Database Systems
Lecture18 – Chapter 18: Concurrency Control
(Part I: Lock-based Concurrency Control)

Beomseok Nam (남범석)
bnam@skku.edu

Last Class

- Serializable schedule in any DBMS
  → "Conflict serializable" schedule

- Swapping method
  - Swap non-conflicting operations
- Dependency graph
  - Topological sorting

- Only if the entire schedule is given, we can determine whether the schedule is serializable or not.

Scheduling at runtime

- Q: What if we do not know the entire schedule ahead of time?

- We need a way to guarantee that all execution schedules are serializable.

- Solution:
  - Use **locks**
  - Use **timestamps**
  - Use **versions**

Lock Types

- S-LOCK: Shared locks for reads
- X-LOCK: Exclusive locks for both reads and writes
- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

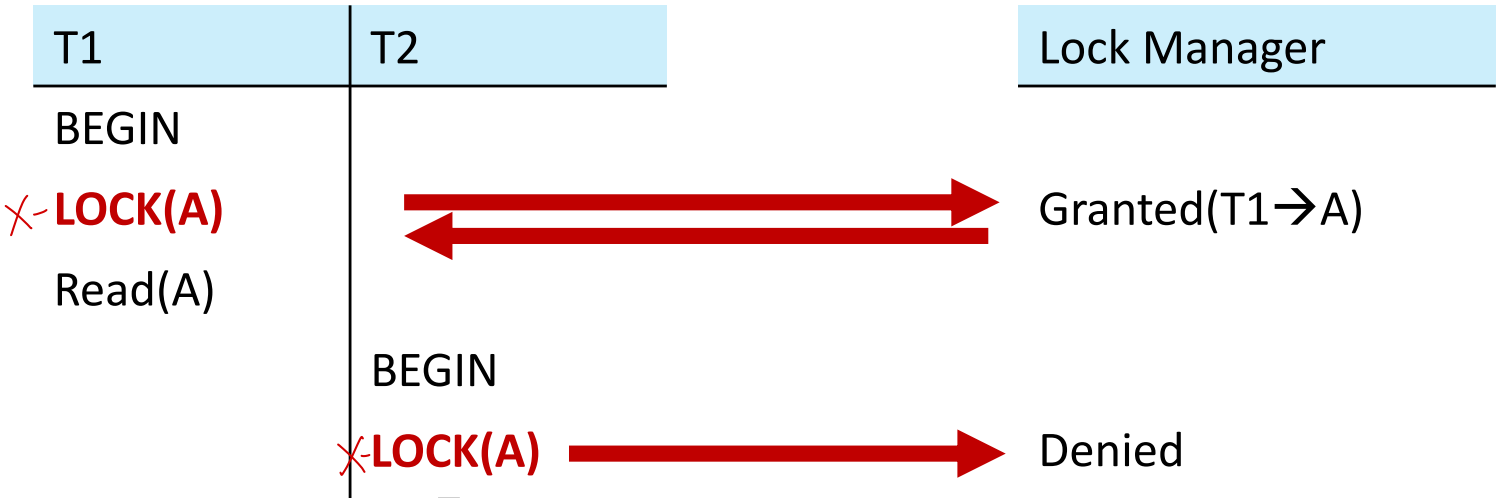
- Transactions request or release locks.
- Lock manager grants or blocks requests.
- Lock manager updates its internal lock-table.
- → It keeps track of what transactions hold what locks and what transactions are waiting to acquire any locks.

Executing with Locks

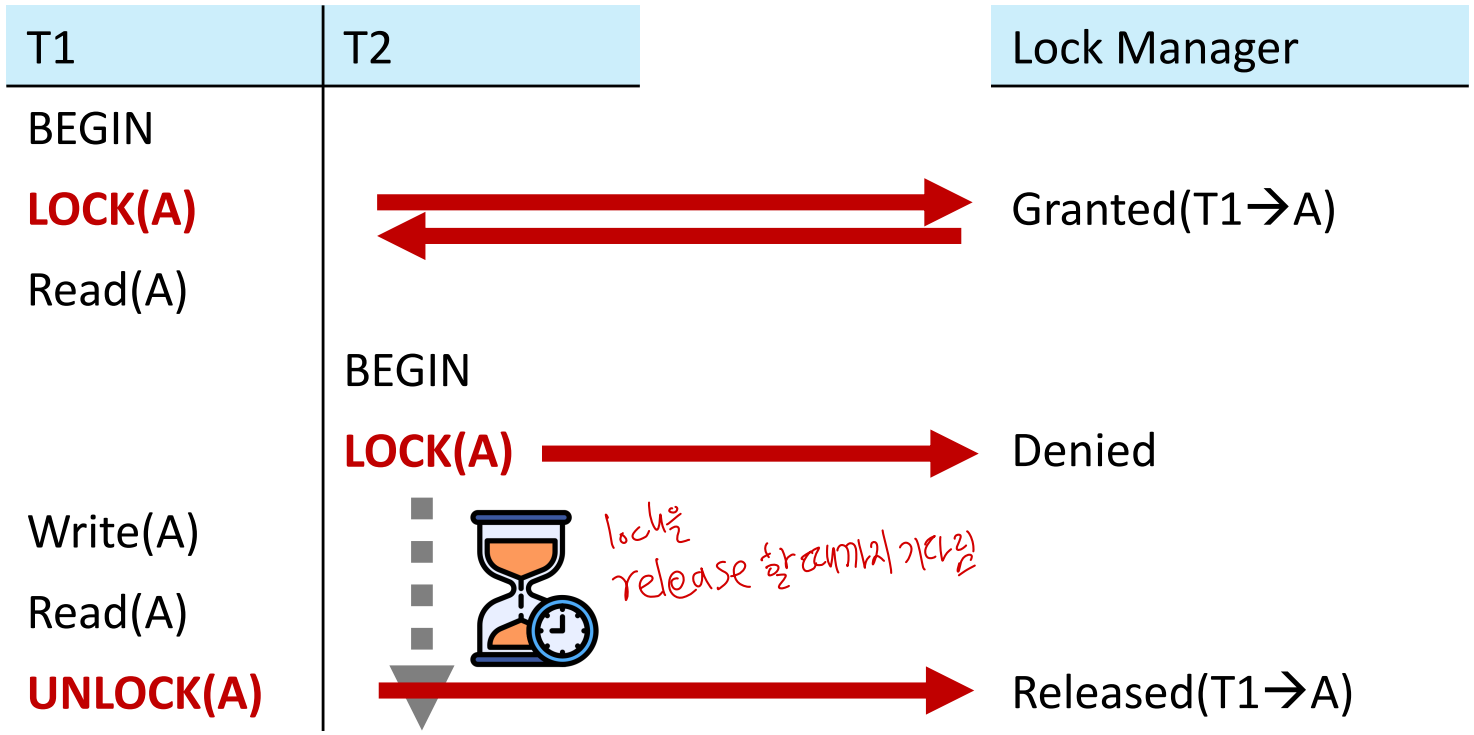
Lock권한주는 시스템

T1	T2	Lock Manager
BEGIN		
X-LOCK(A)		Granted(T1→A)

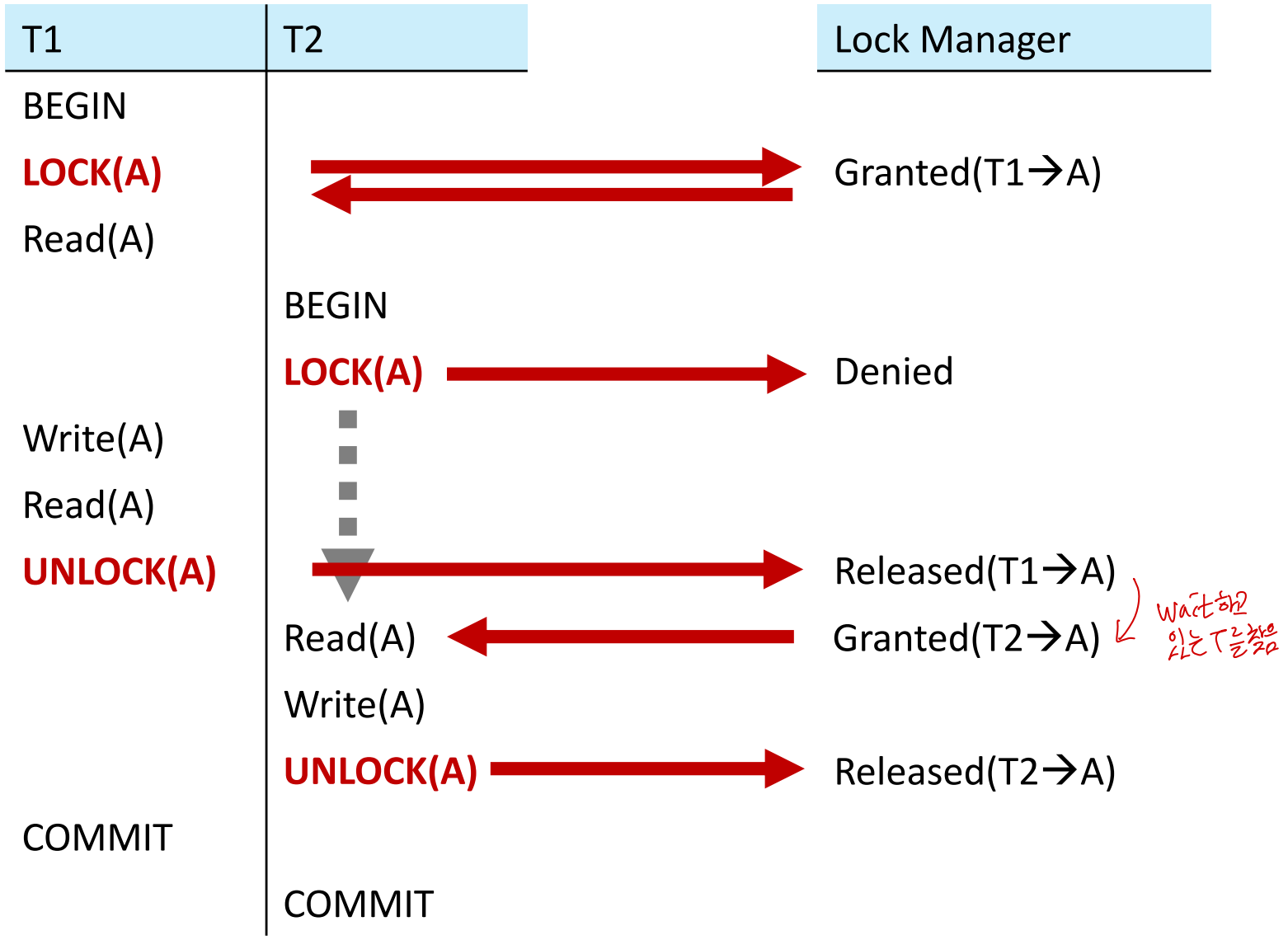
Executing with Locks



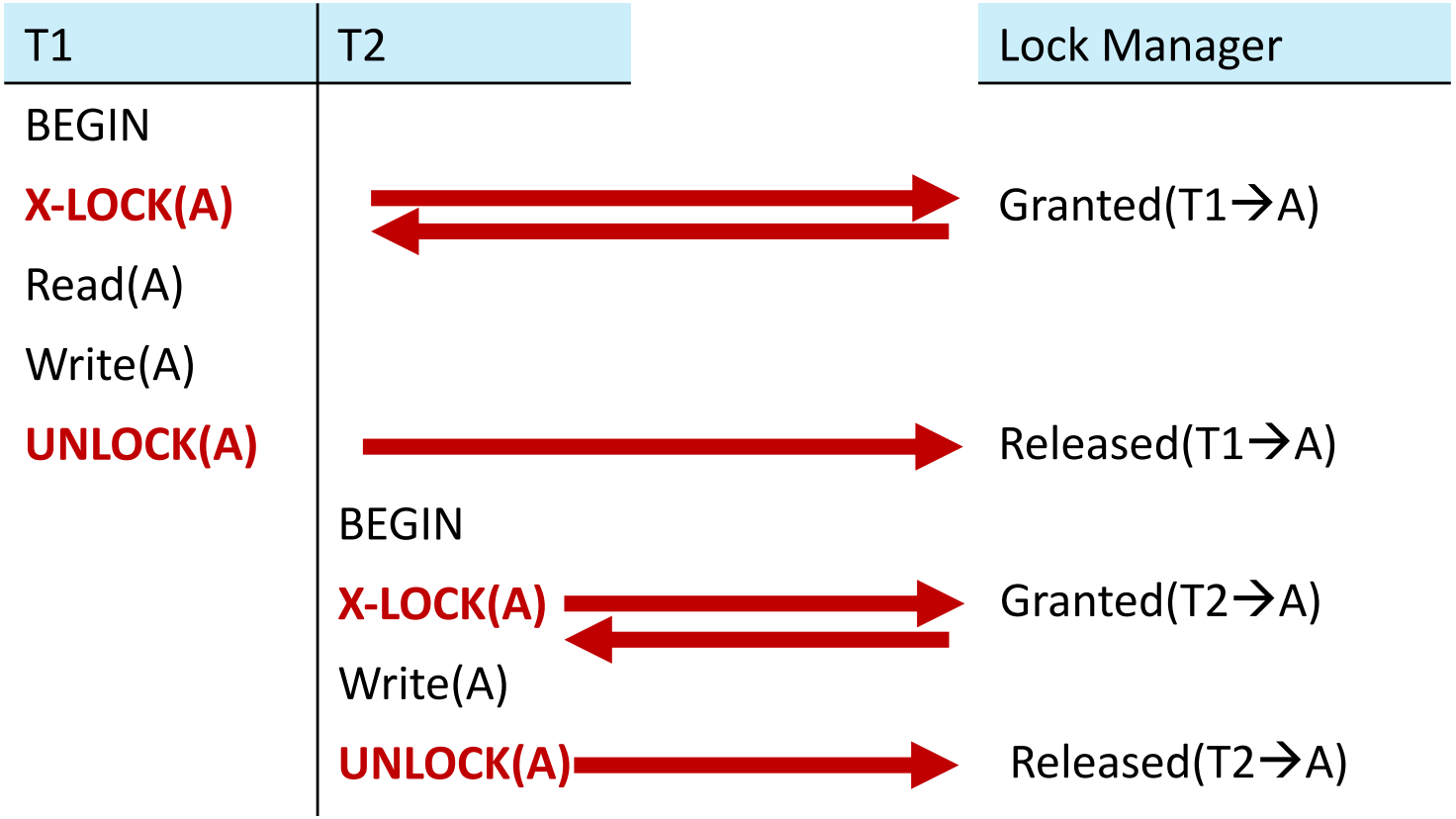
Executing with Locks



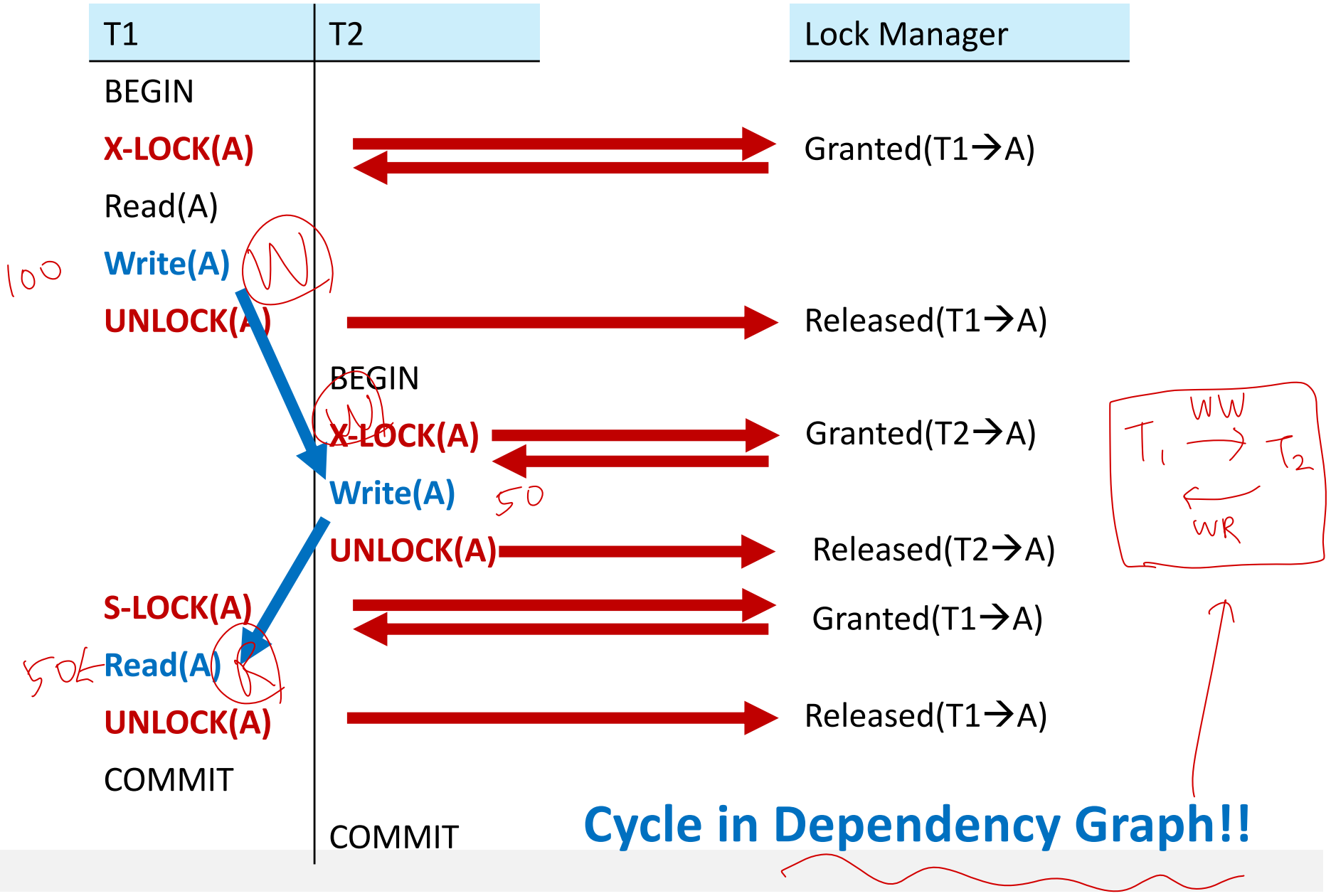
Executing with Locks



What if releasing locks early?



What if releasing locks early?



What if releasing locks early?– Another Example

T1	T2
X-LOCK(A)	
Read(A)	
A=A-100	S-LOCK(A)
Write(A)	
UNLOCK(A)	
	Read(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	Read(B)
	UNLOCK(B)
Read(B)	Print A+B
B=B+100	COMMIT
Write(B)	
UNLOCK(B)	
COMMIT	

Initial Database

A=1000, B=1000

T2 Output

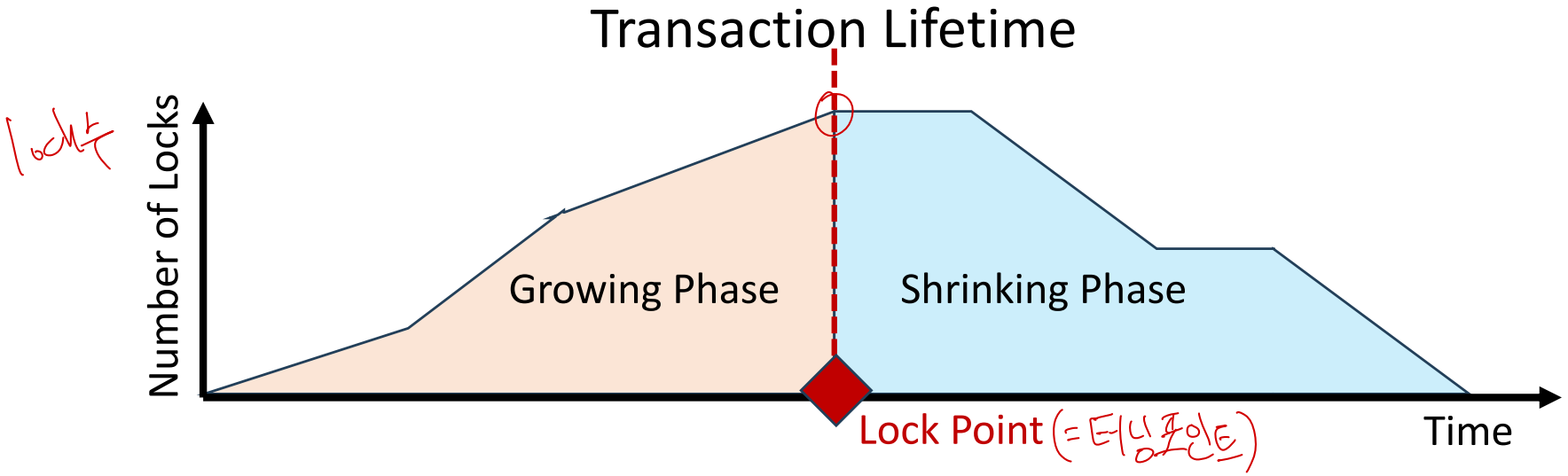
A+B=1900

Locking alone is not sufficient!!

Two-Phase Locking Protocol

Two-Phase Locking (2PL) Protocol

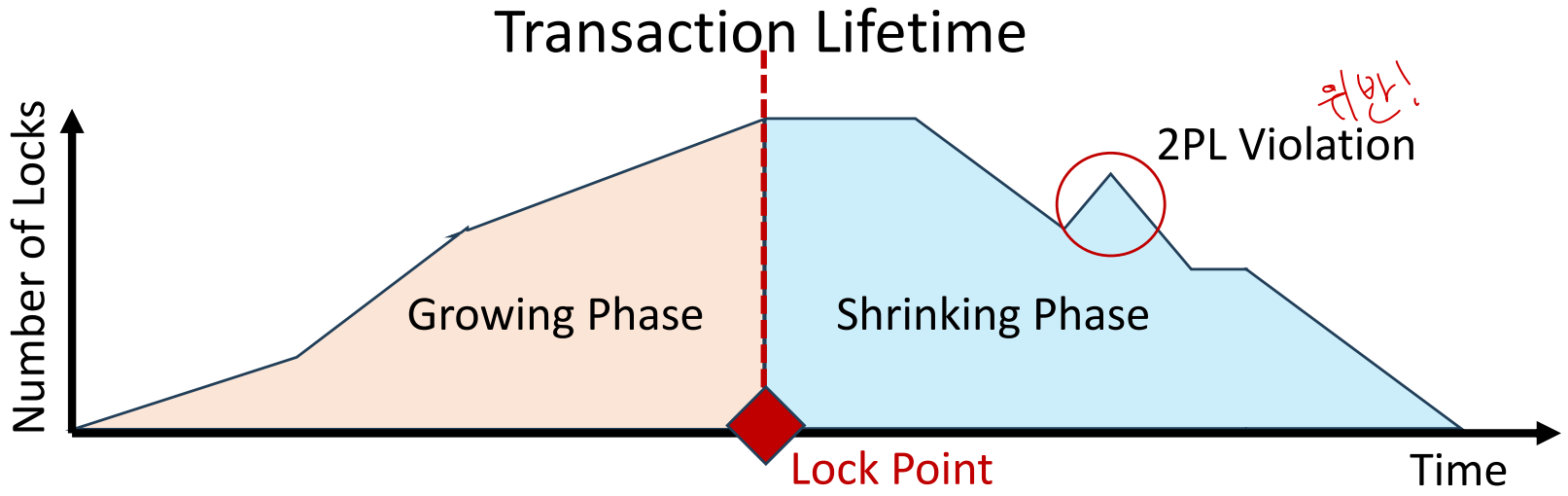
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks



- **Lock point**: a time when a transaction acquired its final lock

Two-Phase Locking (2PL) Protocol

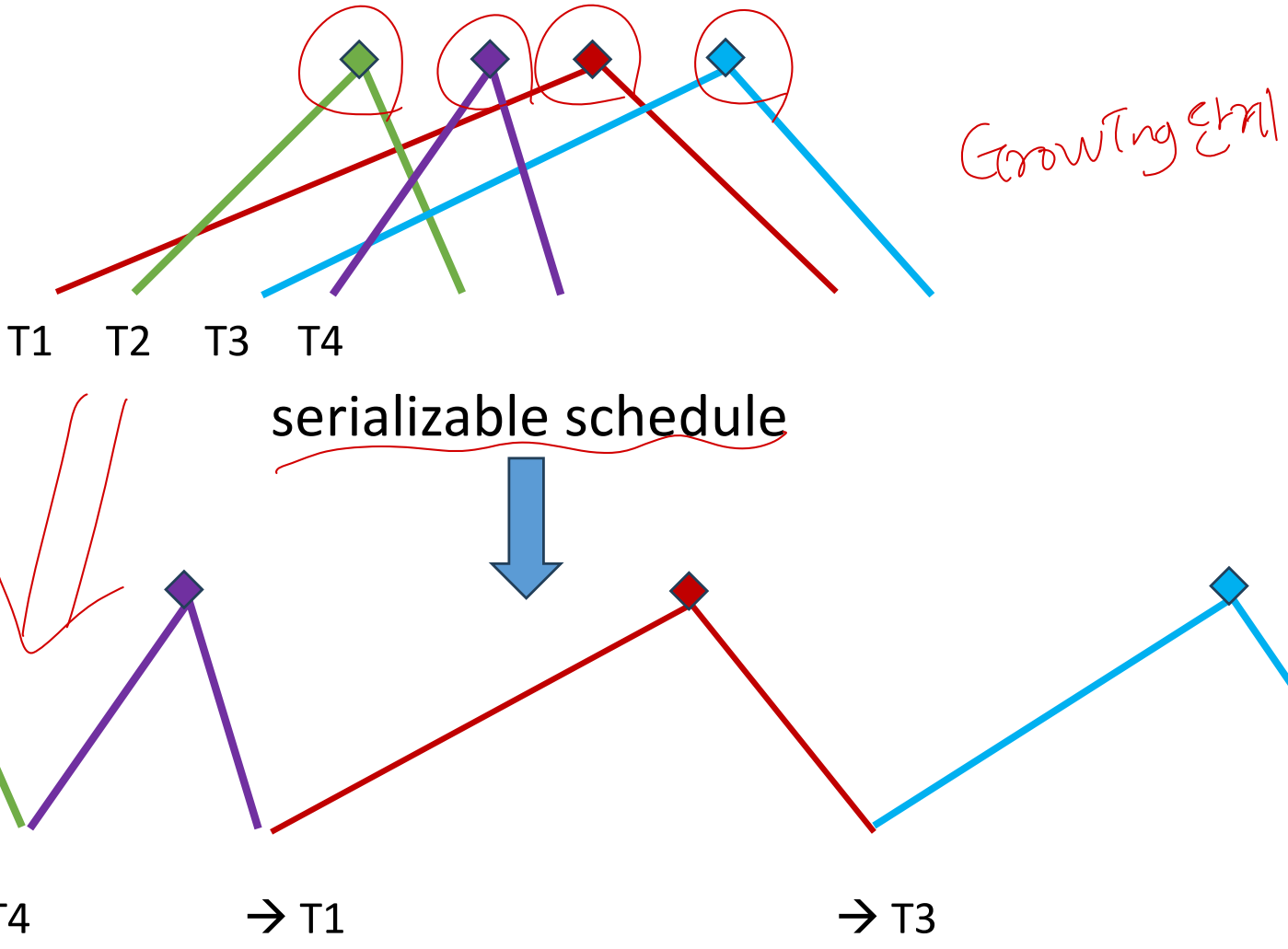
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks



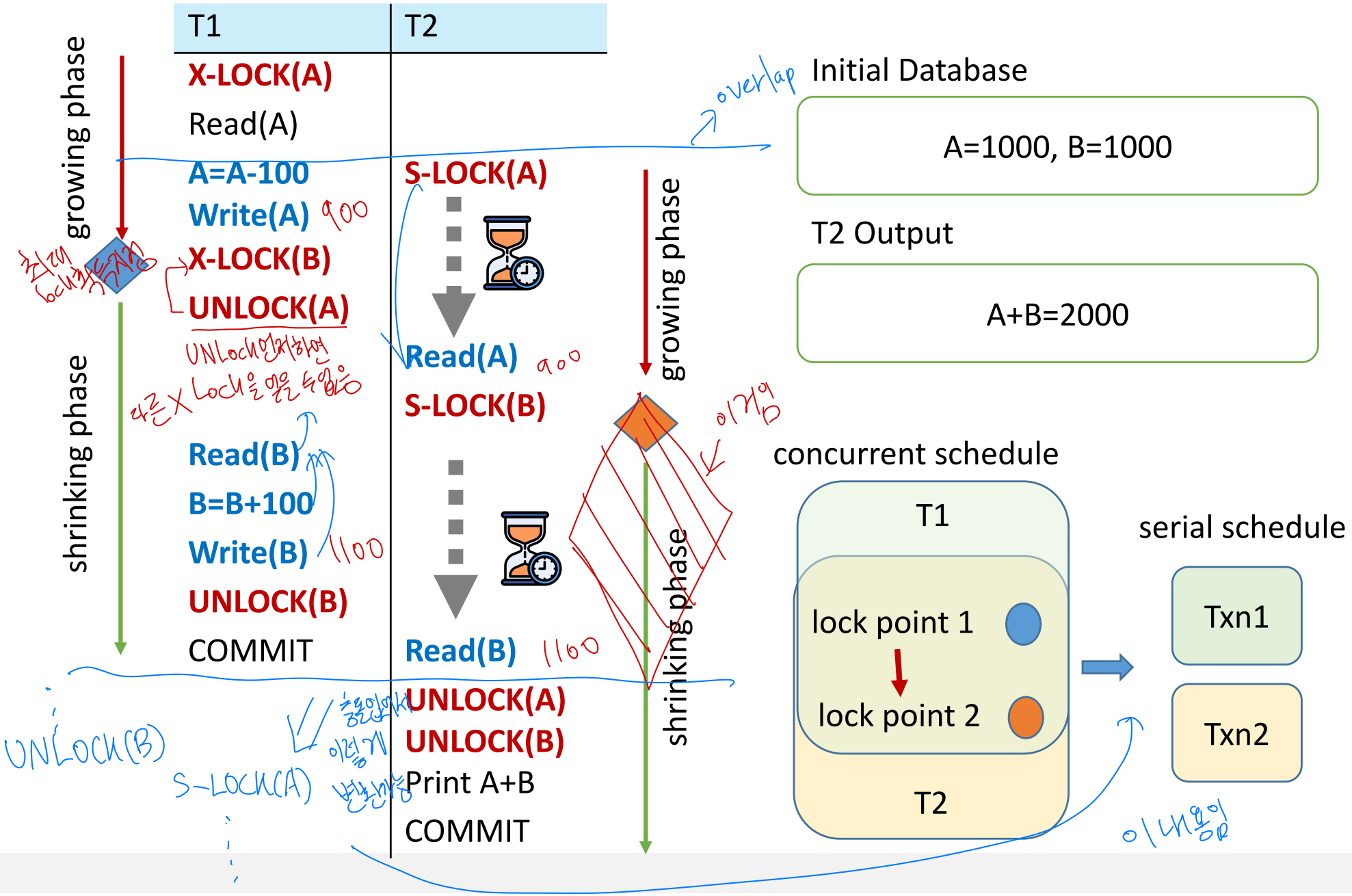
- **Lock point**: a time when a transaction acquired its final lock

Two-Phase Locking (2PL) Protocol

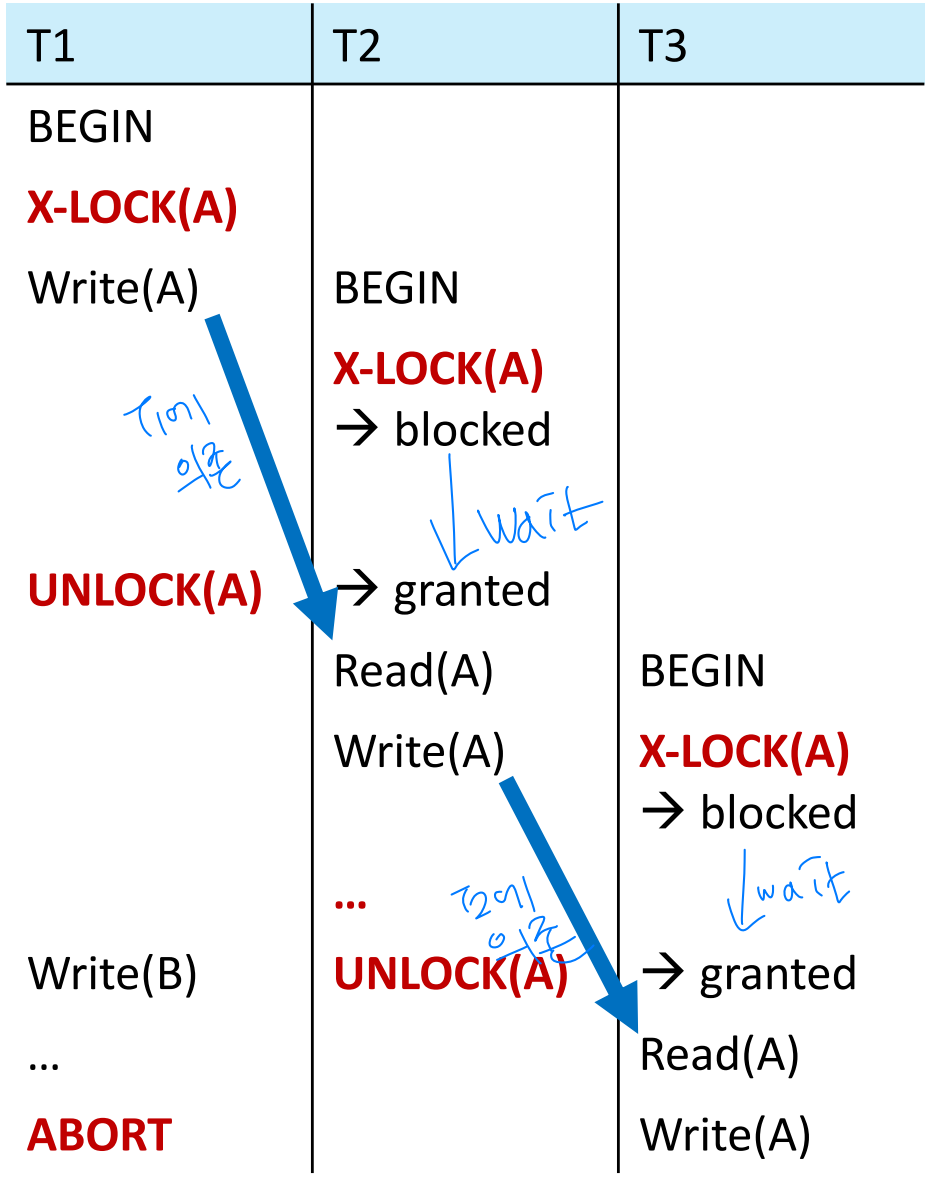
- The order of transactions = order of lock points



Two-Phase Locking (2PL) - Example



Cascading Rollback in 2PL



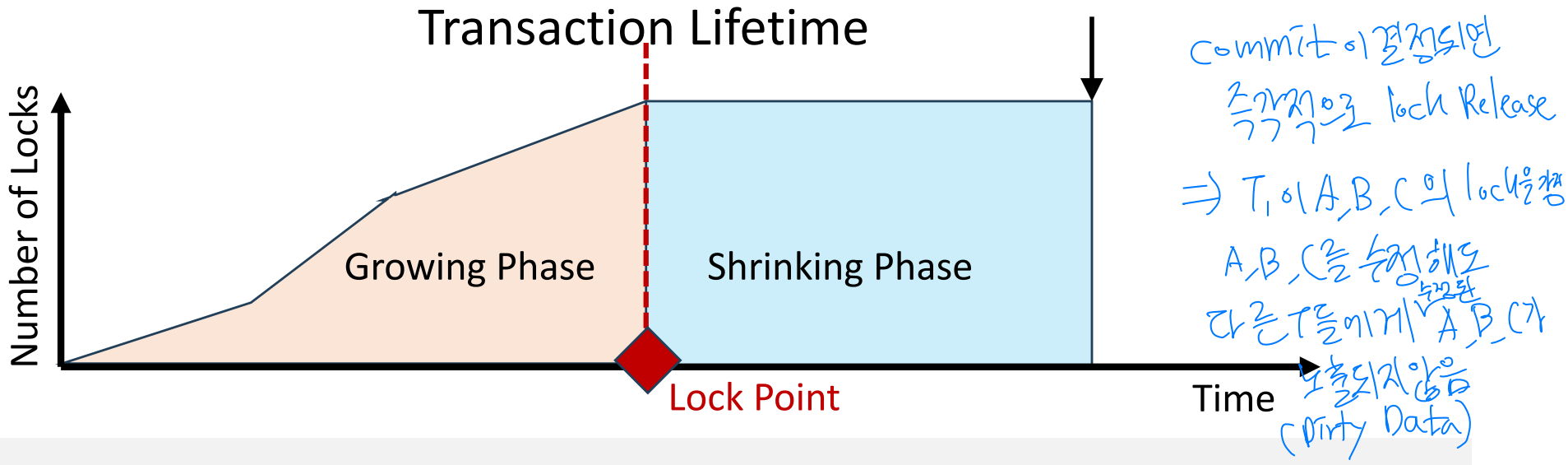
T_2 and T_3 also abort when T_1 aborts

Strict Two-Phase Locking Protocol (2PL)

- **Strict 2PL (a.k.a Rigorous 2PL)** prevents cascading roll-back
  - Transaction is only allowed to release locks after it commits/aborts.
  - A value written by a txn is not read or overwritten by other txns until that txn finishes (No Dirty Read)

Concurrency level을 높게 ↑ 성능↓ But 안전하다 (어떤 경우는 그렇지 않을 수도 있음)

Release all locks when commits/aborts



commit이 끝나면 그 순간에 lock Release

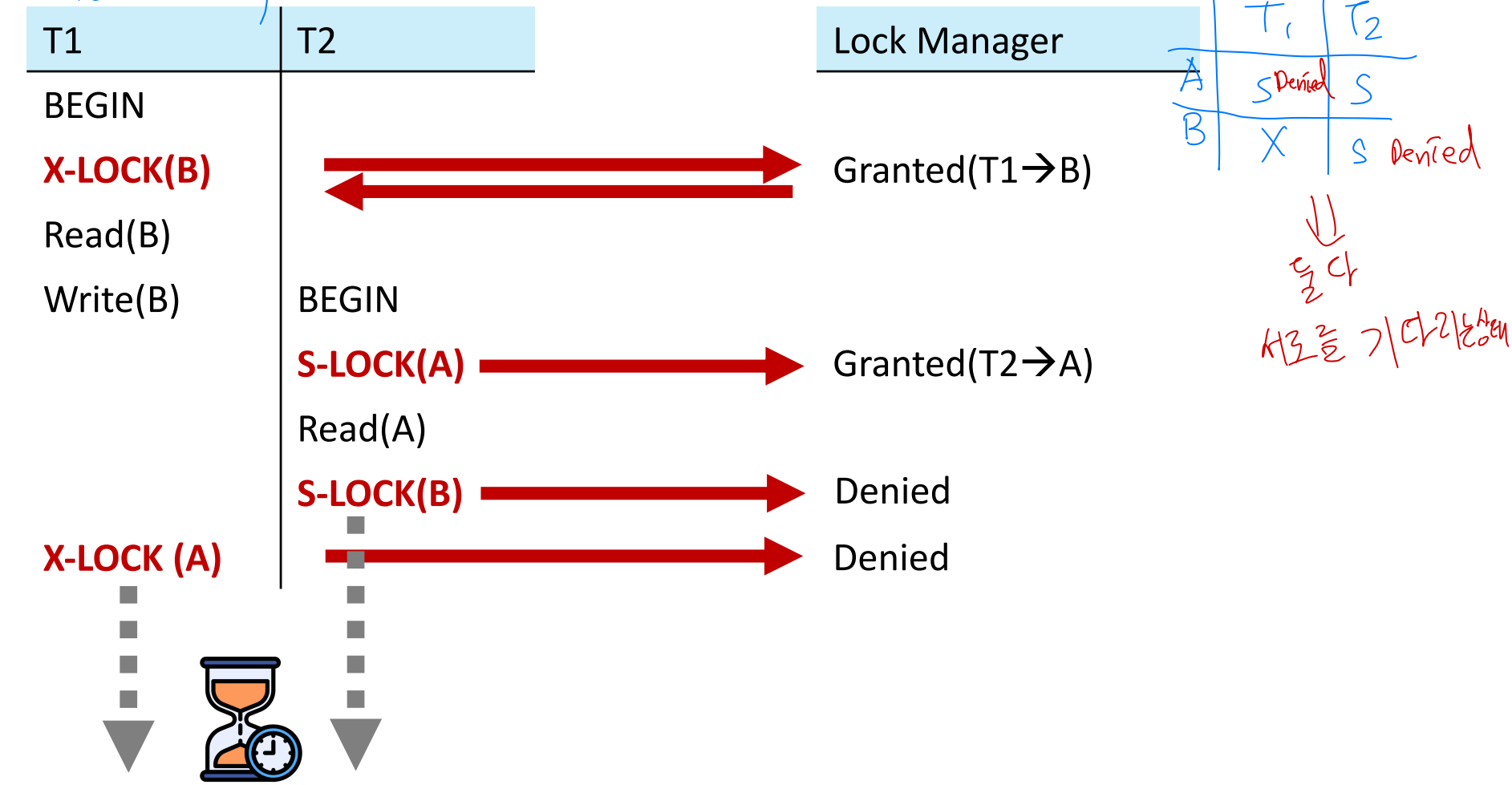
⇒ T_1이 A, B, C의 lock을 가짐

A, B, C를 수정해도 다른 트랜잭션들에게 수정된 A, B, C가 노출되지 않음 (Dirty Data)

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 ensures serialization

Deadlock in 2PL



Neither T_1 nor T_2 can make progress.

Either T_1 or T_2 must be rollbacked and release its lock.

Two ways of dealing with Deadlocks

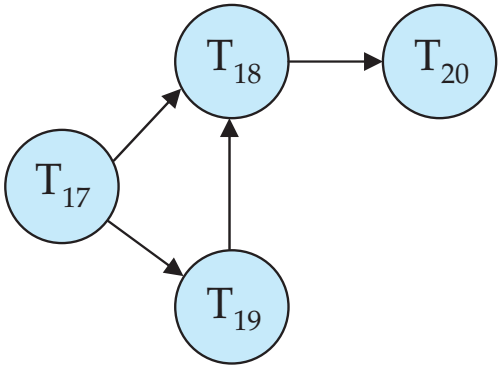
- Approach #1:
  - Deadlock Detection → optimistic 낙관적 가정
    - If a deadlock is detected, select and kill a victim txn

- Approach #2:
  - Deadlock Prevention → Deadlock이 발생할 것을 기다리지 않음 = pessimistic 비관적 가정

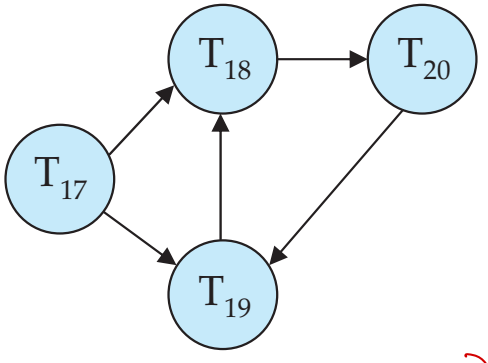
Deadlock Detection

- The DBMS creates a ***waits-for*** graph to keep track of what locks each txn is waiting to acquire:
  - → 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 checks for cycles in waits-for graph and then decides how to break it.



Wait-for graph without a cycle



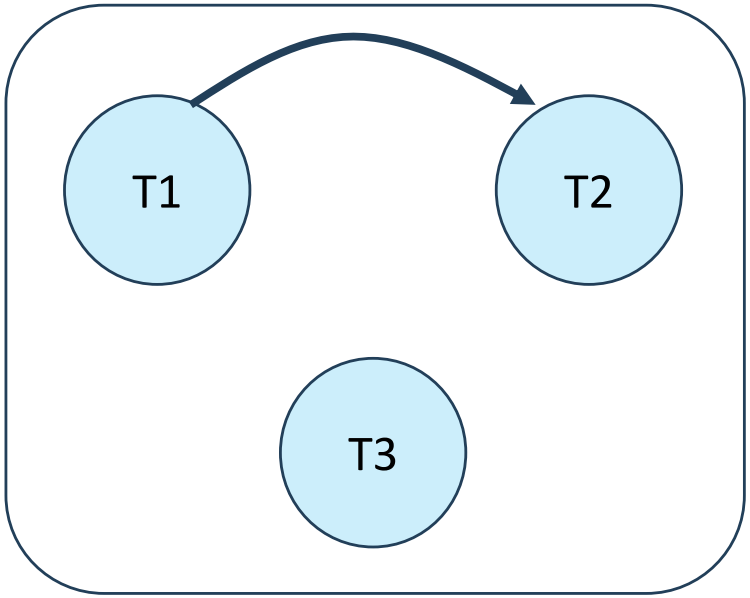
Wait-for graph with a cycle

Deadlock Detection

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

Denied (wait)

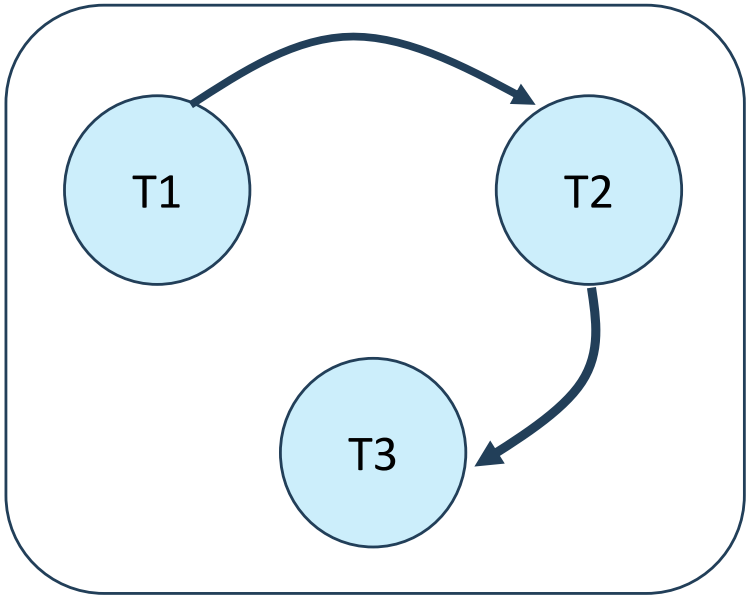
Waits-for Graph



Deadlock Detection

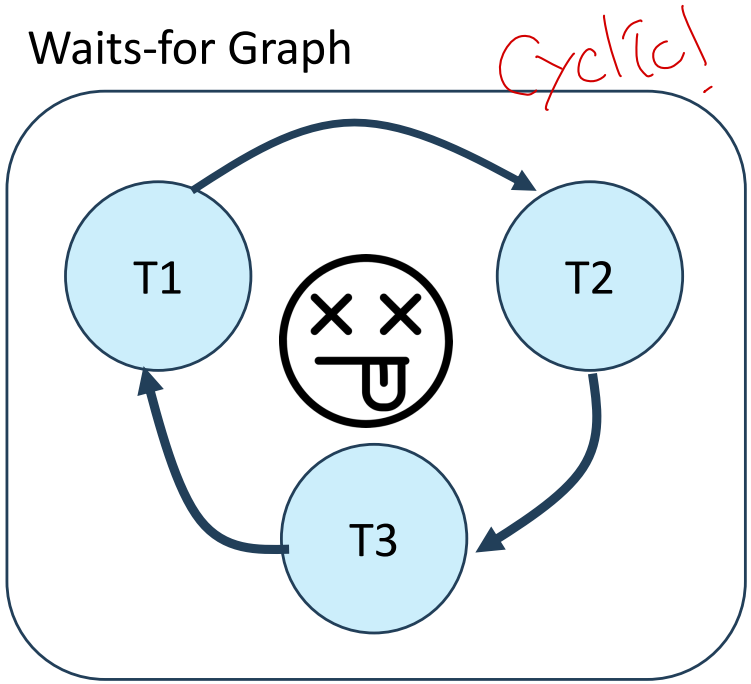
T1	T2	T3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-for Graph



Deadlock Detection

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



Deadlock Handling

- When the DBMS detects a deadlock, it will select a "victim" txn to rollback to break the cycle.

- Selecting the proper victim depends on a lot of different parameters….
  - → By age (lowest timestamp)
  - → By progress (least/most queries executed)
  - → By the # of items already locked
  - → By the # of txns that we have to rollback with it

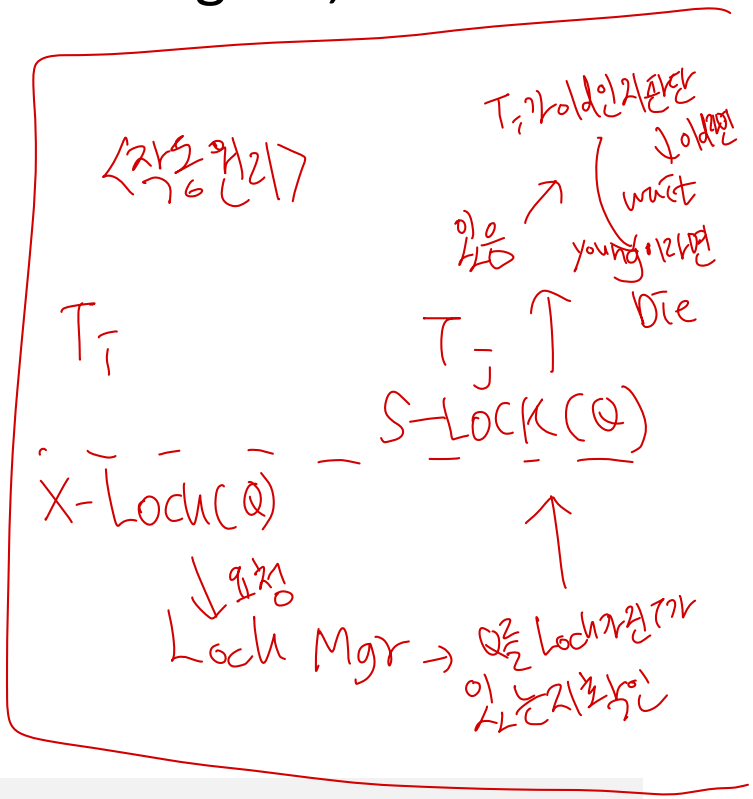
- We also should consider the # of times a txn has been restarted in the past to prevent starvation.

Deadlock Prevention

- When a txn tries to acquire a lock that is held by another txn, the DBMS **kills** one of them **to prevent** a deadlock.
- This approach does not require a *waits-for* graph or detection algorithm.

Deadlock Prevention

- Assign priorities based on timestamps:
  → Older Timestamp = Higher Priority (e.g., $T_1 > T_2$)
- **Wait-Die**
  → If requesting txn has higher priority than holding txn, then requesting txn waits for holding txn.
  → Otherwise requesting txn aborts.



Deadlock Prevention

- Assign priorities based on timestamps:

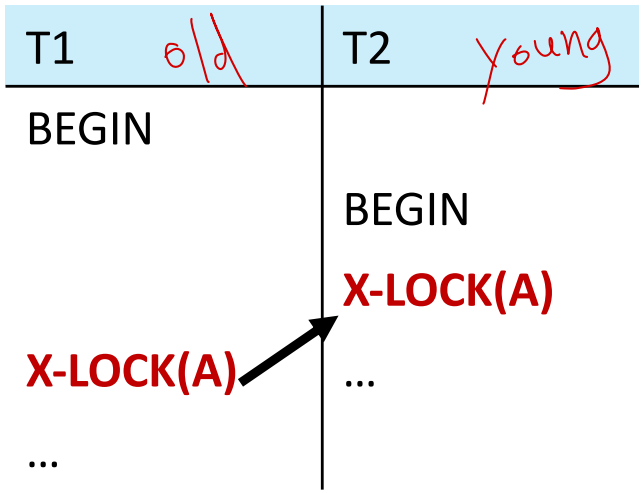
  → Older Timestamp = Higher Priority (e.g., $T_1 > T_2$)

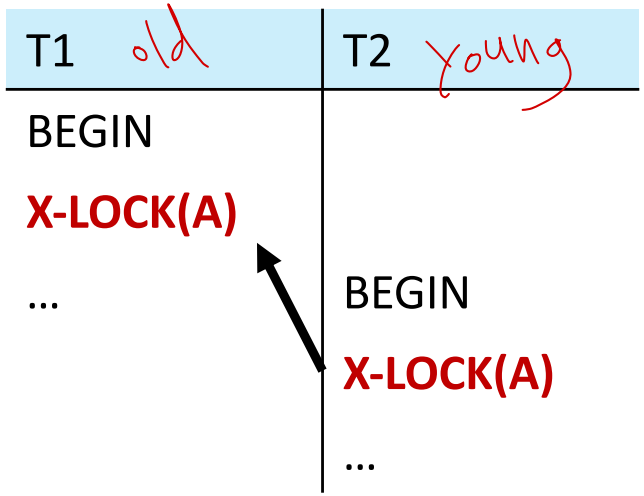
- **Wound-Wait** ("Young Waits for Old")

  → If requesting txn has higher priority than holding txn, then holding txn aborts and releases lock.

  → Otherwise requesting txn waits.

Deadlock Prevention





Wait-Die	Wound-Wait
T1 waits	T2 aborts

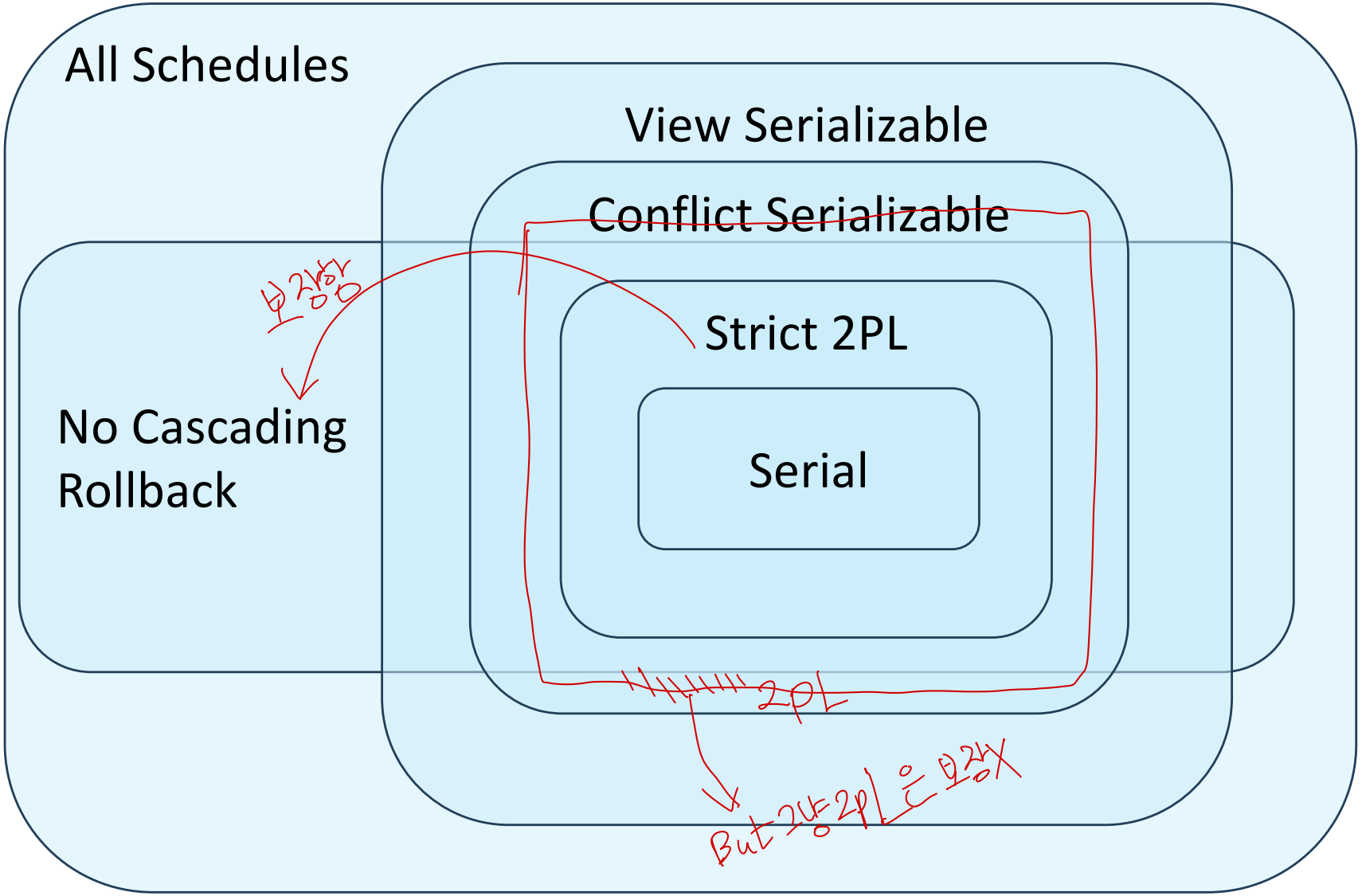
Wait-Die	Wound-Wait
T2 aborts	T2 waits

Deadlock Prevention

- Why do these schemes guarantee no deadlocks?
  - Only one "type" of direction allowed when waiting for a lock.

- When a txn restarts, what is its (new) priority?
  - Its original timestamp to prevent it from getting starved for resources.

Universe of Schedules



Lock Granularities

세분화 (범위)

- When a txn wants to acquire a "lock", the DBMS can decide the granularity (i.e., scope) of that lock.
  → Attribute? Tuple? Page? Table?

- The DBMS should ideally obtain fewest number of locks that a txn needs.

• coarse-grained → 테이블기준 lock (범위가큼)
< 범위가 큰 : 동시성이 작아질 수 있다 (overhead ↓)
범위가 작은 : 동시성이 높아지고, 성능이 좋아진다 (overhead ↑)
관리하는 cost

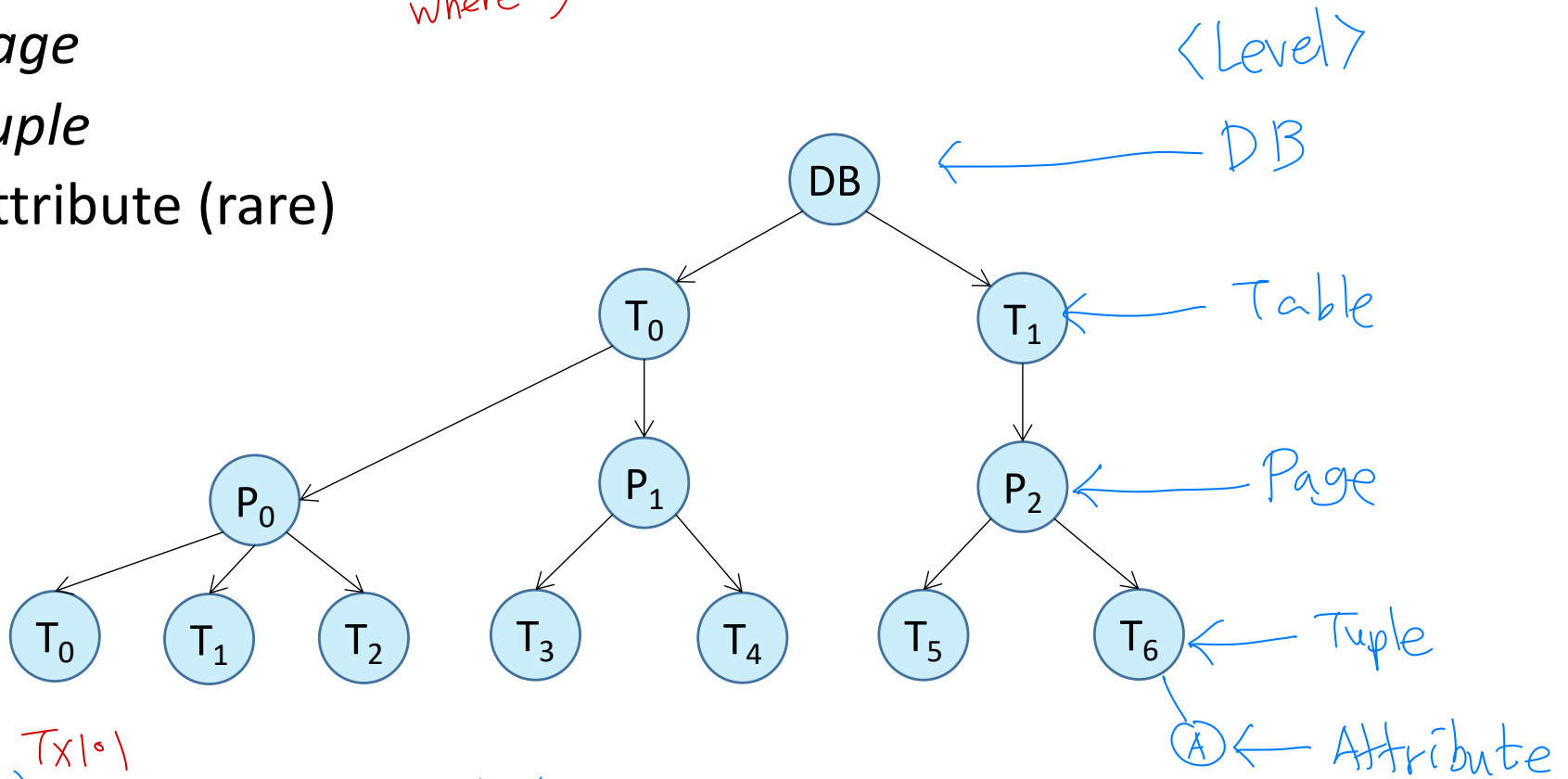
- Trade-off between Concurrency versus Overhead.
  → Fewer Locks, Larger Granularity vs. More Locks, Smaller Granularity.

Database Lock Hierarchy

The levels, starting from the top level are

- *Database*
- *Table*
- *Page*
- *Tuple*
- Attribute (rare)

update from where) → 이 구조는 SIX가 요구됨

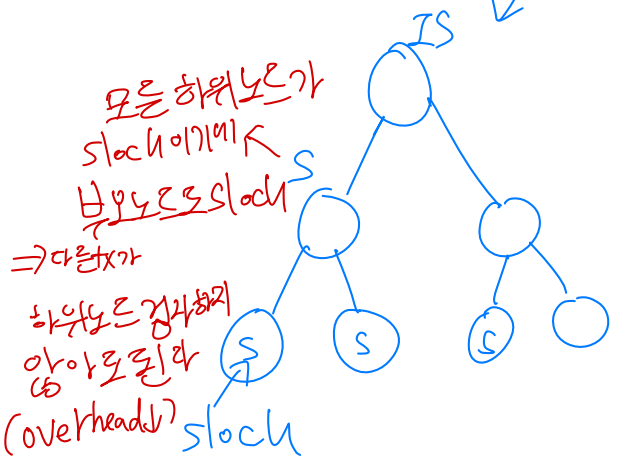


ex) Tx1이 T_3을 업데이트 → T_3: Xlock ⇒ 이상황에서 Tx2가 T_3, T_4을 읽기요청
⇒ P_1, T_0, DB 모두 IX lock 걸림 ⇒ DB, T_0: IS, P_1: S 가되므로 IX vs S는 충돌!
∴ Page 단계에서 false 리턴 ⇒ 하위노드를 방문하지않고 종료됨

Intention Lock Modes

- 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 locked in an intention mode, then some txn is doing explicit locking at a lower level in the tree



Intention Lock Modes

- ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with ***shared*** locks. ⇒ 하위구조에 slock을 가진 tx가 존재한다.
- ***intention-exclusive*** (IX): indicates explicit locking at a lower level with ***exclusive*** locks ⇒ xlock가진 tx존재
- ***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. ⇒ 하위 객체가 모두 slock → 그중에 몇몇이 xlock인 경우

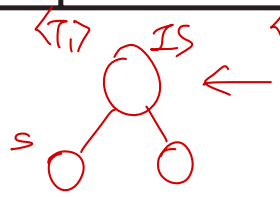
Compatibility Matrix with Intention Lock Modes

- 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

= comparable

흥미로운 관계
1) S - IX (SIX)
⇒ False
왜? S는 모든 하위노드 slock
하지만 IX는 1개라도 조건
X여야해서 충돌!
2) IX - IX (위노드단계)
원소 나뉘어갈수도 있음
→ 그게 아니더라도 하위노드에서
처리하게끔 선점X → True



T2
IS : slock 끼리는 괜찮다 → True
IX : 오른쪽에 Xlock을 얻으면된다. 혹은 왼쪽이어도 위노드단계에서는 괜찮다 → True
S : 괜찮다 → True
SIX : X와 비슷한 이유 → True
X : X는 하위모든 노드가 Xlock을 가져야하는데 이미 왼쪽노드에 slock을 가지므로 충돌 → True

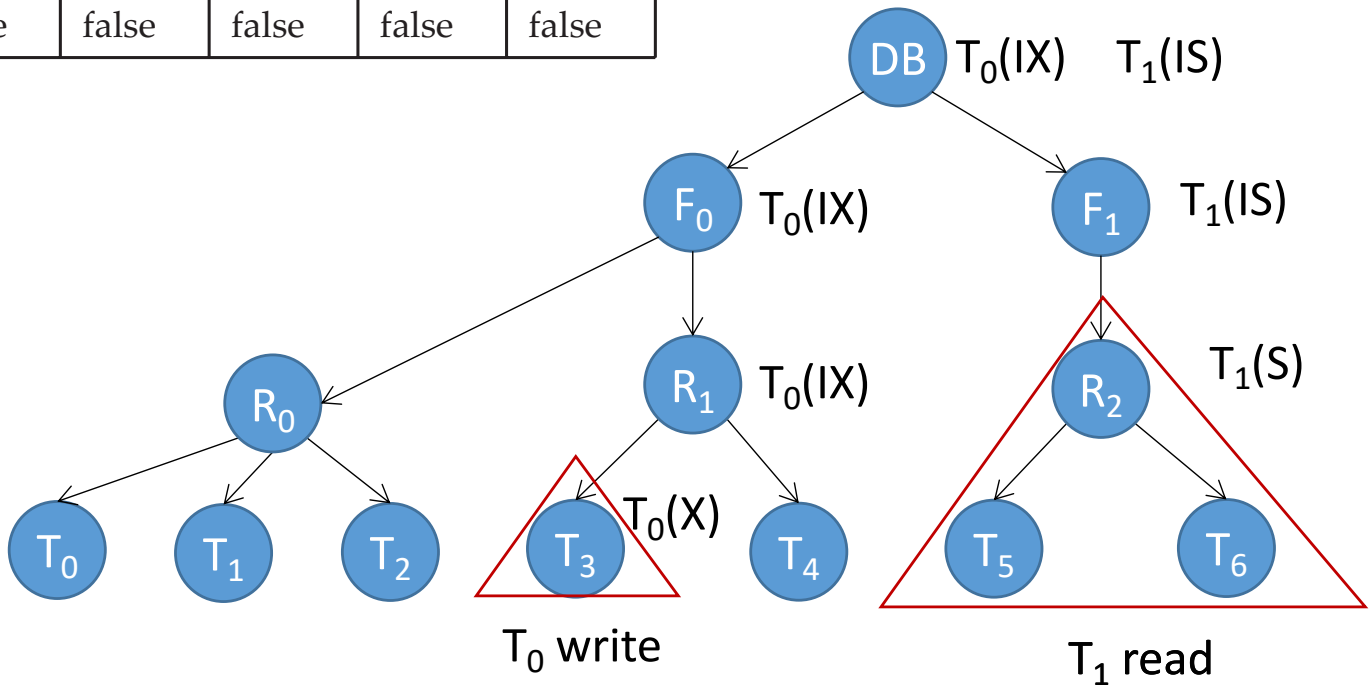
Multiple Granularity Locking Scheme

- Each txn 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.
- To get X, IX, or SIX on a node, must hold at least IX on parent node.

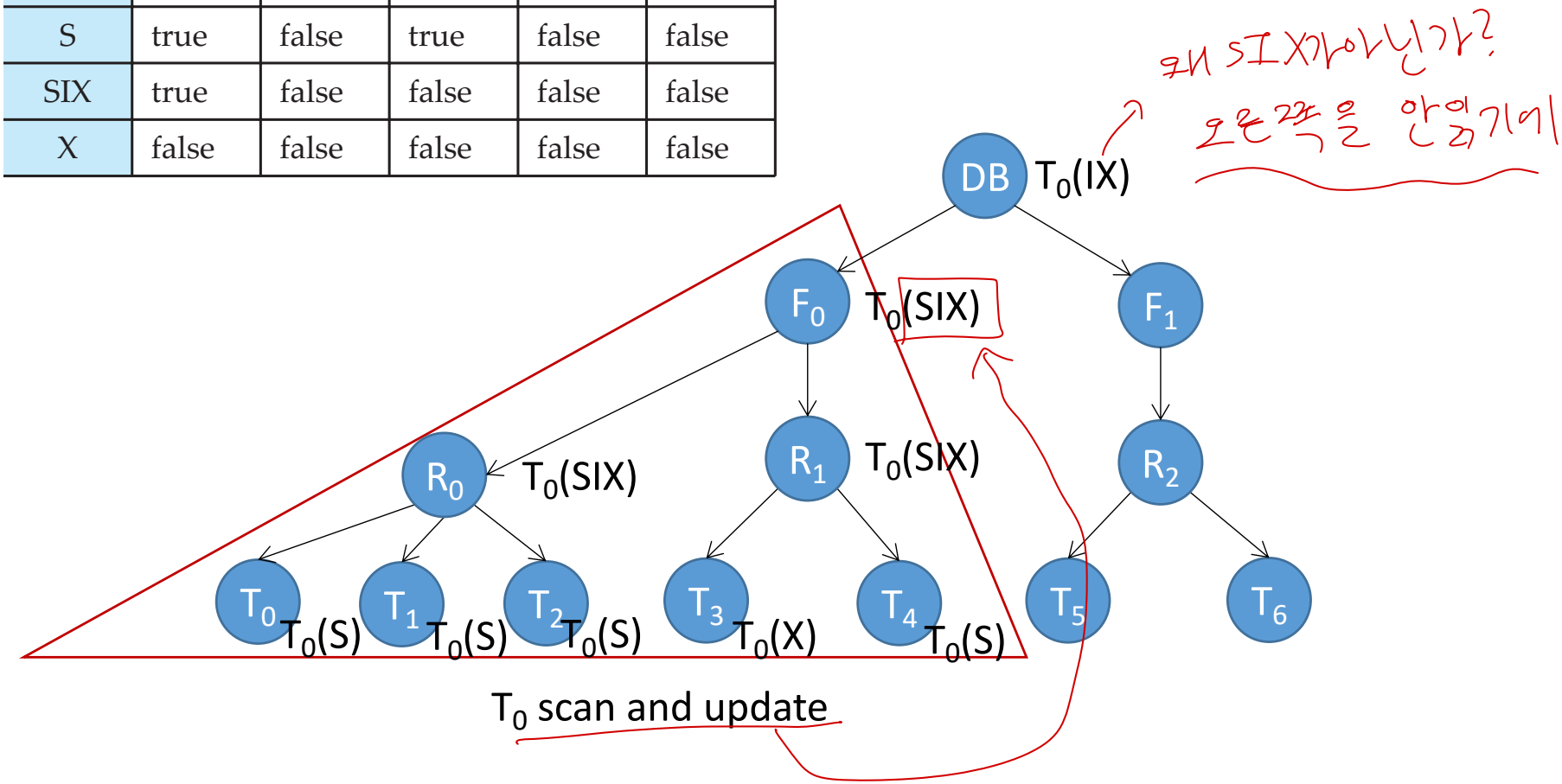
MGL Example

	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



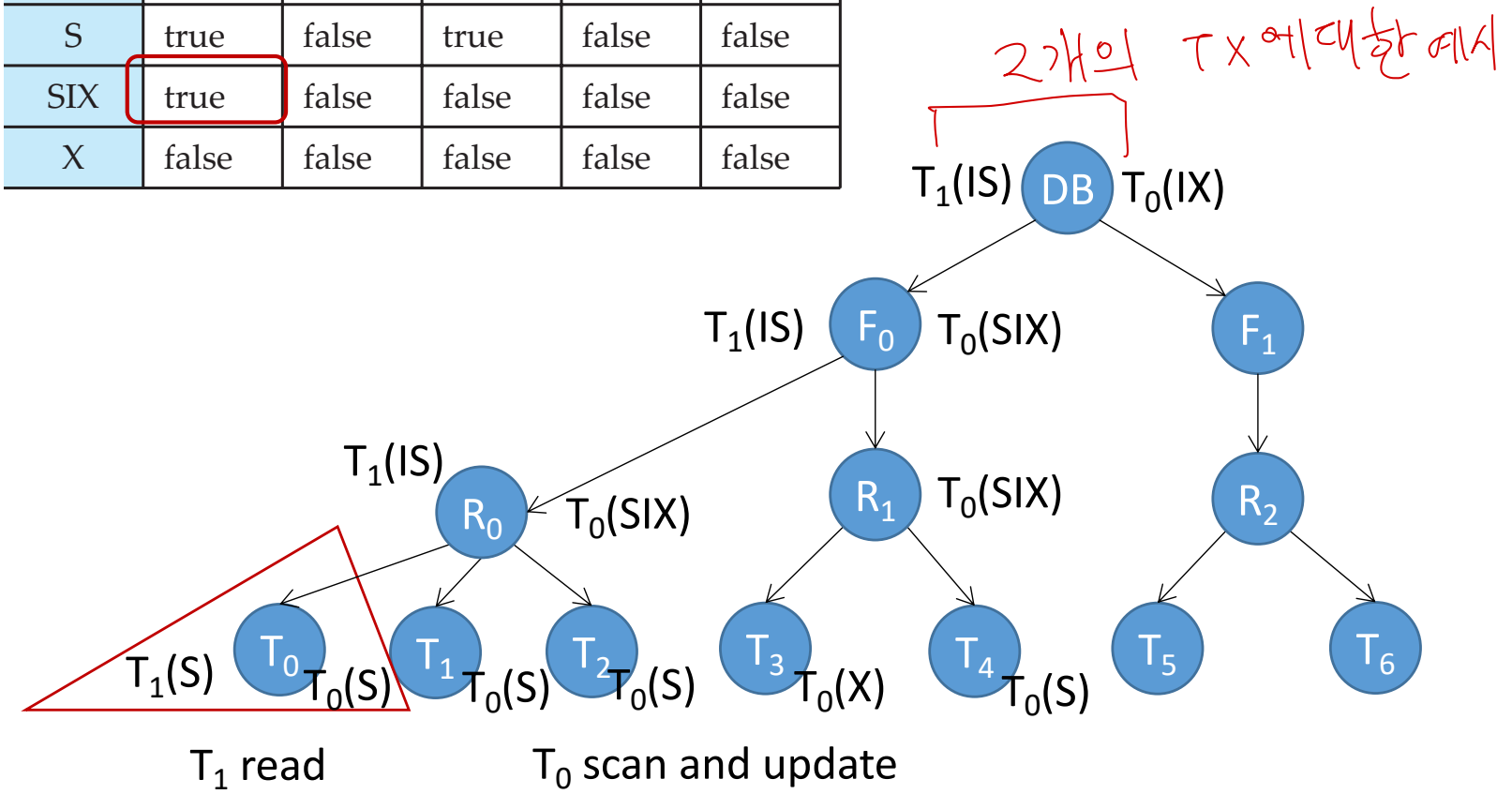
MGL Example

	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



MGL Example

	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

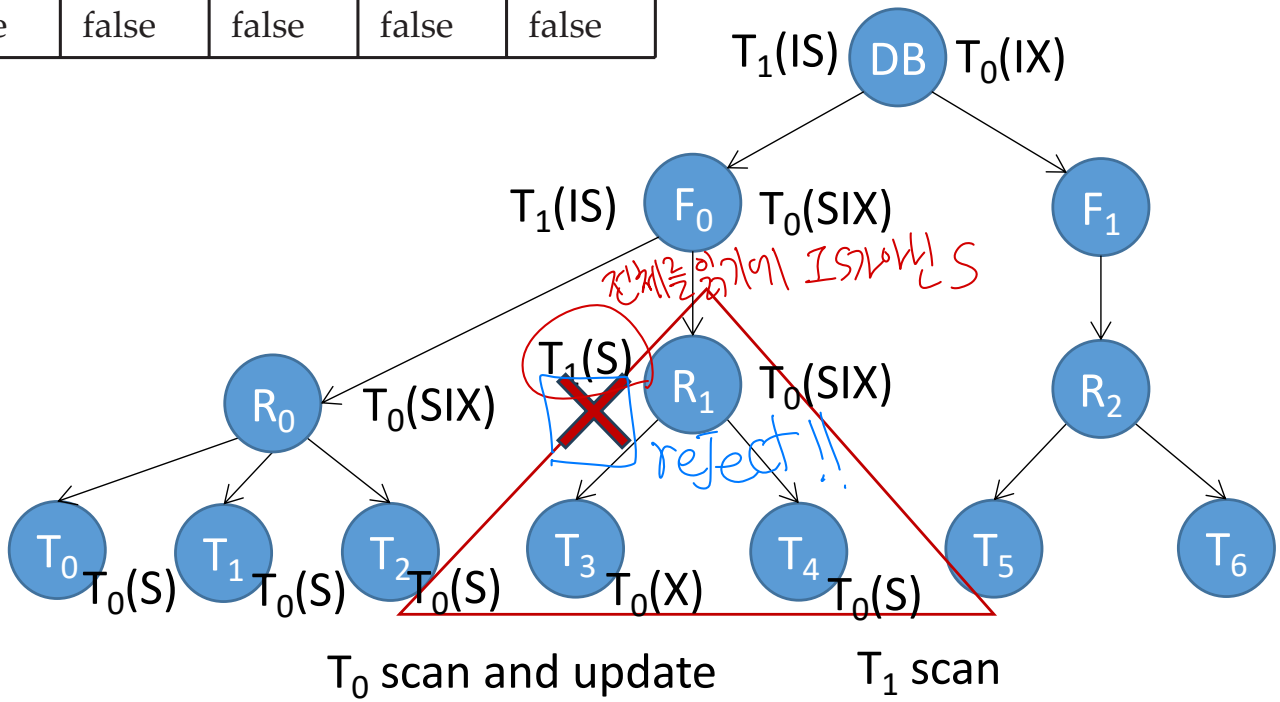


MGL Example

	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

S : entire access

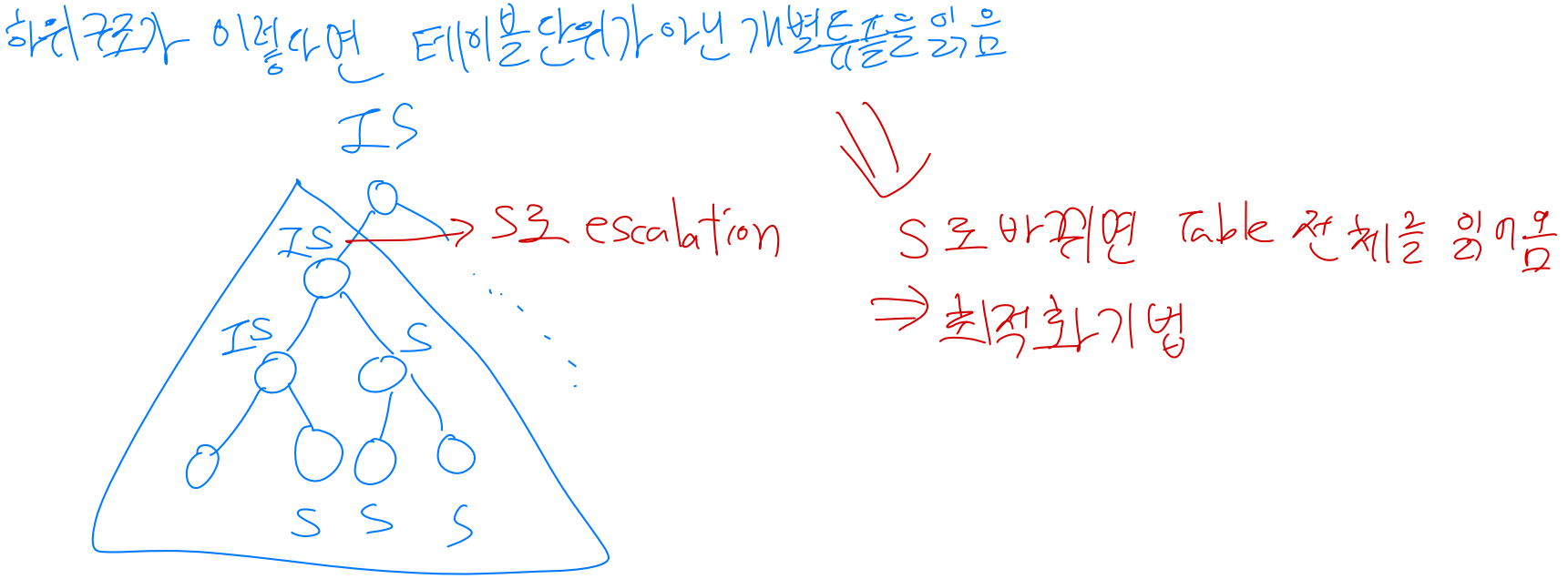
IS : partial access (X로 마찬가지)



Lock Escalation

- The DBMS can automatically switch to coarser-grained locks when a txn acquires too many low-level locks.

- This reduces the number of requests that the lock manager must process.



Locking in Practice

- Applications typically do not acquire a txn's locks manually (i.e., explicit SQL commands). → DBMS가 알아서할테니 app 개발자는 신경 X

- Sometimes you need to provide the DBMS with hints to help it to improve concurrency.
  → Update a tuple after reading it.
  → Skip any tuple that is locked.

- Explicit locks are also useful when doing major changes to the database.

SELECT...FOR UPDATE

- Perform a SELECT and then sets an exclusive lock on the matching tuples.
- Can also set shared locks:
  - → Postgres: FOR SHARE
  - → MySQL: LOCK IN SHARE MODE

팬텀현상피하려 "for update" 키워드사용
↳ DBMS에게 팬텀현상 발생에 대한 힌트를 주는것