



Free Space Management



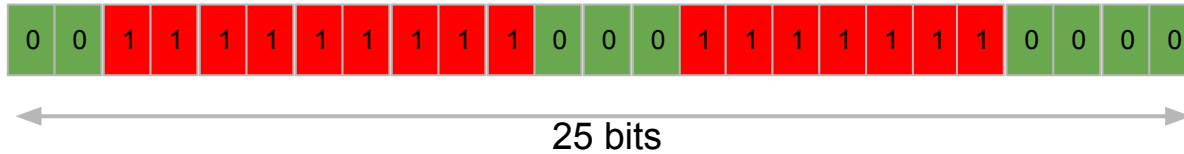
Free Space Management

- Bit Vector/Bitmap
- “Linked-List” (Free List)
 - Simple
 - Grouping
 - Counting
- The “list” can be implemented as a tree (for faster search)

Bitmap / Bit Vector / Array of bits

- Each block is represented by 1-bit
 - 0 = free, 1 = used
- Number of bytes for bitmap = $\frac{1}{8}$ number of total disk blocks
- Fast to locate free blocks: use bitwise operations
 - Searching a particular block can be
- Require extra space to maintain free blocks

BitVector / Bitmap



25-block hard drive requires 25-bit bitmap (4 bytes)

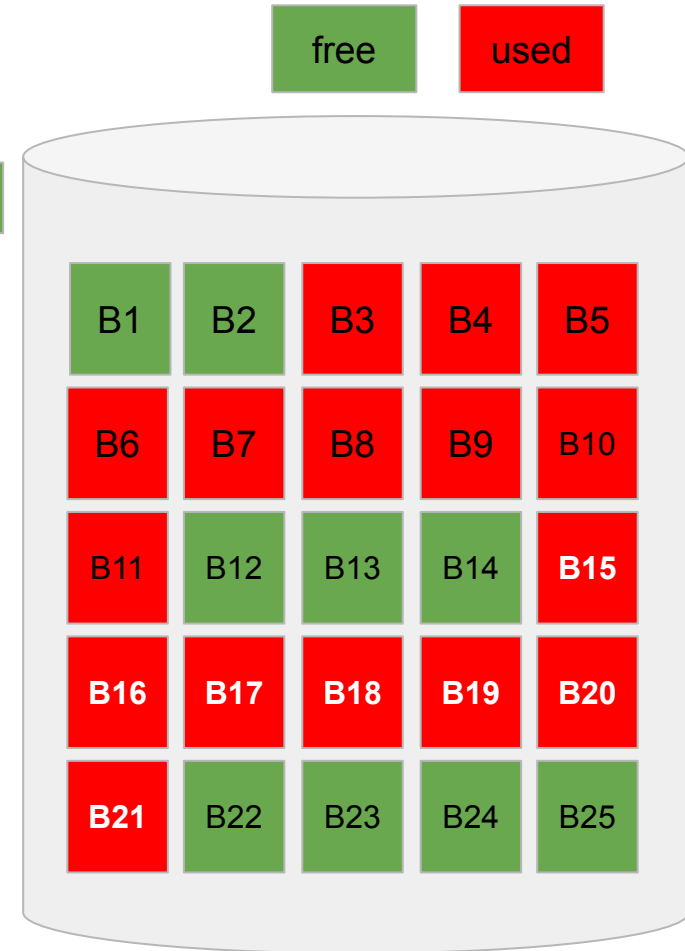
Total Disk Size : D bytes

Block Size : S bytes

disk blocks = $\frac{D}{S} = \# \text{ bits in bitmap}$

bitmap size = $\left\lceil \frac{1}{8} \frac{D}{S} \right\rceil$ bytes

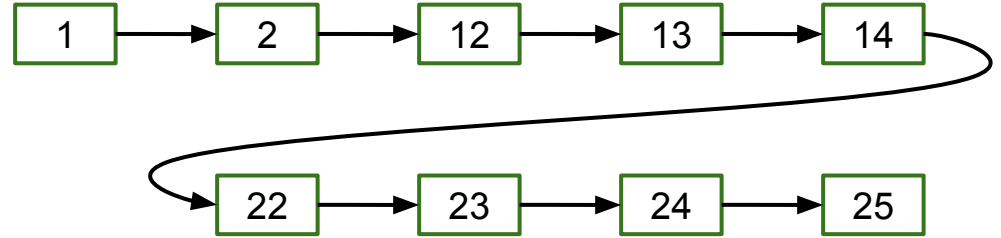
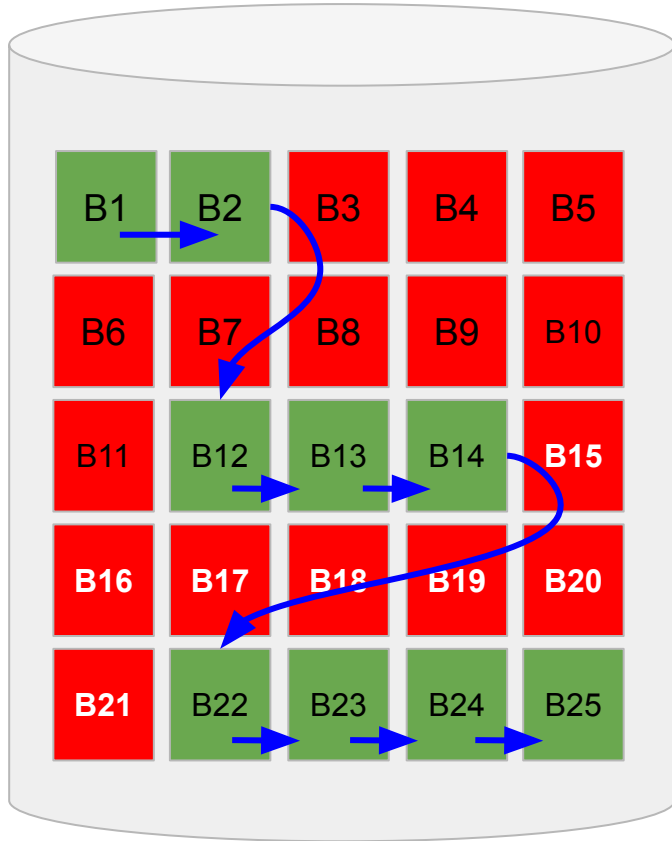
wasted = $\frac{\text{bitmap size}}{\text{Disk size}} = \frac{D/(8S)}{D} = \frac{1}{8S}$



Linked-List of Free Blocks / Free List

- Improvements over bitmap technique
 - *Chain all the free blocks together*
 - The superblock keeps the “head” and “tail” of this list
- Place the “next” pointer inside (at the end) of the freeblocks
- Searching for N free blocks requires reading N blocks in the chain (too many disk I/O)

Simple Linked-List



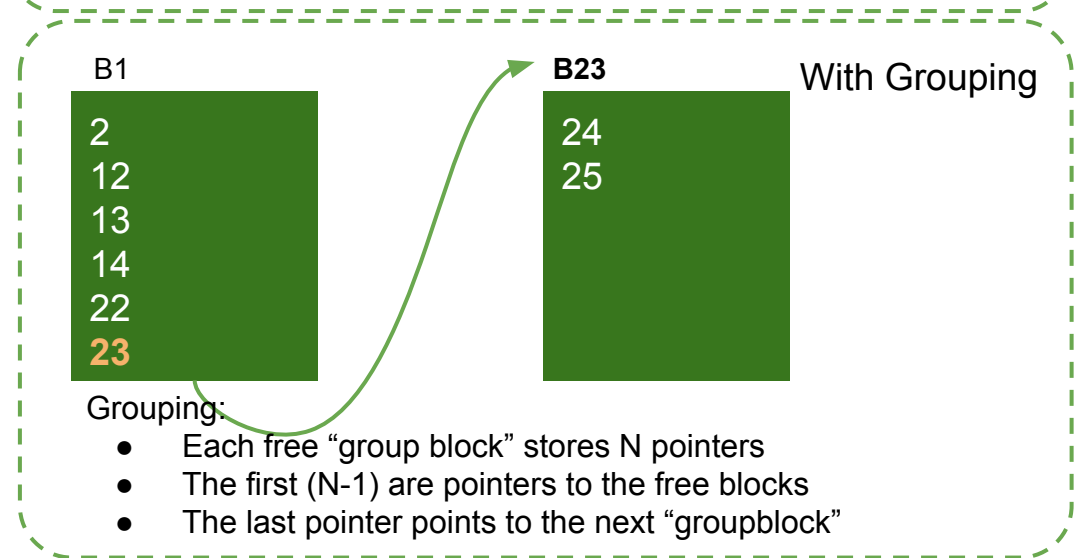
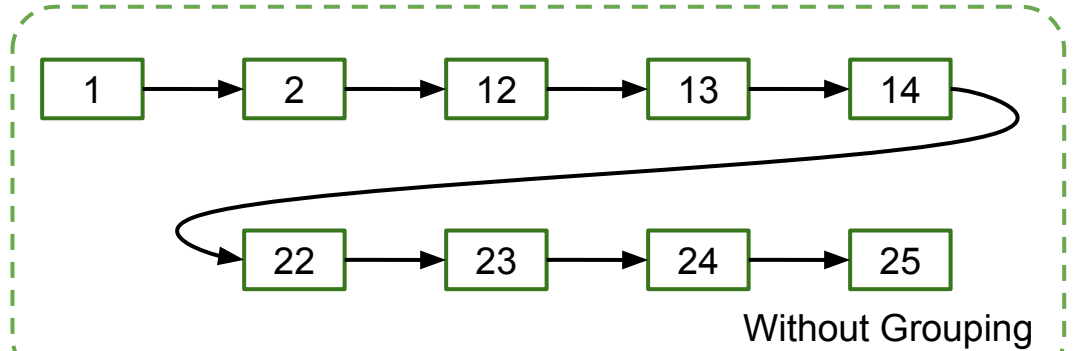
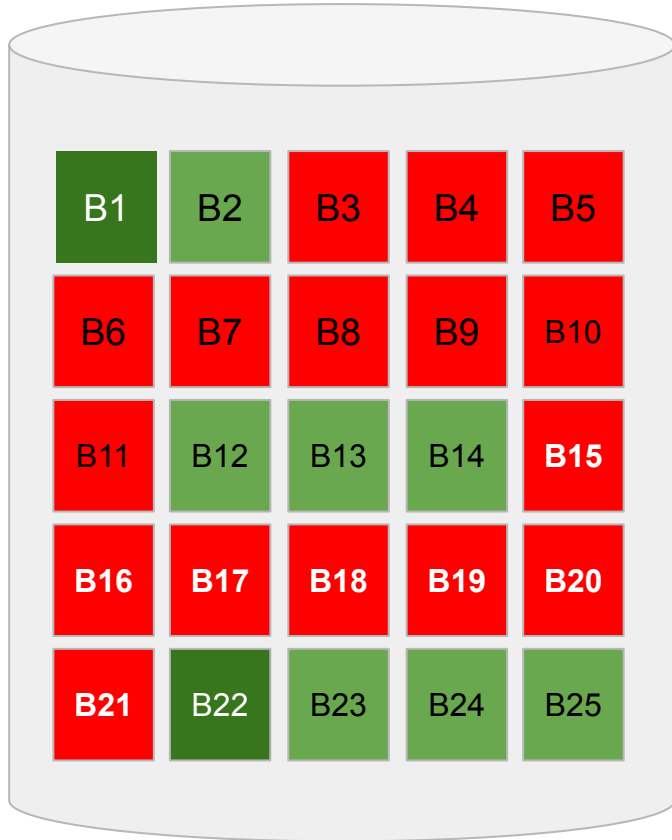
Cost of searching N free blocks = $O(N)$



Packed Linked-List (Group of Block #)

free

used



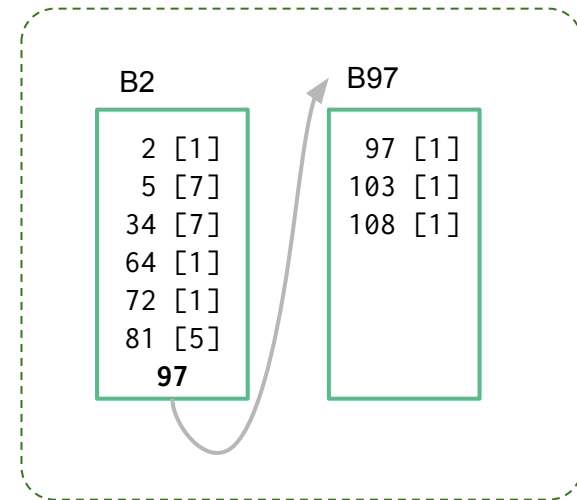
Linked-List & (Group & Range / Count)

Free blocks : 2, **5, 6, 7, 8, 9, 10, 11**, **34, 35, 36, 37, 38, 39, 40**, 64, 72, **81, 82, 83, 84, 85**, 97, 103, 108,

Free blocks (range) : 2, **5-11**, **34-40**, 64, 72, **81-85**, 97, 103, 108,

Free blocks (range) : 2-2, **5-11**, **34-40**, 64-64, 72-72, **81-85**, 97-97, 103-103, 108-108,

Free blocks [count] : 2[1], **5[7]**, **34[7]**, 64[1], 72[1], **81[5]**, 97[1], 103[1], 108[1],



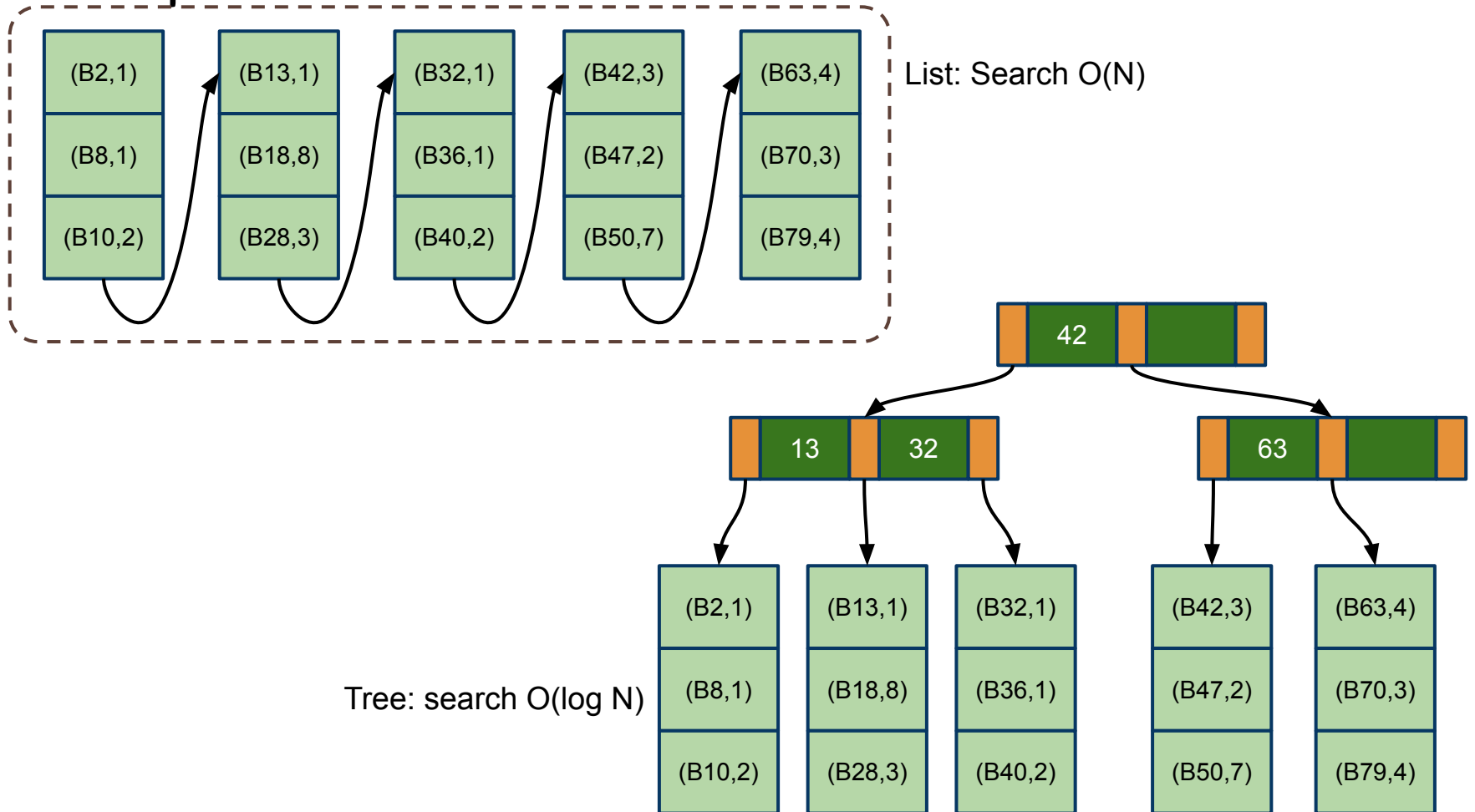
Improvement on size of linked list: 25 \Rightarrow 4 \Rightarrow 2



Upgrade from list to tree?



Transform List into a Tree



Data Recovery

Critical On-Disk Data Structures

A filesystem essentially holds (at least) three separate trees:

- A tree of files and directories (*user-owned*)
- A tree of **free** data blocks (maintained by OS)
- A tree of **free** index blocks (maintained by OS)

These trees originate from the **superblock** (or some kind of “special” block)

Any updates to the FS must guarantee the three trees are **in sync!**

Low-Level Operations in Linux Ext(2|3|4)

To write a file into a (Linux) filesystem

1. Allocate new data blocks
 - a. Update the list of free blocks in the **superblock**
2. Allocate a new inode block
 - a. Update the list of free inode block in the **superblock**
3. Write the file contents into the **data blocks**
4. Update the **inode block**: file metadata and block pointers

When the above sequence *does not run to completion* (i.e. by power failure), the filesystem records only a *partial (inconsistent) data/metadata* of your file

Database Transactions & (Intent) Logs

- To guarantee data integrity
- Multiple operations that update different parts of the DB are performed under one **transaction**: "*BEGIN TRANSACTION*" and "*COMMIT*" / "*ROLLBACK*"
 - In addition to changing the data, a transaction also **logs the intended changes** (insert, delete, update)
 - **The log must be securely saved PRIOR TO the modification of the actual data**
- At recovery time, the contents of the log are compared with the actual data and unfinished transactions can be recovered

Transaction & Logs: Database Example

Transfer \$500 from Account-A to Account-B

	Account-A	Account-B
Current balance	10,000	2,000
Expected balance	9,500	2,500

Log Format: action, target, oldval, newval

```
Update, A, 10000, 9500  
Update, B, 2000, 2500
```

Several Possible States (after a crash)

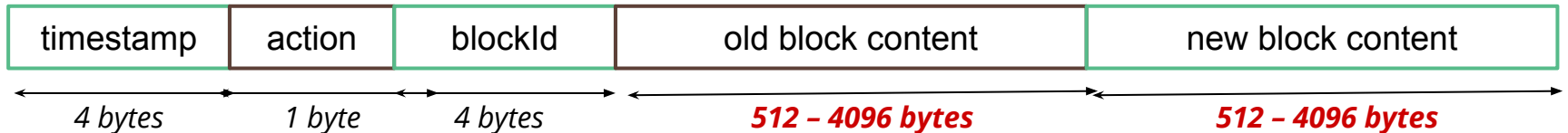
Account-A	Account-B	Explanation	Required Action
9,500	2,000	Failed to update B's account	Update B, Delete Log
10,000	2,500	Failed to update A's account	Update A, Delete Log
10,000	2,000	Failed to update both	Update Both, Delete Log
9,500	2,500	Excellent	Delete Log

Journaling / Log-Structured FS

- Every modification to the FS must be associated with a log entry
 - Log entry format: **timestamp**, **action**, **blockid**, oldvalue, newValue
 - Log entries are created for ALL types of modification (including inode and superblock)
- The log must be saved FIRST before the actual block updates take place
- **Delete** the log after the successful data block updates
- Recovery after system crash: use the existing log entries to restore FS

Log-Structured FS: Advantages & Issues

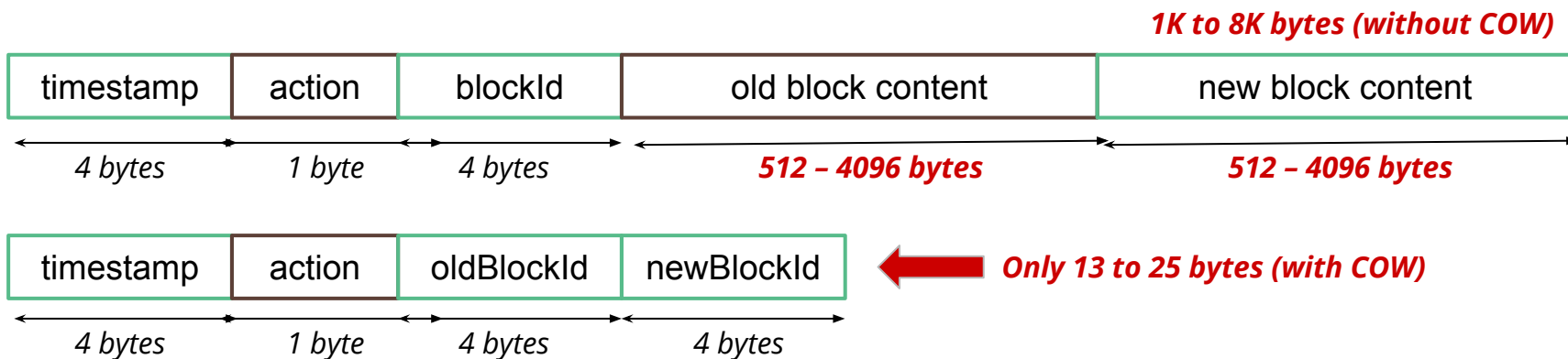
- Performance Improvement (for the user program)
 - As soon as the log entries are safely written, the operation can be **considered complete**
 - The user program **does not have to wait** until the data blocks are updated
 - The system can complete the rest of the operations as if it is “recovering” from a “crash”
- Potential issues
 - Log entries requires lots of space
 - Log entries themselves are corrupted \Rightarrow use checksum to verify log entries



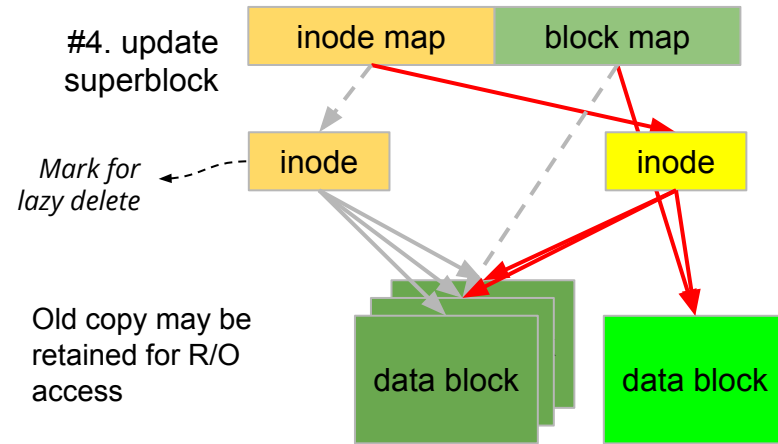
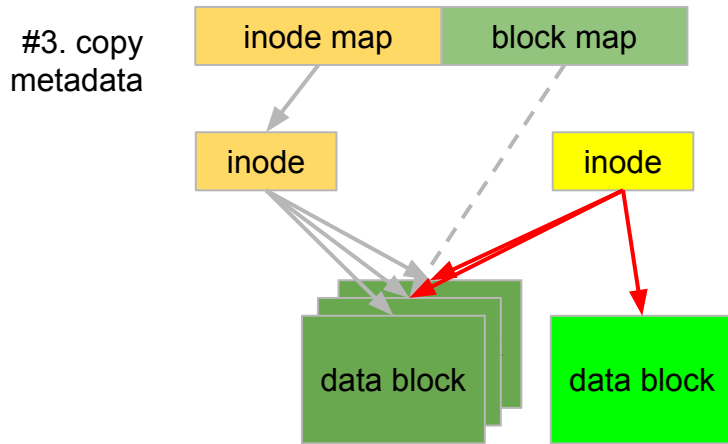
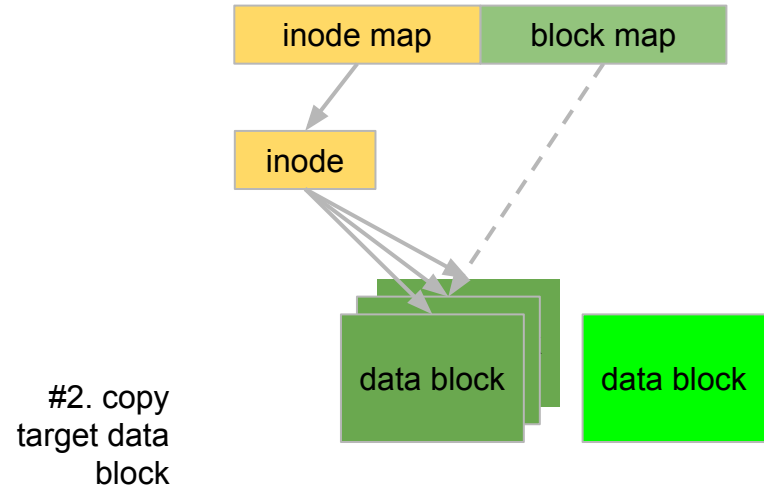
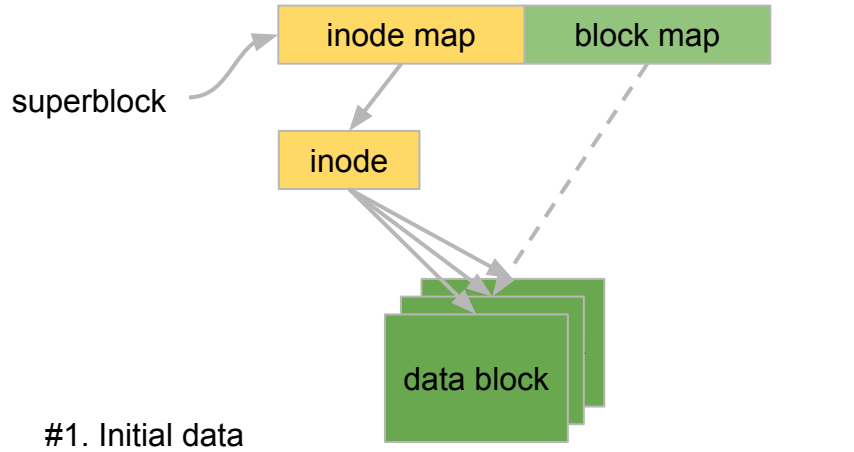



COW: Shorter Log Entries

- Copy-on-Write: when a block is **updated, don't** overwrite it, but create a copy
 - Write the updated contents to a new block, temporarily keep the old block
 - The COW strategy also applies to inode blocks (not only to data blocks)
- Advantage: the disk keeps both old content and updated content
- Journal entries can be made MUCH shorter



Copy-on-Write (on disk blocks)





No COW: Update \Rightarrow Overwrite data blocks
With COW: Update \Rightarrow Redirect pointers




(doesn't it look like git?)


Recovery & Consistency Check

- Superblock, inode, files and directory structures must agree with each other
- A filesystem persistently stores three lists (or trees)
 - The tree structure of files/directories
 - A tree/list of free data blocks
 - A tree/list of free index blocks
- The head (root) pointer of these trees are kept in the superblock

ZFS



ZFS = Filesystem
+ Volume Manager
+ Revision Control



ZFS

- 128-bit block address
 - For sector size of 512 bytes, total disk capacity is $2^{128} \times 512 = 2^{128} \times 2^9 = 2^{137}$ bytes
- Index nodes are generated **on demand**. **What is the advantage?**
 - UFS preallocates index nodes in the superblock (at the time of FS format)
- UFS superblocks = ZFS uberblock
- COW (Copy-On-Write)
 - This strategy allows storing multiple revisions of a file
 - Snapshots = **R/O** copies of older revisions
 - Clones = **R/W** copies of a dataset (similar to a git branch)

How much is 2^{137} bytes?

- 2^{137} bytes = $2^{137} \times 2^3$ bits = 2^{140} bits $\approx 10^{42}$ bits
- Estimated number of atoms on our earth = 10^{50}
- If we were able to store *one bit as one atom*, our hard drive would have been 1/256 the size of the earth!

ZFS Storage Pools

- Design goal: Increasing storage capacity should be as easy as increasing RAM size
 - ZFS allows sysadmin to add new disks on-demand (LVM)
- ZFS virtual devices (vdevs): **block-oriented storage** (HD, SSD, NVRAM, ...)
- Virtual devices are managed into Storage Pools
 - Solve the “partition too small” problem
- Other FSs use 3rd party **logical volume managers** to “expand” capacity

ZFS Data Integrity & Error Checking

- Design Objectives
 - ZFS should be self-healing
 - Filesystems should not require “manual” periodic checking using fsck(8)
- ZFS computes the hash value of (almost) all of its data
 - The hash value of a block is stored in a different block
 - A parent block stores the hash of its children
 - A child block stores the hash of its parent
 - Merkle Tree