Transactions, Concurrency, Recovery

Concepts

Serial schedule

One transaction at a time. no concurrency.

Equivalent schedule

the database state is the same at end of both schedules

Serializable schedule (gold standard)

equivalent to a serial schedule

SQL → R/W Operations

```
UPDATE  accounts
SET     bal = bal + 1000
WHERE   bal > 1M
```

Read all balances for every tuple

Update those with balances > 1000

Does the access method mater?

Why Serializable Schedule? Anomalies

Reading in-between (uncommitted) data

T1: R(A) W(A) R(B) W(B) abort

T2: R(A) W(A) commit

WR conflict or dirty reads

Reading same data gets different values

T1: R(A) R(A) W(A) commit

T2: R(A) W(A) commit

RW conflict or unrepeatable reads

Why Serializable Schedule? Anomalies

Stepping on someone else's writes

T1: W(A) W(B) commit

T2: W(A) W(B) commit

WW conflict or lost writes

Notice: all anomalies involve writing to data that is read/written to.

If we track our writes, maybe can prevent anomalies

Conflict Serializability

What is a conflict?

For 2 operations, if run in different order, get different results

Conflict?	R	W
R	**NO**	YES
W	YES	YES

Conflict Serializability

def: a schedule that is conflict equivalent to a serial schedule

Meaning: you can swap non-conflicting operations to derive a serial schedule.

$\forall$ conflicting operations O1 of T1, O2 of T2

O1 always before O2 in the schedule or

O2 always before O1 in the schedule

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	w	x	y	z
T2:	R(A)	W(A)	R(B)	W(B)

Logical

Conflicts

1x, 2w, 2x, 3z, 4y, 4z

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	w	x	y	z
T2:	R(A)	W(A)	R(B)	W(B)

Serializable

	1	2			3	4		
T1:	R(A)	W(A)			R(B)	W(B)		
			w	x			y	z
T2:			R(A)	W(A)			R(B)	W(B)

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	w	x	y	z
T2:	R(A)	W(A)	R(B)	W(B)

Not Serializable

	1		2		3	4		
T1:	R(A)		W(A)		R(B)	W(B)		
		w		x			y	z
T2:		R(A)		W(A)			R(B)	W(B)

Conflict Serializability

Transaction Precedence Graph

Edge Ti → Tj if:

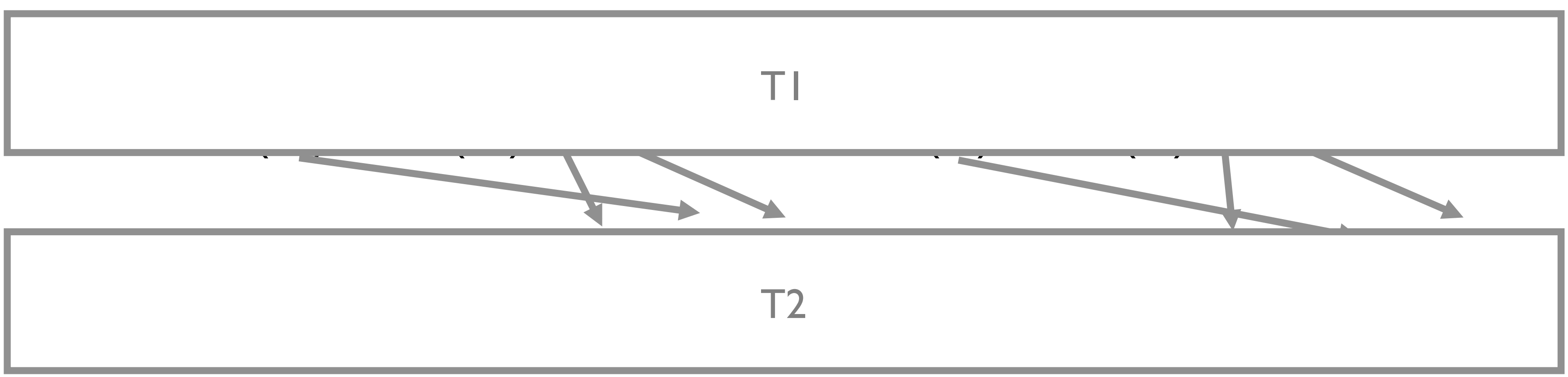
1. Ti read/write A before Tj writes A or
2. Ti writes some A before Tj reads A

If graph is acyclic (does not contain cycles) then conflict serializable!

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

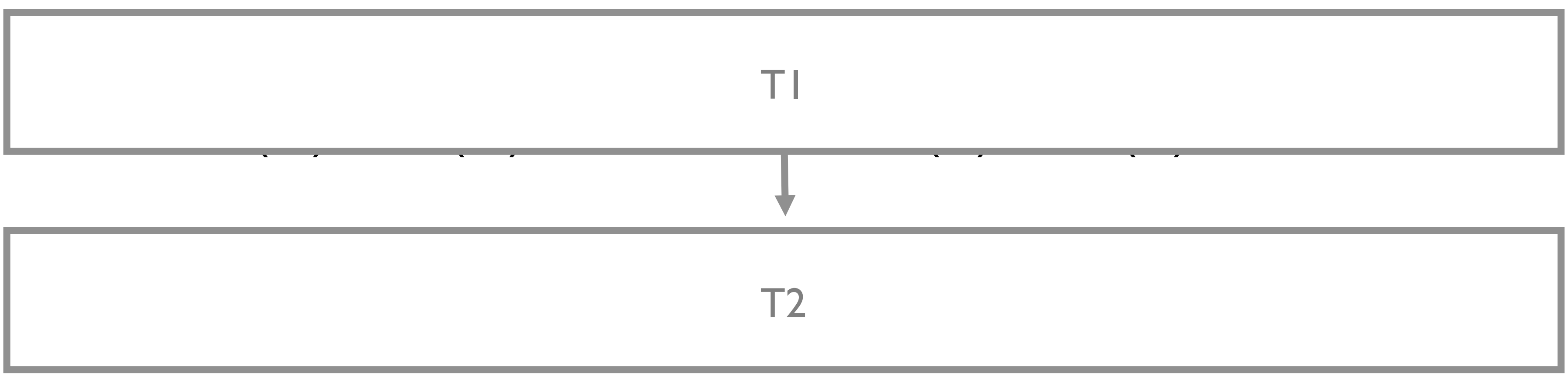
Serializable



Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

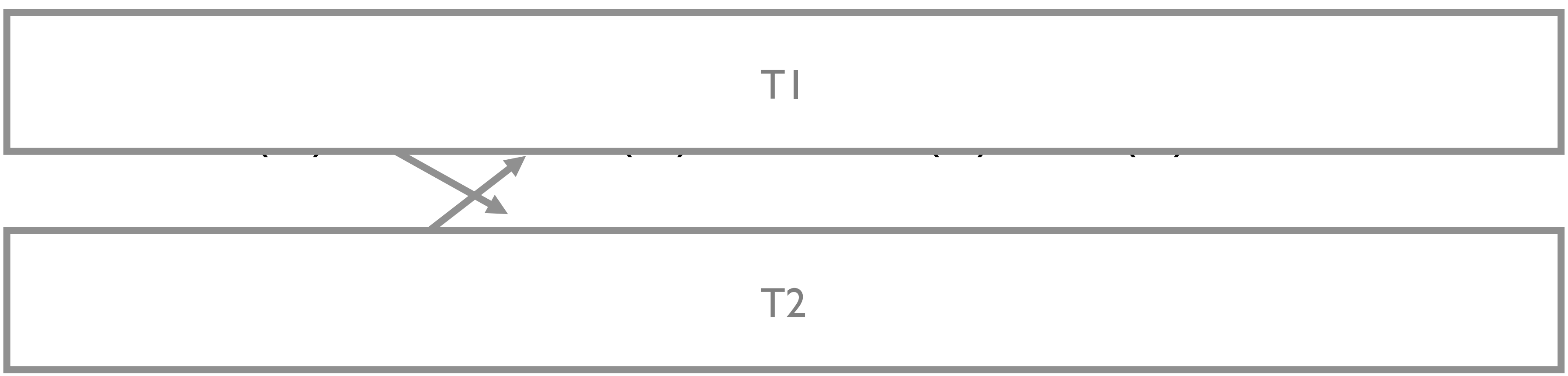
Serializable



Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Not Serializable



Fine, but what about COMMITing?

T1	R(A)	W(A)			R(B)	ABORT
T2			R(A)	COMMIT		

Not recoverable

Promised T2 everything is OK. IT WAS A LIE.

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

Cascading Rollback.

T2 read uncommitted data → T1's abort undos T1's ops & T2's

Lock-based Concurrency Control

Lock the object before reading or writing

Read: Other readers permitted, no writers (shared = S)

Write: Must be only transaction accessing (exclusive = X)

		T1	
	Other Allowed?	S	X
T2	S	Y	N
	X	N	N

Aside: What is a lock?

An abstraction

Hide complex implementation behind a simple to understand and easy to use interface

Aside: How do locks work?

Single CPU era: Operating system implementation

Process A: Give me access to X

OS: Okay! No one is using it

Process B: Give me access to X

OS: I’ll stop running you until A is done

Aside: How do locks work?

Multi CPU era: Special CPU instructions

CPU A: I’m going to write X

CPU B: I’m going to write X

Some hardware somewhere: CPU B wins!

Aside: How do locks work?

How does the hardware work?

No clue

Ancient history example:

CPU A asserts “A exclusive” wire and waits

CPU B asserts “B exclusive” wire and waits

CPU B sees A’s signal and stops; waits and retries

Abstractions

Transactions: Multiple reads/writes

Locks: Exclusive access; wait if not available

CPU instructions: One writer wins

Hardware: Wire delays? Special circuits?

Abstractions

Transactions: Multiple reads/writes

Locks: Exclusive access; wait if not available

CPU instructions: One writer wins

Hardware: Wire delays? Special circuits?

Physics???: Speed of electrons through copper?

Abstractions

Transactions: Multiple reads/writes

Locks: Exclusive access; wait if not available

CPU instructions: One writer wins

Hardware: Wire delays? Special circuits?

Physics???: Speed of electrons through copper?

Build complex systems out of simple interfaces

Lock-based Concurrency Control

Lock the object before reading or writing

Read: Other readers permitted, no writers (shared = S)

Write: Must be only transaction accessing (exclusive = X)

		T1	
	Other Allowed?	S	X
T2	S	Y	N
	X	N	N

Can this schedule happen?

T1	R(A)	W(A)			R(B)	ABORT
T2			R(A)	COMMIT		

OOPS! Maybe

Two-Phase Locking (2PL)

Growing phase: acquire locks

Shrinking phase: release locks

Guarantees serializable schedules!

shrink here

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

Uh Oh, same problem

Lock-based Concurrency Control

Strict two-phase locking (Strict 2PL)

- Growing phase: acquire locks
- Shrinking phase: release locks
- Release locks after commit/abort

Why? Which problem does it prevent?

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

Avoids cascading rollbacks!

Deadlocks

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

Possible for a cycle of transactions to wait forever

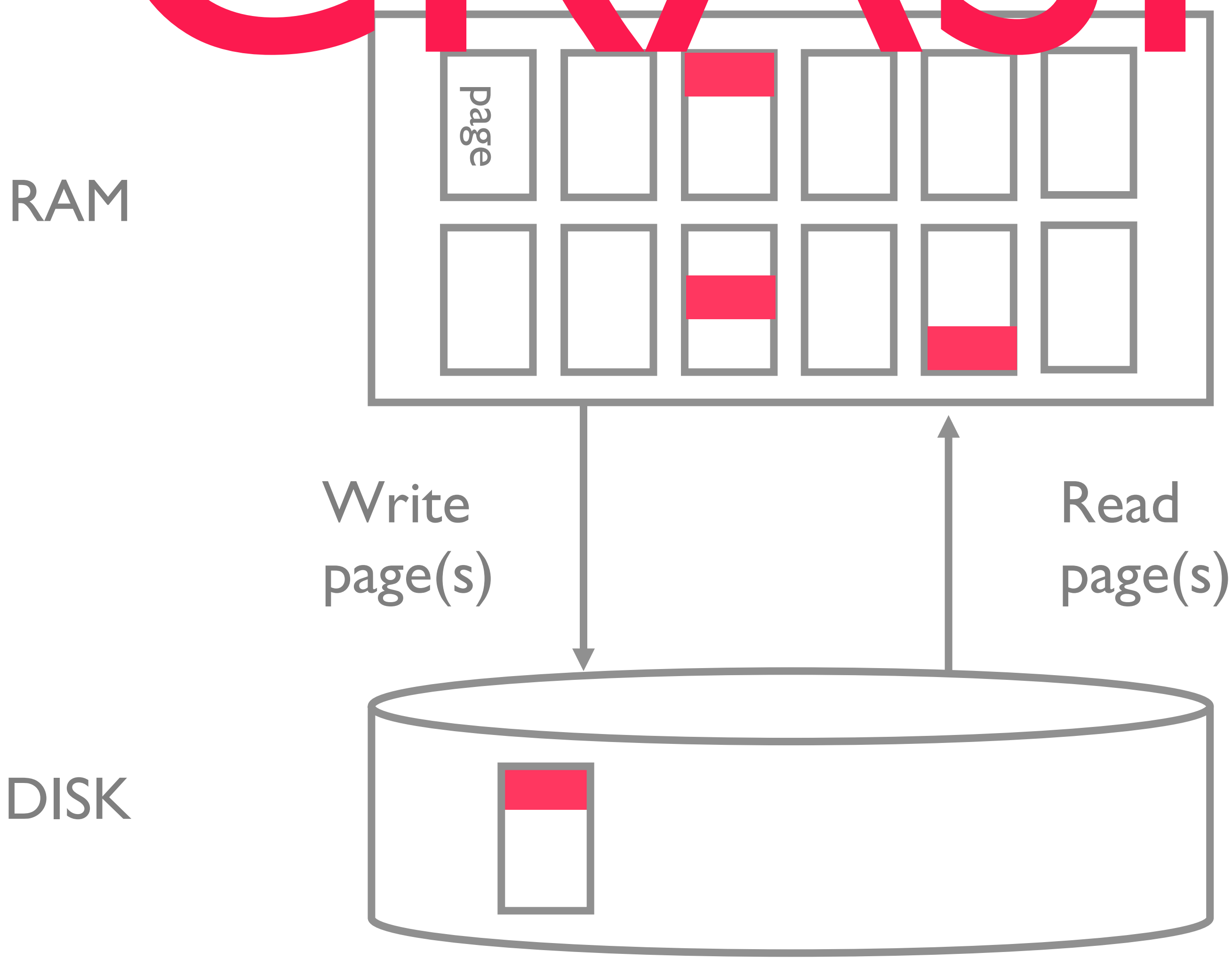
Typical solution: abort txn if waiting too long
(lock timeout)

Review

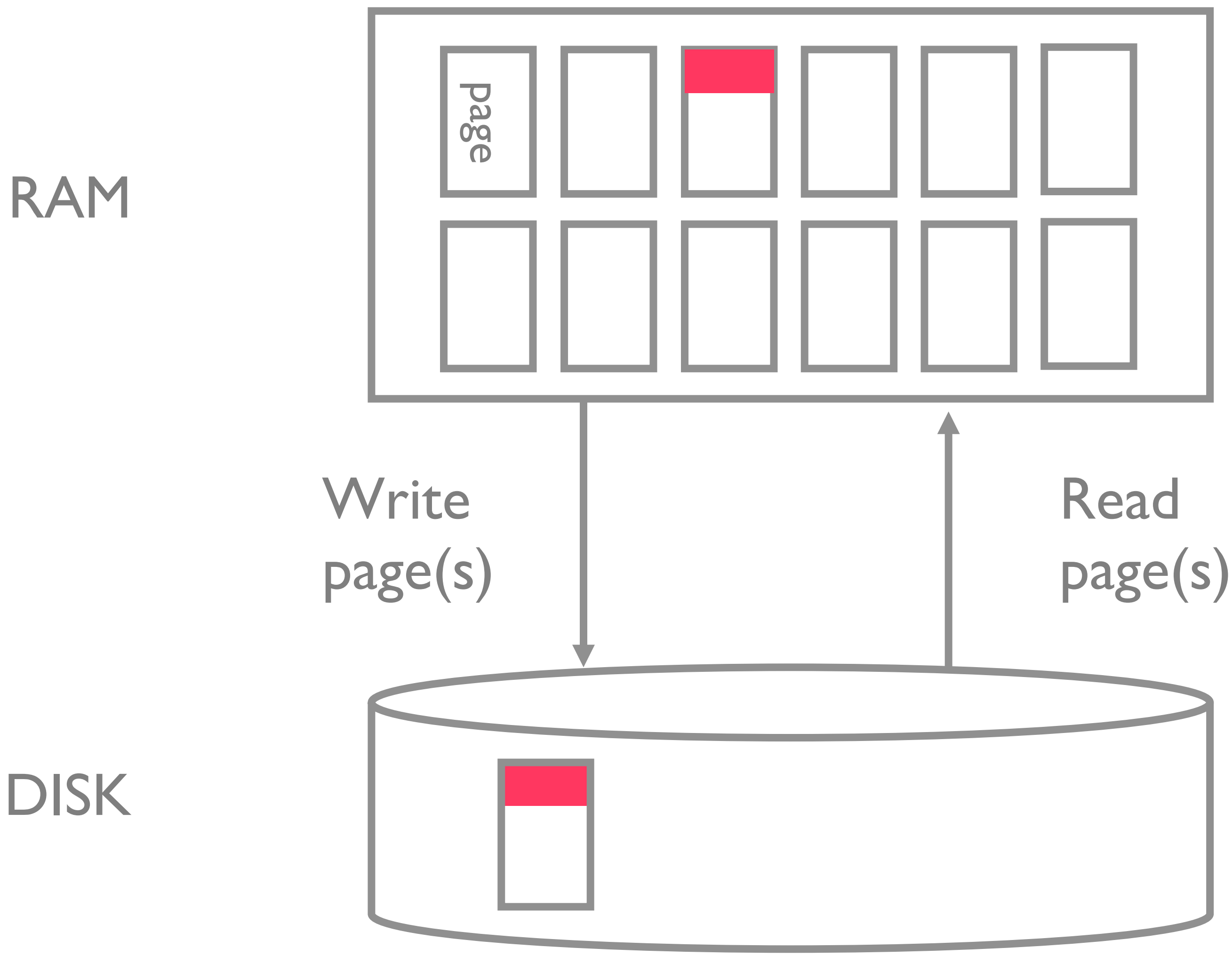
- Issues
  - WR: dirty reads
  - RW: unrepeatable reads
  - WW: lost writes
- Schedules
  - Equivalence
  - Serial
  - Serializable
- Serializability
  - Conflict serializability
  - how to detect
- Conflict Serializable Issues
  - Not recoverable
  - Cascading Rollback
- Strict 2 phase locking
- Deadlocks

Normal Execution

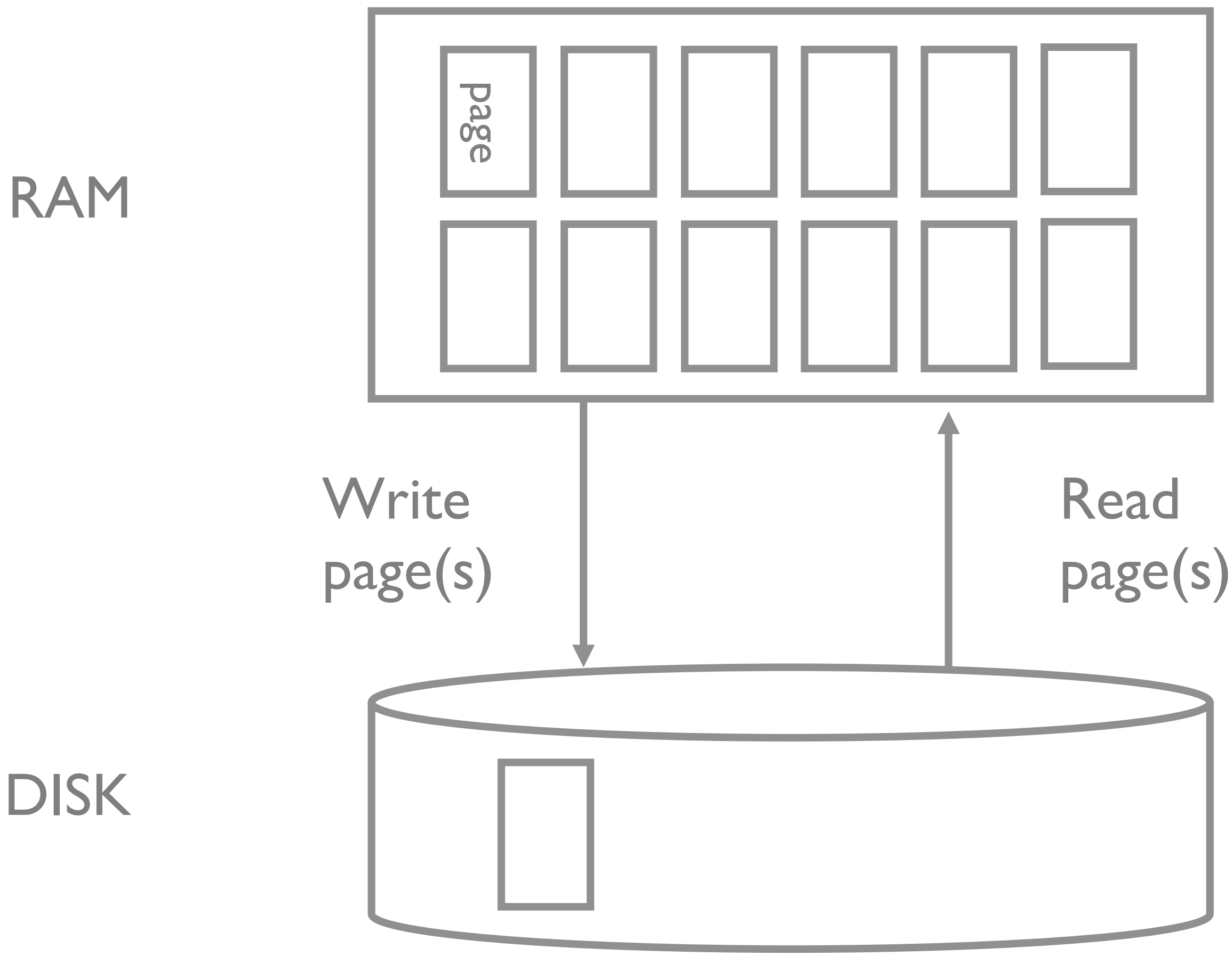
CRASH



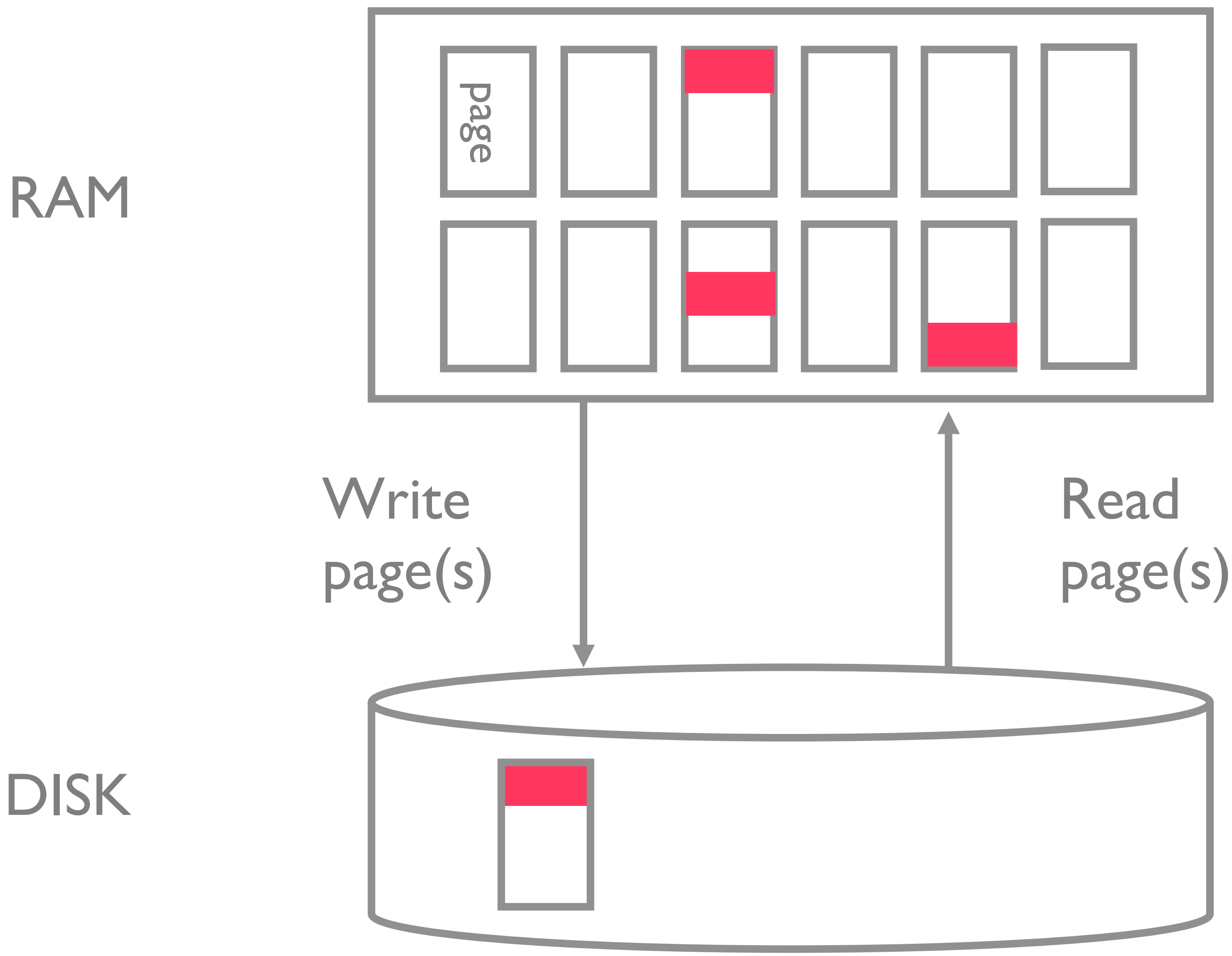
After a Crash



If DB did not say "OK, committed"



If T1 Committed and DB said "OK"



Recovery

Two properties: **Durability**, Atomicity

Assumptions for this class:
- Disk is safe. Memory is not.
- Running strict-2PL

Need to account for
- when pages are modified
- when pages are flushed to disk

Recovery: 2 problems

When could uncommitted ops appear after crash?

- wrote modified pages before commit

If T2 commits, what could make it not durable?

- didn’t write all changed pages to disk

Naïve solution?

Don’t write modified pages before commit!

(called “no steal” in text)

When transaction commits: write all modified pages!

(called “force” in text)

Solved problem?

Naïve solution problems

- Txn modifies 10 pages; crashes after writing 1
  - On recovery: observe partial results
- Txn modifies 10 pages then aborts
  - Re-load all modified pages from disk?
- T1 modifies page A; T2 modifies page A, T1 commits …
  - T1 must wait for T2 to complete (or lock page)
- Huge modification: Too big for memory!
  - Can't do it: need to give up transactions?
- Each transaction needs lots of random IO

Solution: The Log

The source of **truth**

Tables:

A view of "current" data in the log