Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #17

Two-Phase Locking

SPRING 2024 ❯❯ Prof. Jignesh Patel

LAST CLASS

Conflict Serializable

→ Verify using either the "swapping" method or dependency graphs.

→ Any DBMS that says that they support "serializable" isolation does this.

View Serializable

→ No efficient way to verify.

→ No DBMS that supports this.

OBSERVATION

We need a way to guarantee that all execution schedules are correct (i.e., serializable) without knowing the entire schedule ahead of time.

Solution: Use **locks** to protect database objects.

EXECUTING WITH LOCKS

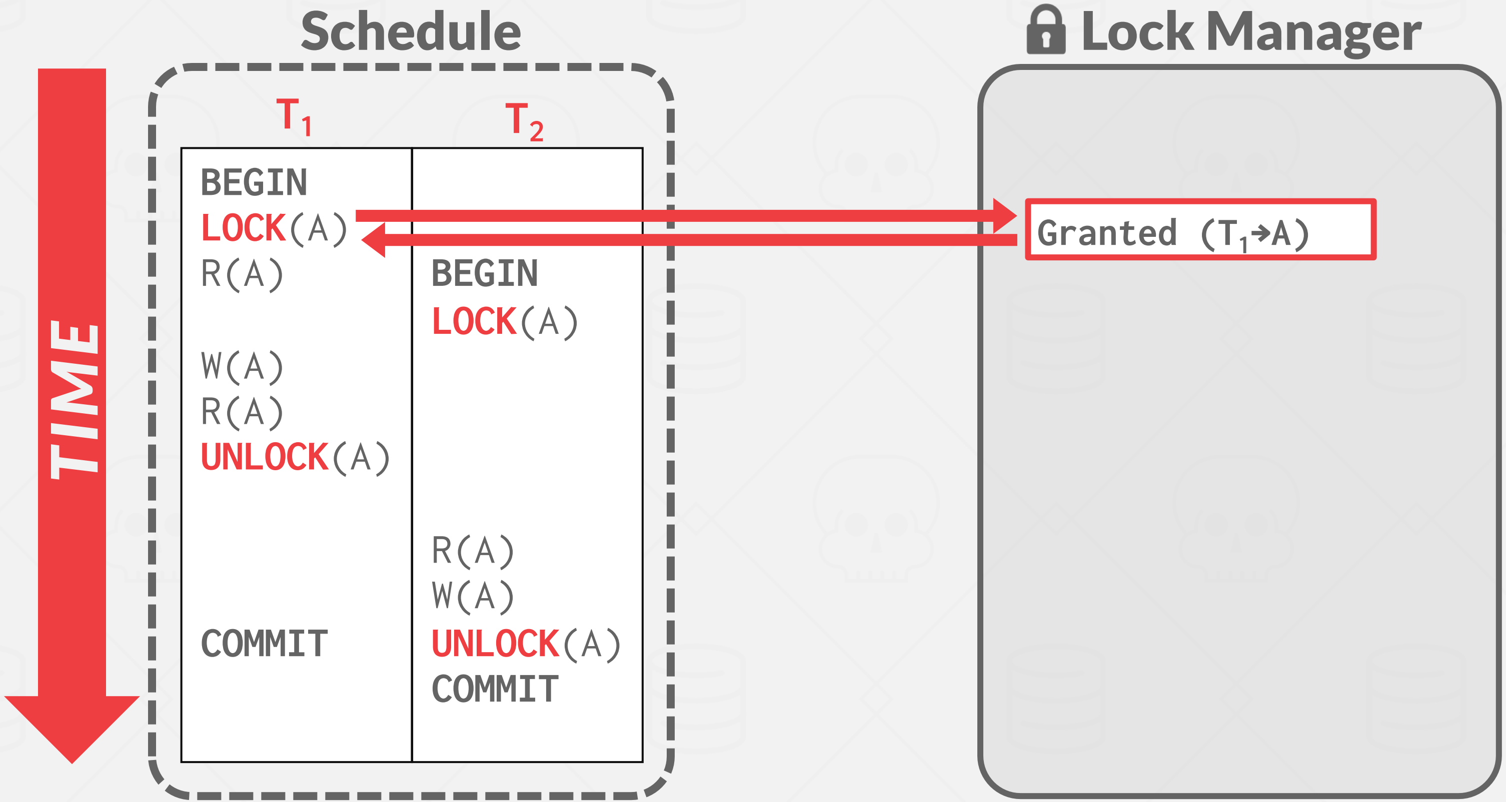
Schedule

TIME

T_1	T_2
BEGIN	
LOCK(A)	
R(A)	BEGIN
	LOCK(A)
W(A)	
R(A)	
UNLOCK(A)	
	R(A)
	W(A)
COMMIT	**UNLOCK**(A)
	COMMIT

Lock Manager

EXECUTING WITH LOCKS



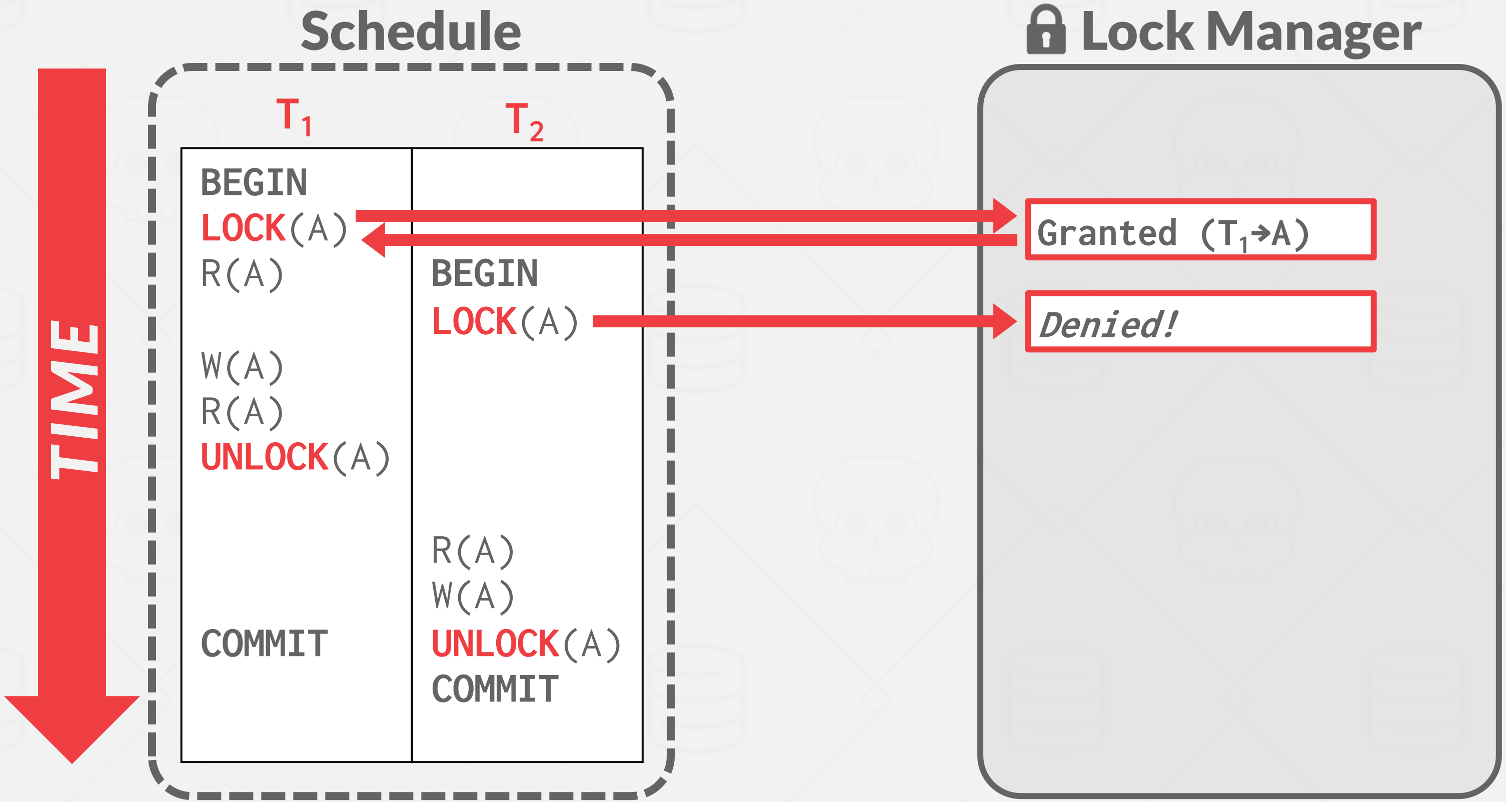
Schedule

T_1	T_2
BEGIN	
LOCK(A)	
R(A)	BEGIN
	LOCK(A)
W(A)	
R(A)	
UNLOCK(A)	
	R(A)
	W(A)
COMMIT	UNLOCK(A)
	COMMIT

Lock Manager

Granted (T_1→A)

EXECUTING WITH LOCKS



Schedule

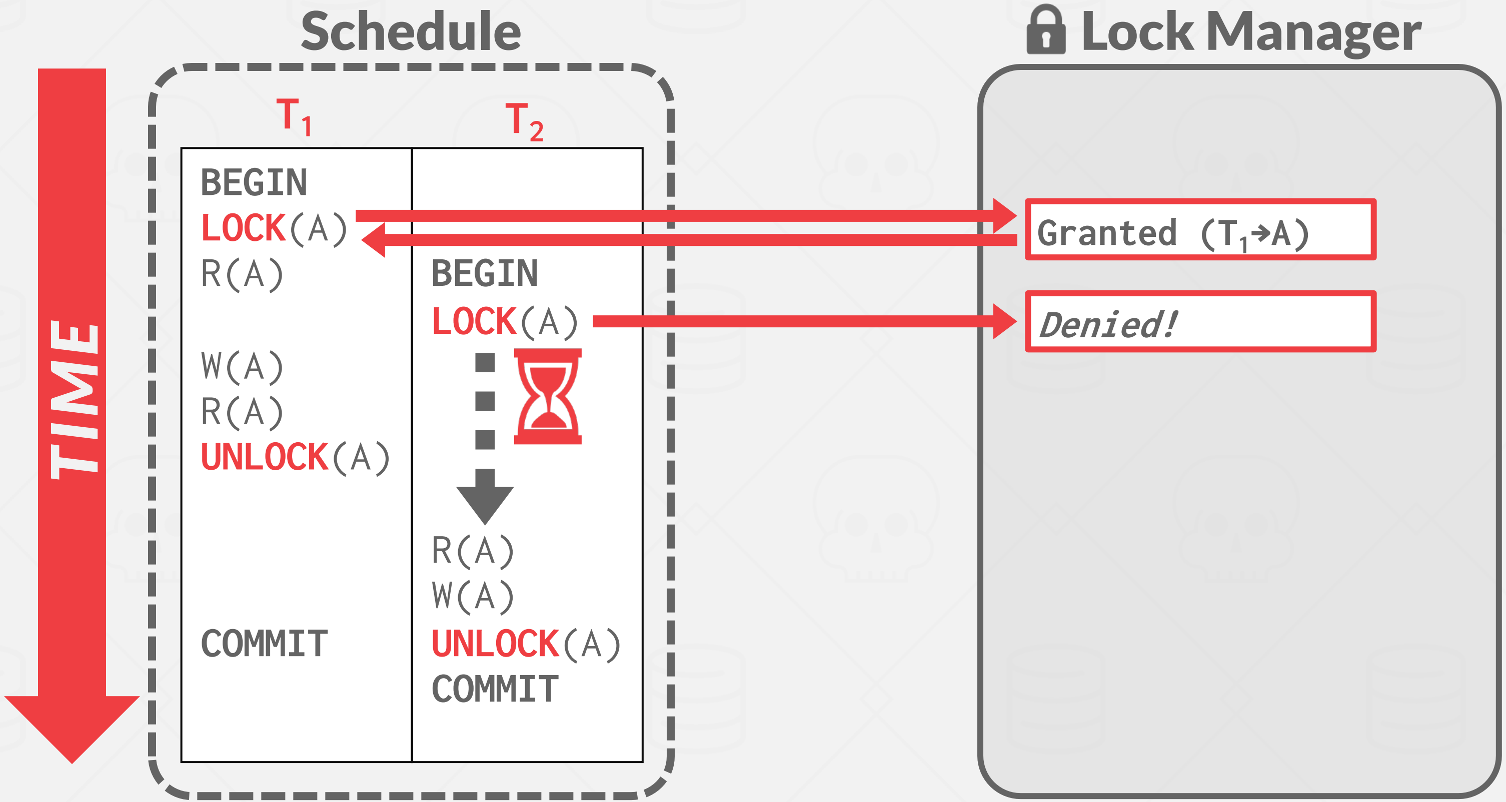
T_1	T_2
BEGIN	
LOCK(A)	
R(A)	BEGIN
	LOCK(A)
W(A)	
R(A)	
UNLOCK(A)	
	R(A)
	W(A)
COMMIT	**UNLOCK**(A)
	COMMIT

Lock Manager

Granted (T_1→A)

Denied!

EXECUTING WITH LOCKS



Schedule

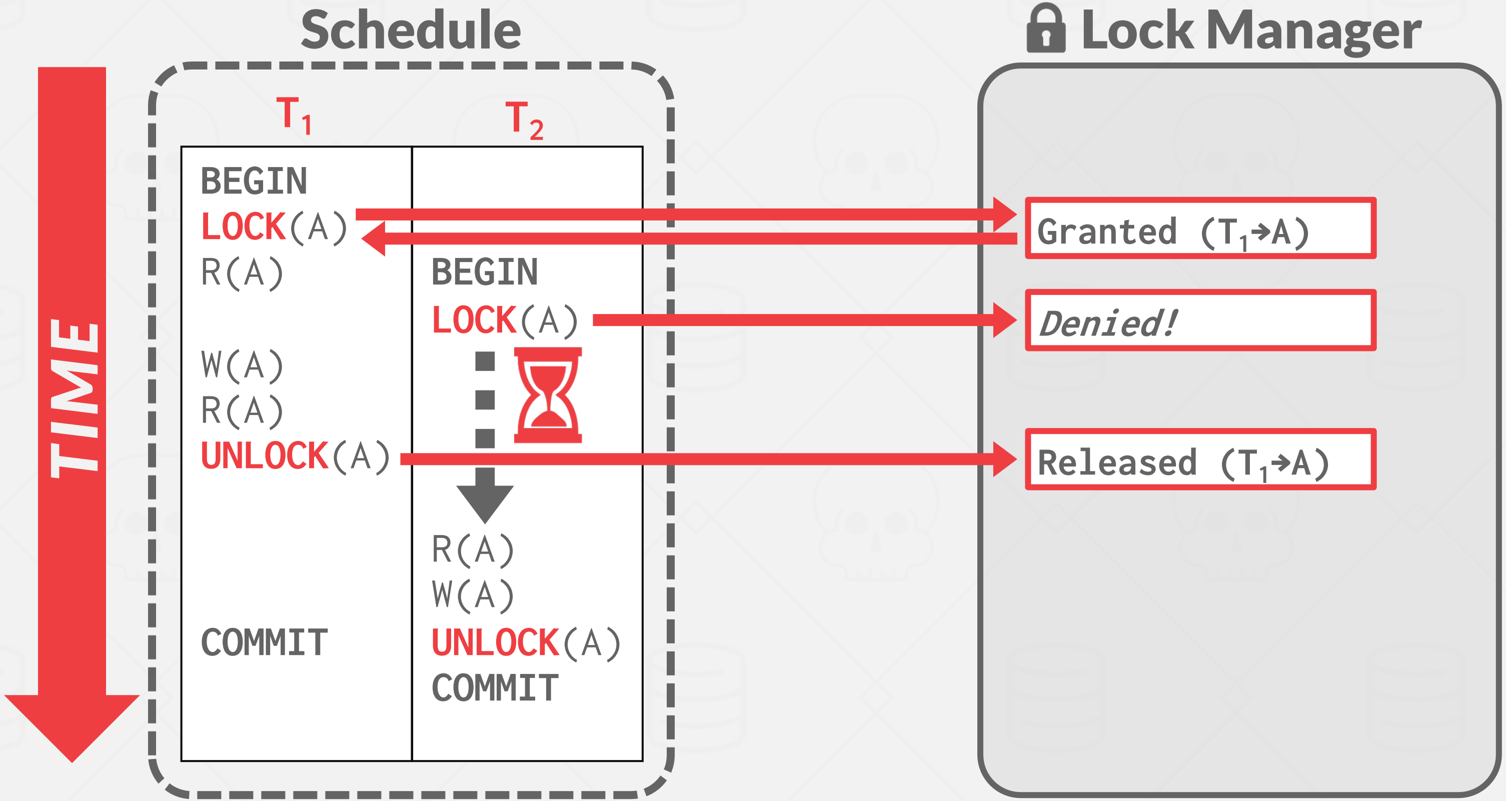
T_1	T_2
BEGIN	
LOCK(A)	
R(A)	BEGIN
	LOCK(A)
W(A)	
R(A)	
UNLOCK(A)	
	R(A)
	W(A)
COMMIT	UNLOCK(A)
	COMMIT

Lock Manager

Granted (T_1→A)

Denied!

EXECUTING WITH LOCKS



Schedule

T_1	T_2
BEGIN	
LOCK(A)	
R(A)	BEGIN
	LOCK(A)
W(A)	
R(A)	
UNLOCK(A)	
	R(A)
	W(A)
COMMIT	**UNLOCK**(A)
	COMMIT

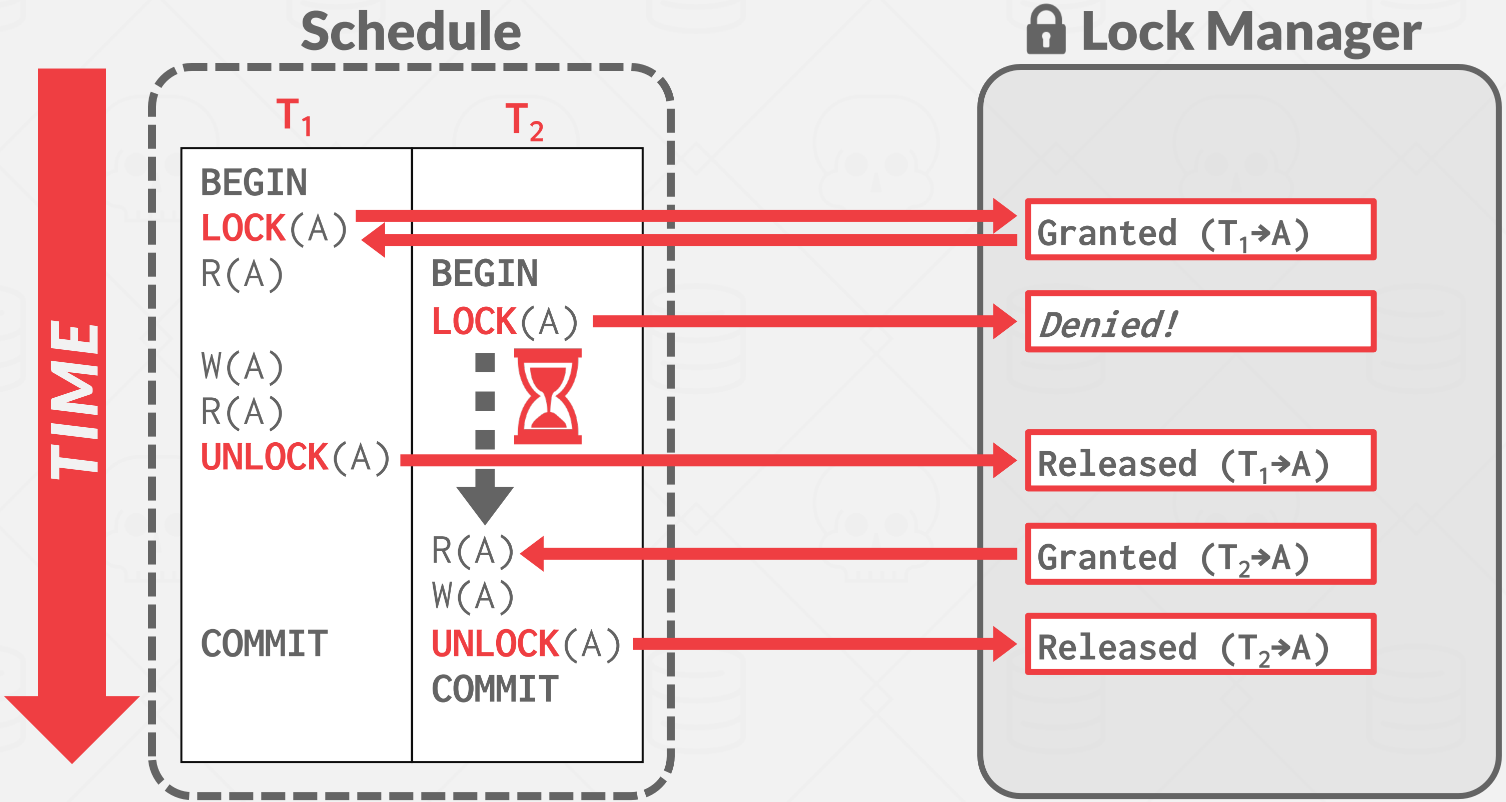
Lock Manager

Granted (T_1→A)

Denied!

Released (T_1→A)

EXECUTING WITH LOCKS



Schedule

T_1	T_2
BEGIN	
LOCK(A)	
R(A)	BEGIN
	LOCK(A)
W(A)	
R(A)	
UNLOCK(A)	
	R(A)
	W(A)
COMMIT	UNLOCK(A)
	COMMIT

Lock Manager

Granted (T_1→A)

Denied!

Released (T_1→A)

Granted (T_2→A)

Released (T_2→A)

TODAY'S AGENDA

Lock Types

Two-Phase Locking

Deadlock Detection + Prevention

Hierarchical Locking

LOCKS VS. LATCHES

	Locks	*Latches*
Separate…	User transactions	Threads
Protect…	Database Contents	In-Memory Data Structures
During…	Entire Transactions	Critical Sections
Modes…	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
…by…	Waits-for, Timeout, Aborts	Coding Discipline
Kept in…	Lock Manager	Protected Data Structure

Source: Goetz Graefe

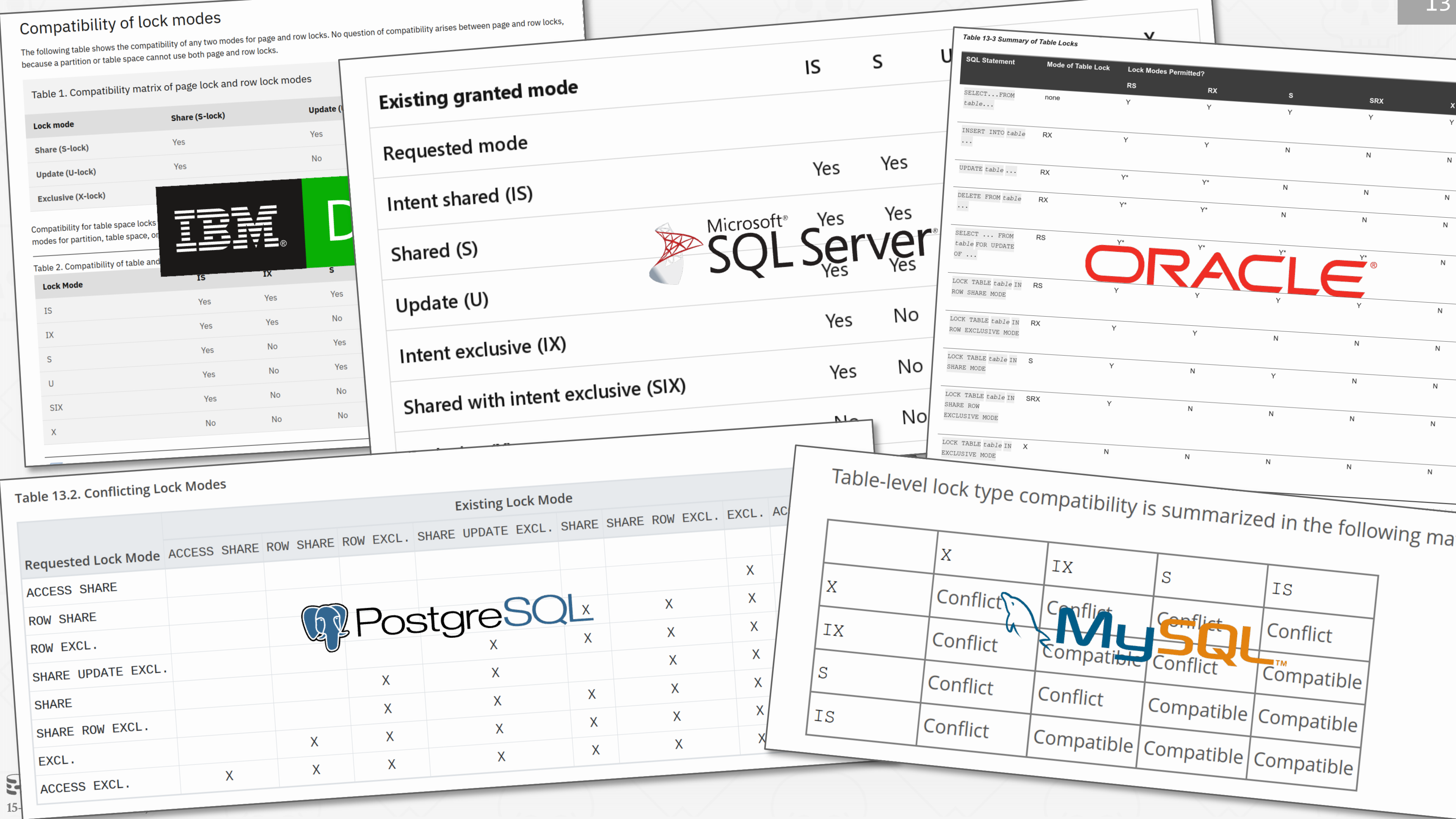
BASIC LOCK TYPES

S-LOCK: Shared locks for reads.

X-LOCK: Exclusive locks for writes.

Compatibility Matrix

	Shared	Exclusive
Shared	✓	X
Exclusive	X	X



Compatibility of lock modes

The following table shows the compatibility of any two modes for page and row locks. No question of compatibility arises between page and row locks, because a partition or table space cannot use both page and row locks.

Table 1. Compatibility matrix of page lock and row lock modes

Lock mode	Share (S-lock)	Update (
Share (S-lock)	Yes	Yes
Update (U-lock)	Yes	No
Exclusive (X-lock)		

Compatibility for table space locks
modes for partition, table space, or

Table 2. Compatibility of table and

Lock Mode	IS	IX	S
IS	Yes	Yes	Yes
IX	Yes	Yes	No
S	Yes	No	Yes
U	Yes	No	Yes
SIX	Yes	No	No
X	No	No	No

Existing granted mode	IS	S	U
Requested mode			
Intent shared (IS)	Yes	Yes	
Shared (S)	Yes	Yes	
Update (U)	Yes	Yes	
Intent exclusive (IX)	Yes	No	
Shared with intent exclusive (SIX)	Yes	No	
	No	No	

Table 13-3 Summary of Table Locks

SQL Statement	Mode of Table Lock	Lock Modes Permitted?				
		RS	RX	S	SRX	X
`SELECT...FROM table...`	none	Y	Y	Y	Y	Y
`INSERT INTO table ...`	RX	Y	Y	N	N	N
`UPDATE table ...`	RX	Y*	Y*	N	N	N
`DELETE FROM table ...`	RX	Y*	Y*	N	N	N
`SELECT ... FROM table FOR UPDATE OF ...`	RS	Y*	Y*	Y*	Y*	N
`LOCK TABLE table IN ROW SHARE MODE`	RS	Y	Y	Y	Y	N
`LOCK TABLE table IN ROW EXCLUSIVE MODE`	RX	Y	Y	N	N	N
`LOCK TABLE table IN SHARE MODE`	S	Y	N	Y	N	N
`LOCK TABLE table IN SHARE ROW EXCLUSIVE MODE`	SRX	Y	N	N	N	N
`LOCK TABLE table IN EXCLUSIVE MODE`	X	N	N	N	N	N

Table 13.2. Conflicting Lock Modes

Requested Lock Mode	Existing Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCL.	SHARE UPDATE EXCL.	SHARE	SHARE ROW EXCL.	EXCL.	AC
ACCESS SHARE								
ROW SHARE							X	
ROW EXCL.					X	X	X	
SHARE UPDATE EXCL.				X	X	X	X	
SHARE			X	X		X	X	
SHARE ROW EXCL.			X	X	X	X	X	
EXCL.		X	X	X	X	X	X	
ACCESS EXCL.	X	X	X	X	X	X	X	

Table-level lock type compatibility is summarized in the following ma

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	Compatible	Conflict	Compatible
S	Conflict	Conflict	Compatible	Compatible
IS	Conflict	Compatible	Compatible	Compatible

EXECUTING WITH LOCKS

Transactions request locks (or upgrades).

Lock manager grants or blocks requests.

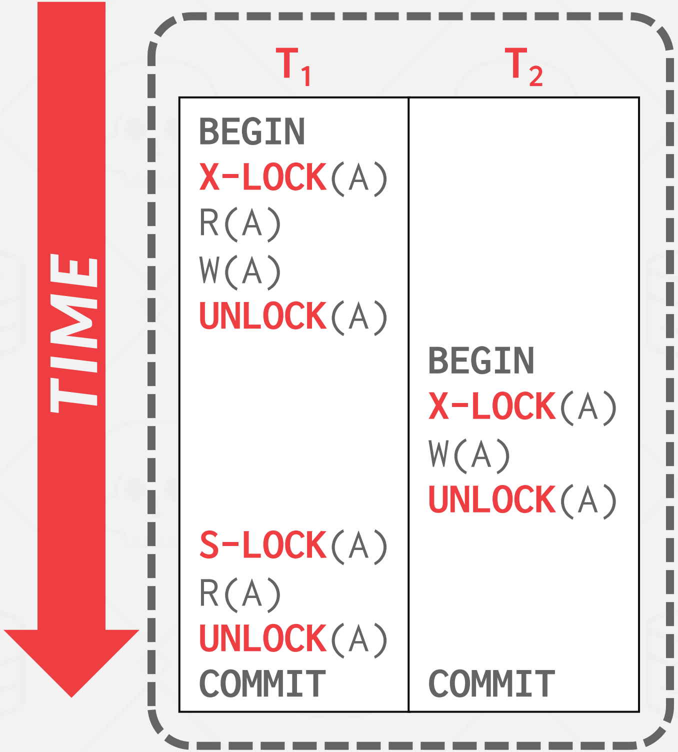
Transactions release locks.

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

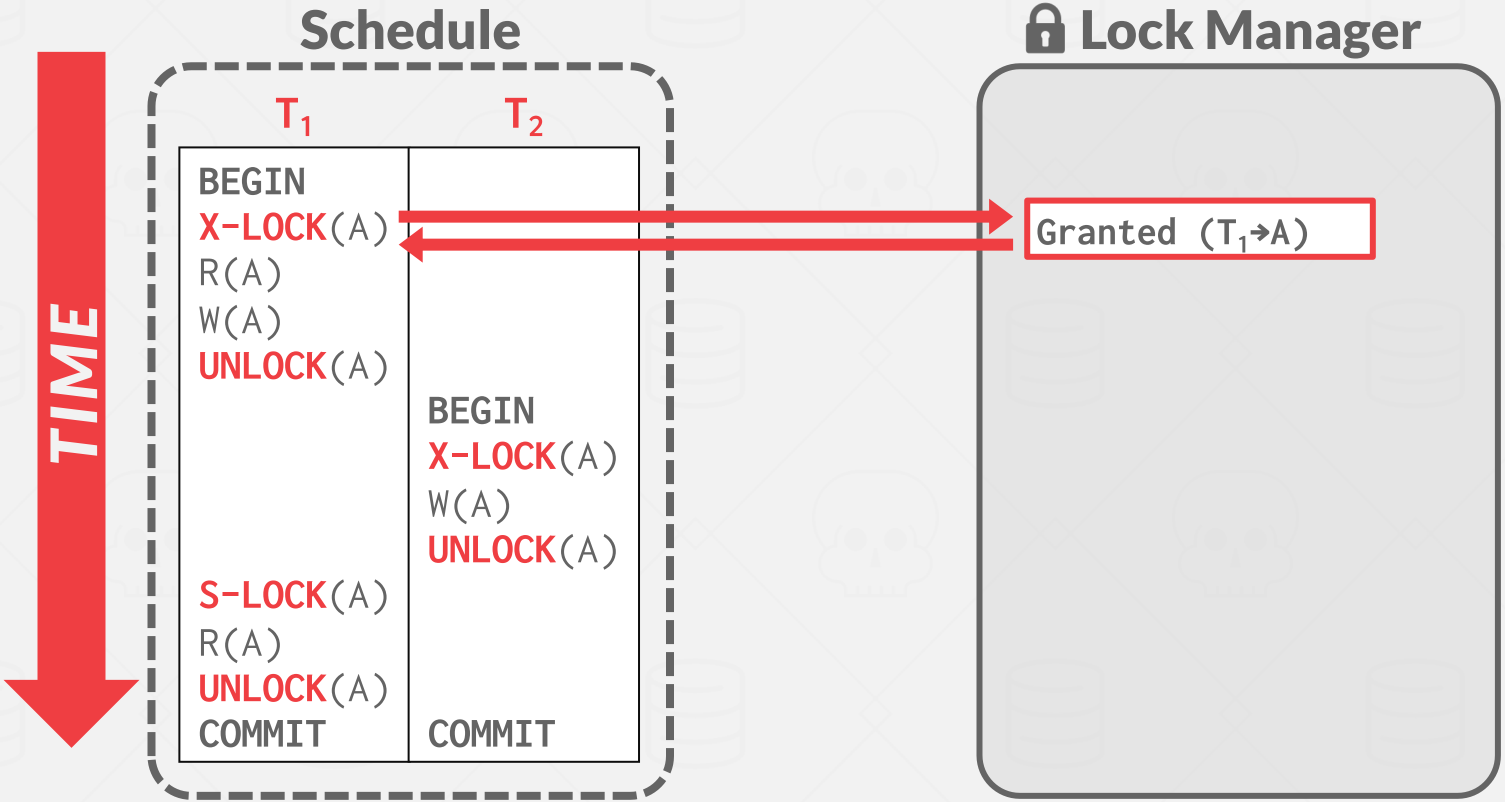
Schedule



T1	T2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

EXECUTING WITH LOCKS



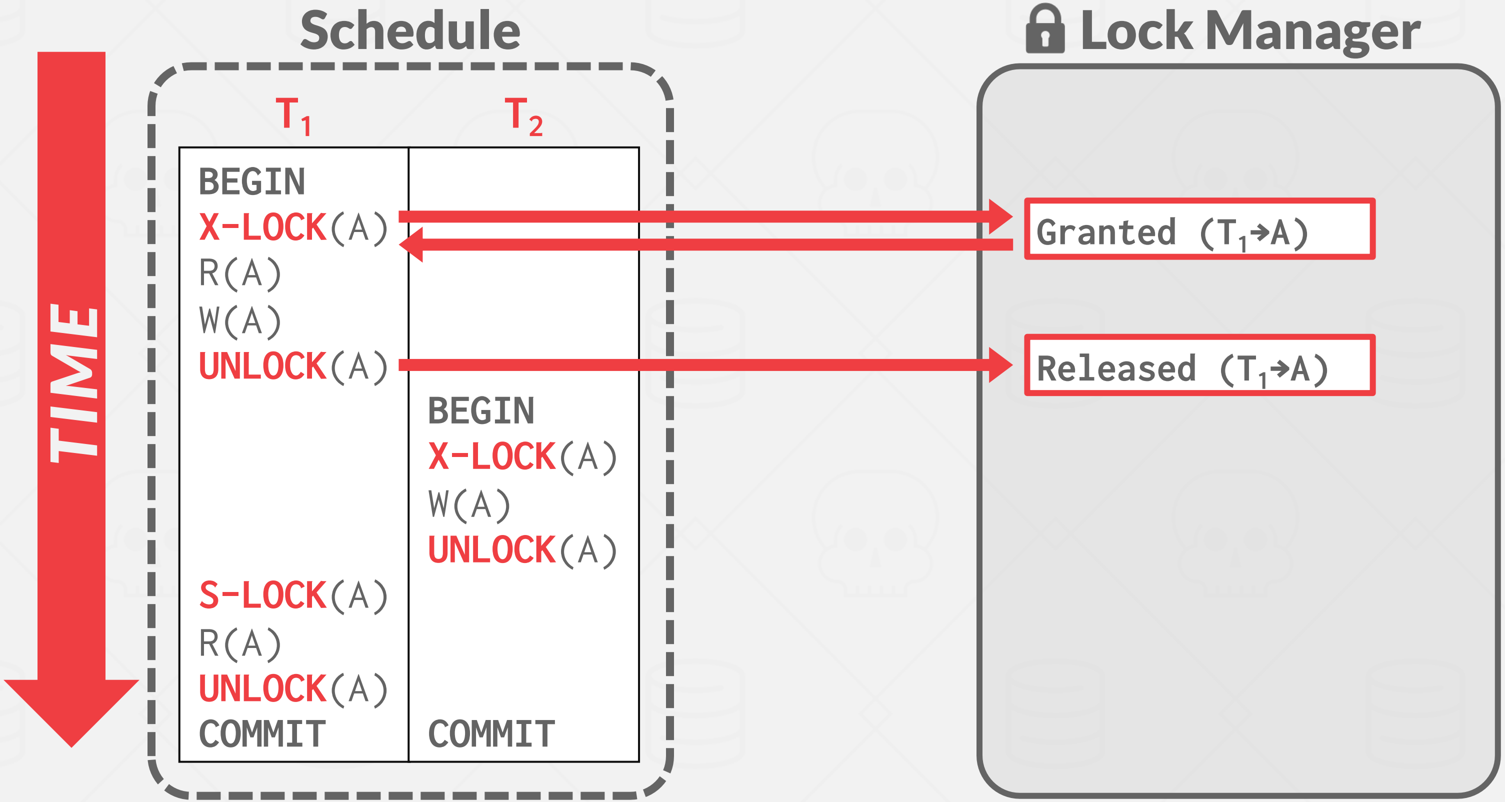
Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

Granted (T_1→A)

EXECUTING WITH LOCKS



Schedule

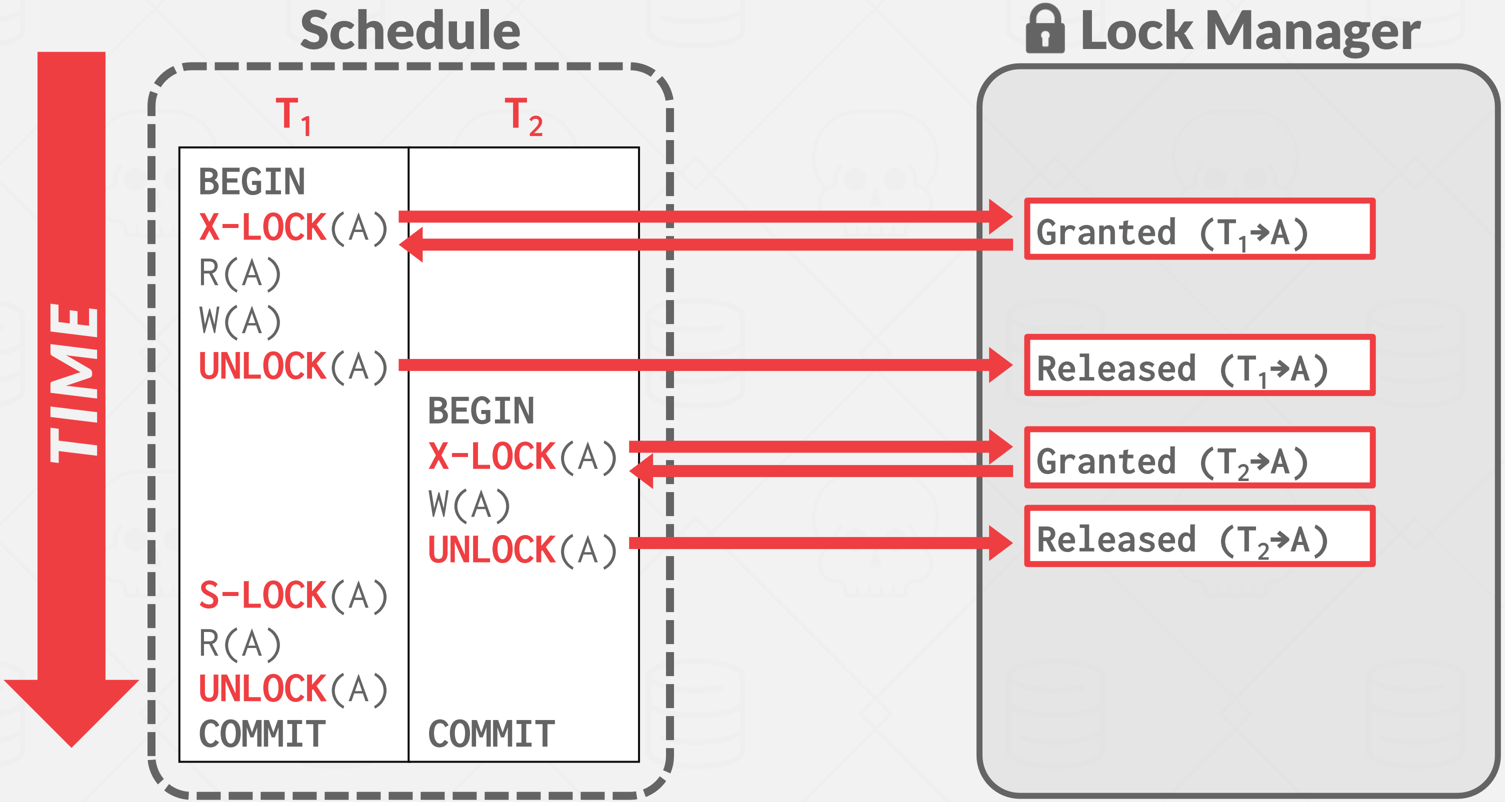
T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

Granted (T_1→A)

Released (T_1→A)

EXECUTING WITH LOCKS



Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

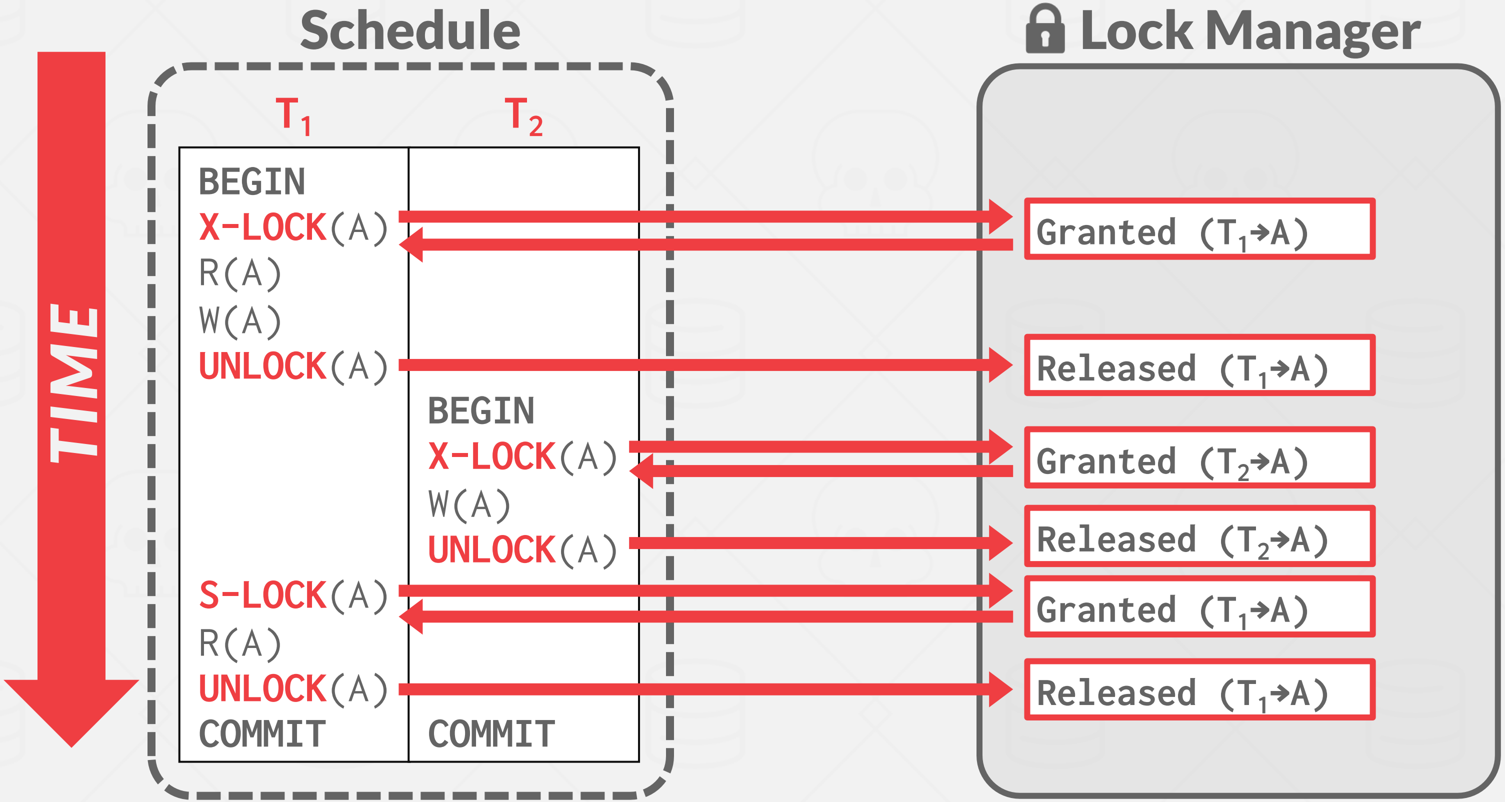
Granted (T_1→A)

Released (T_1→A)

Granted (T_2→A)

Released (T_2→A)

EXECUTING WITH LOCKS



Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

Granted (T_1→A)

Released (T_1→A)

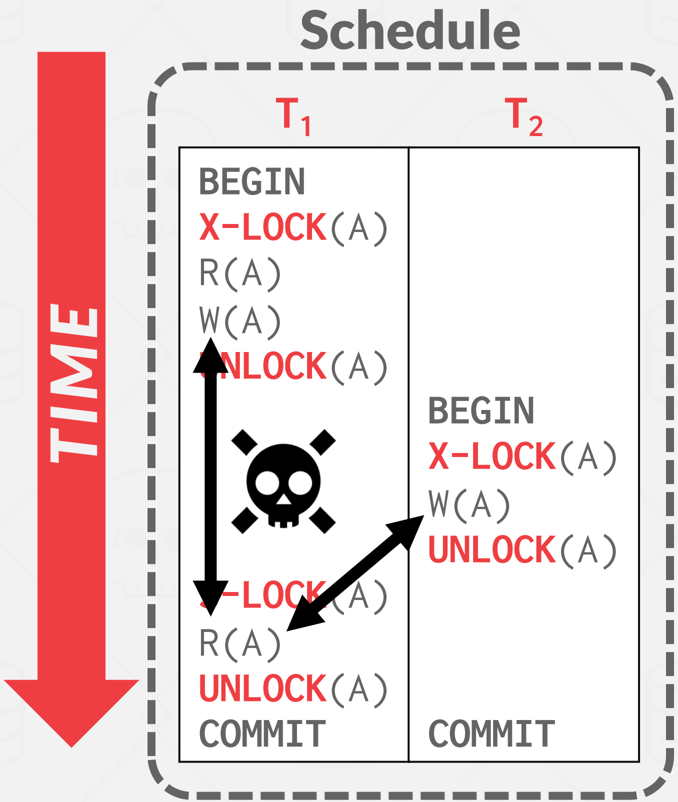
Granted (T_2→A)

Released (T_2→A)

Granted (T_1→A)

Released (T_1→A)

EXECUTING WITH LOCKS



Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
X-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

- Granted (T_1→A)
- Released (T_1→A)
- Granted (T_2→A)
- Released (T_2→A)
- Granted (T_1→A)
- Released (T_1→A)

CONCURRENCY CONTROL PROTOCOL

Two-phase locking (2PL) is a concurrency control protocol that determines whether a txn can access an object in the database at runtime.

The protocol does not need to know all the queries that a txn will execute ahead of time.

TWO-PHASE LOCKING

Phase #1: Growing

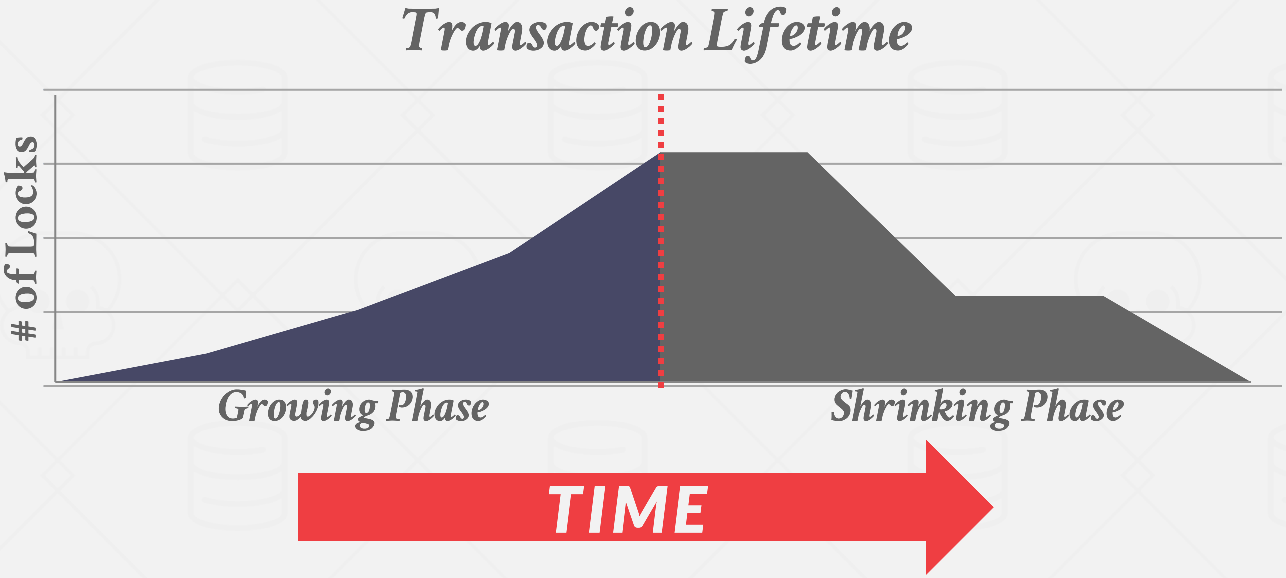
- → Each txn requests the locks that it needs from the DBMS's lock manager.
- → The lock manager grants/denies lock requests.

Phase #2: Shrinking

- → The txn is allowed to only release/downgrade locks that it previously acquired. It cannot acquire new locks.

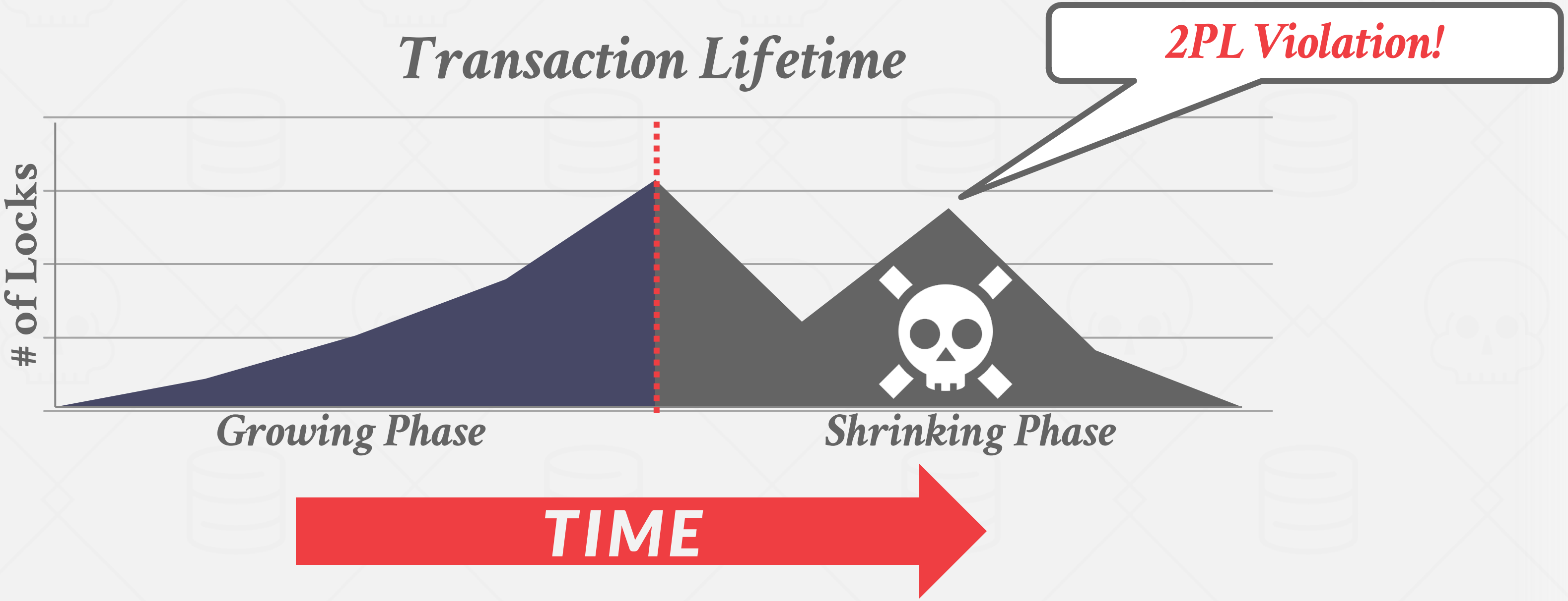
TWO-PHASE LOCKING

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.



TWO-PHASE LOCKING

The txn is not allowed to acquire/upgrade locks after the growing phase finishes.



EXECUTING WITH 2PL

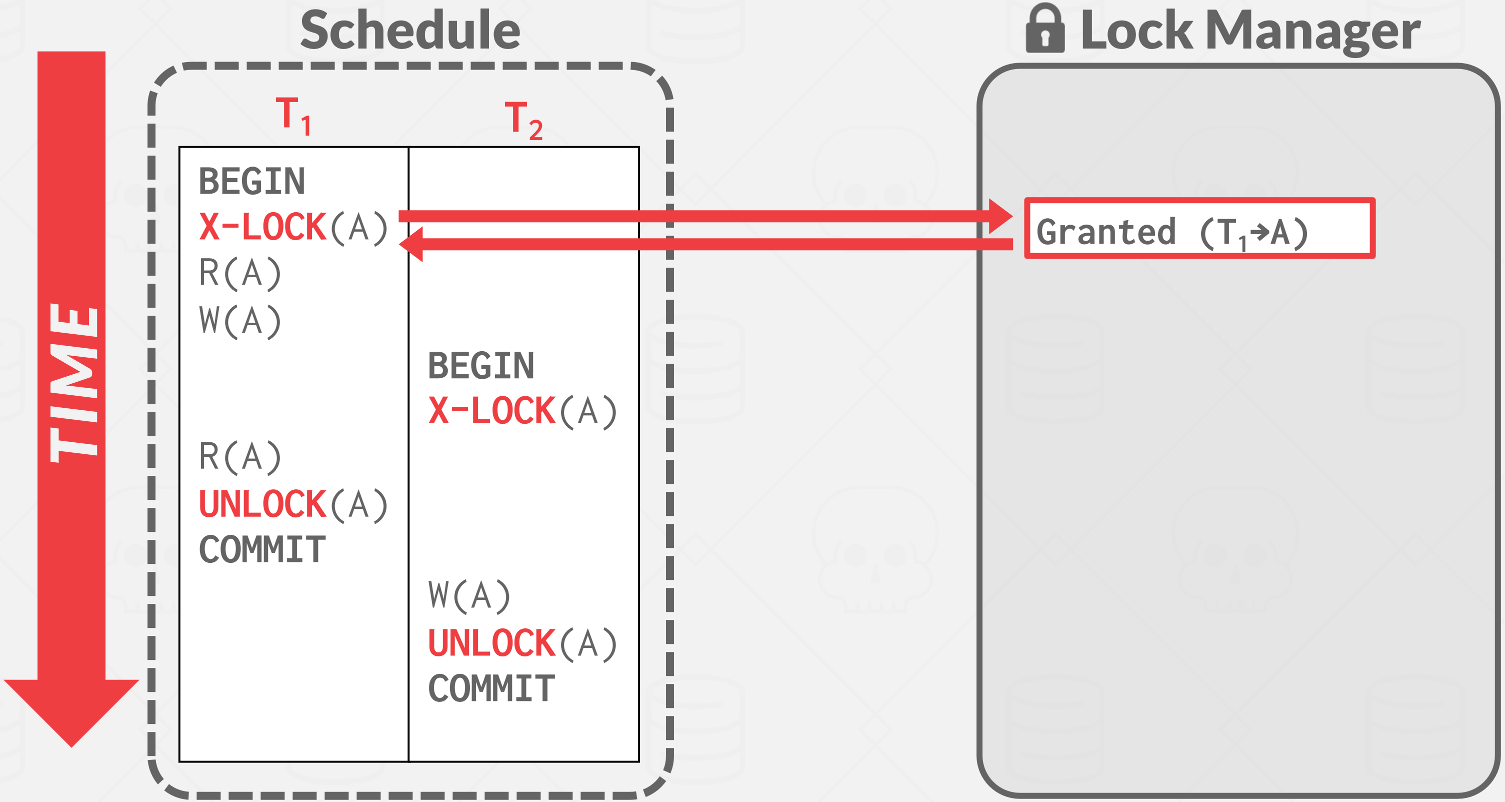
Schedule

TIME

T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
	BEGIN
	X-LOCK(A)
R(A)	
UNLOCK(A)	
COMMIT	
	W(A)
	UNLOCK(A)
	COMMIT

Lock Manager

EXECUTING WITH 2PL



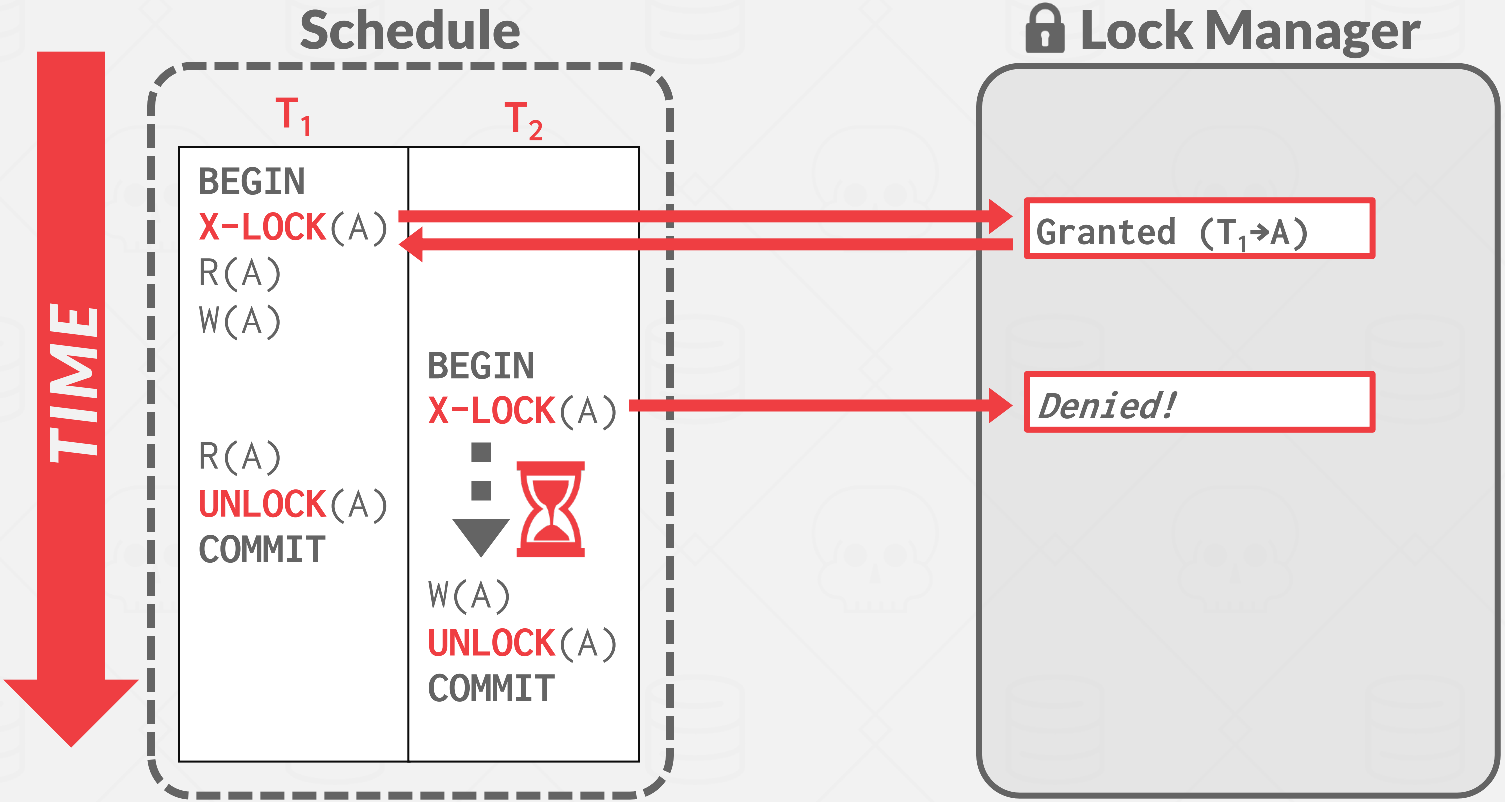
Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
	BEGIN
	X-LOCK(A)
R(A)	
UNLOCK(A)	
COMMIT	
	W(A)
	UNLOCK(A)
	COMMIT

Lock Manager

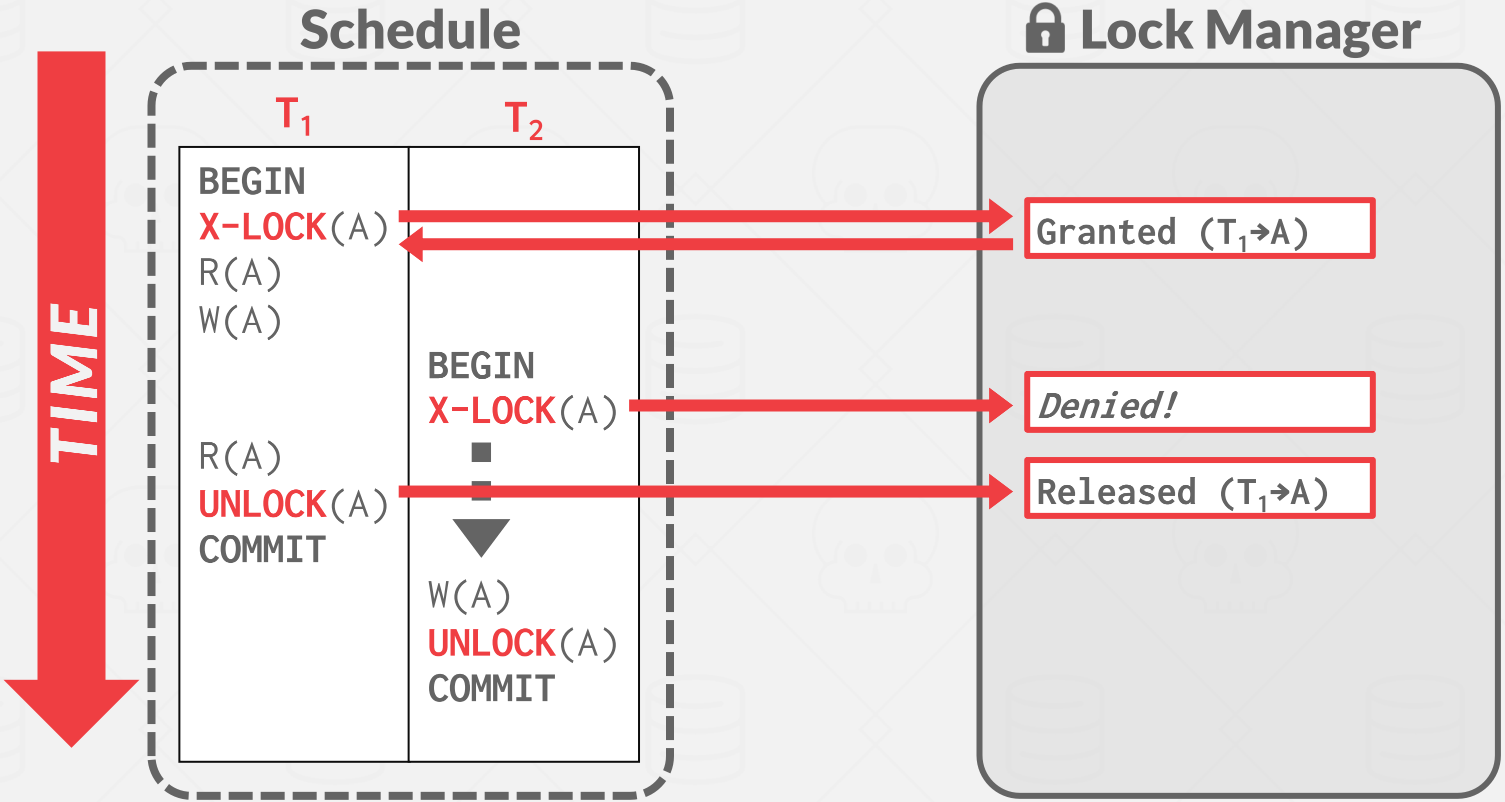
Granted (T_1→A)

EXECUTING WITH 2PL



T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
	BEGIN
	X-LOCK(A)
R(A)	
UNLOCK(A)	
COMMIT	
	W(A)
	UNLOCK(A)
	COMMIT

EXECUTING WITH 2PL



Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
	BEGIN
	X-LOCK(A)
R(A)	
UNLOCK(A)	
COMMIT	
	W(A)
	UNLOCK(A)
	COMMIT

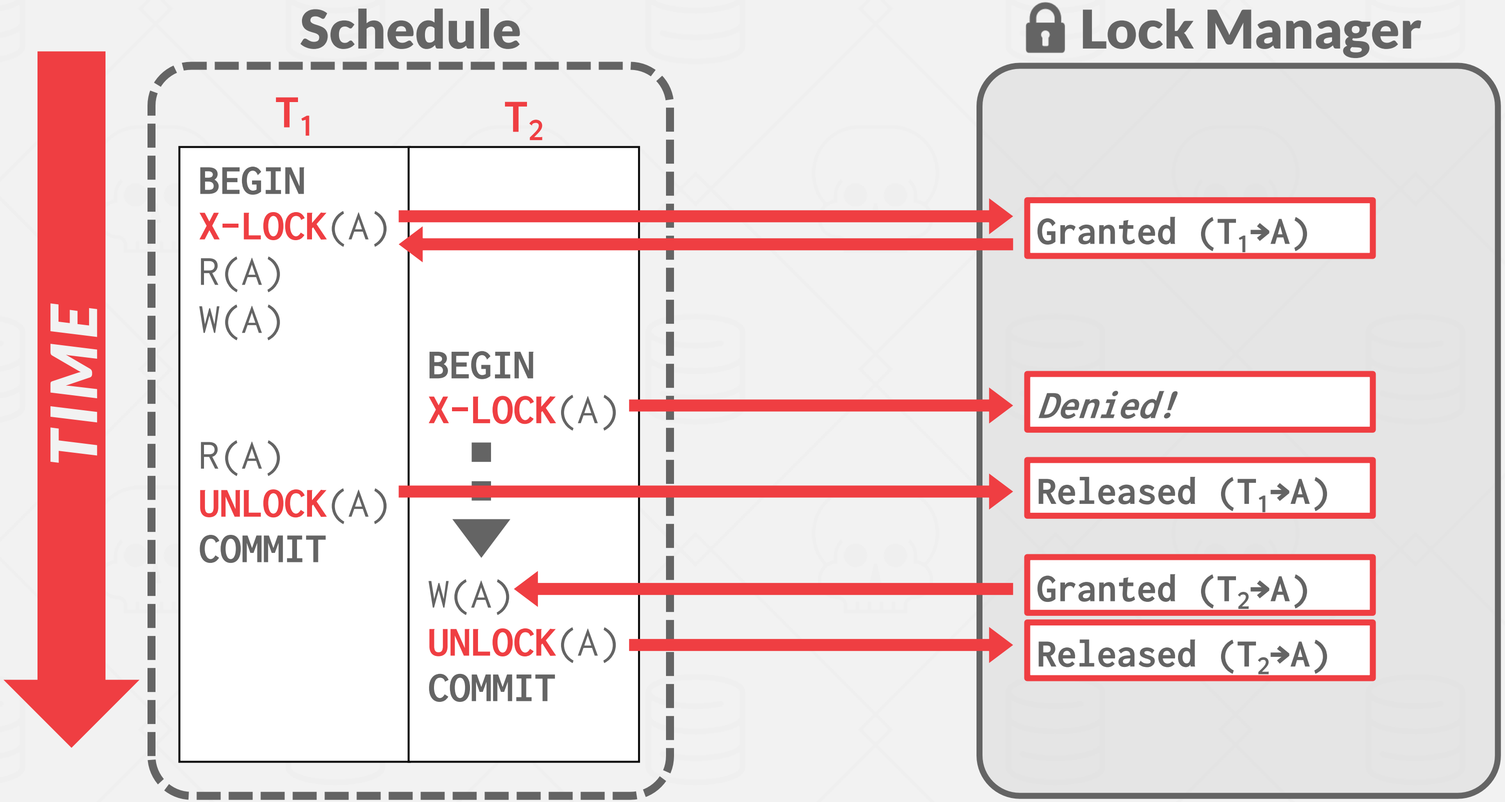
Lock Manager

Granted (T_1→A)

Denied!

Released (T_1→A)

EXECUTING WITH 2PL



T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
	BEGIN
	X-LOCK(A)
R(A)	
UNLOCK(A)	
COMMIT	
	W(A)
	UNLOCK(A)
	COMMIT

Granted (T_1→A)

Denied!

Released (T_1→A)

Granted (T_2→A)

Released (T_2→A)

TWO-PHASE LOCKING

2PL on its own is sufficient to guarantee conflict serializability because it generates schedules whose precedence graph is acyclic.

But it is subject to **cascading aborts**.

2PL – CASCADING ABORTS

Schedule

TIME

T_1	T_2
BEGIN	
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	BEGIN
	X-LOCK(A)
	R(A)
	W(A)
	⋮
R(B)	
W(B)	
⋮	

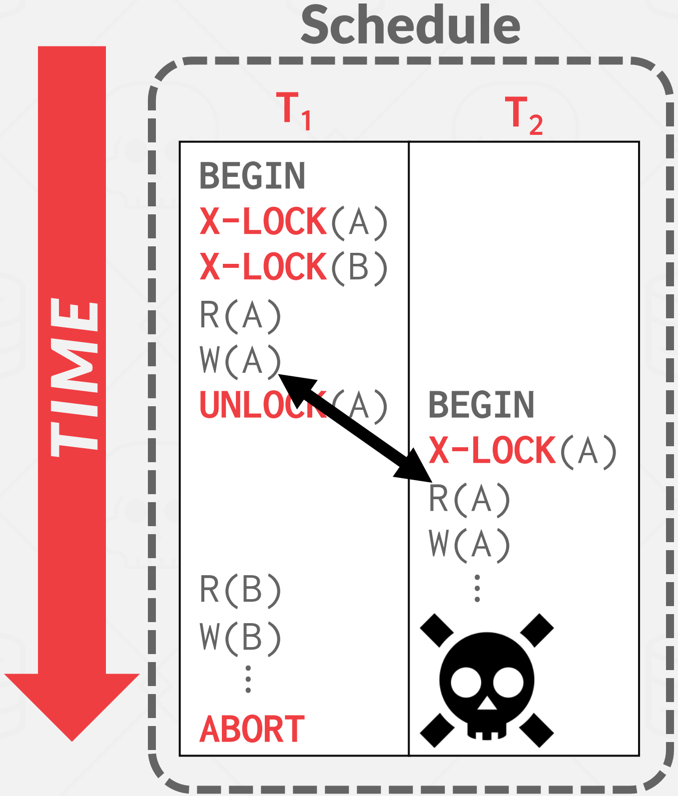
2PL – CASCADING ABORTS

Schedule

TIME

T_1	T_2
BEGIN	
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	BEGIN
	X-LOCK(A)
	R(A)
	W(A)
R(B)	⋮
W(B)	
⋮	
ABORT	

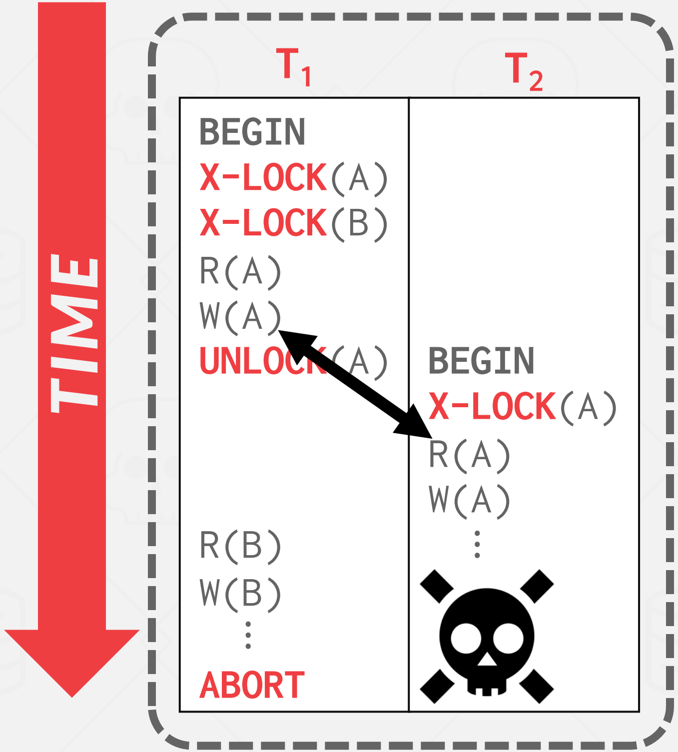
2PL – CASCADING ABORTS



Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	BEGIN
	X-LOCK(A)
	R(A)
	W(A)
	⋮
R(B)	
W(B)	
⋮	
ABORT	

2PL – CASCADING ABORTS



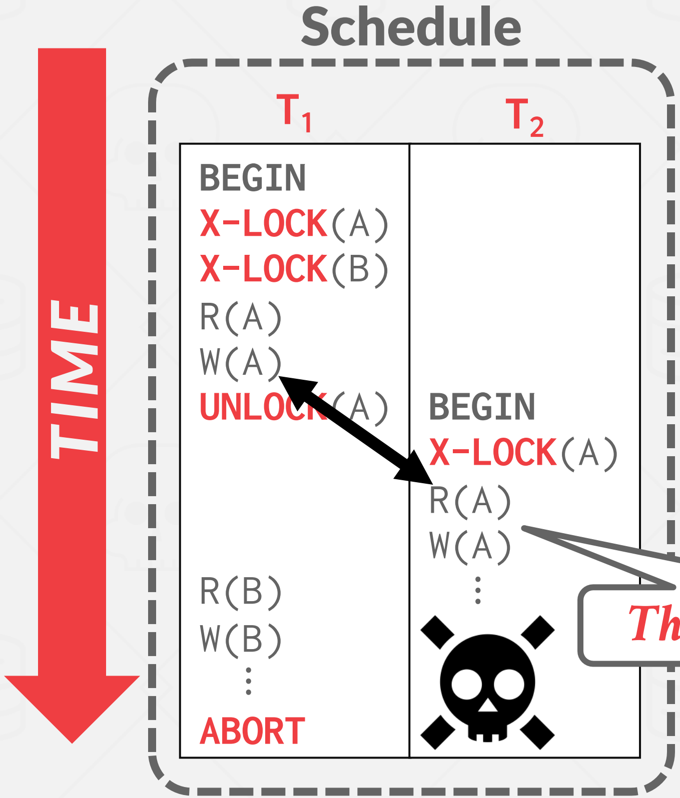
Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	BEGIN
	X-LOCK(A)
	R(A)
	W(A)
	⋮
R(B)	
W(B)	
⋮	
ABORT	

This is a permissible schedule in 2PL, but the DBMS has to also abort T_2 when T_1 aborts.

Any information about T_1 cannot be "leaked" to the outside world.

2PL – CASCADING ABORTS



Schedule

T_1	T_2
BEGIN	
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	BEGIN
	X-LOCK(A)
	R(A)
	W(A)
R(B)	⋮
W(B)	
⋮	
ABORT	

This is all wasted work!

This is a permissible schedule in 2PL, but the DBMS has to also abort T_2 when T_1 aborts.

Any information about T_1 cannot be "leaked" to the outside world.

2PL OBSERVATIONS

There are potential schedules that are serializable but would not be allowed by 2PL because locking limits concurrency.

- → Most DBMSs prefer correctness before performance.

May still have "dirty reads".

- → Solution: **Strong Strict 2PL (aka Rigorous 2PL)**

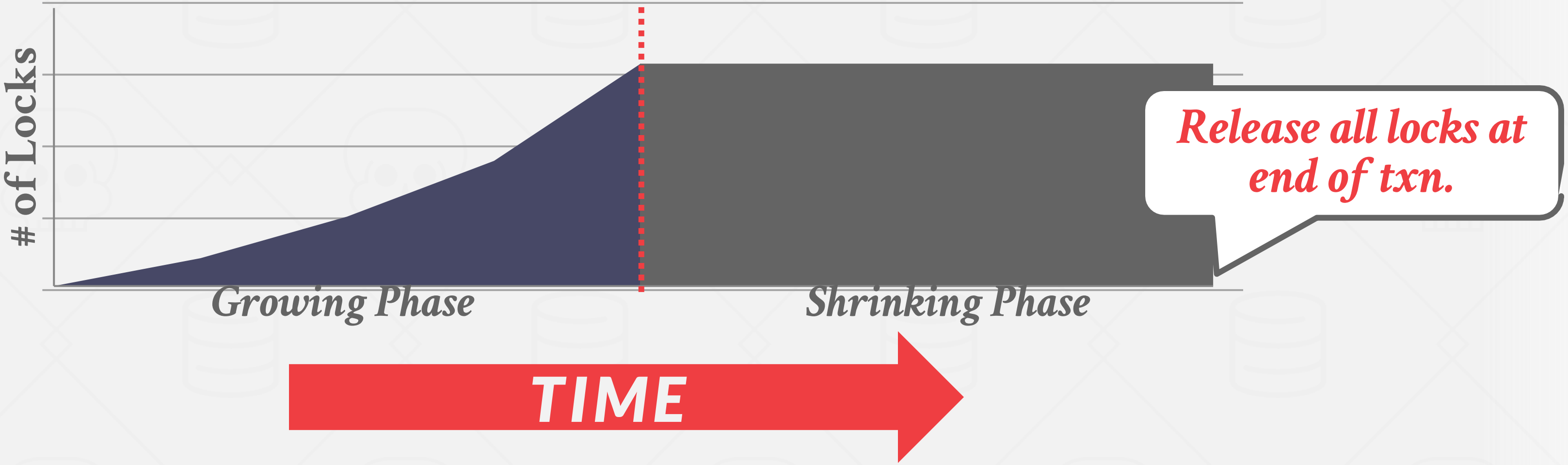
May lead to deadlocks.

- → Solution: **Detection** or **Prevention**

STRONG STRICT TWO-PHASE LOCKING

The txn is only allowed to release locks after it has ended (i.e., committed or aborted).

Allows only conflict serializable schedules, but it is often stronger than needed for some apps.



Release all locks at end of txn.

STRONG STRICT TWO-PHASE LOCKING

A schedule is **strict** if a value written by a txn is not read or overwritten by other txns until that txn finishes.

Advantages:

→ Does not incur cascading aborts.

→ Aborted txns can be undone by just restoring original values of modified tuples.

EXAMPLES

T_1 – Move $100 from Andy's account (**A**) to his bookie's account (**B**).

T_2 – Compute the total amount in all accounts and return it to the application.

T_1

```
BEGIN
A=A-100
B=B+100
COMMIT
```

T_2

```
BEGIN
ECHO A+B
COMMIT
```

NON-2PL EXAMPLE

Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
R(B)	**ECHO A+B**
B=B+100	COMMIT
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

A=1000, B=1000

NON-2PL EXAMPLE

Schedule

TIME

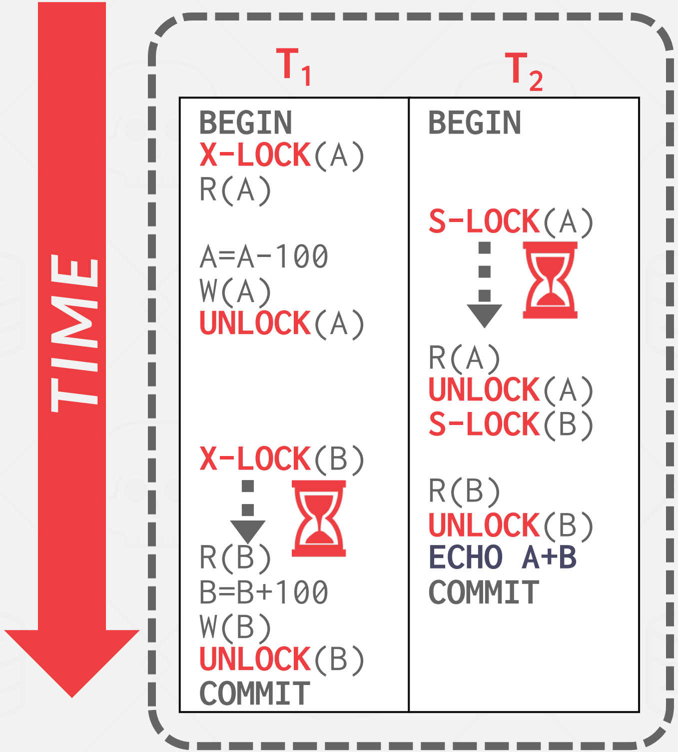
T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
R(B)	ECHO A+B
B=B+100	COMMIT
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

A=1000, B=1000

NON-2PL EXAMPLE

Schedule



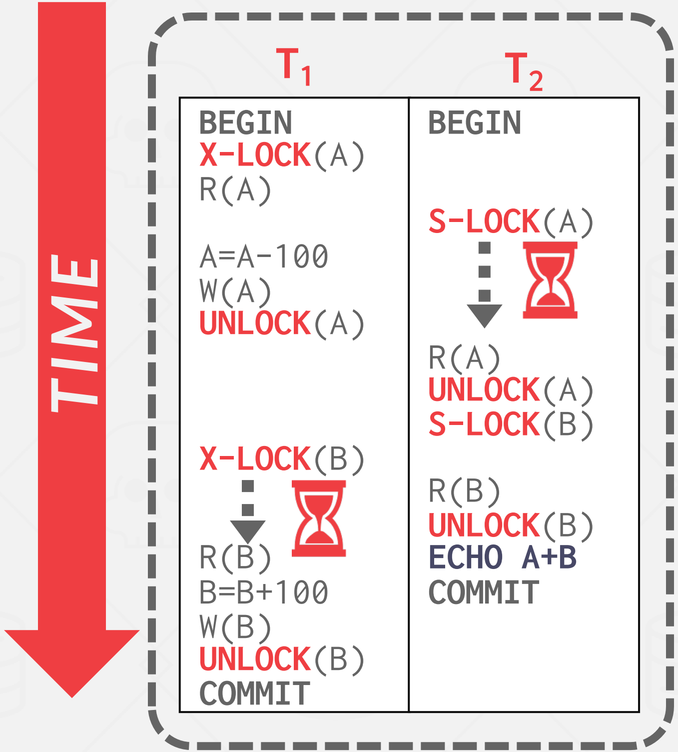
T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
R(B)	ECHO A+B
B=B+100	COMMIT
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

A=1000, B=1000

NON-2PL EXAMPLE

Schedule



T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-100	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
R(B)	ECHO A+B
B=B+100	COMMIT
W(B)	
UNLOCK(B)	
COMMIT	

Initial Database State

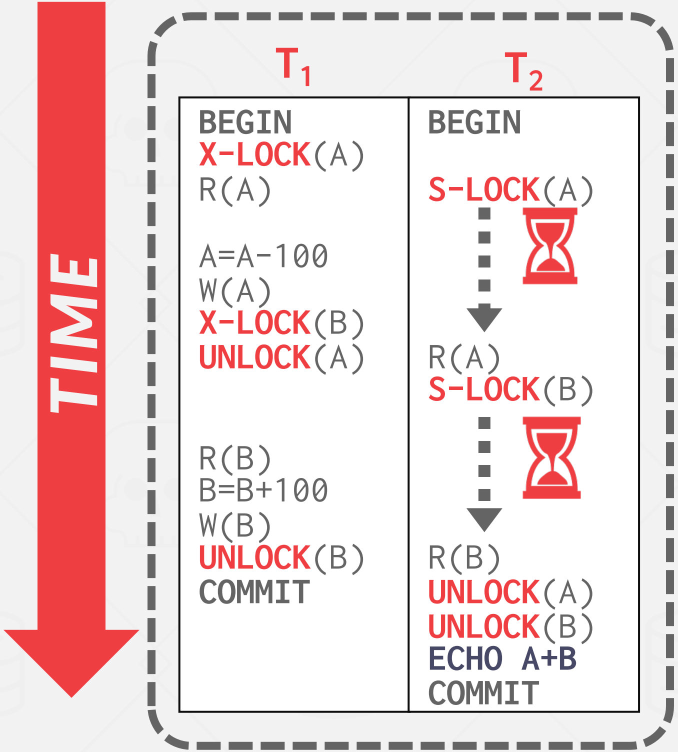
A=1000, B=1000

T_2 Output

A+B=1900

2PL EXAMPLE

Schedule



T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	**S-LOCK**(A)
A=A-100	
W(A)	
X-LOCK(B)	
UNLOCK(A)	R(A)
	S-LOCK(B)
R(B)	
B=B+100	
W(B)	
UNLOCK(B)	R(B)
COMMIT	**UNLOCK**(A)
	UNLOCK(B)
	ECHO A+B
	COMMIT

Initial Database State

A=1000, B=1000

T_2 Output

A+B=2000

STRONG STRICT 2PL EXAMPLE

Schedule

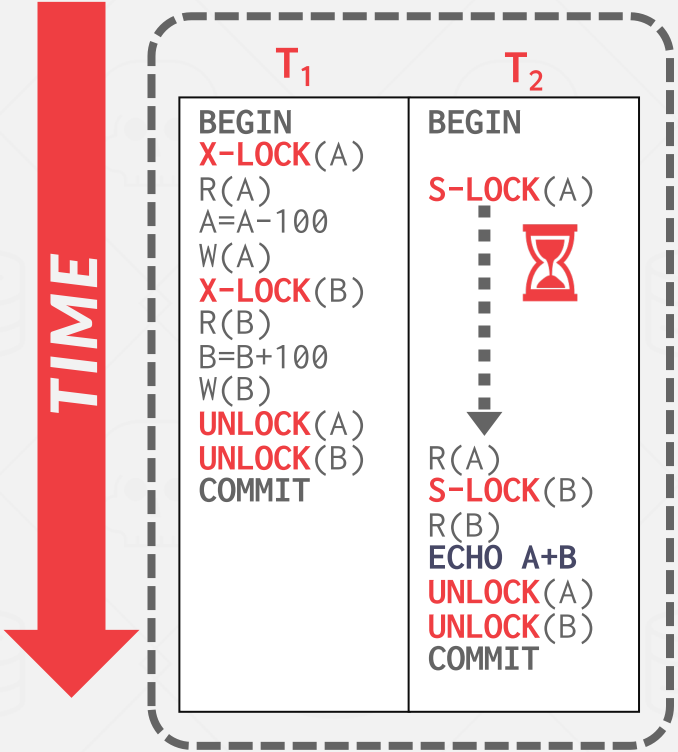
T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	**S-LOCK**(A)
A=A-100	
W(A)	
X-LOCK(B)	
R(B)	
B=B+100	
W(B)	
UNLOCK(A)	
UNLOCK(B)	R(A)
COMMIT	**S-LOCK**(B)
	R(B)
	ECHO A+B
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

Initial Database State

A=1000, **B**=1000

STRONG STRICT 2PL EXAMPLE

Schedule



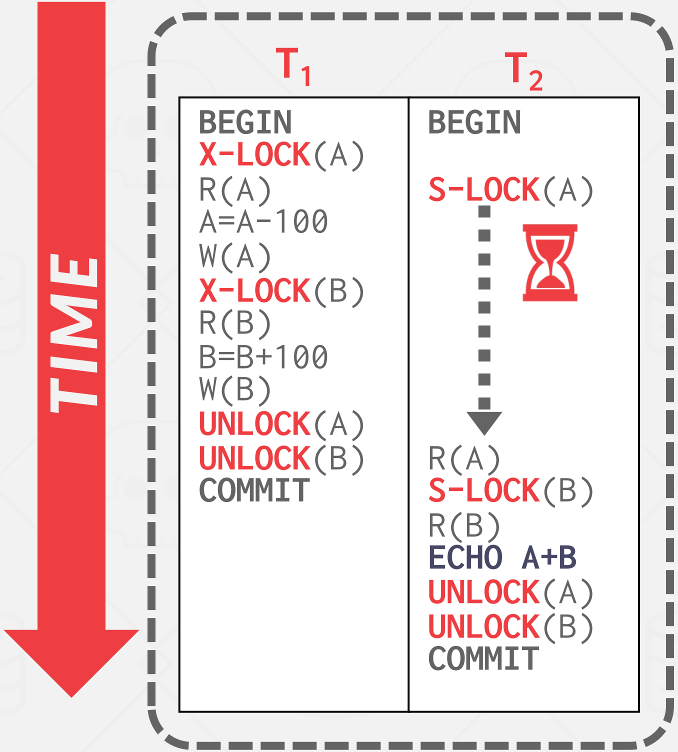
T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-100	
W(A)	
X-LOCK(B)	
R(B)	
B=B+100	
W(B)	
UNLOCK(A)	R(A)
UNLOCK(B)	S-LOCK(B)
COMMIT	R(B)
	ECHO A+B
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

Initial Database State

A=1000, B=1000

STRONG STRICT 2PL EXAMPLE

Schedule



T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-100	
W(A)	
X-LOCK(B)	
R(B)	
B=B+100	
W(B)	
UNLOCK(A)	
UNLOCK(B)	R(A)
COMMIT	S-LOCK(B)
	R(B)
	ECHO A+B
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

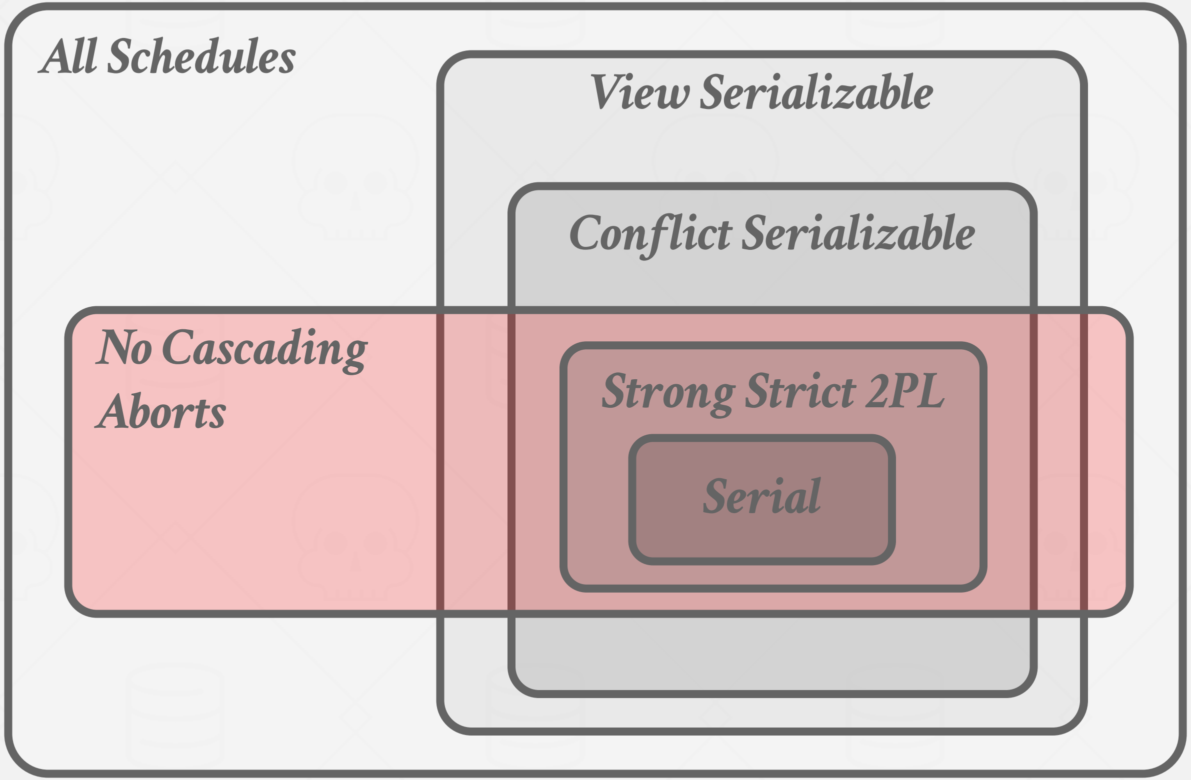
Initial Database State

A=1000, B=1000

T_2 Output

A+B=2000

UNIVERSE OF SCHEDULES



All Schedules

View Serializable

Conflict Serializable

No Cascading Aborts

Strong Strict 2PL

Serial

2PL OBSERVATIONS

There are potential schedules that are serializable but would not be allowed by 2PL because locking limits concurrency.

→ Most DBMSs prefer correctness before performance.

May still have "dirty reads".

→ Solution: **Strong Strict 2PL (aka Rigorous 2PL)**

May lead to deadlocks.

→ Solution: **Detection** or **Prevention**

IT JUST GOT REAL

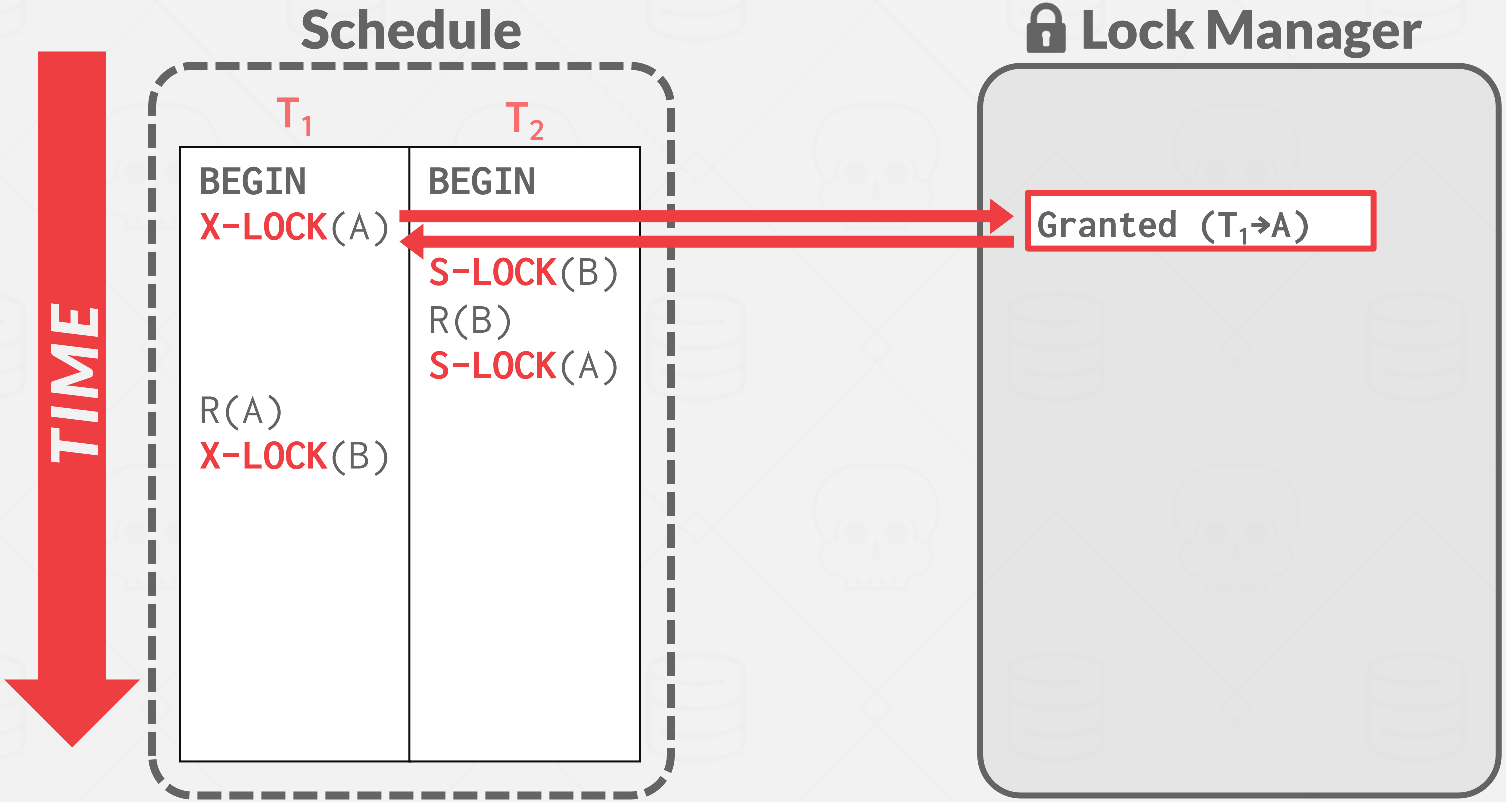
Schedule

TIME

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

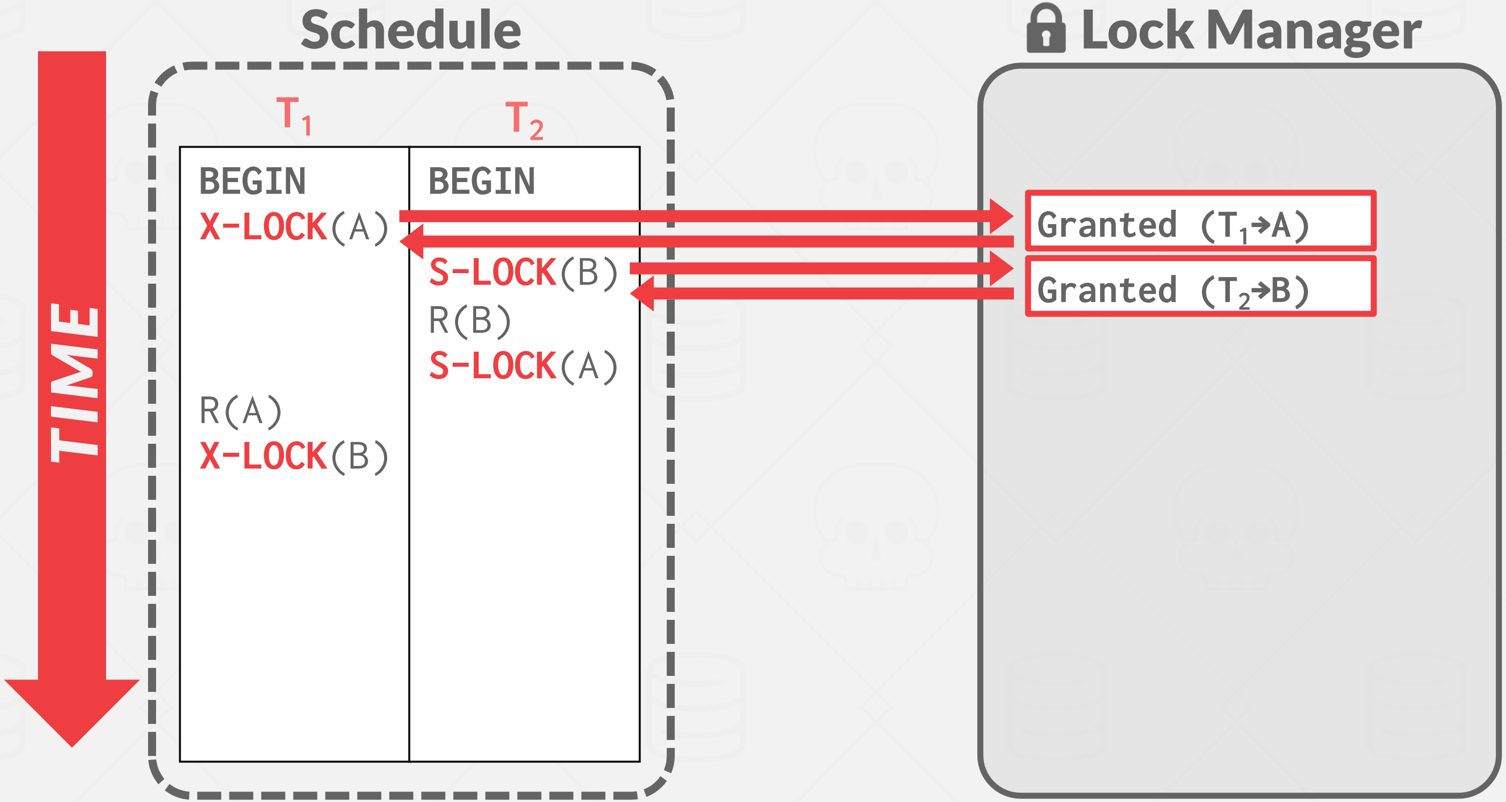
Lock Manager

IT JUST GOT REAL



T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

IT JUST GOT REAL



Schedule

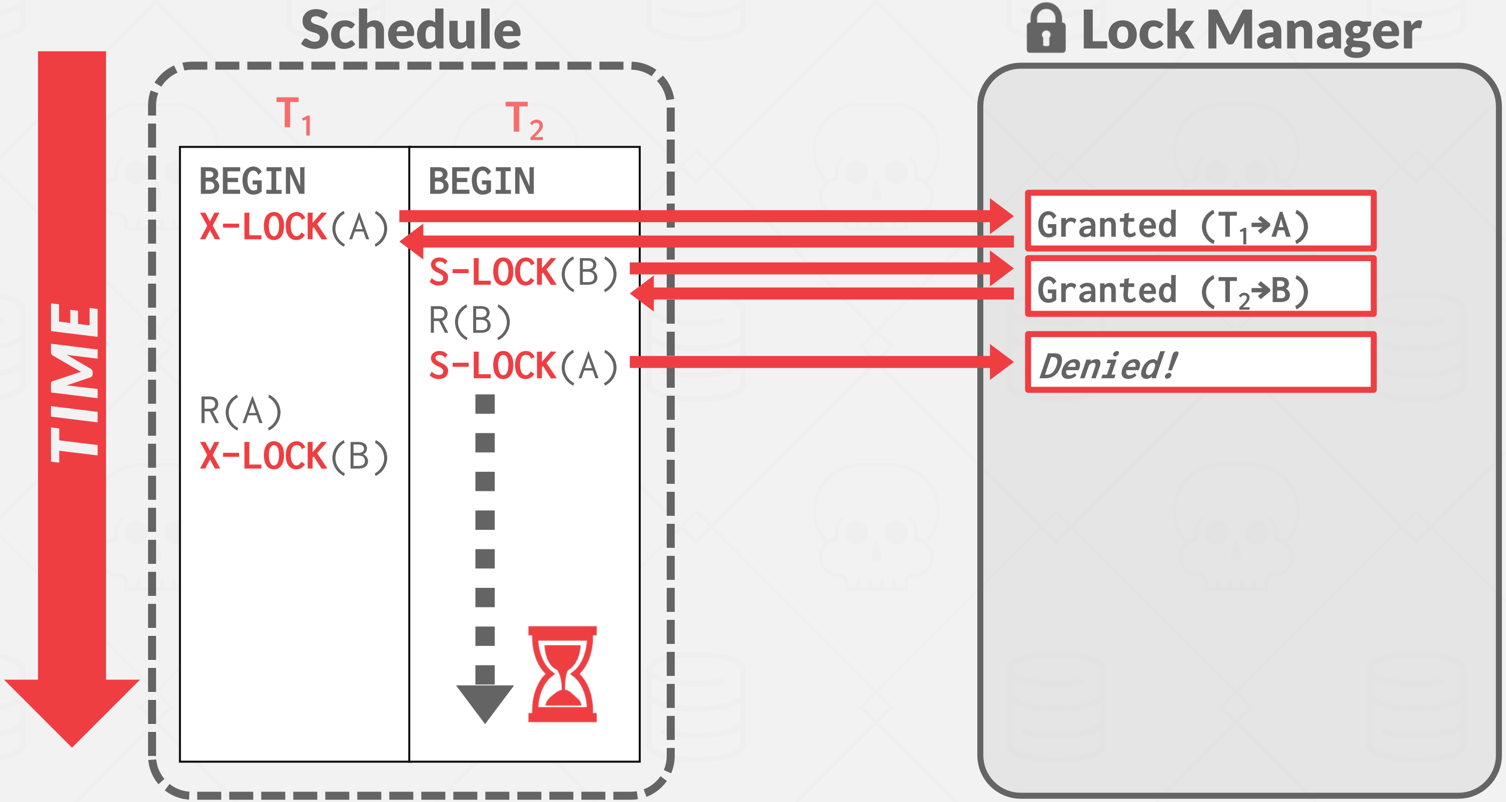
T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

Lock Manager

Granted (T_1→A)

Granted (T_2→B)

IT JUST GOT REAL



Schedule

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

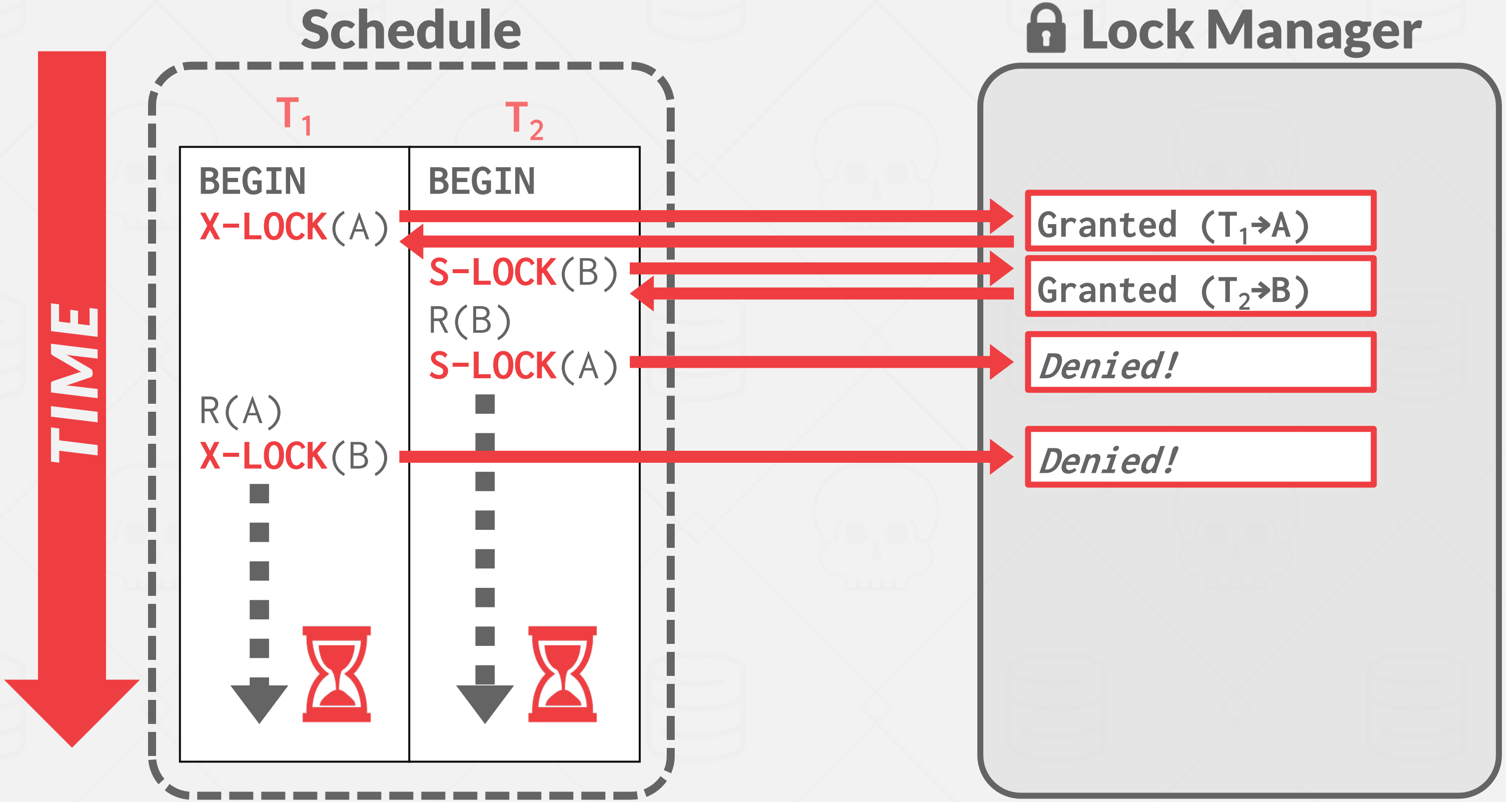
Lock Manager

Granted (T_1→A)

Granted (T_2→B)

Denied!

IT JUST GOT REAL



Schedule

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

Lock Manager

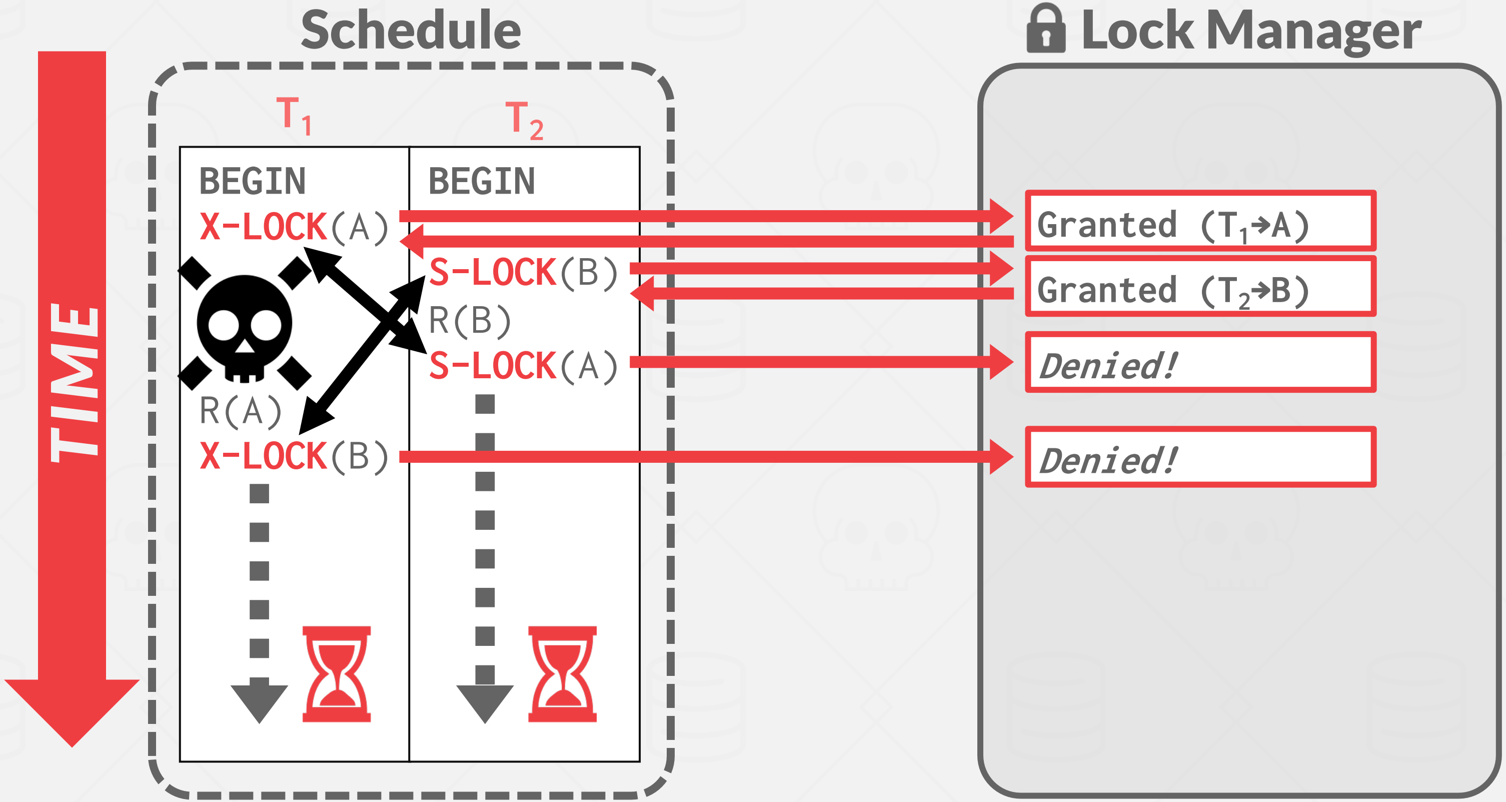
Granted (T_1→A)

Granted (T_2→B)

Denied!

Denied!

IT JUST GOT REAL



Schedule

T_1	T_2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

Lock Manager

Granted (T_1→A)

Granted (T_2→B)

Denied!

Denied!

2PL DEADLOCKS

A **deadlock** is a cycle of transactions waiting for locks to be released by each other.

Two ways of dealing with deadlocks:

- → **Approach #1: Deadlock Detection**
- → **Approach #2: Deadlock Prevention**

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.

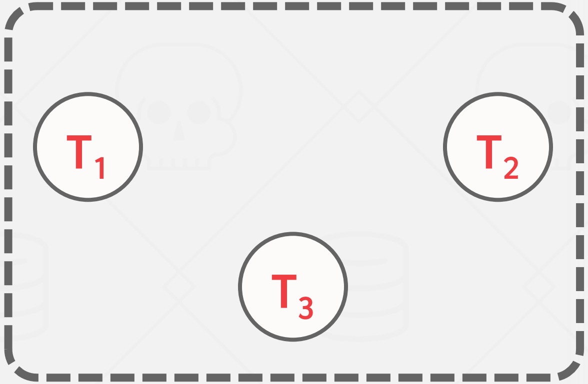
The system periodically checks for cycles in ***waits-for*** graph and then decides how to break it.

DEADLOCK DETECTION

Schedule

T_1	T_2	T_3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-For Graph

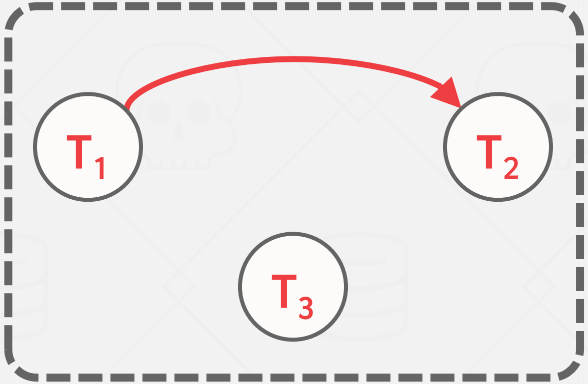


DEADLOCK DETECTION

Schedule

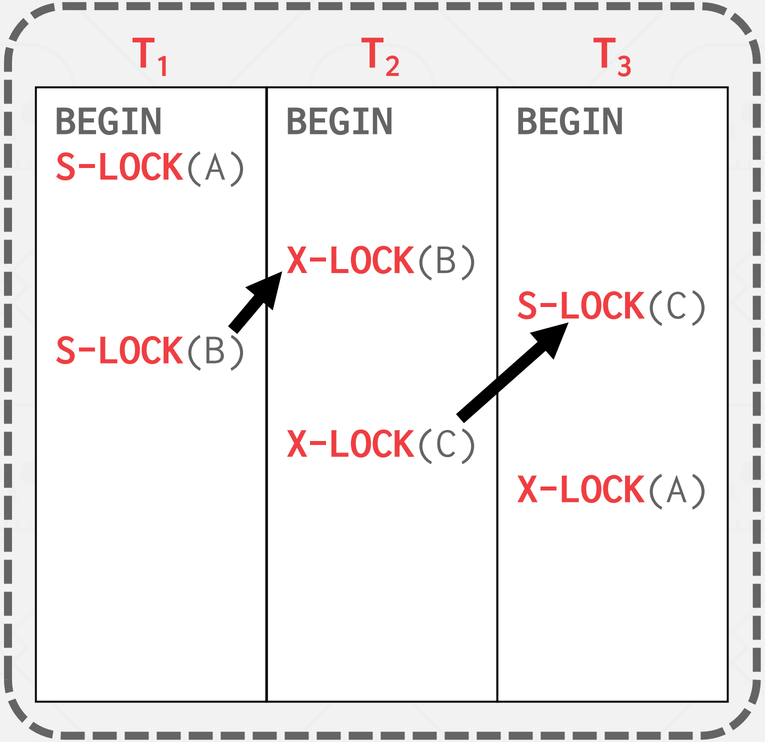
T_1	T_2	T_3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-For Graph



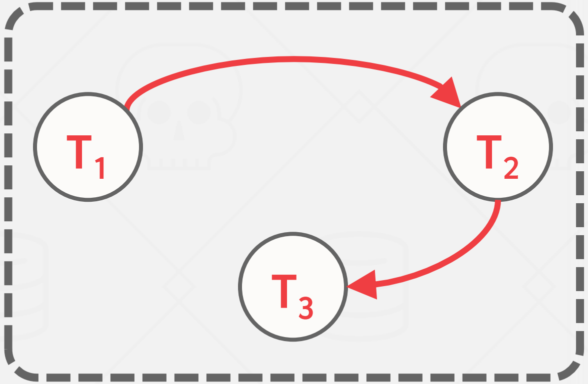
DEADLOCK DETECTION

Schedule



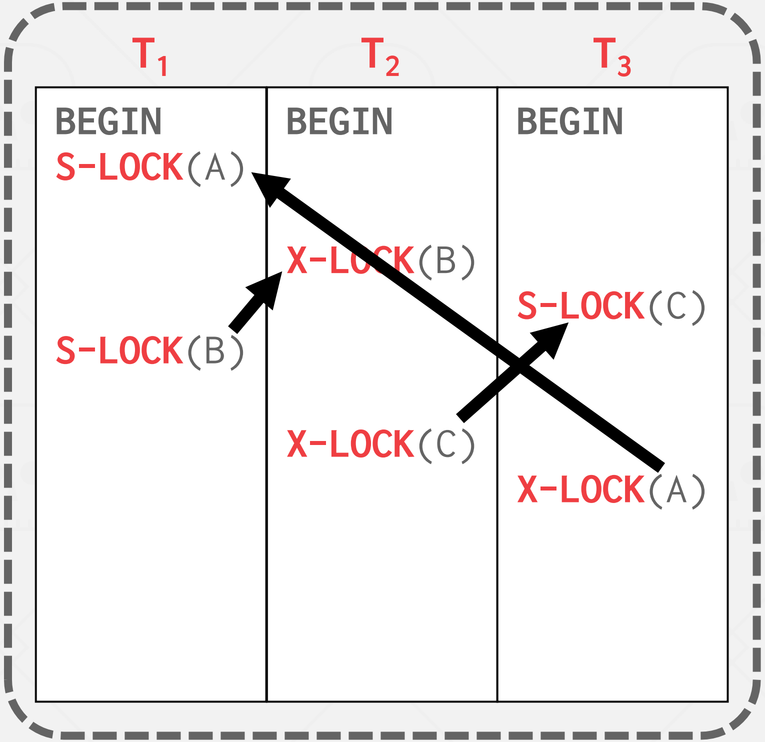
T_1	T_2	T_3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-For Graph



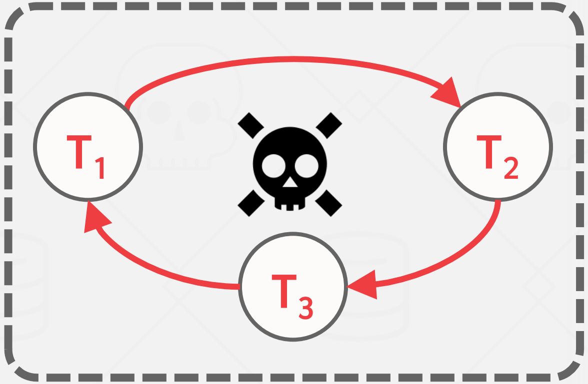
DEADLOCK DETECTION

Schedule



T_1	T_2	T_3
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 HANDLING

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

The victim txn will either restart or abort (more common) depending on how it was invoked.

There is a trade-off between the frequency of checking for deadlocks and how long txns wait before deadlocks are broken.

DEADLOCK HANDLING: VICTIM SELECTION

Selecting the proper victim depends on a lot of different variables….

- → 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 HANDLING: ROLLBACK LENGTH

After selecting a victim txn to abort, the DBMS can also decide on how far to rollback the txn's changes.

Approach #1: Completely

→ Rollback entire txn and tell the application it was aborted.

Approach #2: Partial (Savepoints)

→ DBMS rolls back a portion of a txn (to break deadlock) and then attempts to re-execute the undone queries.

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., $T_1 > T_2$)

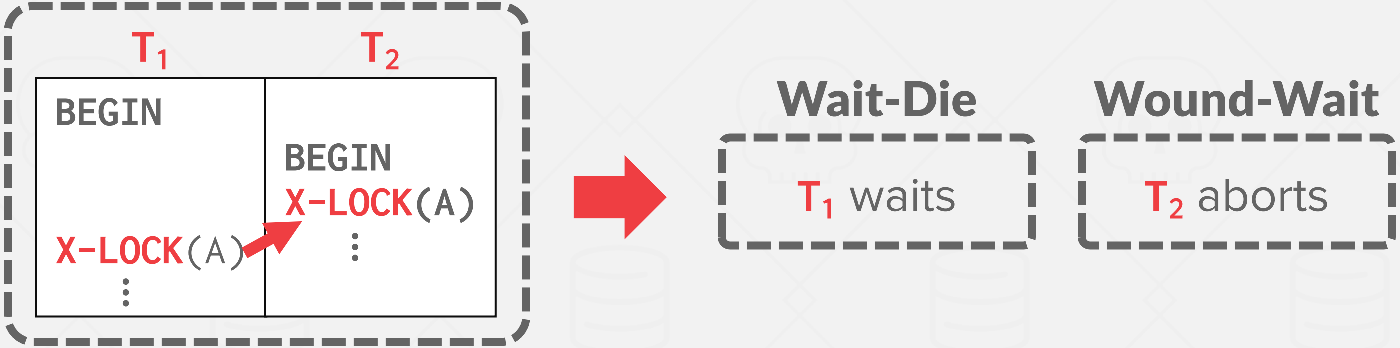
Wait-Die ("Old Waits for Young")

- → If *requesting txn* has higher priority than *holding txn*, then *requesting txn* waits for *holding txn*.
- → Otherwise *requesting txn* aborts.

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



T_1	T_2
BEGIN	
	BEGIN
	X-LOCK(A)
X-LOCK(A)	⋮
⋮	

Wait-Die

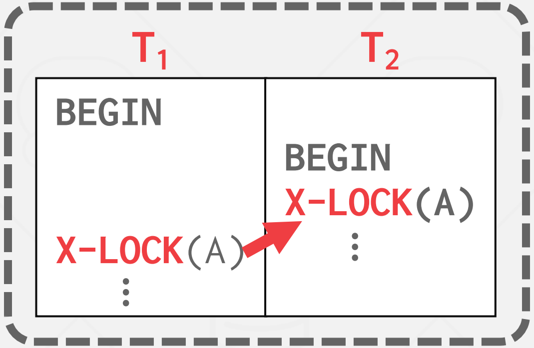
T_1 waits

Wound-Wait

T_2 aborts

T_1	T_2
BEGIN	
X-LOCK(A)	
⋮	
	BEGIN
	X-LOCK(A)
	⋮

DEADLOCK PREVENTION



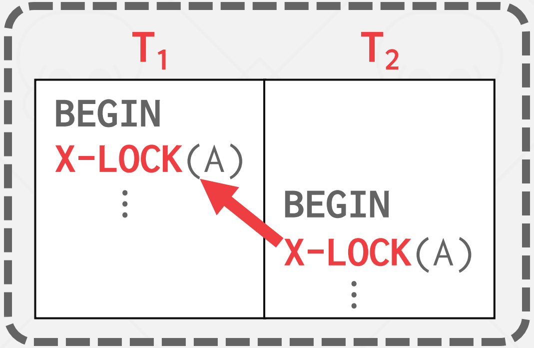
T_1	T_2
BEGIN	
	BEGIN
	X-LOCK(A)
	⋮
X-LOCK(A)	
⋮	

Wait-Die

T_1 waits

Wound-Wait

T_2 aborts



T_1	T_2
BEGIN	
X-LOCK(A)	
⋮	
	BEGIN
	X-LOCK(A)
	⋮

Wait-Die

T_2 aborts

Wound-Wait

T_2 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 like an old man at a corrupt senior center.

OBSERVATION

All these examples have a one-to-one mapping from database objects to locks.

If a txn wants to update one billion tuples, then it must acquire one billion locks.

Acquiring locks is a more expensive operation than acquiring a latch even if that lock is available.

LOCK GRANULARITIES

When a txn wants to acquire a "lock", the DBMS can decide the granularity (i.e., scope) of that lock.

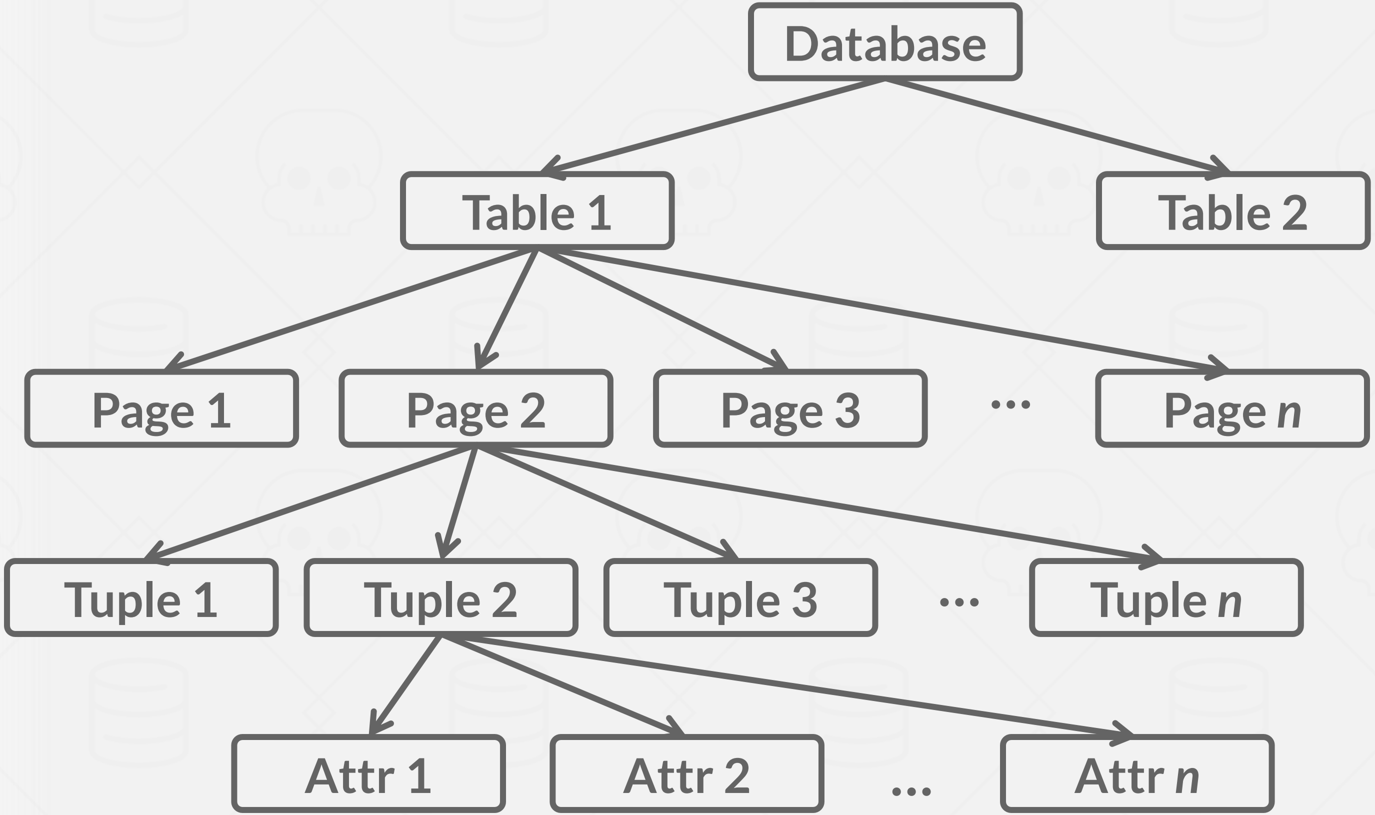
→ Attribute? Tuple? Page? Table?

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

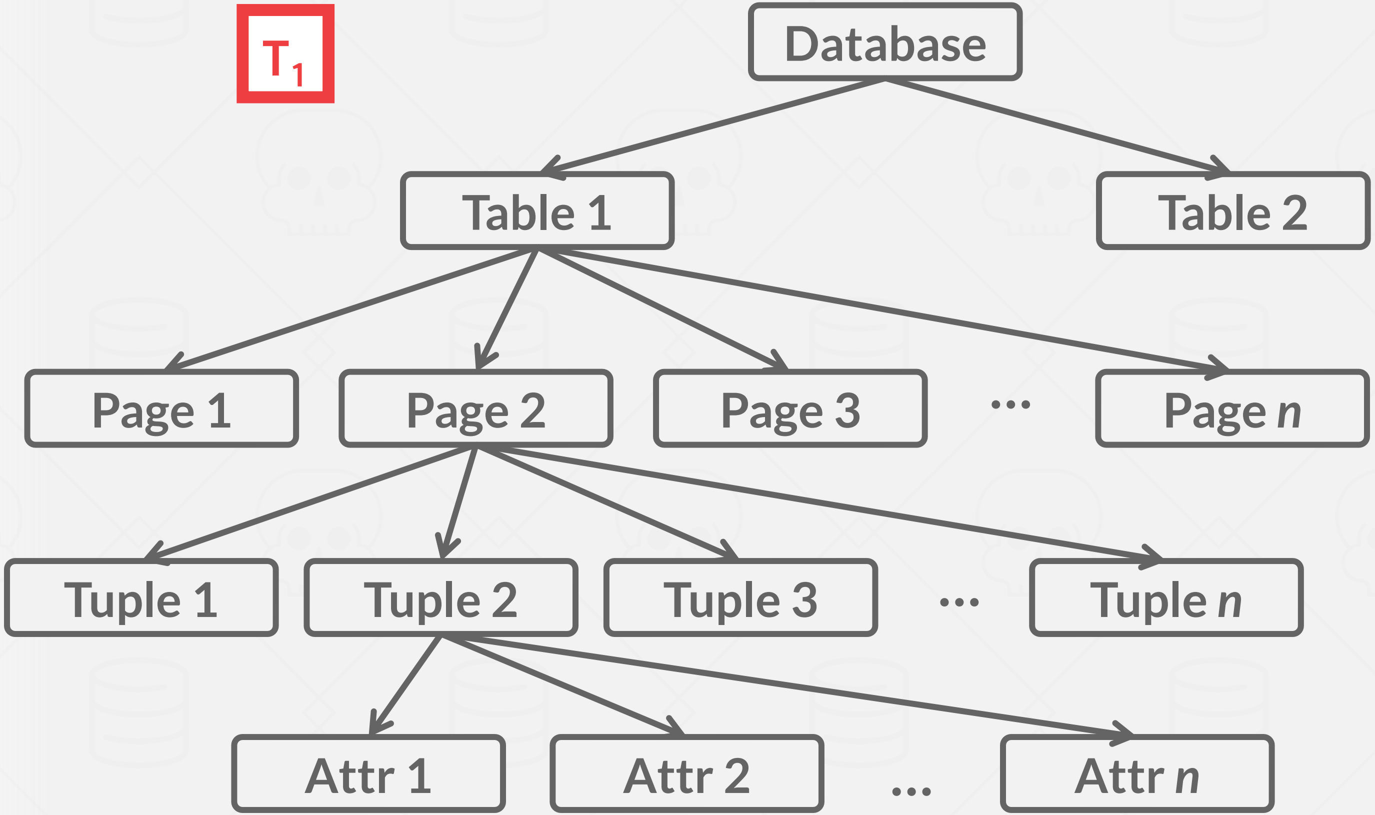
Trade-off between parallelism versus overhead.

→ Fewer Locks, Larger Granularity vs. More Locks, Smaller Granularity.

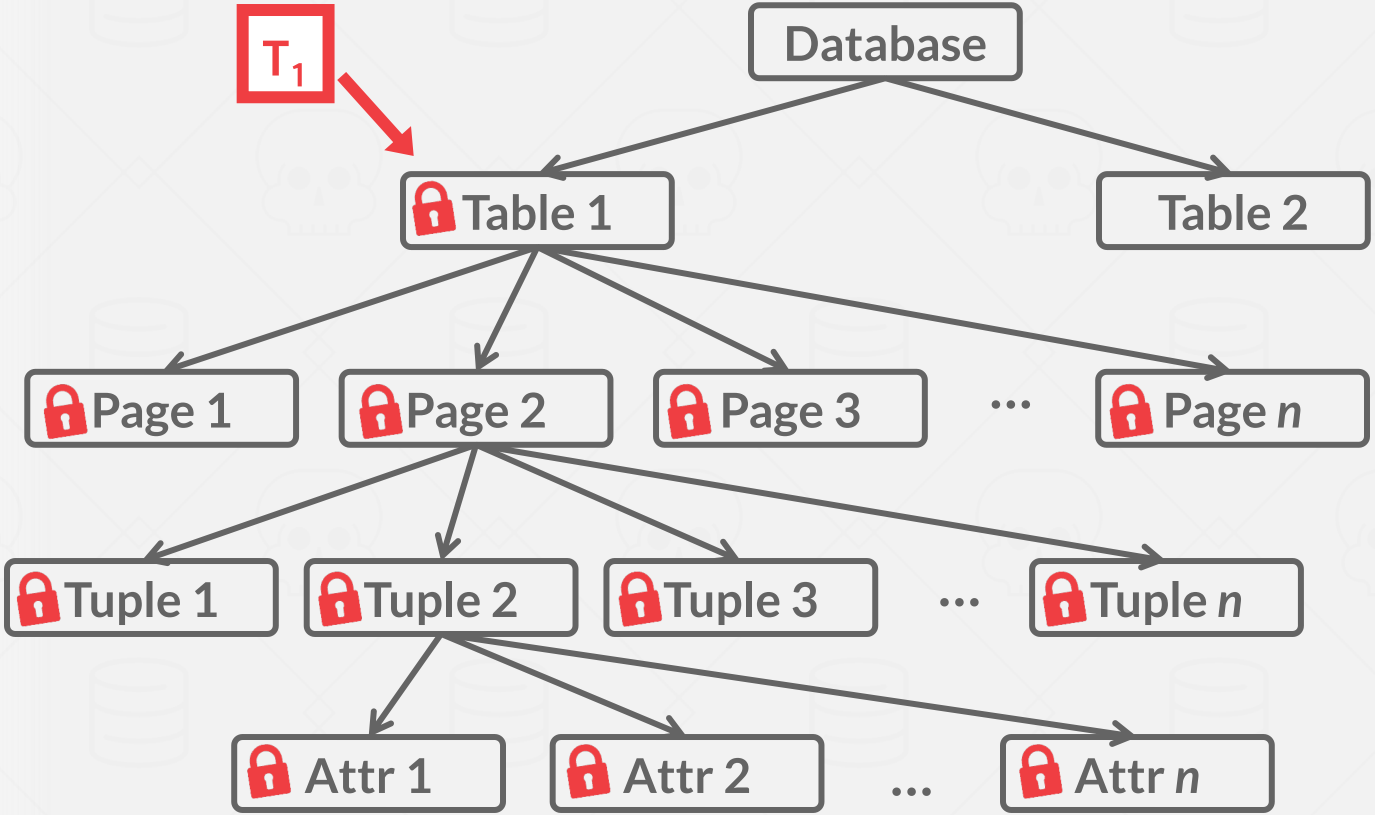
DATABASE LOCK HIERARCHY



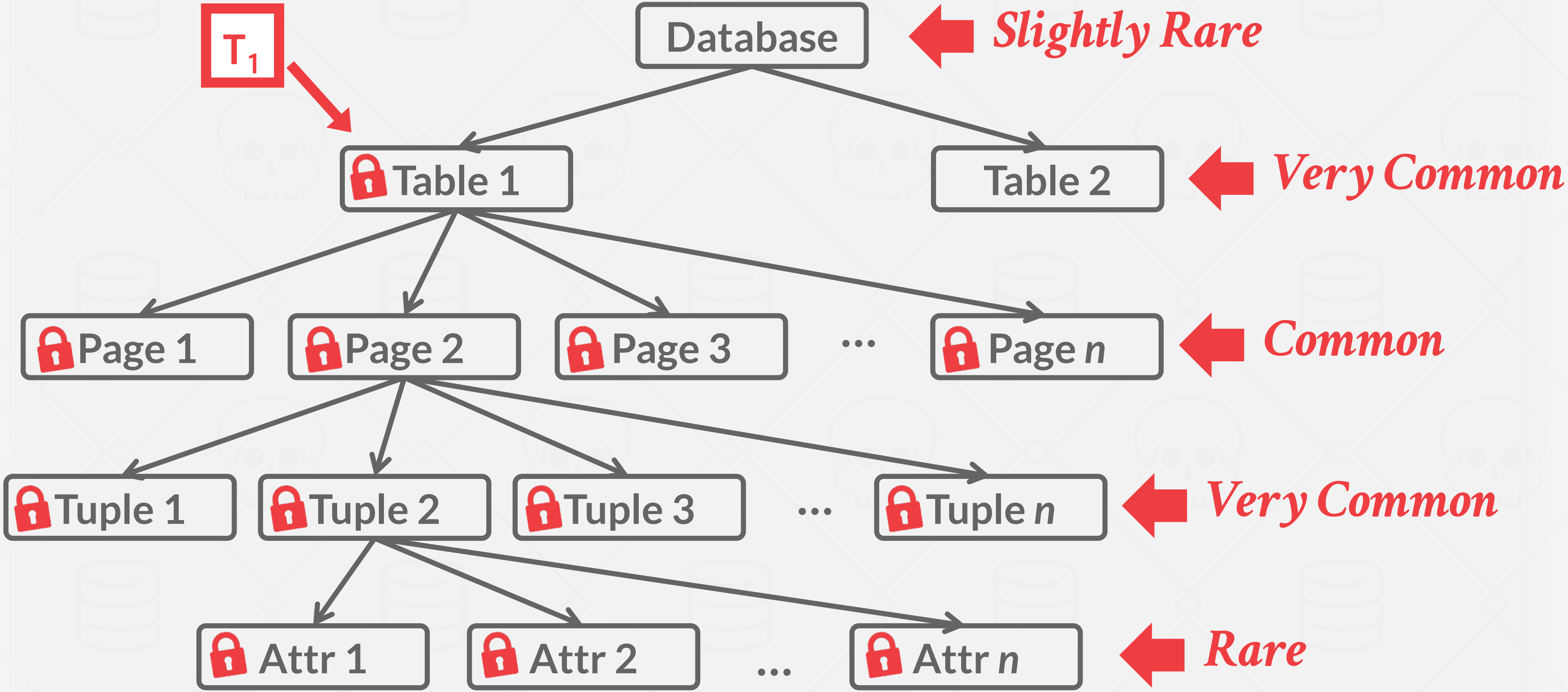
DATABASE LOCK HIERARCHY



DATABASE LOCK HIERARCHY



DATABASE LOCK HIERARCHY



INTENTION LOCKS

An **intention lock** allows a higher-level node to be locked in **shared** or **exclusive** 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 LOCKS

Intention-Shared (IS)

→ Indicates explicit locking at lower level with S locks.

→ Intent to get S lock(s) at finer granularity.

Intention-Exclusive (IX)

→ Indicates explicit locking at lower level with X locks.

→ Intent to get X lock(s) at finer granularity.

Shared+Intention-Exclusive (SIX)

→ The subtree rooted by that node is locked explicitly in S mode and explicit locking is being done at a lower level with X locks.

COMPATIBILITY MATRIX

T_2 Wants

T_1 Holds

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

LOCKING PROTOCOL

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.

EXAMPLE

T_1 – Get the balance of Andy’s off-shore bank account.

T_2 – Increase bookie’s account balance by 1%.

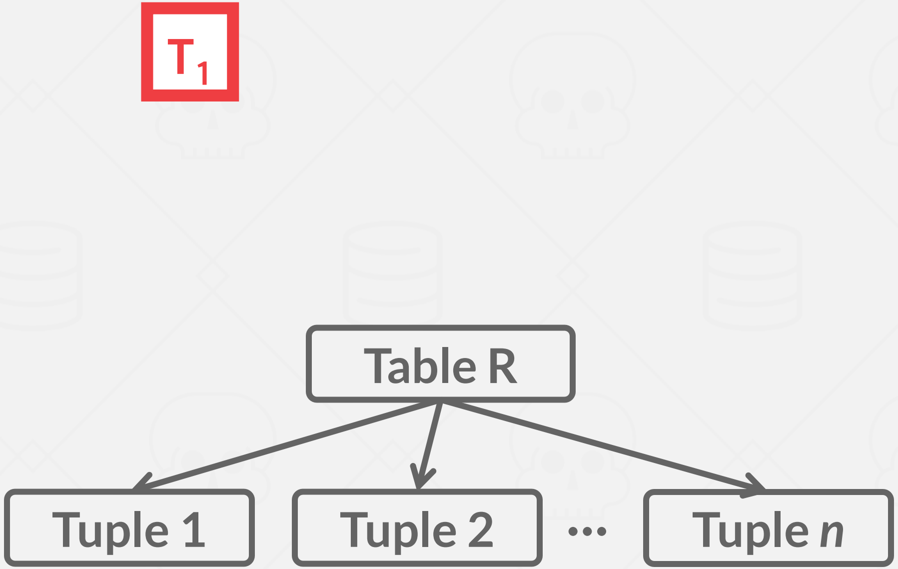
What locks should these txns obtain?

→ **Exclusive** + **Shared** for leaf nodes of lock tree.

→ Special **Intention** locks for higher levels.

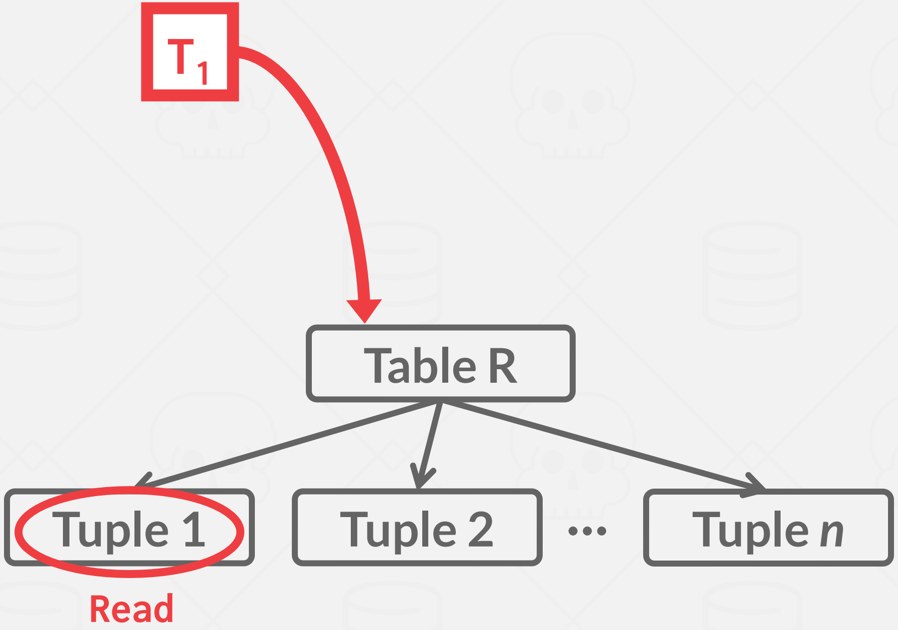
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in **R**.



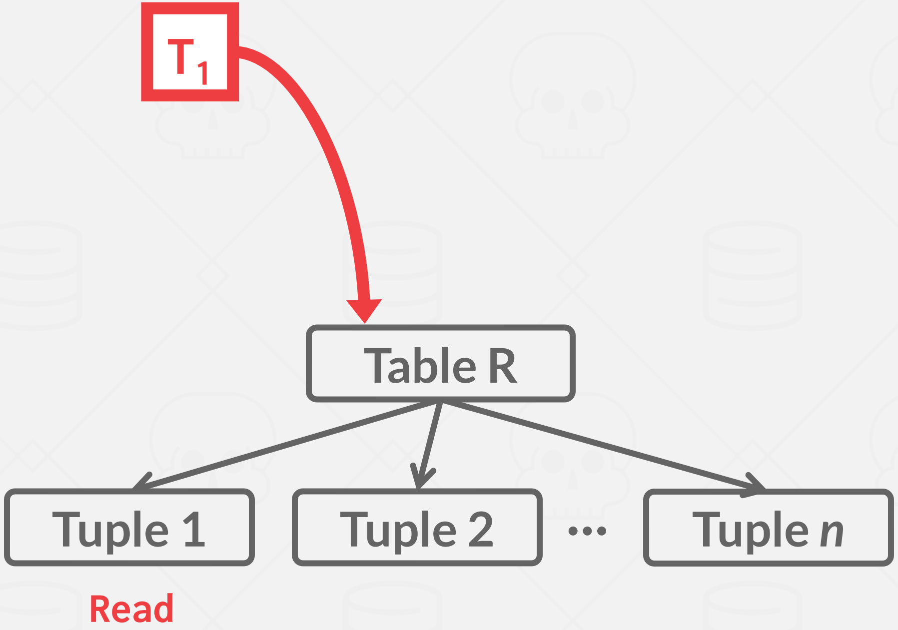
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in **R**.



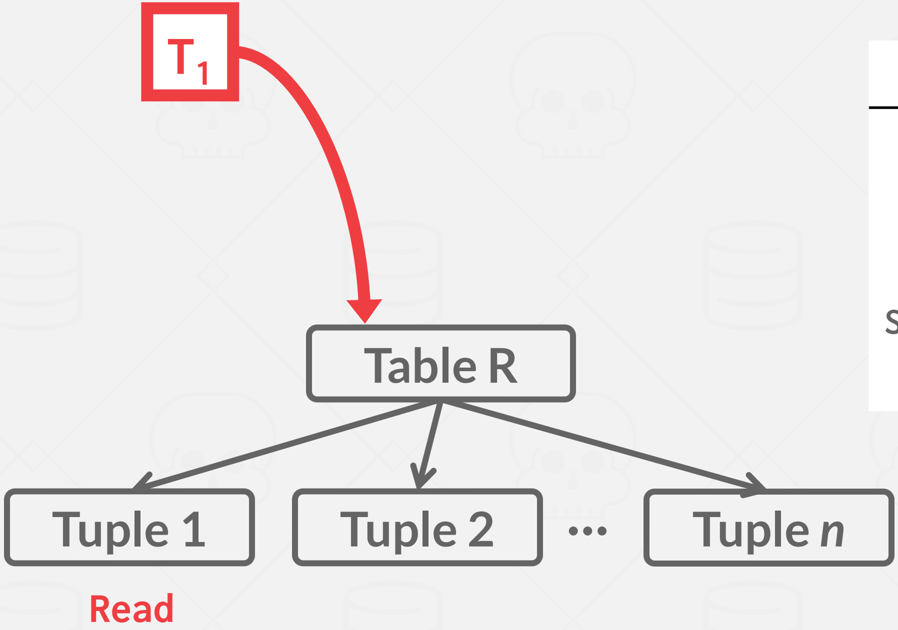
EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in **R**.



EXAMPLE – TWO-LEVEL HIERARCHY

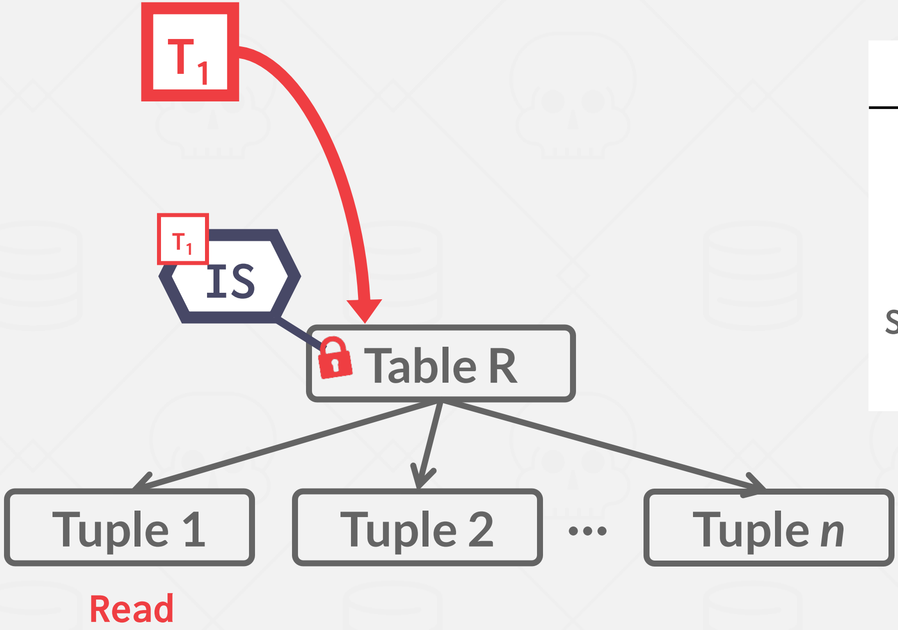
Read Andy's record in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – TWO-LEVEL HIERARCHY

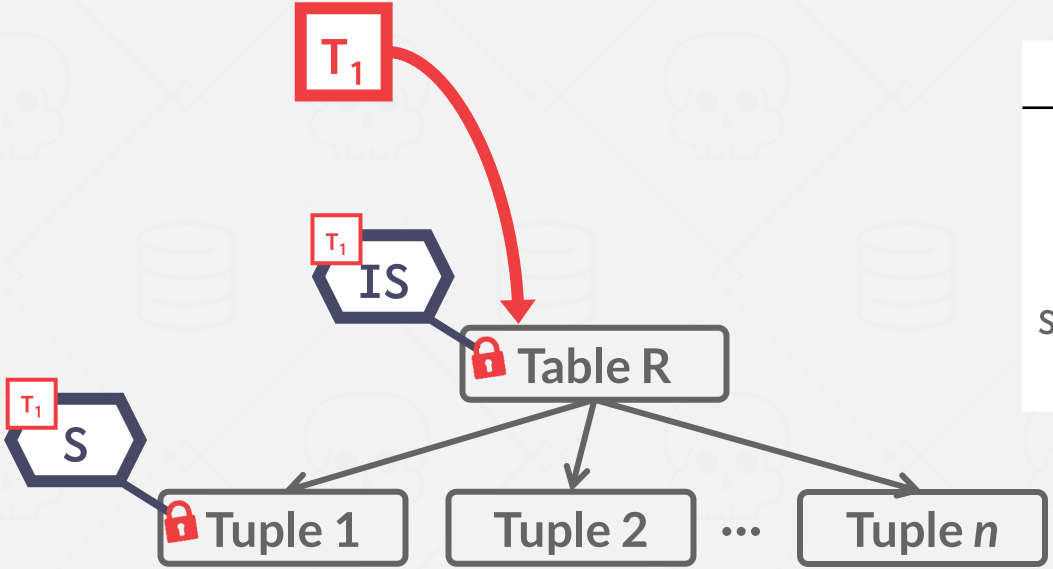
Read Andy's record in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in **R**.

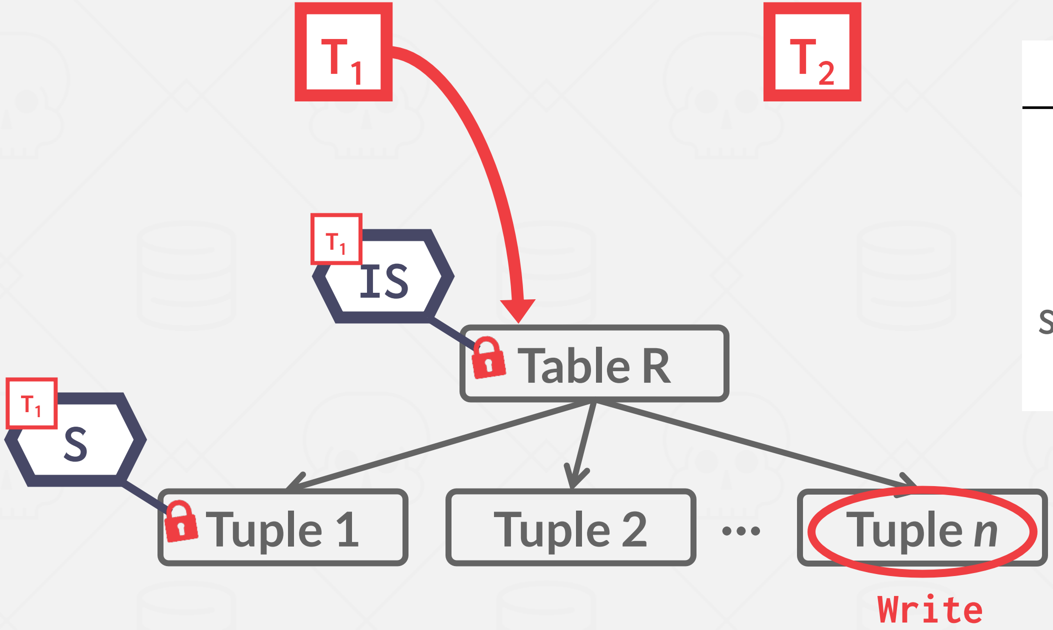


Read

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – TWO-LEVEL HIERARCHY

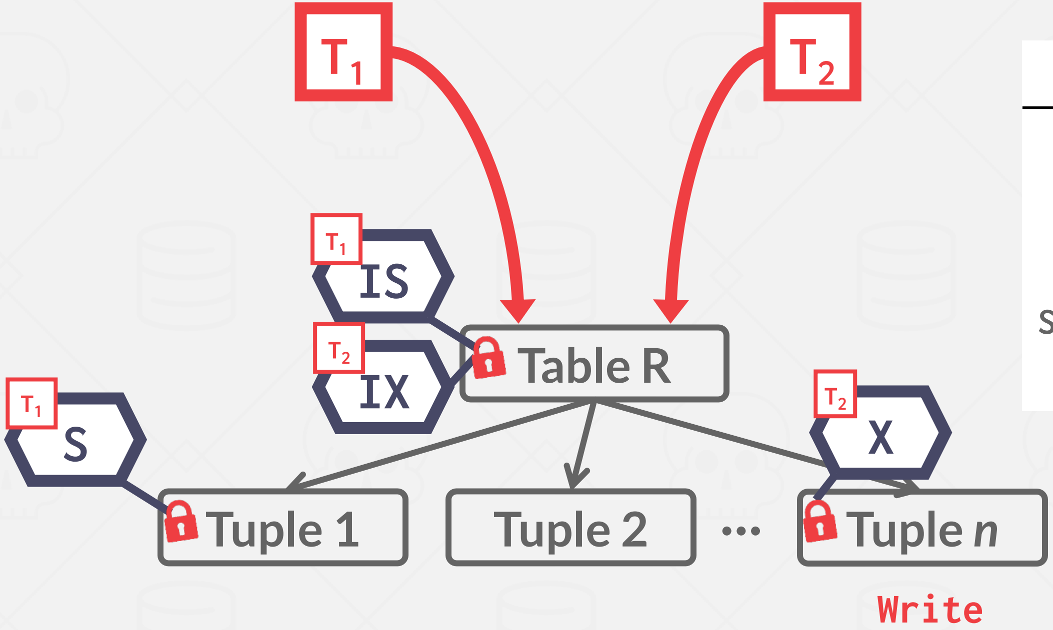
Update bookie's record in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – TWO-LEVEL HIERARCHY

Update bookie's record in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – THREE TXNS

Assume three txns execute at same time:

- → **T_1** – Scan all tuples in **R** and update one tuple.
- → **T_2** – Read a single tuple in **R**.
- → **T_3** – Scan all tuples in **R**.

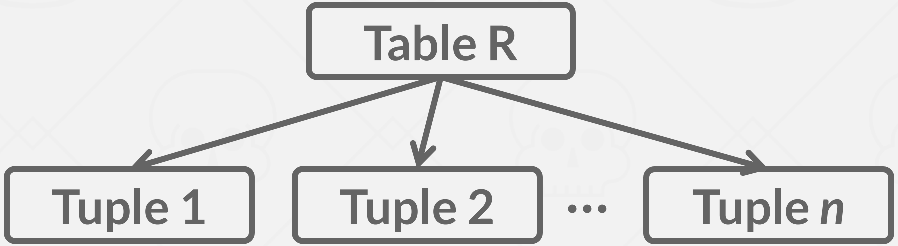


Table R

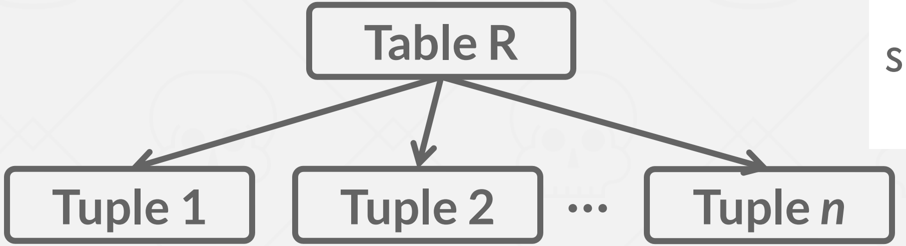
Tuple 1

Tuple 2

…

Tuple *n*

EXAMPLE – THREE TXNS



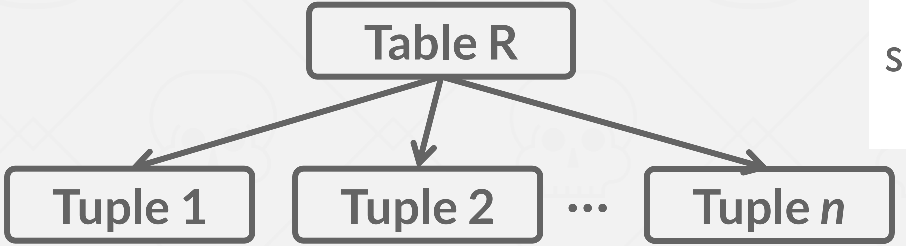
	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

EXAMPLE – THREE TXNS

Scan all tuples in **R** and update one tuple.

T_1

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

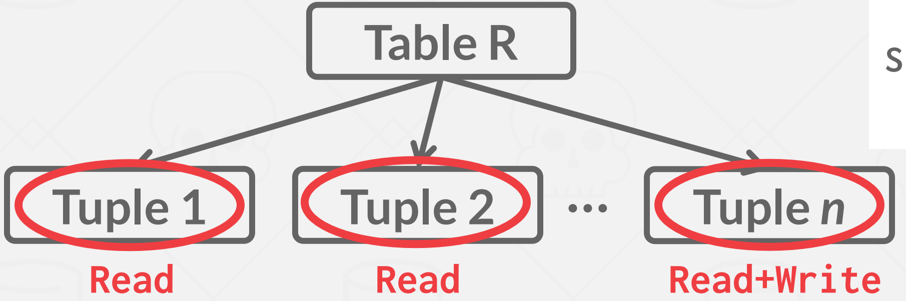


EXAMPLE – THREE TXNS

Scan all tuples in **R** and update one tuple.

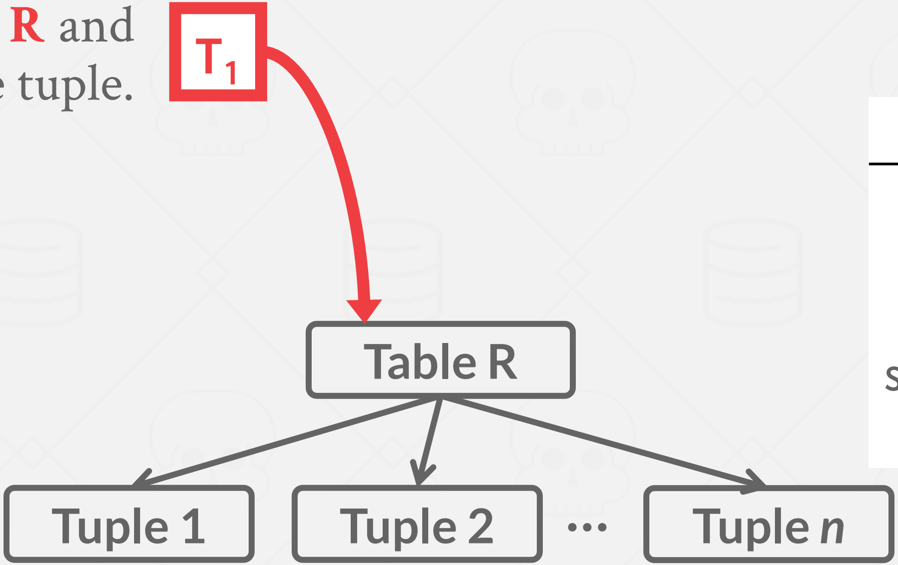
T_1

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×



EXAMPLE – THREE TXNS

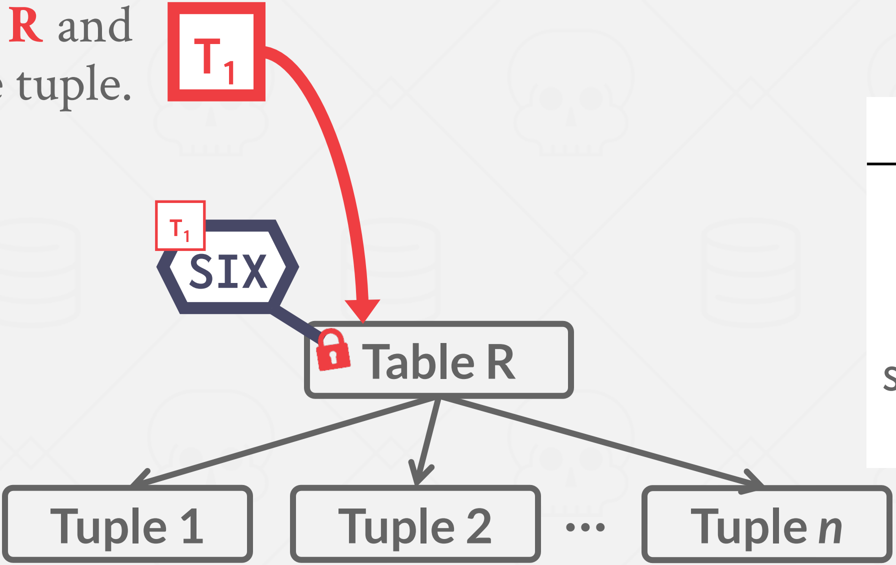
Scan all tuples in **R** and update one tuple.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – THREE TXNS

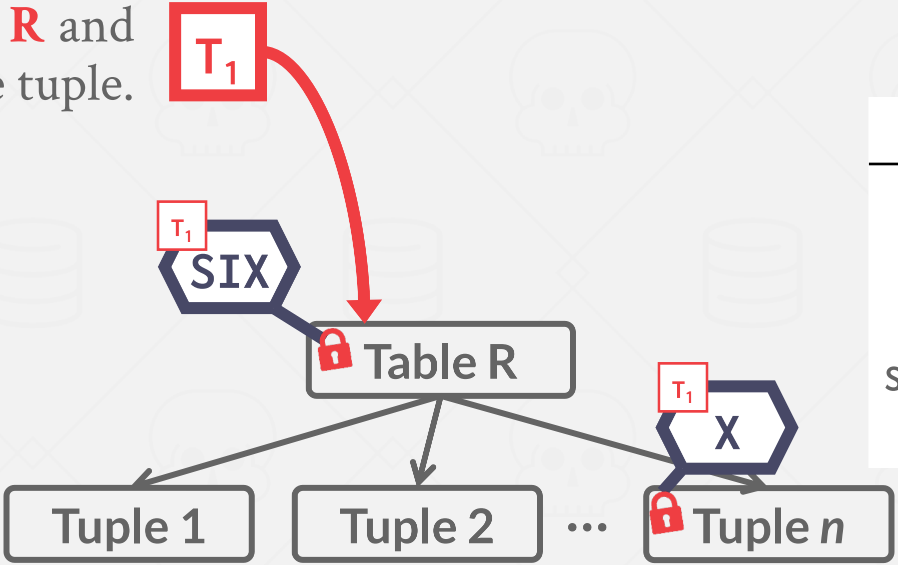
Scan all tuples in **R** and update one tuple.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

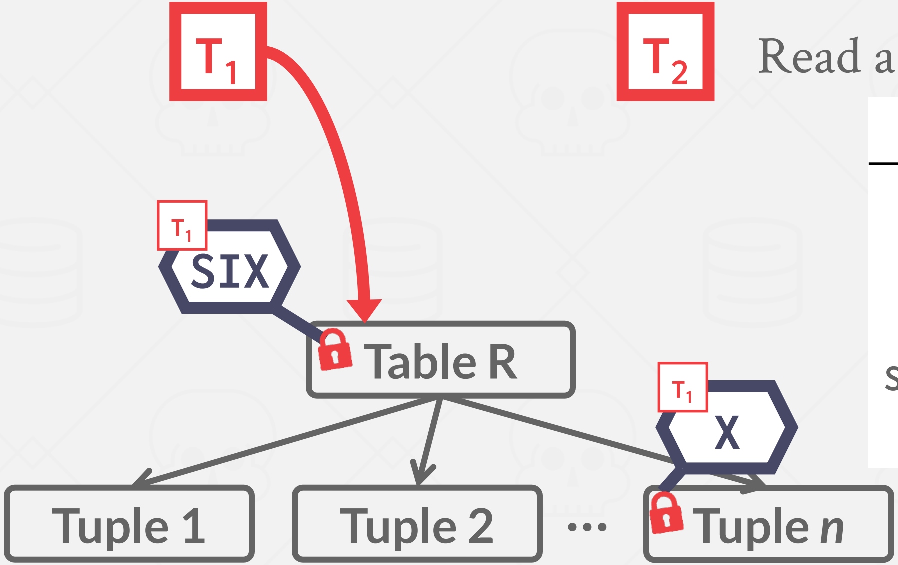
EXAMPLE – THREE TXNS

Scan all tuples in **R** and update one tuple.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

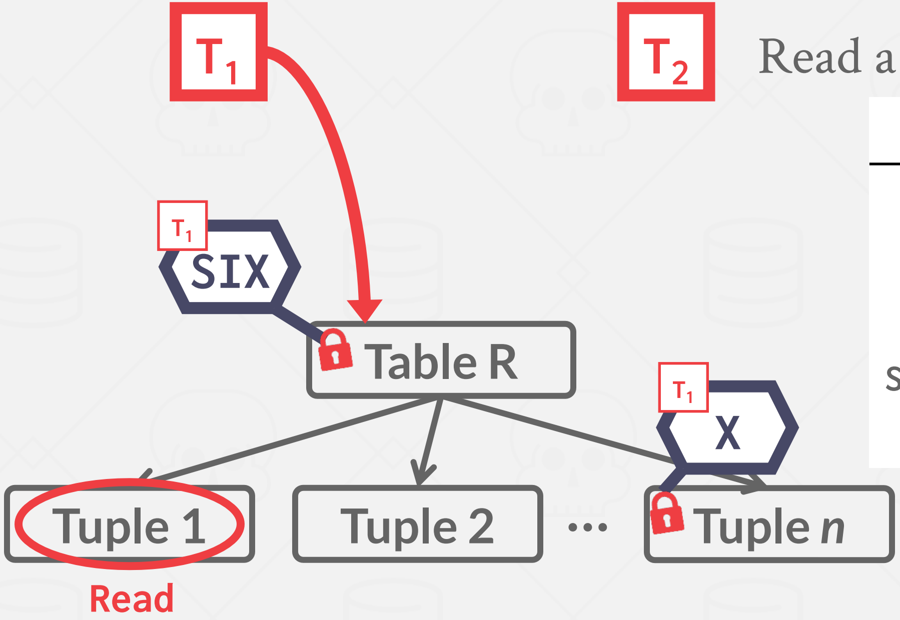
EXAMPLE – THREE TXNS



T_2 Read a single tuple in **R**.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

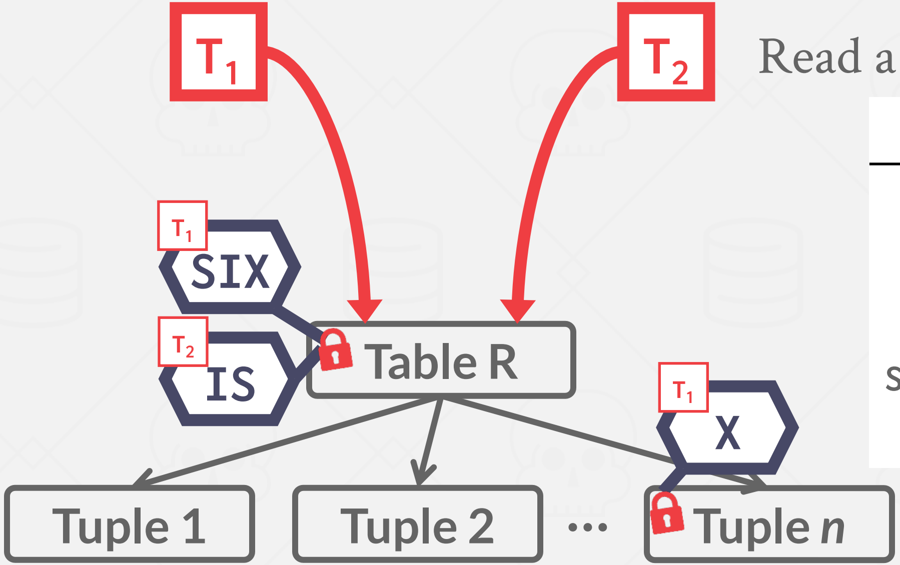
EXAMPLE – THREE TXNS



T_2 Read a single tuple in **R**.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

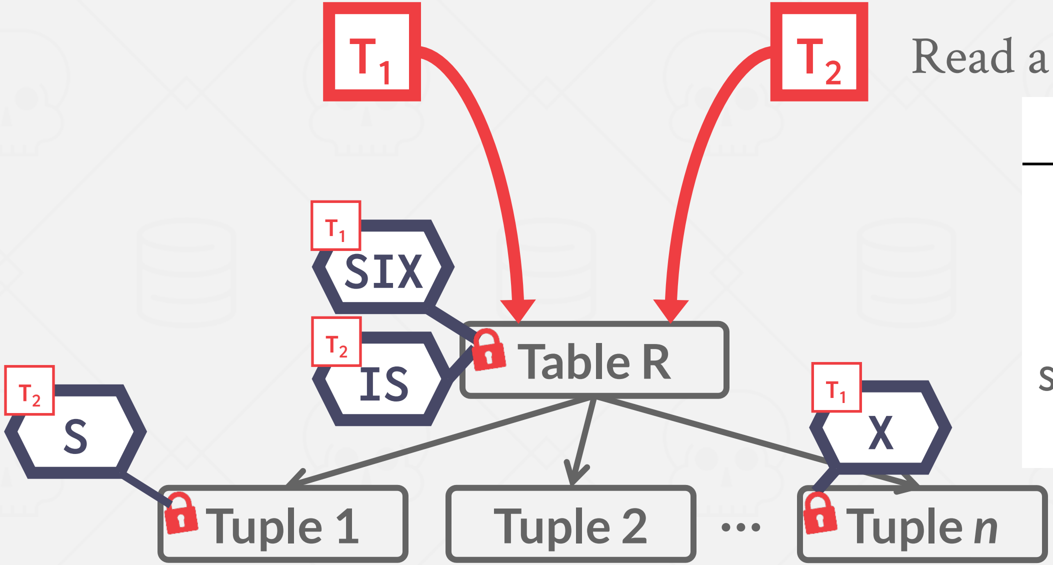
EXAMPLE – THREE TXNS



Read a single tuple in **R**.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – THREE TXNS

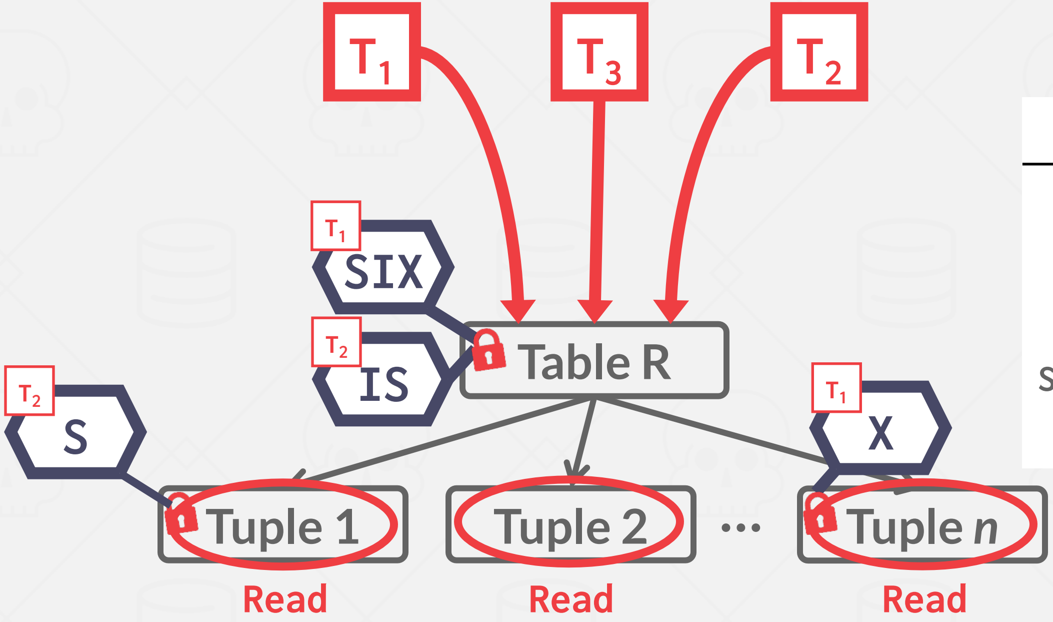


Read a single tuple in **R**.

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

EXAMPLE – THREE TXNS

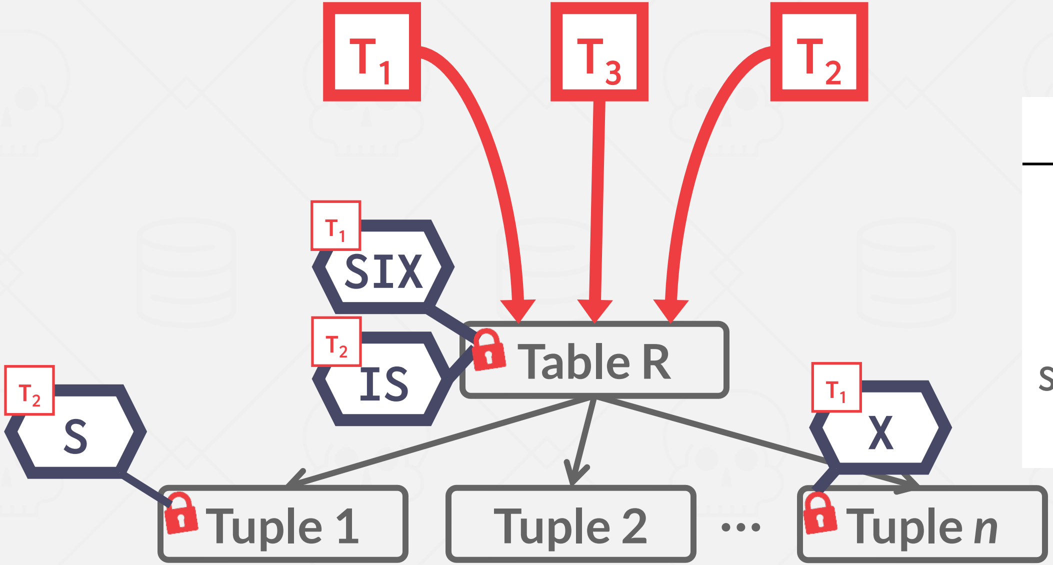
Scan all tuples in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – THREE TXNS

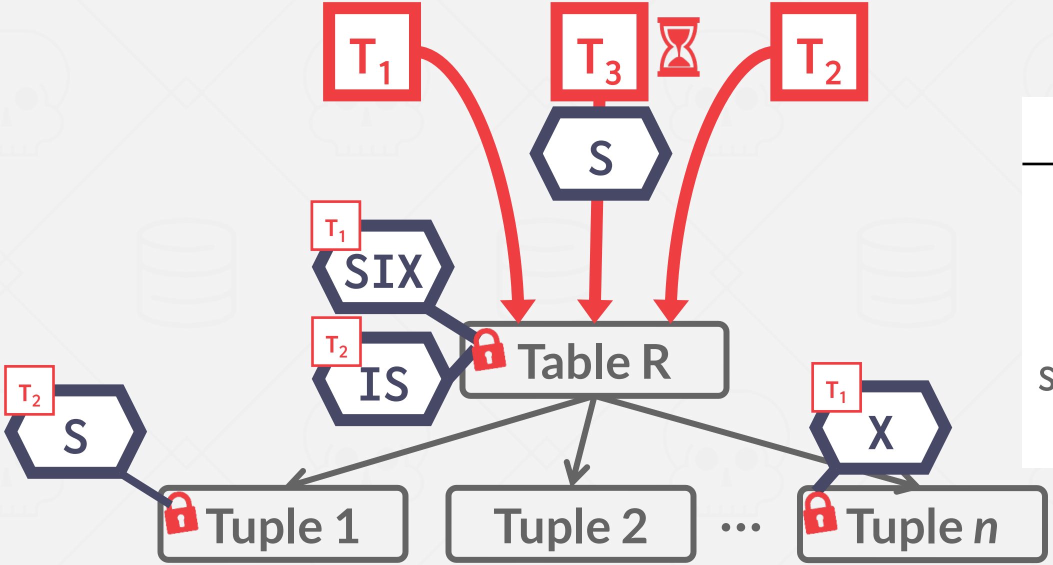
Scan all tuples in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – THREE TXNS

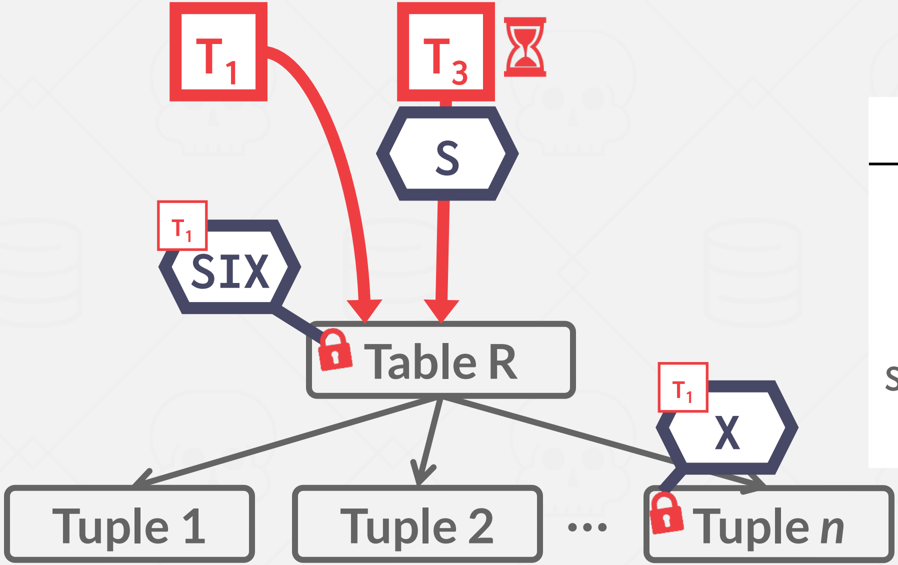
Scan all tuples in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

EXAMPLE – THREE TXNS

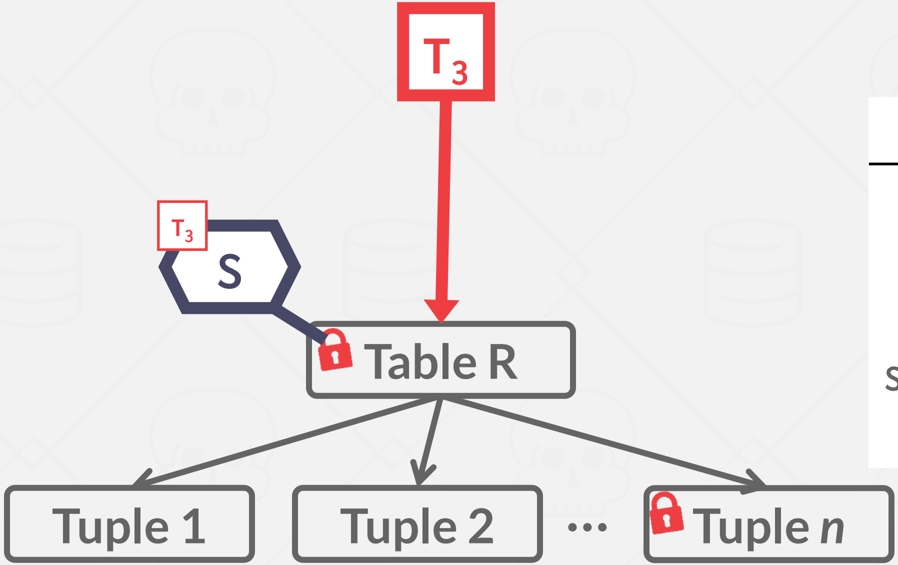
Scan all tuples in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

EXAMPLE – THREE TXNS

Scan all tuples in **R**.



	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

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 don't acquire a txn's locks manually (i.e., explicit SQL commands).

Sometimes you need to provide the DBMS with hints to help it to improve concurrency.

→ Update a tuple after reading it.

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

LOCK TABLE

Explicitly locks a table. Not part of the SQL standard.

- → Postgres/DB2/Oracle Modes: **SHARE**, **EXCLUSIVE**
- → MySQL Modes: **READ**, **WRITE**

```
LOCK TABLE <table> IN <mode> MODE;
```

```
SELECT 1 FROM <table> WITH (TABLOCK, <mode>);
```

```
LOCK TABLE <table> <mode>;
```

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**

```
SELECT * FROM <table>
 WHERE <qualification> FOR UPDATE;
```

CONCLUSION

2PL is used in almost every DBMS.

Automatically generates correct interleaving:

- → Locks + protocol (2PL, SS2PL ...)
- → Deadlock detection + handling
- → Deadlock prevention

NEXT CLASS

Timestamp Ordering Concurrency Control