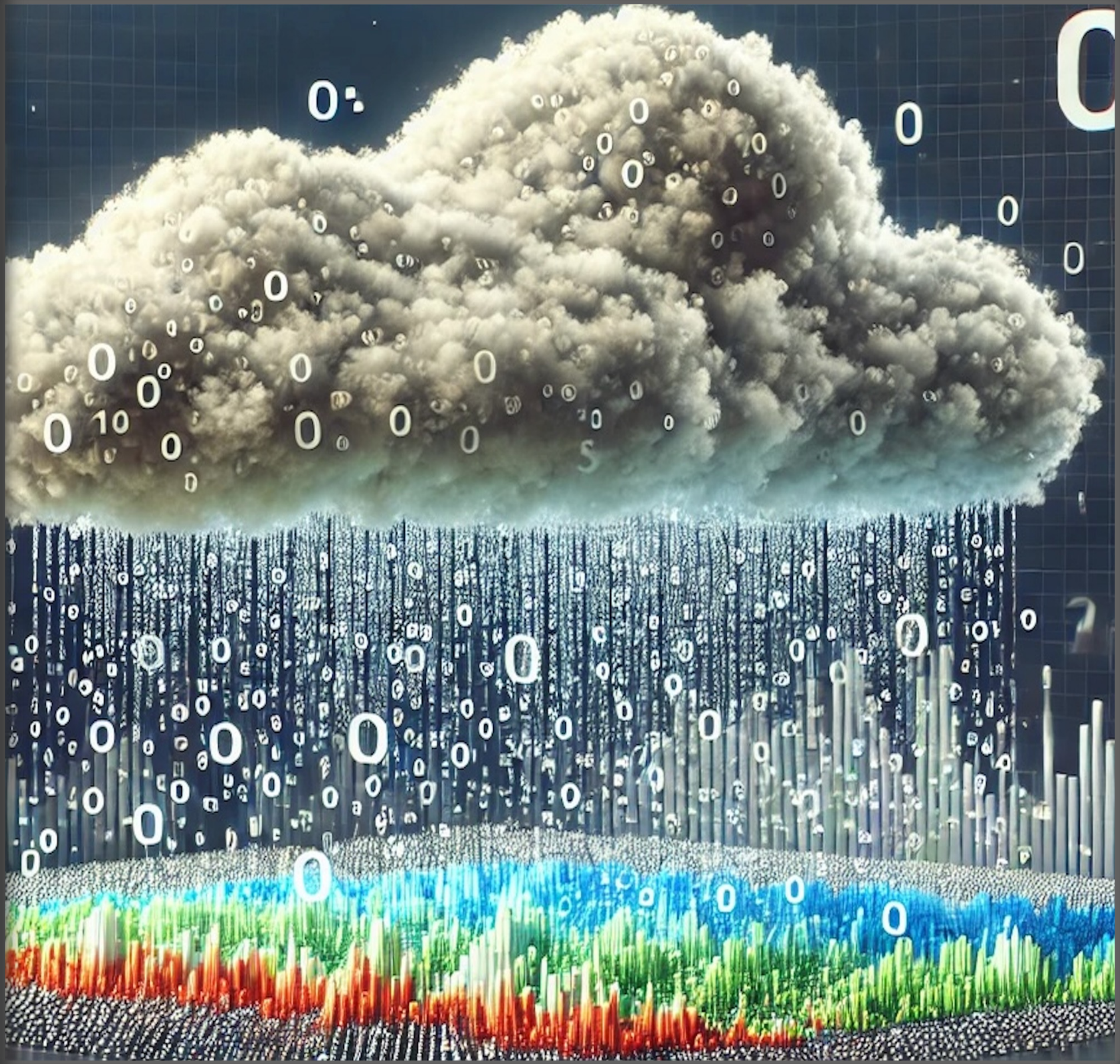
Carnegie Mellon University

Advanced Database Systems (15-721)

Lecture #08

Granularities of Locks and Degrees of Consistency

Fall 2024 » Prof. Jignesh Patel



Announcements

- Talk from Oracle on Tuesday, October 1, @ noon in 6501 GHC.
  - Unifying relational and document/JSON management.

- Exam: Oct 9th in **GHC 8102 between 1-4 pm**. Open book.
  - Start anytime. Stop 90 minutes later.
  - Let me know if you have a conflict by the end of next week 9/20.

Revisit your introduction to DB class material if needed.

Transaction Management

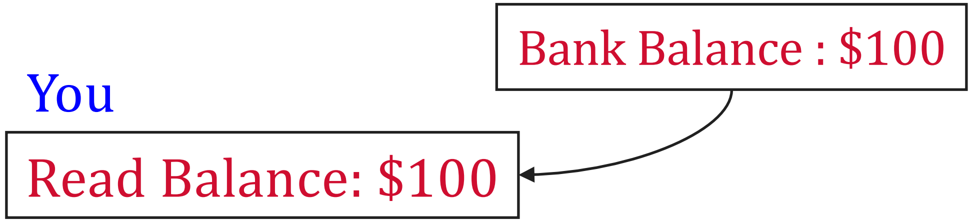
```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```

Bank Balance : $100

Revisit your introduction to DB class material if needed.

Transaction Management

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



You

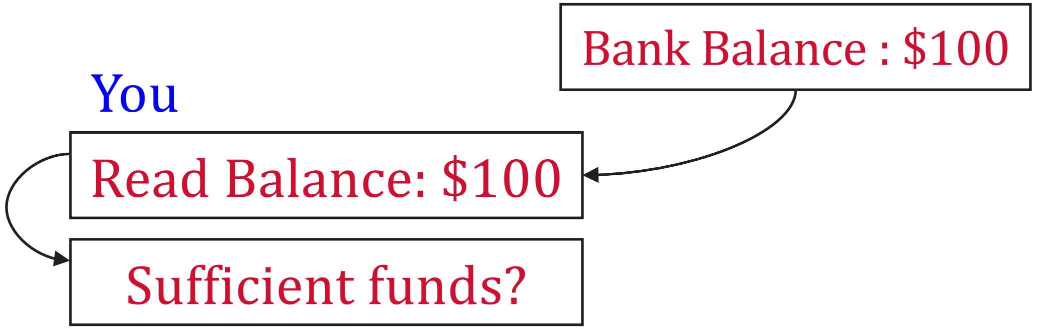
Bank Balance : $100

Read Balance: $100

Revisit your introduction to DB class material if needed.

Transaction Management

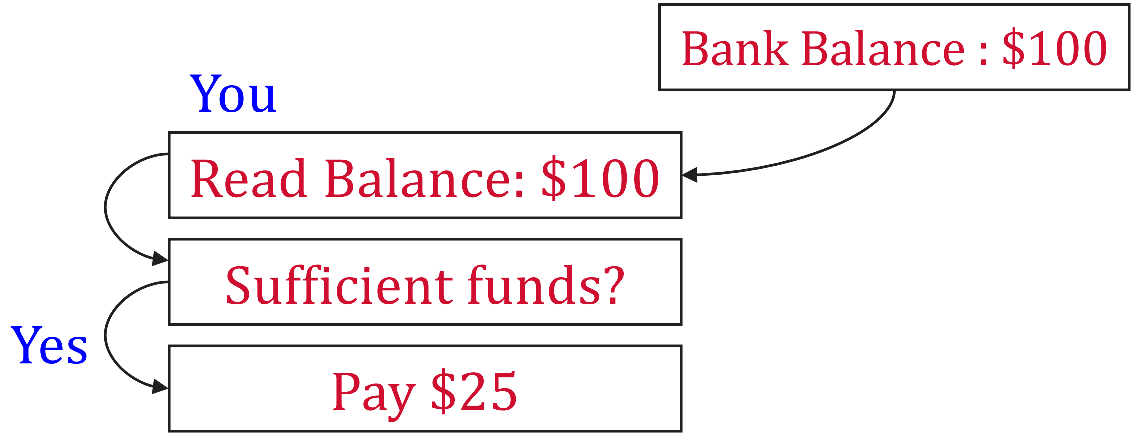
```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Revisit your introduction to DB class material if needed.

Transaction Management

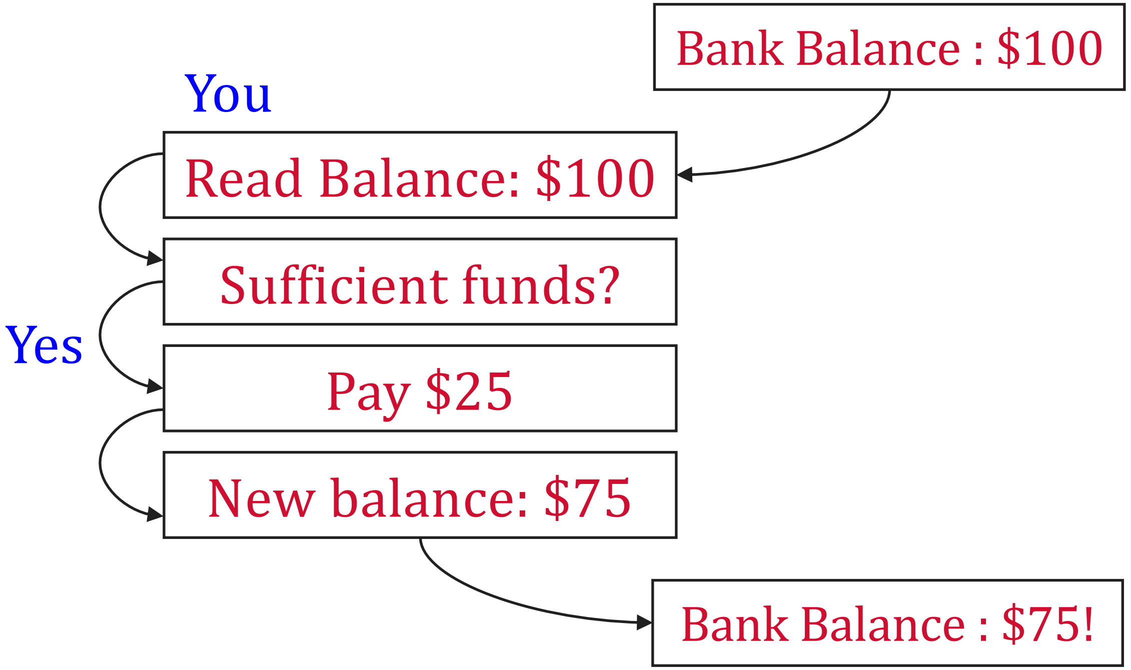
```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Revisit your introduction to DB class material if needed.

TRANSACTION MANAGEMENT

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Bank Balance : $100

You

Read Balance: $100

Sufficient funds?

Yes

Pay $25

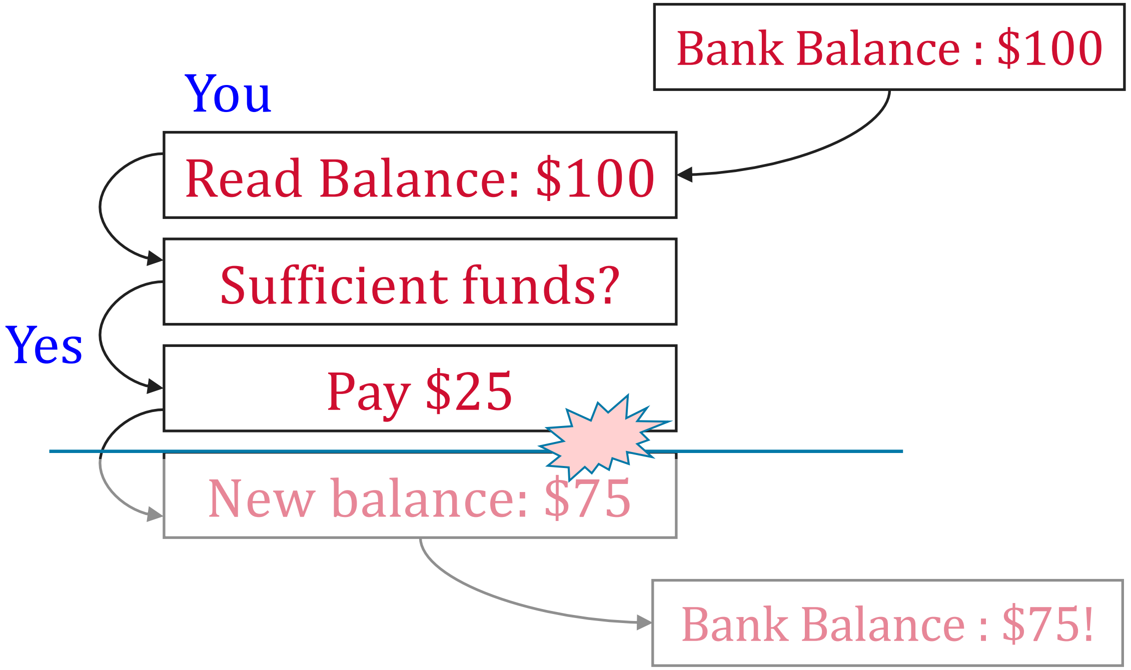
New balance: $75

Bank Balance : $75!

Revisit your introduction to DB class material if needed.

Transaction Management

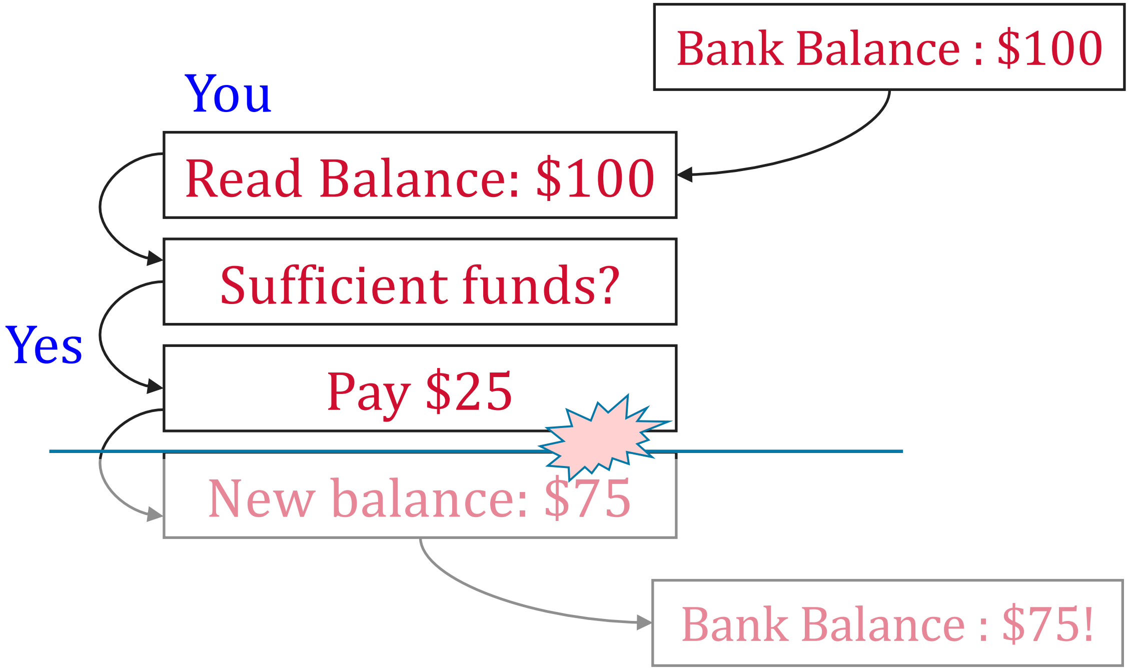
```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Revisit your introduction to DB class material if needed.

TRANSACTION MANAGEMENT

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Bank Balance : $100

You

Read Balance: $100

Sufficient funds?

Yes

Pay $25

New balance: $75

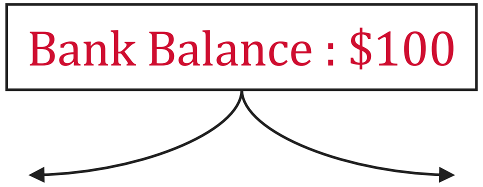
Bank Balance : $75!

Revisit your introduction to DB class material if needed.

Transaction Management

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```

You



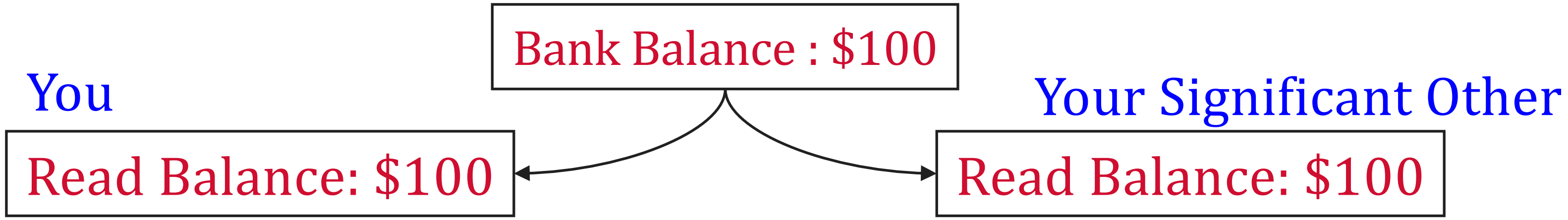
Bank Balance : $100

Your Significant Other

Revisit your introduction to DB class material if needed.

Transaction Management

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



You

Bank Balance : $100

Your Significant Other

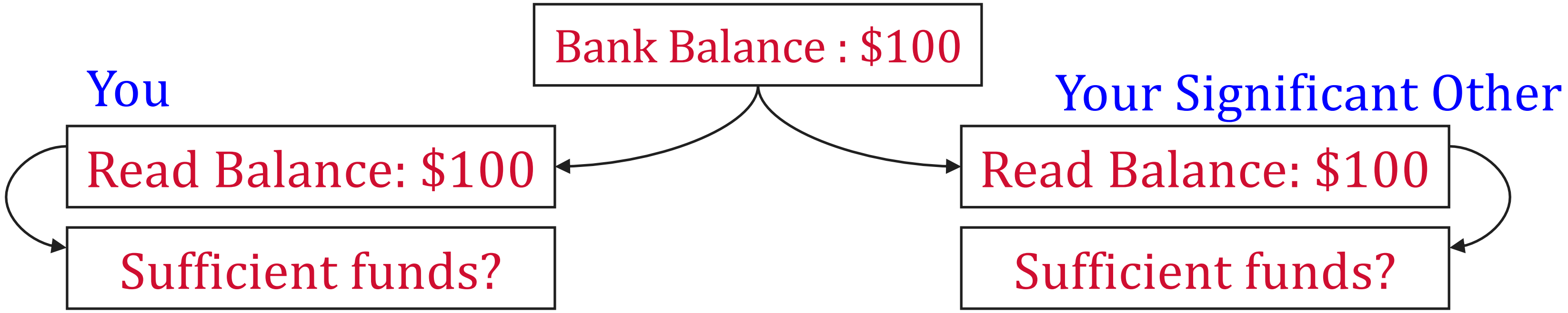
Read Balance: $100

Read Balance: $100

Revisit your introduction to DB class material if needed.

Transaction Management

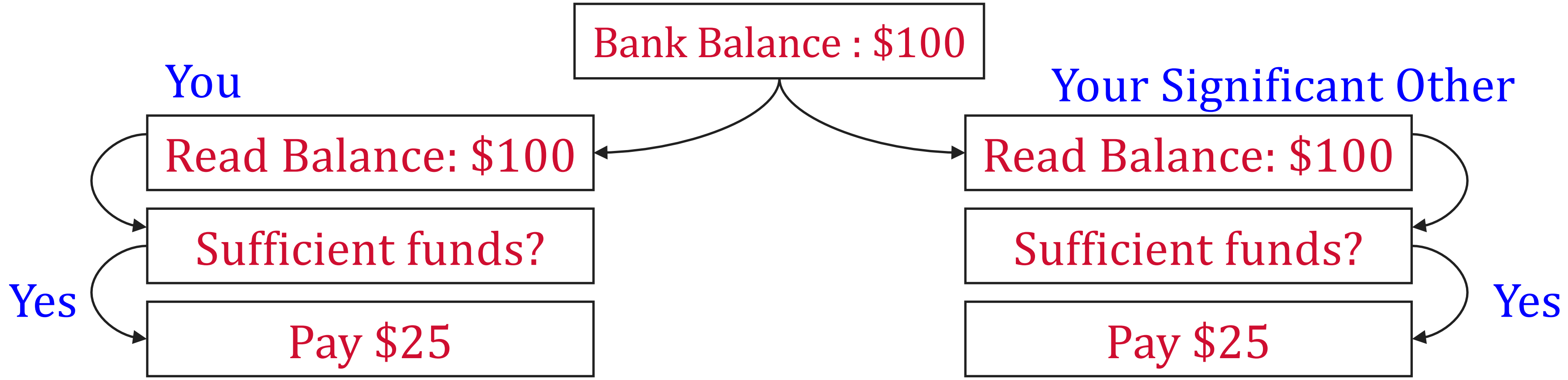
```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Revisit your introduction to DB class material if needed.

Transaction Management

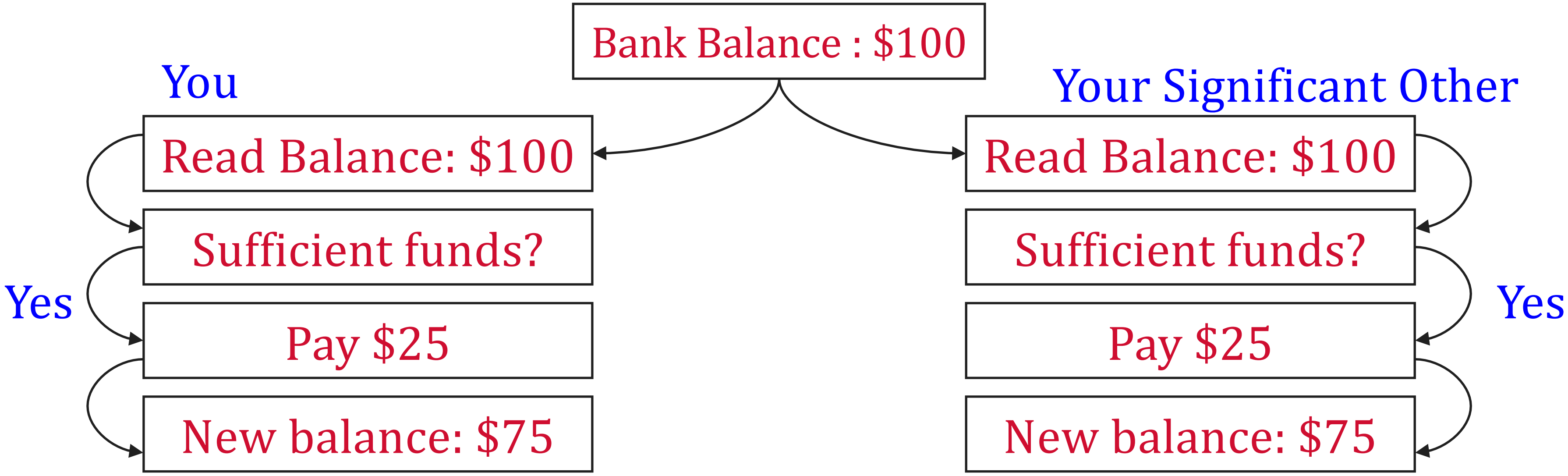
```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Revisit your introduction to DB class material if needed.

TRANSACTION MANAGEMENT

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Bank Balance : $100

You

Read Balance: $100

Sufficient funds?

Yes

Pay $25

New balance: $75

Your Significant Other

Read Balance: $100

Sufficient funds?

Yes

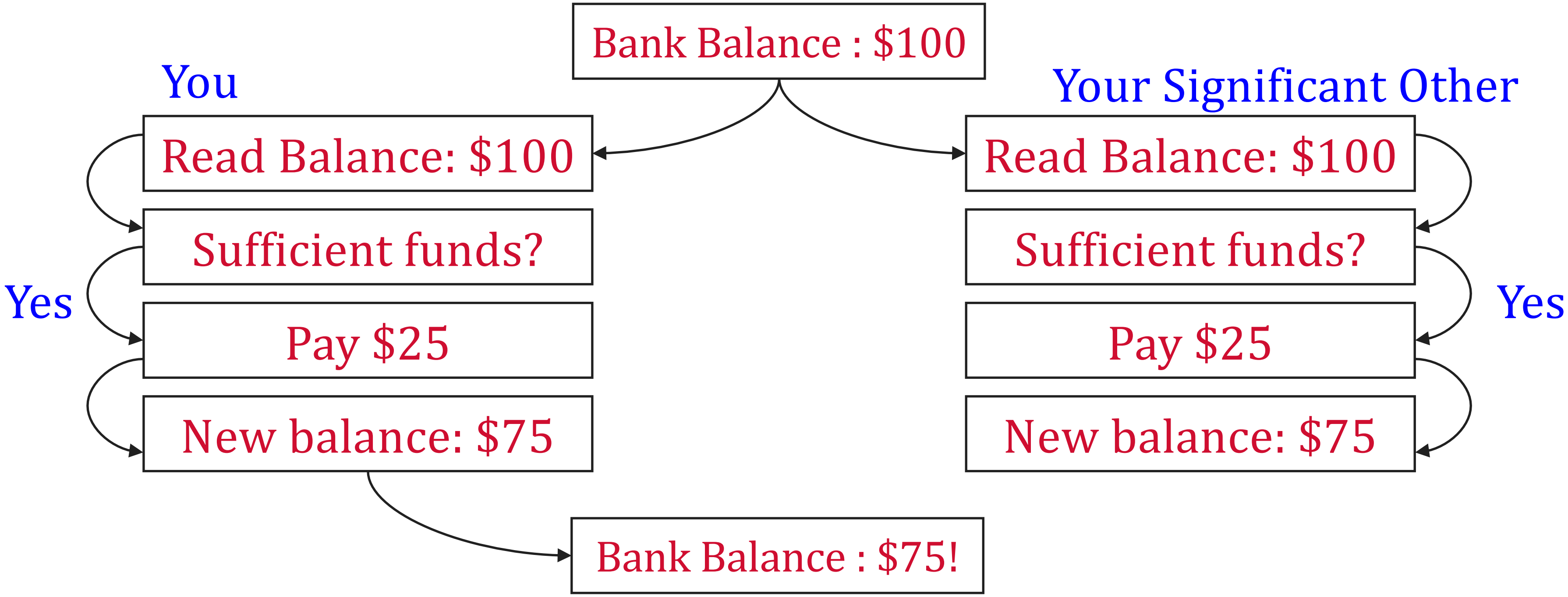
Pay $25

New balance: $75

Revisit your introduction to DB class material if needed.

Transaction Management

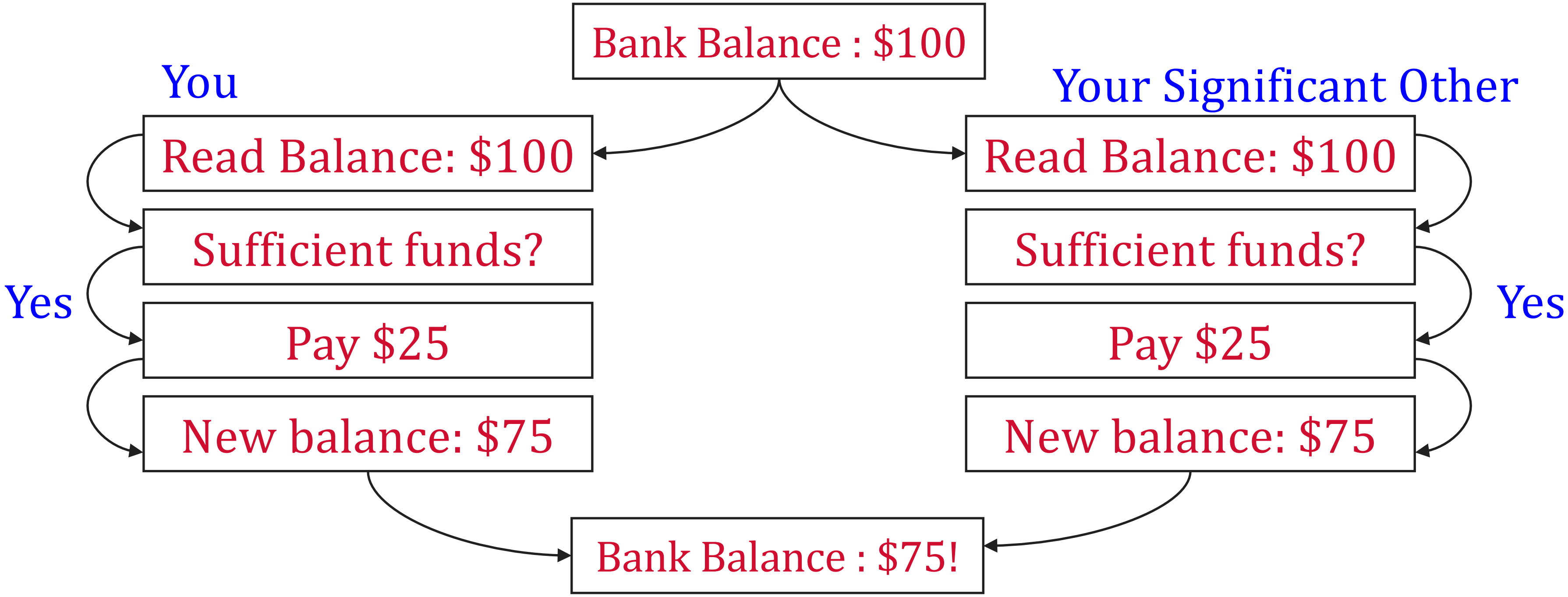
```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Revisit your introduction to DB class material if needed.

Transaction Management

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Revisit your introduction to DB class material if needed.

Transaction Management

Redo/Undo mechanism

Atomicity All actions in txn happen, or none happen.
"All or nothing…"

Integrity Constraints

Consistency If each txn is consistent and the DB starts consistent, then it ends up consistent.
"It looks correct to me…"

Key constraints, CHECKS, TRIGGERS, … hold before and after the txn completes.

Concurrency Control

Isolation Execution of one txn is isolated from that of other txns.
"All by myself…"

Redo/Undo mechanism

Durability If a txn commits, its effects persist.
"I will survive…"

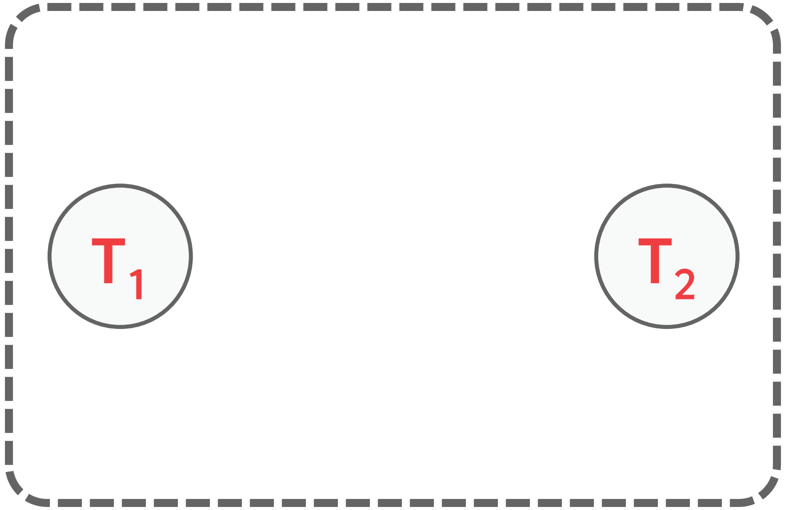
Revisit your introduction to DB class material if needed.

CONCURRENCY CONTROL AND DEPENDENCE GRAPHS

Schedule

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

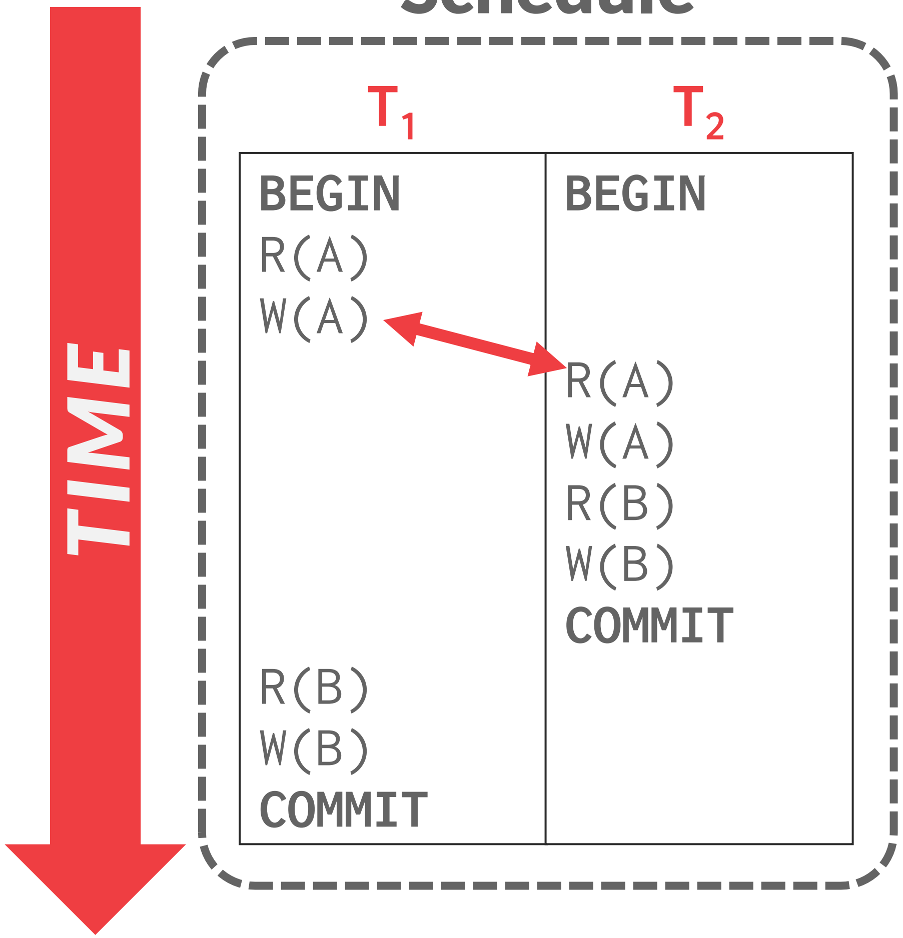
Dependency Graph



Revisit your introduction to DB class material if needed.

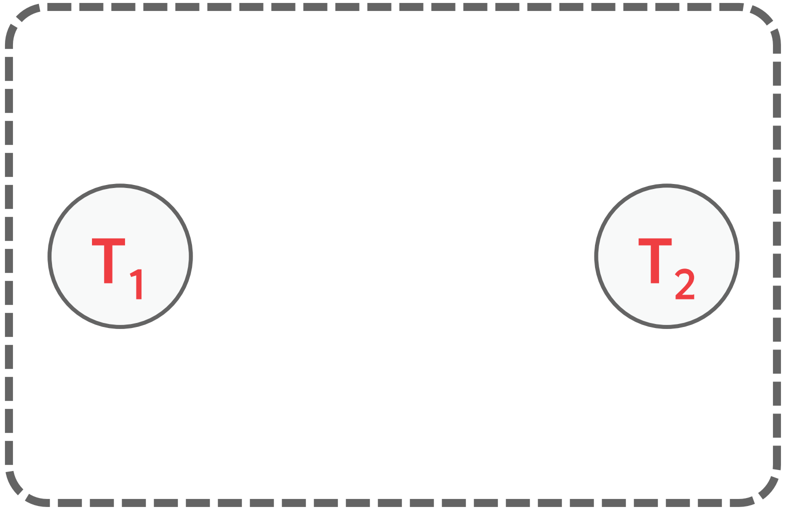
CONCURRENCY CONTROL AND DEPENDENCE GRAPHS

Schedule



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

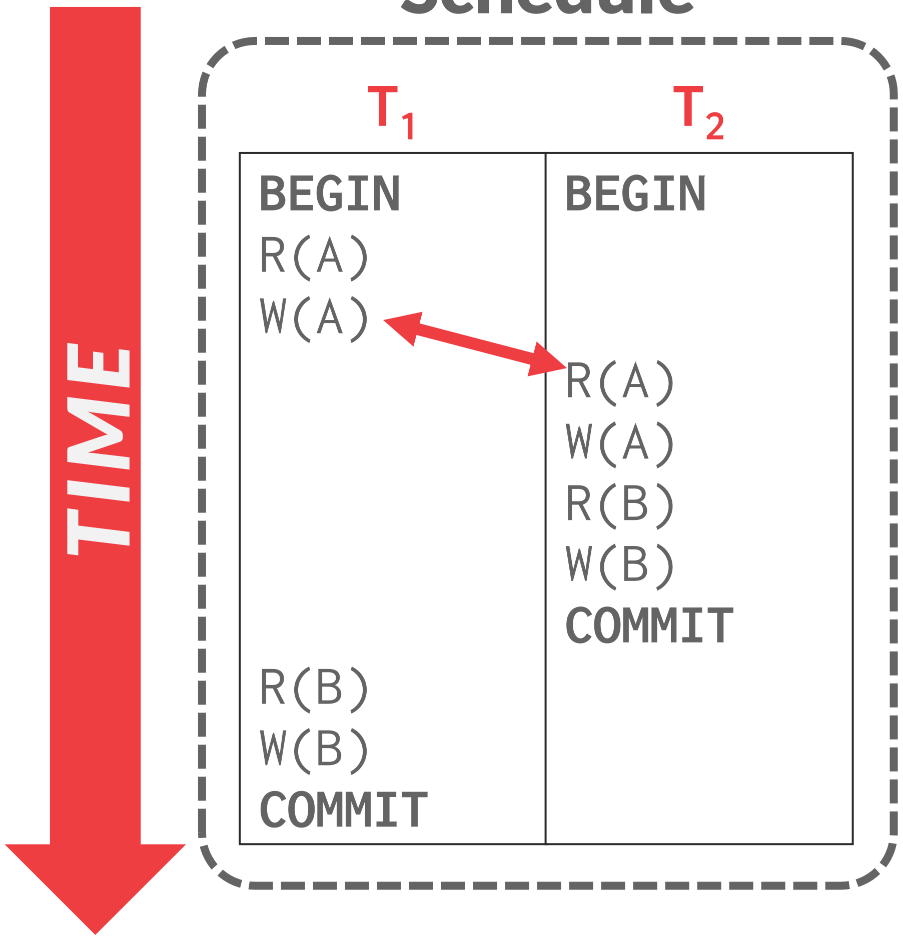
Dependency Graph



Revisit your introduction to DB class material if needed.

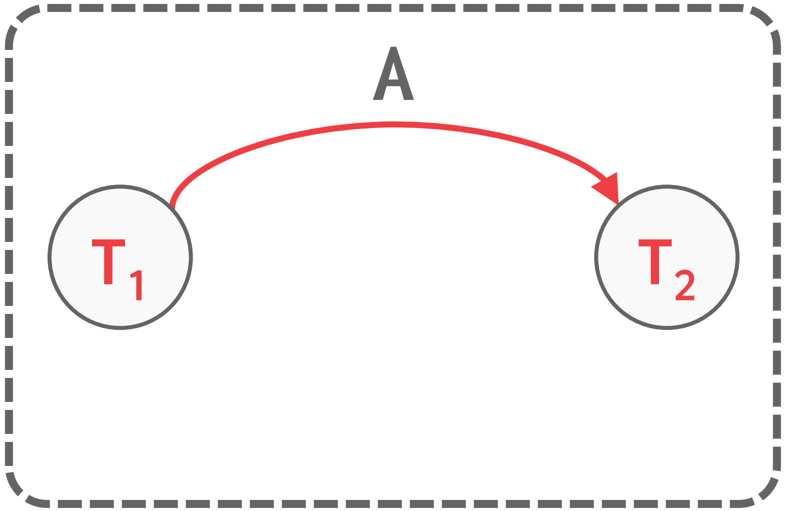
CONCURRENCY CONTROL AND DEPENDENCE GRAPHS

Schedule



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

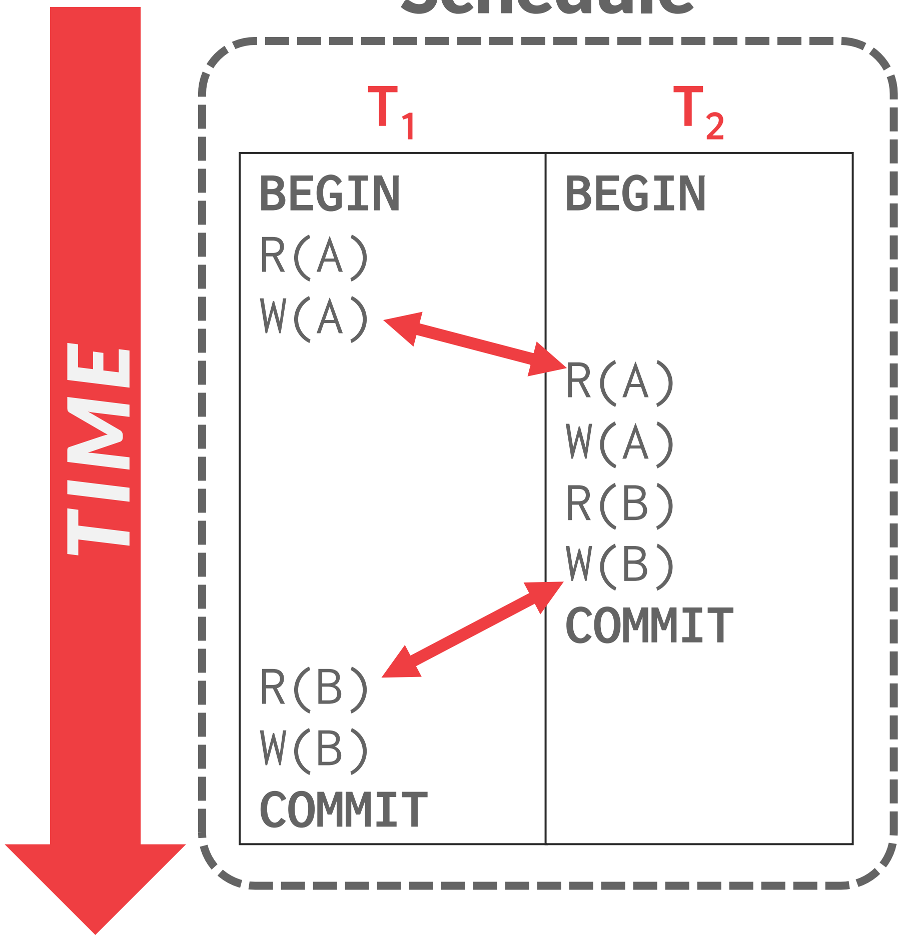
Dependency Graph



Revisit your introduction to DB class material if needed.

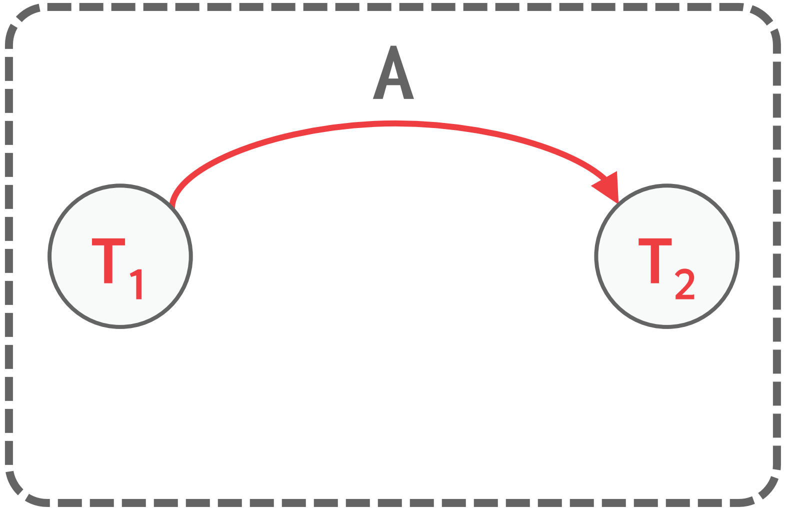
CONCURRENCY CONTROL AND DEPENDENCE GRAPHS

Schedule



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

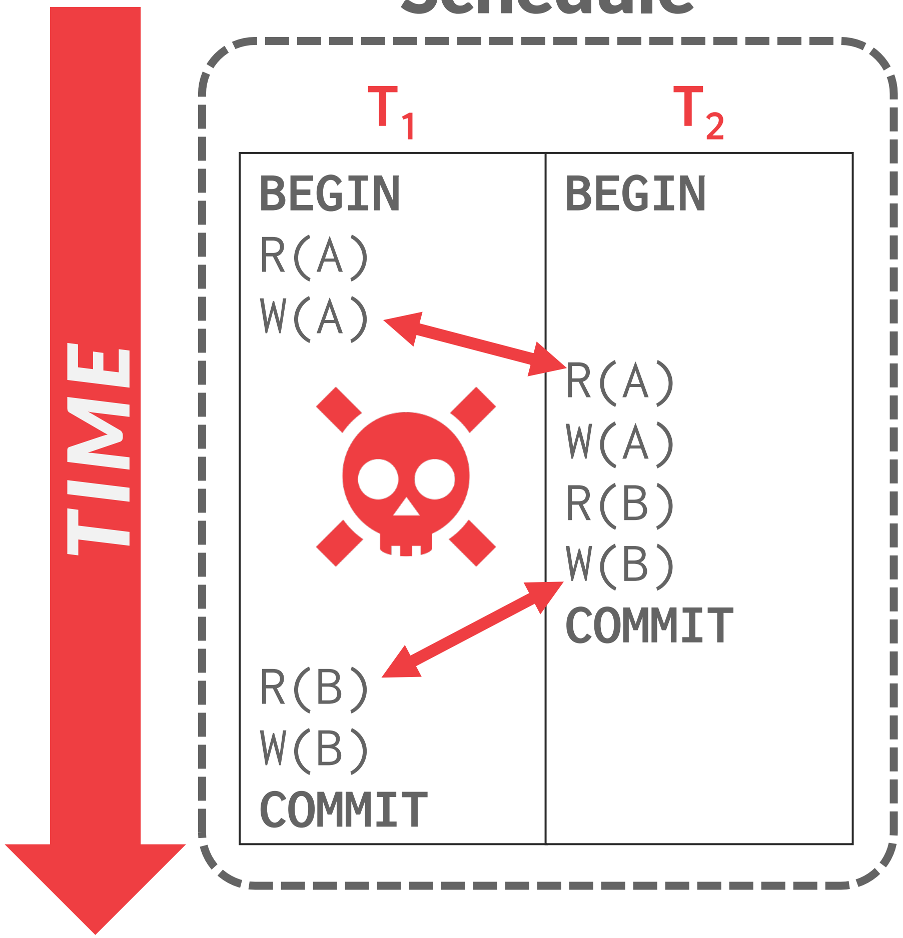
Dependency Graph



Revisit your introduction to DB class material if needed.

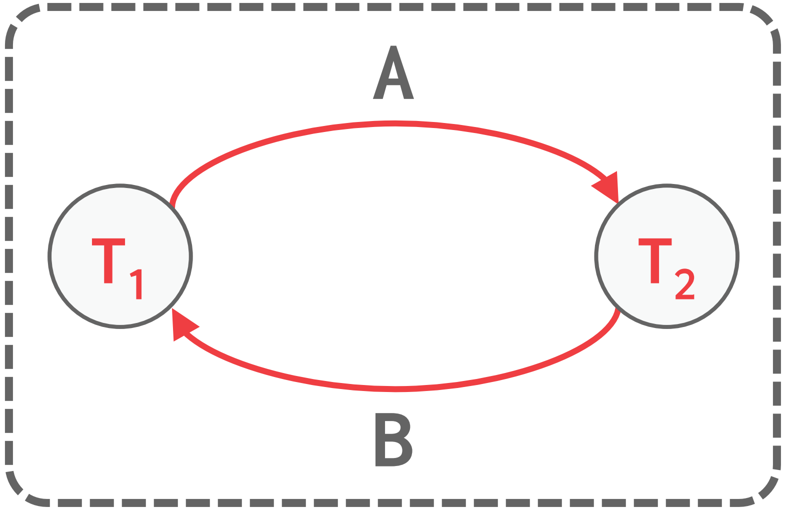
CONCURRENCY CONTROL AND DEPENDENCE GRAPHS

Schedule



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

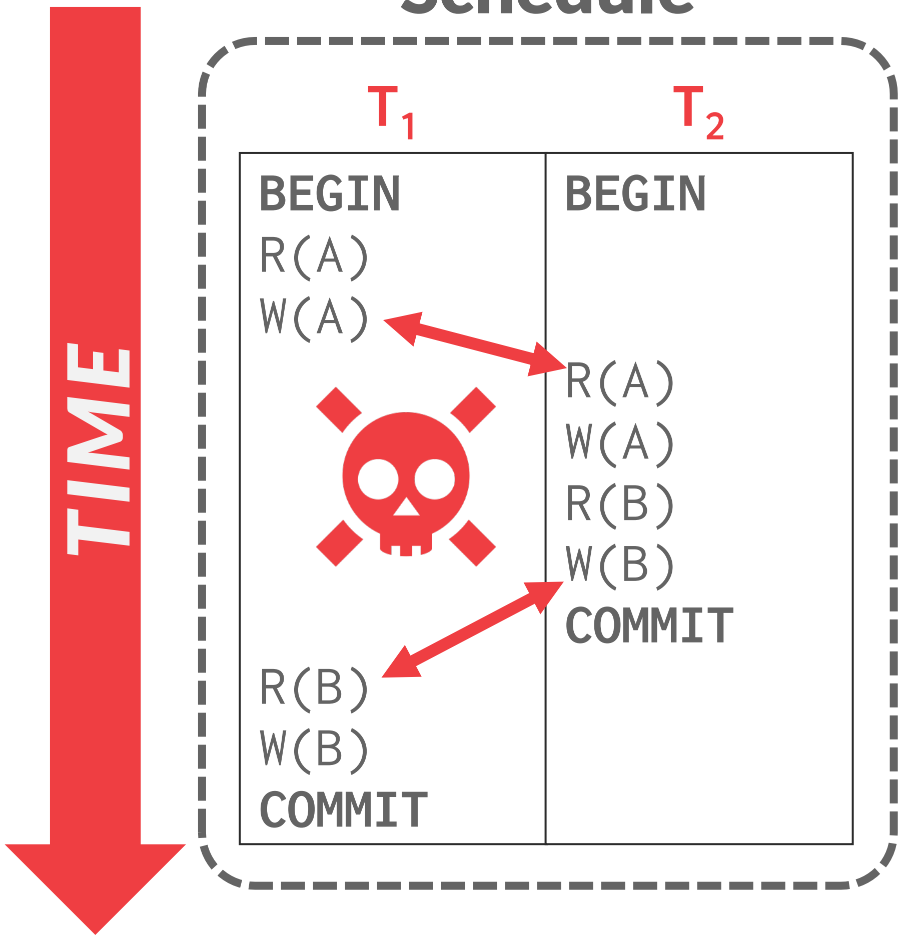
Dependency Graph



Revisit your introduction to DB class material if needed.

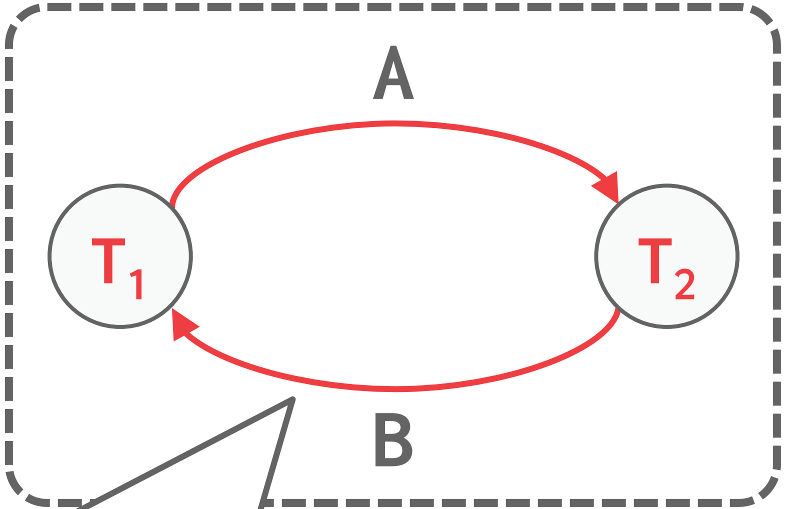
CONCURRENCY CONTROL AND DEPENDENCE GRAPHS

Schedule



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

Dependency Graph



The cycle in the graph reveals the problem. The output of T_1 depends on T_2, and vice-versa.

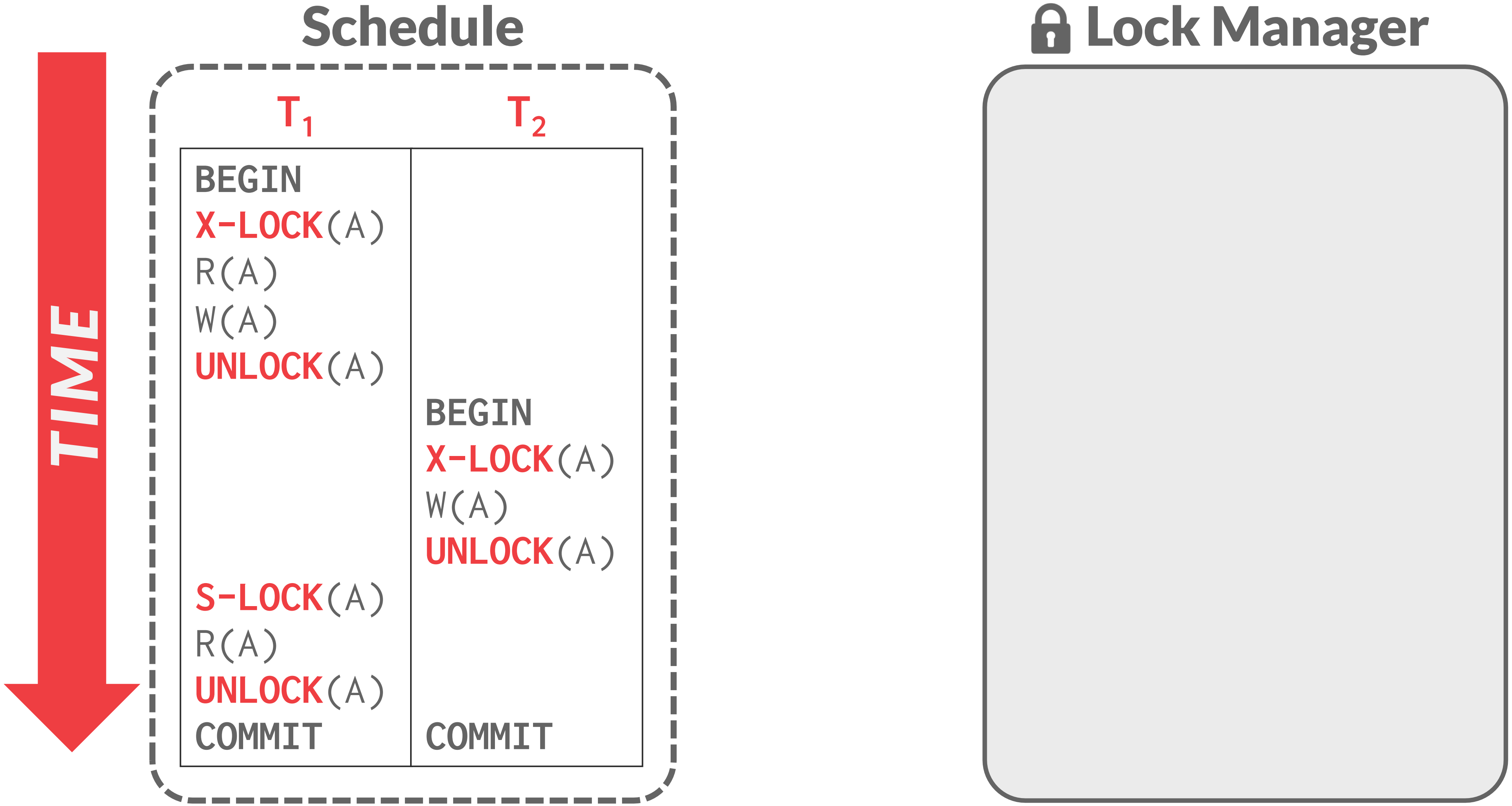
Revisit your introduction to DB class material if needed.

SERIALIZABLE SCHEDULE

- A schedule that is equivalent to some serial execution of the transactions.
- Need to reason about conflicting operations.
- Two operations conflict if:
  - They are by different transactions,
  - They are on the same object and one of them is a write.
- Interleaved Execution Anomalies
  - Read-Write Conflicts (**R-W**). Also called **Unrepeatable Read**.
  - Write-Read Conflicts (**W-R**). Also called **Dirty Read**.
  - Write-Write Conflicts (**W-W**). Also called **Lost Update**.

Revisit your introduction to DB class material if needed.

Locking with S and X Locks at the Row-level

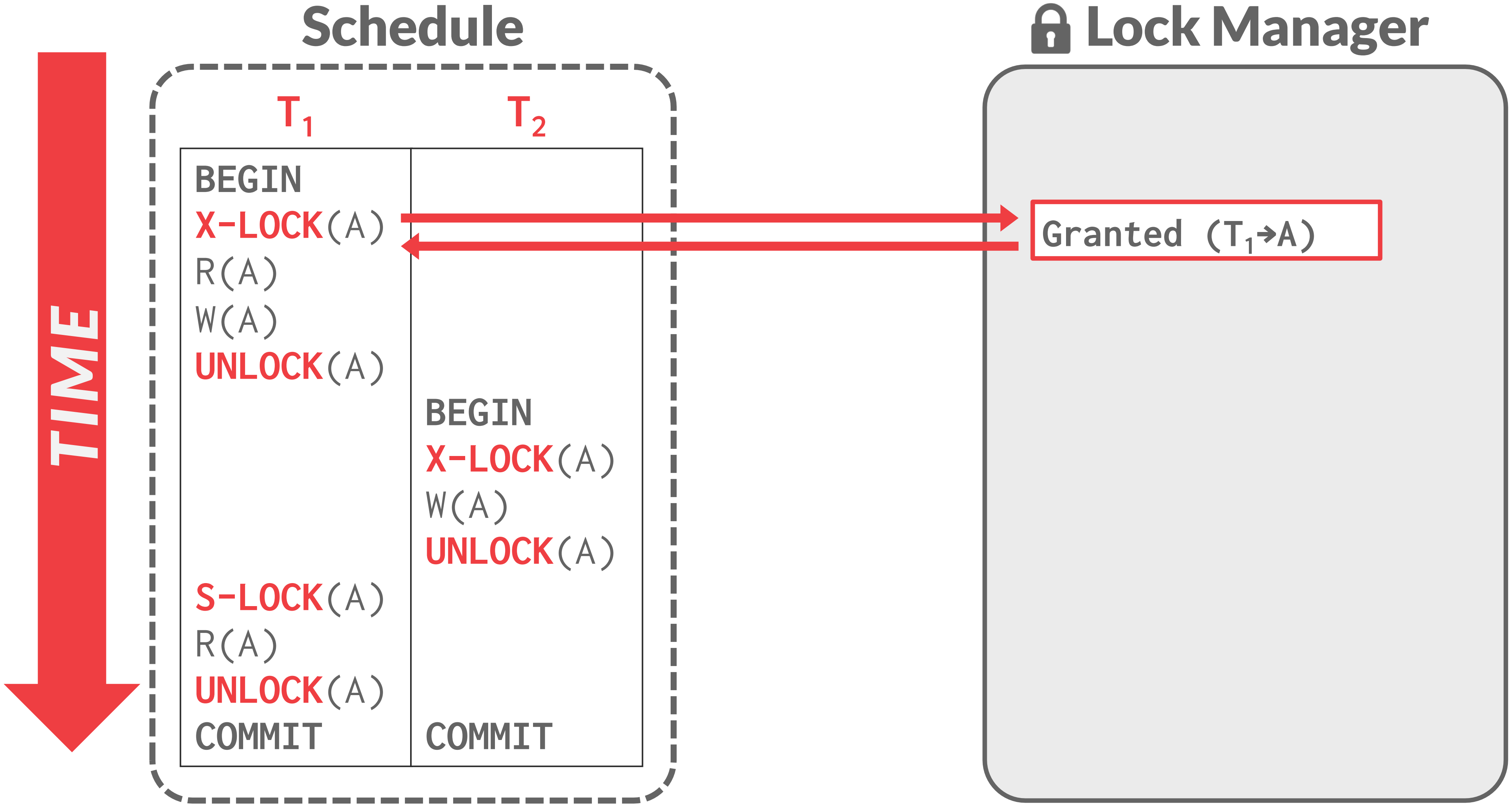


Schedule

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

Revisit your introduction to DB class material if needed.

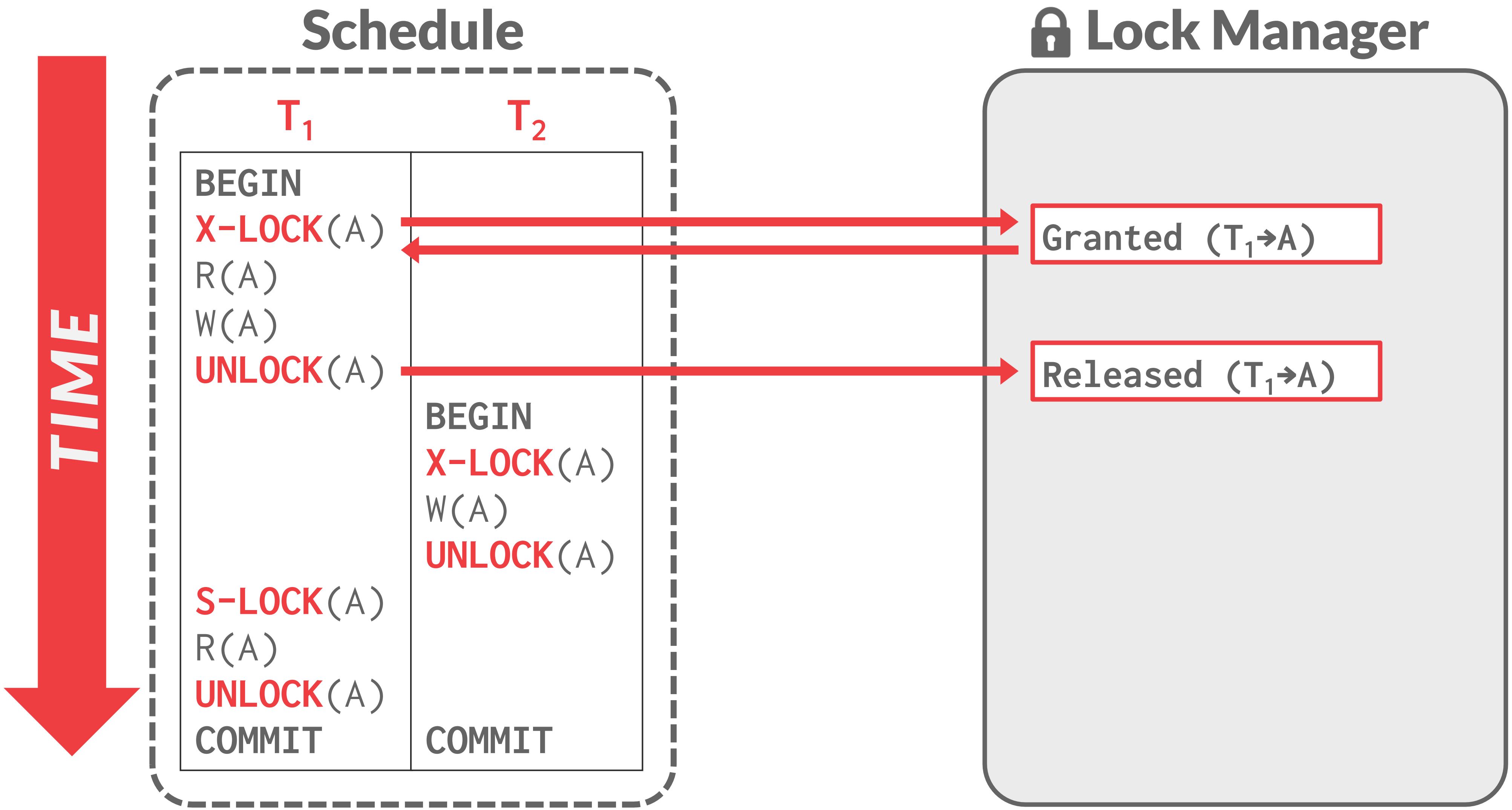
Locking with S and X Locks at the Row-level



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

Revisit your introduction to DB class material if needed.

LOCKING WITH S AND X LOCKS AT THE ROW-LEVEL



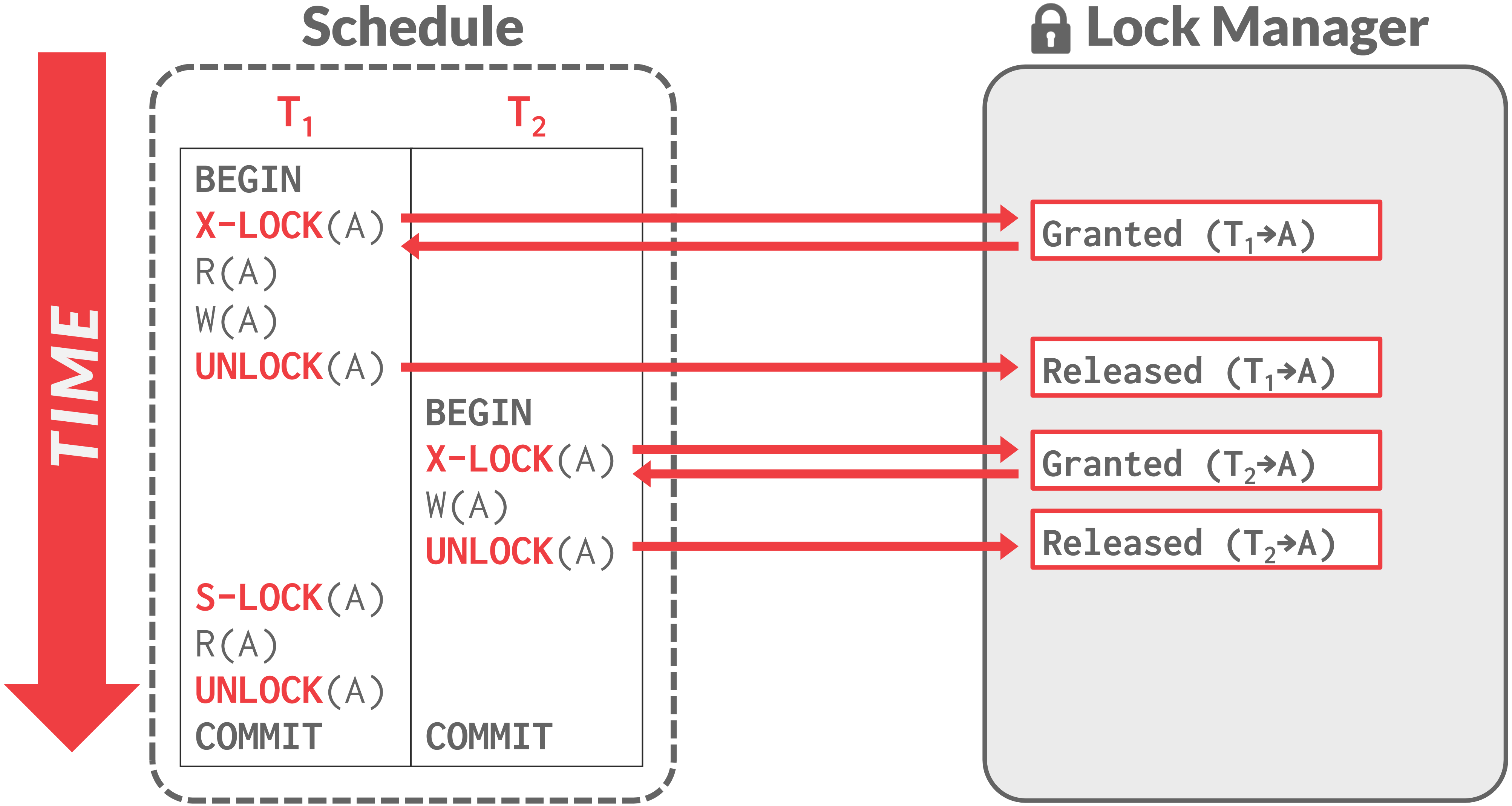
T_1	T_2
BEGIN	
X-LOCK(A)	
R(A)	
W(A)	
UNLOCK(A)	
	BEGIN
	X-LOCK(A)
	W(A)
	UNLOCK(A)
S-LOCK(A)	
R(A)	
UNLOCK(A)	
COMMIT	COMMIT

Granted (T_1→A)

Released (T_1→A)

Revisit your introduction to DB class material if needed.

LOCKING WITH S AND X LOCKS AT THE ROW-LEVEL



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)

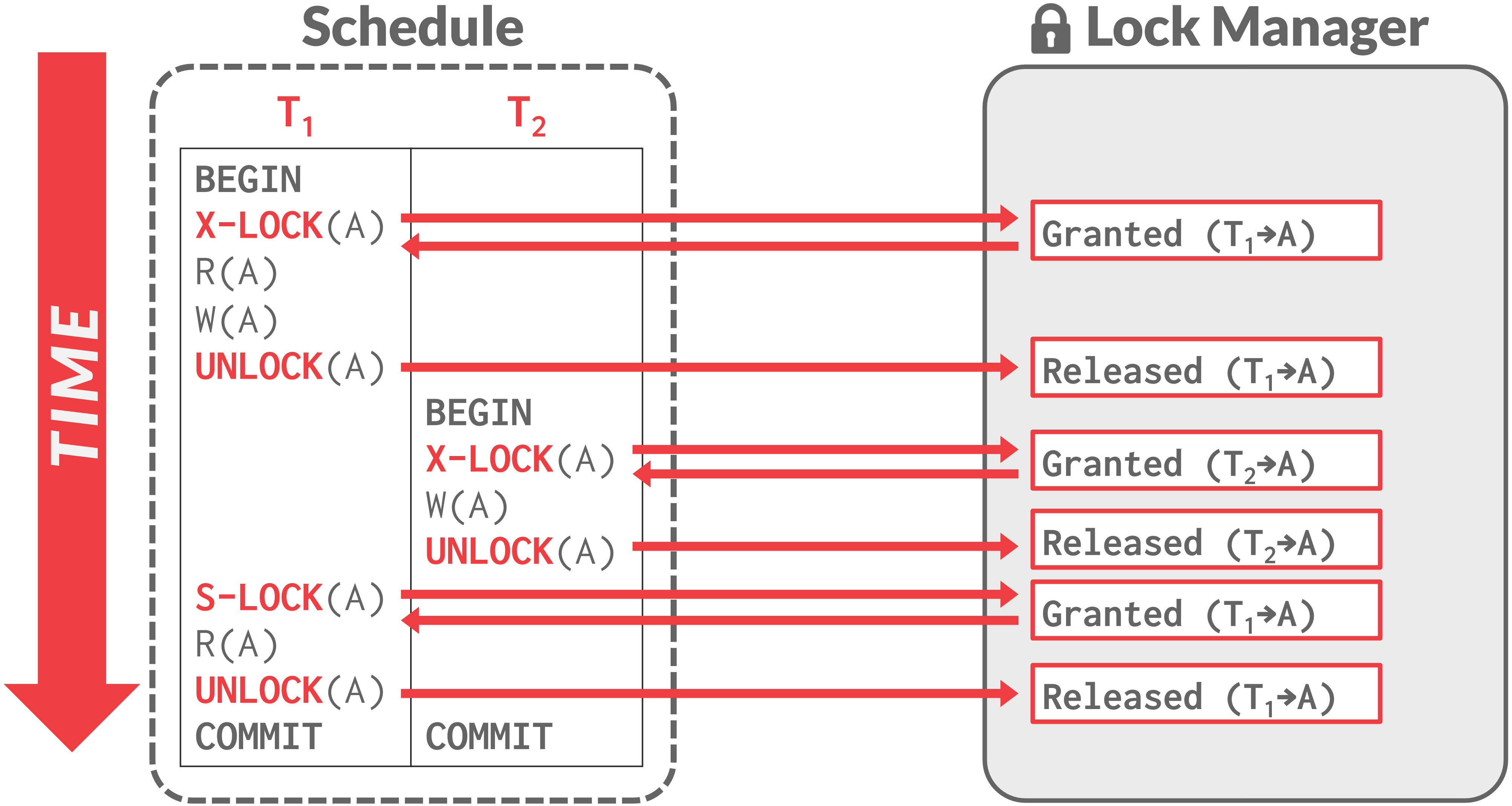
Released (T_1→A)

Granted (T_2→A)

Released (T_2→A)

Revisit your introduction to DB class material if needed.

Locking with S and X Locks at the Row-level



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)

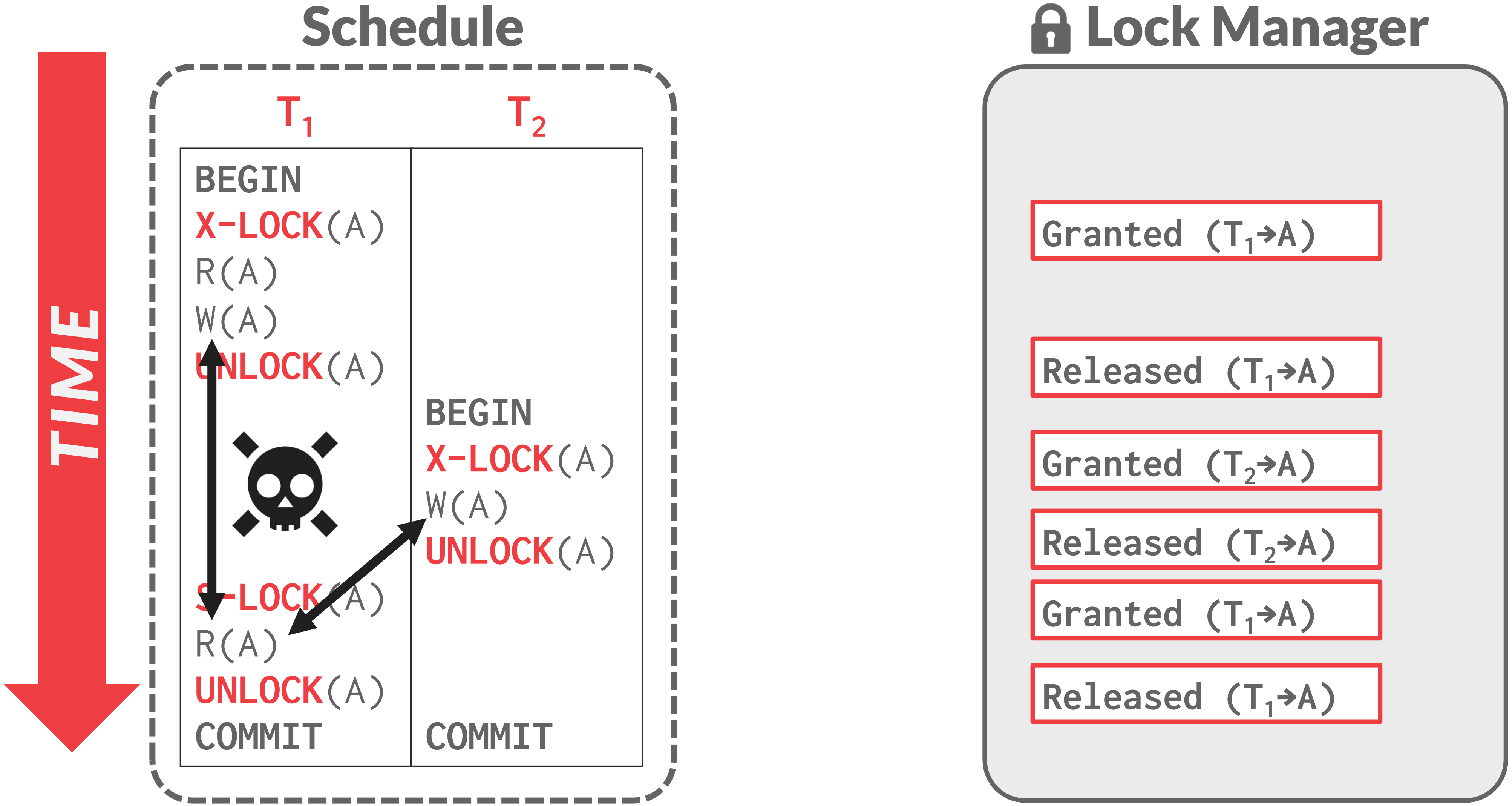
Released (T_2→A)

Granted (T_1→A)

Released (T_1→A)

Revisit your introduction to DB class material if needed.

Locking with S and X Locks at the Row-level



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)

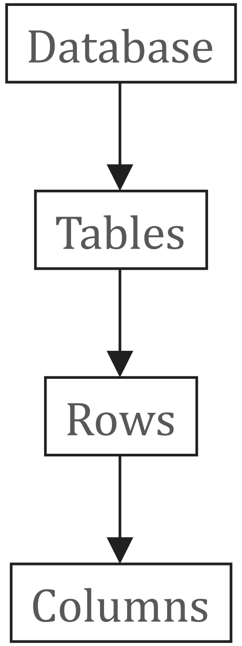
Released (T_2→A)

Granted (T_1→A)

Released (T_1→A)

Granularities Paper: Key Challenge

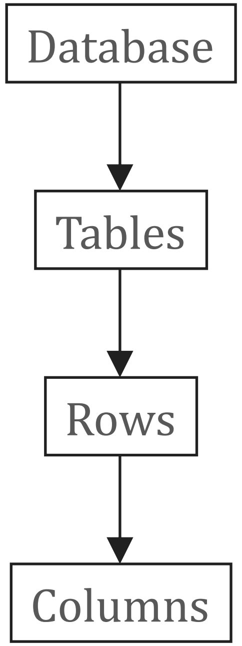
A Resource Hierarchy



Granularities Paper: Key Challenge

- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.

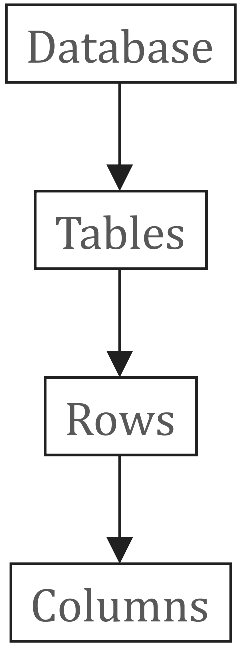
A Resource Hierarchy



Granularities Paper: Key Challenge

- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?

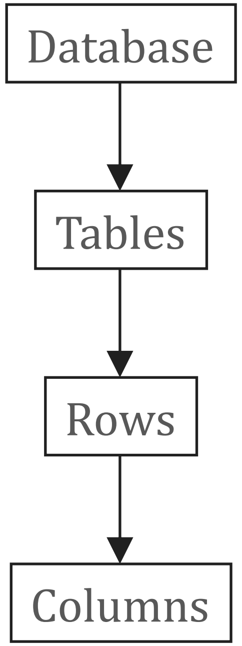
A Resource Hierarchy



GRANULARITIES PAPER: KEY CHALLENGE

- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?
- Want High throughput, but simple methods result in Low throughput.

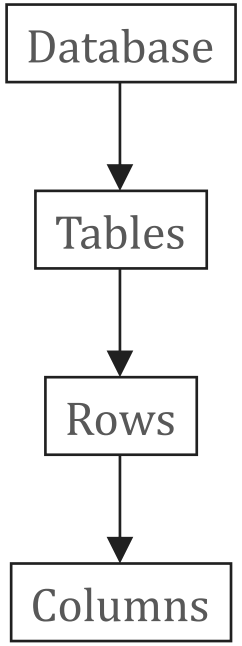
A Resource Hierarchy

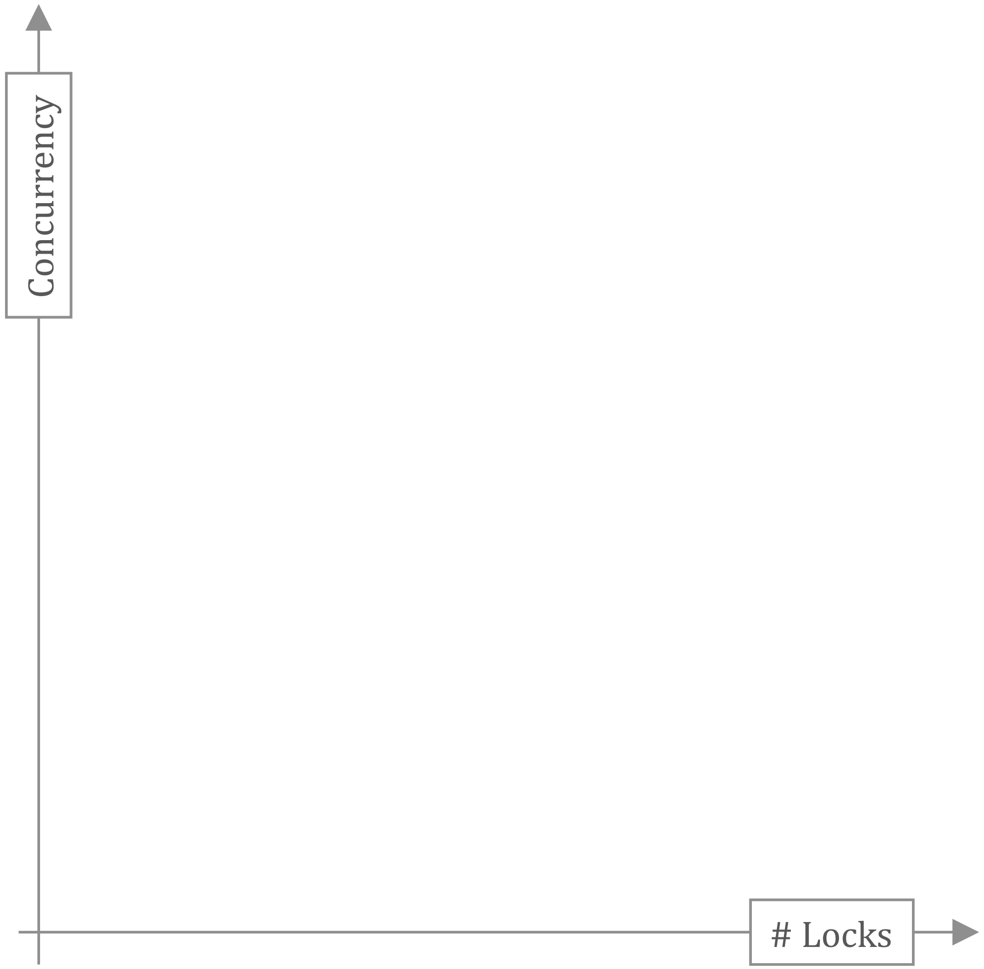


Granularities Paper: Key Challenge

- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?
- Want High throughput, but simple methods result in Low throughput.

A Resource Hierarchy

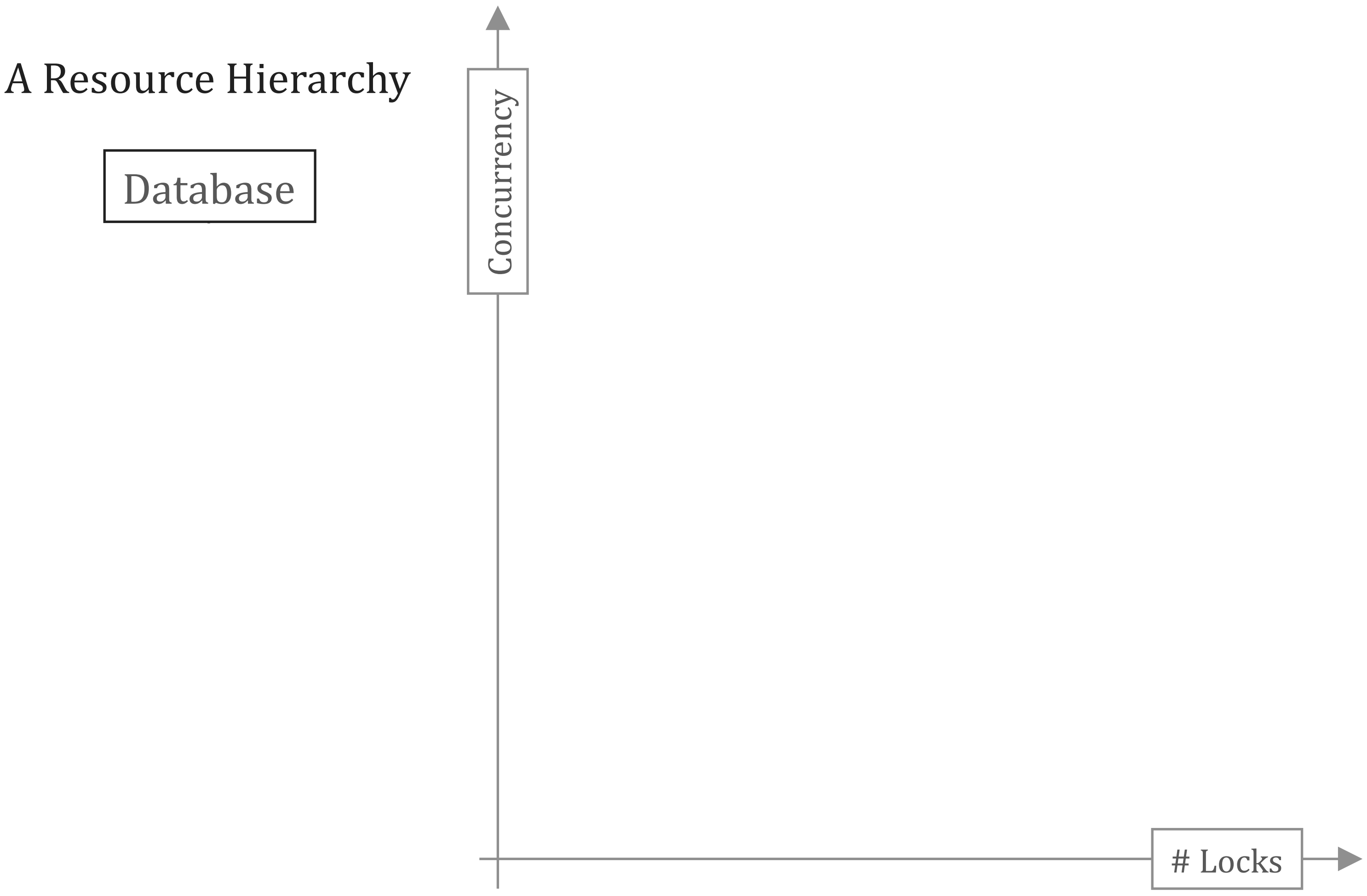




GRANULARITIES PAPER: KEY CHALLENGE

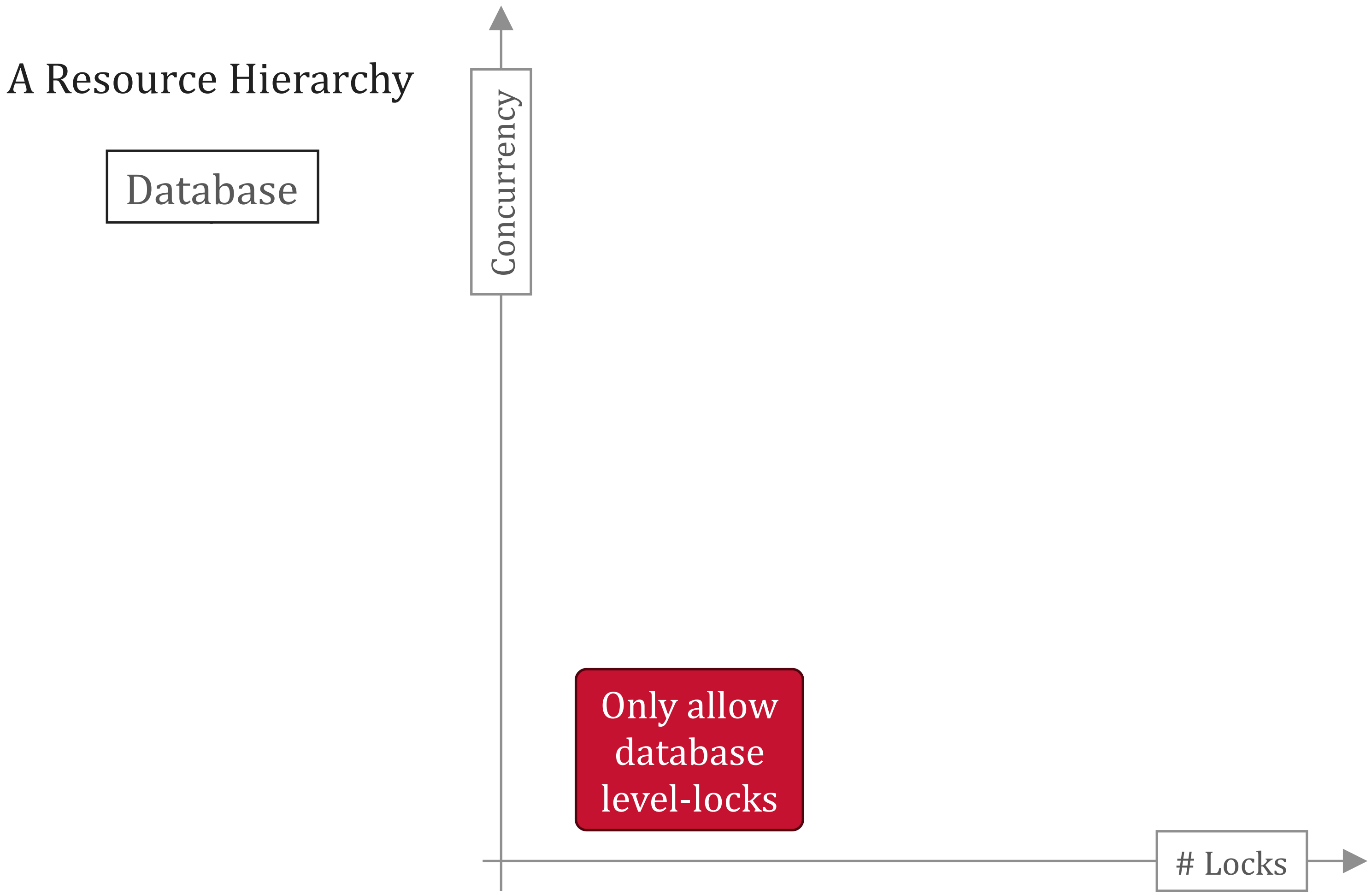
- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?
- Want High throughput, but simple methods result in Low throughput.

A Resource Hierarchy



GRANULARITIES PAPER: KEY CHALLENGE

- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?
- Want High throughput, but simple methods result in Low throughput.



A Resource Hierarchy

Database

Concurrency

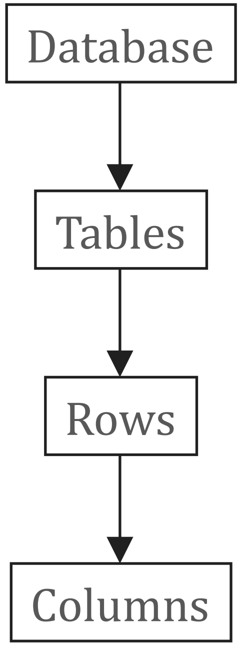
Only allow database level-locks

# Locks

GRANULARITIES PAPER: KEY CHALLENGE

- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?
- Want High throughput, but simple methods result in Low throughput.

A Resource Hierarchy

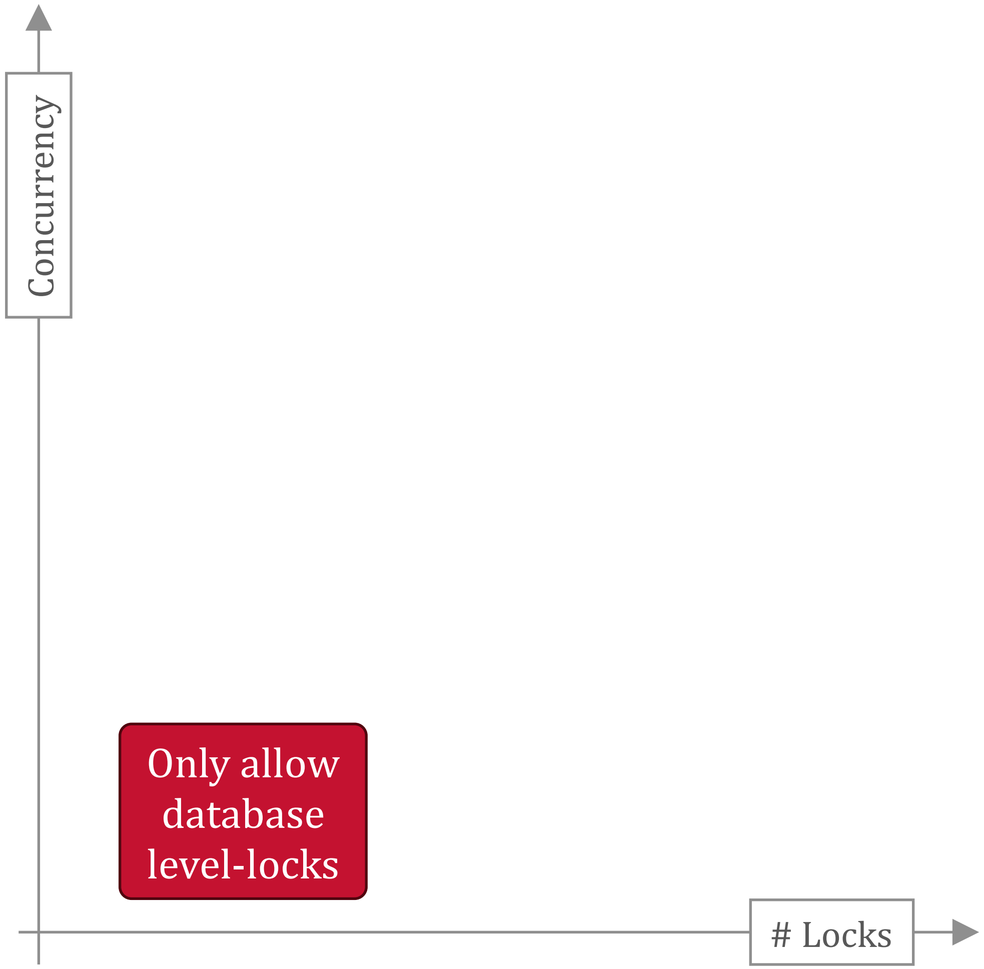


Database

Tables

Rows

Columns

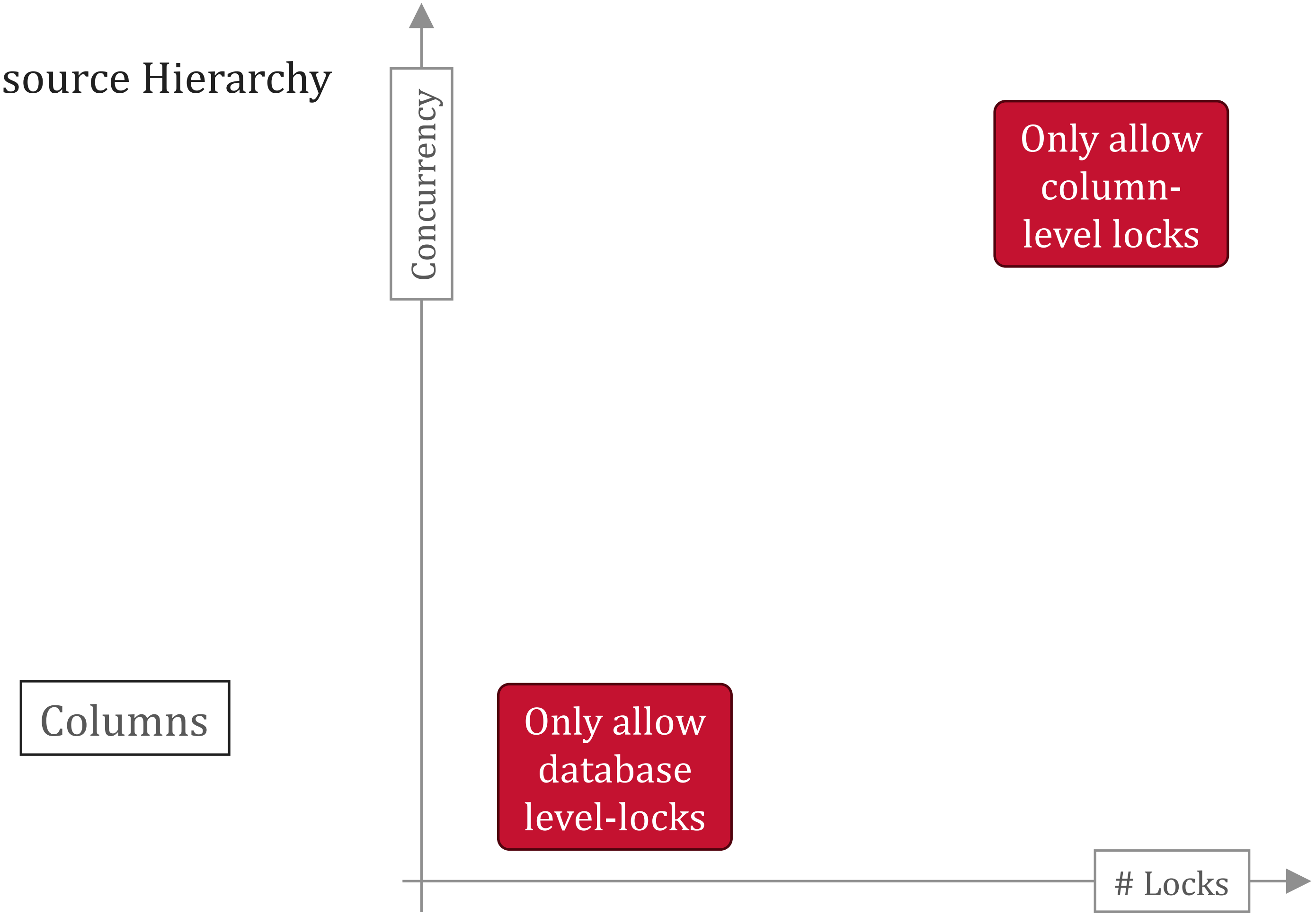


Only allow database level-locks

Granularities Paper: Key Challenge

- Correct (provably so) locking protocol that balances.
    - # locks that are acquired.
    - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?
- Want High throughput, but simple methods result in Low throughput.

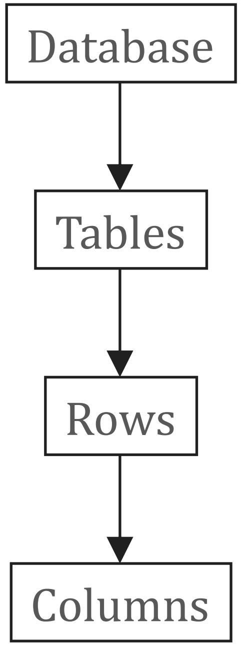
A Resource Hierarchy

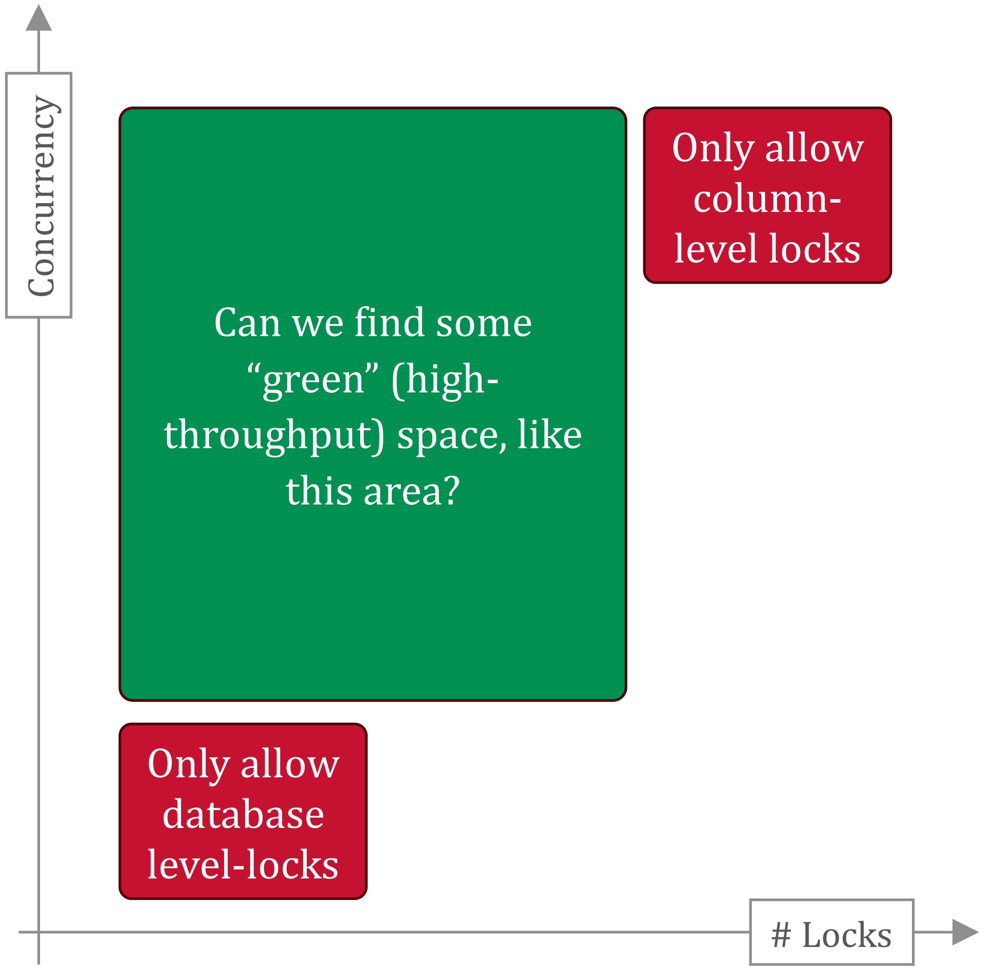

Concurrency
Only allow column-level locks
Columns
Only allow database level-locks
# Locks

GRANULARITIES PAPER: KEY CHALLENGE

- Correct (provably so) locking protocol that balances.
  - # locks that are acquired.
  - Amount of concurrency that is allowed by the protocol.
- Key metric is the transaction (txn) throughput: how many txns can we commit per second?
- Want High throughput, but simple methods result in Low throughput.

A Resource Hierarchy





Concurrency
# Locks
Only allow column-level locks
Only allow database level-locks

Can we find some “green” (high-throughput) space, like this area?

THE SOLUTION (DECEPTIVELY SIMPLE, AND HENCE BRILLIANT)

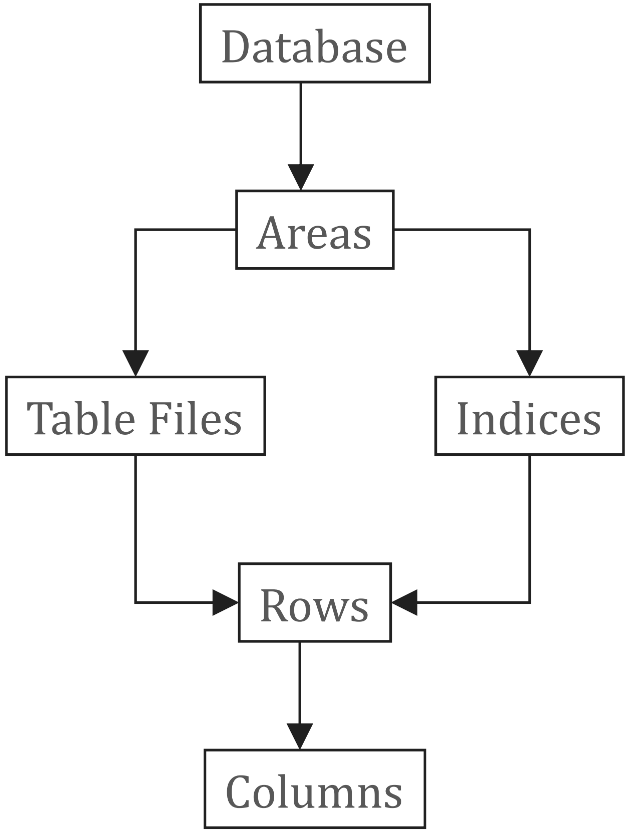
- Work with complex/DAG resource graphs.
- Use a well-formed protocol to acquire locks in top-down manner.
- Introduce the notion of "intention" locks to allow a txn to indicate that they will grab are "regular" lock (S or X) on a resource below in the hierarchy.
- Develop a novel lock-compatibility matrix that allows balancing # locks with the "granularity" of locking.
- Can offer different degrees of consistency (trading performance for lower consistency), allowing concurrent transaction to operate at different consistency levels.

1998 ACM Turing Award

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.

A Resource Hierarchy



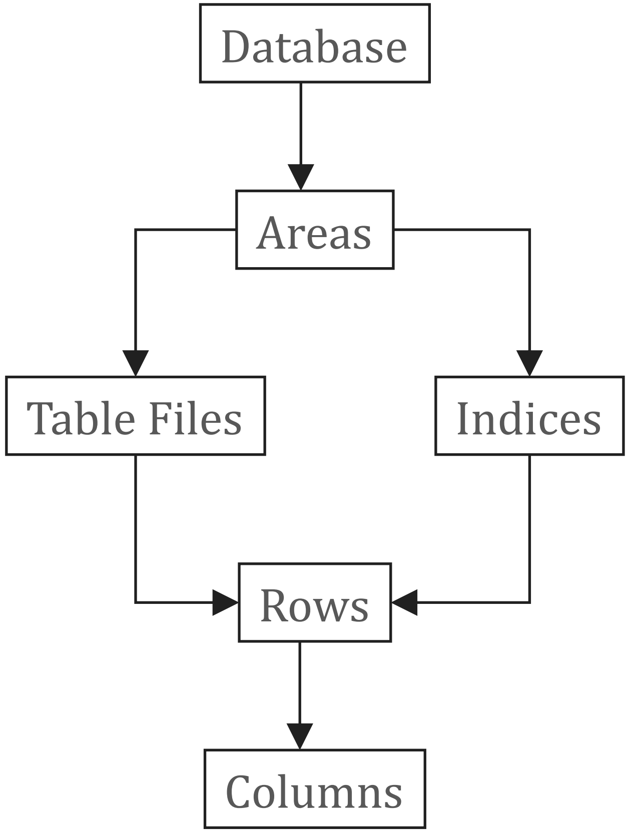
INTENTION LOCKS

- **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.

A Resource Hierarchy



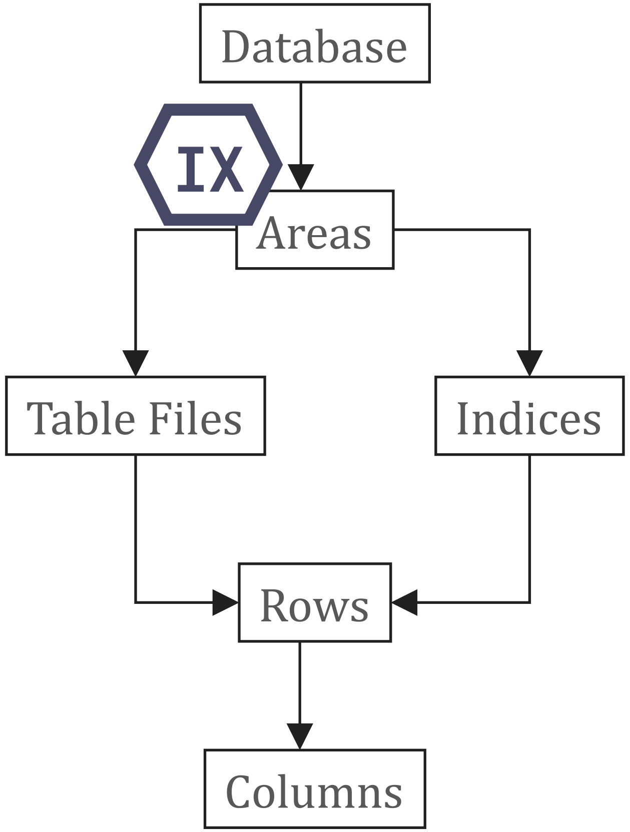
INTENTION LOCKS

- **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.

A Resource Hierarchy



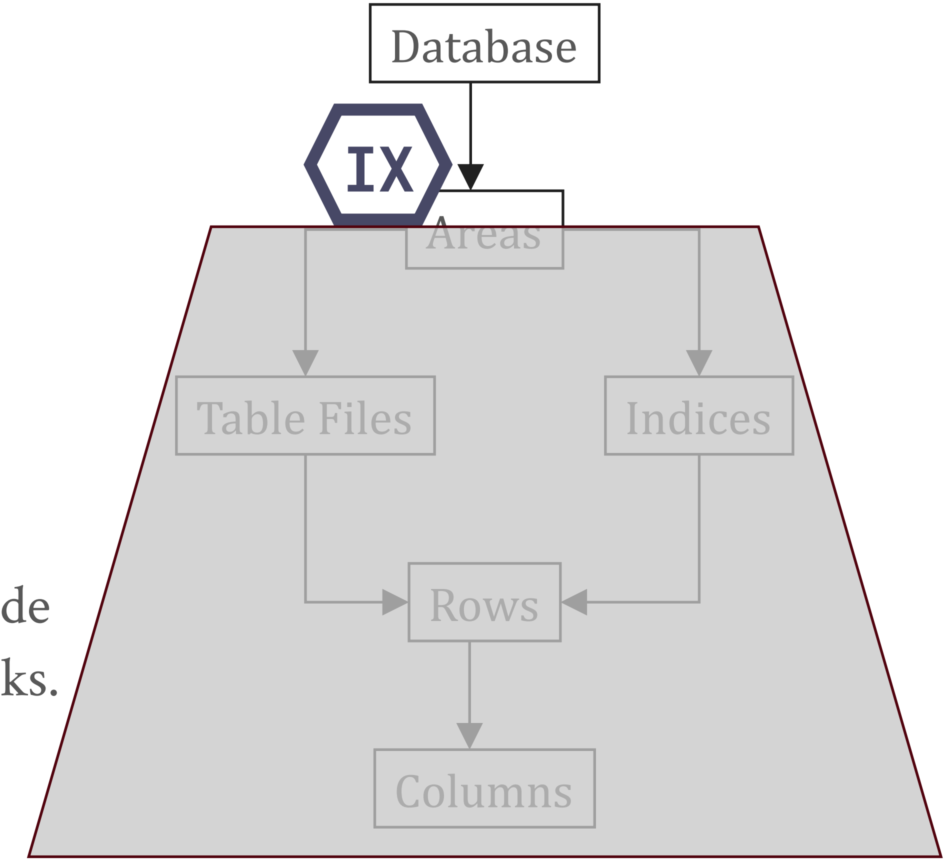
Intention Locks

- **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.

A Resource Hierarchy



An X lock will be acquired somewhere in the gray region.
An intention lock discloses the intent to do additional locking below.

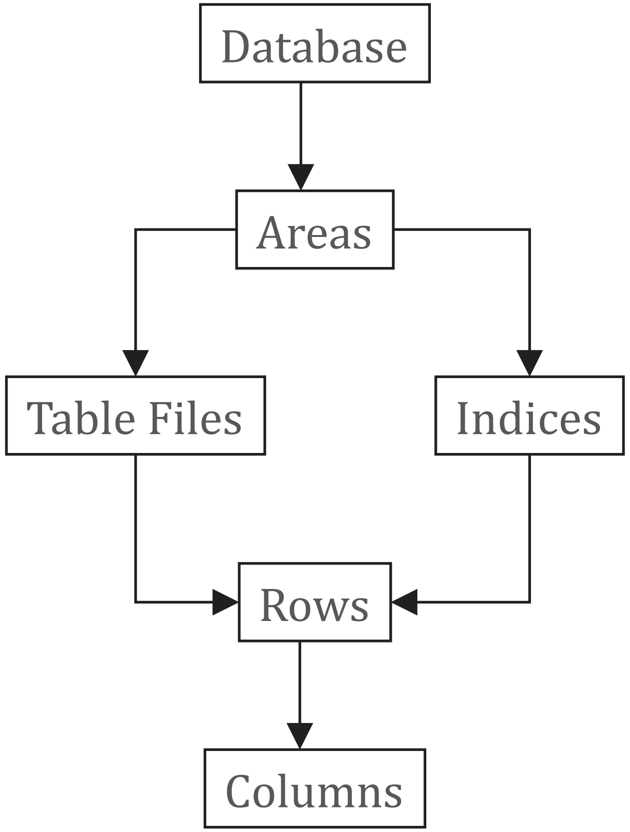
INTENTION LOCKS

- **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.

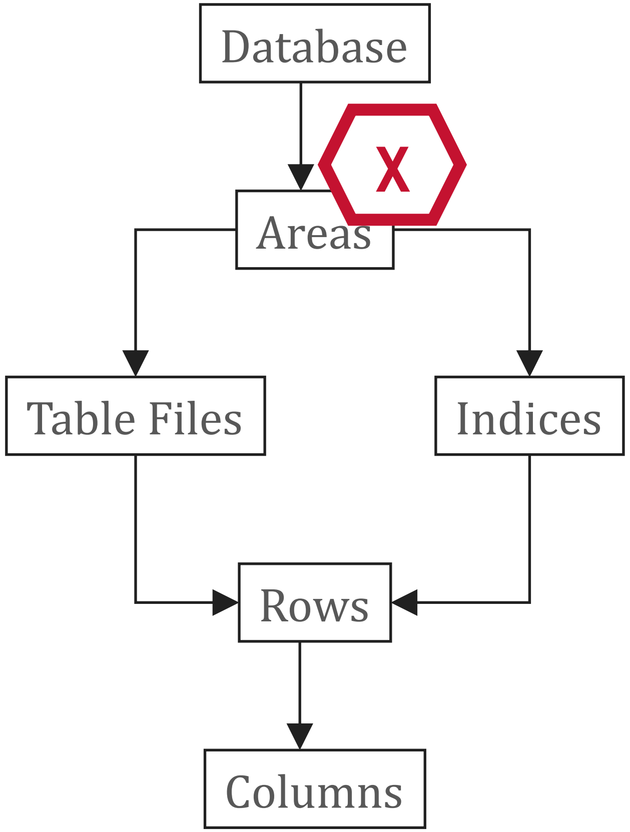
A Resource Hierarchy



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.

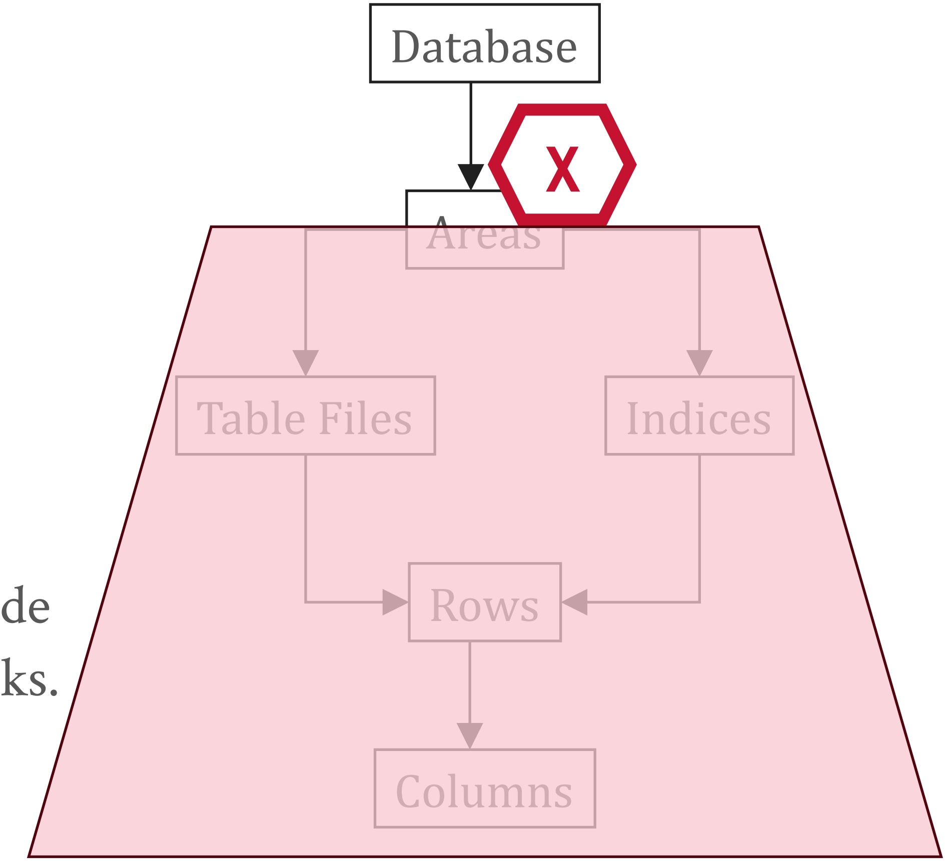
A Resource Hierarchy



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.

A Resource Hierarchy

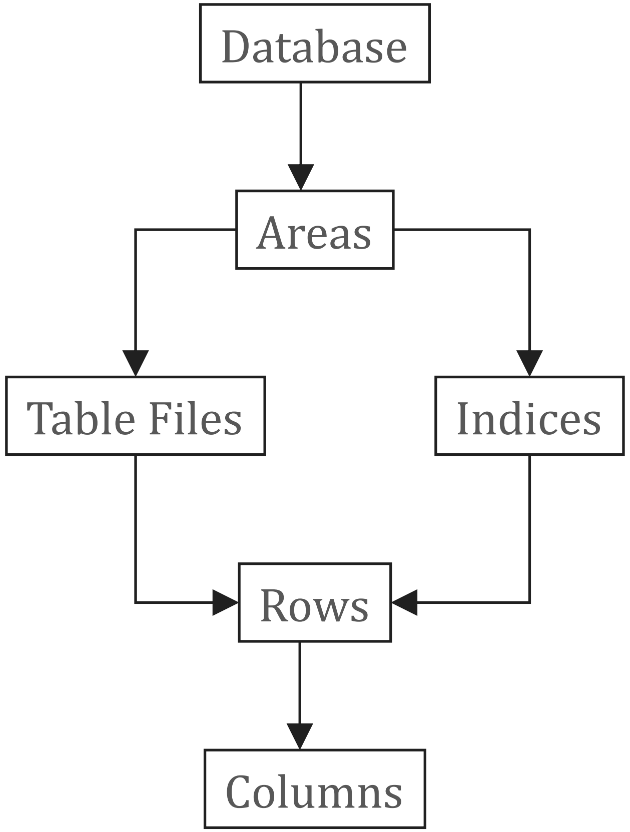


Now, this entire gray region is considered locked in X mode. No need to acquire X locks here. The lock at the top level "covers" this whole region.

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.

A Resource Hierarchy



Compatibility Matrix

T_2 Wants

T_1 Holds

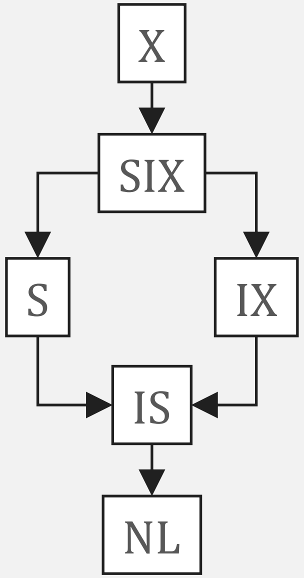
	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

Locking Protocol: Well Formed

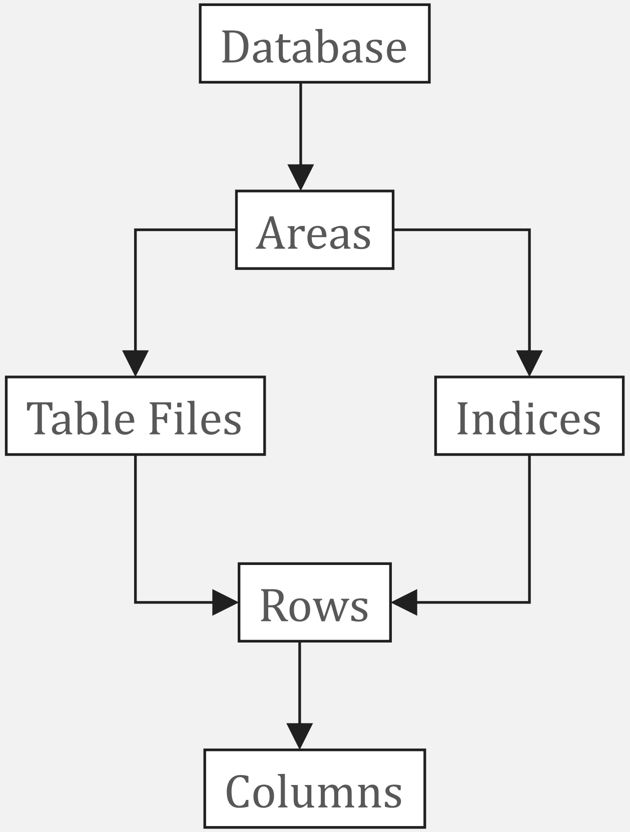
- Each txn obtains an appropriate lock at the highest level of the hierarchy.
- To get **S** or **IS** lock on a node, the txn must hold at least **IS** on parent node.
- To get **X**, **IX**, or **SIX** on a node, must hold at least **IX** on parent node.
- All lock are acquired top-down, so if a txn has an intention lock, every other txn will see that before they acquire lock at a lower level in the resource hierarchy.
- Locks released leaf to root, or all at once at the end of the txn.
- Need non-intention locks somewhere in the resource hierarchy (so can't have txns that only do intention locks).

The partial ordering of the lock modes.

Higher is more restrictive.

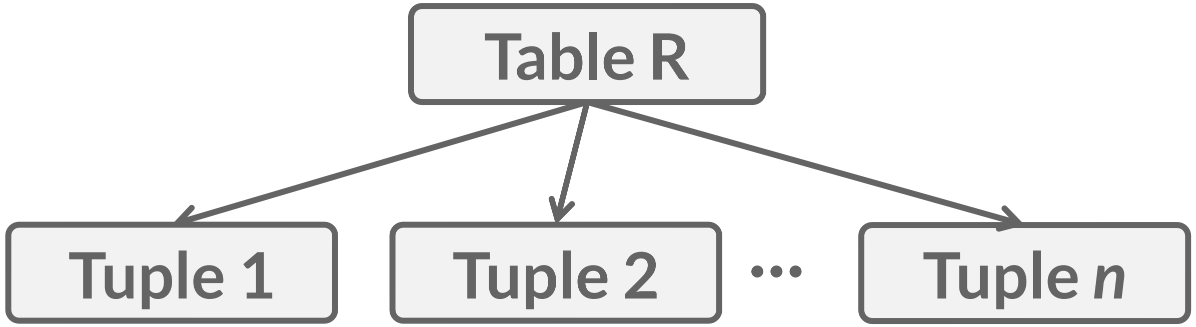


A Resource Hierarchy



Example

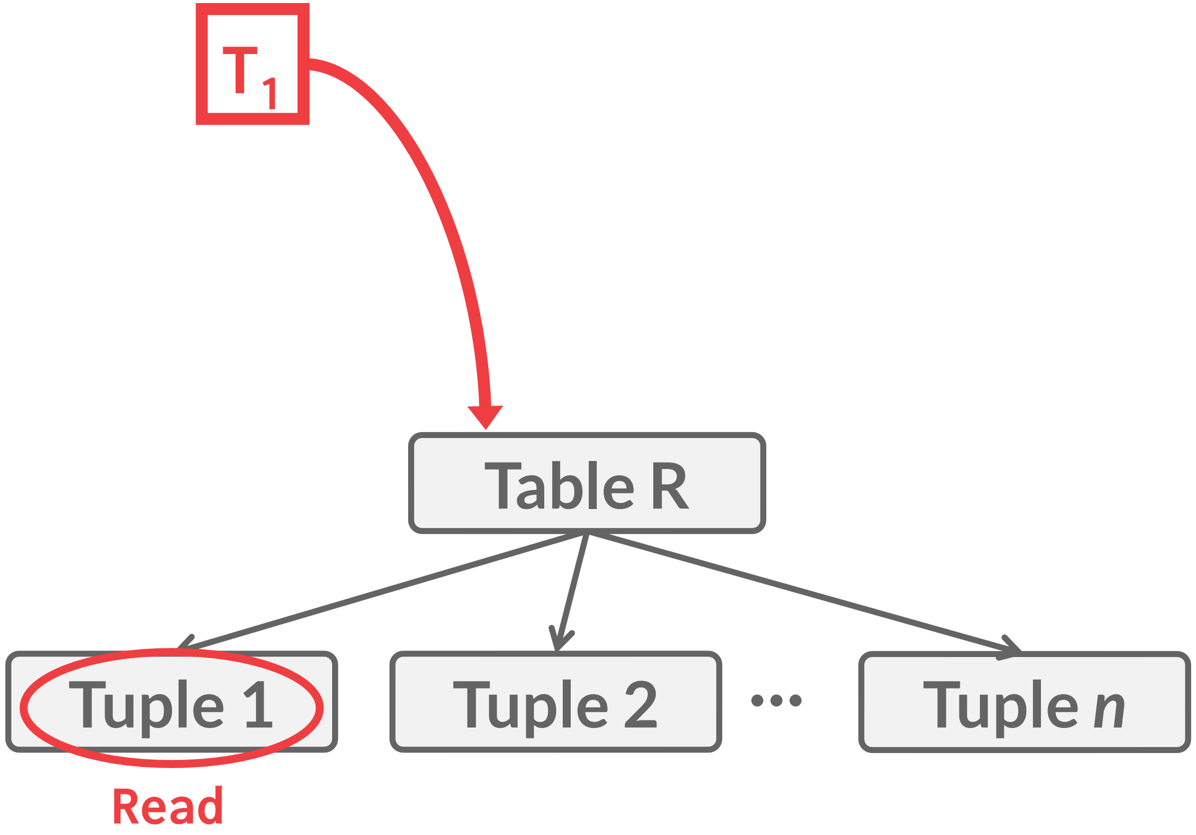
- T_1 – Get the balance of Alice's account.
- T_2 – Increase Bob's account by 1%.
- What locks should these txns obtain?
  - **Exclusive** + **Shared** for leaf nodes of lock tree.
  - Special **Intention** locks for higher levels.



Example

- T_1 – Get the balance of Alice's account.
- T_2 – Increase Bob's account by 1%.
- What locks should these txns obtain?
    - **Exclusive** + **Shared** for leaf nodes of lock tree.
    - Special **Intention** locks for higher levels.

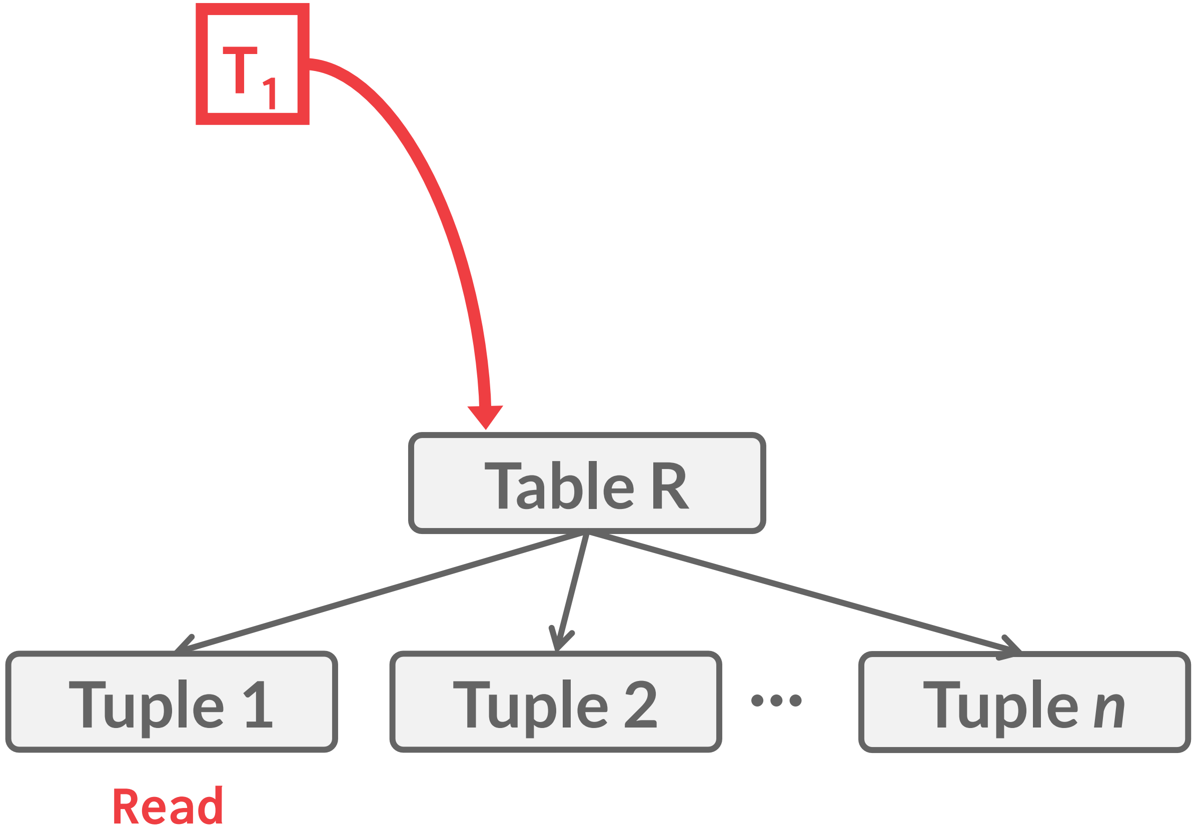
Read Alice's record in **R**.



EXAMPLE

- T_1 – Get the balance of Alice's account.
- T_2 – Increase Bob's account by 1%.
- What locks should these txns obtain?
  - **Exclusive** + **Shared** for leaf nodes of lock tree.
  - Special **Intention** locks for higher levels.

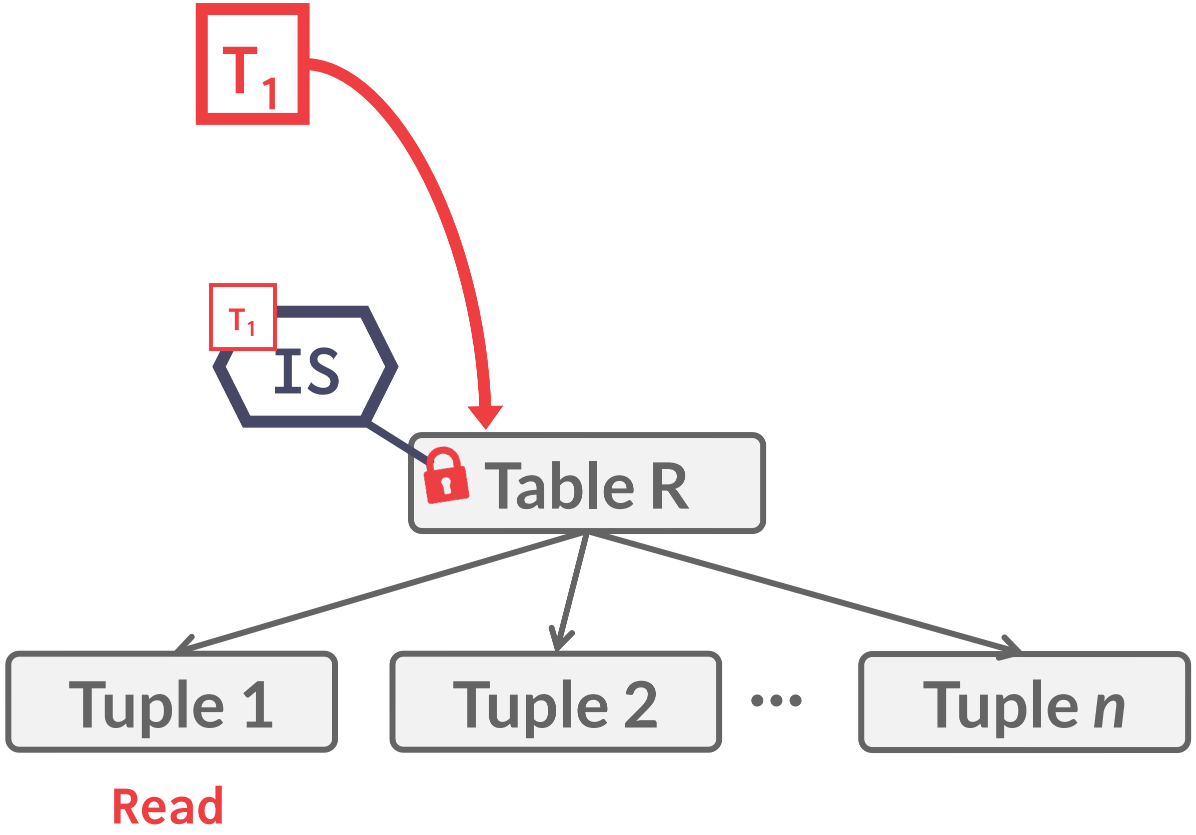
Read Alice's record in **R**.



EXAMPLE

- T_1 – Get the balance of Alice's account.
- T_2 – Increase Bob's account by 1%.
- What locks should these txns obtain?
  - **Exclusive** + **Shared** for leaf nodes of lock tree.
  - Special **Intention** locks for higher levels.

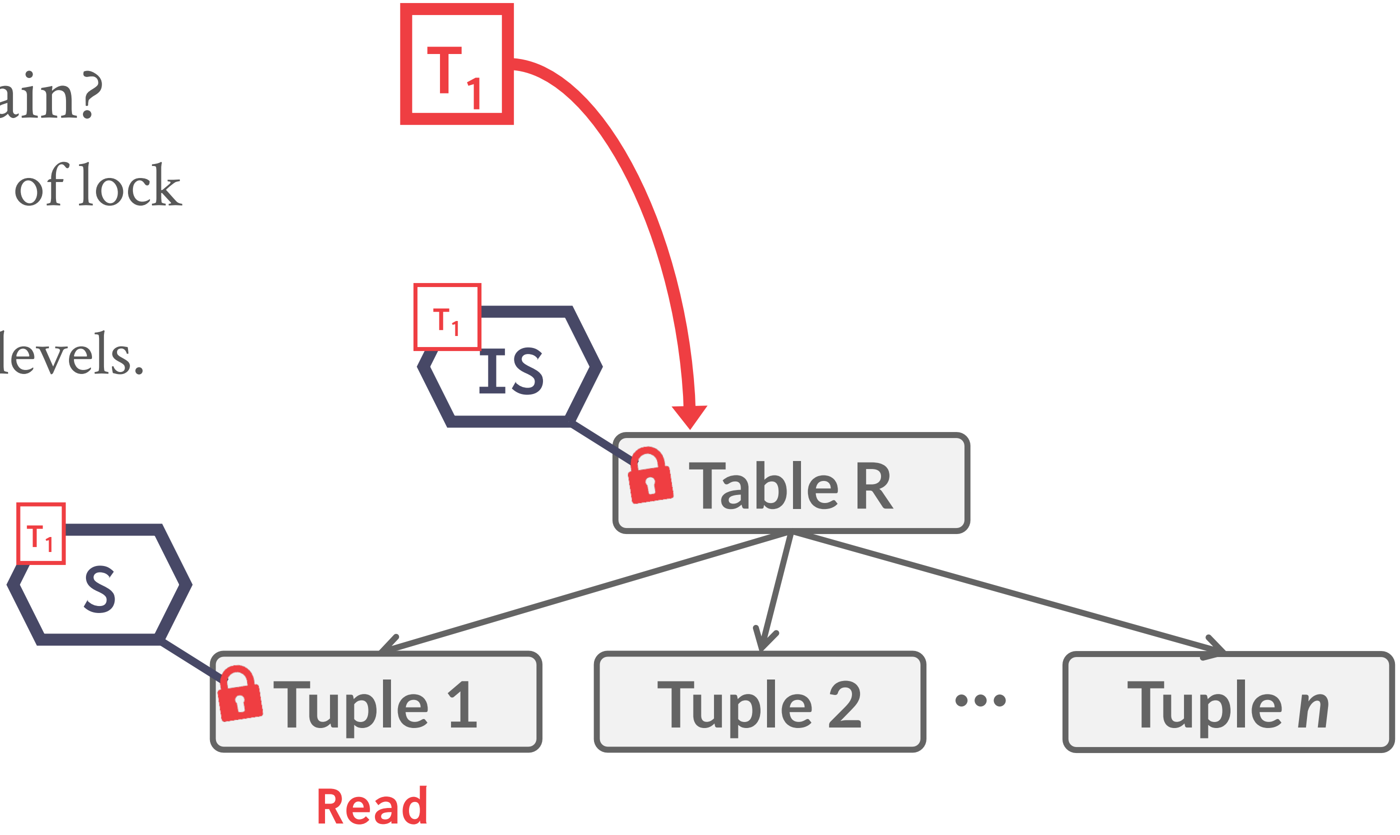
Read Alice's record in **R**.



EXAMPLE

- T_1 – Get the balance of Alice's account.
- T_2 – Increase Bob's account by 1%.
- What locks should these txns obtain?
  - **Exclusive** + **Shared** for leaf nodes of lock tree.
  - Special **Intention** locks for higher levels.

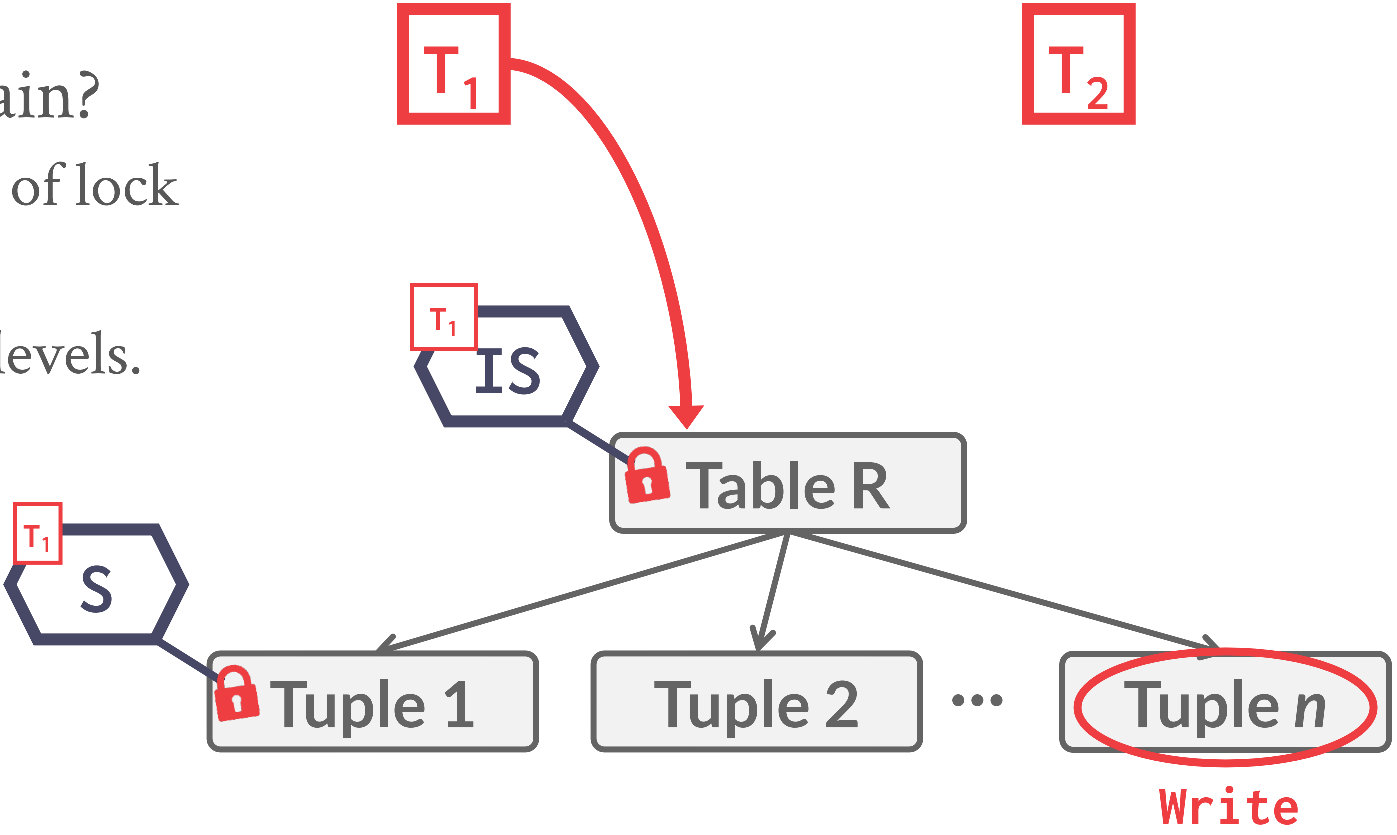
Read Alice's record in **R**.



Example

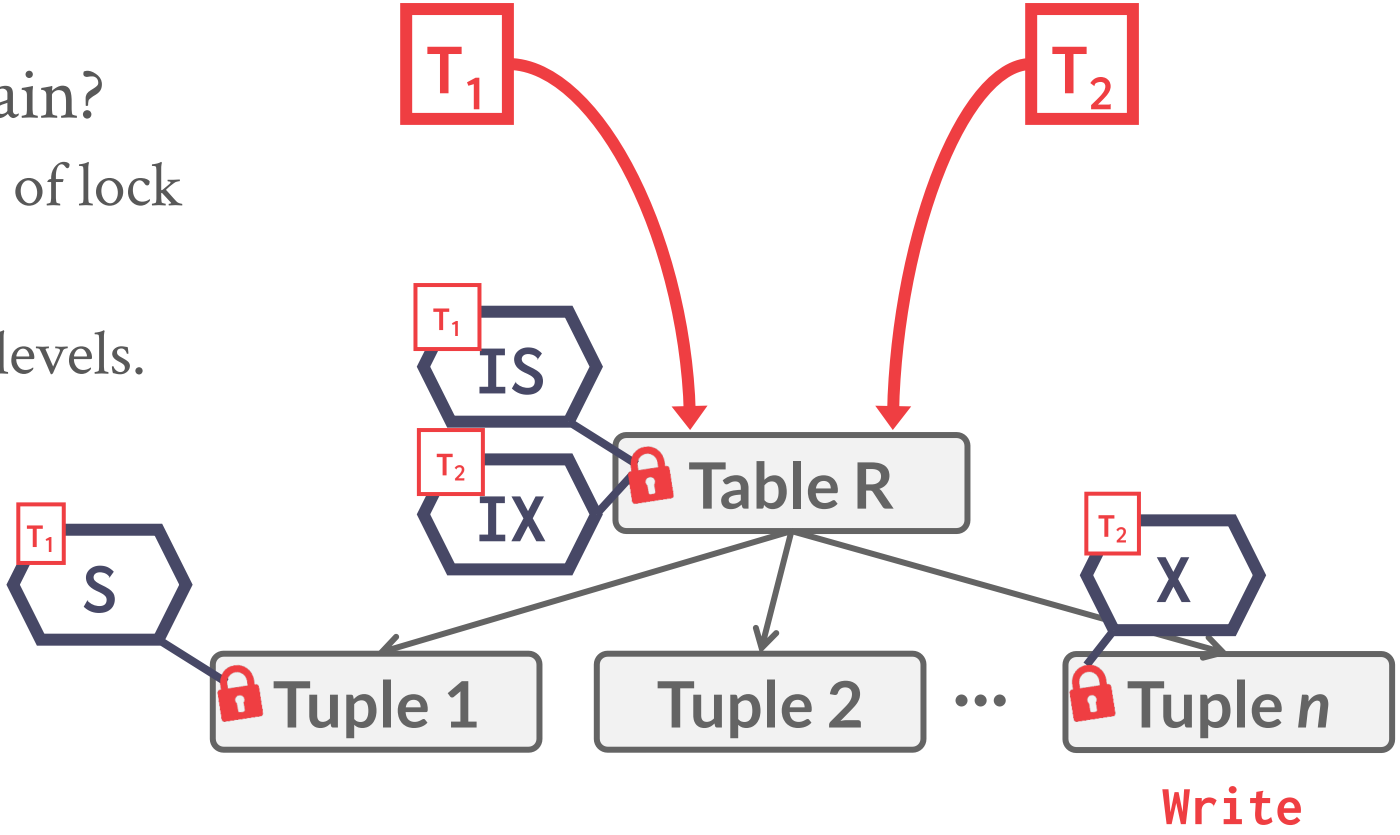
- T_1 – Get the balance of Alice's account.
- T_2 – Increase Bob's account by 1%.
- What locks should these txns obtain?
  - **Exclusive** + **Shared** for leaf nodes of lock tree.
  - Special **Intention** locks for higher levels.

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



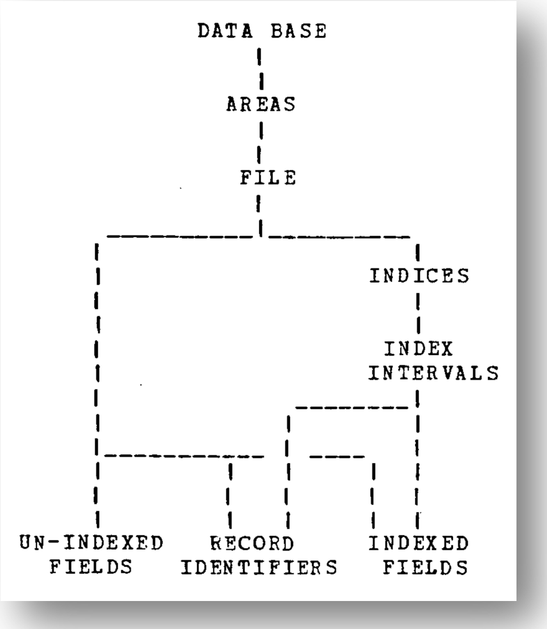
Example

- T_1 – Get the balance of Alice's account.
- T_2 – Increase Bob's account by 1%.
- What locks should these txns obtain?
  - **Exclusive** + **Shared** for leaf nodes of lock tree.
  - Special **Intention** locks for higher levels.



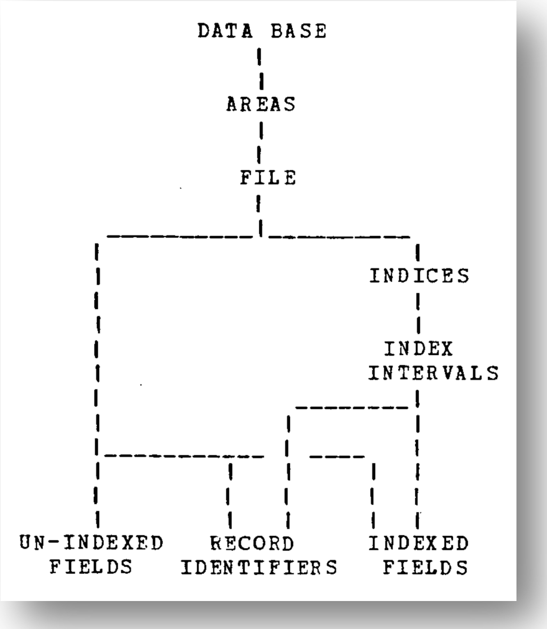
Dynamic Lock Graph

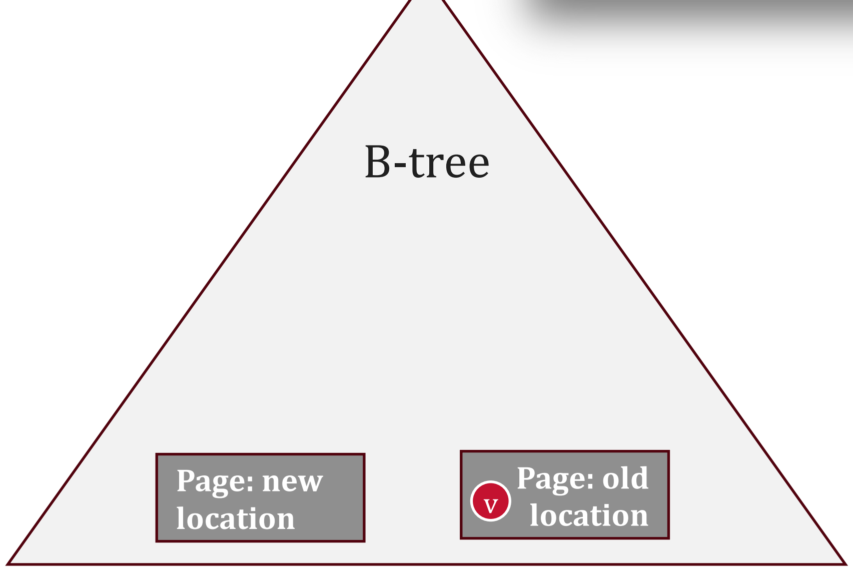
- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.



Dynamic Lock Graph

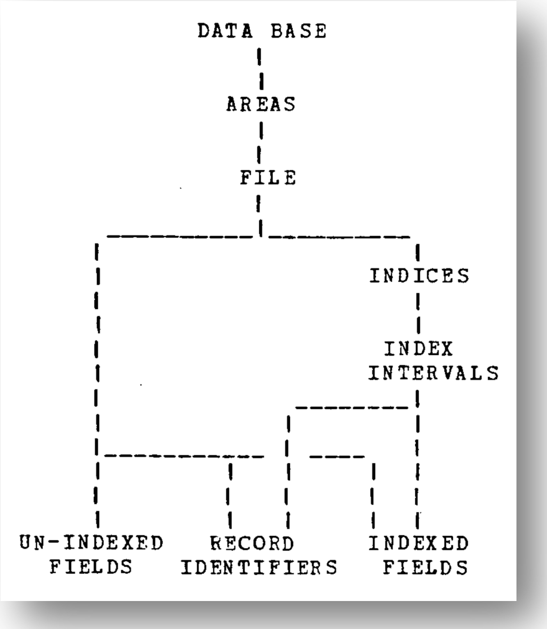
- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.

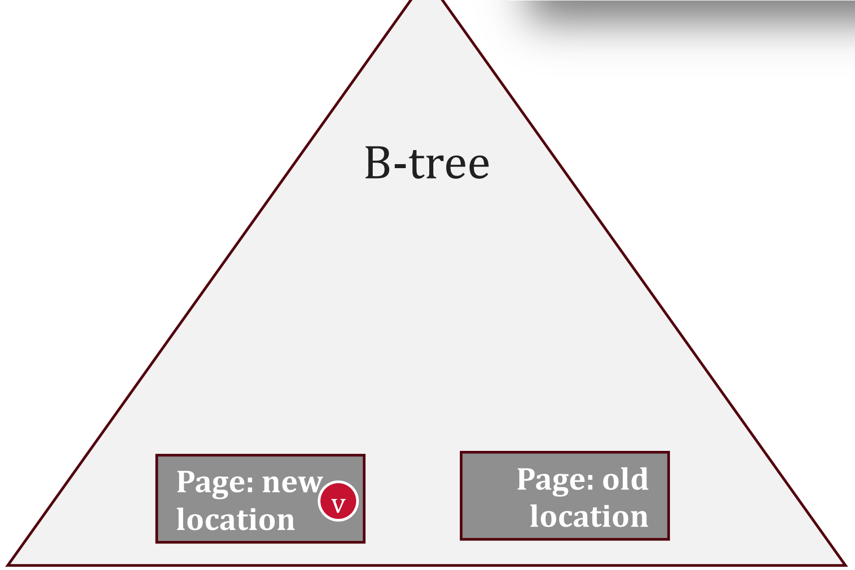




Dynamic Lock Graph

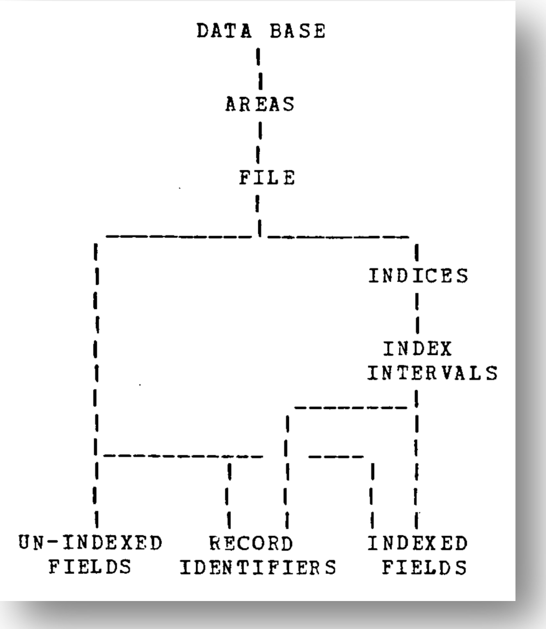
- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.

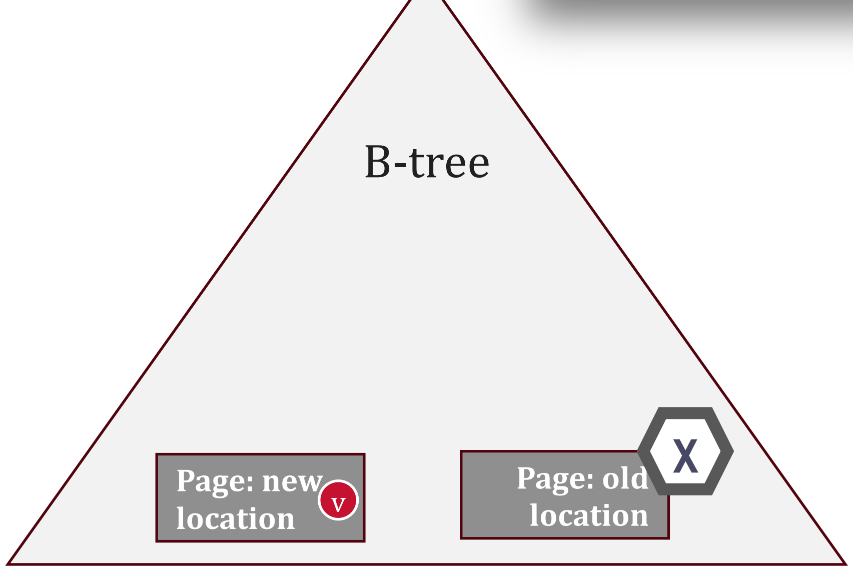




DYNAMIC LOCK GRAPH

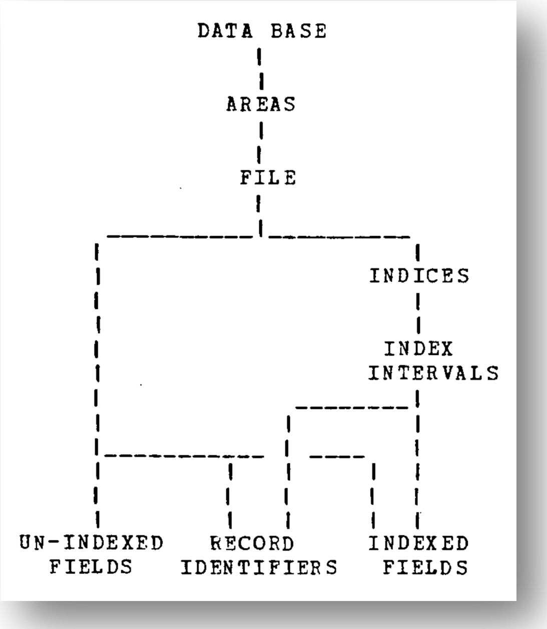
- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.

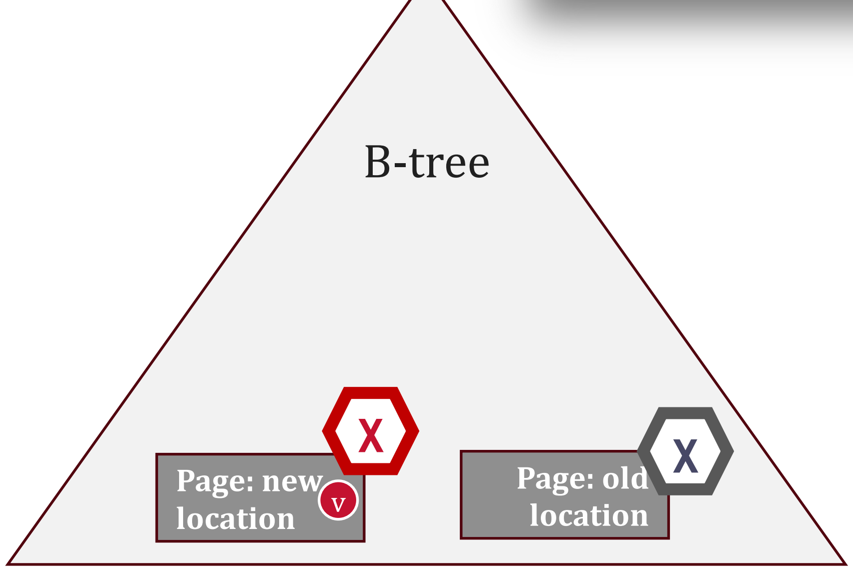




DYNAMIC LOCK GRAPH

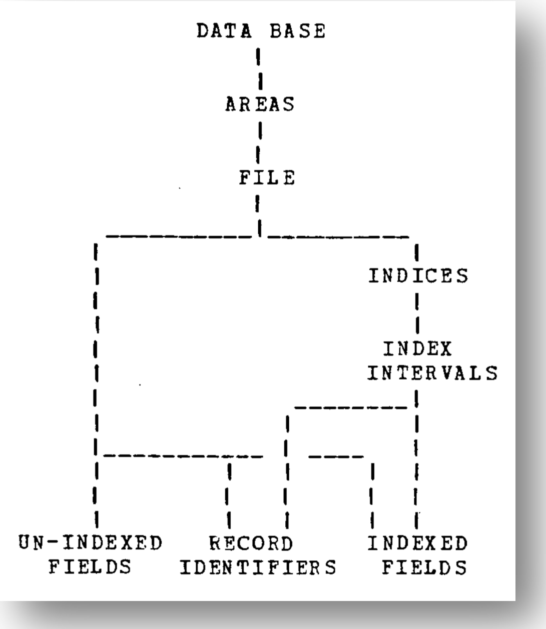
- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.

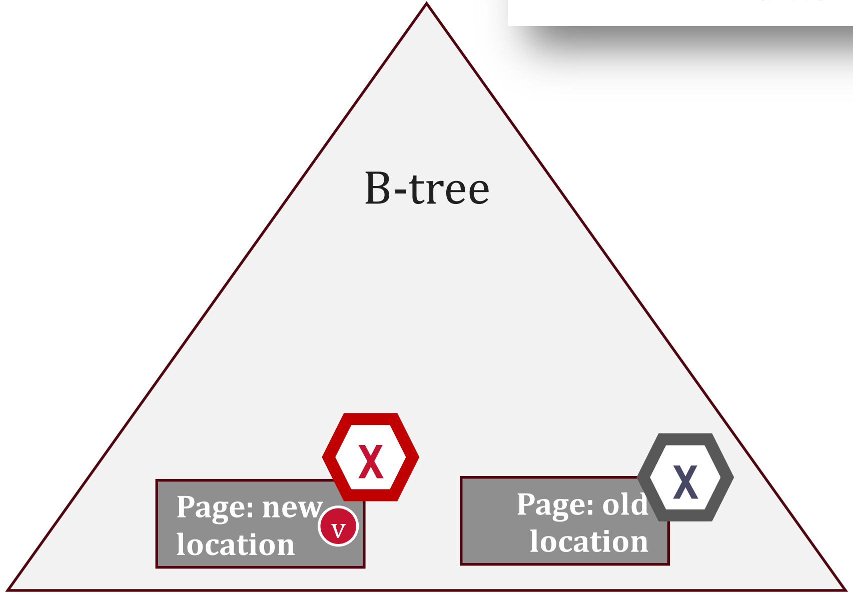




DYNAMIC LOCK GRAPH

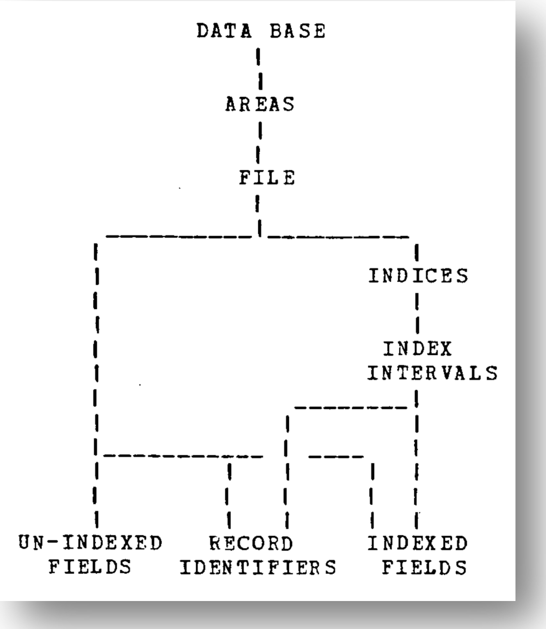
- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.
- Now in the new area, we may not have the appropriate locks on the ancestors.

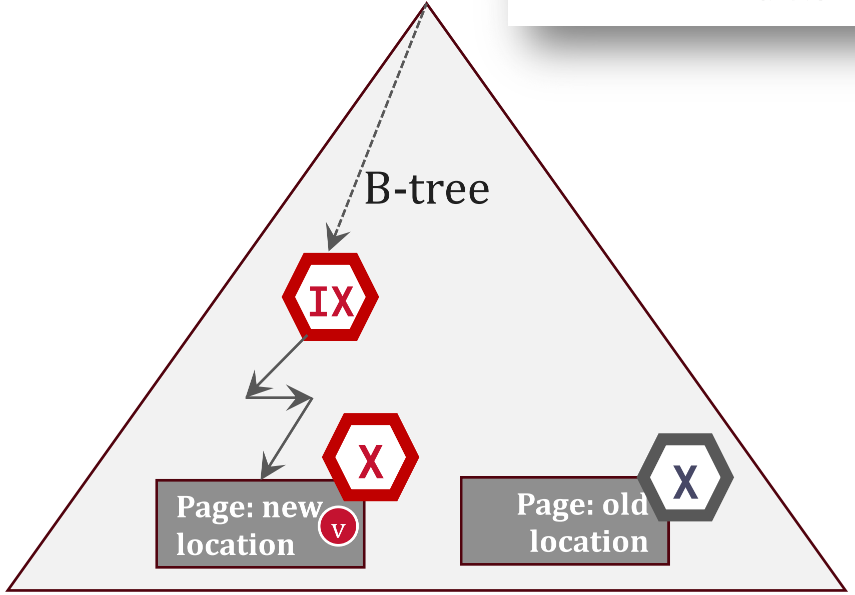




Dynamic Lock Graph

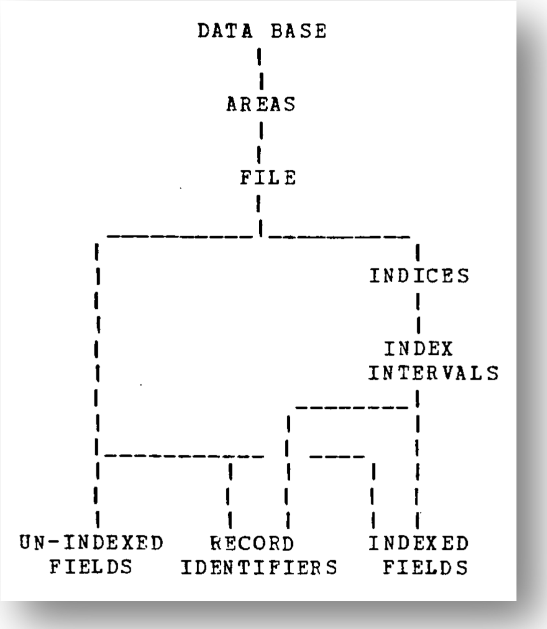
- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.
- Now in the new area, we may not have the appropriate locks on the ancestors.

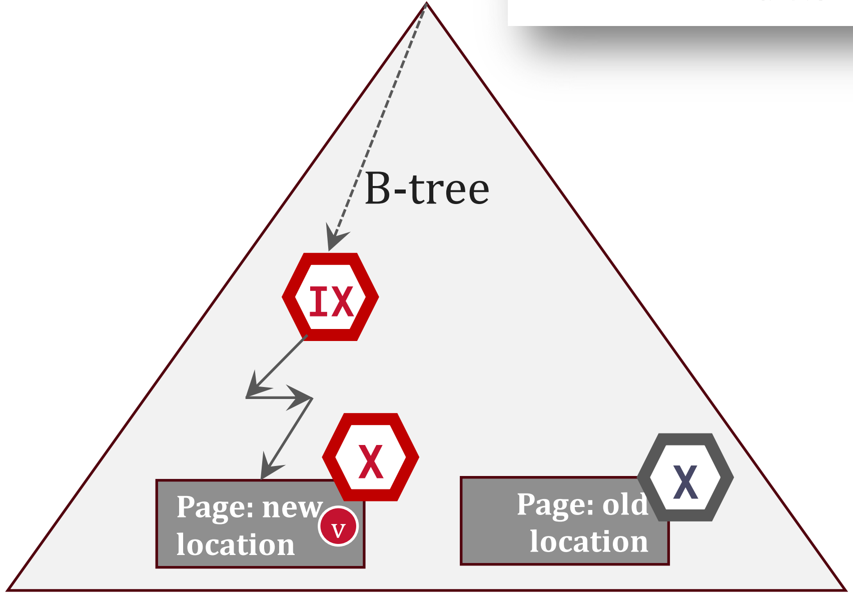




Dynamic Lock Graph

- Data can move around (in the resource graph) in the same transaction; e.g., an update that moves data from one part of an index to another.
- Now in the new area, we may not have the appropriate locks on the ancestors.
- Solution: Before moving data, both the old and new locations must have an X lock, and the well-formed protocol must be preserved so that top-down traversal by another txn does not grab a conflicting lock.

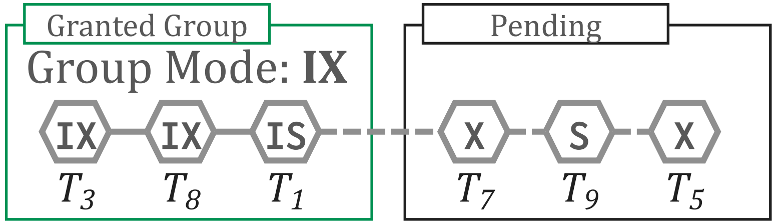




LOCK SCHEDULES AND UPGRADES

- A single resource may have multiple locks. Group mode is the highest level of lock on that resource.

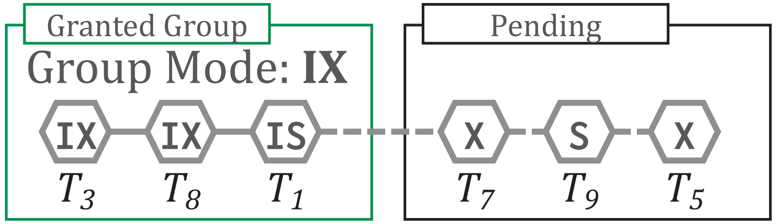
  e.g., if a group has an mix of IS and IX locks, the group mode is IX.



Lock Schedules and Upgrades

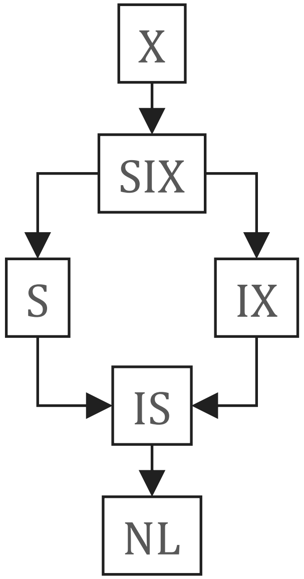
- A single resource may have multiple locks. Group mode is the highest level of lock on that resource.

  e.g., if a group has an mix of IS and IX locks, the group mode is IX.



		New Mode				
		IS	IX	S	SIX	X
Current Mode	IS	IS	IX	S	SIX	X
	IX	IX	IX	SIX	SIX	X
	S	S	SIX	S	SIX	X
	SIX	SIX	SIX	SIX	SIX	X
	X	X	X	X	X	X

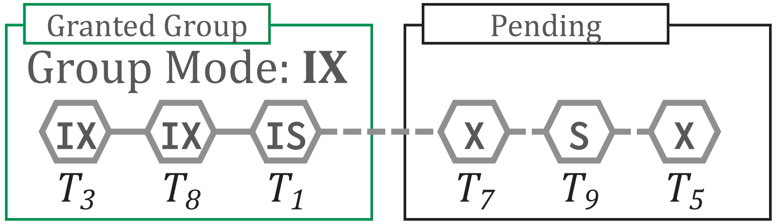
The partial ordering of the lock modes.



Lock Schedules and Upgrades

- A single resource may have multiple locks. Group mode is the highest level of lock on that resource.

  e.g., if a group has an mix of IS and IX locks, the group mode is IX.

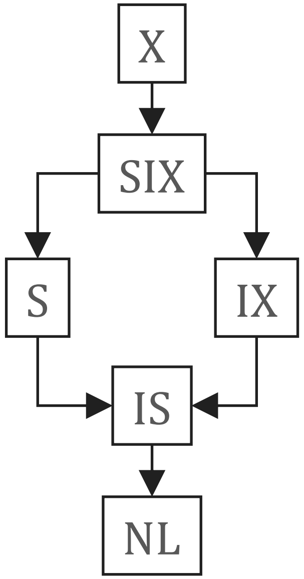


- Some notion of fairness (e.g., FIFO) is needed so some txn does not wait forever on a lock request.
- Lock upgrade request: Give priority to a txn in the pending queue if it is already part of the granted group.

  This txn is already holding a resource. Try to get this txn to finish quickly, and free up this resource.

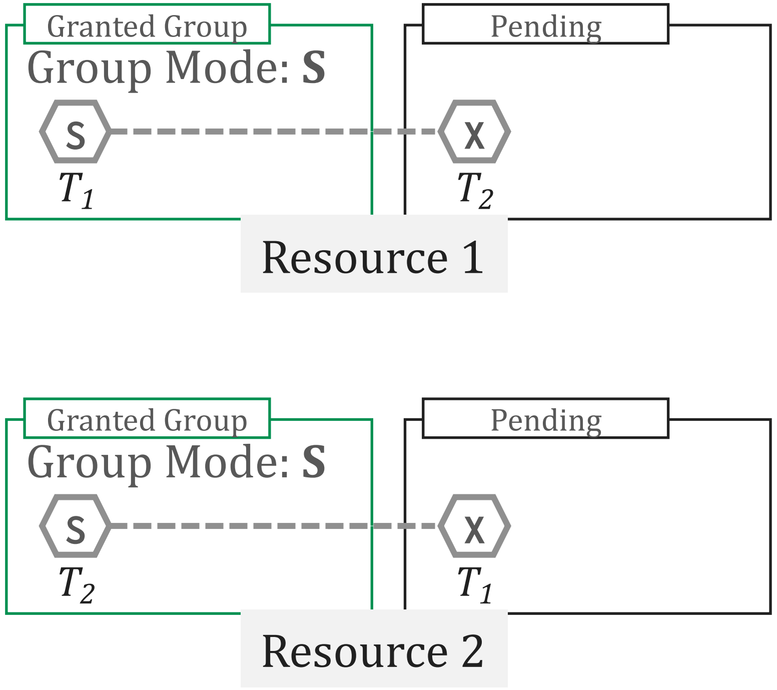
	New Mode				
Current Mode	IS	IX	S	SIX	X
IS	IS	IX	S	SIX	X
IX	IX	IX	SIX	SIX	X
S	S	SIX	S	SIX	X
SIX	SIX	SIX	SIX	SIX	X
X	X	X	X	X	X

The partial ordering of the lock modes.



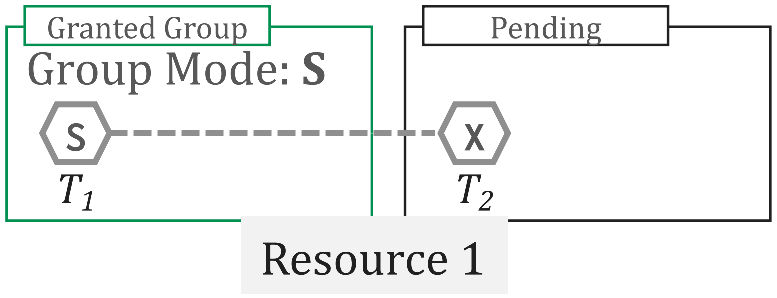
DEADLOCKS

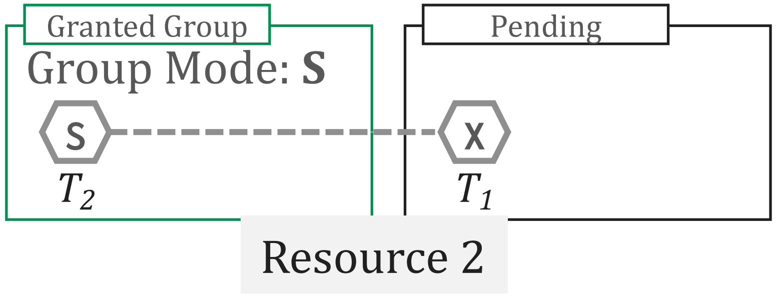
- We can now have deadlocks.

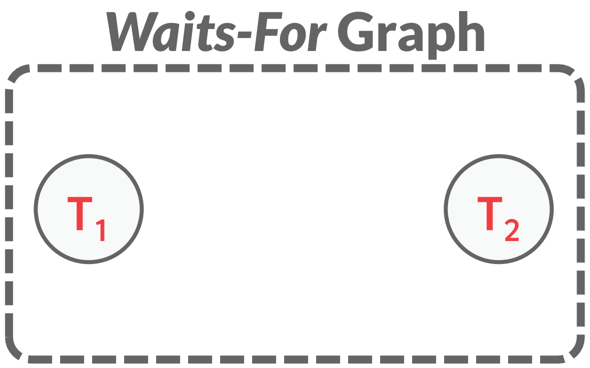


DEADLOCKS

- We can now have deadlocks.

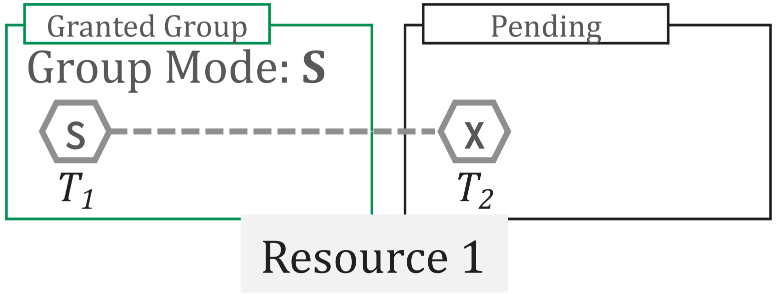


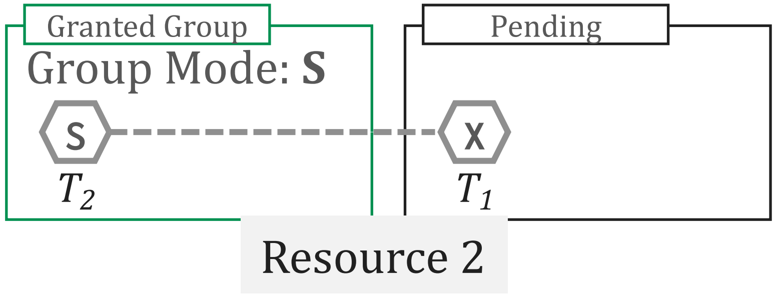




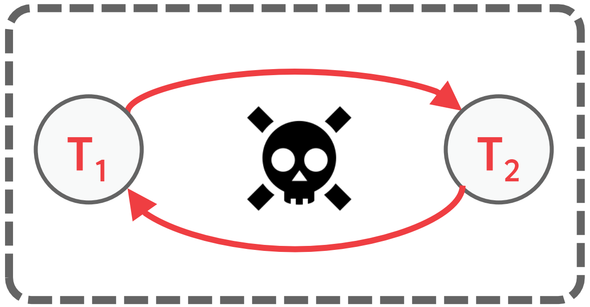
DEADLOCKS

- We can now have deadlocks.
- Need a mechanism to either detect deadlocks, or prevent deadlocks.



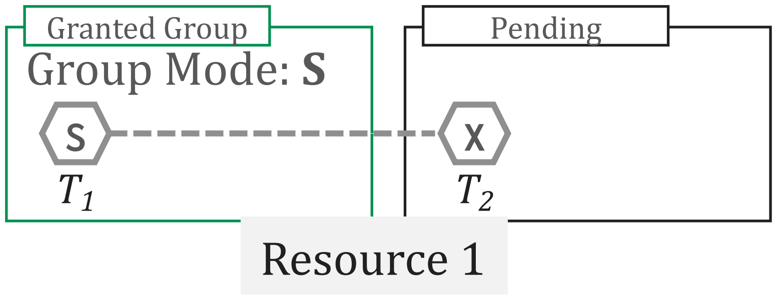


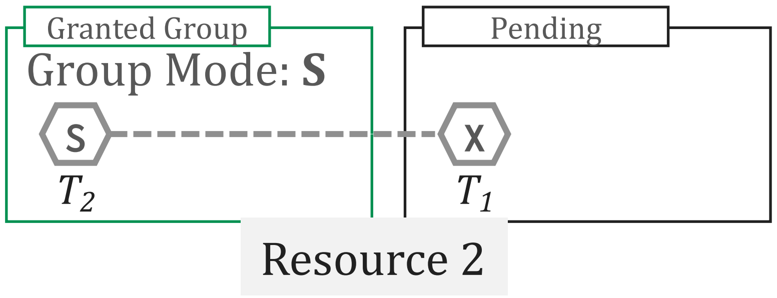
***Waits-For* Graph**

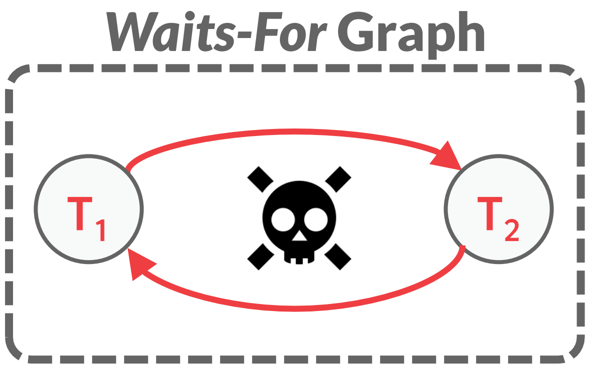


Deadlocks

- We can now have deadlocks.
- Need a mechanism to either detect deadlocks, or prevent deadlocks.
- Deadlock detection: Construct and periodically examine the wait-for-graph. Pick a victim (oldest tnx or newest txn) to break the deadlock. Abort the victim txn and restart it (perhaps after some sleep/delay).
- Deadlock prevention: Abort a txn as soon as it waits for another txn.
  - Wound-wait or wait-die (see the intro DB class for details).



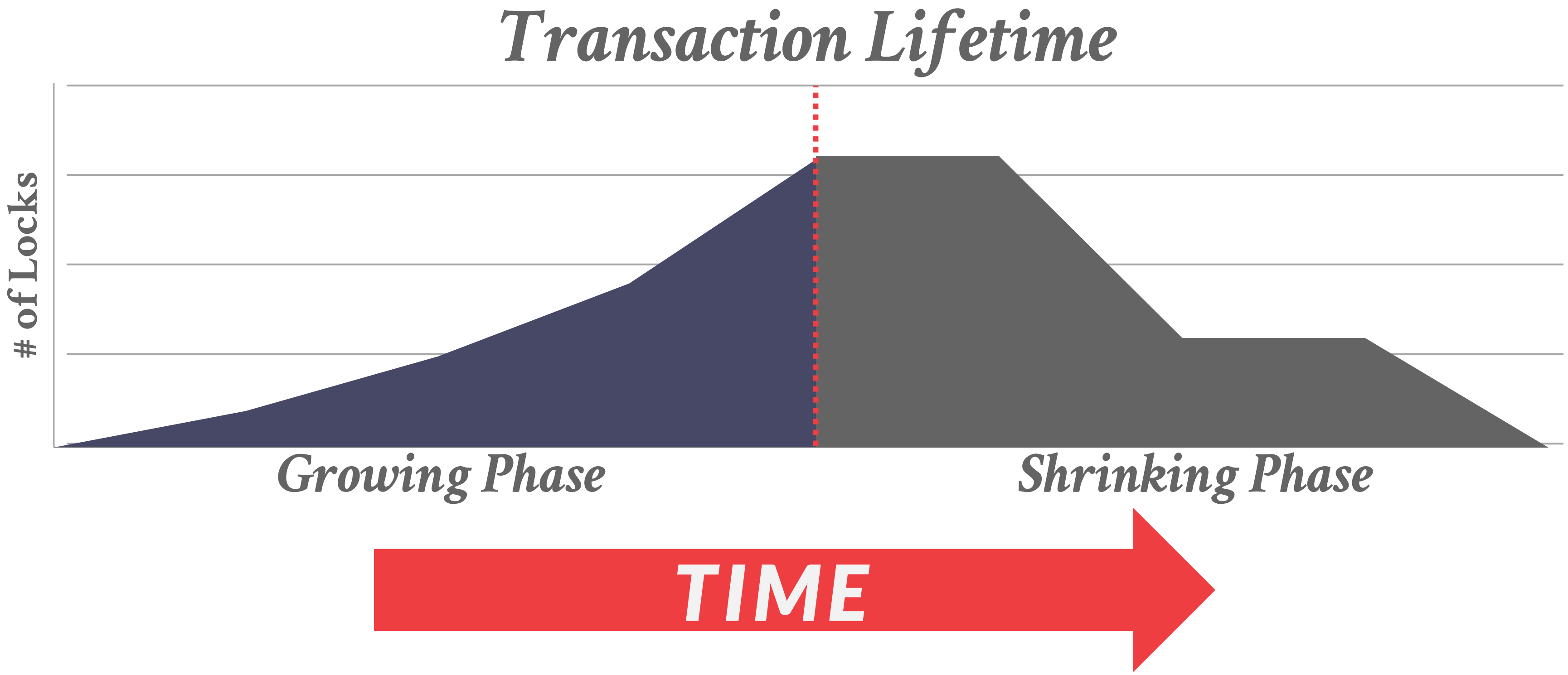




Revisit your introduction to DB class material if needed.

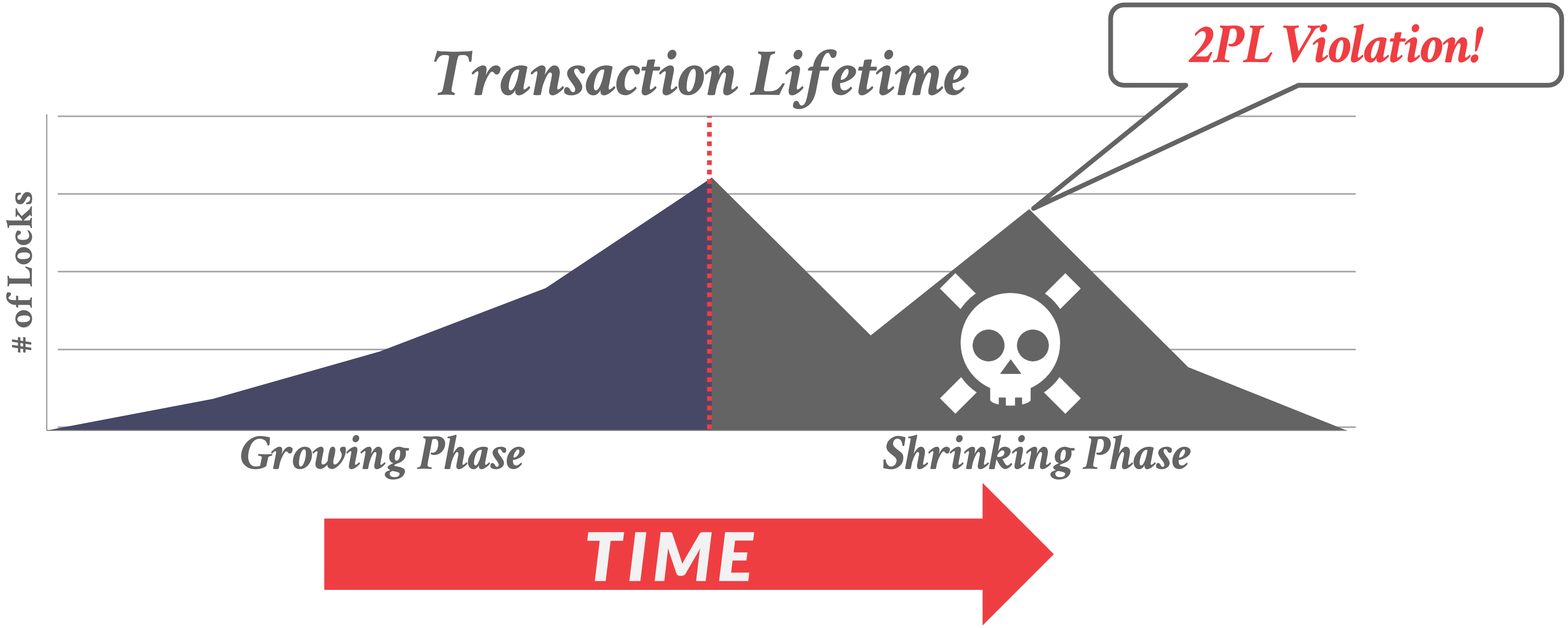
Two Phase Locking (2PL)

- Txn has 2 phases: a growing (acquire lock phase) and a subsequent drop lock phase.



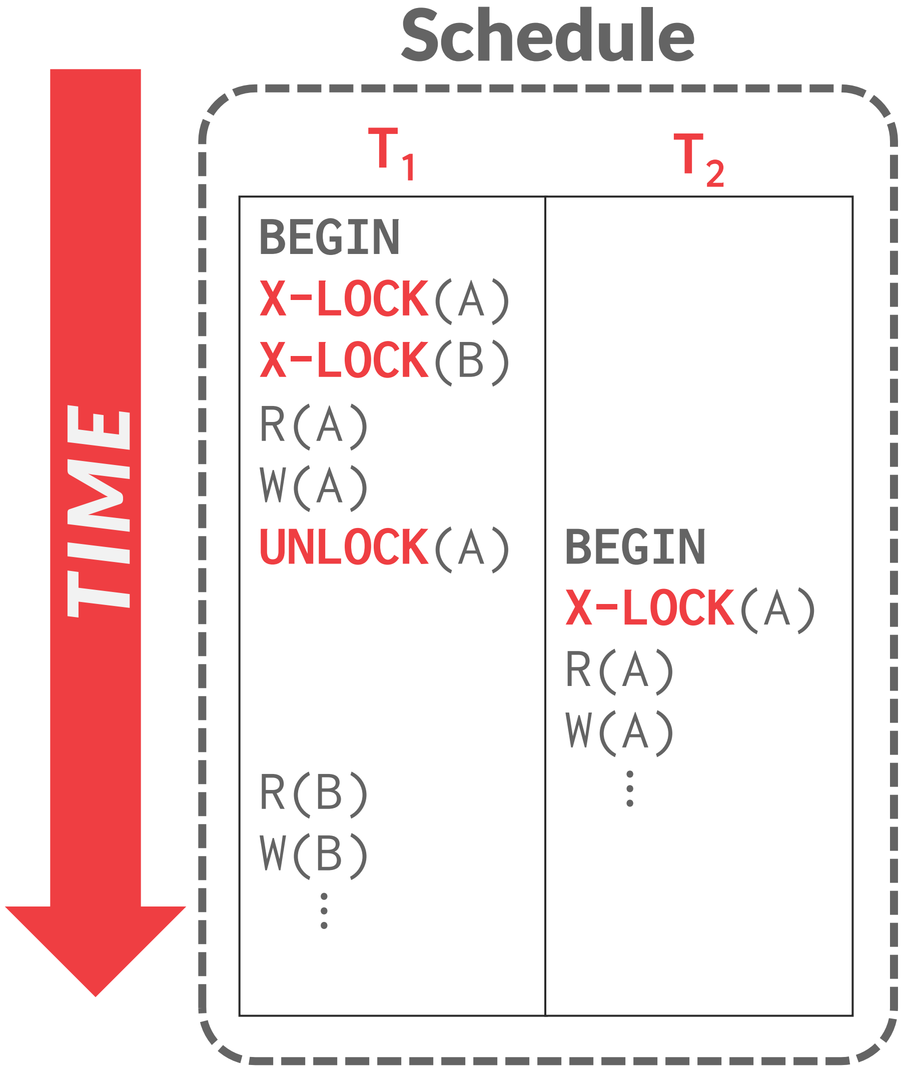
TWO PHASE LOCKING (2PL)

- Txn has 2 phases: a growing (acquire lock phase) and a subsequent drop lock phase.
- Can't acquire a lock after the first lock is released.



Revisit your introduction to DB class material if needed.

2PL Permits Cascading Aborts

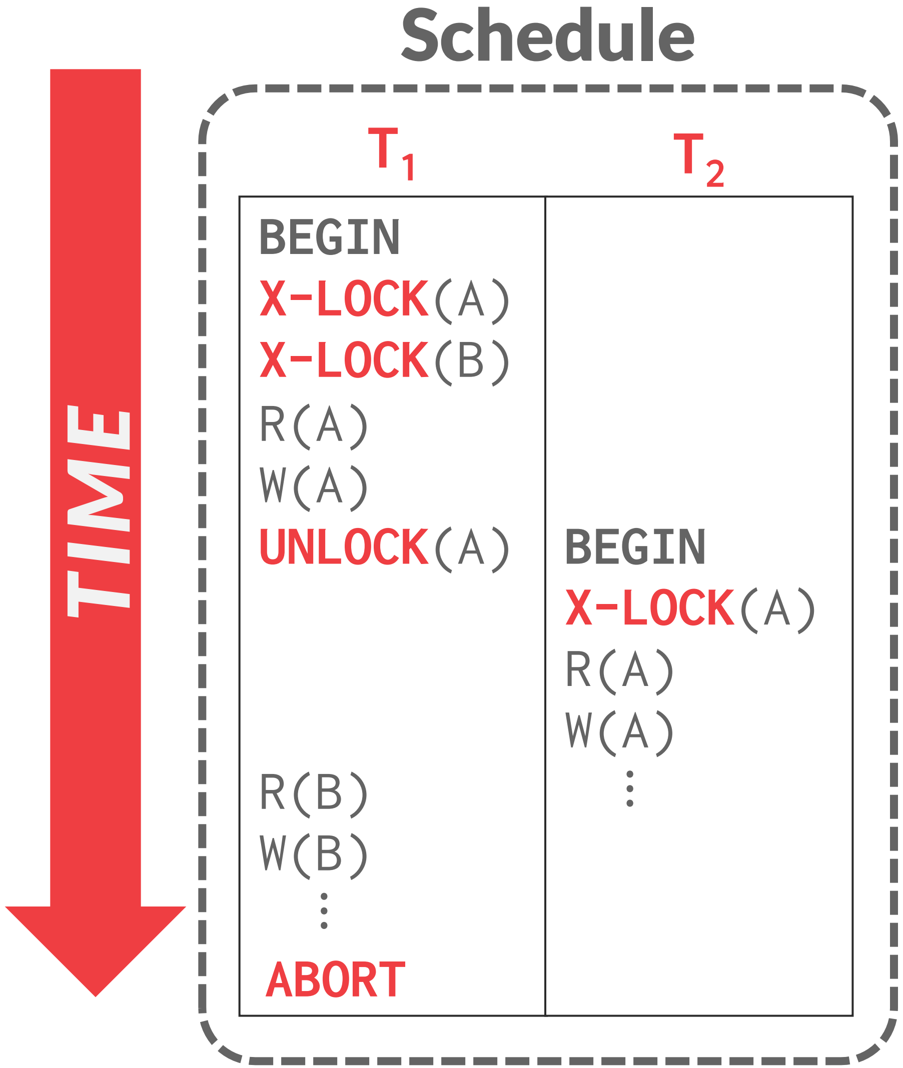


Schedule

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)	
⋮	

Revisit your introduction to DB class material if needed.

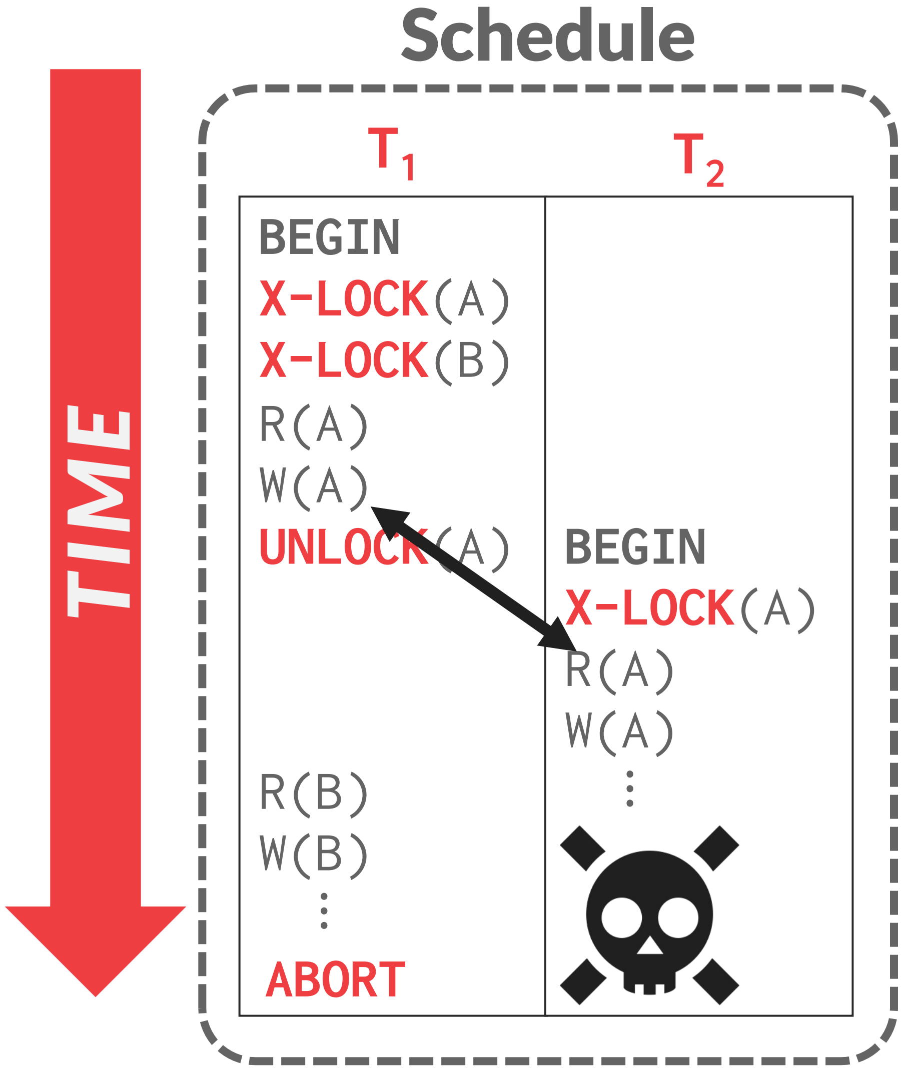
2PL Permits Cascading Aborts



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	

Revisit your introduction to DB class material if needed.

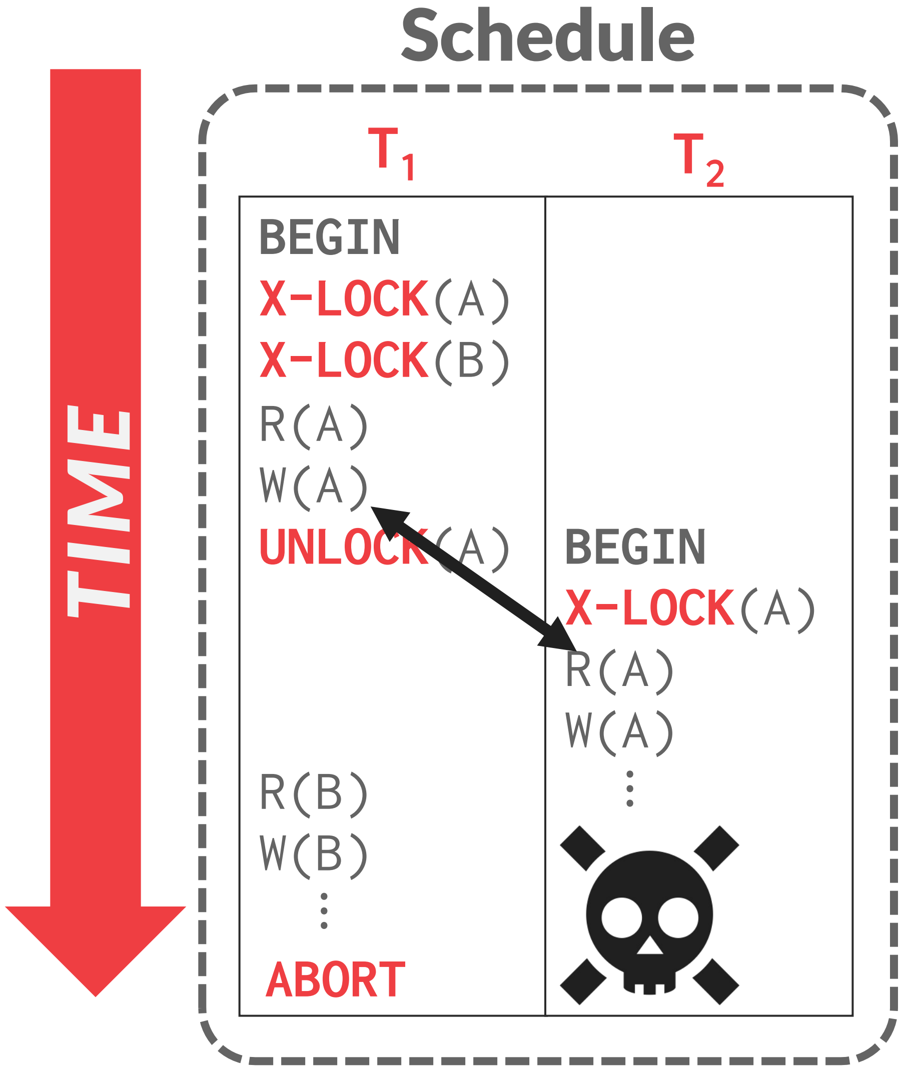
2PL Permits Cascading Aborts



T1	T2
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	

Revisit your introduction to DB class material if needed.

2PL Permits Cascading Aborts



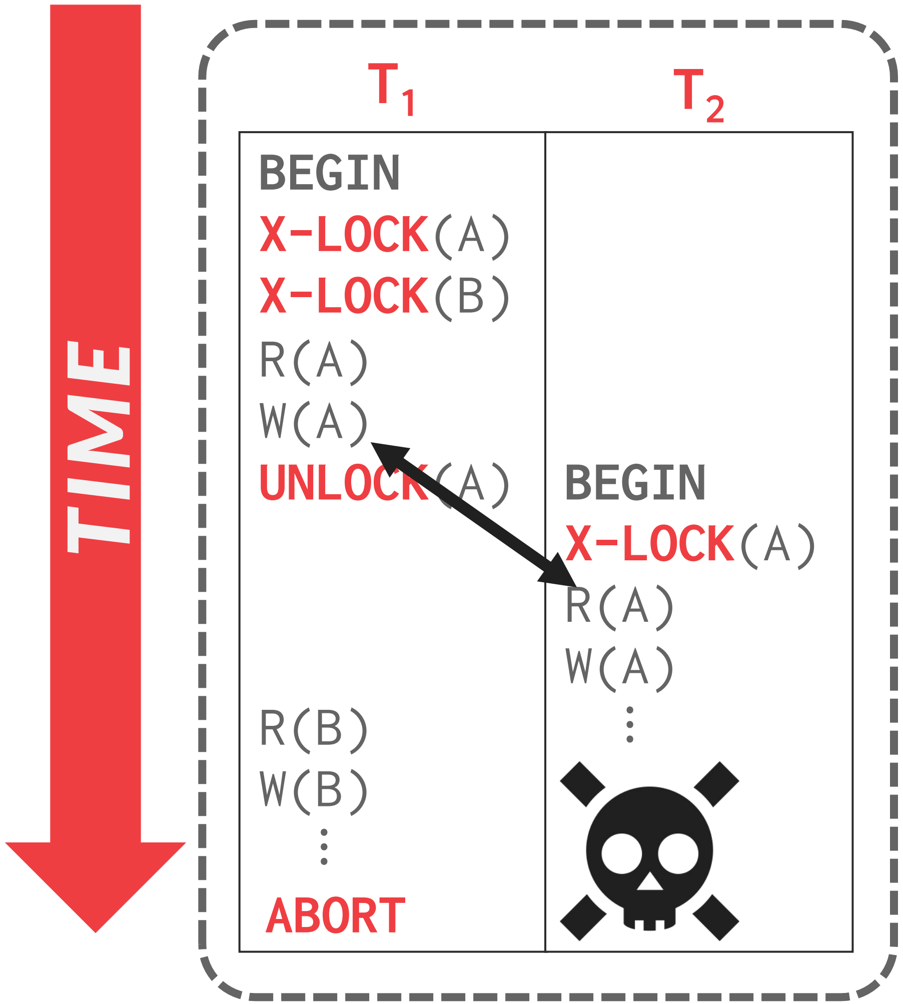
Schedule

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.

2PL Permits Cascading Aborts

Schedule

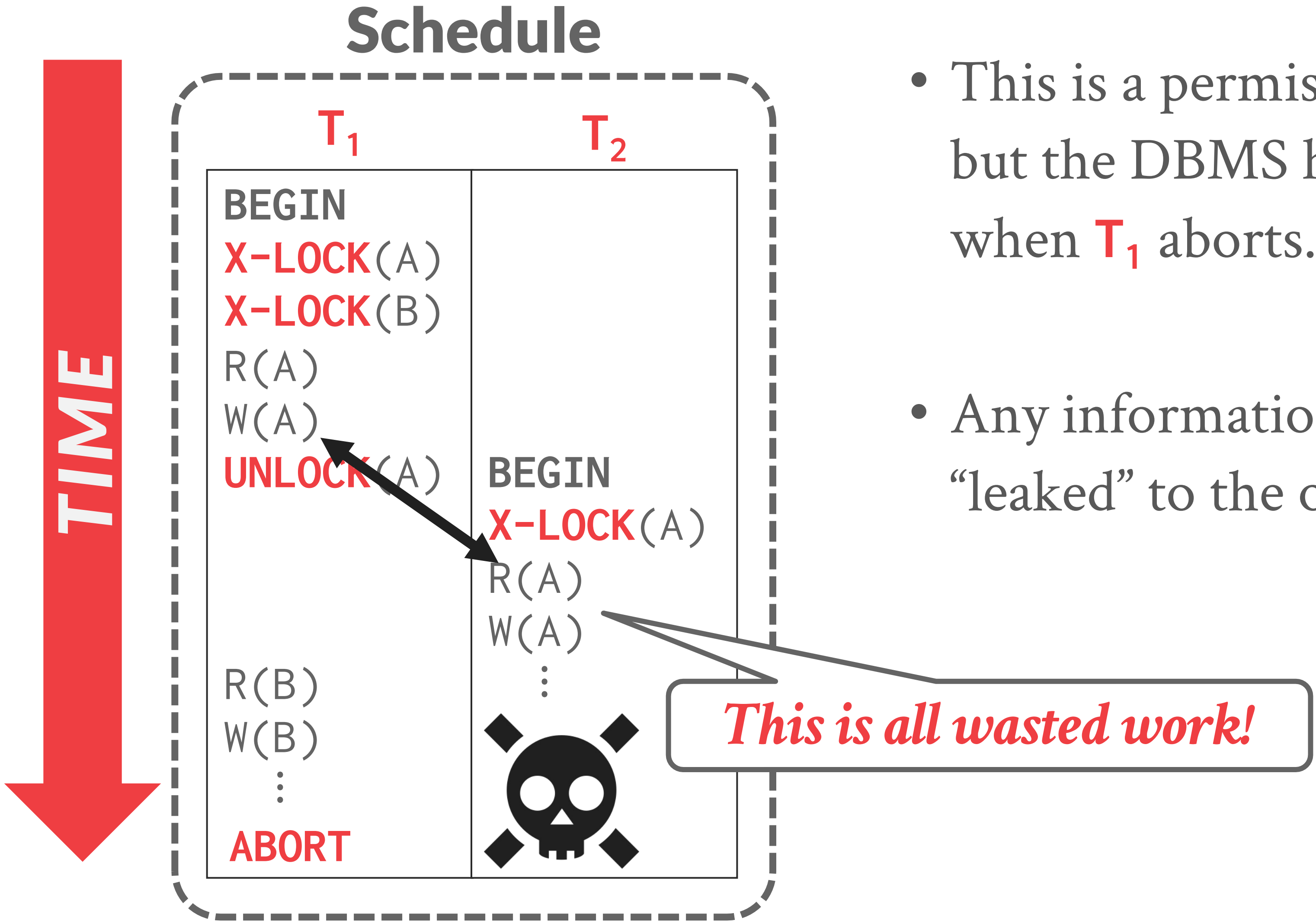


T_1	T_2
BEGIN	
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	BEGIN
	X-LOCK(A)
	R(A)
	W(A)
	⋮
R(B)	
W(B)	
⋮	
ABORT	

- 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.

Revisit your introduction to DB class material if needed.

2PL Permits 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	

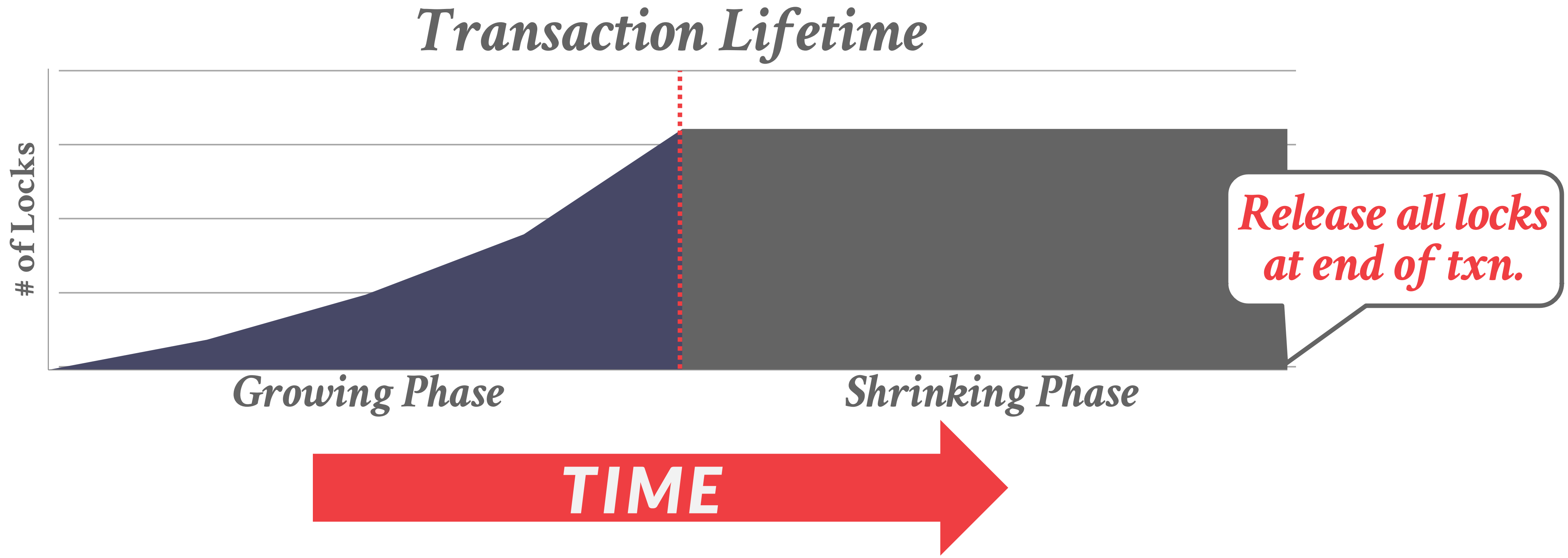
- This is a permissible schedule in 2PL, but the DBMS has to also abort T_2 when T_1 aborts.

- Any information about T_1 cannot be “leaked” to the outside world.

STRONG STRICT TWO PHASE LOCKING

Allows only “conflict serializable” schedules.
ACA – Avoids Cascading Aborts.

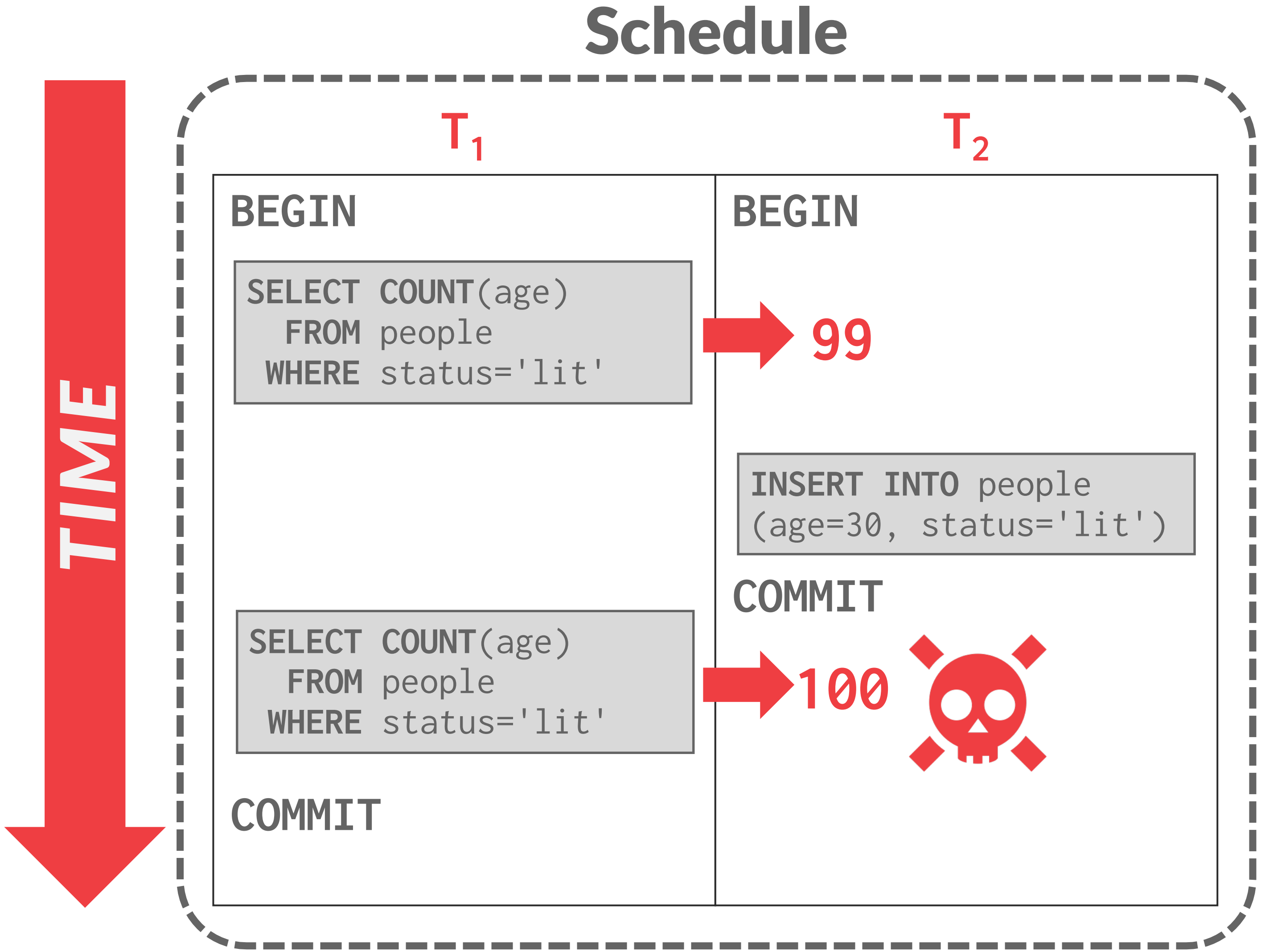
- Txn has 2 phases: a growing (acquire lock phase) and a subsequent drop lock phase.
- Can’t acquire a lock after the first lock is released.
- Txn holds all locks till the end (abort or commit) and drops them then.



Revisit your introduction to DB class material if needed.

PHANTOMS

Lock key ranges in the B-tree to prevent phantoms, aka. predicate locking.



T1:

```
BEGIN
SELECT COUNT(age)
  FROM people
 WHERE status='lit'

SELECT COUNT(age)
  FROM people
 WHERE status='lit'
COMMIT
```

T2:

```
BEGIN
INSERT INTO people
(age=30, status='lit')
COMMIT
```

```
CREATE TABLE people (
  id SERIAL,
  name VARCHAR,
  age INT,
  status VARCHAR
);
```

Revisit your introduction to DB class material if needed.

WEAKER LEVELS OF ISOLATION

- Want to allow various "degrees of consistency" in the same system and concurrent txns.
- Some txns may be ok with lower levels of consistency; e.g., statistics update query.

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

Revisit your introduction to DB class material if needed.

Isolation Levels

- **SERIALIZABLE**: Obtain all locks first; plus index locks, plus strong strict 2PL.
- **REPEATABLE READS**: Same as above, but no index locks.
- **READ COMMITTED**: Same as above, but **S** locks are released immediately.
- **READ UNCOMMITTED**: Same as above but allows dirty reads (no **S** locks).

Part of SQL, and you can explicitly set the isolation levels.

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

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

Isolation Levels and Anomalies

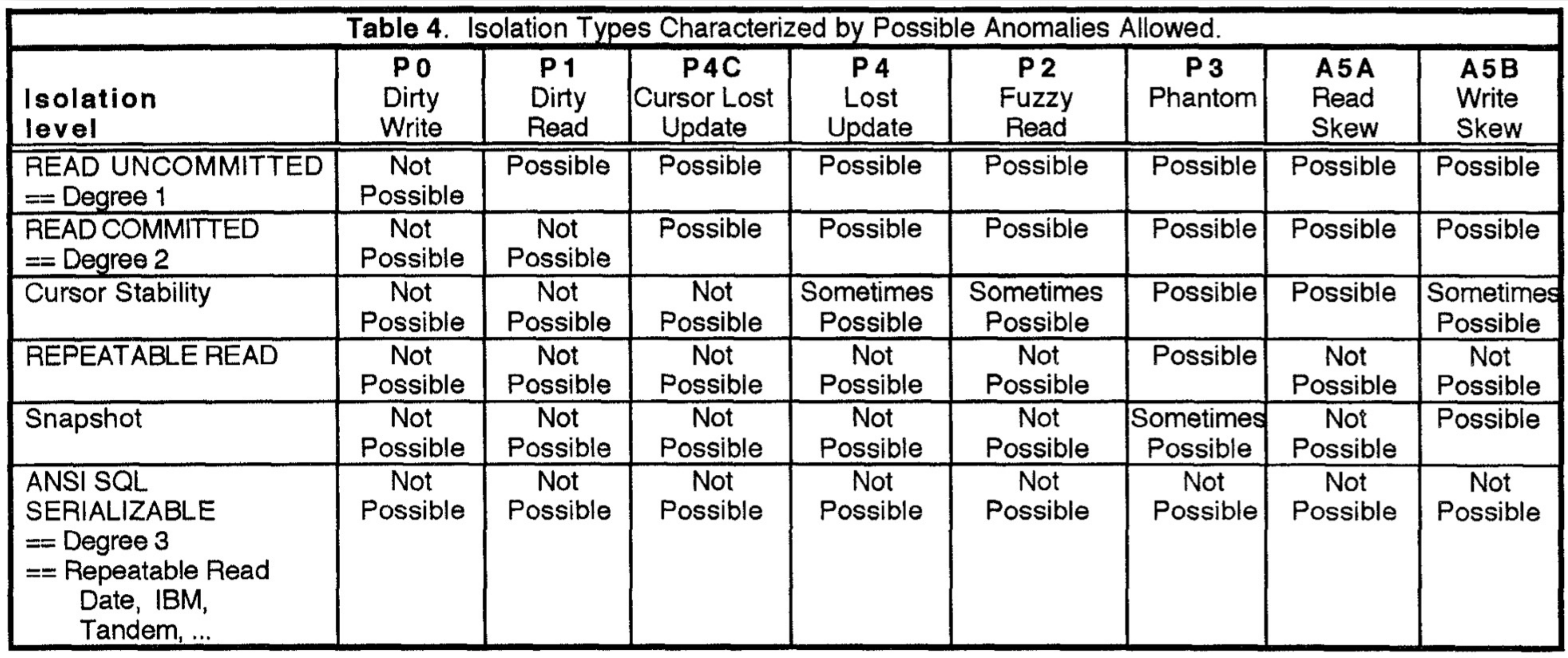


Table 4. Isolation Types Characterized by Possible Anomalies Allowed.

Isolation level	P0 Dirty Write	P1 Dirty Read	P4C Cursor Lost Update	P4 Lost Update	P2 Fuzzy Read	P3 Phantom	A5A Read Skew	A5B Write Skew
READ UNCOMMITTED == Degree 1	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible	Possible
READ COMMITTED == Degree 2	Not Possible	Not Possible	Possible	Possible	Possible	Possible	Possible	Possible
Cursor Stability	Not Possible	Not Possible	Not Possible	Sometimes Possible	Sometimes Possible	Possible	Possible	Sometimes Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Possible	Not Possible	Not Possible
Snapshot	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Sometimes Possible	Not Possible	Possible
ANSI SQL SERIALIZABLE == Degree 3 == Repeatable Read Date, IBM, Tandem, ...	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible	Not Possible

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil:
A Critique of ANSI SQL Isolation Levels. SIGMOD 1995

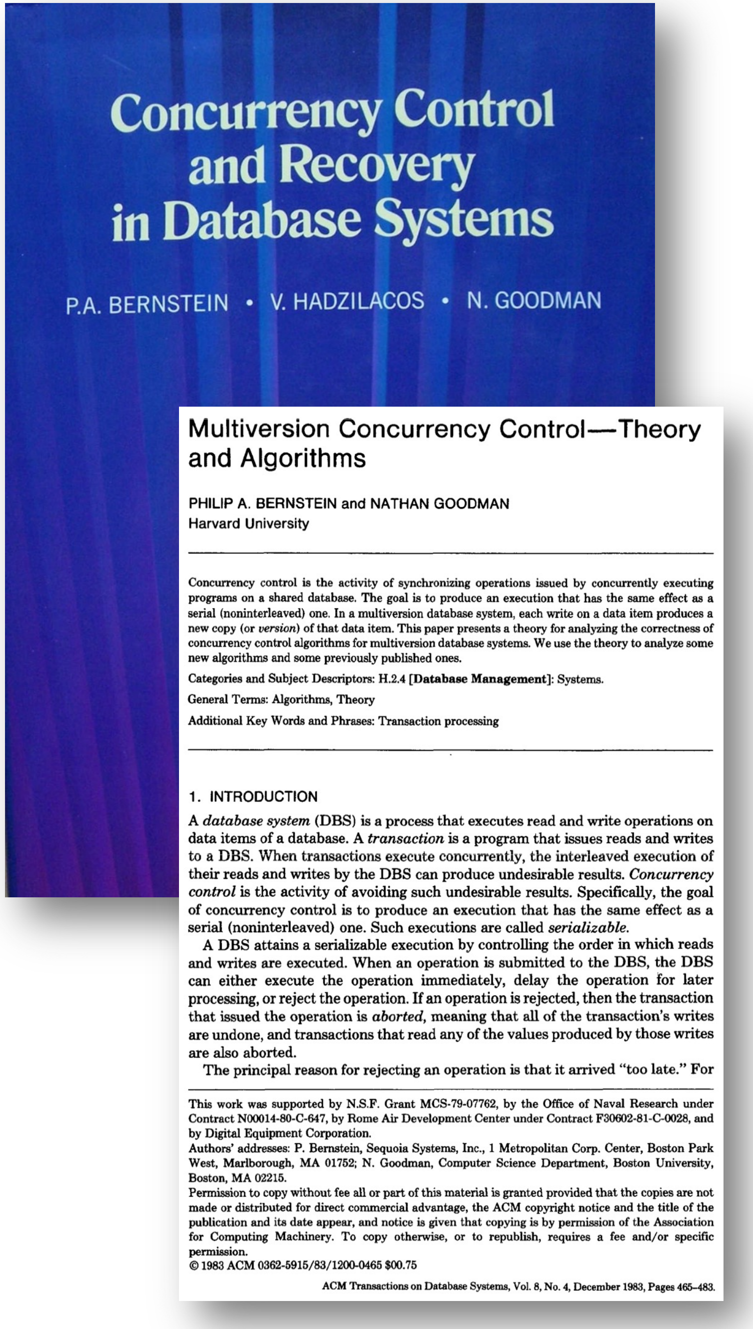
Revisit your introduction to DB class material if needed.

ISOLATION LEVELS

	Default	*Maximum*
Actian Ingres	SERIALIZABLE	SERIALIZABLE
IBM DB2	CURSOR STABILITY	SERIALIZABLE
CockroachDB	SERIALIZABLE	SERIALIZABLE
Google Spanner	STRICT SERIALIZABLE	STRICT SERIALIZABLE
MSFT SQL Server	READ COMMITTED	SERIALIZABLE
MySQL	REPEATABLE READS	SERIALIZABLE
Oracle	READ COMMITTED	SNAPSHOT ISOLATION
PostgreSQL	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
VoltDB	SERIALIZABLE	SERIALIZABLE
YugaByte	SNAPSHOT ISOLATION	SERIALIZABLE

SUMMARY AND OUTLOOK

- This paper directionally set the way concurrency control is implemented in a data platforms (granularity of locking and degrees of consistency), and influenced the SQL standard.
- For the longest time, many database platforms only used pure locking-based protocols for concurrency control.
- But, there were other approaches, including OCC.
- Also, MVCC influences that concurrency protocol. MVCC is about creating versions of data on an update rather than update-in-place and can be used with Locking (or OCC).
    - Revise MVCC from your intro to DB class if you have forgotten it.
- Lot of different way to do concurrency control today with various tradeoffs in the "degree of consistency" and performance.



Concurrency Control and Recovery in Database Systems

P.A. BERNSTEIN • V. HADZILACOS • N. GOODMAN

Multiversion Concurrency Control—Theory and Algorithms

PHILIP A. BERNSTEIN and NATHAN GOODMAN
Harvard University

Concurrency control is the activity of synchronizing operations issued by concurrently executing programs on a shared database. The goal is to produce an execution that has the same effect as a serial (noninterleaved) one. In a multiversion database system, each write on a data item produces a new copy (or *version*) of that data item. This paper presents a theory for analyzing the correctness of concurrency control algorithms for multiversion database systems. We use the theory to analyze some new algorithms and some previously published ones.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems.

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Transaction processing

1. INTRODUCTION

A *database system* (DBS) is a process that executes read and write operations on data items of a database. A *transaction* is a program that issues reads and writes to a DBS. When transactions execute concurrently, the interleaved execution of their reads and writes by the DBS can produce undesirable results. *Concurrency control* is the activity of avoiding such undesirable results. Specifically, the goal of concurrency control is to produce an execution that has the same effect as a serial (noninterleaved) one. Such executions are called *serializable*.

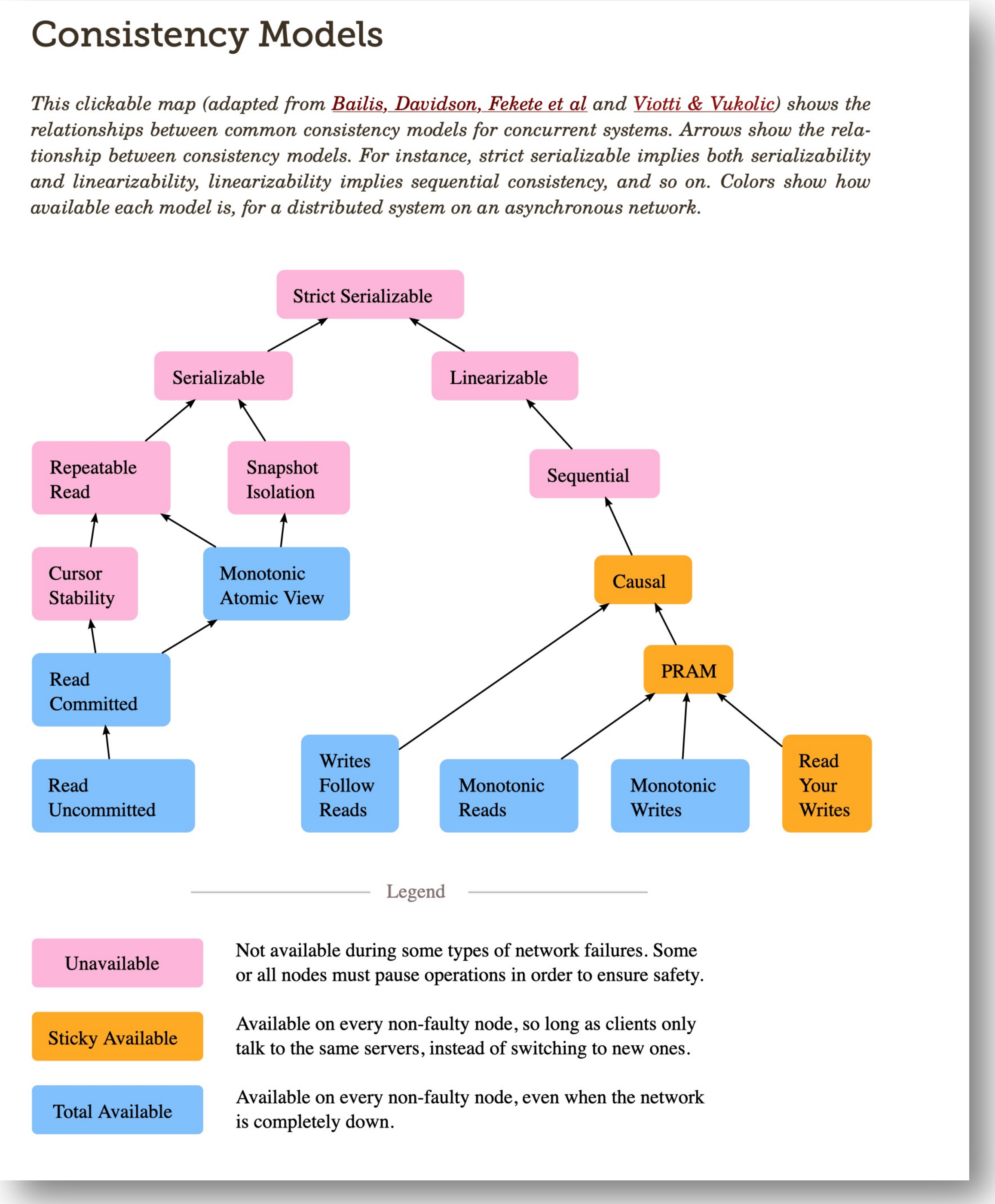
A DBS attains a serializable execution by controlling the order in which reads and writes are executed. When an operation is submitted to the DBS, the DBS can either execute the operation immediately, delay the operation for later processing, or reject the operation. If an operation is rejected, then the transaction that issued the operation is *aborted*, meaning that all of the transaction's writes are undone, and transactions that read any of the values produced by those writes are also aborted.

The principal reason for rejecting an operation is that it arrived "too late." For

This work was supported by N.S.F. Grant MCS-79-07762, by the Office of Naval Research under Contract N00014-80-C-647, by Rome Air Development Center under Contract F30602-81-C-0028, and by Digital Equipment Corporation.

Authors' addresses: P. Bernstein, Sequoia Systems, Inc., 1 Metropolitan Corp. Center, Boston Park West, Marlborough, MA 01752; N. Goodman, Computer Science Department, Boston University, Boston, MA 02215.

ACM Transactions on Database Systems, Vol. 8, No. 4, December 1983, Pages 465–483.



Consistency Models

This clickable map (adapted from Bailis, Davidson, Fekete et al and Viotti & Vukolic) shows the relationships between common consistency models for concurrent systems. Arrows show the relationship between consistency models. For instance, strict serializable implies both serializability and linearizability, linearizability implies sequential consistency, and so on. Colors show how available each model is, for a distributed system on an asynchronous network.

Legend

Unavailable — Not available during some types of network failures. Some or all nodes must pause operations in order to ensure safety.

Sticky Available — Available on every non-faulty node, so long as clients only talk to the same servers, instead of switching to new ones.

Total Available — Available on every non-faulty node, even when the network is completely down.

https://jepsen.io/consistency