Transactions & Concurrency Control

CSC443H1 Database System Technology

Niv Dayan

Please do the course evaluation!

A DB has a notion of consistency, defined by the user

A DB has a notion of consistency, defined by the user

Sum of money in a system should stay constant

A DB has a notion of consistency, defined by the user

Sum of money in a system should stay constant

An account balance cannot drop below zero

A DB has a notion of consistency, defined by the user

Sum of money in a system should stay constant

An account balance cannot drop below zero

A user must have a valid SIN

A DB has a notion of consistency, defined by the user

Sum of money in a system should stay constant

An account balance cannot drop below zero

A user must have a valid SIN

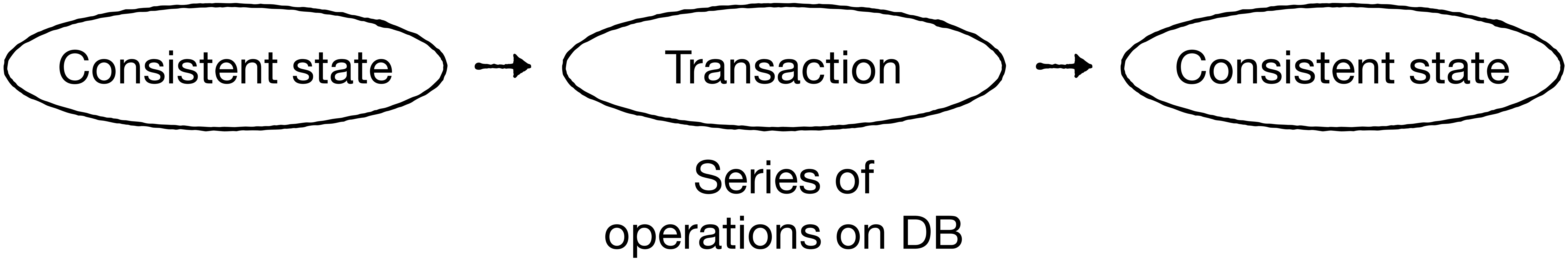
sum(balance) = X

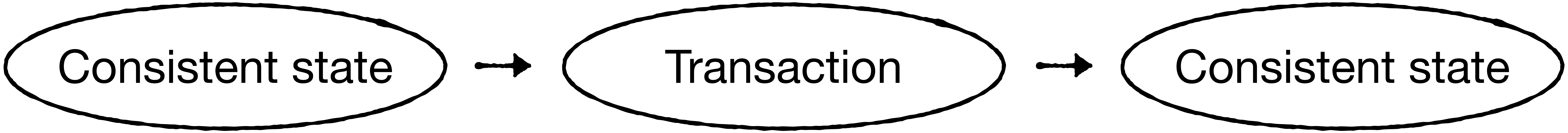
balance ≥ 0

SIN ≠ null

These can be set as integrity constraints

To maintain consistency, database APIs expose the concept of a transaction





Example: Bank transfer

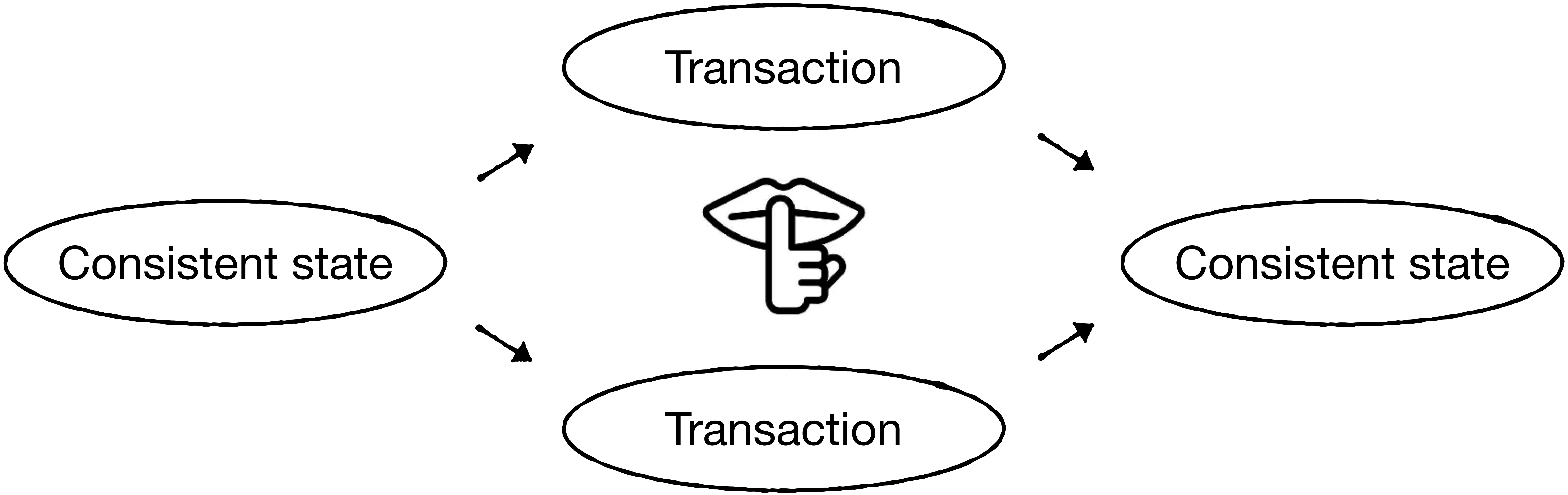
A = A - 100
B = B + 100

Consistent state → Transaction → Consistent state

Example: Bank transfer

A = A - 100
B = B + 100

Both operations must succeed or fail for the system to remain in a consistent state



Transactions do not interact with each other (i.e., by exchanging messages). A transaction is oblivious to all other ongoing transactions.

Transaction API

Begin Transaction

Some commands (e.g., in SQL)

Commit or Abort

Example: Money withdrawal

Begin Transaction

b = Select balance from accounts where id = x

If b ≤ 100

    Abort

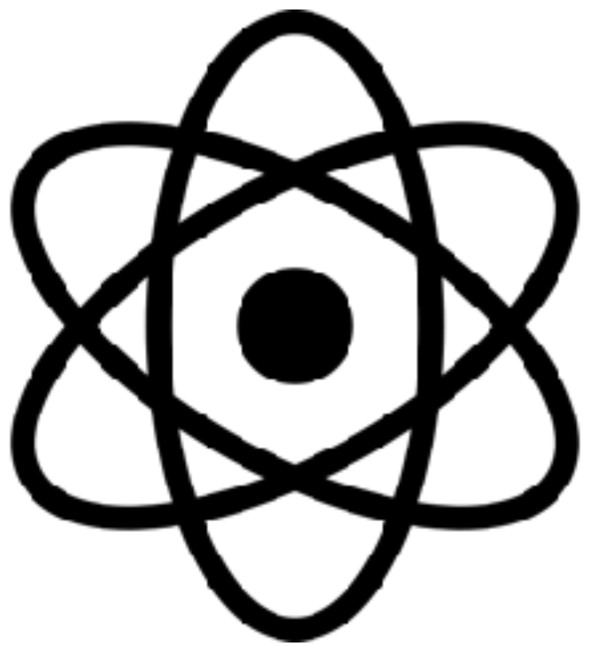
Else

    Update accounts set balance = b - 100 where id = x

    Commit

ACID semantics for Transactions

Atomicity



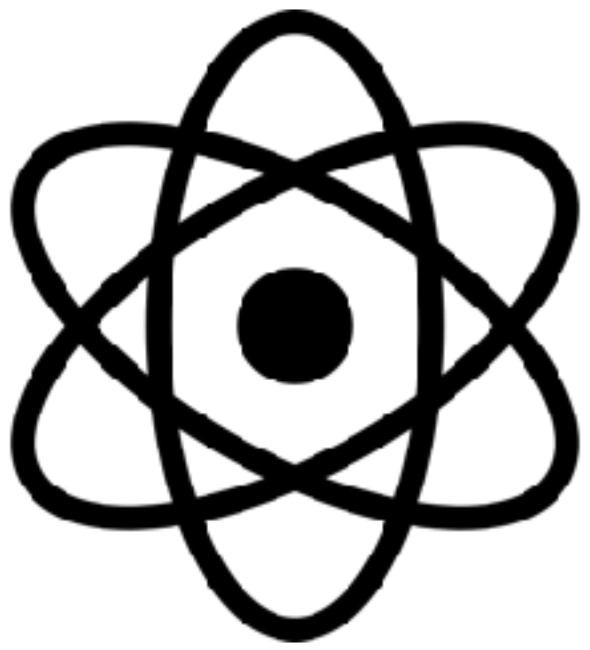
Consistency

Isolation

Durability

ACID

Atomicity



All or nothing

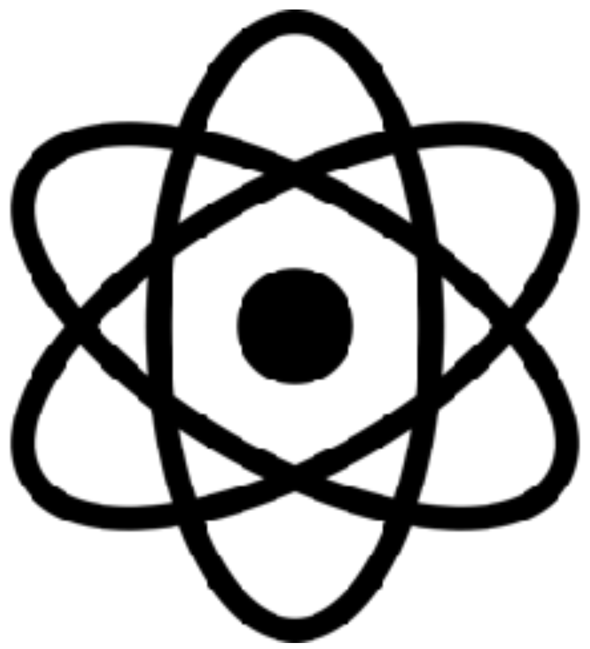
Consistency

Isolation

Durability

ACID

Atomicity



All or nothing

Consistency

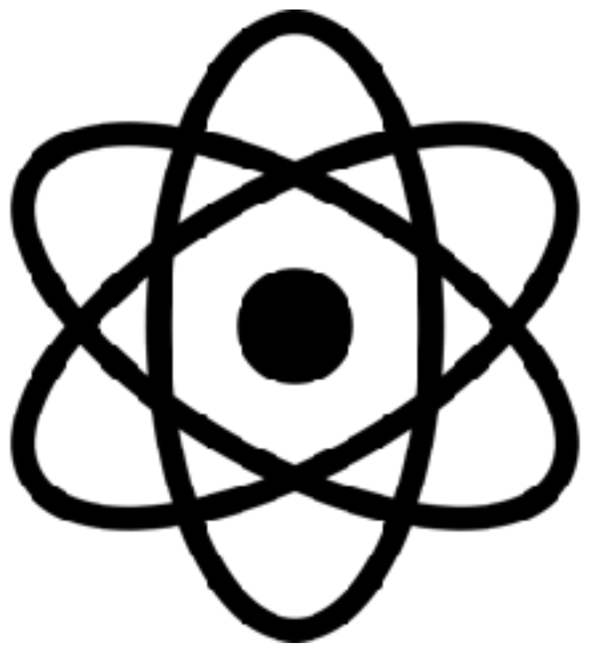
Transition across consistent states

Isolation

Durability

ACID

Atomicity



All or nothing

Consistency

Transition across consistent states

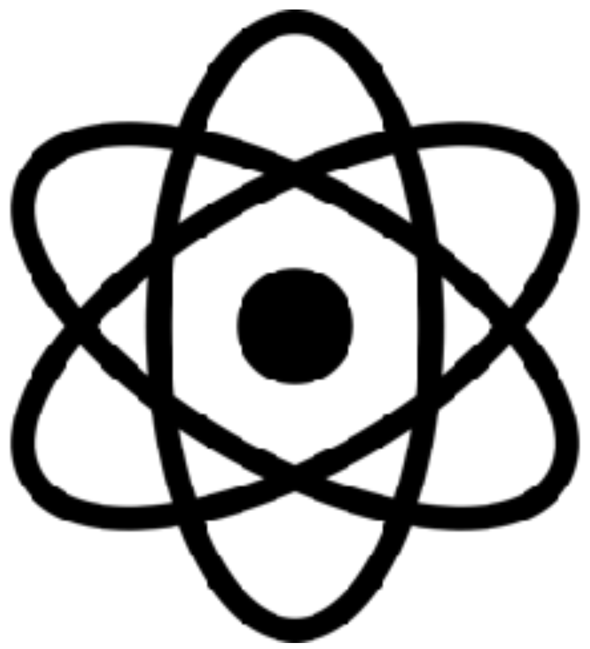
Isolation

Not corrupted by concurrency

Durability

ACID

Atomicity



All or nothing

Consistency

Transition across consistent states

Isolation

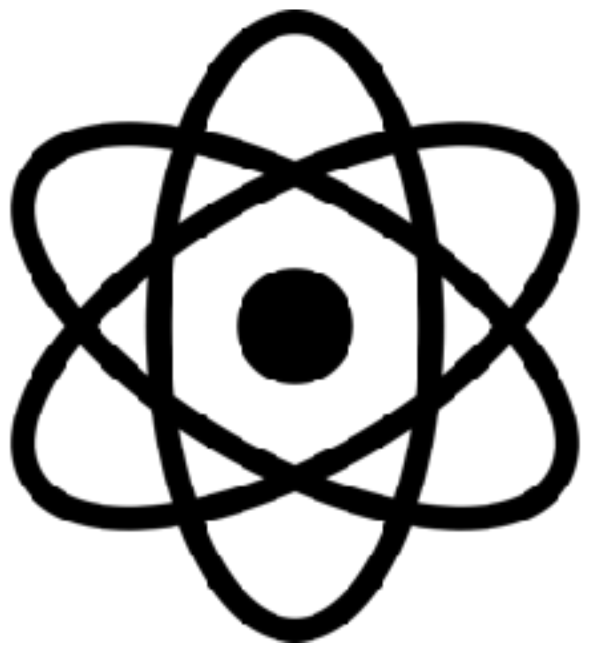
Not corrupted by concurrency

Durability

Recover from failure

Who is responsible for what?

Atomicity



DB

Consistency

User

Isolation

DB

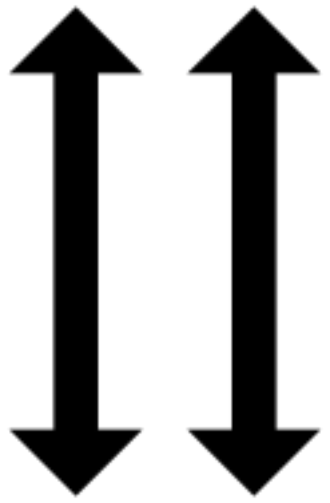
Durability

DB

What Endangers Consistency?

System Failure

Concurrency



System Failure

Power failure

Hardware failure

Data center failure

System Failure

Power failure

Example: Bank transfer

A = A - 100
Power fails
B = B + 100

System Failure

Power failure

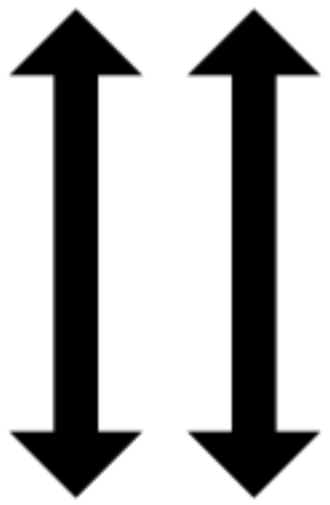
Example: Bank transfer

A = A - 100
Power fails
B = B + 100

Power fails before transaction finishes, leaving the system in an inconsistent state

System Failure

Concurrency



Concurrency

Concurrent transactions can result in an inconsistent state

Concurrency

Concurrent transactions can result in an inconsistent state

Can you think of examples?

Concurrency

Concurrent transactions can result in an inconsistent state

Example 1

Interest & Payments

Example 2

Updates & statistics

Example 1

T1: Interest payment

$$A = A \cdot 1.05$$
$$B = B \cdot 1.05$$

T2: Payments

$$A = A - 100$$
$$B = B + 100$$

Example 1

Time

T1	T2
$A = A \cdot 1.05$	
$B = B \cdot 1.05$	
Commit	
	$A = A - 100$
	$B = B + 100$
	Commit

Example 1

Initialization: **A=1000, B=1000**

T1	T2
$A = A \cdot 1.05$	
$B = B \cdot 1.05$	
Commit	
	$A = A - 100$
	$B = B + 100$
	Commit

Time

Valid Outcome 1: A = 950, B = 1150

Example 1

Initialization: A=1000, B=1000

Time	T1	T2
		A = A - 100
		B = B + 100
		Commit
	A = A · 1.05	
	B = B · 1.05	
	Commit	

Valid Outcome 2: A = 945, B = 1155

Valid Outcome 1: A = 950, B = 1150

Example 1

Initialization: A=1000, B=1000

Time	T1	T2
	$A = A \cdot 1.05$	
		$A = A - 100$
		$B = B + 100$
	$B = B \cdot 1.05$	
	Commit	Commit

Invalid Outcome: A = 950, B = 1155

Valid Outcome 2: A = 945, B = 1155

Valid Outcome 1: A = 950, B = 1150

Example 1

Initialization: A=1000, B=1000

Time

T1	T2
$A = A \cdot 1.05$	
	$A = A - 100$
	$B = B + 100$
$B = B \cdot 1.05$	
Commit	Commit

Dirty read: T2 reads a modified but uncommitted data item from T1

Invalid Outcome: A = 950, B = 1155

Valid Outcome 2: A = 945, B = 1155

Valid Outcome 1: A = 950, B = 1150

Concurrency

Concurrent transactions can result in an inconsistent state

Example 1

Payments & Interest

Example 2

Updates & Statistics

Example 2

T1: Account Updates

Balance += X

T2: Statistics Reporting

count(# accounts)

sum(account balances)

avg(account balances)

Example 2

T1: Account Updates

bob_checking += X

T2: Statistics Reporting

count(# accounts) where user = “bob”

sum(account balances) where user = “bob”

avg(account balances) where user = “bob”

Example 2

T1: Account Updates

bob_checking += X

T2: Statistics Reporting

count(# accounts) where user = “bob”

sum(account balances) where user = “bob”

avg(account balances) where user = “bob”

What can go wrong?

Example 2

Time

T1	T2
checking += X	
	count(# accounts)
	sum(account balances)
	avg(account balances)

Example 2

Time

T1	T2
	count(# accounts)
	sum(account balances)
	avg(account balances)
checking += X	

Example 2

Time

T1	T2
	count(# accounts)
	sum(account balances)
checking += X	
	avg(account balances)

Problem?

Example 2

Time

T1	T2
	count(# accounts)
	sum(account balances)
checking += X	
	avg(account balances)

Problem? **Inconsistent reporting: avg ≠ sum / count**

Example 2

Time

T1	T2
	count(# accounts)
	sum(account balances)
checking += X	
	avg(account balances)

More broadly, this is an **unrepeatable read anomaly:** subsequent reads of the same data are inconsistent as the data was changed in-between

Concurrency

Concurrent transactions can result in an inconsistent state

Example 1

Payments & Interest

Example 2

Updates & Statistics

Problem:

Dirty reads

Unrepeatable reads

Simplest solution: lock whole DB for any modification

Problems?

Simplest solution: lock whole DB for any modification

Problems: terrible for performance

Simplest solution: lock whole DB for any modification

Problems: terrible for performance

While one transaction waits for a storage I/O, another should be able to use the CPU

Simplest solution: lock whole DB for any modification

Problems: terrible for performance

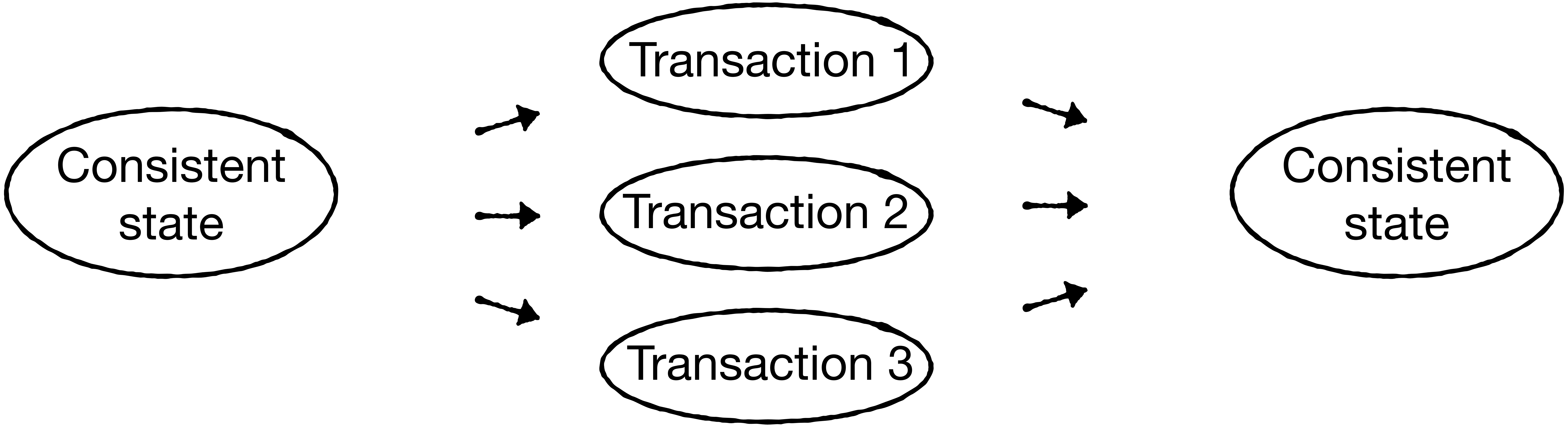
While one transaction waits for a storage I/O, another should be able to use the CPU

A short transaction should not have to wait until a long transaction completes

A good compromise: Serializability

A good compromise: Serializability

Transactions can be concurrent but must result in a consistent state



A good compromise: Serializability

Transactions can be concurrent but must result in a consistent state

Any given state once a transaction commits should have been achievable through a serial execution of all committed transactions thus far

A good compromise: Serializability

Transactions can be concurrent but must result in a consistent state

Equivalence to any serial execution is considered correct

How to achieve concurrency & serializability?

How to achieve concurrency & serializability? **Strict Two-Phase Locking**

How to achieve concurrency & serializability? **Strict Two-Phase Locking**

A transaction is divided into two phases

Phase 1 Phase 2

How to achieve concurrency & serializability? **Strict Two-Phase Locking**

A transaction is divided into two phases

Phase 1

Lock a data item (e.g., row) when it is accessed for the first time

Phase 2

How to achieve concurrency & serializability? **Strict Two-Phase Locking**

A transaction is divided into two phases

Phase 1

Lock a data item (e.g., row) when it is accessed for the first time

Reads take shared locks

Writes take exclusive locks

Phase 2

How to achieve concurrency & serializability? **Strict Two-Phase Locking**

A transaction is divided into two phases

Phase 1

Lock a data item (e.g., row) when it is accessed for the first time

Reads take shared locks

Writes take exclusive locks

Phase 2

Release all locks during commit

How to achieve concurrency & serializability? **Strict Two-Phase Locking**

A transaction is divided into two phases

Phase 1

Lock a data item (e.g., row) when it is accessed for the first time

Phase 2

Release all locks during commit

Invariant: a data item that has been accessed by this transaction cannot be modified by another transaction until this transaction commits

Let’s fix our running examples

Example 1

Interest & Payments

Example 2

Updates & Statistics

Problems: Dirty reads | Unrepeatable reads

Example 1

T1: Interest payment

$$A = A \cdot 1.05$$
$$B = B \cdot 1.05$$

T2: Payments

$$A = A - 100$$
$$B = B + 100$$

Example 1

Time ↓

T1	T2
$A = A \cdot 1.05$	
	$A = A - 100$
	$B = B + 100$
$B = B \cdot 1.05$	
Commit	Commit

Example 1

Time	T1	T2
	ex_lock(A)	
	$A = A \cdot 1.05$	
	ex_lock(B)	
	$B = B \cdot 1.05$	
	Commit	
		ex_lock(A)
		$A = A - 100$
		ex_lock(B)
		$B = B + 100$
		Commit

Exclusive locks ensure T2 cannot modify A until T1 commits

Let’s fix our running examples

Example 1

Interest & Payments

Example 2

Updates & Statistics

Problems:	Dirty reads	Unrepeatable reads

Fixing Example 2

Time

T1	T2
	count(# accounts)
	sum(account balances)
checking += X	
	avg(account balances)

Fixing Example 2

Time	T1	T2
		count(# accounts)
		sum(account balances)
	checking += X	
		avg(account balances)

Take shared locks on each row it reads

Fixing Example 2

Time

T1	T2
	count(# accounts)
	sum(account balances)
Request ex_lock(X)	
	avg(account balances)
	Commit
checking += X	

T1 has to wait

Fixing Example 2

Time

T1	T2
	count(# accounts)
	sum(account balances)
Request ex_lock(X)	
	avg(account balances)
	Commit
checking += X	

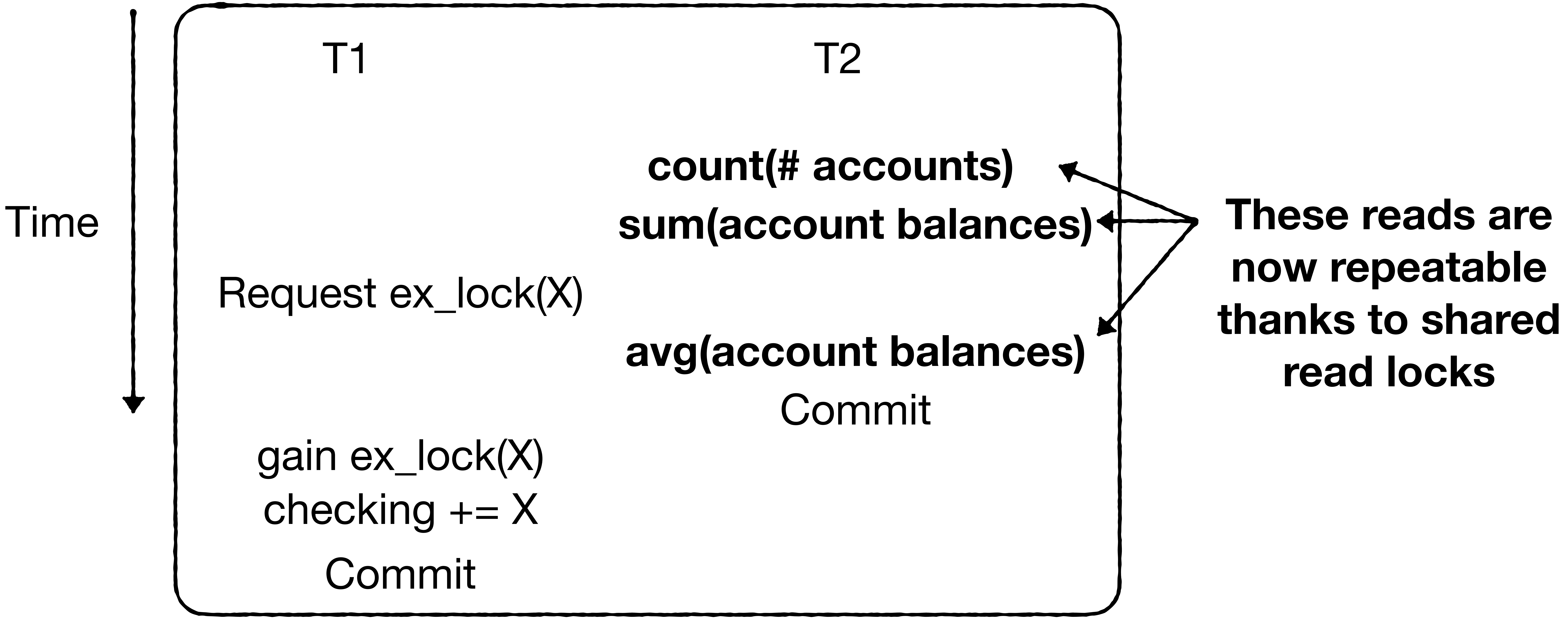
T2 releases locks

Fixing Example 2

Time

T1	T2	
	count(# accounts)	
	sum(account balances)	
Request ex_lock(X)		
	avg(account balances)	
	Commit	
gain ex_lock(X)		**T1 gains lock**
checking += X		
Commit		

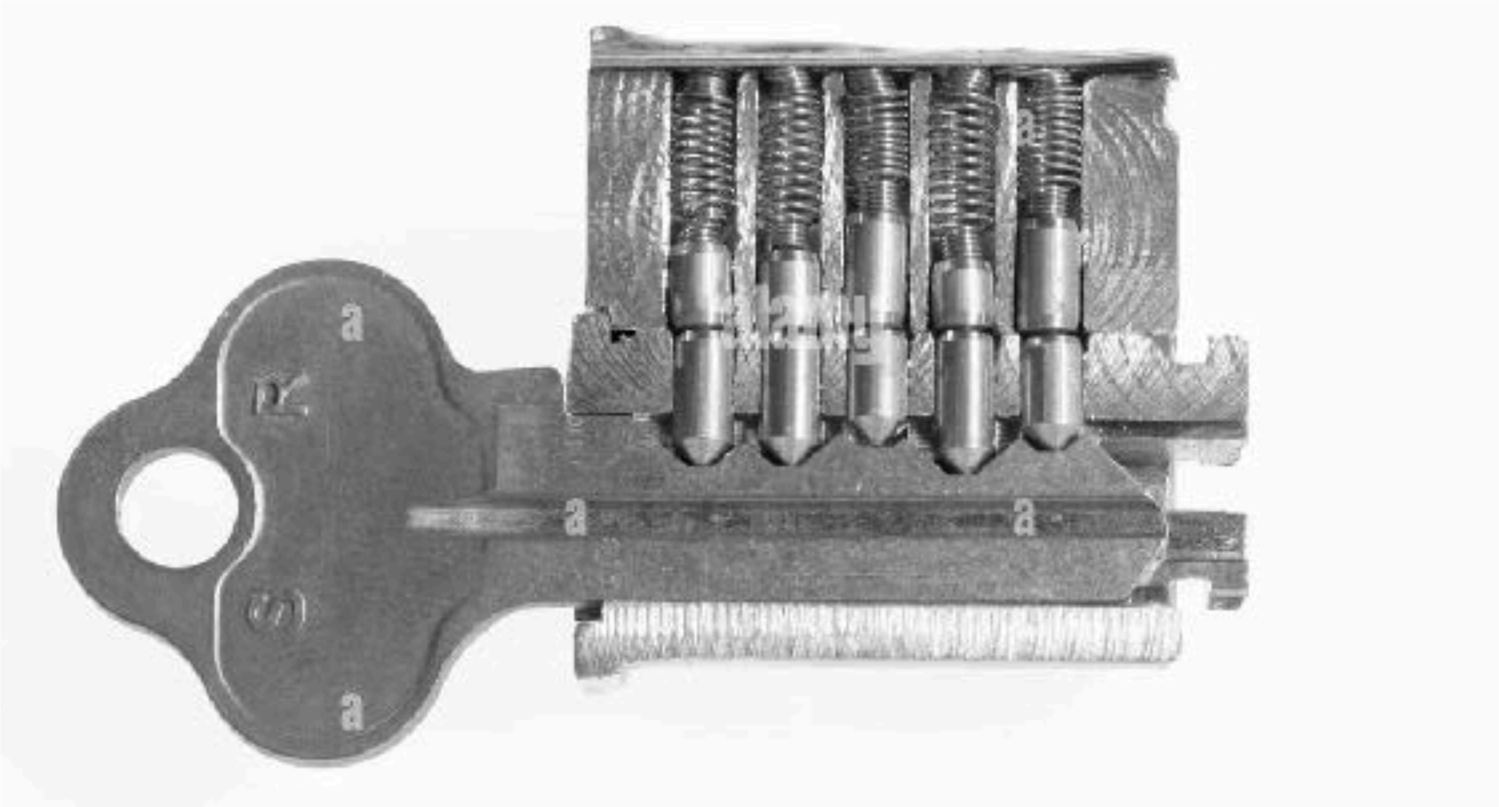
Fixing Example 2



Both examples now work correctly

	Example 1 Interest & Payments	Example 2 Updates & Statistics
		
Problems:	Dirty reads	Unrepeatable reads
Solution:	**Exclusive write locks**	**Shared read locks**

How are locks implemented?



Lock Manager - a hash table

Object ID

Lock Manager - a hash table

Object ID → (**type**, lock **count**, **queue** of waiting requests)

Lock Manager - a hash table

Object ID → (**type**, lock count, queue of waiting requests)

Shared/exclusive

Lock Manager - a hash table

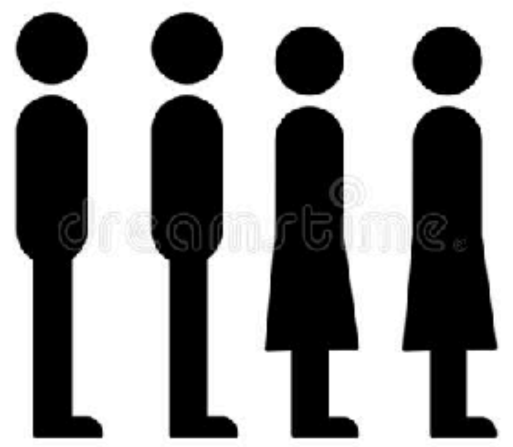
Object ID → (type, **lock count**, queue of waiting requests)

In case of shared lock

Lock Manager - a hash table

Object ID → (type, lock count, **queue of waiting requests**)

What to invoke next when this lock is released



Lock Manager - a hash table

Object ID → (type, lock count, queue of waiting requests)

e.g., Row, Page, Table

Lock Manager - a hash table

Object ID → (type, lock count, queue of waiting requests)

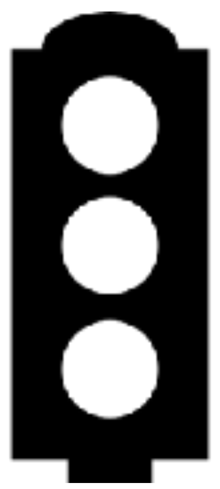
We lock an object the first time it is accessed by a transaction

Lock Manager - a hash table

Object ID → (type, lock count, queue of waiting requests)

We lock an object the first time it is accessed by a transaction

Locking table implemented via OS locking primitives (e.g., mutexes)



An important distinction

Locks

Latches

An important distinction

	Locks	Latches
		
Separate:	**Transactions**	**Threads**

An important distinction

	Locks	Latches
		
Separate:	Transactions	Threads
Protect:	**DB content**	**In-memory data structures (LRU queue)**

An important distinction

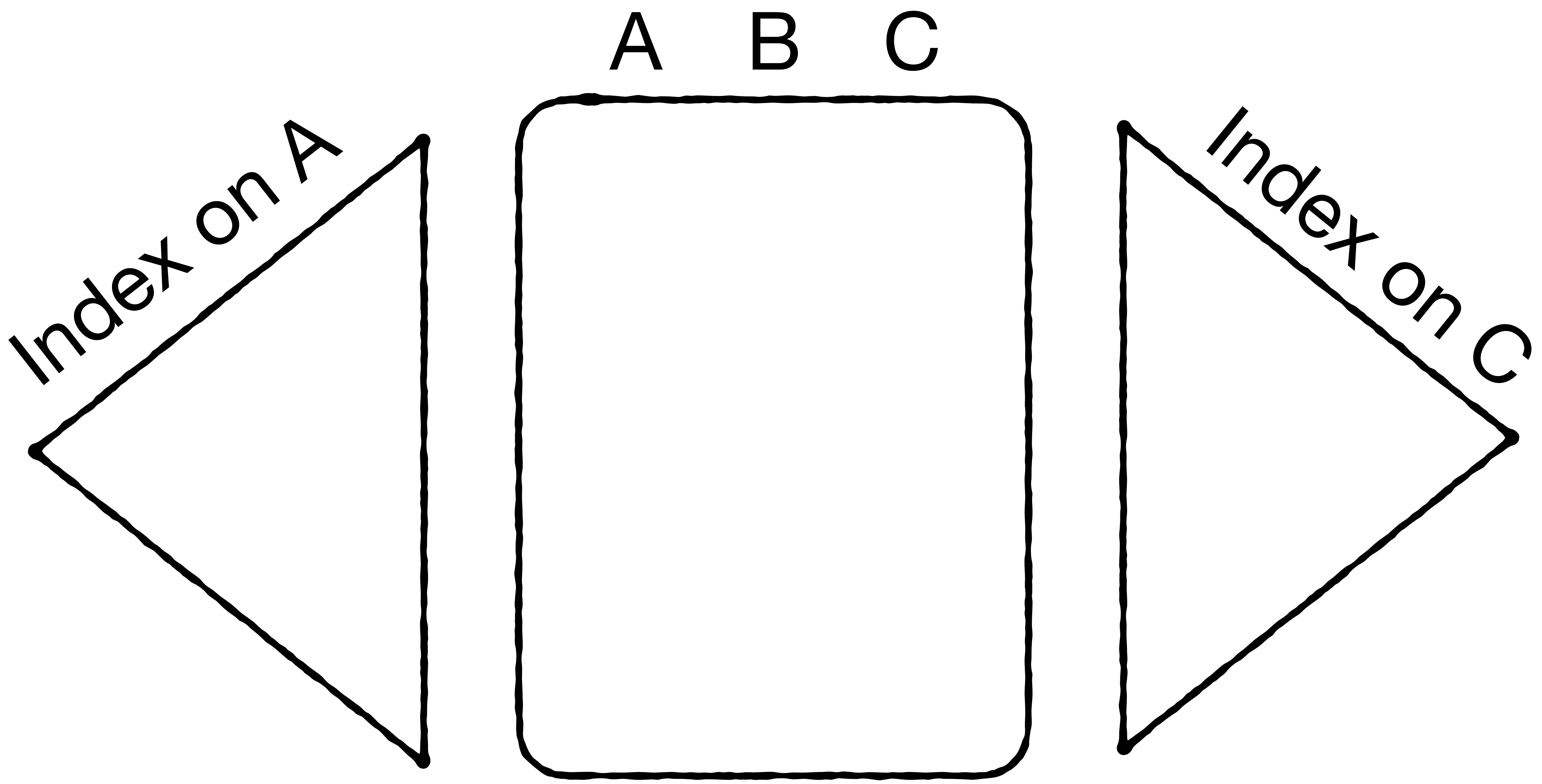
	Locks	Latches
		
Separate:	Transactions	Threads
Protect:	DB content	In-memory data structures (LRU queue)
Duration:	**Transaction**	**Critical section**

An important distinction

	Locks	Latches
		
Separate:	Transactions	Threads
Protect:	DB content	In-memory data structures (LRU queue)
Duration:	Transaction	Critical section
Implementation:	**Lock manager**	**Spin-locks**

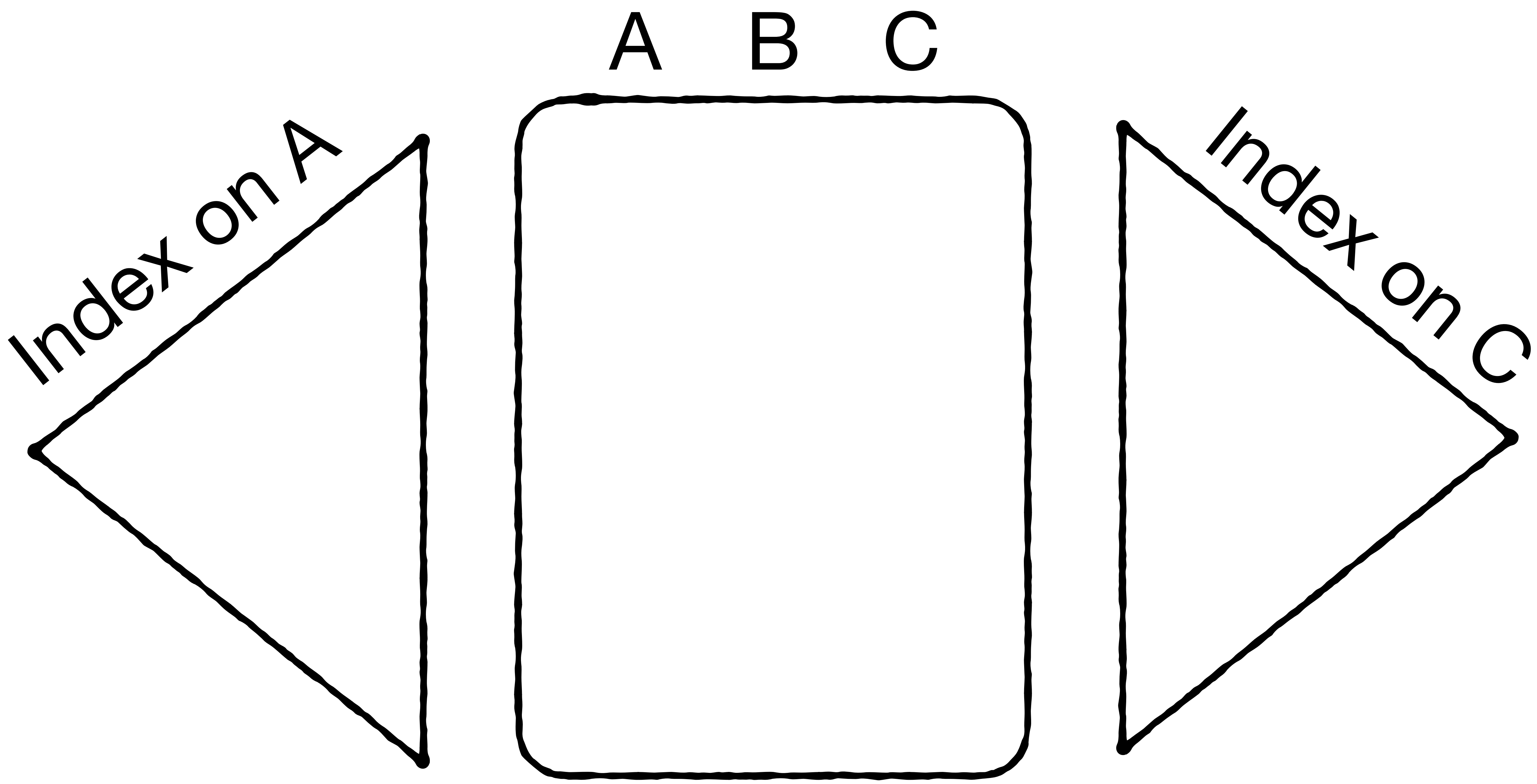
The DB modifies relevant indexes as a part of a transaction

e.g., update table set C = '…' where A '…'



The DB modifies relevant indexes as a part of a transaction

e.g., update table set C = '…' where A '…'



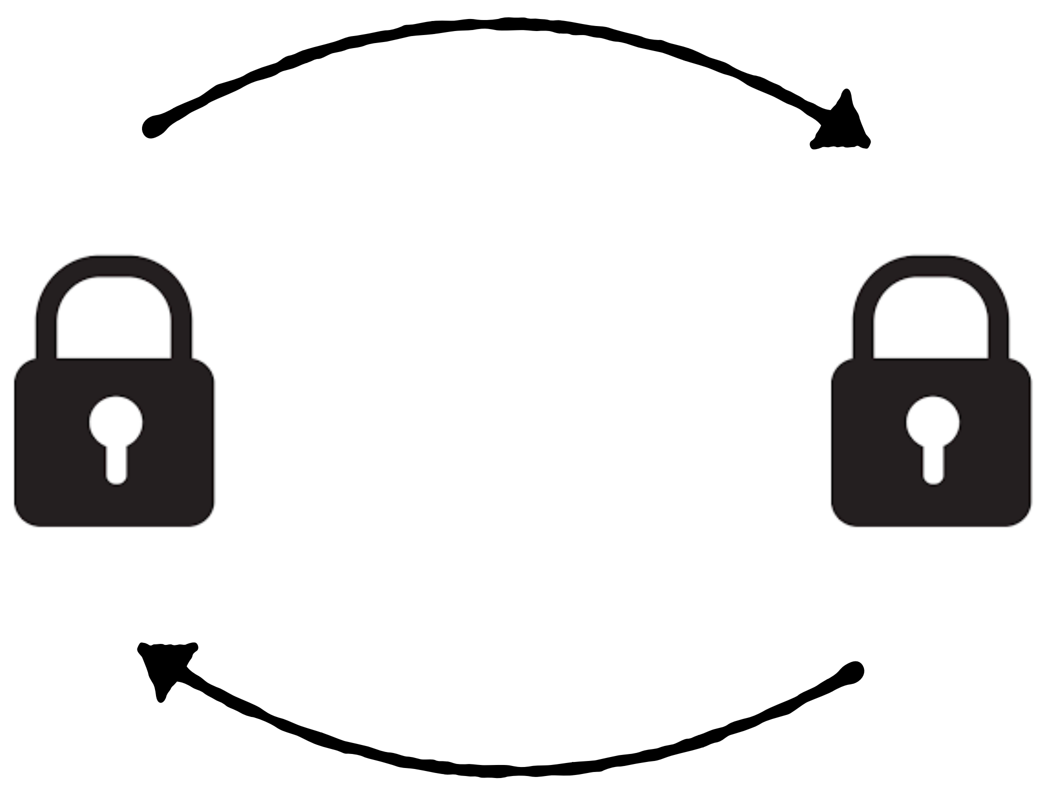
Schedule

(1) Begin transaction
(2) Search index A
(3) Access row and update C
(4) Update entry in C's index
(5) Commit

Anything to make you uneasy about two-phase locking so far?

Anything to make you uneasy about two-phase locking so far?

Deadlocks: transactions waiting on each other's locks forever



Deadlock example: concurrent payments

T1: A to B	T2: B to A
A = A - 100	B = B - 100
B = B + 100	A = A + 100

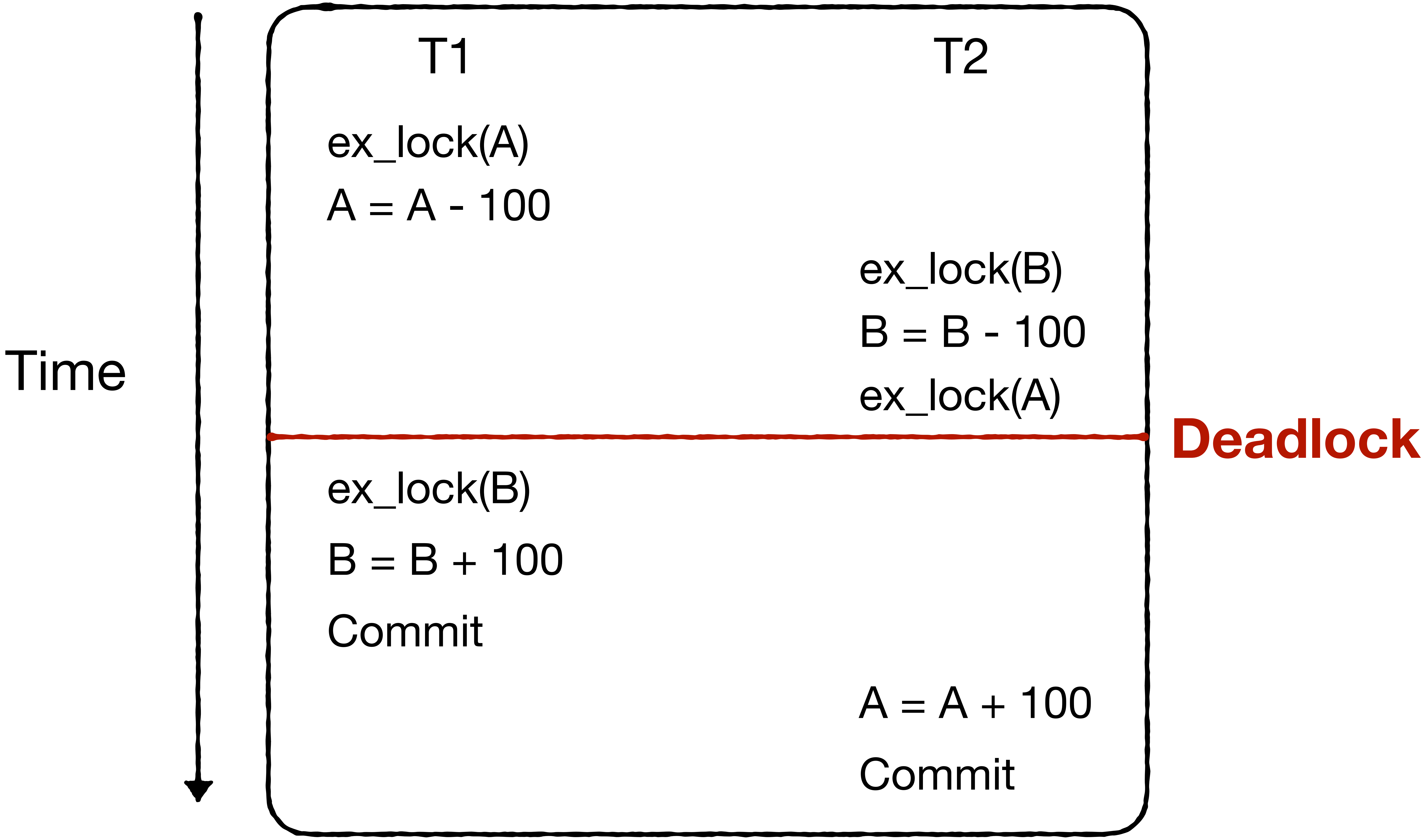
Deadlock example: concurrent payments

Time ↓

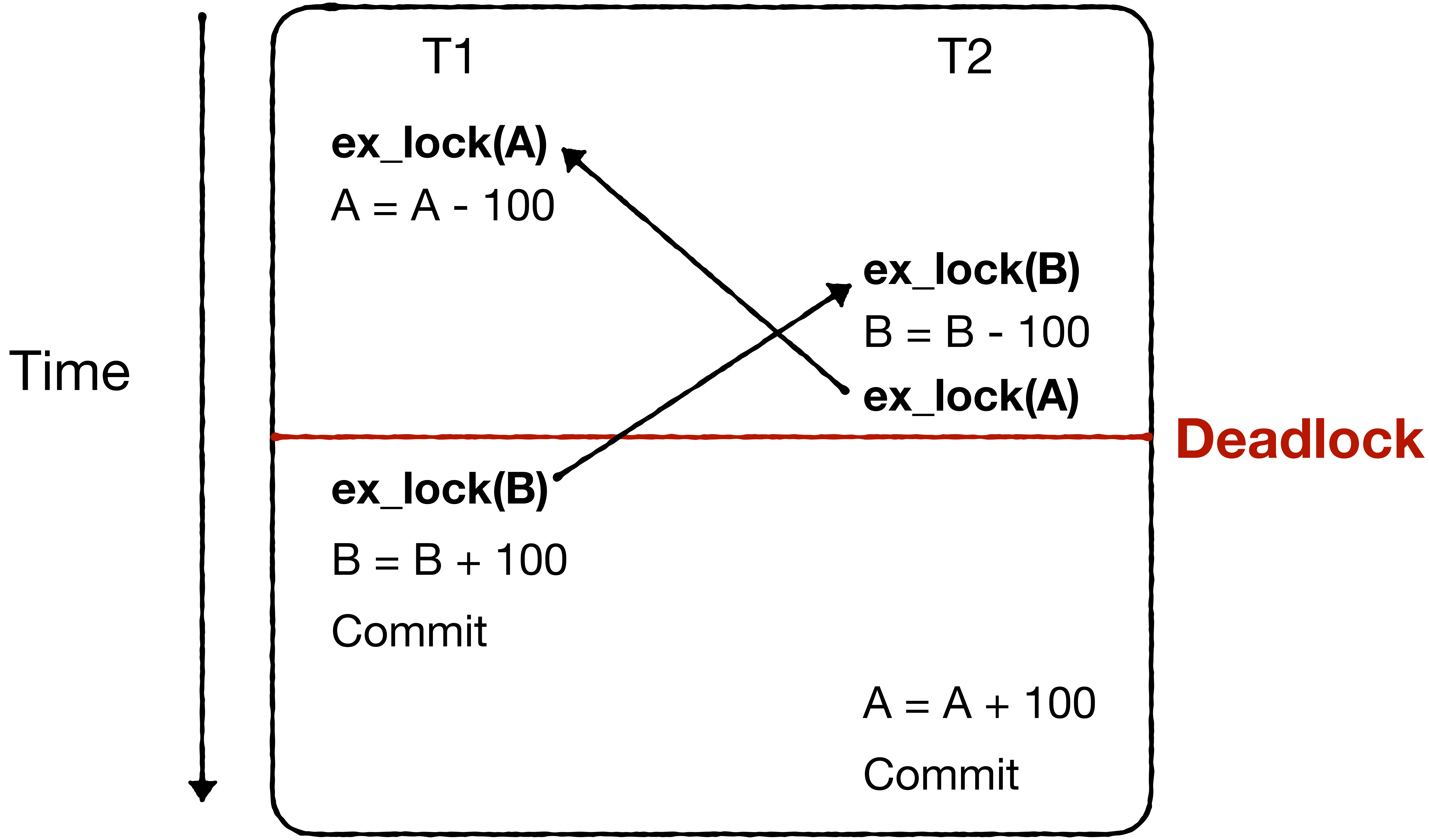
T1	T2
ex_lock(A)	
A = A - 100	
ex_lock(B)	
B = B + 100	
Commit	
	ex_lock(B)
	B = B - 100
	ex_lock(A)
	A = A + 100
	Commit

This is ok

Deadlock example: concurrent payments



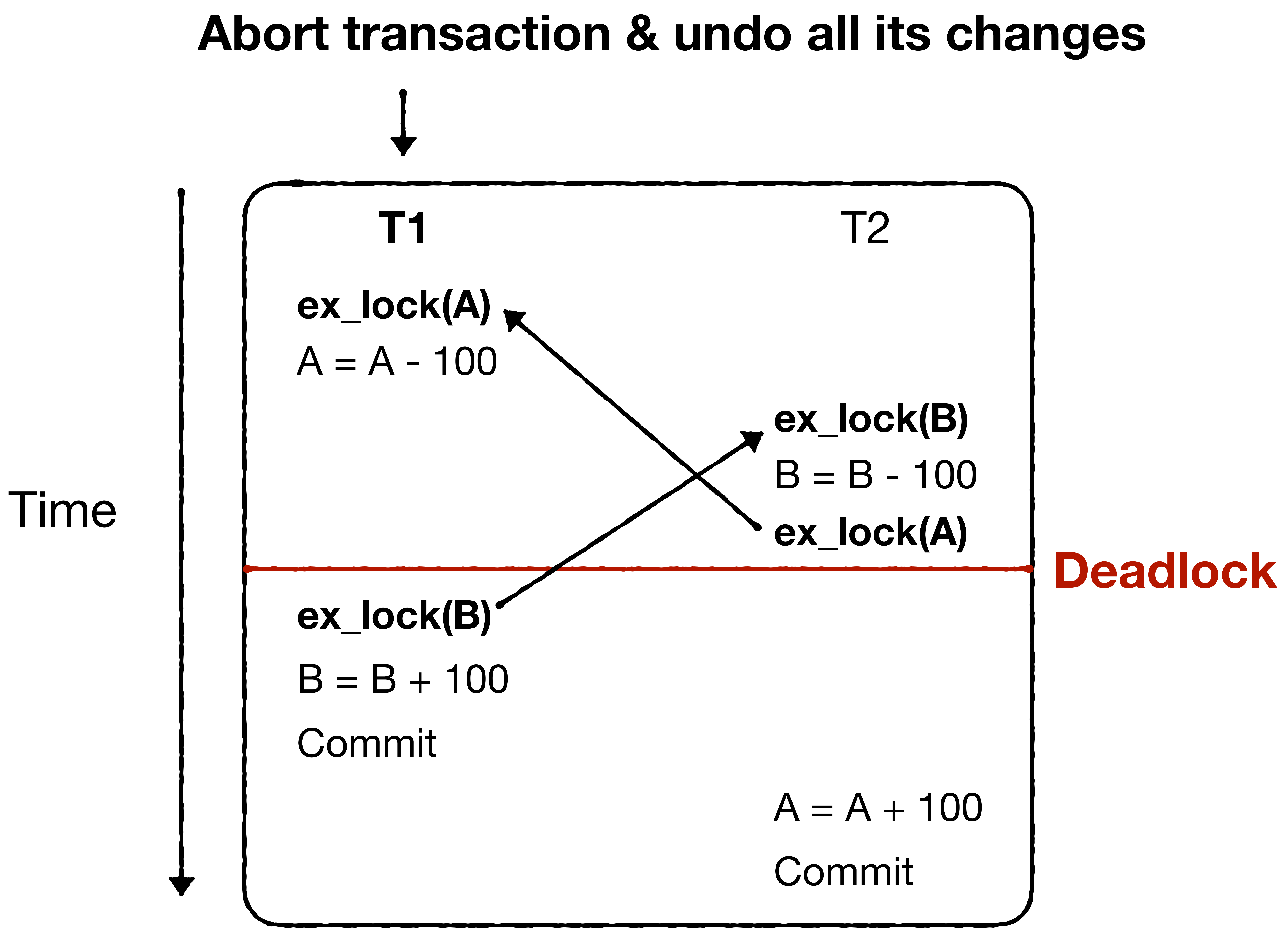
Deadlock example: concurrent payments



What should to do when we detect a deadlock?

What should to do when we detect a deadlock?

Abort transaction & undo all its changes


Abort transaction & undo all its changes
Time
T1
T2
ex_lock(A)
A = A - 100
ex_lock(B)
B = B - 100
ex_lock(A)
Deadlock
ex_lock(B)
B = B + 100
Commit
A = A + 100
Commit

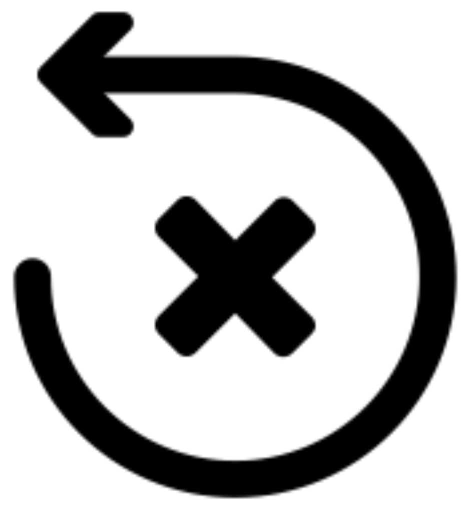
Abort transaction & undo all its changes

How to undo?

Abort T1	T2
ex_lock(A)	
A = A - 100	
	ex_lock(B)
	B = B - 100
	ex_lock(A)
ex_lock(B)	
B = B + 100	
Commit	
	A = A + 100
	Commit

Deadlock

How to undo?



How to undo?

record before-image for all changes to the DB in a sequential log.

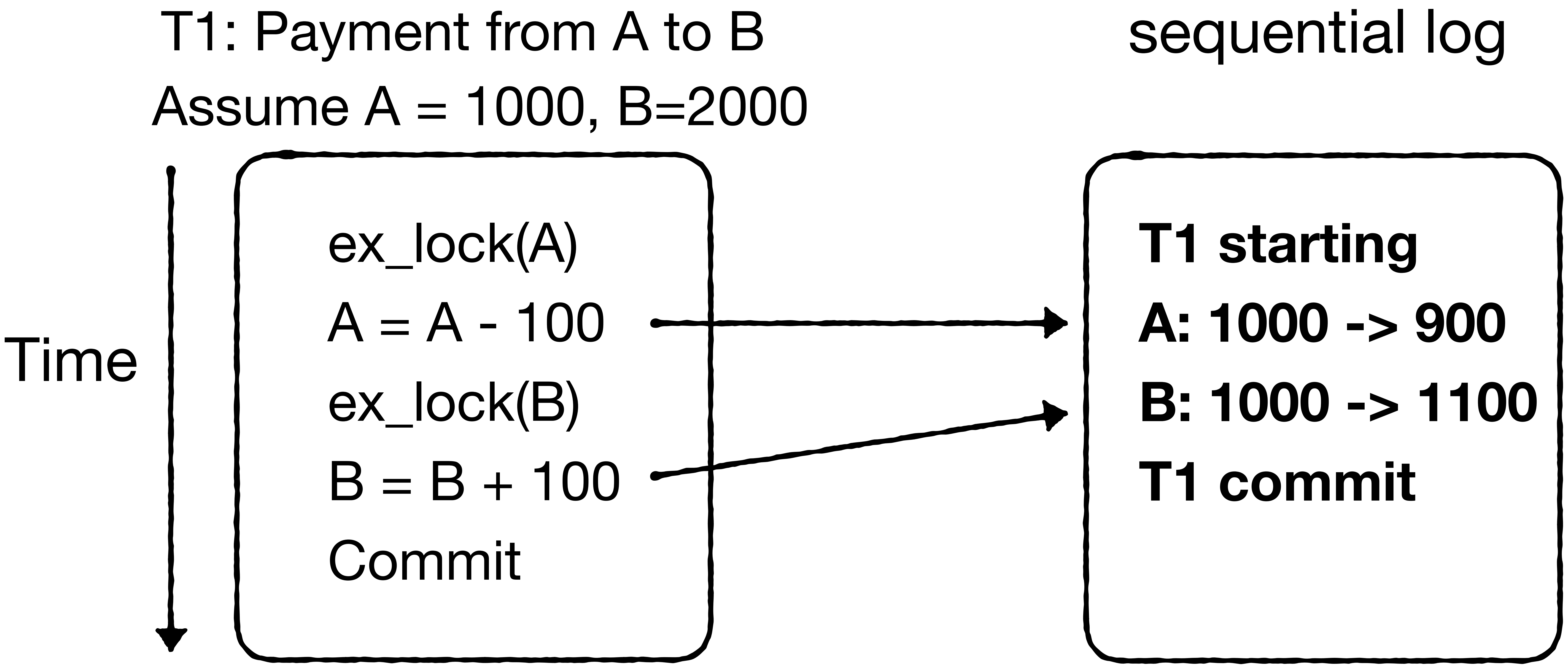
T1: Payment from A to B

Assume A = 1000, B=2000

Time

```
ex_lock(A)
A = A - 100
ex_lock(B)
B = B + 100
Commit
```

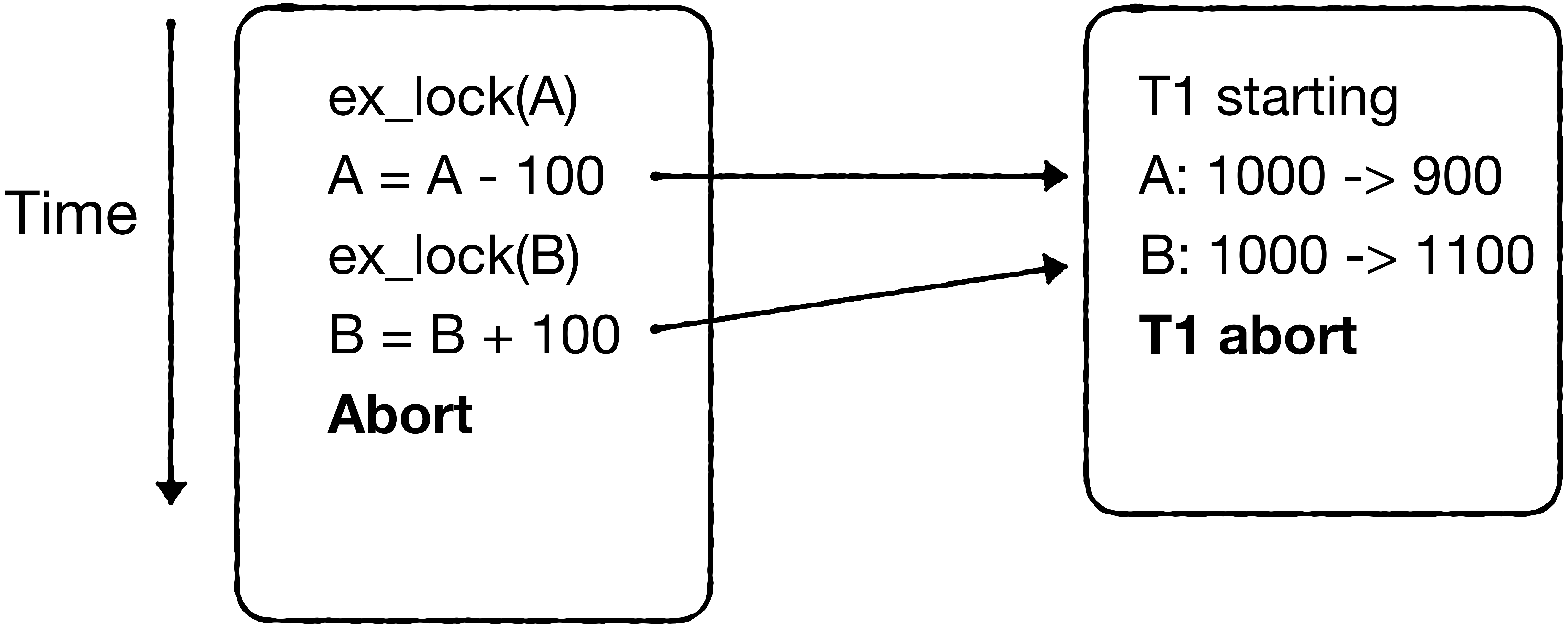
record before-image for all changes to the DB in a sequential log.


T1: Payment from A to B
Assume A = 1000, B=2000
sequential log
Time
ex_lock(A)
A = A - 100
ex_lock(B)
B = B + 100
Commit
T1 starting
A: 1000 -> 900
B: 1000 -> 1100
T1 commit

T1: Payment from A to B

Assume A = 1000, B=2000

sequential log



If transaction aborts before completing, we undo its changes via its before-images in the log

T1: Payment from A to B

Assume A = 1000, B=2000

Time

ex_lock(A)

A = A - 100

ex_lock(B)

B = B + 100

Abort

Write (A)

Write (B)

Release locks

sequential log

T1 starting

A: **1000** -> 900

B: **1000** -> 1100

T1 abort

If transaction aborts before completing, we undo its changes via its before-images in the log

How can we detect & prevent deadlocks?

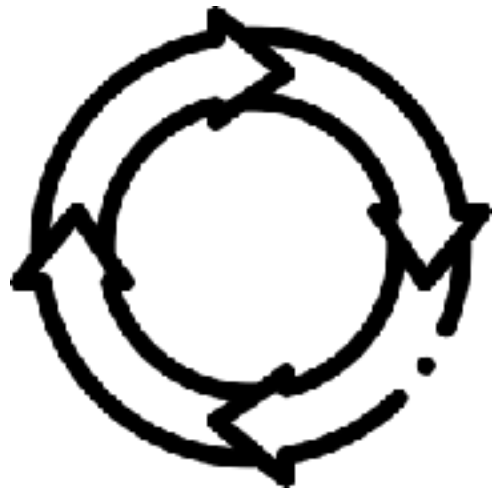
How can we detect & prevent deadlocks?

Timeouts

Conservative Two Phase Locking

Abort on wait

Cycle Detection



Timeouts

Abort a transaction after waiting a certain time for a lock

Pros?

Cons?

Timeouts

Abort a transaction after waiting a certain time for a lock

Pros? **Simple**

Cons? **Speculative - wastes work aborting on non-deadlocks**

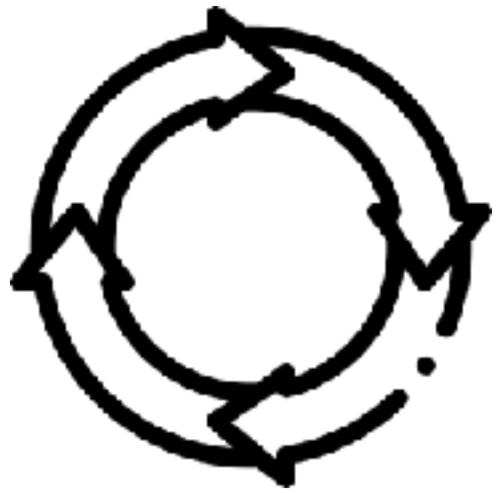
How can we detect & prevent deadlocks?

Timeouts

Conservative Two Phase Locking

Abort on wait

Cycle Detection



Conservative Two Phase Locking

A transaction is divided into two phases

Phase 1

Take all locks the transaction could possibly need

Phase 2

Release all locks during commit

Conservative Two Phase Locking

A transaction is divided into two phases

Phase 1

Take all locks the transaction could possibly need

Phase 2

Release all locks during commit

Pros: no deadlocks

Con:

Conservative Two Phase Locking

A transaction is divided into two phases

Phase 1

Take all locks the transaction could possibly need

Phase 2

Release all locks during commit

Pros: no deadlocks

Con: takes more locks and holds them for longer

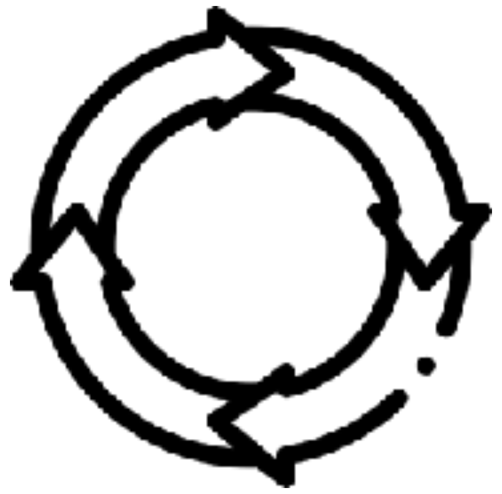
How can we detect & prevent deadlocks?

Timeouts

Conservative Two Phase Locking

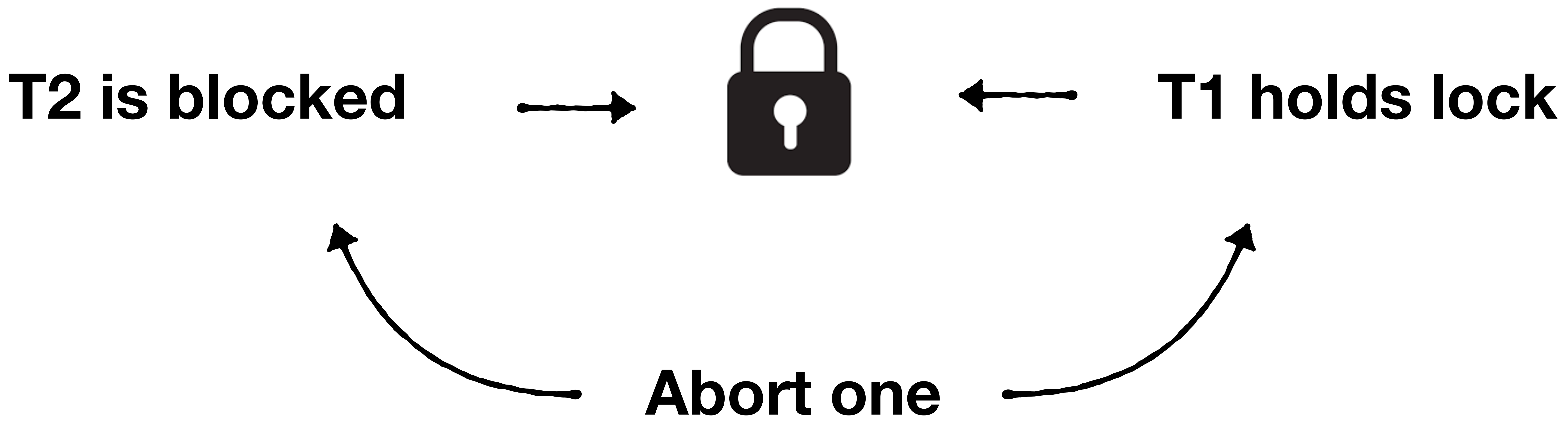
Abort on Wait

Cycle Detection



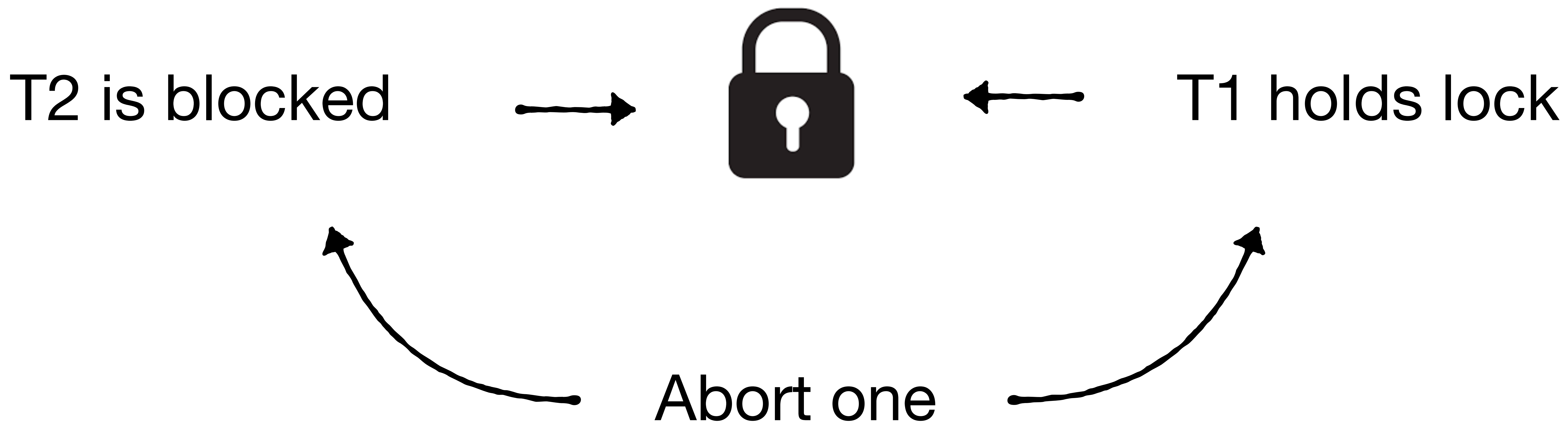
Abort on Wait

When a transaction is blocked, abort it or the transaction holding the lock.



Abort on Wait

When a transaction is blocked, abort it or the transaction holding the lock.



Pros: no deadlocks

Con: defeatist (wastes work, or aborts many transactions)

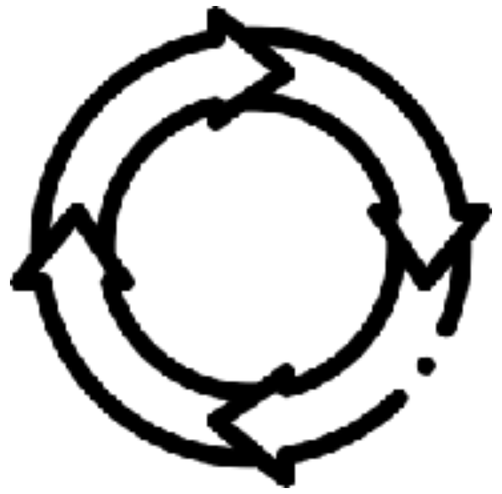
How can we detect & prevent deadlocks?

Timeouts

Conservative Two Phase Locking

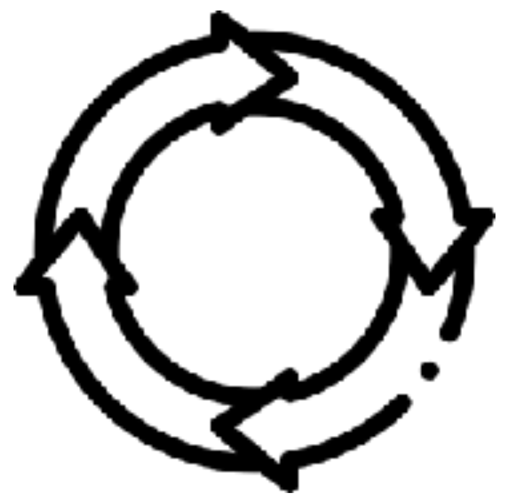
Abort on wait

Cycle Detection



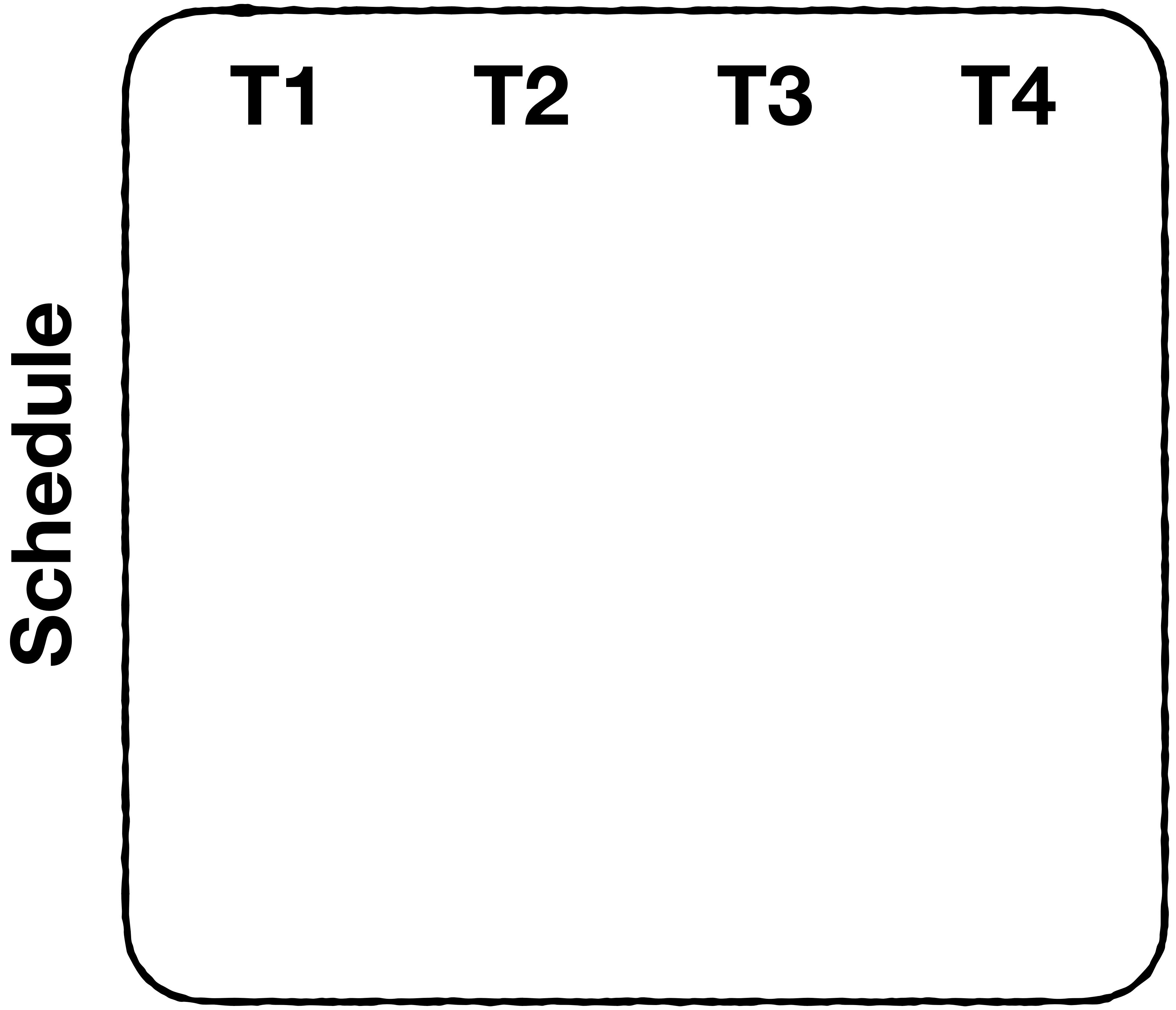
Cycle Detection

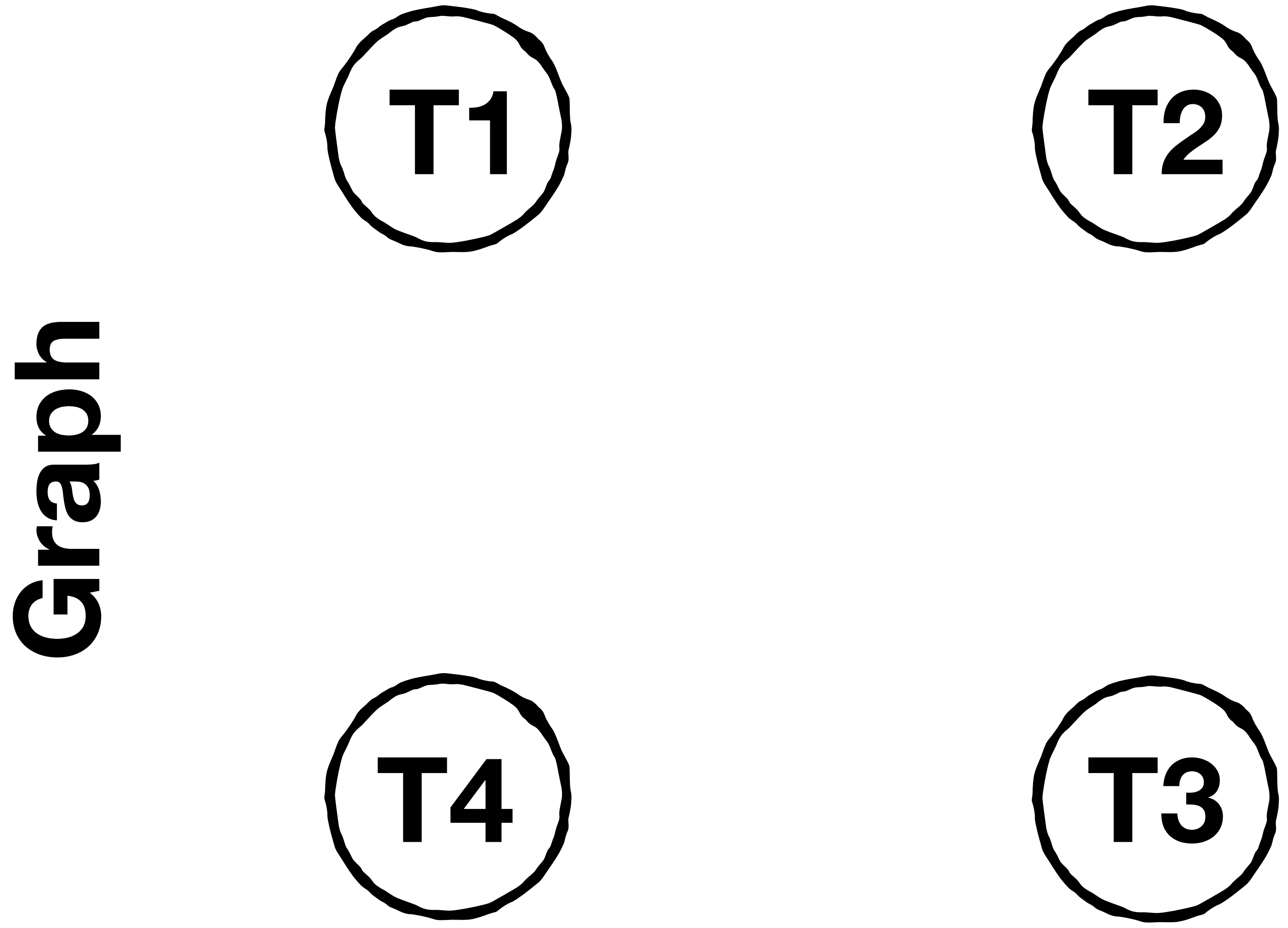
The DB can maintain graph of transactions waiting on each other.



Cycle Detection

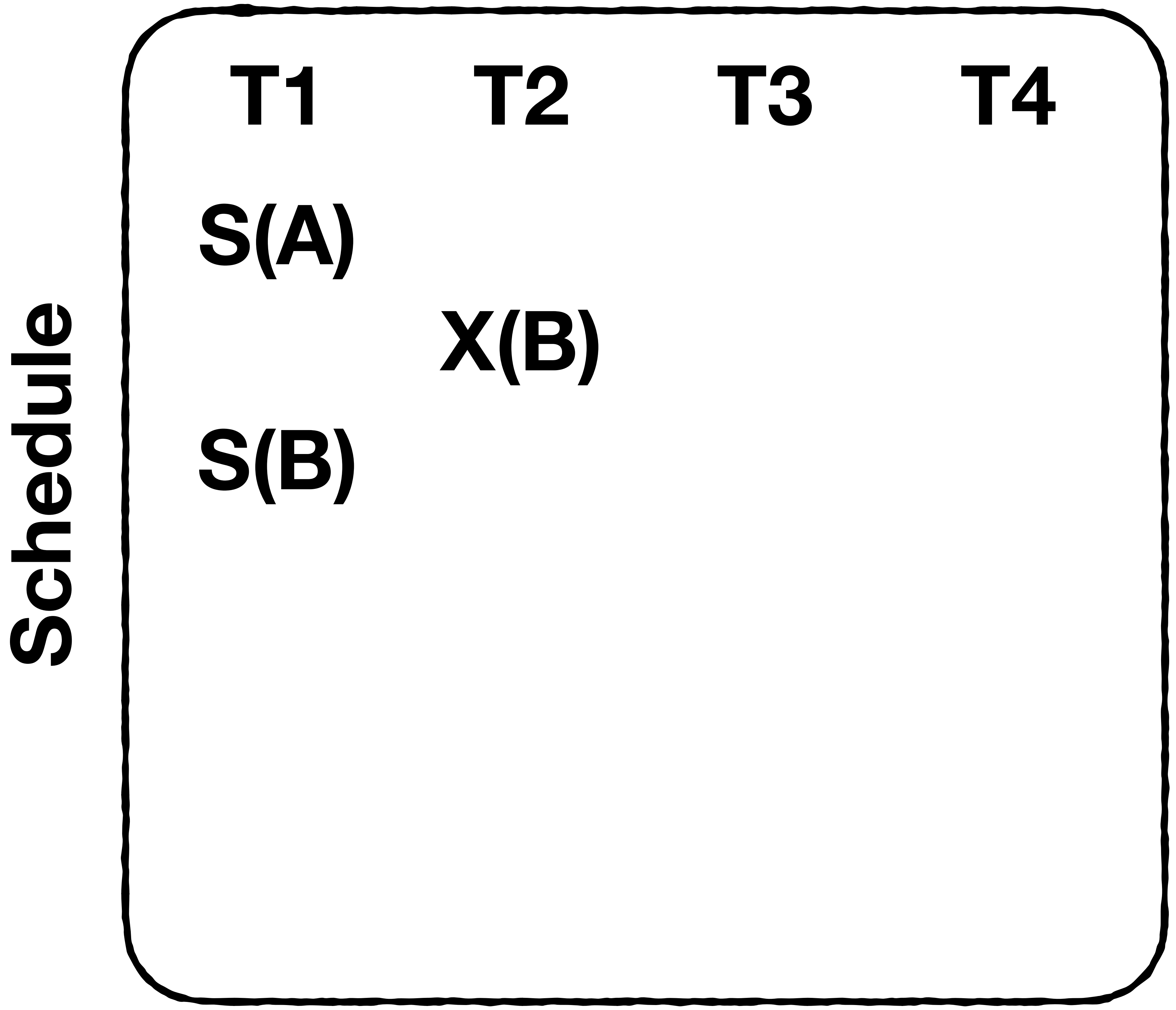
The DB can maintain graph of transactions waiting on each other.

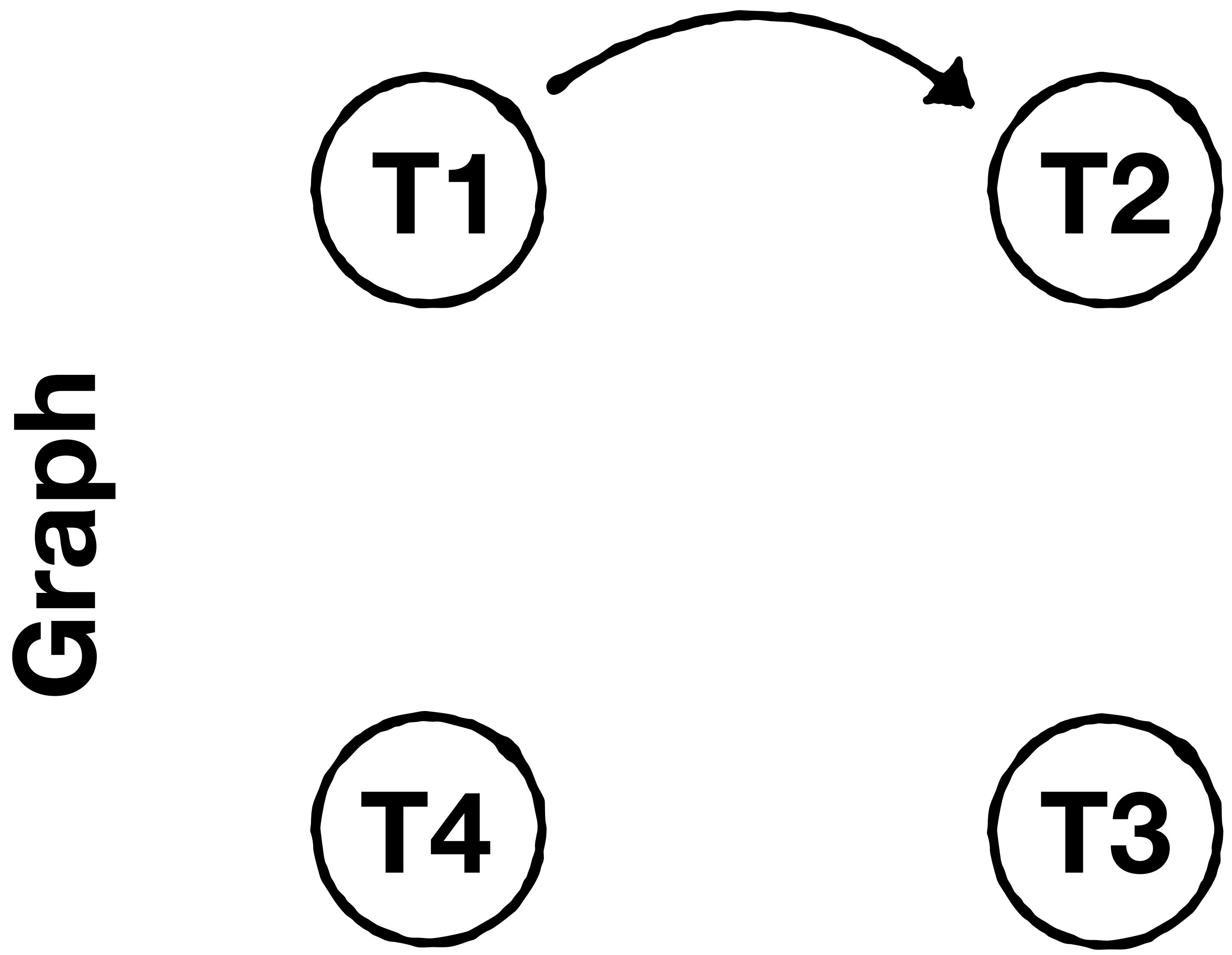




Cycle Detection

The DB can maintain graph of transactions waiting on each other.





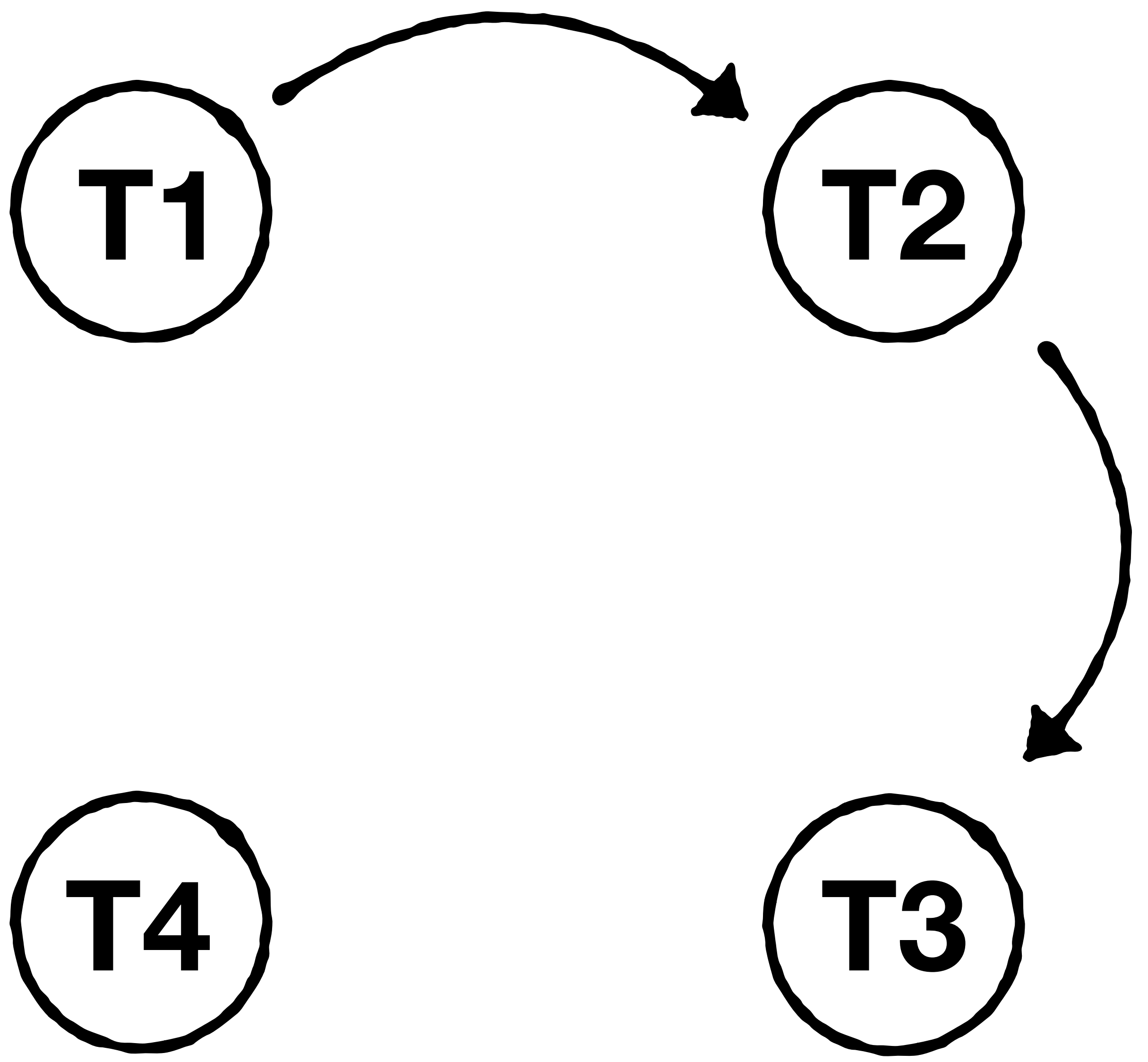
Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Schedule

T1	T2	T3	T4
S(A)			
	X(B)		
S(B)			
		S(C)	
	X(C)		

Graph



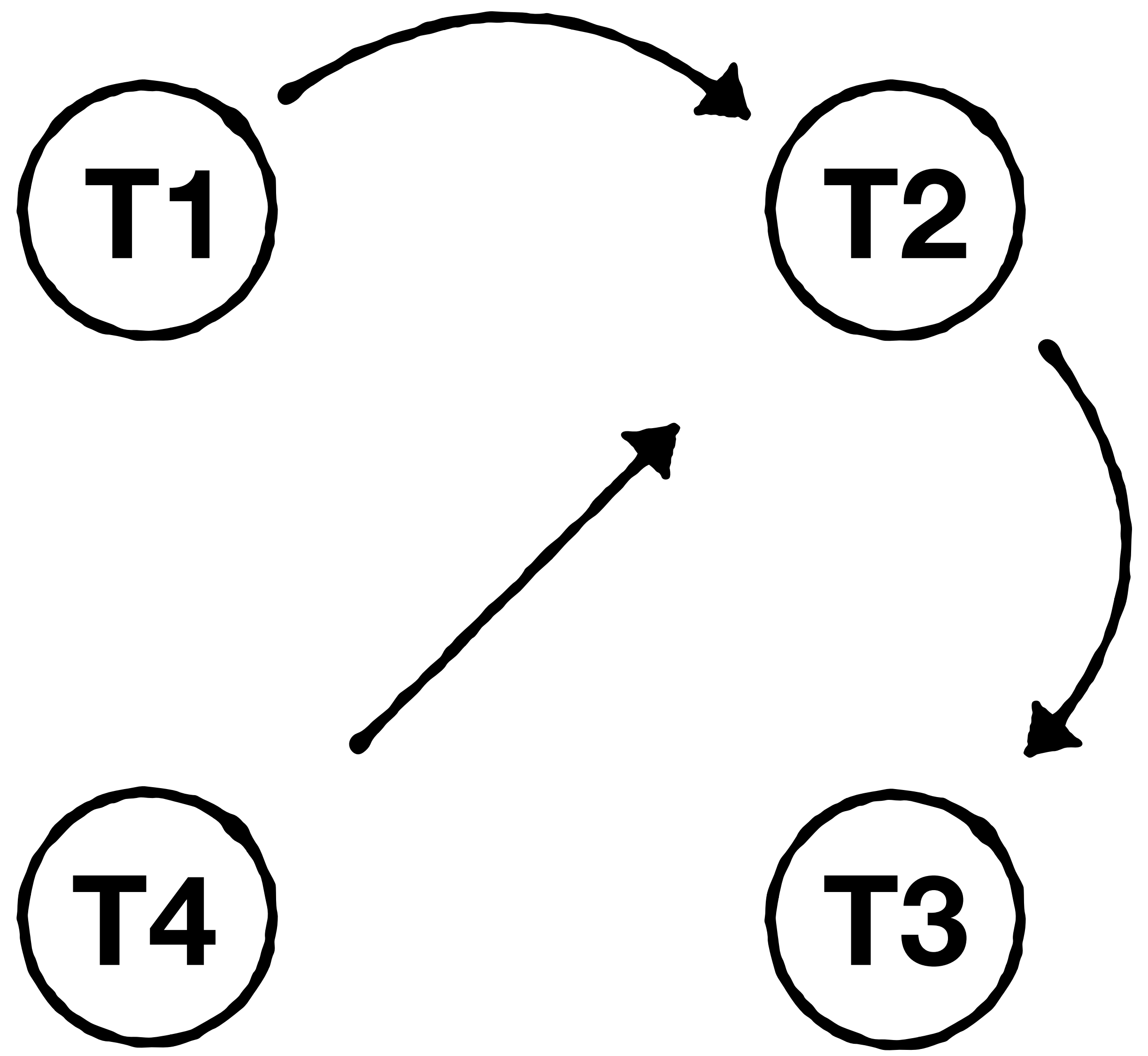
Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Schedule

T1	T2	T3	T4
S(A)			
	X(B)		
S(B)			
		S(C)	
	X(C)		
			X(B)

Graph



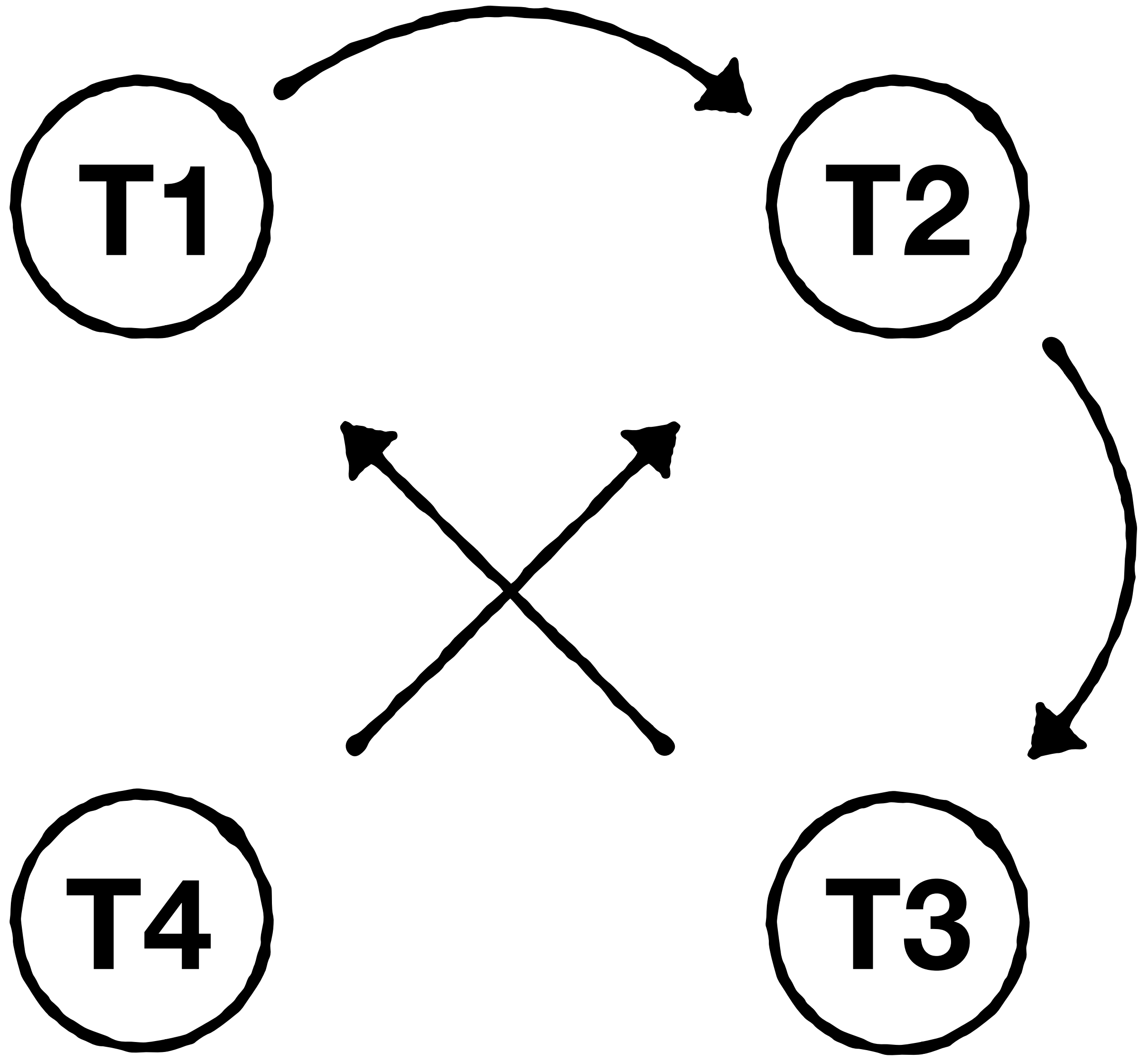
Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Schedule

T1	T2	T3	T4
S(A)			
	X(B)		
S(B)			
		S(C)	
	X(C)		
			X(B)
		X(A)	

Graph

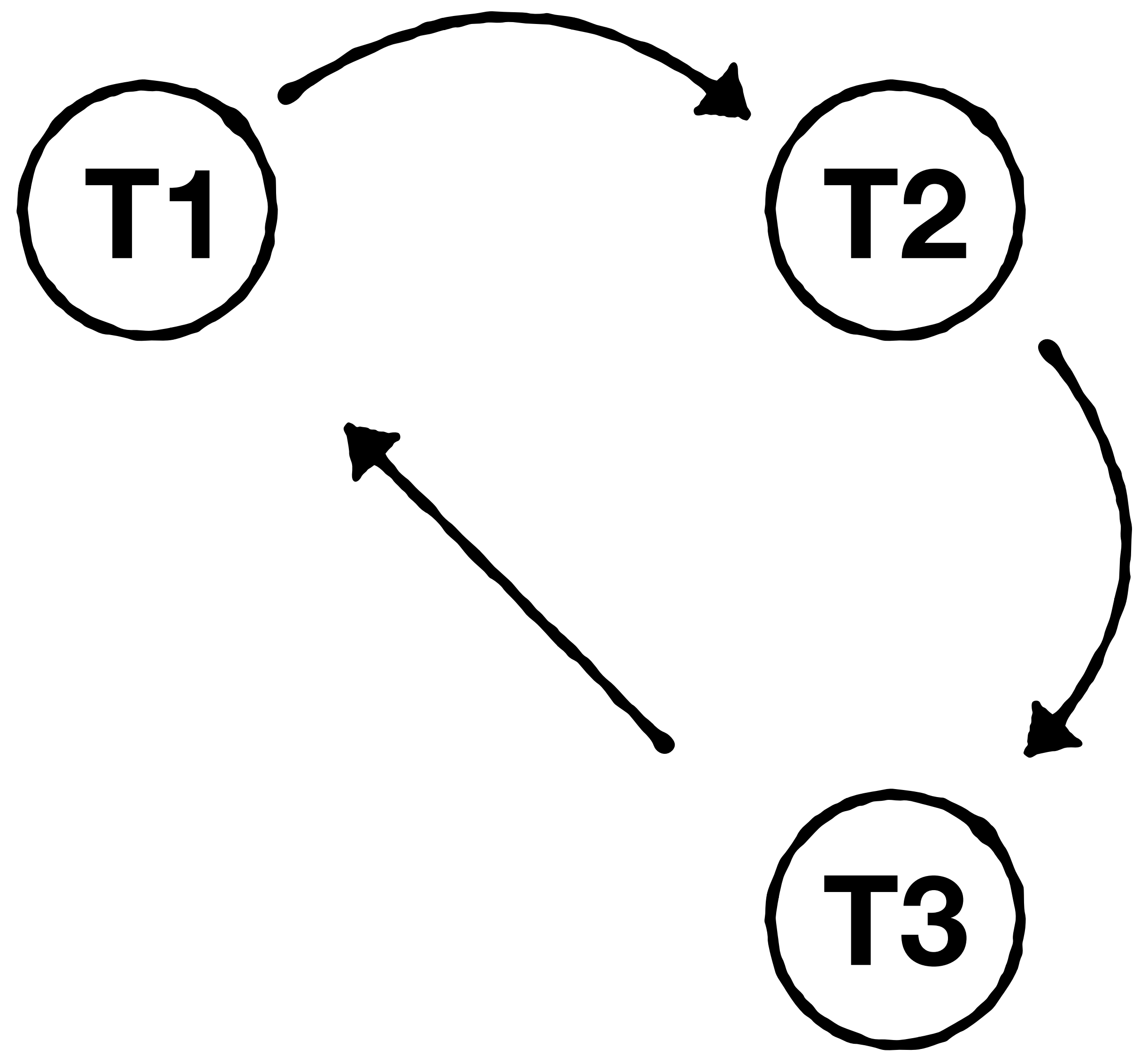


Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Other approaches are speculative

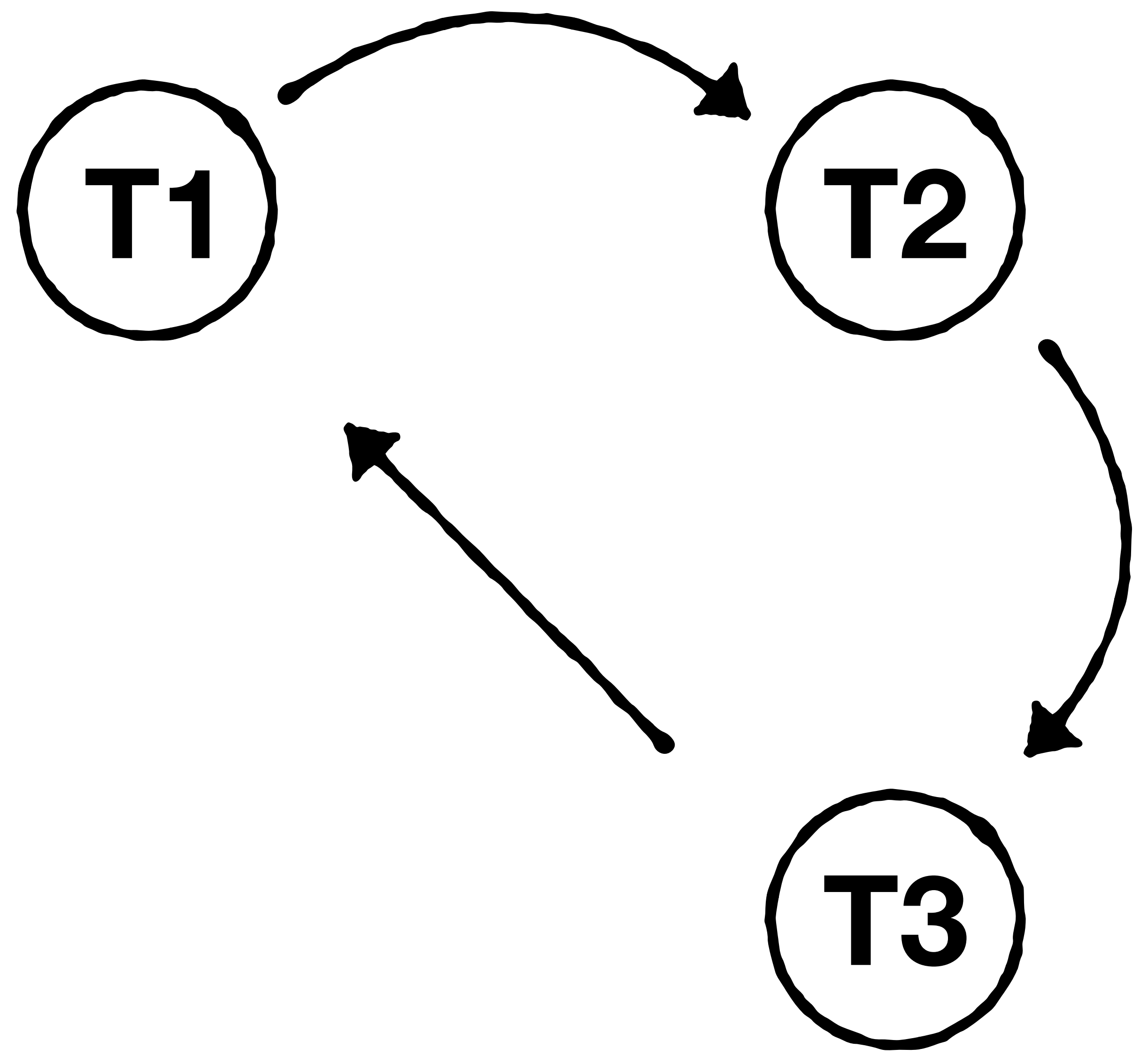
Here we detect a deadlock for sure



Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Should we abort T1, T2 or T3?

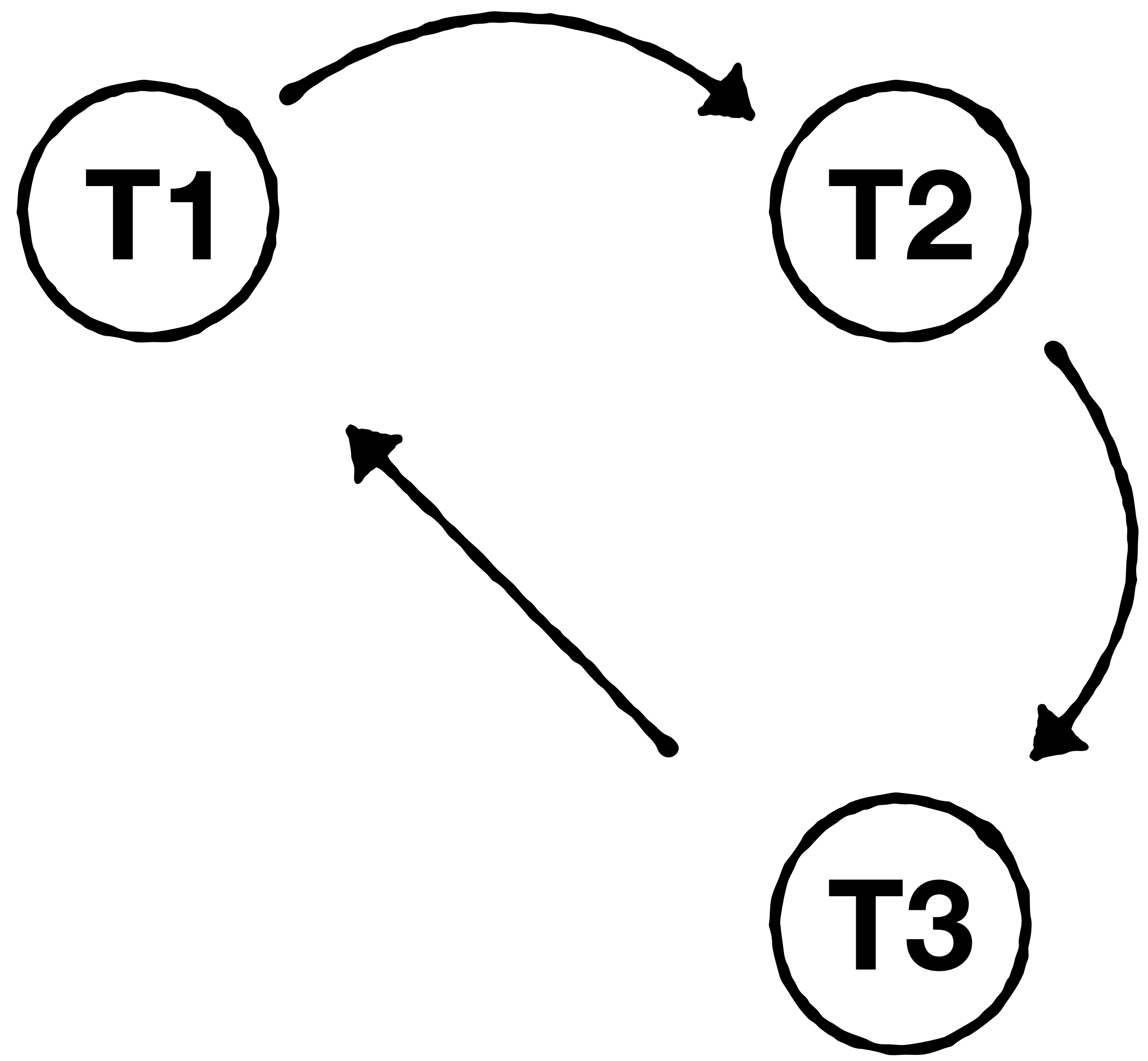


Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Abort the transaction that:

(1) has done the least work



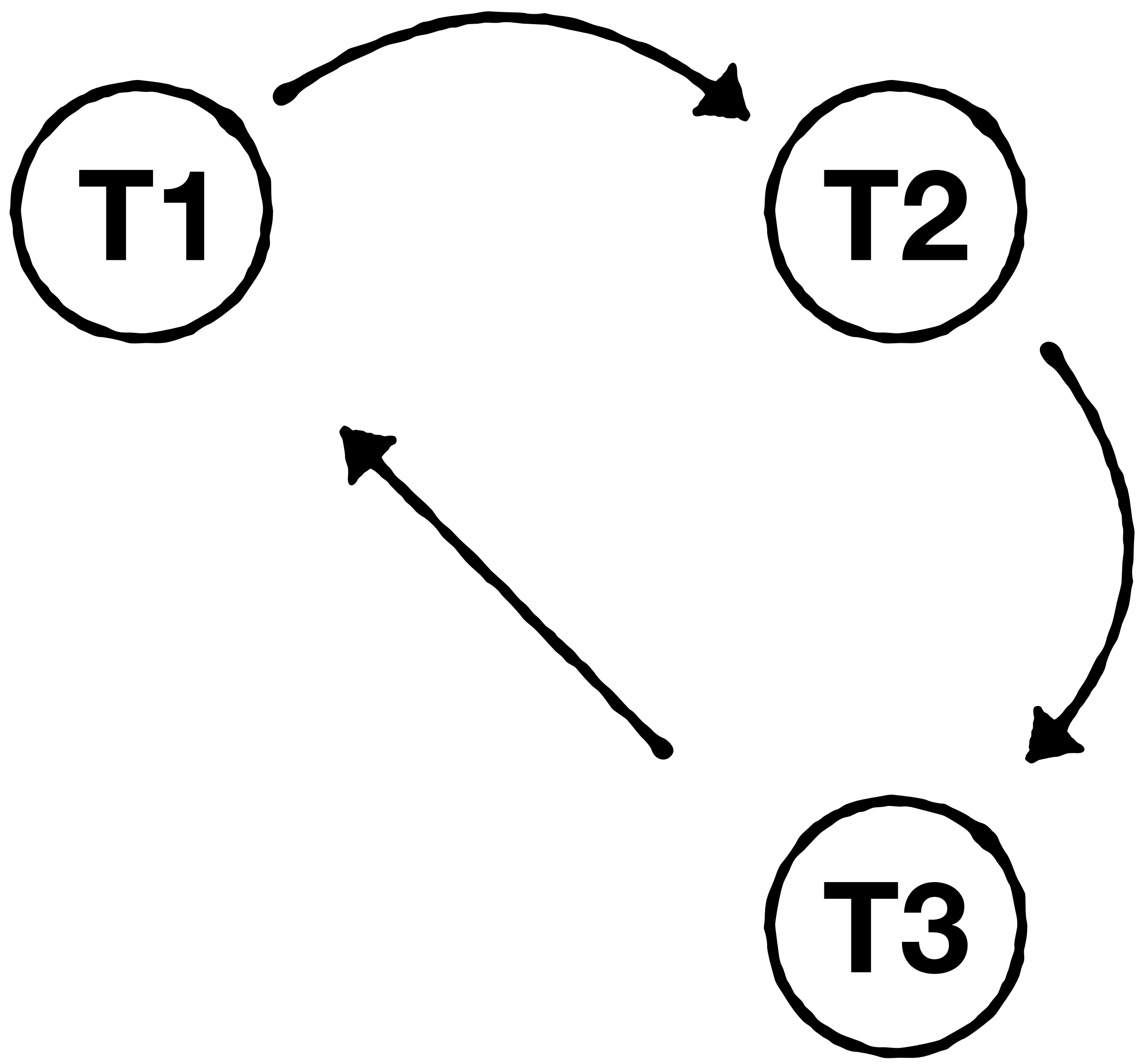
Cycle Detection

The DB can maintain graph of transactions waiting on each other.

Abort the transaction that:

(1) has done the least work

(2) Farthest from completion



Cycle Detection

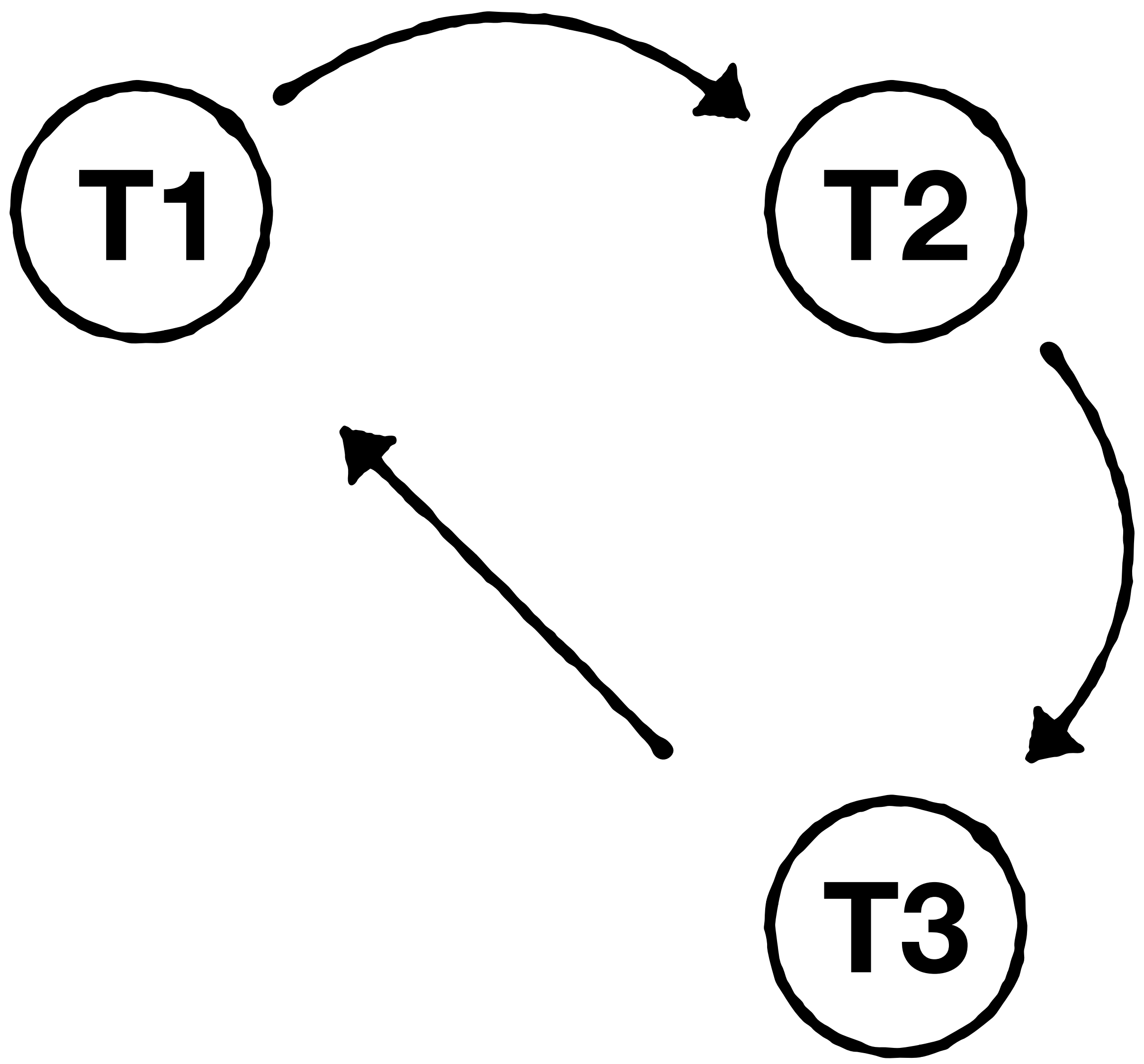
The DB can maintain graph of transactions waiting on each other.

Abort the transaction that:

(1) has done the least work

(2) Farthest from completion

(3) Been aborted the least times

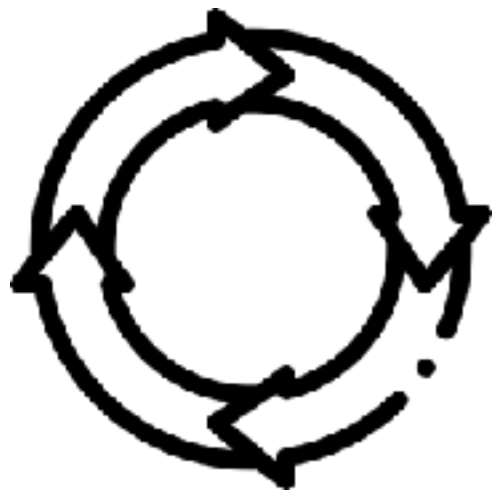


Timeouts

Conservative Two Phase Locking

Abort on wait

Cycle Detection



There is still a problem

Let’s return to our running examples

	Example 1	Example 2
	Interest & Payments	Updates & Statistics
		
Problems:	Dirty reads	Unrepeatable reads
Solution:	**Exclusive write locks**	**Shared read locks**

Let’s return to our running examples

Example 1

Interest & Payments

Dirty reads

Exclusive write locks

Example 2

Updates & Statistics

Unrepeatable reads

Shared read locks

Example 3

Inserts & Statistics

?

?

Example 3

T1: insert account

Insert A = 0

T2: statistics reporting

count(# accounts)

sum(account balances)

avg(account balances)

Example 3

Time

T1	T2
Insert A = 0	
Commit	
	count(# accounts)
	sum(account balances)
	avg(account balances)

Time

T1	T2
	count(# accounts)
	sum(account balances)
	avg(account balances)
Insert A = 0	
Commit	

Time

T1	T2
	count(# accounts)
	sum(account balances)
Insert A = 0	
Commit	
	avg(account balances)

Problem?

T1 | T2

Time

count(# accounts) ← **Only locks existing rows**

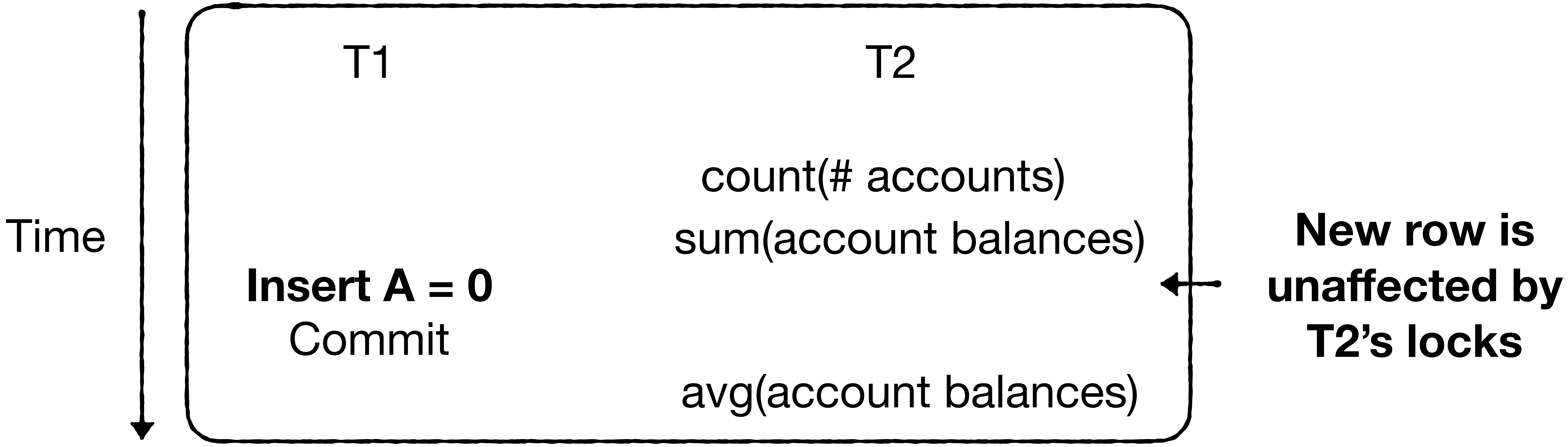
sum(account balances)

Insert A = 0

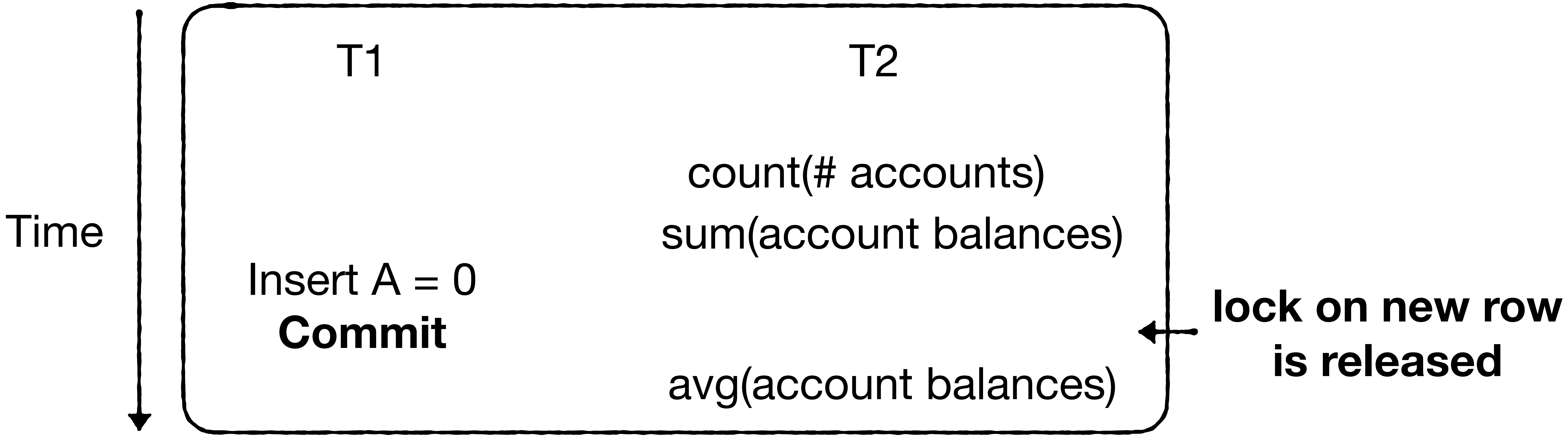
Commit

avg(account balances)

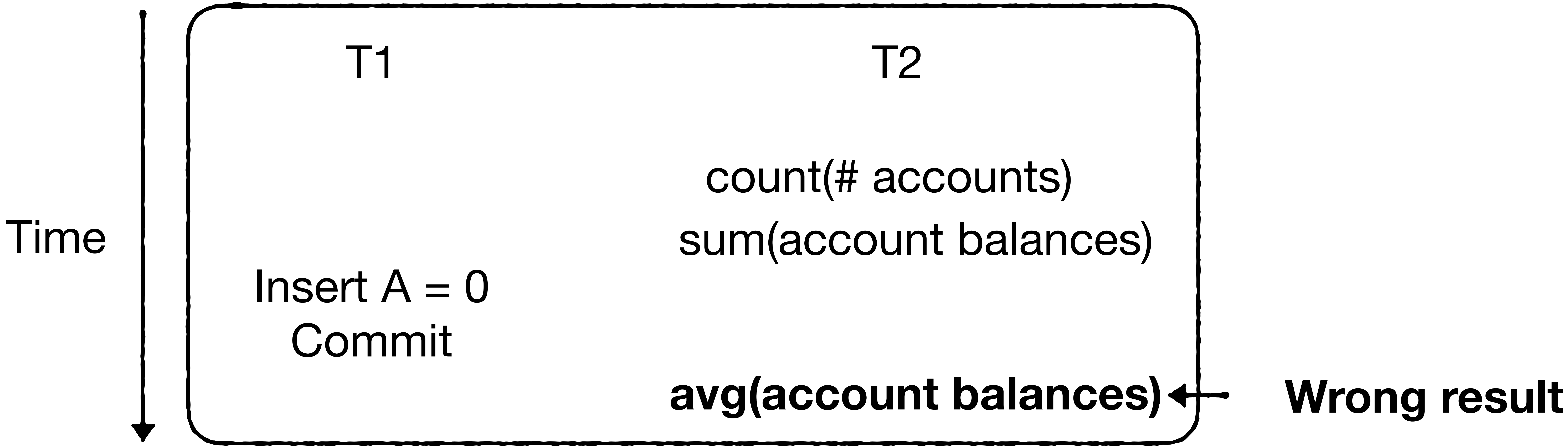
Problem?


T1
T2
Time
count(# accounts)
sum(account balances)
Insert A = 0
Commit
avg(account balances)
New row is unaffected by T2's locks

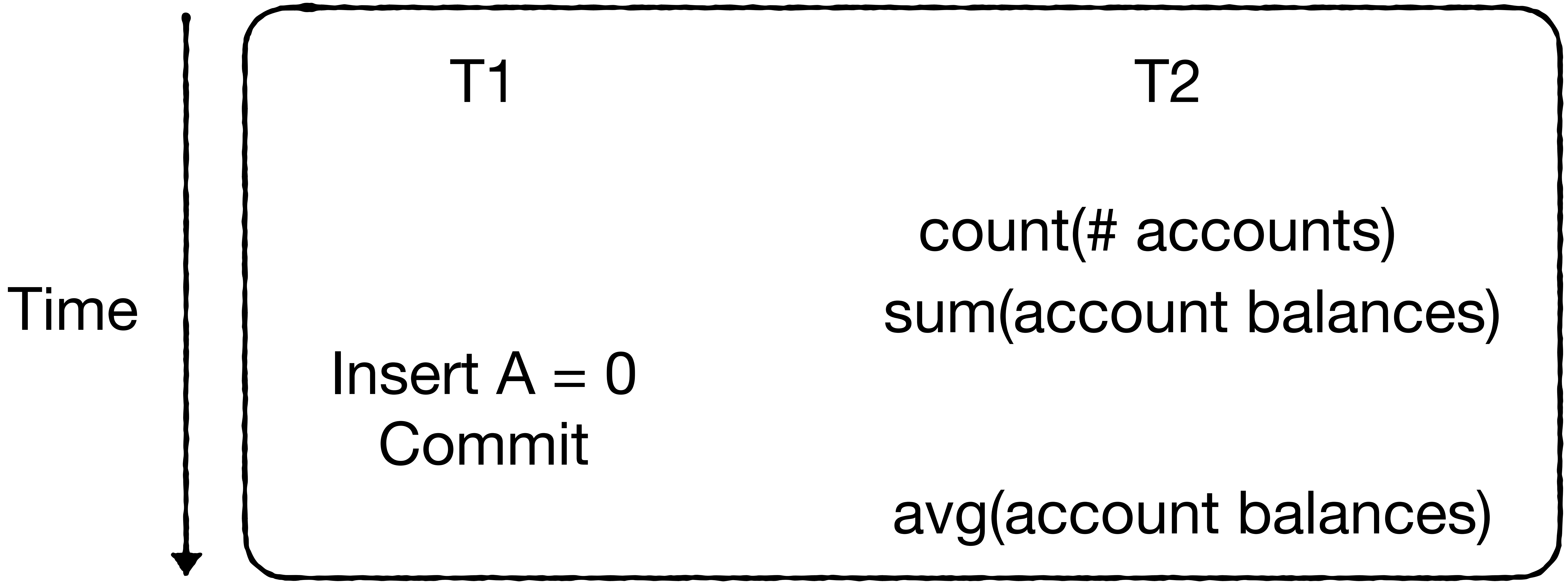
Problem?


T1
T2
count(# accounts)
Time
sum(account balances)
Insert A = 0
Commit
lock on new row is released
avg(account balances)

Problem?



Problem? **Inconsistent reporting: avg ≠ sum / count**



More broadly, this is a **phantom read**: a transaction accesses a set of rows twice, and qualifying rows are added in-between

How can we prevent phantom reads?

Time

T1	T2
	count(# accounts)
	sum(account balances)
Insert A = 0	
Commit	
	avg(account balances)

More broadly, this is a **phantom read**: a transaction accesses a set of rows twice, and qualifying rows are added in-between

How can we prevent phantom reads?

lock table

Aggressive

How can we prevent phantom reads?

lock table

Aggressive

Predicate locking

Complex & expensive

How can we prevent phantom reads?

lock table

Aggressive

Predicate locking

Complex & expensive

Select * from table where **ID=x**

Lock “id=x”

How can we prevent phantom reads?

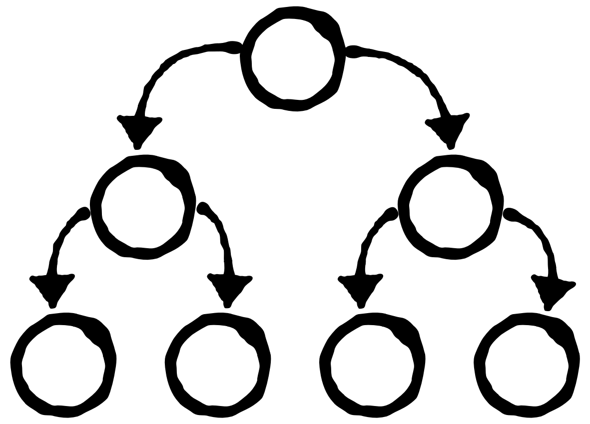
lock table

Aggressive

Predicate
locking

Complex &
expensive

**Index
locking**

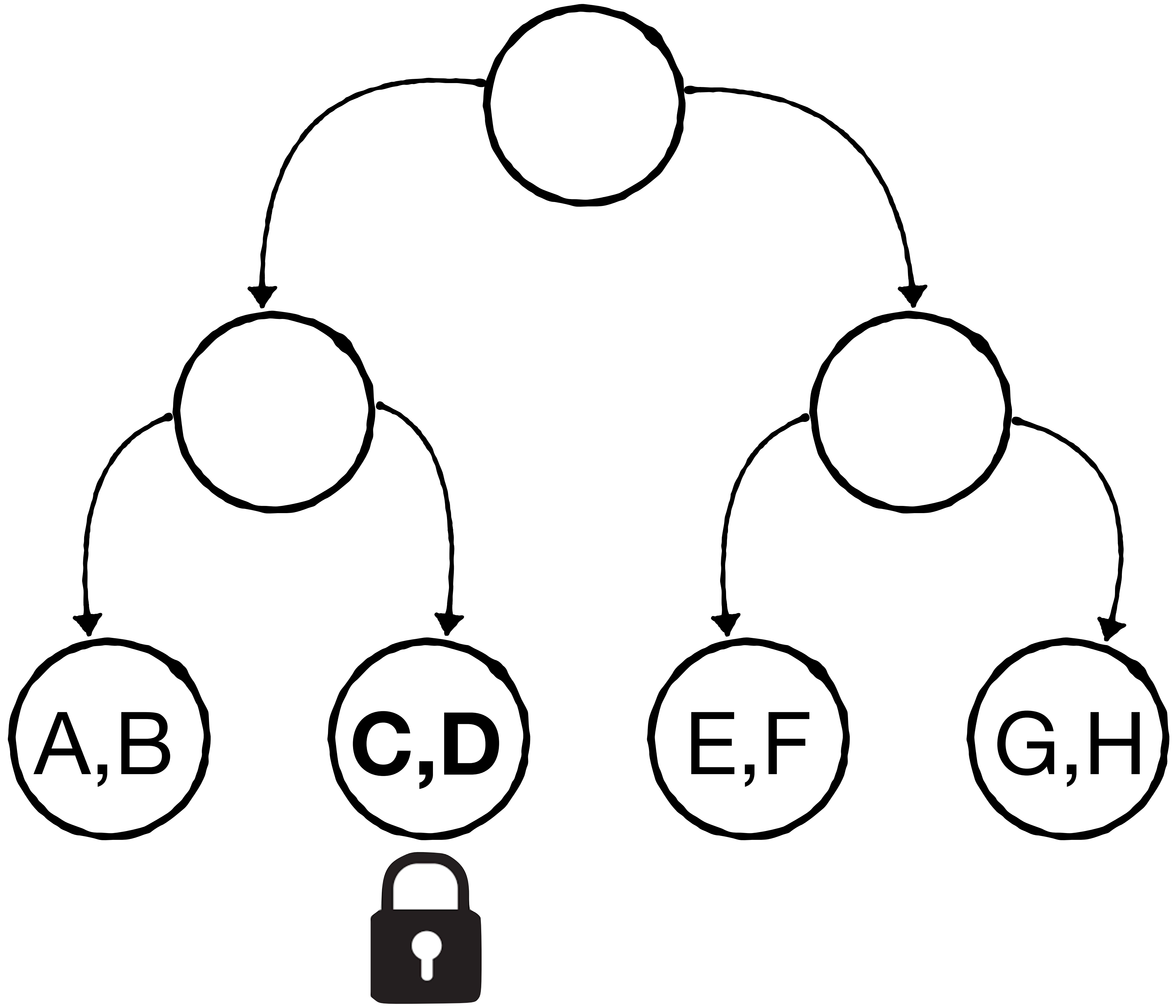


**Good but
requires an index**

Index Locking

Lock B-tree leaf storing relevant range

Example: Select * from accounts where **ID="Cindy"**

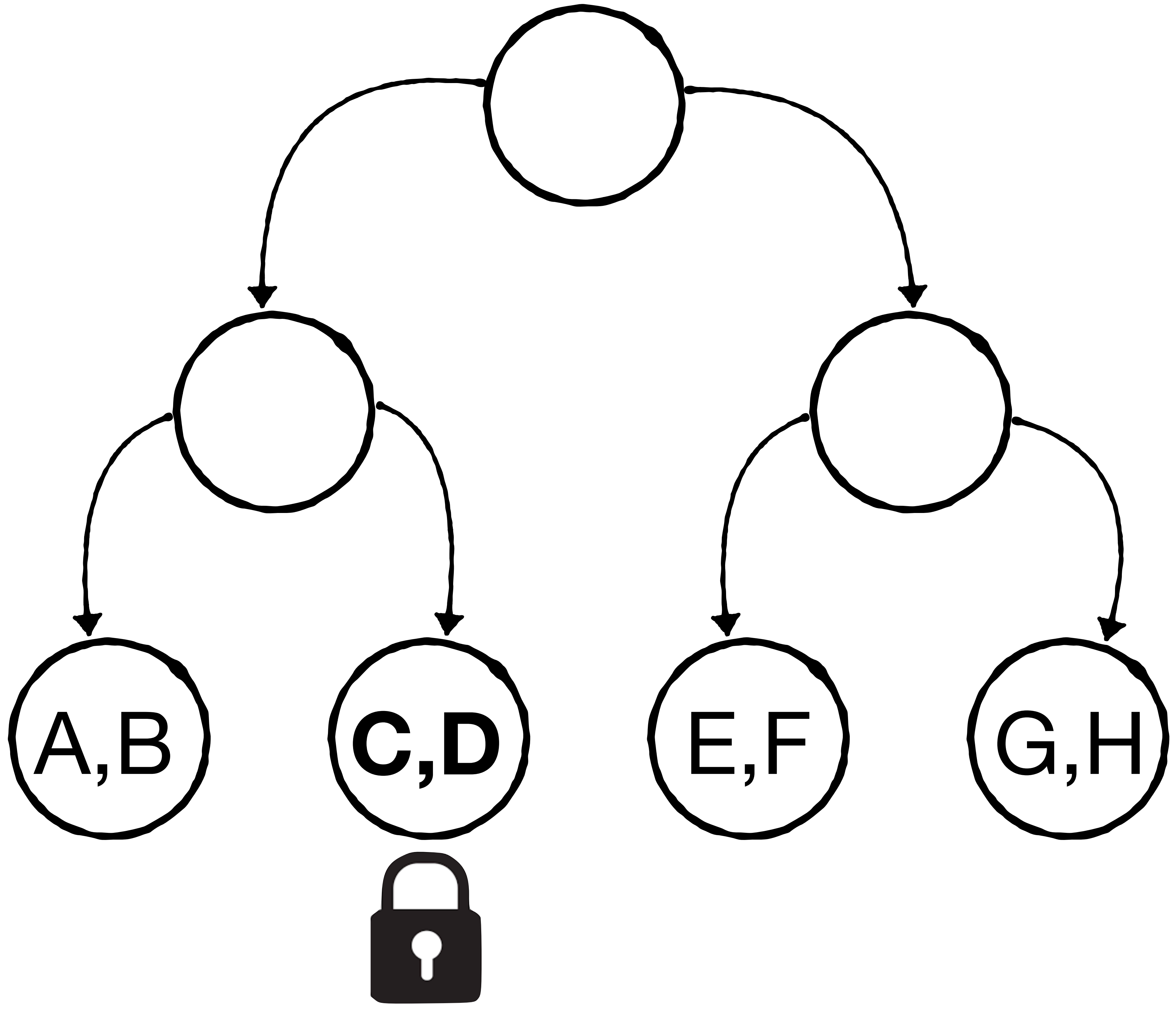


Index Locking

Lock B-tree leaf storing relevant range

Example: Select * from accounts
where **ID="Cindy"**

So we do not have to lock the whole table as long as we have an index :)



Example 1

Interest & Payments

Dirty reads

Exclusive write locks

Example 2

Updates & Statistics

Unrepeatable reads

Shared read locks

Example 3

Inserts & Statistics

Phantom Read

Range locks

Example 1

Interest & Payments

Dirty reads

Exclusive write locks

Example 2

Updates & Statistics

Unrepeatable reads

Shared read locks

Example 3

Inserts & Statistics

Phantom Read

Range locks

More correct but slower

Example 1

Interest & Payments

Dirty reads

Exclusive write locks

Example 2

Updates & Statistics

Unrepeatable reads

Shared read locks

Example 3

Inserts & Statistics

Phantom Read

Range locks

Compromising correctness is fine for some applications

ANSI/ISO **Transaction Isolation Levels** - part of the SQL standard

	Dirty read	**Unrepeatable read**	**Phantom reads**
Read uncommitted			
Read committed			
Repeatable reads			
Serializable			

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

	Dirty read	Unrepeatable read	Phantom reads
Read uncommitted	**Possible**	**Possible**	**Possible**
Read committed			
Repeatable reads			
Serializable			

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

	Dirty read	Unrepeatable read	Phantom reads
Read uncommitted	Possible	Possible	Possible
Read committed	**Not Possible**	**Possible**	**Possible**
Repeatable reads			
Serializable			

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

	Dirty read	Unrepeatable read	Phantom reads
Read uncommitted	Possible	Possible	Possible
Read committed	Not Possible	Possible	Possible
Repeatable reads	**Not Possible**	**Not Possible**	**Possible**
Serializable			

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

	Dirty read	Unrepeatable read	Phantom reads
Read uncommitted	Possible	Possible	Possible
Read committed	Not Possible	Possible	Possible
Repeatable reads	Not Possible	Not Possible	Possible
Serializable	**Not Possible**	**Not Possible**	**Not Possible**

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

Read uncommitted	**Exclusive write locks not held until a transaction ends**
Read committed	
Repeatable reads	
Serializable	

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

Read uncommitted	Exclusive write locks not held until a transaction ends
Read committed	**Shared read locks not held until a transaction ends**
Repeatable reads	
Serializable	

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

Read uncommitted	Exclusive write locks not held until a transaction ends
Read committed	Shared read locks not held until a transaction ends
Repeatable reads	**Range locks not employed (e.g., no tree/table locking)**
Serializable	

ANSI/ISO Transaction Isolation Levels - part of the SQL standard

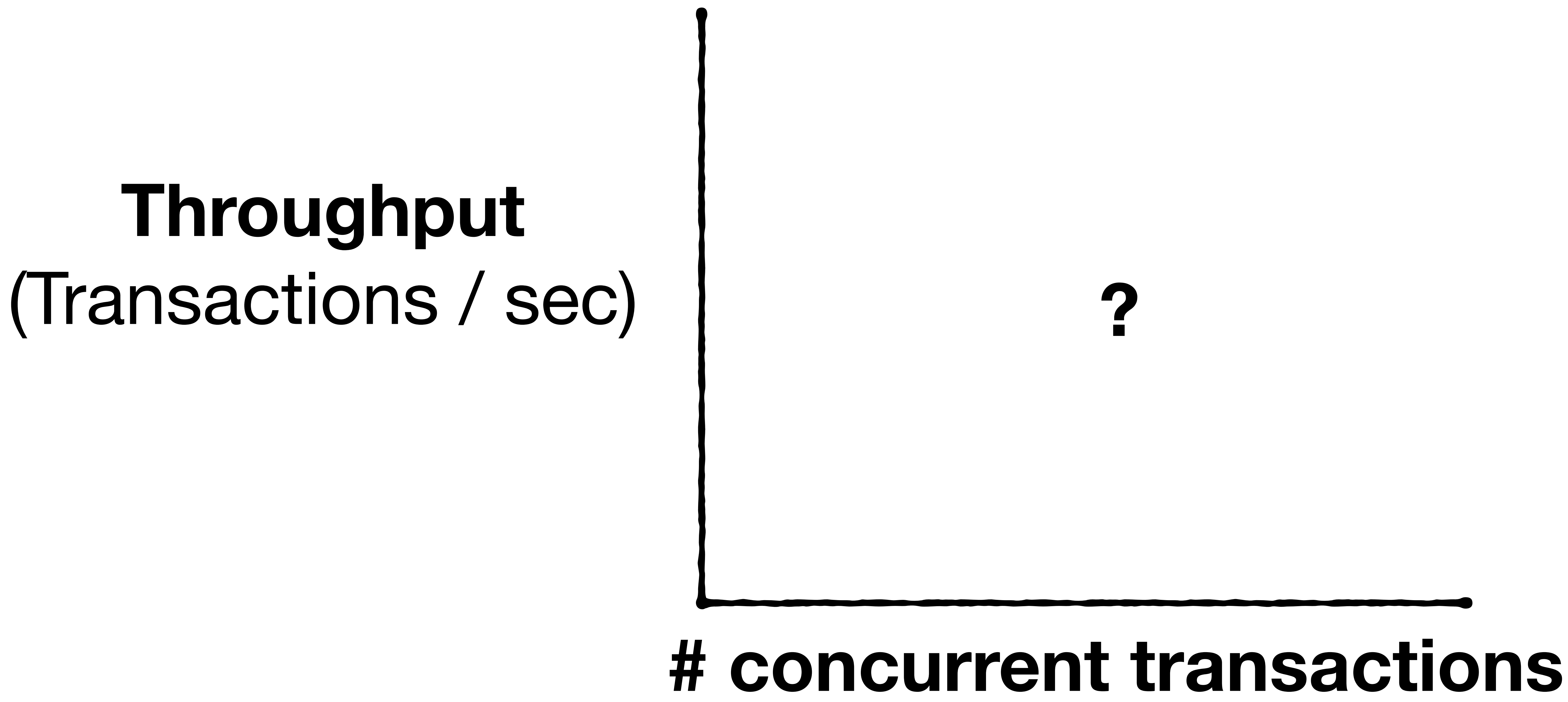
Read uncommitted	Exclusive write locks not held until a transaction ends
Read committed	Shared read locks not held until a transaction ends
Repeatable reads	Range locks not employed (e.g., no tree/table locking)
Serializable	**Everything is fully correct**

The default in many relational DB systems

Repeatable reads

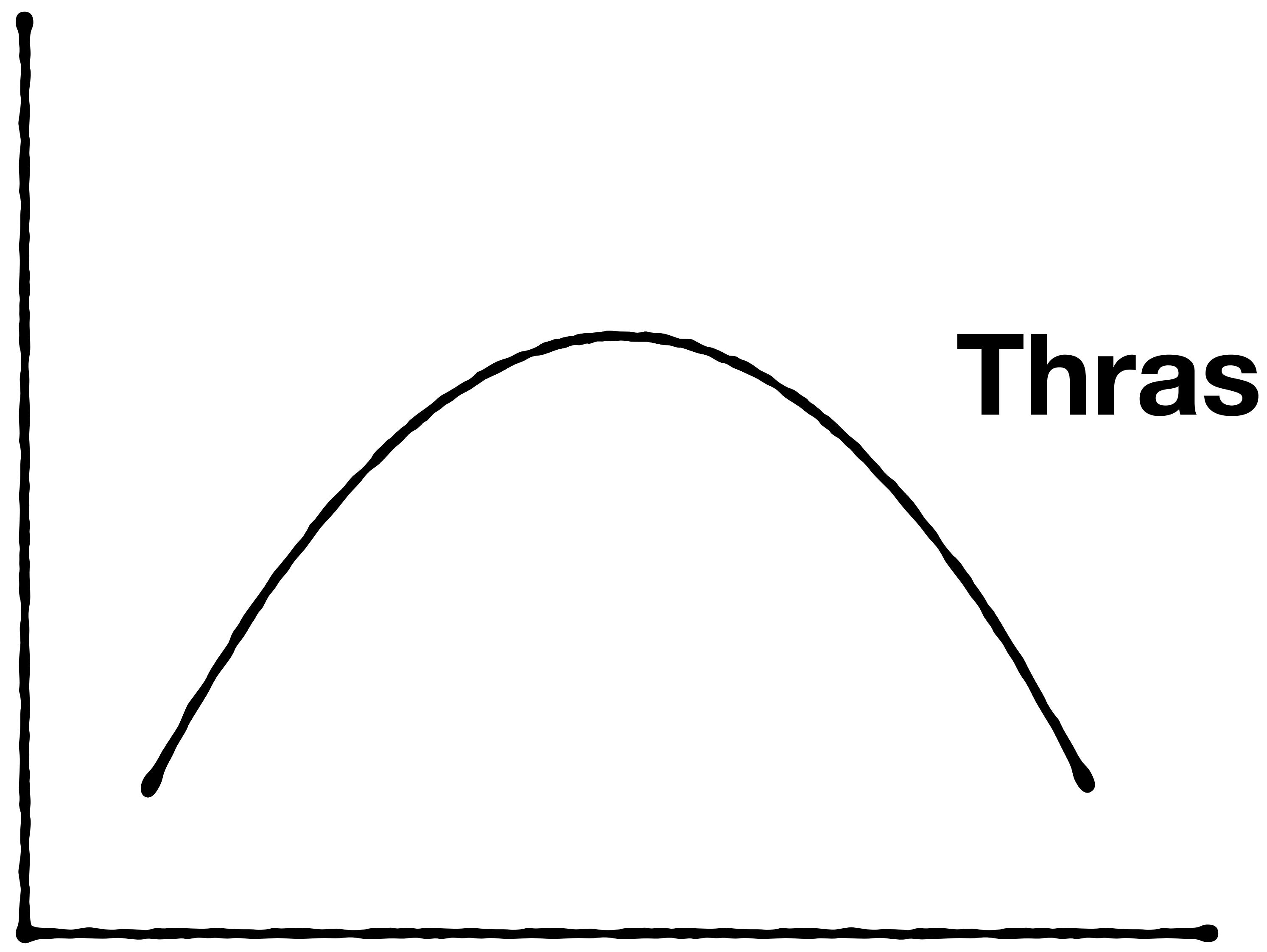
Range locks not employed (e.g., no tree/table locking)

What do you think this curve should look like?



What do you think this curve should look like?

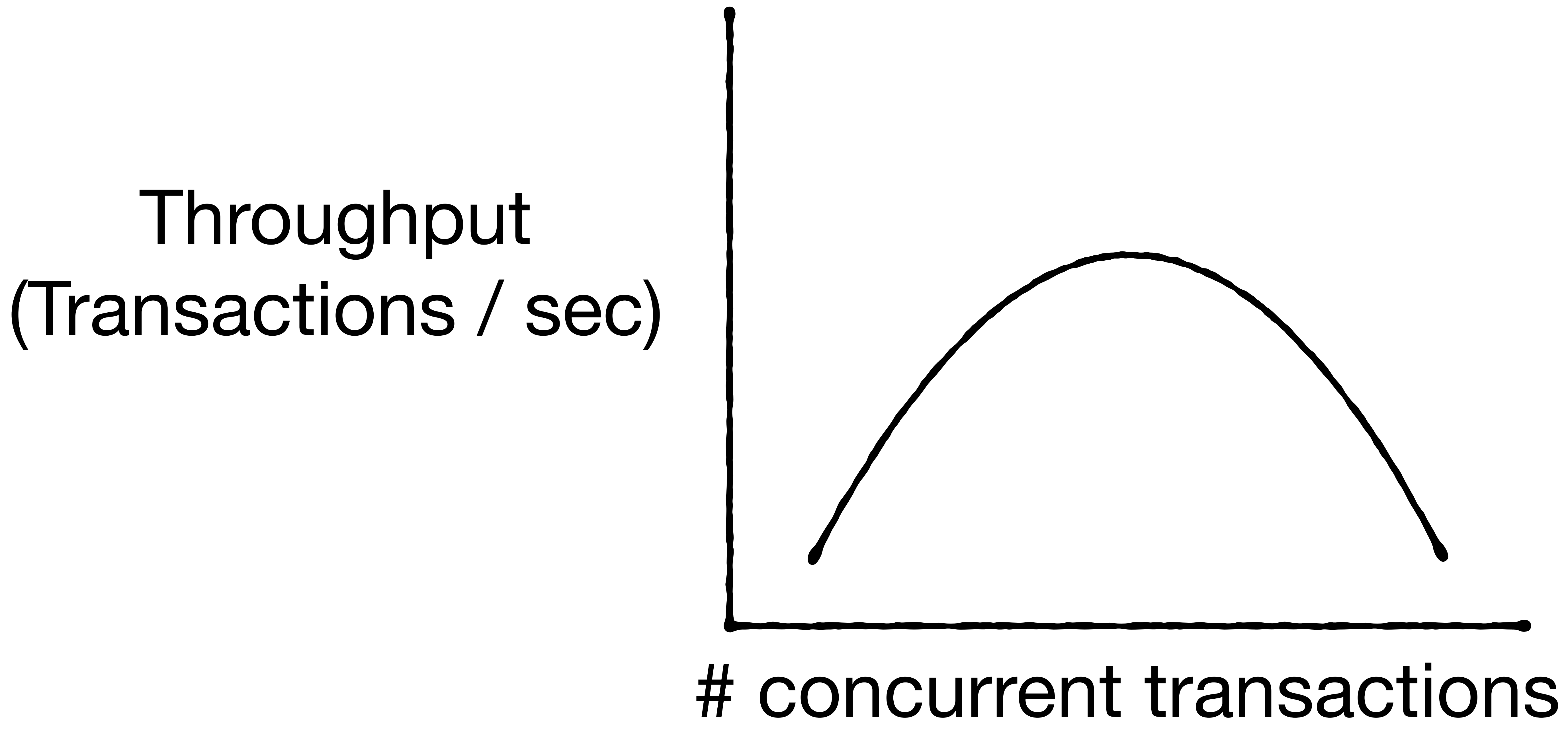
Throughput
(Transactions / sec)



Thrashing due to locks

concurrent transactions

What do you think this curve should look like?

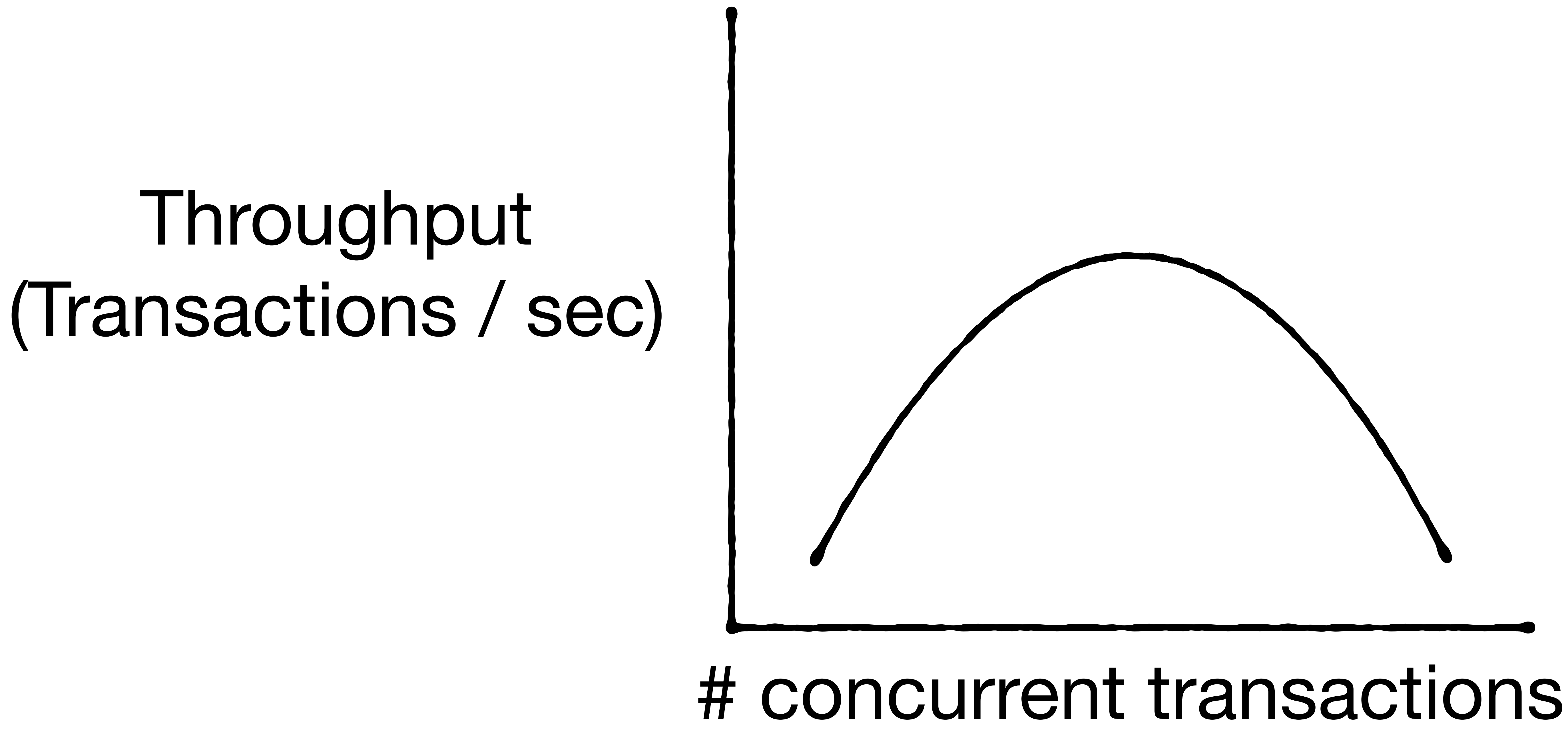


How can we address this?

(1)

(2)

What do you think this curve should look like?



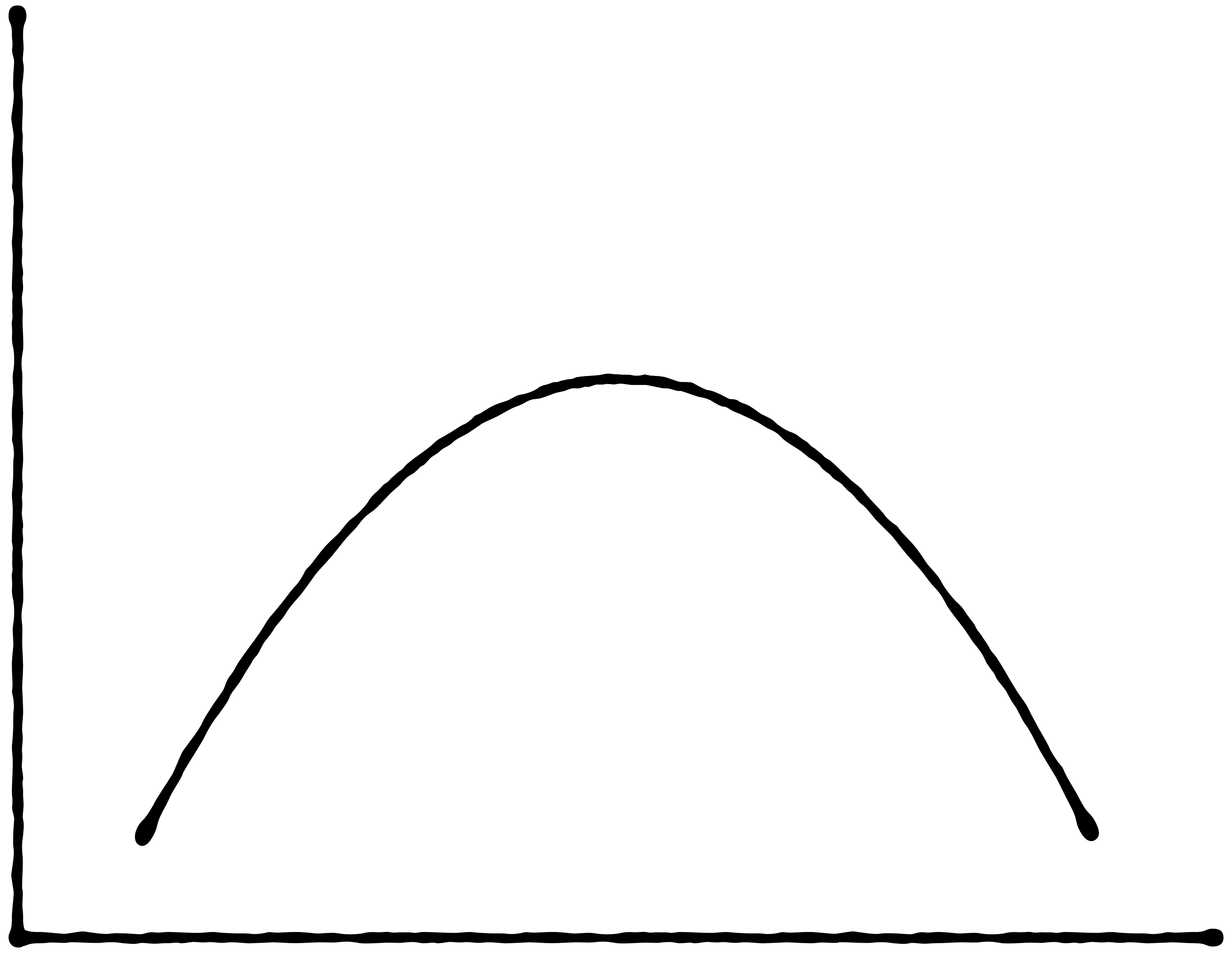
How can we address this?

(1) Design the application such that transactions are short

(2)

What do you think this curve should look like?

Throughput
(Transactions / sec)



concurrent transactions

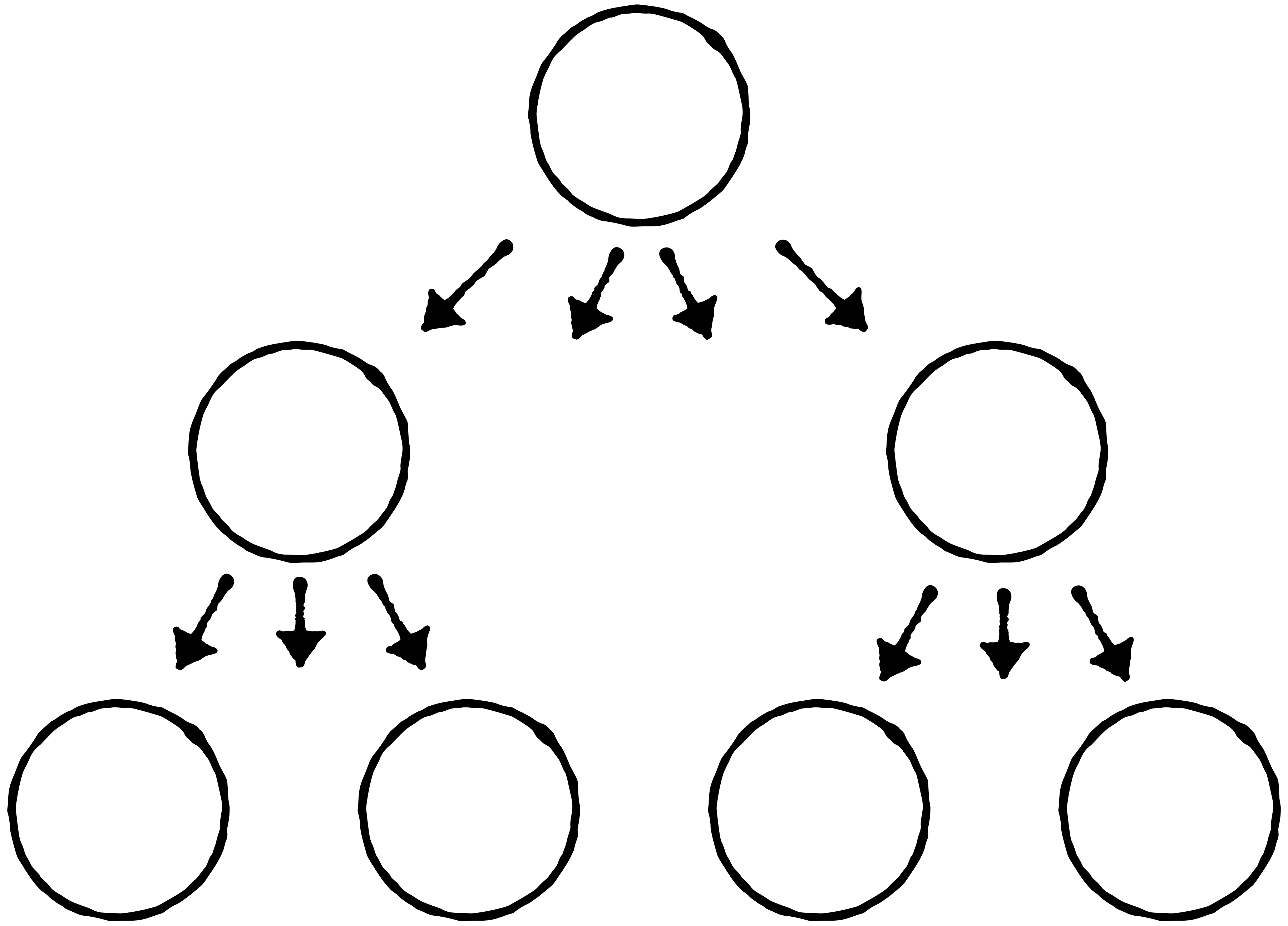
How can we address this?

(1) Design the application such that transactions are short

(2) **Restrict the maximum number of concurrent transactions**

With Strict Two-Phase Locking, a transaction locks all objects on its path until it commits

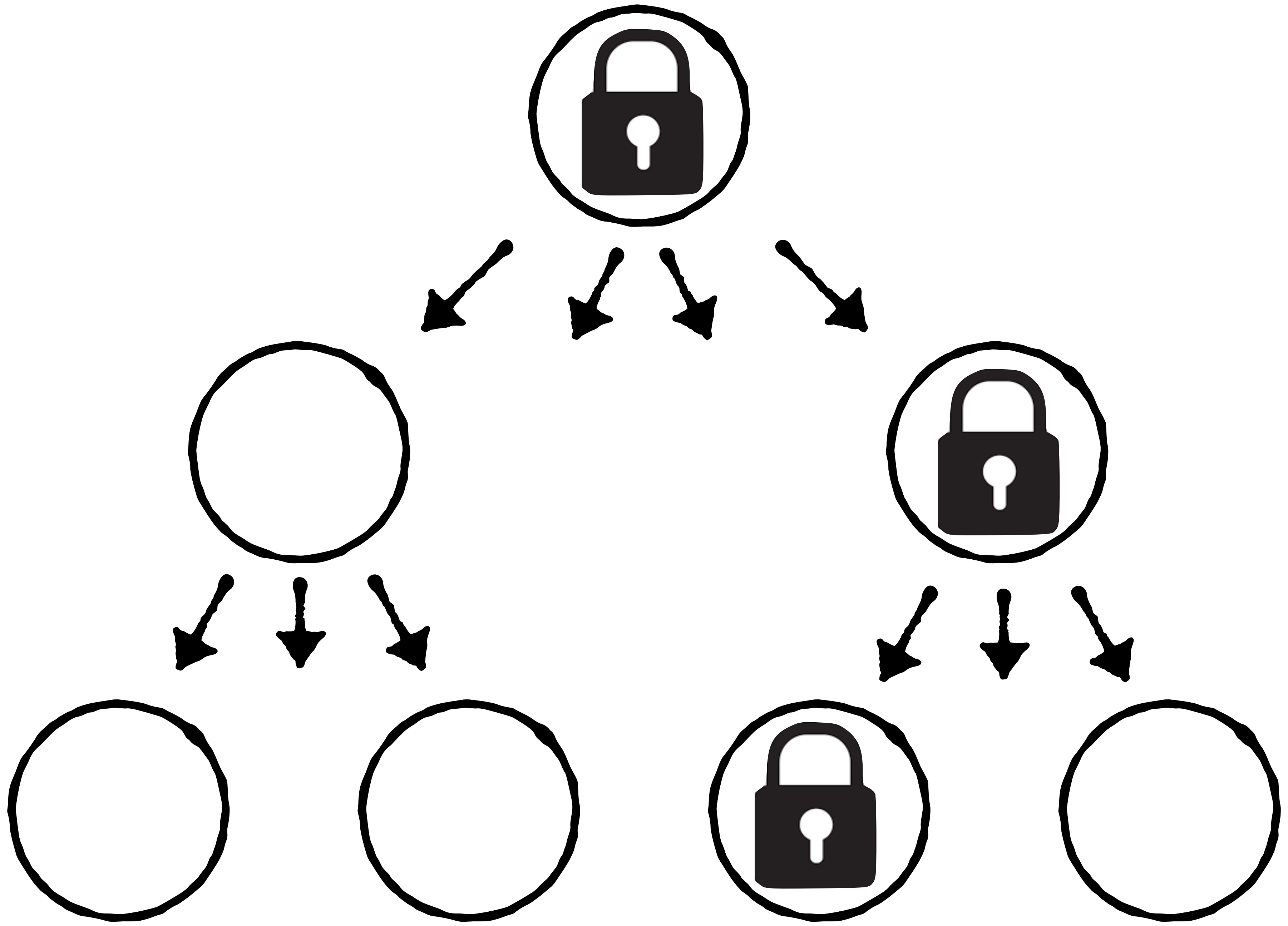
Why is this a problem for B-tree update?



With Strict Two-Phase Locking, a transaction locks all objects on its path until it commits

Why is this a problem for B-tree update?

Lots of locking contention at root and upper levels

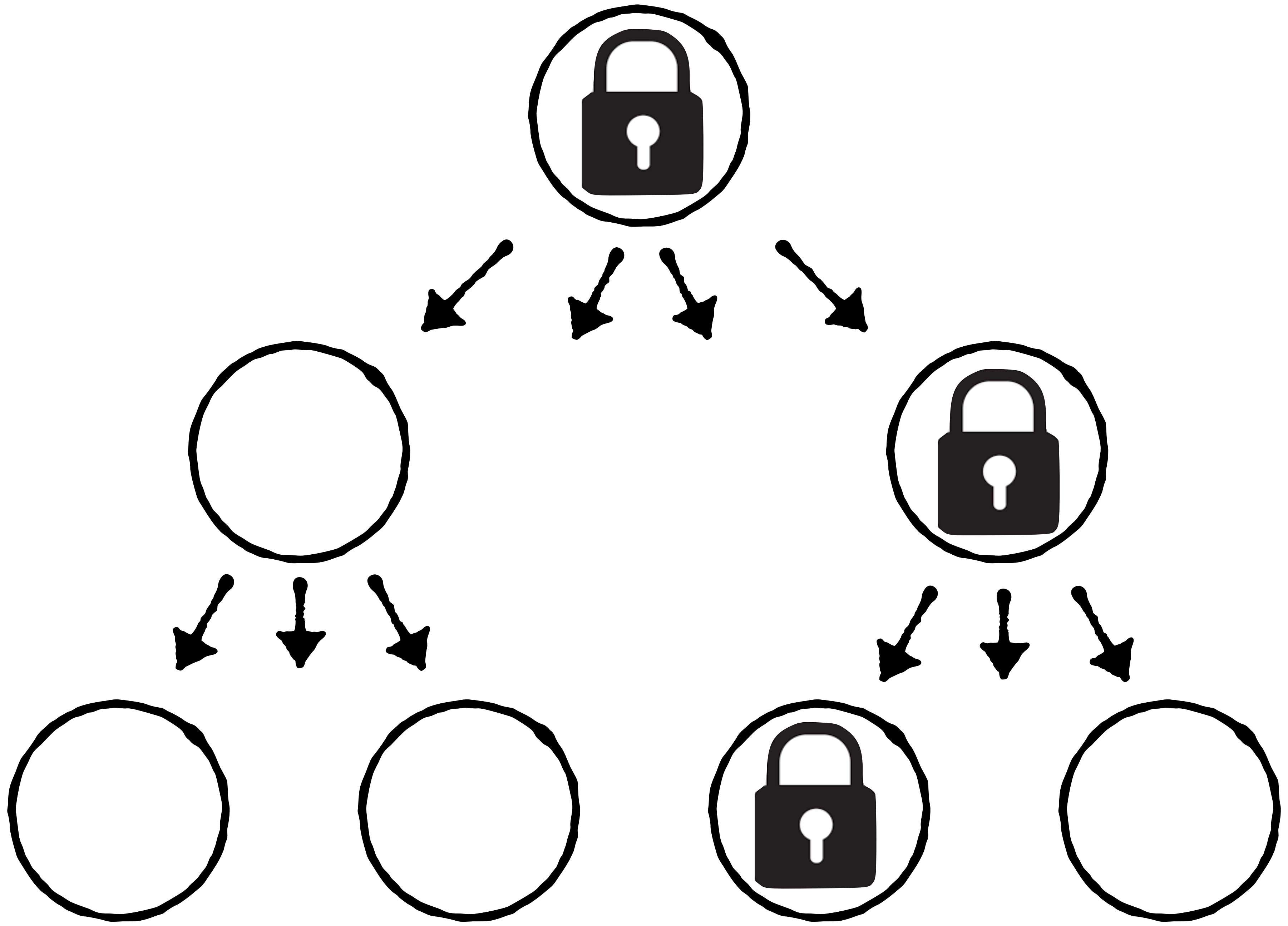


With Strict Two-Phase Locking, a transaction locks all objects on its path until it commits

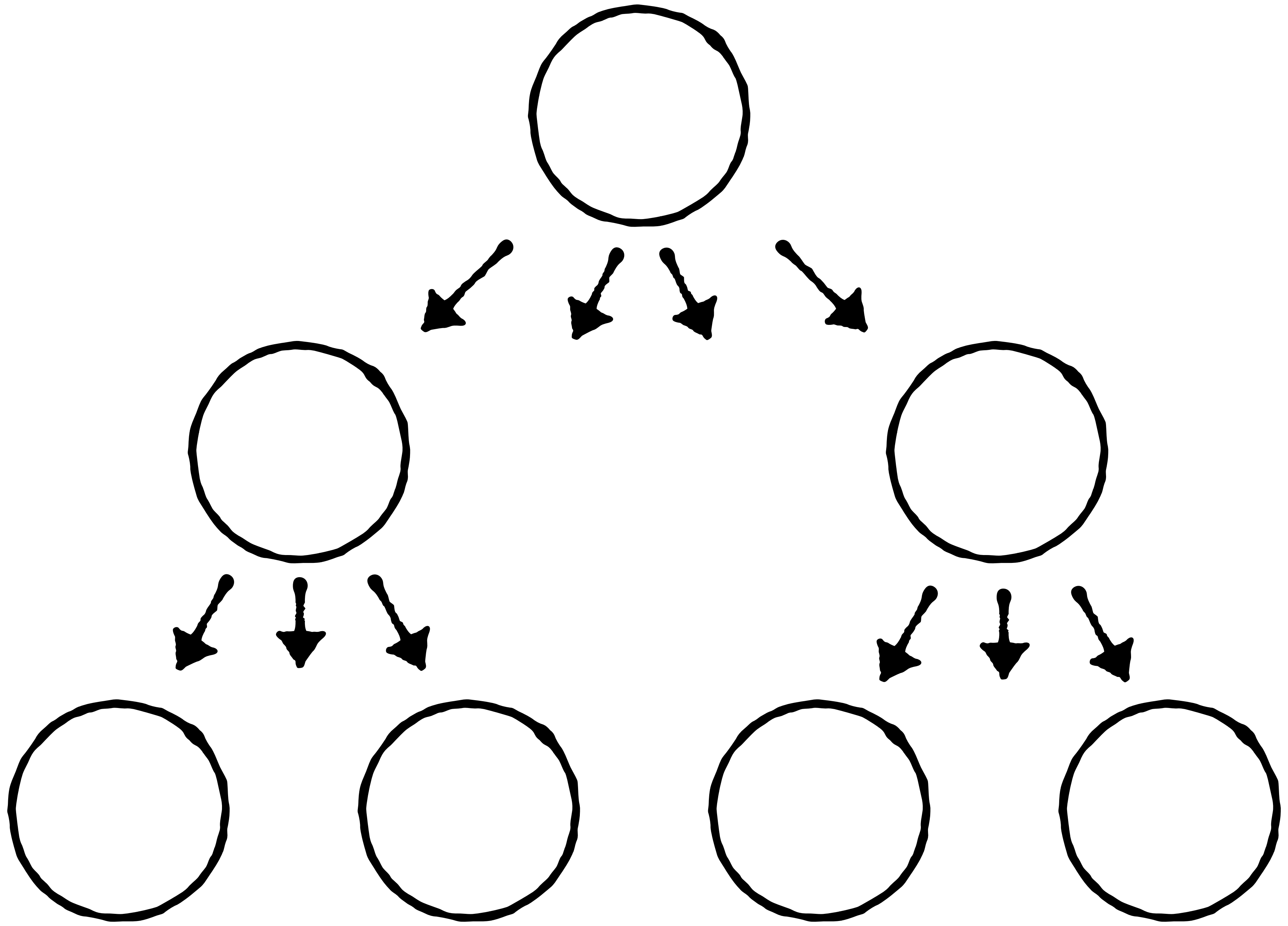
Why is this a problem for B-tree update?

Lots of locking contention at root and upper levels

Any solutions?

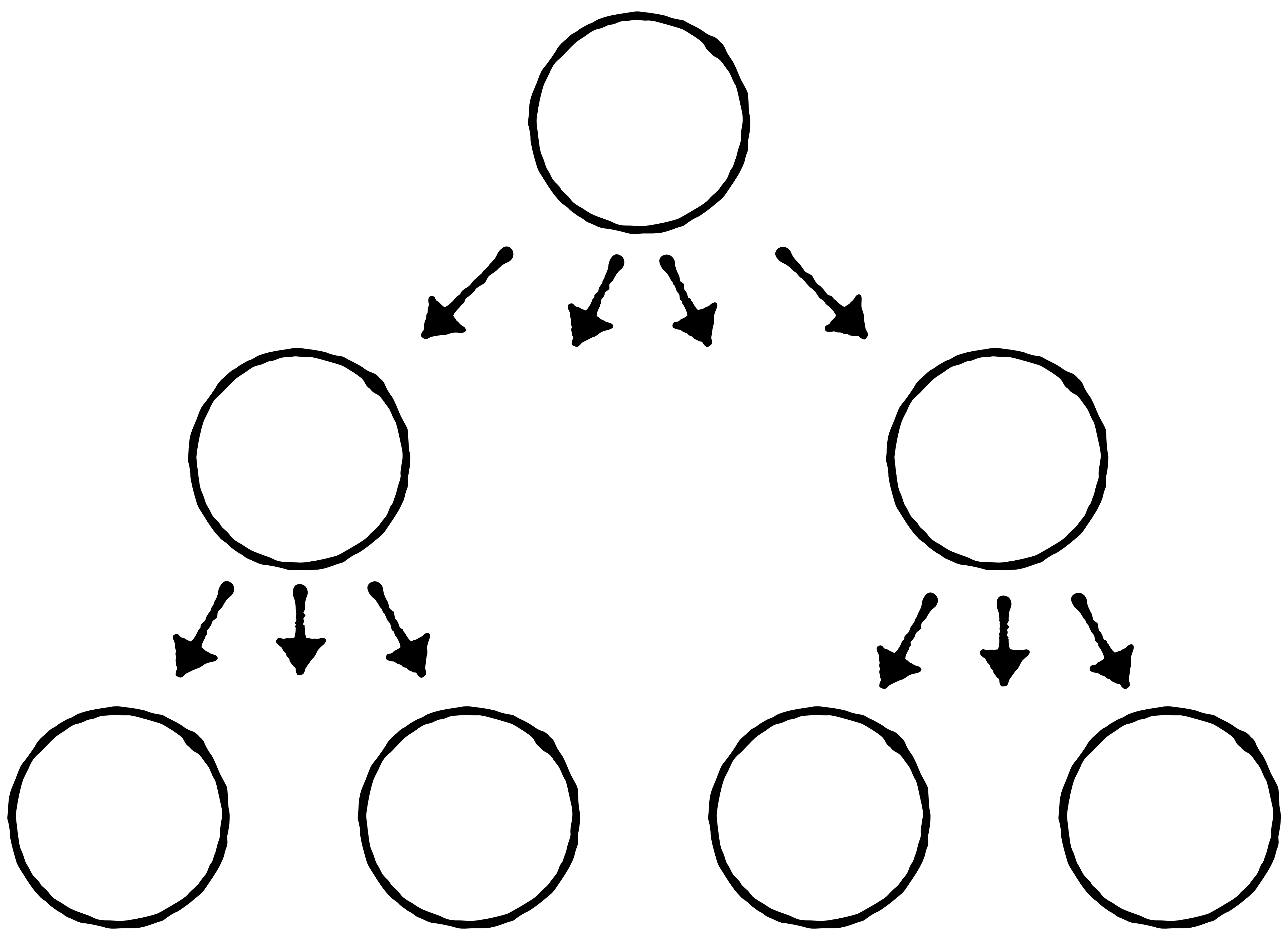


What must we lock at minimum to ensure correctness?



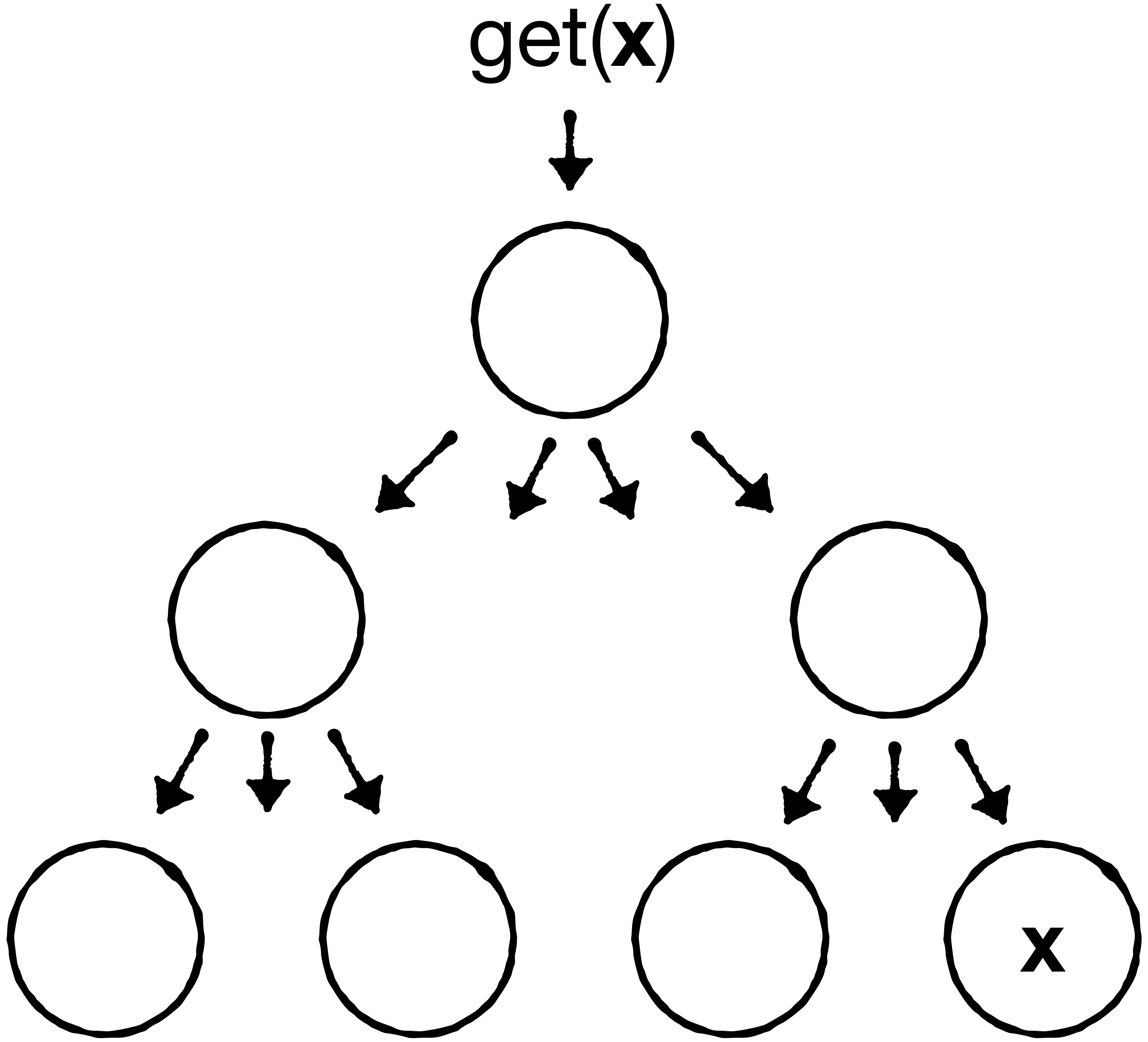
What must we lock at minimum to ensure correctness?

(1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



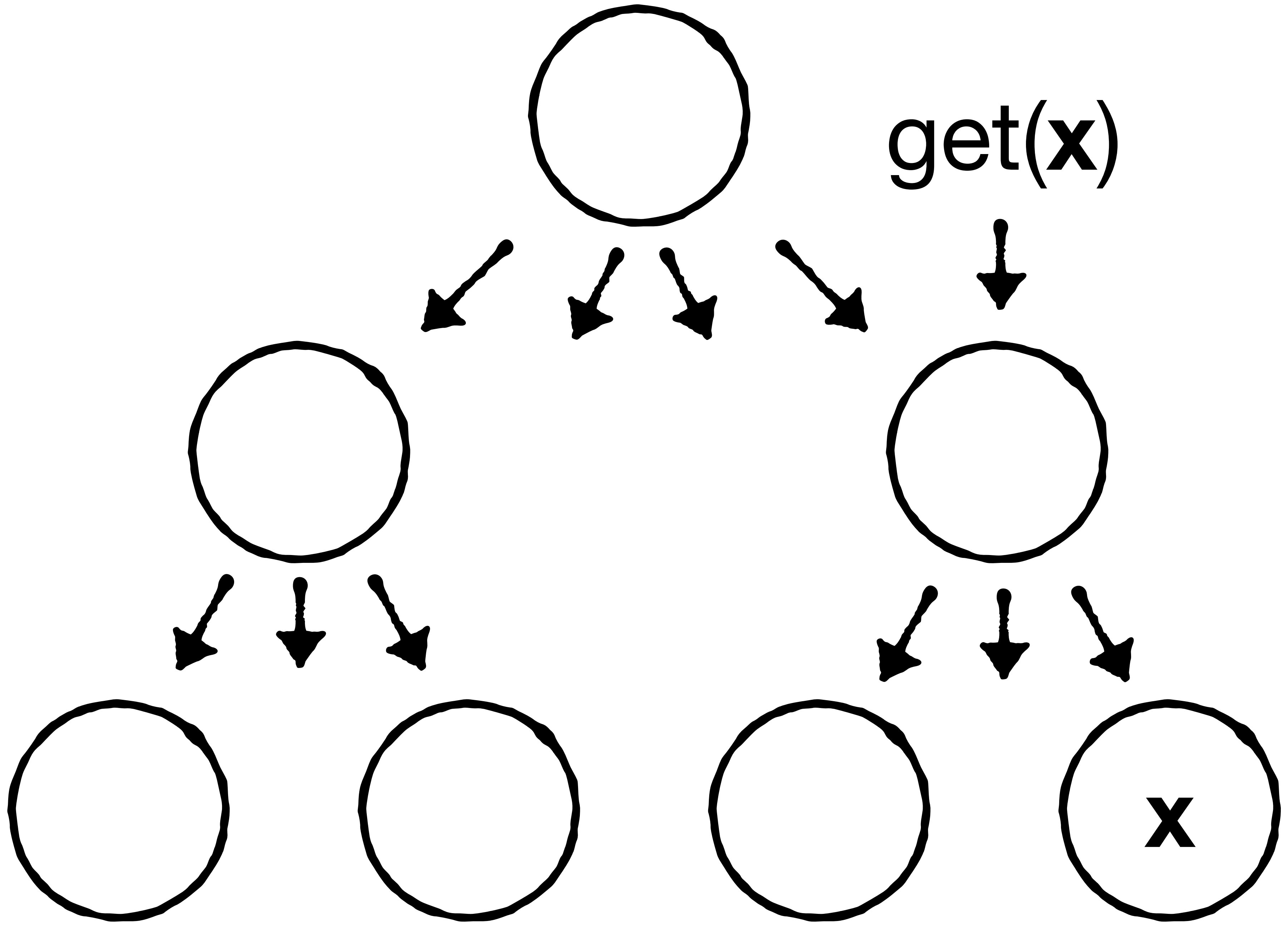
What must we lock at minimum to ensure correctness?

(1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



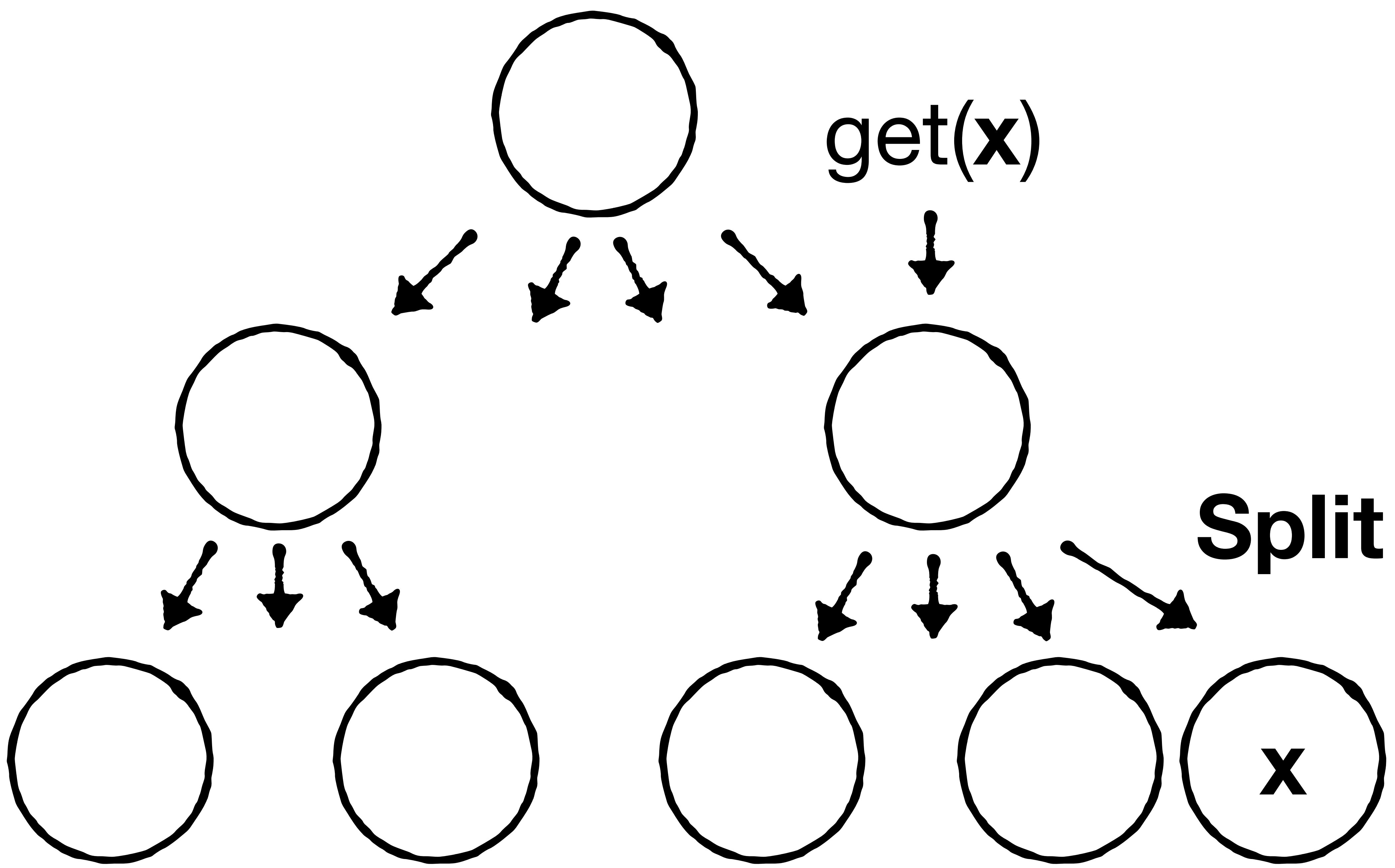
What must we lock at minimum to ensure correctness?

(1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



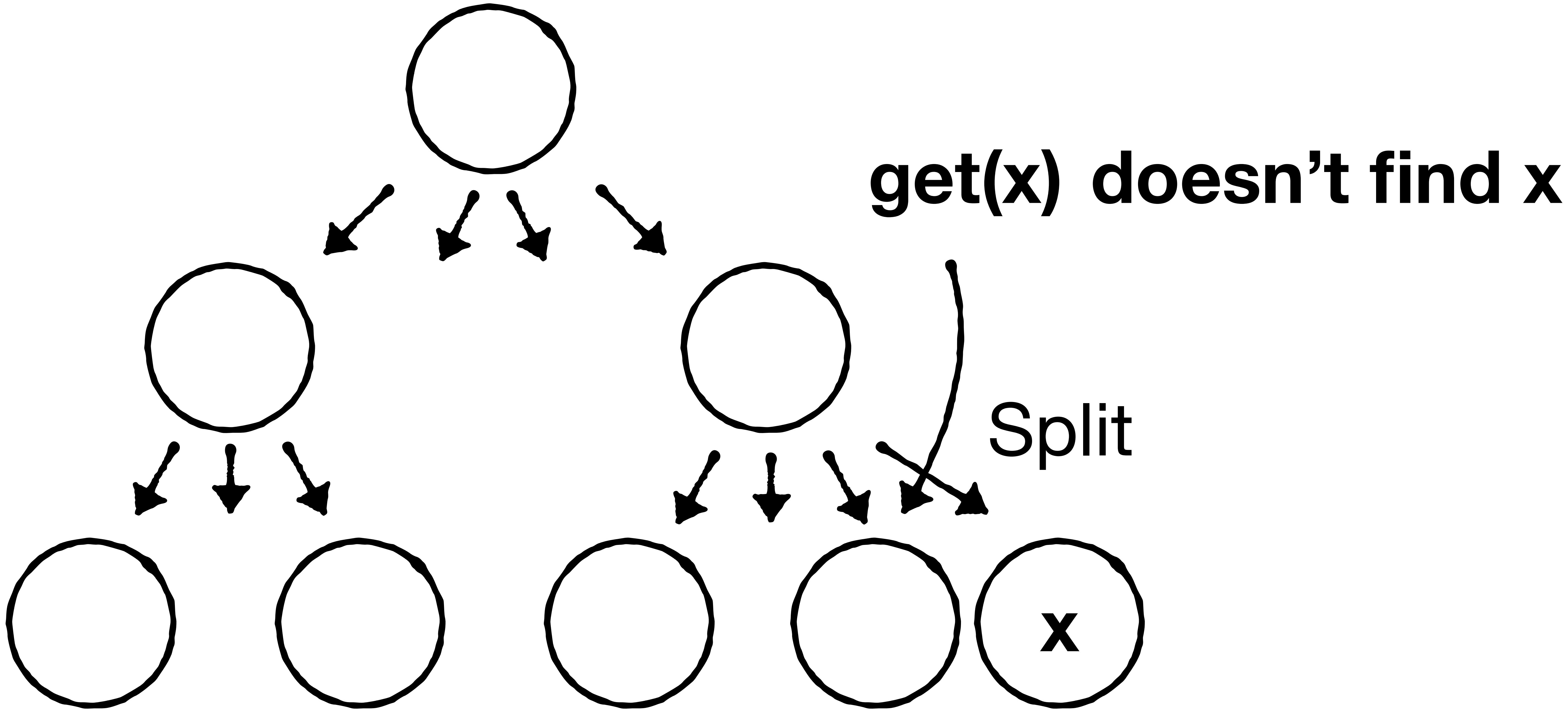
What must we lock at minimum to ensure correctness?

(1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



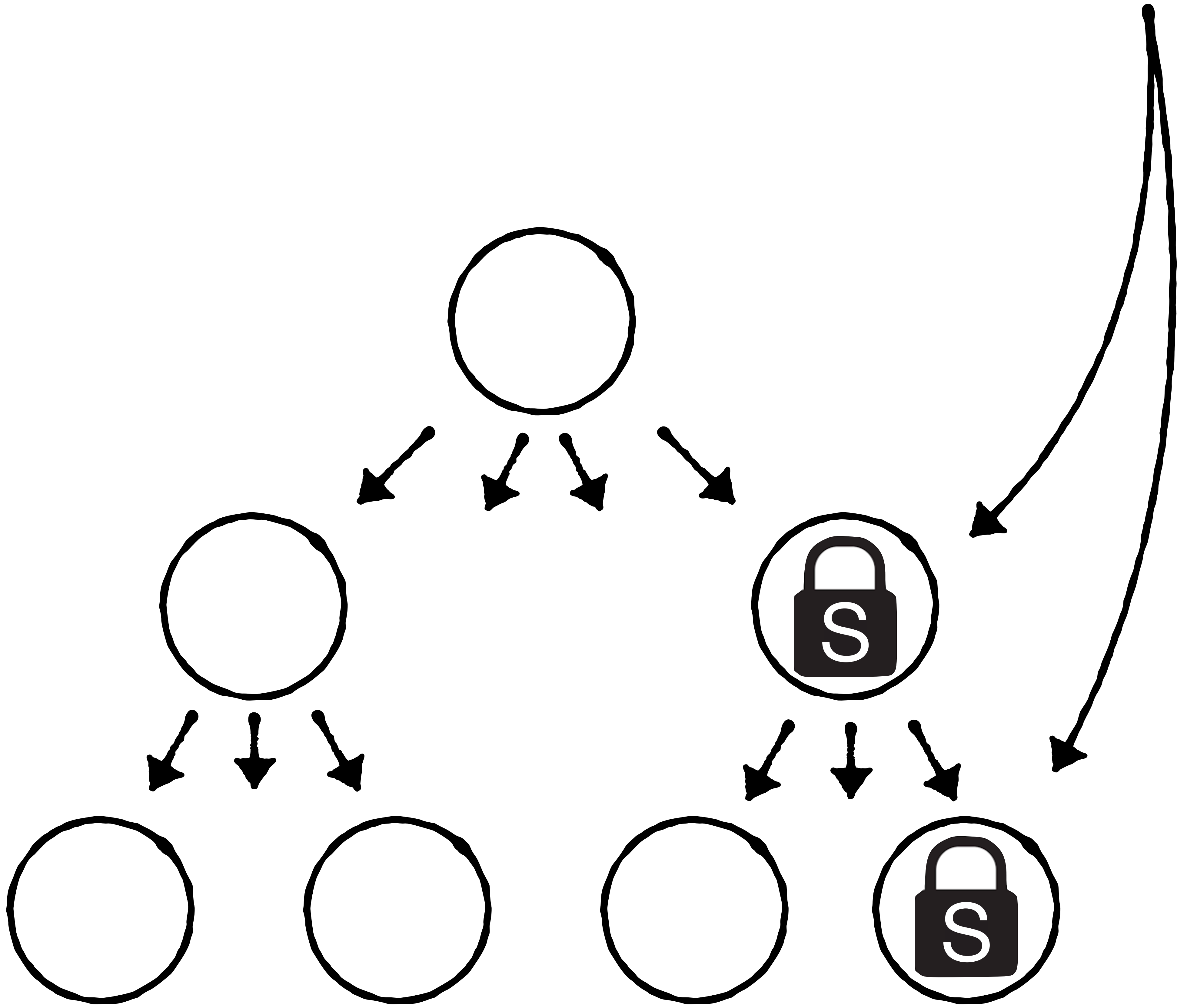
What must we lock at minimum to ensure correctness?

(1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



What must we lock at minimum to ensure correctness?

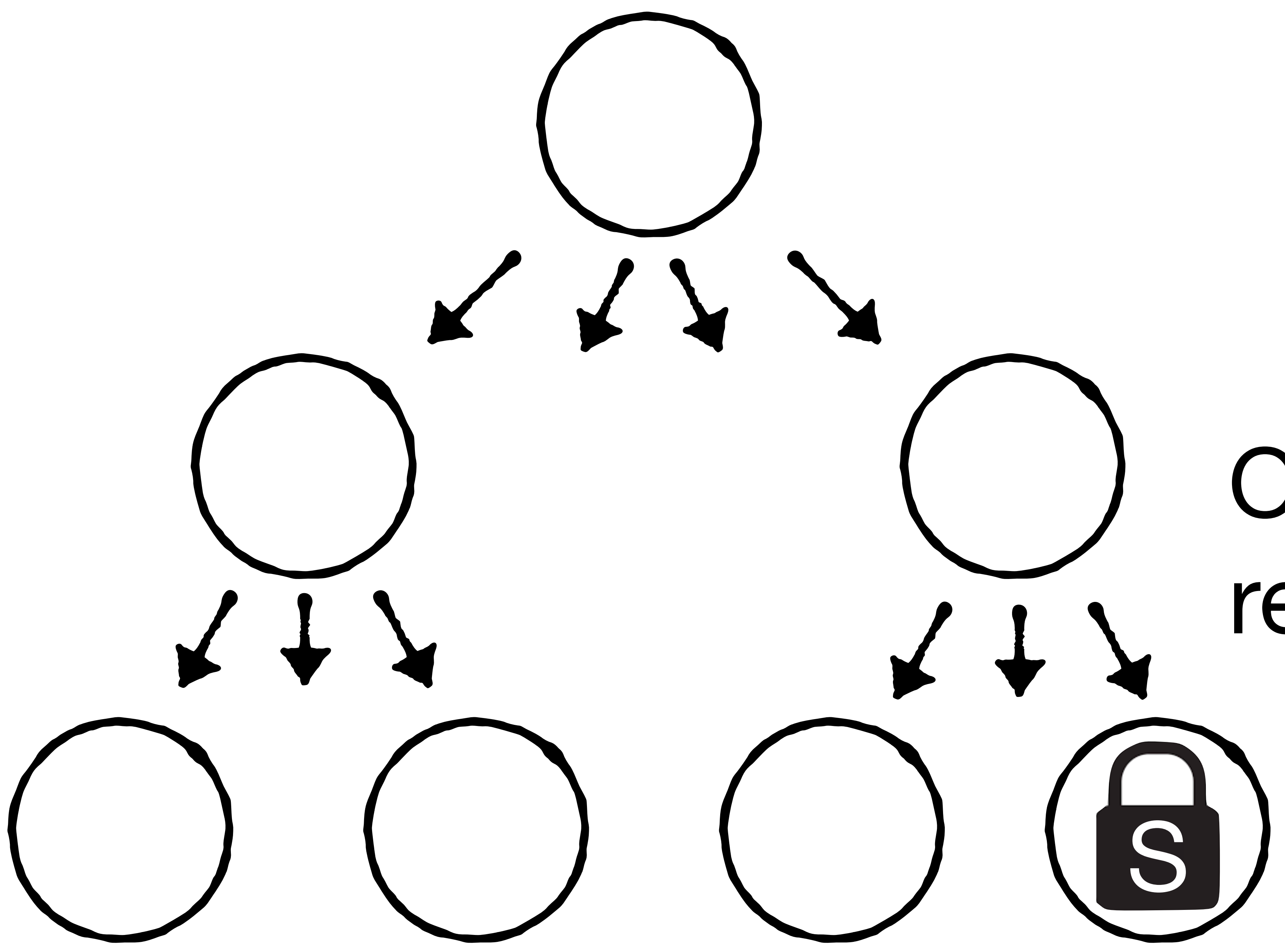
(1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



Double shared locks solve the problem

What must we lock at minimum to ensure correctness?

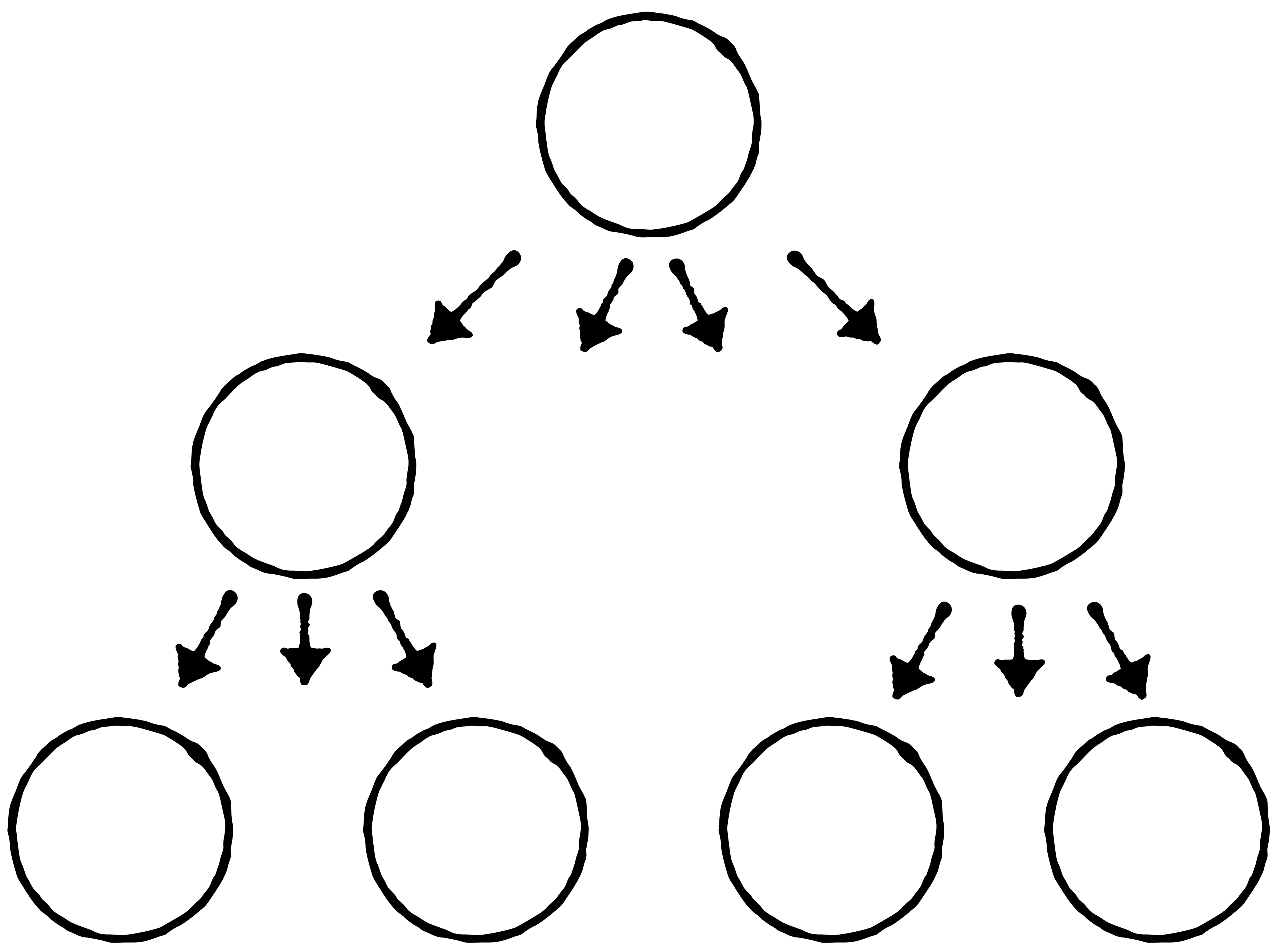
(1) We must maintain lock on parent as we acquire lock on child. Otherwise, another transaction may split child before we get to it



Once we hold lock on child, release lock on parent

What must we lock at minimum to ensure correctness?

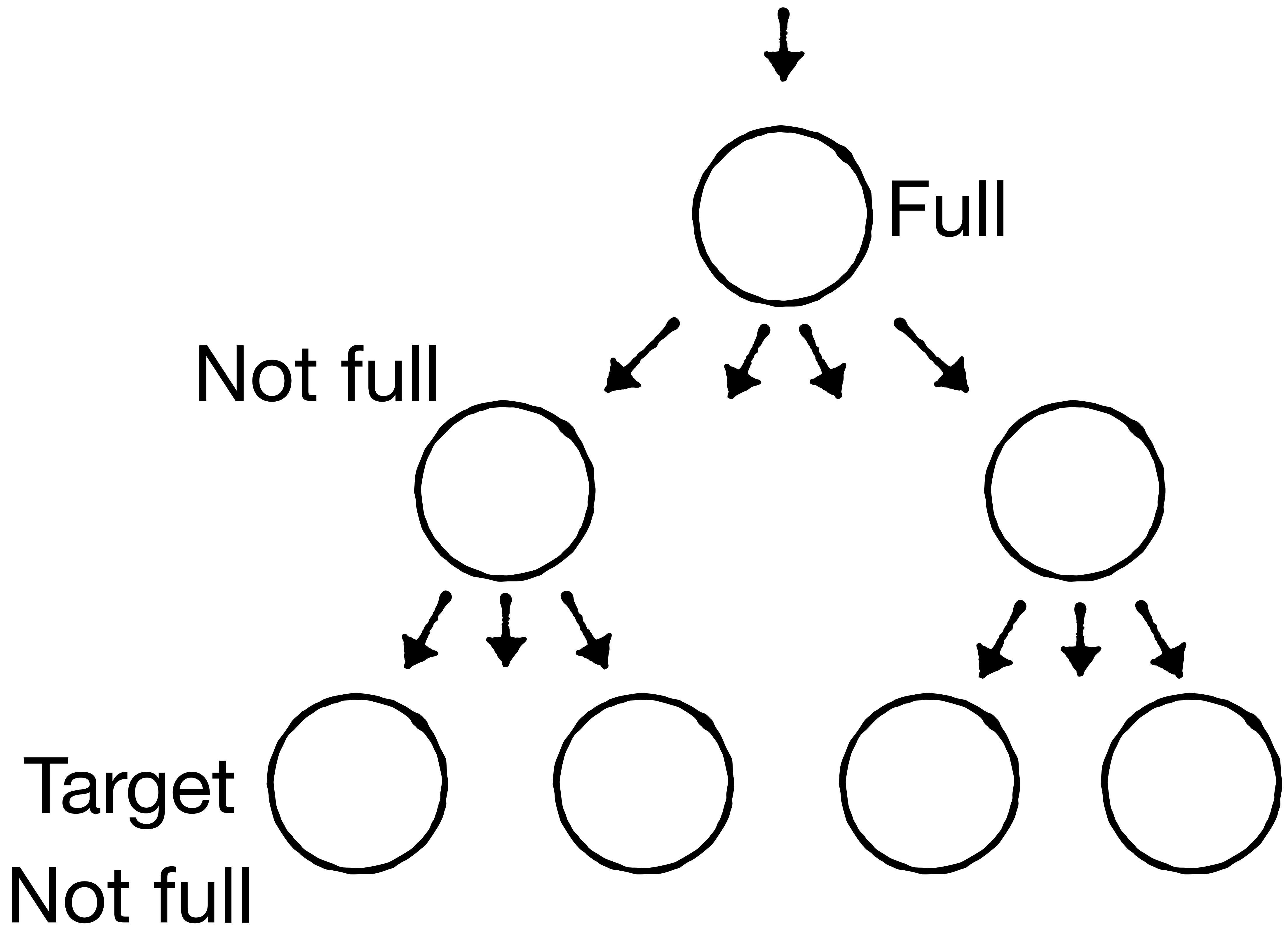
(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



What must we lock at minimum to ensure correctness?

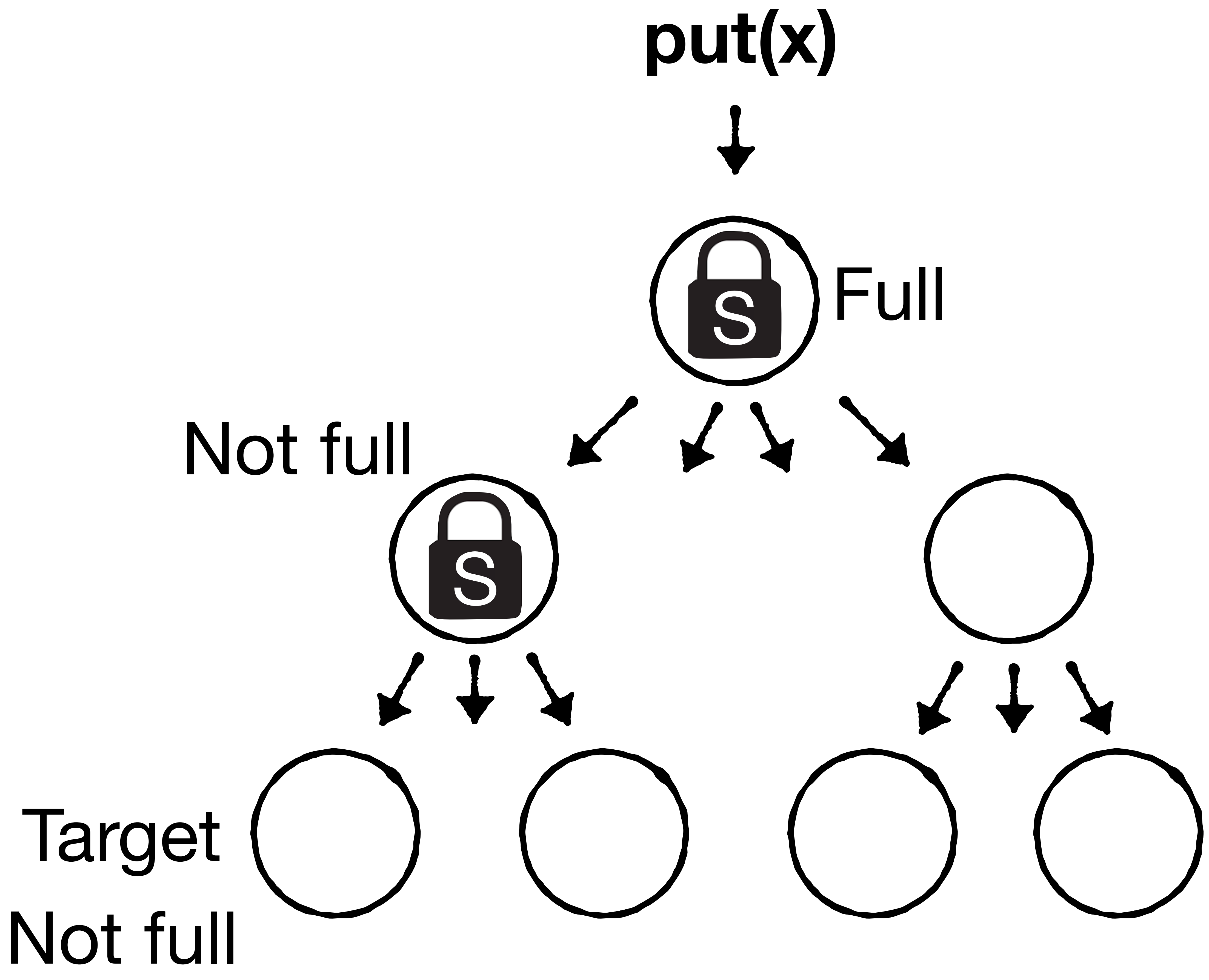
(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards

put(x)



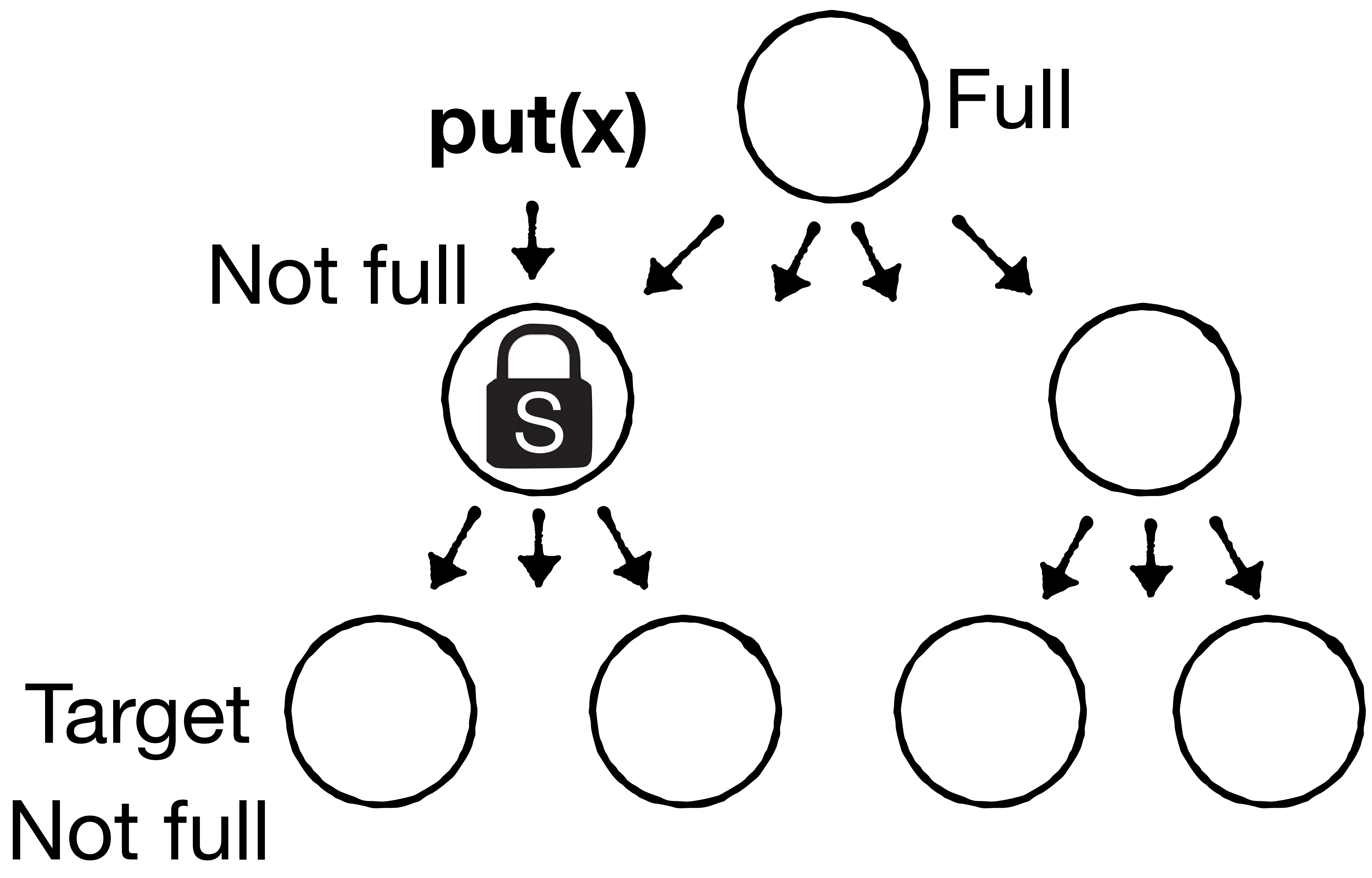
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



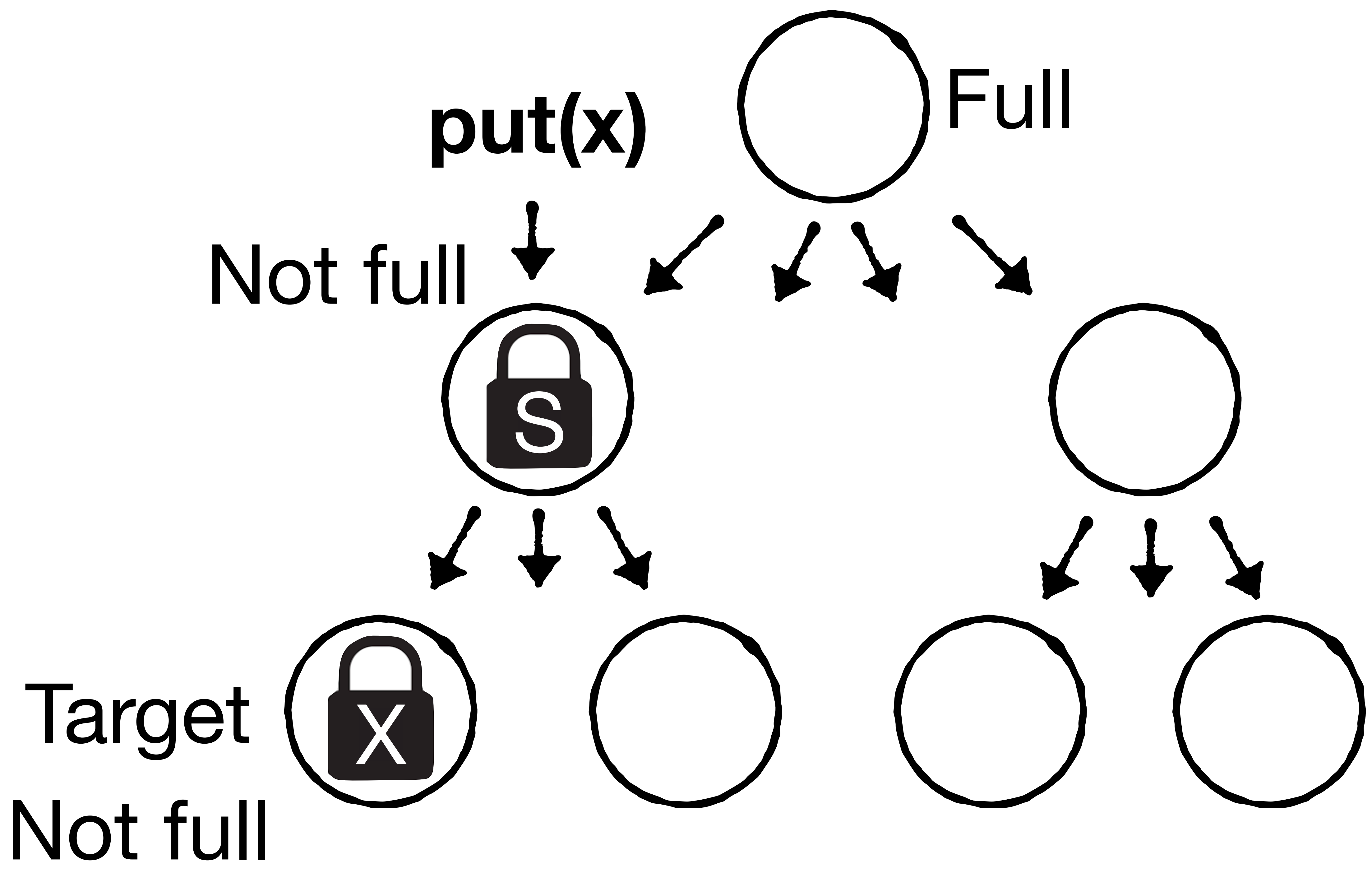
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



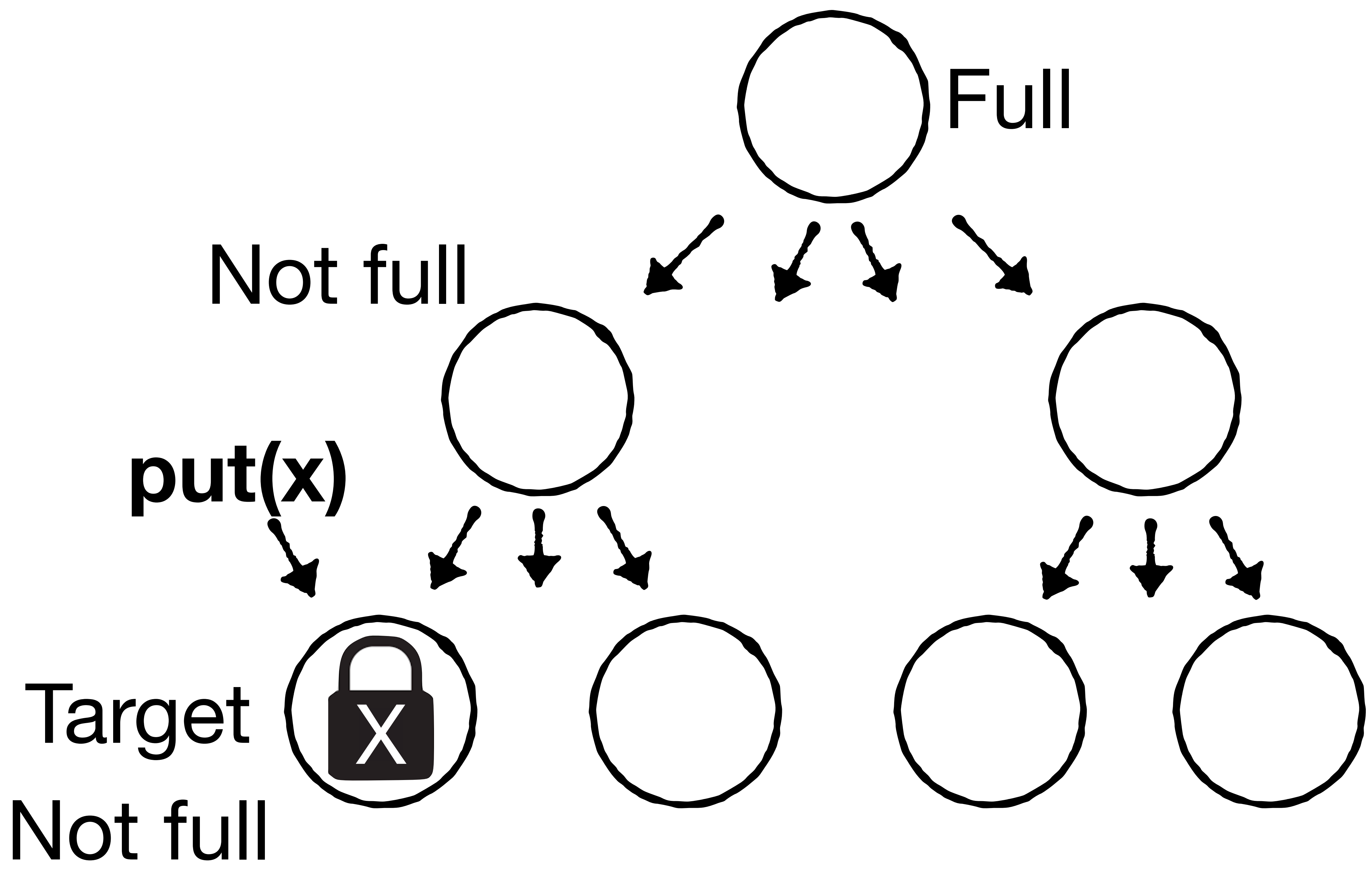
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



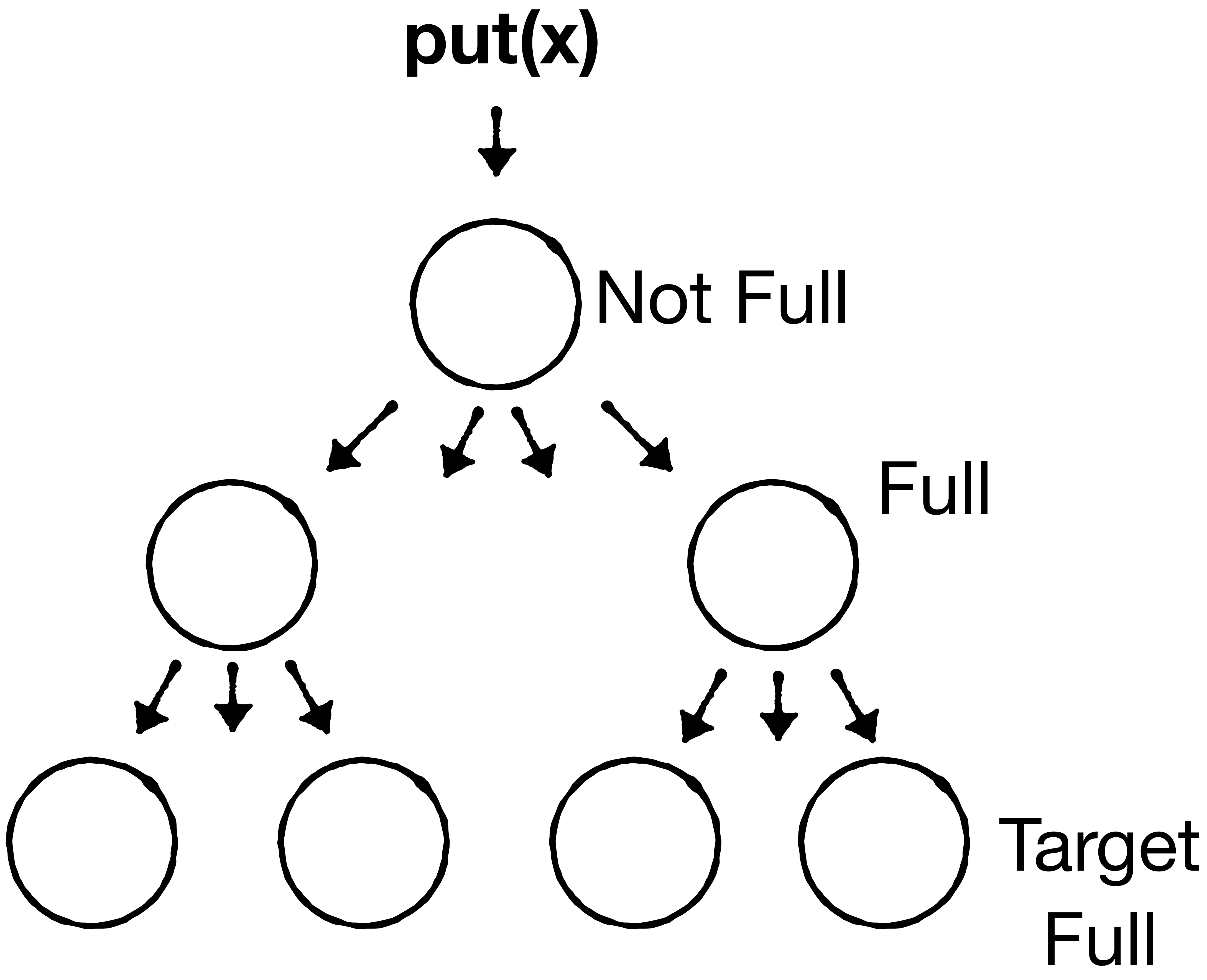
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



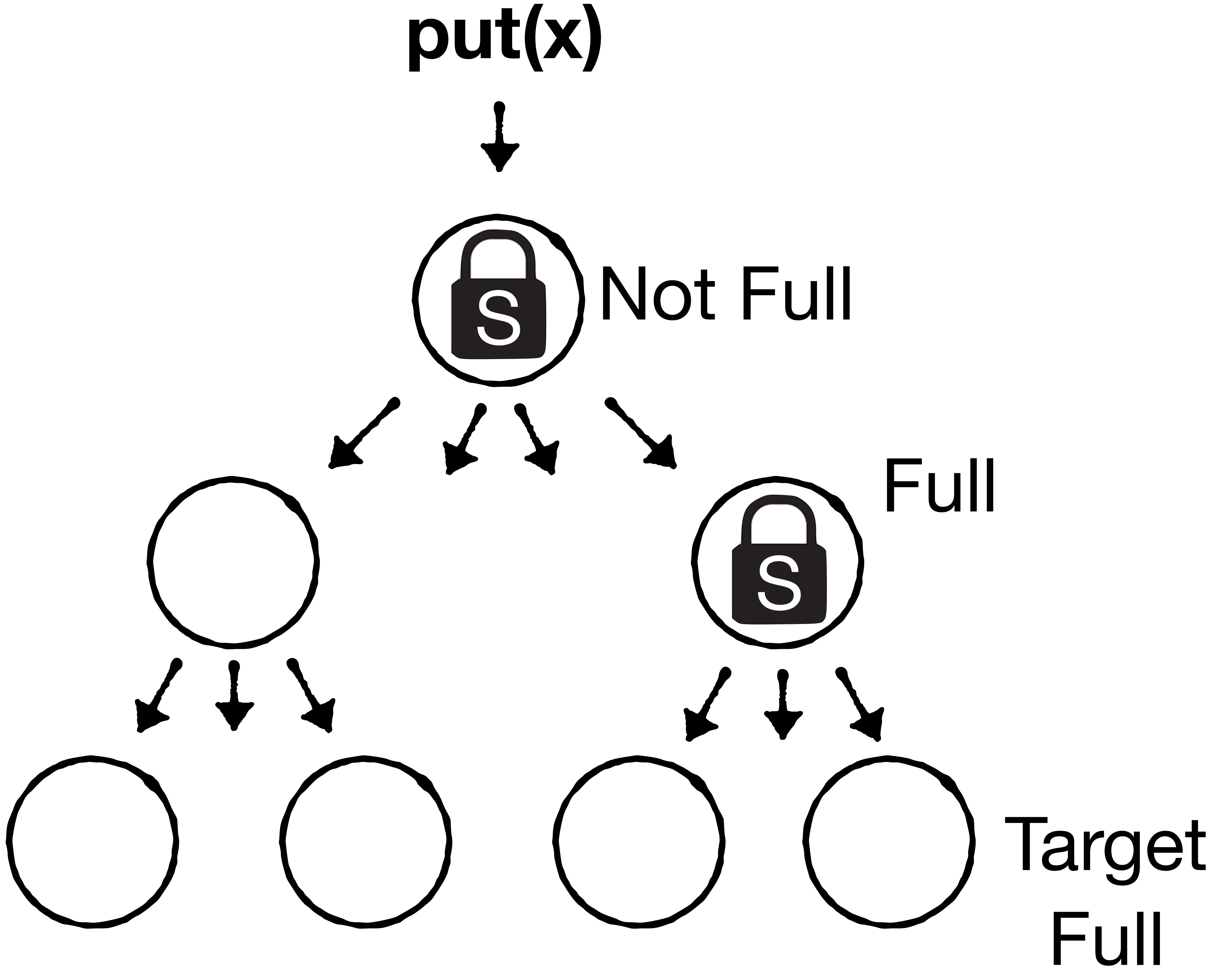
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



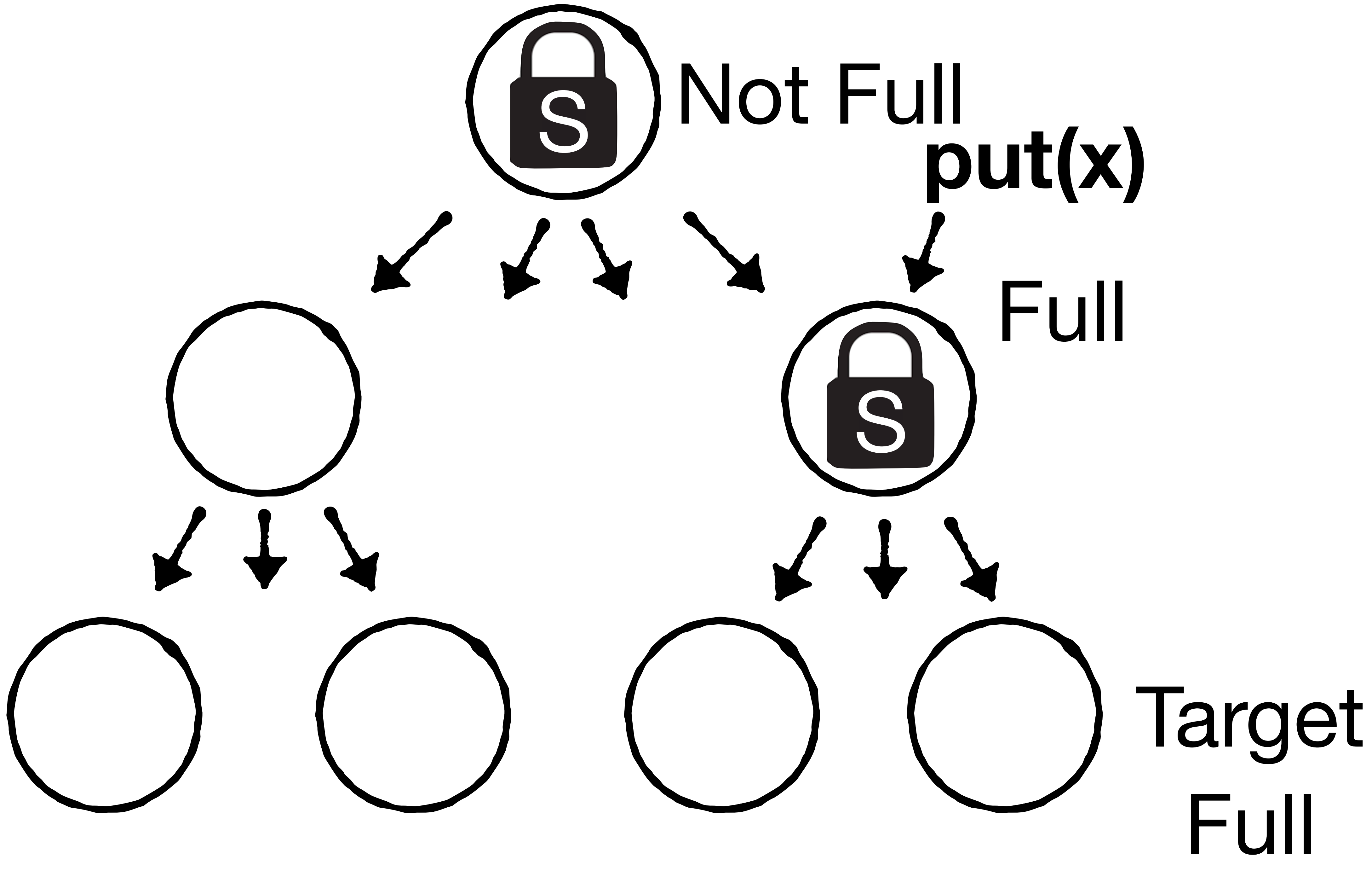
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



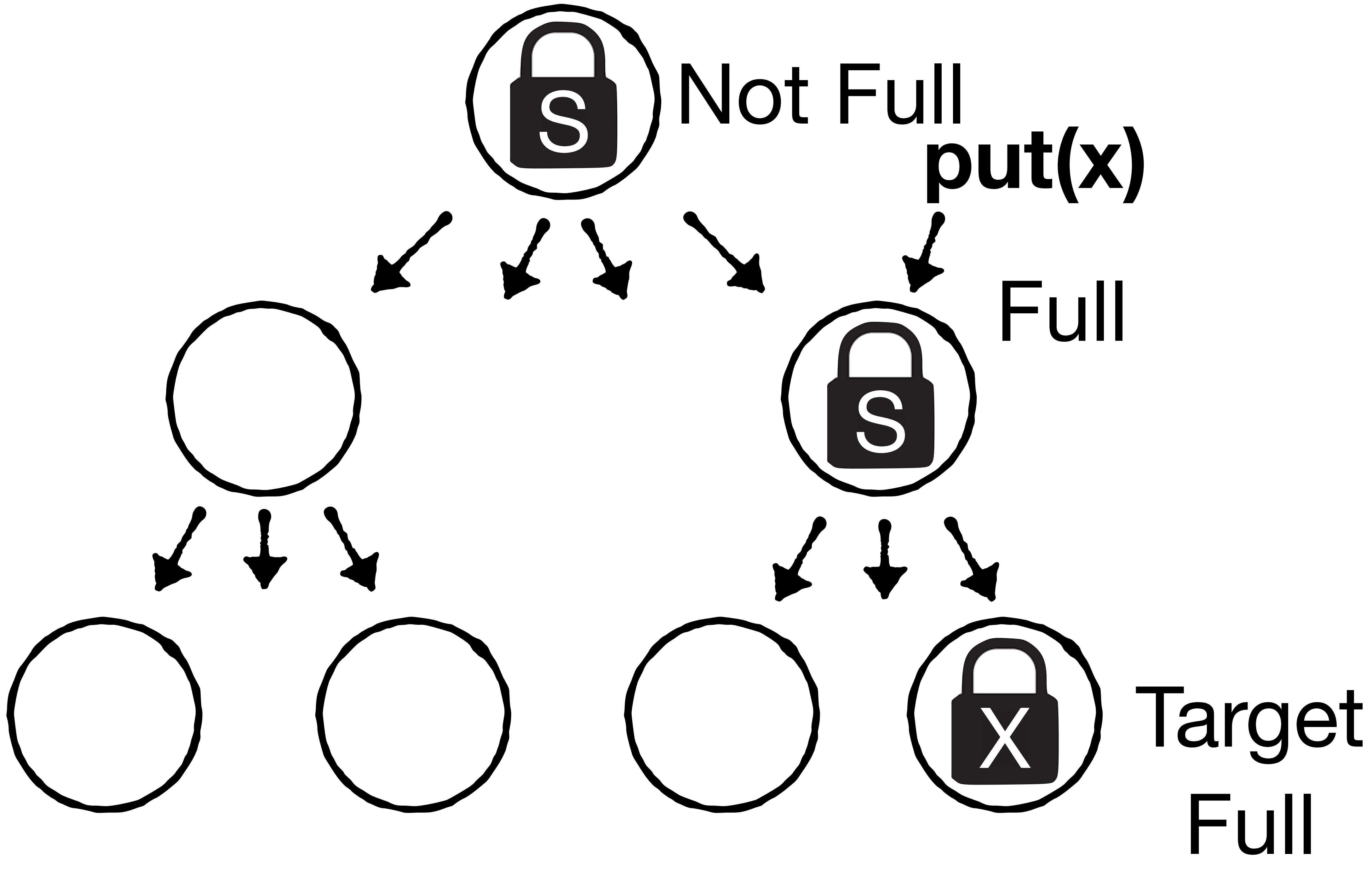
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



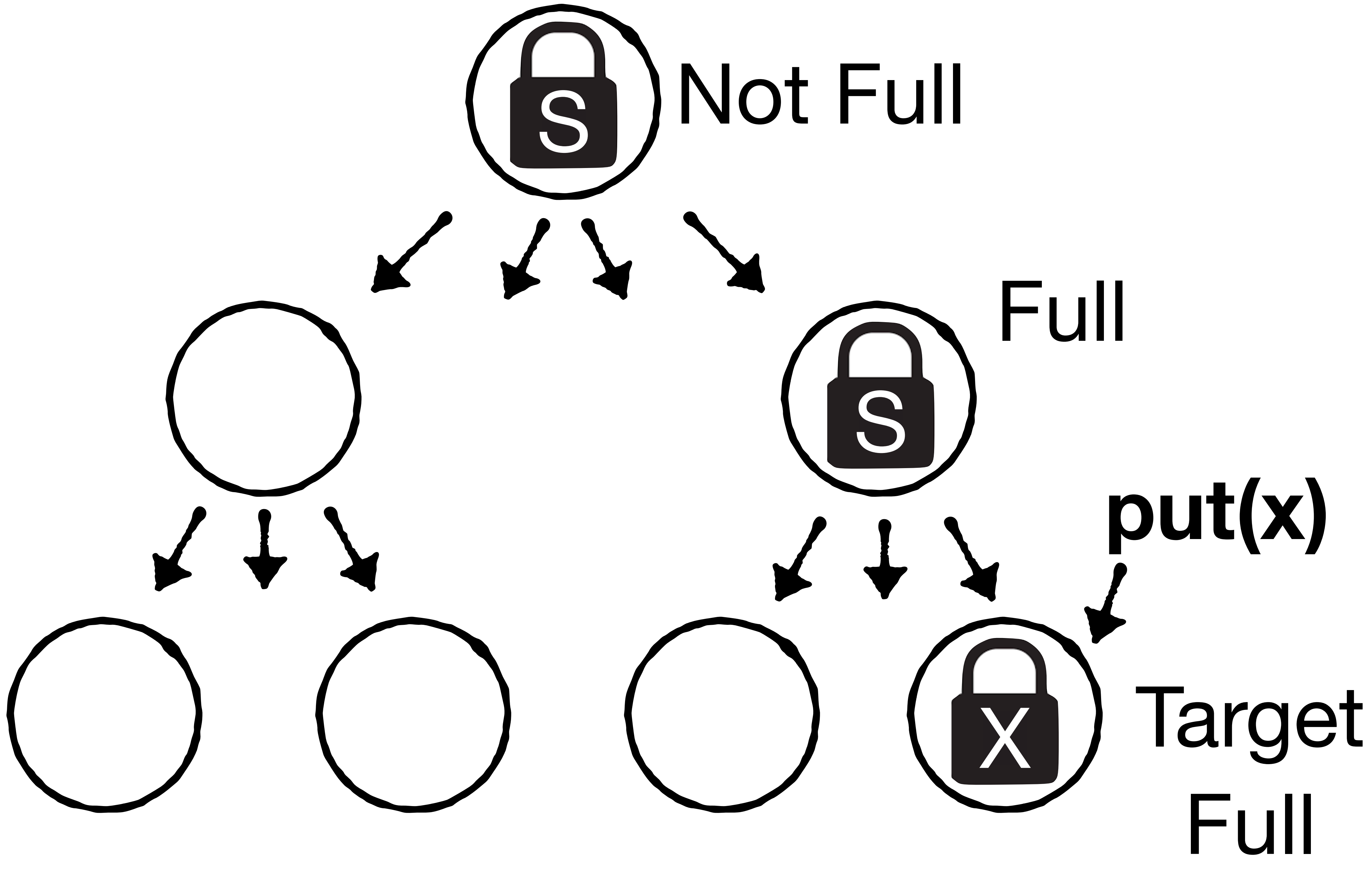
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



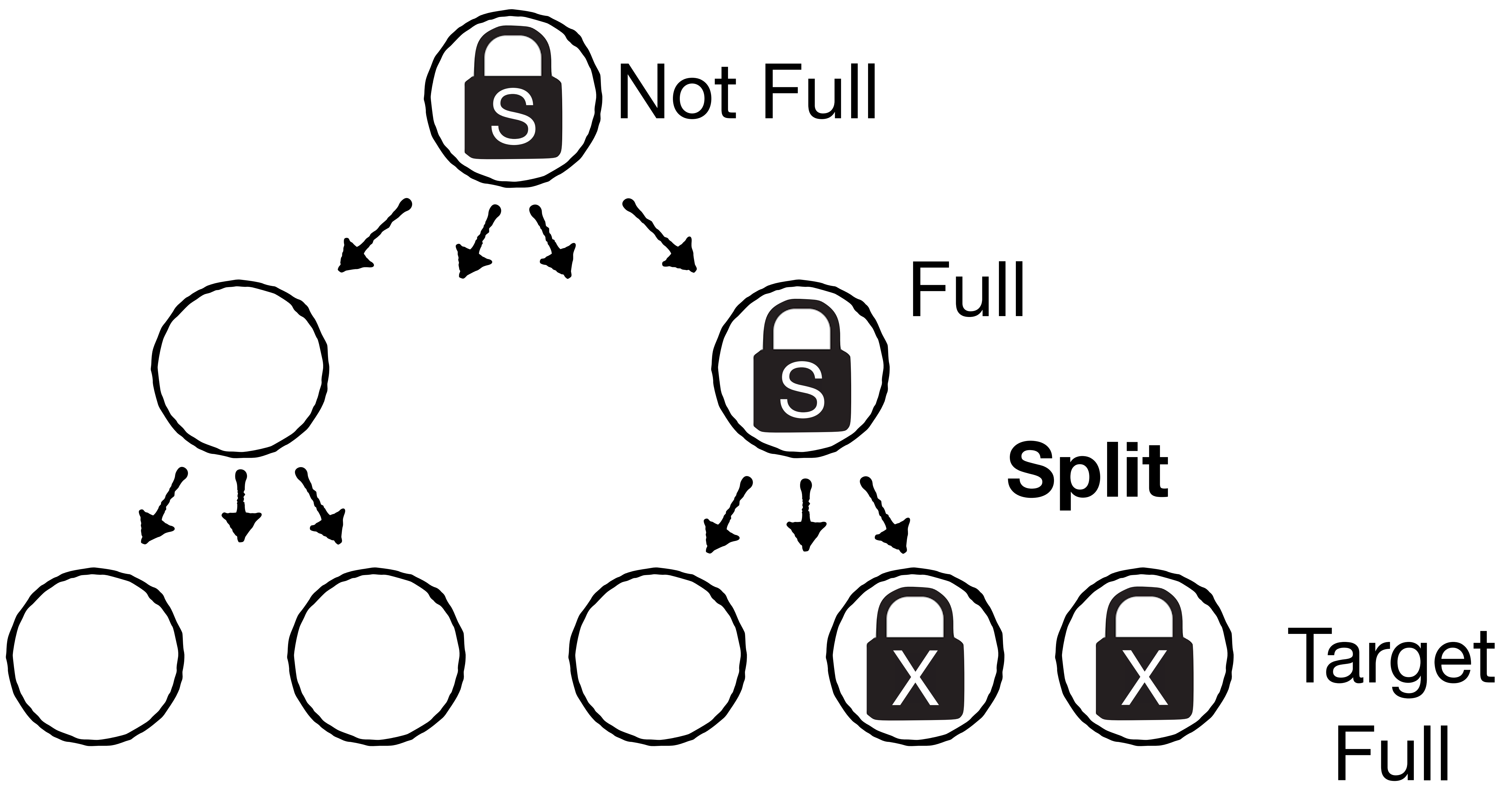
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



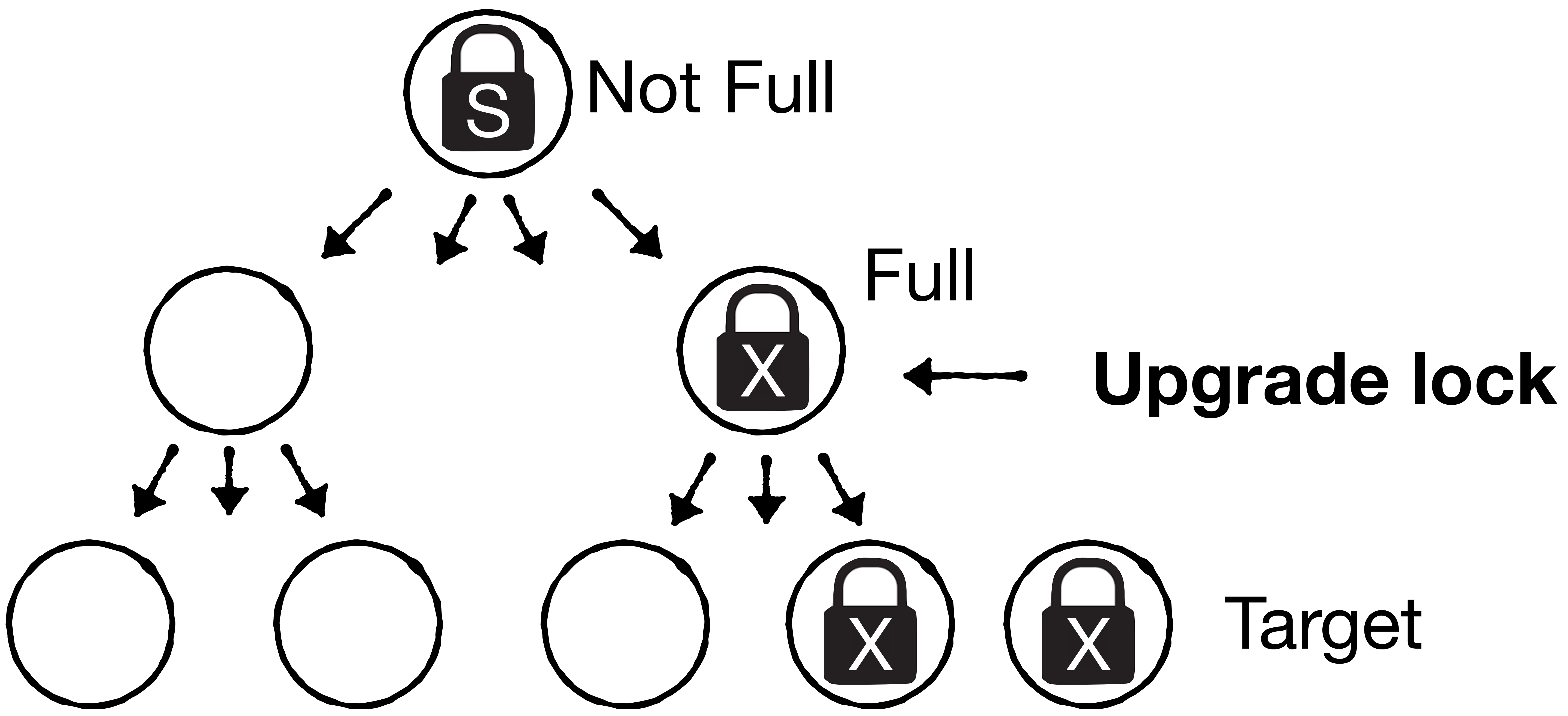
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



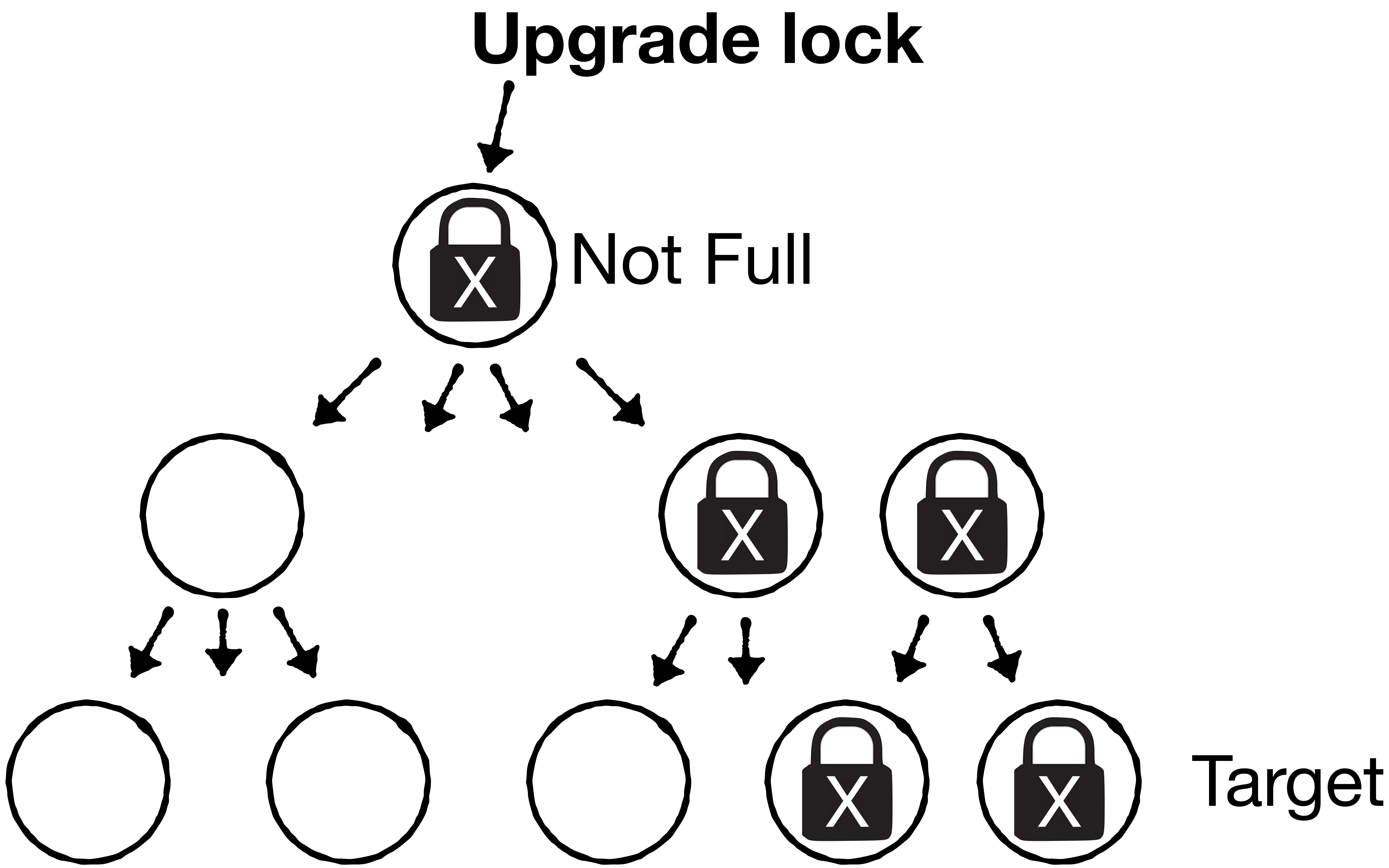
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



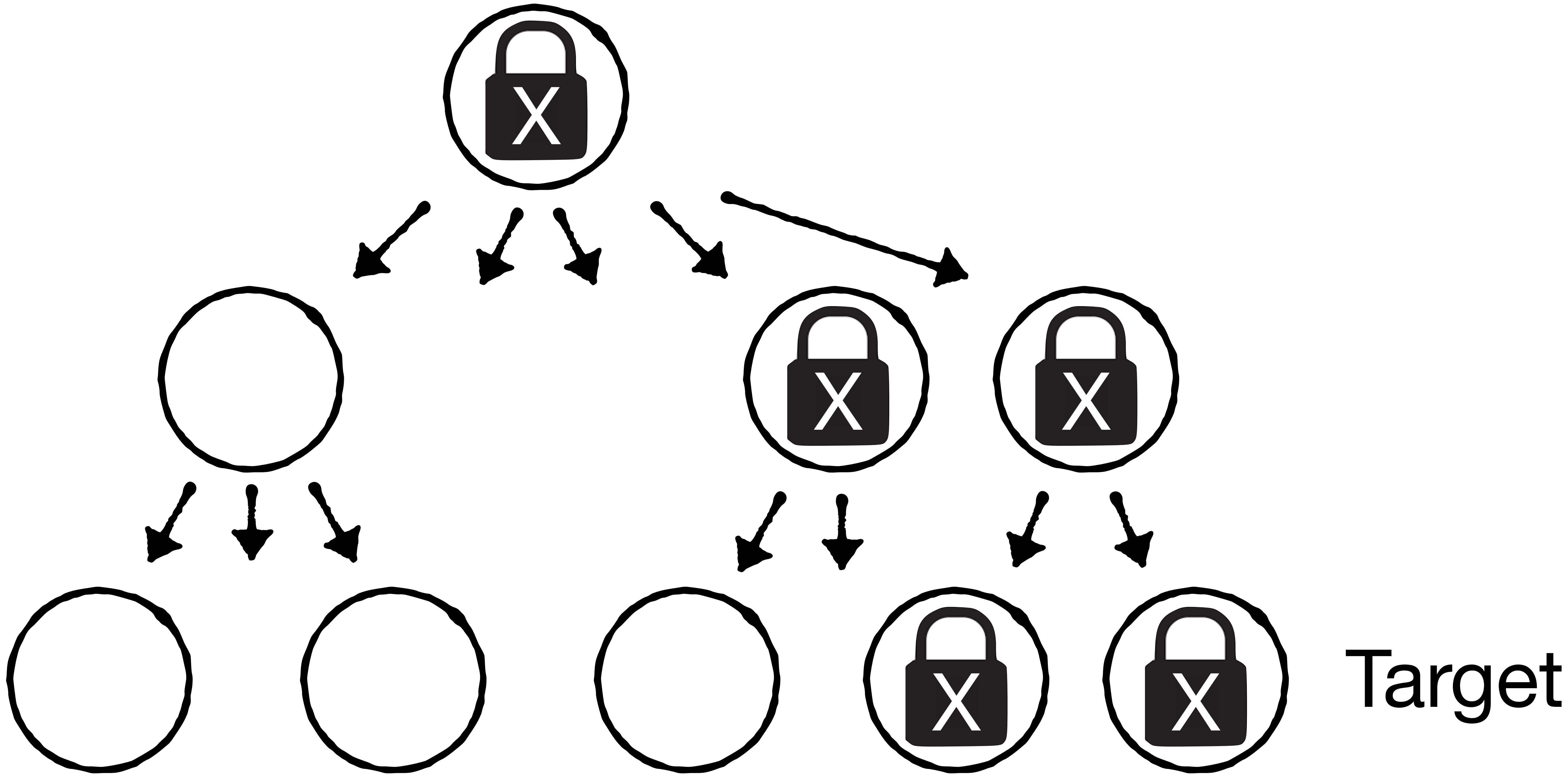
What must we lock at minimum to ensure correctness?

(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards

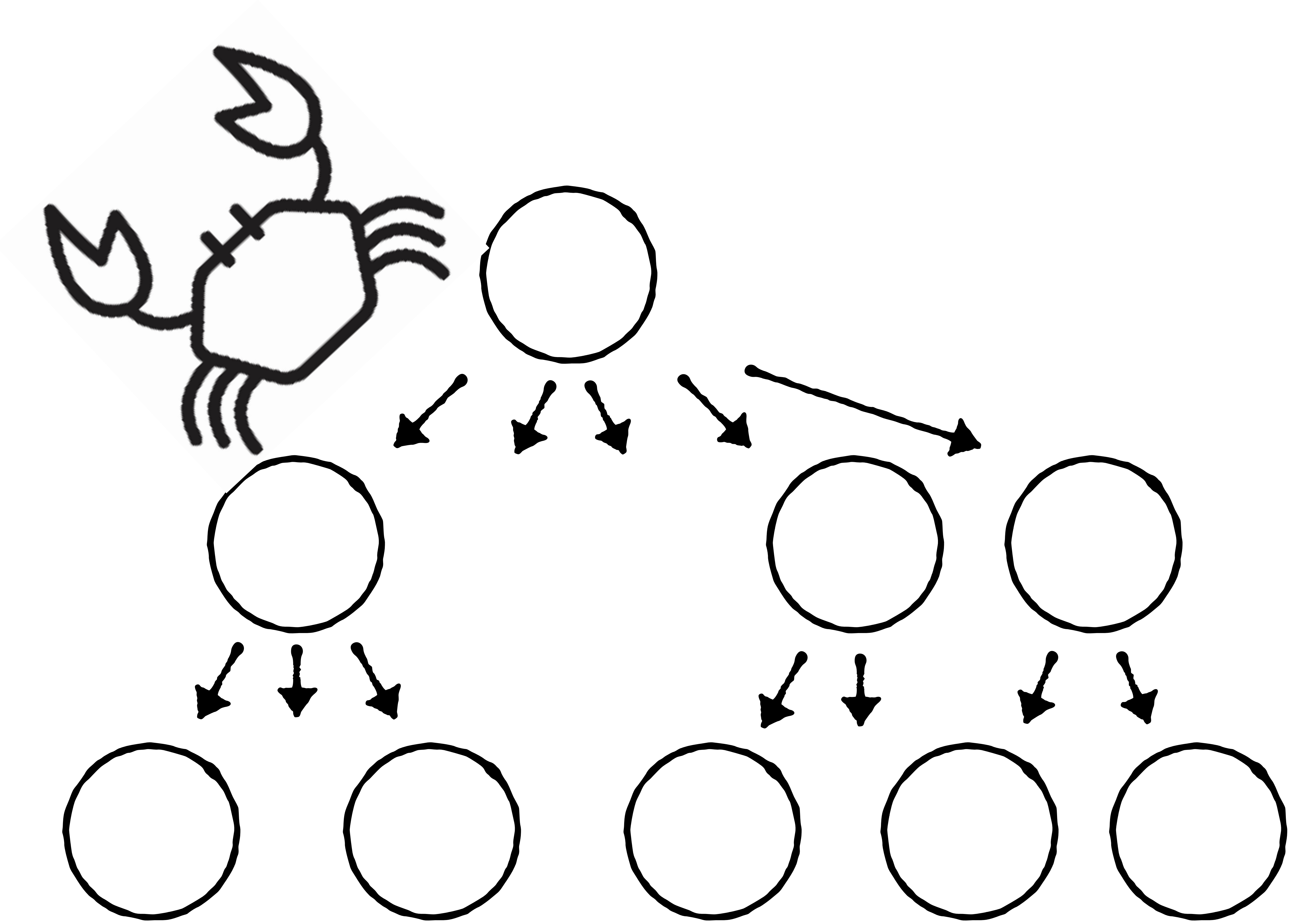


What must we lock at minimum to ensure correctness?

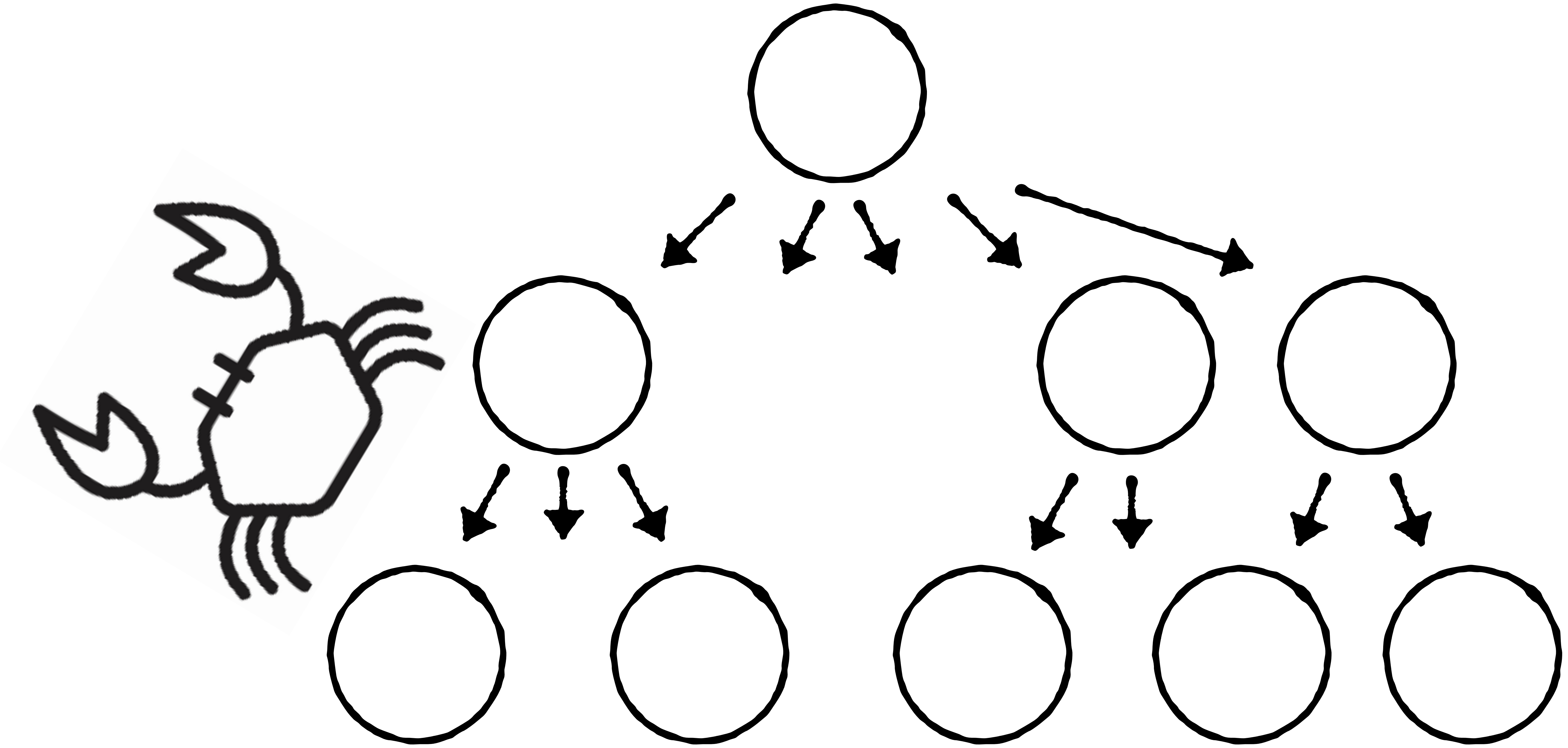
(2) Once we reach child, we only need to hold a lock on a parent if the child is full, as a split would propagate upwards



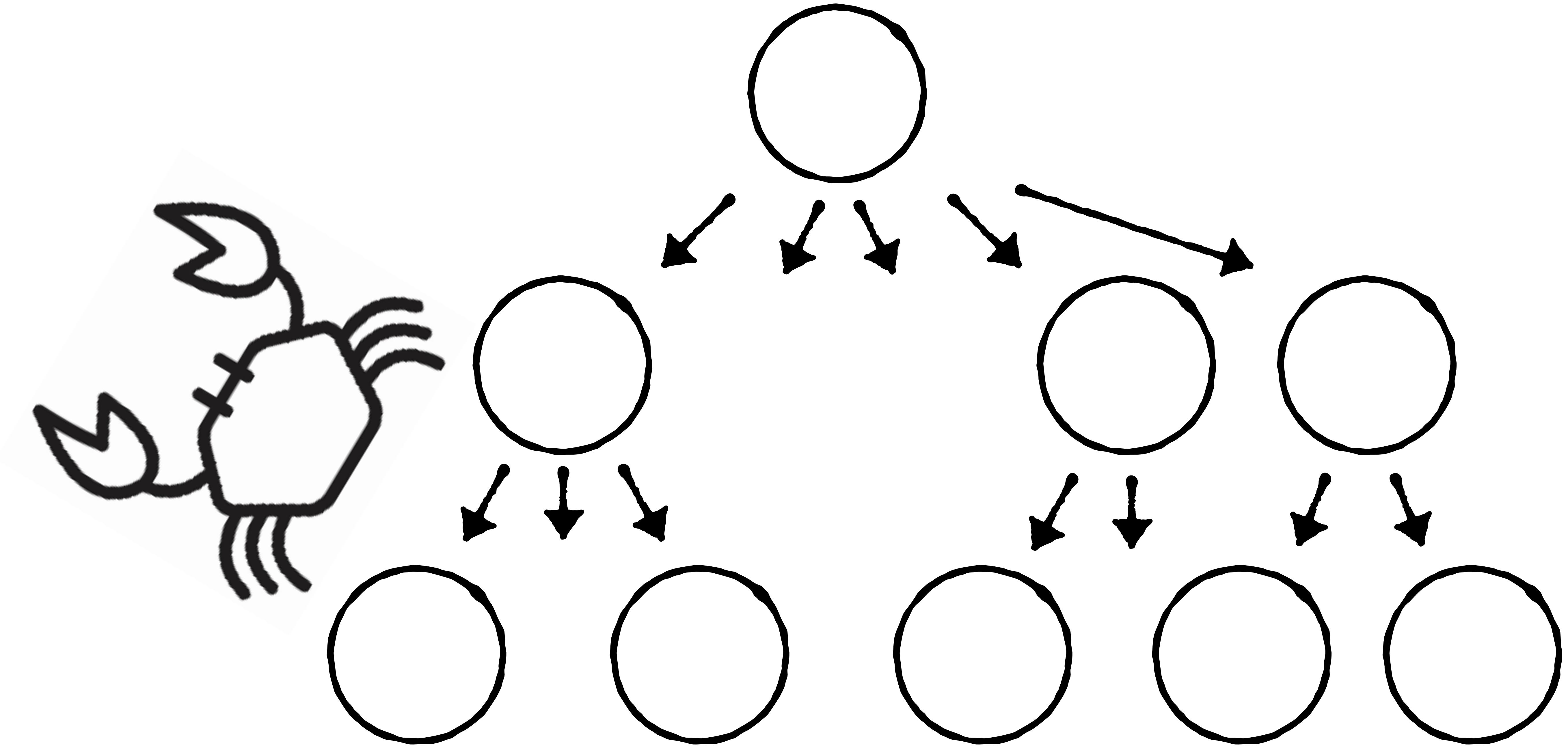
Protocol known as Lock-Coupling, or Crabbing



Protocol known as Lock-Coupling, or Crabbing



Shows that while strict two-phase locking is correct, it can sometimes be relaxed while maintaining correctness



And now, office hours