Welcome back!

PERSISTENCE: SOLID-STATE DEVICES

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

Regrade Piazza

Project 5 grades out, Project 6 (this week)

Project 7 Issues?!?

→ Simplify what we grade

→ manual grades

Midterm 3 conflicts (today!?)

Percentiles → 5 pm today

AGENDA / LEARNING OUTCOMES

How to design a filesystem that performs better for small writes?

How do SSDs differ from hard drives?

RECAP

LFS STRATEGY

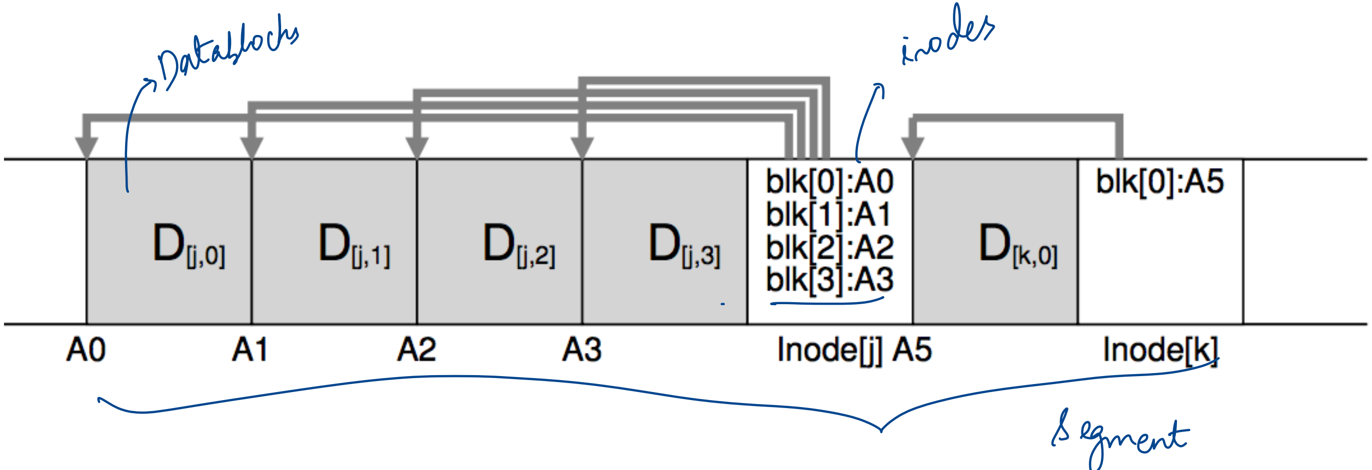
FFS → fast file System → fixed layout → random writes → inode, bitmap etc.

File system buffers writes in main memory until "enough" data

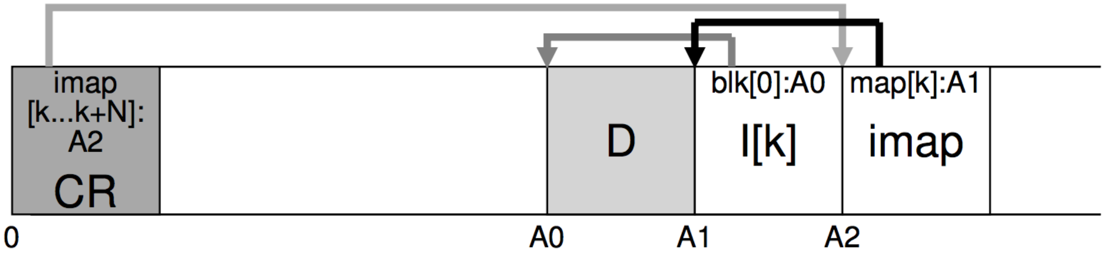
– Enough to get good sequential bandwidth from disk (MB)

Write buffered data sequentially (good) to new **segment** on disk

Never overwrite old info: old copies left behind



READING IN LFS



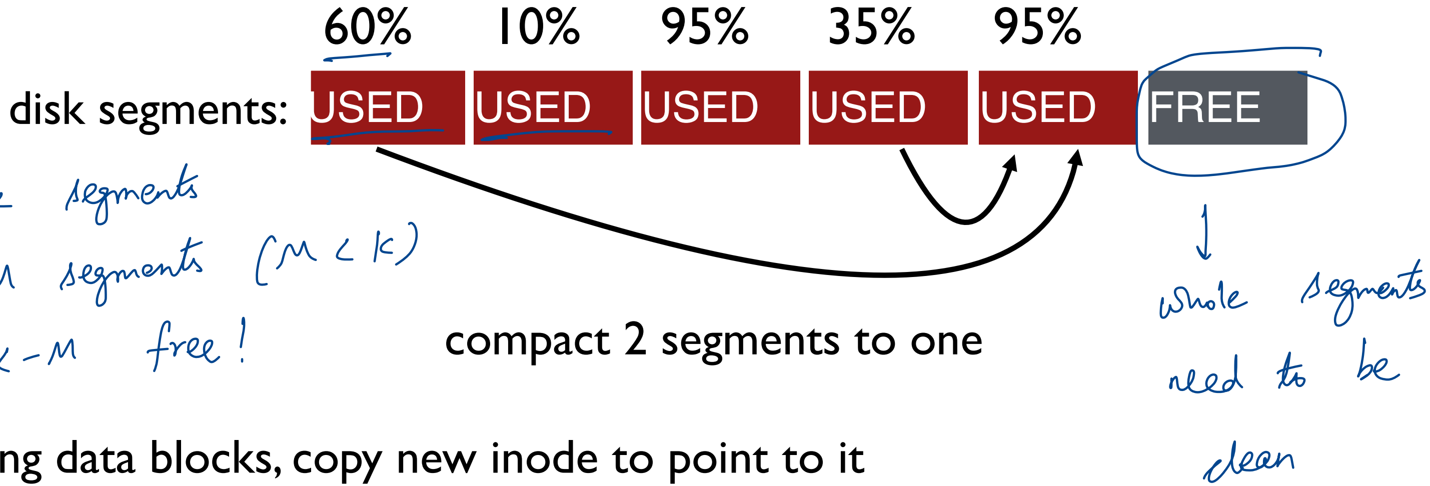
1. Read the Checkpoint region
2. Read all imap parts, cache in mem
3. To read a file:
   1. Lookup inode location in imap
   2. Read inode
   3. Read the file block

new data structure
ptrs to disk location
with inodes

imap cached in memory

GARBAGE COLLECTION

you want to free up space used by older versions (data blocks, inodes)



Pick any k segments
get M segments ($M < k$)
$\Rightarrow$ k - M free!

compact 2 segments to one

When moving data blocks, copy new inode to point to it
When move inode, update imap to point to it

SEGMENT SUMMARY

Inode

Is an inode the latest version?

Check imap to see if this inode is pointed to

Fast!

imap

inode → disk

5 → 25

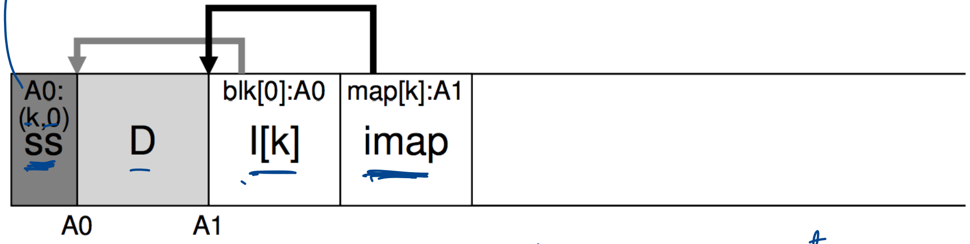
Data

```
(N, T) = SegmentSummary[A];

inode = Read(imap[N]);

if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

inode and offset for the data block



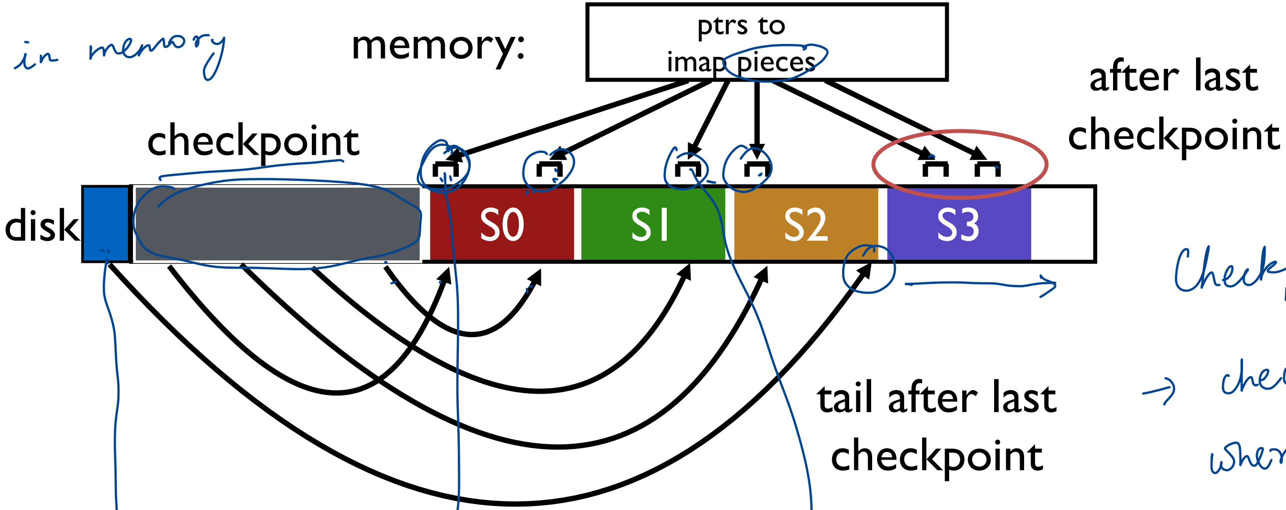
Inode: Alive → Copy this to new segment
update imap to point new segment

Dead

CRASH RECOVERY

imap : inode → disk loc
1 → 24
2 → 48
7 → 73
⋮

imap on crash, recovery
↳ cached in memory



Slow, Simple
→ Scan all segments

segment tail

(2, 48)

(7, 73)

Checkpoint region →
→ checkpointing where imap ptrs are located

CHECKPOINT SUMMARY

Checkpoint occasionally (e.g., every 30s)

↳ timeout

write to specific location on disk

Upon recovery:

- read checkpoint to find most imap pointers and segment tail
- find rest of imap pointers by reading past tail → simple but only for segments after the tail

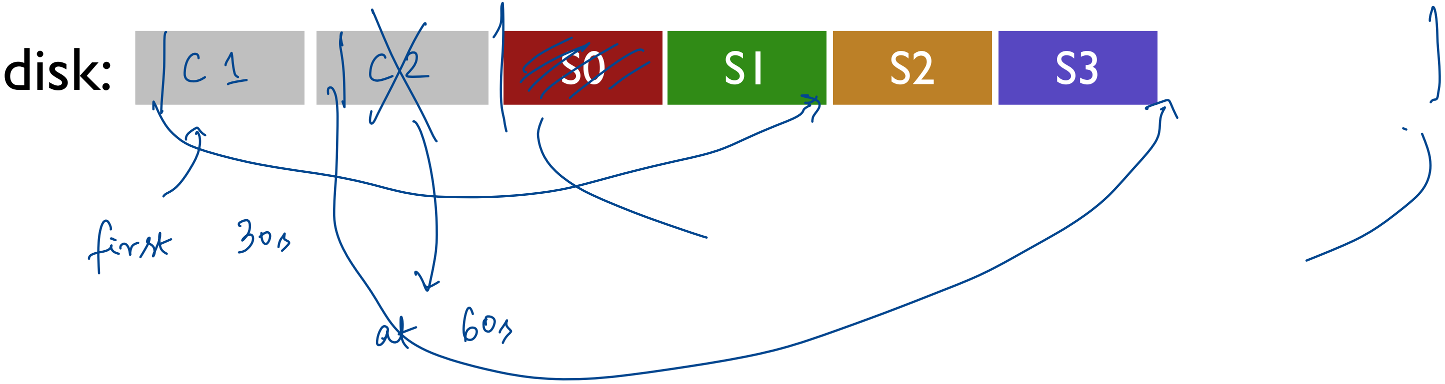
What if crash during checkpoint?

CHECKPOINT STRATEGY

Have two checkpoint regions

Only overwrite one checkpoint at a time

Use checksum/timestamps to identify newest checkpoint



QUIZ 30

https://tinyurl.com/cs537-sp23-quiz30

```
block 100: [("." 0), (".." 0), ("foo" 1)]     // a data block
block 101: [size=1,ptr=100,type=d]            // an inode
block 102: [size=0,ptr=-,type=r]              // an inode
block 103: [imap: 0->101,1->102]              // a piece of the imap
```

file

inode 1 → 102

empty   most recent version

creat ("/foo")

```
block 104: [SOME DATA]                               // a data block
block 105: [SOME DATA]                               // a data block
block 106: [size=2,ptr=104,ptr=105,type=r]           // an inode
block 107: [imap: 0->101,1->106]                     // a piece of the imap
```

updated

write ("/foo", Two data blocks)

LFS VS FFS

File System Logging Versus Clustering: A Performance Comparison

Margo Seltzer, Keith A. Smith
Harvard University

Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan
University of California, Berkeley

→ FFS better → ext2, ext3

↓

A Critique of Seltzer's LFS Measurements

John Ousterhout / john.ousterhout@scriptics.com

Until ... SSDs enter the picture

SSDS

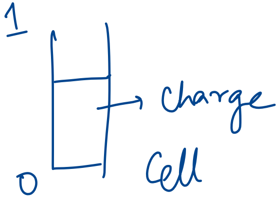
NAND FLASH

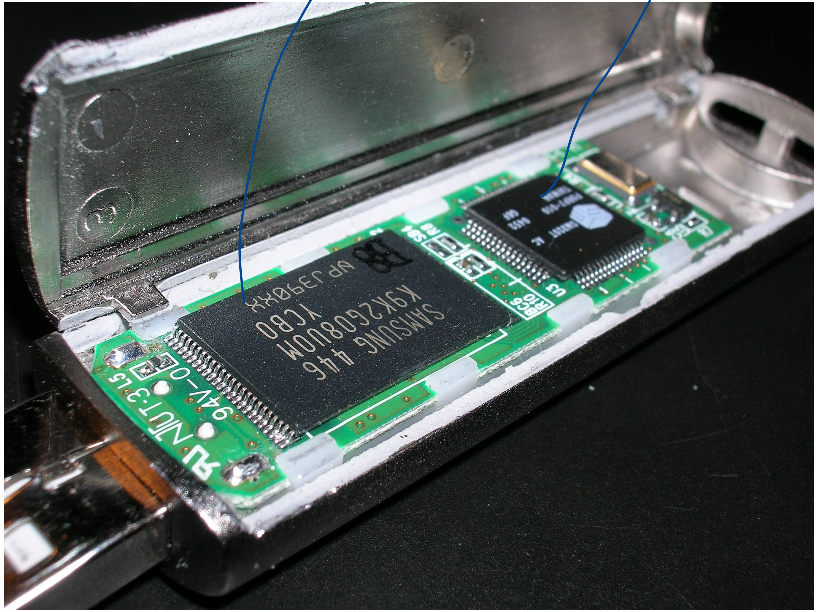
Single Level Cell (SLC) = 1 bit per cell
(faster, more reliable)

Multi Level Cell (MLC) = 2 bits per cell
(slower, less reliable) → 00, 01, 10, 11

density → capacity

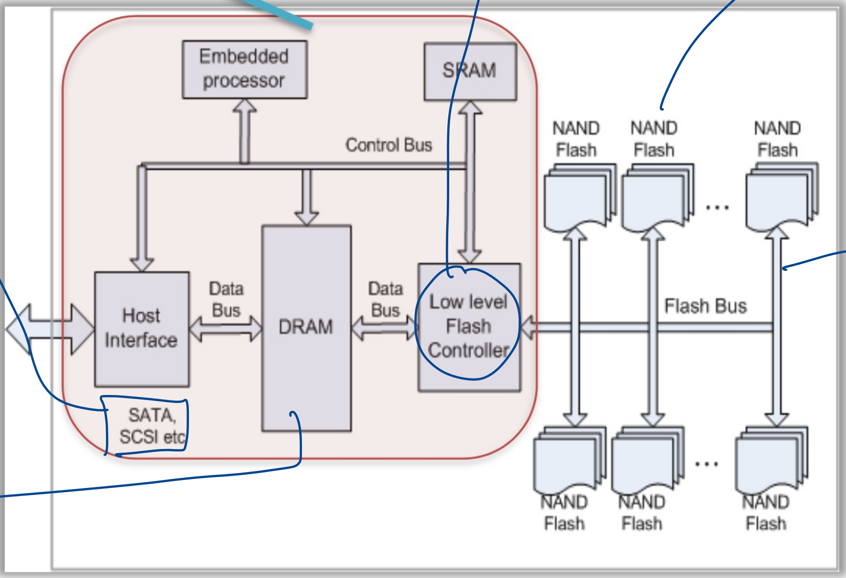
Triple Level Cell (TLC) = 4 bits per cell
(even more so)





SSD STRUCTURE

Flash Translation Layer
(Proprietary firmware)



Simplified block diagram of an SSD

firmware → resides on device / device driver

array

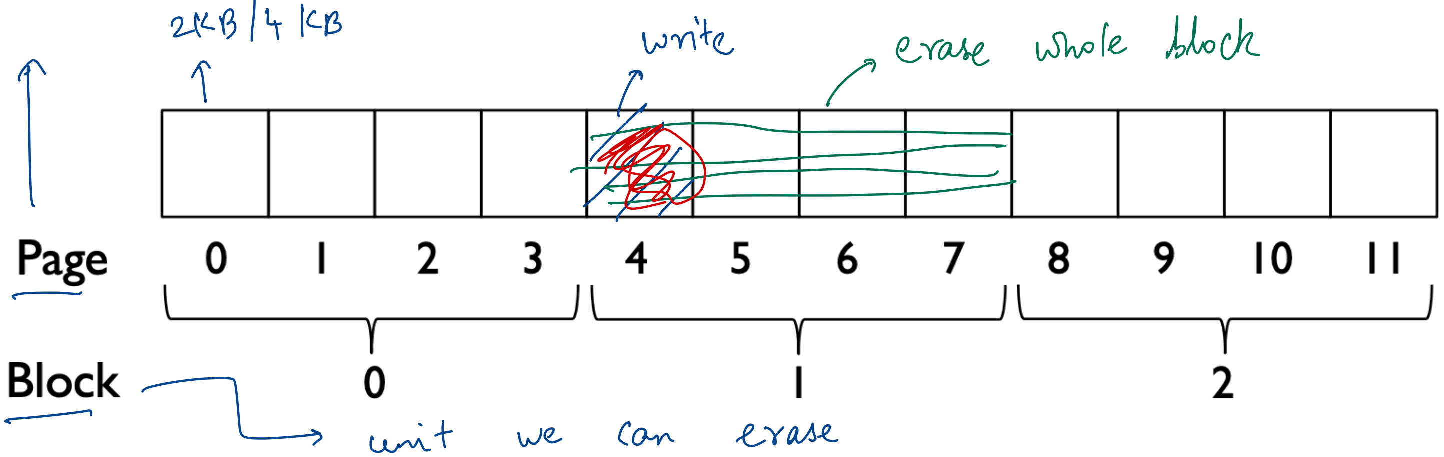
Same interface as disk drives

DRAM

caching metadata

SSD PROPERTIES

Unit read or write



Page ~ 4KB,
Block ~ 128 KB
or 256 KB

Read

→ read a single page → random accesses.

READ 0
≤ return the page to you

Write → Erase block that has this page

write to the page

Block 1 - erased

prob. of failure increases from "wear"

Failures: Block likely to fail after a certain number of erases (~10000 for MLC flash, ~100,000 for SLC flash)

SSD OPERATIONS

Read a page: Retrieve contents of entire page (e.g., 4 KB)

- Cost: 25—75 microseconds
- Independent of page number, prior request offsets

Erase a block: Resets each page in the block to all 1s

- Cost: 1.5 to 4.5 milliseconds
- Much more expensive than reading!
- Allows each page to be written

Program (i.e., write) a page: Change selected 1s to 0s

- Cost is 200 to1400 microseconds
- Faster than erasing a block, but slower than reading a page

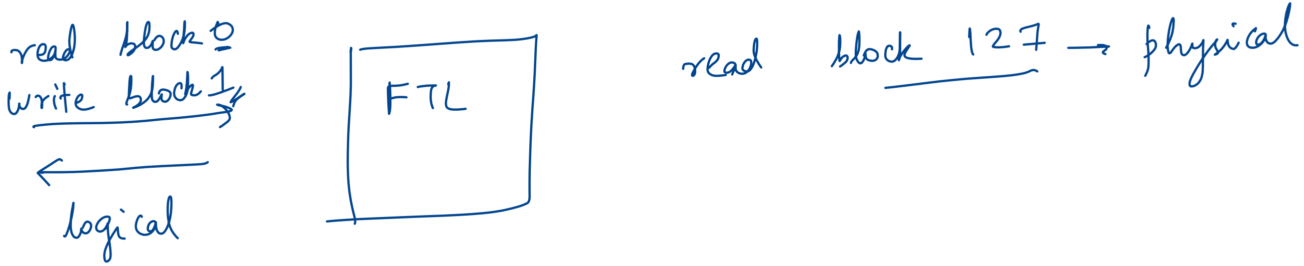
FLASH TRANSLATION LAYER

1. Translate reads/writes to logical blocks into reads/erases/programs on physical

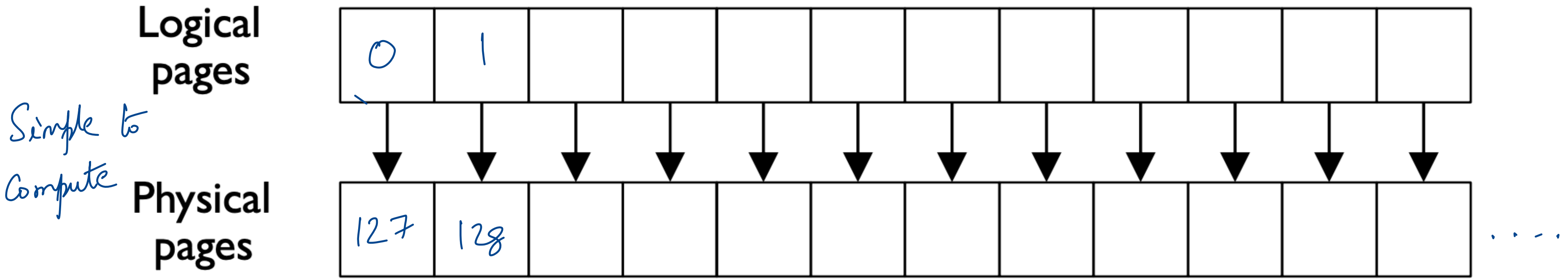
2. Reduce write amplification (extra copying needed to deal with block-level erases)

3. Implement wear leveling (distribute writes equally to all blocks)

Typically implemented in hardware in the SSD, but in software for some SSDs



FTL: DIRECT MAPPING



Cons?

- write (page 0) → erase on block 127
  write (page 0) ⇒ wear is very high

write amplification
- each read entire block, erase, write back pages

FTL: LOG-BASED MAPPING

Idea: Treat the physical blocks like a log. Modifications to pages go to end of the log

logical physical

Table: 100 → 0 , 154 → 1   Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a1	c5											Chip
State:	V	~~E~~ V	E	E	i	i	i	i	i	i	i	i	

V → valid

E → ready to be written

i → need to be erased before write

all blocks are used for writes and writes are spread out

FTL: LOG-STRUCTURED ADVANTAGES

Avoids expensive read-modify-write behavior → minimizes write amplification

Better wear levelling: writes get spread across pages, even if there is spatial locality in writes at logical level

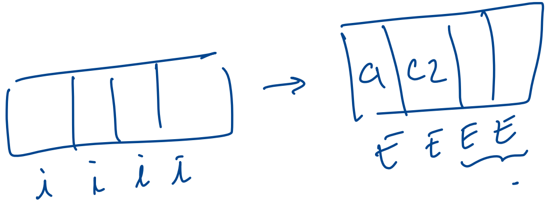
Challenges? Garbage!

GARBAGE COLLECTION

Table: 100 → 0 101 → 1 2000 → 2 2001 → 3 Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	a1	a2	b1	b2									Chip
State:	V	V	V	V	i	i	i	i	i	i	i	i	

Before



write 100 , c1 → new operations
write 101 , c2

Table: 100 → 4 101 → 5 2000 → 2 2001 → 3 Memory

Block:	0				1				2				
Page:	00	01	02	03	04	05	06	07	08	09	10	11	Flash
Content:	~~a1~~	~~a2~~	b1	b2	c1	c2							Chip
State:	V	V	V	V	V	V	E	E	i	i	i	i	

garbage

Program a page in state E
↓
valid

GARBAGE COLLECTION

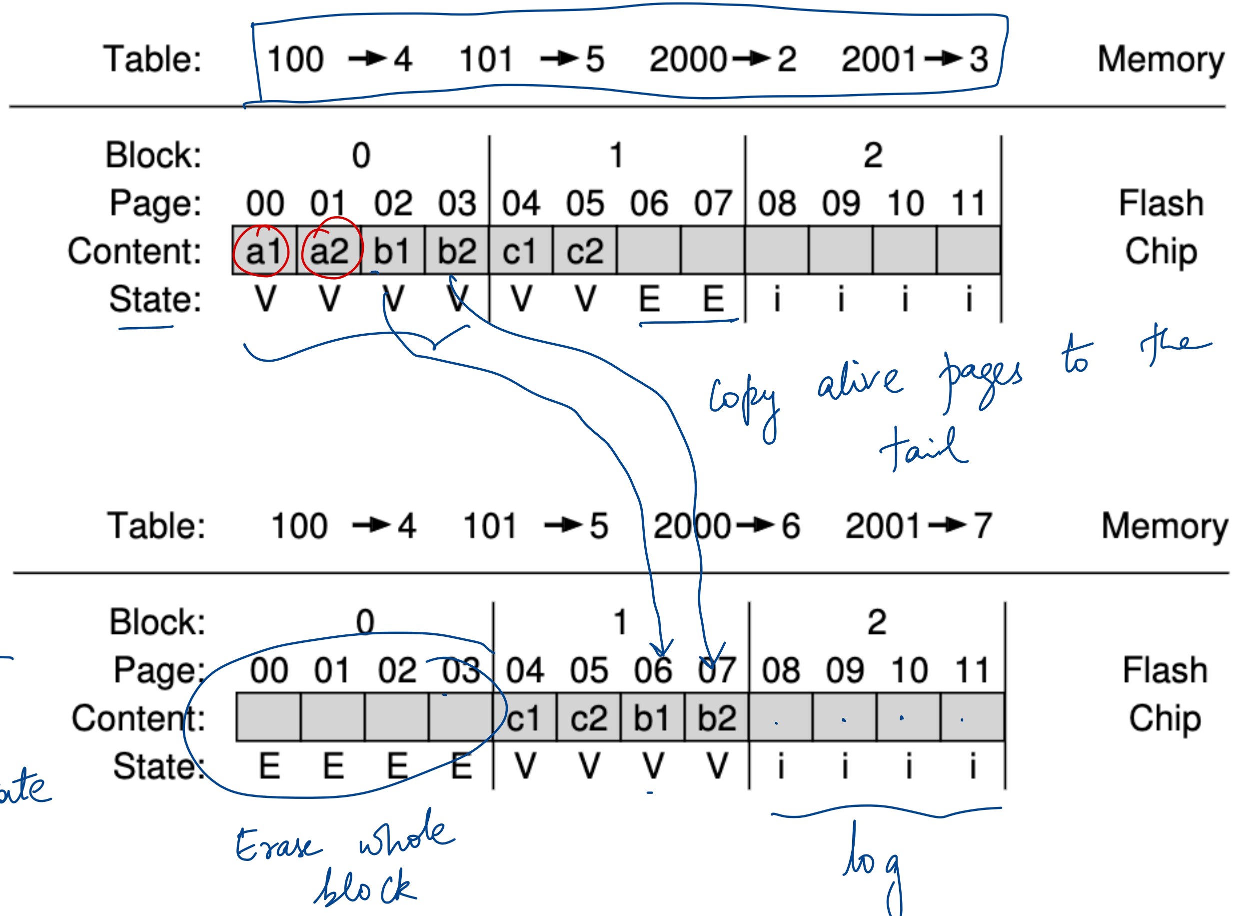
Steps:

Read all pages in physical block

Write out the alive entries to the end of the log

Erase block (freeing it for later use)

Start invalid state
→ Erase block
→ all pages E state



OVERHEADS

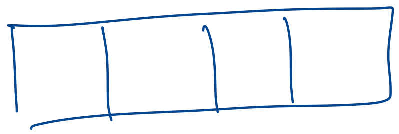
Garbage collection requires extra read+write traffic

Overprovisioning makes GC less painful

- SSD exposes logical space that is smaller than the physical space
- By keeping extra, "hidden" pages around, the SSD tries to defer GC to a background task (thus removing GC from critical path of a write)

Occasionally shuffle live (i.e., non-garbage) blocks that never get overwritten

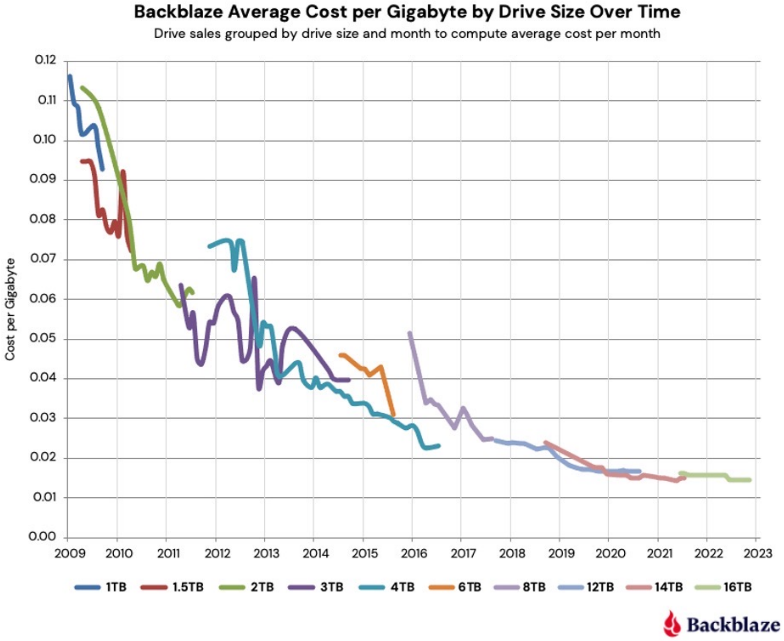
- Enforces wear levelling



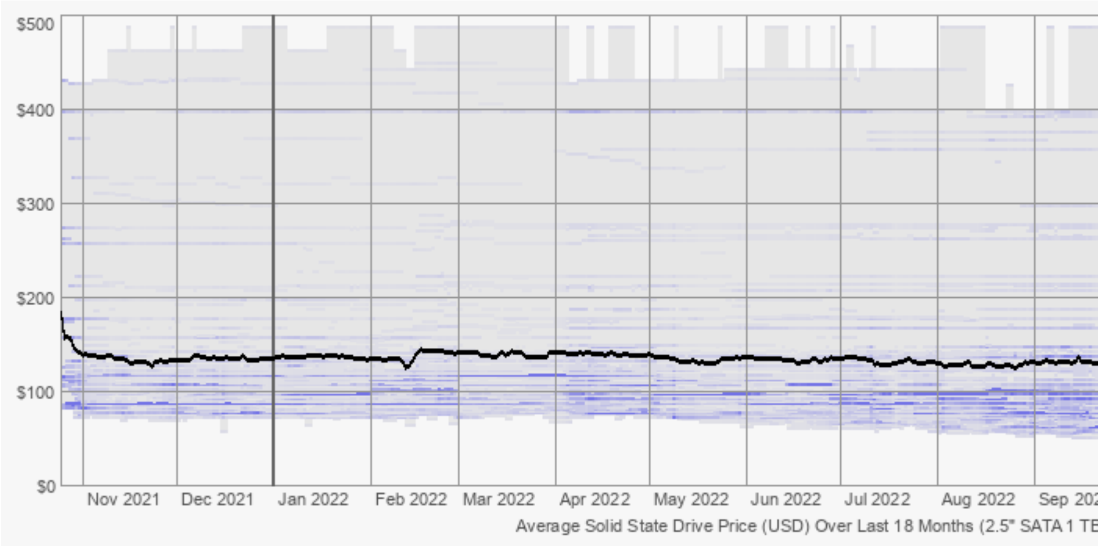
OVERALL PERFORMANCE

Device	Random Reads (MB/s)	Random Writes (MB/s)	Sequential Reads (MB/s)	Sequential Writes (MB/s)
Samsung 840 Pro SSD	103	287	421	384
Seagate 600 SSD	84	252	424	374
Intel SSD 335 SSD	39	222	344	354
Seagate Savvio 15K.3 HDD	2	2	223	223

COST?



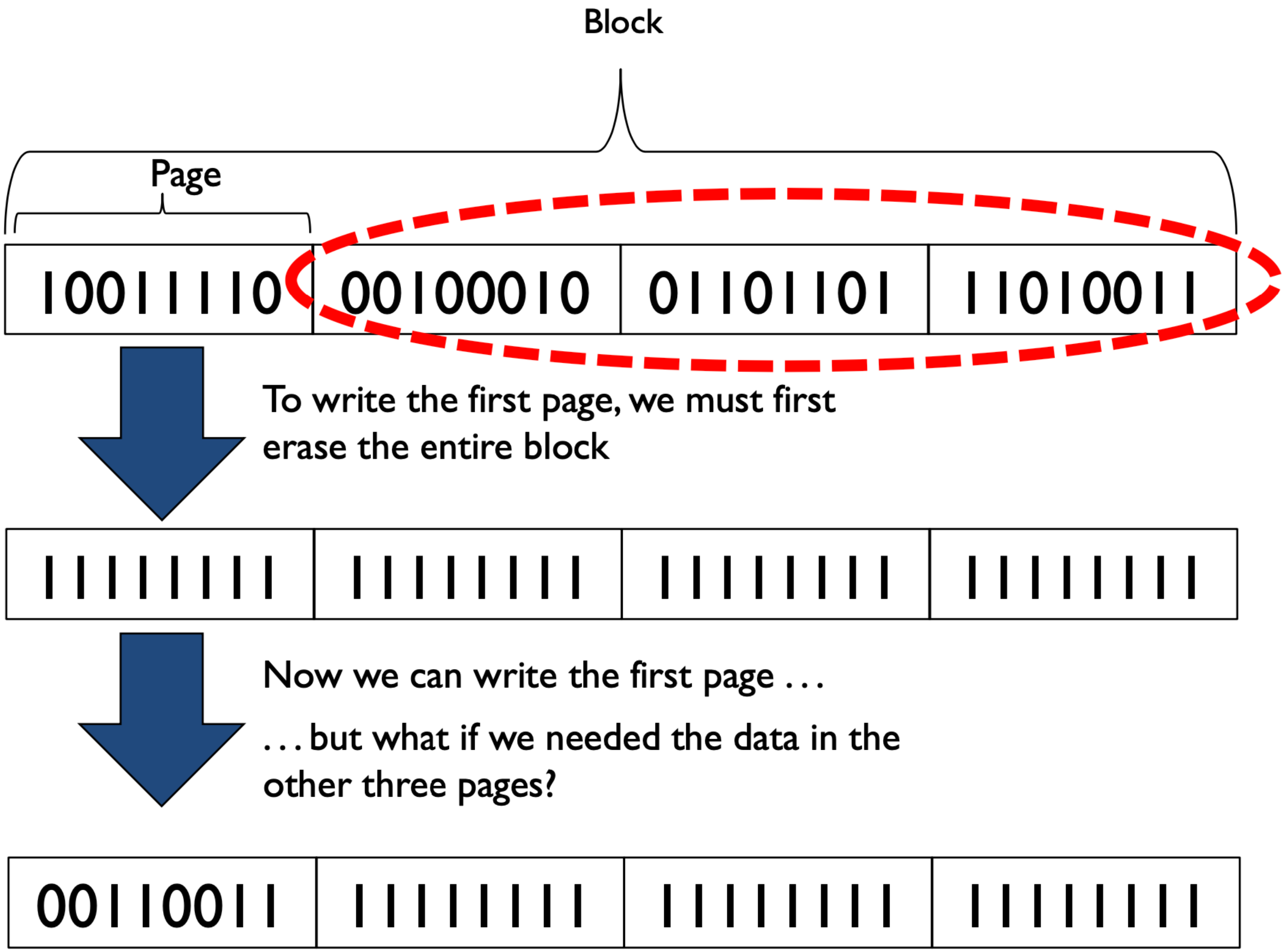
~1.5 cents / GB



1TB ~ $150 on average
~15 cents / GB

NEXT STEPS

Next class: Distributed Systems!


Block
Page
10011110
00100010
01101101
11010011
To write the first page, we must first erase the entire block
11111111
11111111
11111111
11111111
Now we can write the first page . . .
. . . but what if we needed the data in the other three pages?
00110011
11111111
11111111
11111111