MC 302– DBMS: Concurrency Control

Goonjan Jain

Dept. of Applied Mathematics

Delhi Technological University

Outline

- Serializability
- Locking –
  - 2PL
  - Variations
- Deadlocks

Formal Properties of Schedules

- Levels of serializability
  - Conflict serializability – all DBMSs support this
  - View serializability – harder but allows more concurrency

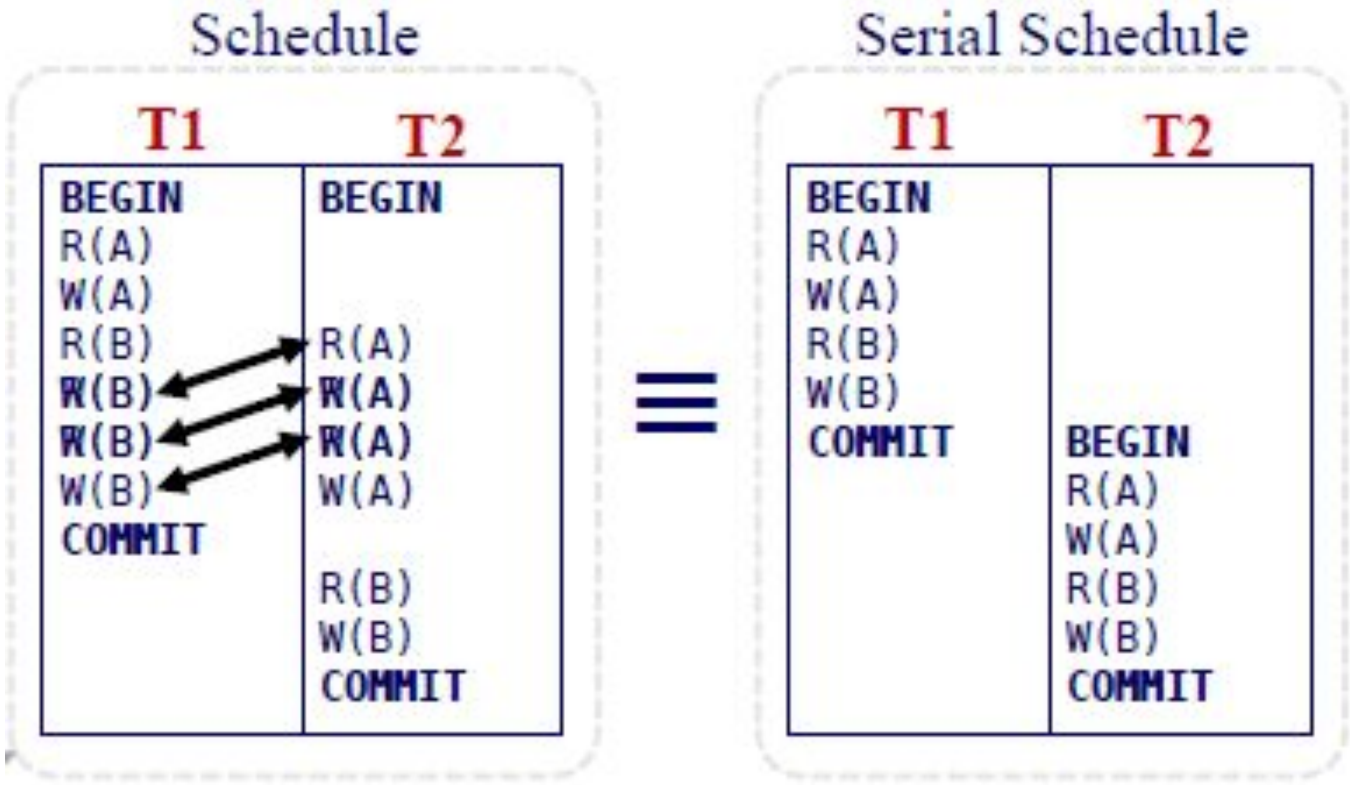
- Conflicting operations- Two operations conflict if:
  - They are by different transactions,
  - They are on the same object and at least one of them is a write.

Conflict Serializable Schedules

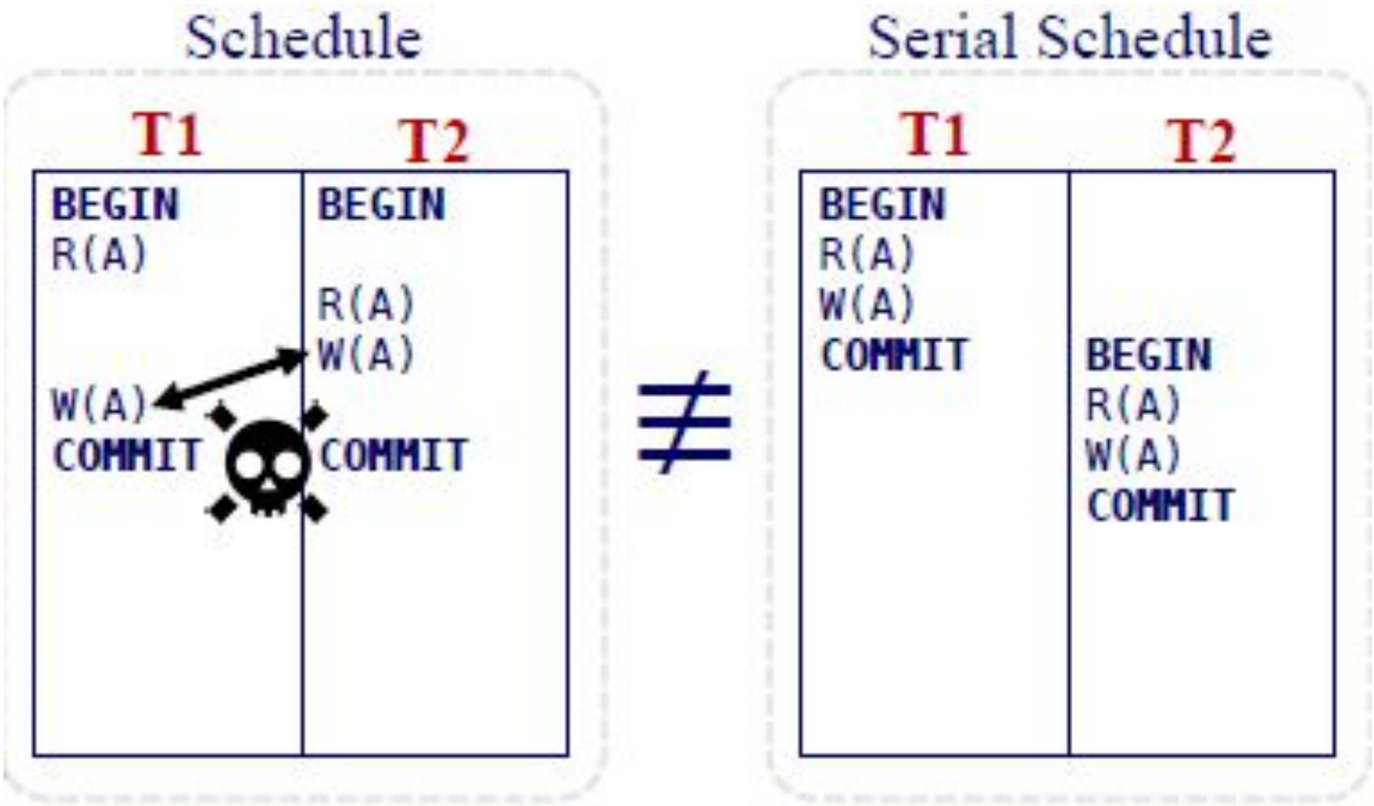
- Two schedules are ***conflict equivalent*** iff:
  - They involve the same actions of the same transactions, and
  - Every pair of conflicting actions is ordered the same way.

- Schedule S is ***conflict serializable*** if:
  - S is conflict equivalent to some serial schedule.
  - Able to transform S into a serial schedule by swapping consecutive nonconflicting operations of different transactions.

Conflict Serializability Intuition



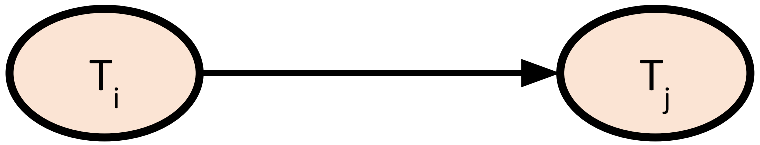
Conflict Serializability Intuition



Serializability

- **Q:** Are there any faster algorithms to figure this out other than transposing operations?
- A: Dependency Graphs

- Dependency Graphs:
  - One node per transaction.
  - Edge from T_i to T_j if:
    - An operation O_i of T_i conflicts with an operation O_j of T_j and
    - O_i appears earlier in the schedule than O_j.
  - Also known as a "precedence graph"



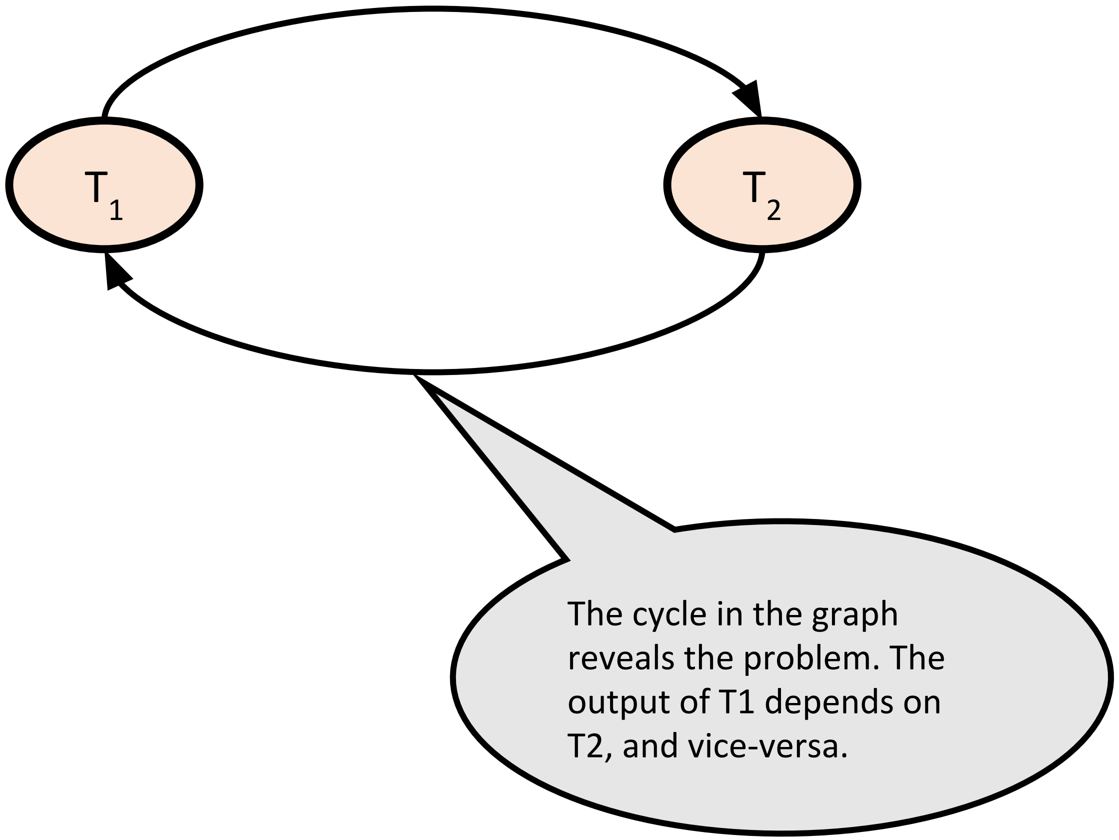
- **Theorem:** A schedule is *conflict serializable* if and only if its dependency graph is acyclic.

Dependency Graphs – Example 1

Schedule

T1	T2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph

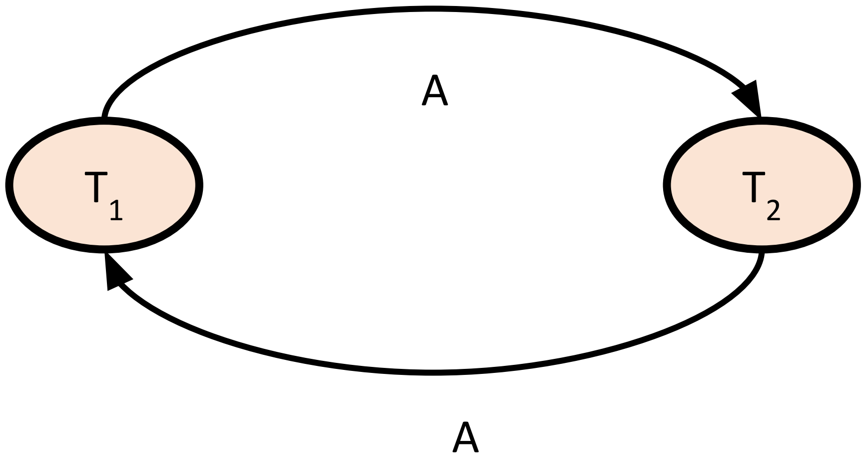


Example 2 – Lost update

Schedule

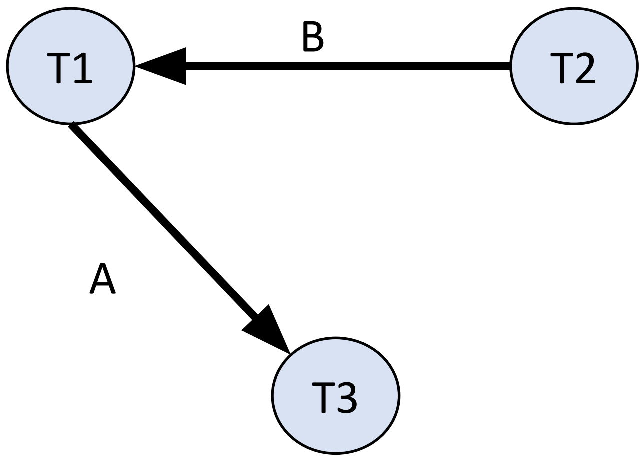
T1	T2
BEGIN R(A) A = A-1 W(A) COMMIT	BEGIN R(A) A = A -1 W(A) COMMIT

Dependency Graph



Example 3

T1	T2	T3
BEGIN		
R(A)		
W(A)		BEGIN
		R(A)
		W(A)
	BEGIN	COMMIT
	R(B)	
	W(B)	
R(B)	COMMIT	
W(B)		
COMMIT		



Is this equivalent to a serial execution?

- **A:** Yes (T2, T1, T3)

– Notice that T3 should go after T2, although it starts before it!

- Need an algorithm for generating serial schedule from an acyclic dependency graph.

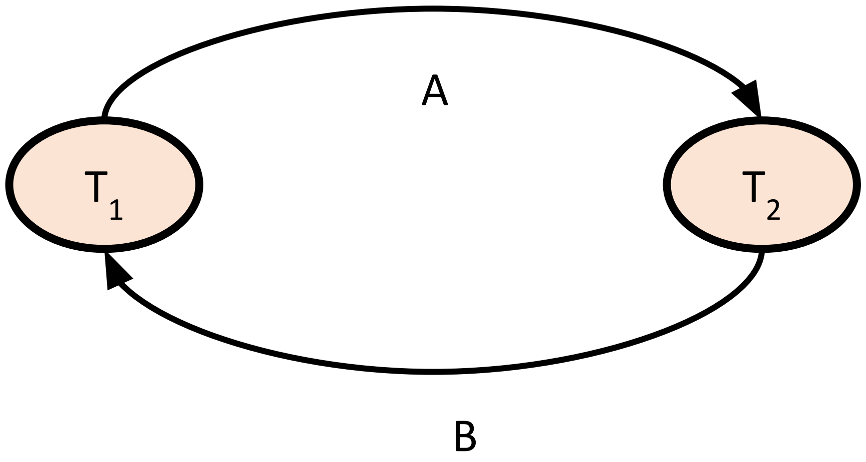
– ***Topological Sorting***

Example 4 – Inconsistent Analysis

Schedule

T1	T2
BEGIN	BEGIN
R(A)	
A = A-1	
W(A)	
	R(A)
	SUM = A
	R(B)
	SUM += B
	ECHO(SUM)
R(B)	COMMIT
B = B+1	
W(B)	
COMMIT	

Dependency Graph

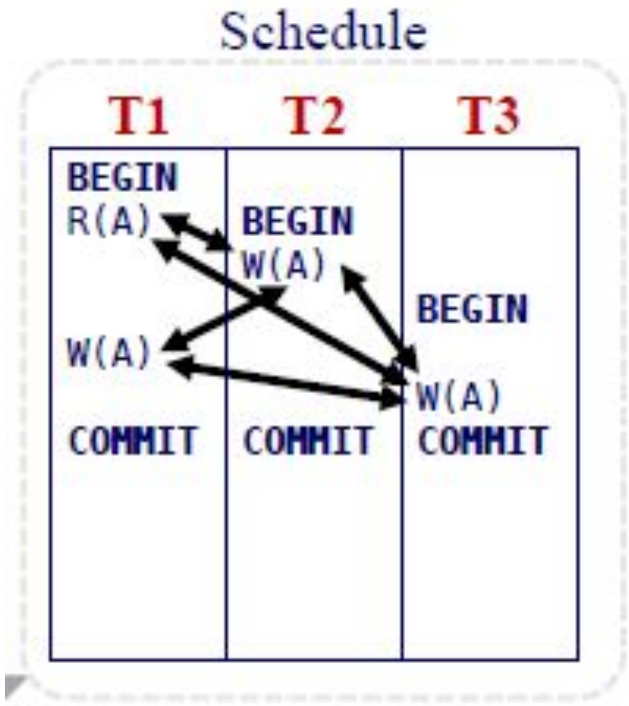


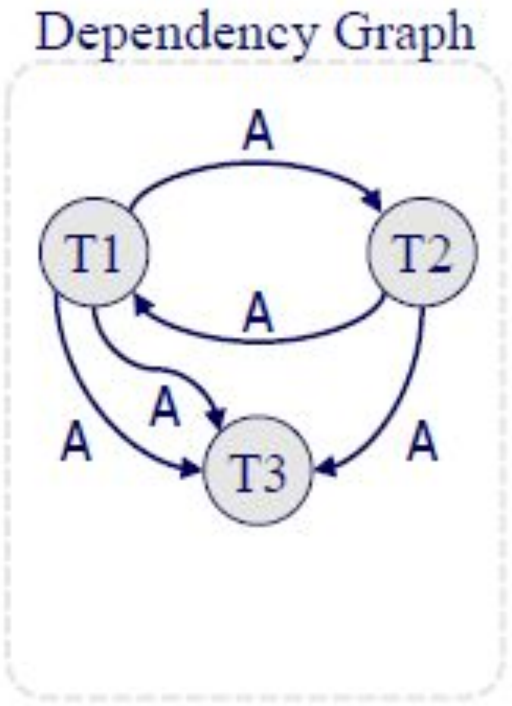
Is it possible to create a schedule similar to this that is "correct" but still not conflict serializable?

View Serializability

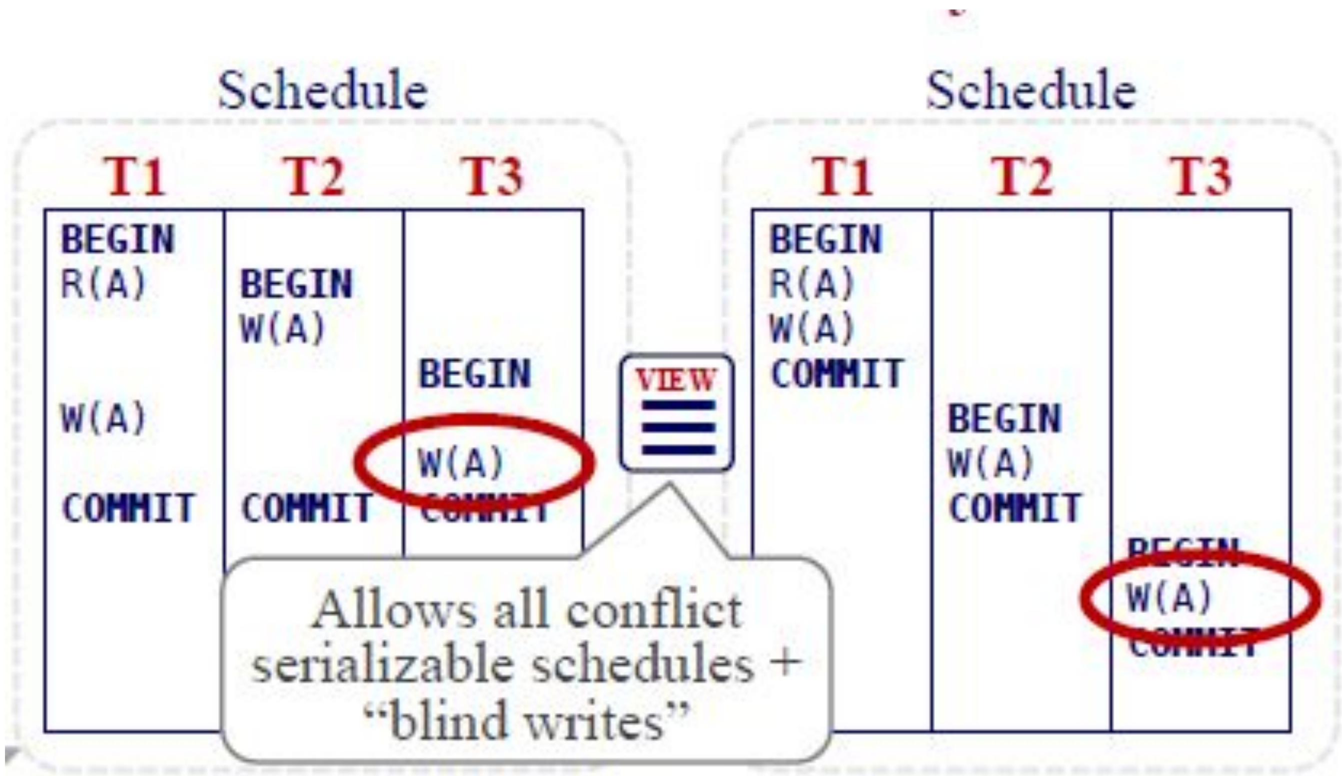
- Alternative (weaker) notion of serializability.
- Schedules S1 and S2 are ***view equivalent*** if:
  - If T1 reads initial value of A in S1, then T1 also reads initial value of A in S2.
  - If T1 reads value of A written by T2 in S1, then T1 also reads value of A written by T2 in S2.
  - If T1 writes final value of A in S1, then T1 also writes final value of A in S2.

View Serializability



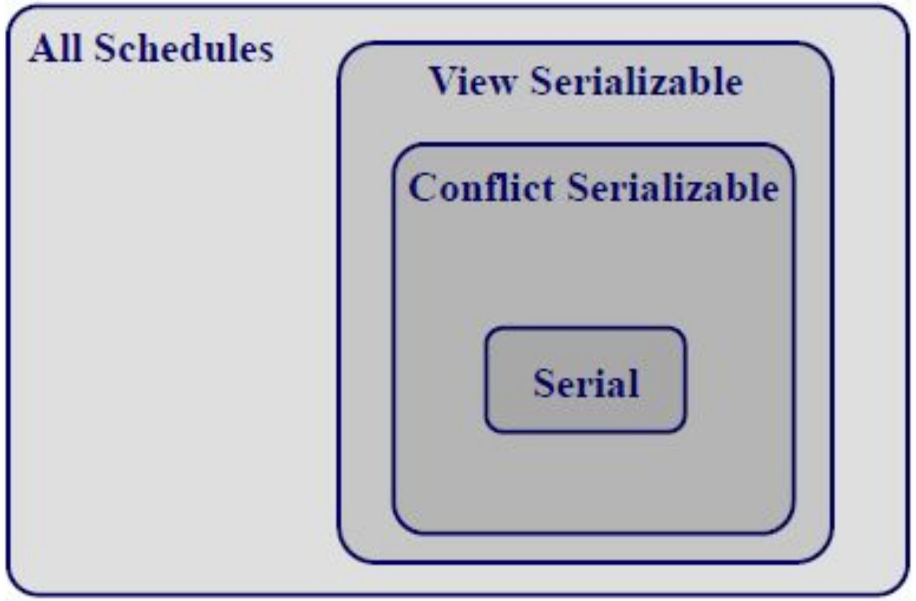


View Serializability



Serializability

- **View Serializability** allows (slightly) more schedules than **Conflict Serializability** does.
  - But is difficult to enforce efficiently.
- In practice, **Conflict Serializability** is what gets used, because it can be enforced efficiently.



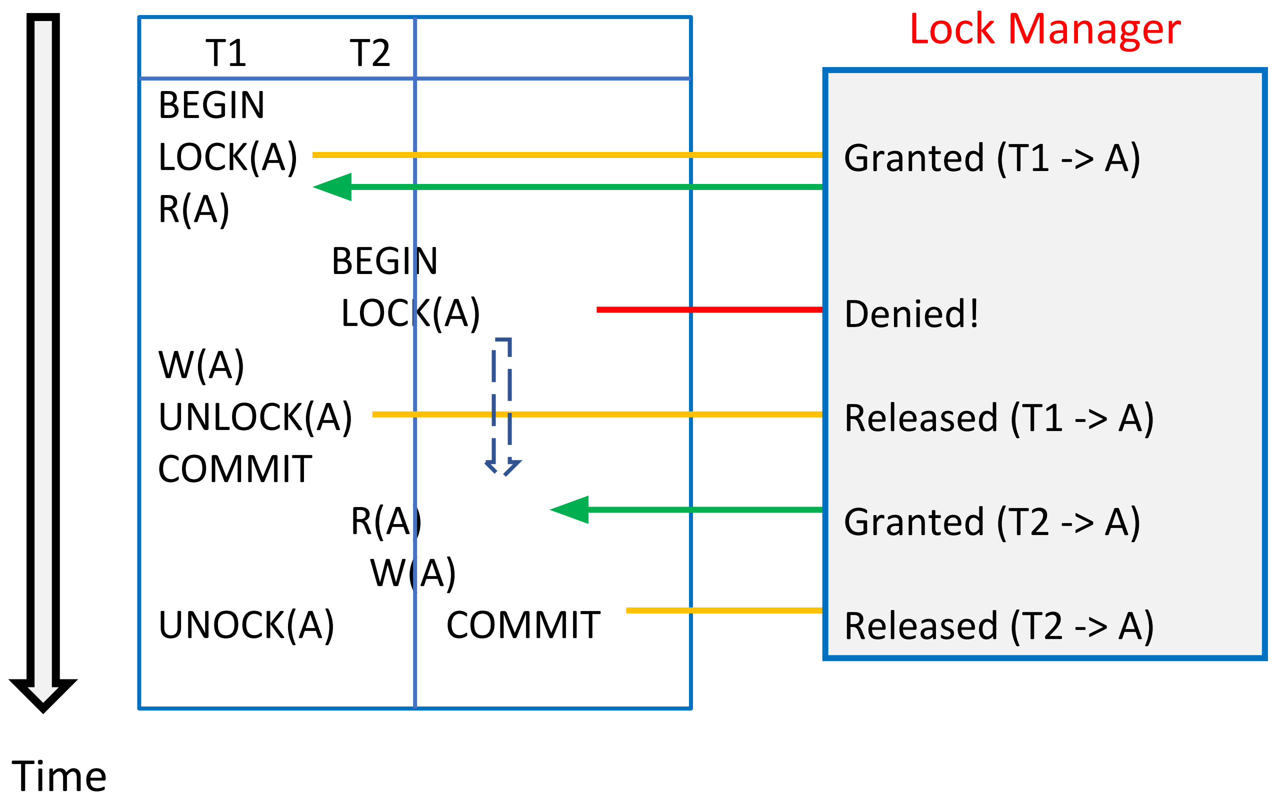
Locking Based Concurrency Control

- Lost update problem – without locks

T1	T2	
BEGIN		
R(A)		
	BEGIN	
R(A)		
W(A)		
COMMIT		
	W(A)	
COMMIT		

Executing with locks

- With locks – lock manager grants/denies lock requests



Executing with locks

- **Q:** If a transaction only needs to read ‘A’, should it still get a lock?
- **A:** Yes, but you can get a shared lock.

Lock Types

- Basic Lock Types:
  - S-Lock: Shared Locks (Reads)
  - X-Lock: Exclusive Locks (writes)

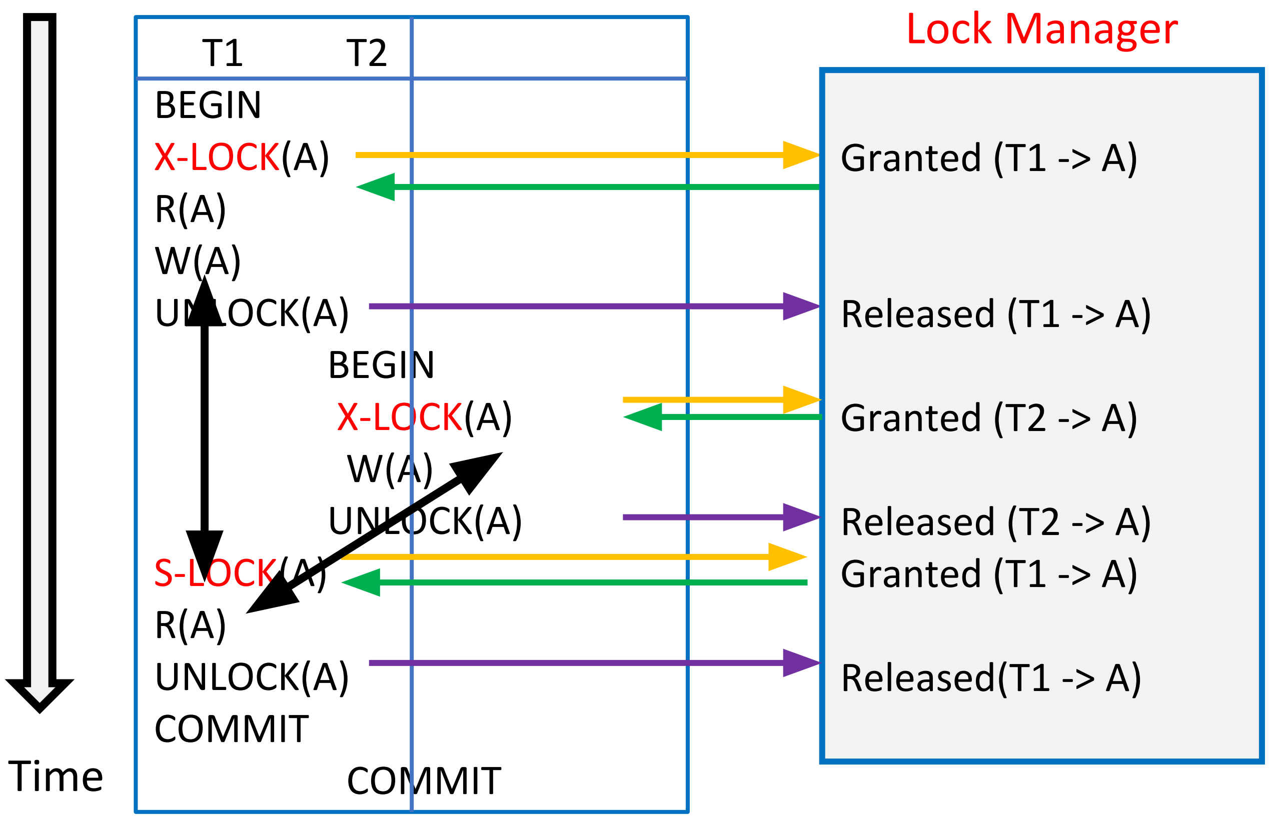
Compatibility Matrix

T2 wants / T1 has	Shared	Exclusive
Shared	Y	N
Exclusive	N	N

Executing with locks

- Transactions request locks (or upgrades)
  - Lock manager grants or blocks requests
  - Transactions release locks
  - Lock manager updates lock-table
- *But this is not enough…*

- *Inconsistent Analysis*



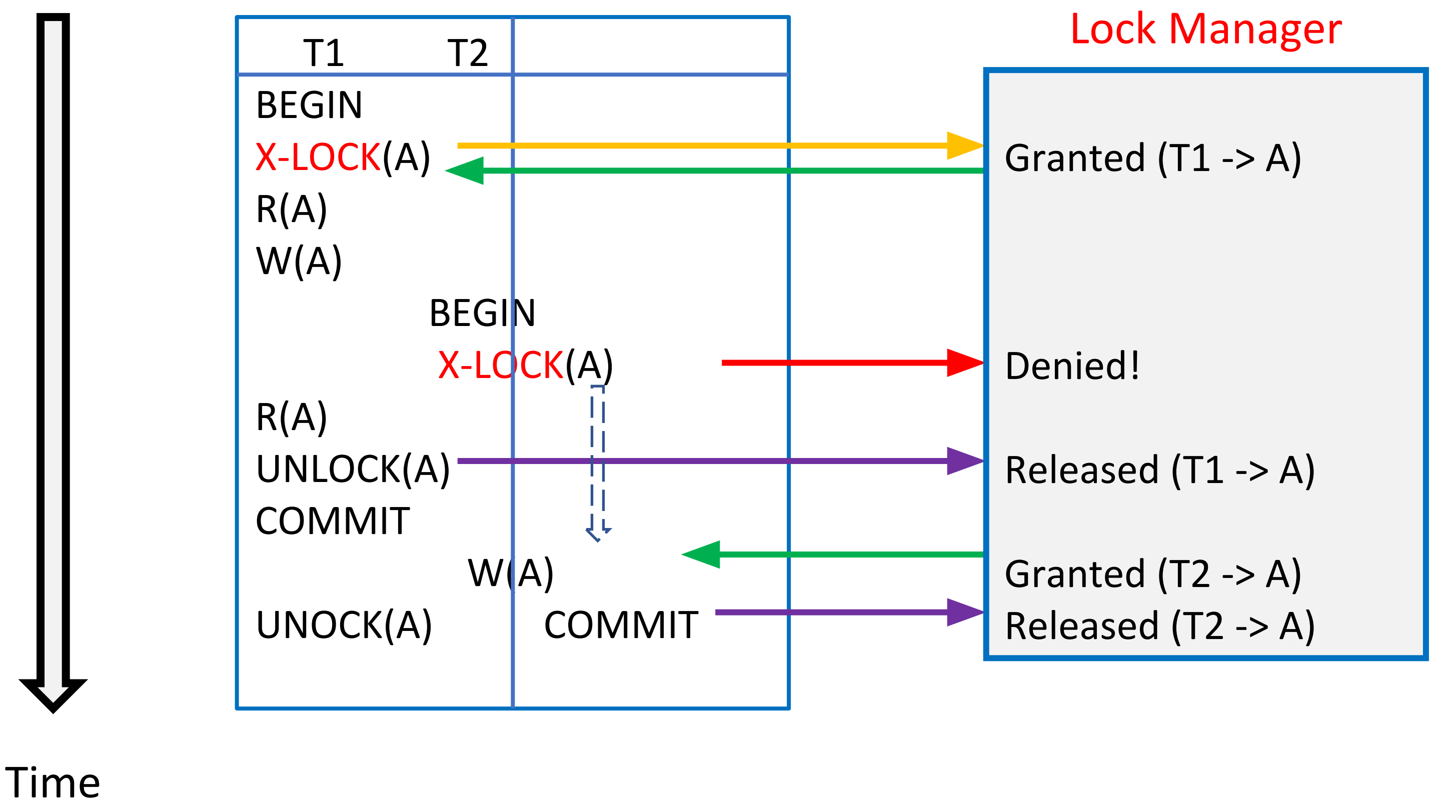
Concurrency Control

- We need to use a well-defined protocol that ensures that transactions execute correctly.
- Two categories:
  - Two-Phase Locking (2PL)
  - Timestamp Ordering (T/O) – discuss in future classes

Two-Phase Locking

- **Phase 1: Growing**
  - Each transaction requests the locks that it needs from the DBMS’s lock manager.
  - The lock manager grants/denies lock requests.
- **Phase 2: Shrinking**
  - The transaction is allowed to only release locks that it previously acquired.
  - It cannot acquire new locks.
- The transaction is not allowed to acquire/upgrade locks after the growing phase finishes.

Executing with 2PL

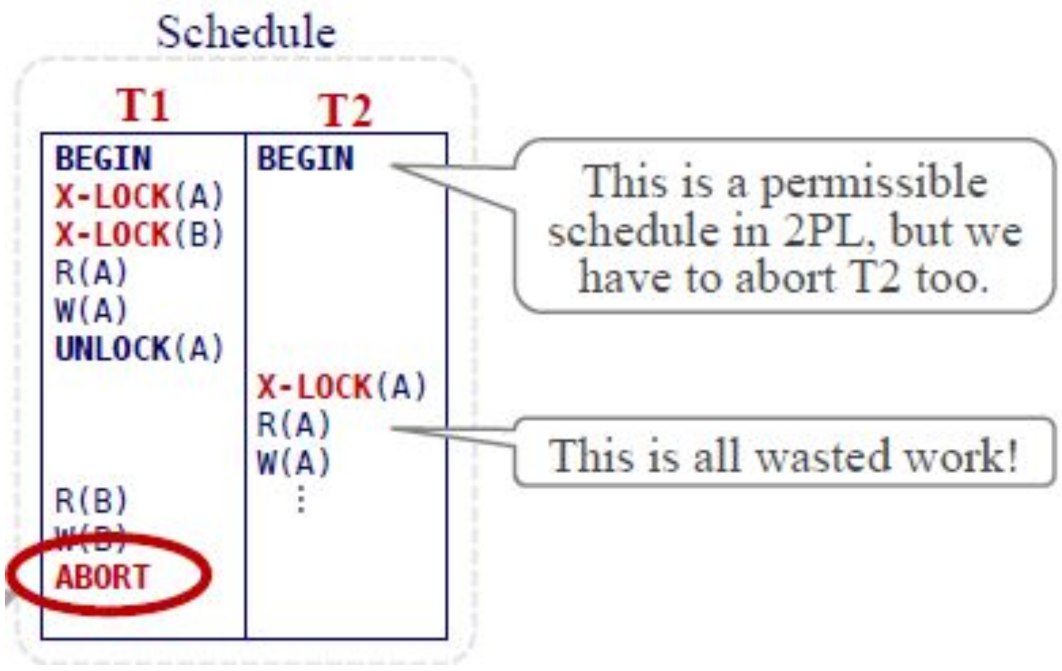


Lock Management

- Lock and unlock requests handled by the DBMS’s *lock manager* (LM).
- LM contains an entry for each currently held lock:
  - Pointer to a list of transactions holding the lock.
  - The type of lock held (shared or exclusive).
  - Pointer to queue of lock requests.
- When lock request arrives see if any other transaction holds a conflicting lock.
  - If not, create an entry and grant the lock
  - Else, put the requestor on the wait queue
- All lock operations must be atomic.
- Lock upgrade: The transaction that holds a shared lock upgrade to hold an exclusive lock.

Two-Phase Locking

- 2PL –
  - sufficient to guarantee conflict serializability (i.e., precedence graph is acyclic),
  - But, subject to *cascading aborts*.



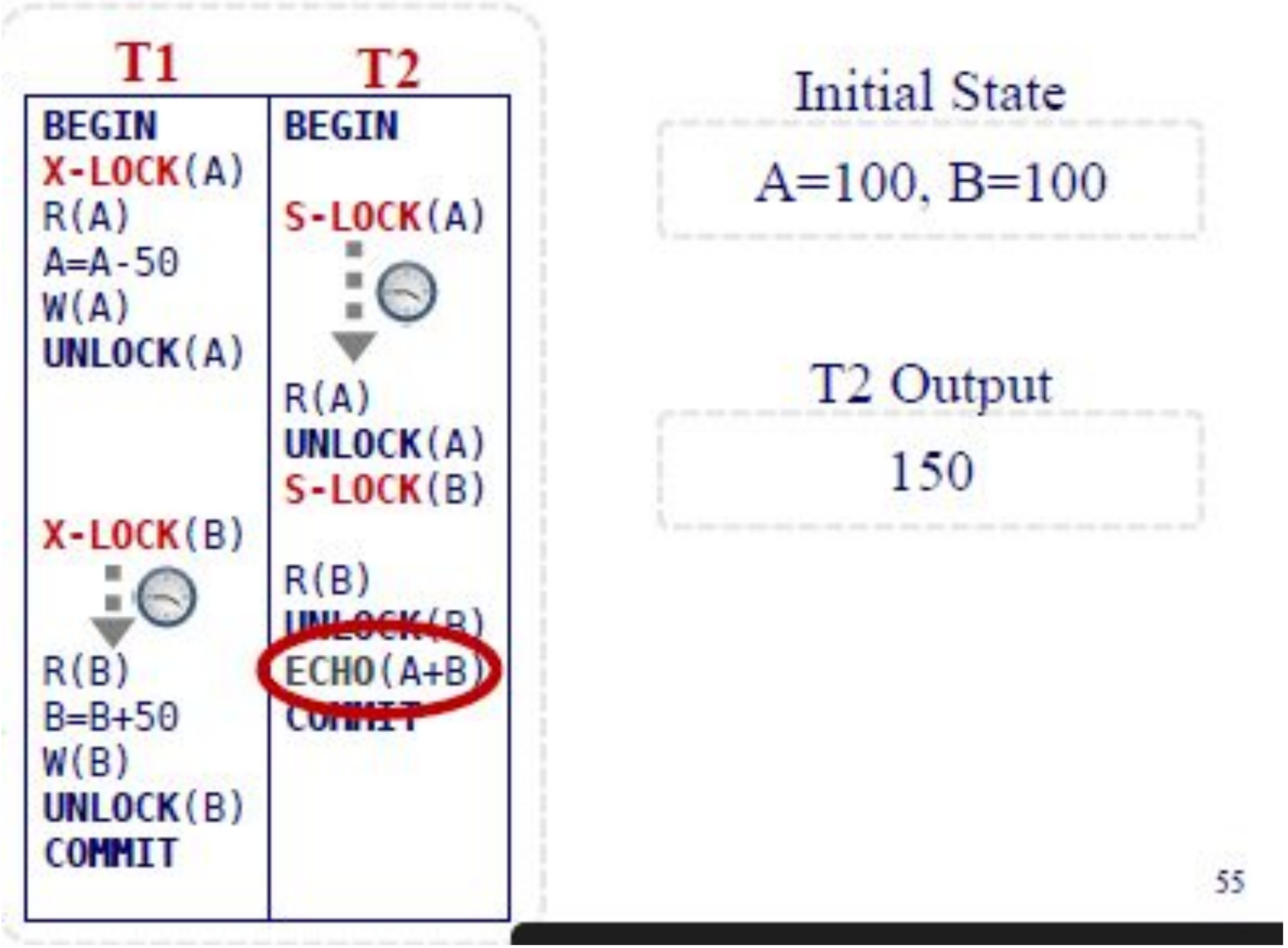
2PL Observations

- There are schedules that are serializable but would not be allowed by 2PL.
- Locking limits concurrency.
- May lead to deadlocks.
- May still have “dirty reads”
  - Solution: **Strict 2PL**

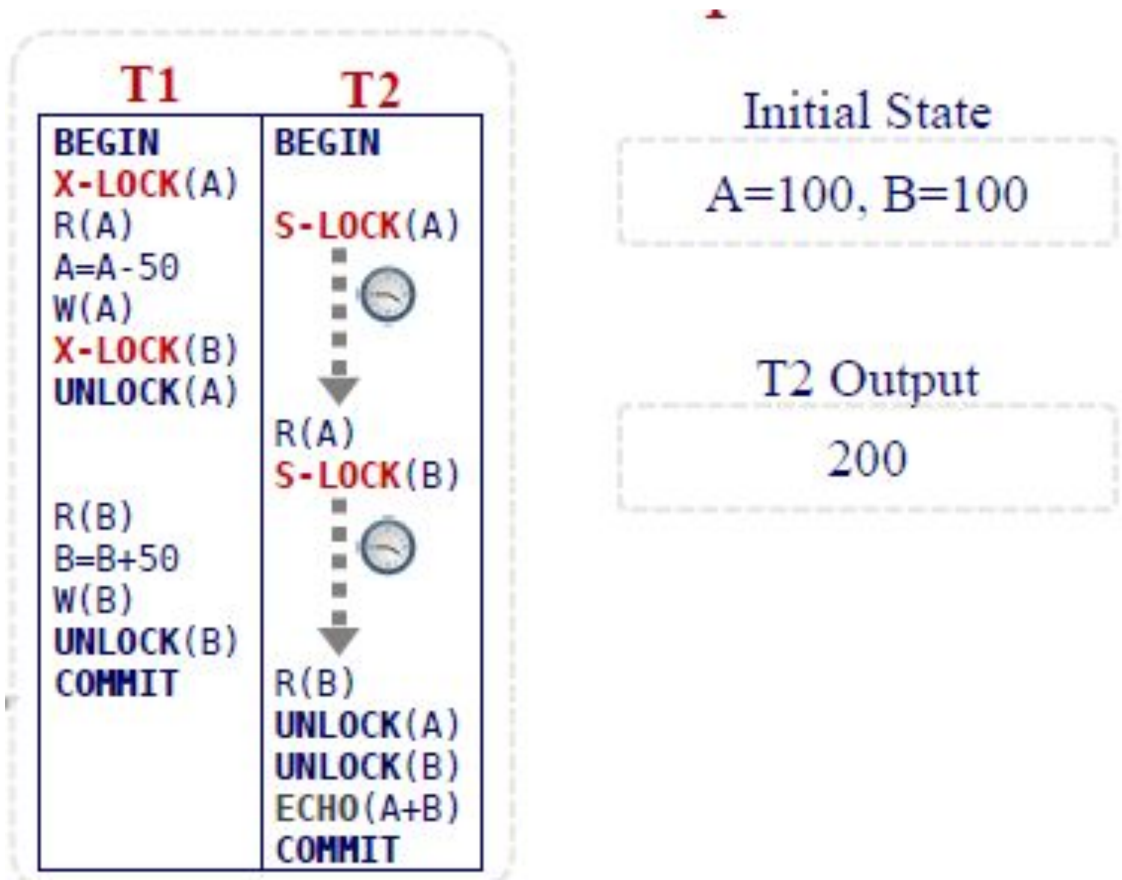
Strict Two-Phase Locking

- The transaction is not allowed to acquire/upgrade locks after the growing phase finishes.
- Allows only conflict serializable schedules, but it is actually stronger than needed.
- A schedule is ***strict*** if a value written by a transaction is not read or overwritten by other transactions until that transaction finishes.
- Advantages:
  - Recoverable.
  - Do not require cascading aborts.
  - Aborted transactions can be undone by just restoring original values of modified tuples.

Non 2PL Example



2PL Example


T1
T2
BEGIN
X-LOCK(A)
R(A)
A=A-50
W(A)
X-LOCK(B)
UNLOCK(A)
R(B)
B=B+50
W(B)
UNLOCK(B)
COMMIT
BEGIN
S-LOCK(A)
R(A)
S-LOCK(B)
R(B)
UNLOCK(A)
UNLOCK(B)
ECHO(A+B)
COMMIT
Initial State
A=100, B=100
T2 Output
200

Strict 2PL Example

T1	T2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-50	
W(A)	
X-LOCK(B)	
R(B)	
B=B+50	
W(B)	
UNLOCK(A)	
UNLOCK(B)	R(A)
COMMIT	S-LOCK(B)
	R(B)
	ECHO(A+B)
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

Initial State

A=100, B=100

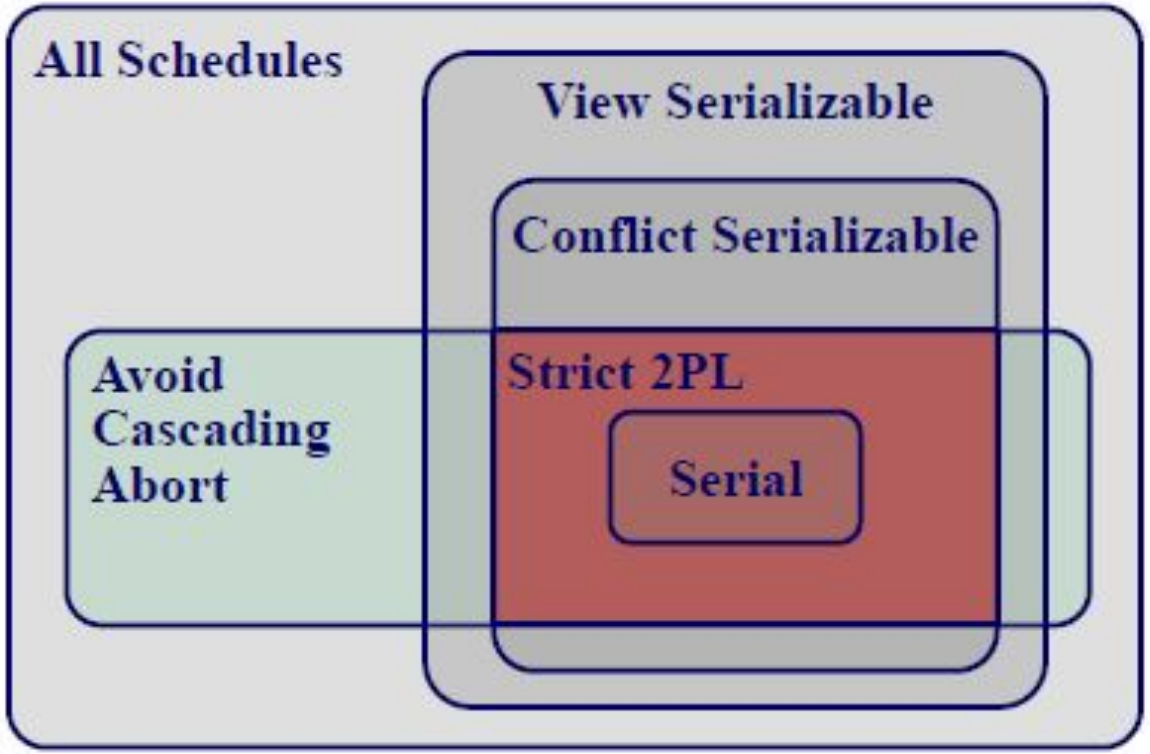
T2 Output

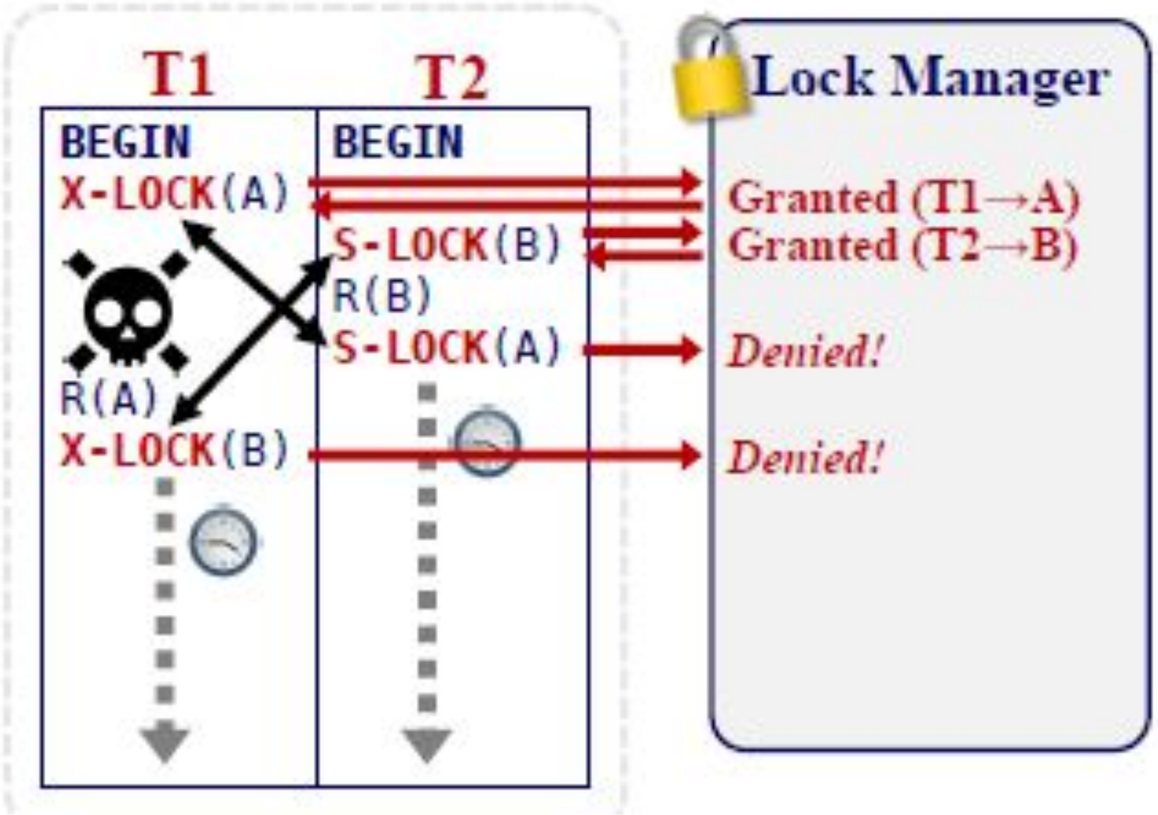
200

Strict Two-Phase Locking

- Transactions hold all of their locks until commit.
- Good:
  - Avoids “dirty reads” etc.
- Bad:
  - Limits concurrency even more
  - And still may lead to deadlocks

Schedules


All Schedules
View Serializable
Conflict Serializable
Avoid
Cascading
Abort
Strict 2PL
Serial


T1
T2
Lock Manager
BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)
BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)
Granted (T1→A)
Granted (T2→B)
Denied!
Denied!

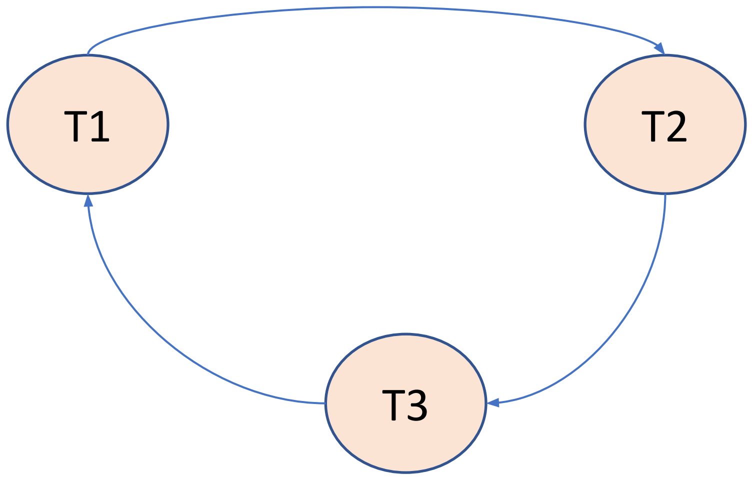
Deadlocks

- **Deadlock:** Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection
- Many systems just punt and use timeouts
  - What are the dangers with this approach?

Deadlock Detection

- The DBMS creates a ***waits-for*** graph:
  - Nodes are transactions
  - Edge from T_i to T_j if T_i is waiting for T_j to release a lock
- The system periodically check for cycles in ***waits-for*** graph.

T1	T2	T3
BEGIN	**BEGIN**	**BEGIN**
S-LOCK(A)		
S-LOCK(D)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)



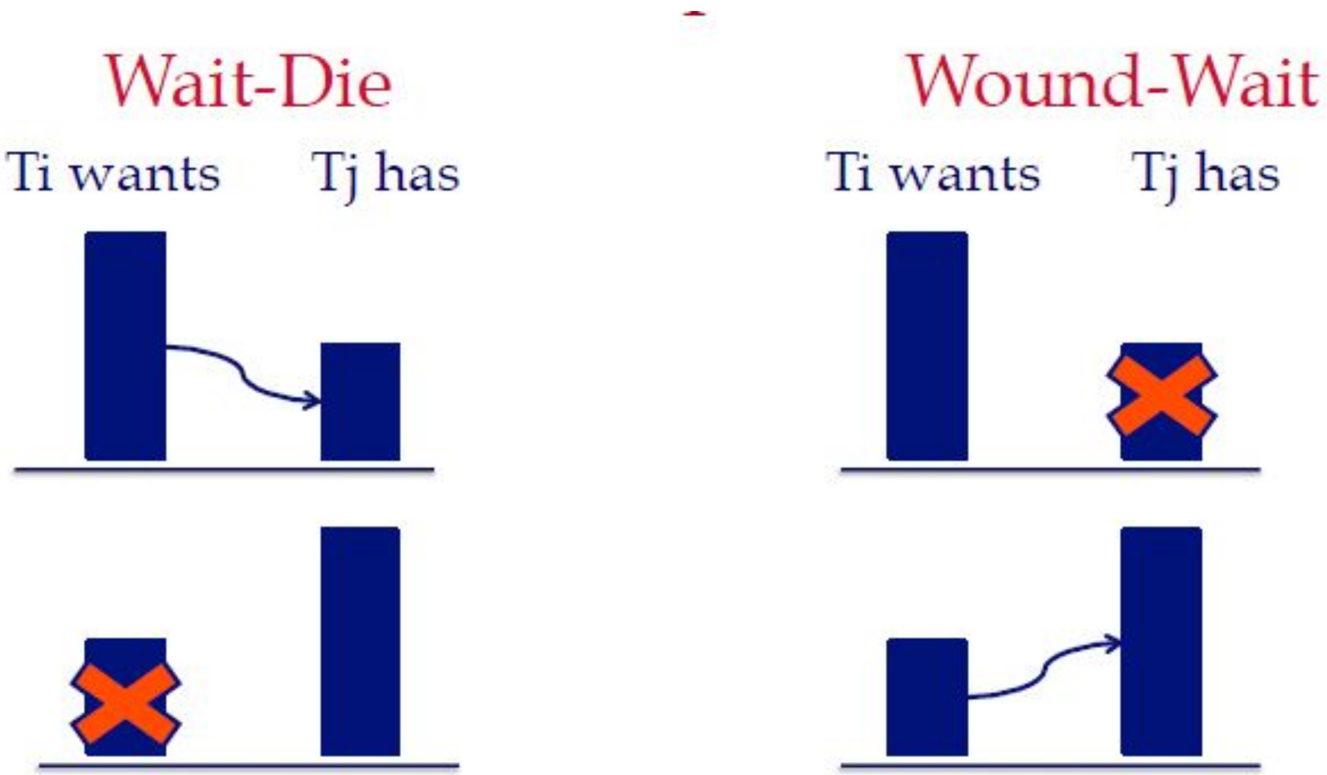
Deadlock Detection

- What do we do when we find a deadlock?
- Select a “victim” and rollback it back to break the deadlock.
- Which one do we choose?
- It depends…
  - By age (lowest timestamp)
  - By progress (least/most queries executed)
  - By the # of items already locked
  - By the # of transactions that we have to rollback with it
- We also should consider the # of times a transaction has been restarted in the past.
- How far do we rollback?
- It depends…
  - Completely
  - Minimally (i.e., just enough to release locks)

Deadlock Prevention

- When a transaction tries to acquire a lock that is held by another transaction, kill one of them to prevent a deadlock.
- Assign priorities based on timestamps:
  - Older → higher priority (e.g., T1 > T2)
- Two different prevention policies:
  - **Wait-Die:** If T1 has higher priority, T1 waits for T2; otherwise T1 aborts (“old wait for young”)
  - **Wound-Wait:** If T1 has higher priority, T2 aborts; otherwise T1 waits (“young wait for old”)

Deadlock Prevention



Deadlock Prevention

T1	T2
BEGIN	
	BEGIN
	X-LOCK(A)
X-LOCK(A)	

Wait-Die

T1 Waits

Wound-Wait

T2 aborted

T1	T2
BEGIN	
X-LOCK(A)	
	BEGIN
	X-LOCK(A)

Wait-Die

T2 Aborted

Wound-Wait

T1 waits

Deadlock Prevention

- Why do these schemes guarantee no deadlocks?
- Only one “type” of direction allowed.
- When a transaction restarts, what is its (new) priority?
- Its original timestamp. Why?

Performance Problems

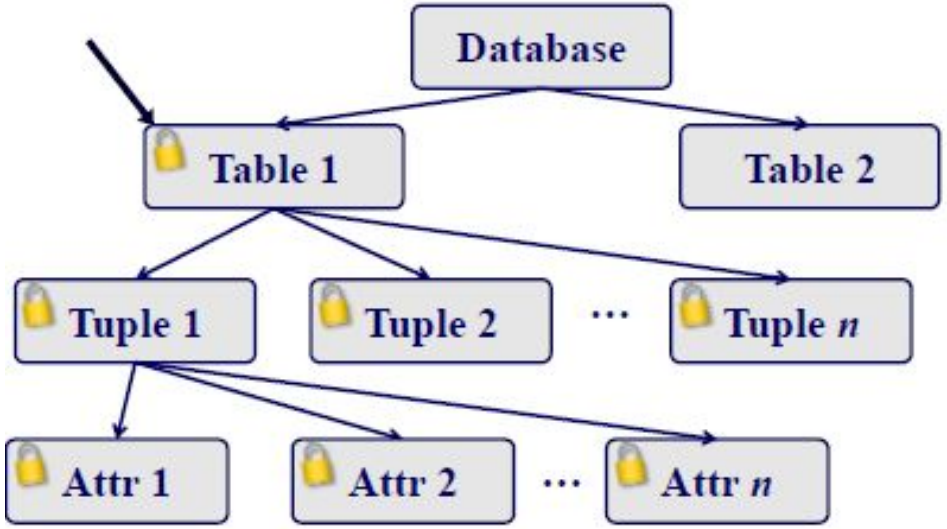
- Executing more transactions can increase the throughput.
- But there is a tipping point where adding more transactions actually makes performance worse.

Quiz

- is there a serial schedule (= interleaving) that is not serializable?
- is there a serializable schedule that is not serial?
- can 2PL produce a non-serializable schedule? (assume no deadlocks)
- is there a serializable schedule that can not be produced by 2PL?

Lock Granularities

- When we say that a transaction acquires a “lock”, what does that actually mean?
  - On a field? Record? Page? Table?
- Ideally, each transaction should obtain fewest number of locks that is needed…



Intention Locks

- Intention locks allow a higher level node to be locked in **S** or **X** mode without having to check all descendent nodes.
- If a node is in an intention mode, then explicit locking is being done at a lower level in the tree.

Intention Locks

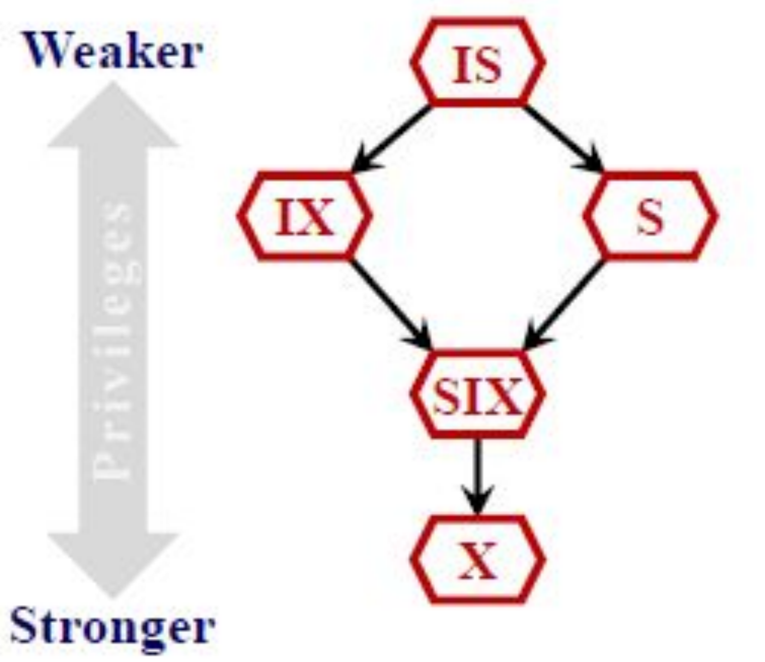
- Useful in practice as each transaction only needs a few locks.
- Intention locks help improve concurrency:
- **Intention-Shared** (**IS**): Indicates explicit locking at a lower level with shared locks.
- **Intention-Exclusive** (**IX**): Indicates locking at lower level with exclusive or shared locks.
- **Shared+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.

Compatibility Matrix

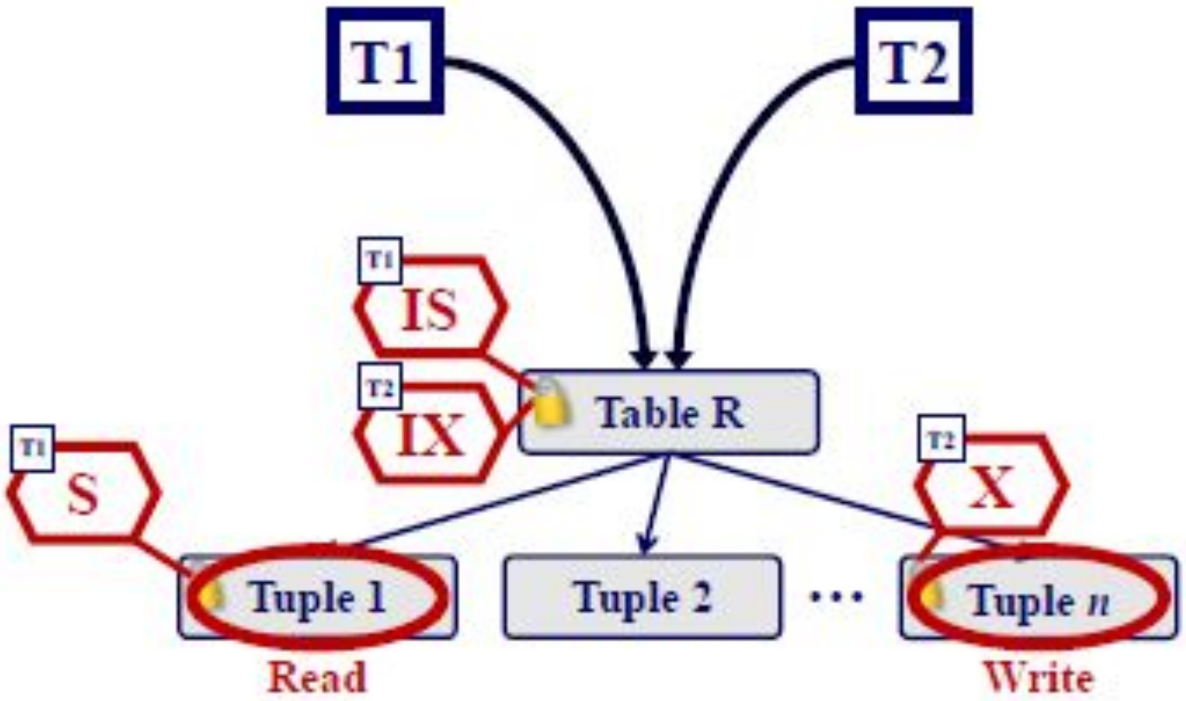
		T2 Wants				
		IS	IX	S	SIX	X
T1 Holds	IS	✔	✔	✔	✔	✘
	IX	✔	✔	✘	✘	✘
	S	✔	✘	✔	✘	✘
	SIX	✔	✘	✘	✘	✘
	X	✘	✘	✘	✘	✘

Multiple Granularity Protocol

- Each transaction obtains appropriate lock at highest level of the database hierarchy.
- To get **S** or **IS** lock on a node, the txn must hold at least **IS** on parent node.
  - What if transaction holds **SIX** on parent? **S** on parent?
- To get **X**, **IX**, or **SIX** on a node, must hold at least **IX** on parent node.

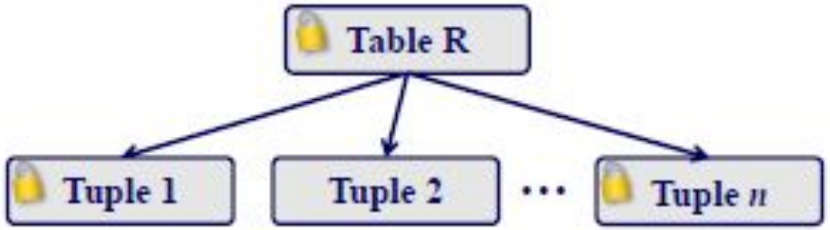


Example – Two Level Hierarchy



Example

- Assume three transactions execute at same time:
- **T1:** Scan **R** and update a few tuples.
- **T2:** Scan a portion of tuples in **R**.
- **T3:** Scan all tuples in **R**.



Example

- **T1:** Get an **SIX** lock on **R**, then get **X** lock on tuples that are updated.
- **T2:** Get an **IS** lock on **R**, and repeatedly get an **S** lock on tuples of **R**.
- **T3:** Two choices:
    - T3 gets an **S** lock on **R**.
    - OR, T3 could behave like T2; can use ***lock escalation*** to decide which.