Transactions and Locking

Database Systems: The Complete Book
Ch 18.4, 19

Transaction Correctness

- Reliability in database transactions guaranteed by ACID
- A - Atomicity (“Do or Do Not, there is nothing like try”) - usually ensured by logs
- C - Consistency (“Within the framework of law”) - usually ensured by integrity constraints, validations, etc.
- I - Isolation (“Execute in parallel or serially, the result should be same”) - usually ensured by locks
- D - Durability (“once committed, remain committed”) - usually ensured at hardware level

What could go wrong?

Reading uncommitted data

(write-read/WR conflicts; aka “Dirty Reads”)

```
T1: R(A),W(A),              R(B),W(B),ABRT
T2:           R(A),W(A),CMT,
```

Unrepeatable Reads

(read-write/RW conflicts)

```
T1: R(A),              R(A),W(A),CMT
T2:      R(A),W(A),CMT,
```

What could go wrong?

Overwriting Uncommitted Data

(write-write/WW conflicts)

```
T1: W(A),              W(B),CMT
T2:       W(A),W(B),CMT,
```

Schedule

An ordering of read and write operations.

Serial Schedule

No interleaving between transactions **at all**

Serializable Schedule

Guaranteed to produce equivalent output
to a serial schedule

Conflict Equivalence

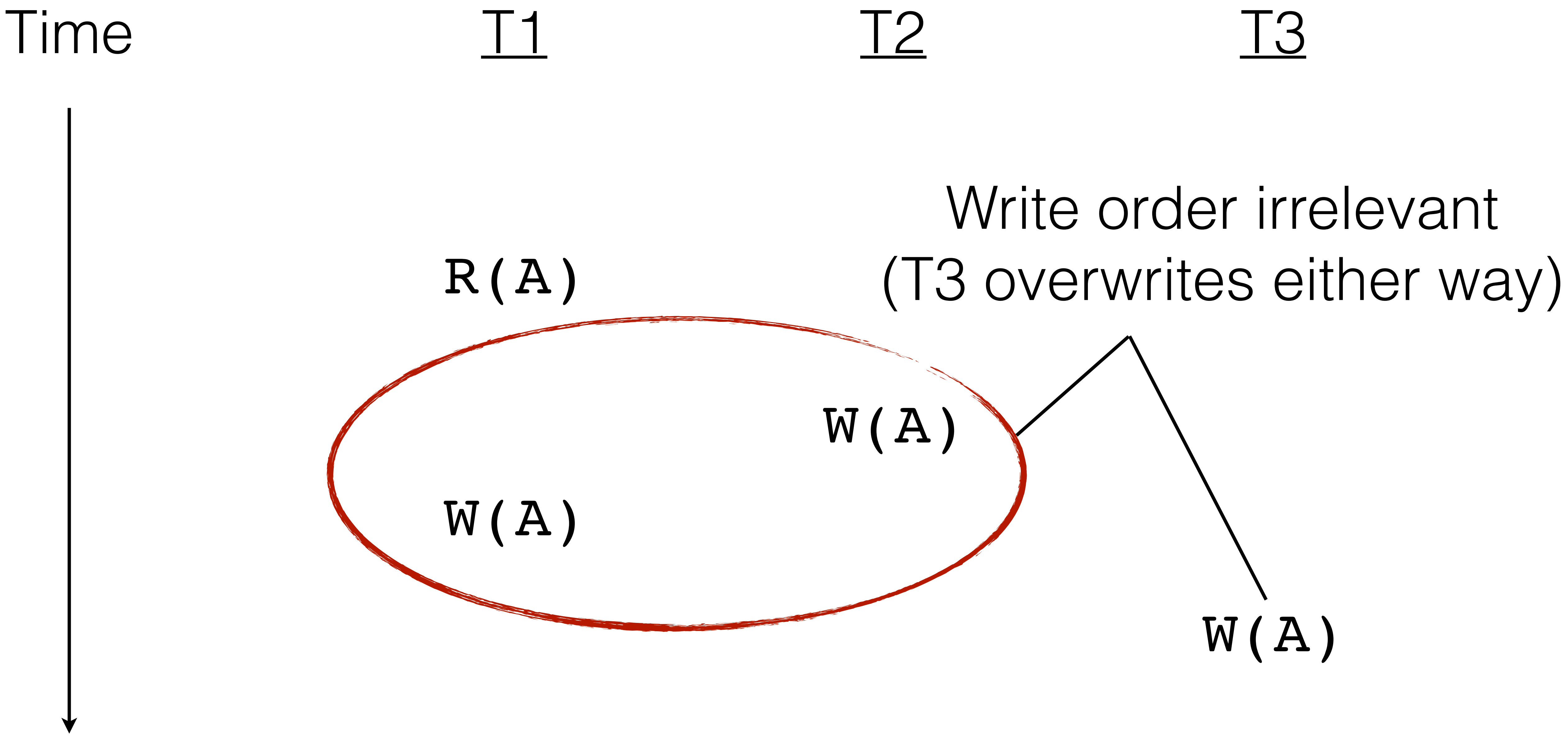
Possible Solution: Look at read/write, etc… conflicts!

Allow operations to be reordered as long as conflicts are ordered the same way

Conflict Equivalence: Can reorder one schedule into another without reordering conflicts.

Conflict Serializability: Conflict Equivalent to a serial schedule.

Example



Time

T1

T2

T3

`R(A)`

`W(A)`

`W(A)`

Write order irrelevant
(T3 overwrites either way)

`W(A)`

View Serializability

Possible Solution: Look at data flow!

View Equivalence: All reads read from the same writer
Final write in a batch comes from the same writer

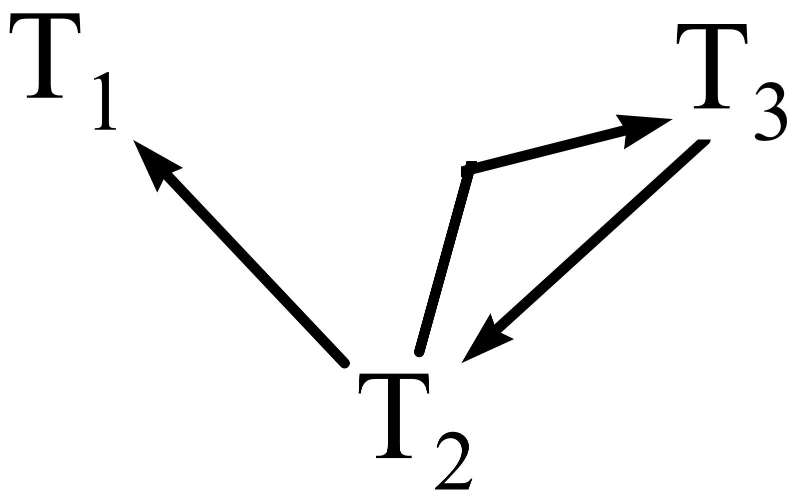
View Serializability: View Equivalent to a serial schedule.

Enforcing Serializability

- Conflict Serializability:
  - Does locking enforce conflict serializability?
- View Serializability
  - Is view serializability stronger, weaker, or incomparable to conflict serializability?
- What do we need to enforce either fully?

How to detect conflict serializable schedule?

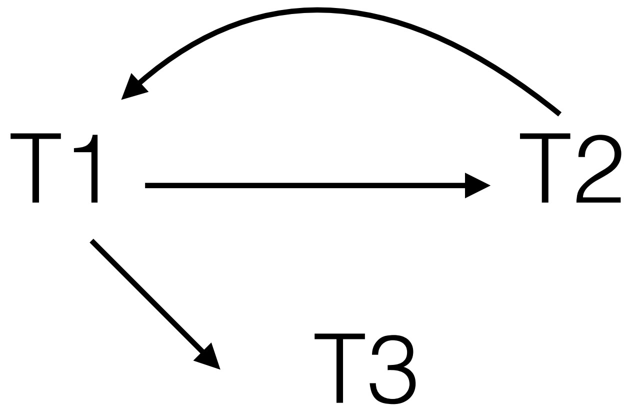
T1	T2	T3
W(a)		
	R(b)	
		W(d)
W(b)		
	R(d)	
		W(d)



Precedence Graph

Cycle!
Not Conflict serializable

Not conflict serializable but view serializable

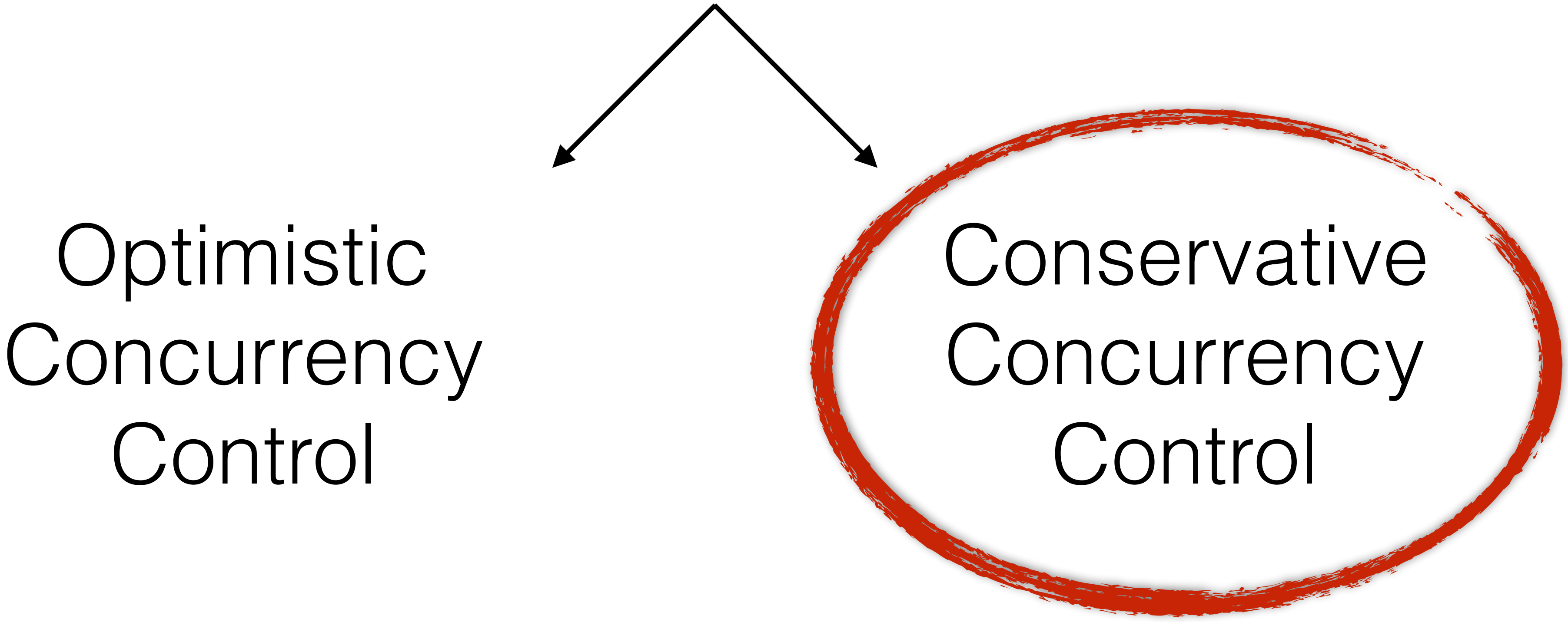


Satisfies 3 conditions of view serializability

T1	T2	T3
W(y)		
	W(y)	
	W(x)	
W(x)		
		W(x)

Every view serializable schedule which is not conflict serializable has blind writes.

How can conflicts be avoided?



Optimistic Concurrency Control

Conservative Concurrency Control

Conservative Concurrency Control

- How can bad schedules be detected?
- What problems does each approach introduce?
- How do we resolve these problems?

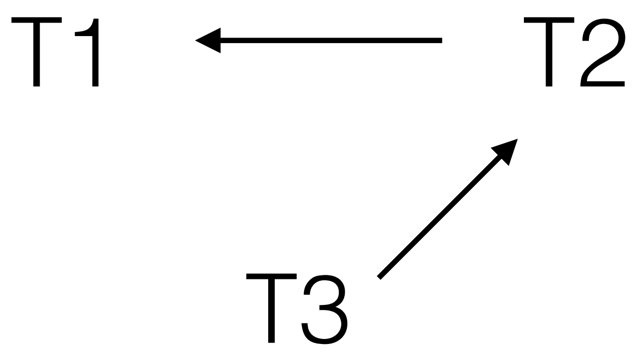
Two-Phase Locking

- Phase 1: Acquire (do not release) locks.
- Phase 2: Release (do not acquire) locks.

Why?

Can we do even better?

Example



Acyclic -
Conflict Serializable
2PL exists

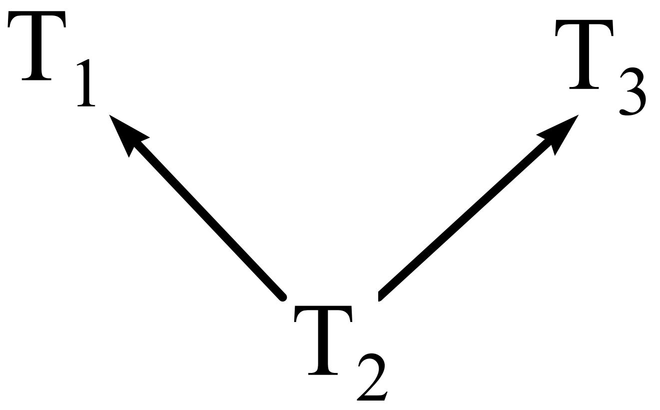
T1	T2	T3
		R(d)
W(a)		
	R(b)	
		W(d)
W(b)		
	R(d)	

Example

T1	T2	T3
		L(d) R(d)
L(a) W(a)		
	L(b) R(b)	
		W(d) R-L(d)
	L(d) R-L(b)	
L(b) R-L(a) W(b) R-L(b)		
	R(d) R-L(d)	

Need for shared and exclusive locks

T1	T2	T3
		L(d) R(d)
L(a) W(a)		
	L(b) R(b)	
L(b) W(b)		
	R(d)	
		W(d)

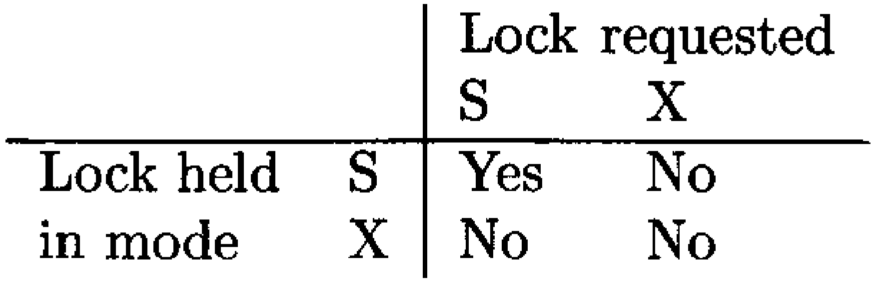


Precedence Graph

It is conflict Serializable but requires granular control of locks

Need for shared and exclusive locks

T1	T2	T3
		SL(d) R(d)
XL(a) W(a)		
	SL(b) SL(d) R(b) R-SL(b)	
XL(b) W(b) R-XL(b)		
	R(d) R-SL(d)	
		XL(d) W(d) R-XL(d)



		Lock requested	
		S	X
Lock held	S	Yes	No
in mode	X	No	No

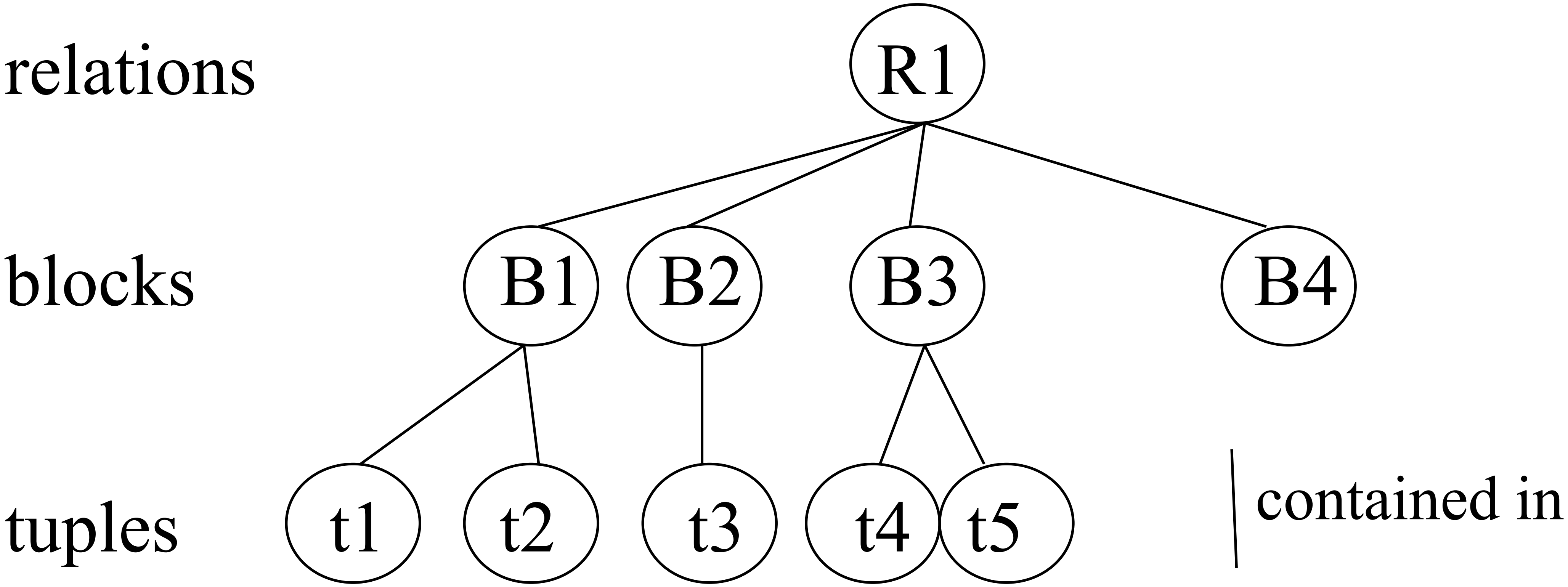
Reader/Writer (S/X)

- When accessing a DB Entity…
  - Table, Row, Column, Cell, etc…
- Before reading: Acquire a Shared (S) lock.
  - Any number of transactions can hold S.
- Before writing: Acquire an Exclusive (X) lock.
  - If a transaction holds an X, no other transaction can hold an S or X.

What do we lock?

Is it safe to allow some transactions to lock tables while other transactions to lock tuples?

New Lock Modes



Hierarchical Locks

- Lock Objects Top-Down
  - Before acquiring a lock on an object, an xact must have at least an intention lock on its parent!
- For example:
  - To acquire a S on an object, an xact must have an IS, IX on the object's parent (why not S, SIX, or X?)
  - To acquire an X (or SIX) on an object, an xact must have a SIX, or IX on the object's parent.

New Lock Modes

Lock Mode(s) Currently Held By Other Xacts

Lock Mode Desired

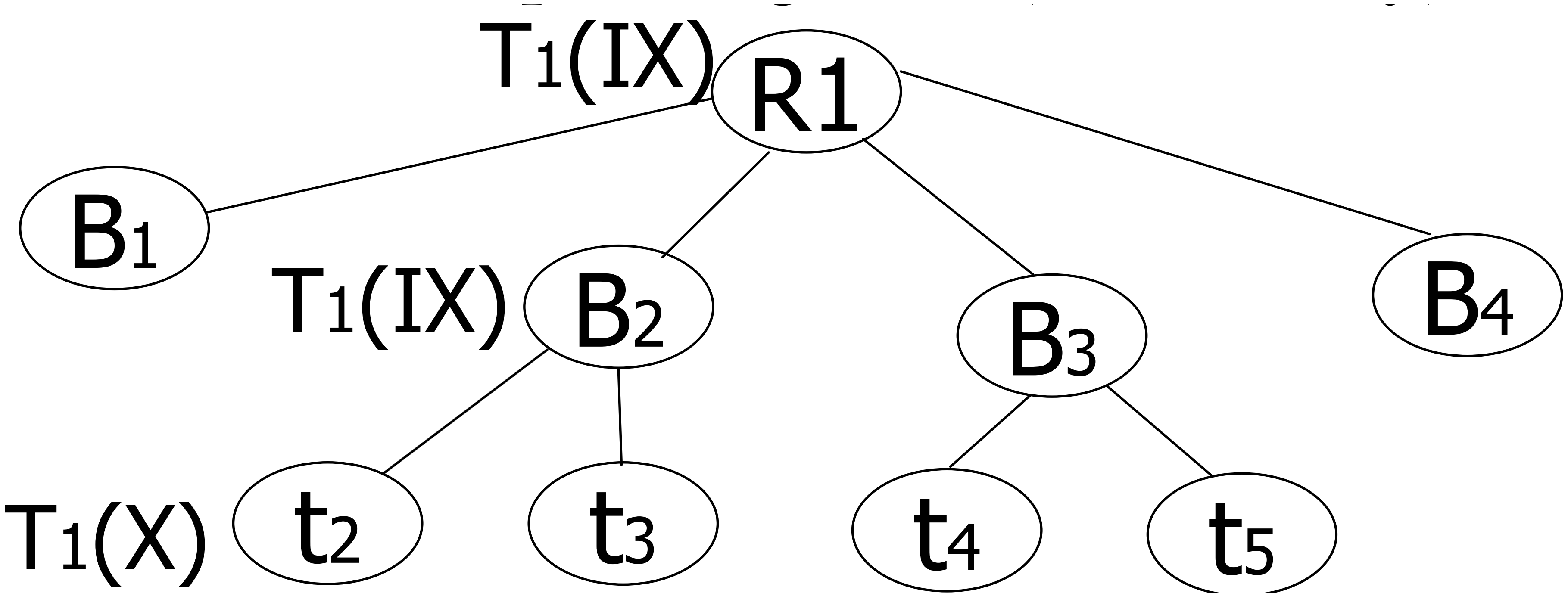
	None	IS	IX	S	X
None	valid	valid	valid	valid	valid
IS	valid	valid	valid	valid	fail
IX	valid	valid	valid	fail	fail
S	valid	valid	fail	valid	fail
X	valid	fail	fail	fail	fail

Example

- An I lock for a super-element constrains the locks that the same transaction can obtain at a subelement.
- If Ti has locked the parent element P in IS, then Ti can lock child element C in IS, S.
- If Ti has locked the parent element P in IX, then Ti can lock child element C in IS, S, IX, X.

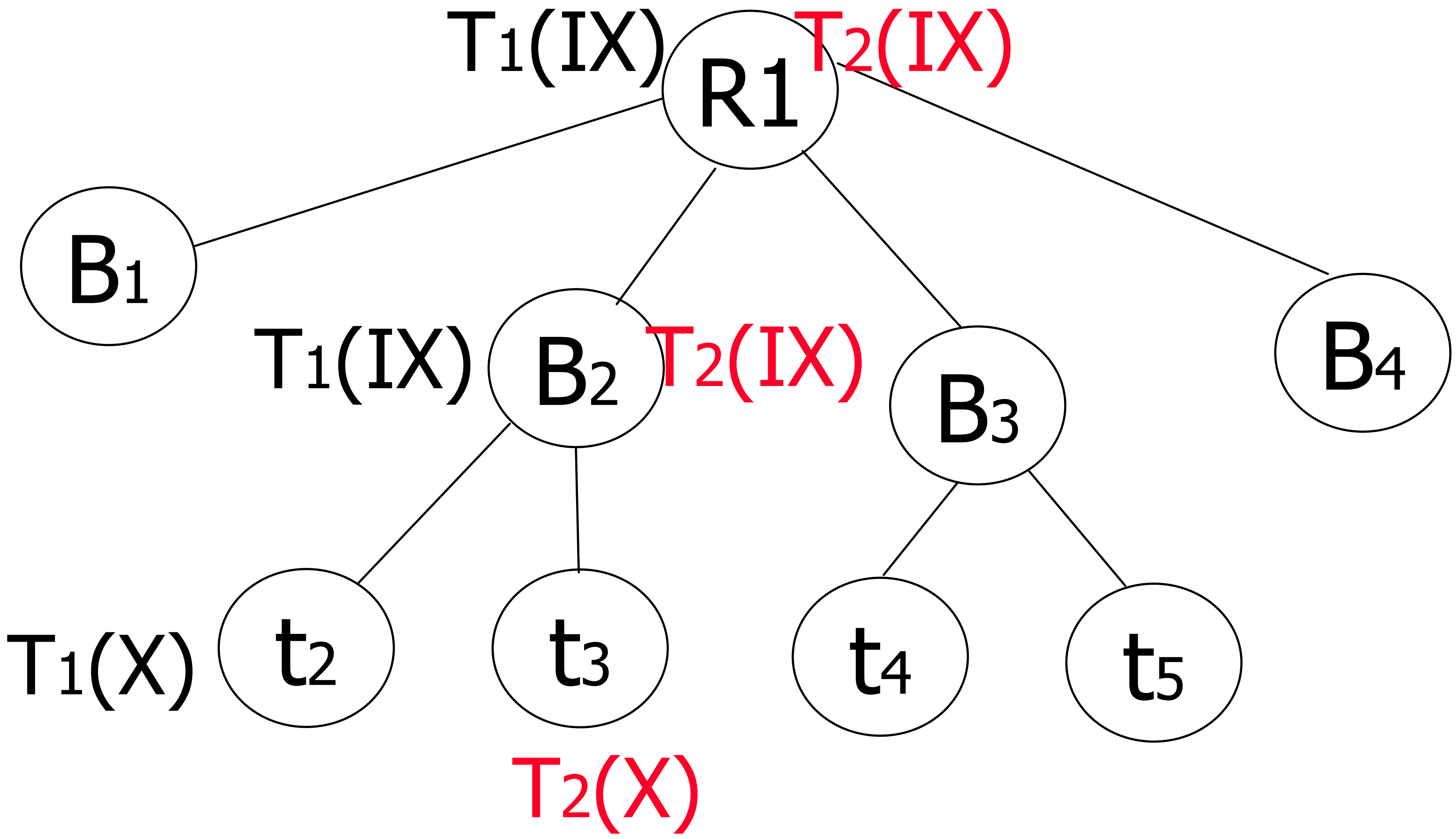
Example

- T1 wants exclusive lock on tuple t2



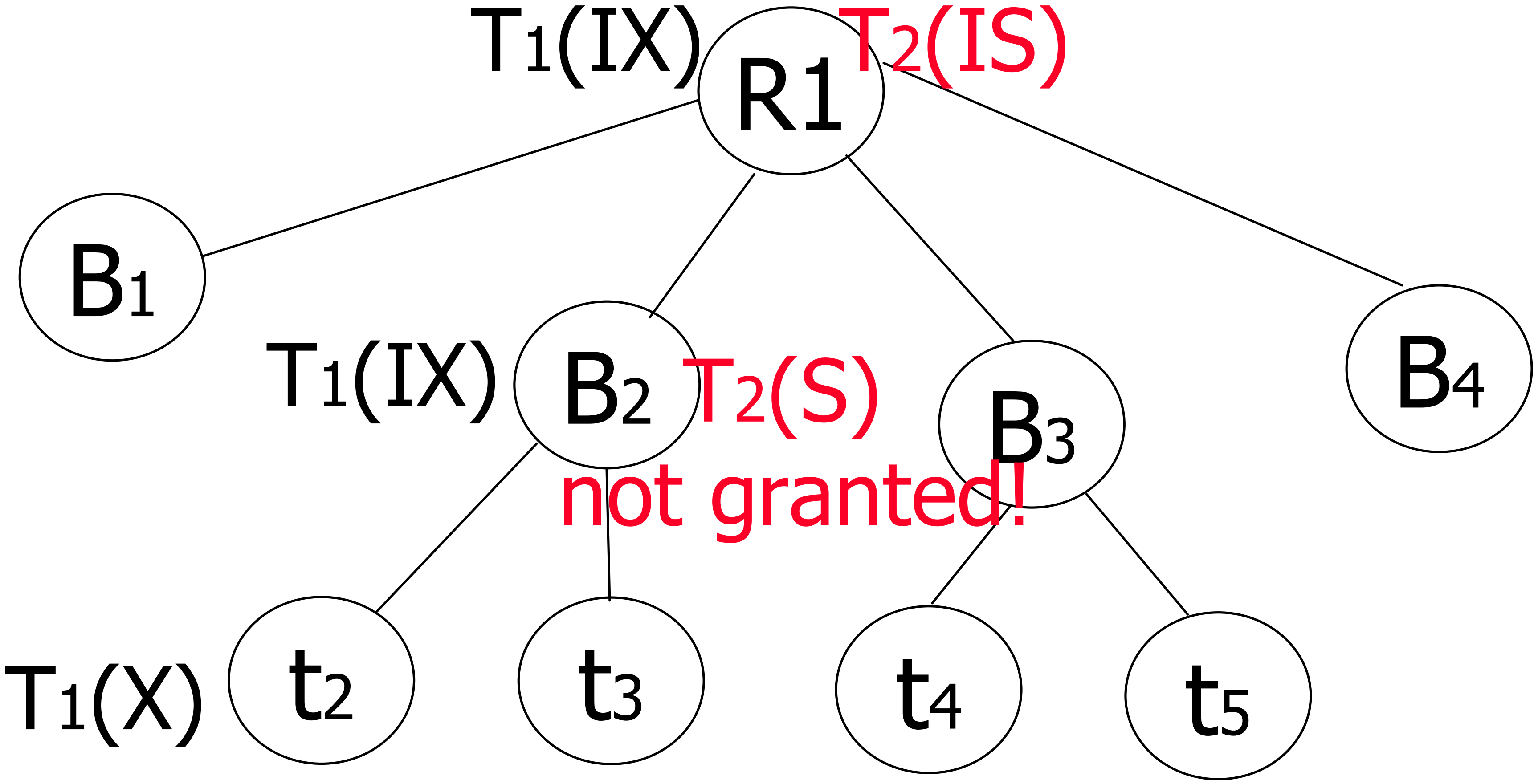
Example

- T2 wants to request an X lock on tuple t3



Example

T2 wants to request an S lock on block B2



Deadlocks

- Deadlock: A cycle of transactions waiting on each other's locks
  - Problem in 2PL; xact can't release a lock until it completes
- How do we handle deadlocks?
  - **Anticipate**: Prevent deadlocks before they happen.
  - **Detect**: Identify deadlock situations and abort one of the deadlocked xacts.

Deadlock Detection

- **Baseline**: If a lock request can not be satisfied, the transaction is blocked and must wait until the resource is available.
- Create a waits-for graph:
  - Nodes are transactions
  - Edge from T_i to T_k if T_i is waiting for T_k to release a lock.
- Periodically check for cycles in the graph.

Example

T_1: $l_1(A)$; $r_1(A)$; `A := A+100`; $w_1(A)$; $l_1(B)$; $u_1(A)$; $r_1(B)$; `B := B+100`; $w_1(B)$; $u_1(B)$;

T_2: $l_2(B)$; $r_2(B)$; `B := B*2`; $w_2(B)$; $l_2(A)$; $u_2(B)$; $r_2(A)$; `A := A*2`; $w_2(A)$; $u_2(A)$;

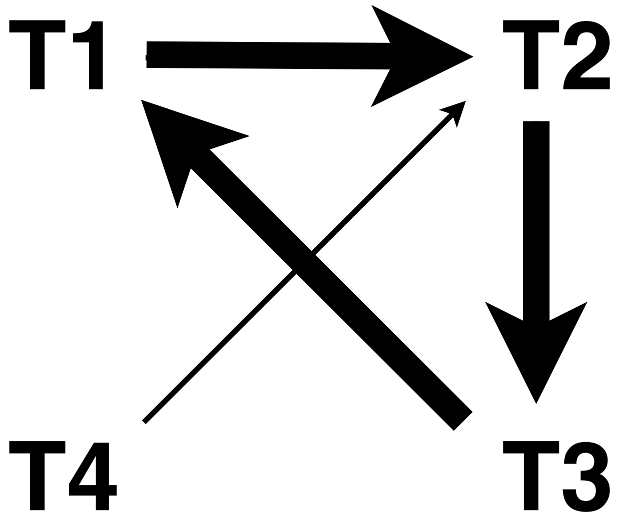
T_1	T_2	A	B
		25	25
$l_1(A)$; $r_1(A)$;			
	$l_2(B)$; $r_2(B)$;		
`A := A+100`;			
	`B := B*2`;		
$w_1(A)$;		125	
	$w_2(B)$;		50
$l_1(B)$ **Denied**	$l_2(A)$ **Denied**		

Example

Time	T1	T2	T3	T4
	`S(A)`			
	`R(A)`			
		`X(B)`		
		`W(B)`		
	`S(B)`			
			`S(C)`	
			`R(C)`	
		`X(C)`		
				`X(B)`
			`X(A)`	

Example

Time	T1	T2	T3	T4
	S(A)			
	R(A)			
		X(B)		
		W(B)		
	S(B)			
			S(C)	
			R(C)	
		X(C)		
				X(B)
			X(A)	



Handling Deadlocks

Approach 1

Avoid getting into deadlocks

Approach 2

Detect (and fix) deadlocks after they occur

Avoiding Deadlocks

Approach: Require transactions to follow an invariant that is guaranteed to be deadlock free.

Avoiding Deadlocks

Example: Give each Lock an ID #.
Only allow locks to be acquired in order of their ID.

Example

Time	T1	T2	T3	T4
	`S(A)`			
	`R(A)`			
		`X(B)`		
		`W(B)`		
	`S(B)`			
			`S(C)`	
			`R(C)`	
		`X(C)`		
				`X(B)`
			`X(A)`	

Out of Order
(T3 is not valid)

Avoiding Deadlock

Alternative: Acquire all locks at the start.

Example

Time	T1	T2	T3	T4
	`S(A)`			
	`R(A)`			
		`X(B)`		
		`W(B)`		
	`S(B)`			
			`S(C)`	
			`R(C)`	
		`X(C)`		
				`X(B)`
			`X(A)`	

Example

Time	T1	T2	T3	T4
↓	`S(A)`			
	`R(A)`			
		`X(B)`		
		`W(B)`		
	`S(B)`			
			`X(A)`	
	——	A released ——→	`S(C)`	
		`X(C)`		
				`X(B)`
		– C released →	`R(C)`	

Avoiding Deadlocks

Pro: No Deadlocks… Ever

Con: Not all transactions are supported.

or

Con: Transactions need to maintain all locks
that might possibly ever be required at all times.

Handling Deadlocks

Approach 1

Avoid getting into deadlocks

Approach 2

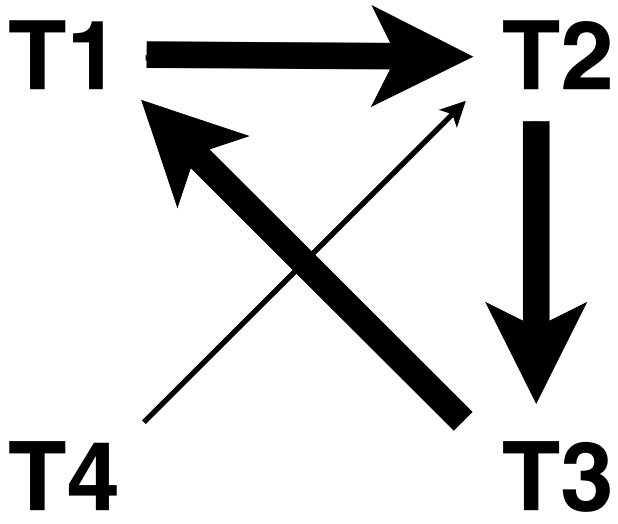
Detect (and fix) deadlocks after they occur

Deadlock Detection

- **Baseline**: If a lock request can not be satisfied, the transaction is blocked and must wait until the resource is available.
- Create a waits-for graph:
  - Nodes are transactions
  - Edge from T_i to T_k if T_i is waiting for T_k to release a lock.
- Periodically check for cycles in the graph.

Example

Time	T1	T2	T3	T4
	S(A)			
	R(A)			
		X(B)		
		W(B)		
	S(B)			
			S(C)	
			R(C)	
		X(C)		
				X(B)
			X(A)	



Deadlock Detection

What happens when a deadlock is detected?

(and get restarted)

Deadlock Detection

Default: Kill as many deadlocked transactions as needed.
(killed transactions may be restarted or “replayed”)

Optional: App-specific recovery logic

Detecting Deadlocks

Pro: No limitations on transactions

Pro: Best-case is faster than upfront acquisition

Con: Worst-case is much much slower.

Con: Cycle detection is slow and expensive

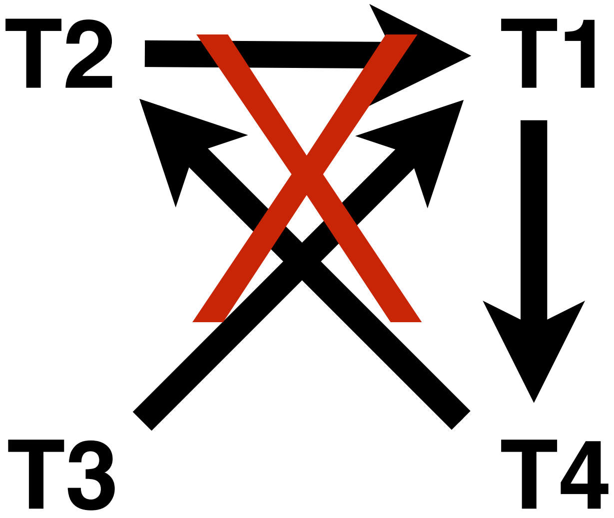
Simpler Detection Schemes

Approach: Accept false positives for faster deadlock detection

Simpler Detection Schemes

- **Trivial Solution**: Time-outs.
- **Invariant-Based Solution**: Enforce monotonicity property about which transactions are allowed to block which transactions.

Simpler Detection Scheme 1



Intuition: Never block on an ‘older’ transaction

Simpler Detection Scheme 1

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)

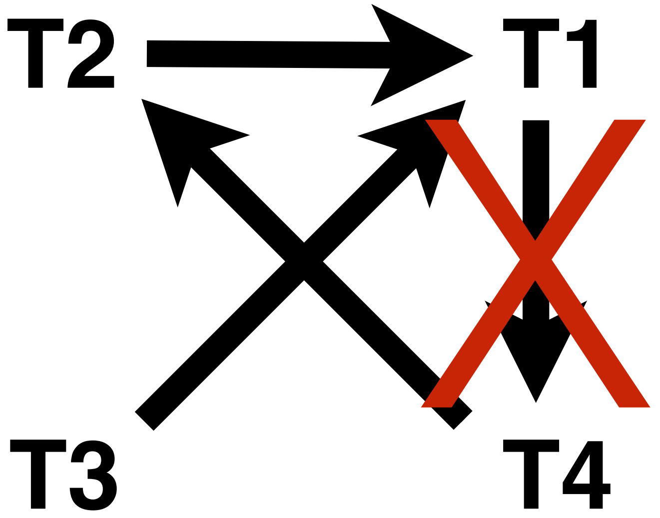
the invariant is preserved

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)

avoid deadlock by killing T2

“Wait-Die”

Simpler Detection Scheme 2



Intuition: Never block on a ‘younger’ transaction

Simpler Detection Scheme 1

T1 holds a lock on A
T2 tries to acquire the lock on A (and would block)

the invariant is preserved

T2 holds a lock on A
T1 tries to acquire the lock on A (and would block)

avoid deadlock by killing T2 and giving T1 the lock

“Wound-Wait”

Which transaction?

Policy 1: Wait-Die
“Those in power stay in power”
Blocking Xact Dies

Policy 2: Wound-Wait
“Take everything you can”
Blocking Xact Kills Other

Simpler Detection Schemes

Preserve fairness: A killed transaction is restarted with the same timestamp

Managing Deadlocks

- Approach 1: Avoidance
  - Invariant on lock acquisition order.
  - Aquire all locks upfront.
- Approach 2: Recovery
  - Detect cycles (or conditions that indicate cycles)
  - Kill/Restart transactions until there are no cycles.