Transactions & Update Correctness

April 11, 2017

Correctness

- Data Correctness (Constraints)
- Query Correctness (Plan Rewrites)
- **Update Correctness (Transactions)**

What could go wrong?

- **Parallelism**: What happens if two updates modify the same data?
  - Maximize use of IO / Minimize Latencies.
- **Persistence**: What happens if something breaks during an update?
  - When is my data safe?

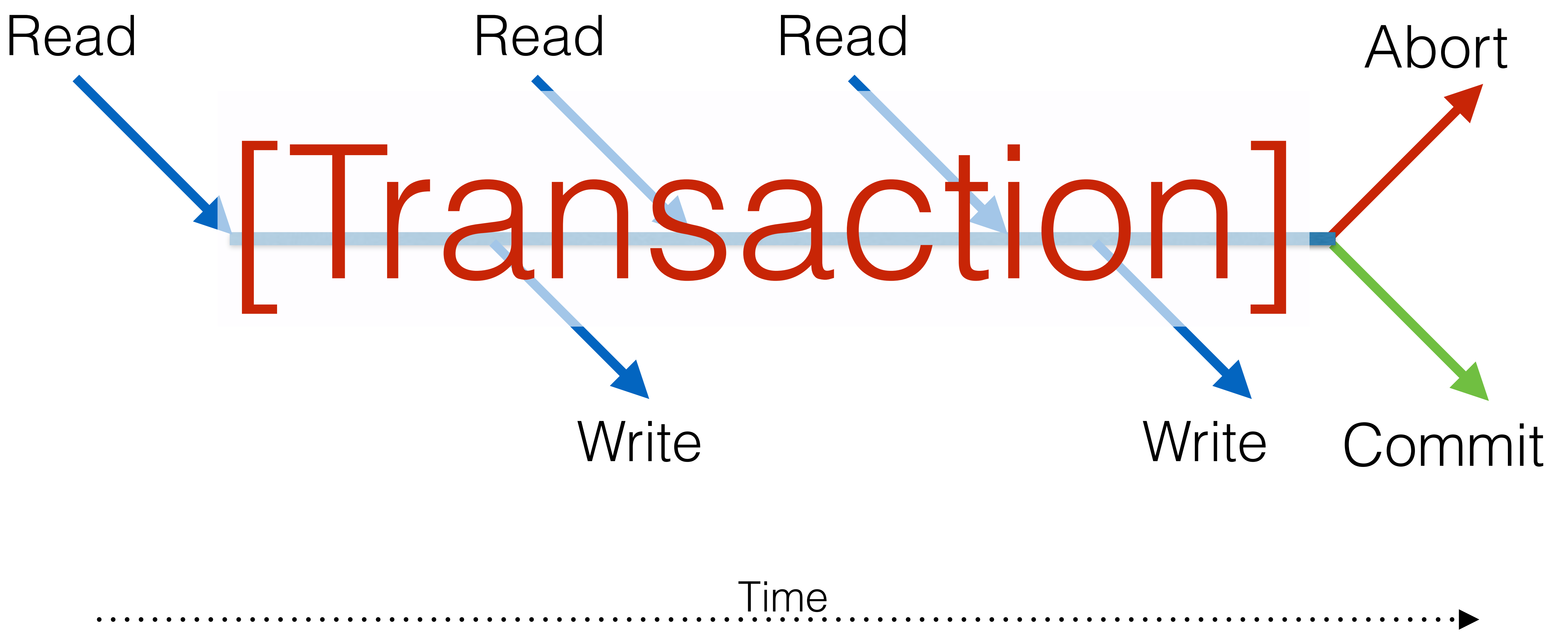
What does it mean for a database operation to be correct?

What is an Update?

- INSERT INTO …?
- UPDATE … SET … WHERE …?
- Non-SQL?

Can we abstract?

Abstract Update Operatons



Transaction

What does it mean for a ~~database operation~~ to be correct?

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

Atomicity

- A transcation completes by committing, or terminates by aborting.
    - Logging is used to undo aborted transactions.
- **Atomicity**: A transaction is (or appears as if it were) applied in one ‘step’, independent of other transactions.
    - All ops in a transaction commit or abort together.

Isolation

```
T1: BEGIN A=A+100,  B=B-100  END
T2: BEGIN A=1.06*A, B=1.06*B END
```

- Intuitively, T1 transfers $100 from A to B and T2 credits both accounts with interest.
- What are possible interleaving errors?

Example: Schedule

Time	T1	T2
	`A=A+100`	
		`A=1.06*A`
	`B=B-100`	
		`B=1.06*B`

OK!

Example: Schedule

Time	T1	T2
	`A=A+100`	
		`A=1.06*A`
		`B=1.06*B`
	`B=B-100`	

Not OK!

Example:The DBMS's View

Time	T1	T2
	`R(A)`	
	`W(A)`	
		`R(A)`
		`W(A)`
		`R(B)`
		`W(B)`
	`R(B)`	
	`W(B)`	

Not OK!

What went wrong?

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

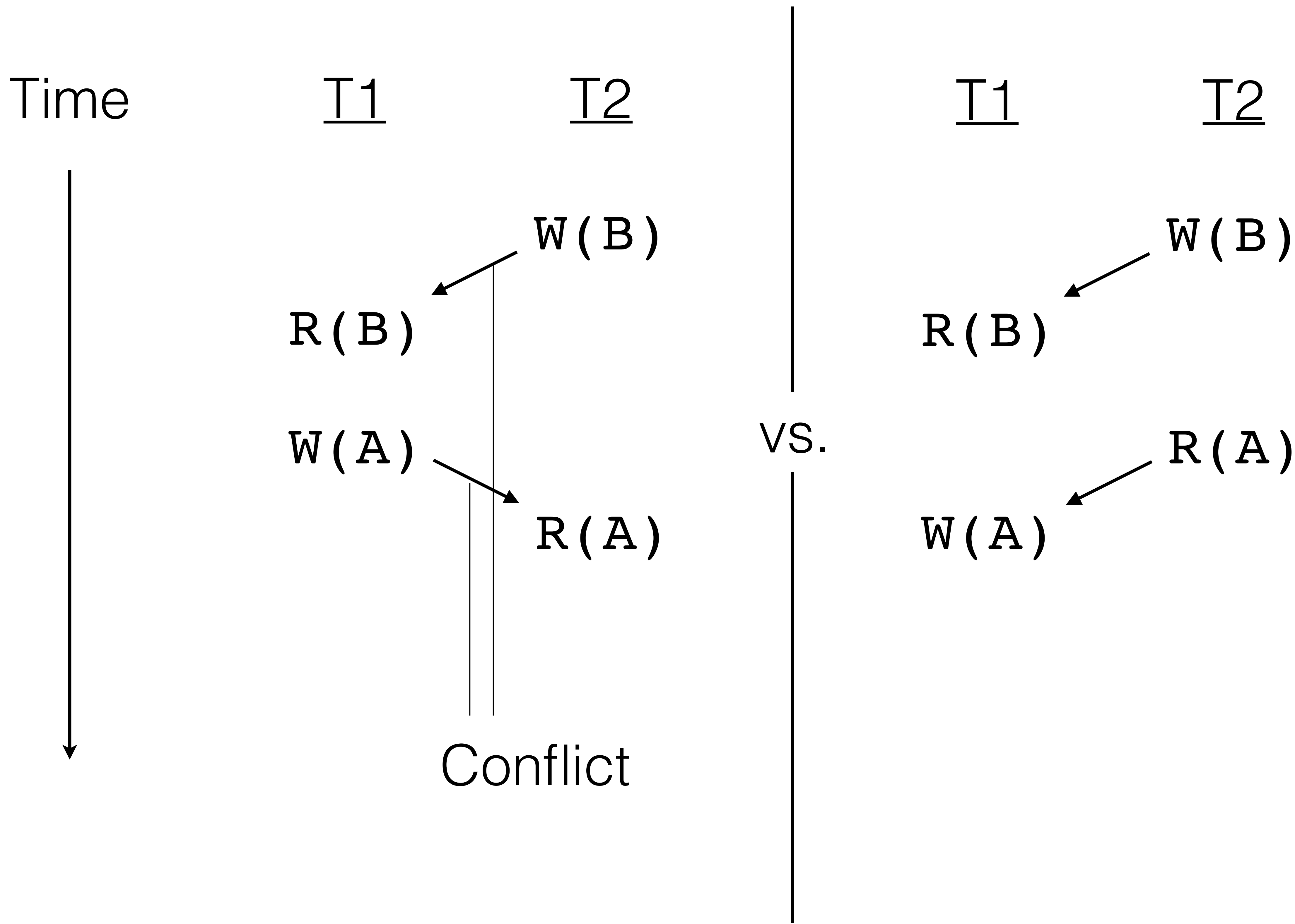
<u>Conflict Equivalence</u>: Can reorder one schedule into another without reordering conflicts.

<u>Conflict Serializability</u>: Conflict Equivalent to a serial schedule.

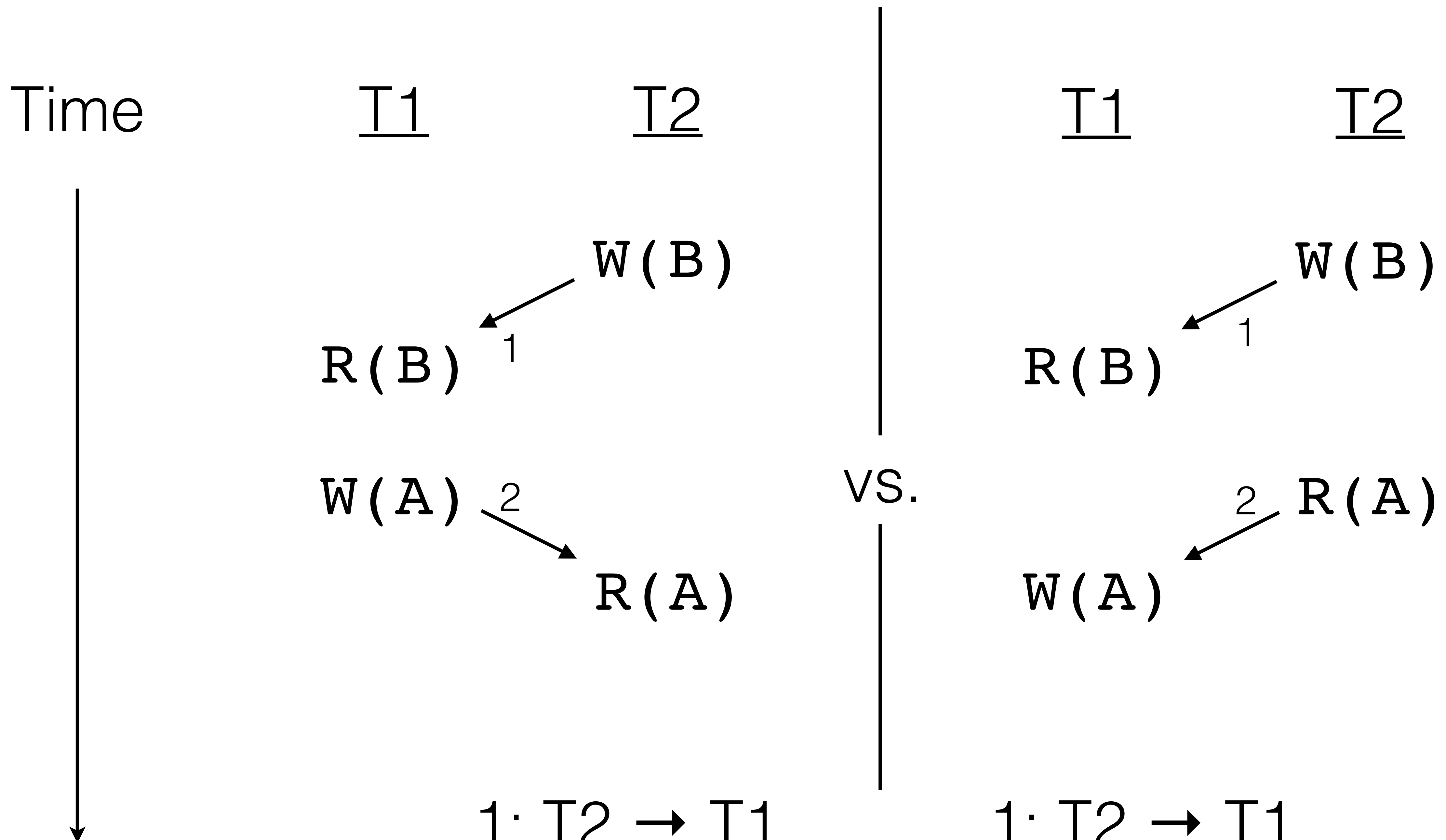
Conflict Serializability

- **Step 1:** Serial Schedules are Always Correct
- **Step 2:** Schedules with the same operations and the same conflict ordering are conflict-equivalent.
- **Step 3:** Schedules conflict-equivalent to an always correct schedule are also correct.
    - … or conflict serializable

Example



Example



1: T2 → T1
2: T1 → T2

≠

1: T2 → T1
2: T2 → T1

Equivalence

- Look at the actual effects
  - Can’t determine effects without running
- Look at the conflicts
  - Too strict
- Look at the possible <u>effects</u>

Example

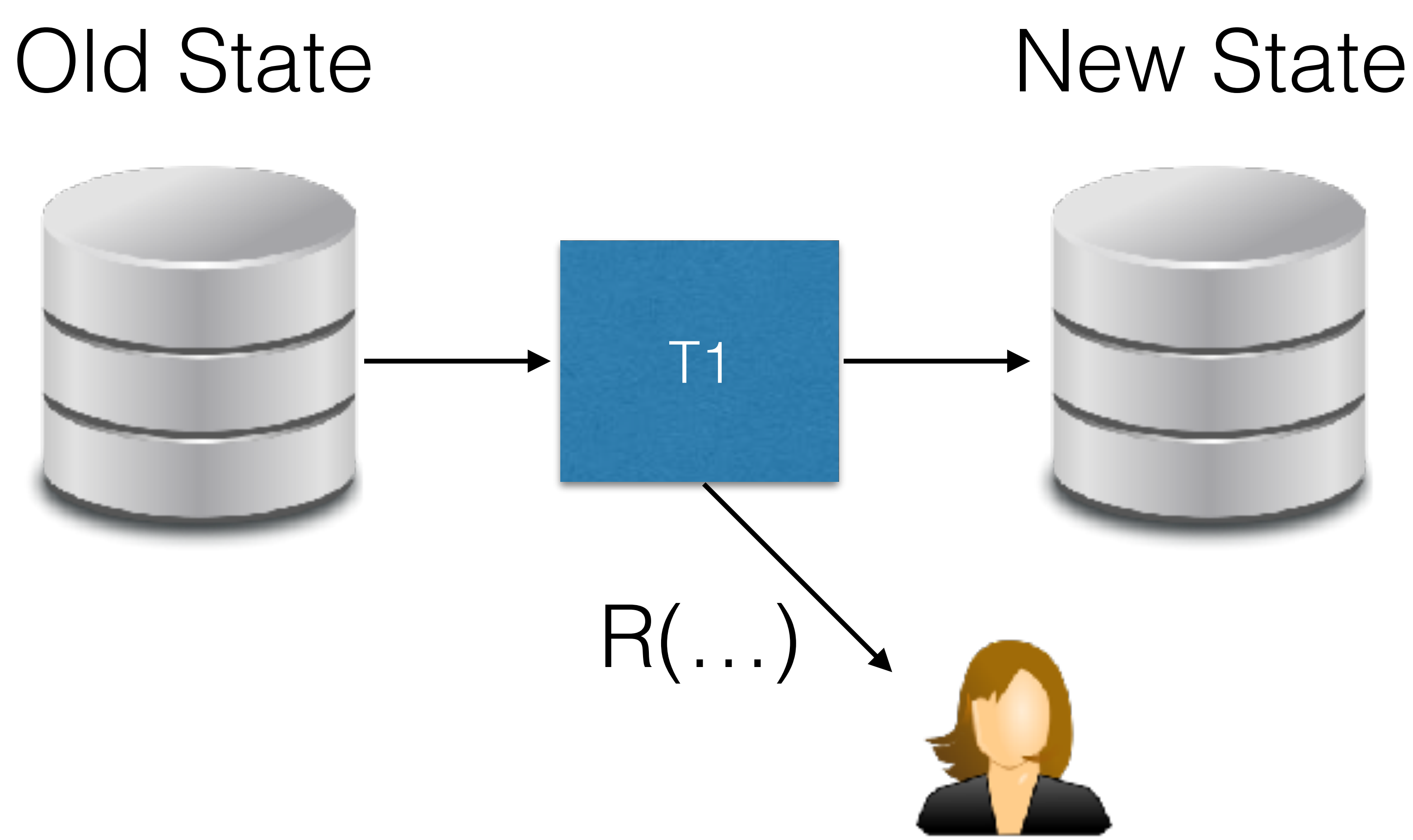
Time	T1	T2	T3
	`R(A)`		
	`W(A)`		
		`W(A)`	
			`W(A)`

Example

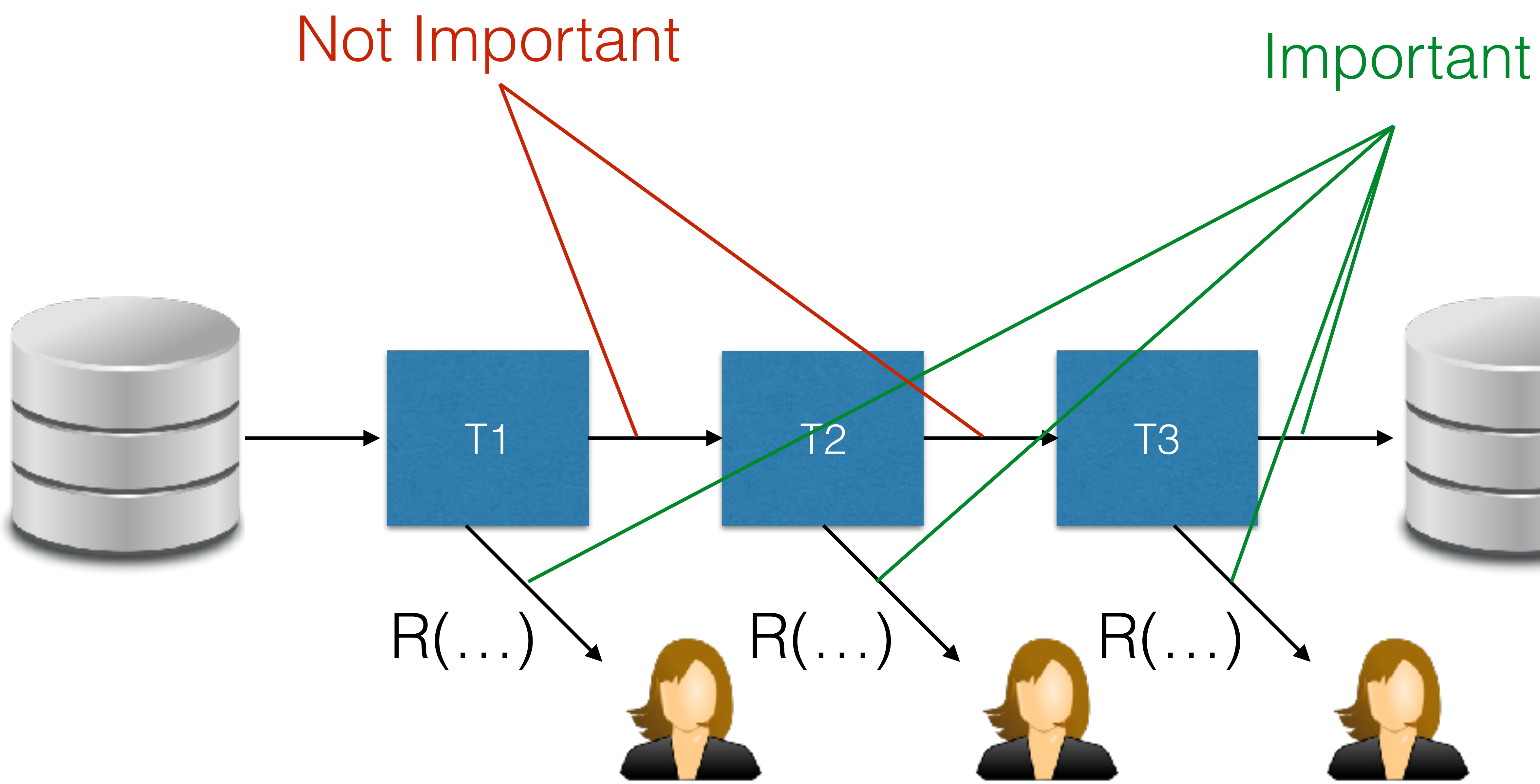
Write order irrelevant
(T3 overwrites either way)

Time	T1	T2	T3
	`R(A)`		
		`W(A)`	
	`W(A)`		
			`W(A)`

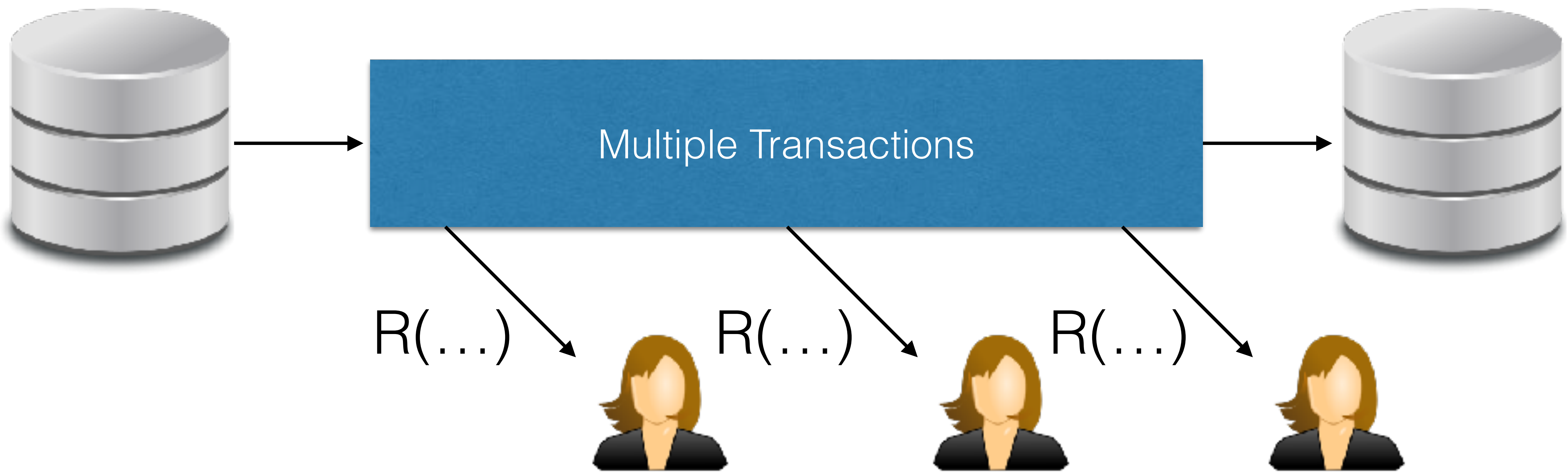
Information Flow



Information Flow



Information Flow



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.

View Equivalence

- For all Reads R
  - If R reads old state in S1, R reads old state in S2
  - If R reads Ti’s write in S1, R reads the the same write in S2
- For all values V being written.
  - If W is the last write to V in S1, W is the last write to V in S2
- If these conditions are satisfied, S1 and S2 are <u>view-equivalent</u>

View Serializability

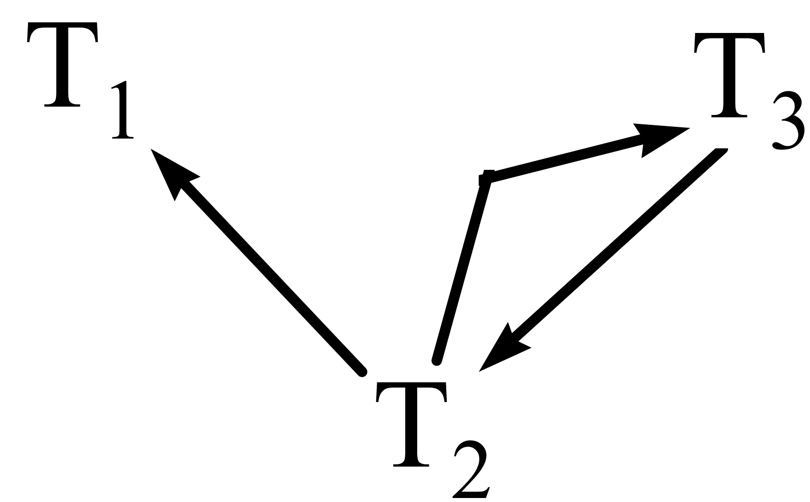
- **Step 1:** Serial Schedules are Always Correct
- **Step 2:** Schedules with the same information flow are view-equivalent.
- **Step 3:** Schedules view-equivalent to an always correct schedule are also correct.
  - … or view serializable

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?

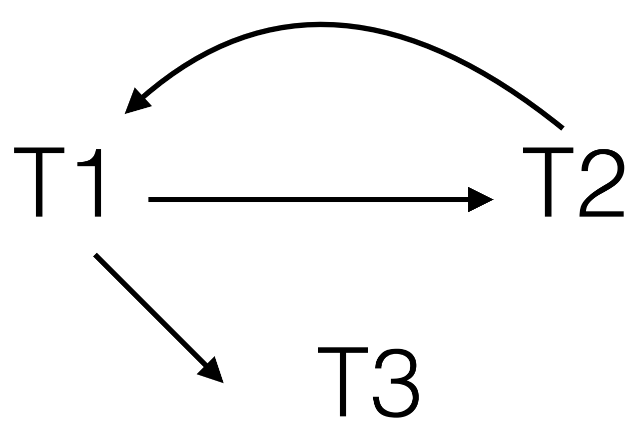
T1	T2	T3
W(a)		
	R(b)	
		W(d)
W(b)		
	R(d)	
		W(d)



Precedence Graph

Cycle!
Not Conflict serializable

Not conflict serializable but view serializable



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

Satisfies 3 conditions of view serializability

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?

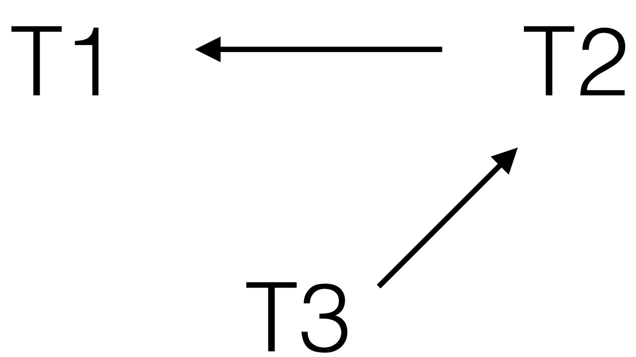
Two-Phase Locking

- 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

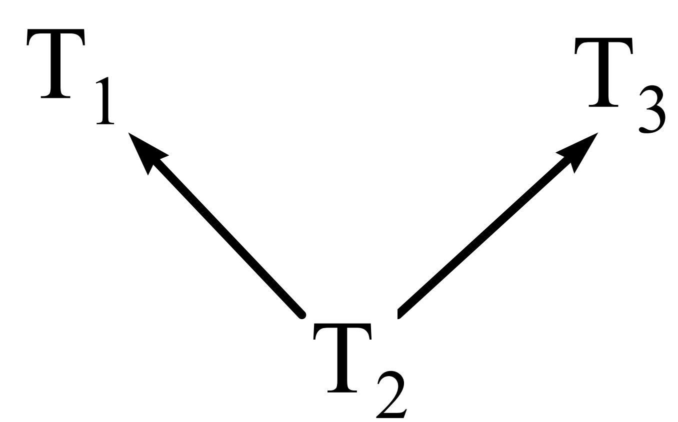
T1	T2	T3
		R(d)
W(a)		
	R(b)	
		W(d)
W(b)		
	R(d)	

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 in mode	S	Yes	No
	X	No	No

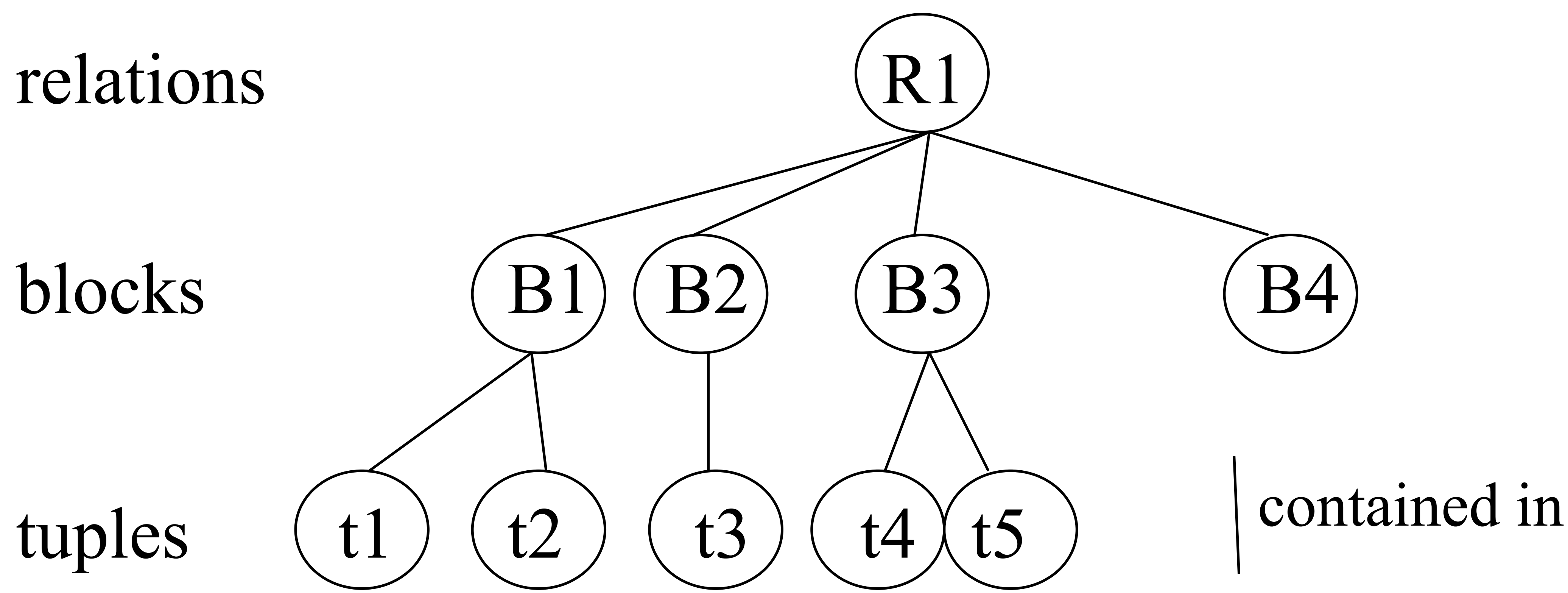
Reader/Writer (S/X)

- 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

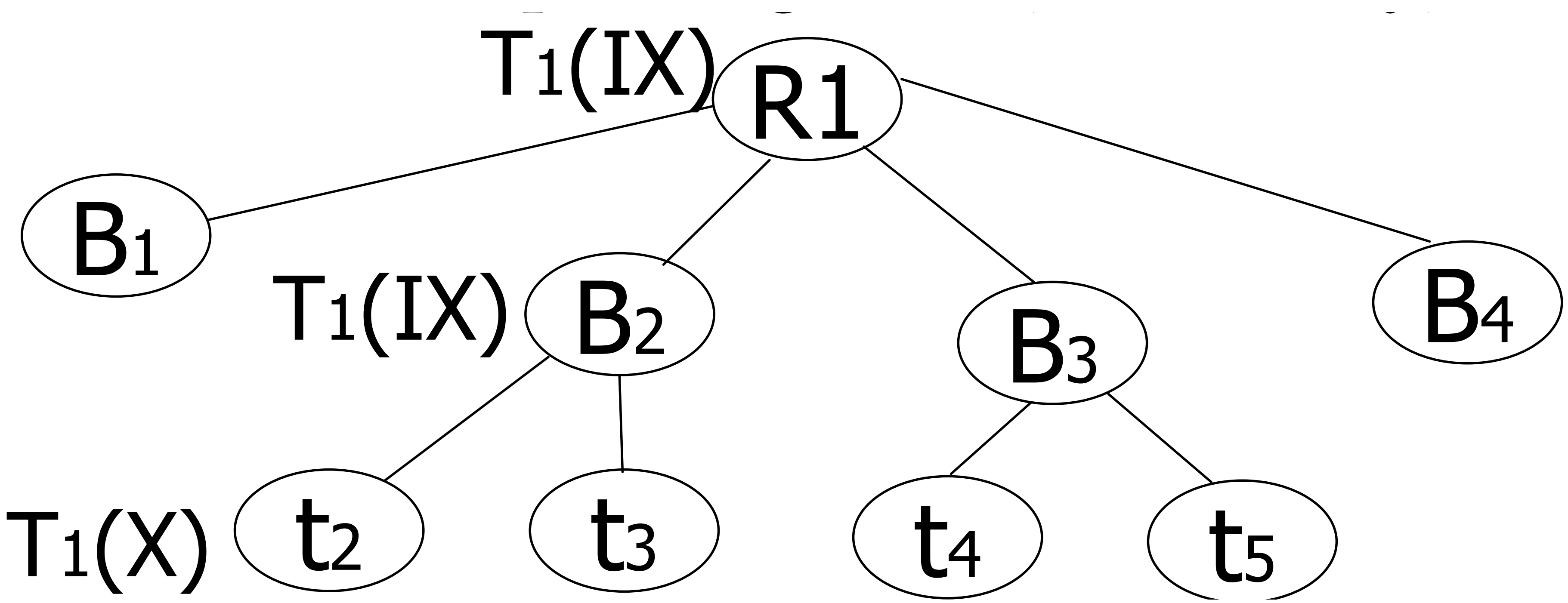
	None	IS	IX	S	X
None	valid	valid	valid	valid	valid
IS	valid	valid	valid	valid	fail
IX	valid	valid	valid	fail	fail
S	valid	valid	fail	valid	fail
X	valid	fail	fail	fail	fail

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.

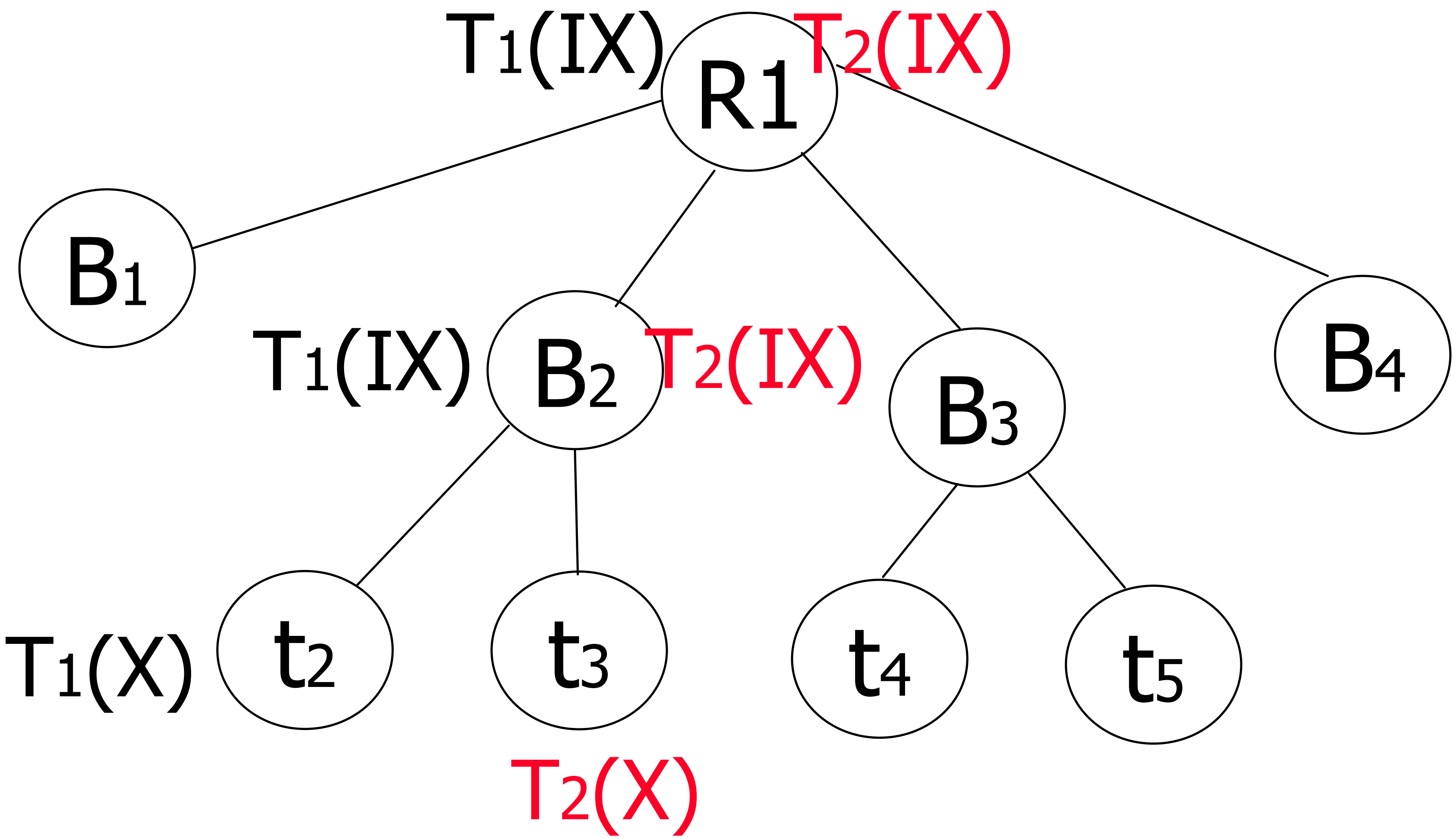
Example

- T1 wants exclusive lock on tuple t2



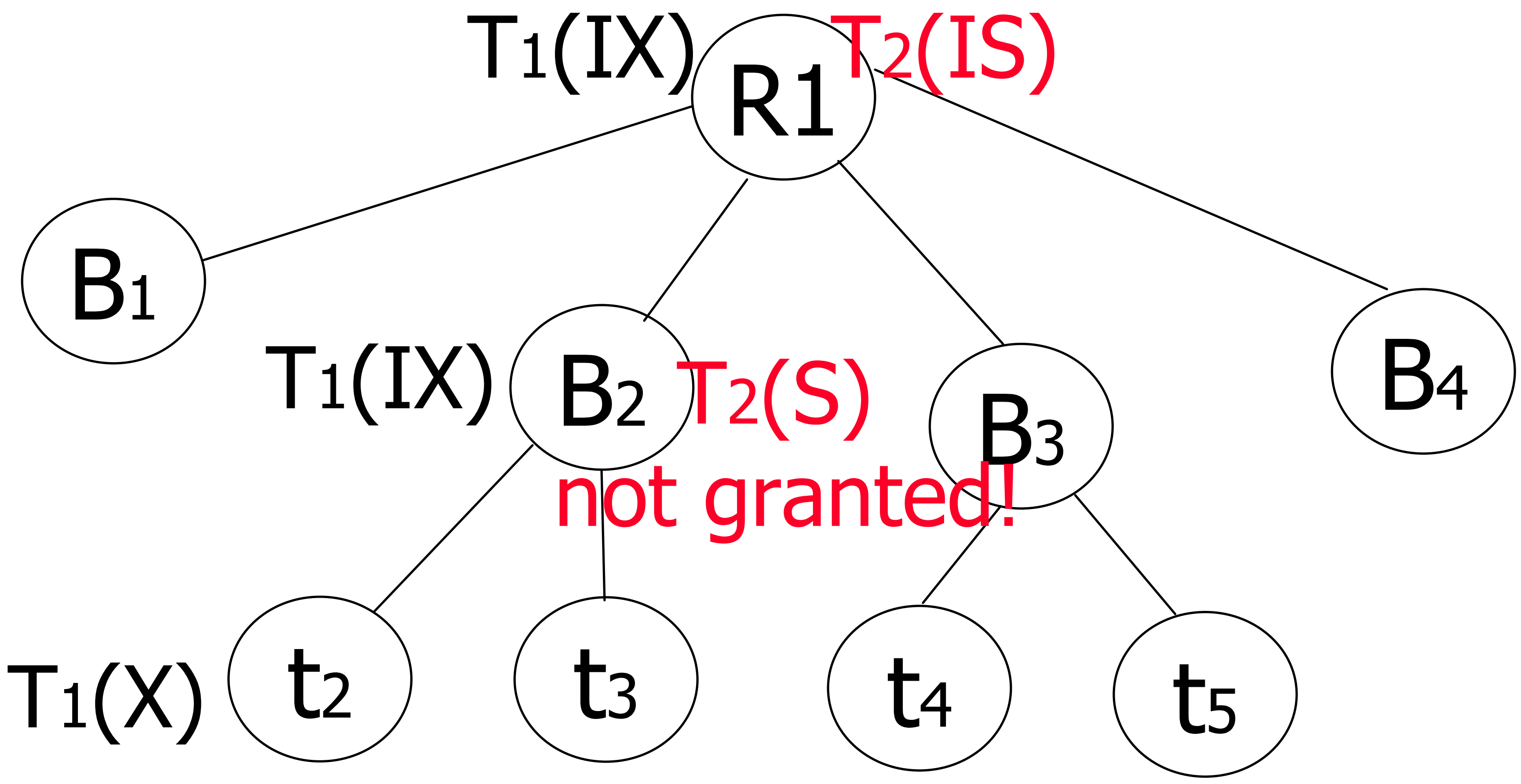
Example

- 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

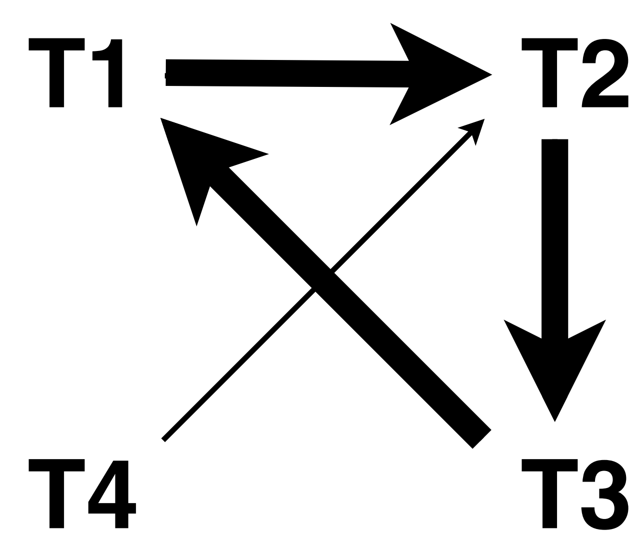
- **Baseline**: If a lock request can not be satisfied, the transaction is blocked and must wait until the resource is available.
- Create a waits-for graph:
  - Nodes are transactions
  - Edge from T_i to T_k if T_i is waiting for T_k to release a lock.
- Periodically check for cycles in the graph.

Example

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)	



How do we avoid deadlock?

Avoid Deadlock Situations

React to Deadlock Situations

Deadlock Prevention

- Ensure that dependencies are monotonic (and consequently acyclic)
- Assign each transaction a priority based on the timestamp at which it starts.
- When a transaction fails to acquire a lock:
  - Wait if monotonicity would be preserved.
  - Kill one transaction otherwise.

Deadlock Prevention

- Policy 1 (Wait-Die): If T_i has a higher priority, wait for T_k, otherwise T_i aborts.
- Policy 2 (Wait-Wound): If T_i has a higher priority, T_k aborts, otherwise T_i waits.
- Protect fairness by restarting the aborted transaction with its original timestamp.