins
- ReadSet(T_i) – set of object read by transaction T_i
- WriteSet(T_i) – set of objects modified by T_i

Validation: Test 1

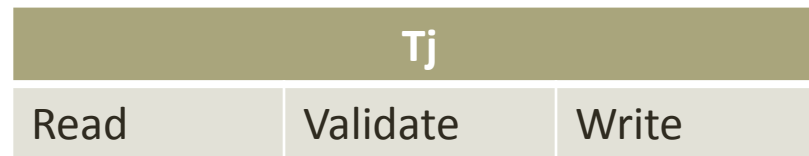
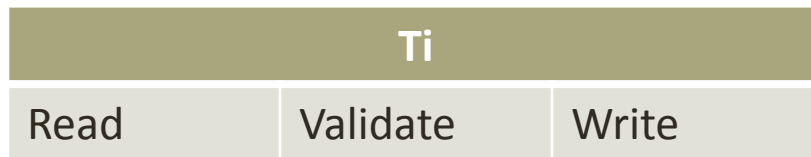
- For all i and j such that $T_i < T_j$ check that T_i completes before T_j begin

T_i		
Read	Validate	Write

T_j		
Read	Validate	Write

Validation: Test 2

- For all i and j such that $T_i < T_j$ check that
 - T_i completes before T_j begins its Write phase
 - $\text{WriteSet}(i) \cap \text{Readset}(T_j)$ is empty



Validation: Test 3

- For all i and j such that $T_i < T_j$ check that
 - T_i completes Read phase before T_j does
 - $\text{WriteSet}(T_i) \cap \text{Readset}(T_j)$ is empty
 - $\text{WriteSet}(T_i) \cap \text{WriteSet}(T_j)$ is empty

T_i		
Read	Validate	Write

T_j		
Read	Validate	Write

Optimistic Concurrency Control

- Locking prevents conflicts on resources
- Disadvantages
 - Lock management overhead
 - Deadlock detection/prevention
 - Lock contention for heavily used objects
- If conflicts are rare – then just resolve conflicts before a transaction commits

Summary

- There are several lock-based concurrency control schemes
 - Strict 2PL, 2PL
 - Conflicts between transactions can be detected in the precedence graph
- Lock manager keeps track of locks issued. Deadlocks can be prevented or detected.
 - Prevention via waits-for graph.
- Optimistic concurrency control aims to minimize the cost of CC
 - Best when reads common and writes are rare