L10
Transactions, Concurrency, Recovery

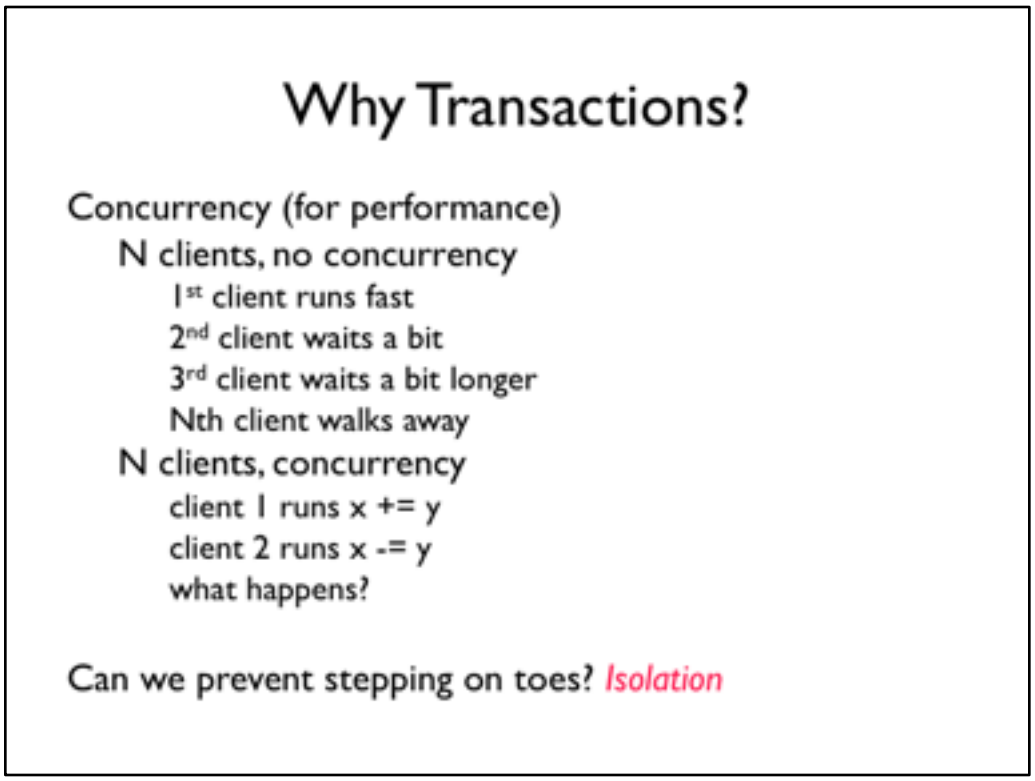
Eugene Wu

Fall 2015

Overview

Why do we want transactions?

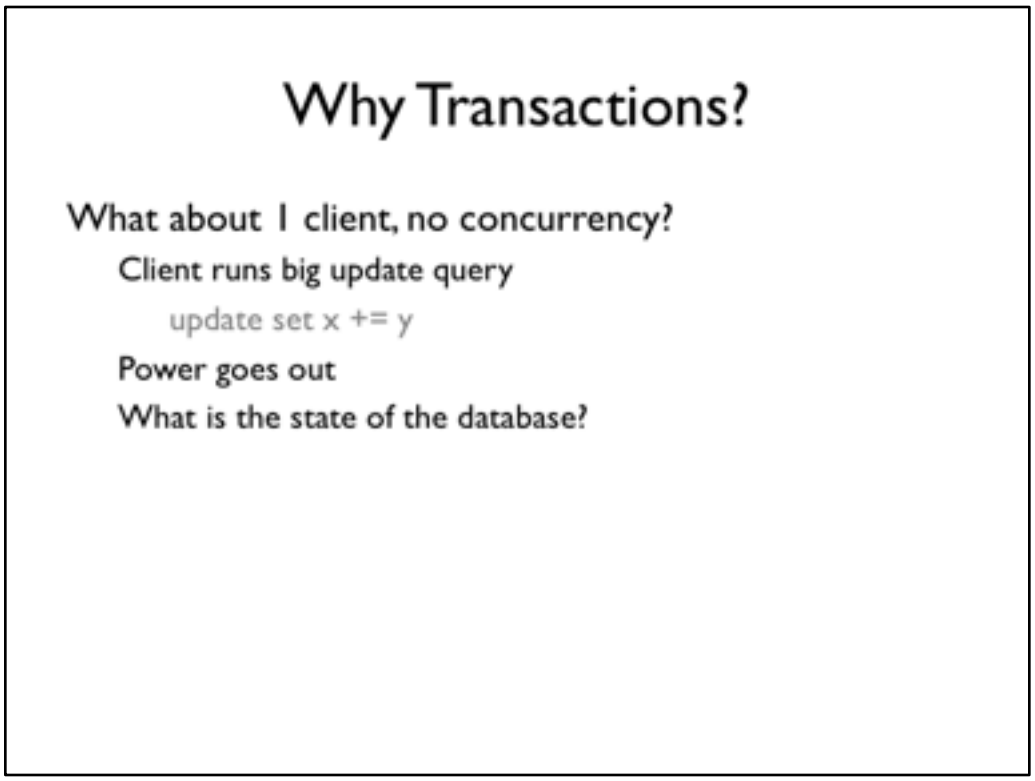
What guarantees do we want from transactions?



Tell story about lydia's storing entire chat room contents as a single json , with write skew multiple users read the json file, and write at the same time, so writes are lost.

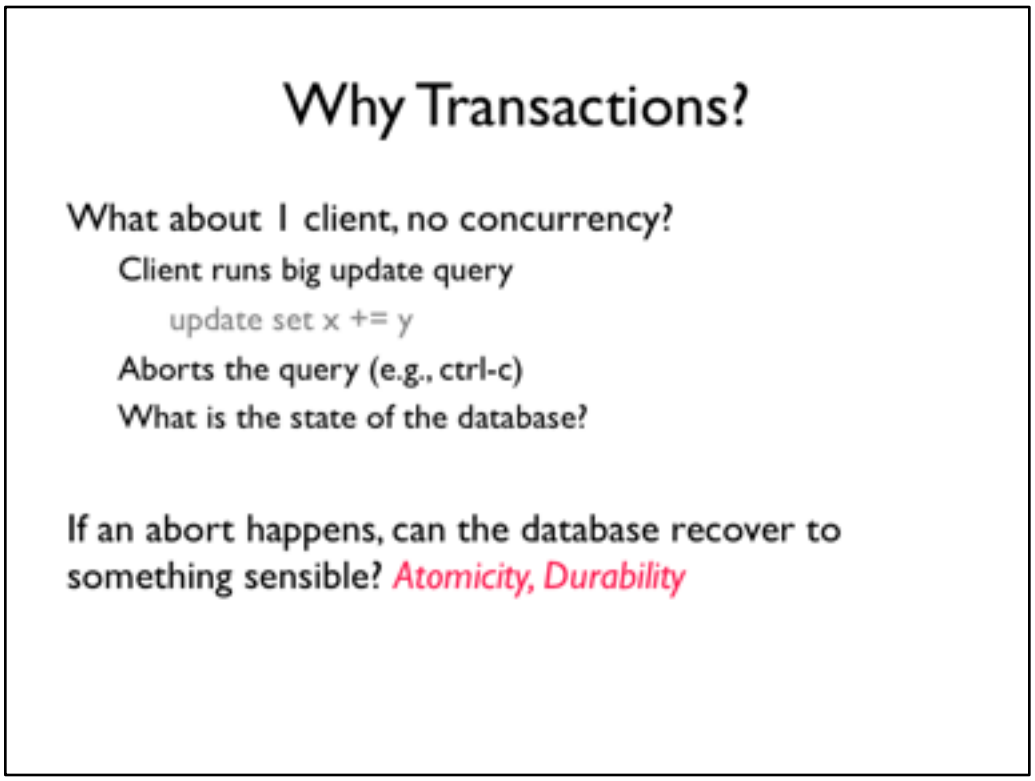
```
x += y
a1 = read(x)
b1 = read(y)
store(a1 + b1)
x -= y
a2 = read(x)
b2 = read(y)
store(a2 – b2)
```

```
x += y
a1 = read(x)
a2 = read(x)
b2 = read(y)
store(a2 – b2)
b1 = read(y)
store(a1 + b1)
```



durability: big update query finished but database doesn't contain all of it
atomicity: only half the updates finished, database is left in an in-between state

This seems pretty difficult



durability: big update query finished but database doesn't contain all of it
atomicity: only half the updates finished, database is left in an in-between state

This seems pretty difficult

Transactions

Transaction: a sequence of actions

action = read object, write object, commit, abort

API between app semantics and DBMS's view

User's view

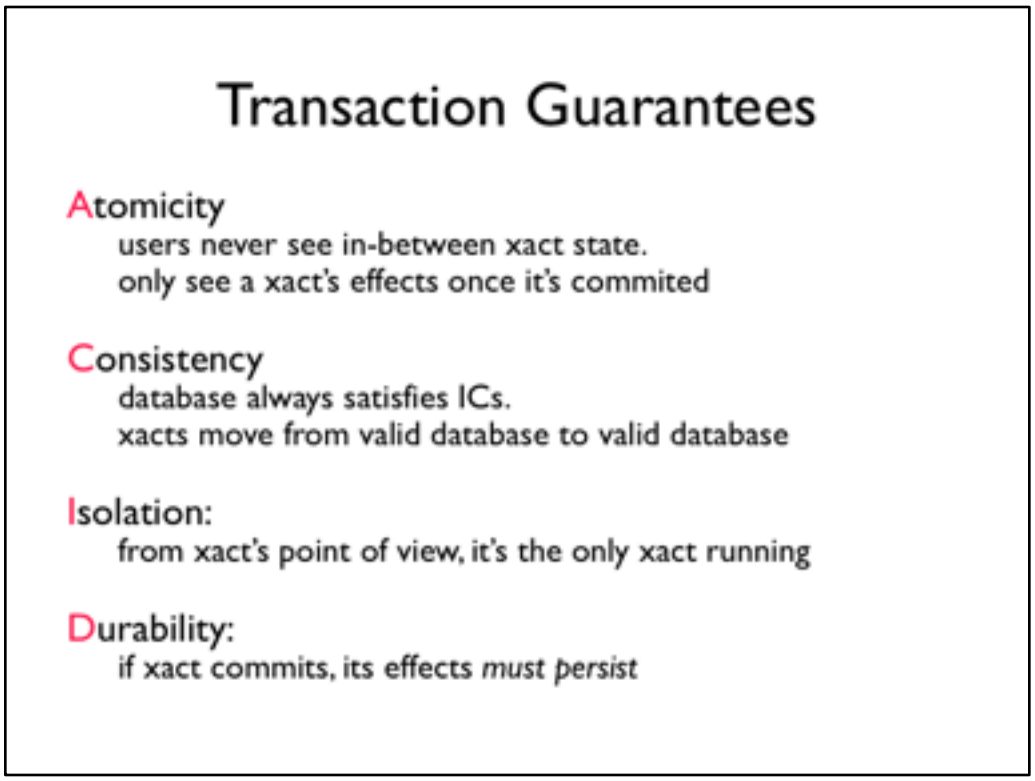
T1: begin	A=A+100	B=B-100	END
T2: begin	A=1.5*A	A=1.5*B	END

DBMS's logical view

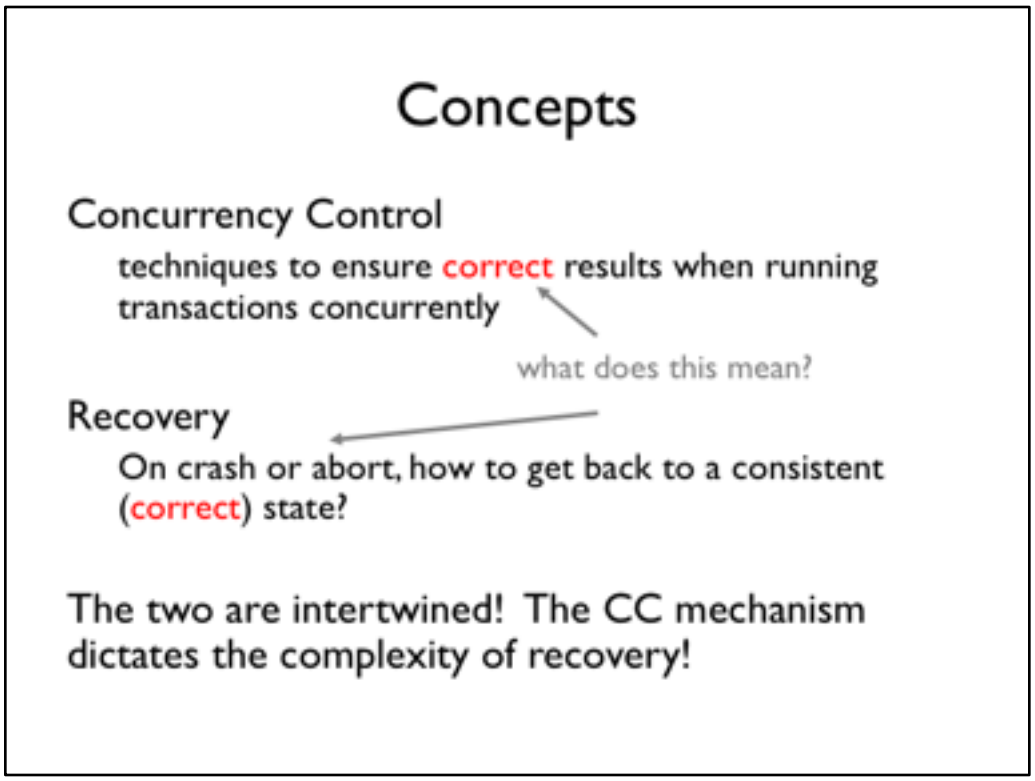
T1: begin	r(A) w(A)	r(B) w(B)	END
T2: begin	r(A) w(A)	r(B) w(A)	END

aborts can happen because

- user tells it to abort
- DBMS finds it causes some data anomalies
- xact hits some error (reads bad value, can't get more resources) and aborts itself



atomicity: go back to prev slide, say user never sees the database in the middle of xact1. Only at beginning if xact is still running, or after if it's committed
and by user, it also means the other xacts

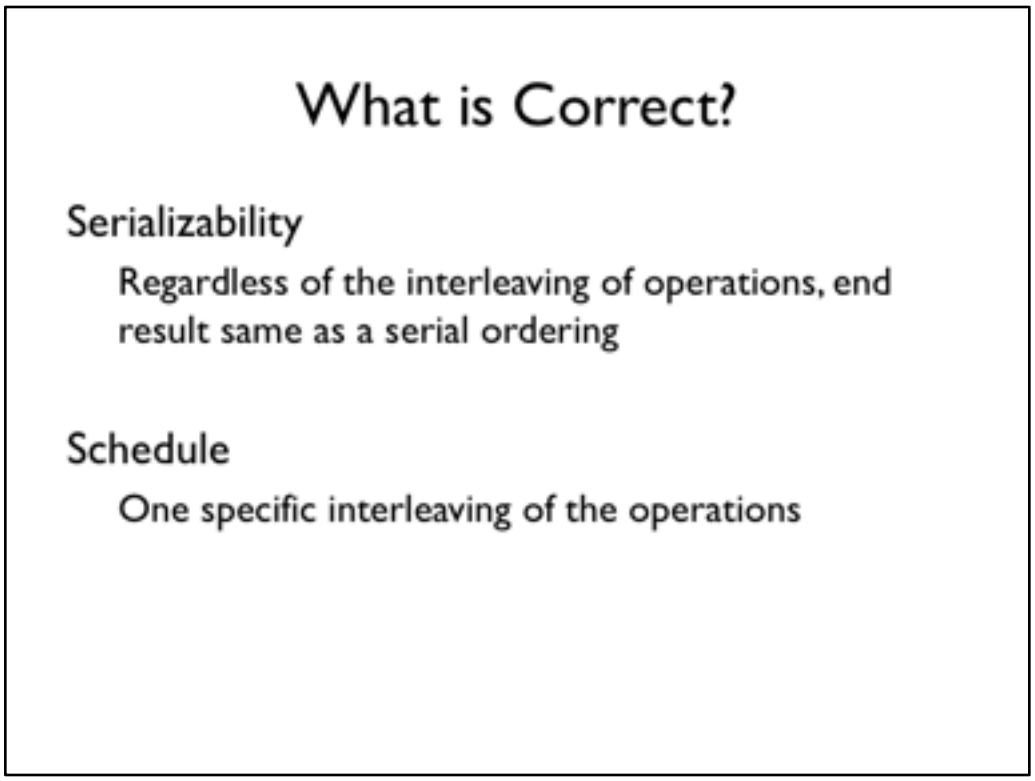


Because the type of concurrency control we use will have a dramatic effect on how recovery works.

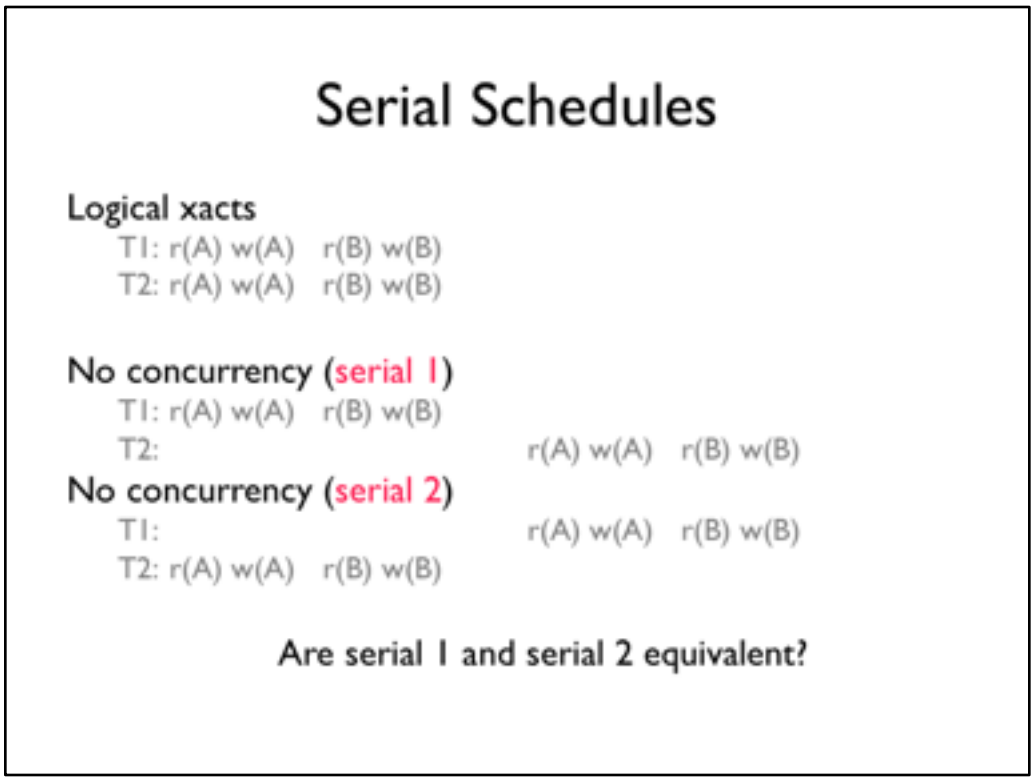
Suppose, for example, that we didn't have multi-query transactions, and each query executed serially?

How would that affect recovery?

It'd be a lot easier, because there'd be only one outstanding action at a time. First, we need some definitions.



this means we interleave the execution of transactions, but to have the end result be as though those concurrent actions had run in some serial order.



What does atomicity say? instantaneous – all or nothing right?
Under atomicity, is serial 1 allowed?
is serial 2 allowed?
are they both allowed?

any serial schedule is allowed.
xacts can be ordered in any way they are serially, DBMS doesn't care, but can't arbitrarily order their operations

Why is this a good abstraction? If transactions are instantaneous, no way of guaranteeing transaction order anyways.
Networks cause message reordering, time itself as we know is relative.
The act of communication causes many of these issues, and ordering communication is very expensive and often doesn't make sense.
If you and I both use the same ATM card to withdraw all money from an account, and you are in China and I'm in Bali, who knows which is allowed to go first? Does it matter?

GOLD STANDARD

There's a stronger form of concurrency we won't talk about called linearizability, which takes reality into account. It imagines, if there's a global observer that sees

everything (an out of band observer), the execution order is the same as the observer's.

More Example Schedules

Logical xacts

T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concurrency (bad)

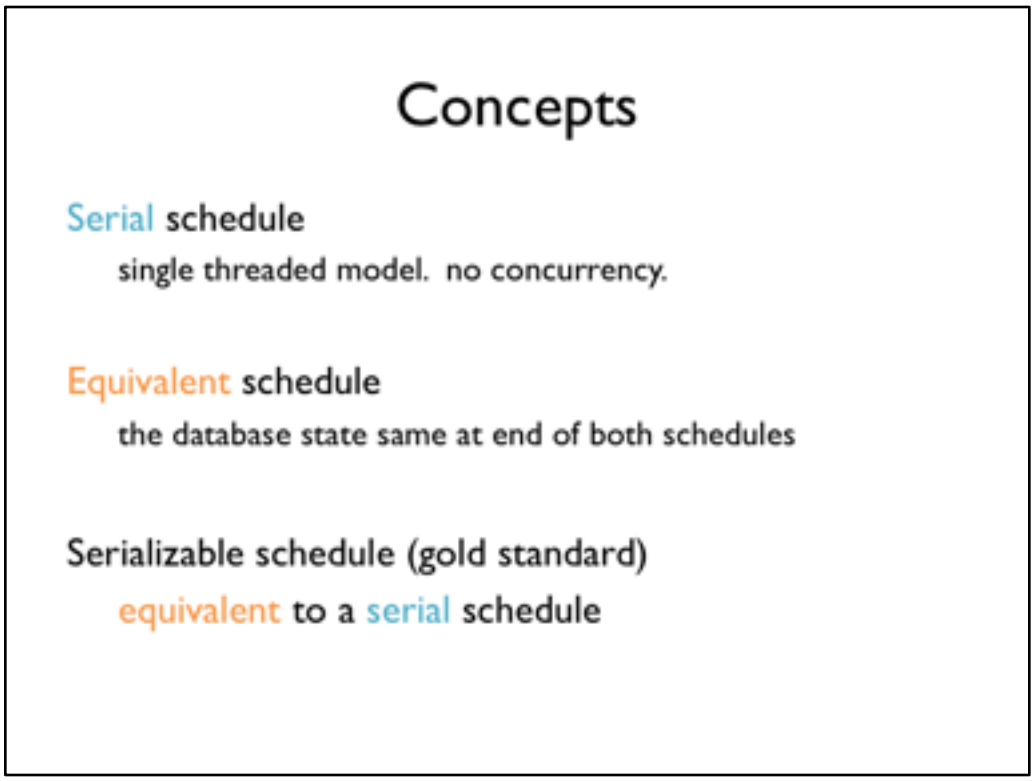
T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Concurrency (same as serial 1!)

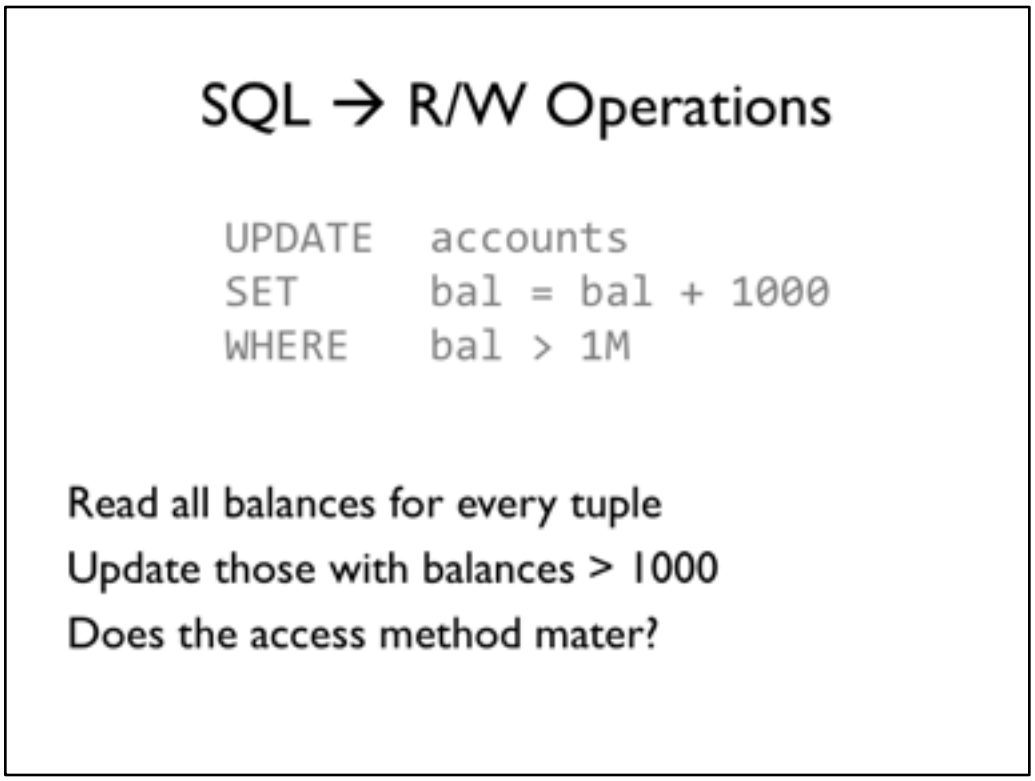
T1: r(A) w(A) r(A) w(B)
T2: r(A) w(A) r(B) w(B)

Why is the first concurrency example bad? Because T1 wrote a value of A (say 10), but when it reads the value again later, it could see a different value (what T2 wrote)
T1's w(A) was a lost write
T1's second read was a dirty read – it read an intermediate state in T2

What does "SAME" as serial 1 mean?
The end result.



There are different types of serializable schedules, and we will focus on one type called conflict serializable



Does the exact tuples we read depend on the access method?
If scan – read every tuple
If index, only tuples that match the predicate (plus the other tuples on the data pages, maybe)

Why is there this difference? Because our abstraction is individual read and write operations.

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

what are we trying to avoid?

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

We will define this notion called conflict serializability that is a way to determine if these anomalies do occur

R(A) W(A): first read 1 then write 2, if swapped, write 2 and read 2
Same with WR
W(A) W(A): Lost the first write. Affects future reads.

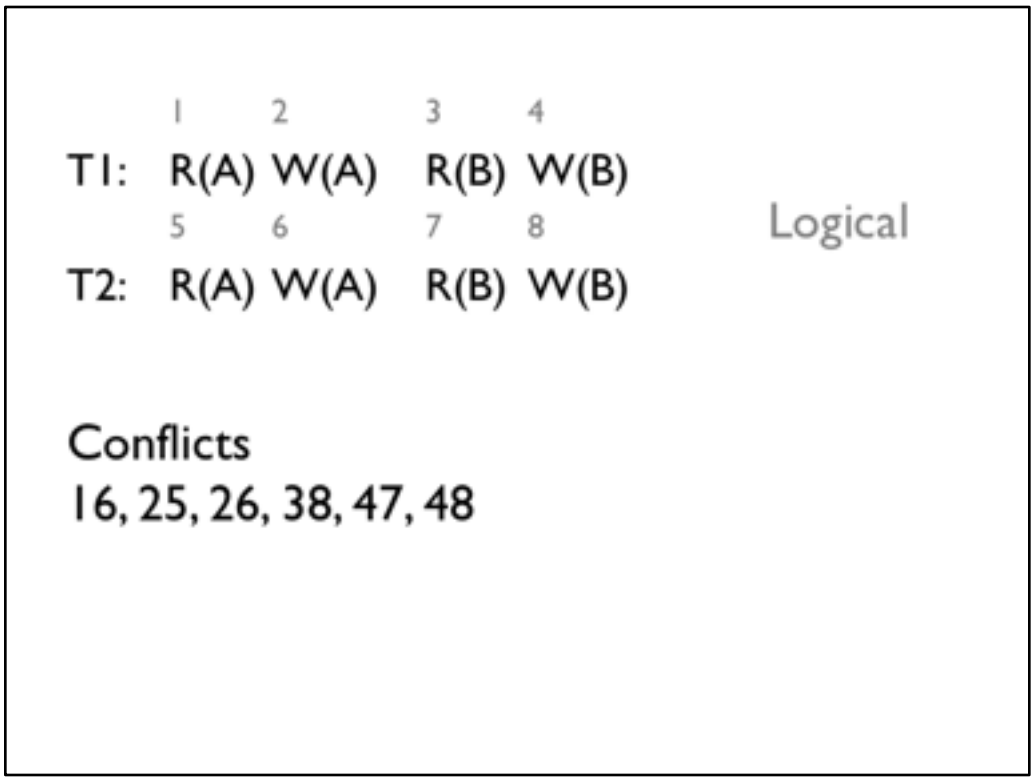
Conflict Serializability

def: possible to swap non-conflicting operations to derive a serial schedule.

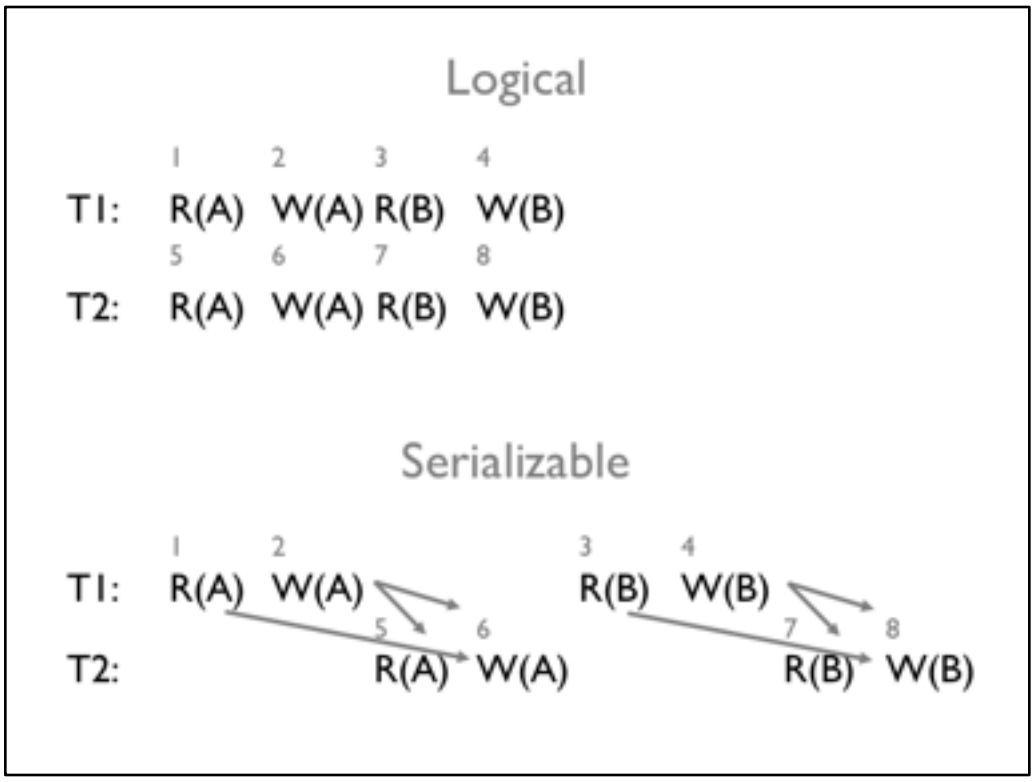
∀ conflicting operations O1 of T1, O2 of T2

O1 always before O2 in the schedule or

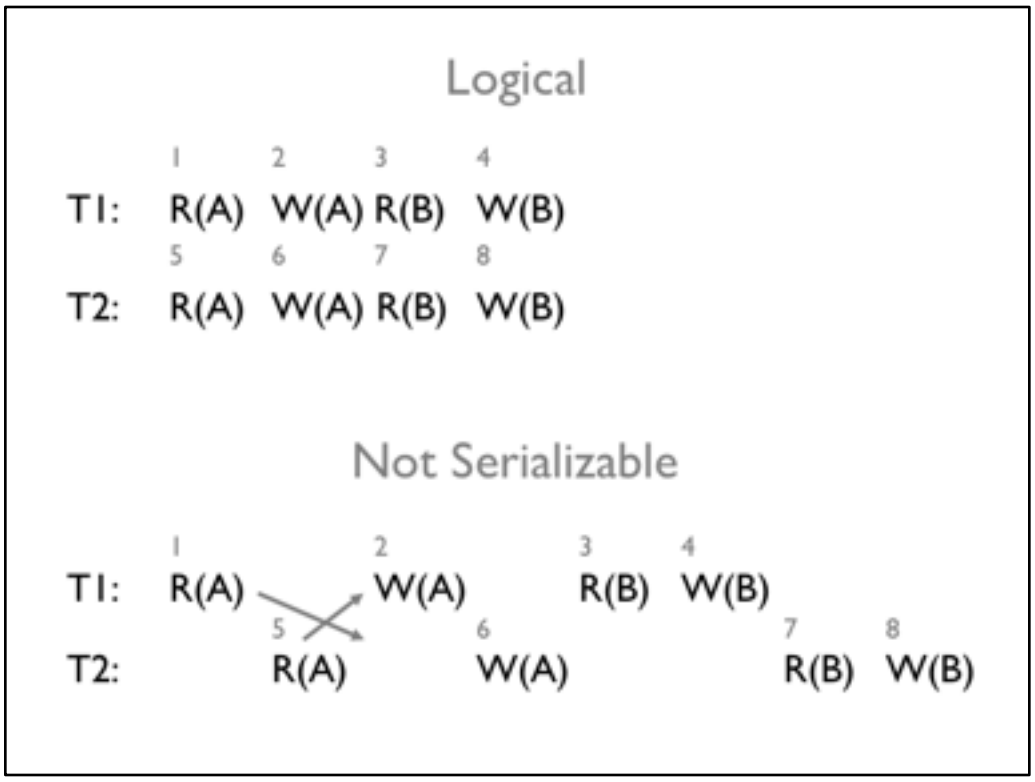
O2 always before O1 in the schedule



Consider some examples of serializable and non-serializable schedules



Why is this serializable?
Swap 36, 35 to get 1 2 3 5 6 4 7 8
Swap 46, 45 to get 1 2 3 4 5 6 7 8



we can't swap 2 with anything, since it's conflicting with 5 and 6!

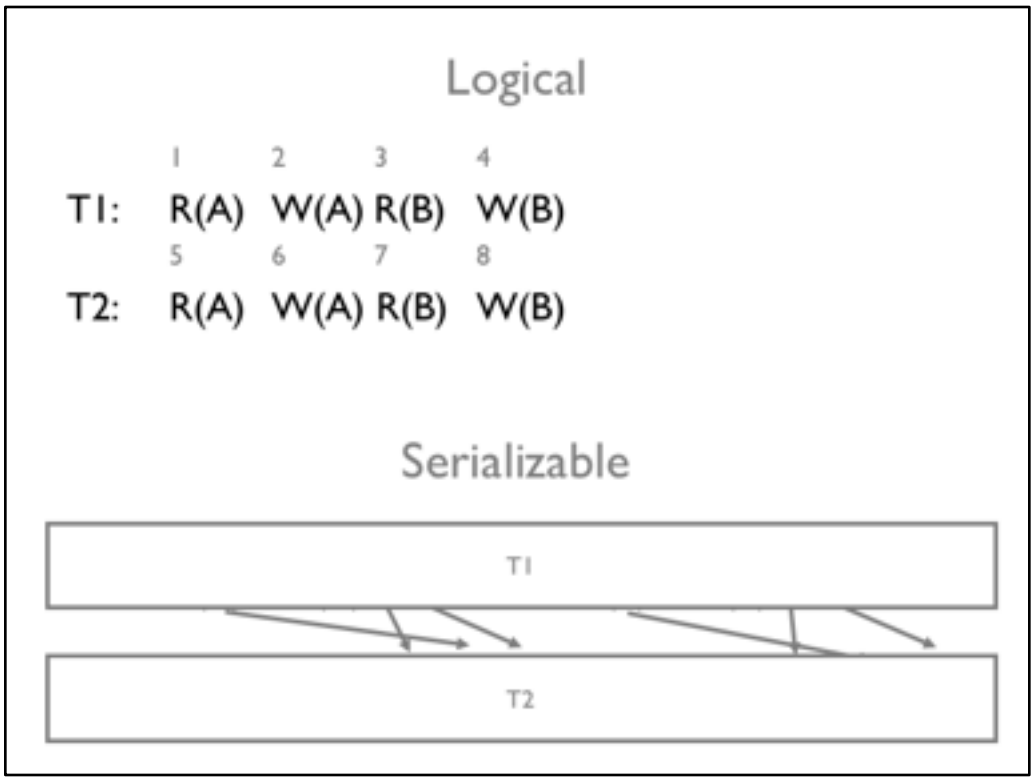
Conflict Serializability

Transaction Precedence Graph

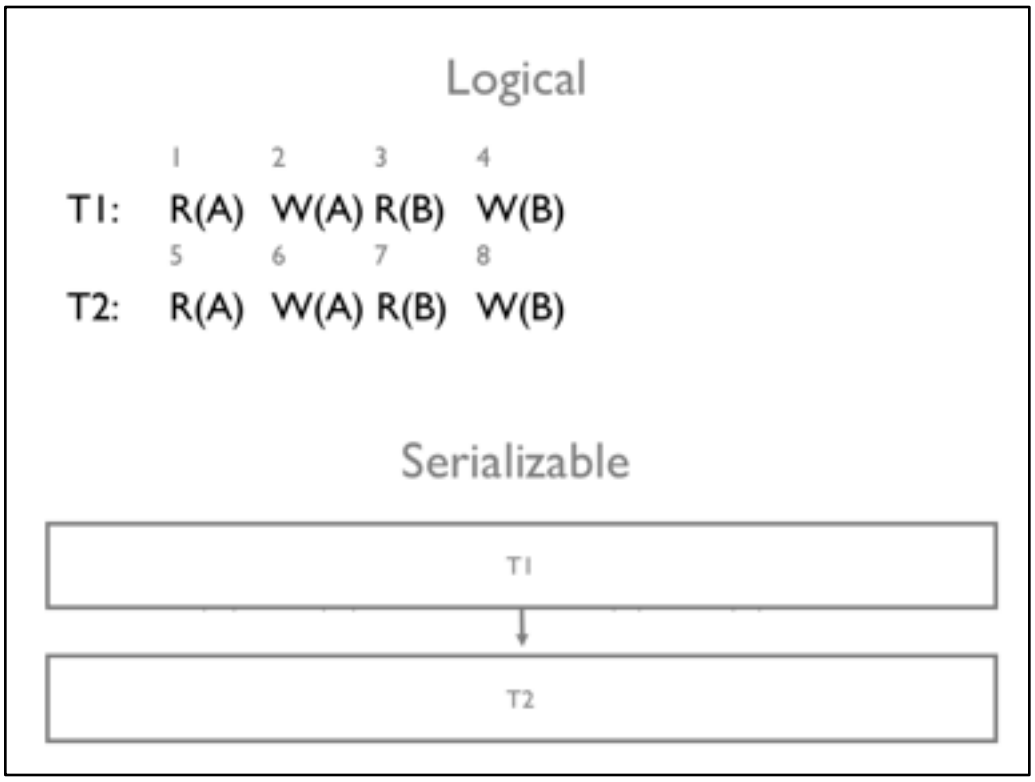
Edge Ti → Tj if:

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

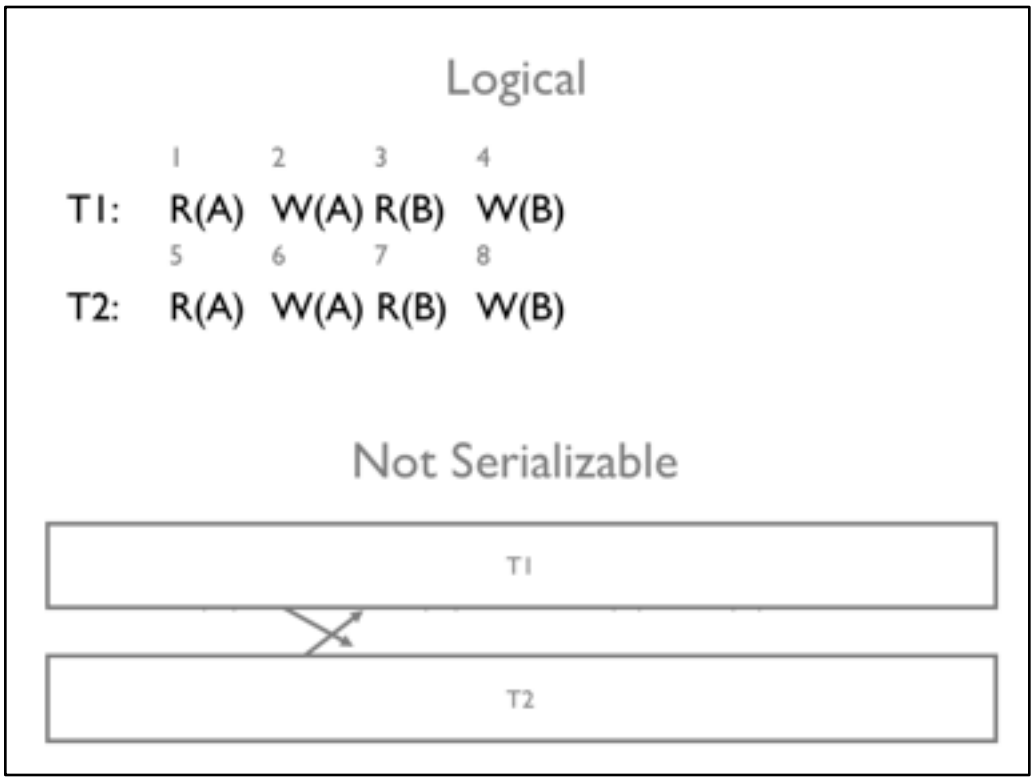
If graph is acyclic (does not contain cycles) then conflict serializable!



Why is this serializable?
Swap 36, 35 to get 1 2 3 5 6 4 7 8
Swap 46, 45 to get 1 2 3 4 5 6 7 8



Why is this serializable?
Swap 36, 35 to get 1 2 3 5 6 4 7 8
Swap 46, 45 to get 1 2 3 4 5 6 7 8



we can't swap 2 with anything, since it's conflicting with 5 and 6!

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

Must get a shared(read) or exclusive(write) lock BEFORE op

If other xact has lock, can get if lock table says so

YES

		T1	
	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	

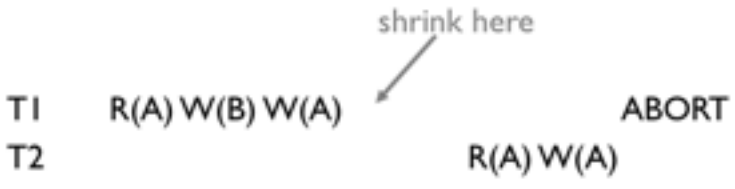
Is this schedule possible if we obey this locking criteria?
YES! We didn't say when we release locks! In this case, W(A) could get the lock, release it immediately, so T2 gets the shared lock. :(sadness

Lock-based Concurrency Control

Two-phase locking (2PL)

Growing phase: acquire locks

Shrinking phase: release locks



Uh Oh, same problem

How powerful is this!? What part of ACID does this guarantee?

A simple set of locking rules and ensure ISOLATION!! Super cool. This idea is used everywhere.
But doesn't ensuer atomicity and durability.

Lock-based Concurrency Control

Strict two-phase locking (Strict 2PL)

Growing phase: acquire locks

Shrinking phase: release locks

Hold onto locks until commit/abort

Why? Which problem does it prevent?

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

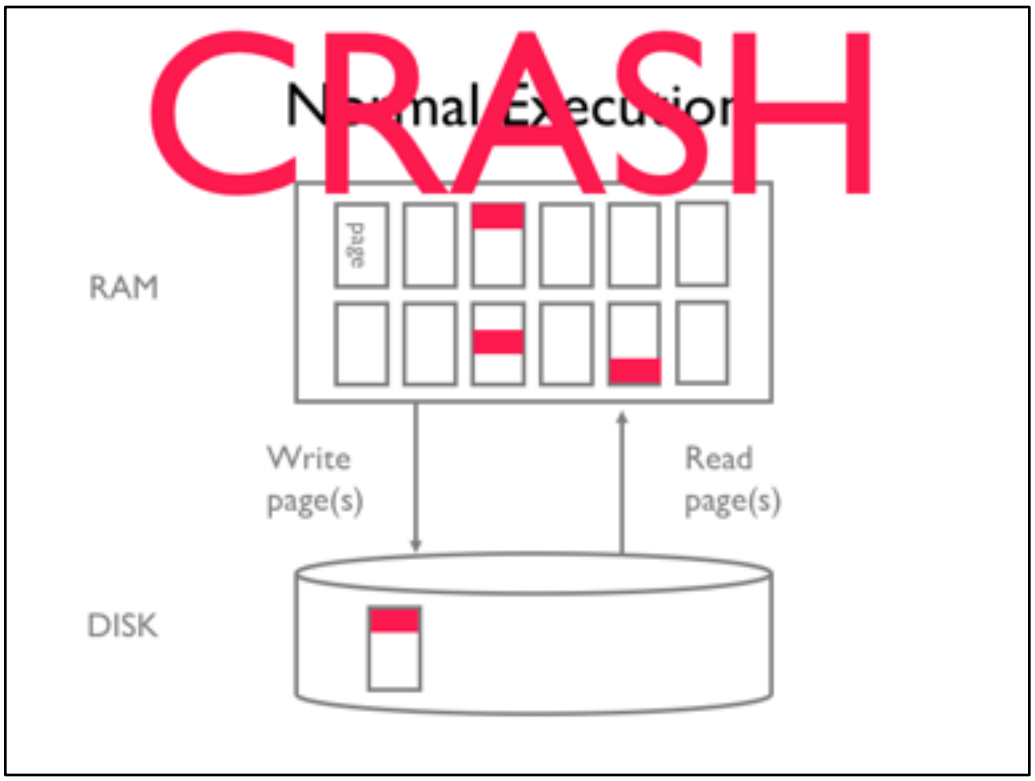
Guarantees serializable schedules! Avoids cascading rollbacks!

How powerful is this!? What part of ACID does this guarantee?

A simple set of locking rules and ensure ISOLATION!! Super cool. This idea is used everywhere.
But doesn't ensuer atomicity and durability.

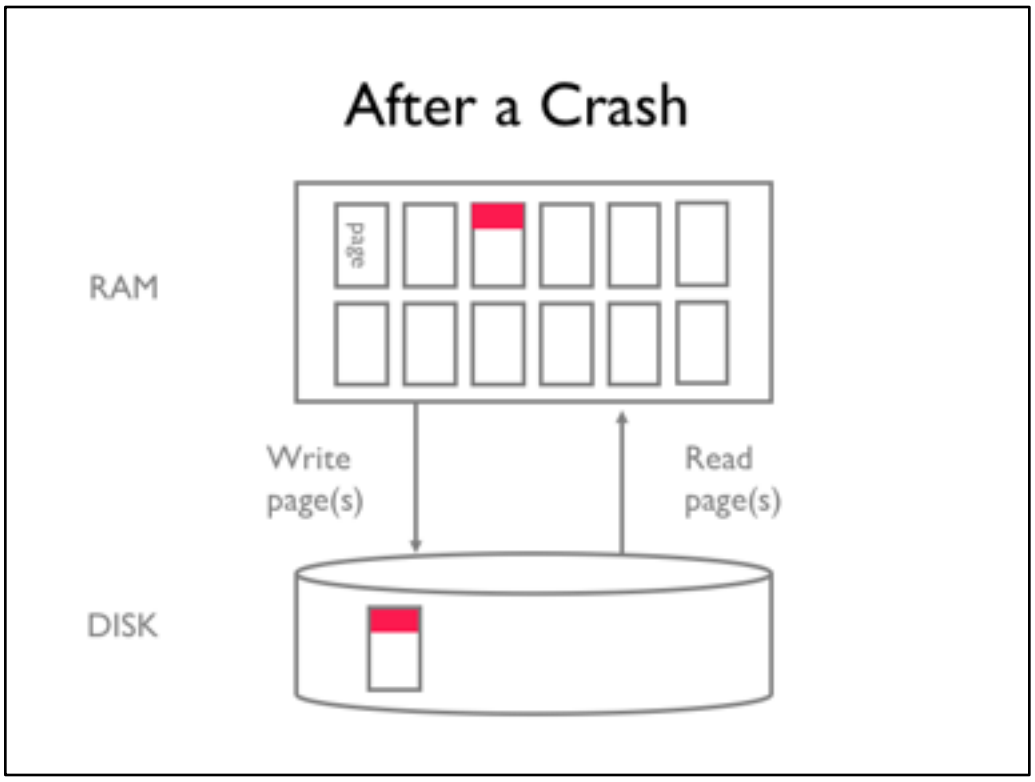
Review

- Issues
  - TR: 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

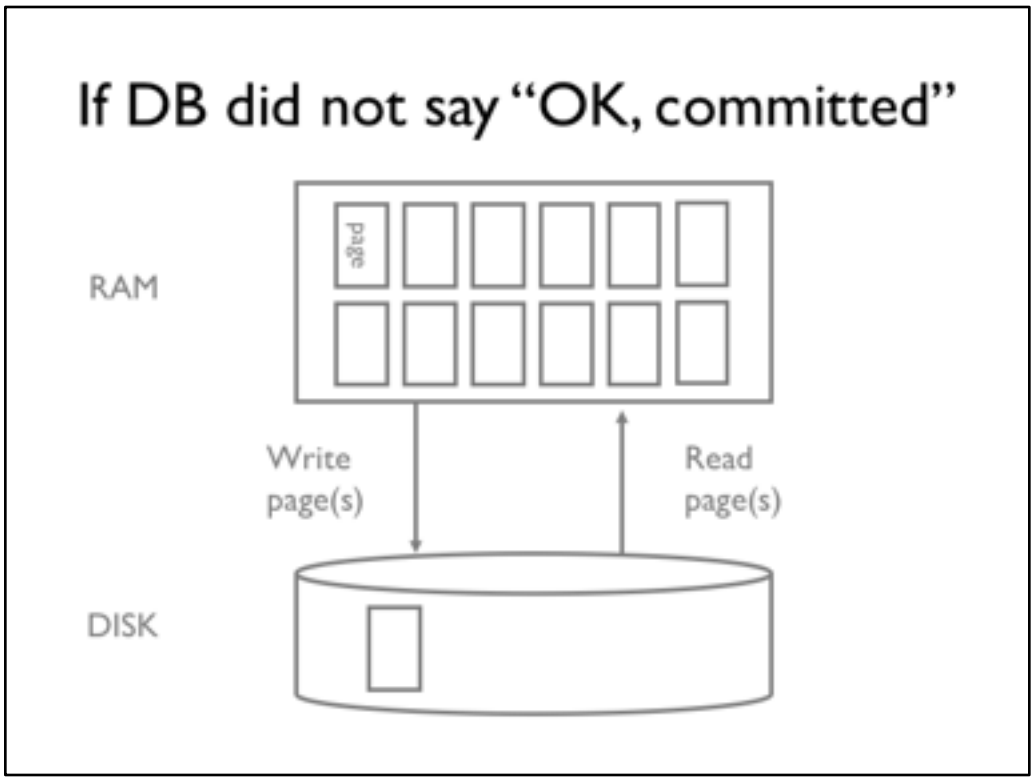


let's say transaction T1 is modifying pages, and each red bar is a set of modifications. Then T1 may modify two pages,
the database decides to write out one of them, then it modifies a third page, and commits.

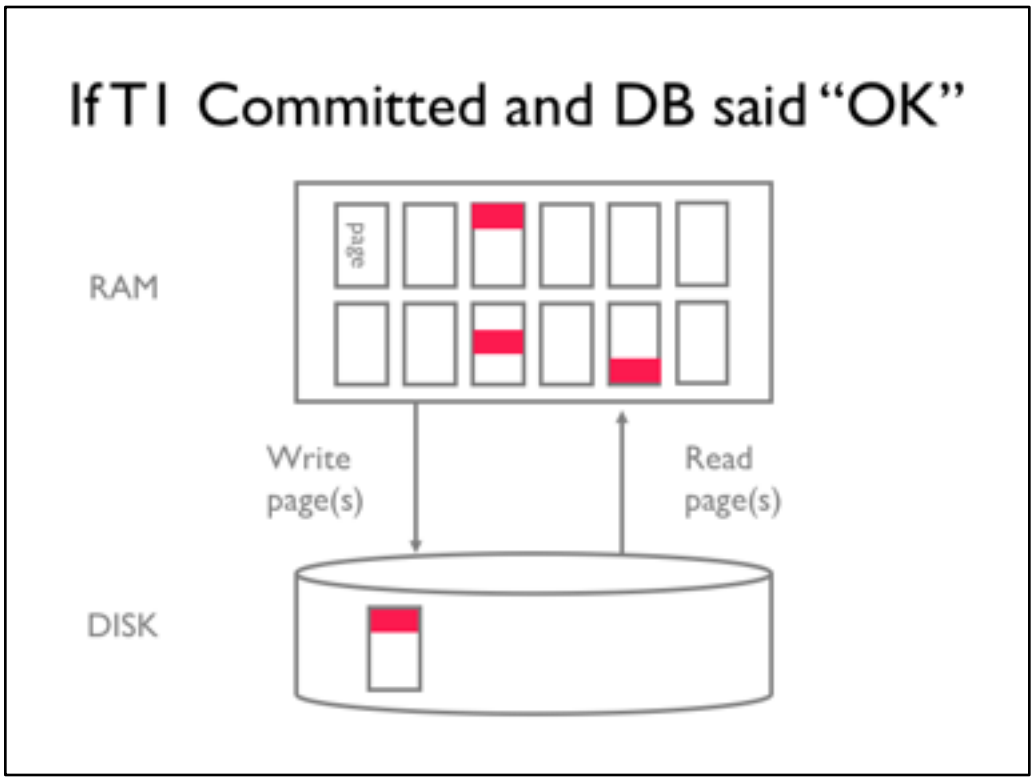
This state of the world is perfectly possible, because we have never discussed when the database needs to write pages to disk. So if we go backwards, a crash could happen at ANY POINT IN TIME



We had only written one of the modified pages out, so we read it from disk. Hmm, is this ok?



If T2 had not committed then all pages should be in their pre-T1 state



If T1 had committed, there should be three pages that are changed.
What's a problem? Given the data we have on disk, could we recover?
But the disk only has the one modified page!
Need some additional information.

Recovery

Two properties: Atomicity, Durability

Assumption in class
- Disk is safe. Memory is not.
- Running strict-2PL

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

Atomicity: how to undo aborted xacts
Durability: if I commit, it's there until the end of time (or when really bad things happen)

Why is this assumption important? if disk is faulty, then we need replication, or to understand enough about how things can go wrong e.g., what you are willing to accept, and design for that.
There's no _perfect_ recovery. It's only good enough recovery.
Many assume indpendent failures, at less than some rate.
Mountains and bombs

If might we miss durability?

Recovery

Deal with 2 cases

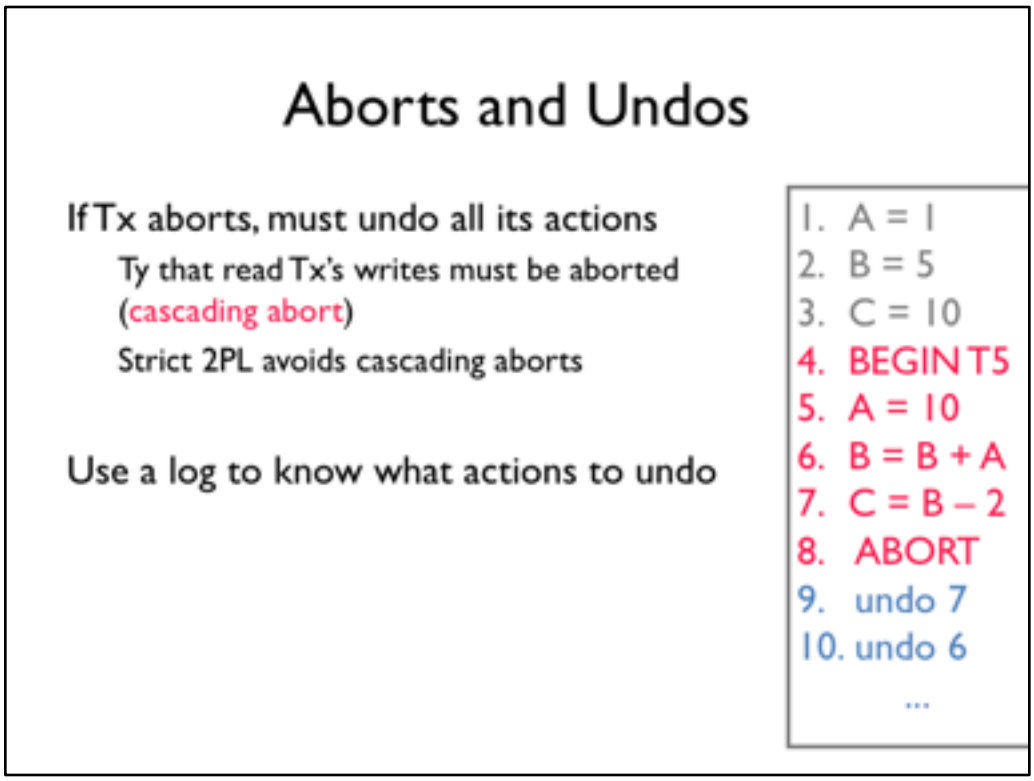
If T2 commits, what could make it not durable?
didn't write all changed pages to disk

When could uncommitted ops appear after crash?
wrote modified pages before commit

Atomicity: how to undo aborted xacts
Durability: if I commit, it's there until the end of time (or when really bad things happen)

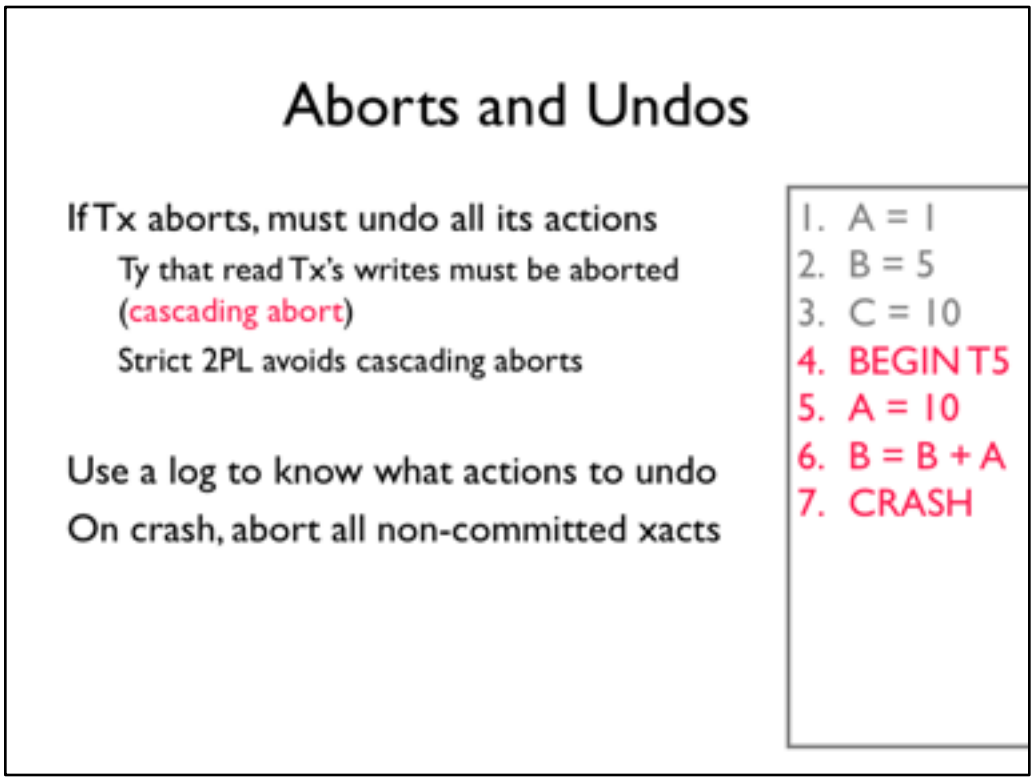
Why is this assumption important? if disk is faulty, then we need replication, or to understand enough about how things can go wrong e.g., what you are willing to accept, and design for that.
There's no _perfect_ recovery. It's only good enough recovery.
Many assume indpendent failures, at less than some rate.
Mountains and bombs

If might we miss durability?



Even avoiding cascading aborts, need to worry about undoing multiple ongoing transactions.

Let's say we have a database here
and T5 begins its transaction
it writes 10 to A….
Then it aborts, meaning the database should be as if T5 never even started. What do we do?
undo C
undo B
undo A



Let's say we have a database here
and T5 begins its transaction
it writes 10 to A….
Then it aborts, meaning the database should be as if T5 never even started. What do we do?
undo C
undo B
undo A

Logs

Log is the *ground truth*

Log records

- writes: old & new value
- commit/abort actions
- xact id & xact's previous log record

Persist log records (write to disk) *before* data pages persisted

Is this enough?

physical representation of the log holds the same data as our database – physical independence?

Durability

Bad scenario

- T1 writes to A
- T1 commits, log record written to disk
- start writing page with A to disk
- *crash*

Can undo help us?

Need to redo T1, otherwise no durability!

Logs

Log is the *ground truth*

Log records

- writes: old & new value
- commit/abort actions
- xact id & xact's previous log record

Write ahead logging (WAL)

1. Persist log records (write to disk) *before* data pages persisted
2. Persist all log records *before* commit

(1) guarantees UNDO info
(2) guarantees REDO info

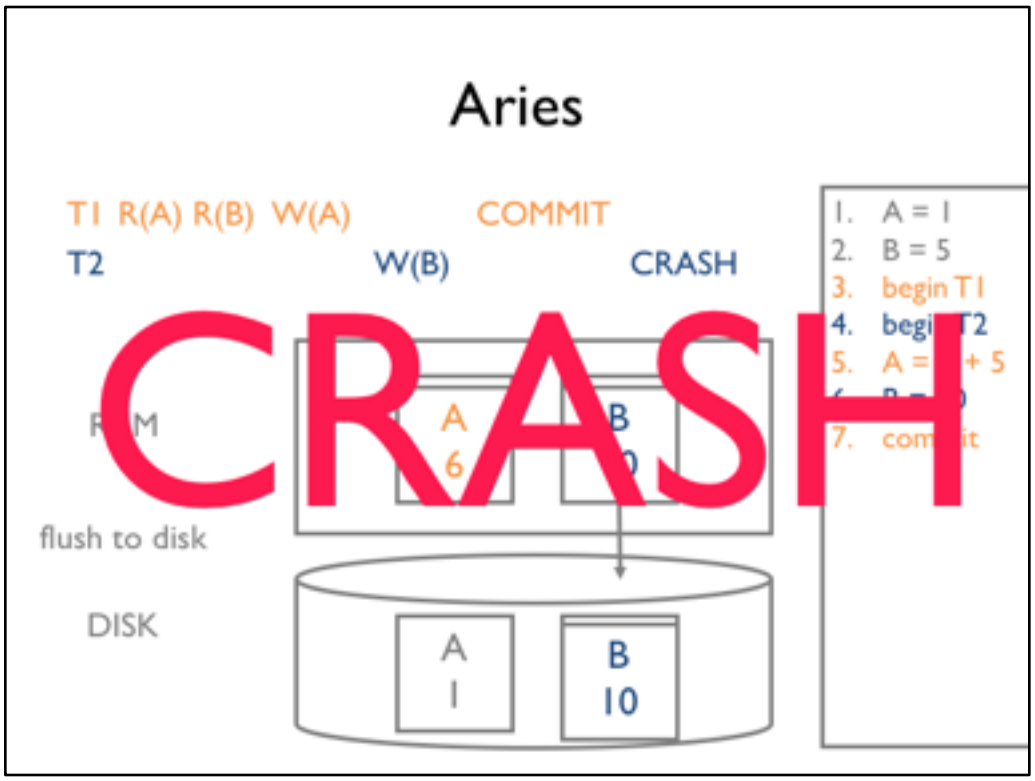
physical representation of the log holds the same data as our database – physical independence?

Aries Recovery Algorithm

3 phases

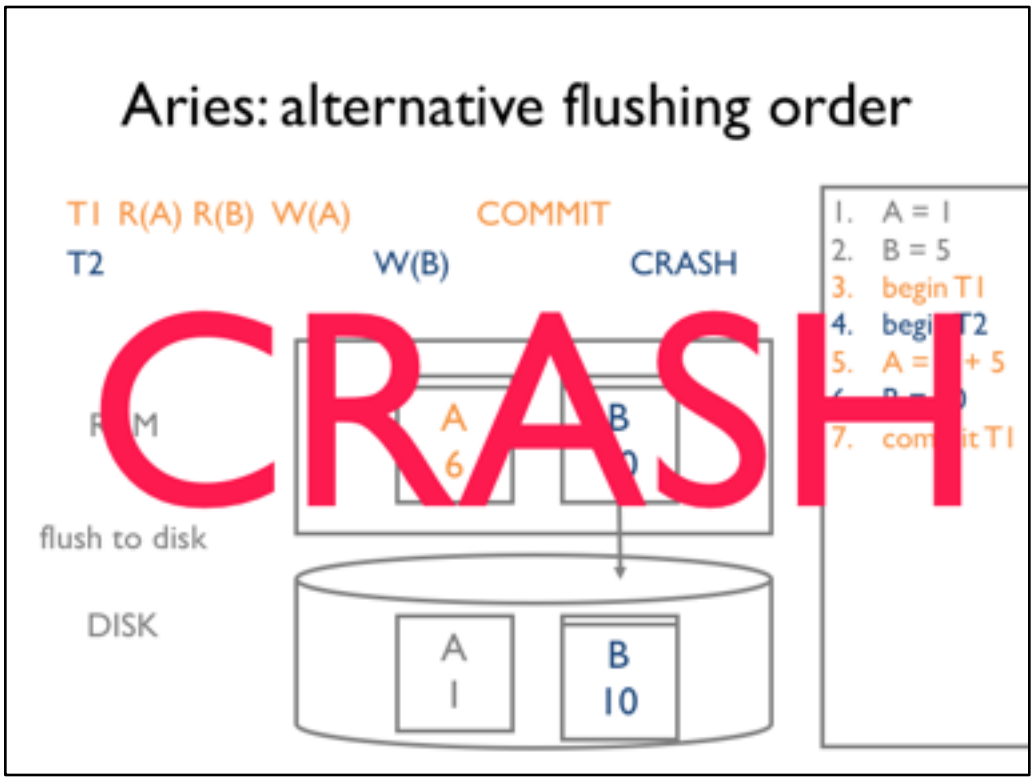
- Analyze the log to find status of all xacts
  - Commited or in flight?
- Redo xacts that were committed
- Undo partial (in flight) xacts

Recovery is *extremely* tricky and *must be correct*



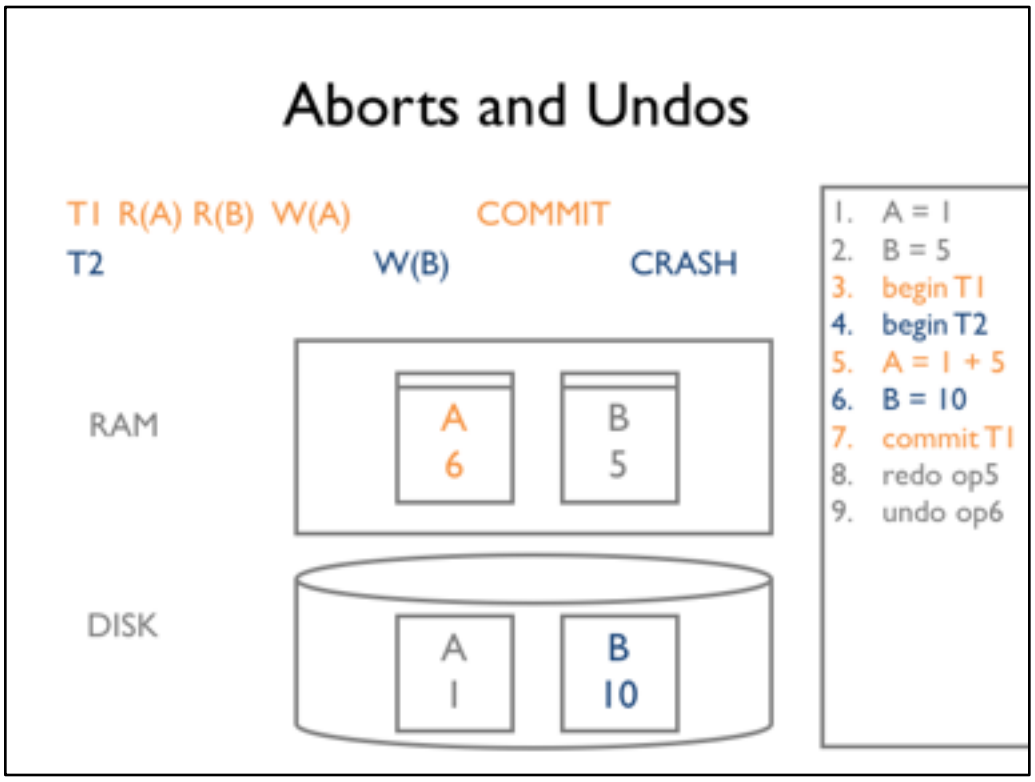
Even avoiding cascading aborts, need to worry about undoing multiple ongoing transactions.

Let's say we have a database here
and T5 begins its transaction
it writes 10 to A….
Then it aborts, meaning the database should be as if T5 never even started. What do we do?
undo C
undo B
undo A



Even avoiding cascading aborts, need to worry about undoing multiple ongoing transactions.

Let's say we have a database here
and T5 begins its transaction
it writes 10 to A….
Then it aborts, meaning the database should be as if T5 never even started. What do we do?
undo C
undo B
undo A



What if you crash DURING this recovery? need to make sure even that is handled

Summary

Recovery depends on what failures are tolerable

Buffer pool can write RAM pages to disk any time

Recovery Manager ensures durability and atomicity via redo and undo

You should know

What transactions/schedules/serializable are

Can identify conflict serializable schedules

Can identify schedule anomalies

Can identify strict 2PL executions

Understand WAL and what it provides

Given an executed schedule, run the proper sequence of undo/redos