Lecture 23: Reliable Storage

CS 105 December 5, 2019

File System Goals

- **Persistence:** maintain/update user data + internal data structures on persistent storage devices
- **Flexibility:** need to support diverse file types and workloads
- **Performance:** despite limitations of disks
- **Reliability:** must store data for long periods of time despite OS crashes or hardware malfunctions

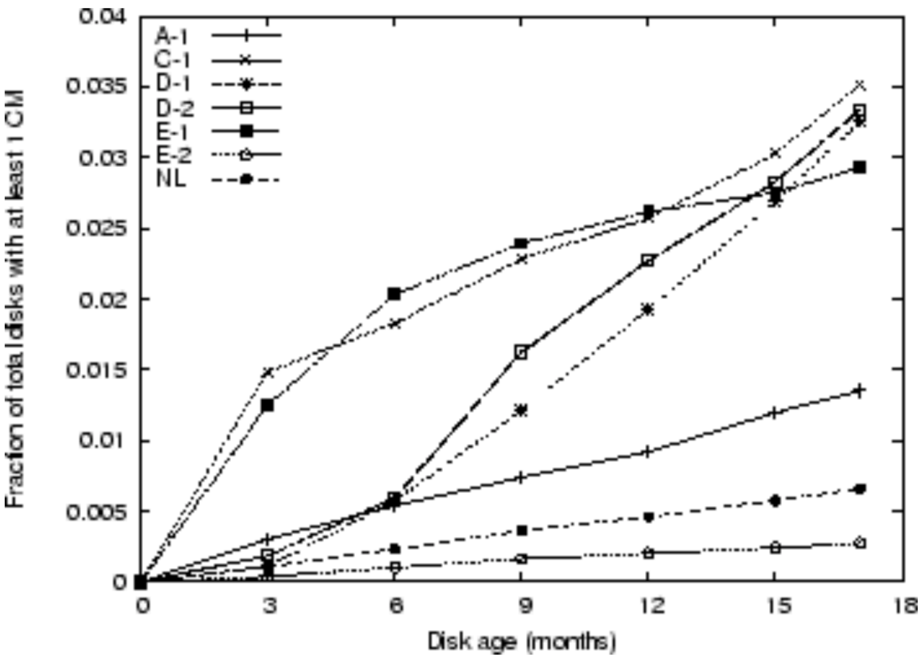
Types of Failures

- Isolated Disk Sectors
  - **Transient:** data corrupted but new data can be successfully written to / read from sector
  - **Permanent:** physical malfunction (magnetic coating, scratches, contaminants)

- Full Disk Failure
  - Damage to disk head, electronic failure, wear out

Data Corruption

- data corruption can be caused by write interference, head height, leaked charge, cosmic rays, etc.
- approximately one sector will be corrupted per 10^14 bits read (about a 2% chance if you read a 2TB disk)



Checksums

- a checksum is the result of a function that takes a chunk of data (e.g., a 4KB block) and returns a short summary (e.g., 4 or 8 bytes)
- File systems can store checksums for metadata and/or file contents

- Example:
  - xor
  - addition
  - Fletcher check-sum
  - cyclic redundancy check (CRC)

XOR-based Checksum

- Consider a 16-byte data block

365e	c4cd	ba14	8a92	ecef	2c3a	40be	f666

- Represented in binary, we get

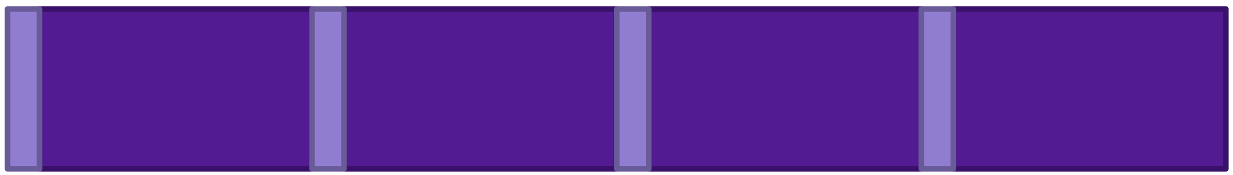
00110110	01011110	11000100	11001101
10111010	00010100	10001010	10010010
11101100	11101111	00101100	00111010
01000000	10111110	11110110	01100110

- We then perform an XOR over each column to compute the checksum

00100000	00011011	10010100	00000011	= 0x201b9403

Using Checksums

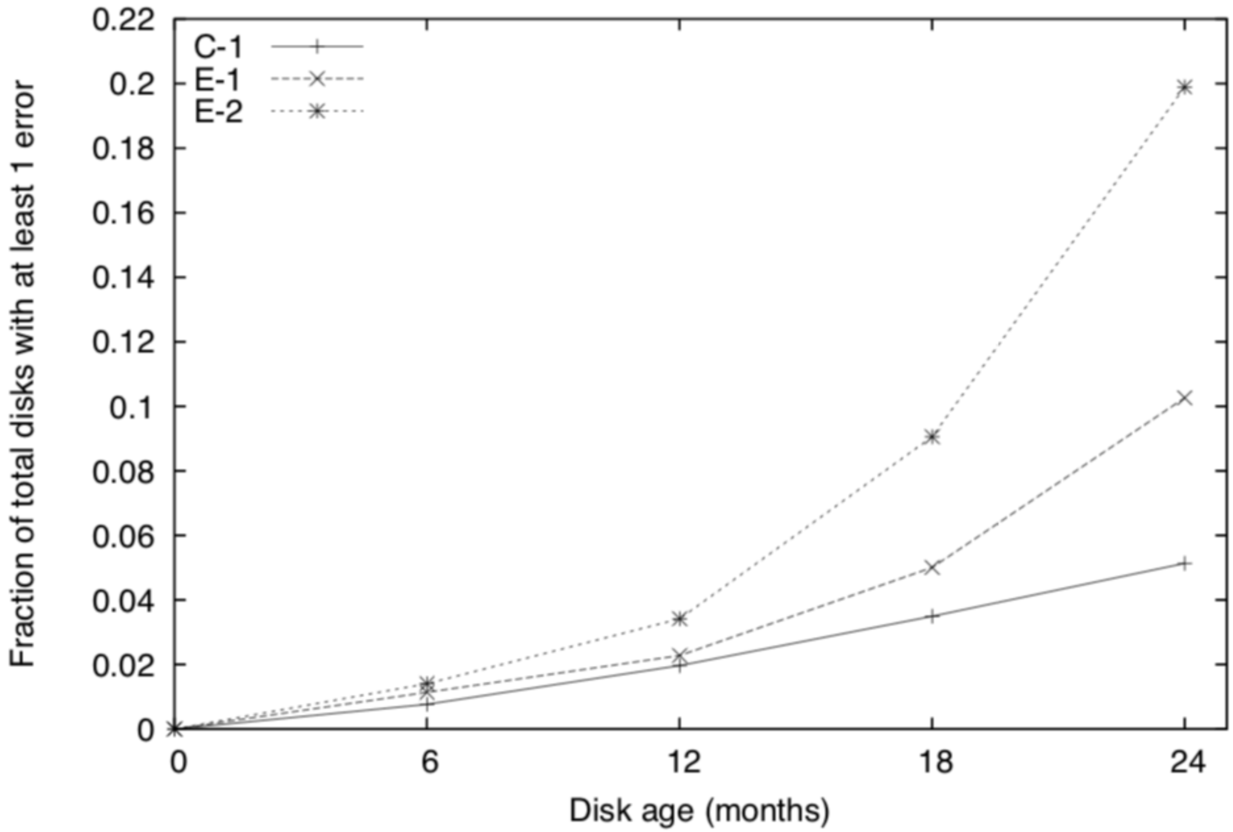
- datablocks are stored with checksums



- When reading a datablock D from disk, the OS also reads its stored checksum $C_S(D)$. It then computes the checksum of the datablock $C_C(D)$ and check whether $C_C(D) == C_S(D)$.

Latent-Sector Errors

- latent-sector errors arise when a disk sector (or group of sectors) has been damaged in some way
- Example: head crash



Error Correcting Codes

- an error-correcting code is a redundant encoding of data that allows information to be recovered from a corrupted copy
- used by disks to automatically correct for disk errors
- balances storage overhead versus error rate

Read-Solomon Coding

- Consider a 16-byte data block d

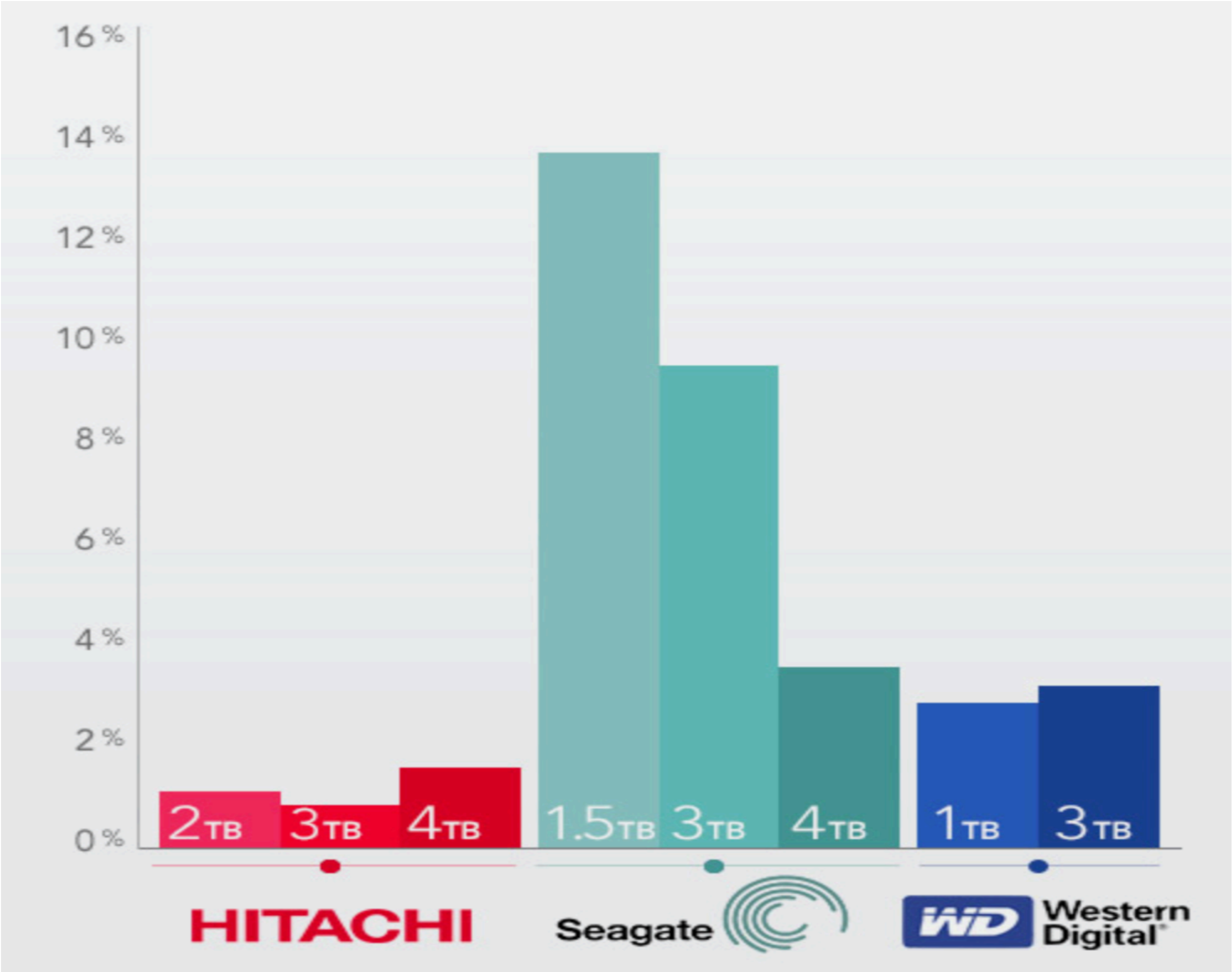
365e	c4cd	ba14	8a92	ecef	2c3a	40be	f666

- It is mapped to a polynomial $p_d(x) = d_0 + d_1 x + \ ... + d_{15} x^{15}$ evaluated modulo 2^8

- The error-correction code is simply p_d evaluated at n different points

Full Disk Error

- Damage to disk head, electronic failure, wear out

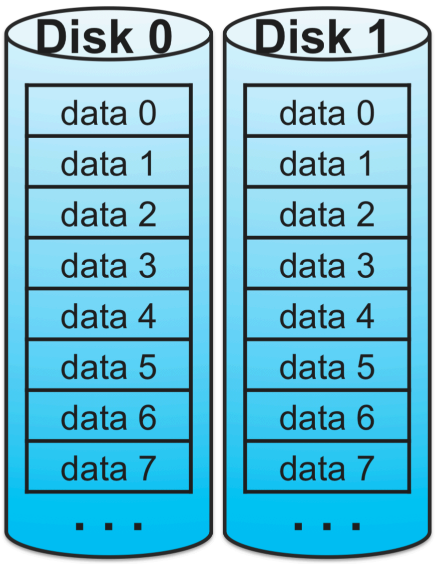


RAID

- a redundant array of inexpensive disks (RAID) is a system that spreads data redundantly across multiple disks in order to tolerate individual disk failures

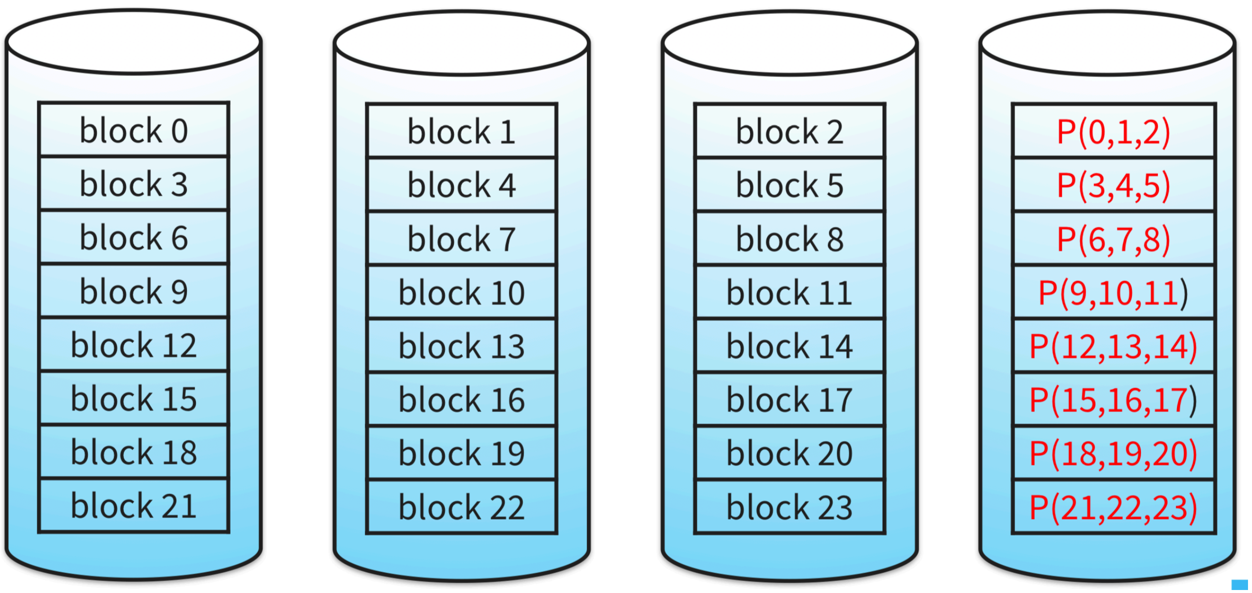
RAID-1: Mirroring

- Each block is stored on 2 separate disks.
- Read either copy
  - If error is detected, read other copy
- write both copies (in parallel)



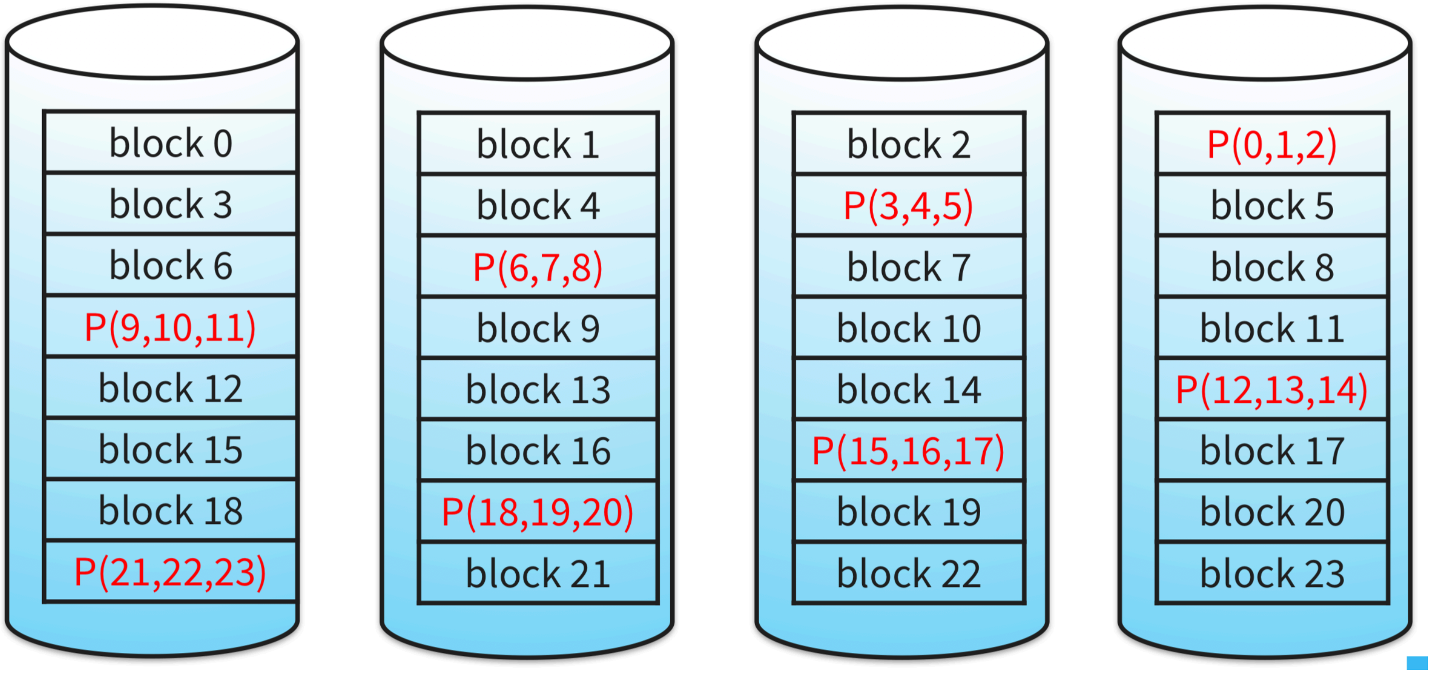
RAID-4: Parity for Errors

- block-level striping with a dedicated parity disk
- RAID-2 and RAID-3 are variants that stripe at the bit and byte levels (not used in practice)
- parity disk becomes the bottleneck



RAID-5: Rotating Parity

- write-load for parity block spread across all disks



Scrubbing

- most data is rarely accessed
- many systems utilize disk scrubbing, that is , periodically reading through every block of the system and checking whether checksums are still valid

Tolerating Crash Failures

- If a processor crashes then only some blocks on a disk might get updated.
  - Data is lost
  - On-disk data structures might become inconsistent.
  - E.g.
    - starting state: A, B
    - update: A -> A’ and B -> B’
    - Possible result when there is a crash: A, B’ or A’, B
- Solutions:
  - Add fsync: programmer forces writes to disk
  - Detect and recover
  - Fault-tolerant disk update protocols

File System Consistency Checks

- fsck (UNIX) & scandisk (Windows)

- observation: writing a new data block involves 3 writes (write data block, update freelist, update inode)

Detection Algorithm for File Blocks:

- Build table with info about each block
  - initially each block is unknown (except superblock)
- Scan through the inodes and the freelist
  - Keep track in the table
  - If block already in table, note error
- Finally, see if all blocks have been visited

Fault-tolerant Disk Update

- Use Journaling (aka) Write-Ahead Logging

- **Idea**: Protocol where performing a **single** disk write causes multiple disk writes to take effect.

- **Implementation**: New on-disk data structure ("journal") with a sequence of blocks containing updates *plus …*

Journal-Update Protocol Step

- write x; write y; write z
- E.g., write to inode, write to freelist (bitmap), write to datablock

- *implemented by:*
  - Append to journal: TxBegin, x, y, z
  - Wait for completion of disk writes.
  - Append to journal: TxEnd
  - Wait for completion of disk write.
  - Write x, y, z to final locations in file system

- Recovery protocol for TxBegin ... TxEnd:
  - if TxEnd present then redo writes to final locations
  - else ignore journal entries following TxBegin