University of Michigan

23 Two-Phase Locking

Database Management Systems
EECS 484
Fall 2024

LM
Lin Ma
Computer Science and
Engineering Division

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.
→ Lin doesn't know of any DBMS that supports this.

EXAMPLE

Schedule

TIME

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

EXAMPLE

Schedule

TIME

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

OBSERVATION

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

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

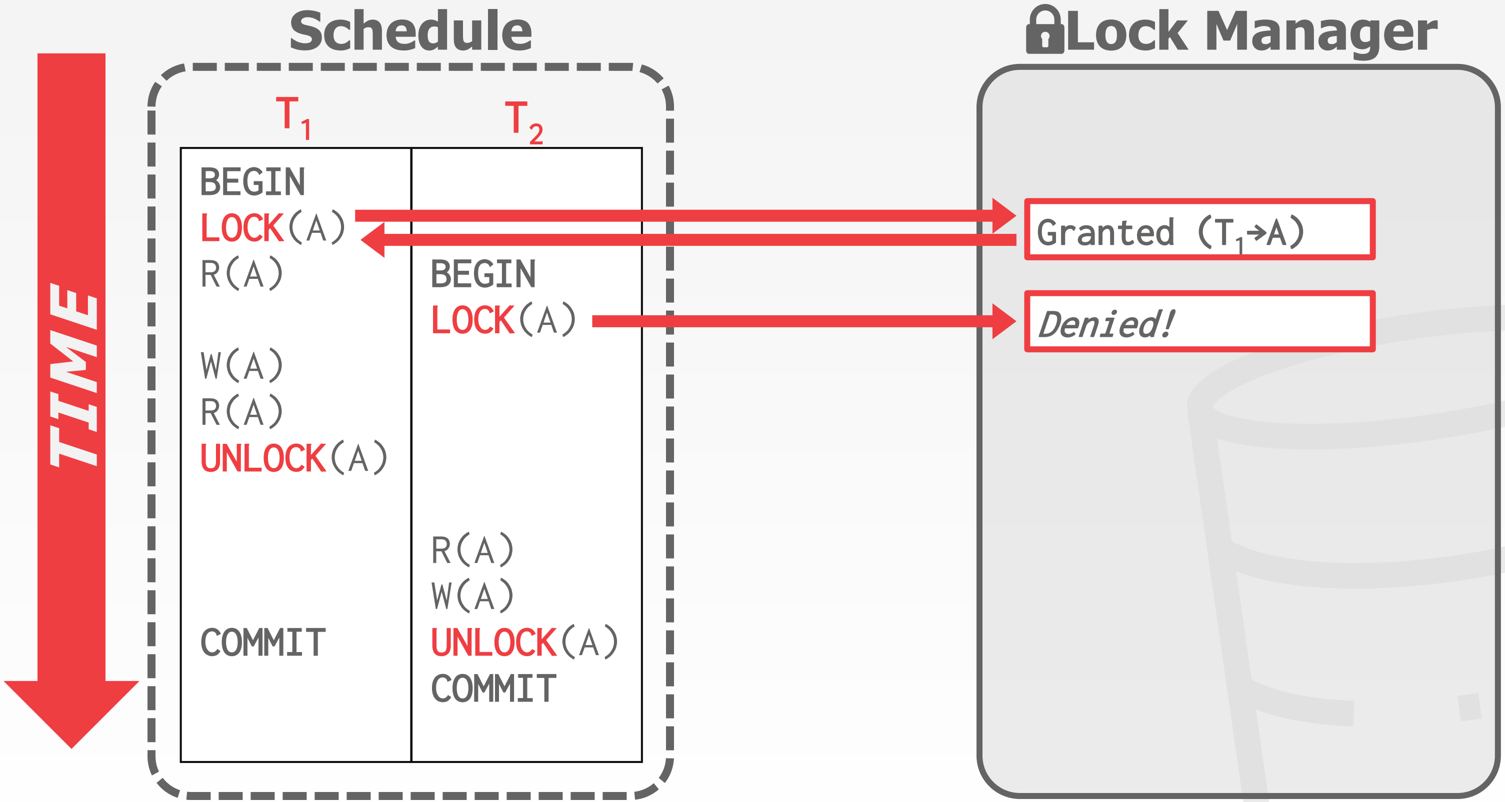
TIME

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

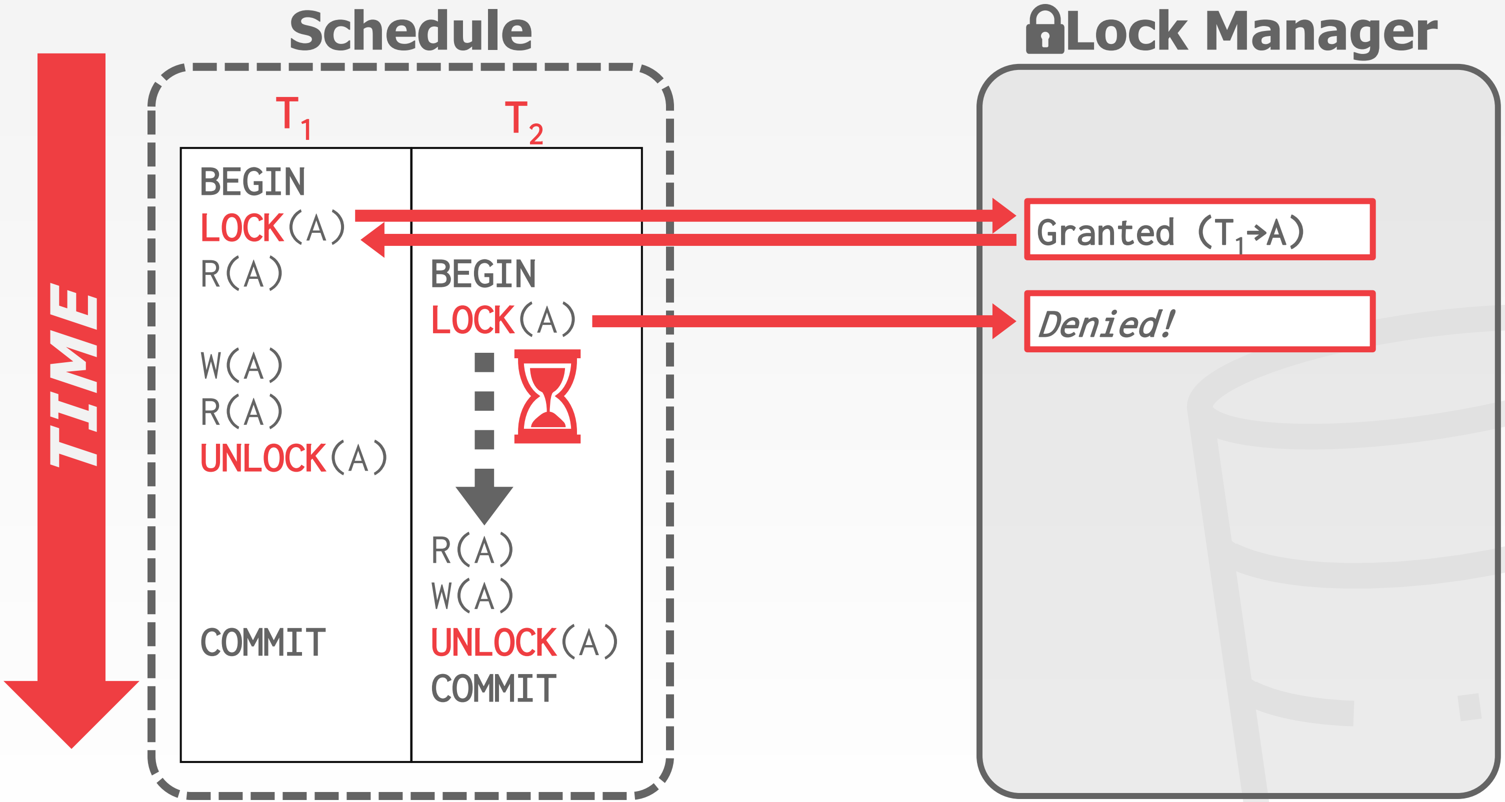
Lock Manager

Granted (T_1→A)

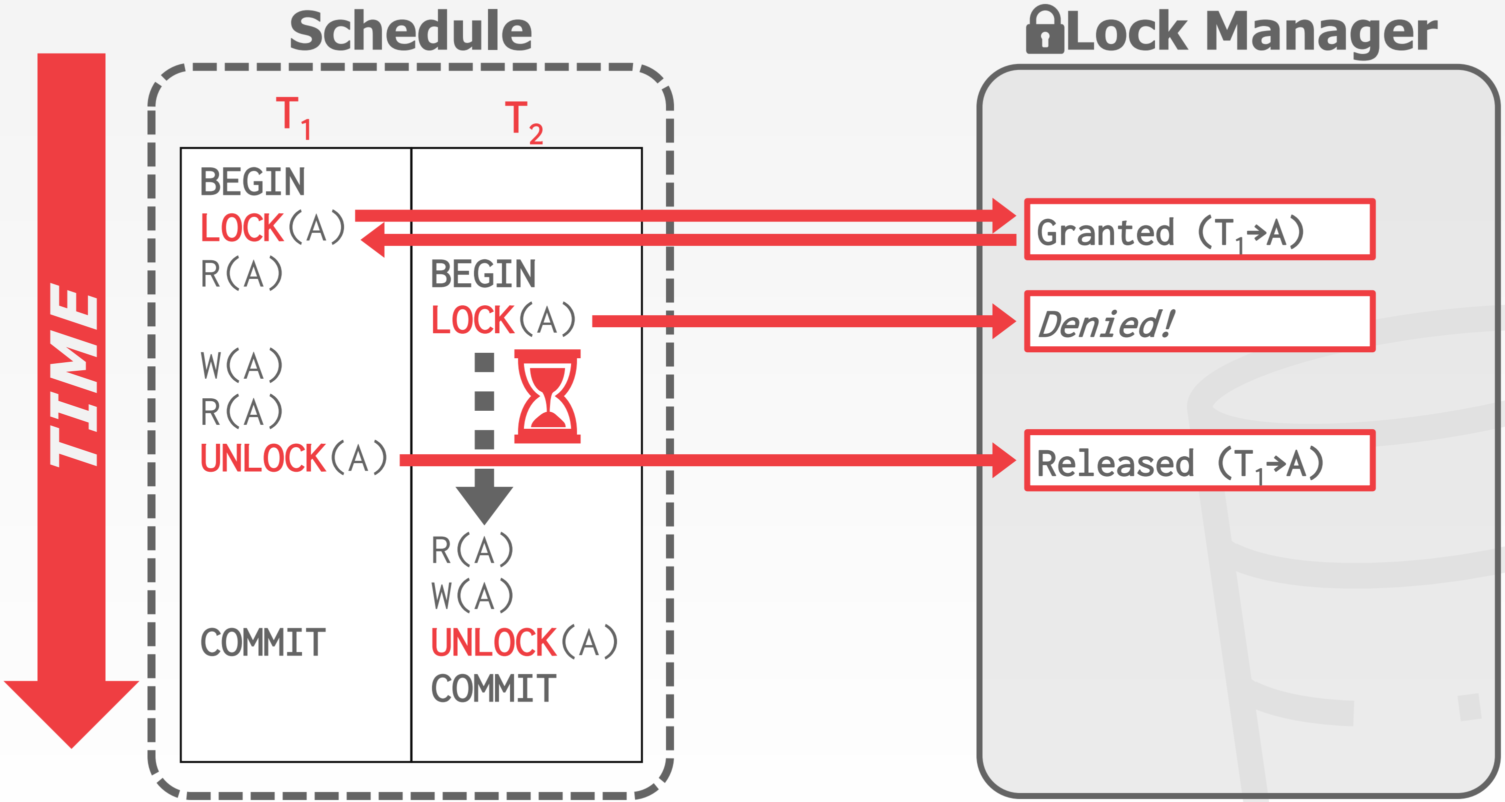
EXECUTING WITH LOCKS



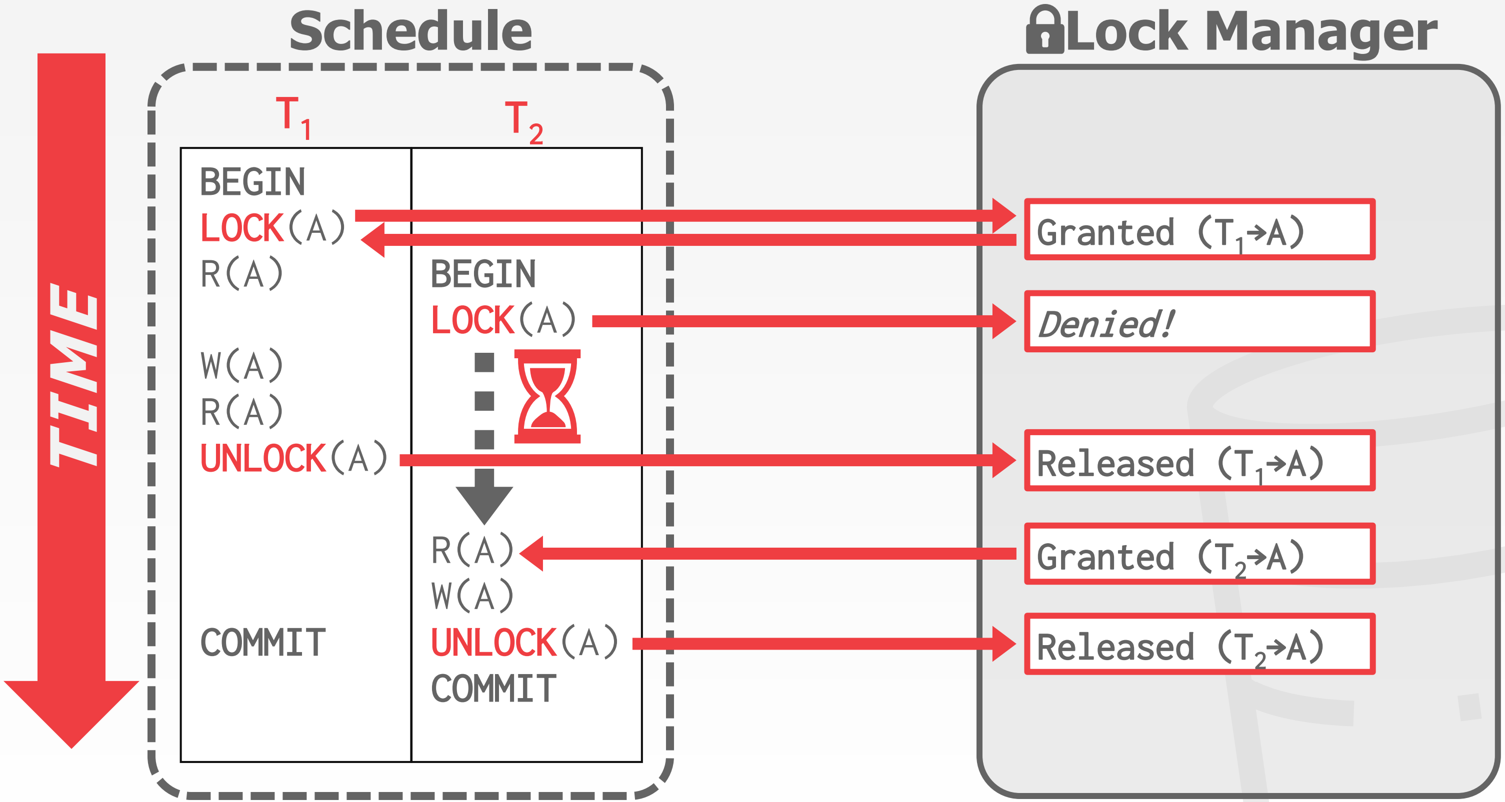
EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



TODAY'S AGENDA

Lock Types

Two-Phase Locking

Deadlock Detection + Prevention

Isolation Levels

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

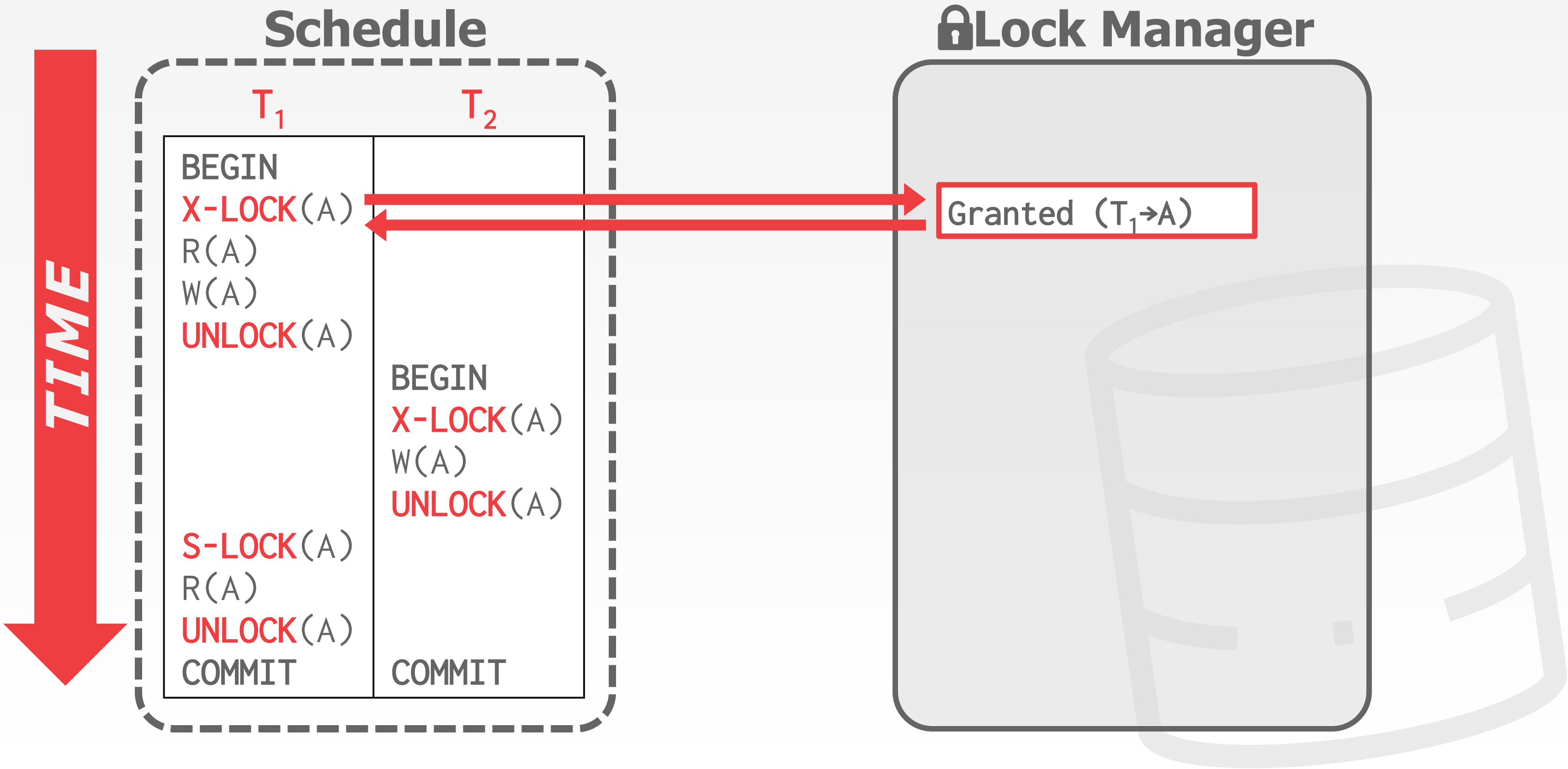
Schedule

TIME

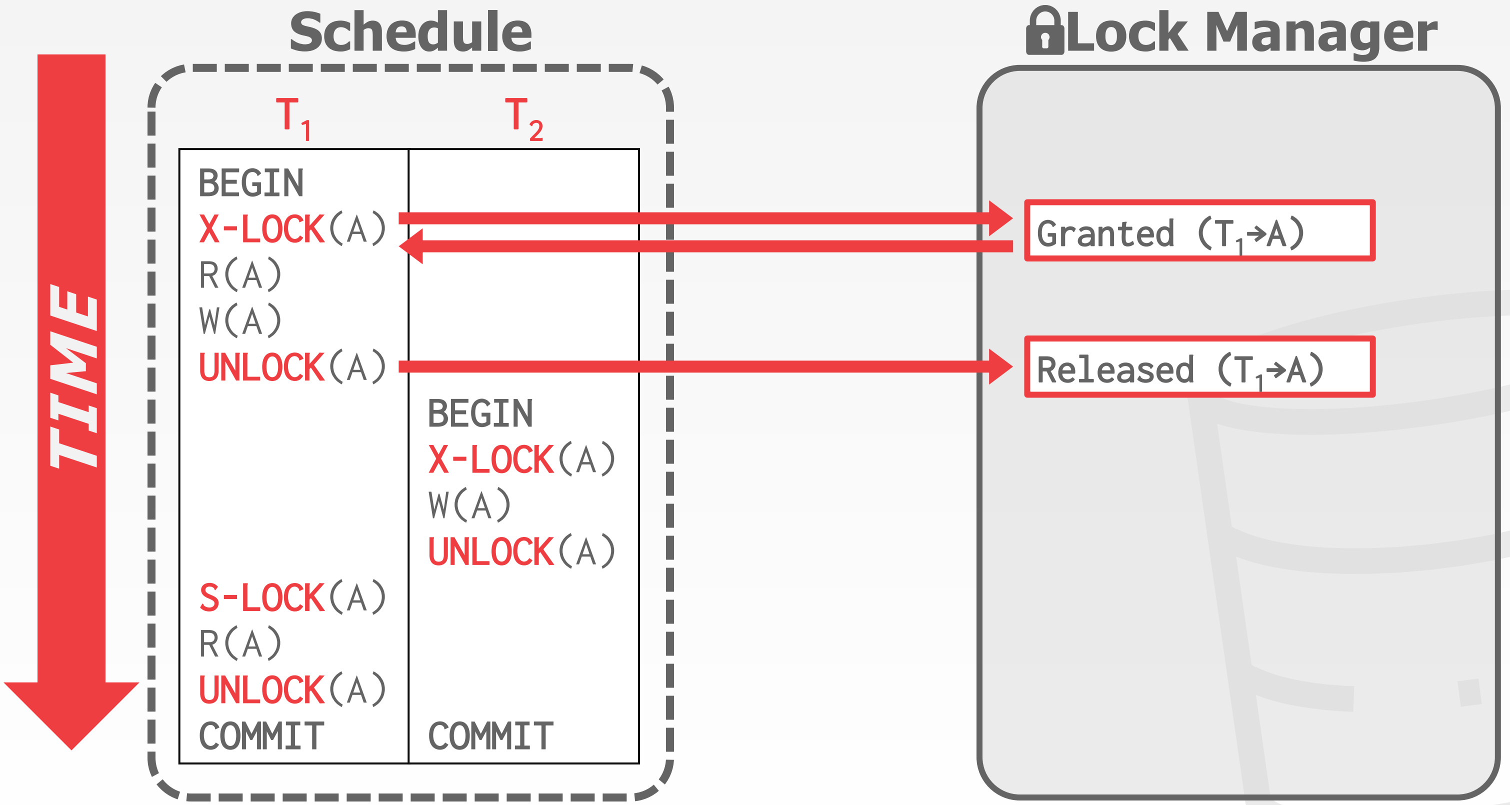
T$_1$	T$_2$
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Lock Manager

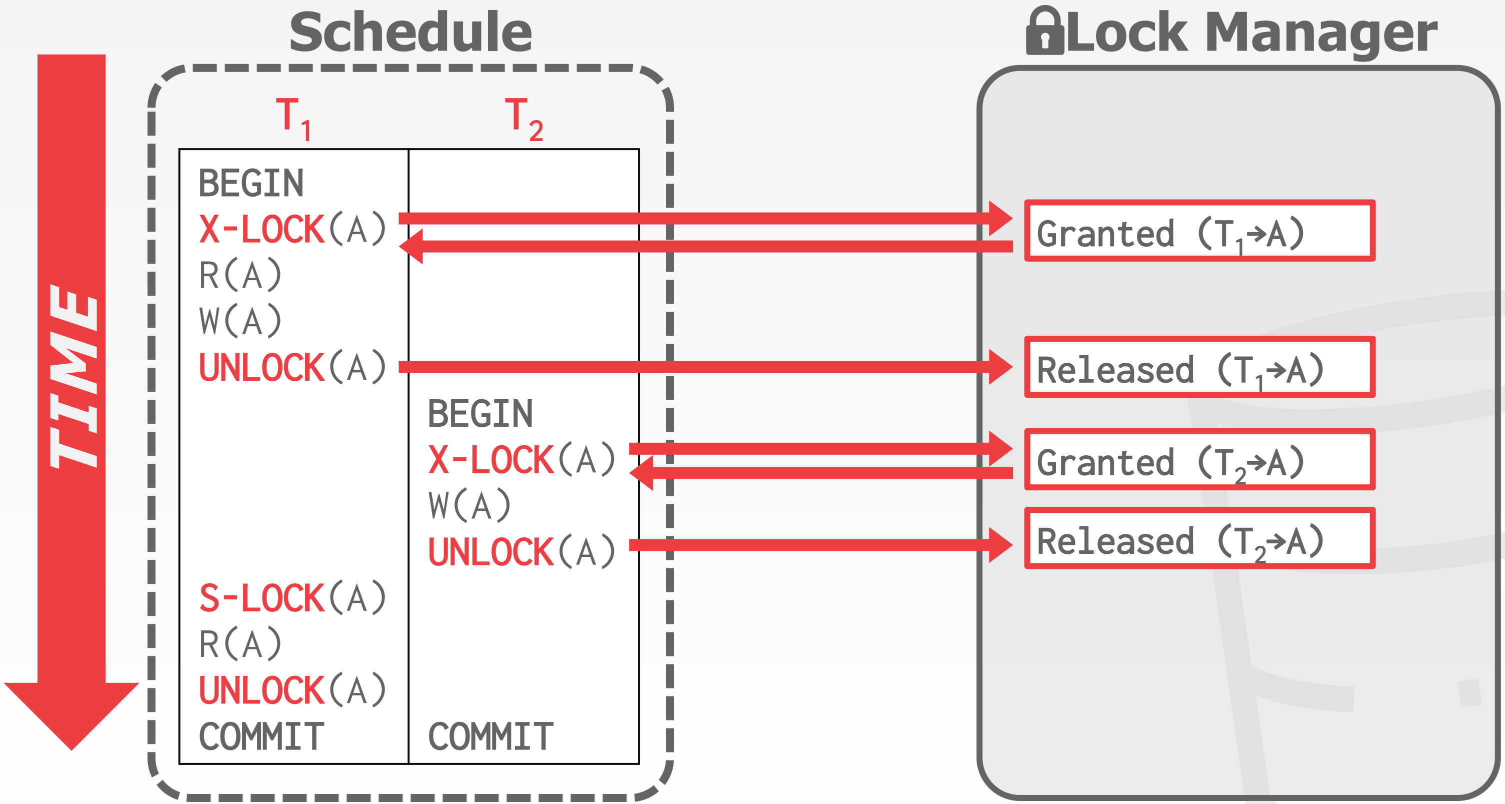
EXECUTING WITH LOCKS



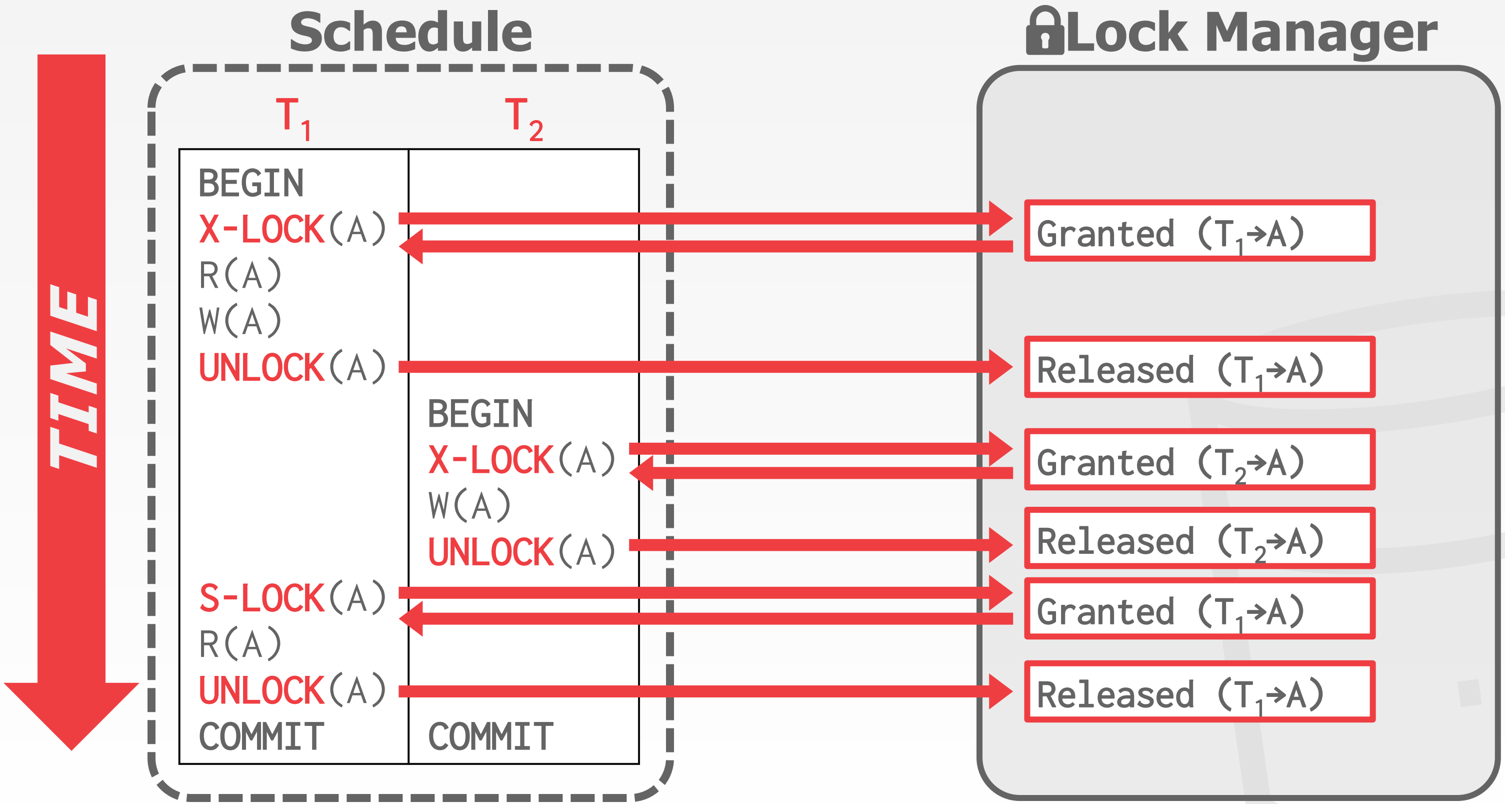
EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



EXECUTING WITH LOCKS



EXECUTING WITH LOCKS

Schedule

TIME

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 ($T_1 \rightarrow A$)

Released ($T_1 \rightarrow A$)

Granted ($T_2 \rightarrow A$)

Released ($T_2 \rightarrow A$)

Granted ($T_1 \rightarrow A$)

Released ($T_1 \rightarrow A$)

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 <u>not</u> 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.

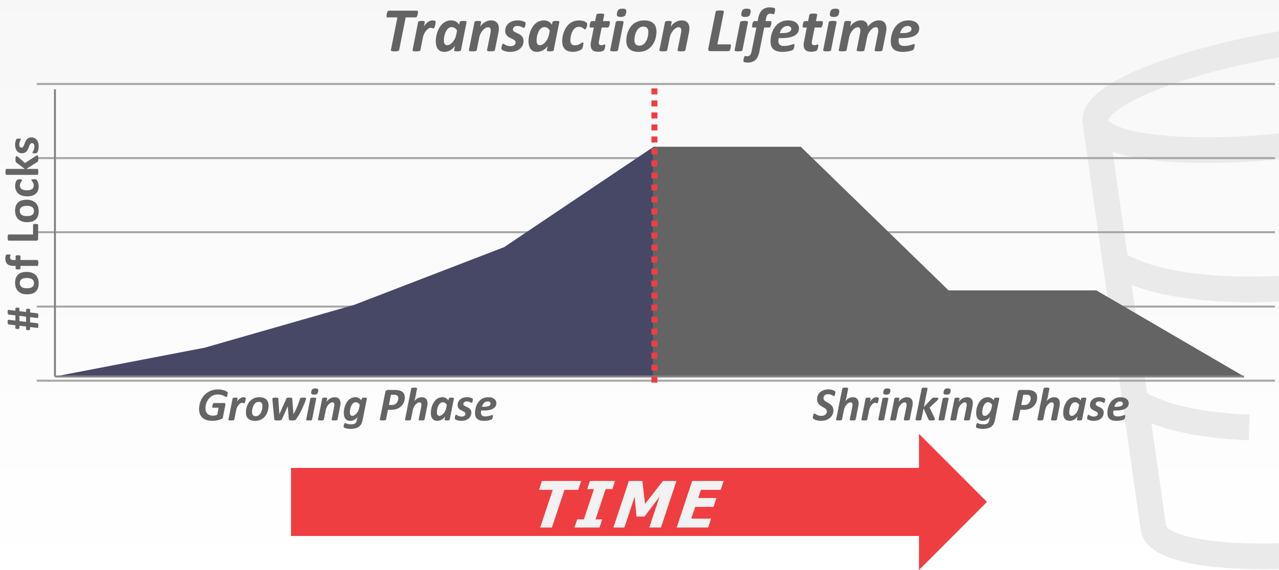
→ The lock manager grants/denies lock requests.

Phase #2: Shrinking

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

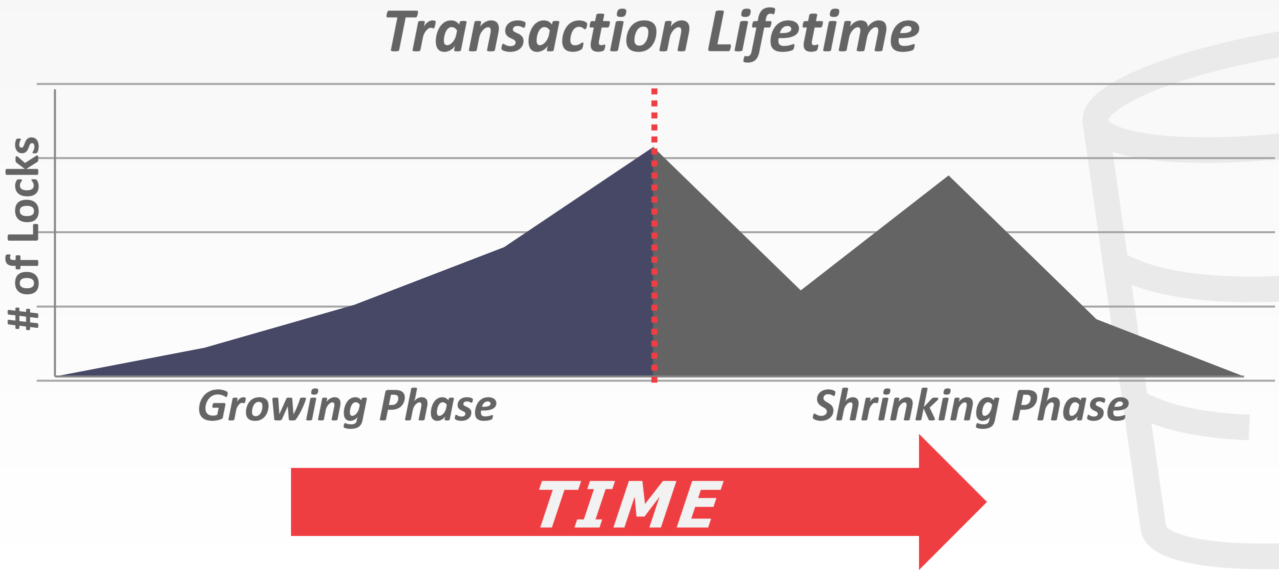
TWO-PHASE LOCKING

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



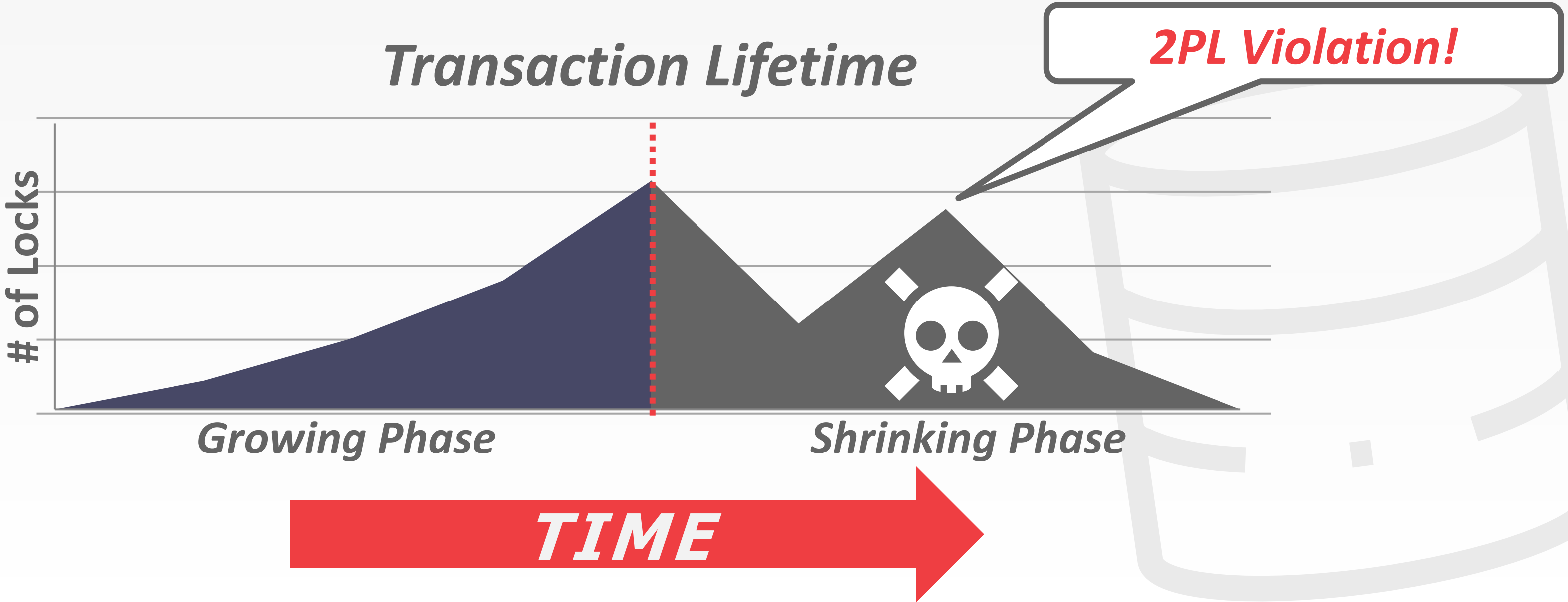
TWO-PHASE LOCKING

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



TWO-PHASE LOCKING

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



EXECUTING WITH 2PL

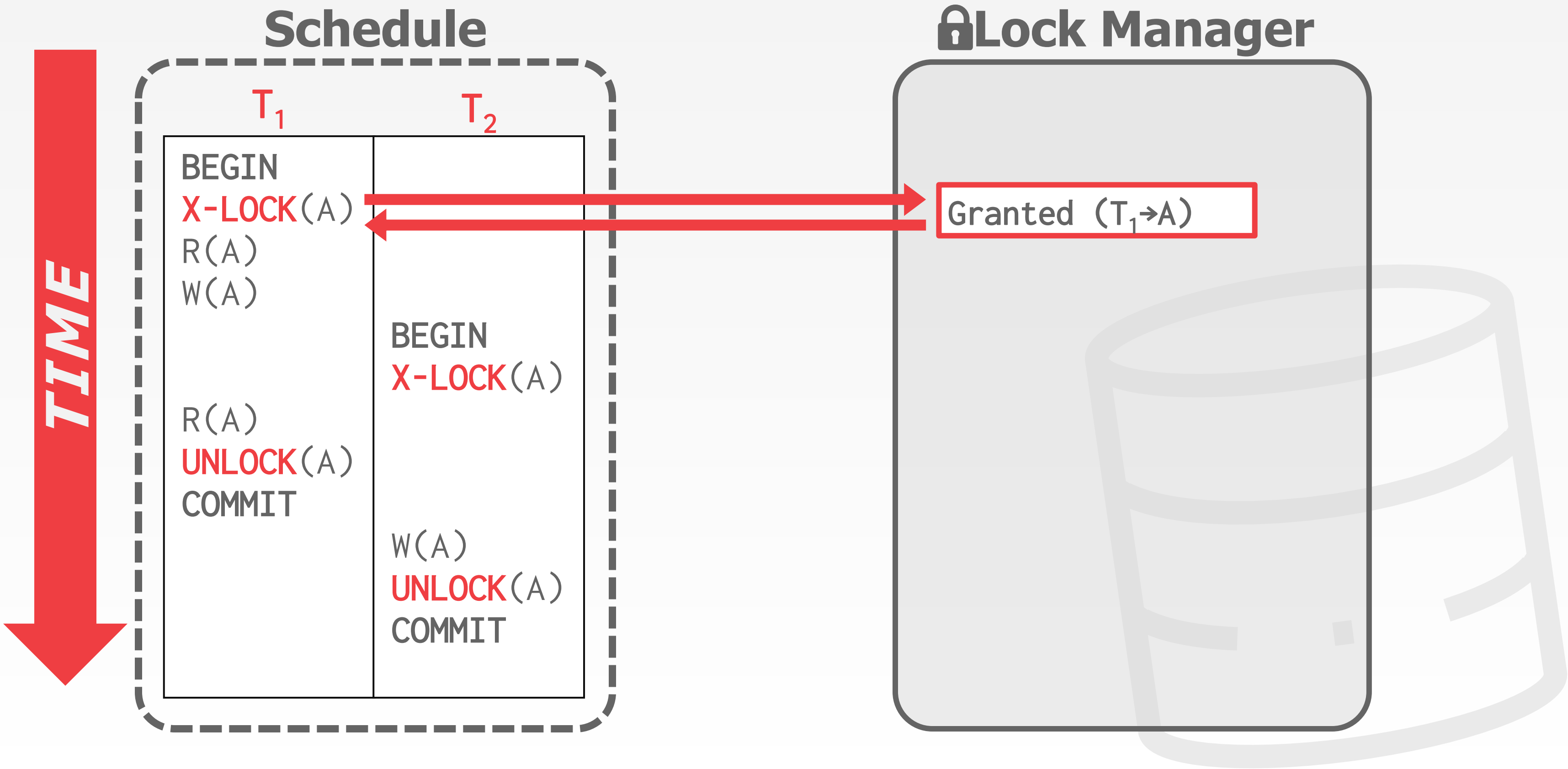
Schedule

TIME

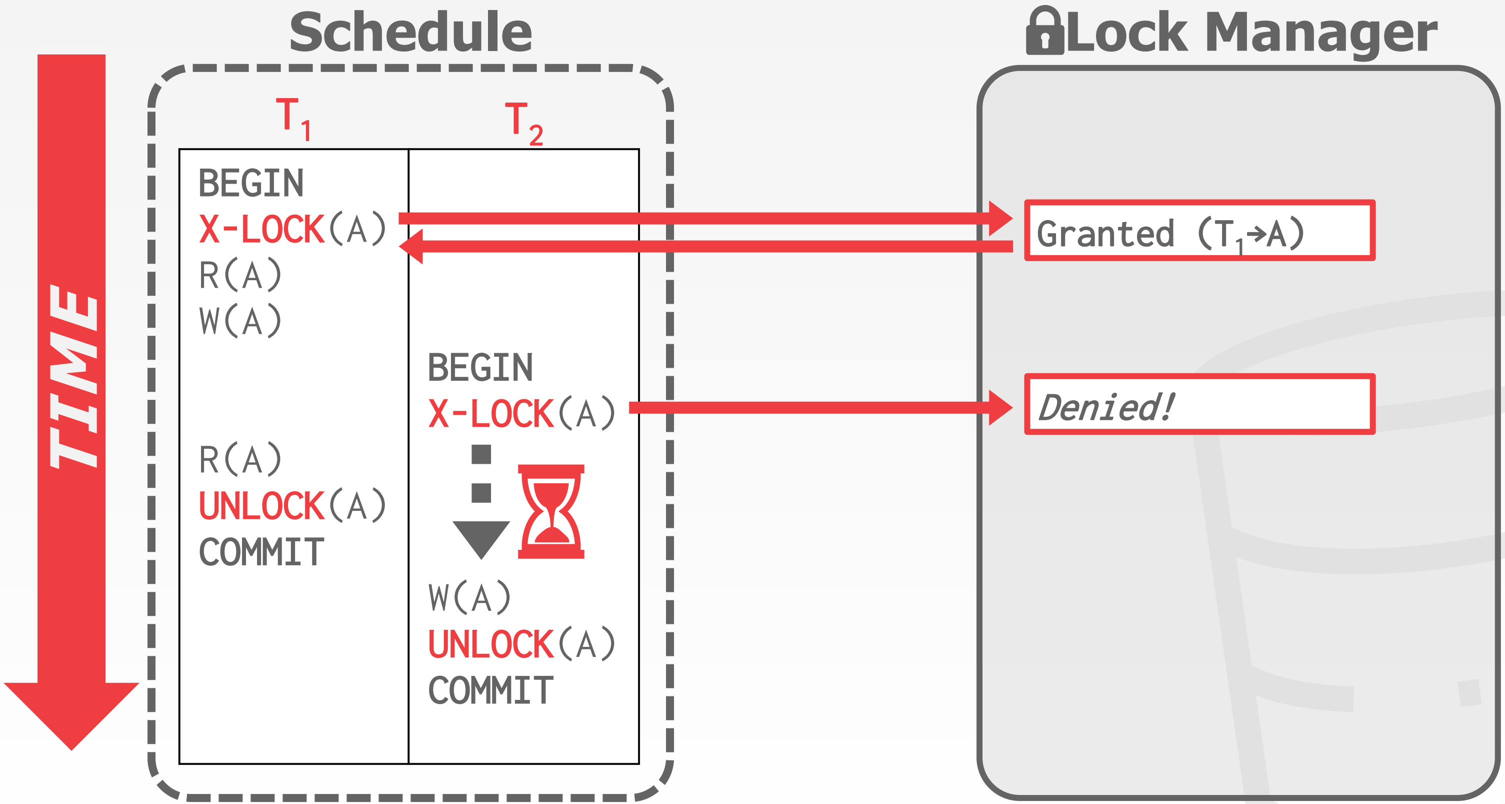
T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
	BEGIN
	X-LOCK(A)
R(A)	
UNLOCK(A)	
COMMIT	
	W(A)
	UNLOCK(A)
	COMMIT

Lock Manager

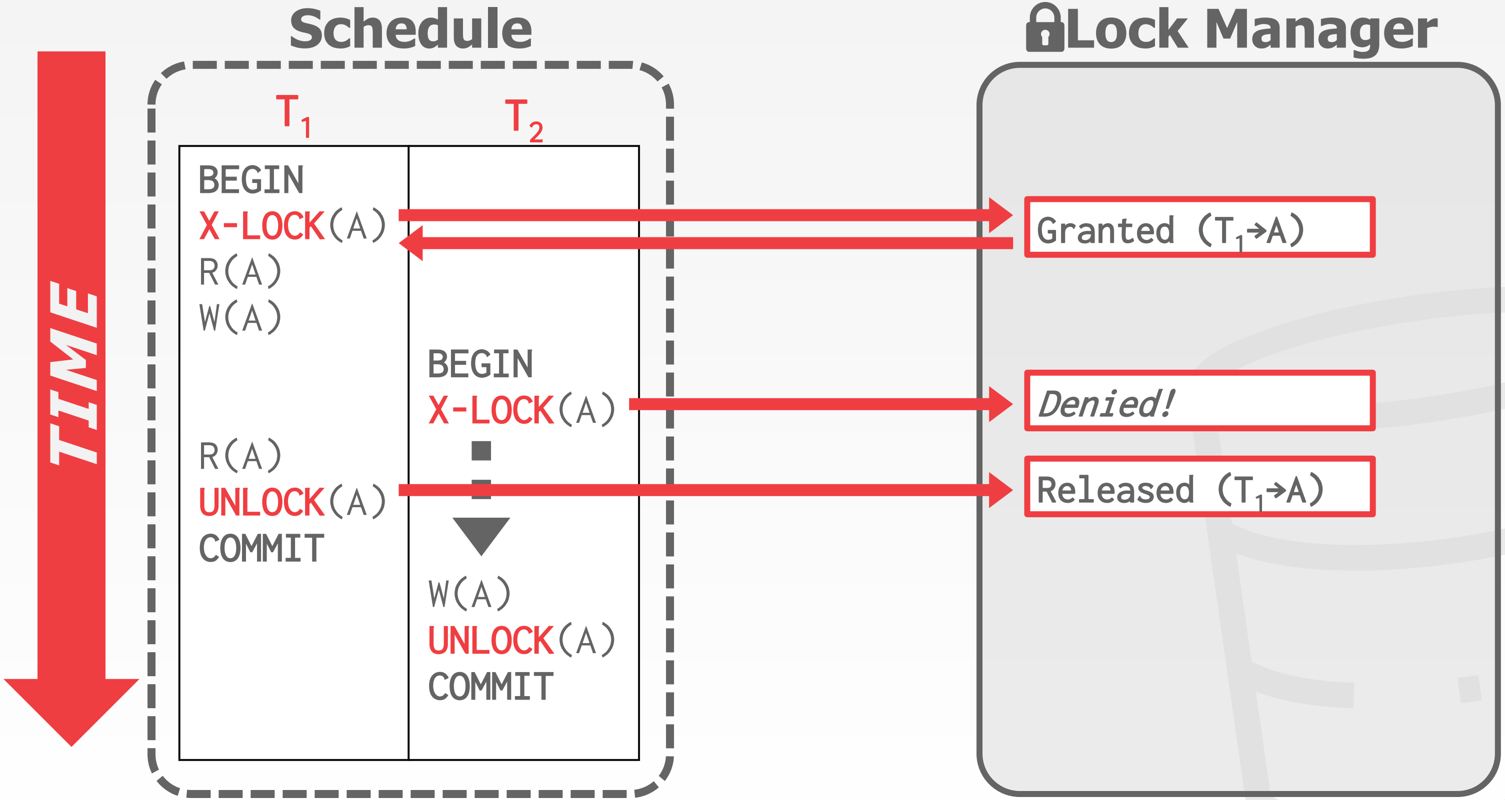
EXECUTING WITH 2PL



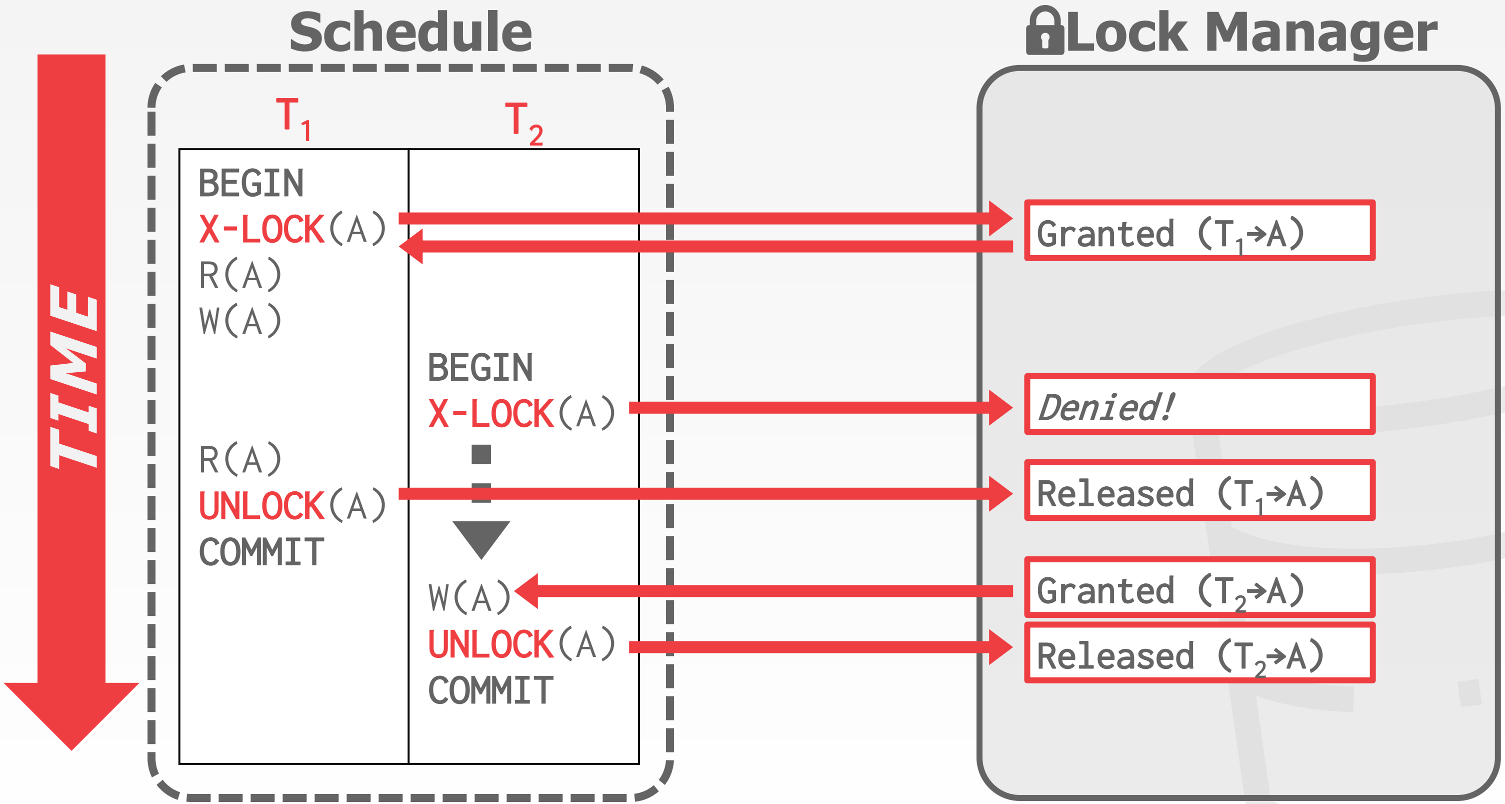
EXECUTING WITH 2PL



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

Schedule

TIME

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	

2PL – CASCADING ABORTS

Schedule

TIME

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	

2PL – CASCADING ABORTS

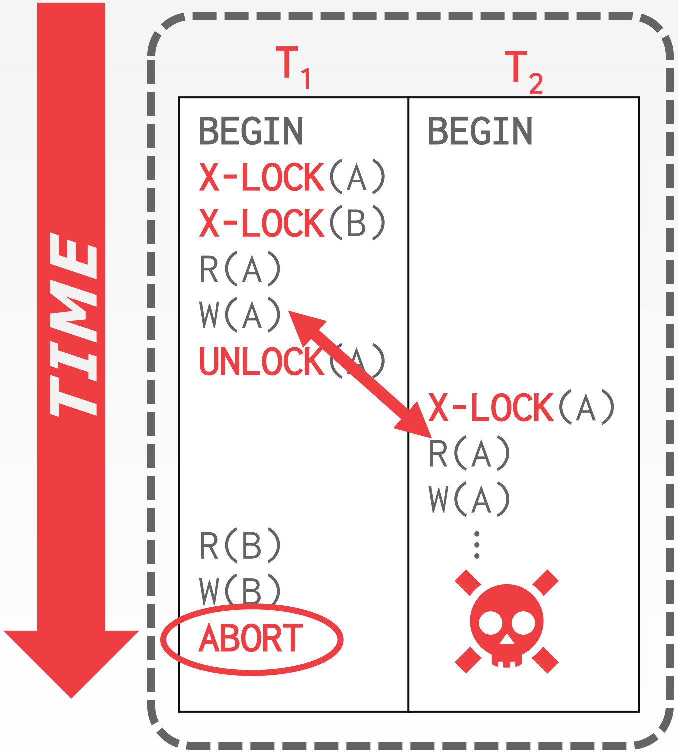
Schedule

TIME

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	

2PL – CASCADING ABORTS

Schedule



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

TIME

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	

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.

→ Locking limits concurrency.

May still have "dirty reads".

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

May lead to deadlocks.

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

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)**

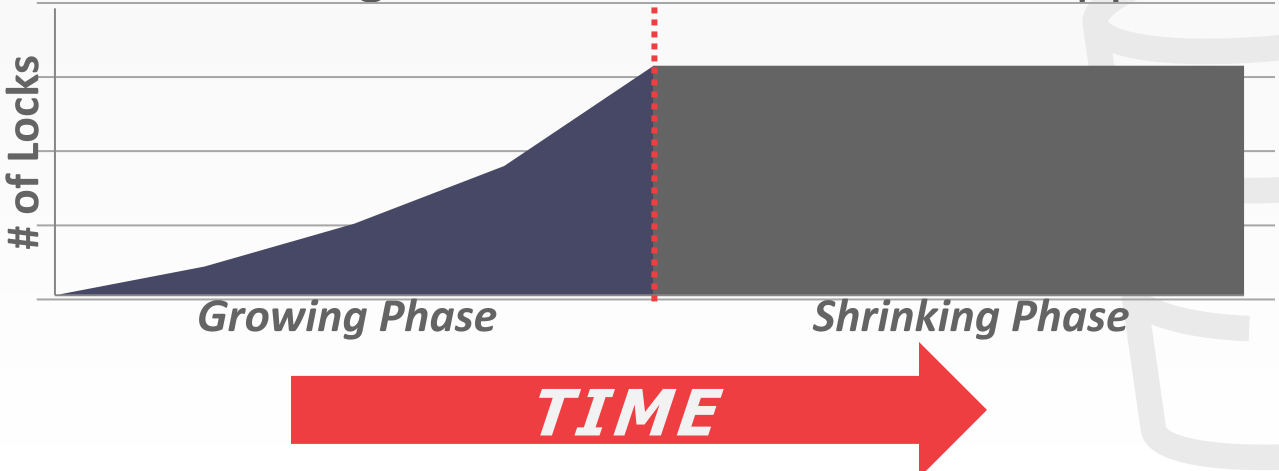
May lead to deadlocks.

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

STRONG STRICT TWO-PHASE LOCKING

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

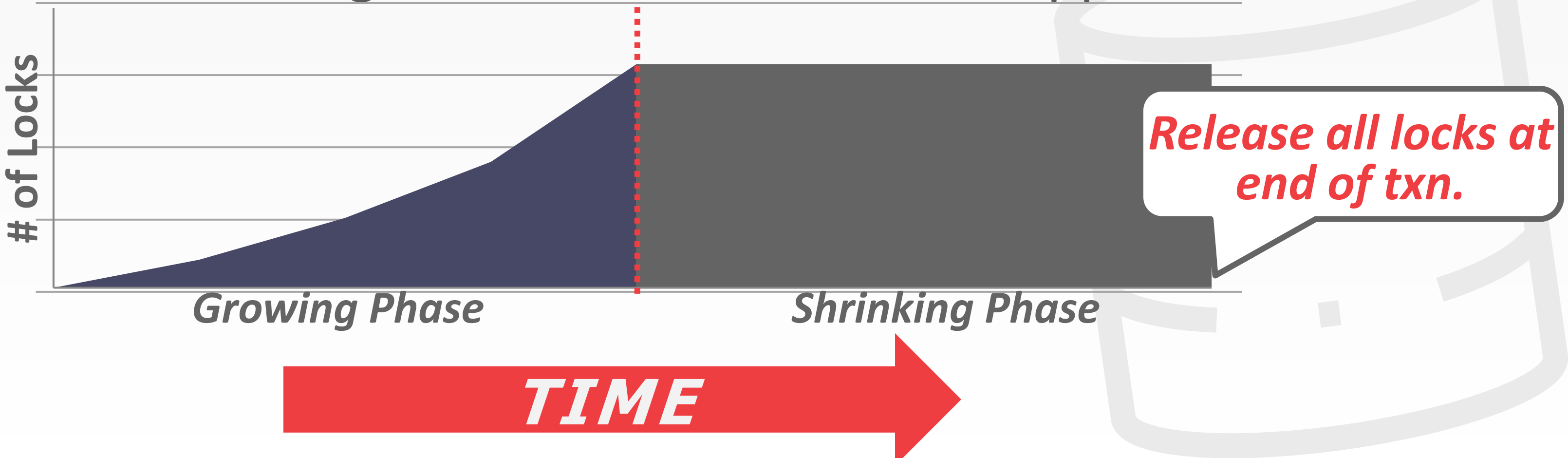
Allows only conflict serializable schedules, but it is often stronger than needed for some apps.



STRONG STRICT TWO-PHASE LOCKING

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

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 Lin's account (A) to his friend'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
```

EXAMPLES

T_1 – Move $100 from Lin's account (A) to his friend'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	

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

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)
	ECHO A+B
R(B)	
	COMMIT
B=B+100	
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

T_2 Output

A+B=1900

2PL EXAMPLE

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

Initial Database State

A=1000, B=1000

2PL EXAMPLE

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

Initial Database State

A=1000, B=1000

2PL EXAMPLE

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

Initial Database State

A=1000, B=1000

2PL EXAMPLE

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

Initial Database State

A=1000, B=1000

T_2 Output

A+B=2000

STRONG STRICT 2PL EXAMPLE

Schedule

TIME

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

TIME

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

TIME

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

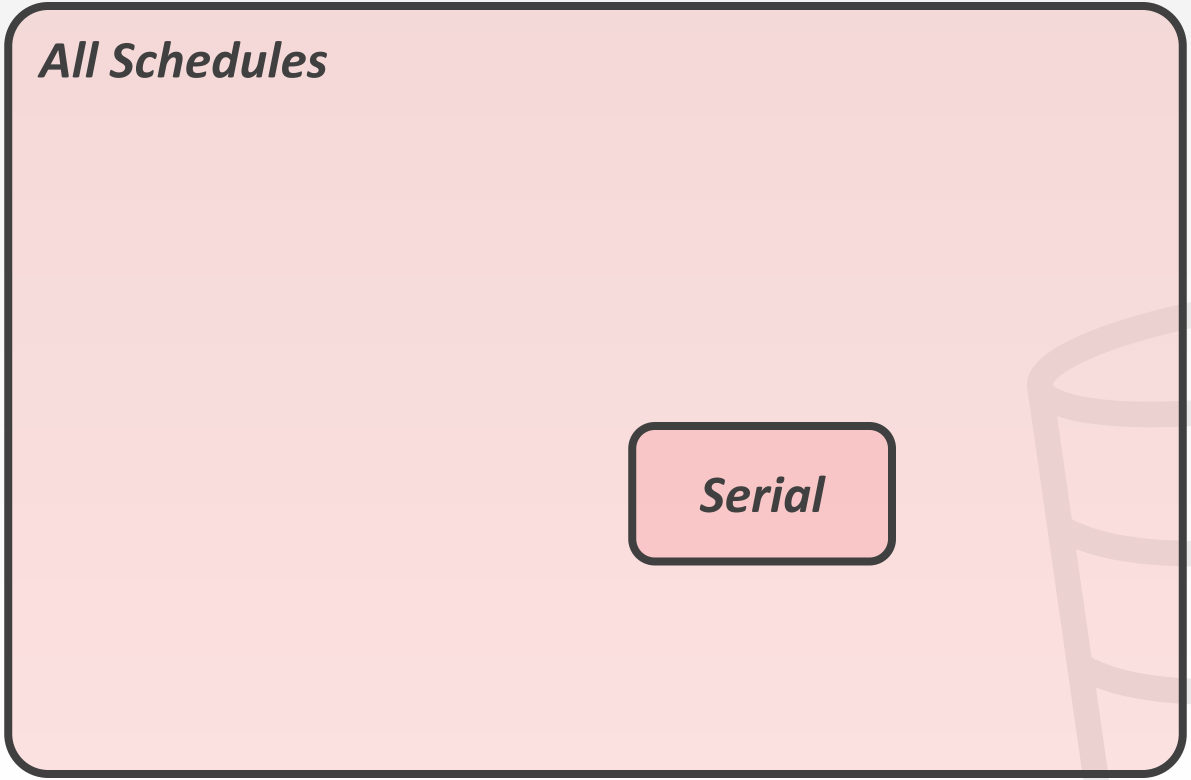
T_2 Output

A+B=2000

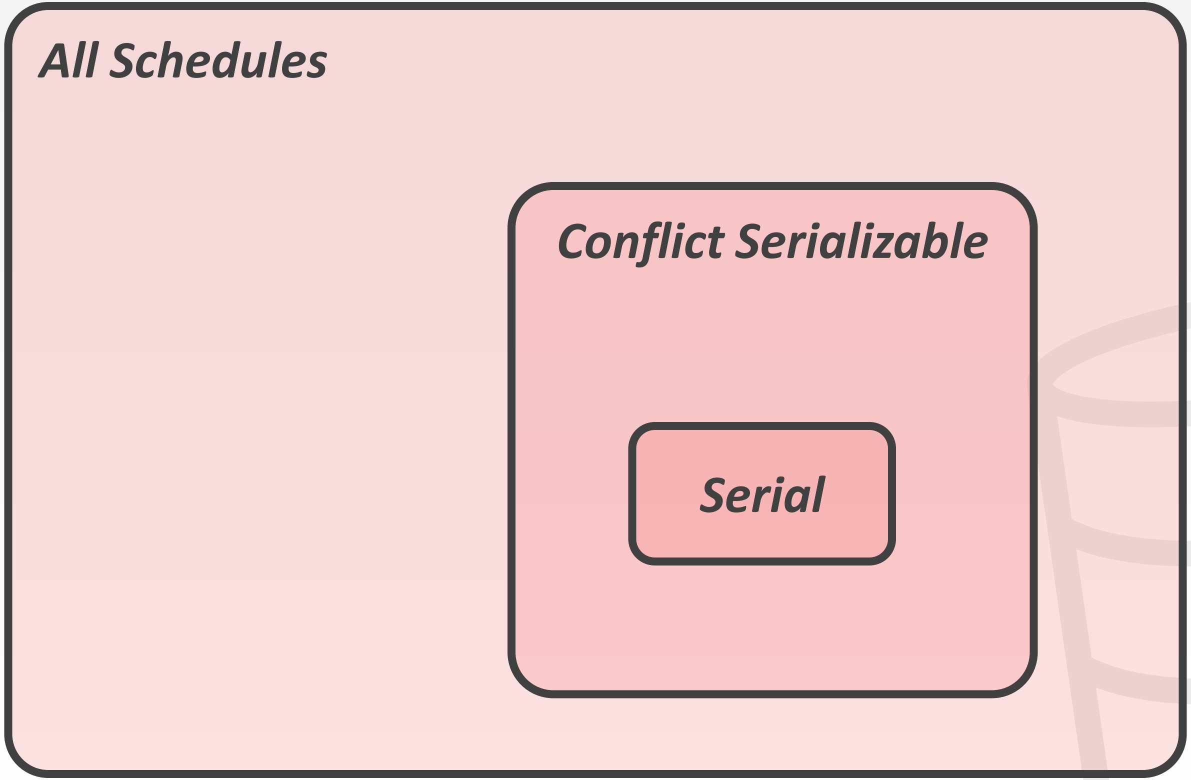
UNIVERSE OF SCHEDULES

All Schedules

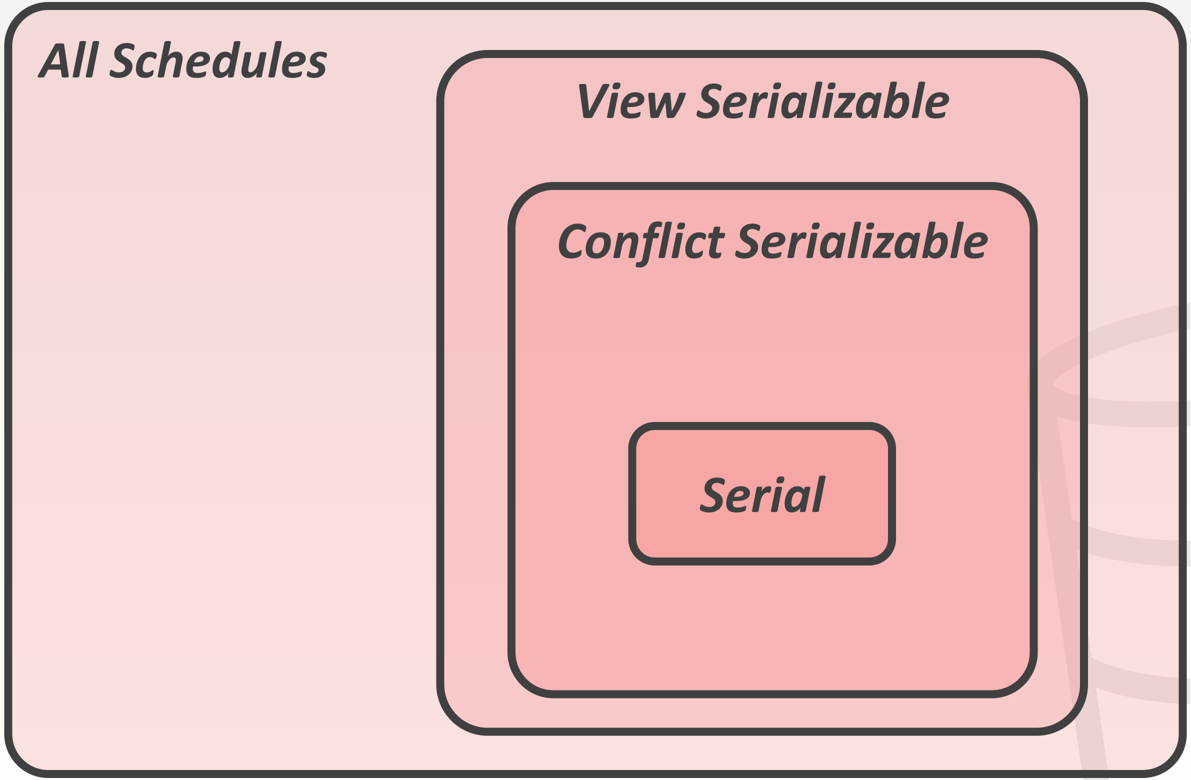
UNIVERSE OF SCHEDULES



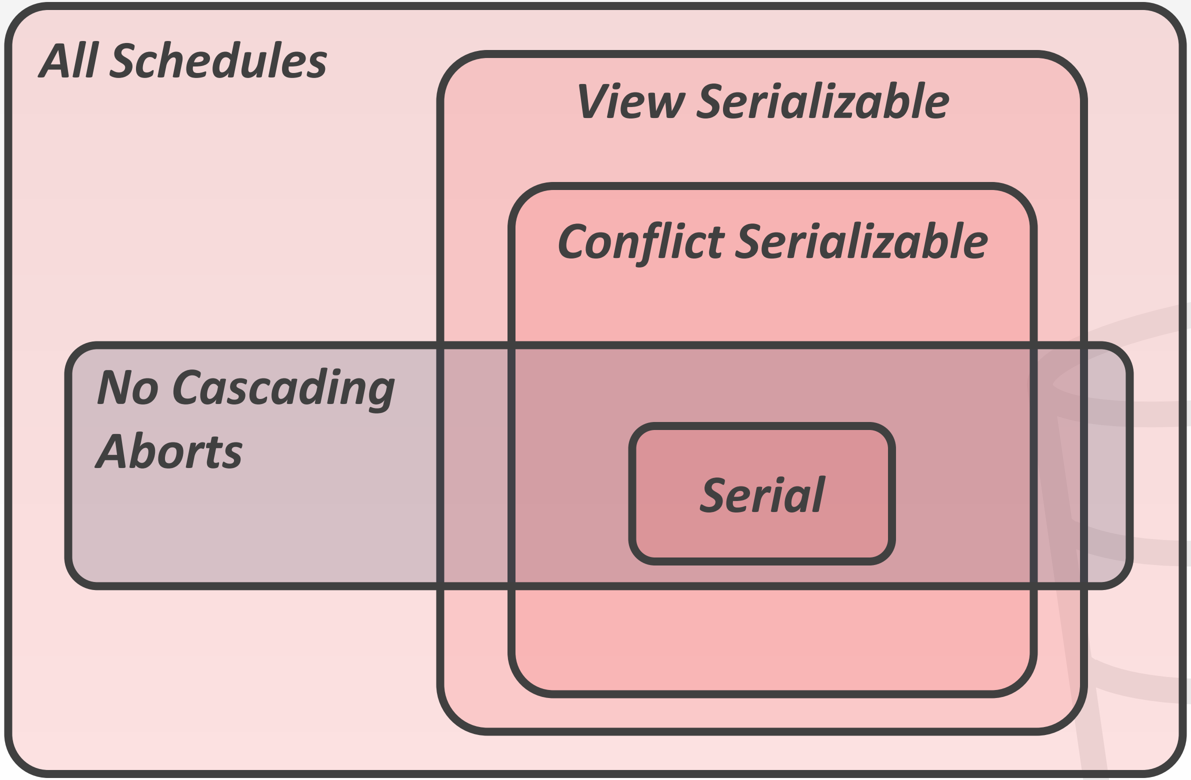
UNIVERSE OF SCHEDULES



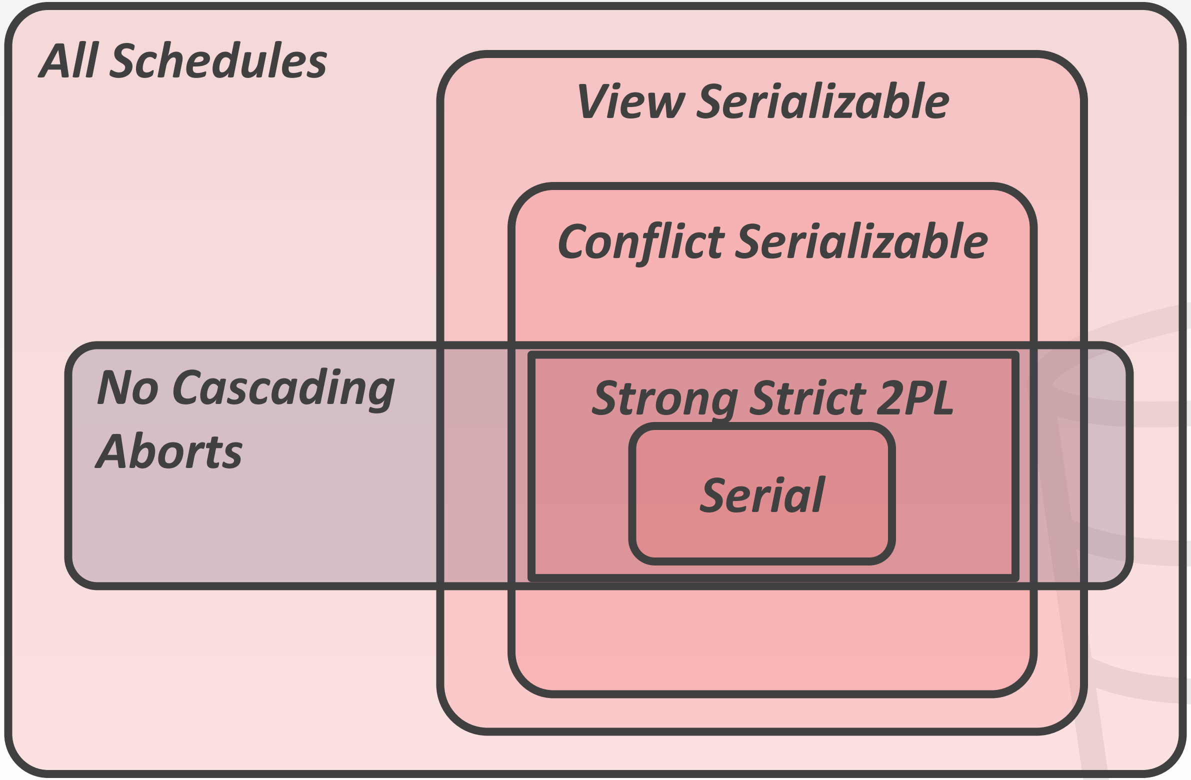
UNIVERSE OF SCHEDULES



UNIVERSE OF SCHEDULES



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**

2PL DEADLOCKS

Schedule

TIME

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

Lock Manager

2PL DEADLOCKS

Schedule

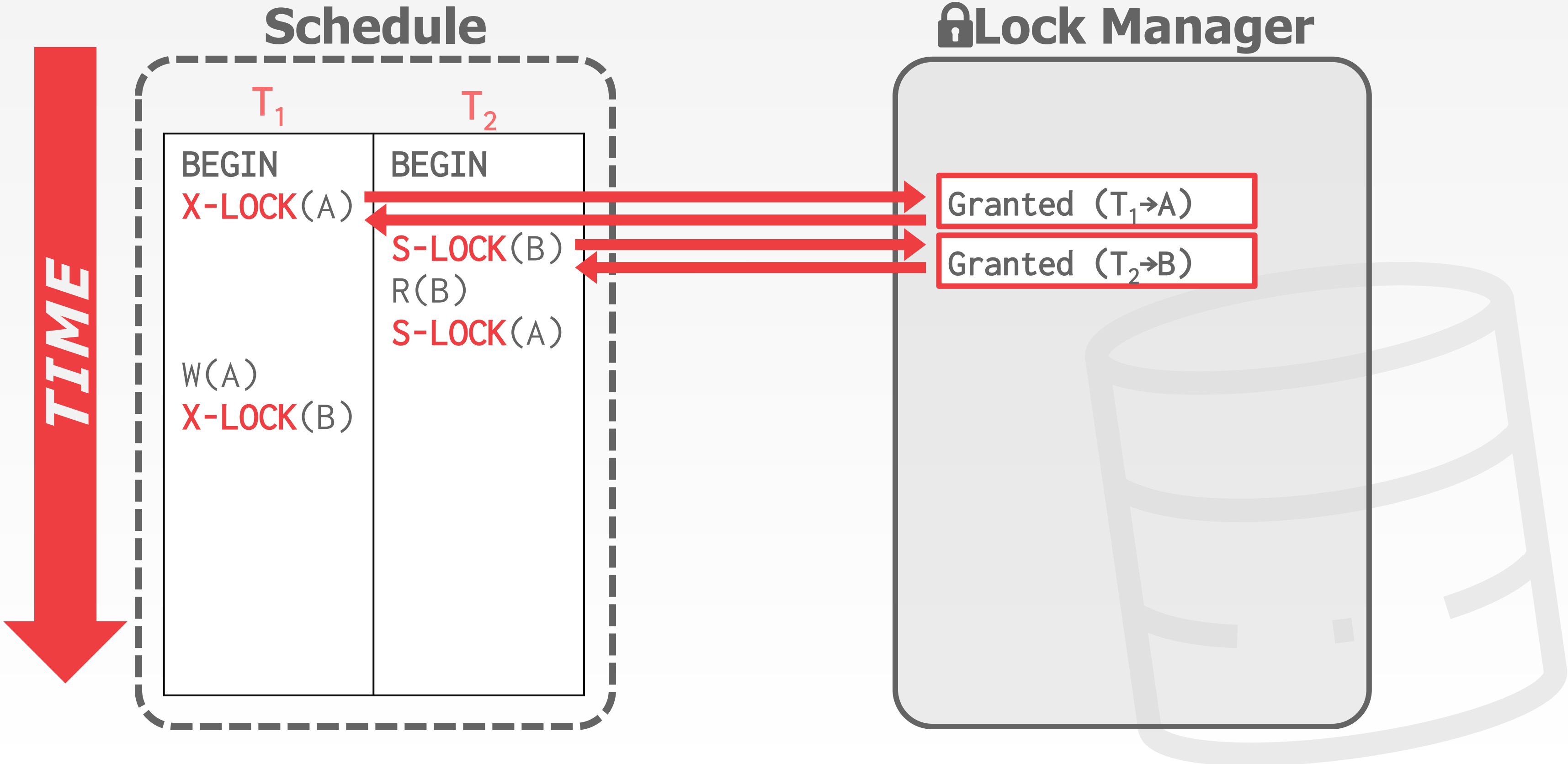
TIME

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

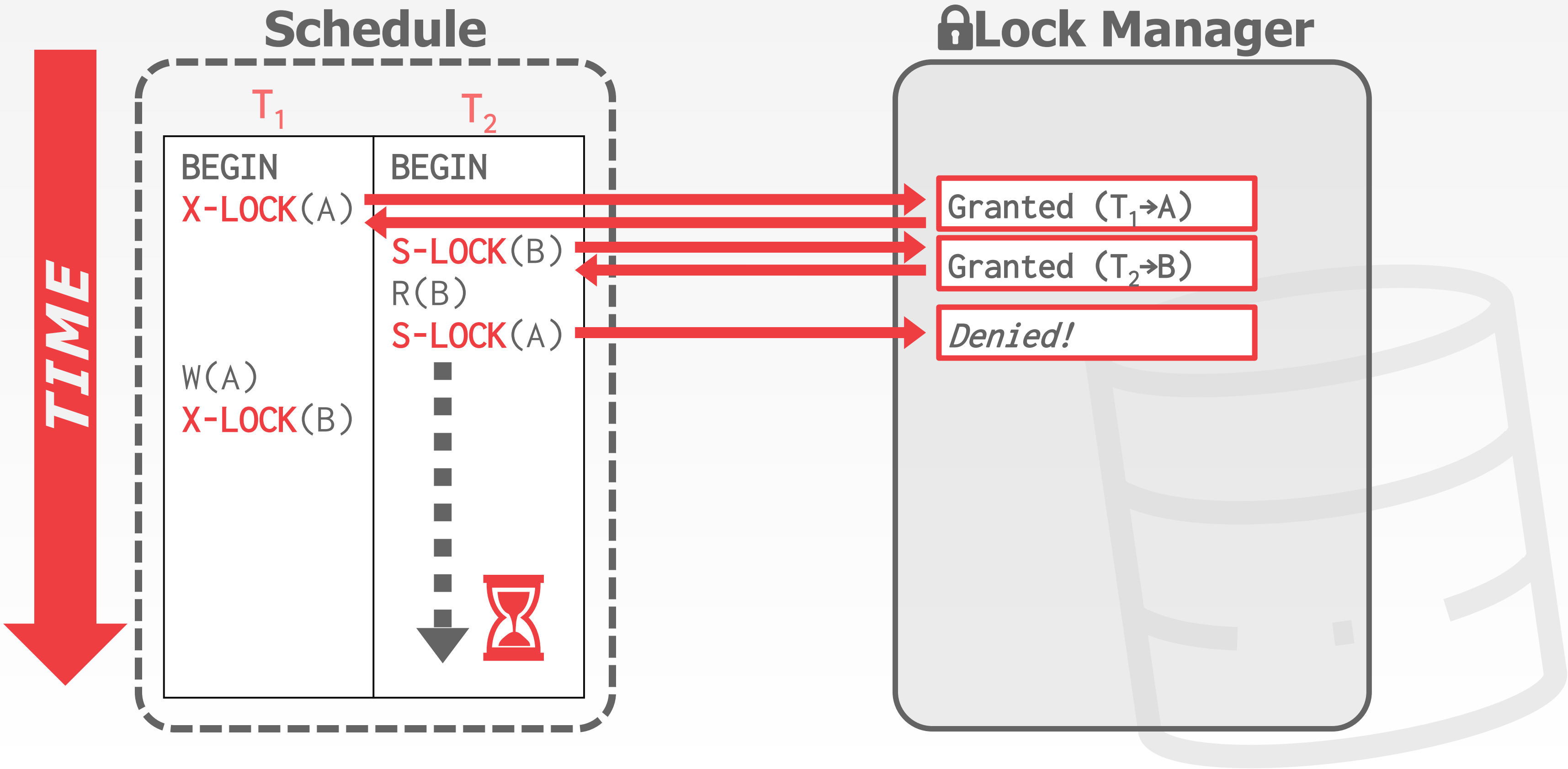
Lock Manager

Granted (T_1→A)

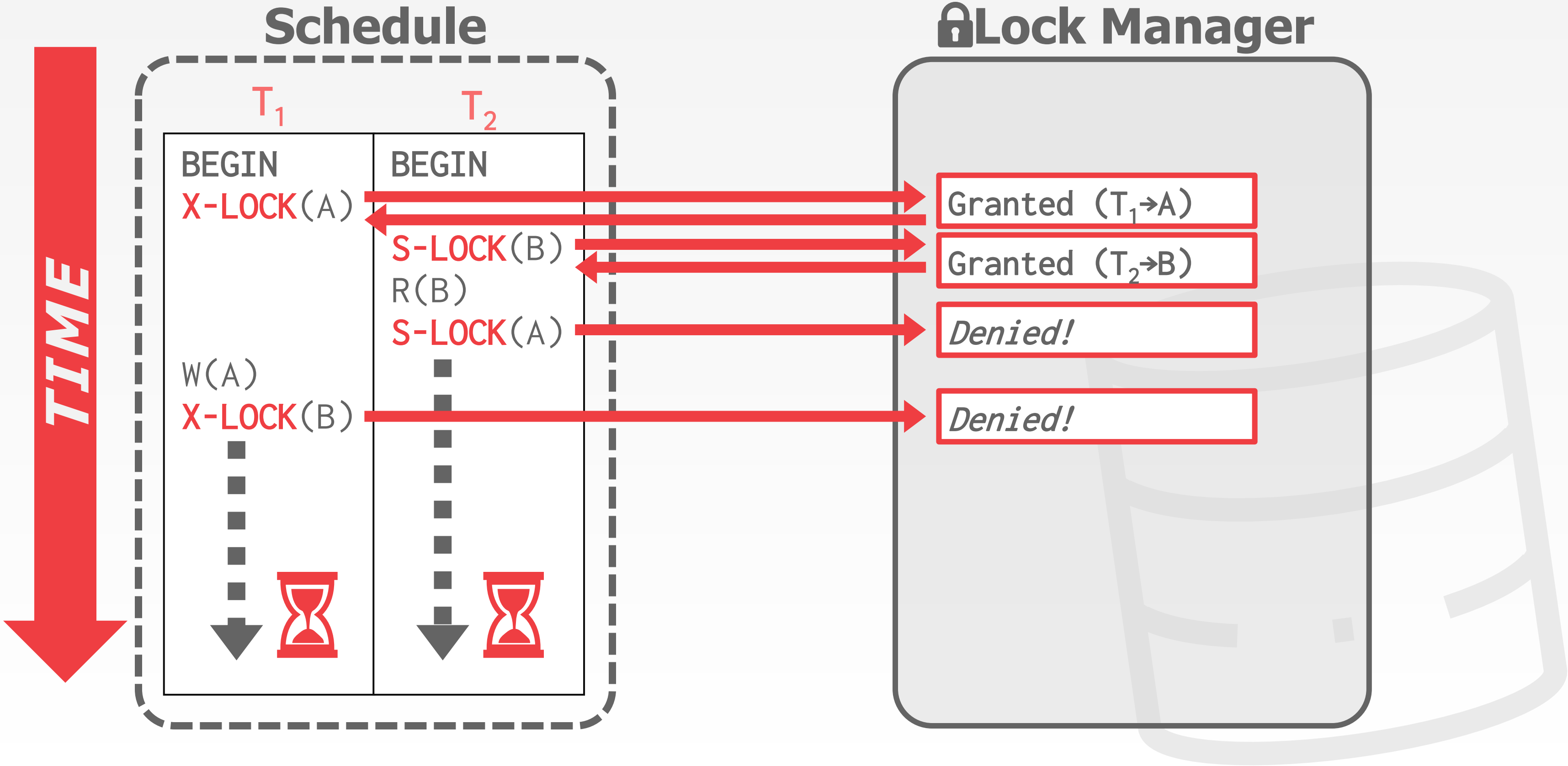
2PL DEADLOCKS



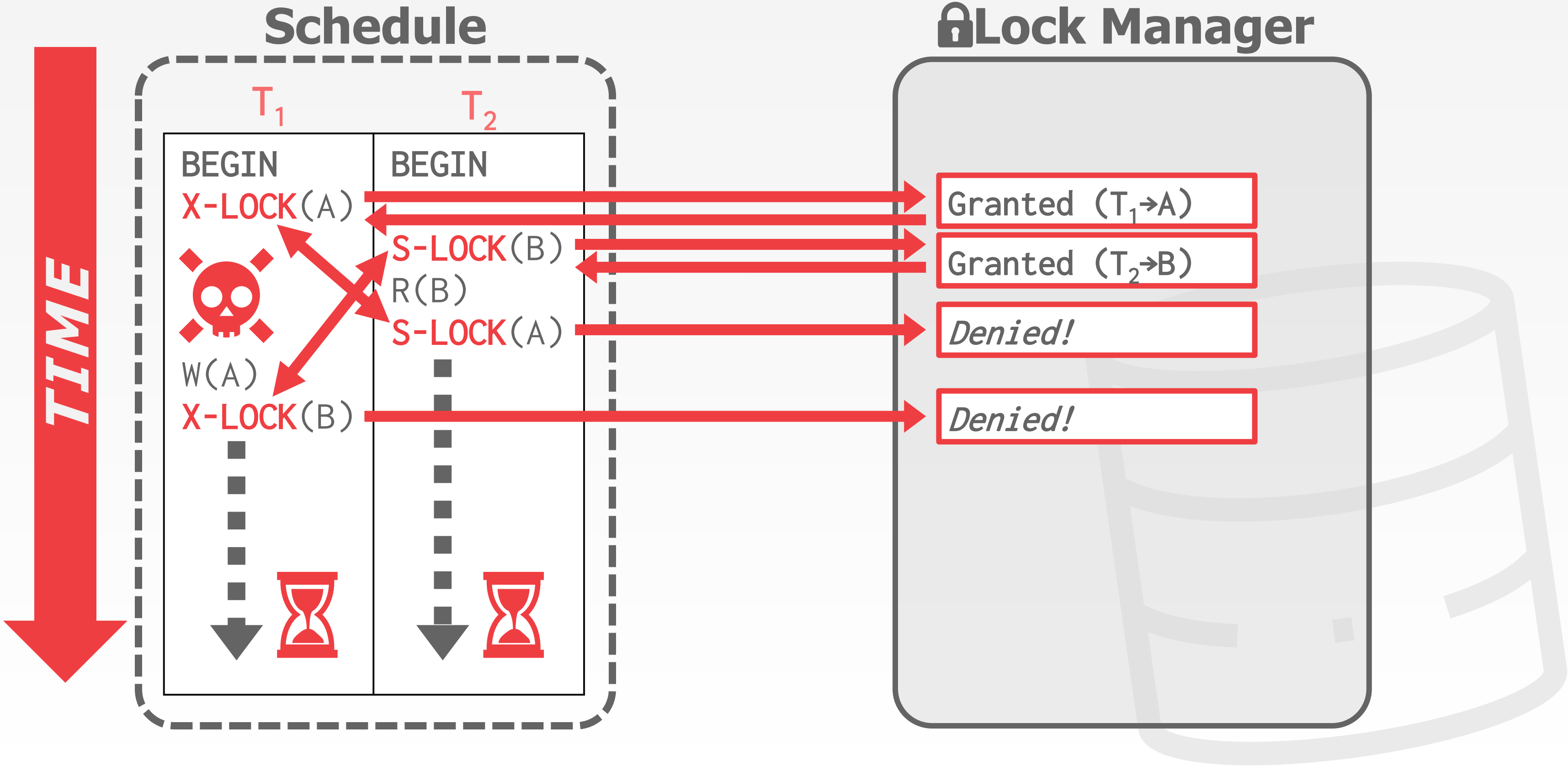
2PL DEADLOCKS



2PL DEADLOCKS



2PL DEADLOCKS



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.

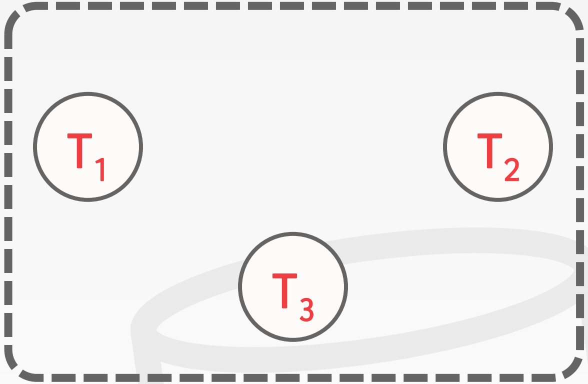
DEADLOCK DETECTION

Schedule

TIME ↓

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



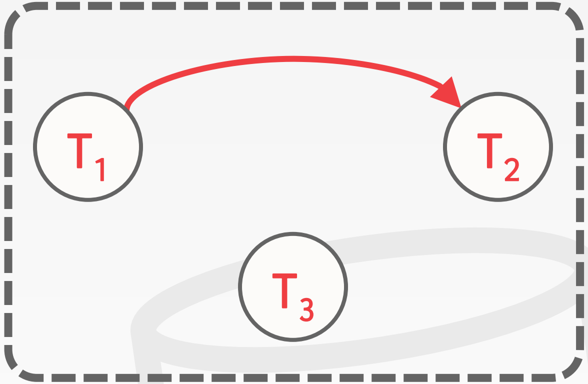
DEADLOCK DETECTION

Schedule

TIME

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



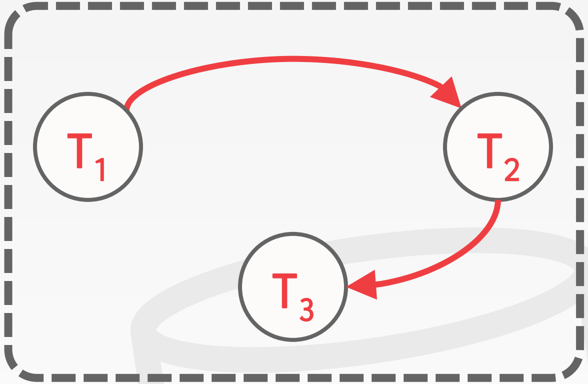
DEADLOCK DETECTION

TIME

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



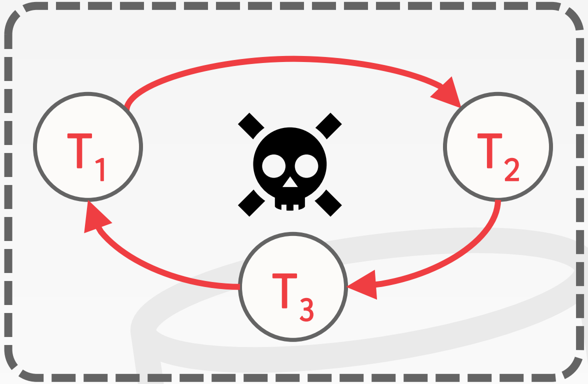
DEADLOCK DETECTION

Schedule

TIME

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 have to wait before deadlocks are broken.

DEADLOCK HANDLING: VICTIM SELECTION

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

DEADLOCK HANDLING: VICTIM SELECTION

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

→ By age (lowest timestamp)

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)

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

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

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

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

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)	⋮
⋮	

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)	⋮
⋮	

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

Wait-Die

T_1 waits

Wound-Wait

T_2 aborts

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?

When a txn restarts, what is its (new) priority?

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?

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?

WEAKER LEVELS OF ISOLATION

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency and limit performance.

We may want to use a weaker level of consistency to improve scalability.

ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:

→ Dirty Reads
→ Unrepeatable Reads
→ Phantom Reads (Unprotected Inserts/Deletes)

ISOLATION LEVELS

Isolation (High→Low)

SERIALIZABLE: No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS: Phantoms may happen.

READ COMMITTED: Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED: All of them may happen.

ISOLATION LEVELS

	Dirty Read	*Unrepeatable Read*	*Phantom*
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe

SQL-92 ISOLATION LEVELS

You set a txn's isolation level before you execute any queries in that txn.

Not all DBMS support all isolation levels in all execution scenarios
→ Replicated Environments

The default depends on implementation…

```
SET TRANSACTION ISOLATION LEVEL
  <isolation-level>;
```

```
BEGIN TRANSACTION ISOLATION LEVEL
  <isolation-level>;
```

ISOLATION LEVELS (2013)

	Default	*Maximum*
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITTED	READ COMMITTED
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: Peter Bailis

ISOLATION LEVELS (2013)

	Default	*Maximum*
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITTED	READ COMMITTED
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: Peter Bailis

ISOLATION LEVELS (2013)

	Default	*Maximum*
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITTED	READ COMMITTED
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: Peter Bailis

SQL-92 ACCESS MODES

```
SET TRANSACTION <access-mode>;
```

```
BEGIN TRANSACTION <access-mode>;
```

You can provide hints to the DBMS about whether a txn will modify the database during its lifetime.

Only two possible modes:

→ READ WRITE (Default)

→ READ ONLY

Not all DBMSs will optimize execution if you set a txn to in READ ONLY mode.

CONCLUSION

2PL is used in almost DBMS.

Automatically generates correct interleaving:

→ Locks + protocol (2PL, SS2PL ...)

→ Deadlock detection + handling

→ Deadlock prevention

Default DBMS isolation levels are usually weaker than serializable

NEXT CLASS

Logging