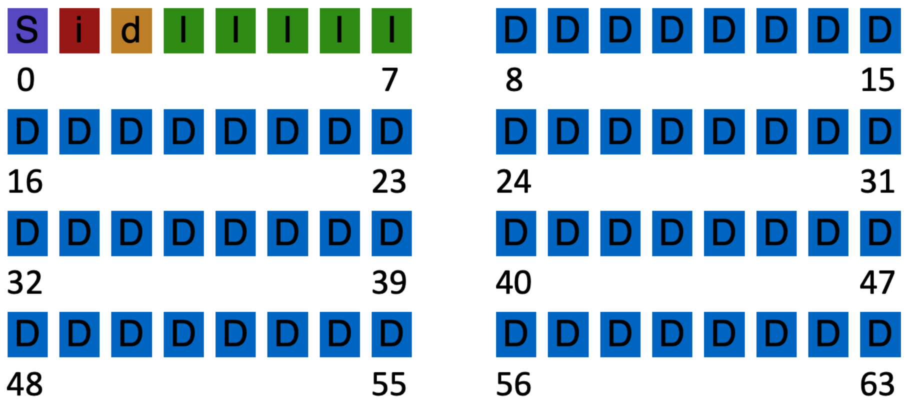
Crash Consistency



Inconsistency: a result of redundancy (non-independence)

Knowing A limits the possible values of B.

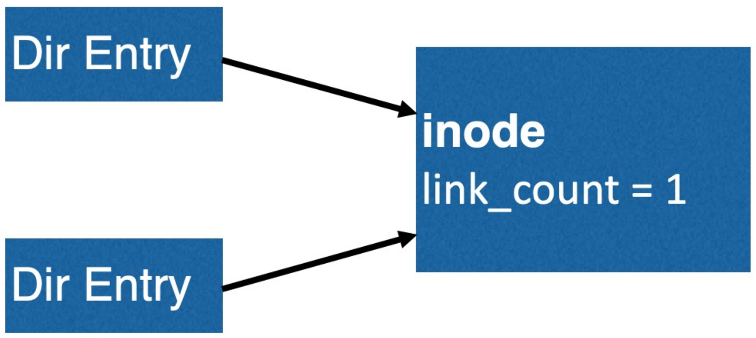
Inode pointer
Data block bitmap

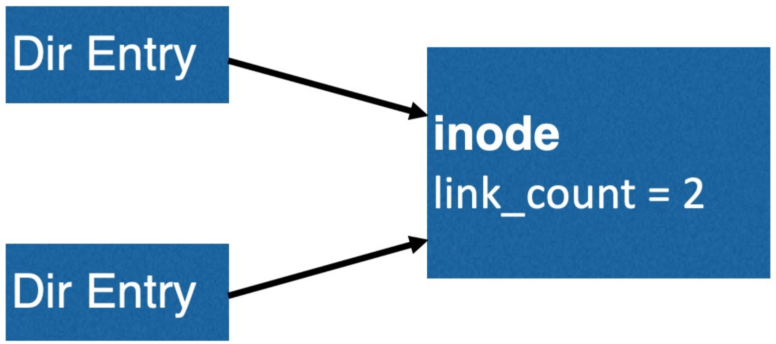
Inode link count
Directory entry

Superblock total block count
Inode pointer

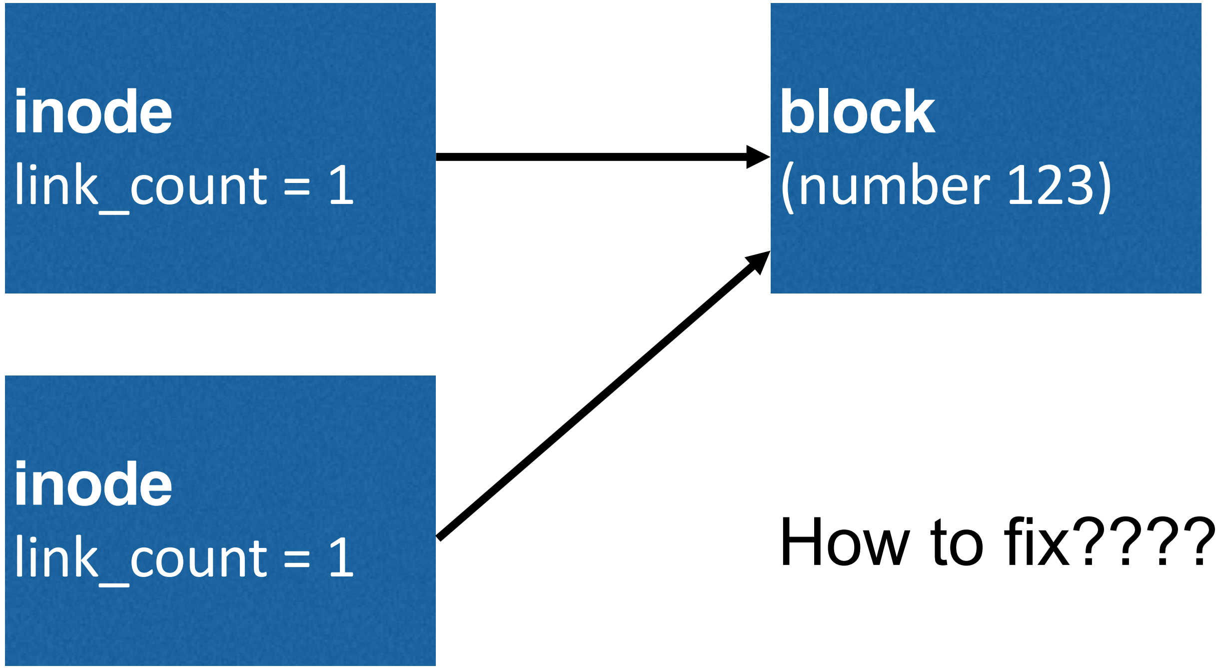
Filesystem checker: after a crash, look at data structures on disk, and make them consistent.

100s of checks & fixes



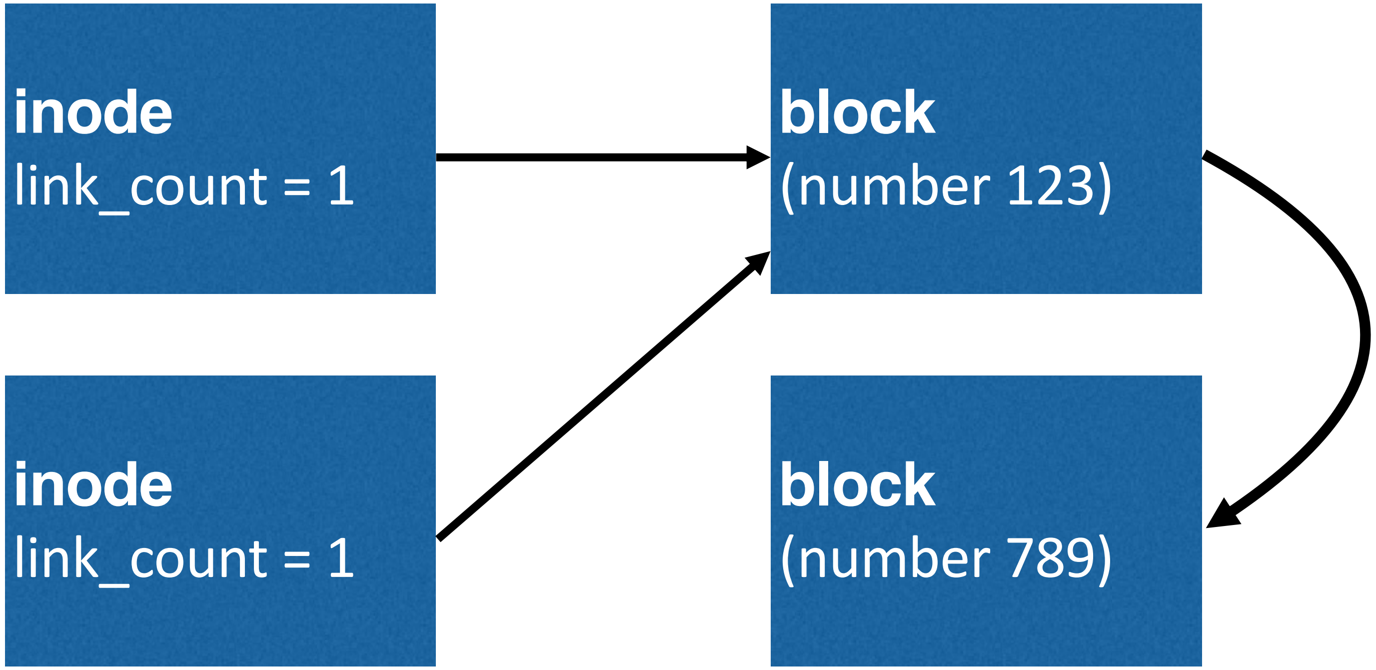


Duplicate Pointers

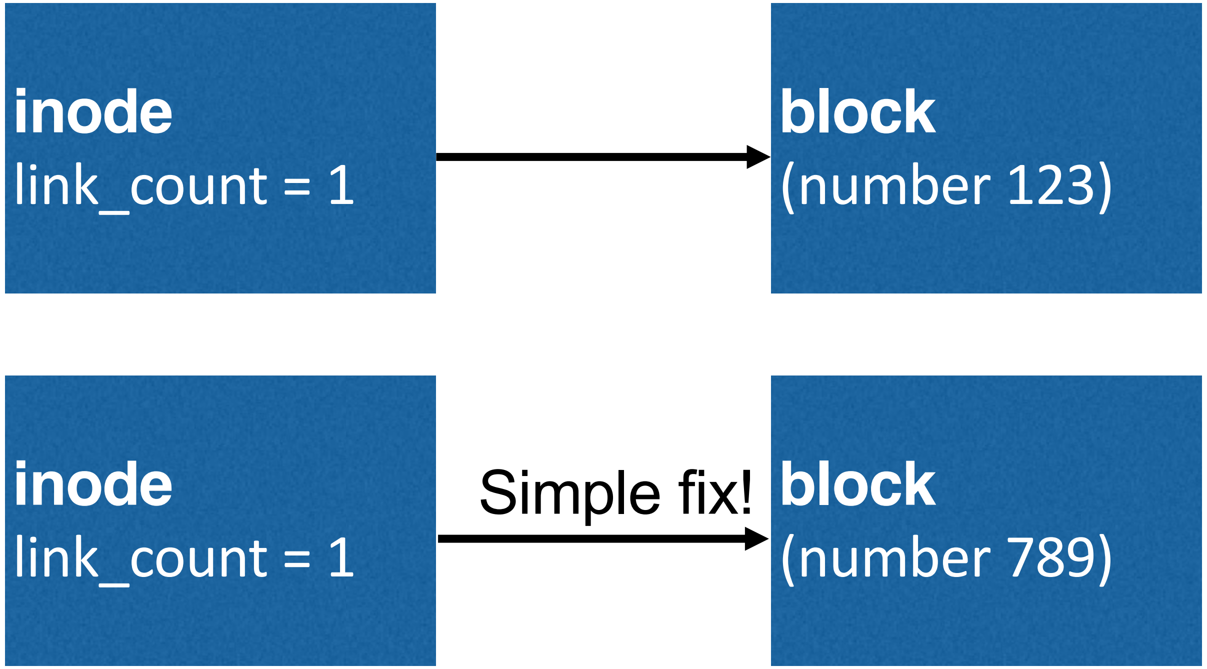


How to fix????

Duplicate Pointers

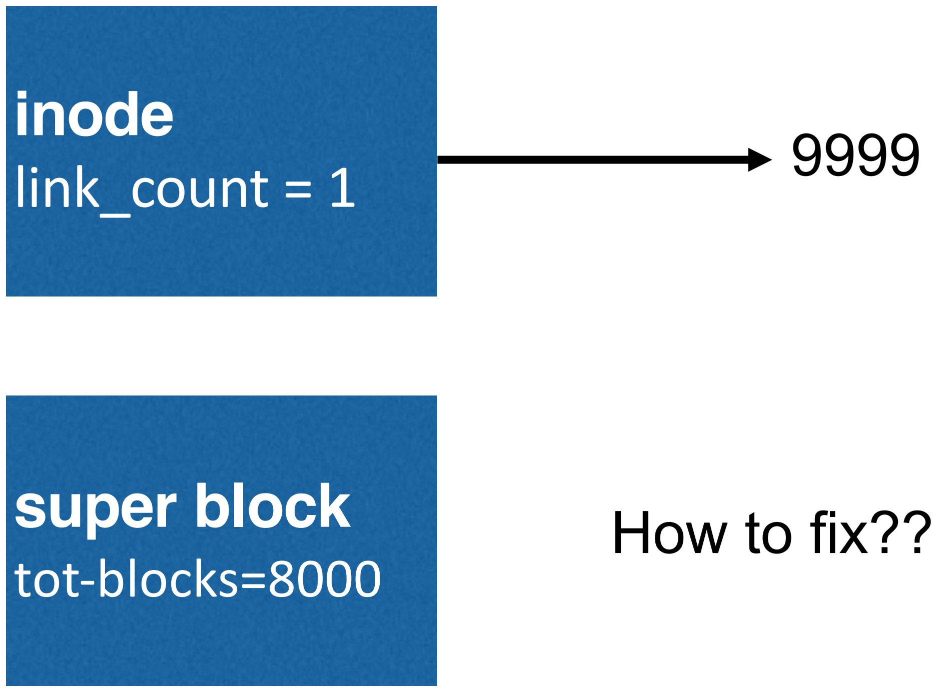


Duplicate Pointers



But is this correct?

Bad Pointer



How to fix???

Bad Pointer

inode
link_count = 1

Simple fix! (But is this correct?)

super block
tot-blocks=8000

Problems with fsck

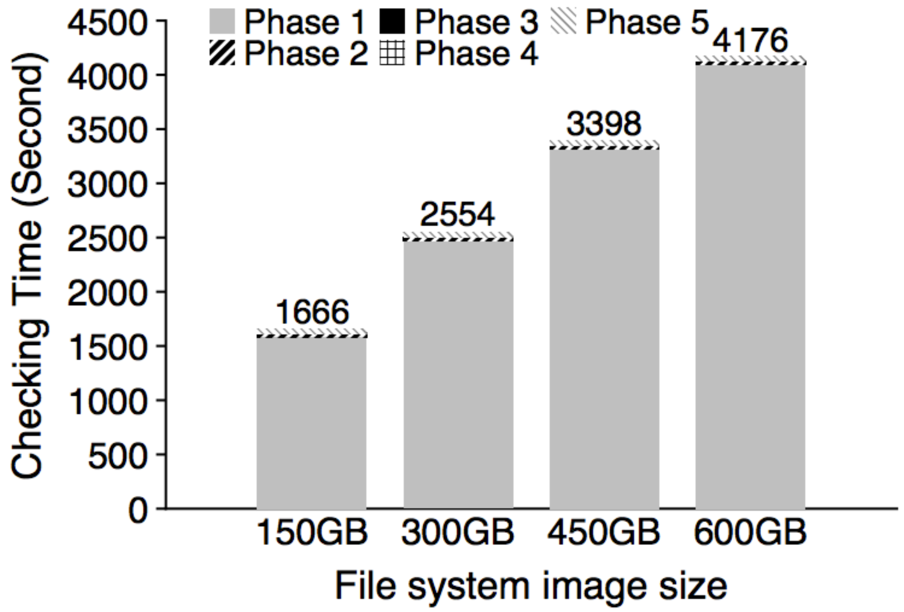
Problem 1:

- Not always obvious how to fix file system image

- Don’t know “correct” state, just a consistent one

- Easy way to get consistency: reformat disk!

Problem 2: fsck is very slow



Checking a 600GB disk takes ~70 minutes

ffsck: The Fast File System Checker

Ao Ma, EMC Corporation and University of Wisconsin—Madison; Chris Dragga,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

Consistency Solution #2: Journaling

Goals

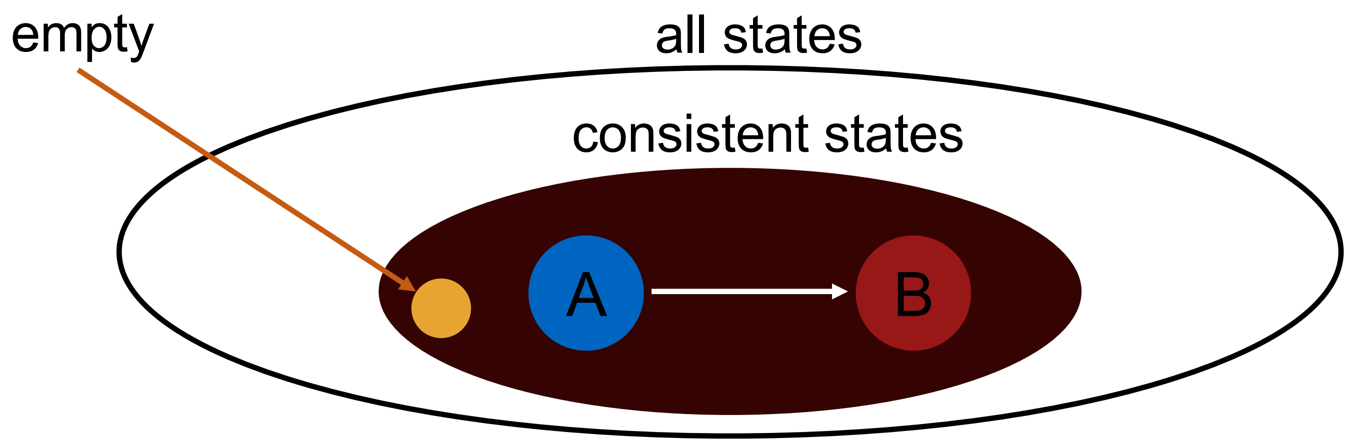
- Ok to do some **recovery work** after crash, but not to read entire disk
- Don't move file system to just any consistent state, get **correct** state (in most cases)

Strategy

- Atomicity
- Definition of atomicity for **concurrency**
  - operations in critical sections are not interrupted by operations on related critical sections
- Definition of atomicity for **persistence**
  - collections of writes are not interrupted by crashes;
    either (all new) or (all old) data is visible

Consistency vs Correctness

Say a set of writes moves the disk from state A to B



fsck gives consistency
Atomicity gives A or B.

Journaling: General Strategy

Never delete ANY old data, until, ALL new data is safely on disk

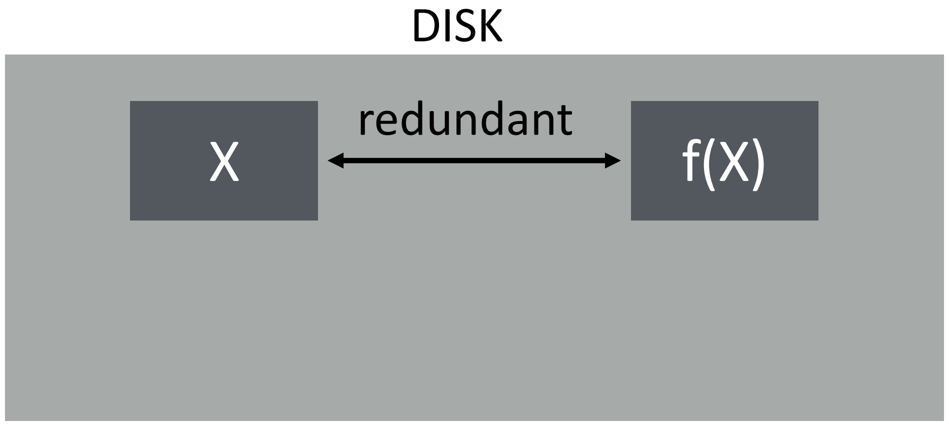
Ironically, adding redundancy to fix the problem caused by redundancy.

Do a little extra work during regular operation, to avoid A LOT OF extra work during recovery

Also referred to as write-ahead logging

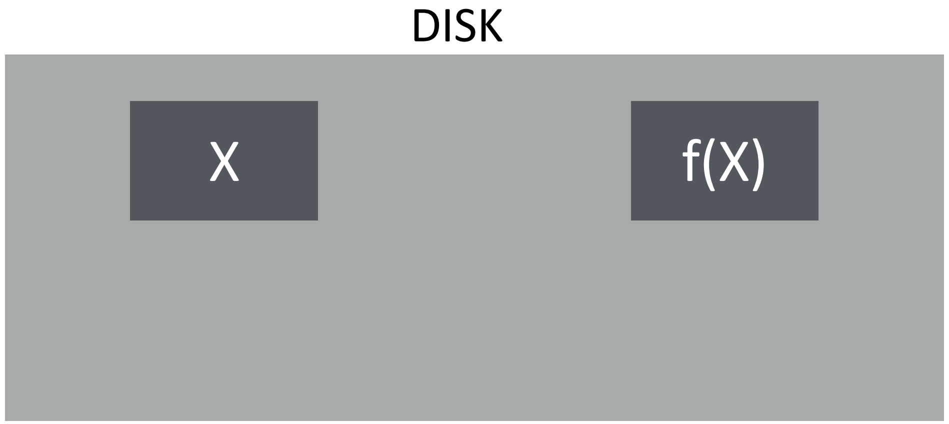
Fight Redundancy with Redundancy

Want to replace X with Y. Original:



Fight Redundancy with Redundancy

Want to replace X with Y. Original:

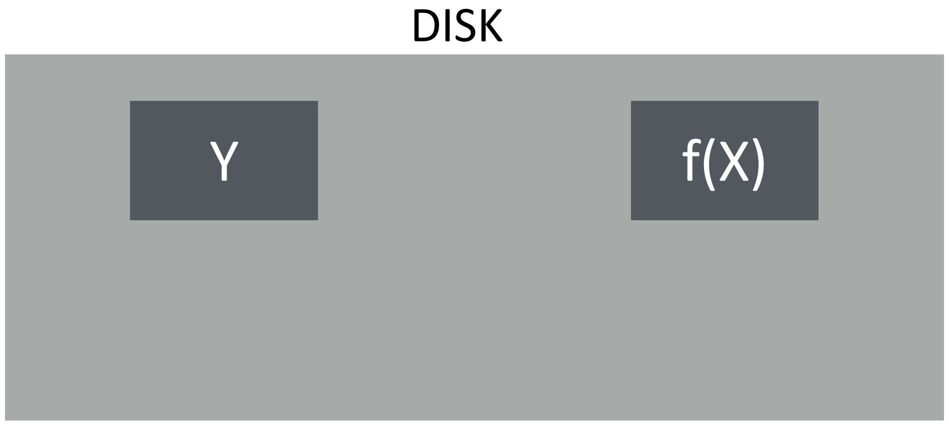


Good time to crash?

Yes, good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

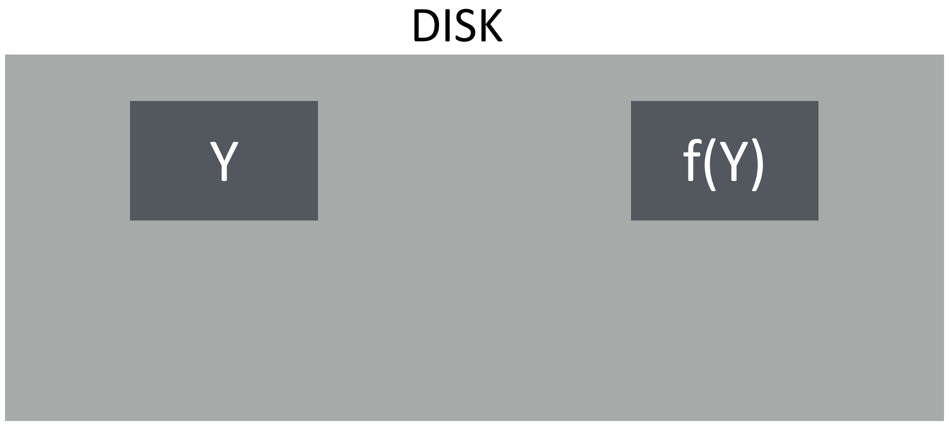


Good time to crash?

bad time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. Original:

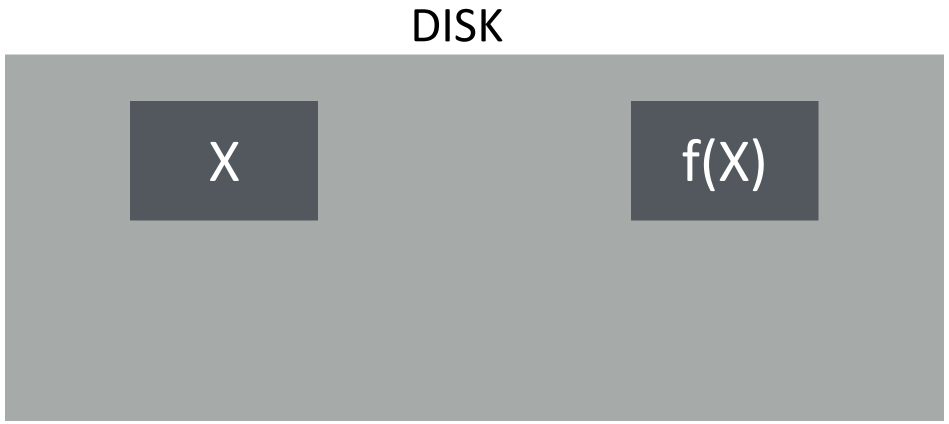


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. **With journal:**

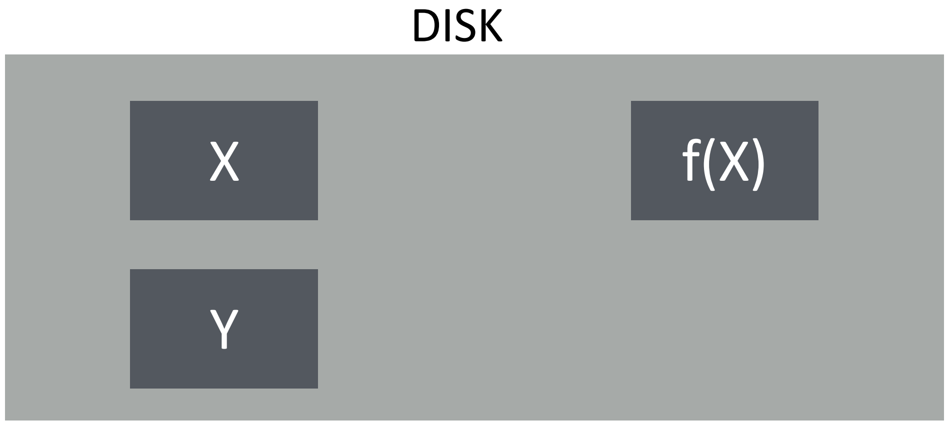


Good time to crash?

good time to crash

Fight Redundancy with Redundancy

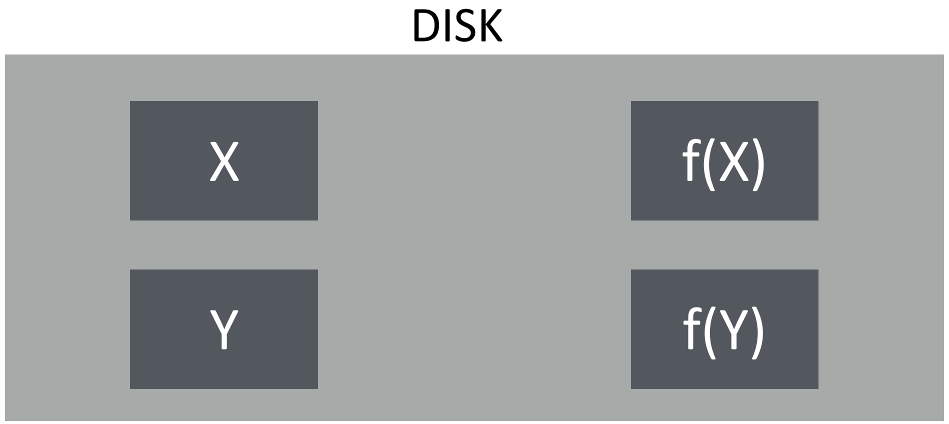
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

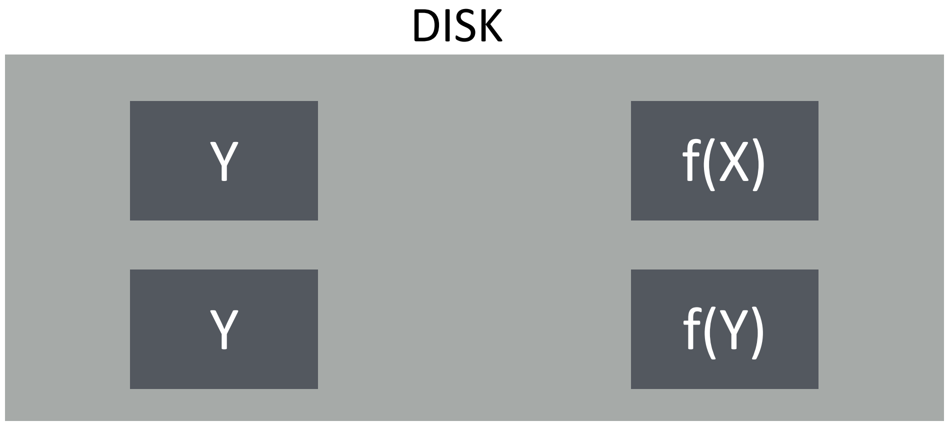
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

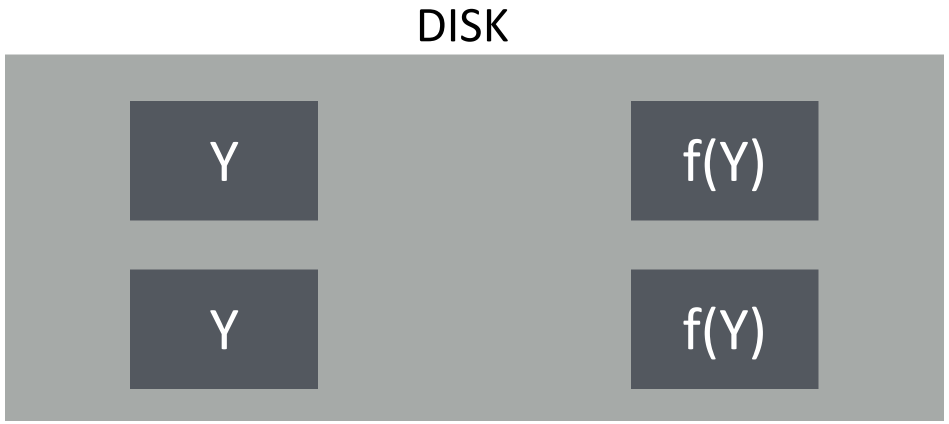
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

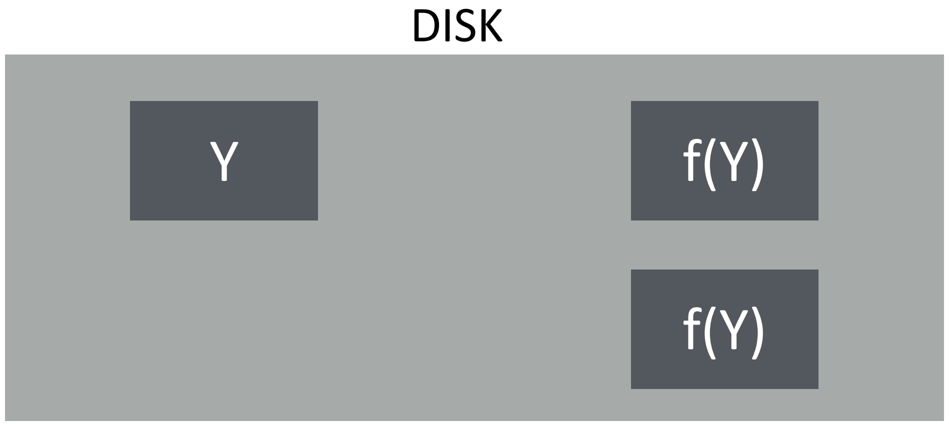
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

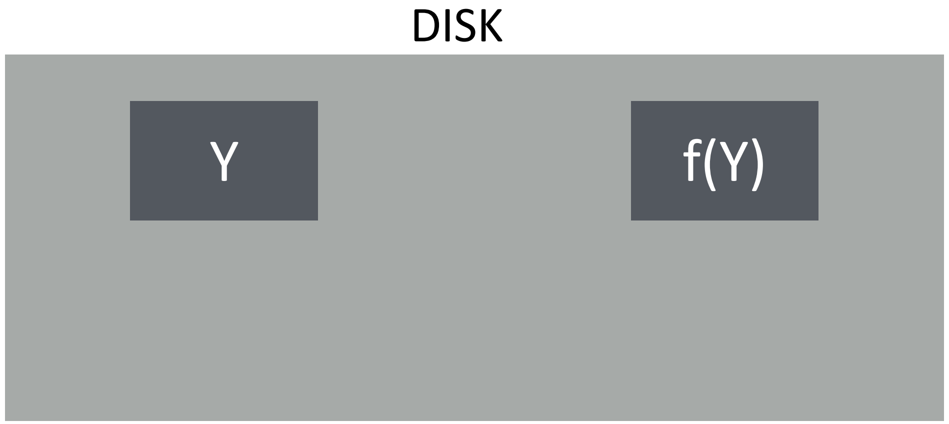
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

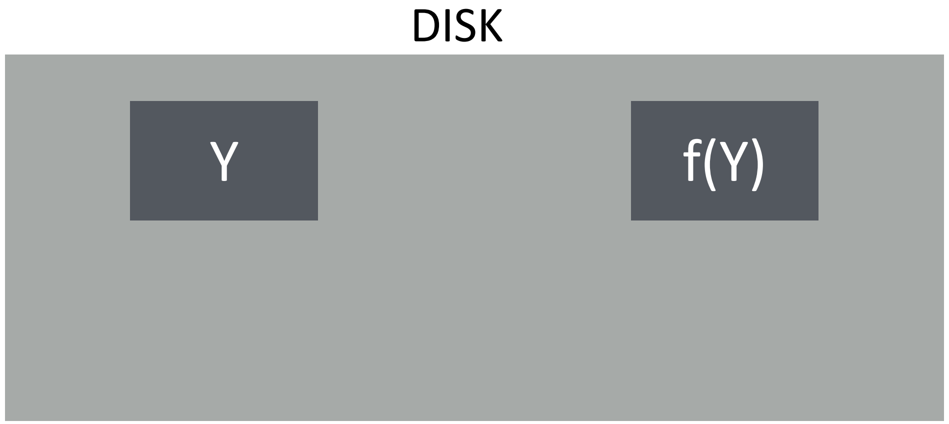
Want to replace X with Y. With journal:



good time to crash

Fight Redundancy with Redundancy

Want to replace X with Y. With journal:



With journaling, it's always a good time to crash!

Inconsistency: how do we fix it?

Develop algorithm to atomically update two blocks:
Write 10 to block 0; write 5 to block 1

Assume these are only blocks in file system.

Assume: only 1 block, not multiple, can be written in one shot

Usage Scenario: Block 0 stores Alice's bank account;
Block 1 stores Bob's bank account; transfer $2 from Alice to Bob

Time	Block 0	Block 1
1	12	3
2	12	5
3	10	5

don't crash here!

A wrong update algorithm can lead to inconsistent states (non-atomic updates)

Initial Solution: Journal New Data

Suppose we make updates on a copy of each block first

(note: must allocate space for these copies)

Time	Block 0	Block 1	J:2	J:3	J:valid:4
1	12	3	0	0	0
2	12	3	10	0	0
3	12	3	10	5	0
4	12	3	10	5	1
5	10	3	10	5	1
6	10	5	10	5	1
7	10	5	10	5	0

Crash here? (times 1–3)
→ Old data

Crash here? (times 4–7)
→New data

Let's understand behavior if crash occurs after each write

Note: every step assumes previous update committed to disk

Scenario: Block 0 stores Alice's bank account; Block 1 stores Bob's bank account; transfer $2 from Alice to Bob

```
void update_accounts(int cash1, int cash2) {
      write(cash1 to block 2) // Alice backup
      write(cash2 to block 3) // Bob backup
      write(1 to block 4)     // backup is safe
      write(cash1 to block 0) // Alice
      write(cash2 to block 1) // Bob
      write(0 to block 4)     // discard backup
}
```

Suppose the machine failed somewhere along the way…

```
void recovery() {
      if(read(block 4) == 1) {
            write(read(block 2) to block 0) // restore Alice
            write(read(block 3) to block 1) // restore Bob
            write(0 to block 4)             // discard backup
      } // no recovery needed if !(read(block 4) == 1)
}
```

Terminology

Extra blocks are called a journal

* (all blocks considered same: inode, superblock, …)

The writes to the journal are a journal transaction

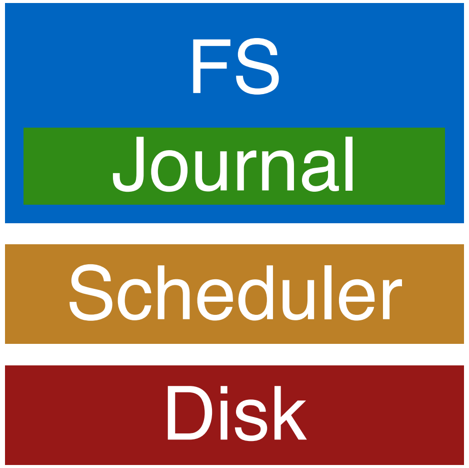
The last valid bit written is a journal commit block

* journal commit relies on disk committing single-block writes fully or not at all

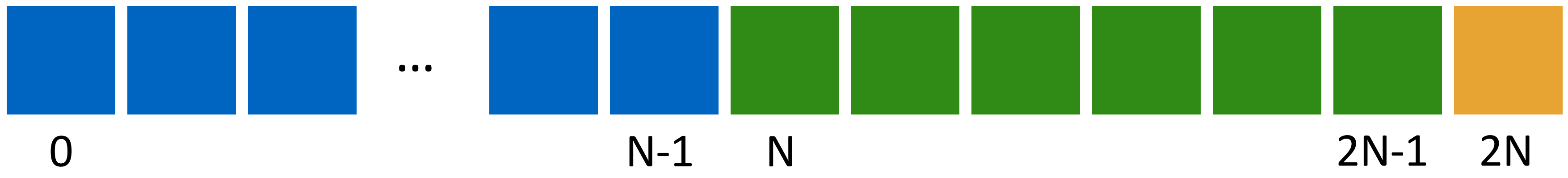
The writing of the actual data (in place) is called checkpoint

Approach described above: Data journaling

File System Integration



Problems with data journaling approach



Disadvantages?

- slightly < half of disk space is usable

- transactions copy all the data (1/2 disk bandwidth!)

- disk commit forces hardware to seek to random locations one after another

Fix #1: Small Journals

Still need to first write all new data elsewhere before overwriting new data

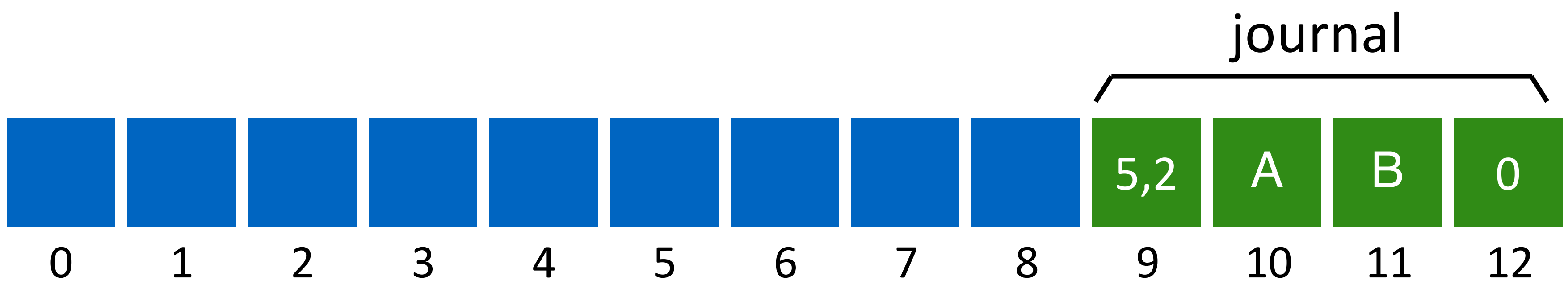
Goal:

- Reuse small area as backup for any block

How?

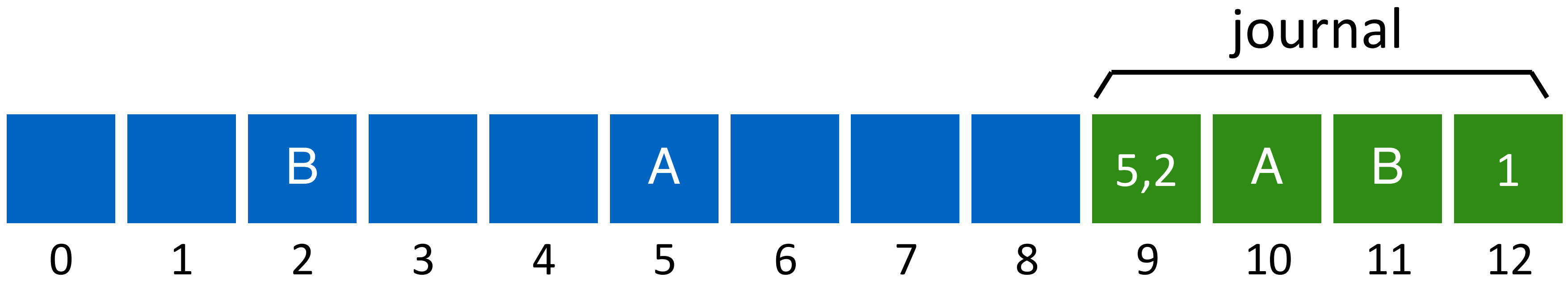
- Store block numbers in a transaction "header” in journal

New Layout



transaction: write A to block 5; write B to block 2

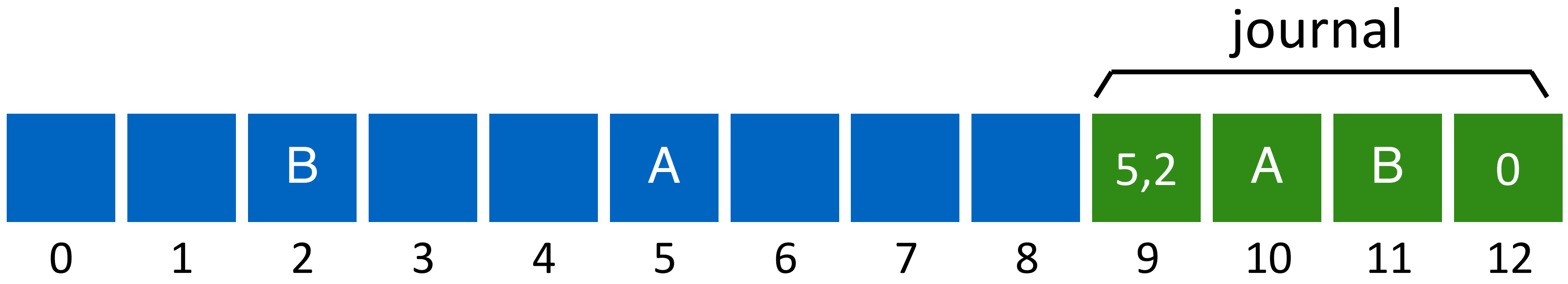
New Layout



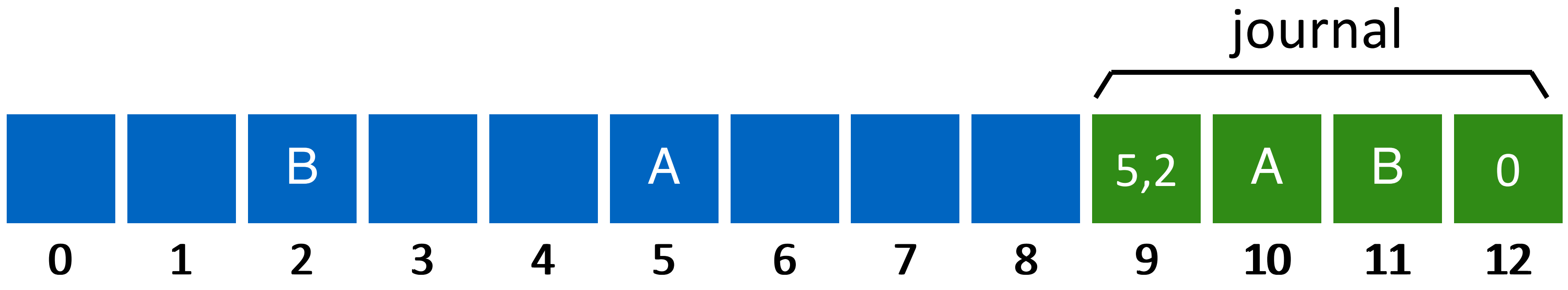
transaction: write A to block 5; write B to block 2

Checkpoint: Writing new data to in-place locations

New Layout

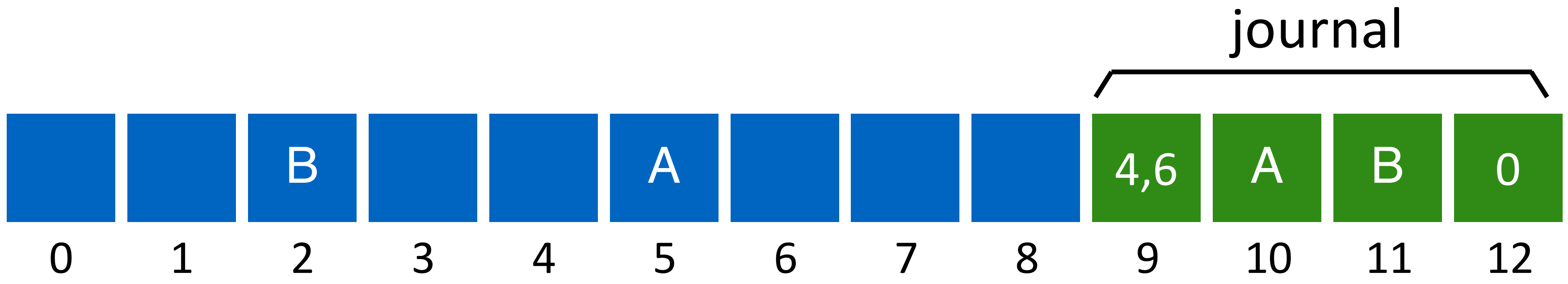


New Layout



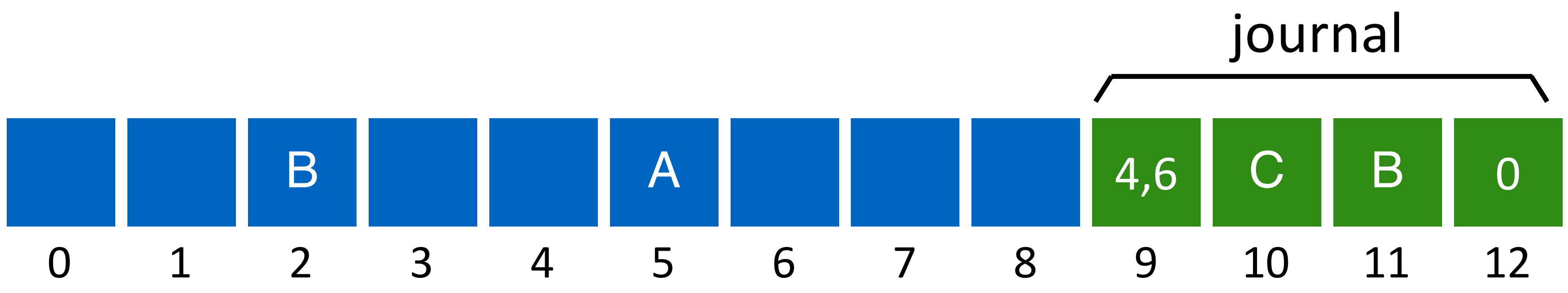
transaction: write C to block 4; write T to block 6

New Layout



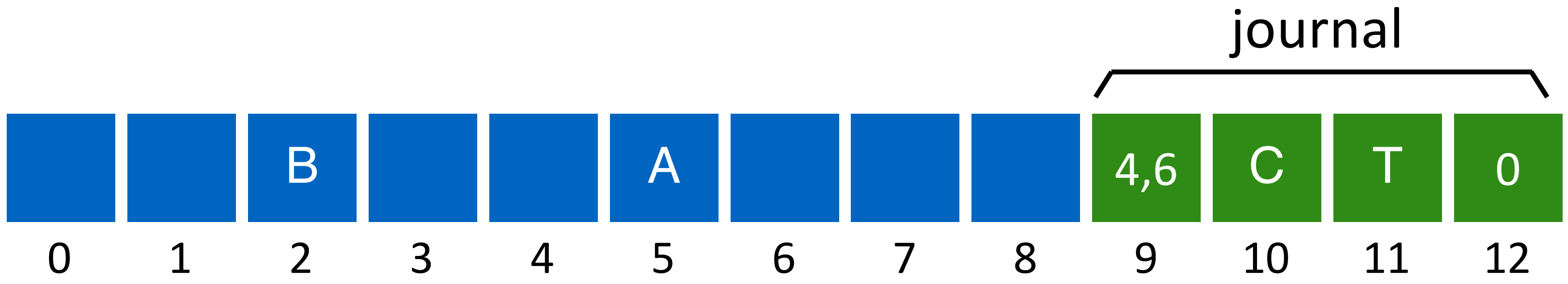
transaction: write C to block 4; write T to block 6

New Layout



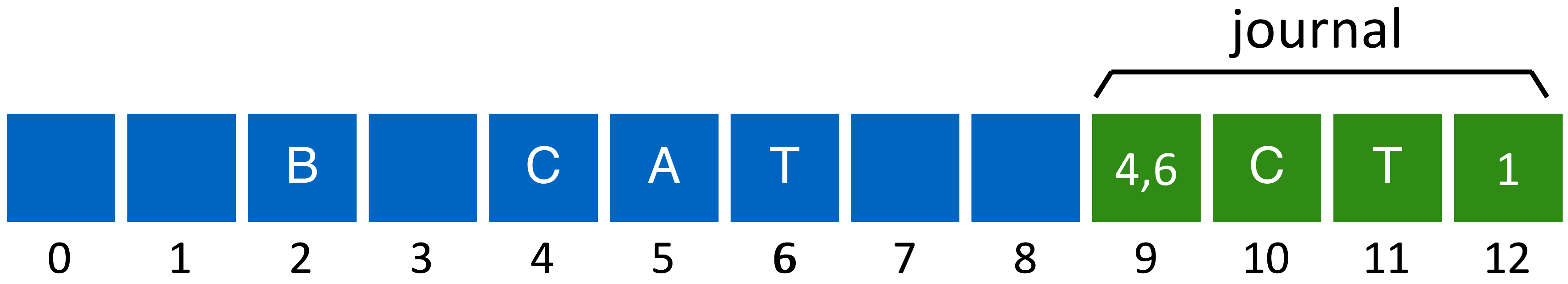
transaction: write C to block 4; write T to block 6

New Layout



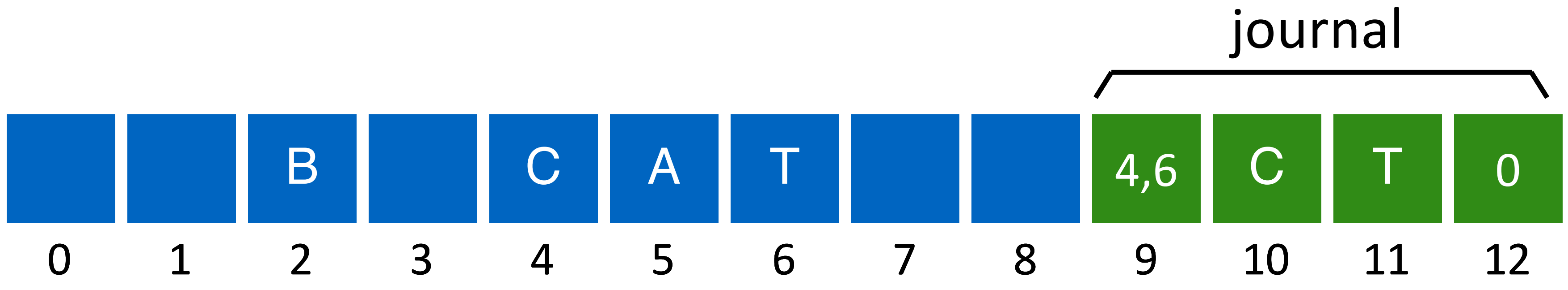
transaction: write C to block 4; write T to block 6

New Layout



transaction: write C to block 4; write T to block 6

New Layout

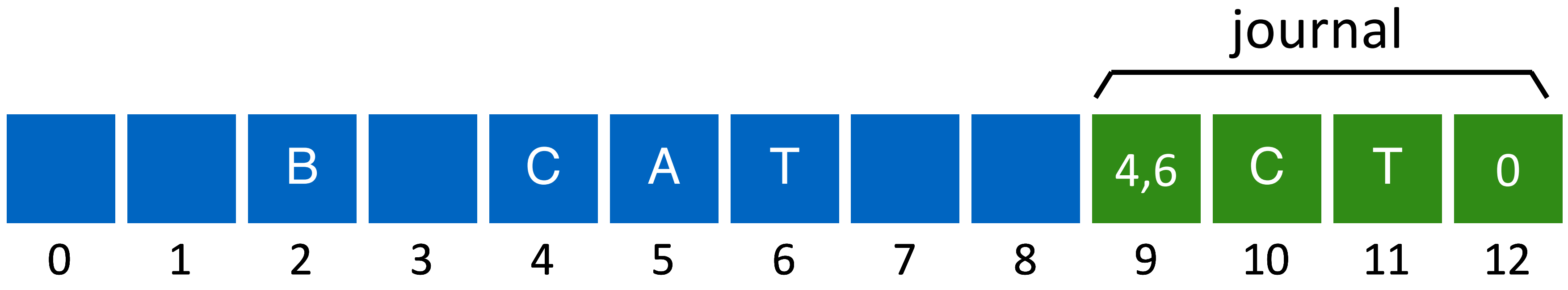


transaction: write C to block 4; write T to block 6

Optimizations

1. Reuse small area for journal
2. Barriers - (fsync)
3. Checksums
4. Circular journal
5. Metadata journal

Correctness depends on Ordering



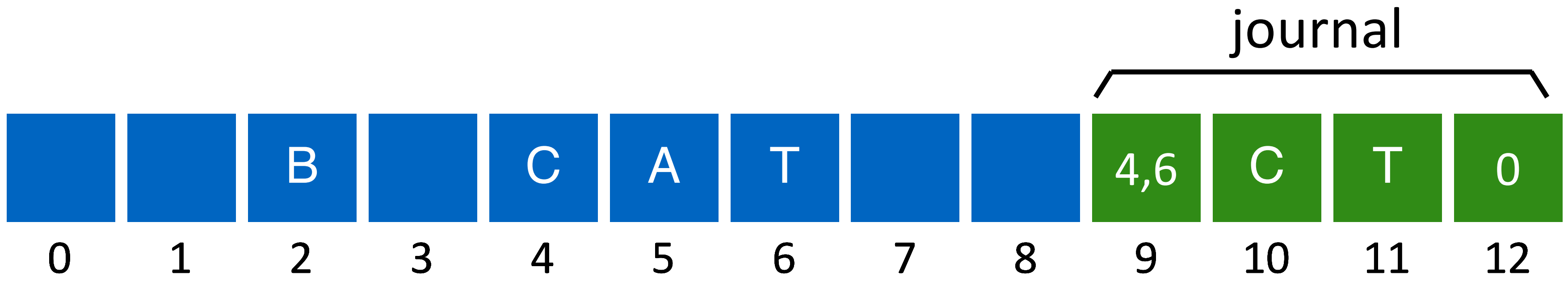
transaction: write C to block 4; write T to block 6

write order: 9, 10, 11, 12, 4, 6, 12

Enforcing total ordering among these writes is inefficient (random writes)

Instead: Use barriers w/ disk cache flush at key points (Q: when?)

Ordering



transaction: write C to block 4; write T to block 6

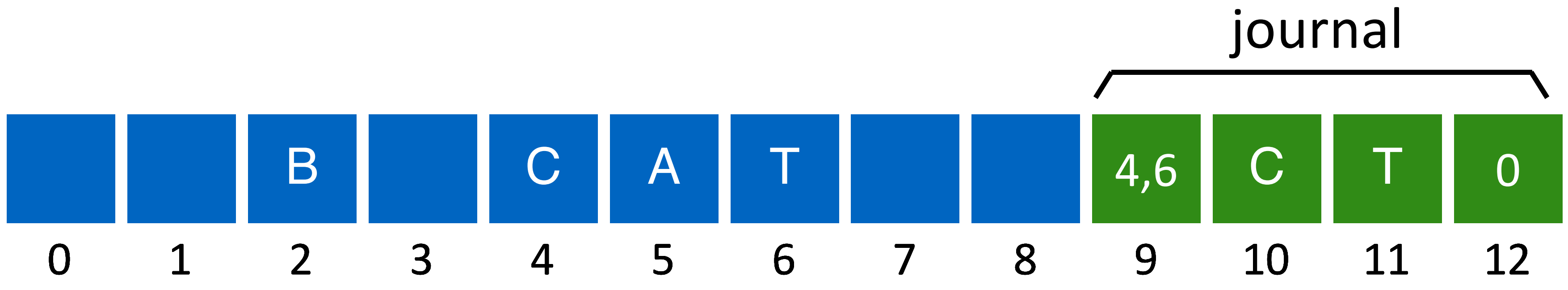
write order: 9,10,11 | 12 | 4,6 | 12

Use barriers at key points in time:
1) Before journal commit, ensure journal transaction entries complete
2) Before checkpoint, ensure journal commit complete
3) Before free journal, ensure in-place updates complete
Force disk controller to commit data through fsync()/sync()

Optimizations

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Metadata journal

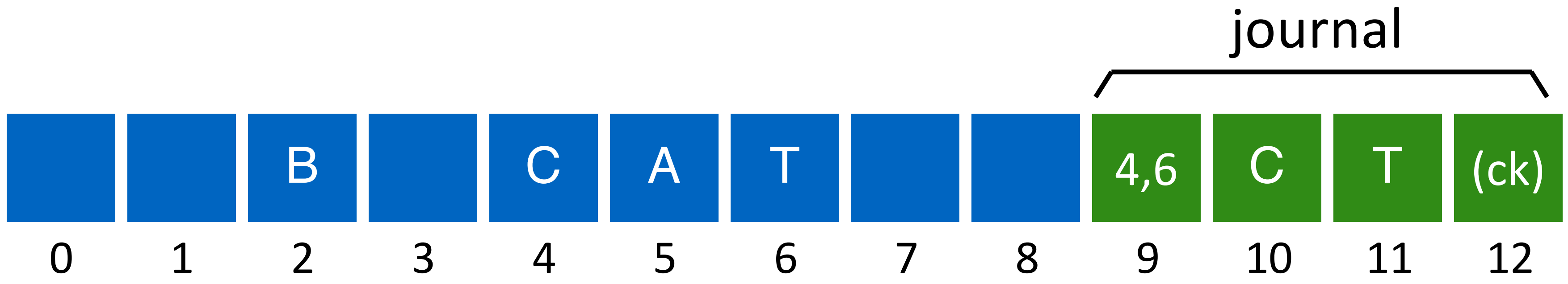
Checksums to avoid txn commit barrier



write order: 9,10,11 | 12 | 4,6 | 12

Can we get rid of barrier between (9, 10, 11) and 12 ?

Checksums to avoid txn commit barrier



write order: 9,10,11,12 | 4,6 | 12

An easy-to-compute function f. If x != y, likely f(x) != f(y)

In last transaction block, store checksum of rest of transaction

data in 12 = checksum(9, 10, 11)

During recovery:

If checksum does not match transaction, treat transaction as not committed

Optimizations

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Metadata journal

Write Buffering Optimization

Note: after journal write, there is no rush to checkpoint

- If system crashes, still have persistent copy of written data!

Journaling is sequential, checkpointing is random

Solution? Delay checkpointing for some time

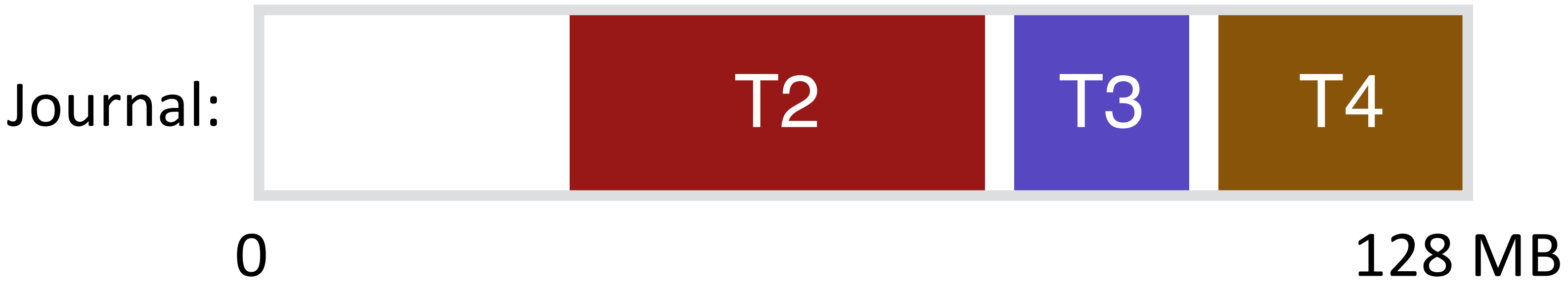
Difficulty: need to reuse journal space

Solution: keep many transactions for un-checkpointed data

Circular Buffer

Keep data also in memory until checkpointed on disk

Circular Buffer

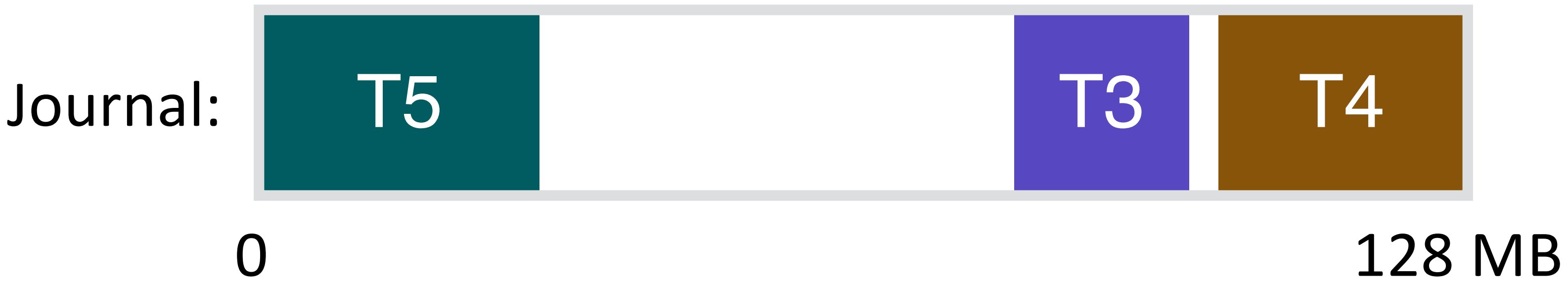


checkpoint and cleanup

Circular Buffer

New transaction reuses cleaned-up space

Circular Buffer



checkpoint and cleanup

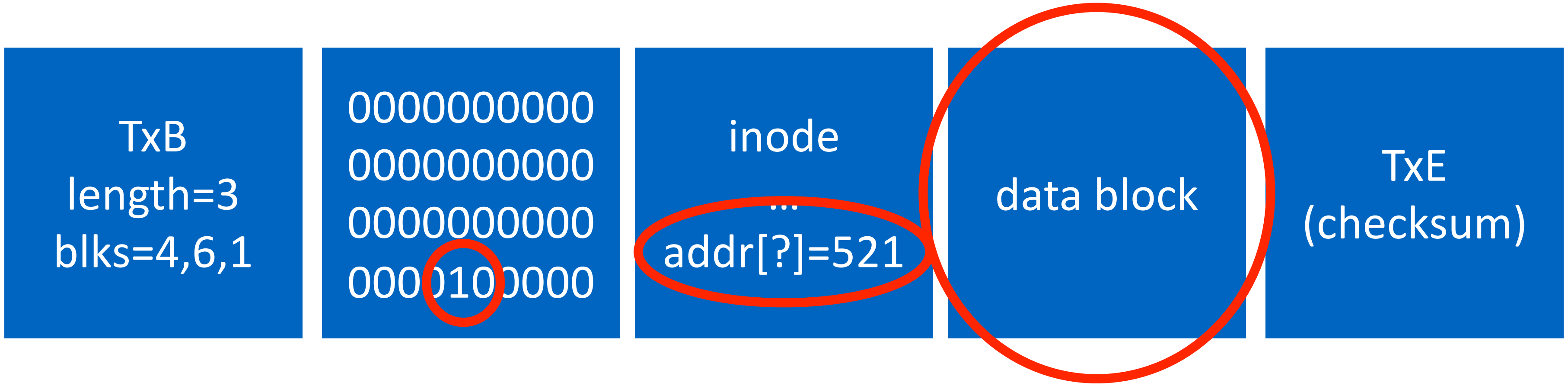
Optimizations

1. Reuse small area for journal
2. Barriers
3. Checksums
4. Circular journal
5. Metadata journal

Data Journal

Example: adding a new data block when appending to a file

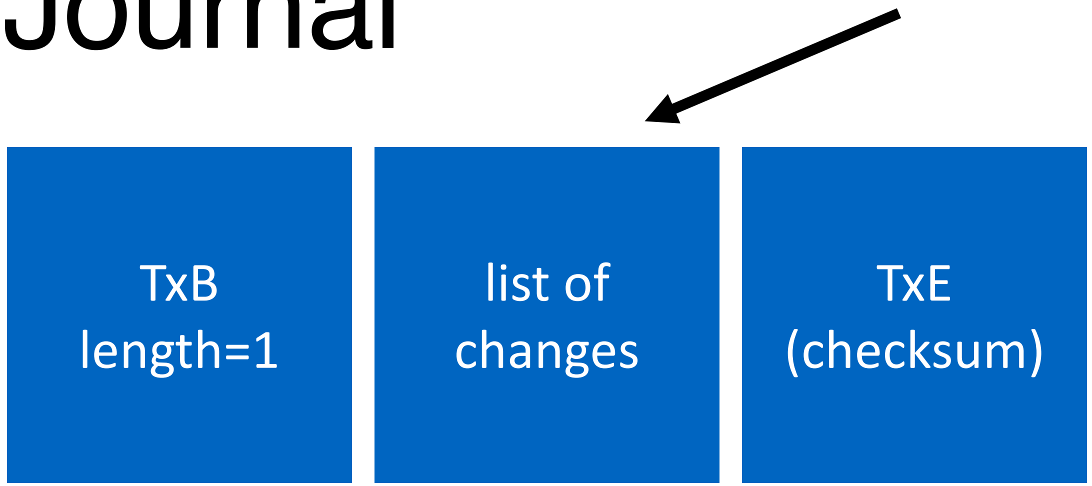
Data Journal



Example: adding a new data block when appending to a file

Metadata Journal

Should “changes” include data blocks?



Metadata journals record changes to bytes, not contents of new blocks

Tradeoff: More work upon recovery!

Need to read existing contents of in-place data and (re-)apply changes

Logical journaling

Option 1: avoid writing disk blocks twice

Observation: some blocks (e.g., user data) could be considered less important

Strategy: journal only metadata changes, including:

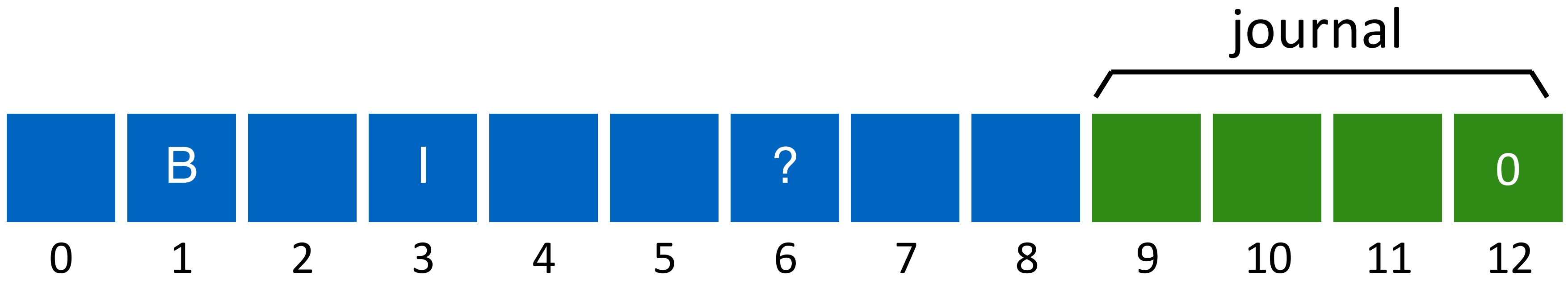
superblock, bitmaps, inodes, indirects, directories

For regular data, write it back whenever convenient. Problem?

Files may contain garbage if fail before writing the data.

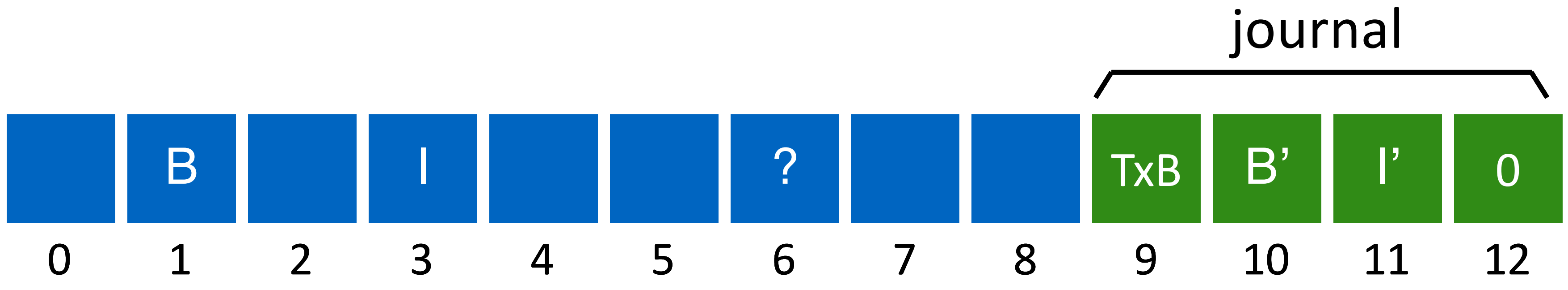
Unordered metadata journaling

Unordered Metadata Journal



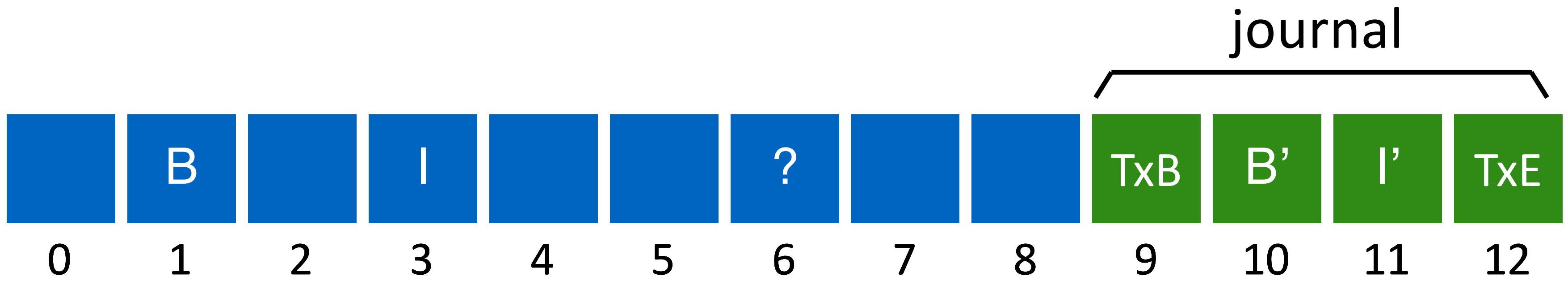
transaction: append to inode I

Unordered Metadata Journal



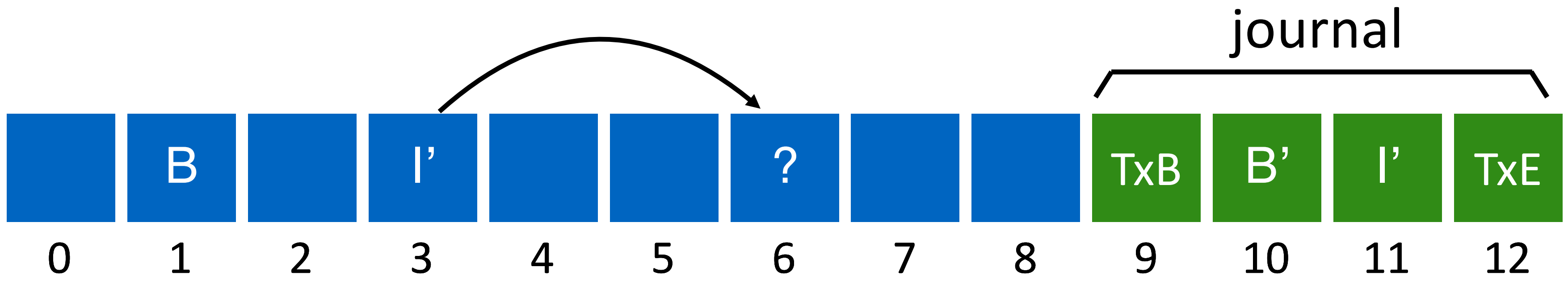
transaction: append to inode I

Unordered Metadata Journal



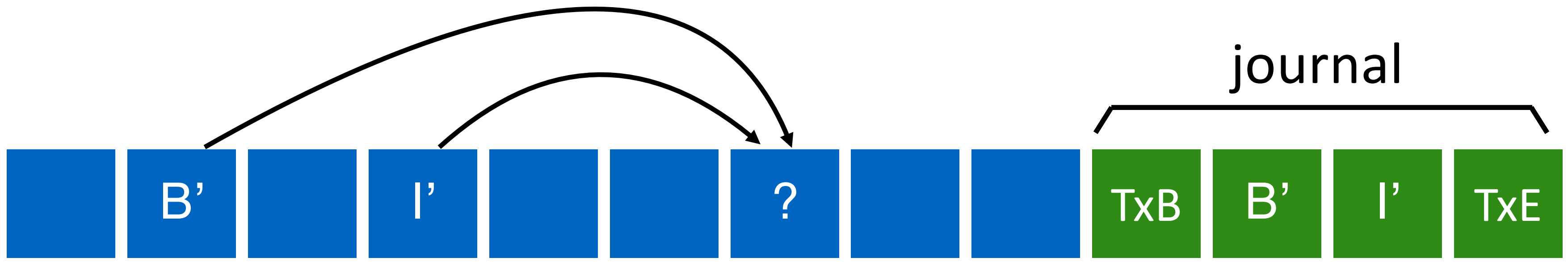
transaction: append to inode I

Unordered Metadata Journal



transaction: append to inode I

Unordered Metadata Journal



transaction: append to inode I

what if we crash now?

Point to garbage data?
Possibly leak sensitive data?

Solutions?

Option 2: Ordered Metadata Journaling

Still only journal metadata

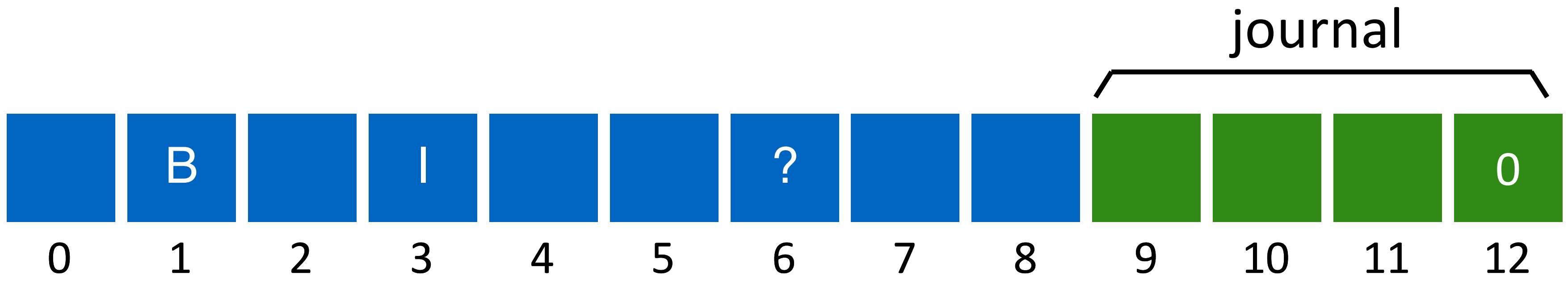
But write data block **before** the transaction commits

No leaks of sensitive data or data loss if metadata consistent

Tip: write the "pointed-to" thing first before writing the pointer

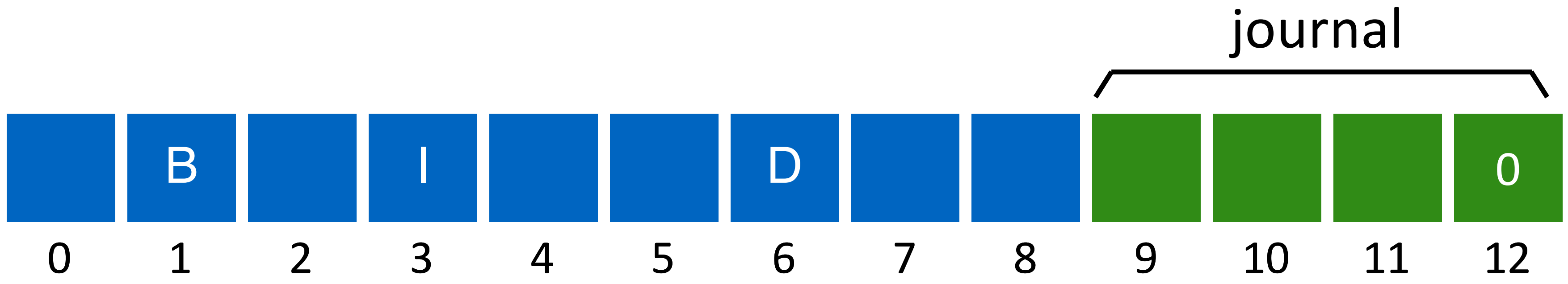
(more generally applicable)

Ordered Journal



transaction: append to inode I

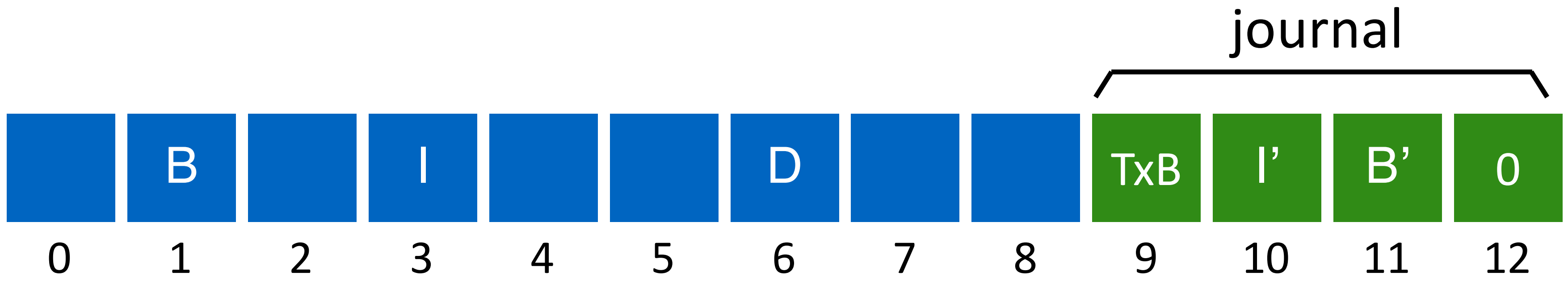
Ordered Journal



transaction: append to inode I

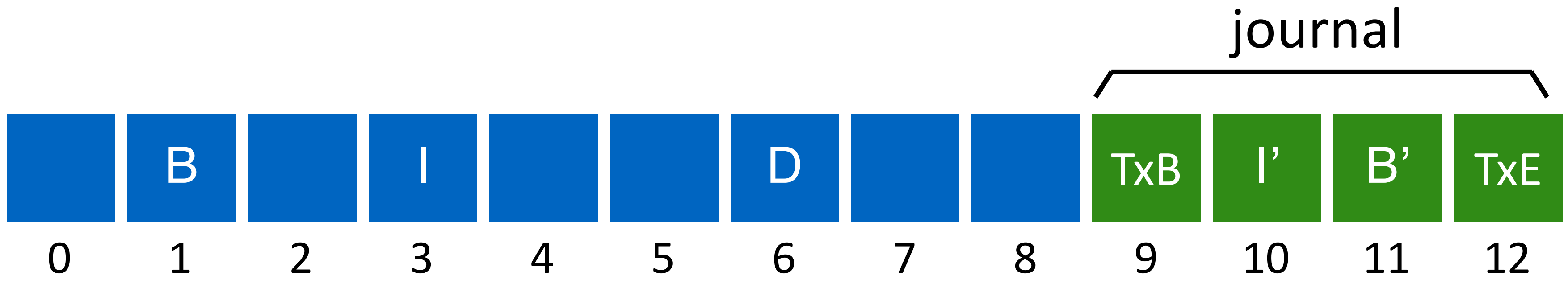
What happens if crash now?
B indicates D currently free, I does not point to D;
Lose D, but that might be acceptable

Ordered Journal



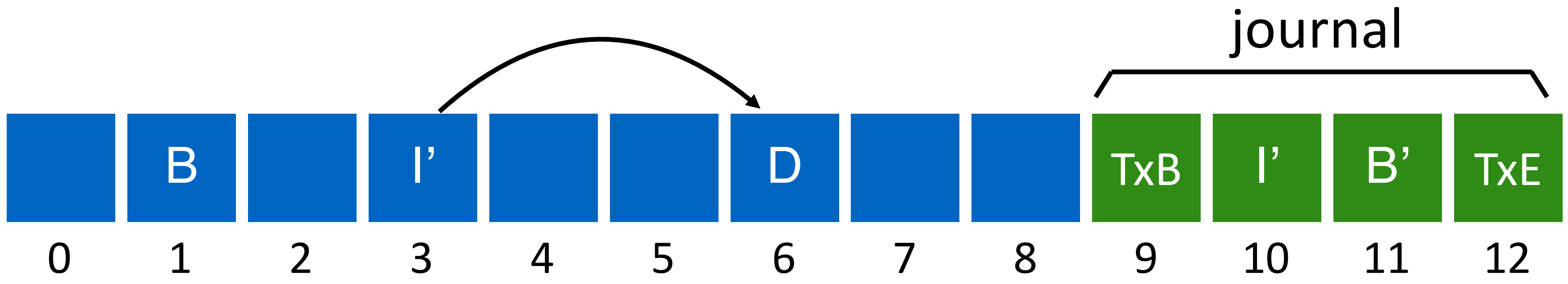
transaction: append to inode I

Ordered Journal



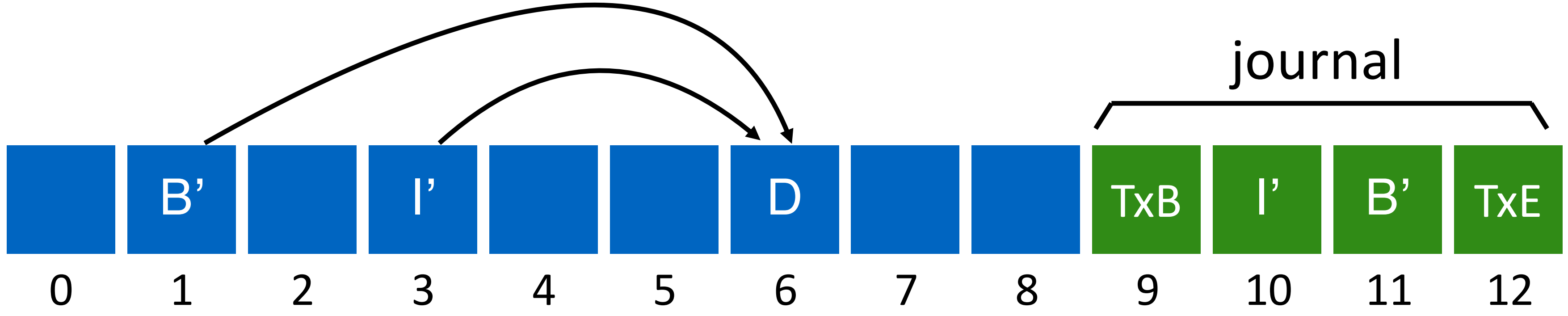
transaction: append to inode I

Ordered Journal



transaction: append to inode I

Ordered Journal



transaction: append to inode I

Summary

Most modern file systems use journals

- Ordered metadata journaling mode is popular

FSCK is still useful for weird cases: bit flips, filesystem bugs, …

Some file systems don't use journals, but still usually write new data before deleting old (copy-on-write file systems)

Need for crash consistency makes persistent storage different from physical (main) memory

Operating Systems

Rutgers University | New Brunswick

Outro

Summary

- An OS is a set of abstractions, mechanisms, policies to access your machine hardware

- OS work with, rely on, and support hardware capabilities
  - When hardware changes, OS support must change

- Virtualization: getting an app to use machine as if it's own
- Concurrency: doing things simultaneously on a machine
- Persistence: accessing and storing data that remains after failure

OK, now what?

Go about life as usual (1/3)

- But live with a deeper appreciation of how your machines work
- Example: When you buy more memory, what do you expect to run faster, and what won't?
- What does your machine hardware guarantee? What doesn't it?

Put your knowledge to use in tech work (2/3)

- You’ve programmed significantly in this course. In future:
- Become a power-user of the machine
- Debug and optimize performance for your software
  - Why is ML inference slow? I have enough memory, so why does my program run slower when I ask for more memory?
- How do you design a complex system?
- What principles should you use to organize functionality? What functionality goes where?

Go deeper (3/3)

- Use your knowledge to solve a problem you care about

- Learn more about computer systems
  - Rutgers CS curriculum: CS 519, 552, 546, 539, 553, 545, …

- Push the boundaries of the field
  - Talk to me about research

Thanks & all the best!