KOÇ
UNIVERSITY

# Database Management Systems

## Concurrency Control

**M. Emre Gürsoy**

Assistant Professor
Department of Computer Engineering
www.memregursoy.com

# Introduction

- We have been talking about:
  - "Good" schedules versus "bad" schedules (e.g., conflict serializability)
  - How to check if a schedule is "good" or "bad"

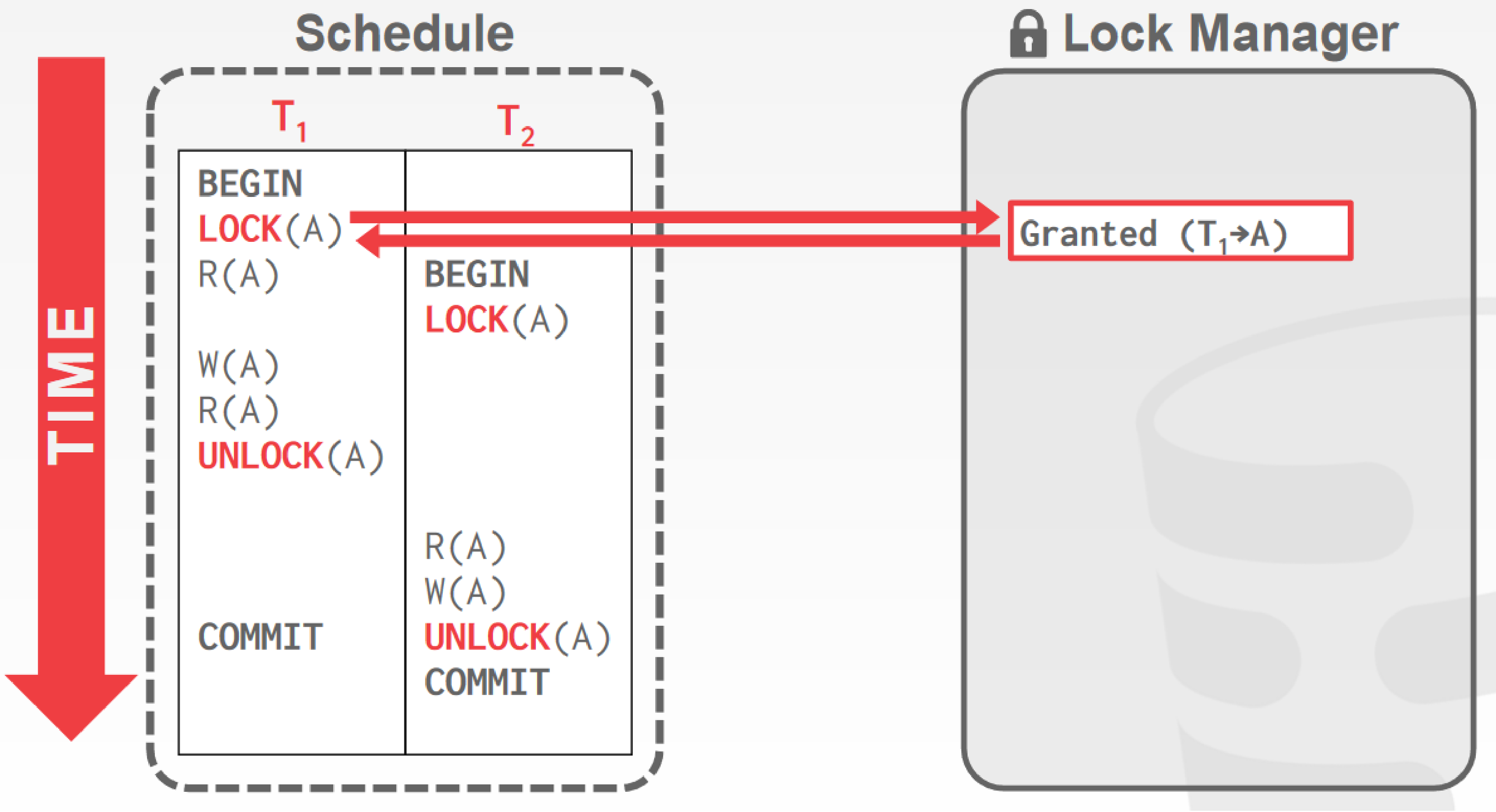
- But our previous methods have two shortcomings:
  - They require us to know all transactions and all actions ahead of time (all Rs and Ws)
  - They don't tell us how to create a “good” schedule

- Now we'll learn about **locking** for **concurrency control**

# Intuition

- Execution using locks:
    - Transactions request locks.
    - Lock manager grants or denies requests.
    - Once transactions are done with the DB object, they release their locks (unlock).
    - Lock manager keeps track of:
        - which transactions hold which locks on which objects
        - which transactions are waiting to acquire locks

# Intuition



TIME

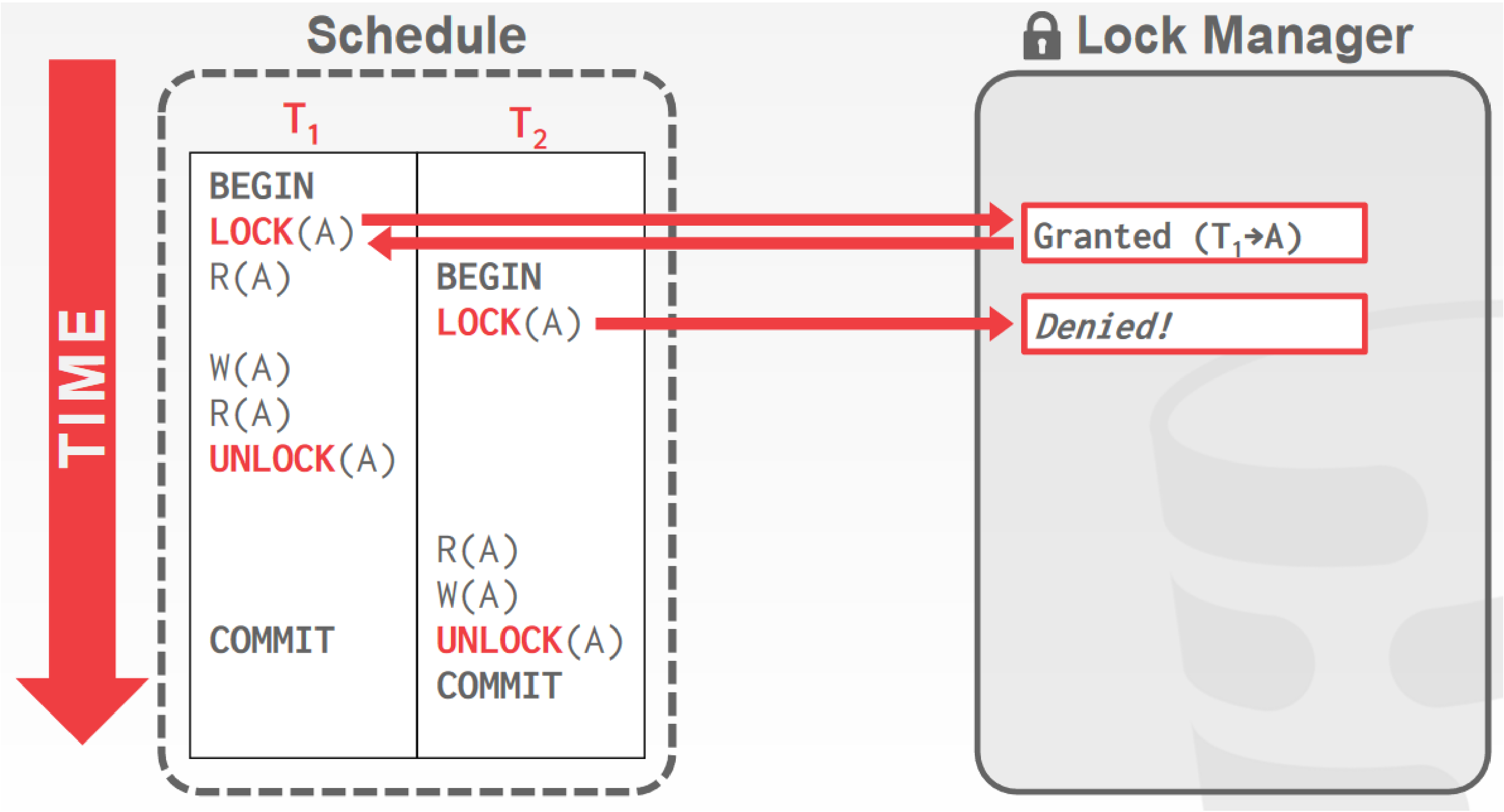
Schedule

| $T_1$ | $T_2$ |
|---|---|
| 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)

# Intuition

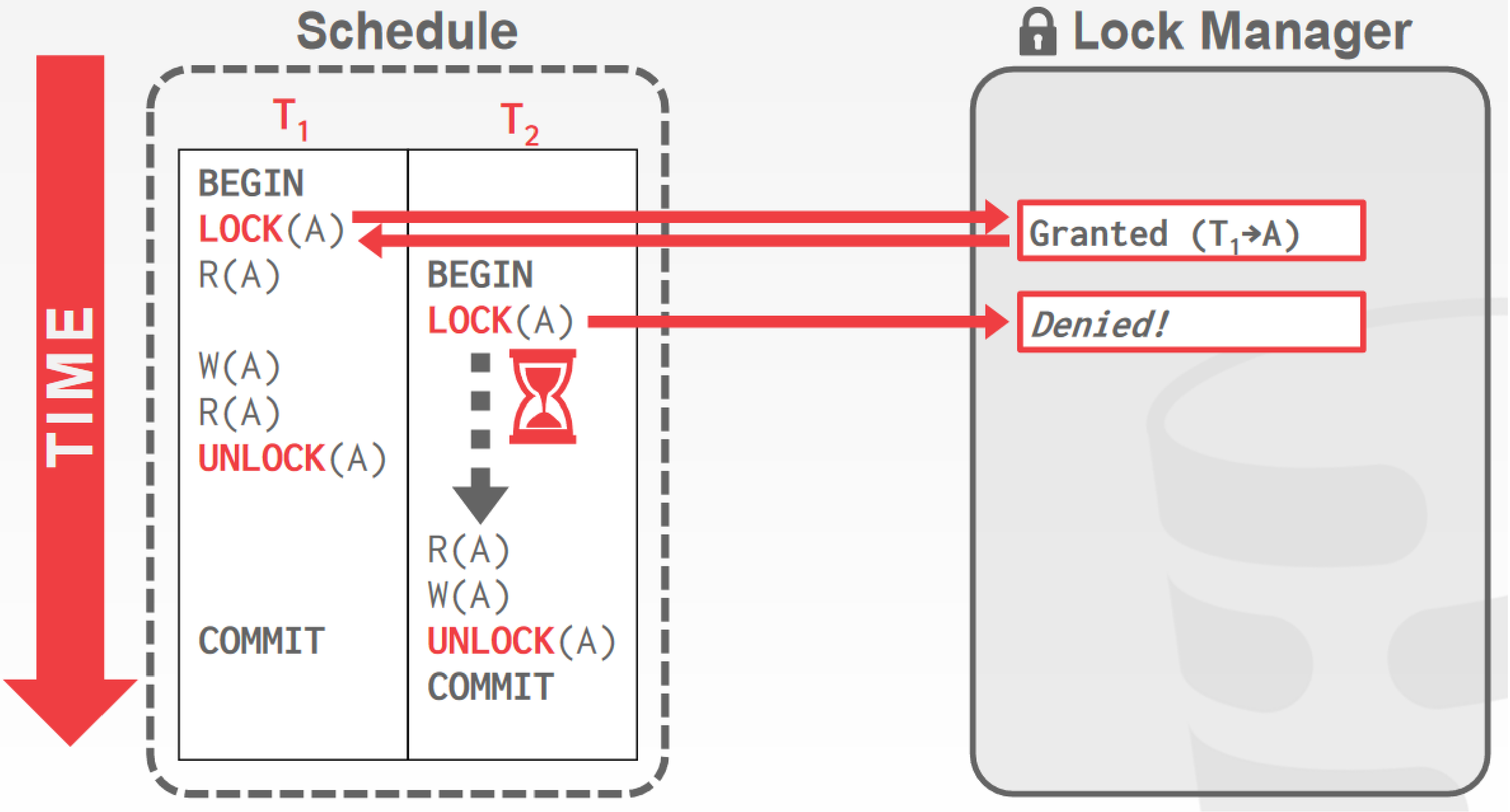


| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| LOCK(A) | |
| R(A) | BEGIN |
| | LOCK(A) |
| W(A) | |
| R(A) | |
| UNLOCK(A) | |
| | R(A) |
| | W(A) |
| COMMIT | UNLOCK(A) |
| | COMMIT |

Granted ($T_1$→A)

*Denied!*

# Intuition



Schedule

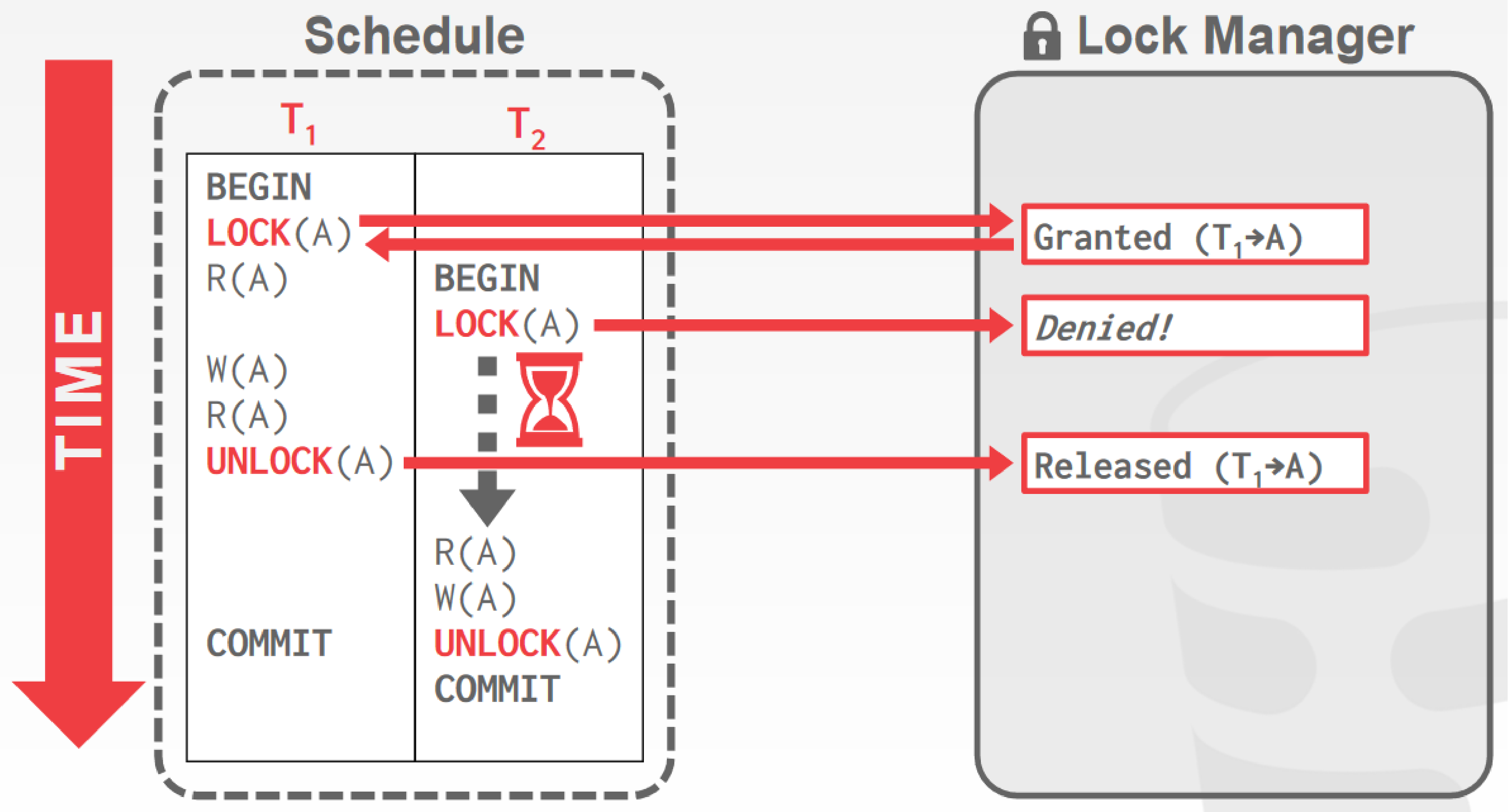
| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| LOCK(A) | |
| R(A) | BEGIN |
| | LOCK(A) |
| W(A) | |
| R(A) | |
| UNLOCK(A) | |
| | R(A) |
| | W(A) |
| COMMIT | UNLOCK(A) |
| | COMMIT |

Lock Manager

Granted ($T_1$→A)

*Denied!*

# Intuition



## 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 |

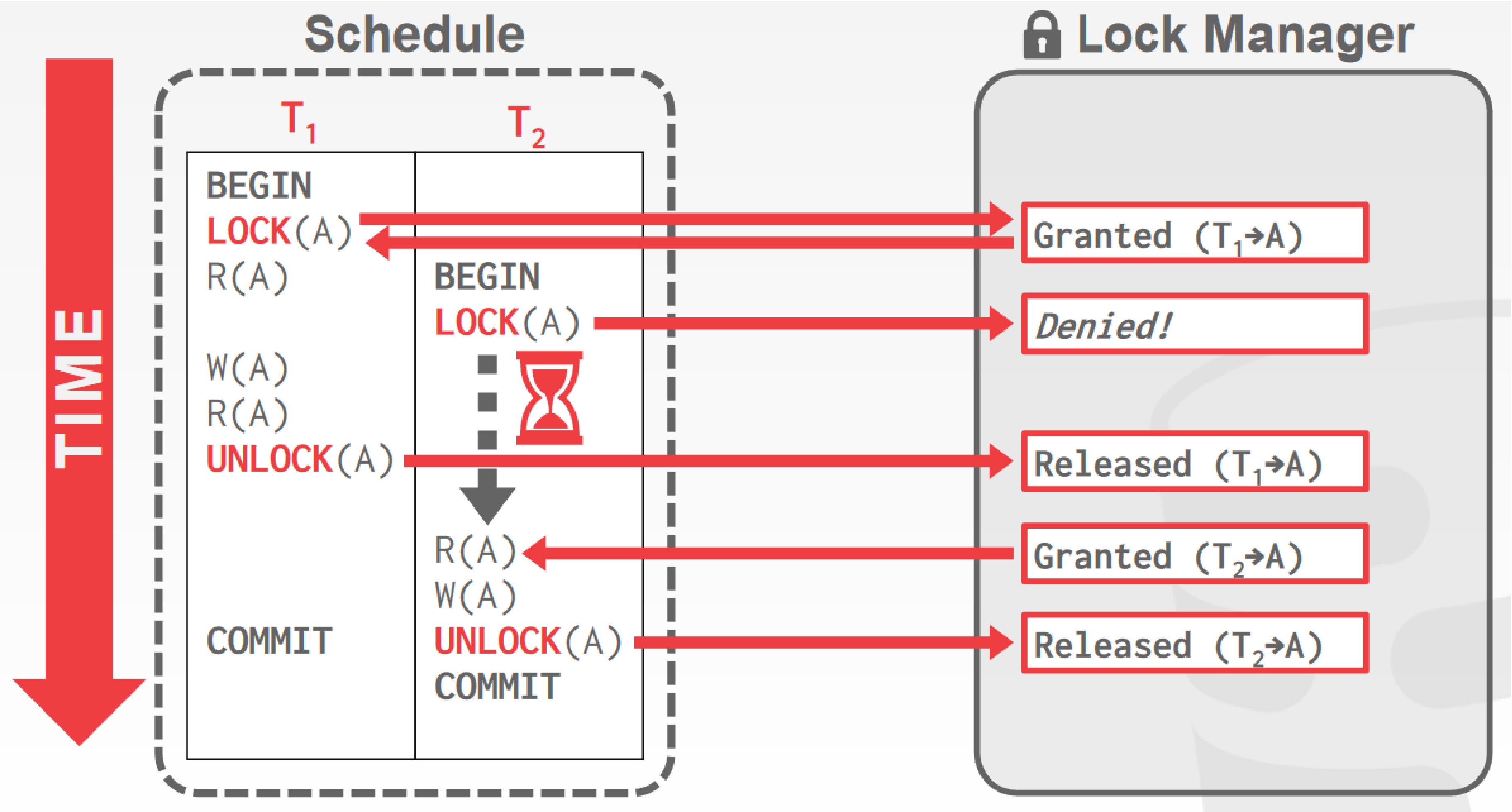
## Lock Manager

Granted ($T_1$→A)

*Denied!*

Released ($T_1$→A)

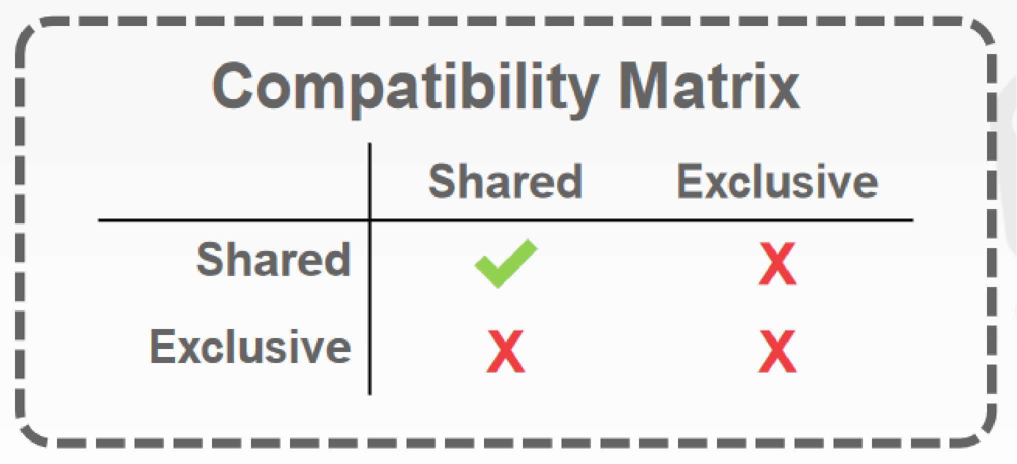
# Intuition


Schedule
Lock Manager
TIME
T1
T2
BEGIN
LOCK(A)
R(A)
BEGIN
LOCK(A)
W(A)
R(A)
UNLOCK(A)
R(A)
W(A)
COMMIT
UNLOCK(A)
COMMIT
Granted (T1→A)
Denied!
Released (T1→A)
Granted (T2→A)
Released (T2→A)

# Lock Types

- Two types of locks:
  - **Shared locks (S-lock):** for reads
  - **Exclusive locks (X-lock):** for writes

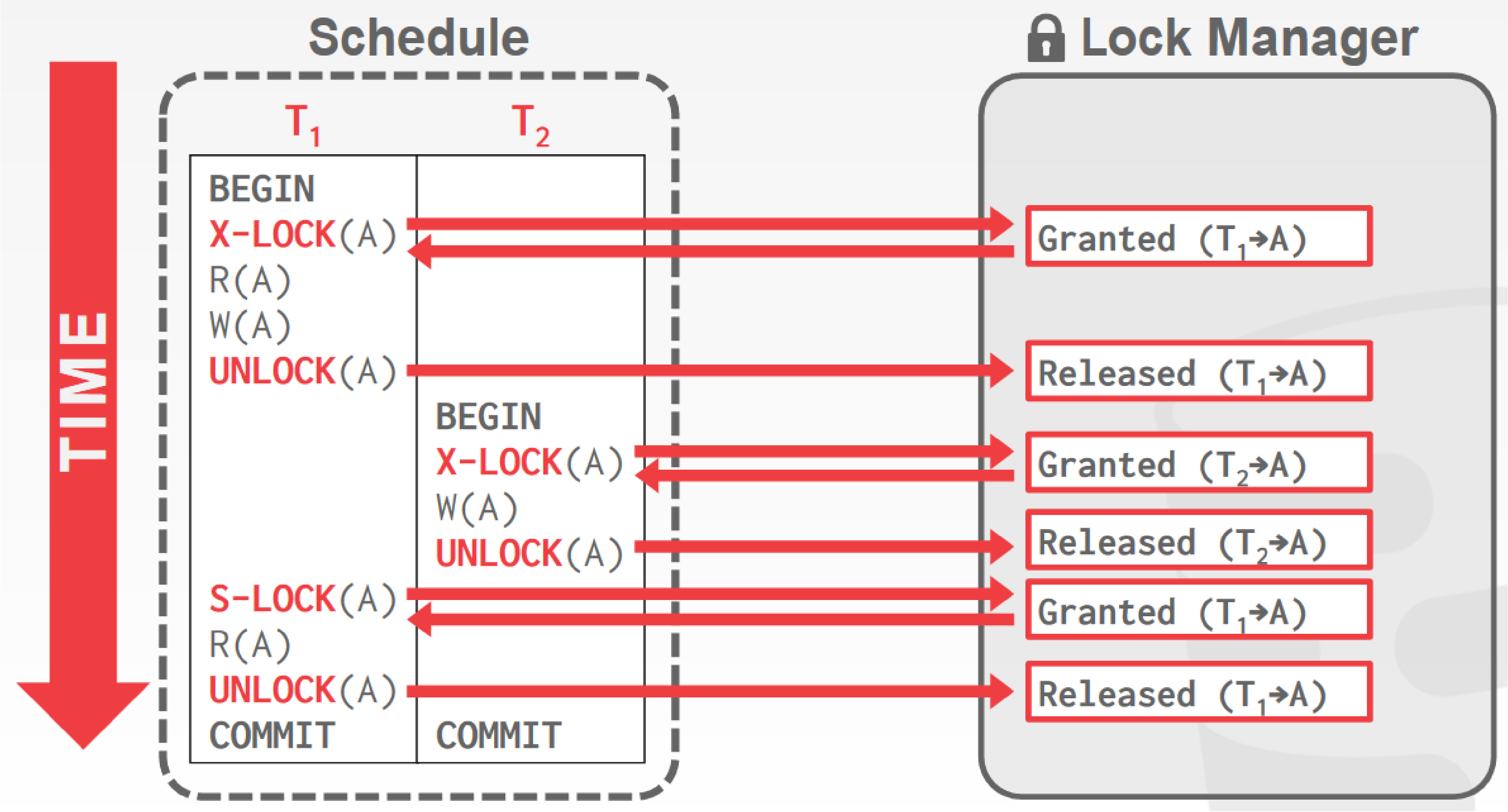
- Why not use shared locks for writes?
- Why not use exclusive locks for reads?



Compatibility Matrix

| | Shared | Exclusive |
|---|---|---|
| Shared | ✓ | X |
| Exclusive | X | X |

# 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)

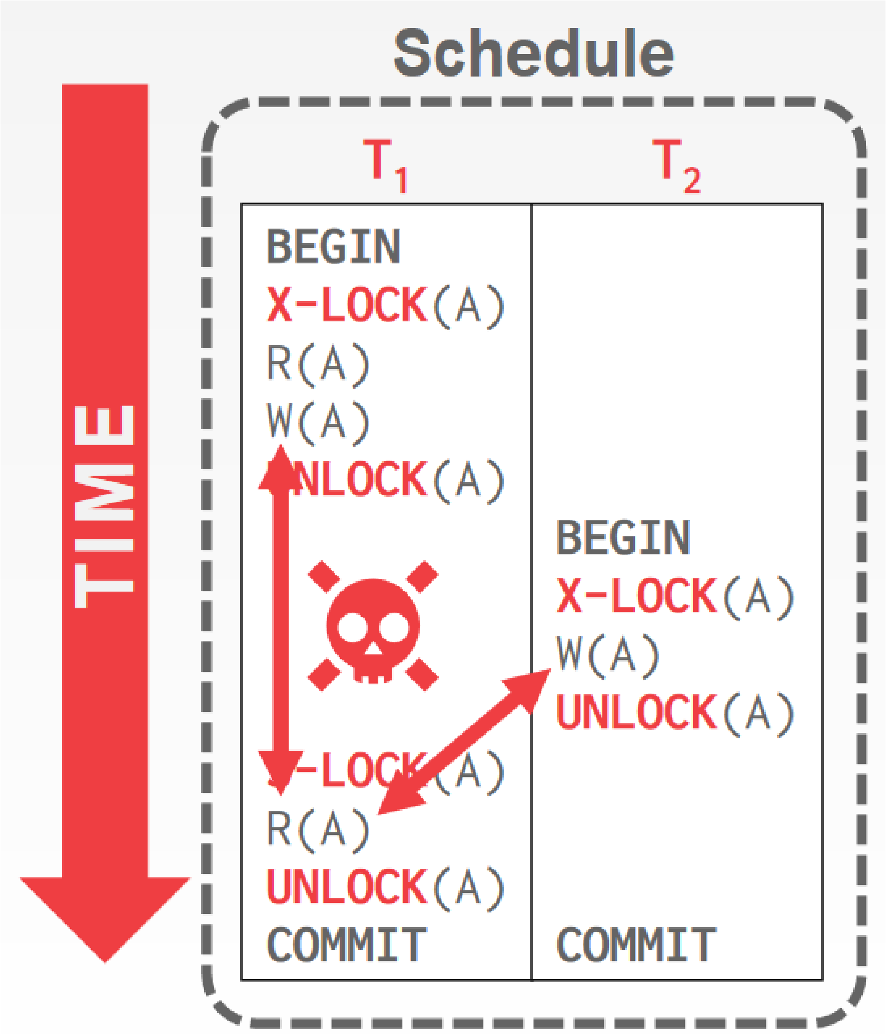
Released ($T_2$→A)

Granted ($T_1$→A)

Released ($T_1$→A)

**Do you see a problem here?**

# 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 |

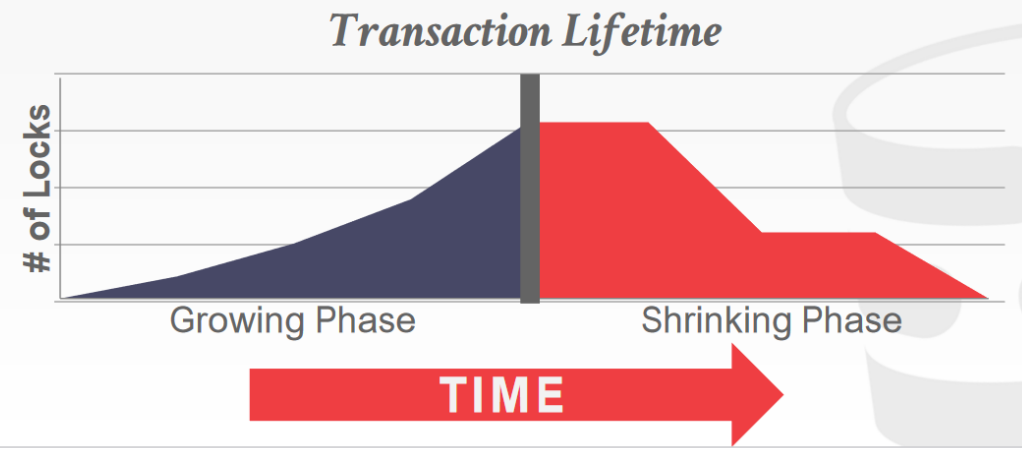
$T_1$ is expecting to read what it wrote but ends up reading what $T_2$ wrote.

What if $T_2$ decides to abort later?

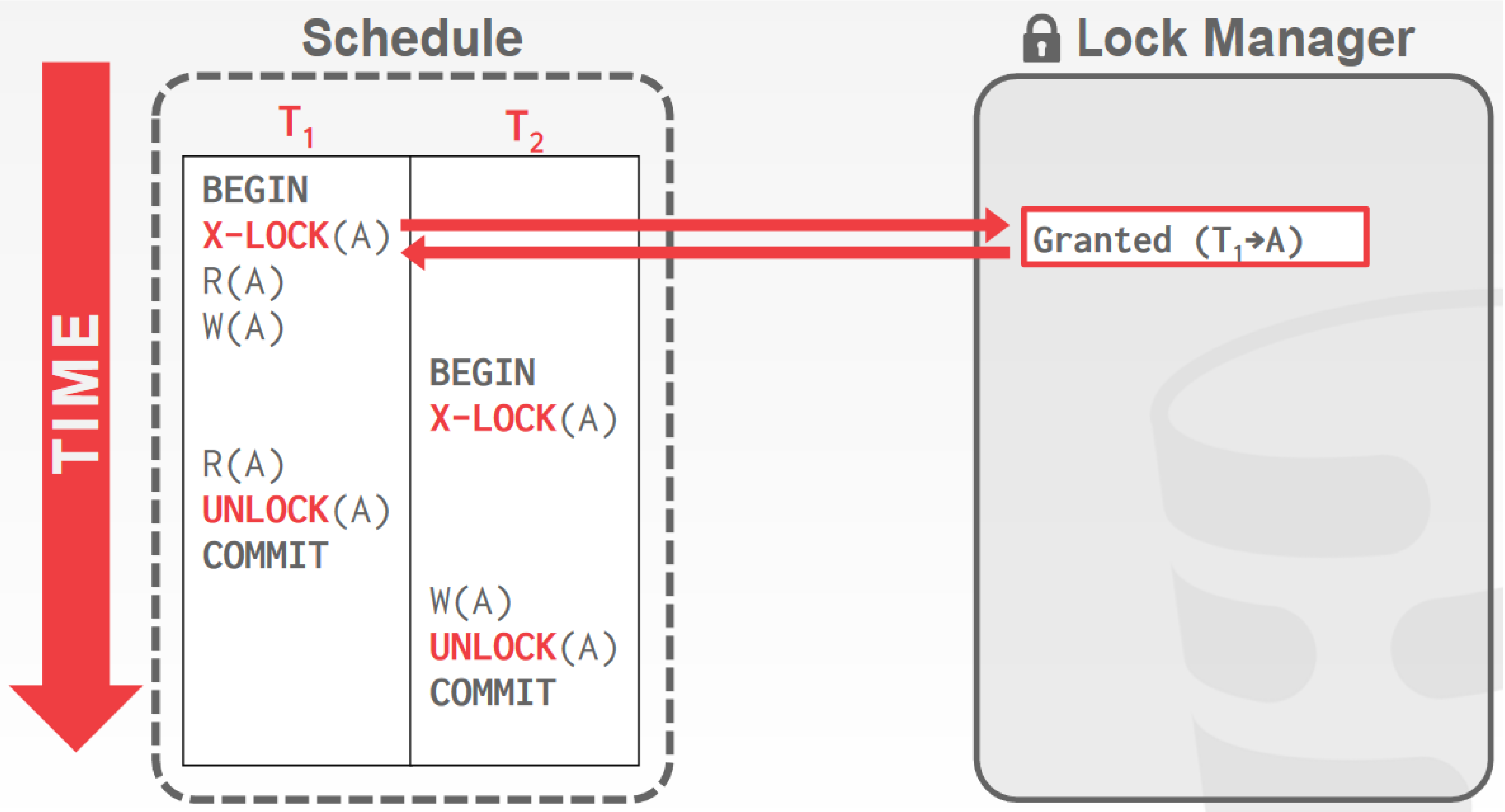
**Just using S-locks and X-locks is not sufficient, thus we need concurrency control protocols.**

# Two-Phase Locking

- **Two-phase locking (2PL)** is a commonly used concurrency control protocol.
  - Phase #1: **Growing**
    - Each transaction requests locks from DBMS lock manager.
    - Lock manager grants or denies lock requests.
  - Phase #2: **Shrinking**
    - The transaction is allowed to release previously acquired locks, but it is not allowed to acquire new locks.



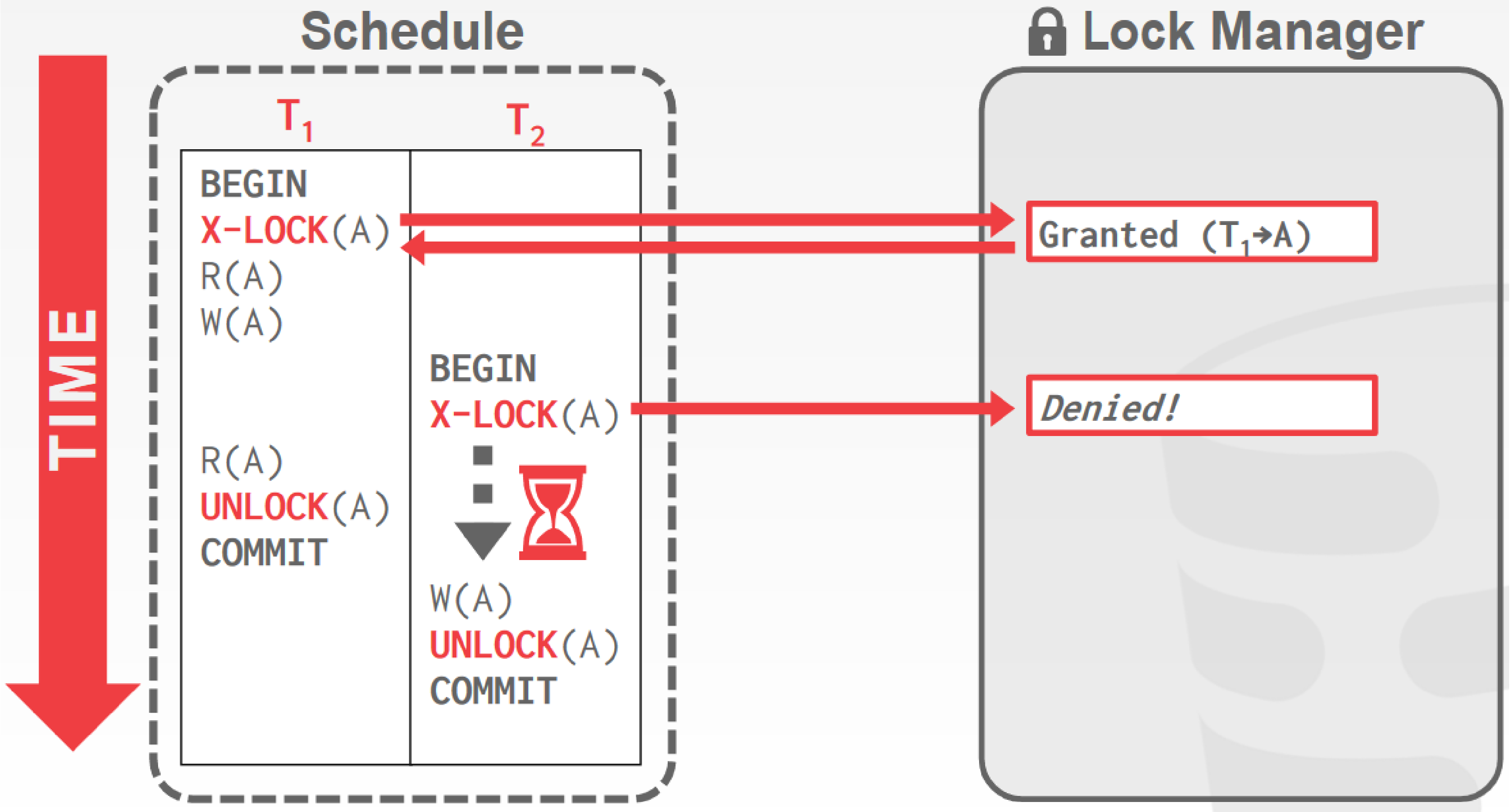
# Executing with 2PL



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

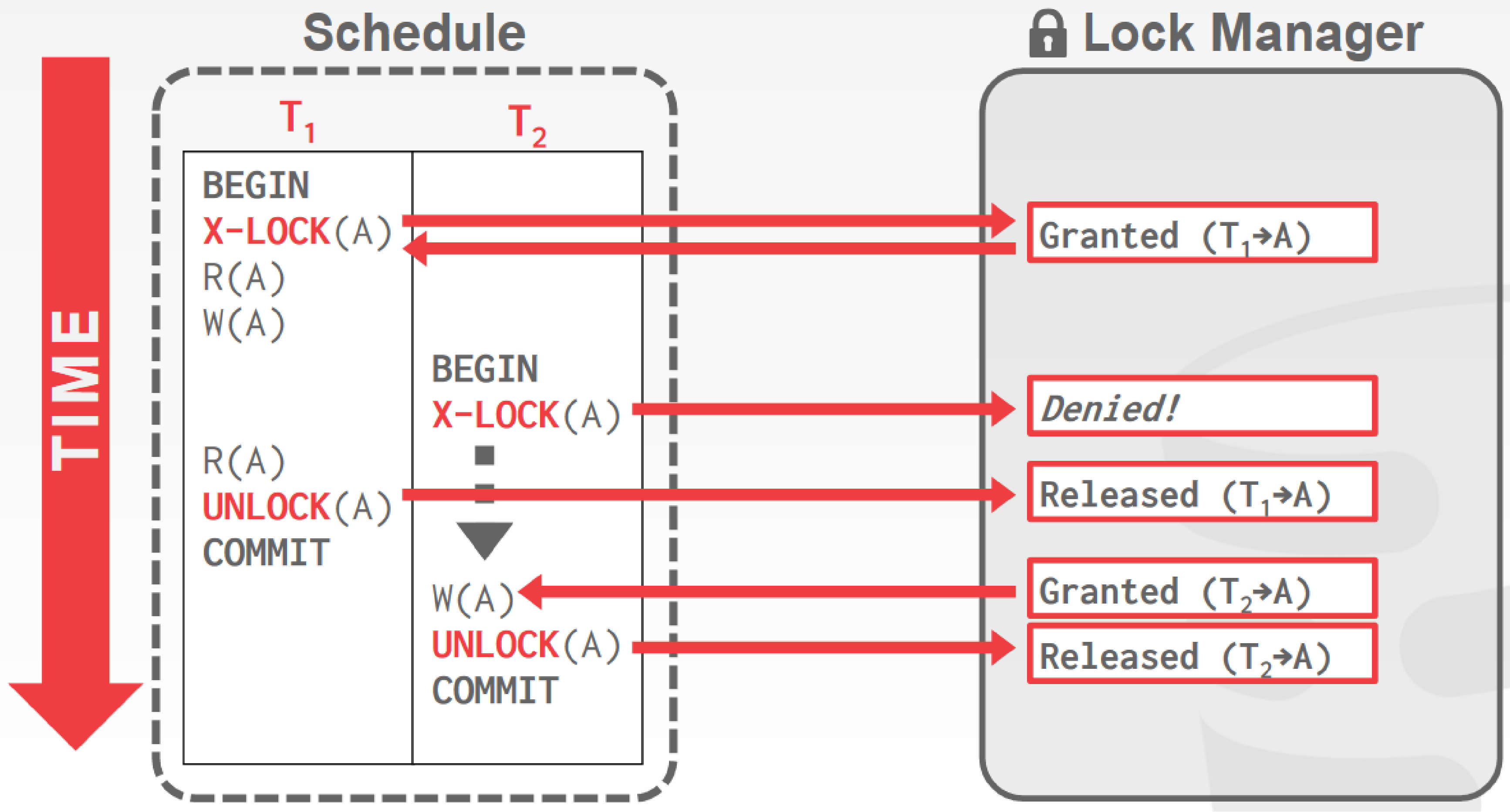
**What is the main difference with the previous case?**

# 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



TIME

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!*

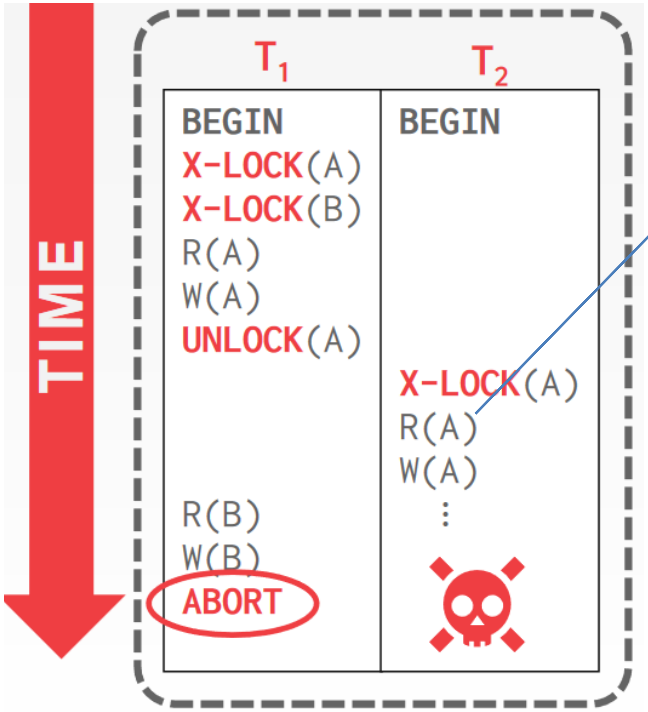
Released ($T_1$→A)

Granted ($T_2$→A)

Released ($T_2$→A)

# Cascading Aborts

- 2PL achieves conflict serializability for transactions that commit, but it is subject to the **cascading aborts** problem.
  - When a transaction aborts, it causes other transactions to also have to abort.



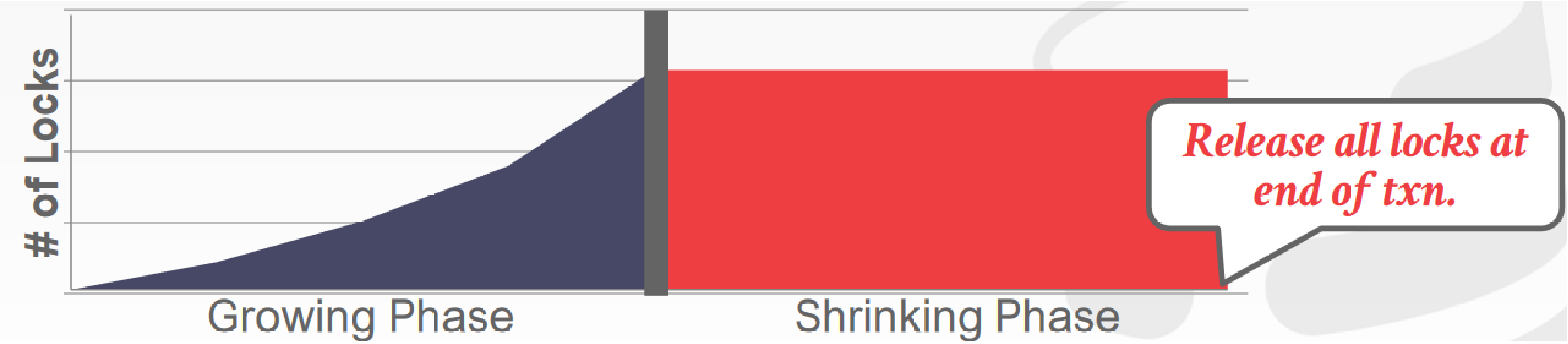
| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| X-LOCK(B) | |
| R(A) | |
| W(A) | |
| UNLOCK(A) | |
| | X-LOCK(A) |
| | R(A) |
| | W(A) |
| | ⋮ |
| R(B) | |
| W(B) | |
| ABORT | ☠ |

But $T_2$ had read what $T_1$ wrote and assumed that it was true!

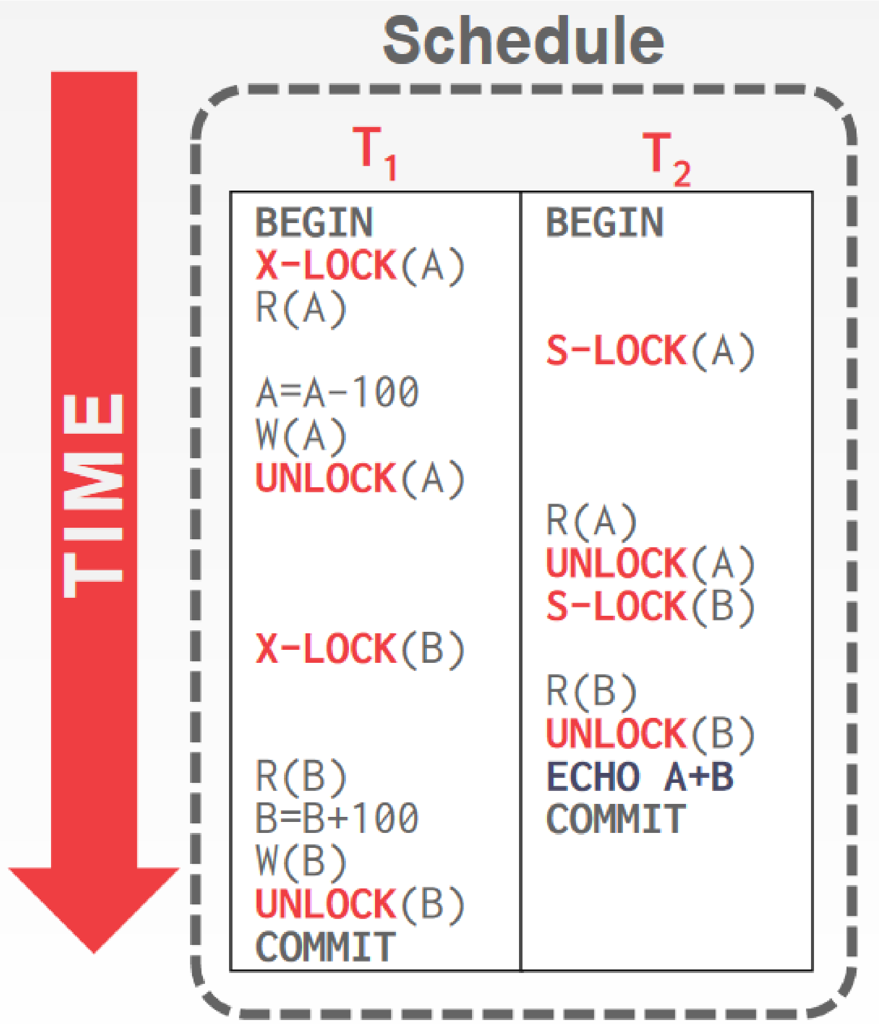
So, when $T_1$ aborts we must also abort $T_2$.

# Strict 2PL

- All locks held by a transaction are released when the transaction completes.
  - No unlocking in the middle of a transaction.
  - If I have an X-lock on A, noone else can get an S-lock or X-lock on A before I commit or abort.
  - Hence, noone can read/write on A before I am finished.
  - Hence, no cascading aborts.



# Example (Non-2PL)



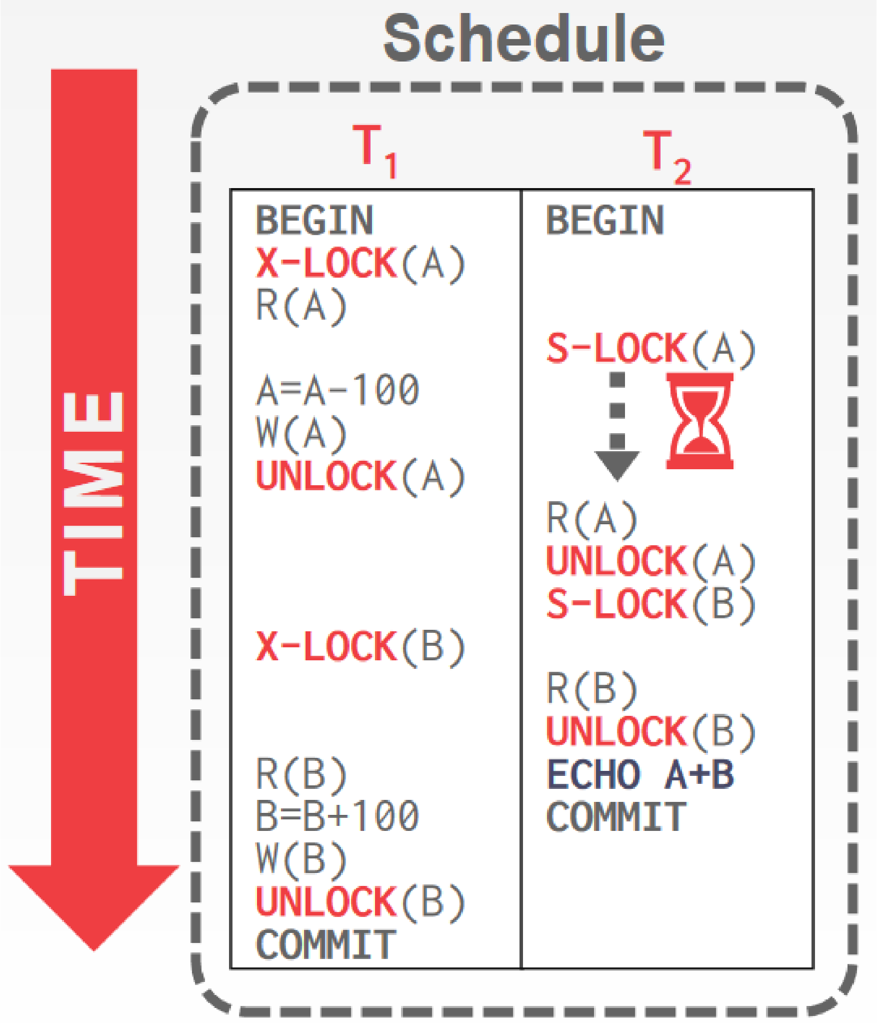
Schedule

| $T_1$ | $T_2$ |
|---|---|
| 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 | |

Assume A=1000, B=1000 initially.

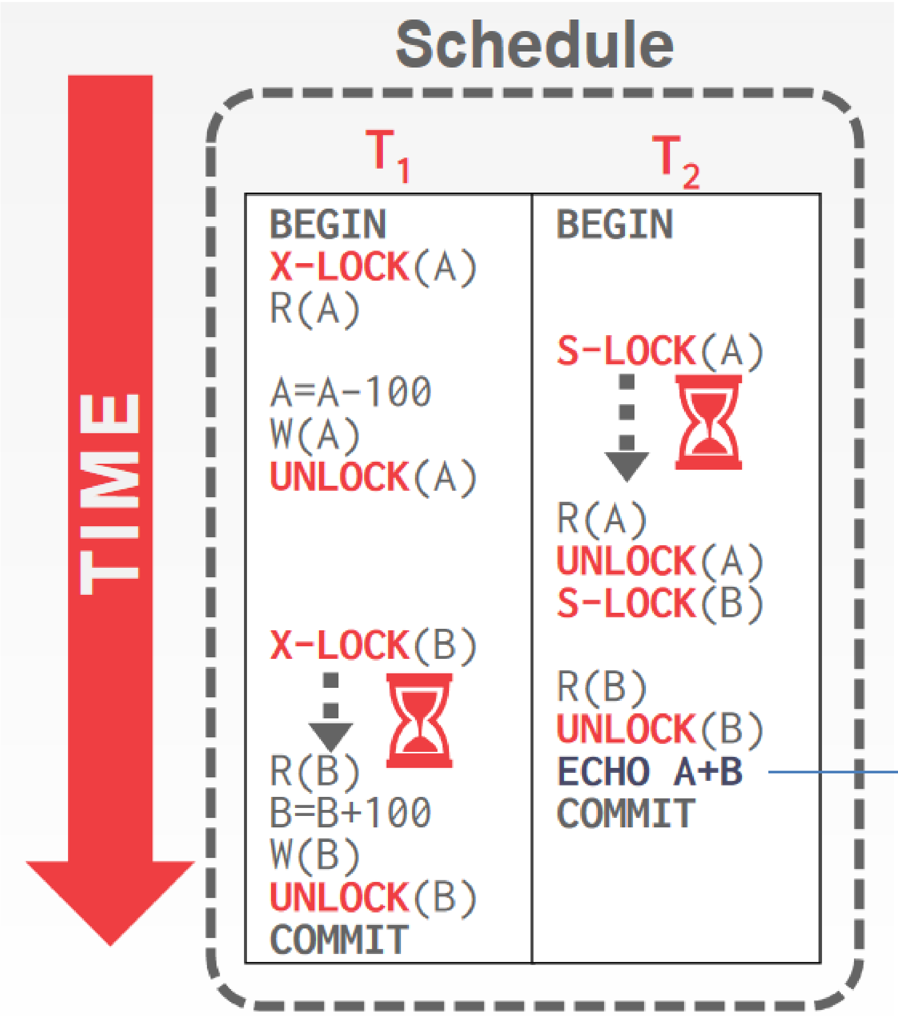
What are the two transactions intending to do?

# Example (Non-2PL)



| $T_1$ | $T_2$ |
|---|---|
| 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 | |

# Example (Non-2PL)



Schedule

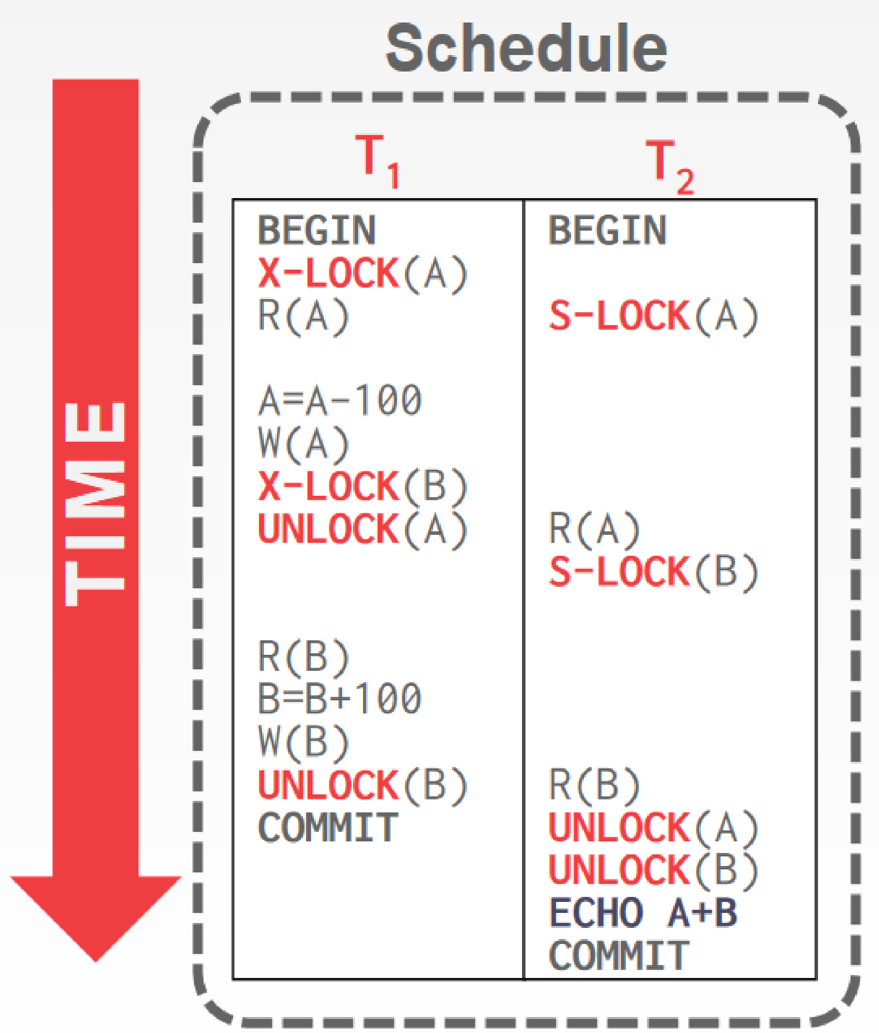
| T1 | T2 |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | |
| | S-LOCK(A) |
| A=A-100 | |
| W(A) | |
| UNLOCK(A) | |
| | R(A) |
| | UNLOCK(A) |
| | S-LOCK(B) |
| X-LOCK(B) | |
| | R(B) |
| | UNLOCK(B) |
| R(B) | ECHO A+B |
| B=B+100 | COMMIT |
| W(B) | |
| UNLOCK(B) | |
| COMMIT | |

Assume A=1000, B=1000 initially.

What gets printed out?

What are the final values of A and B?

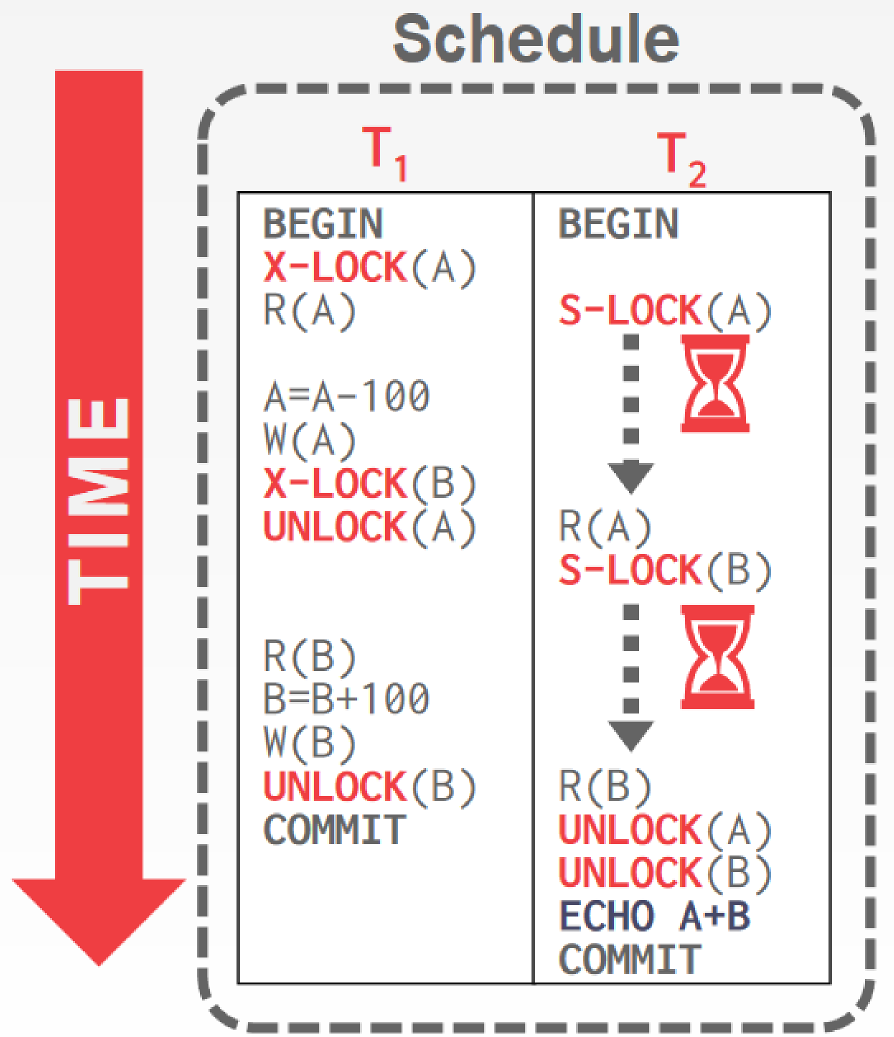
# Example (2PL)


Schedule
TIME

| $T_1$ | $T_2$ |
|---|---|
| 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 |

Assume A=1000, B=1000 initially.

# Example (2PL)

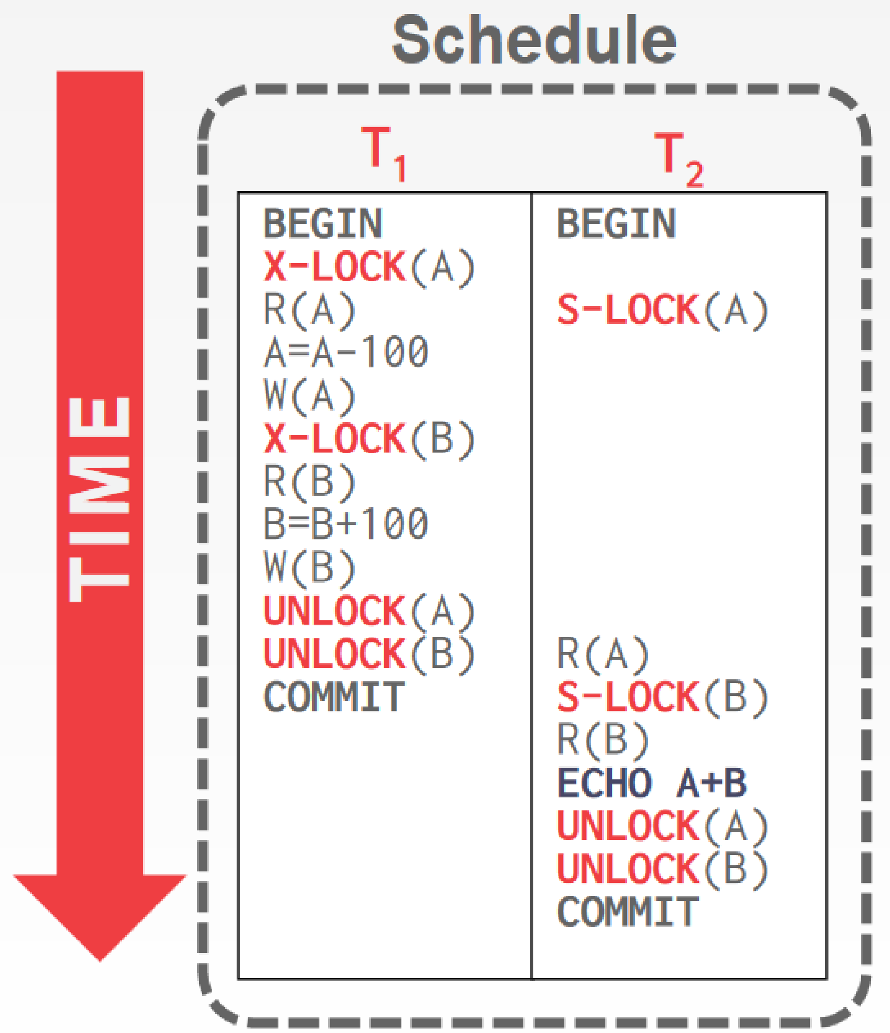


| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| R(A) | S-LOCK(A) |
| | |
| A=A-100 | |
| W(A) | |
| X-LOCK(B) | |
| UNLOCK(A) | R(A) |
| | S-LOCK(B) |
| | |
| R(B) | |
| B=B+100 | |
| W(B) | |
| UNLOCK(B) | R(B) |
| COMMIT | UNLOCK(A) |
| | UNLOCK(B) |
| | ECHO A+B |
| | COMMIT |

What gets printed out?

What are the final values of A and B?

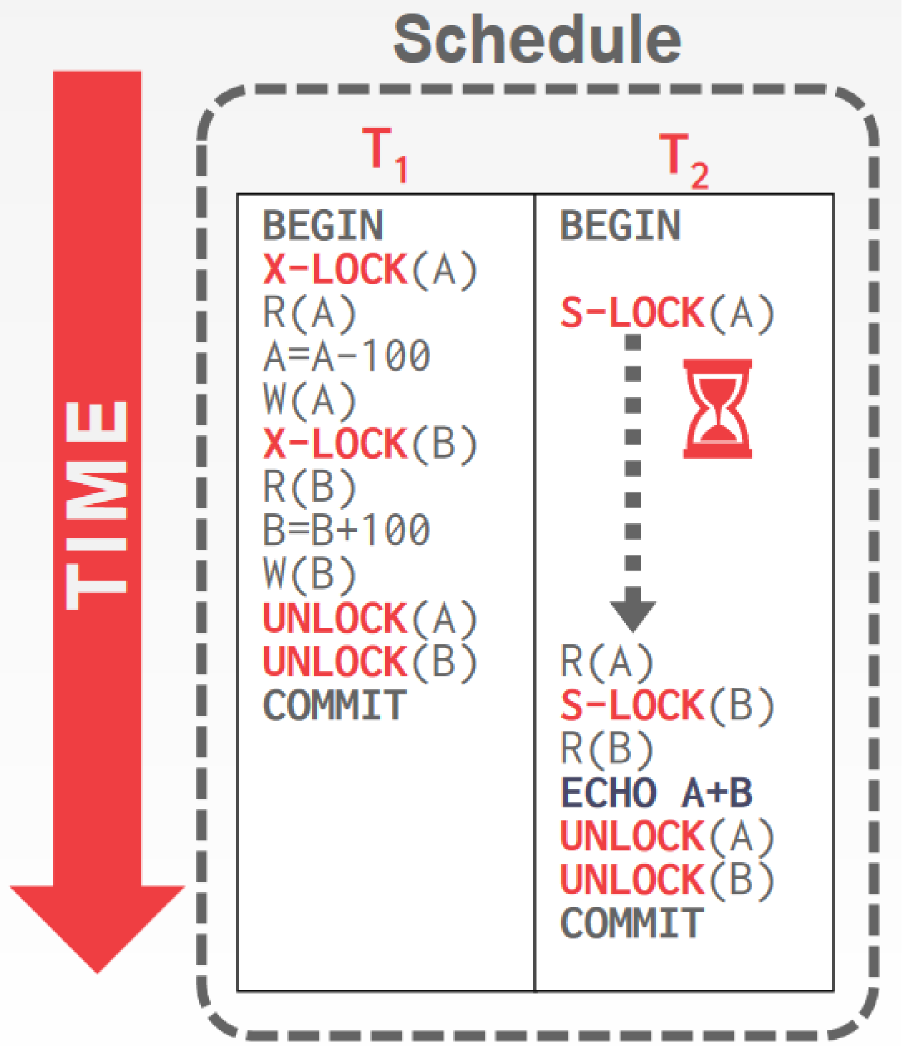
# Example (Strict 2PL)



| $T_1$ | $T_2$ |
|---|---|
| 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 |

Assume A=1000, B=1000 initially.

# Example (Strict 2PL)



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 |

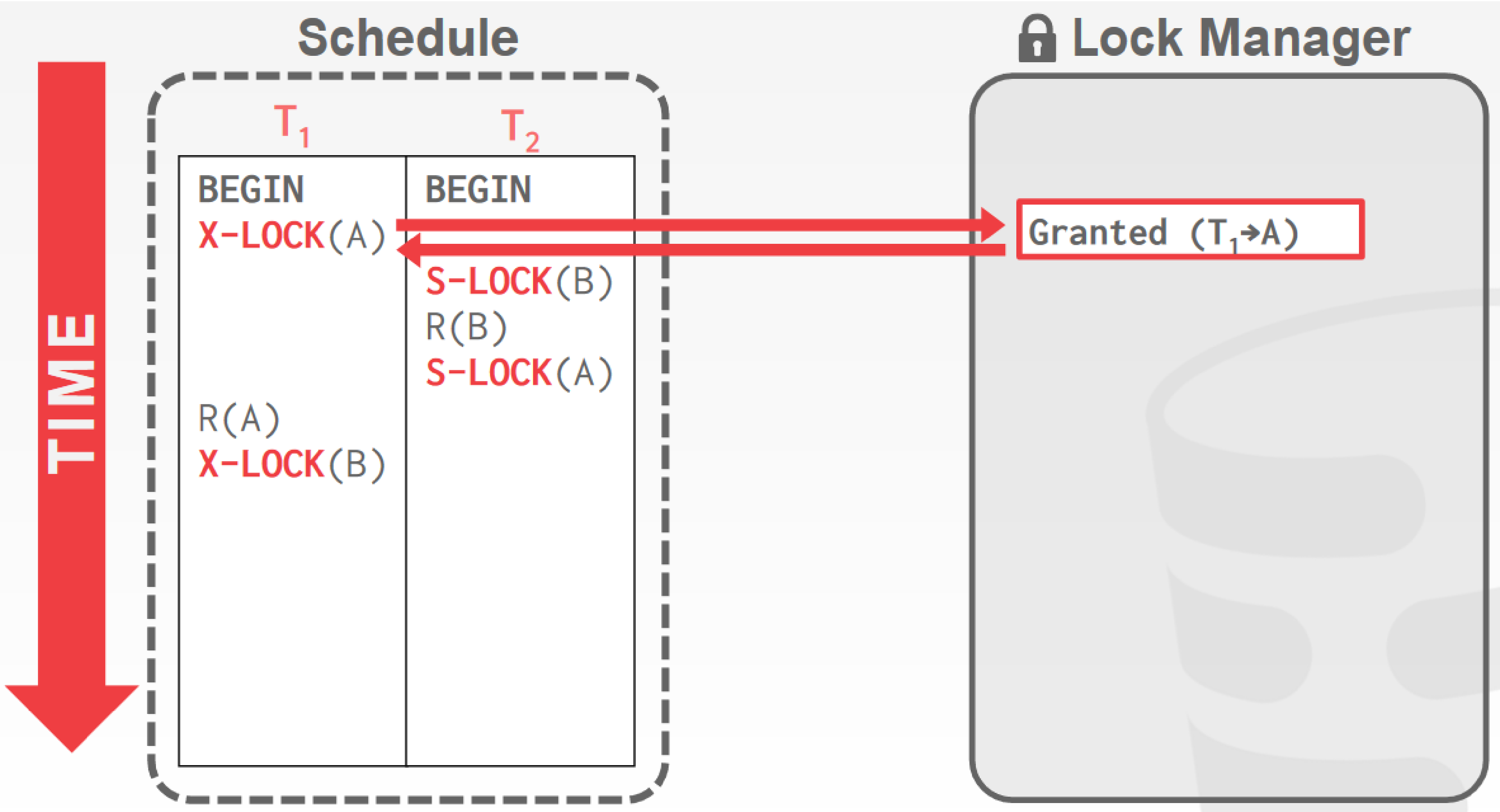
What gets printed out?

What are the final values of A and B?

# Deadlocks

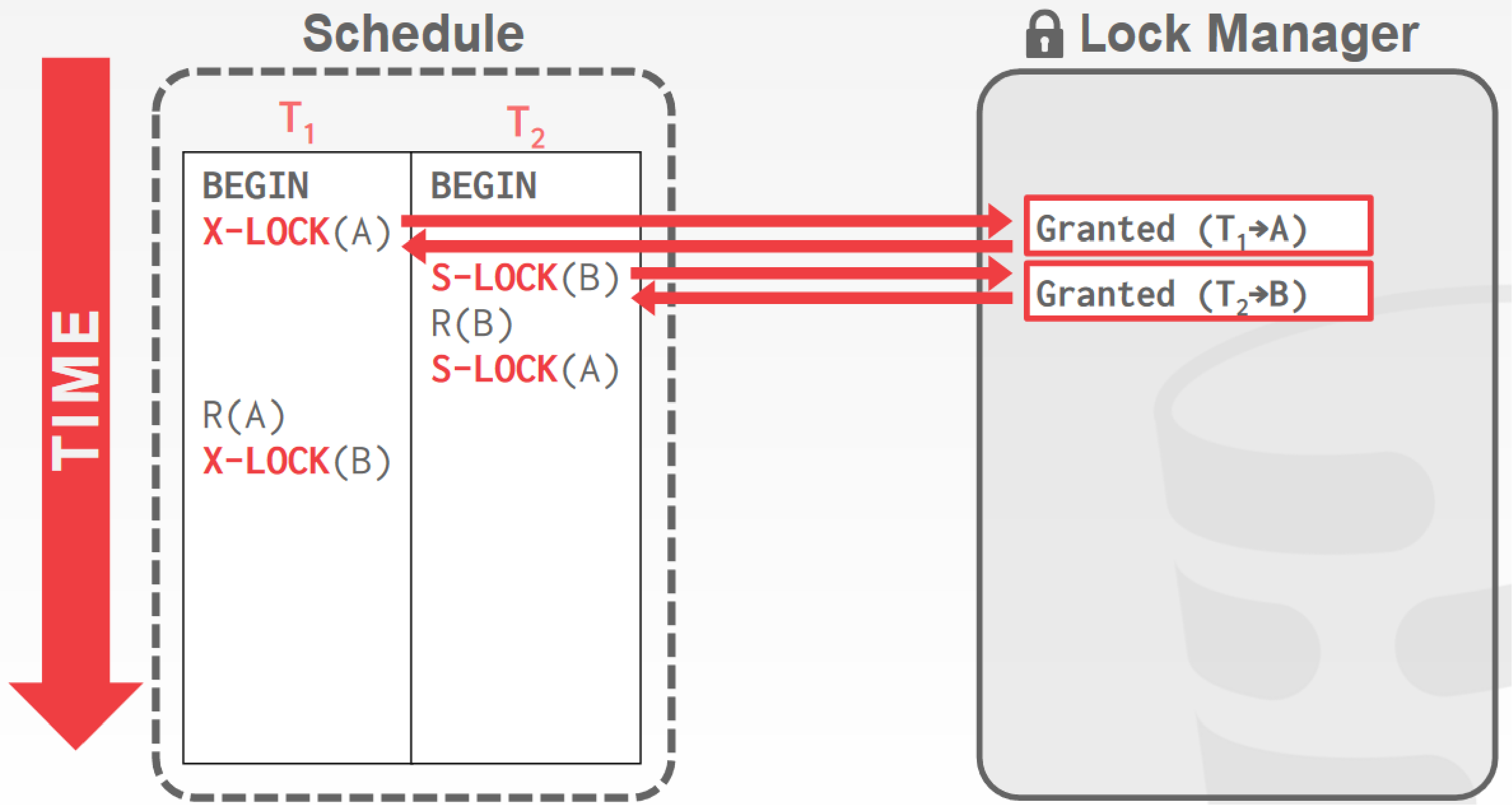
- 2PL and strict 2PL can lead to deadlocks.
- A **deadlock** is a cycle of transactions waiting for locks to be released by each other.
  - $T_1$ locks A, needs access to B in order to continue.
  - $T_2$ locks B, needs access to A in order to continue.
- Two issues related to deadlocks:
  - Deadlock detection
  - Deadlock prevention

# Deadlocks



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

# Deadlocks

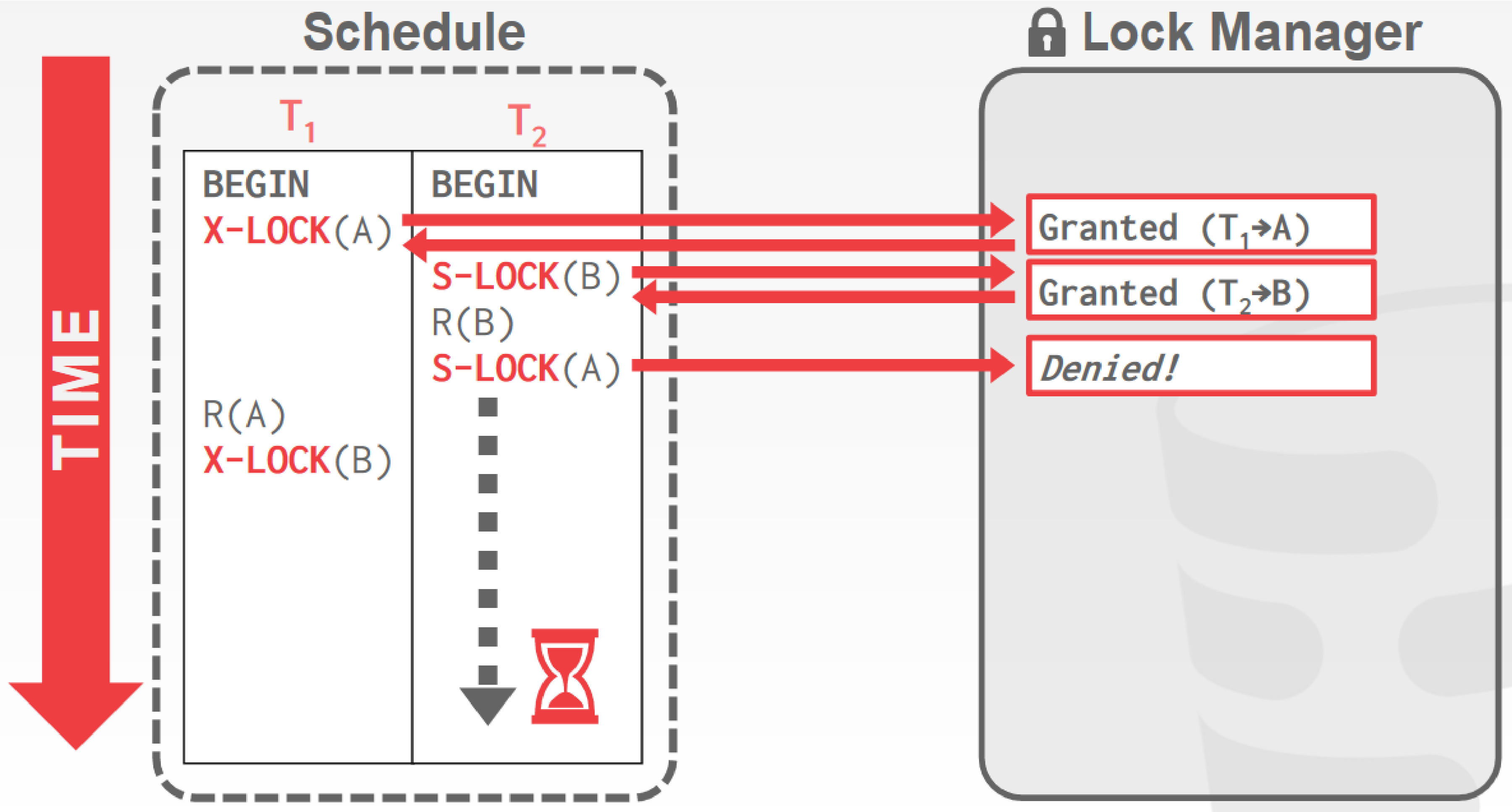


| $T_1$ | $T_2$ |
|---|---|
| BEGIN | BEGIN |
| X-LOCK(A) | |
| | S-LOCK(B) |
| | R(B) |
| | S-LOCK(A) |
| R(A) | |
| X-LOCK(B) | |

Granted ($T_1$→A)

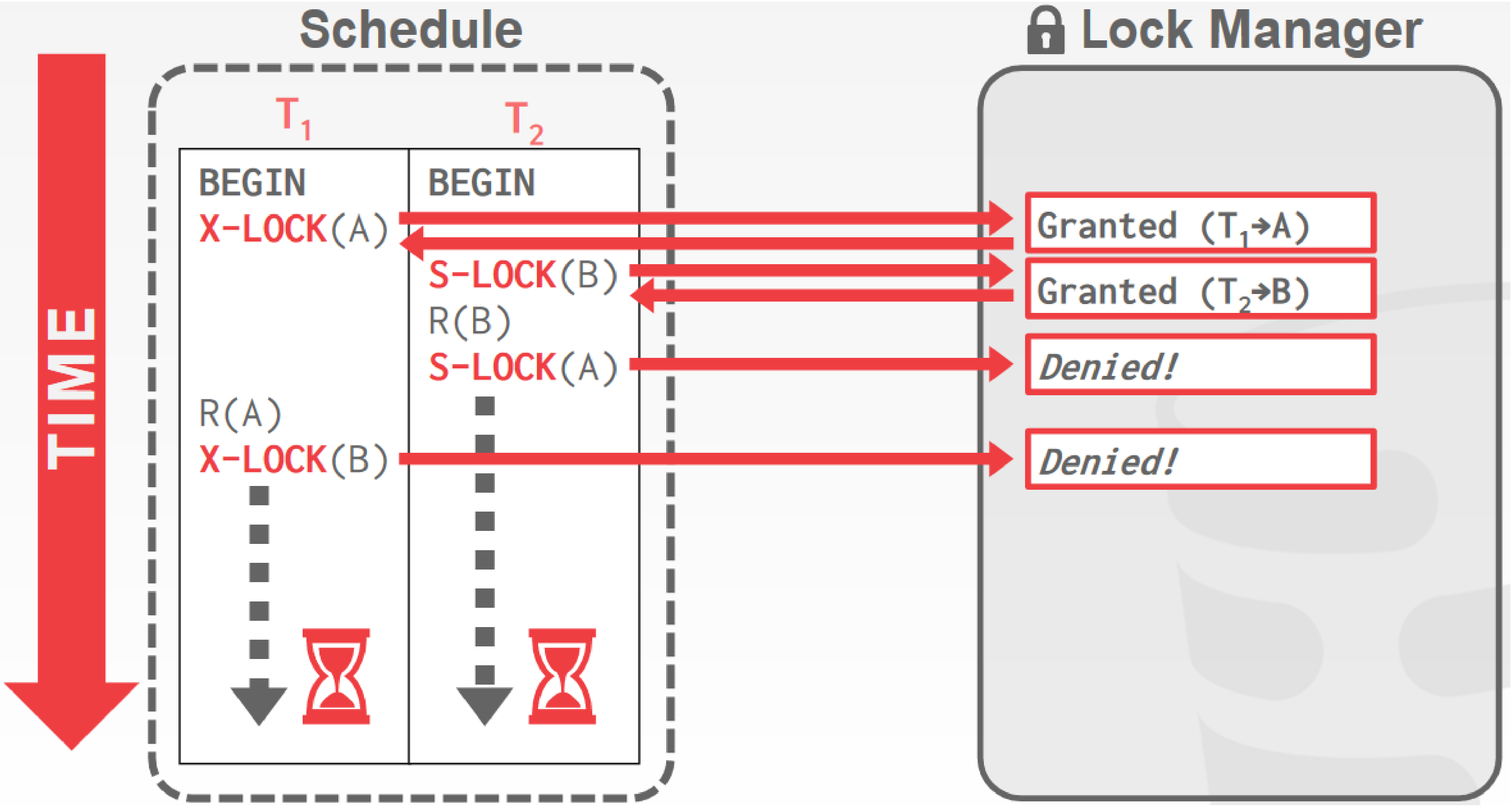
Granted ($T_2$→B)

# Deadlocks



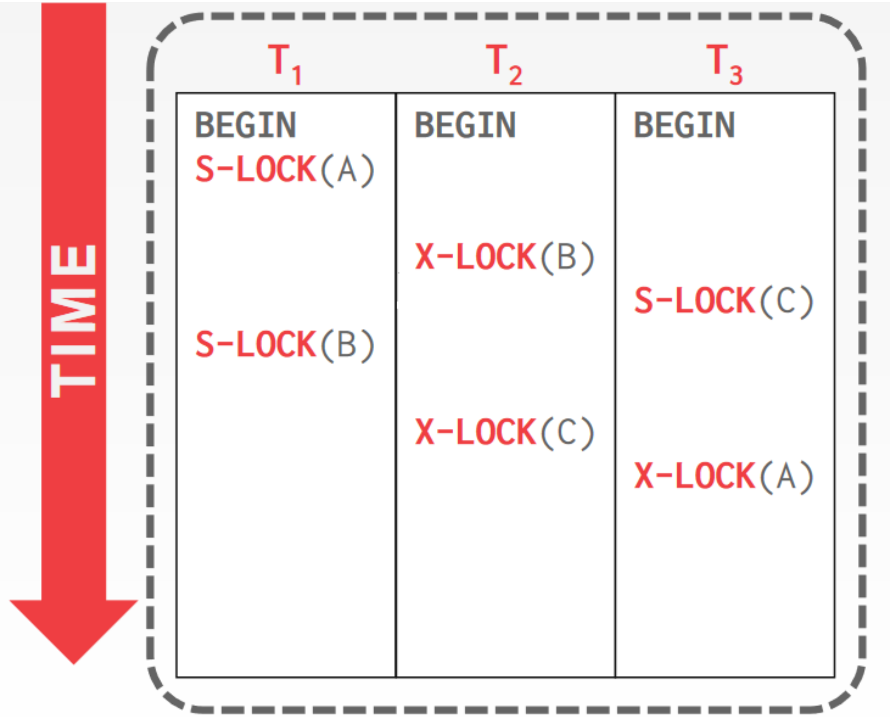
Schedule
TIME
T1
T2
BEGIN
X-LOCK(A)
R(A)
X-LOCK(B)
BEGIN
S-LOCK(B)
R(B)
S-LOCK(A)
Lock Manager
Granted (T1→A)
Granted (T2→B)
Denied!

# Deadlocks

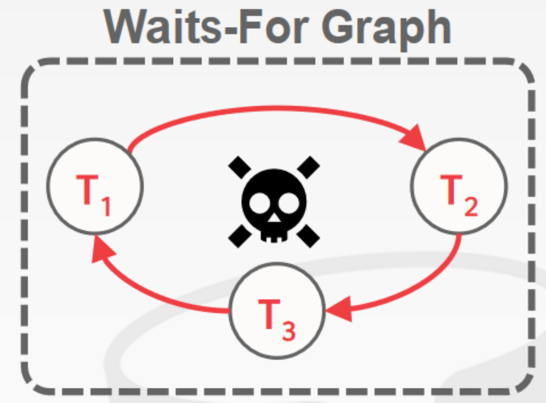

Schedule
Lock Manager
TIME
T1
T2
BEGIN
BEGIN
X-LOCK(A)
S-LOCK(B)
R(B)
S-LOCK(A)
R(A)
X-LOCK(B)
Granted (T1→A)
Granted (T2→B)
Denied!
Denied!

# Deadlock Detection

- We create a **waits-for graph**:
  - Each node is a transaction
  - Edge from $T_i$ to $T_j$ if $T_i$ is waiting for $T_j$ to release a lock
- Cycle in the graph means there's a deadlock



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



# Deadlock Detection

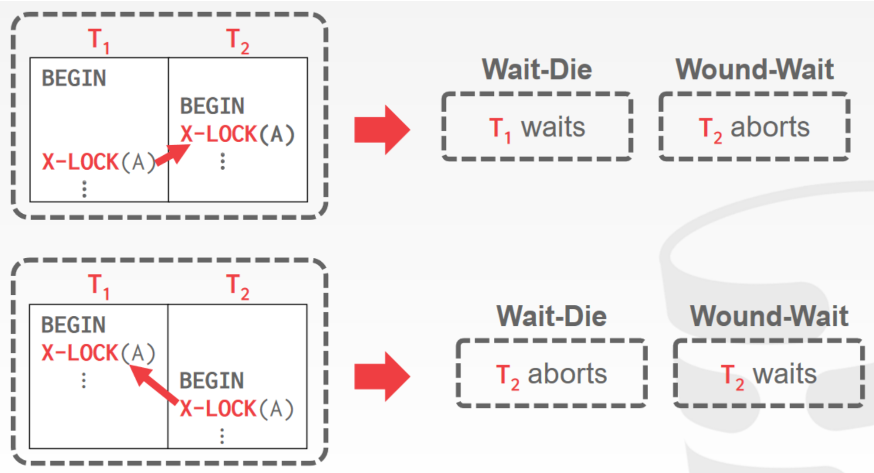
- When the DBMS detects a deadlock using the waits-for graph, it will select a victim transaction to break the cycle.
    - The victim is forced to restart or abort.

- How to select the proper victim? Possible heuristics:
    - By age (e.g., lowest timestamp)
    - By progress (e.g., fewest queries executed)
    - # of objects locked
    - ...

- We should keep in mind whether a transaction has been selected as a victim before, to avoid starvation.

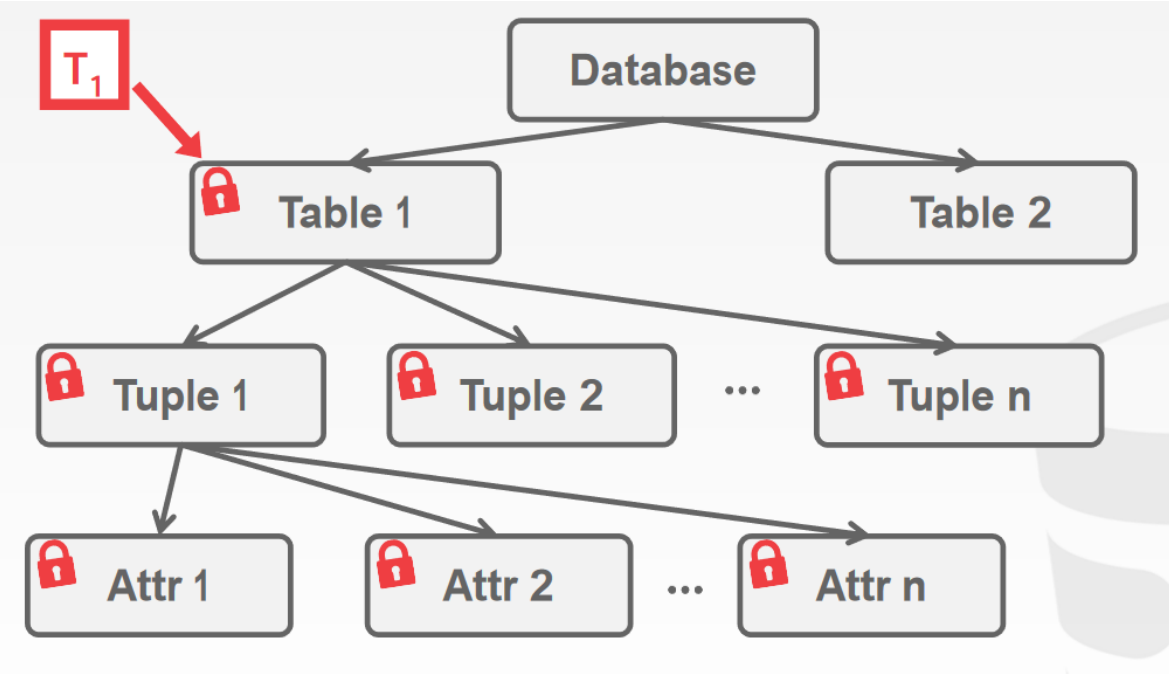
# Deadlock Prevention

- When $T_i$ tries to acquire a lock that is held by $T_j$, the DBMS "stops" one of them to prevent a deadlock.
  - Doesn't require a waits-for graph
- How to determine which transaction to stop?
  - **Wait-Die:** If $T_i$ started earlier than $T_j$, then $T_i$ waits for $T_j$. Otherwise $T_i$ aborts.
  - **Wound-Wait:** If $T_i$ started earlier than $T_j$, then $T_j$ aborts and releases lock. Otherwise $T_i$ waits.



# Multi-Granularity Locking

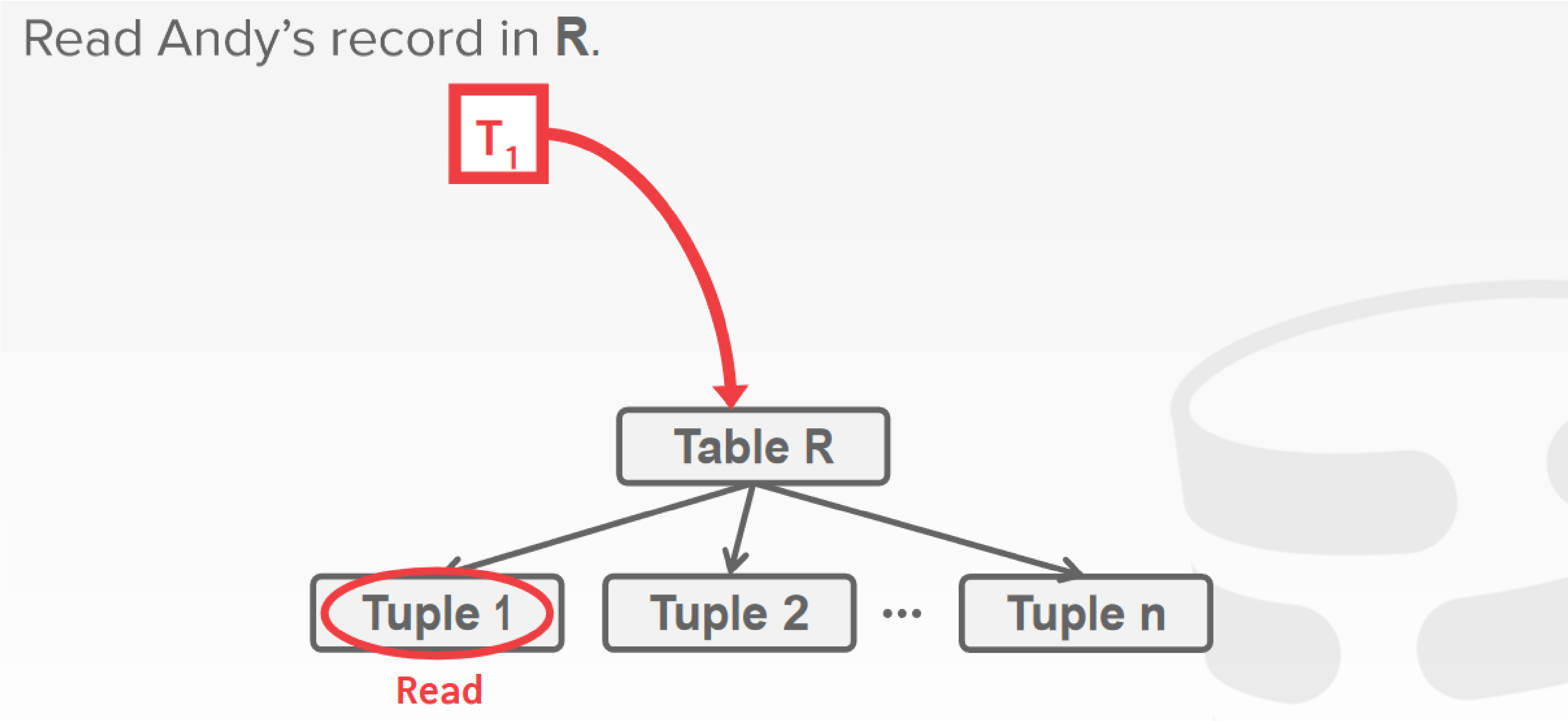
- What are these database objects we have been locking?
  - Tables? Tuples? Attributes?
  - Locking too coarse: many conflicts, deadlocks
  - Locking too granular: can be inefficient



# Multi-Granularity Locking

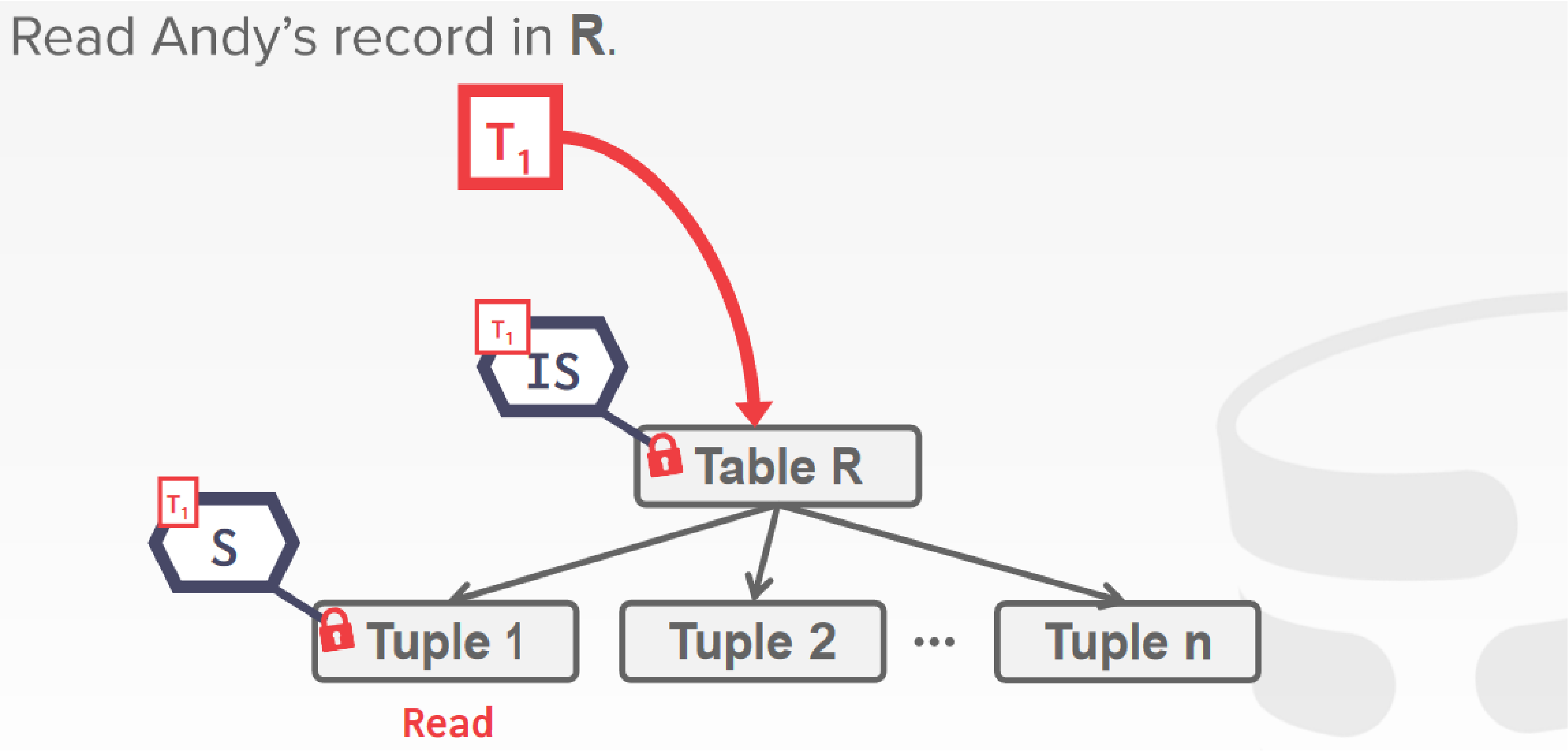
- An **intention lock** allows a higher level node to be locked without having to check all descendant nodes.
  - If a node is intention locked, then explicit locking is being done at a lower level in the tree.
- Five types of locks:
  - S lock: shared lock (same as before)
  - X lock: exclusive lock (same as before)
  - IS lock: intention-shared lock
    - You get IS at the parent, then S at one or more descendants
  - IX lock: intention-exclusive lock
    - You get IX at the parent, then X or S at one or more descendants
  - SIX lock: S + IX together

# Example



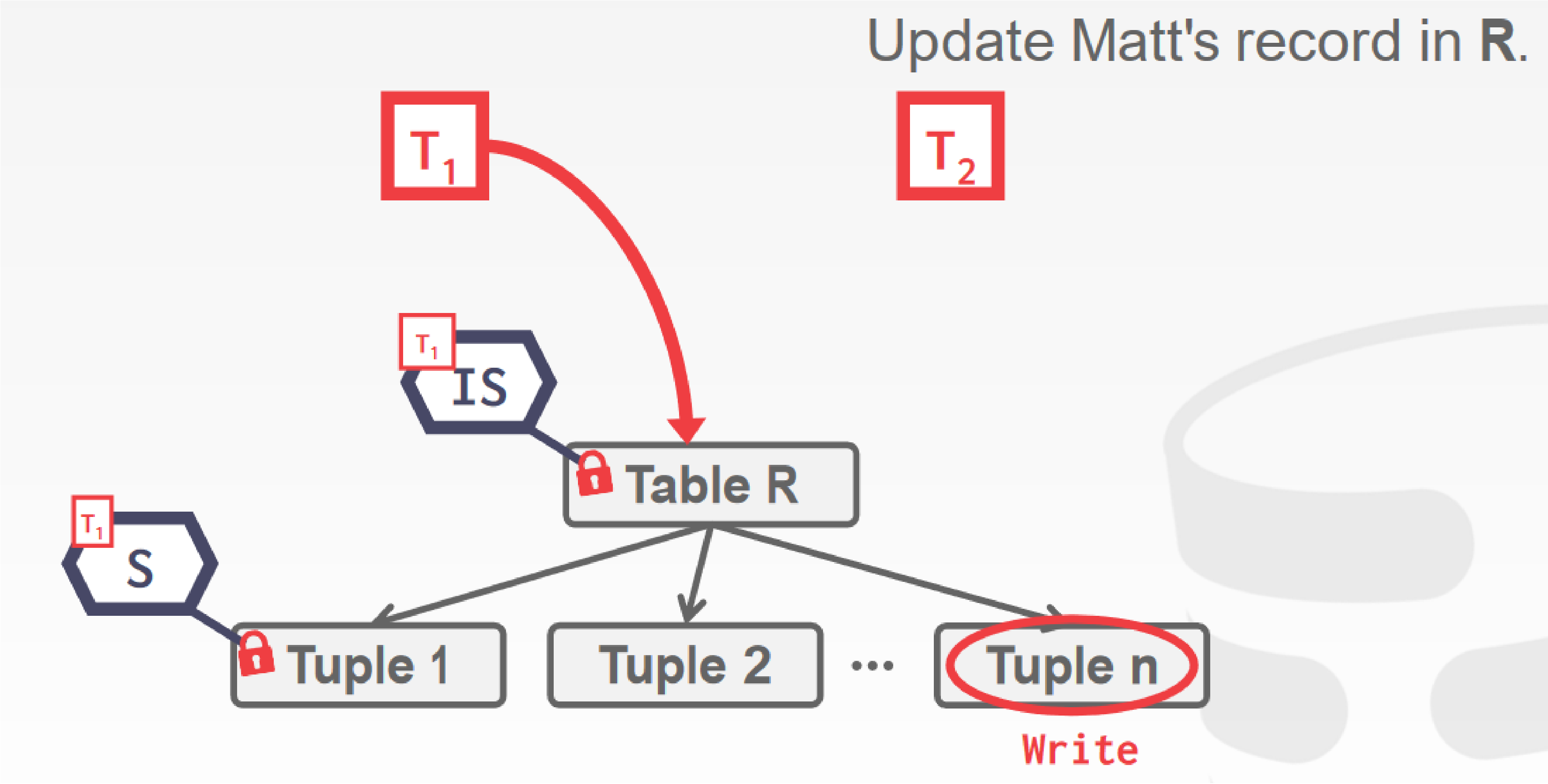
Read Andy's record in **R**.

# Example

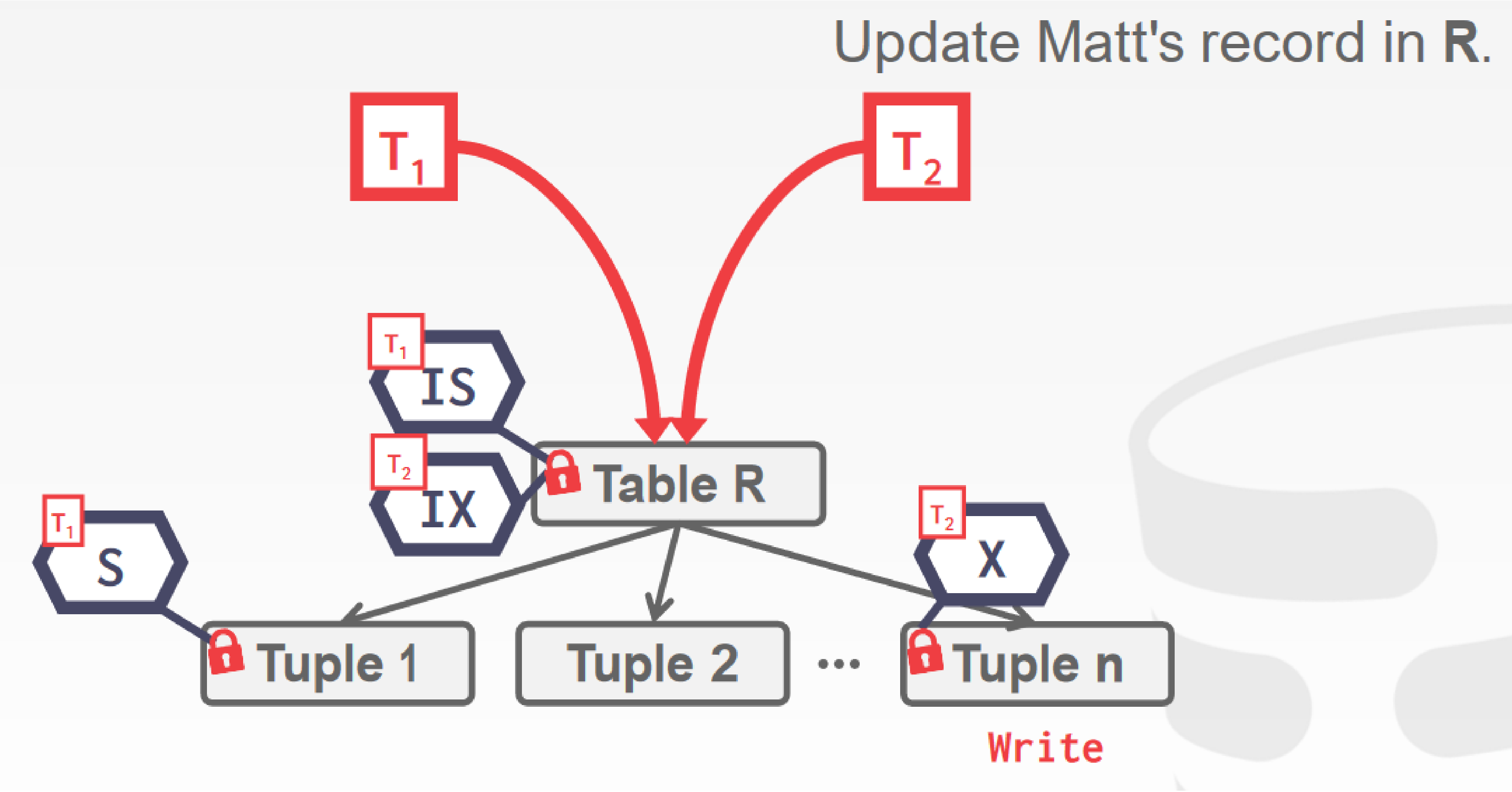


Read Andy’s record in **R**.

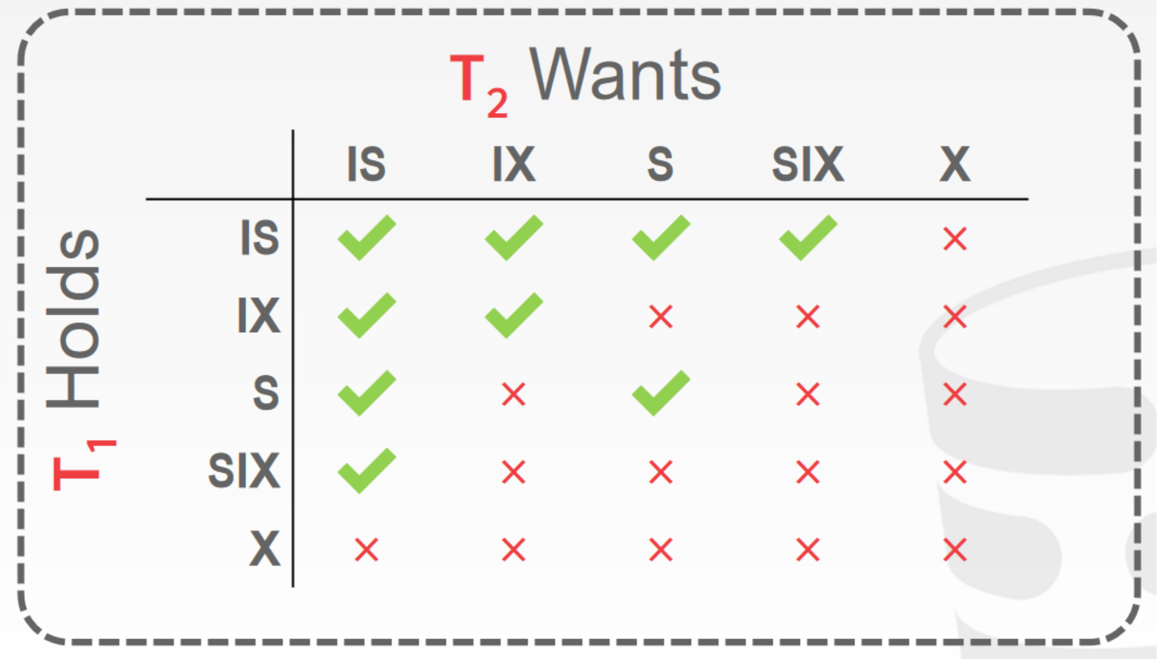
# Example


Update Matt's record in R.
$T_1$
$T_2$
$T_1$
IS
Table R
$T_1$
S
Tuple 1
Tuple 2
…
Tuple n
Write

# Example


Update Matt's record in **R**.
$T_1$
$T_2$
$T_1$
IS
$T_2$
IX
Table R
$T_1$
S
$T_2$
X
Tuple 1
Tuple 2
...
Tuple n
Write

# Compatibility



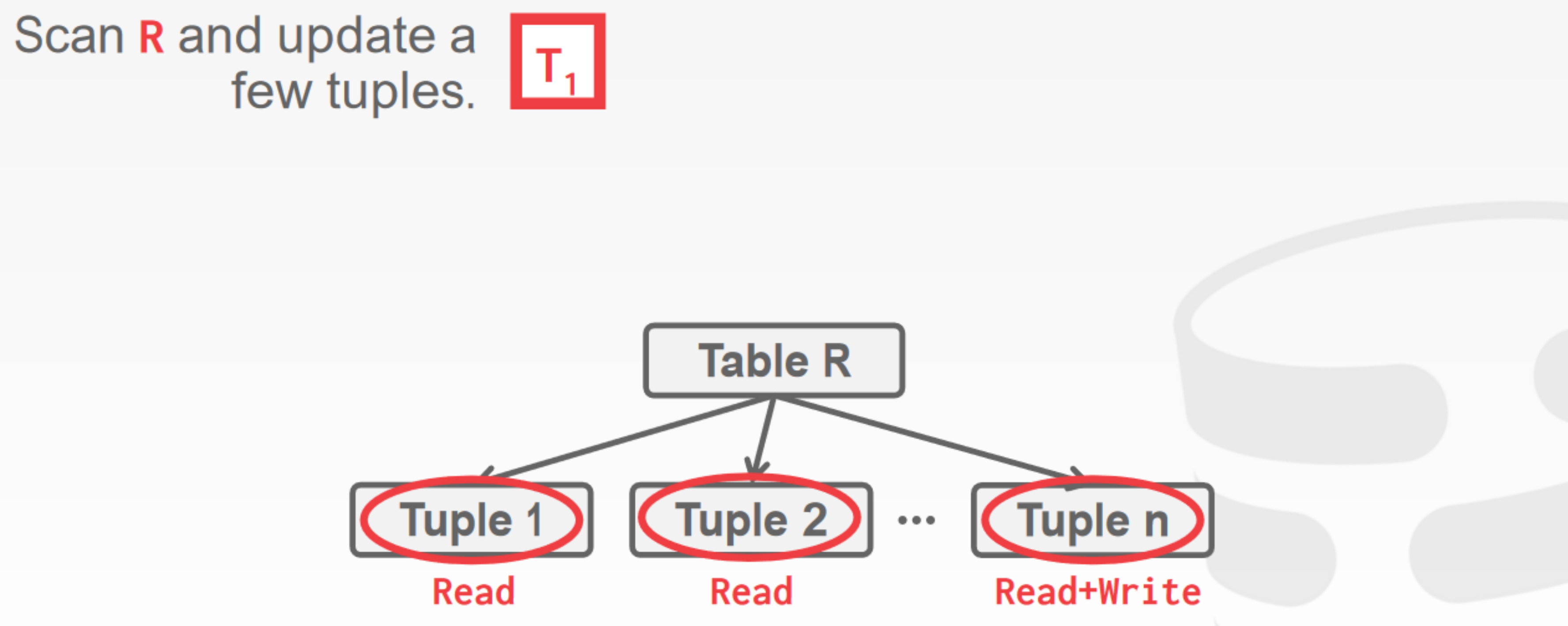
$T_2$ Wants

$T_1$ Holds

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | ✓ | ✓ | ✓ | ✓ | × |
| IX | ✓ | ✓ | × | × | × |
| S | ✓ | × | ✓ | × | × |
| SIX | ✓ | × | × | × | × |
| X | × | × | × | × | × |

- If transactions' locks are not compatible, they cannot execute concurrently!

# Example



Scan R and update a few tuples.

$T_1$

Table R

Tuple 1

Read

Tuple 2

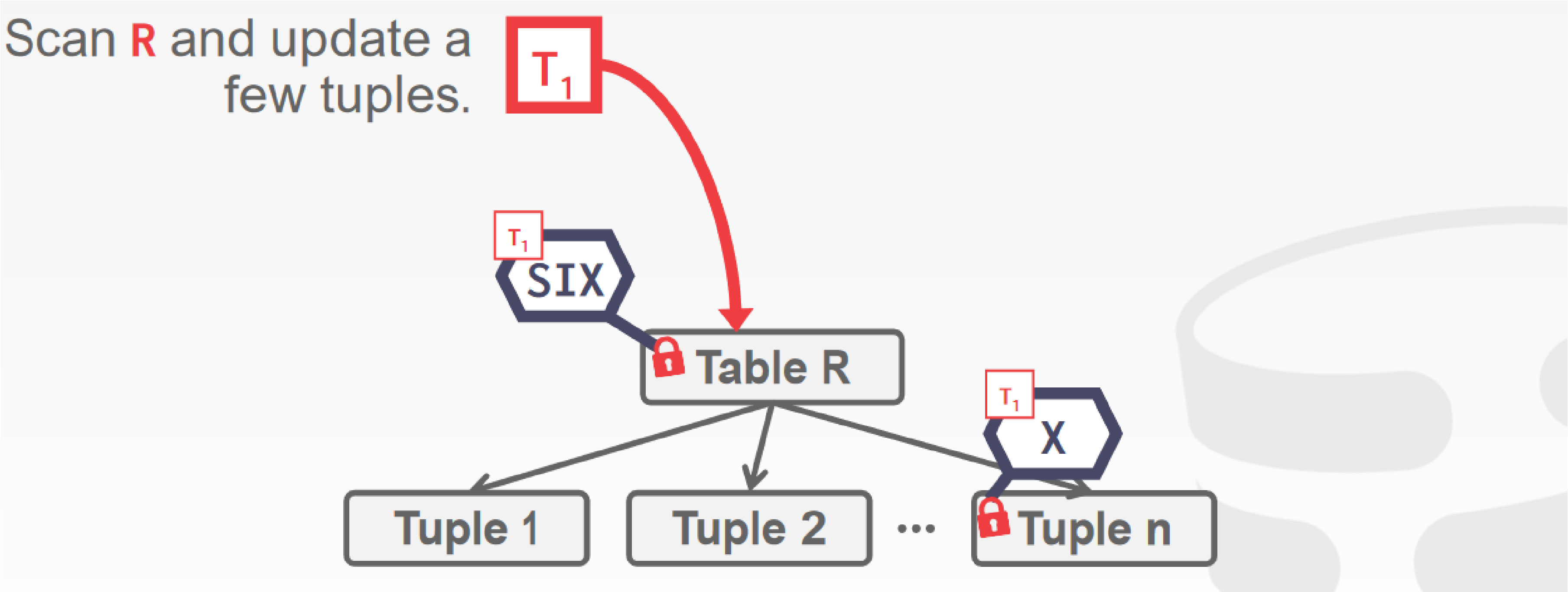
Read

...

Tuple n

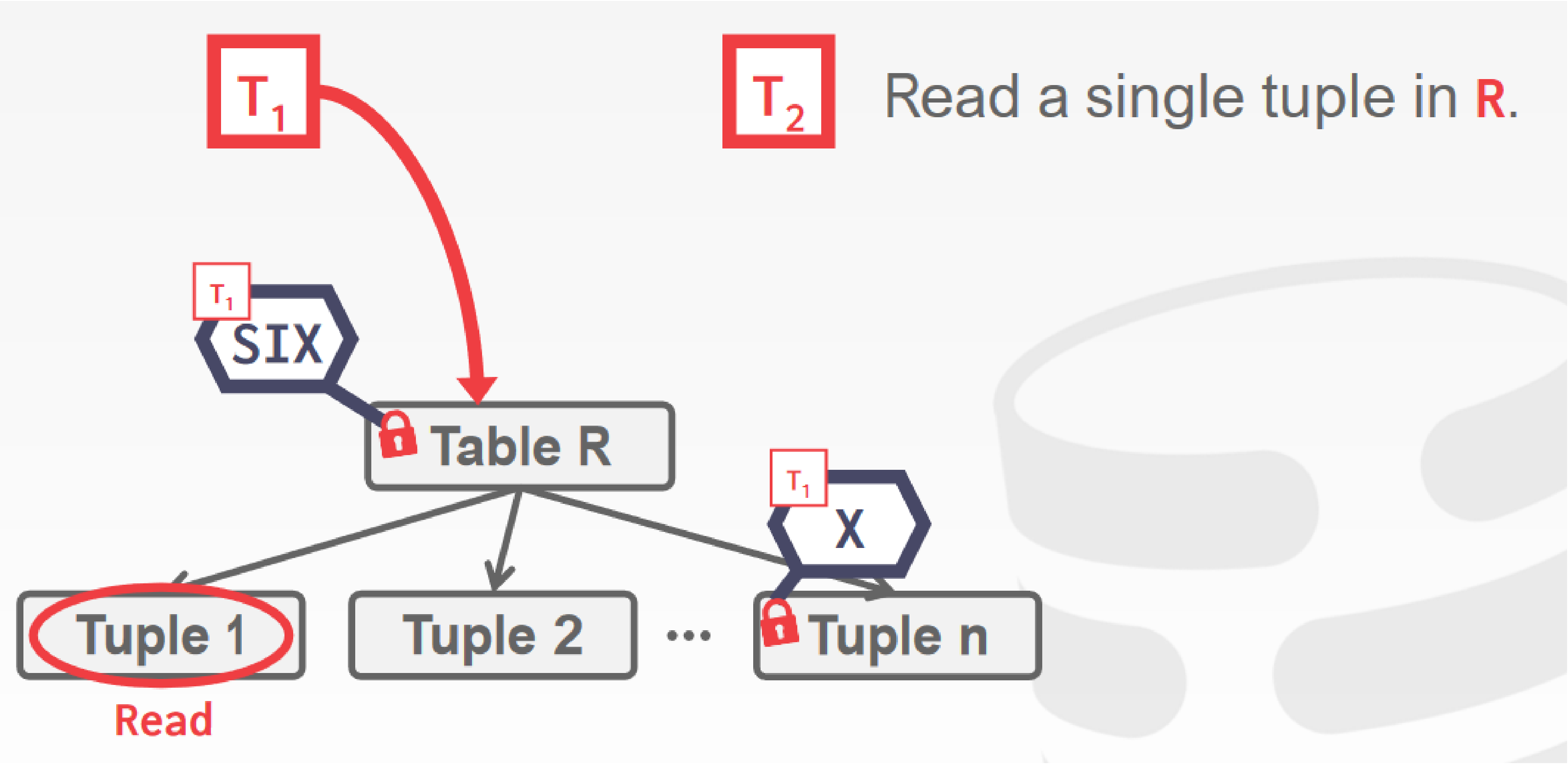
Read+Write

# Example

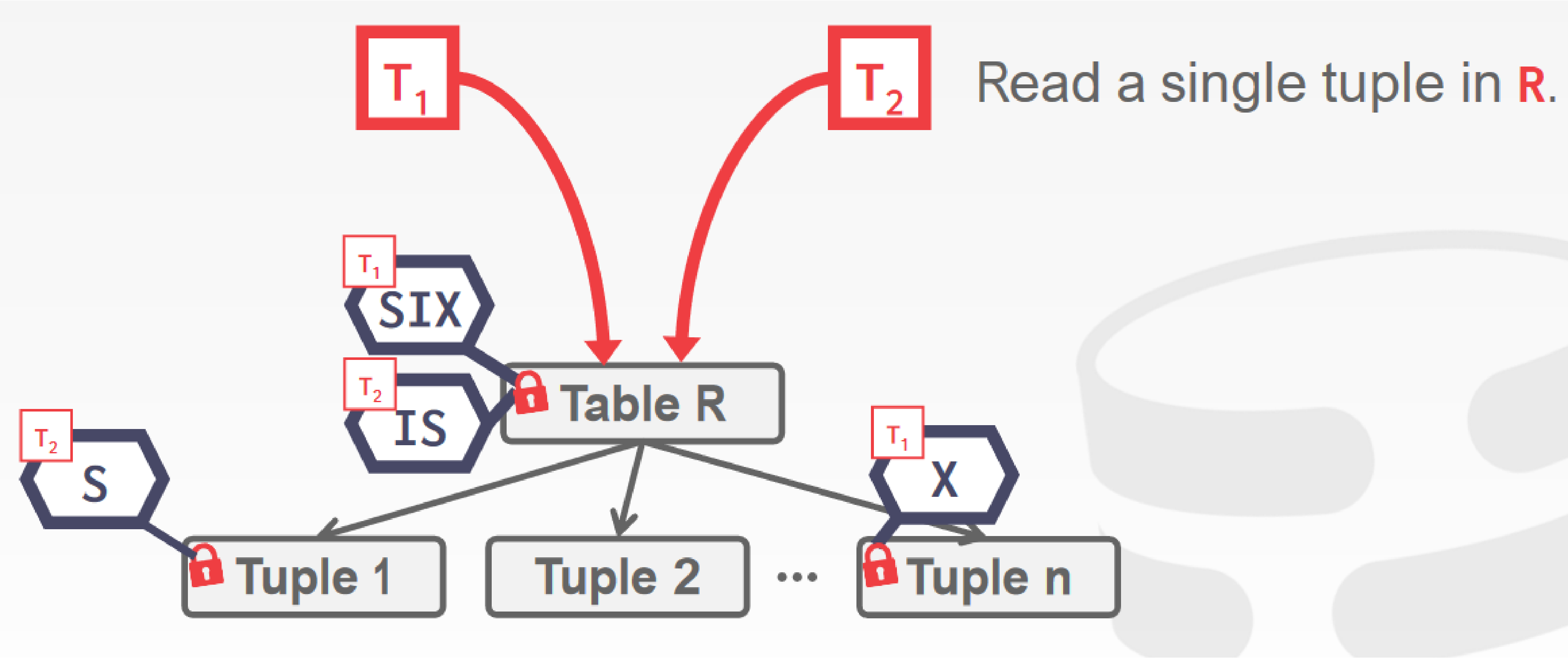


Scan R and update a few tuples.

# Example

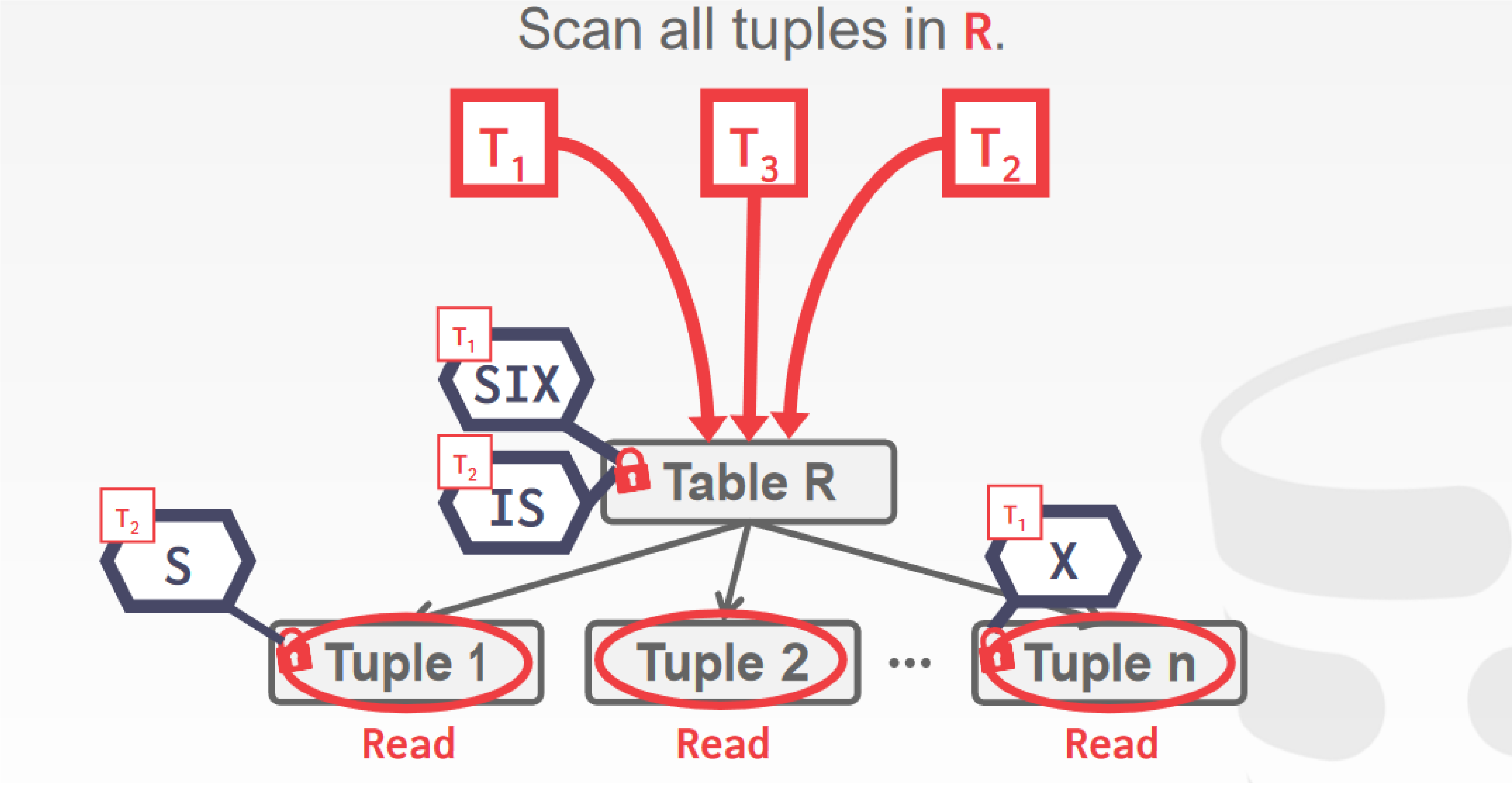

T1
T2
Read a single tuple in R.
T1
SIX
Table R
T1
X
Tuple 1
Tuple 2
…
Tuple n
Read

# Example

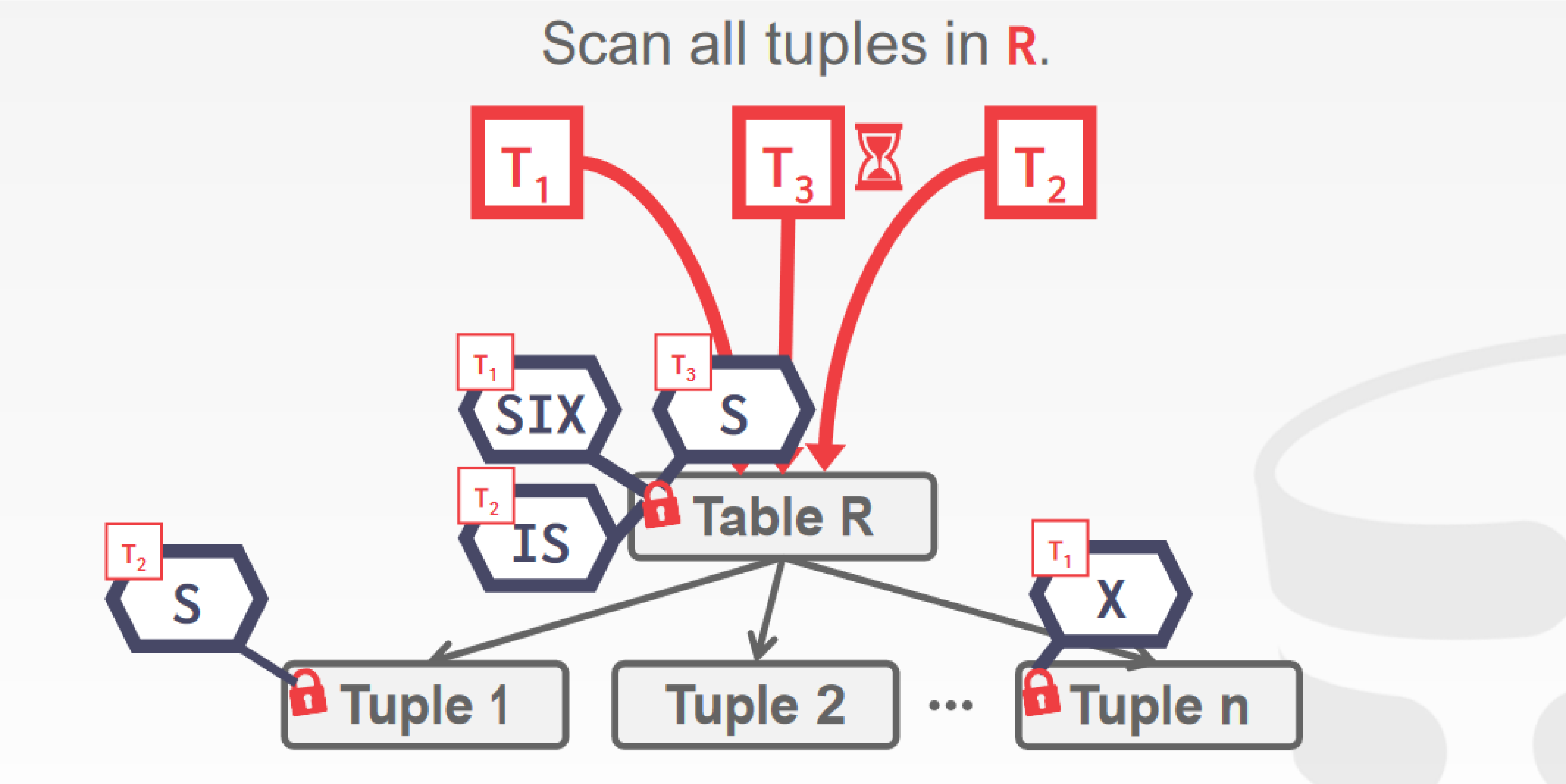


$T_1$
$T_2$
Read a single tuple in R.
$T_1$
SIX
$T_2$
IS
$T_2$
S
Table R
$T_1$
X
Tuple 1
Tuple 2
…
Tuple n

# Example


Scan all tuples in R.
$T_1$
$T_3$
$T_2$
$T_1$
SIX
$T_2$
IS
Table R
$T_2$
S
$T_1$
X
Tuple 1
Tuple 2
...
Tuple n
Read
Read
Read

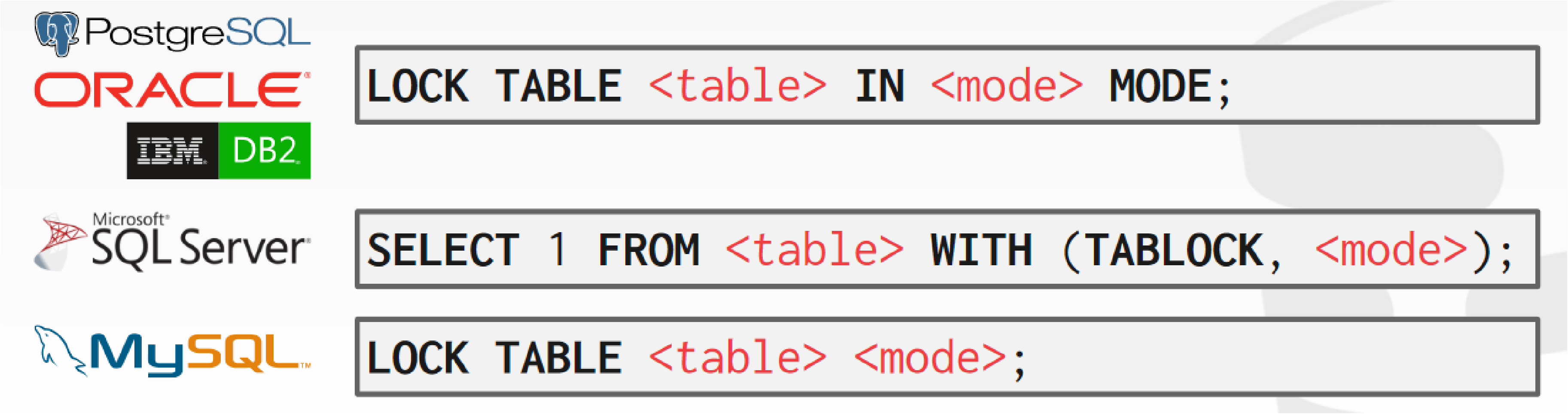
# Example



**Are the locks of these transactions compatible?**

# Locking in Practice

- Different DBMSs use different locking methods
- The DBMS can automatically take care of locking, unlocking, etc.
- But there are methods for explicit, manual locking
  - Depends on which DBMS you are using
    - PostgreSQL: Locking in SHARE, EXCLUSIVE, and other modes
    - MySQL: Locking in READ,WRITE modes


PostgreSQL
ORACLE
IBM DB2

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

Microsoft SQL Server

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

MySQL

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