Recovery

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

Concept: Sequence of changes

Before any change: write it to a sequential file
(write ahead of the change: write-ahead log)

On commit/abort: write "commit/abort" record

On recovery: replay complete transactions

Required: ordered; on recovery: no "holes"

Log with multiple changes

T1 modifies pages A, Z, T2 modifies A, C

1. T1: A: i=5
2. T2: A: j=2
3. T1: Z: k=3
4. T1: commit
5. T2: C: l=9
6. T2: commit

Crash: Truncated log

1. T1: A: i=5
2. T2: A: j=2
3. T1: Z: k=3
4. T1: commit

Result: apply T1 actions, don't apply T2 actions

Aside: how do we ensure “no holes”?

E.g. Can’t recover if log contains:

1. T1: A: i=5
2. ???
3. T1: commit

Build complex system on simpler parts

Aside: Durability abstractions

Transaction: All or nothing, any random pages

Log: On crash, always have a complete prefix

How do we implement the log?

Disk: Write a page

- What happens if we crash in the middle?
- Old page, new page, corrupted page

Aside: Durability abstractions

Disk: Flush pages

When complete:

1. Read: returns new value, survives power
2. Writes to other pages: will not change

Solution: careful ordering of writes, flushes;

Don’t overwrite pages

Aside: How do disks implement flush?

Old disks:

Wait for the write to hardware to complete

New disks (flash, shingled recording):

Firmware/software reorders writes

Aside: Durability abstractions

Transaction: All or nothing, random pages

Log: On crash, always have a complete prefix

Disk: After flush, writes will survive

Shingled disk: Re-write adjacent data

??? Physics: No idea how this works

Redo-only log

Rule: To write a modified page to disk:

- All writers must have committed

  (no uncommitted changes that might be aborted)

- Log record is on disk

On abort: Need to undo changes

Keep "undo" information in memory

When do disk writes happen?

On commit: Write log to disk, wait to complete (sequential IO)

After commit: Write modified pages when needed, or when idle?

“free” the log? Complicated; not covered

Disadvantage: Big transactions

What about transactions bigger than memory?

Need to write uncommitted changes to disk

Solution: ARIES algorithm (IBM again; 1992)

Idea: Write both undo and redo records

Can write page if undo records on disk

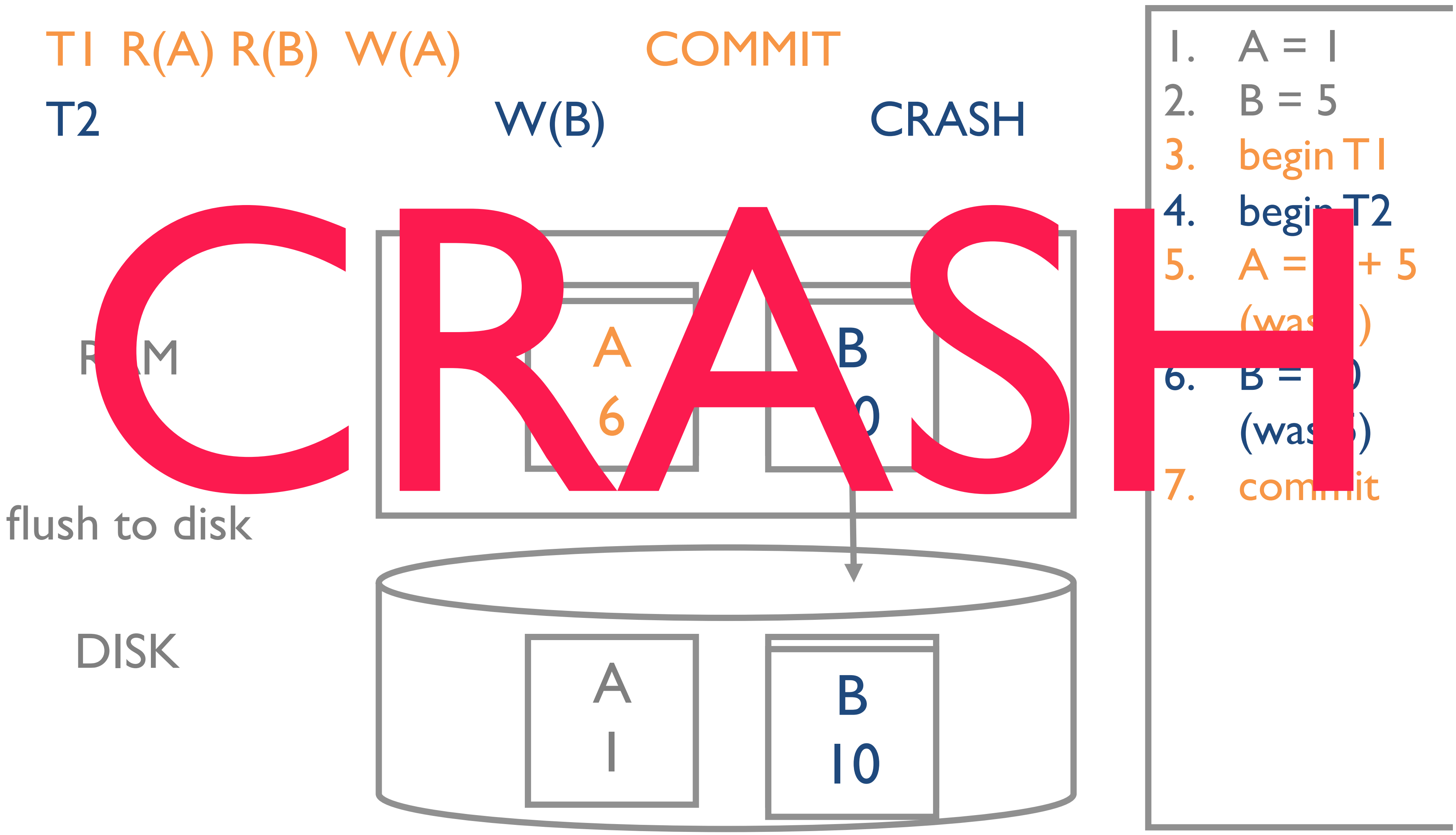
Aries Recovery Algorithm

3 phases

- Analyze the log to find status of all xacts
  - Committed or in flight?
- Redo xacts that were committed
  - Now at the same state at the point of the crash
- Undo partial (in flight) xacts

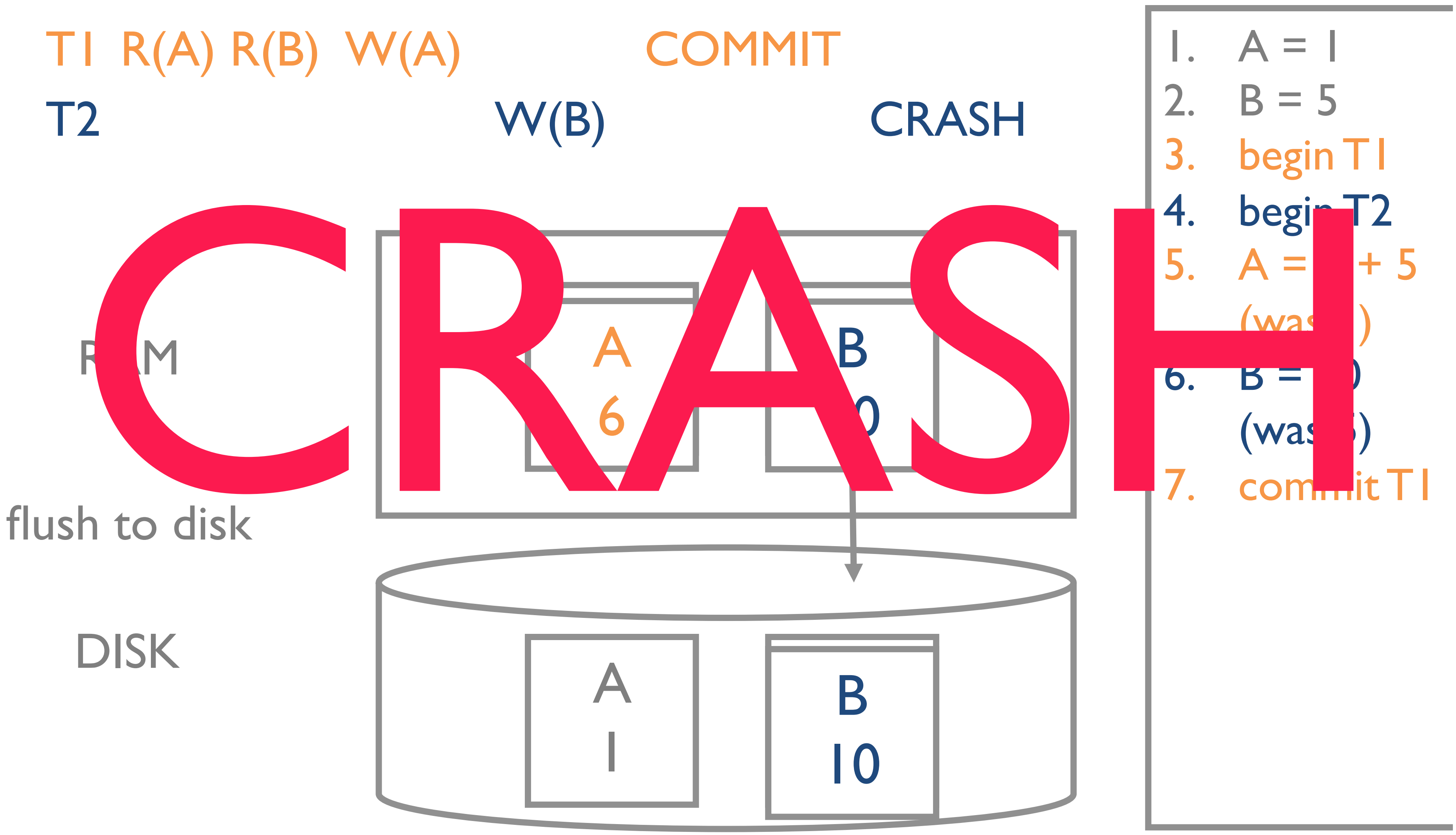
Recovery is *extremely* tricky and *must be correct*

Aries



T1 R(A) R(B) W(A) COMMIT
T2 W(B) CRASH
RAM
A
6
B
flush to disk
DISK
A
1
B
10
1. A = 1
2. B = 5
3. begin T1
4. begin T2
5. A = + 5
(was)
6. B =
(was)
7. commit
CRASH

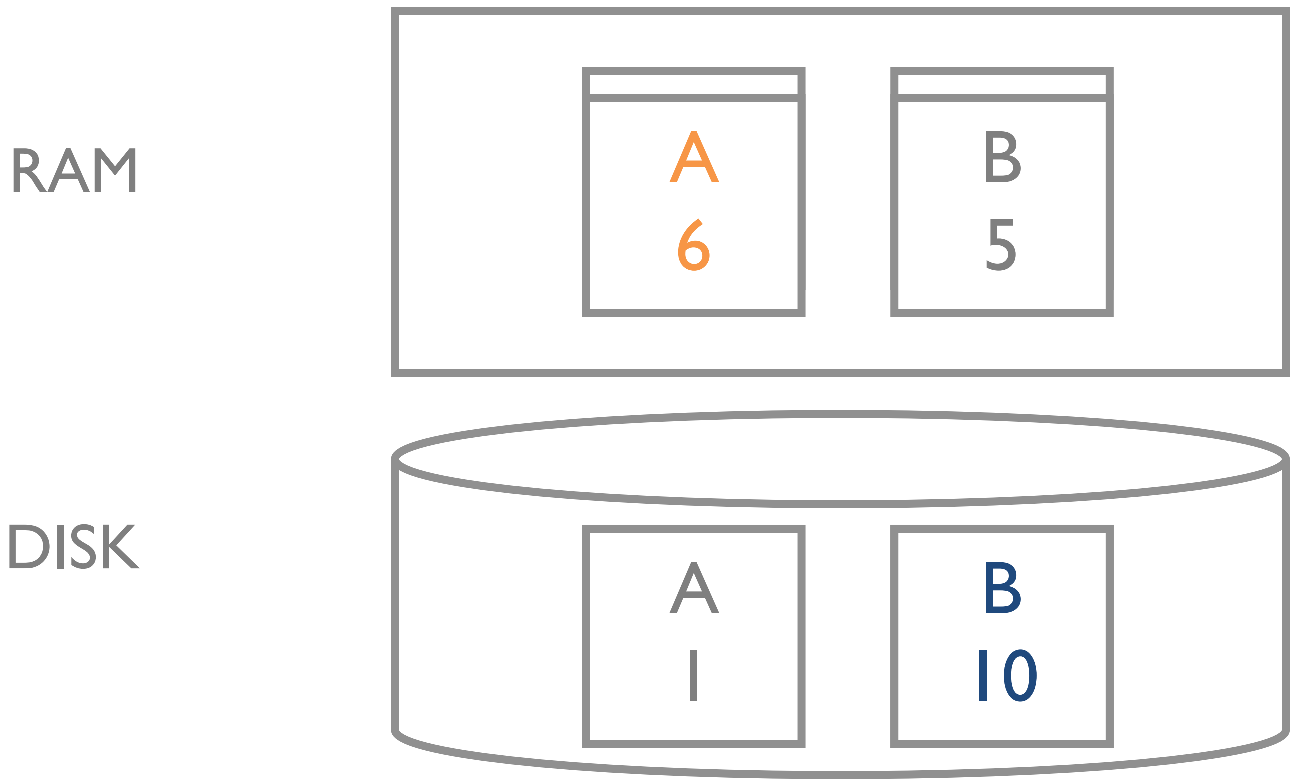
Aries: alternative flushing order



Aborts and Undos

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

T2 W(B) CRASH



1. A = 1
2. B = 5
3. begin T1
4. begin T2
5. A = 1 + 5 (was 1)
6. B = 10 (was 5)
7. commit T1
8. redo op5
9. undo op6

Durability/Recovery = The Log

Recovery depends on what failures are tolerable

Write-ahead log: Write changes to log before disk

Undo/redo OR ARIES: Permits transactions bigger than memory

Log both undo and redo information

Write pages to disk any time

On recovery: redo all committed txt, undo all non-committed txns

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, and a log file, run the proper sequence of undo/redos