

# L10

## Transactions, Concurrency, Recovery

Eugene Wu  
Fall 2015

Remember in the history of SQL, the CODASYL folks were making the performance argument, that there's no way that SQL could run fast? Well history has proved them wrong, and we'll talk enough to give you a feel for how we went about making this fast

## Overview

Why do we want transactions?

What guarantees do we want from transactions?

# Why Transactions?

Concurrency (for performance)

N clients, no concurrency

1<sup>st</sup> client runs fast

2<sup>nd</sup> client waits a bit

3<sup>rd</sup> client waits a bit longer

Nth client walks away

N clients, concurrency

client 1 runs  $x += y$

client 2 runs  $x -= y$

what happens?

Can we prevent stepping on toes? *Isolation*

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)
```

# Why Transactions?

What about 1 client, no concurrency?

Client runs big update query

update set  $x += y$

Power goes out

What is the state of the database?

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

# Why Transactions?

What about 1 client, no concurrency?

Client runs big update query

update set x += y

Aborts the query (e.g., ctrl-c)

What is the state of the database?

If an abort happens, can the database recover to something sensible? *Atomicity, Durability*

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

# Transaction Guarantees

## Atomicity

users never see in-between xact state.  
only see a xact's effects once it's committed

## Consistency

database always satisfies ICs.  
xacts move from valid database to valid database

## Isolation:

from xact's point of view, it's the only xact running

## Durability:

if xact commits, its effects *must persist*

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



# Concepts

## Concurrency Control

techniques to ensure **correct** results when running transactions concurrently

what does this mean?

## Recovery

On crash or abort, how to get back to a consistent (**correct**) state?

The two are intertwined! The CC mechanism dictates the complexity of recovery!

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.

# What is Correct?

## Serializability

Regardless of the interleaving of operations, end result same as a serial ordering

## Schedule

One specific interleaving of the operations

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.

# Serial Schedules

## Logical xacts

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

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

## No concurrency (**serial 1**)

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

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

## No concurrency (**serial 2**)

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

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

Are serial 1 and serial 2 equivalent?

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 I!)

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.

# Concepts

## Serial schedule

single threaded model. no concurrency.

## Equivalent schedule

the database state same at end of both schedules

## Serializable schedule (gold standard)

equivalent to a serial schedule

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

## 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 matter?

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

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

Conflicts

16, 25, 26, 38, 47, 48

Consider some examples of serializable and non-serializable schedules

	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)

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

Swap 46, 45 to get 1 2 3 4 5 6 7 8

### 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

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

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

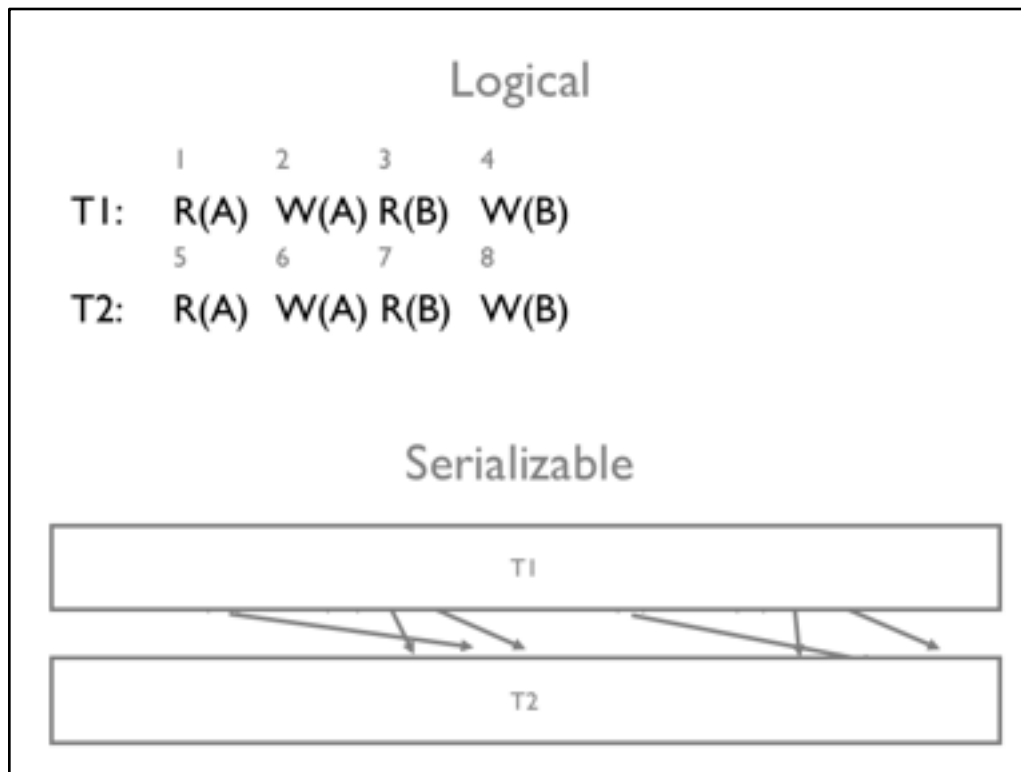
## Conflict Serializability

### Transaction Precedence Graph

Edge  $T_i \rightarrow T_j$  if:

1.  $T_i$  read/write A before  $T_j$  writes A or
2.  $T_i$  writes some A before  $T_j$  reads A

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

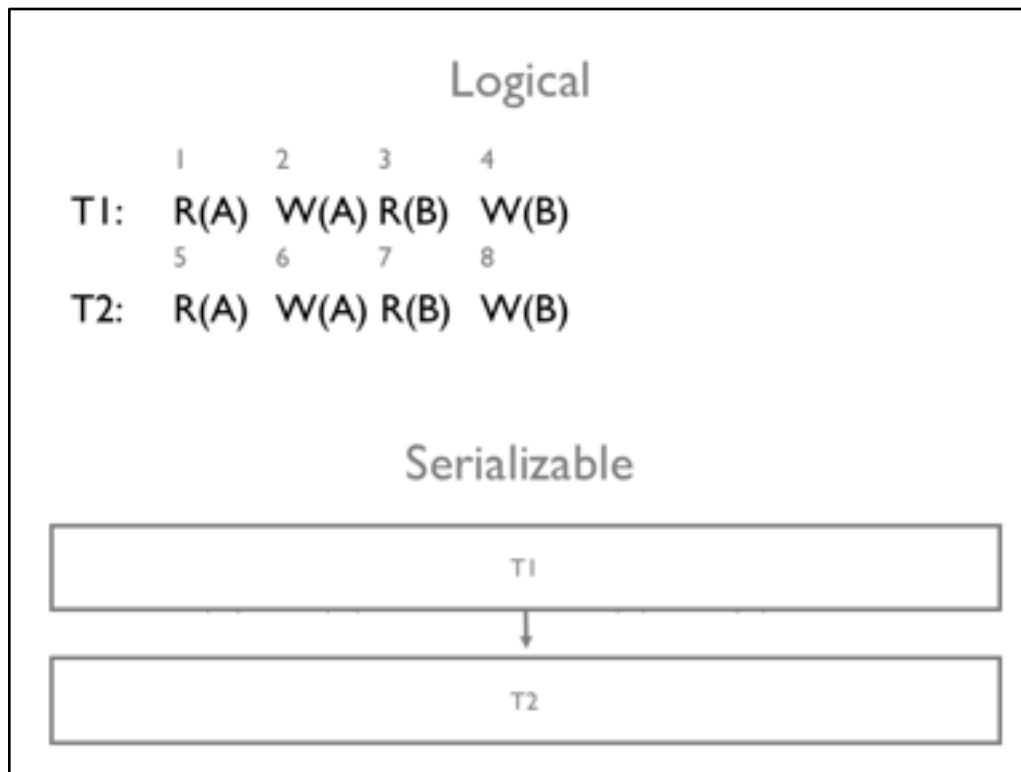


Why is this serializable?

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

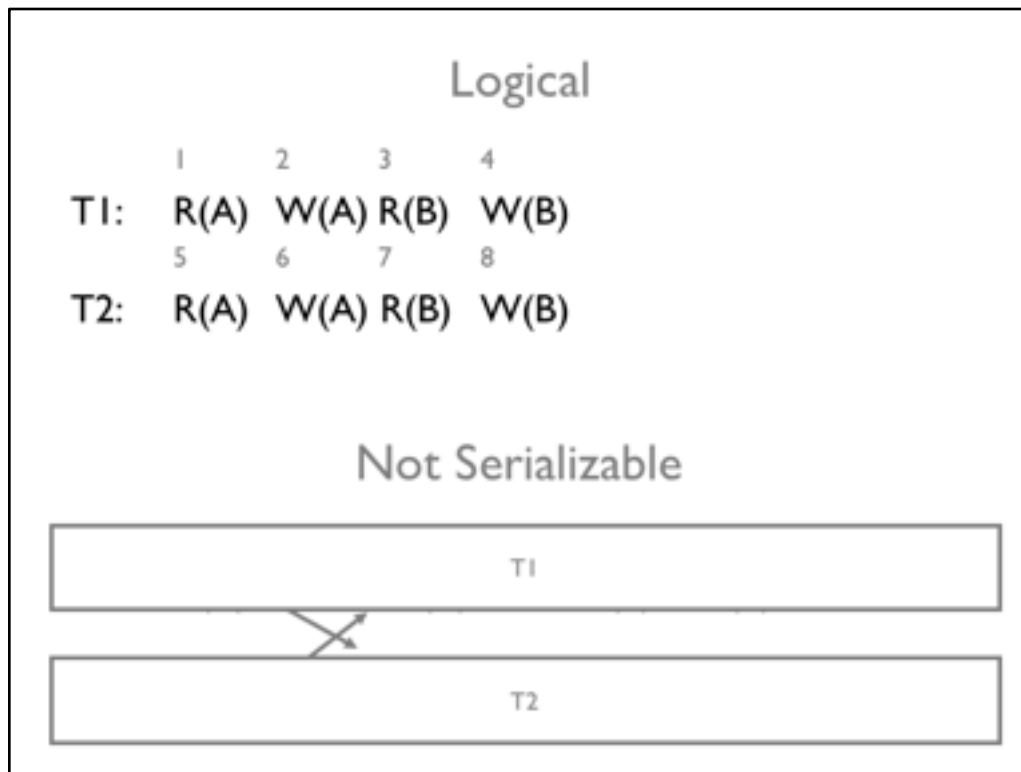




Why is this serializable?

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 undoes 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?

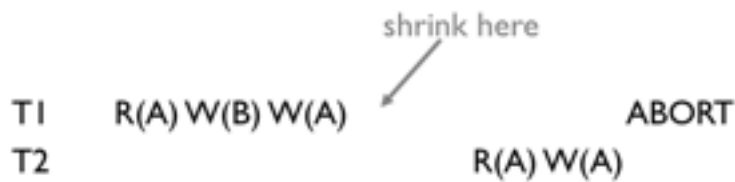
YES! We didn't say when we release locks! In this case, W(A) could get the lock, release it immediately, so T2 gets the shared lock. :( sadness

# 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 ensure 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 ensure atomicity and durability.

# Review

## Issues

TR: dirty reads  
RW: unrepeatable reads  
WW: lost writes

## Schedules

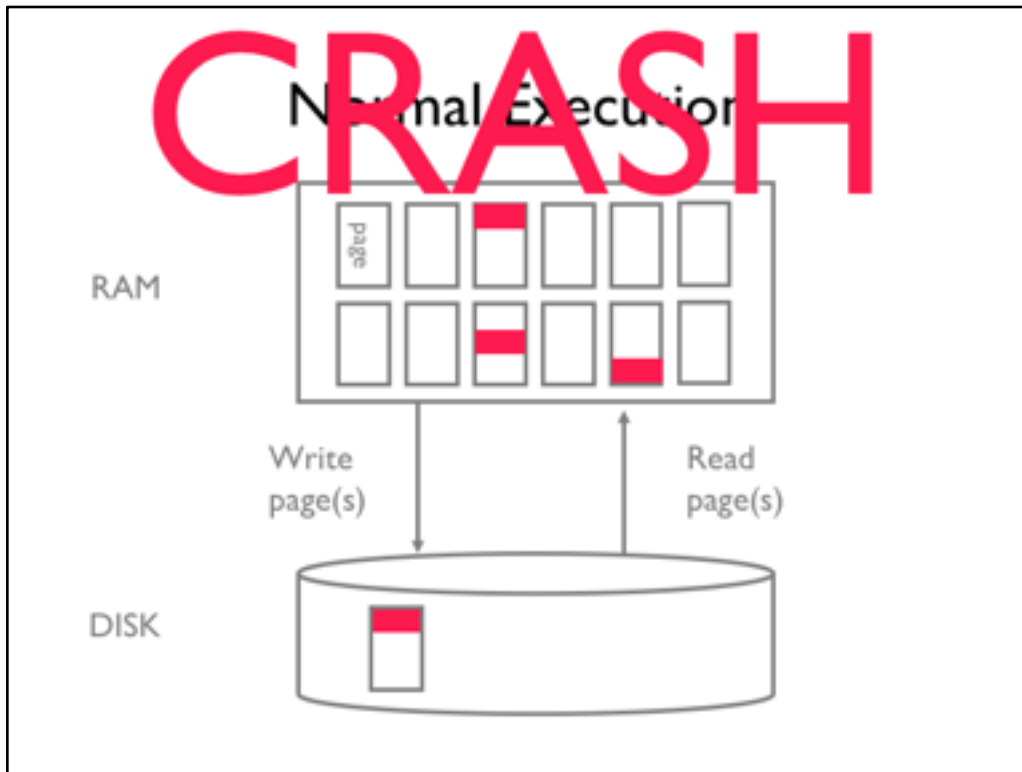
Equivalence  
Serial  
Serializable

## Serializability

Conflict serializability  
how to detect

## Conflict Serializable Issues

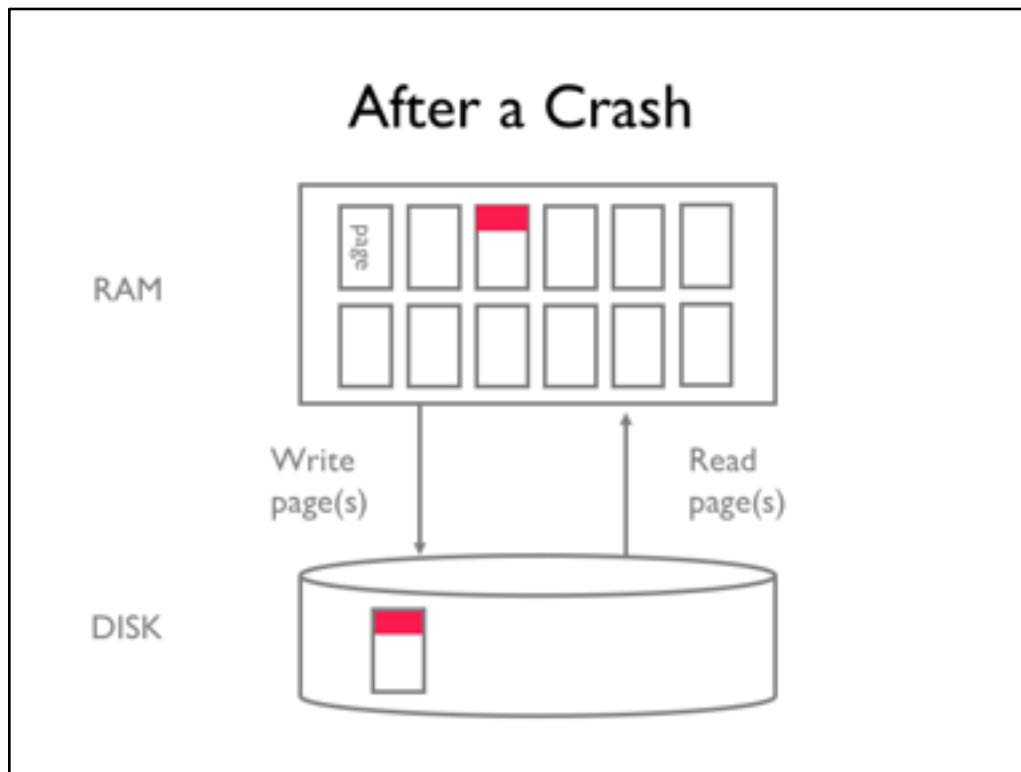
Not recoverable  
Cascading Rollback  
Strict 2 phase locking



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.

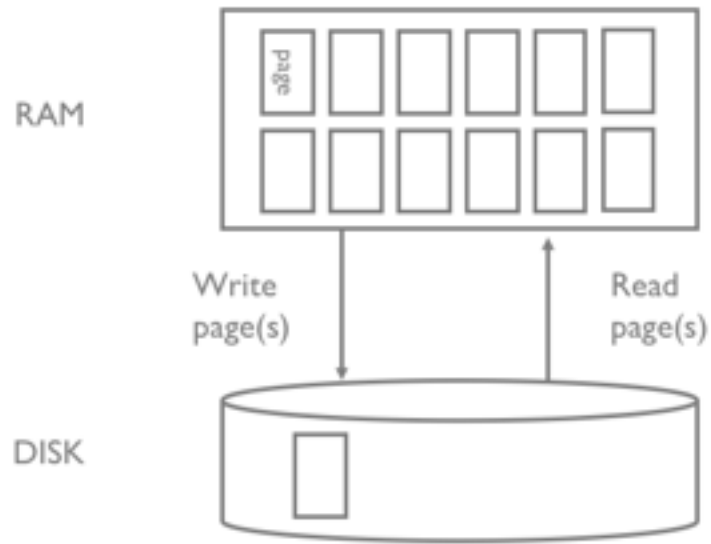
This state of the world is perfectly possible, because we have never discussed when the database needs to write pages to disk. So if we go backwards, a crash could happen at ANY POINT IN TIME





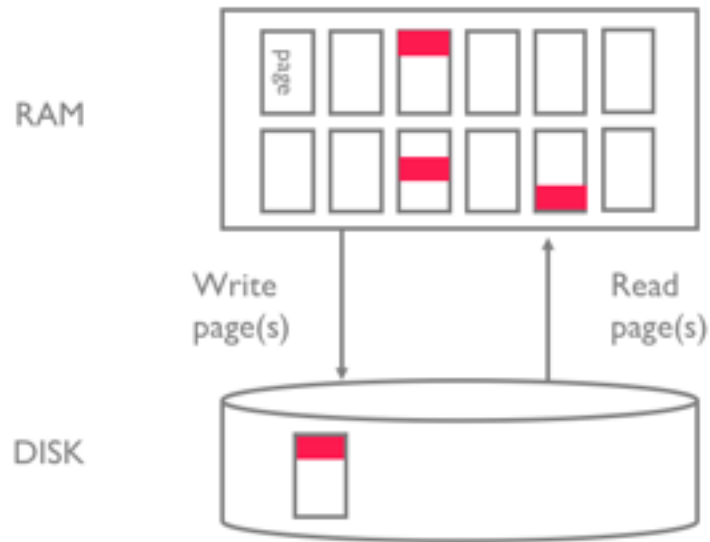
We had only written one of the modified pages out, so we read it from disk. Hmm, is this ok?

If DB did not say “OK, committed”



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

## If T1 Committed and DB said “OK”



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.

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 independent 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 independent failures, at less than some rate.

Mountains and bombs

If might we miss durability?

## Aborts and Undos

If Tx aborts, all of its actions must be undone.

Ty that read Tx's writes must be aborted  
(cascading abort)

Strict 2PL avoids cascading aborts

Use a log to know what actions to undo

```
1. A = 1
2. B = 5
3. C = 10
4. BEGIN T5
5. A = 10
6. B = B + A
7. C = B - 2
8. ABORT
9. undo 7
10. undo 6
...
```

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

## Aborts and Undos

If Tx aborts, all of its actions must be undone.

Ty that read Tx's writes must be aborted  
(cascading abort)

Strict 2PL avoids cascading aborts

Use a log to know what actions to undo

On crash, abort all non-committed xacts

1.  $A = 1$
2.  $B = 5$
3.  $C = 10$
4. BEGIN T5
5.  $A = 10$
6.  $B = B + A$
7. CRASH

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 records

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

## Write ahead logging (WAL)

- log records stored on disk (persisted) *before* data pages can be persisted
- log is the *ground truth*

So far: use log to undo partial transactions



# 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!

## Aries Recovery Algorithm

3 phases

1. Analyze the log to find status of all xacts
2. Redo xacts that were committed
3. Undo partial xacts

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