# ECE 350 Real-time Operating Systems

# Lecture 12: Reliable Storage

Prof. Seyed Majid Zahedi

https://ece.uwaterloo.ca/~smzahedi

# Outline

- Problem posed by machine/disk failures
- Transaction concept (ASID properties)
- Transactional file systems
  - Journaling and logging
- Copy-on-write file systems
- Redundant arrays of independent disks (RAID)

# Recall: Important "abilities"

- Reliability: ability of system or component to perform its required functions under stated conditions for specified time
  - System is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Data must survive system crashes, disk crashes, other problems
- Availability: probability that system can accept and process requests
  - Can build highly-available systems, despite unreliable components
    - Involves independence of failures and redundancy
  - Often as "nines" of probability. 99.9% is "3-nines of availability"
- Durability: ability of system to recover data despite faults
  - This idea is fault-tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone

# File System Reliability

- Single logical file operation can involve updates to multiple physical storage blocks
  - inode, indirect block, data block, bitmap, …
  - With sector/page remapping, single update to physical storage block can require multiple (even lower-level) updates to sectors/pages
- What can happen if disk loses power or software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of block may only partially complete
- How do we guarantee consistency regardless of when crash occurs?

# Reliability Take 1: Careful Ordering

- Sequence operations in specific order
  - Careful design to allow sequence to be interrupted safely
- Post-crash recovery
  - Read data structures to see if there were any operations in progress
  - Clean up/finish as needed
- Approach taken by
  - FAT and FFS (`fsck`) to protect file-system structure/metadata
  - Many app-level recovery schemes (e.g., Word, emacs autosaves)

# Append Data to File in FAT

Normal operation

- Allocate data block
- Write data
- Write new MFT entry to point to data block
- Update file tail to point to new MFT entry
- Update access time at head of file

Recovery

- Scan MFT
- If entry is unlinked, delete data block
- If access time is incorrect, update

# Create New File in FFS

Normal operation

- Allocate data block
- Write data block
- Allocate inode
- Write inode block
- Update bitmap of free blocks and inodes
- Update directory with file name → inode number
- Update modify time for directory

Recovery

- Scan inode table
- If any unlinked files (not in any directory), delete or put in lost & found directory
- Compare free block bitmap against inode trees
- Scan directories for missing update/access times
- Recovery time is proportional to storage size

# Some General Rules for Careful Ordering

- Never write a pointer before initializing the block it points to (e.g., indirect block)

- Never reuse resource (e.g., inode, disk block, etc.) before nullifying all pointers to it

- Never clear last pointer to resource before setting new one (e.g., `mv`)

# Careful Ordering: Discussion

- + Works with minimal support from storage drive
- + Works for most multi-step operations

- – Time-consuming recovery after each failure
- – Slow updates due to sync barriers between dependent operations
- – Hard to turn every operation to safely-interruptible sequence of writes
- – Hard to achieve consistency when multiple operations run concurrently

# Reliability Take 2: Transaction Concept

- Transaction is group of operations providing ACID properties
- Atomicity: operations appear to logically happen as group, or not at all
  - At physical level, only single disk/flash write is atomic
- Consistency: transactions maintain data integrity
  - Each transaction moves system from one legal state to another
    - E.g., balance cannot be negative
    - E.g., cannot reschedule meeting on February 30
- Isolation: each transaction is not affected by other in-progress transactions
  - If multiple transactions execute concurrently, for each pair of transactions A and B, it either appears that A executed entirely before B or vice versa
- Durability: if transaction commits, its effects persist despite crashes

# Typical Structure

- Begin transaction – get transaction id
- Do bunch of updates
  - If any fail along the way, roll-back
  - Or, if any conflicts with other transactions, roll-back
- Commit the transaction

# “Classic” Transaction Example

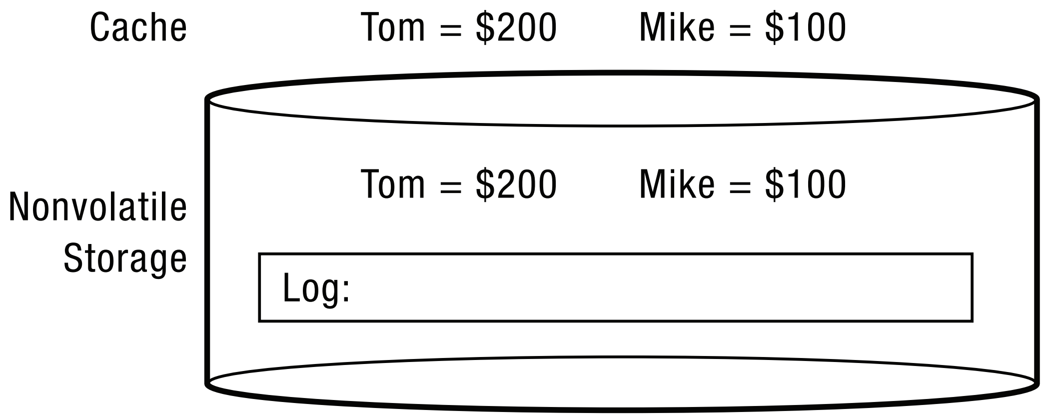
- Transfer $100 from Tom’s account to Mike’s account

```
BEGIN;

UPDATE accounts SET balance = balance - 100.00 WHERE name = ‘Tom';

UPDATE accounts SET balance = balance + 100.00 WHERE name = ‘Mike';

COMMIT;
```

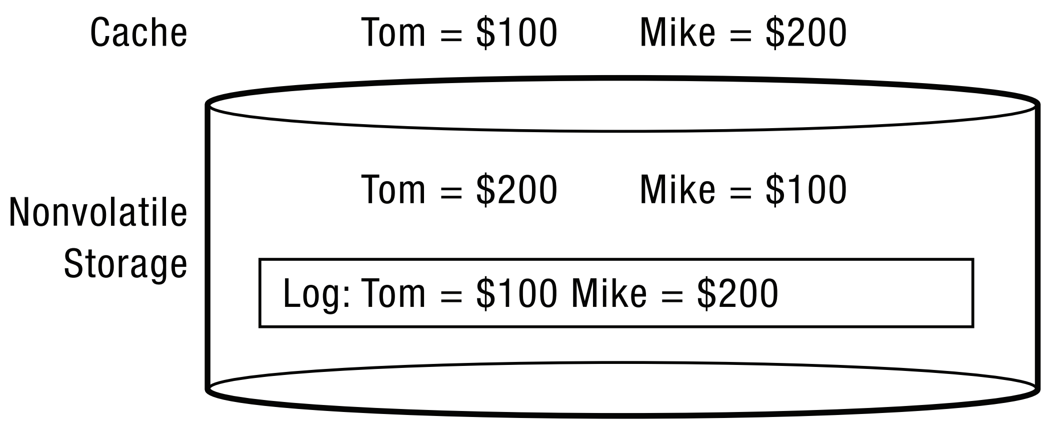
# Atomicity and Durability: Redo Logging

- Prepare: append all updates to log
- Commit: append commit record to log
  - Single disk write to make transaction durable
- Redo: copy all updates to disk
- Garbage collect: reclaim space in log
- Recover: execute recovery routine if system crashes
  - Scan sequentially through log, do the following for each type of record
    - Update record: add to list of updates planned for this transaction
    - Commit record: redo all planned updates for this transaction
    - Roll-back record: discard list of updates planned for this transaction
  - At the end of log, discard updates for transactions without commit record
  - Garbage collect log

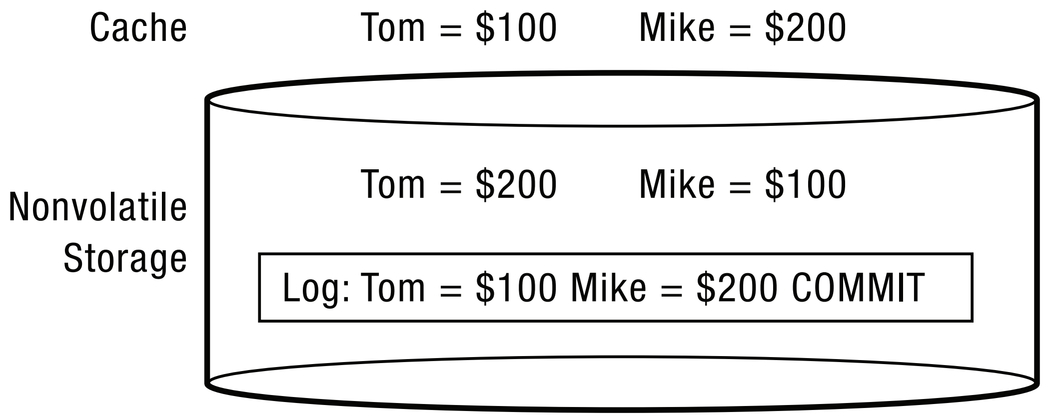
# Redo Logging Example: Before Transaction Start



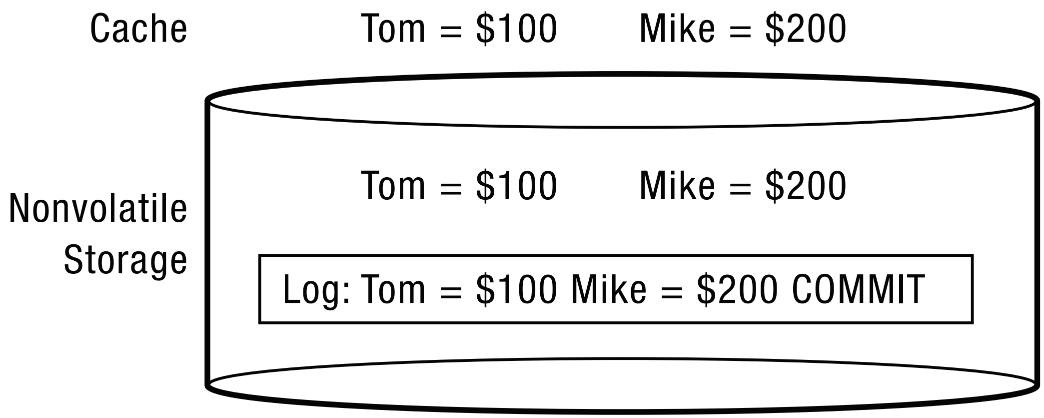
# Redo Logging Example: After Updates Are Logged



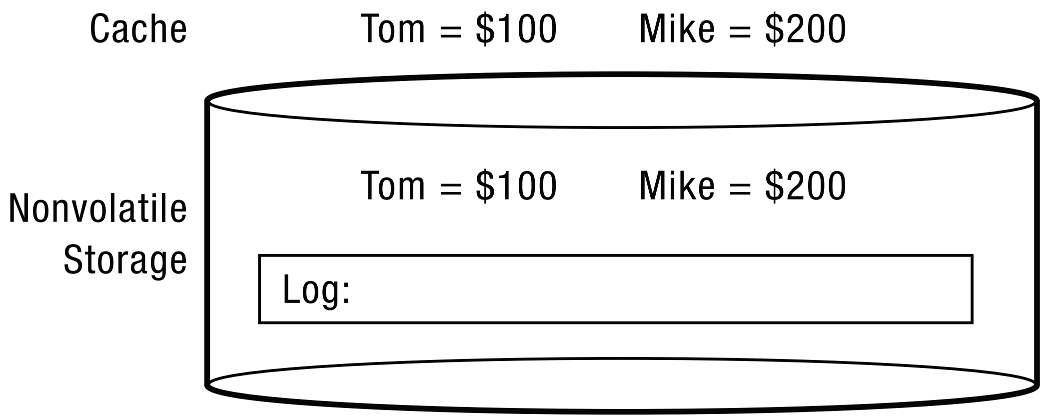
# Redo Logging Example: After Commit Logged



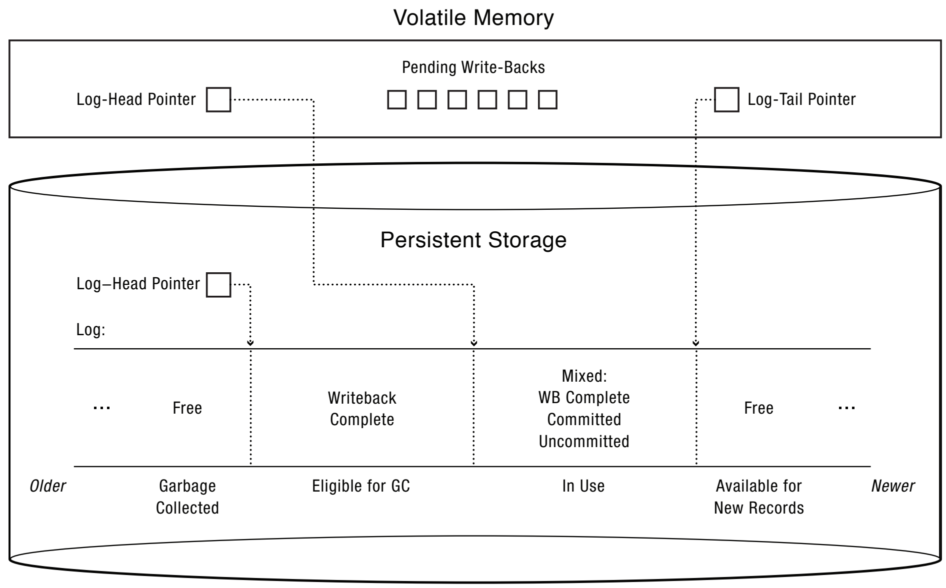
# Redo Logging Example: After Redo



# Redo Logging Example: After Garbage Collection



# Redo Log Implementation




- Volatile memory has pointers to head and tail log
- New transaction records are appended to log's tail and cached in volatile memory
- Redo asynchronously writes pending updates for committed transactions
- Garbage collector periodically advances persistent log-head pointer so that recovery can skip at least some of committed transactions

# Isolation: Two-phase Locking

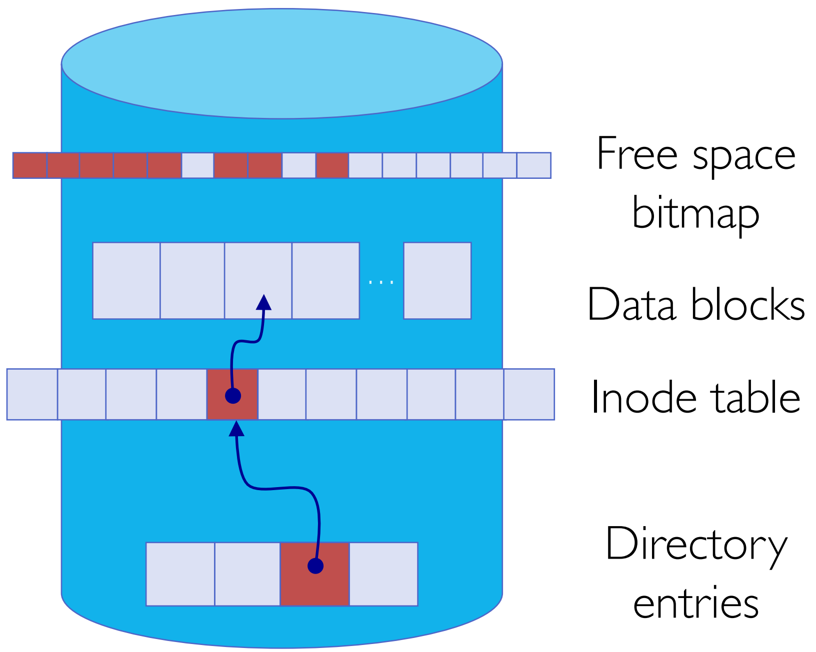
- Expanding phase: locks may be acquired but not released
- Contracting phase: locks may be released but not acquired
- Transactions release locks only AFTER they commit
  - Prevents transactions from seeing results of non-committed ones
  - Could lead to deadlock
- If set of transactions deadlocks, one or more can be forced to roll back, release their locks, and restart at some later time
- Serializability: with two phase locking and redo logging, transactions appear to occur in sequential order

# Transactional File Systems

- Journaling: apply updates to system metadata using transactions, but updates to users' files are done in place
  - E.g., NTFS, Apple HFS+, Linux XFS, JFS,
  - E.g., Ext3 and Ext4 use journaling by default
- Logging: apply all updates (metadata & data) using transactions
  - E.g., Ext3 and Ext4 can be configured to use logging

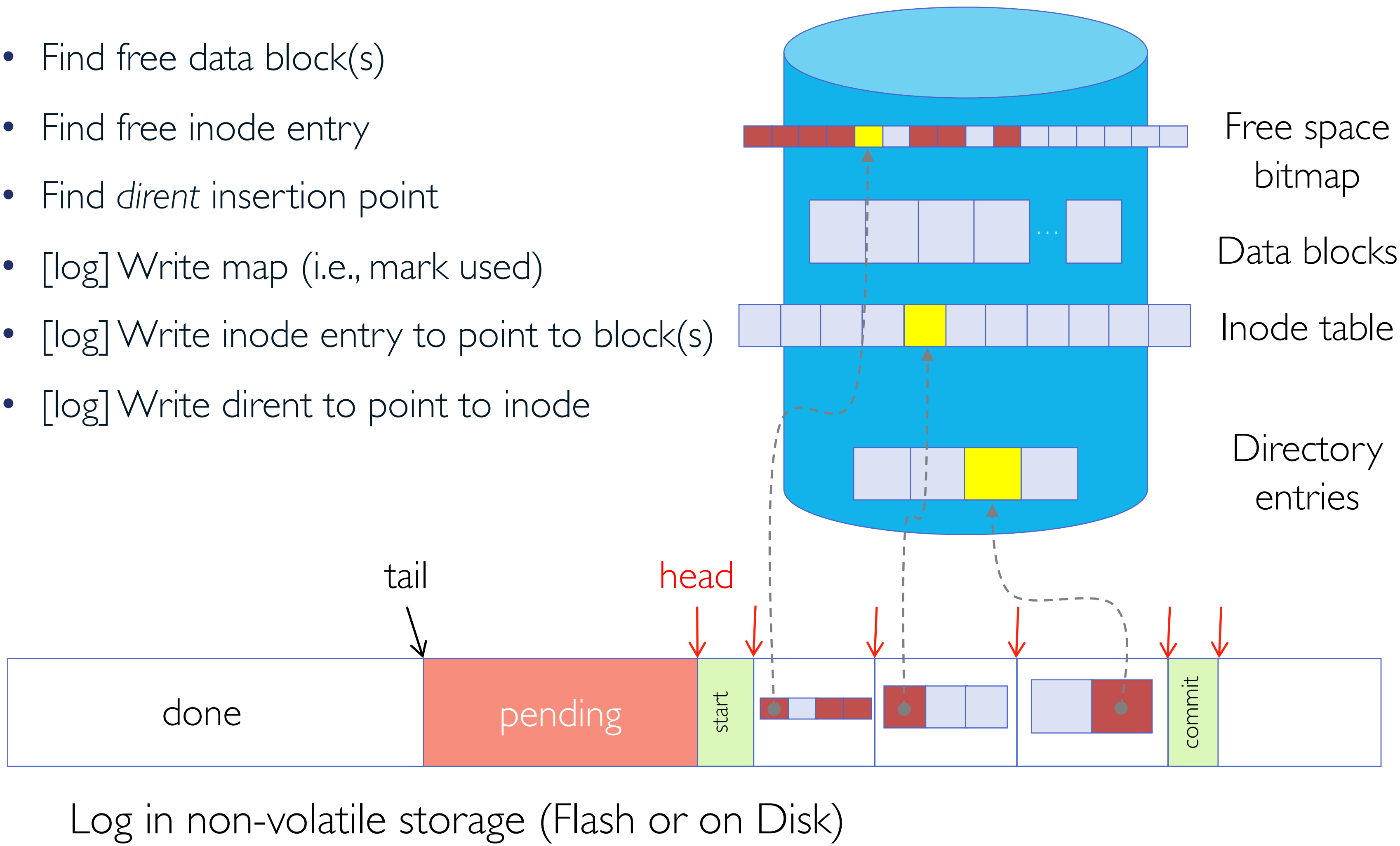
# Example: Creating File

- Find free data block(s)
- Find free inode entry
- Find *dirent* insertion point
- Write map (i.e., mark used)
- Write inode entry to point to block(s)
- Write dirent to point to inode



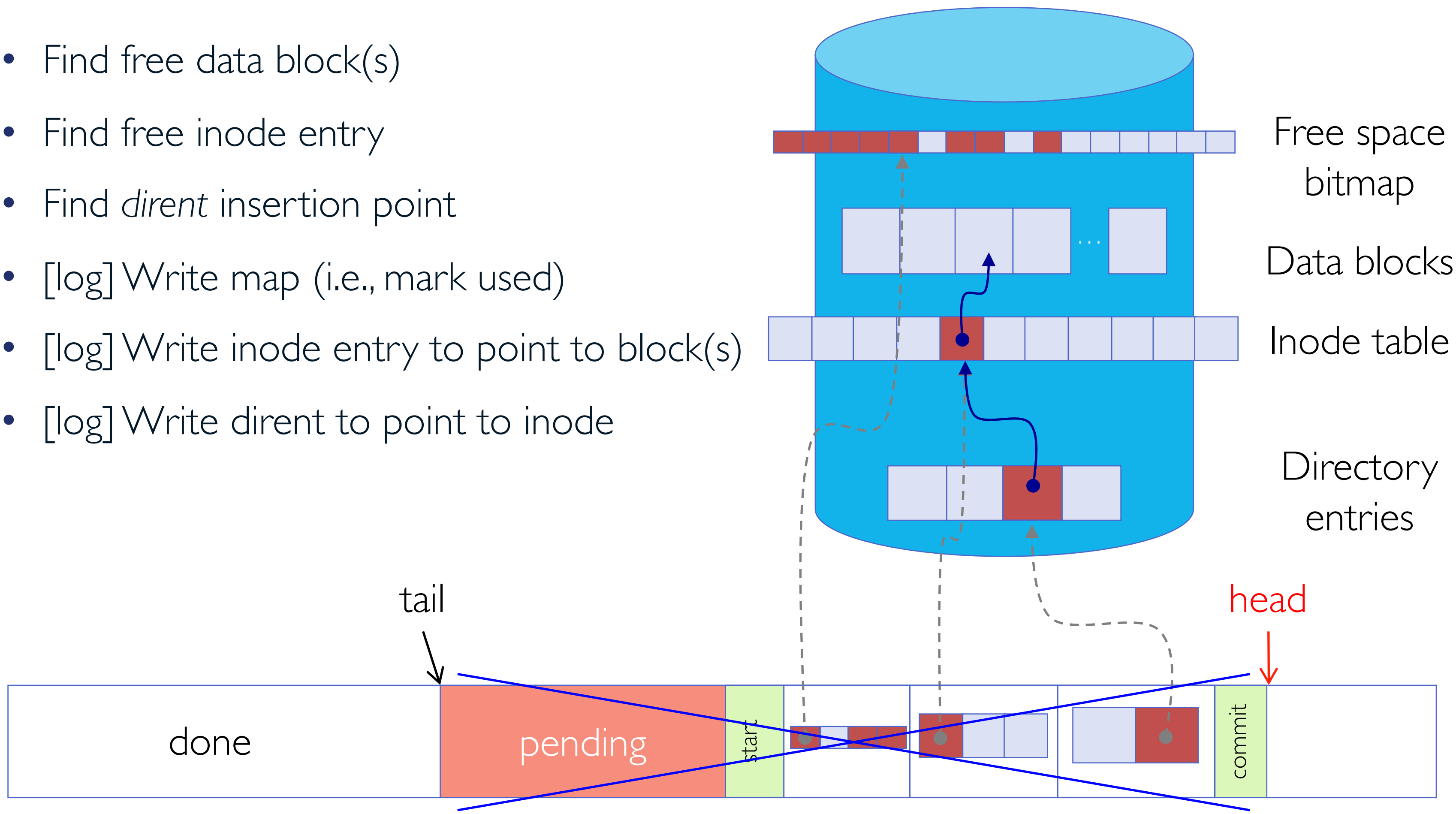
# Example: Creating File (as Transaction)

- Find free data block(s)
- Find free inode entry
- Find *dirent* insertion point
- [log] Write map (i.e., mark used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode



# Example: Creating File (as Transaction)

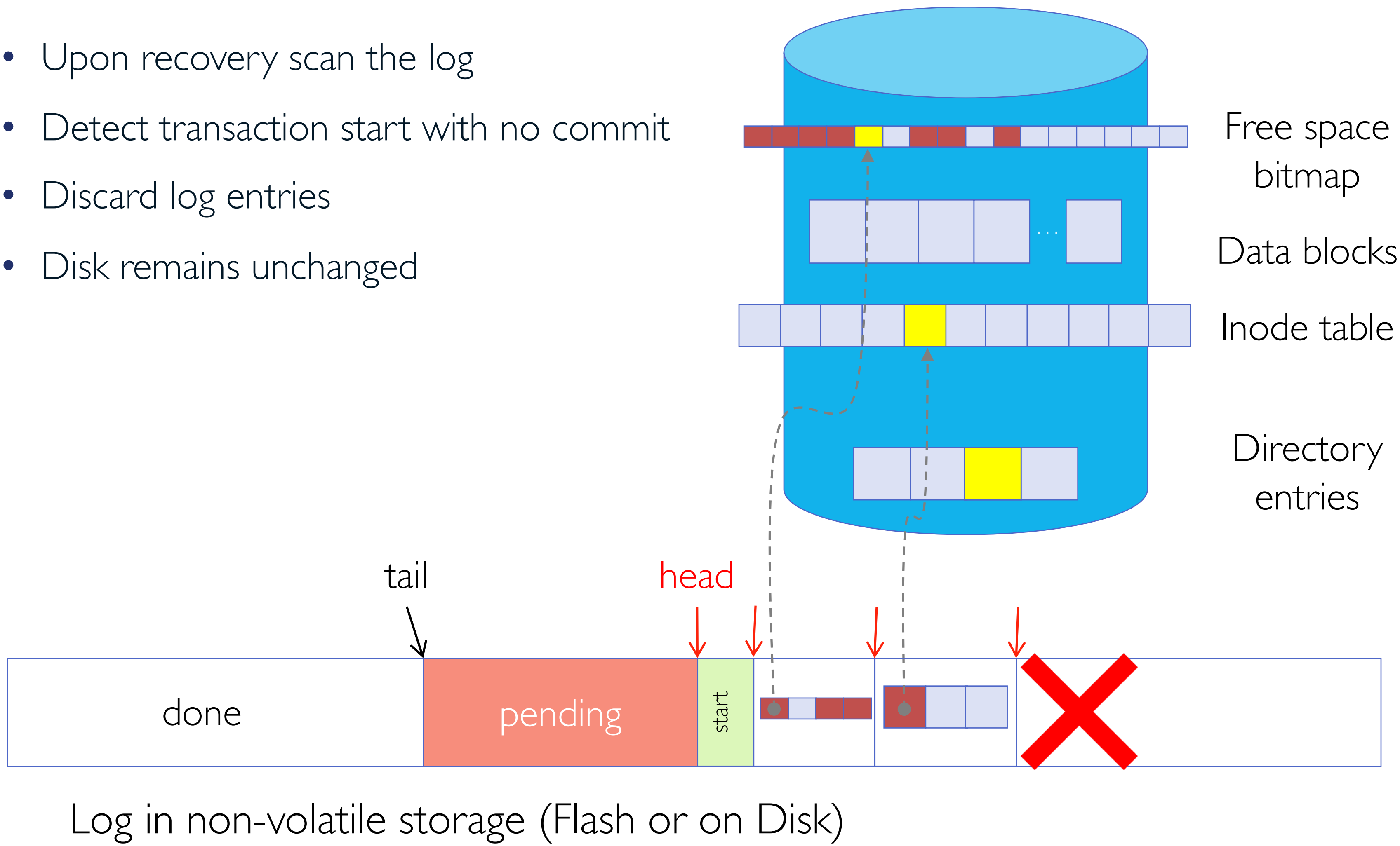
- Find free data block(s)
- Find free inode entry
- Find *dirent* insertion point
- [log] Write map (i.e., mark used)
- [log] Write inode entry to point to block(s)
- [log] Write dirent to point to inode




Log in non-volatile storage (Flash or on Disk)

# Example: Recovery After Failure During Logging

- Upon recovery scan the log
- Detect transaction start with no commit
- Discard log entries
- Disk remains unchanged



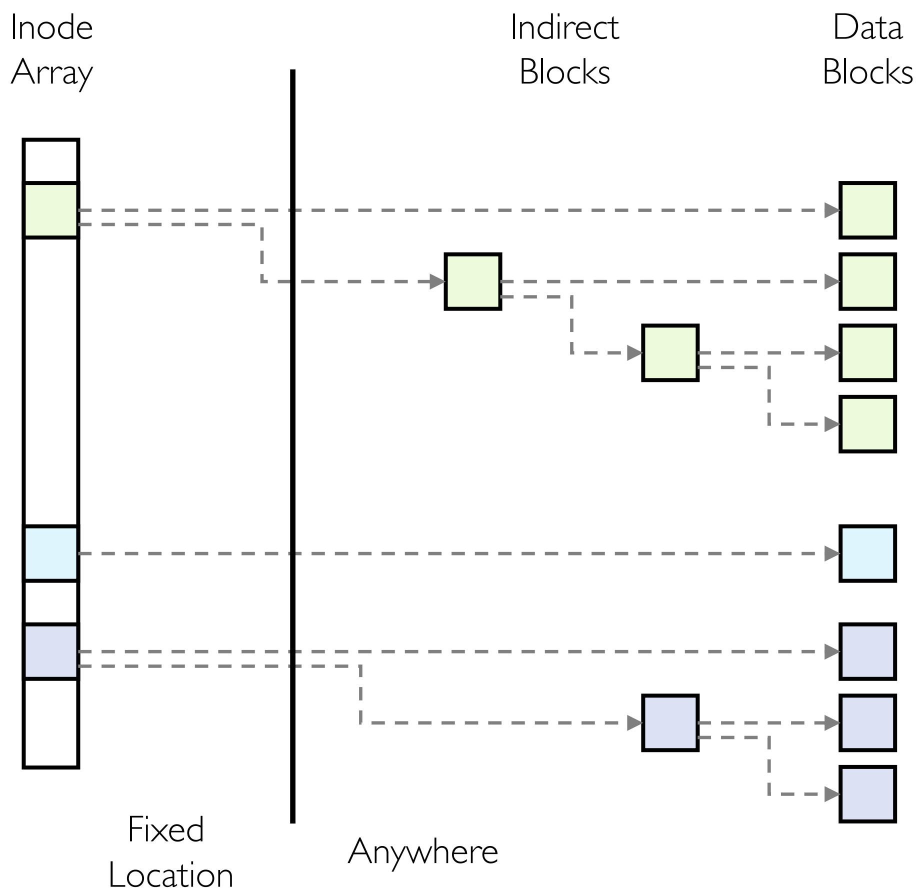
# Copy-on-Write (COW) File System

- For each update, write new version of file system
  - Never update in place
  - Reuse existing unchanged disk blocks

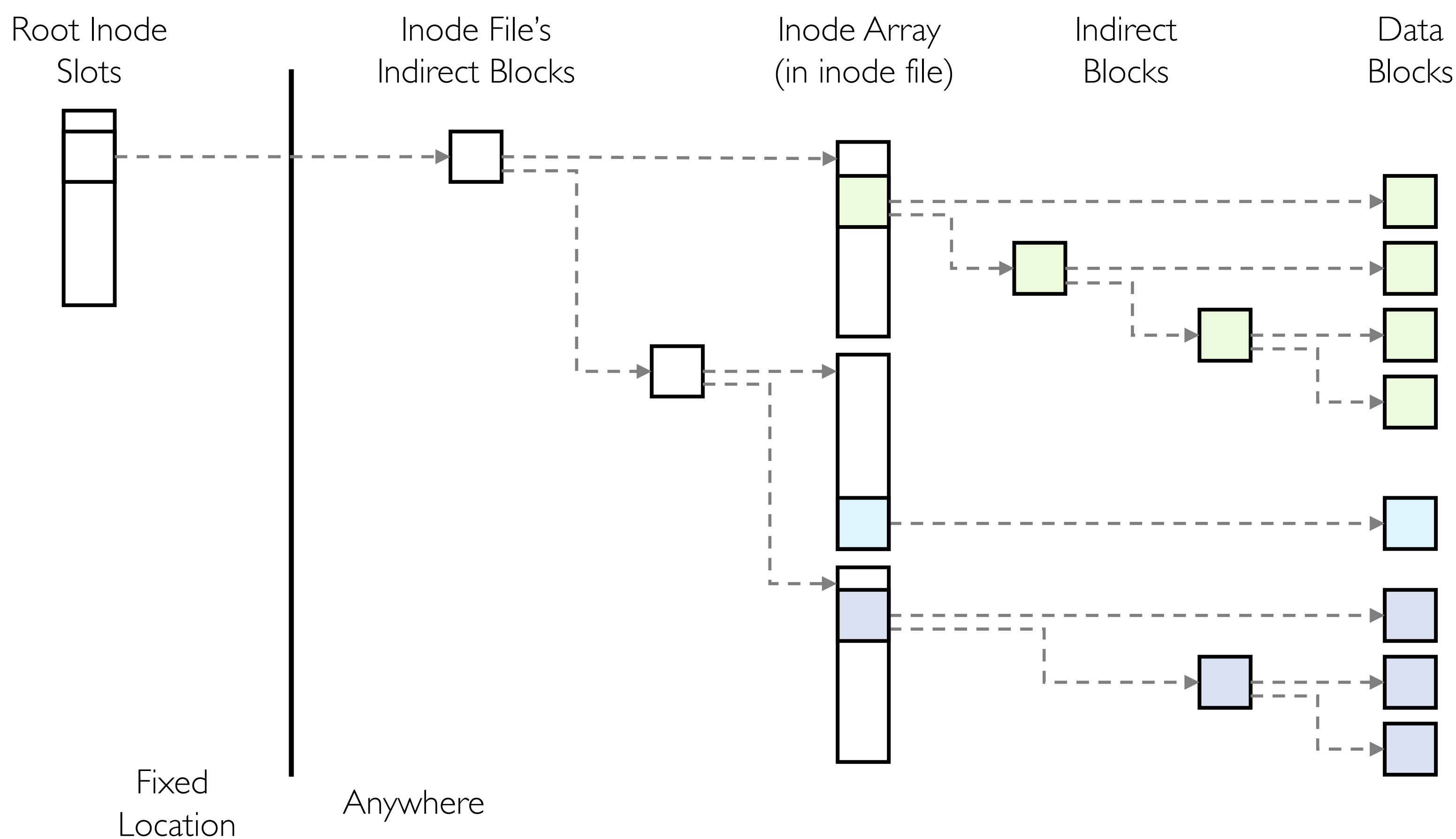
- Seems expensive! But …
  - Updates can be batched
  - Almost all disk writes can occur in parallel

- Approach taken in network file server appliances
  - NetApp's Write Anywhere File Layout (WAFL)
  - ZFS (Sun/Oracle) and OpenZFS

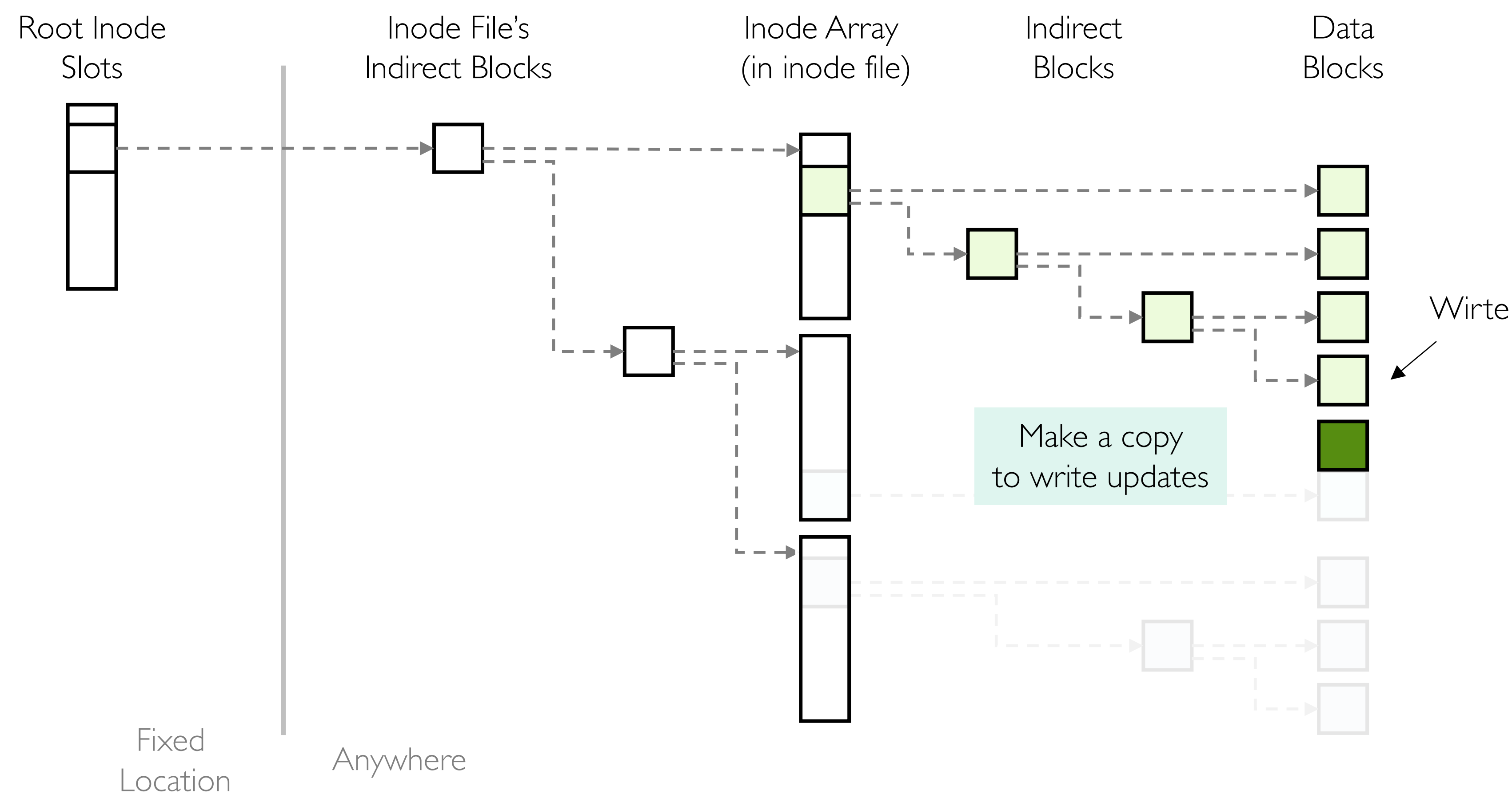
# Recall: Traditional, Update-in-place File System



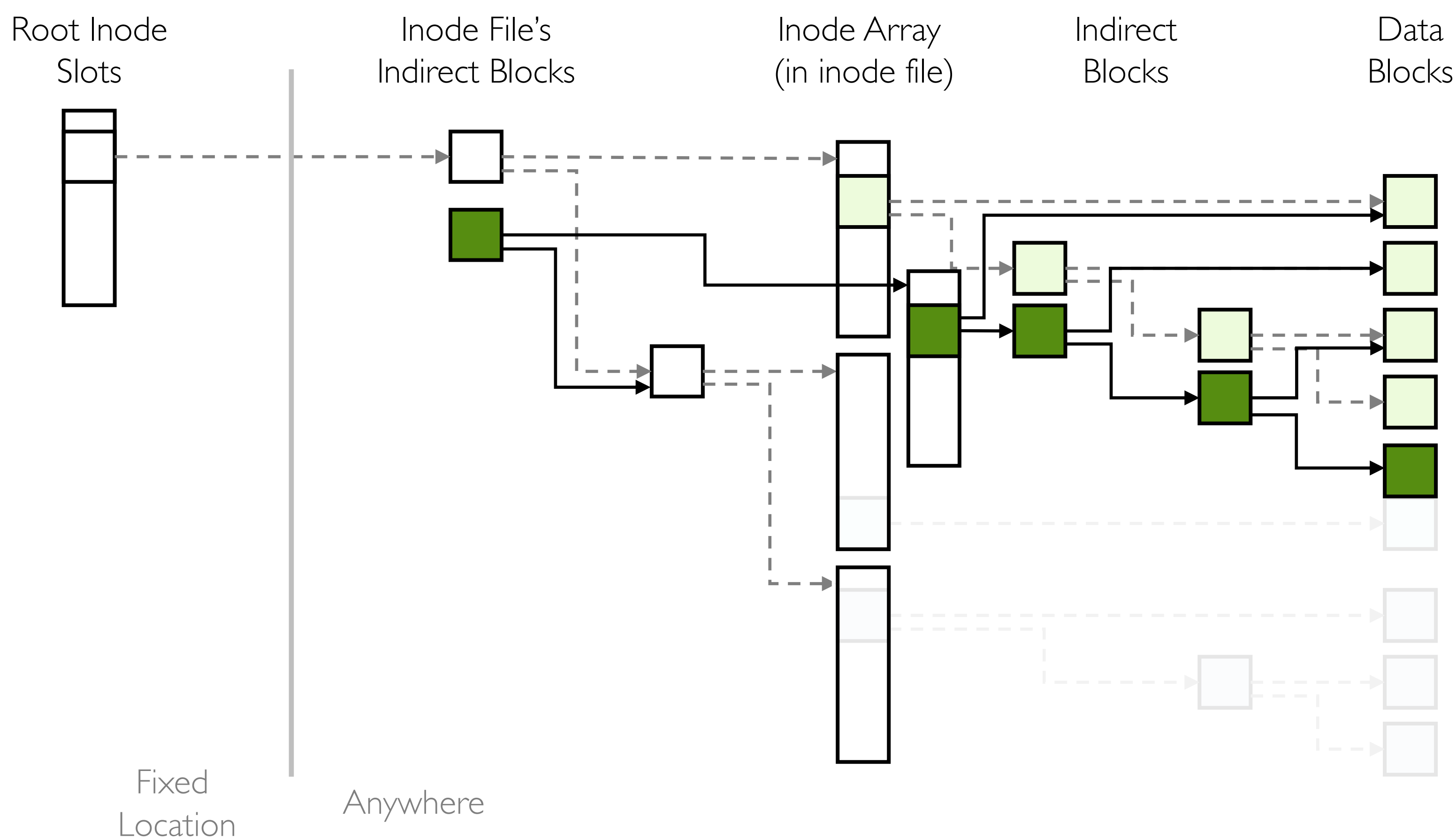
# Simple Copy-on-write File System



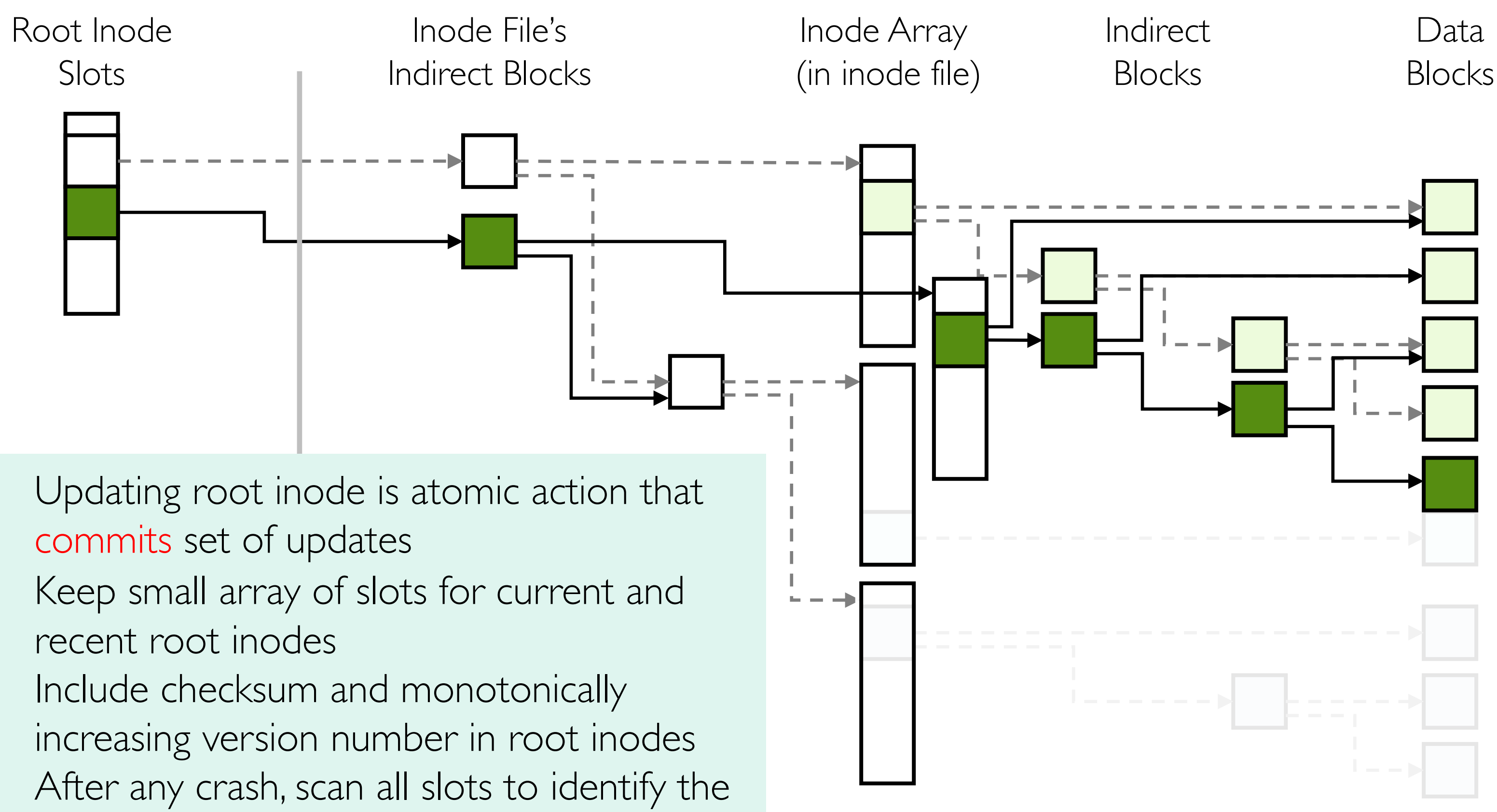
# Simple Copy-on-write File System



# Simple Copy-on-write File System

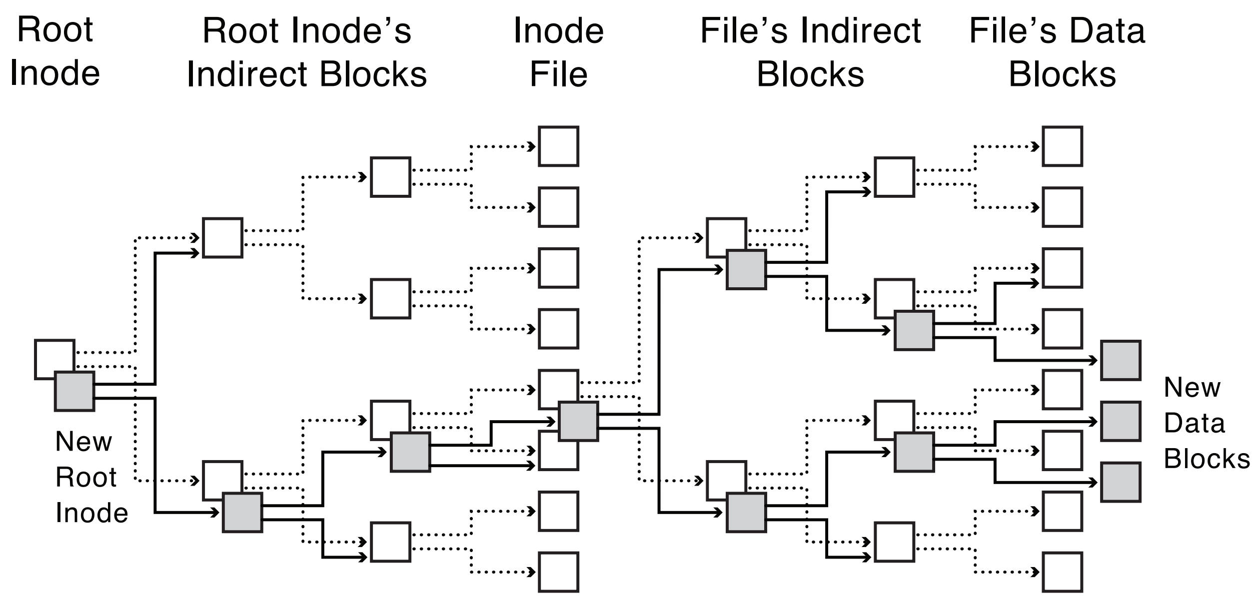


# Simple Copy-on-write File System



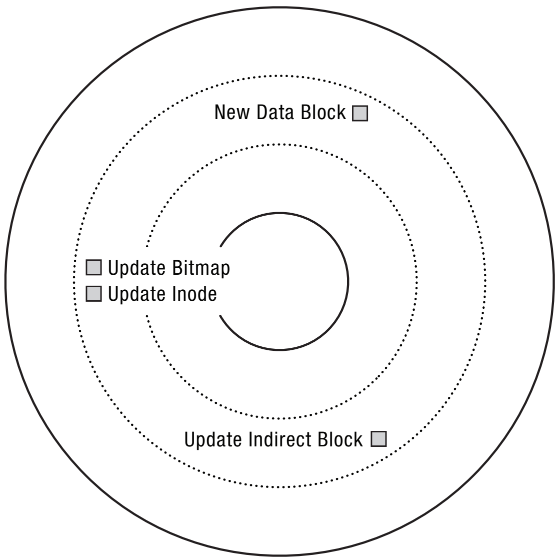

- Updating root inode is atomic action that commits set of updates
- Keep small array of slots for current and recent root inodes
- Include checksum and monotonically increasing version number in root inodes
- After any crash, scan all slots to identify the newest root inode that includes correct checksum

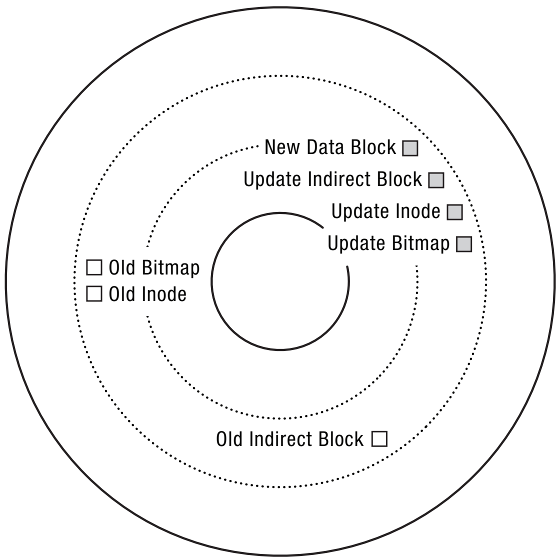
# Copy-on-write Batch Update




- Updates to file's inode and indirect blocks may be amortized over multiple writes to different blocks of the file
- Updates to the rood inode and root inode's indirect blocks may be amortized over multiple writes to different files

# Update-in-place vs. Copy-on-write File Systems




- Update-in-place file system updates data and metadata in their existing locations
- Copy-on-write file system makes new copies on every update
  - Updates can be grouped into single write to free range of sequential blocks on disk
  - Sequential writes are much faster than random ones → excellent write performance even though copy-on-write writes more data than update-in-place does

# Copy-on-write File Systems: Garbage Collection

- Write efficiency requires contiguous sequences of free blocks
  - Updates leave dead blocks scattered
- Read efficiency requires data read together to be in the same block group
  - Write anywhere leaves related data scattered
- Solution: background coalescing of live/dead blocks

# Copy-on-write File Systems: Discussion

- + Correct behavior regardless of failures
- + Fast recovery (root block array)
- + High throughput (best if updates are batched)

- – Potential for high latency
- – Small changes require many writes
- – Garbage collection essential for performance

# Outline

- Problem posed by machine/disk failures
- Transaction concept (ASID properties)
- Transactional file systems
  - Journaling and logging
- Copy-on-write file systems
- Redundant arrays of independent disks (RAID)

# How to Make File System Durable?

- Disk drives use error correcting codes (ECC) to deal with small defects
  - Can allow recovery of data from small media defects

- Writes should survive in short term
  - Either abandon delayed writes or
  - Use battery-backed RAM (non-volatile RAM - NVRAM) for dirty blocks in buffer cache

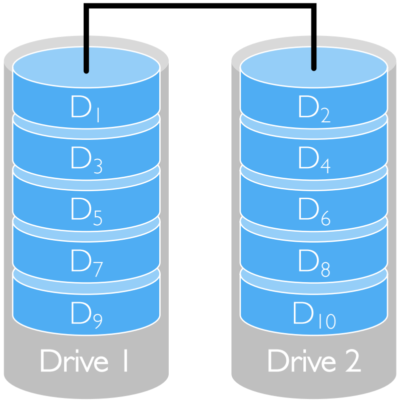
- Data should survive in long term
  - Need to replicate! Keep more than one copy of data!
  - Important element: independence of failure
    - Could put copies on one disk, but if disk head fails …
    - Could put copies on different disks, but if server fails …
    - Could put copies on different servers, but if building is struck by lightning …
    - Could put copies on servers in different continents …

# RAID: Redundant Arrays of Inexpensive Disks

- Invented by David Patterson, Garth A. Gibson, and Randy Katz in 1987
- Data stored on multiple disks (redundancy)
- Implemented either in software or hardware
  - In hardware case, done by disk controller; file system may not even know that there is more than one disk in use
- Initially, five levels of RAID (more now!)

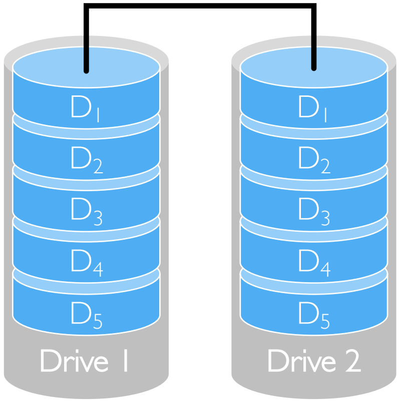
# RAID 0: Striping

- Multiple drives (at least 2) are used to store data
- Data is split into blocks written across all drives
- + Offers great performance
  - Read and write operations can be parallelized over multiple drives
- + Uses entire capacity and is easy to implement
- – Not fault-tolerant
  - If one drive fails, data cannot be restored (not suitable for mission-critical systems)



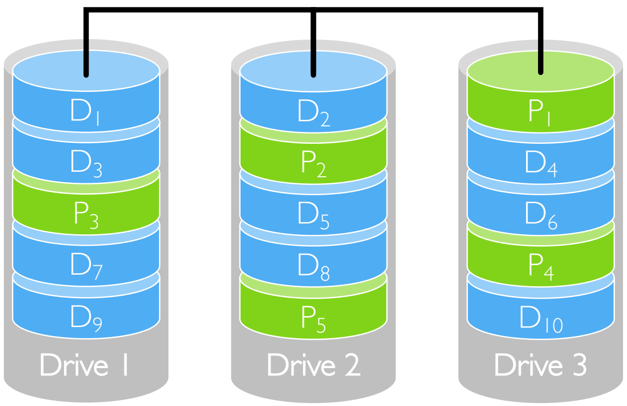
# RAID 1: Mirroring

- Data is stored twice
- Each drive is fully duplicated onto its *"shadow"*
- + Offers low-overhead data recovery
  - Disk failure ⇒ replace disk & copy data to new disk
  - Hot spare: idle disk already attached to system to be used for immediate replacement
- – 100% capacity overhead
- – Bandwidth sacrificed on write
  - Logical write = two physical writes



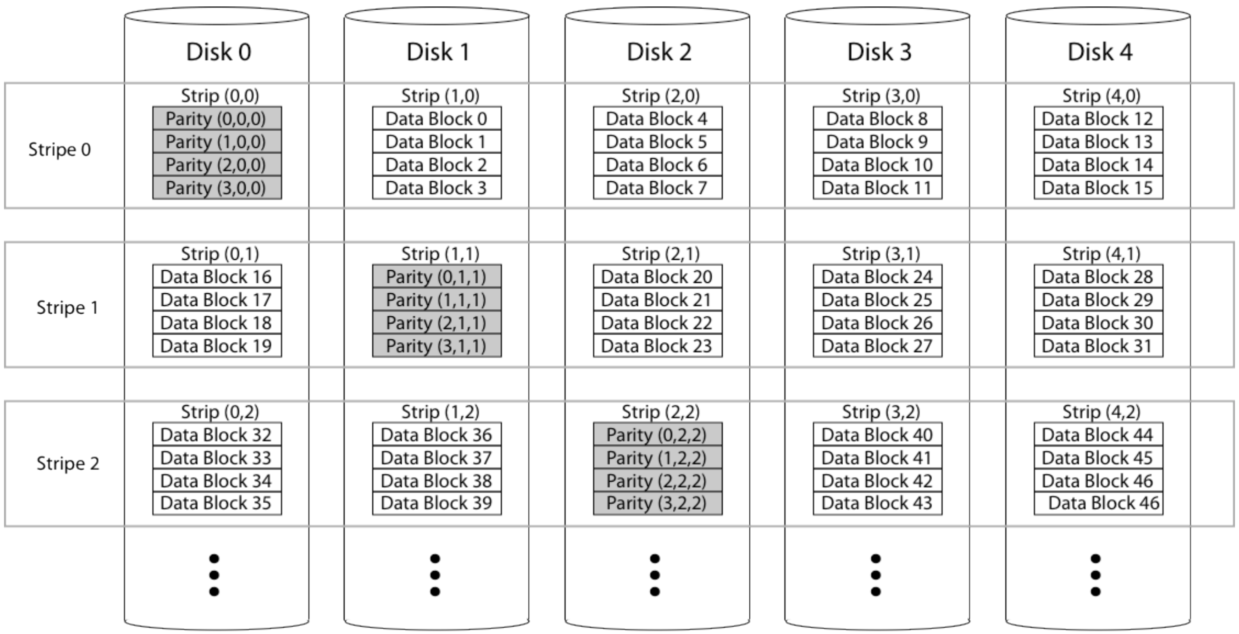
# RAID 5: Striping With Parity

- With G drives, for every G – 1 blocks, save one parity block (XOR of all data blocks)
  - $P = D_1 \oplus D_2 \oplus D_3 \oplus \ldots \oplus D_{G-1}$
  - If $D_1$ is lost, it can be restored using P and other blocks: $D_1 = P \oplus D_2 \oplus \ldots \oplus D_{G-1}$
- Parity for given set of blocks must be updated each time any of blocks is updated
  - To balance load, parity blocks are stored rotationally across all drives




- + Offers high availability with low capacity overhead
- – High recovery overhead
- – Complex implementation with high-overhead writes (requires 4 I/O operations)
  - Read old data
  - Read old parity
  - Write new data
  - Write new parity
    - Remove old data from parity ($P_{tmp} = P_{old} \oplus D_{old}$)
    - Calculate new parity ($P_{new} = P_{tmp} \oplus D_{new}$)

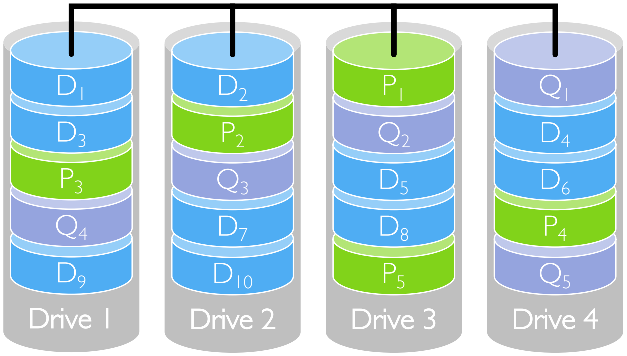
# RAID 5: Closer Look




- To balance parallelism versus sequential-access efficiency, single strip of several sequential blocks is placed on one drive before shifting to another
- Set of G – 1 data strips and their parity strip is called stripe

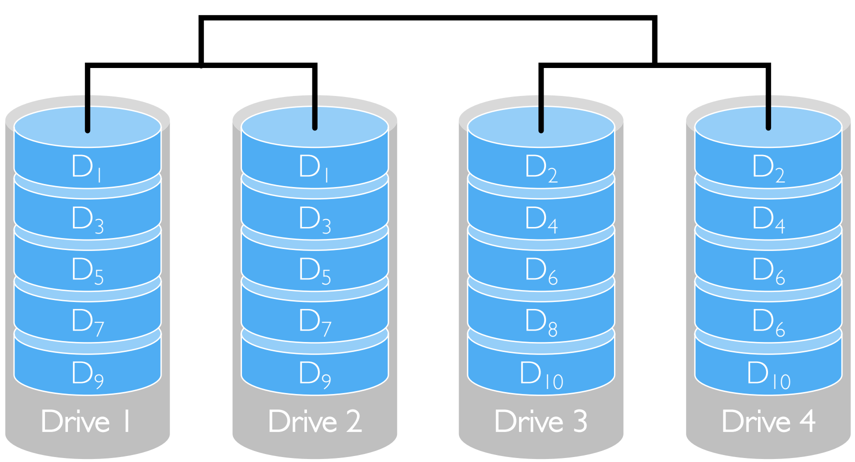
# RAID 6: Striping With Double Parity

- With RAID 5, time to repair failed drive is too long, another drive might fail in process!
- RAID 6 is like RAID 5, but with two parity data
  - Simple XOR will no longer work
  - We need something more complex
- Example: with G drives and k-bit blocks, if $G - 2 \le k$
  - $P = D_1 \oplus D_2 \oplus D_3 \oplus \ldots \oplus D_{G-2}$
  - $Q = D_1 \oplus \text{shift}(D_2) \oplus \text{shift}^2(D_3) \oplus \ldots \oplus \text{shift}^{G-3}(D_{G-2})$
  - If $D_1$ is lost, recover it using P and other blocks (like RAID 5)
  - If $D_1$ and P are lost, recover it using Q and other blocks
    - $D_1 = Q \oplus \text{shift}(D_2) \oplus \text{shift}^2(D_3) \oplus \ldots \oplus \text{shift}^{G-3}(D_{G-2})$
  - If D1 and D2 are lost, recover them using P, Q, and other blocks
    - $D_1 \oplus D_2 = P \oplus D_3 \oplus \ldots \oplus D_{G-2}$
    - $D_1 \oplus \text{shift}(D_2) = Q \oplus \text{shift}^2(D_3) \oplus \ldots \oplus \text{shift}^{G-3}(D_{G-2})$
    - System of 2k equations in 2k unknowns which uniquely determines the lost data
  - If $G - 2 > k$ or if we need more parity blocks, then general option is Reed-Solomon system
- + More reliable than RAID 5
  - Tolerates two drive failures
- – Even more slower writes than RAID 5
- – Higher capacity overhead than RAID 5



# RAID 10: Combining RAID 1 & RAID 0

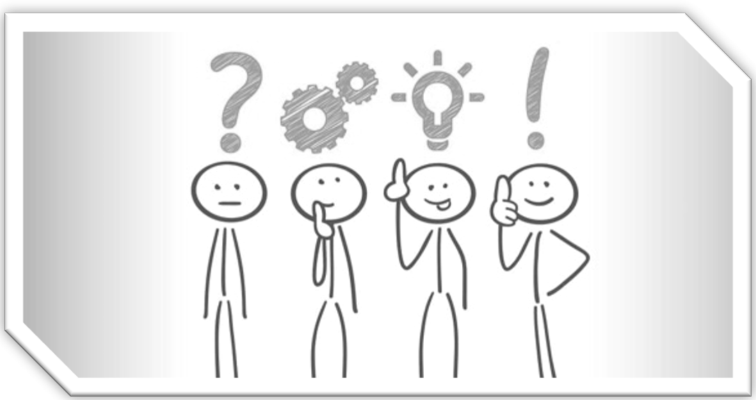
- RAID 10: RAID 1 + RAID 0
- + Advantages of RAID 1 and RAID 0
- – Higher capacity overhead than RAID 5 and 6



# Summary

- Transactions
  - ACID properties: atomicity, consistency, isolation, and durability
  - Redo logging and two-phase locking
- Transactional file systems
  - Journaling and logging
- Copy-on-write file systems
  - Write new version of file system on each update
- Redundant arrays of inexpensive disks (RAID)
  - RAID 0, 1, 5, 6, and 10

# Questions?



# Acknowledgment


- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny