

# 17

## Two-Phase Locking



Intro to Database Systems  
15-445/15-645  
Fall 2020

AP

Andy Pavlo  
Computer Science  
Carnegie Mellon University

# ADMINISTRIVIA

---

**Project #3** is due Sun Nov 22<sup>nd</sup> @ 11:59pm.

**Homework #4** is due Sun Nov 8<sup>th</sup> @ 11:59pm.



# ADMINISTRIVIA

---

Sign up for the student-run discussion groups.

- Small group of at most 10 students where you can discuss the implementation details of the projects.
- You can share test code, but you are not allowed to share implementation code.

See [Piazza@906](#) for more details.



# UPCOMING DATABASE TALKS

---

## MySQL Query Optimizer

→ Monday Nov 2<sup>nd</sup> @ 5pm ET



## EraDB "Magical Indexes"

→ Monday Nov 9<sup>th</sup> @ 5pm ET



## FaunaDB Serverless DBMS

→ Monday Nov 16<sup>th</sup> @ 5pm ET



# LAST CLASS

---

## **Conflict Serializable**

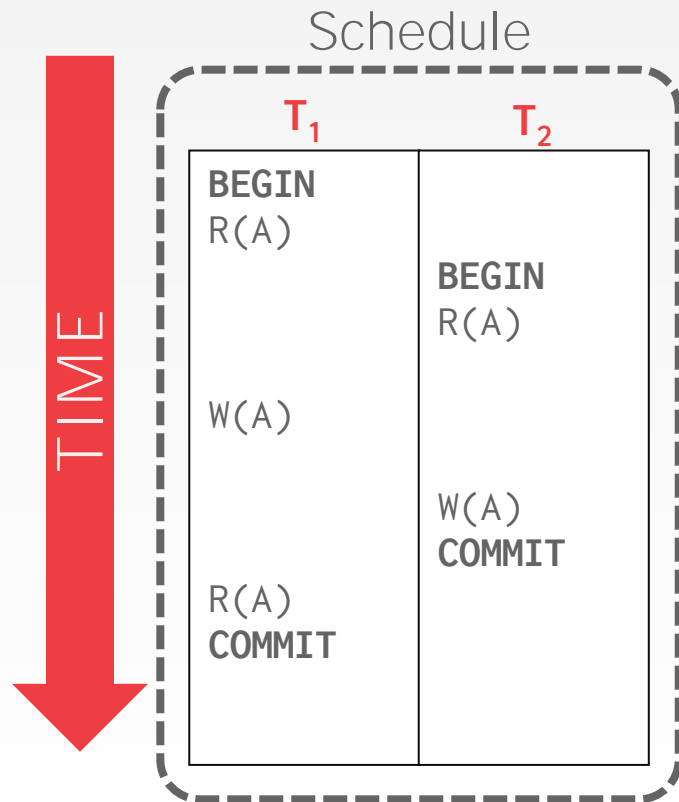
- Verify using either the "swapping" method or dependency graphs.
- Any DBMS that says that they support "serializable" isolation does this.

## **View Serializable**

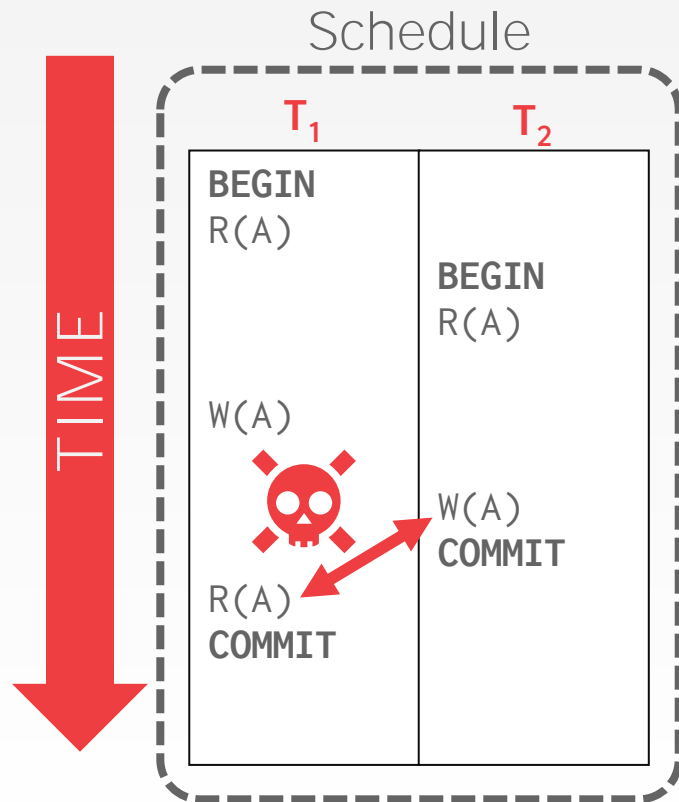
- No efficient way to verify.
- Andy doesn't know of any DBMS that supports this.



# EXAMPLE



# EXAMPLE



# OBSERVATION

---

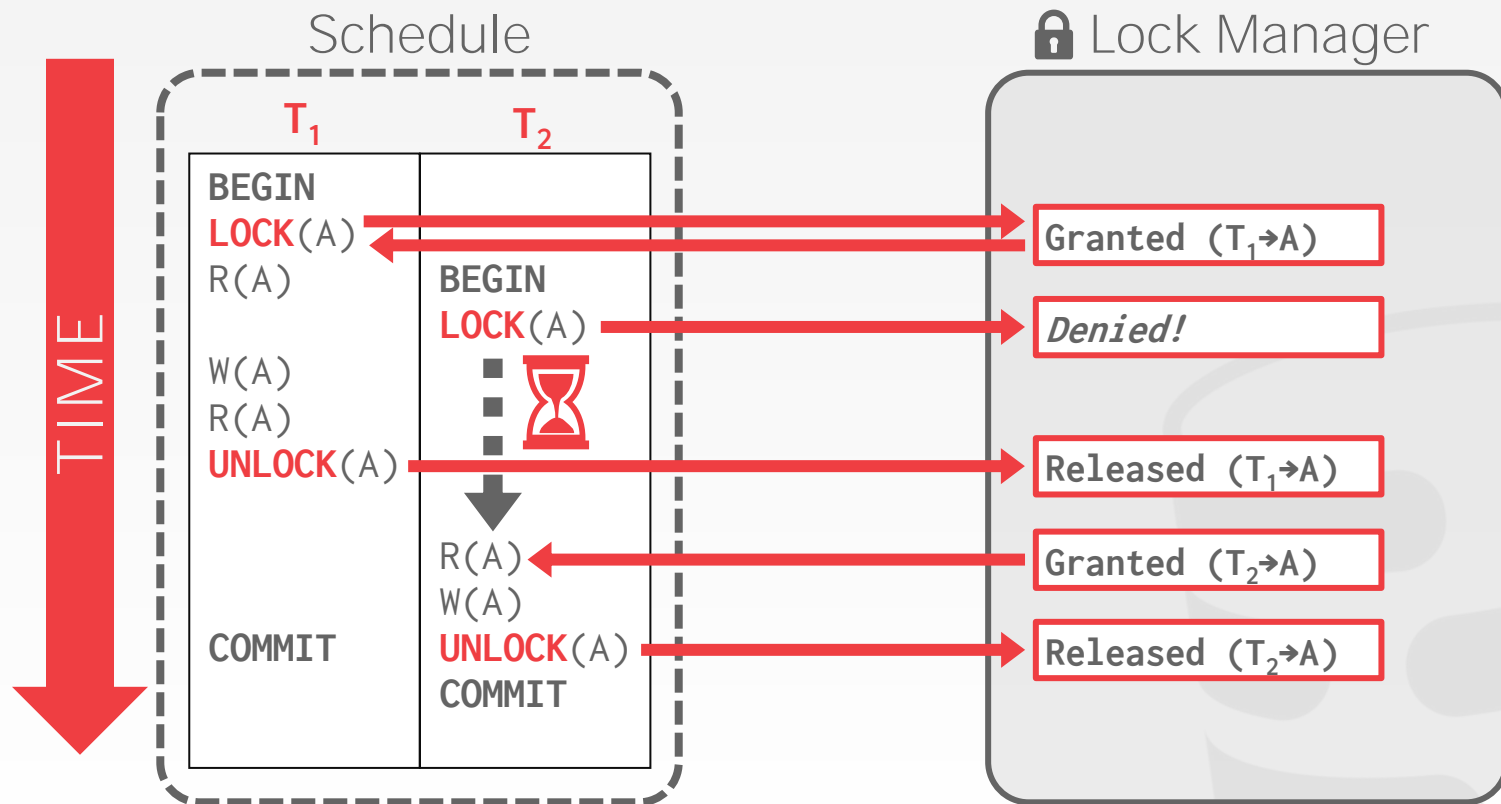
We need a way to guarantee that all execution schedules are correct (i.e., serializable) without knowing the entire schedule ahead of time.

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





# EXECUTING WITH LOCKS



# TODAY'S AGENDA

---

Lock Types

Two-Phase Locking

Deadlock Detection + Prevention

Hierarchical Locking

Isolation Levels



# LOCKS VS. LATCHES

	<i><b>Locks</b></i>	<i><b>Latches</b></i>
<b>Separate...</b>	User transactions	Threads
<b>Protect...</b>	Database Contents	In-Memory Data Structures
<b>During...</b>	Entire Transactions	Critical Sections
<b>Modes...</b>	Shared, Exclusive, Update, Intention	Read, Write
<b>Deadlock</b>	Detection & Resolution	Avoidance
<b>...by...</b>	Waits-for, Timeout, Aborts	Coding Discipline
<b>Kept in...</b>	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

# EXECUTING WITH LOCKS

---

Transactions request locks (or upgrades).

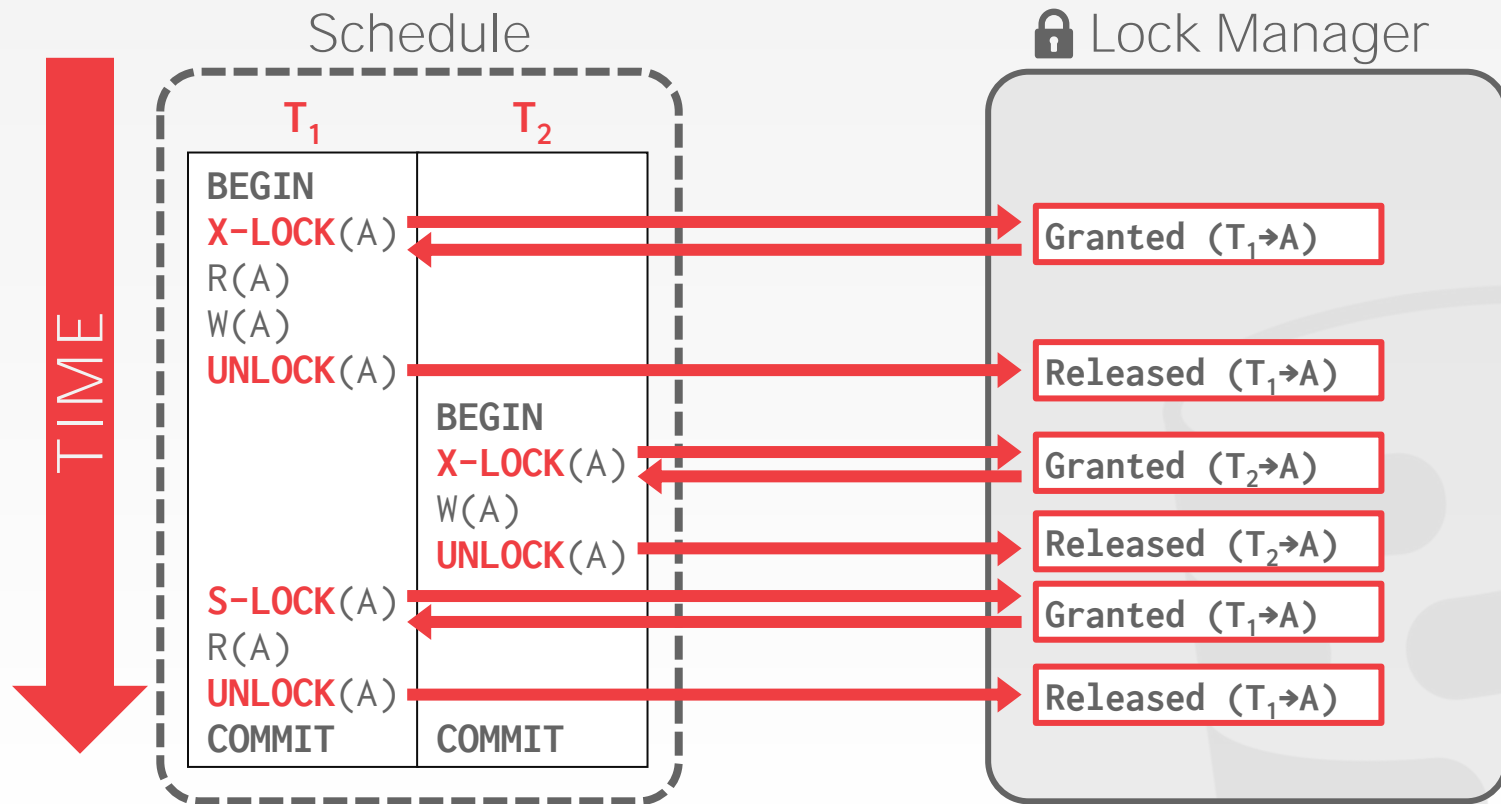
Lock manager grants or blocks requests.

Transactions release locks.

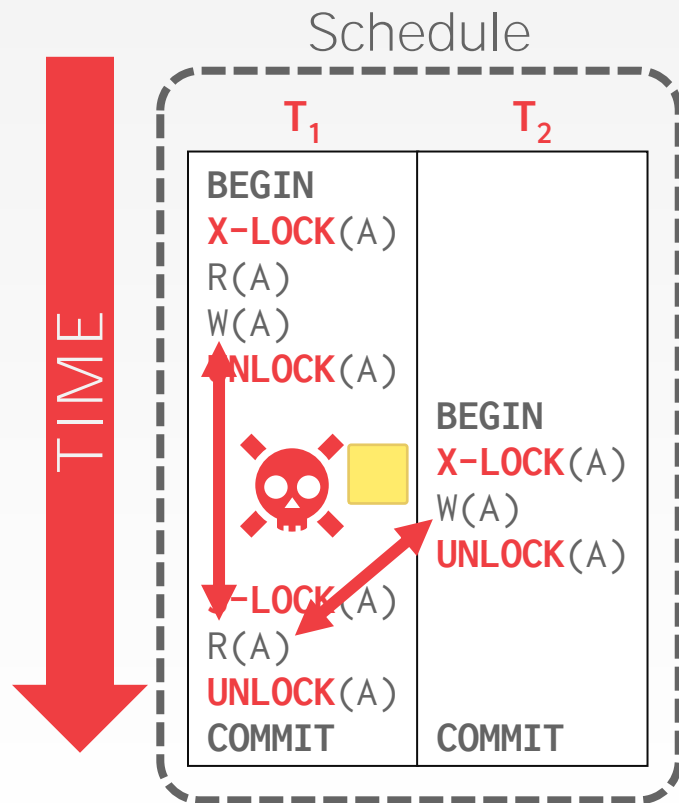
Lock manager updates its internal lock-table.

→ It keeps track of what transactions hold what locks and what transactions are waiting to acquire any locks.

# EXECUTING WITH LOCKS



# EXECUTING WITH LOCKS



## Lock Manager

Granted (T<sub>1</sub>→A)

Released (T<sub>1</sub>→A)

Granted (T<sub>2</sub>→A)

Released (T<sub>2</sub>→A)

Granted (T<sub>1</sub>→A)

Released (T<sub>1</sub>→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 on the fly.

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



# TWO-PHASE LOCKING

---

## Phase #1: Growing

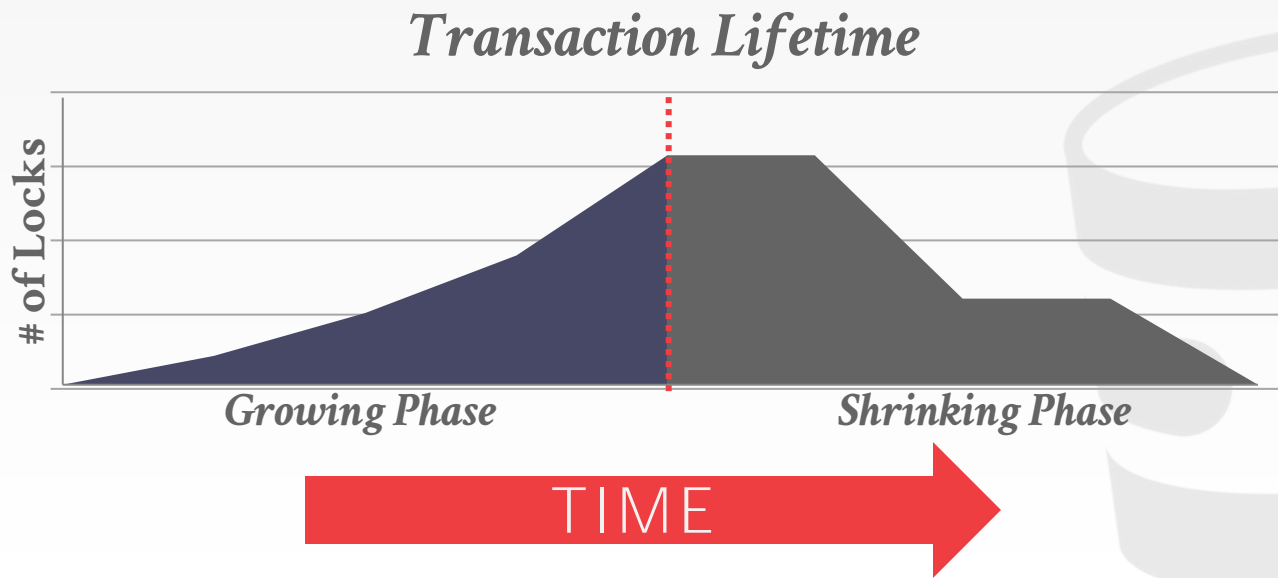
- Each txn requests the locks that it needs from the DBMS's lock manager.
- The lock manager grants/denies lock requests.

## Phase #2: Shrinking

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

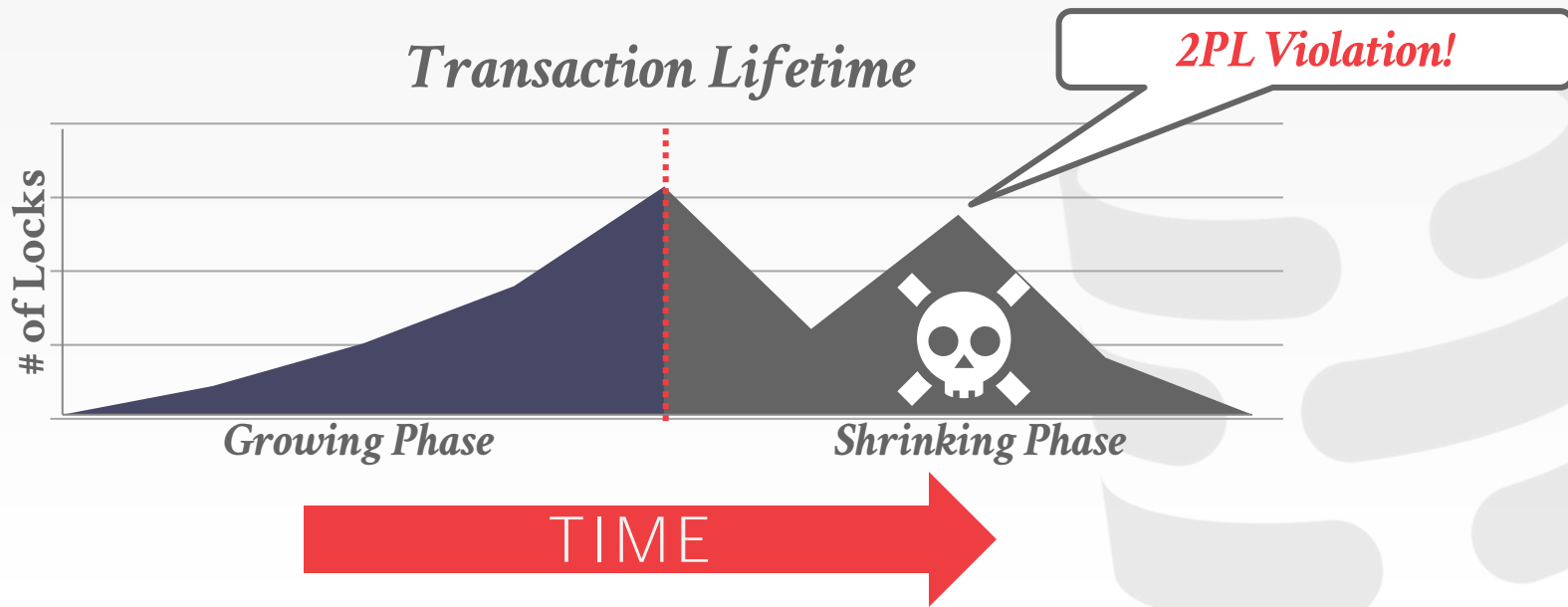
# TWO-PHASE LOCKING

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

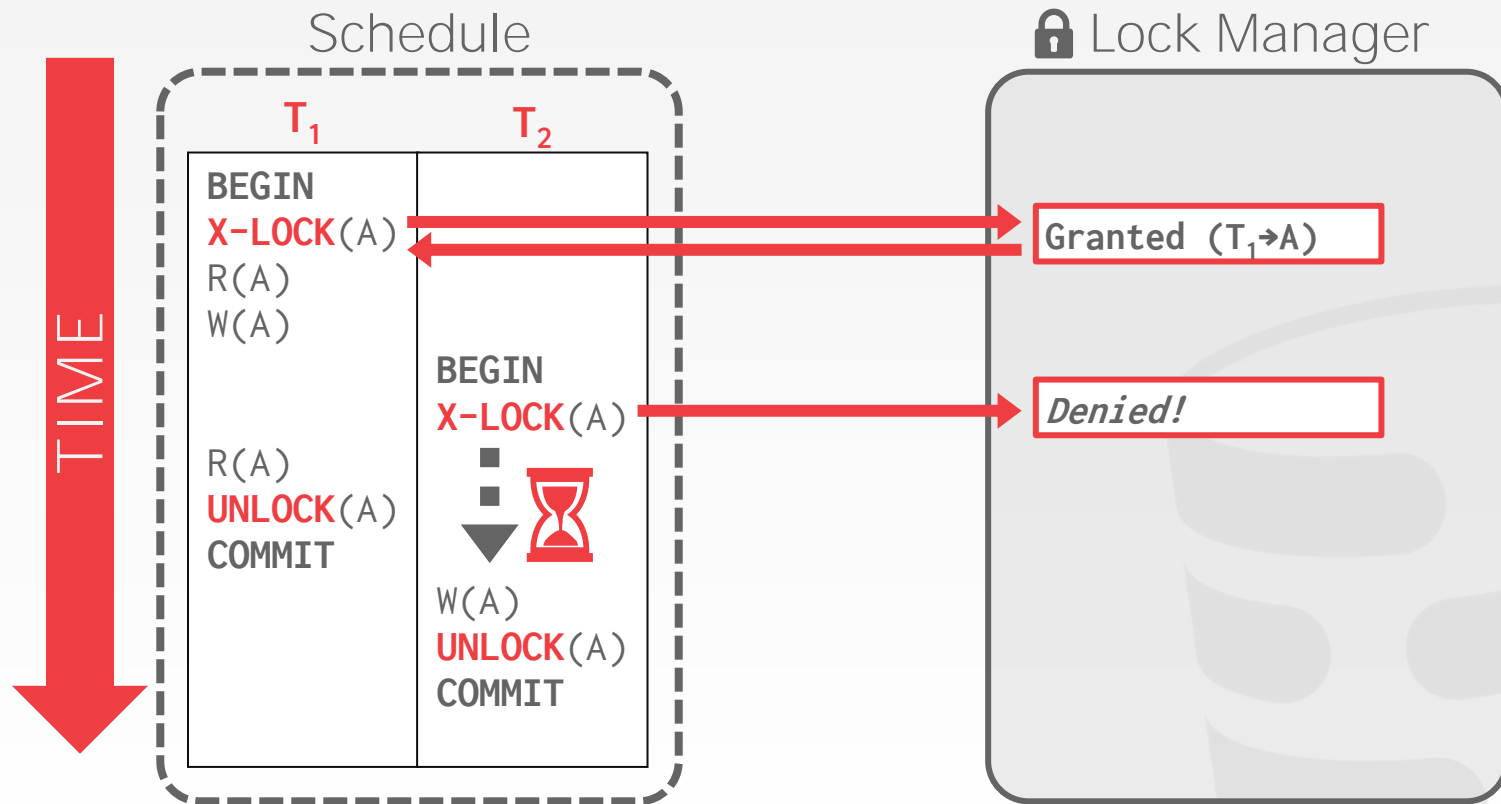


# TWO-PHASE LOCKING

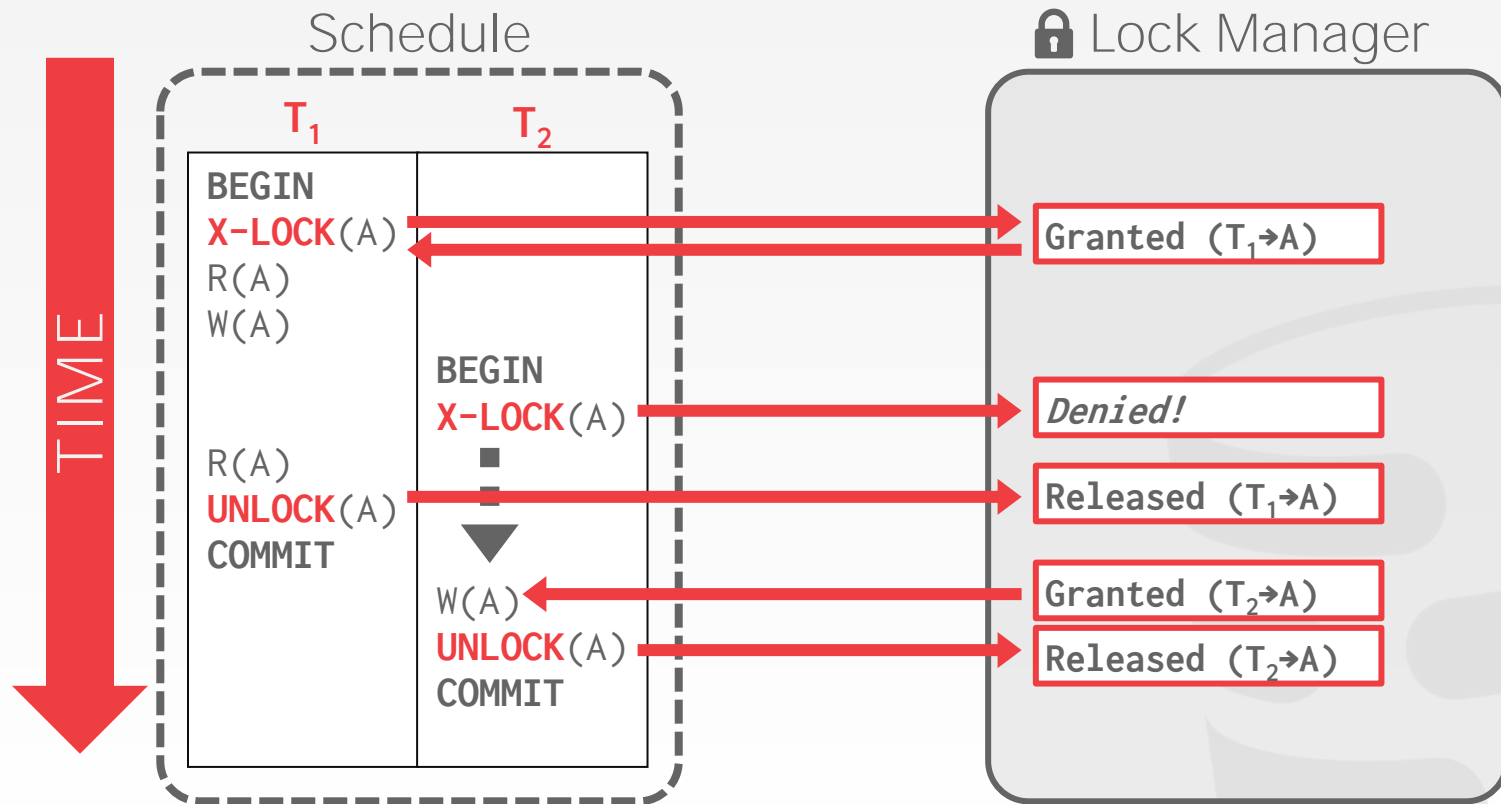
The txn is not allowed to acquire/upgrade locks after the growing phase finishes.



# EXECUTING WITH 2PL



# EXECUTING WITH 2PL



# TWO-PHASE LOCKING

---

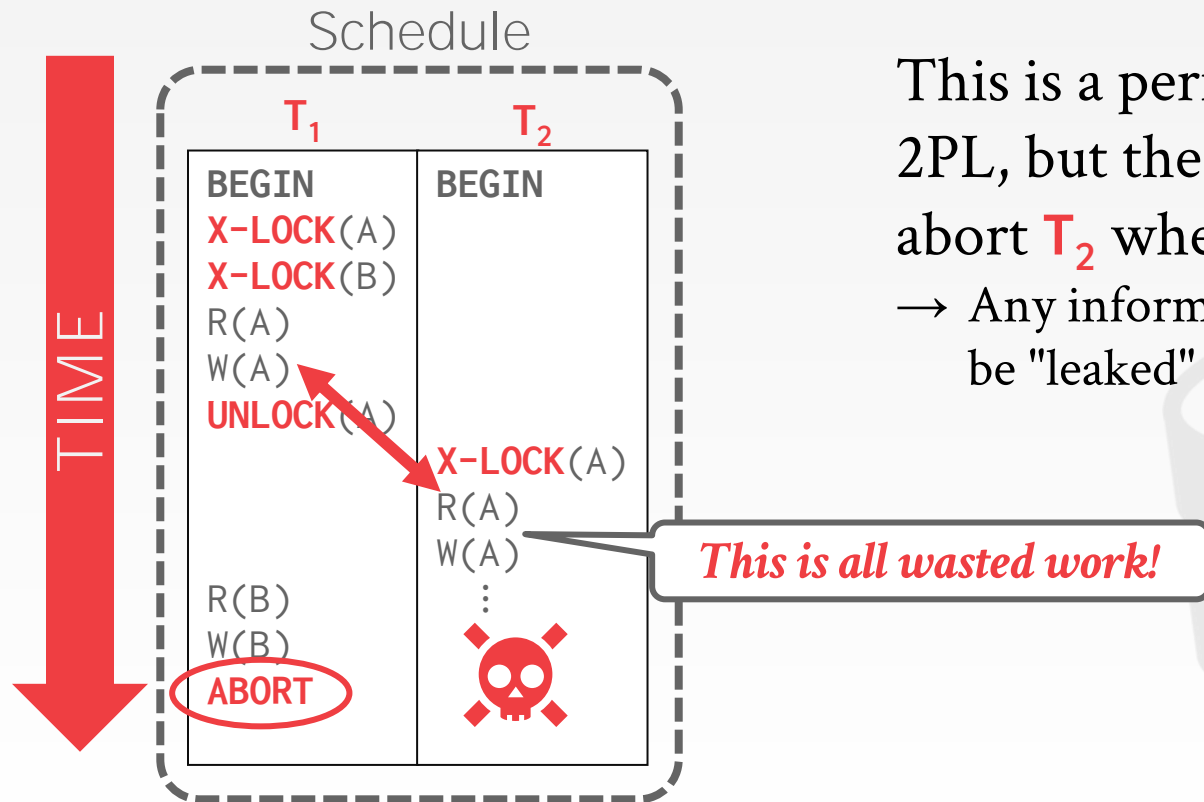
2PL on its own is sufficient to guarantee conflict serializability.

→ It generates schedules whose precedence graph is acyclic.

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



## 2PL – CASCADING ABORTS



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.

→ Locking limits concurrency.

May still have "dirty reads".

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

May lead to deadlocks.

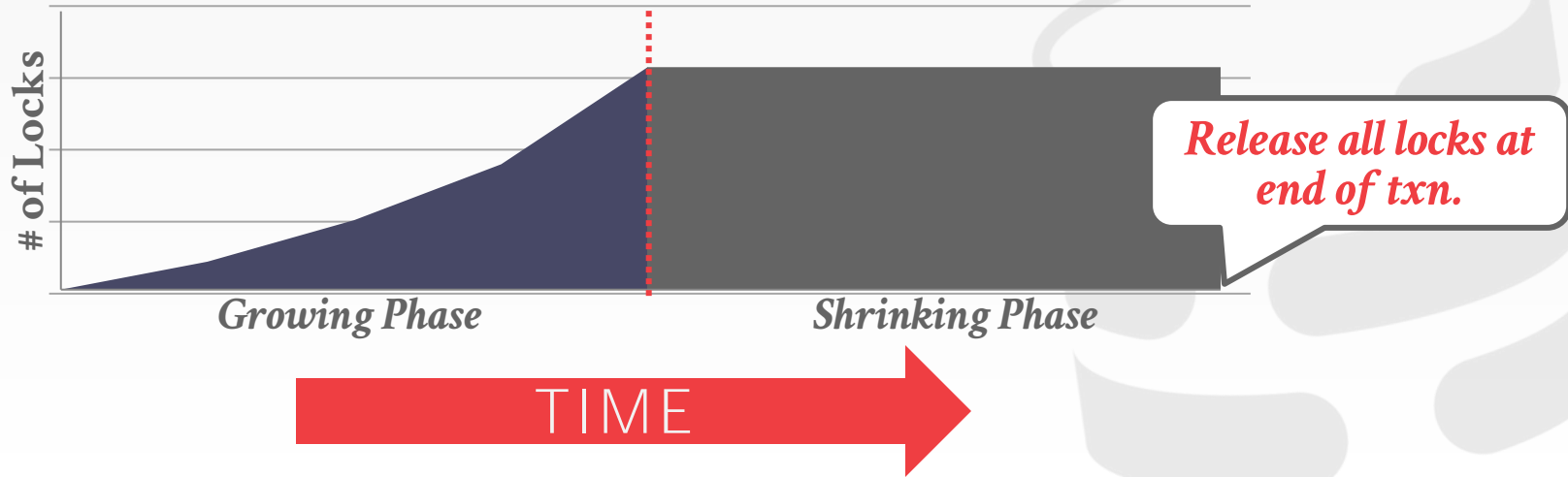
→ Solution: **Detection** or **Prevention**



# STRONG STRICT TWO-PHASE LOCKING

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

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



# 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.

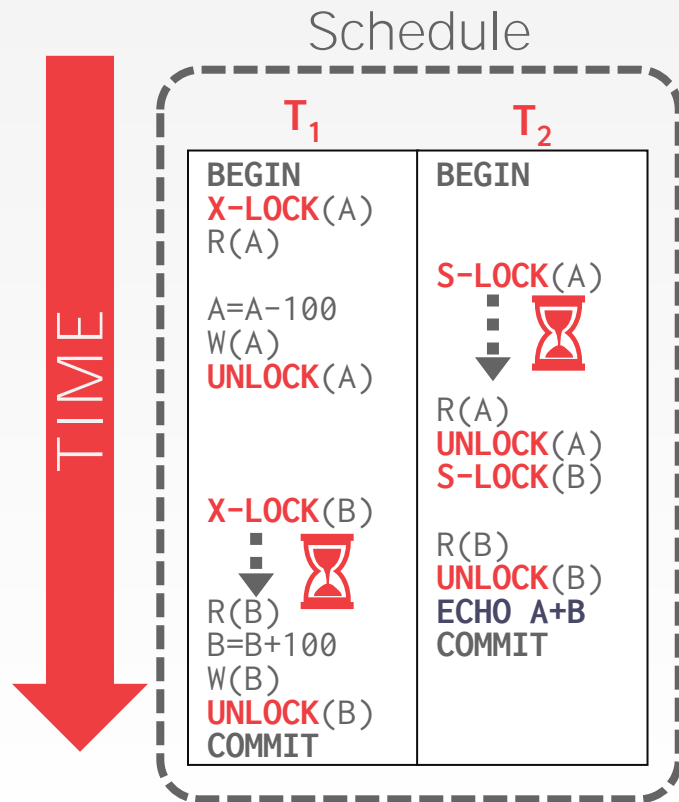
$T_1$

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

$T_2$

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

# NON-2PL EXAMPLE



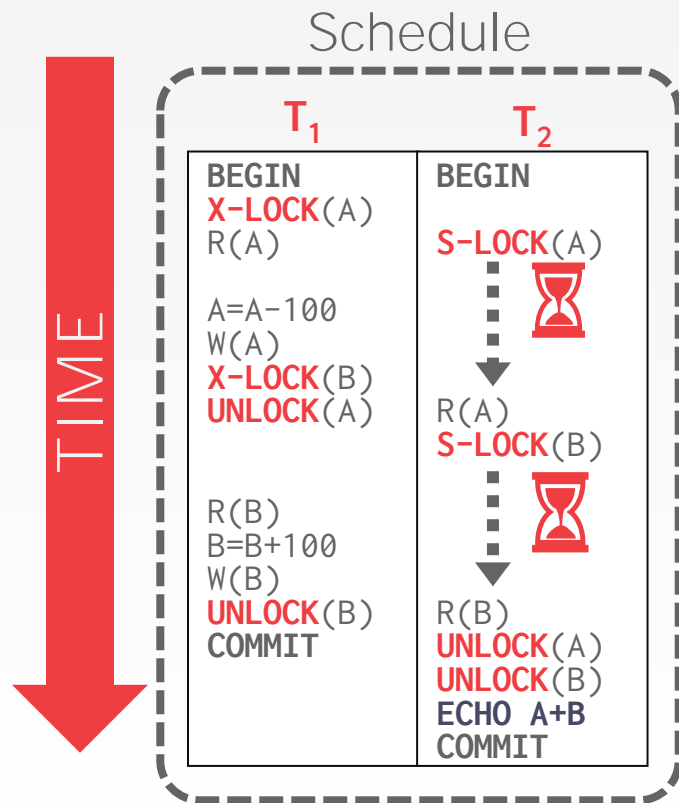
Initial Database State

**A**=1000, **B**=1000

$T_2$  Output

**A+B**=1100

# 2PL EXAMPLE



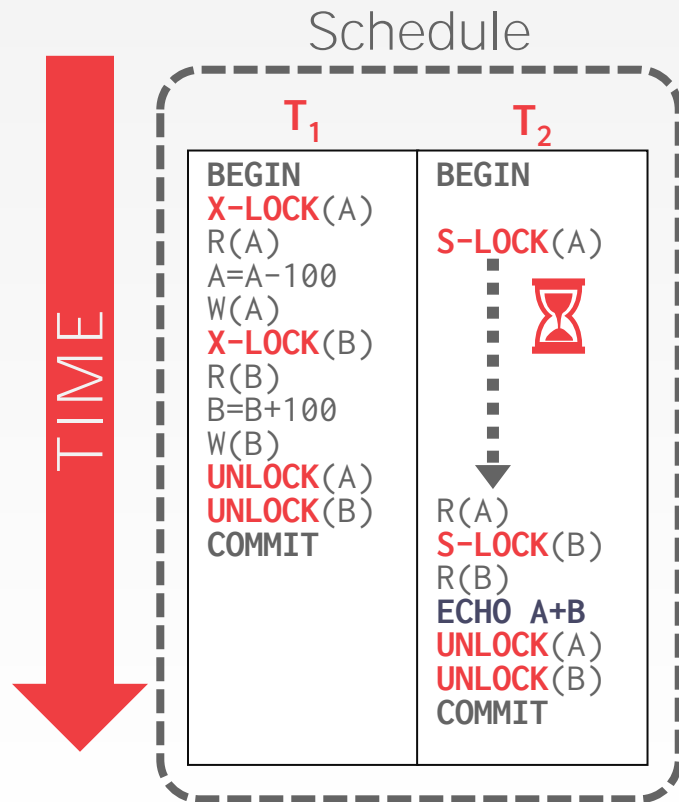
Initial Database State

**A**=1000, **B**=1000

$T_2$  Output

**A+B**=2000

# STRONG STRICT 2PL EXAMPLE



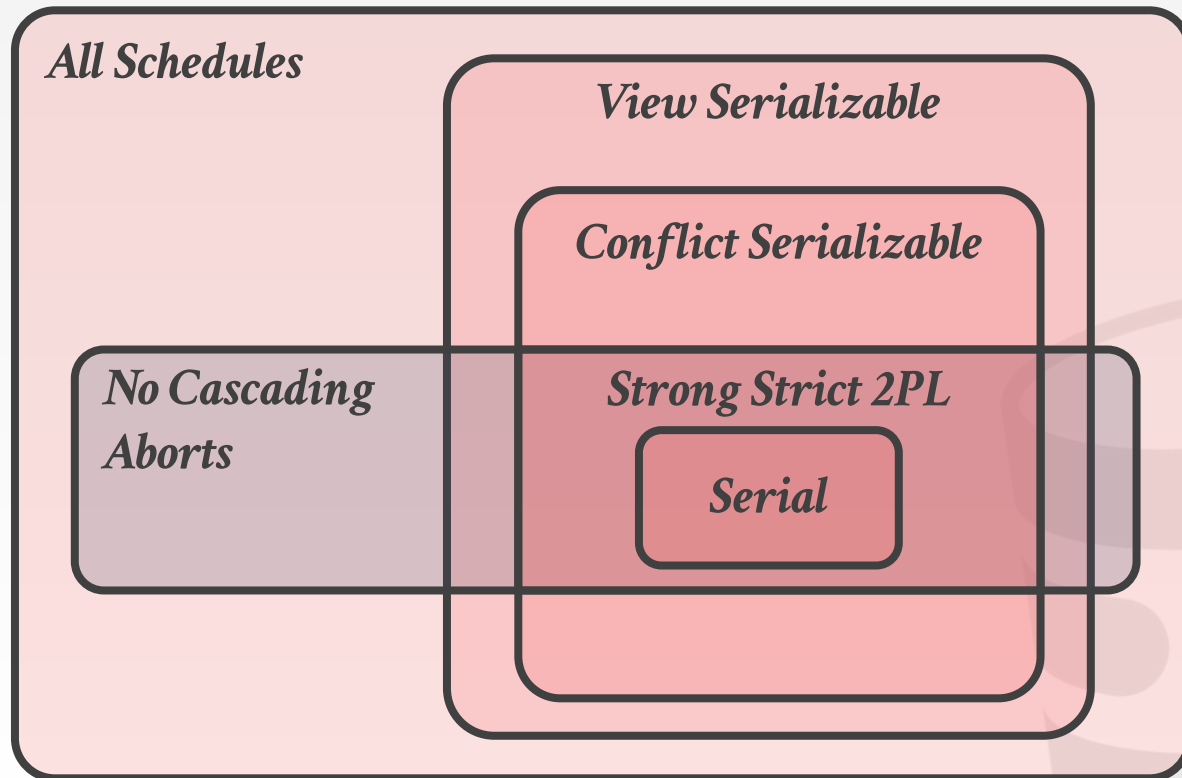
Initial Database State

A=1000, B=1000

$T_2$  Output

A+B=2000

# UNIVERSE OF SCHEDULES



## 2PL OBSERVATIONS

---

There are potential schedules that are serializable but would not be allowed by 2PL.

→ Locking limits concurrency.

May still have "dirty reads".

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

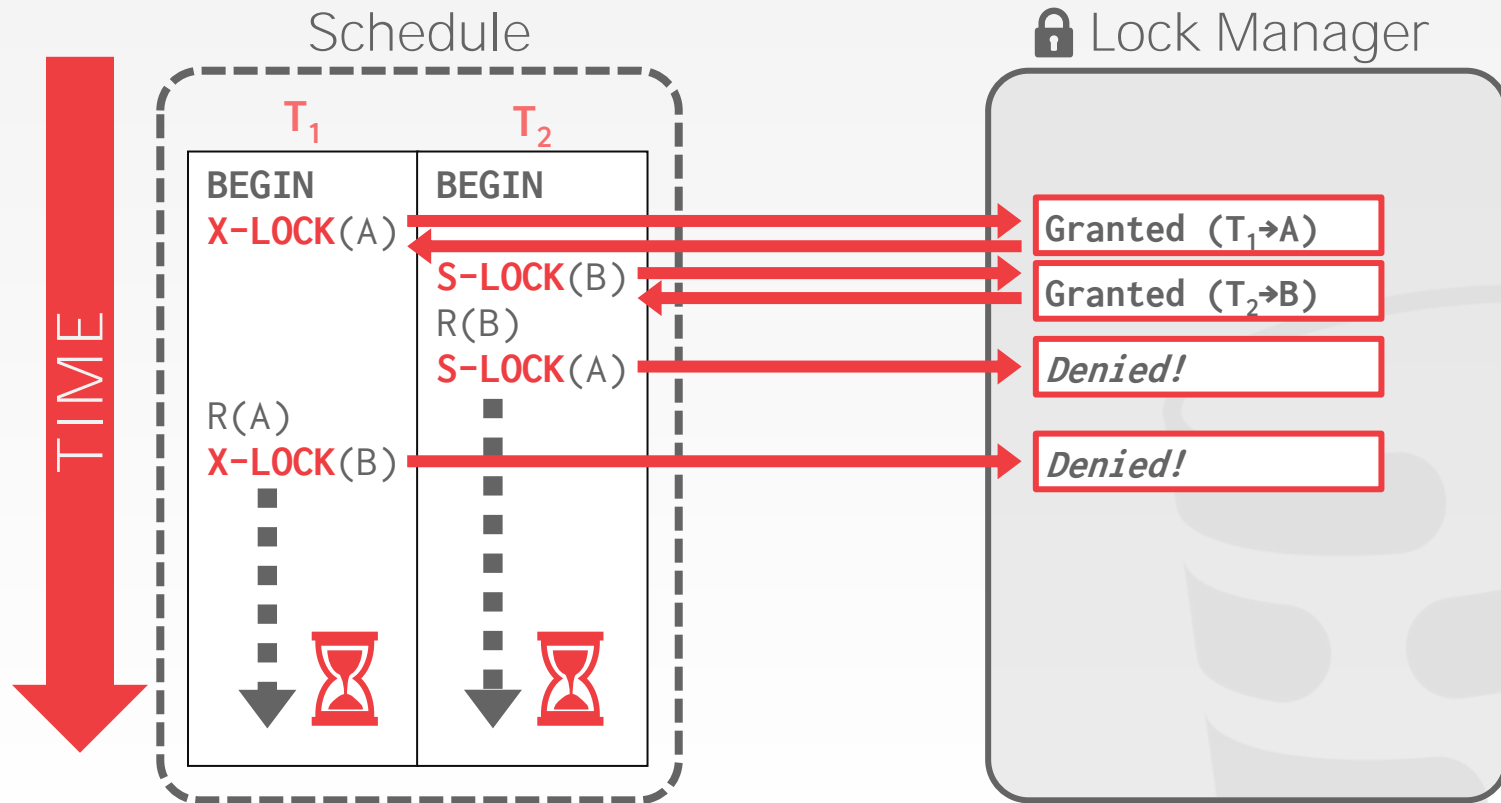
May lead to deadlocks.

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

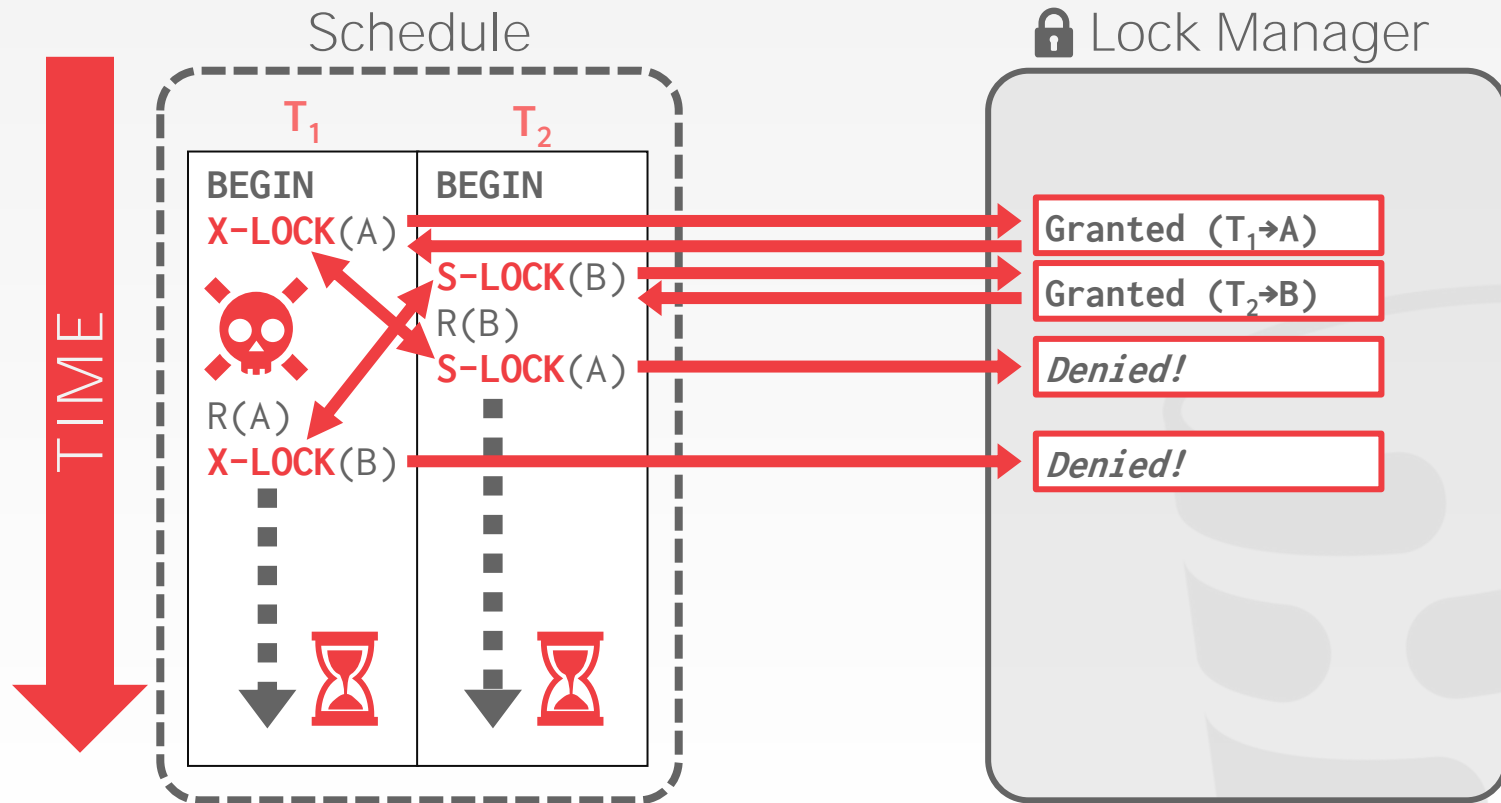




# SHIT JUST GOT REAL, SON



# SHIT JUST GOT REAL, SON



## 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

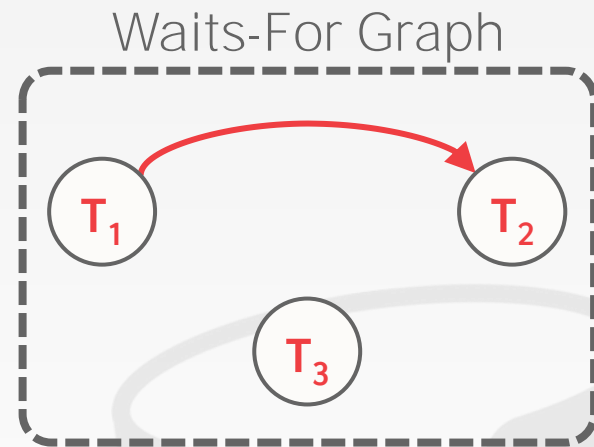
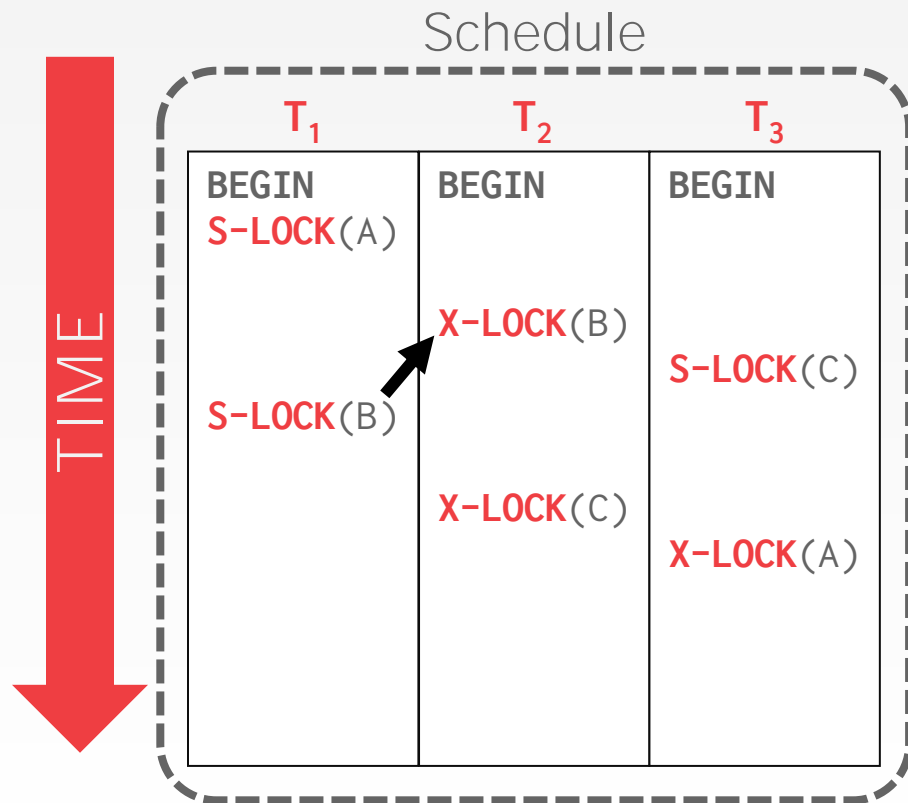
---

The DBMS creates a **waits-for** graph to keep track of what locks each txn is waiting to acquire:

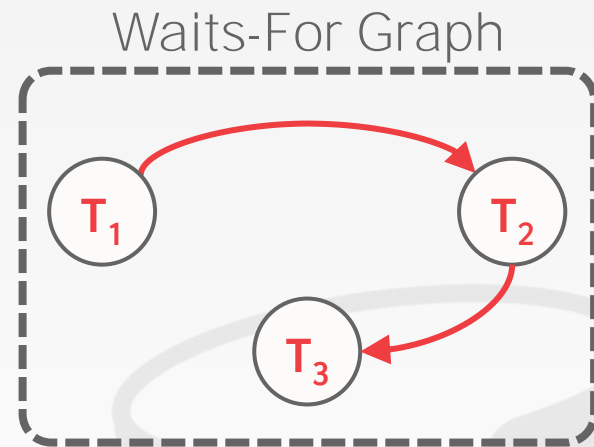
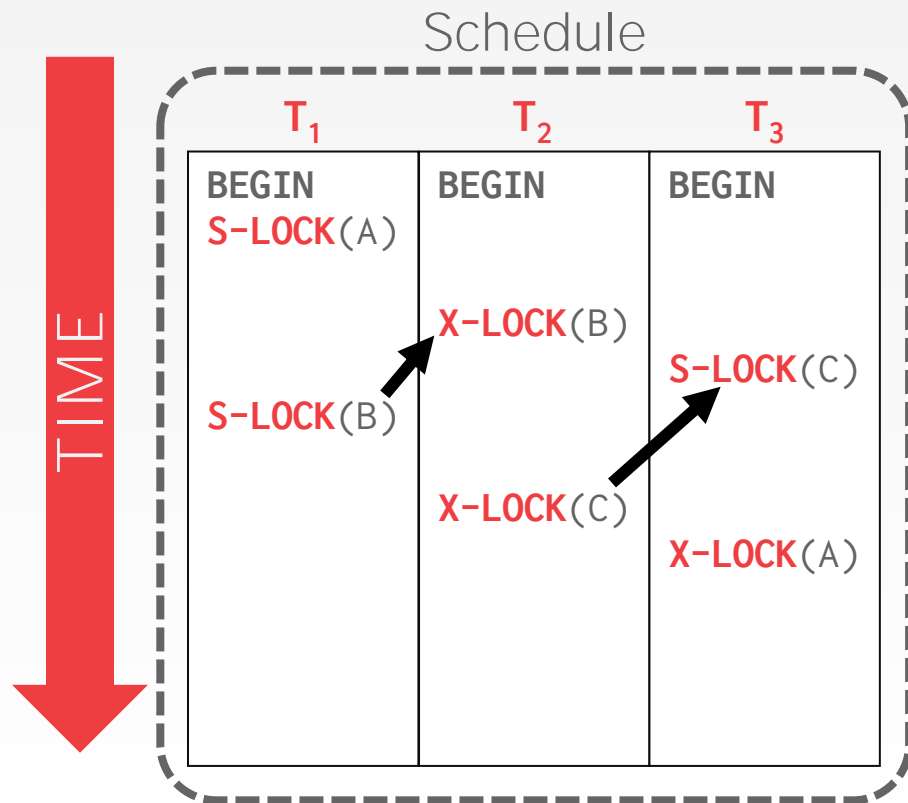
- Nodes are transactions
- Edge from  $T_i$  to  $T_j$  if  $T_i$  is waiting for  $T_j$  to release a lock.

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

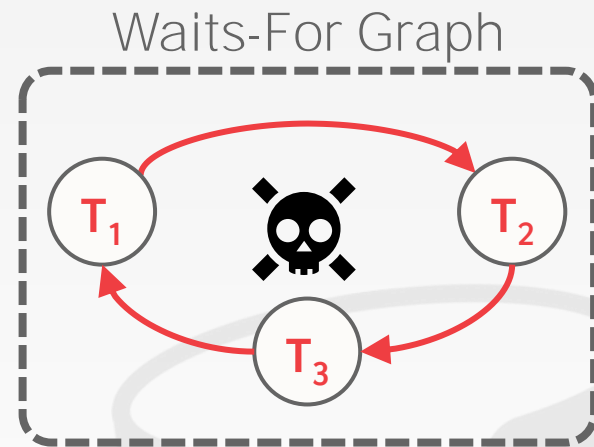
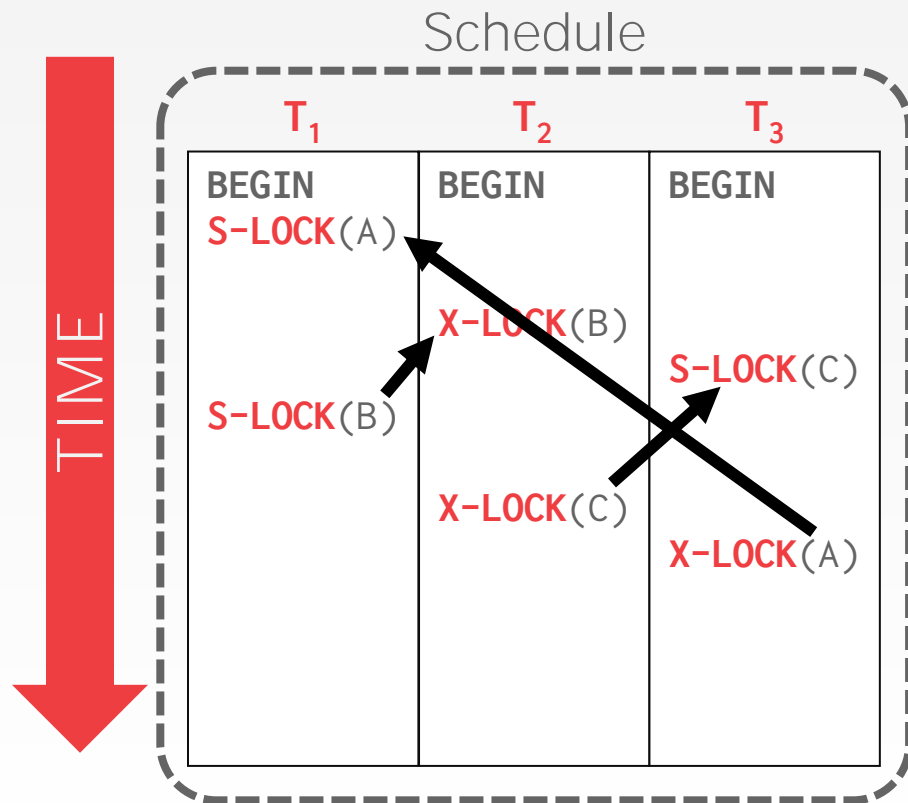
# DEADLOCK DETECTION



# DEADLOCK DETECTION



# DEADLOCK DETECTION



# 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 have to 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**

**Approach #2: Minimally**



# 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*.

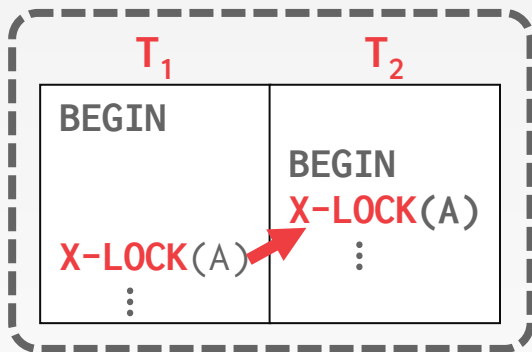
→ Otherwise *requesting txn* aborts.

## 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.

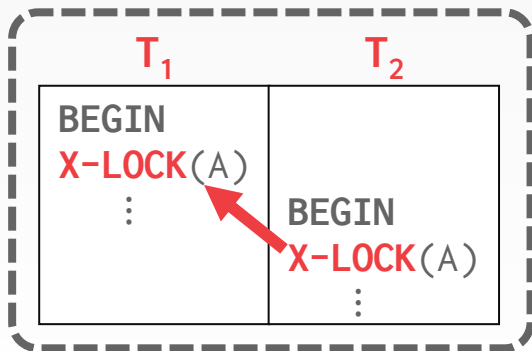
# DEADLOCK PREVENTION



Wait-Die



Wound-Wait



Wait-Die



Wound-Wait



# 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. Why?

# 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?

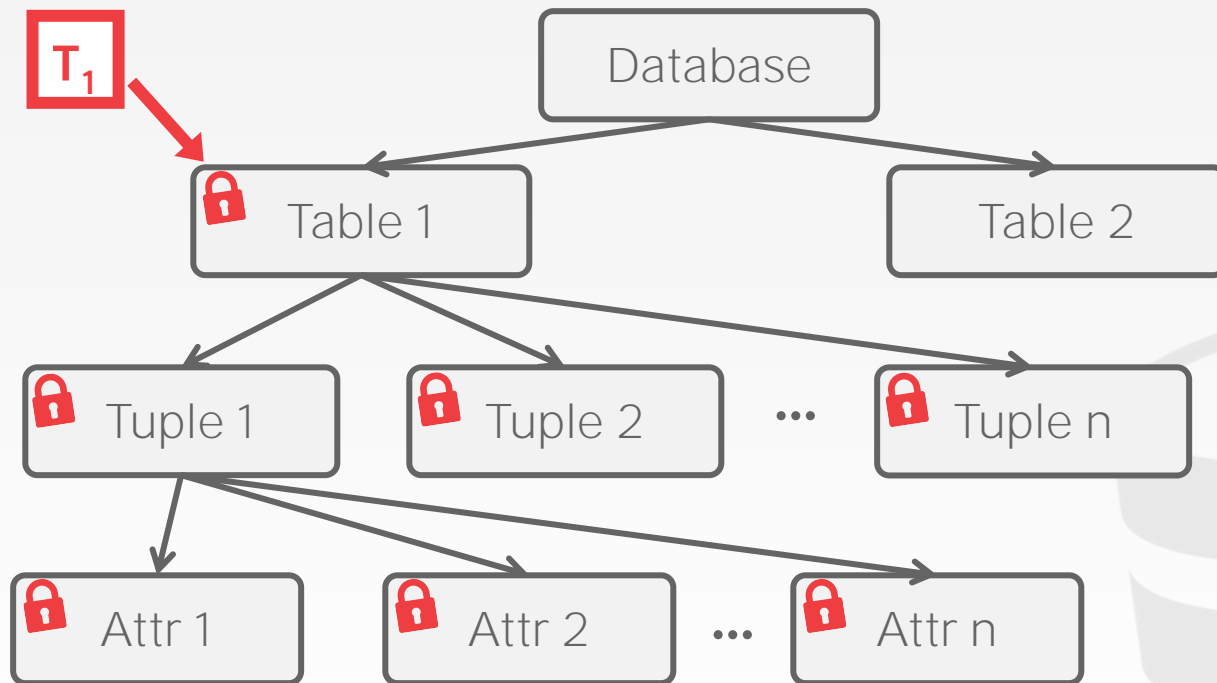
The DBMS should ideally obtain fewest number of locks that a txn needs.

Trade-off between parallelism versus overhead.

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



# DATABASE LOCK HIERARCHY



## EXAMPLE

---

$T_1$  – Get the balance of Andy's shady off-shore bank account.

$T_2$  – Increase Biden's bank account balance by 1%.

*What locks should these txns obtain?*

- Exclusive + Shared for leaf nodes of lock tree.
- Special Intention locks for higher levels.

# 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 shared locks.

## Intention-Exclusive (**IX**)

→ Indicates explicit locking at lower level with exclusive 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

		$T_2$ Wants				
$T_1$ Holds		IS	IX	S	SIX	X
	IS	✓	✓	✓	✓	×
	IX	✓	✓	×	×	×
	S	✓	×	✓	×	×
	SIX	✓	×	×	×	×
	X	×	×	×	×	×

# LOCKING PROTOCOL

---

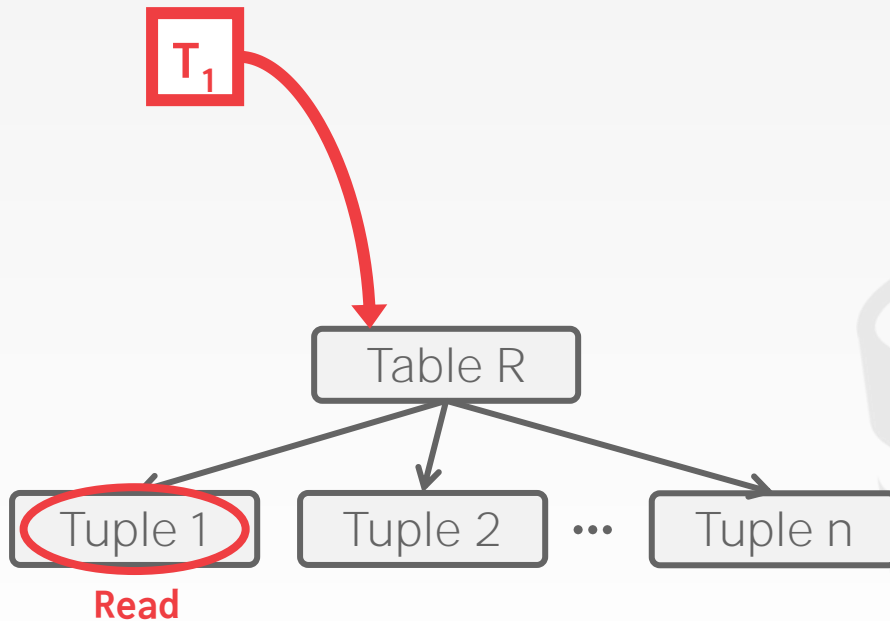
Each txn obtains appropriate lock at highest level of the database hierarchy.

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.

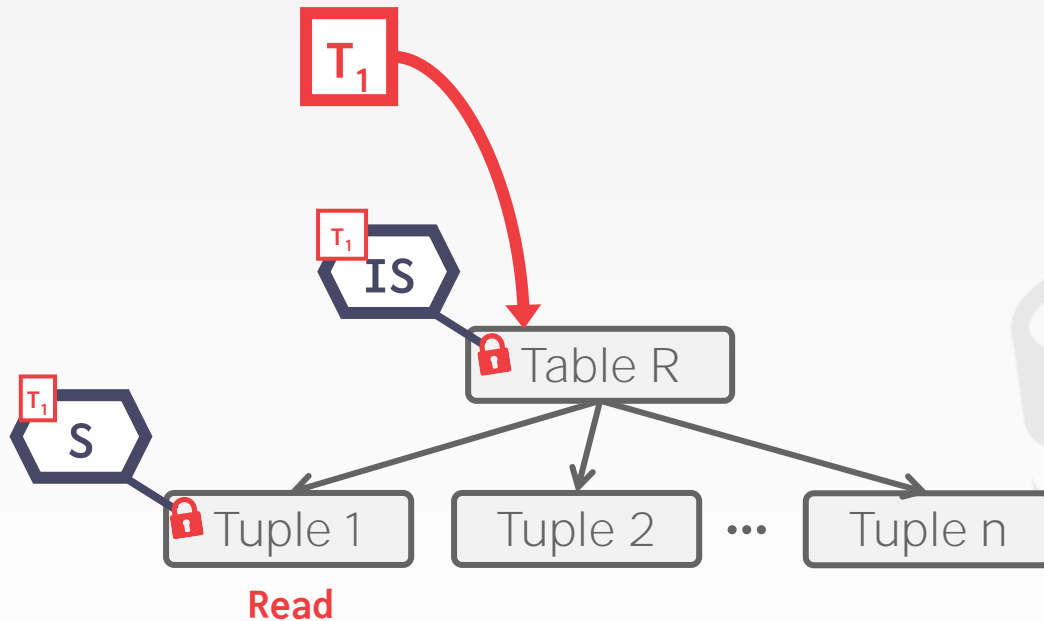
# EXAMPLE – TWO-LEVEL HIERARCHY

Read Andy's record in R.



# EXAMPLE – TWO-LEVEL HIERARCHY

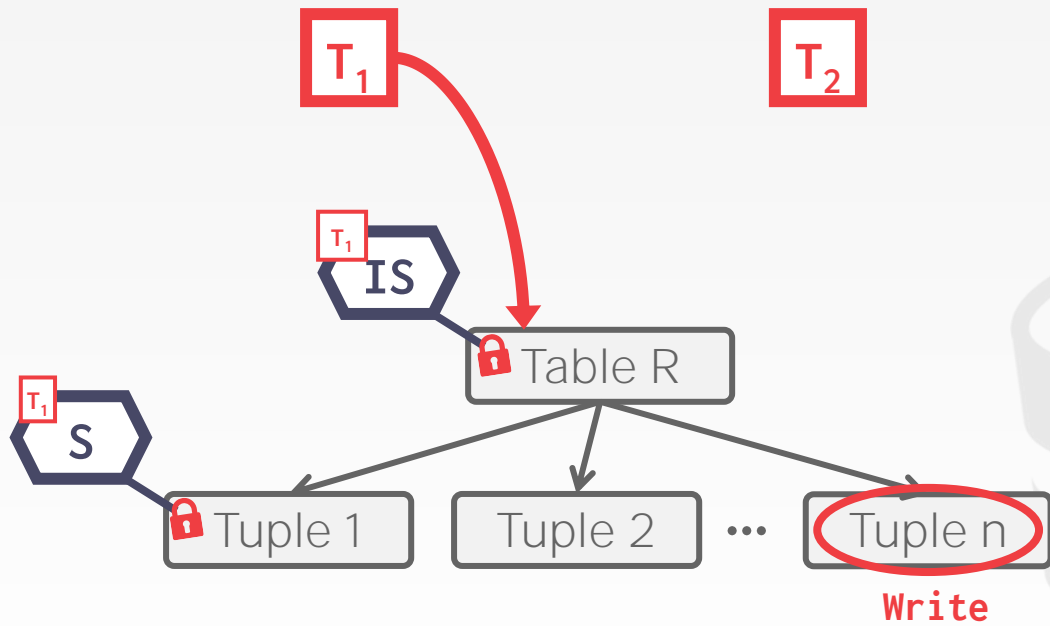
Read Andy's record in R.





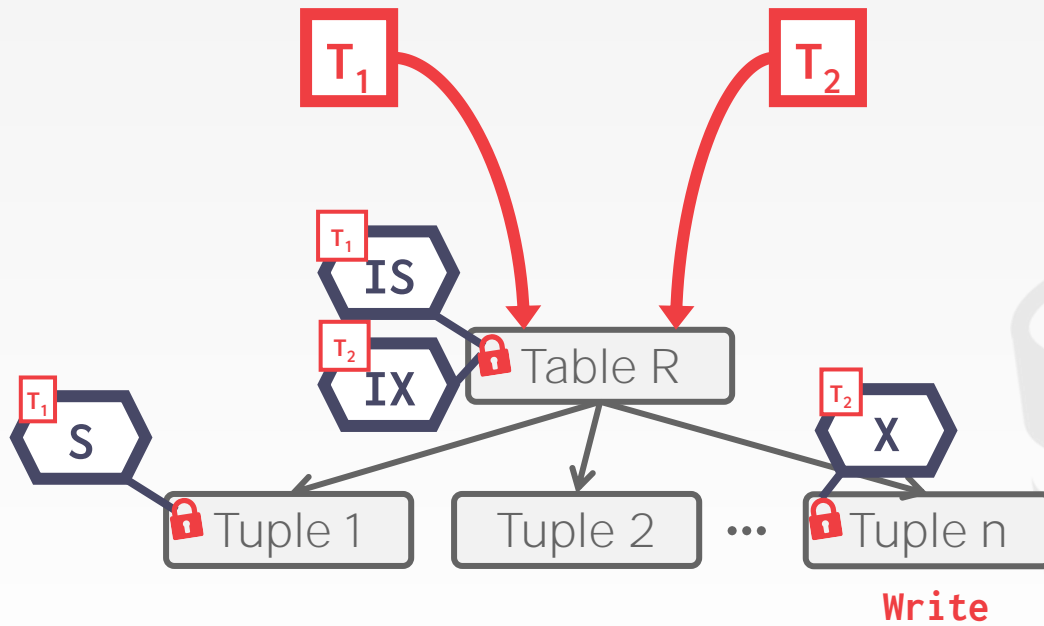
# EXAMPLE – TWO-LEVEL HIERARCHY

Update Biden's record in R.



# EXAMPLE – TWO-LEVEL HIERARCHY

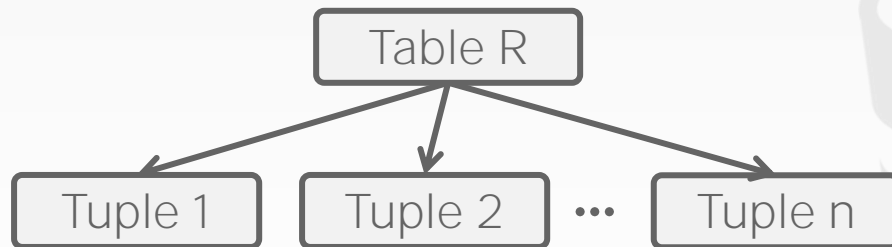
Update Biden's record in R.



## EXAMPLE – THREESOME

Assume three txns execute at same time:

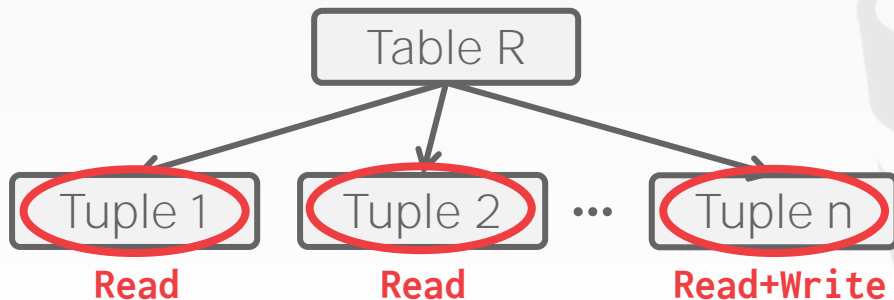
- $T_1$  – Scan **R** and update a few tuples.
- $T_2$  – Read a single tuple in **R**.
- $T_3$  – Scan all tuples in **R**.



## EXAMPLE – THREESOME

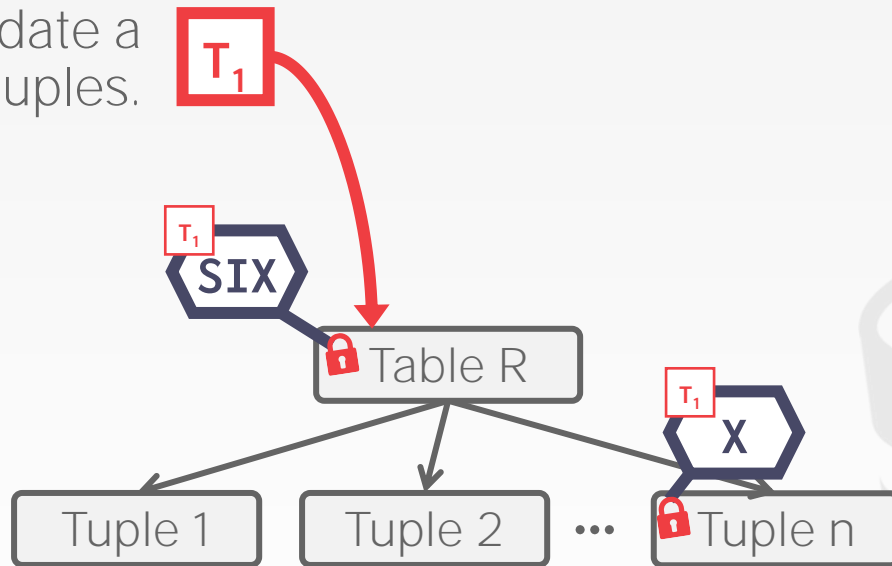
Scan **R** and update a few tuples.

**T<sub>1</sub>**

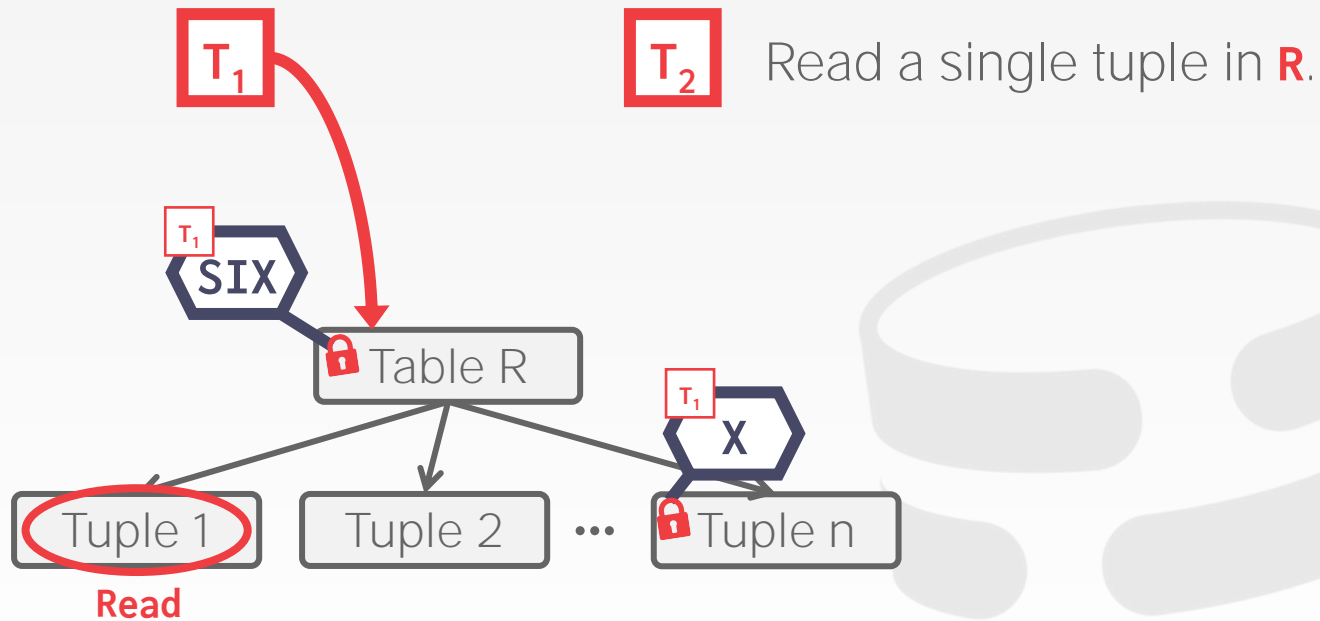


# EXAMPLE – THREESOME

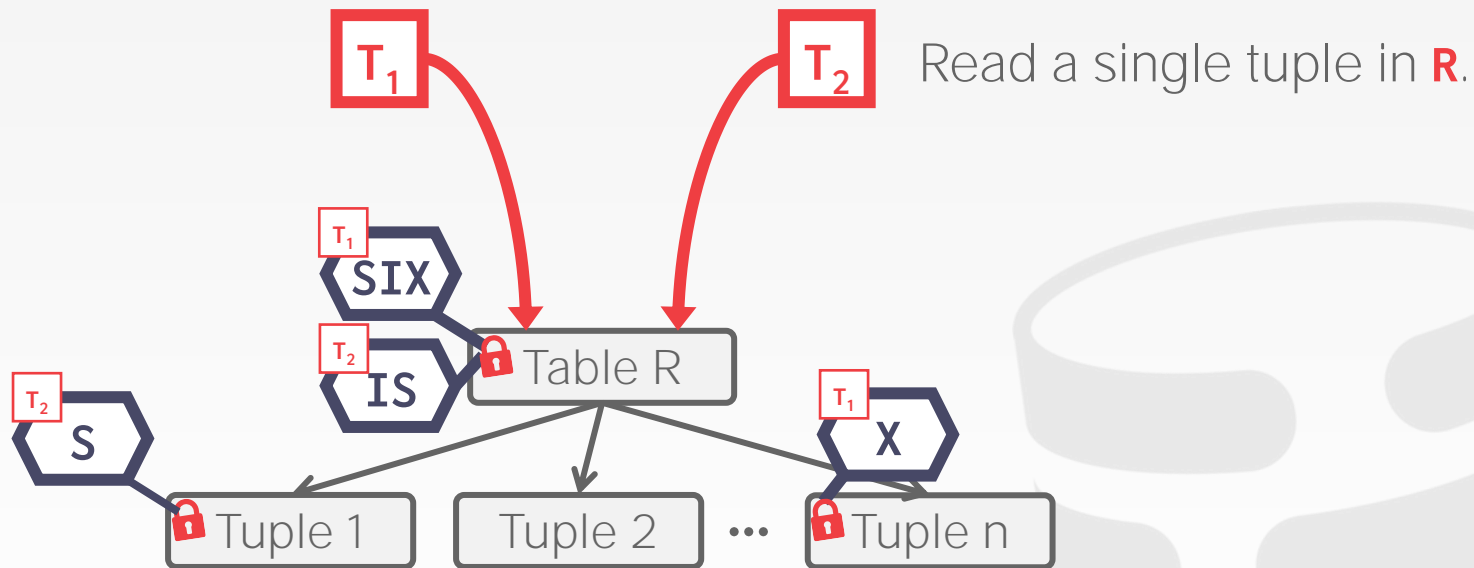
Scan **R** and update a few tuples.



## EXAMPLE – THREESOME

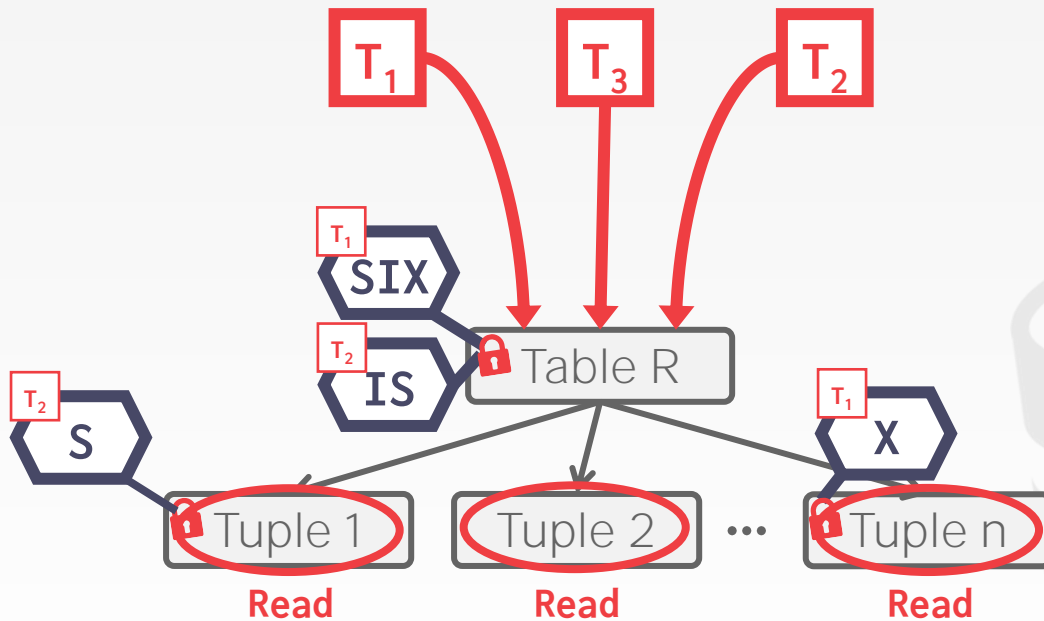


## EXAMPLE – THREESOME



# EXAMPLE – THREESOME

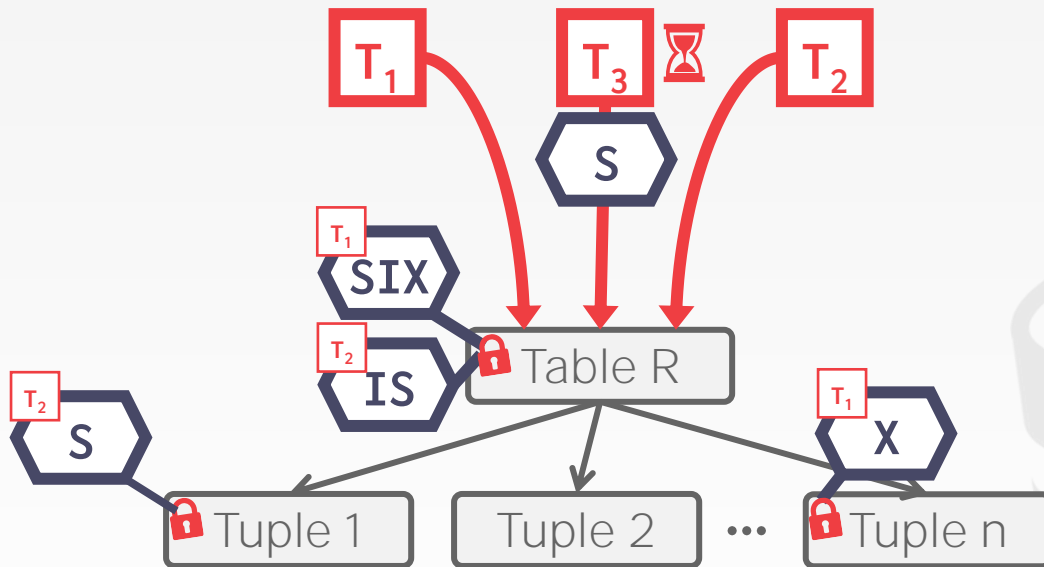
Scan all tuples in **R**.





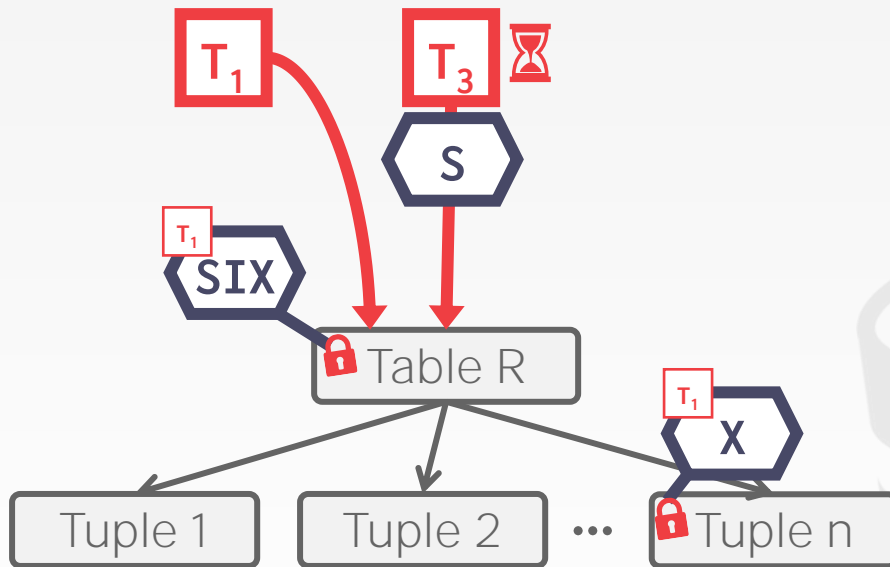
# EXAMPLE – THREESOME

Scan all tuples in **R**.



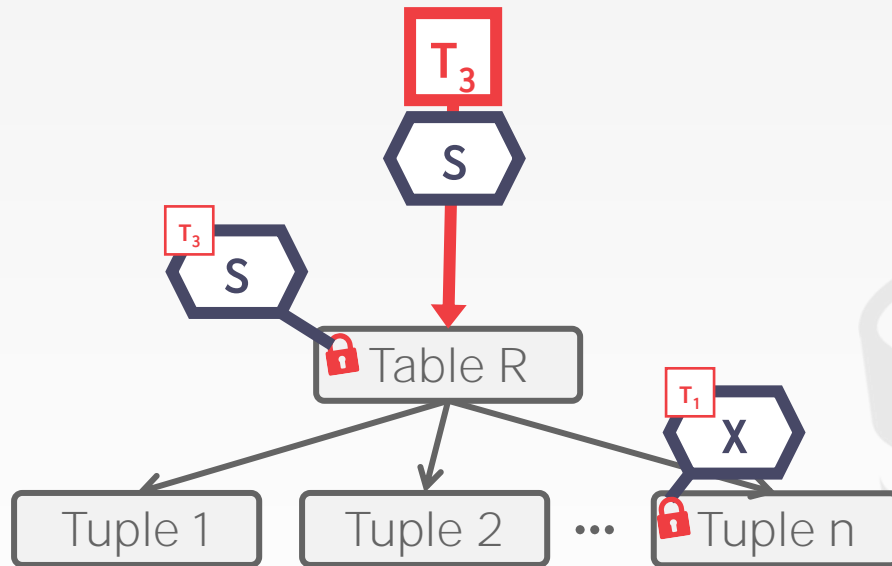
# EXAMPLE – THREESOME

Scan all tuples in **R**.



# EXAMPLE – THREESOME

Scan all tuples in **R**.



# MULTIPLE LOCK GRANULARITIES

---

Hierarchical locks are useful in practice as each txn only needs a few locks.

Intention locks help improve concurrency:

- **Intention-Shared (IS)**: Intent to get **S** lock(s) at finer granularity.
- **Intention-Exclusive (IX)**: Intent to get **X** lock(s) at finer granularity.
- **Shared+Intention-Exclusive (SIX)**: Like **S** and **IX** at the same time.

# LOCK ESCALATION

---

Lock escalation dynamically asks for coarser-grained locks when too many low-level locks acquired.

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

# LOCKING IN PRACTICE

---

You typically don't set locks manually in txns.

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

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 DBMS.

Automatically generates correct interleaving:

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



# NEXT CLASS

---

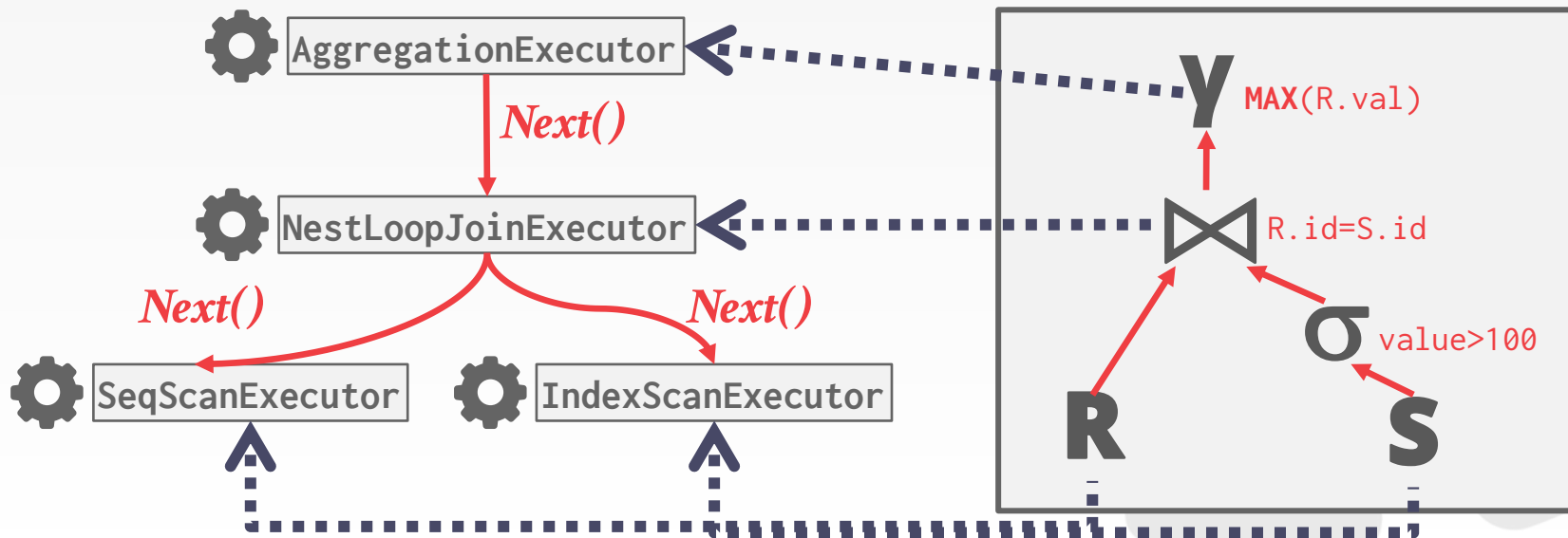
## Timestamp Ordering Concurrency Control



# PROJECT #3 – QUERY EXECUTION

You will build a query execution engine in your DBMS.

```
SELECT MAX(R.val)
FROM R JOIN S
      ON R.id = S.id
WHERE S.value > 100
```



# PROJECT #3 – TASKS

---

Install Tables + Indexes in Catalog

Plan Node Executors

- Access Methods: Sequential Scan, Index Scan
- Modifications: Insert, Update, Delete
- Miscellaneous: Nest Loop Join, Index Join, Hash-based Aggregation, Limit/Offset

<https://15445.courses.cs.cmu.edu/fall2020/project3/>

# DEVELOPMENT HINTS

---

Implement the **Insert** and **Sequential Scan** executors first so that you can populate tables and read from it.

You do **not** need to worry about transactions.

The aggregation hash table does **not** need to be backed by your buffer pool (i.e., use STL)

Gradescope is for meant for grading, **not** debugging. Write your own local tests.

# THINGS TO NOTE

---

Do **not** change any file other than the ones that you submit to Gradescope.

Rebase on top of the latest BusTub master branch.

Post your questions on Piazza or come to TA office hours.

# PLAGIARISM WARNING

---

Your project implementation must be your own work.

- You may **not** copy source code from other groups or the web.
- Do **not** publish your implementation on Github.

Plagiarism will **not** be tolerated.  
See [CMU's Policy on Academic Integrity](#) for additional information.

