L17: Transactions & Concurrency Control

CS3200 Database design (fa18 s2)

https://northeastern-datalab.github.io/cs3200/

Version 11/8/2018

Announcements!

- Exam2: Non-SQL part will be graded on Gradescope; feedback from you comparing exam1 vs exam2 setup welcome (after we have posted the results)
- Monday is Veteran's day (no class)
- Today
  - Transactions, some repetition (based on OH feedback) & practice, locking
- Next week
  - Lessons-learned from exam 2
  - NoSQL

Transactions

Turing Awards to Database Researchers

- Charles Bachman 1973 for CODASYL (network data model)

- Edgar Codd 1981 for relational databases

- Jim Gray 1998 for transactions
  - (describing the requirements for reliable transaction processing, memorably called the ACID test)

- Michael Stonebraker 2014 for various contributions (e.g. INGRES)

History of data models

- What Goes Around Comes Around" by Stonebraker, Hellerstein, ~2005 http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.113.5640
- "This paper provides a summary of 35 years of data model proposals, grouped into 9 different eras. We discuss the proposals of each era, and show that there are only a few basic data modeling ideas, and most have been around a long time. Later proposals inevitably bear a strong resemblance to certain earlier proposals. Hence, it is a worthwhile exercise to study previous proposals. In addition, we present the lessons learned from the exploration of the proposals in each era. Most current researchers were not around for many of the previous eras, and have limited (if any) understanding of what was previously learned. There is an old adage that he who does not understand history is condemned to repeat it. By presenting “ancient history”, we hope to allow future researchers to avoid replaying history."
- (cp. to recent references to Plato's republic in a May 2016 NewYork Magazin article by Andrew Sullivan titled "Democracies end when they are too democratic")

Transactions

- User programs may do many things on the data retrieved.
  - E.g., operations on Bob's bank account.
  - E.g. transfer of money from account A to account B.
  - E.g., search for a ticket, think about it…, and buy it.
- But the DBMS is only concerned about what data is read from/written to the database.
- A **transaction** is DBMS's abstract view of a user program, simply, a sequence of reads and writes.

Transaction Properties: ACID

a-tomos: undividable

- Atomic
  - either all actions of a txn are carried out, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (in some serial order)
- Durable
  - Once a txn has committed, its effects remain in the database (persist)

ACID continues to be a source of great debate!
BASE (Basic Availability, Soft-state, Eventual Consistency)

The World Without Transactions

- Just write applications that talk to databases

- Rely on operating systems for scheduling, and for concurrency control

- What can go wrong ?
  - Several famous anomalies
  - Other anomalies are possible (but not famous)

The Problem

- Multiple transactions are running concurrently
  T1, T2, …
- They read/write some common elements
  A1, A2, B, X, Y, …
- How can we prevent unwanted interference ?
- The **SCHEDULER** is responsible for that

Scheduling: the concept of Serializability

Why Interleave transactions?

Why? To <u>improve performance</u> (execution time).

- Individual TXNs might be slow; we don't want to block other users during TXN (<u>memory hierarchy</u>: let some TXNs use CPU while others accessing HD)
- By interleaving queries, users do not have to wait for other user's transactions to complete fully before their own transaction begins.
- E.g. without interleaving: user A begins a transaction that takes 10 sec to complete. User B begins a transaction, but has to wait an additional 10 seconds before the database would begin processing user B's request.

But: Interleaving TXNs might lead to <u>anomalies</u>

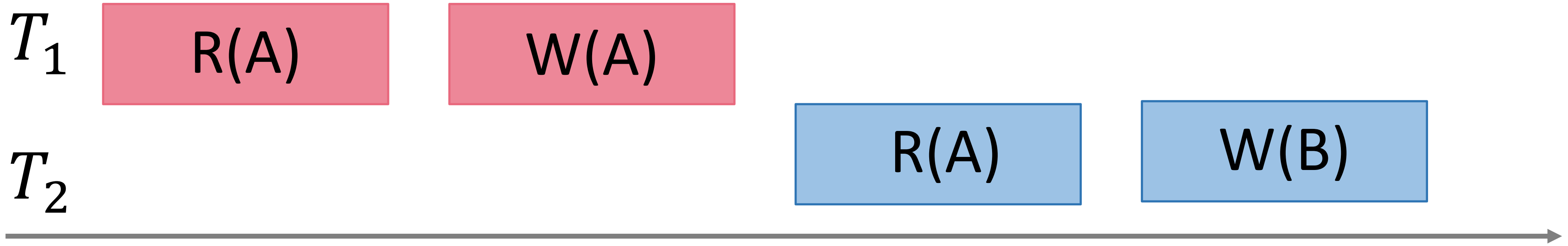
- we want to avoid those; so we need to be smart about how to avoid "bad" things from happening (in a principled way)

How? TXNs must be as if executed serially!

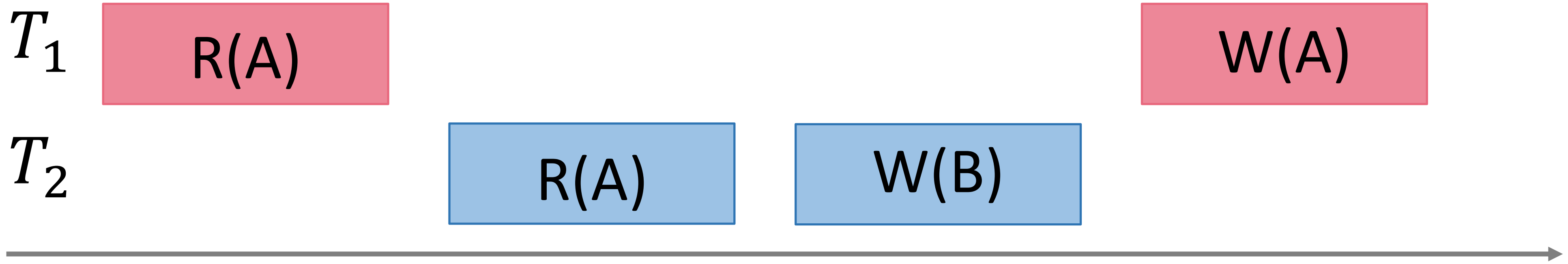
- That way <u>isolation</u> and <u>consistency</u> are maintained

Schedules

- **Schedule** – A chronological sequence of actions from all transactions
- **Serial Schedule** – A schedule in which transactions are aligned such that one transaction is executed first before the next transaction is executed.



- **Interleaved Schedule** – A schedule in which actions of different transactions are interleaved



Serializability

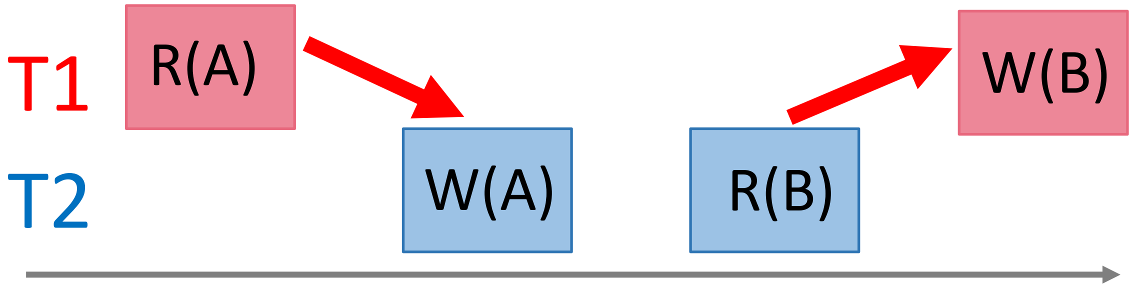
- A **serializable schedule** is a schedule that is equivalent to some serial execution of the transactions.
- The execution sequence within a transaction cannot be changed, but two transactions can have their instructions **interleaved**
  - Notice: interleaving does no harm if two transactions are working on different segments of data
  - If the two transactions are working on same data ("**conflicts**"), interleaving may or may not bring the database to an inconsistent state

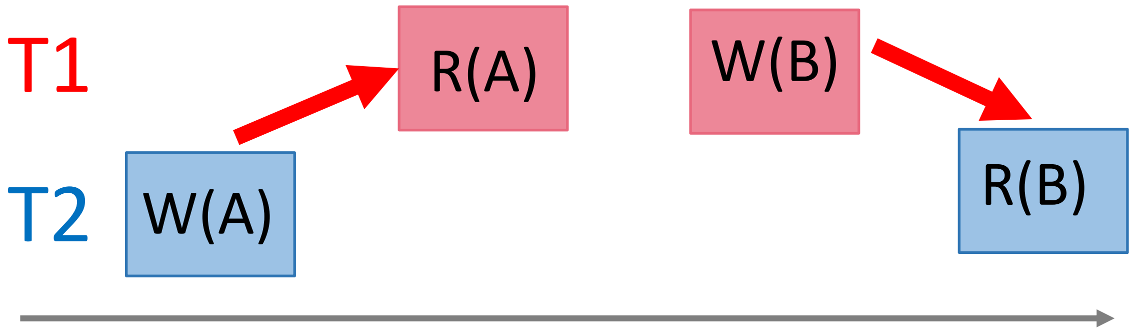
Conflict Types

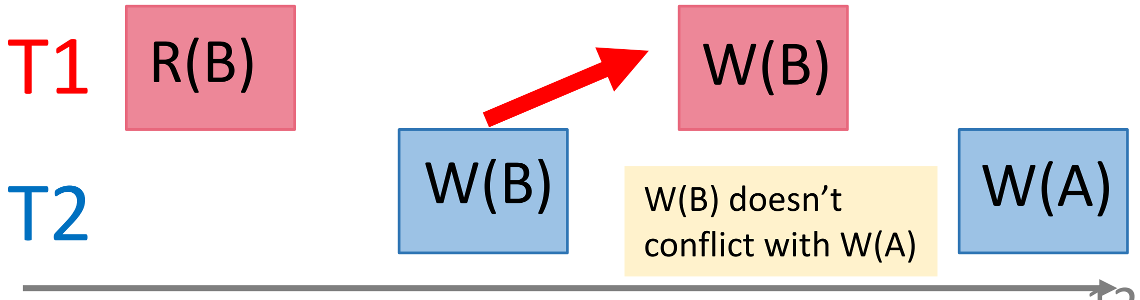
Two actions **conflict** if they are part of different TXNs, involve the same data item, and at least one of them is a write

3 types of conflicts:

- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)







Interleaving anomalies occur with/because of these conflicts between TXNS **BUT these conflicts can occur w/o causing anomalies – Conflict serializable**

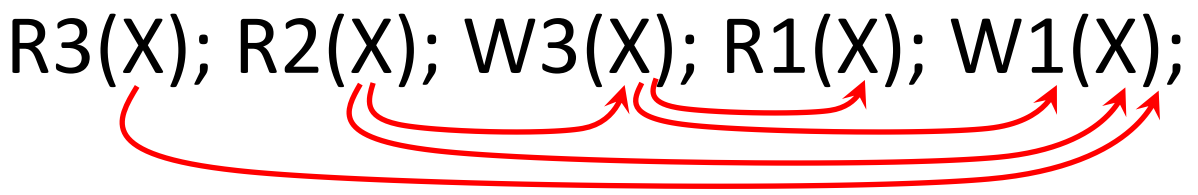
How to resolve: Conflict Equivalence

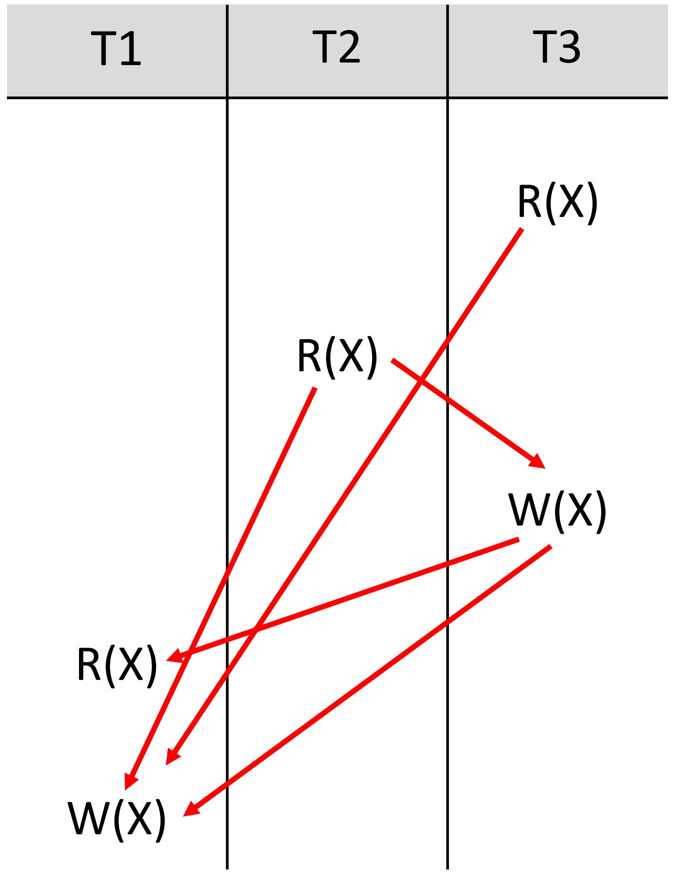
- Two schedules having multiple transactions with conflicting operations are **conflict equivalent** if and only if:
  - Both schedules contain the same set of transactions
  - The order of conflicting pairs of operation is maintained in both the schedules
- Schedules which are conflict equivalent with another serial schedule are **conflict serializable**

Practice

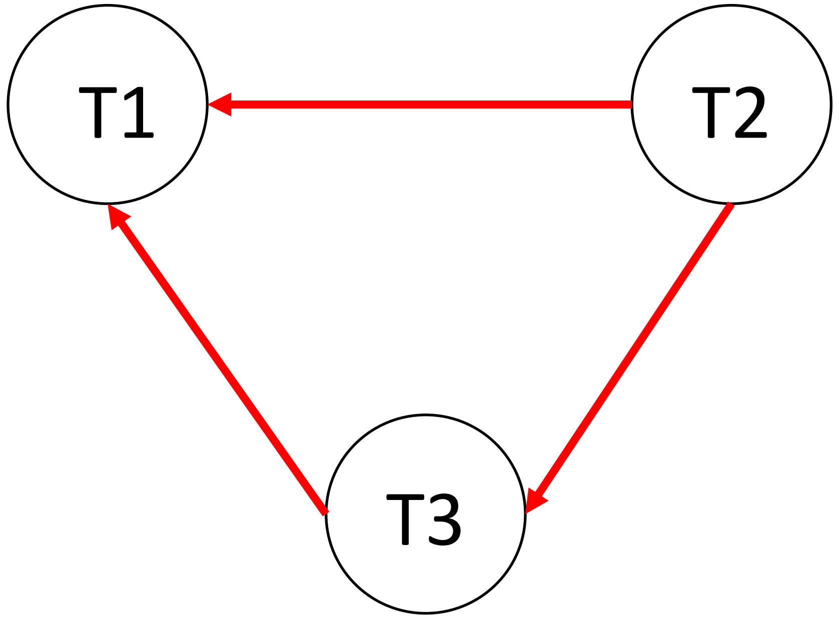
Example transactions: Serializable or not?

R3(X); R2(X); W3(X); R1(X); W1(X);





Serialization graph



No directed cycles: Serializable

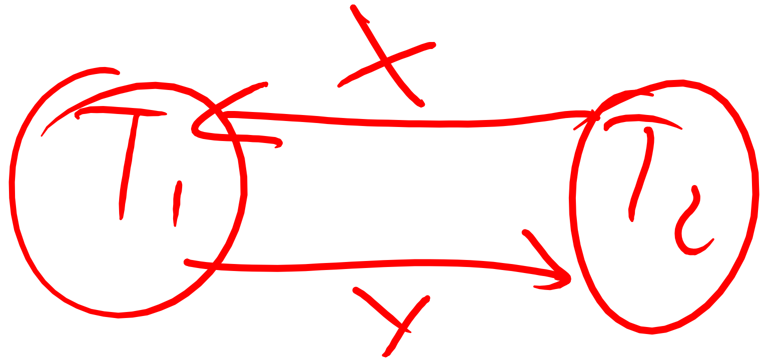
Equivalent serial schedule: R2(X); R3(X); W3(X); R1(X); W1(X);

Example transactions 2: Serializable or not?

- $w_1(Y)$, $r_2(X)$,$r_2(Y)$, $w_1(X)$, c_1, $r_2(Z)$, $w_2(Y)$, $w_2(Z)$, c_2
- [Draw on whiteboard]

Example transactions 2: Serializable or not?

- $w_1(Y)$, $r_2(X)$,$r_2(Y)$, $w_1(X)$, c_1, $r_2(Z)$, $w_2(Y)$, $w_2(Z)$, c_2
- [Draw on whiteboard]



- No: because of X, T_2 must precede T_1, but because of Y the opposite is true

Example

T1	T2
READ(A, t)	READ(A,s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

T1	T2	A	B
READ(A, t)		25	25
t := t+100			
WRITE(A, t)			
READ(B, t)			
t := t+100			
WRITE(B,t)			
	READ(A,s)		
	s := s*2		
	WRITE(A,s)		
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		

Serial schedule: (T1,T2)

A Serial Schedule

T1	T2	A	B
READ(A, t)		25	25
t := t+100			
WRITE(A, t)		125	
READ(B, t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)		
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		

Serial schedule: (T1,T2)

A Serial Schedule

T1	T2	A	B
READ(A, t)		25	25
t := t+100			
WRITE(A, t)		125	
READ(B, t)			
t := t+100			
WRITE(B,t)			125
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Serial schedule: (T1,T2)

A Serial Schedule (version 2)

T1	T2	A	B
	READ(A,s)	25	25
	s := s*2		
	WRITE(A,s)	50	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(A, t)			
t := t+100			
WRITE(A, t)		150	
READ(B, t)			
t := t+100			
WRITE(B,t)			150

Serial schedule: (T2,T1)

Serializable Schedule

- A schedule is serializable if it is equivalent to a serial schedule

A schedule S is <u>serializable</u>, if there is a serial schedule S’, such that for <u>every initial database</u> state, the effects of S and S’ are the same

A schedule that is not serial

T1	T2	A	B
READ(A, t)		25	25
t := t+100			
WRITE(A, t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B, t)			
t := t+100			
WRITE(B,t)			
	READ(B,s)		125
	s := s*2		
	WRITE(B,s)		250

Notice:
But it is serializable

A schedule that is not serial, and non-serializable

T1	T2	A	B
READ(A, t)		25	25
t := t+100			
WRITE(A, t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(B, t)			
t := t+100			
WRITE(B,t)			150

Transaction Semantics

Is this serializable?

T1	T2	A	B
READ(A, t)		25	25
t := t+100			
WRITE(A, t)		125	
	READ(A,s)		
	s := s+200		
	WRITE(A,s)	325	
	READ(B,s)		
	s := s+200		
	WRITE(B,s)		225
READ(B, t)			
t := t+100			
WRITE(B,t)			325

Ignoring Details

- Serializability is undecidable!
- Scheduler should not look at transaction details
- Assume worst case updates
  - Only care about reads r(A) and writes w(A)
  - Not the actual values involved

Notation

actions

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

transaction

schedule

$r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$

Conflict Serializability

Conflicts:

Two actions by same transaction T_i: $r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element: $w_i(X); w_j(X)$

Read/write by T_i, T_j to same element: $w_i(X); r_j(X)$ $r_i(X); w_j(X)$

Conflict Serializability

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions.
  - Every pair of conflicting actions is ordered the same way.
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.
- Given a set of X's, conflict serializable schedules are a subset of serializable schedules.
  - There are serializable schedules that can't be detected using conflict serializability.

Conflict Serializability

- A schedule is conflict serializable if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); w_1(B); w_2(A); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

The Precedence Graph Test

Is a schedule conflict-serializable ?

Simple test:

- Build a graph of all transactions T_i
- Edge from T_i to T_j if T_i makes an action that conflicts with one of T_j and comes first
- The test: if the graph has no cycles, then it is conflict serializable !

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

$$\begin{tikzcd}
1 \arrow[r, "B"] & 2 \arrow[r, "A"] & 3
\end{tikzcd}$$

This schedule is conflict-serializable

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

$$\begin{tikzcd}
1 \arrow[r, "B"'] & 2 \arrow[l, bend right, "B"'] \arrow[r, "A"] & 3
\end{tikzcd}$$

This schedule is NOT conflict-serializable

Conflict Serializability

- A serializable schedule need not be conflict serializable

Lost write

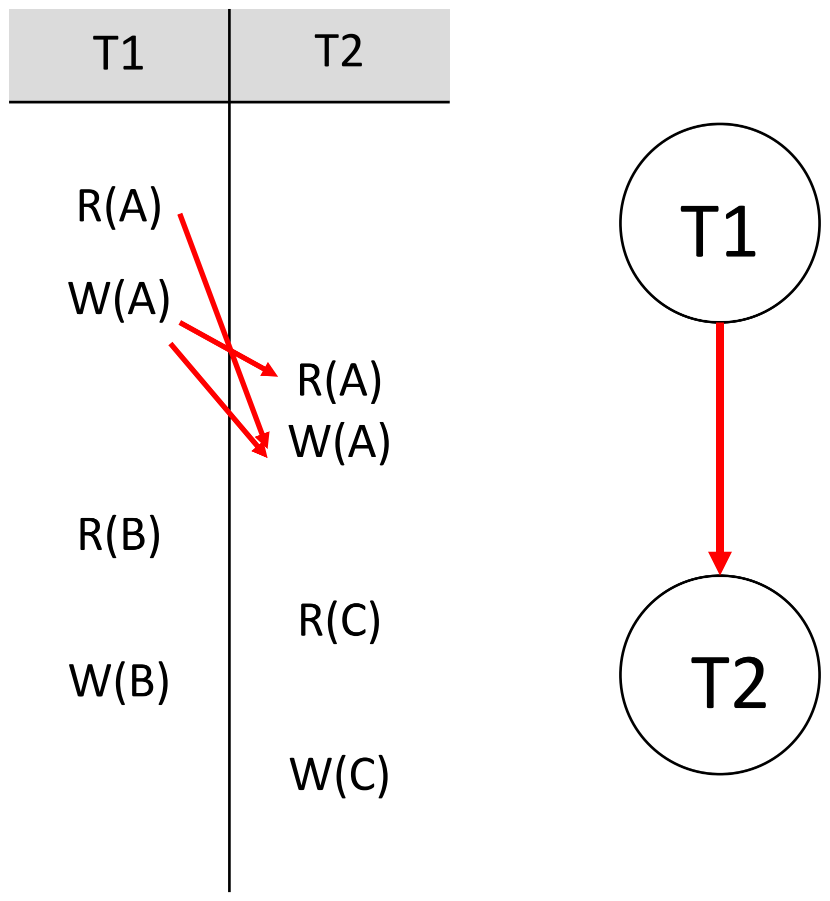
$w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$

$$\not\Downarrow$$

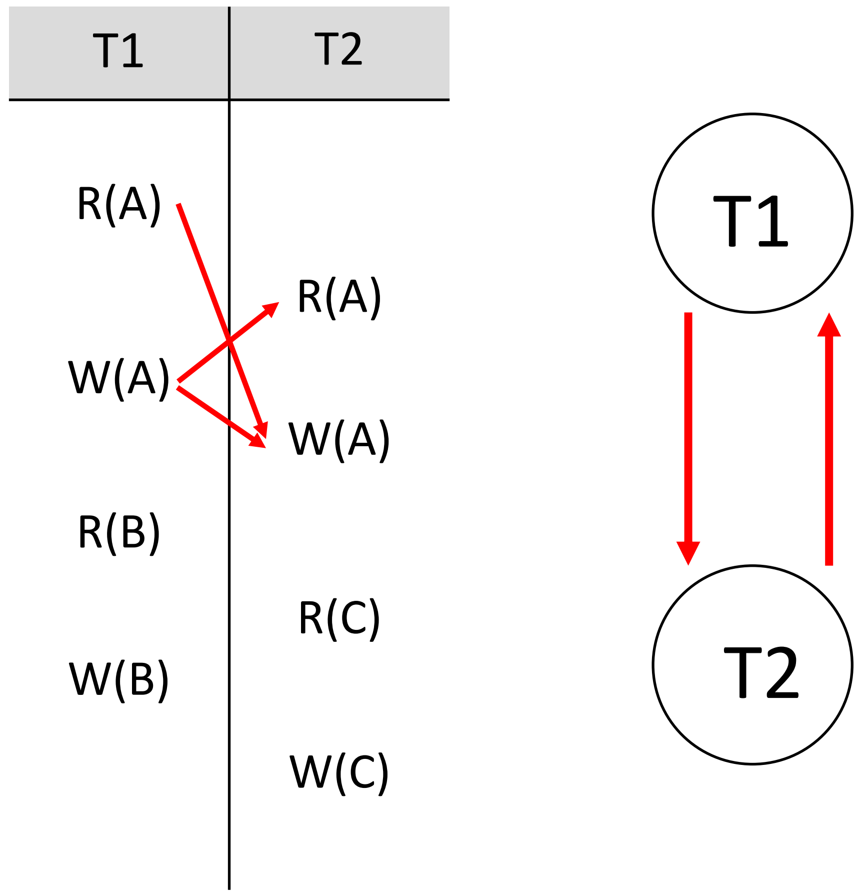
$w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$

Equivalent, but can’t swap

Another look at "Good" vs "Bad"



T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
	R(C)
W(B)	
	W(C)



T1	T2
R(A)	
	R(A)
W(A)	
	W(A)
R(B)	
	R(C)
W(B)	
	W(C)

Locking

How to enforce (conflict) serializability?

Simple idea:

- Each element has a unique lock
- Each transaction must first acquire the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must release the lock(s)

Some things still can go wrong. Let's "experience" some problems.

Notation

L_i(A) = transaction T_i acquires lock for element A

U_i(A) = transaction T_i releases lock for element A

Example

T1	T2
L_1(A); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A); L_1(B)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s); U_2(A);
	L_2(B); **DENIED…**
READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	
	…**GRANTED;** READ(B,s)
	s := s*2
	WRITE(B,s); U_2(B);

Scheduler has ensured a conflict-serializable schedule

Example

T1	T2
L_1(A); READ(A, t)	
t := t+100	
WRITE(A, t); U_1(A)	
	L_2(A); READ(A,s)
	s := s*2
	WRITE(A,s); U_2(A);
	L_2(B); READ(B,s)
	s := s*2
	WRITE(B,s); U_2(B);
L_1(B); READ(B, t)	
t := t+100	
WRITE(B,t); U_1(B);	

Locks did not enforce conflict serializability ☹ !!

Strict Two-Phase Locking ("2PL")

- Two-phase locking is a way to deal with concurrency, because it guarantees conflict serializability (if it completes…)

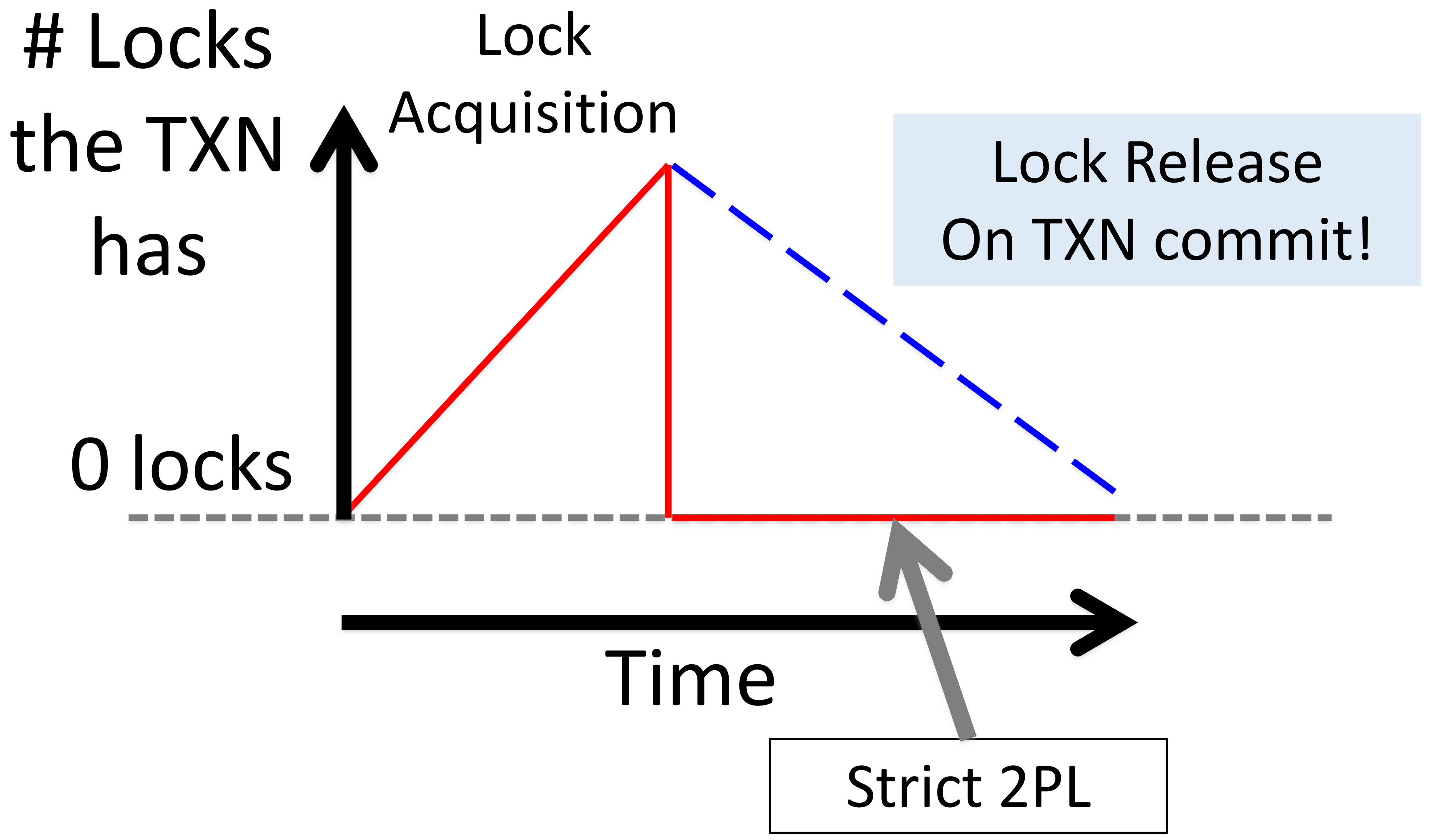
- Also (conceptually) straightforward to implement, and transparent to the user!

Strict Two-Phase Locking (Strict 2PL) Protocol:

- TXNs obtain:
- An <u>X (exclusive) lock</u> on object before <u>writing</u>
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An <u>S (shared) lock</u> on object before <u>reading</u>
  - If a TXN holds, no other TXN can get an X lock on that object
- All locks held by a TXN are released when TXN completes.

Note: Terminology here- “exclusive”, “shared”- meant to be intuitive- no tricks!

Picture of 2-Phase Locking (2PL)



Strict 2PL

Theorem: Strict 2PL allows only schedules whose dependency graph is acyclic

Proof Intuition: In strict 2PL, if there is an edge $T_i \rightarrow T_j$ (i.e. T_i and T_j conflict) then T_j needs to wait until T_i is finished – so *cannot* have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable $\Rightarrow$ serializable schedules

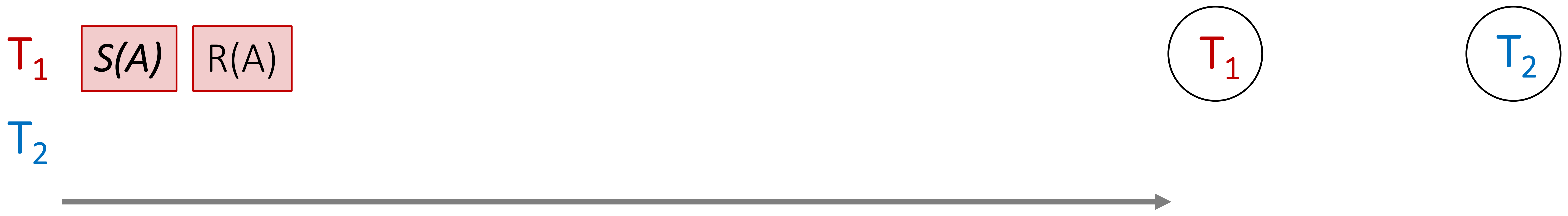
Strict 2PL

- If a schedule follows strict <u>2PL and locking</u>, it is <u>conflict serializable</u>…
  - …and thus serializable
  - …and thus maintains isolation & consistency!

- Not all serializable schedules are allowed by strict 2PL.

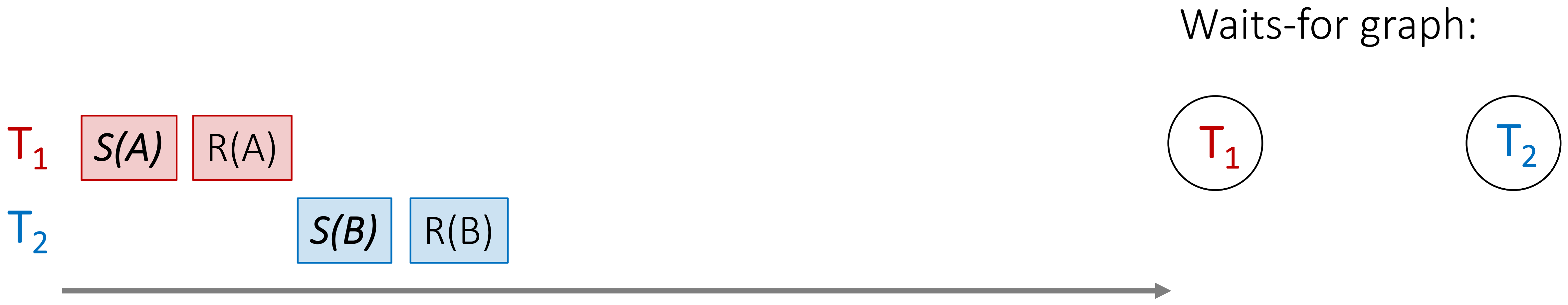
- So let’s use strict 2PL, what could go wrong?

Deadlock Detection: Example



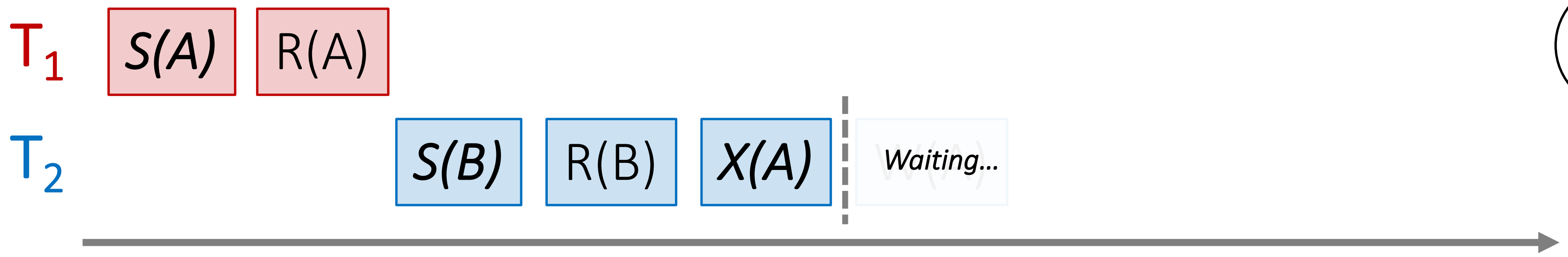
First, T_1 requests a shared lock on A to read from it

Deadlock Detection: Example

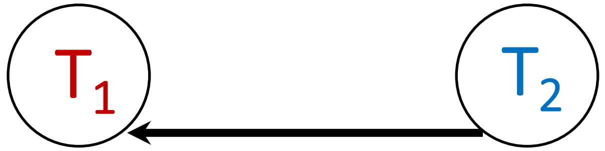


Next, T_2 requests a shared lock on B to read from it

Deadlock Detection: Example

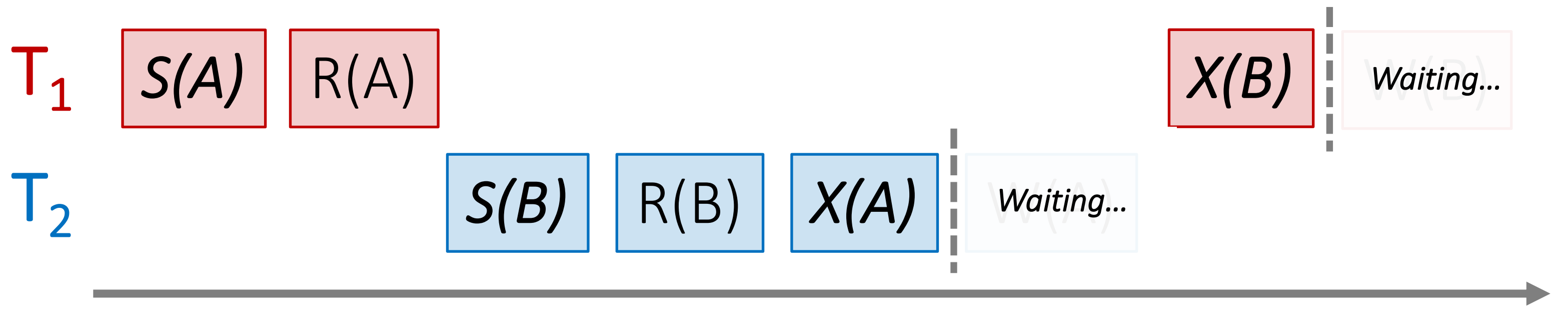


Waits-for graph:

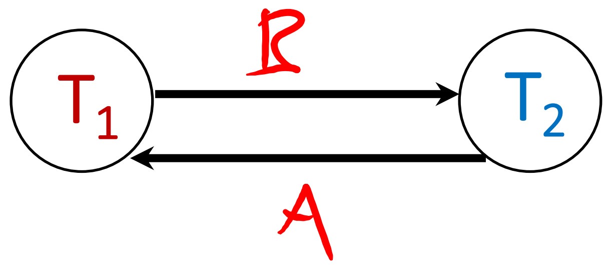


T_2 then requests an exclusive lock on A to write to it- **now T_2 is waiting on T_1…**

Deadlock Detection: Example



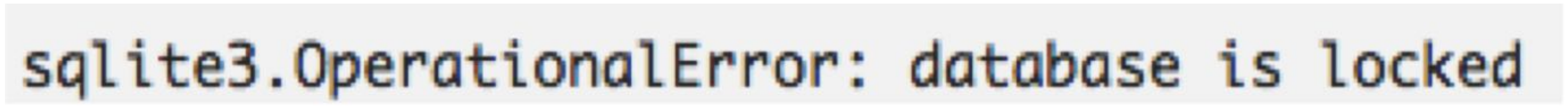
Waits-for graph:



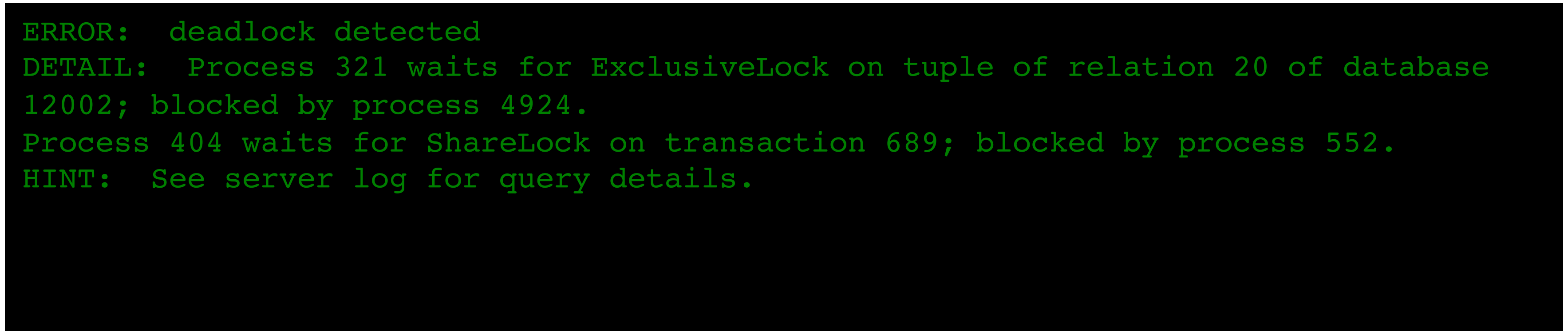
Cycle = DEADLOCK

Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2… DEADLOCK!**

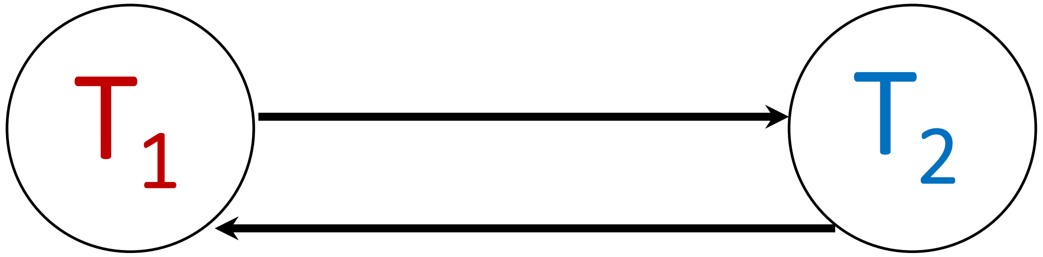
The problem? Deadlock!??!



```
sqlite3.OperationalError: database is locked
```



```
ERROR:  deadlock detected
DETAIL:  Process 321 waits for ExclusiveLock on tuple of relation 20 of database
12002; blocked by process 4924.
Process 404 waits for ShareLock on transaction 689; blocked by process 552.
HINT:  See server log for query details.
```



$$\begin{tikzcd} T_1 \arrow[r, bend left] & T_2 \arrow[l, bend left] \end{tikzcd}$$

Deadlock

Deadlock

- Transaction T1 waits for a lock held by T2;
- But T2 waits for a lock held by T3;
- While T3 waits for . . . .
- . . .
- . . .and T73 waits for a lock held by T1 !!

- Could be avoided, by ordering all elements, or deadlock detection + rollback

Deadlocks

- <u>Deadlock</u>: Cycle of transactions <u>waiting</u> for locks to be released by each other.

- Two ways of dealing with deadlocks:
  - Deadlock <u>detection</u>
  - Deadlock <u>prevention</u>

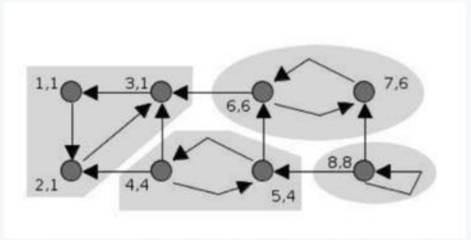
Deadlock Detection

- Create the waits-for graph:
  - Nodes are transactions
  - There is an edge from $T_i \rightarrow T_j$ if T_i is waiting for T_j to release a lock
- Periodically check for (and break) cycles in the waits-for graph

Finding directed cycles in a graph in O(|V|+|E|)

Tarjan's algorithm is an algorithm in graph theory for finding the strongly connected components of a directed graph. It runs in linear time, matching the time bound for alternative methods including Kosaraju's algorithm and the path-based strong component algorithm. Tarjan's algorithm is named for its inventor, Robert Tarjan.[1]

Tarjan's strongly connected components algorithm



Tarjan's algorithm animation

Data structure	Graph
Worst-case performance	$O(\vert V\vert + \vert E\vert)$

Donald Knuth described Tarjan's algorithm as one of Knuth's favorite implementations in his book *The Stanford GraphBase*.[5]

He also wrote:[6]

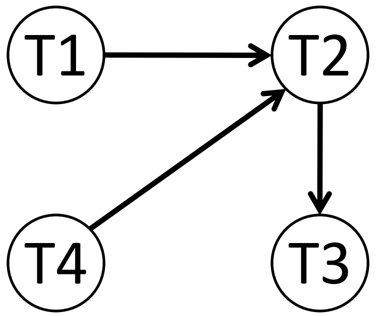
> The data structures that he devised for this problem fit together in an amazingly beautiful way, so that the quantities you need to look at while exploring a directed graph are always magically at your fingertips. And his algorithm also does topological sorting as a byproduct.

Source: https://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

Deadlock: example

T1	T2	T3	T4
L(A)			
R(A)			
	L(B)		
	W(B)		
L(B)			
		L(C)	
		R(C)	
	L(C)		
			L(B)

Waits-for graph


$$\begin{tikzcd} T1 \arrow[r] & T2 \arrow[d] \\ T4 \arrow[ur] & T3 \end{tikzcd}$$

Deadlock: example

T1	T2	T3	T4
L(A)			
R(A)			
	L(B)		
	W(B)		
L(B)			
		L(C)	
		R(C)	
	L(C)		
			L(B)
		L(A)	

Waits-for graph

$$\begin{tikzcd} T1 \arrow[r] & T2 \arrow[d] \\ T4 \arrow[ur] & T3 \arrow[ul] \end{tikzcd}$$

Deadlock!

Most systems do deadlock detection

Deadlock prevention

T_i requests a lock conflicting with T_j

- Wait-die:
  - If T_i has higher priority, it waits; otherwise it is aborted
- Wound-wait:
  - If T_i has higher priority, abort T_j; otherwise T_i waits

Conservative 2PL

- Acquire all locks at the beginning

Summary

- Concurrency achieved by <u>interleaving TXNs</u> such that <u>isolation & consistency</u> are maintained
  - We formalized a notion of <u>serializability</u> that captured such a “good” interleaving schedule

- We defined <u>conflict serializability</u>, which implies serializability

- <u>Careful Locking</u> allows only conflict serializable schedules
  - If the schedule completes… (it may deadlock!)