CS 60038: Advances in Operating Systems Design

File Systems

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty

sandipc@cse.iitkgp.ac.in

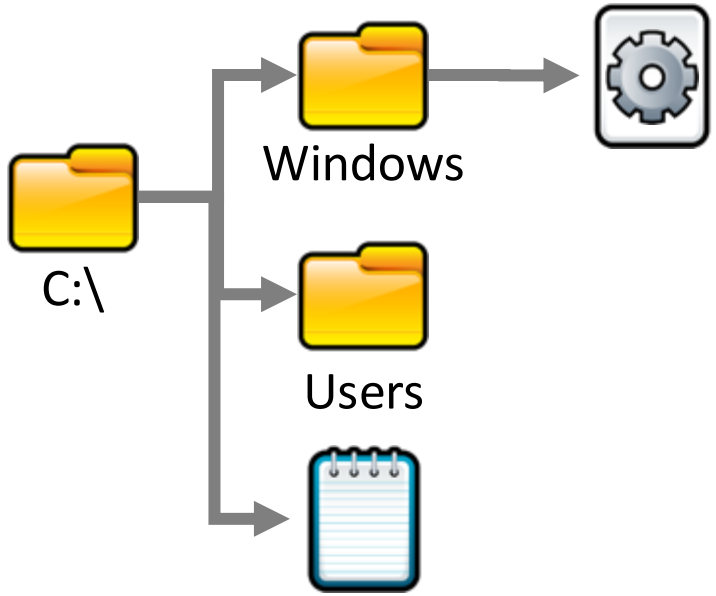
File Allocation Tables (FAT)

- Simple file system popularized by MS-DOS
  - First introduced in 1977
  - Most devices today use the FAT32 spec from 1996
  - FAT12, FAT16, VFAT, FAT32, etc.
- Still quite popular today
  - Default format for USB sticks and memory cards
  - Used for EFI boot partitions
- Name comes from the index table used to track directories and files

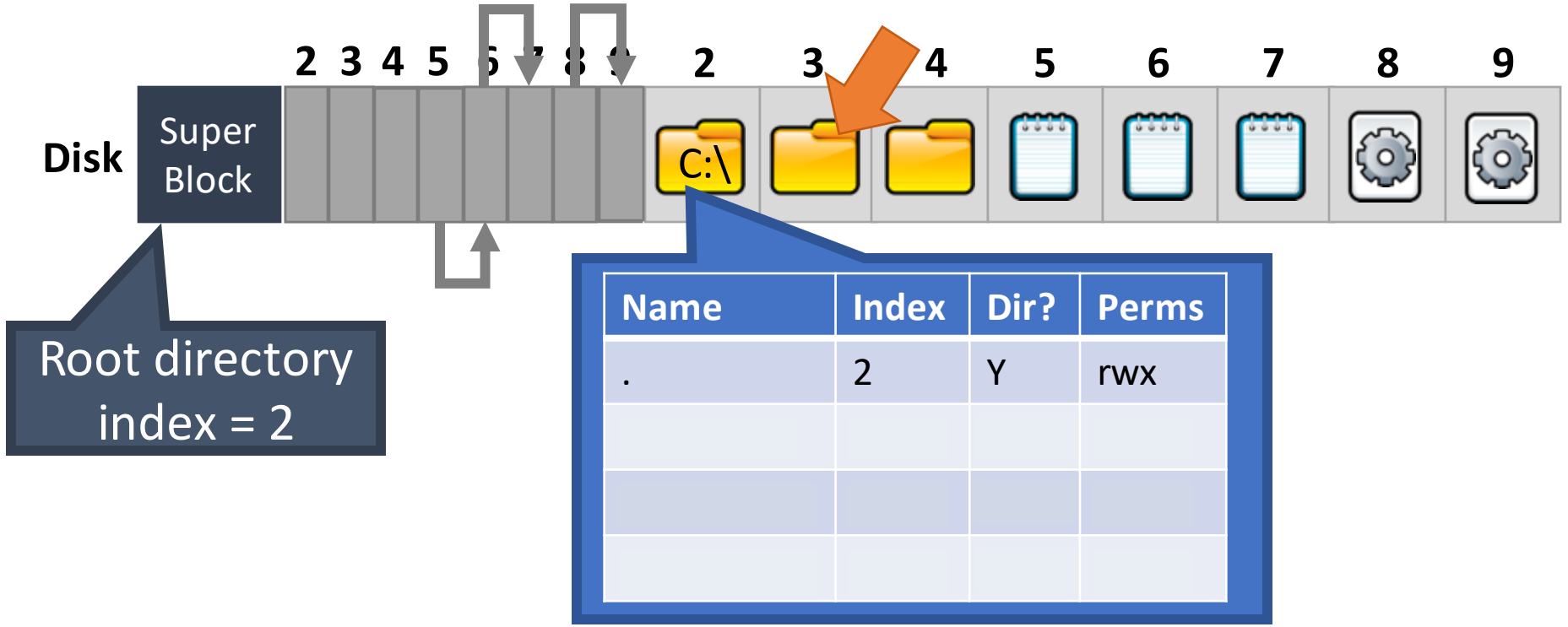
- Stores basic info about the file system
- FAT version, location of boot files
- Total number of blocks
- Index of the root directory in the FAT

- File allocation table (FAT)
- Marks which blocks are free or in-use
- **Linked-list structure** to manage large files

- Store file and directory data
- Each block is a fixed size (4KB – 64KB)
- Files may span multiple blocks



- Directories are special files
  - File contains a list of entries inside the directory
- Possible values for FAT entries:
  - 0 – entry is empty
  - 1 – reserved by the OS
  - 1 < N < 0xFFFF – next block in a chain
  - 0xFFFF – end of a chain



ext2 inodes

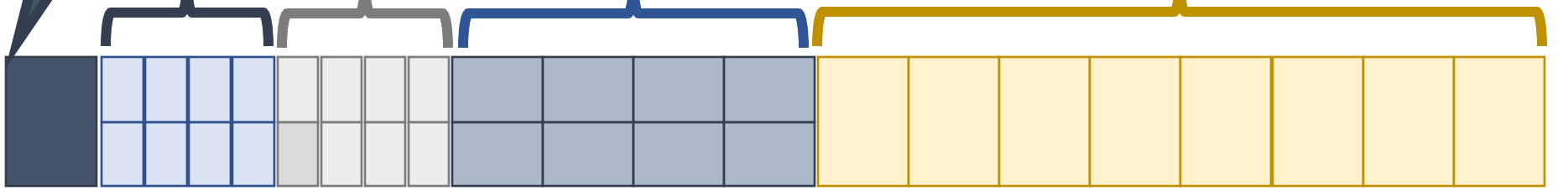
- Super block, storing:
  - Size and location of bitmaps
  - Number and location of inodes
  - Number and location of data blocks
  - Index of root inodes

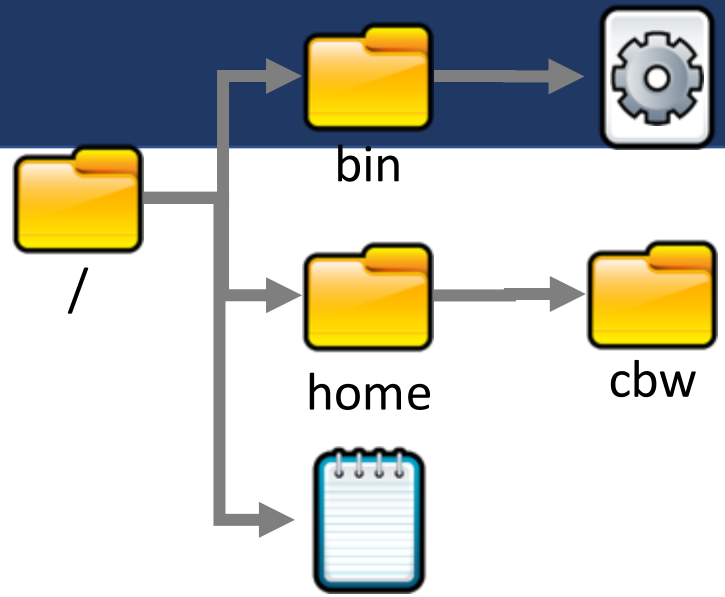
Bitmap of free & used data blocks

Bitmap of free & used inodes

- Table of inodes
- Each inode is a file/directory
- Includes meta-data and lists of associated data blocks

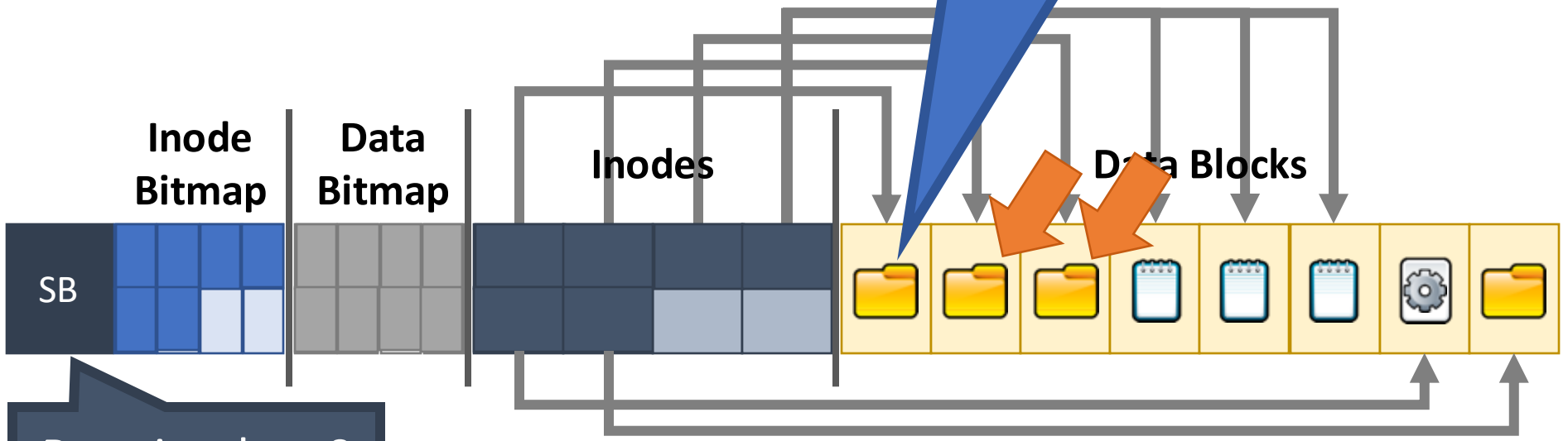
Data blocks (4KB each)





- Directories are files
- Contains the list of entries in the directory

- Each inode can directly point to 12 blocks
- Can also indirectly point to blocks at 1, 2, and 3 levels of depth



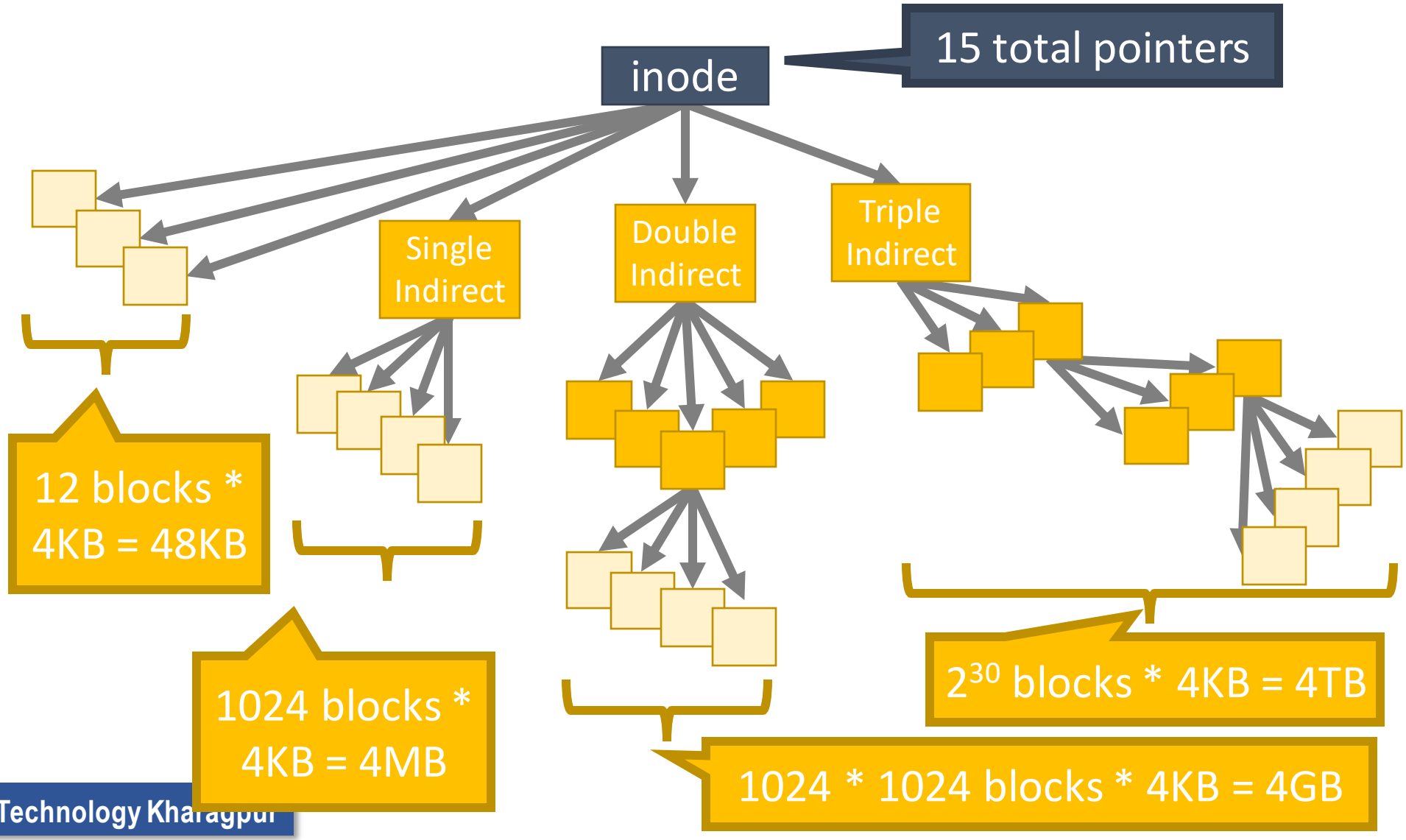
Root inode = 0

ext2 inodes

Size (bytes)	Name	What is this field for?
2	mode	Read/write/execute?
2	uid	User ID of the file owner
4	size	Size of the file in bytes
4	time	Last access time
4	ctime	Creation time
4	mtime	Last modification time
4	dtime	Deletion time
2	gid	Group ID of the file
2	links_count	How many hard links point to this file?
4	blocks	How many data blocks are allocated to this file?
4	flags	File or directory? Plus, other simple flags
60	block	15 direct and indirect pointers to data blocks

inode Block Pointers

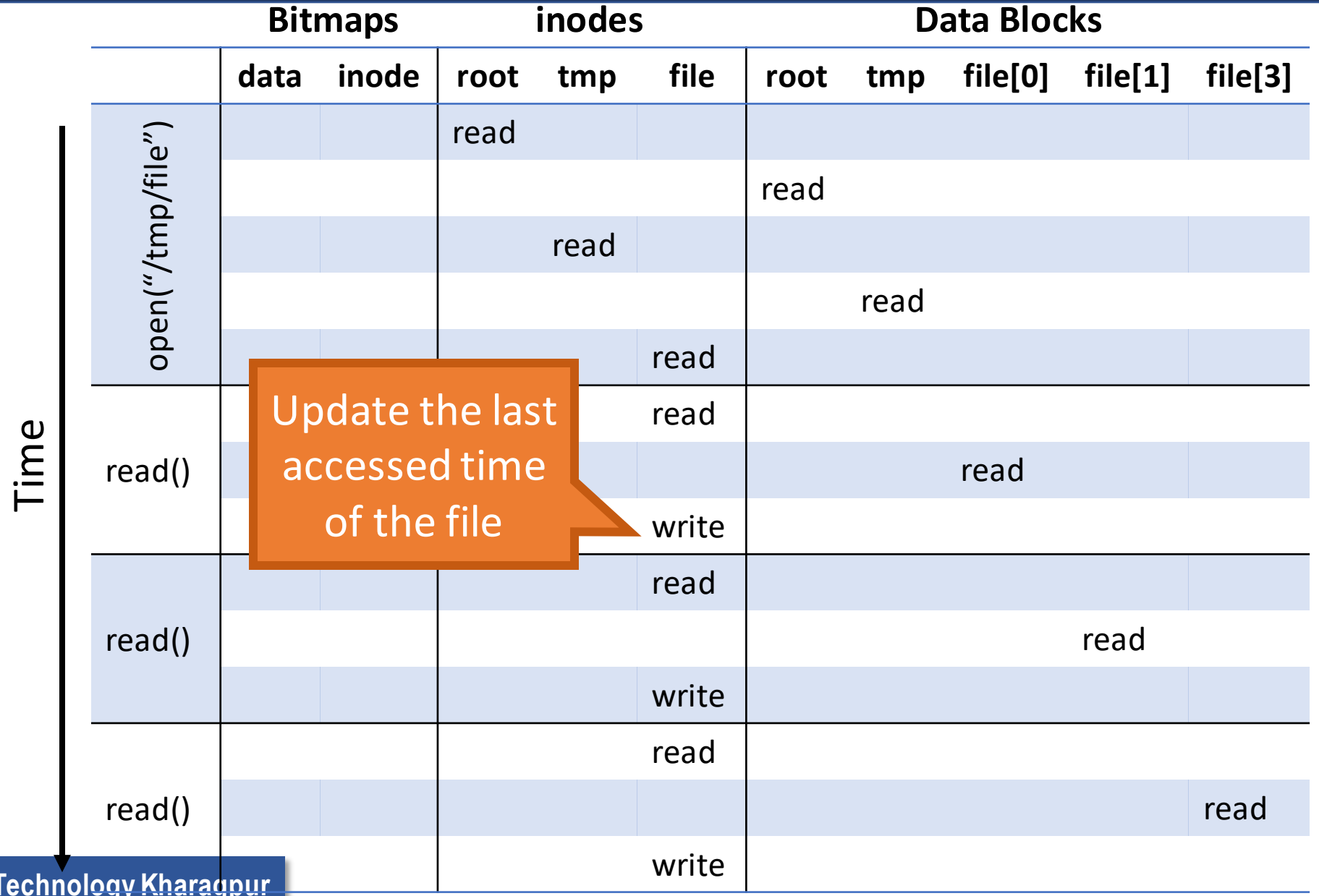
- Each inode is the root of an unbalanced tree of data blocks



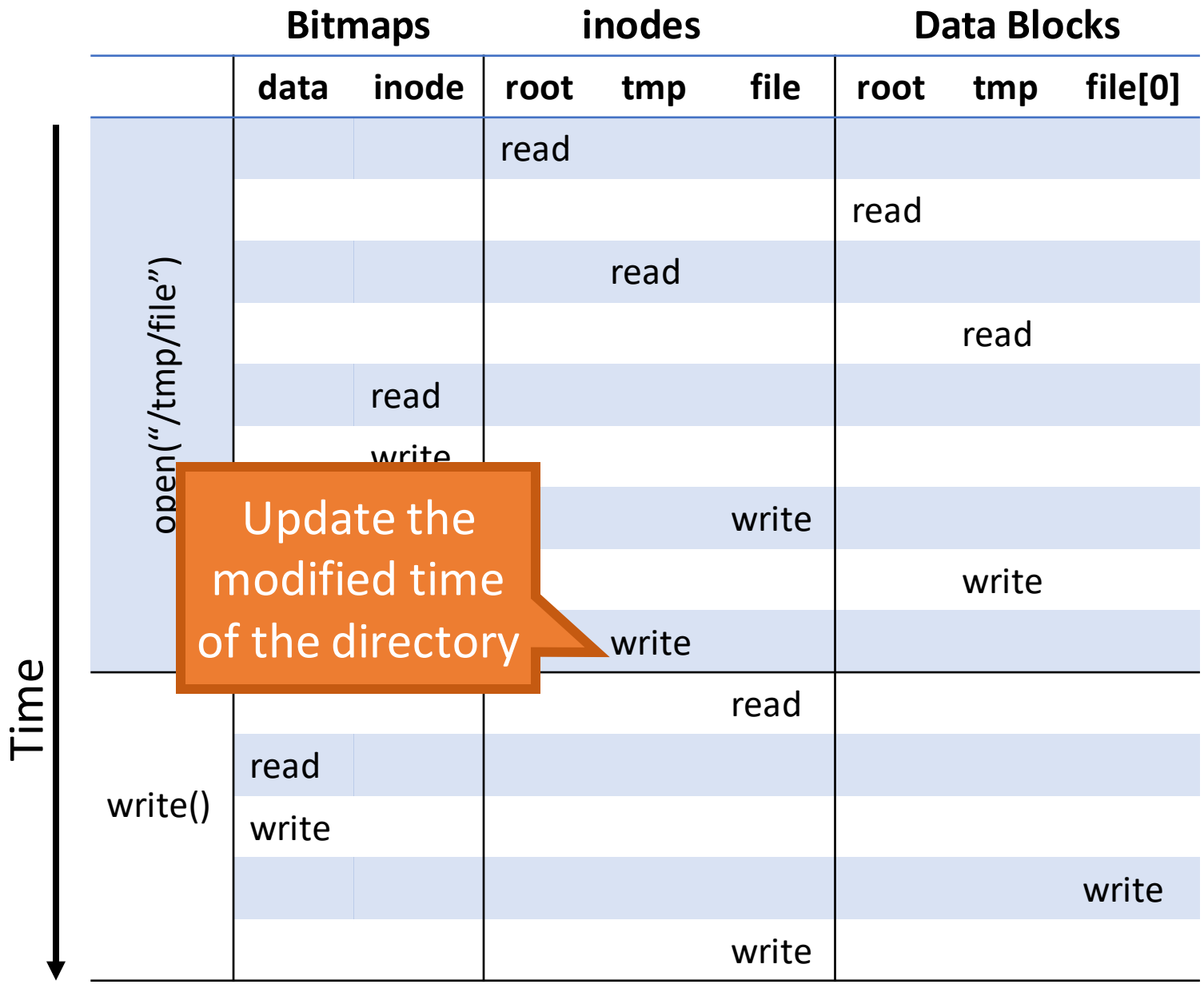
Advantages of inodes

- Optimized for file systems with many small files
  - Each inode can directly point to 48KB of data
  - Only one layer of indirection needed for 4MB files
- Faster file access
  - Greater meta-data locality → less random seeking
  - No need to traverse long, chained FAT entries
- Easier free space management
  - Bitmaps can be cached in memory for fast access
  - inode and data space handled independently

File Reading Example

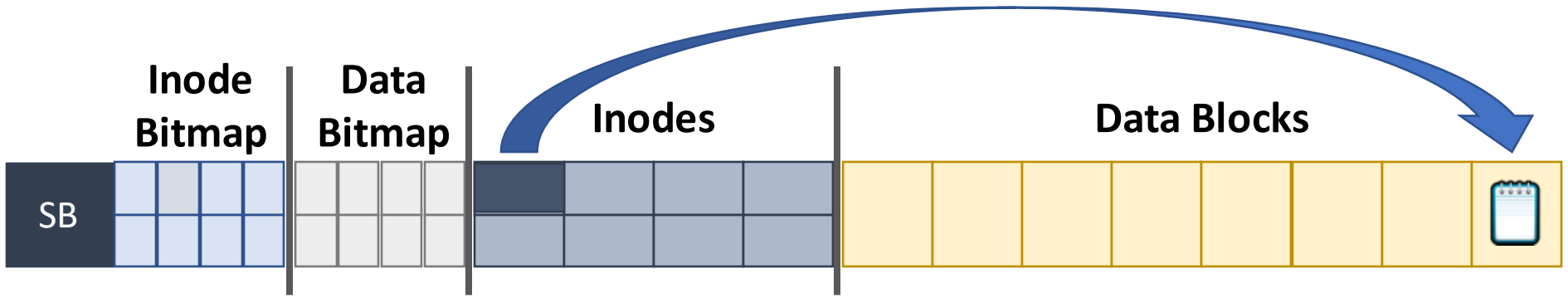


File Create and Write Example



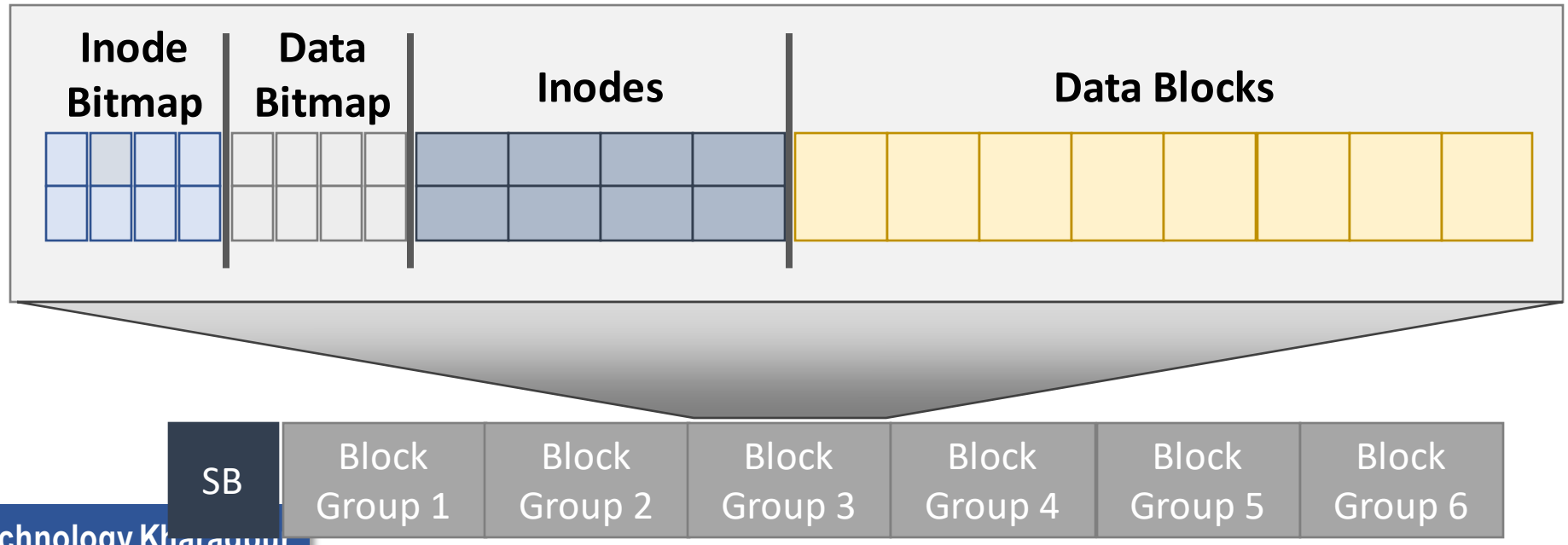
ext: The Good and the Bad

- The Good – ext file system (inodes) support:
  - All the typical file/directory features
  - Hard and soft links
  - More performant (less seeking) than FAT
- The Bad: poor locality
  - ext is optimized for a particular file size distribution
  - However, it is not optimized for spinning disks
  - inodes and associated data are far apart on the disk!



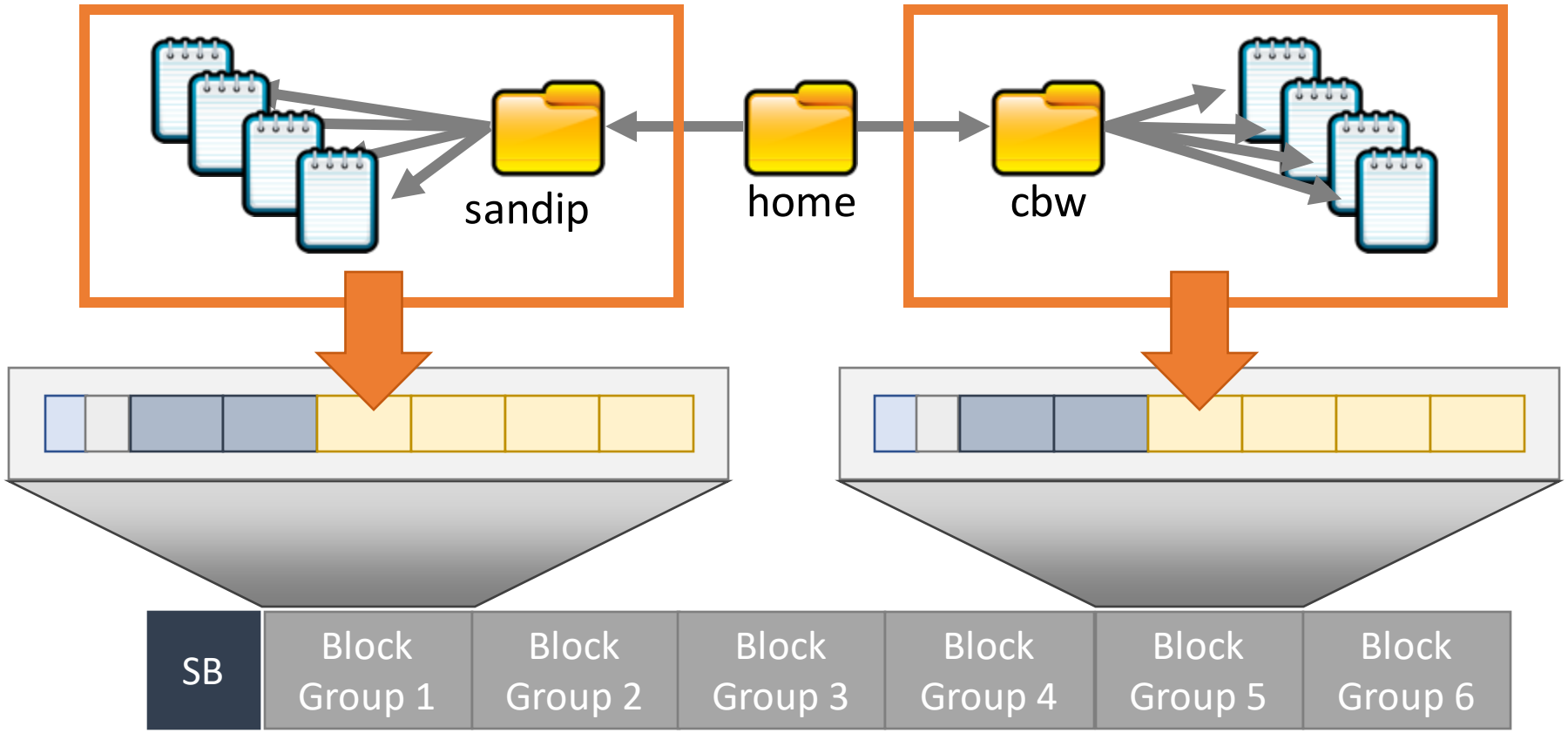
Block Groups

- In ext, there is a single set of key data structures
  - One data bitmap, one inode bitmap
  - One inode table, one array of data blocks
- In ext2, each block group contains its own key data structures



Allocation Policy

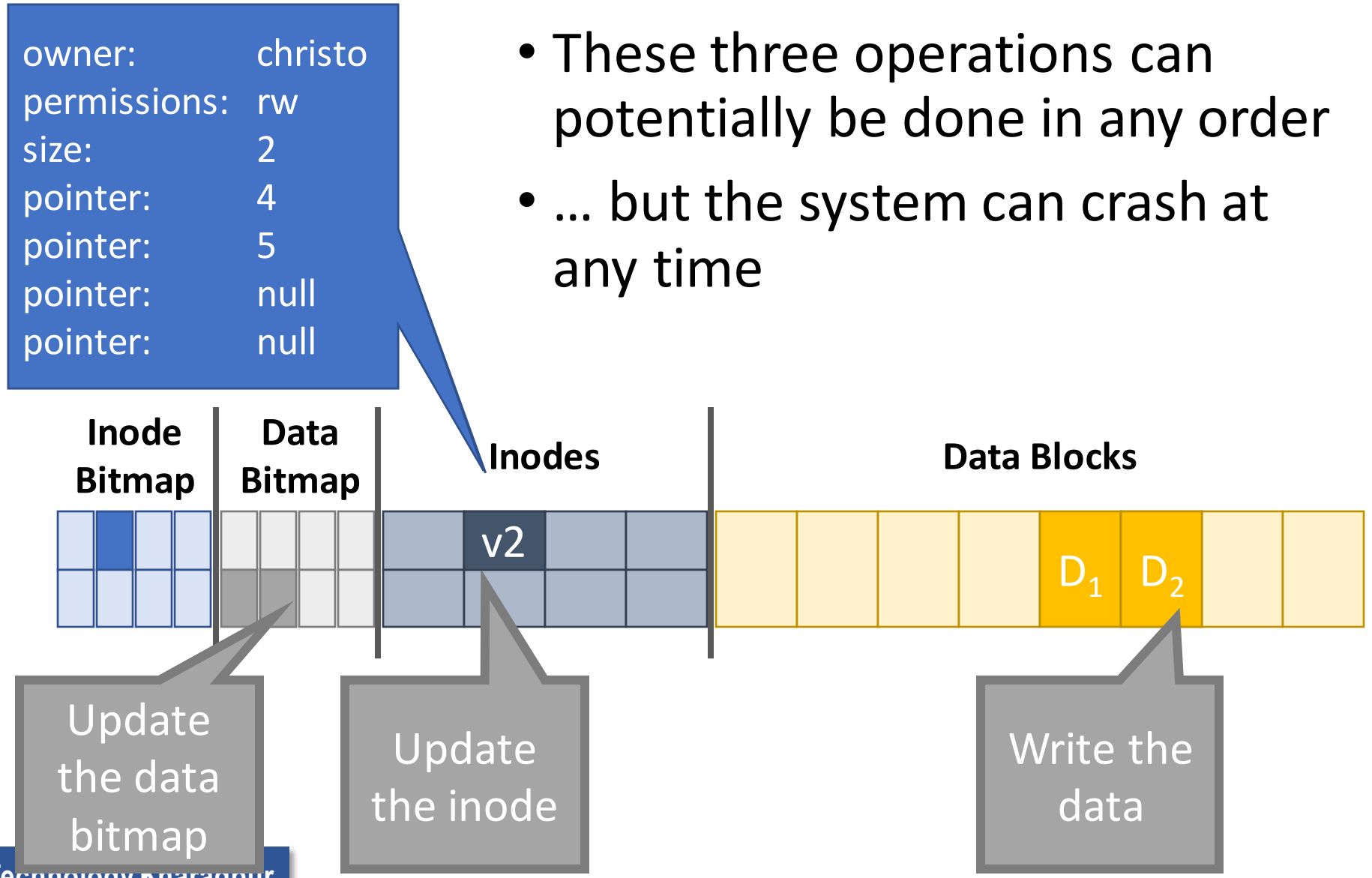
- ext2 attempts to keep related files and directories within the same block group



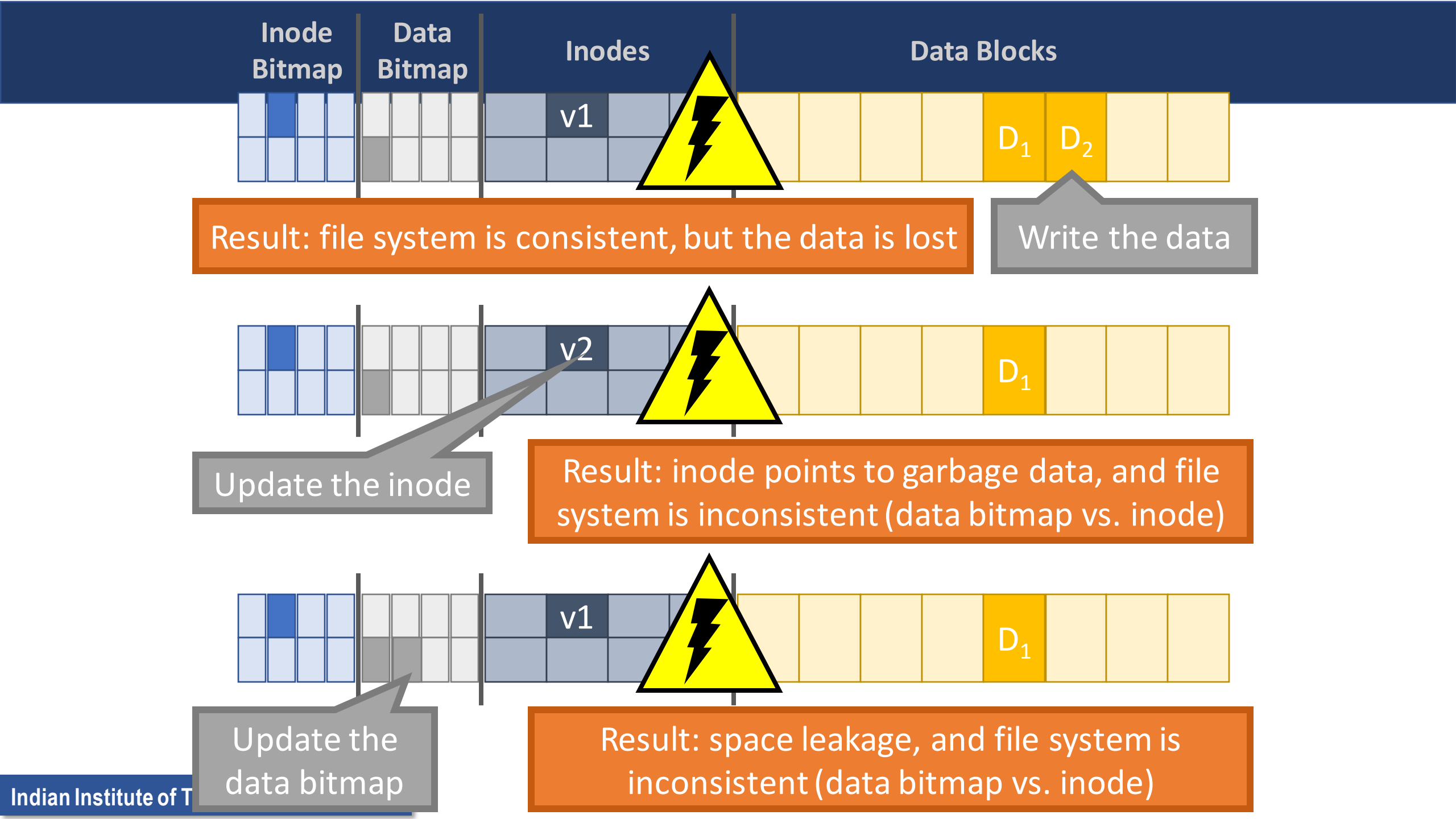
Maintaining Consistency

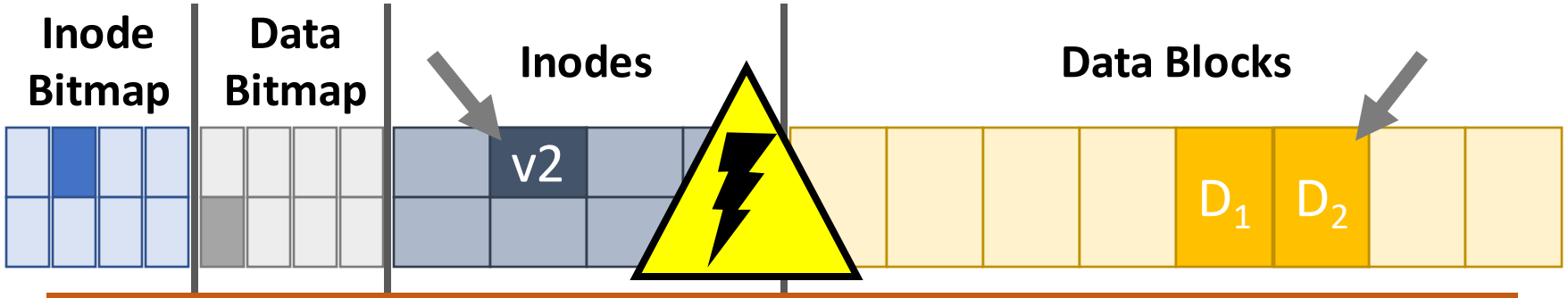
- Many operations results in multiple, independent writes to the file system
  - Example: append a block to an existing file
  1. Update the free data bitmap
  2. Update the inode
  3. Write the user data
- What happens if the computer crashes in the middle of this process?

File Append Example

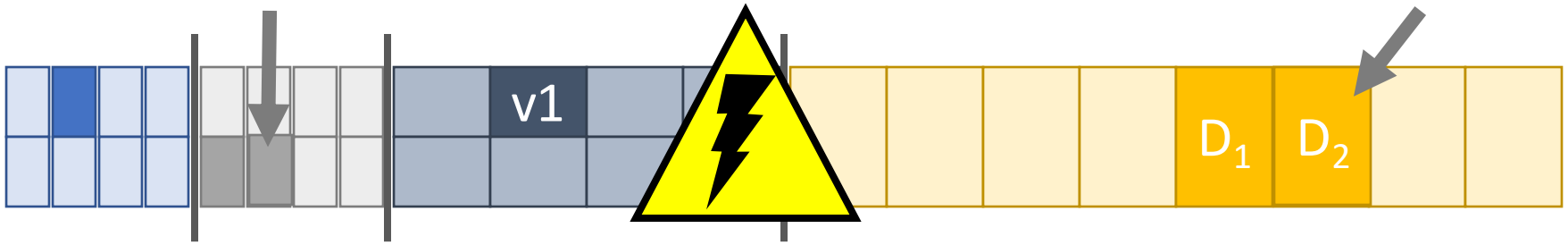


- These three operations can potentially be done in any order
- … but the system can crash at any time

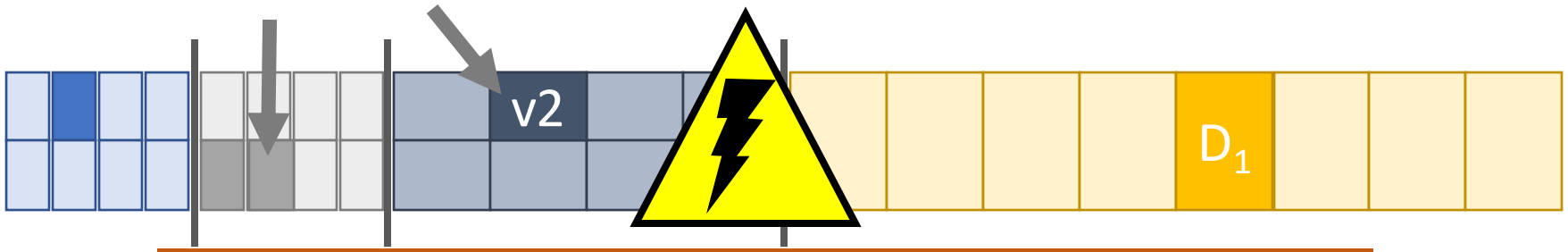

Inode Bitmap
Data Bitmap
Inodes
Data Blocks
v1
D1
D2
Result: file system is consistent, but the data is lost
Write the data
v2
D1
Update the inode
Result: inode points to garbage data, and file system is inconsistent (data bitmap vs. inode)
v1
D1
Update the data bitmap
Result: space leakage, and file system is inconsistent (data bitmap vs. inode)



Result: inode points to data, but file system is inconsistent



Result: file system is inconsistent, and the data is useless since it's not associated with an inode



Result: file system is consistent, but the inode points to garbage data

The Crash Consistency Problem

- The disk guarantees that sector writes are atomic
  - No way to make multi-sector writes atomic
- How to ensure consistency after a crash?
  1. Don't bother to ensure consistency
     - Accept that the file system may be inconsistent after a crash
     - Run a program that fixes the file system during bootup
     - File system checker (*fsck*)
  2. Use a transaction log to make multi-writes atomic
     - Log stores a history of all writes to the disk
     - After a crash the log can be "replayed" to finish updates
     - Journaling file system

Approach 1: File System Checker

- Key idea: fix inconsistent file systems during bootup
  - Unix utility called *fsck* (*chkdsk* on Windows)
  - Scans the entire file system multiple times, identifying and correcting inconsistencies
- Why during bootup?
  - No other file system activity can be going on
  - After fsck runs, bootup/mounting can continue

fsck Tasks

- **Superblock:** validate the superblock, replace it with a backup if it is corrupted
- **Free blocks and inodes:** rebuild the bitmaps by scanning all inodes
- **Reachability:** make sure all inodes are reachable from the root of the file system
- **inodes:** delete all corrupted inodes, and rebuild their link counts by walking the directory tree
- **directories:** verify the integrity of all directories
- … and many other minor consistency checks

fsck: the Good and the Bad

- Advantages of *fsck*
  - Doesn't require the file system to do any work to ensure consistency
  - Makes the file system implementation simpler
- Disadvantages of *fsck*
  - Very complicated to implement the *fsck* program
    - Many possible inconsistencies that must be identified
    - Many difficult corner cases to consider and handle
  - *fsck* is **super slow**
    - Scans the entire file system multiple times
    - Imagine how long it would take to fsck a 40 TB RAID array

Approach 2: Journaling

- Problem: *fsck* is slow because it checks the entire file system after a crash
  - What if we knew where the last writes were before the crash, and just checked those?
- Key idea: make writes transactional by using a write-ahead log
  - Commonly referred to as a journal
- Ext3 and NTFS use journaling

Write-Ahead Log

- Key idea: writes to disk are first written into a log
  - After the log is written, the writes execute normally
  - In essence, the log records transactions
- What happens after a crash…
  - If the writes to the log are interrupted?
    - The transaction is incomplete
    - The user's data is lost, but the file system is consistent
  - If the writes to the log succeed, but the normal writes are interrupted?
    - The file system may be inconsistent, but…
    - The log has exactly the right information to fix the problem

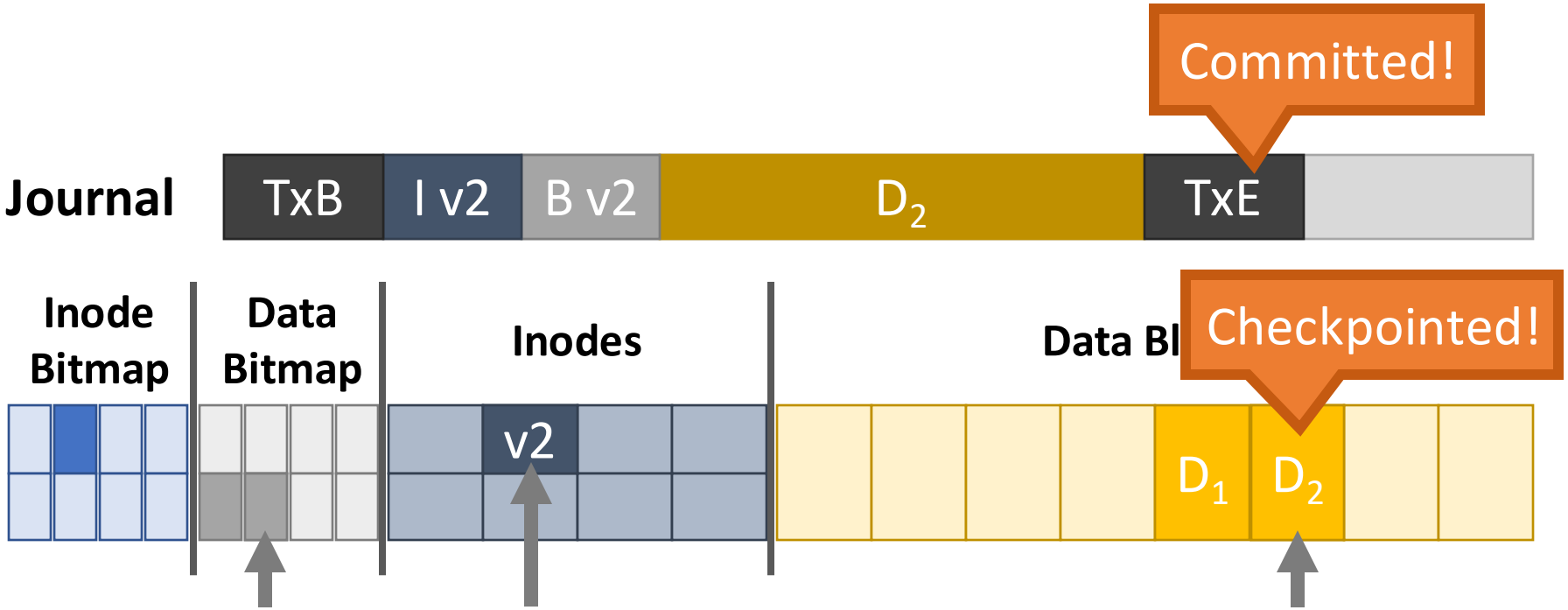
Data Journaling Example

- Assume we are appending to a file
  - Three writes: inode v2, data bitmap v2, data D_2
- Before executing these writes, first log them

1. Begin a new transaction with a unique ID=*k*
2. Write the updated meta-data block(s)
3. Write the file data block(s)
4. Write an end-of-transaction with ID=*k*

Commits and Checkpoints

- We say a transaction is committed after all writes to the log are complete
- After a transaction is committed, the OS checkpoints the update

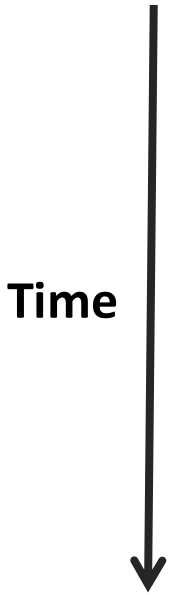


- Final step: free the checkpointed transaction

Journal Implementation

- Journals are typically implemented as a circular buffer
  - Journal is **append-only**
- OS maintains pointers to the front and back of the transactions in the buffer
  - As transactions are freed, the back is moved up
- Thus, the contents of the journal are never deleted, they are just overwritten over time

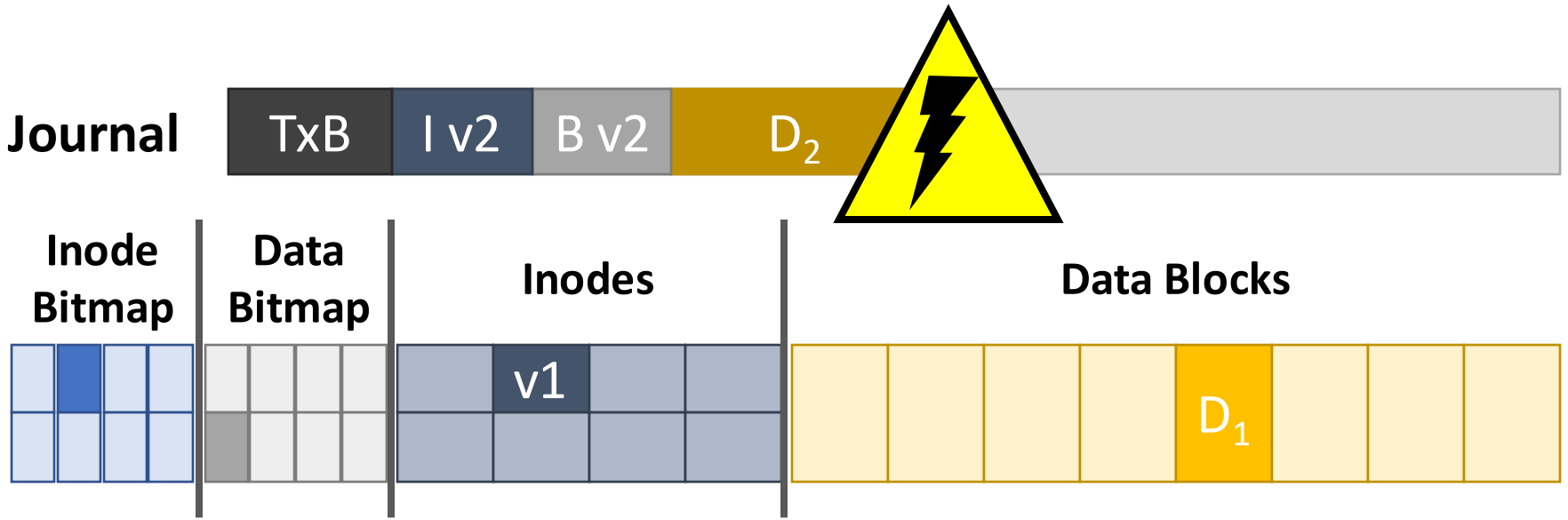
Data Journaling Timeline



Journal				File System	
TxB	**Meta-data**	**Data**	**TxE**	**Meta-data**	**Data**
Issue	Issue	Issue	Issue		
	Complete				
Complete					
			Complete		
		Complete			
				Issue	Issue
					Complete
				Complete	

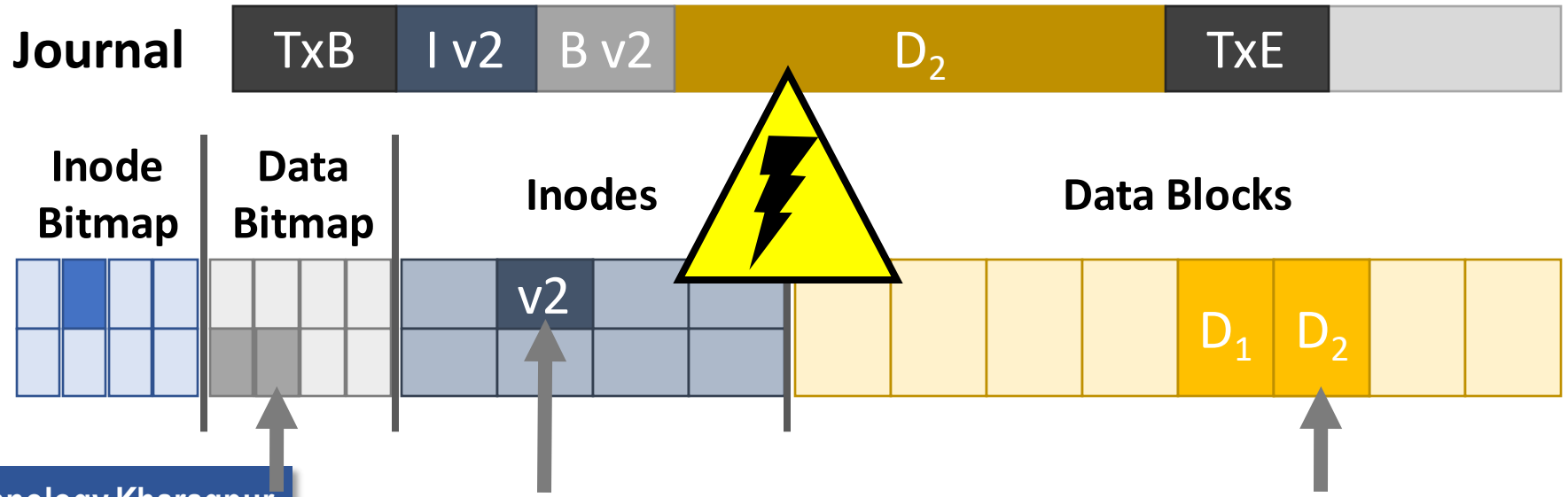
Crash Recovery (1)

- What if the system crashes during logging?
  - If the transaction is not committed, data is lost
  - But, the file system remains consistent



Crash Recovery (2)

- What if the system crashes during the checkpoint?
  - File system may be inconsistent
  - During reboot, transactions that are committed but not free are replayed in order
  - Thus, no data is lost and consistency is restored



Corrupted Transactions

- Problem: the disk scheduler may not execute writes in-order
  - Transactions in the log may appear committed, when in fact they are invalid

- Solution: add a checksum to TxB
- During recovery, reject transactions with invalid checksums
- Implemented on Linux in ext4

- Transaction looks valid, but the data is missing!
- During replay, garbage data is written to the file system

Journaling: The Good and the Bad

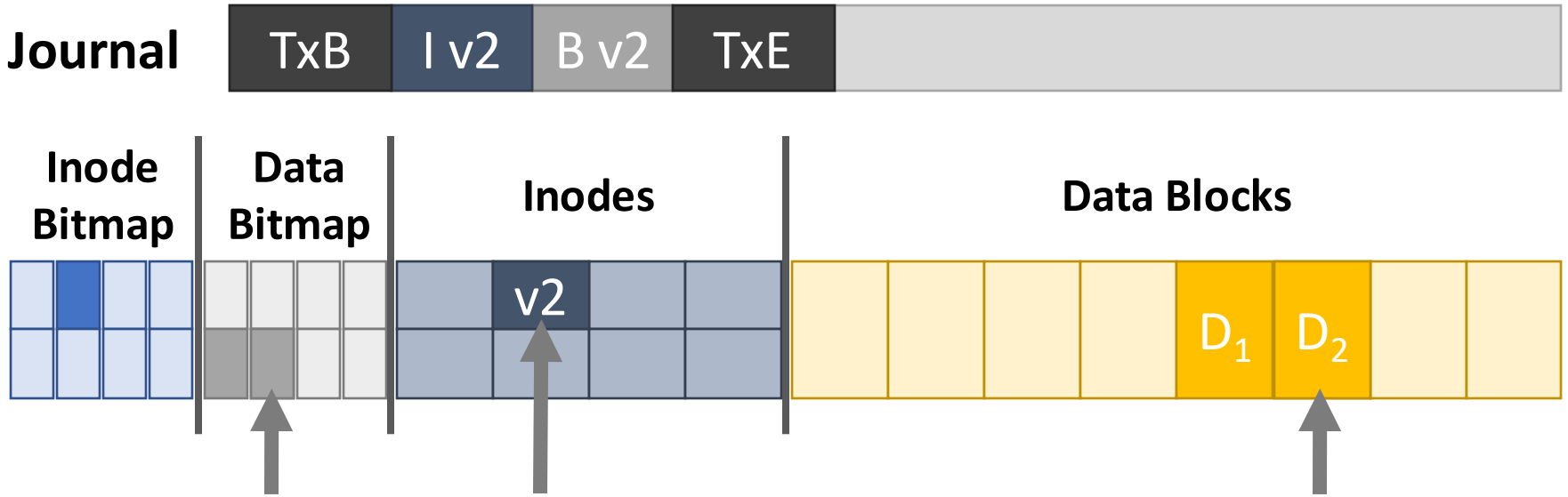
- Advantages of journaling
  - Robust, fast file system recovery
    - No need to scan the entire journal or file system
  - Relatively straight forward to implement
- Disadvantages of journaling
  - Write traffic to the disk is doubled
    - Especially the file data, which is probably large

Making Journaling Faster

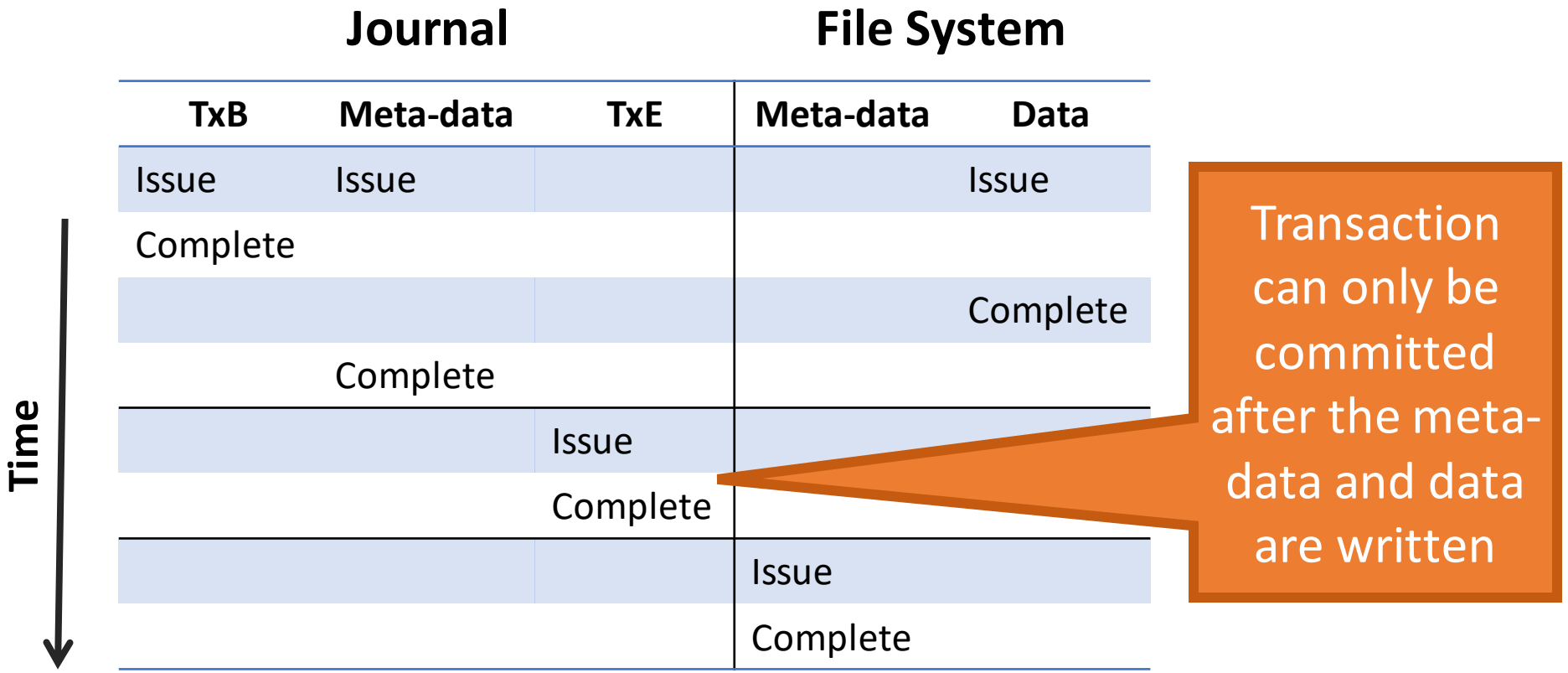
- Journaling adds a lot of write overhead
- OSes typically batch updates to the journal
  - Buffer sequential writes in memory, then issue one large write to the log
  - Example: ext3 batches updates for 5 seconds
- Tradeoff between performance and persistence
  - Long batch interval = fewer, larger writes to the log
    - Improved performance due to large sequential writes
  - But, if there is a crash, everything in the buffer will be lost

Meta-Data Journaling

- The most expensive part of data journaling is writing the file data twice
  - Meta-data is small (~1 sector), file data is large
- ext3 implements meta-data journaling



Meta-Journaling Timeline



Journaling Wrap-Up

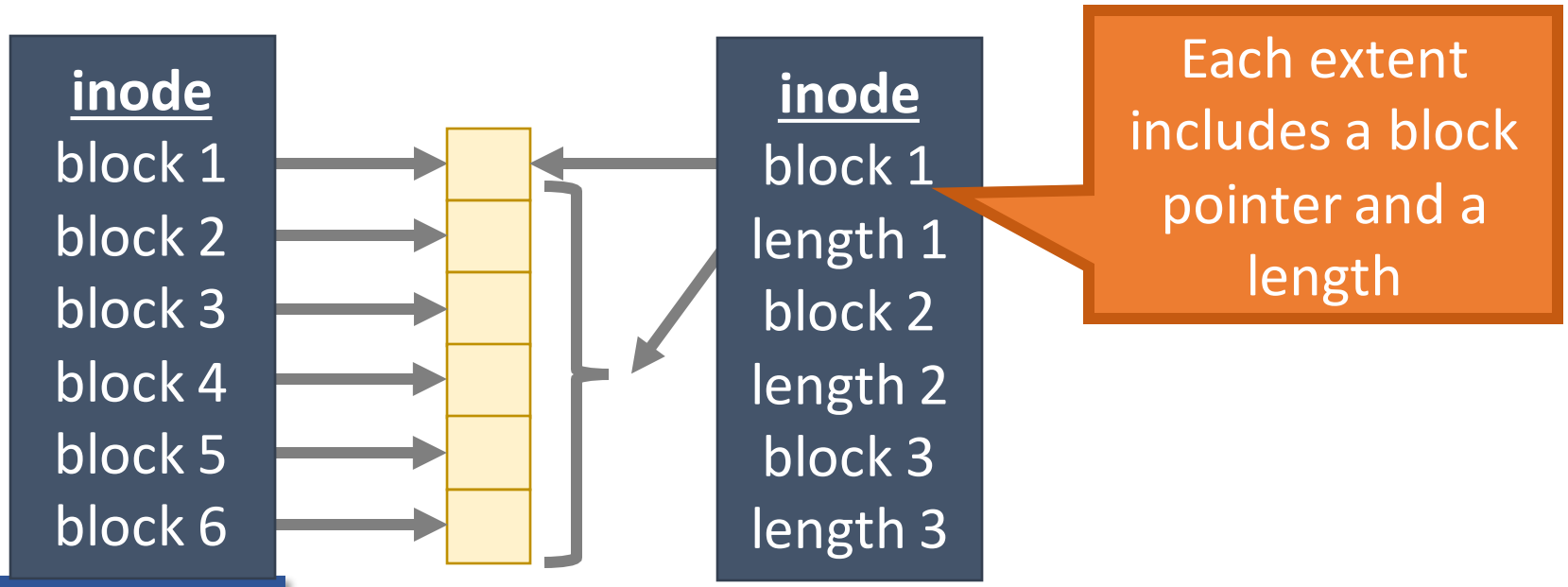
- Today, most OSes use journaling file systems
  - ext3/ext4 on Linux
  - NTFS on Windows
- Provides excellent crash recovery with relatively low space and performance overhead
- Next-gen OSes will likely move to file systems with copy-on-write semantics
  - btrfs and zfs on Linux

Revisiting inodes

- Recall: inodes use indirection to acquire additional blocks of pointers
- Problem: inodes are not efficient for large files
  - Example: for a 100MB file, you need 25600 block pointers (assuming 4KB blocks)
- This is unavoidable if the file is 100% fragmented
  - However, what if large groups of blocks are contiguous?

From Pointers to Extents

- Modern file systems try hard to minimize fragmentation
  - Since it results in many seeks, thus low performance
- Extents are better suited for contiguous files



Implementing Extents

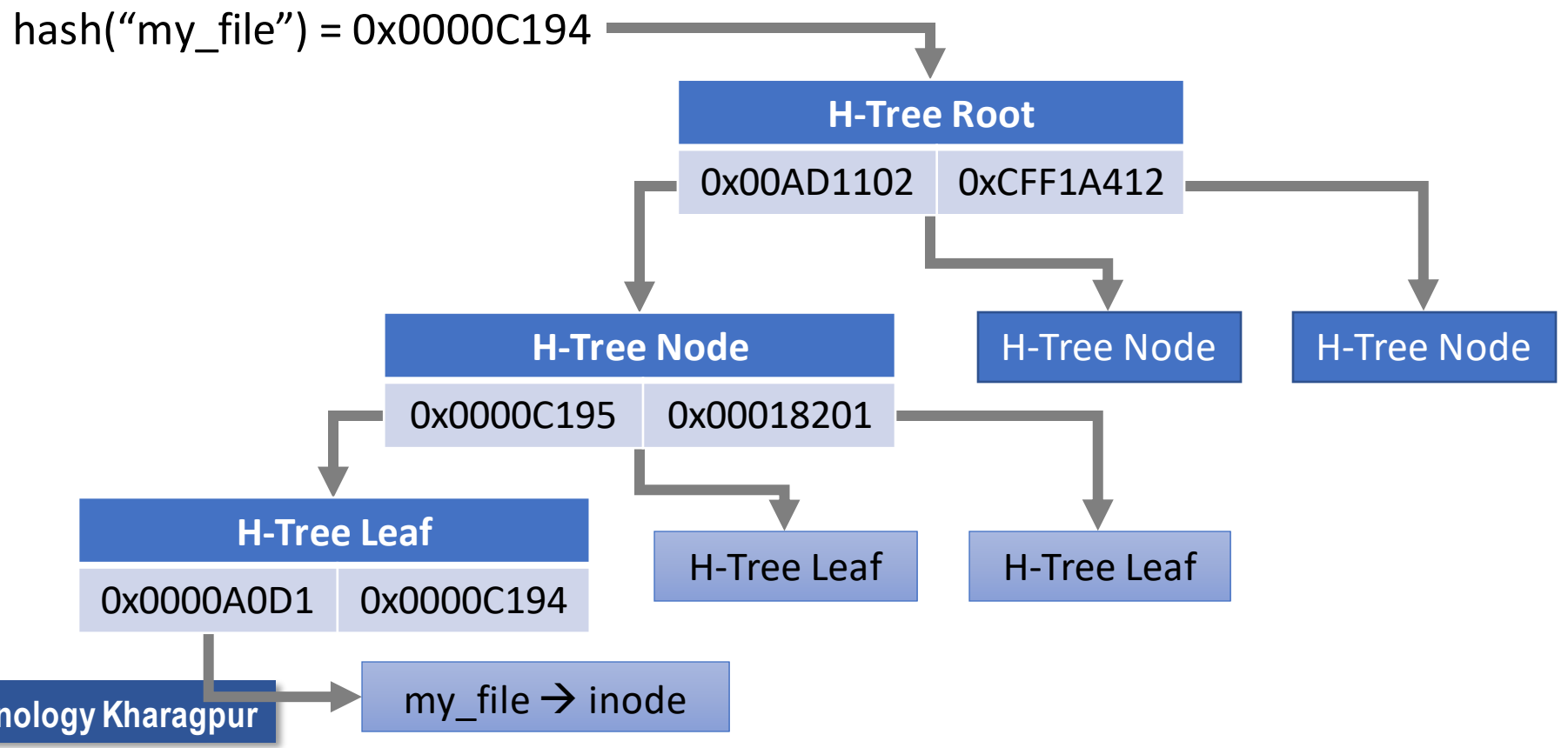
- ext4 and NTFS use extents
- ext4 inodes include 4 extents instead of block pointers
  - Each extent can address at most 128MB of contiguous space (assuming 4KB blocks)
  - If more extents are needed, a data block is allocated
  - Similar to a block of indirect pointers

Revisiting Directories

- In ext, ext2, and ext3, each directory is a file with a list of entries
  - Entries are not stored in sorted order
  - Some entries may be blank, if they have been deleted
- Problem: searching for files in large directories takes O(n) time
  - Practically, you can't store >10K files in a directory
  - It takes way too long to locate and open files
- ext4 and NTFS encode directories as B-Trees to improve lookup time to O(log N)

Example B-Tree

- ext4 uses a B-Tree variant known as a H-Tree
  - The *H* stands for *hash* (sometime called B+Tree)
- Suppose you try to open("my_file", "r")

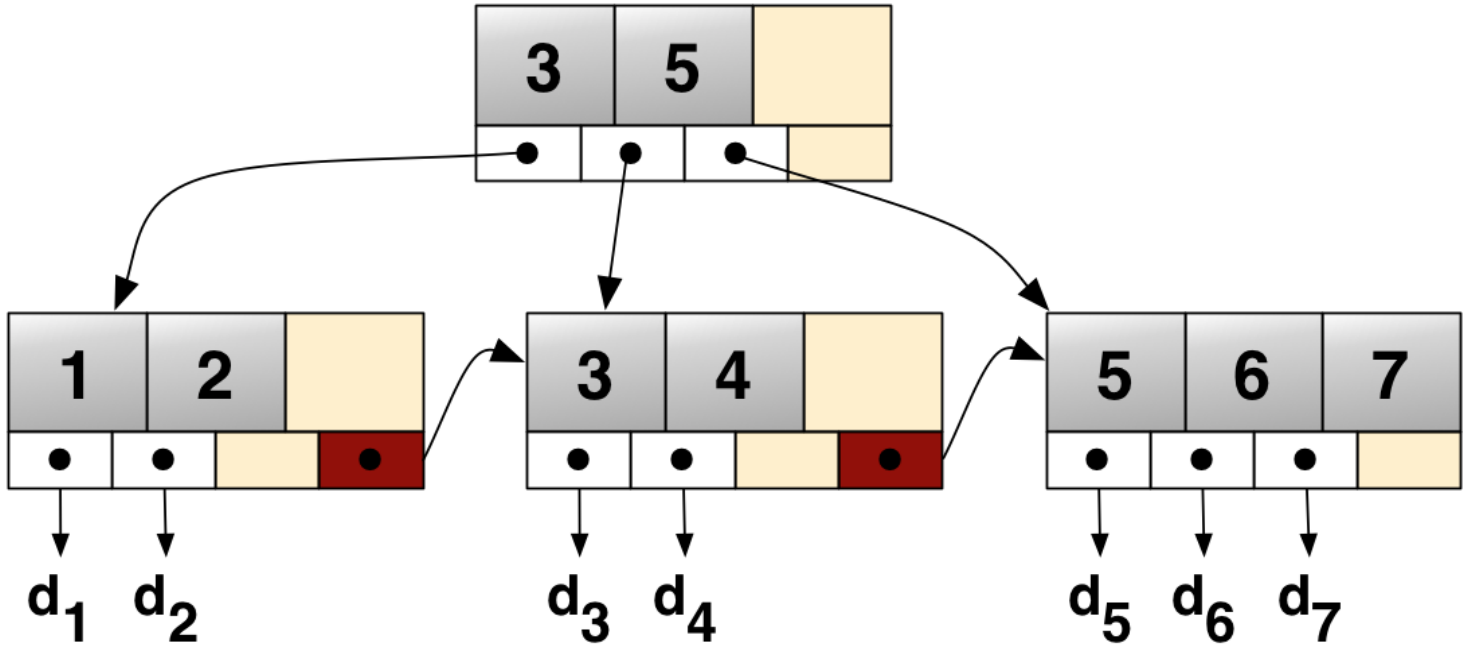


Copy on Write (COW) and B (Read B+) Trees

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

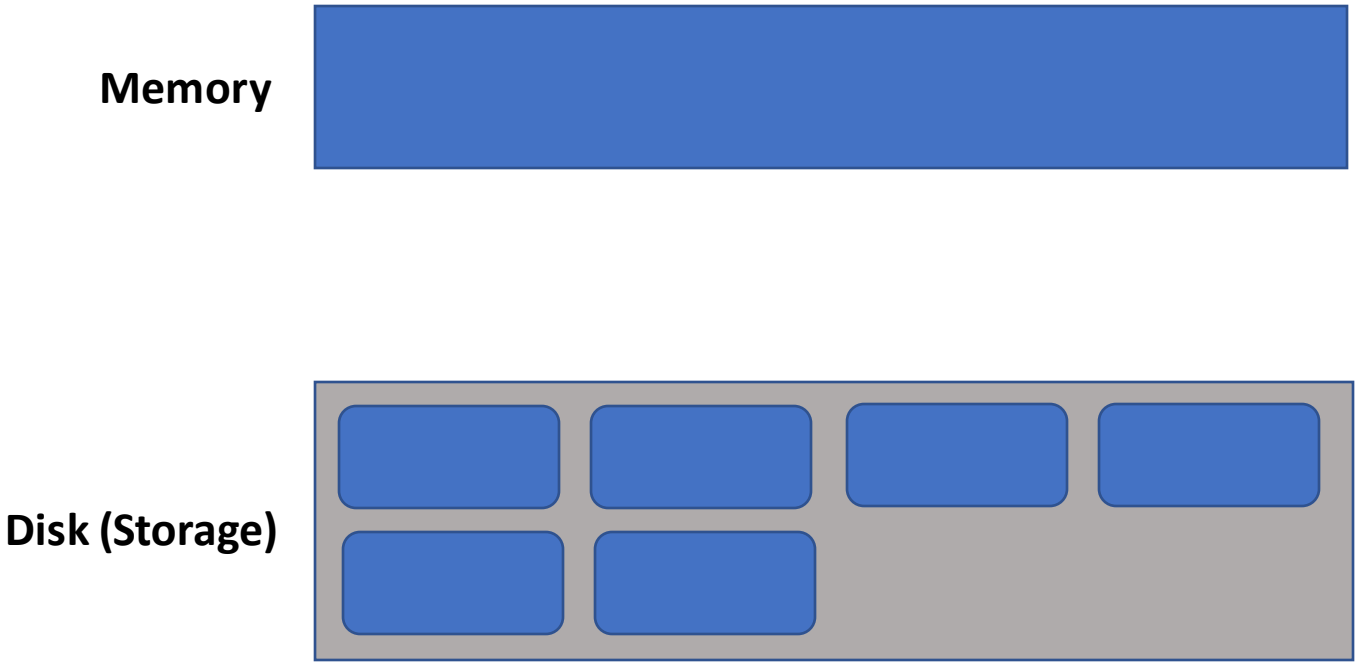


B Trees in File Systems

- B Trees are widely used for file system representation (WAFL, ZFS, BTRFS)
  - Logarithmic time key search, insert and remove
  - Well represents sparse files
- The File System as a large tree made up of fixed size pages
- **Shadowing:** Technique to support atomic updates over persistent data structures
  - Implement snapshots, crash recovery, write-batching, RAID

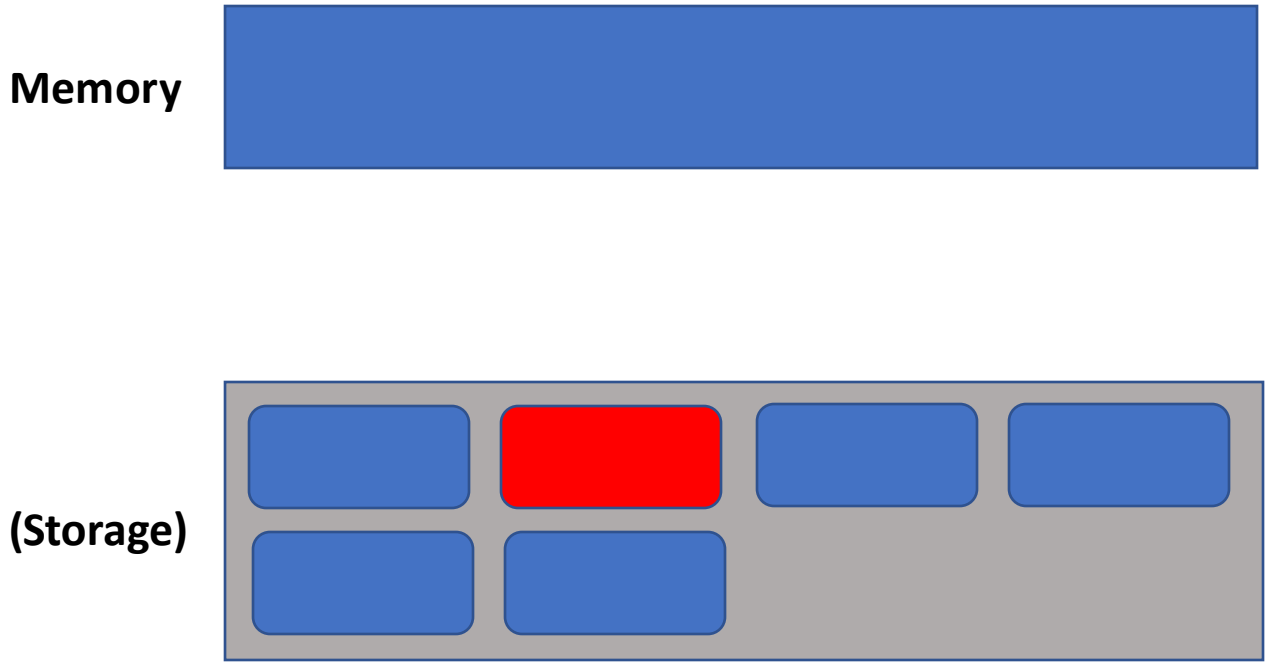
Shadowing

- To update an on-disk page (the page is in the disk, not available in the memory)
  - Read the entire page in the memory
  - Modify the page
  - Write in an alternate location



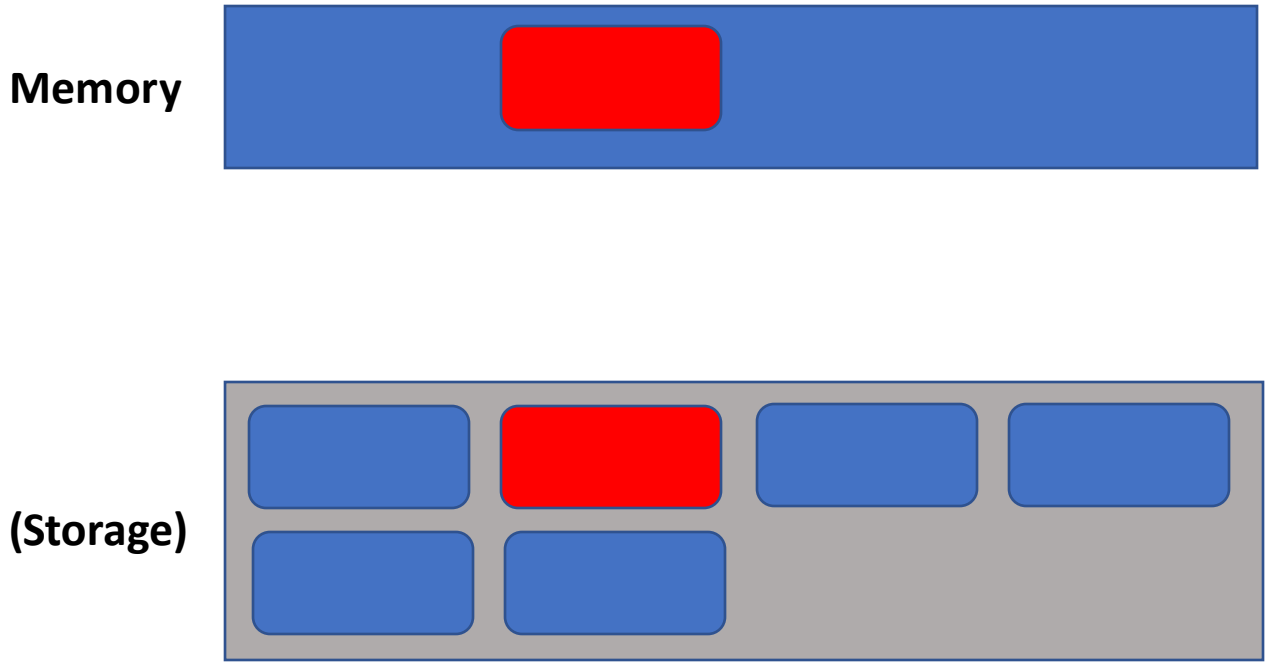
Shadowing

- To update an on-disk page (the page is in the disk, not available in the memory)
  - Read the entire page in the memory
  - Modify the page
  - Write in an alternate location



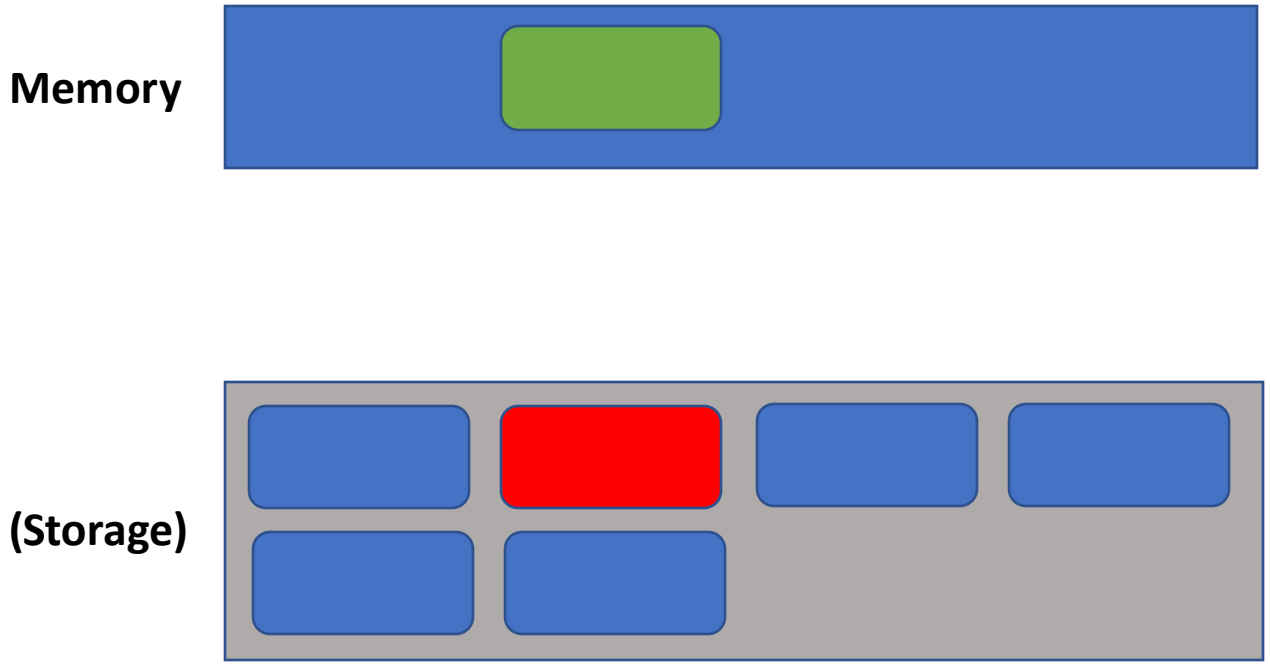
Shadowing

- To update an on-disk page (the page is in the disk, not available in the memory)
  - Read the entire page in the memory
  - Modify the page
  - Write in an alternate location



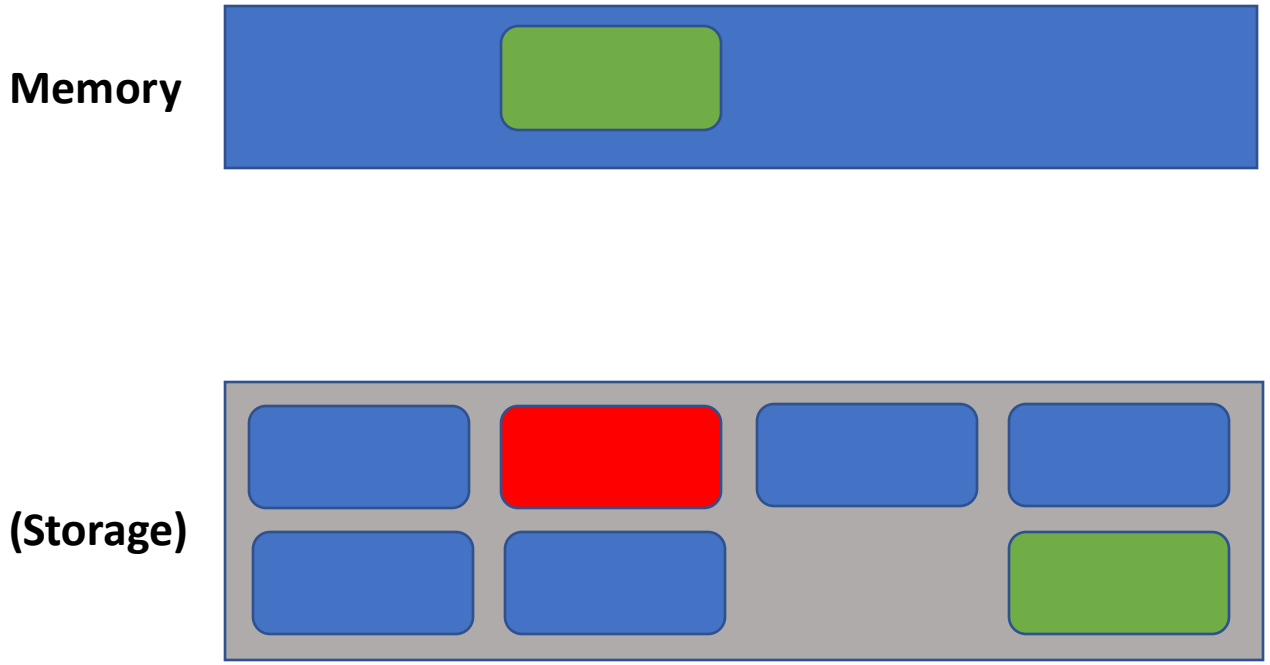
Shadowing

- To update an on-disk page (the page is in the disk, not available in the memory)
  - Read the entire page in the memory
  - Modify the page
  - Write in an alternate location



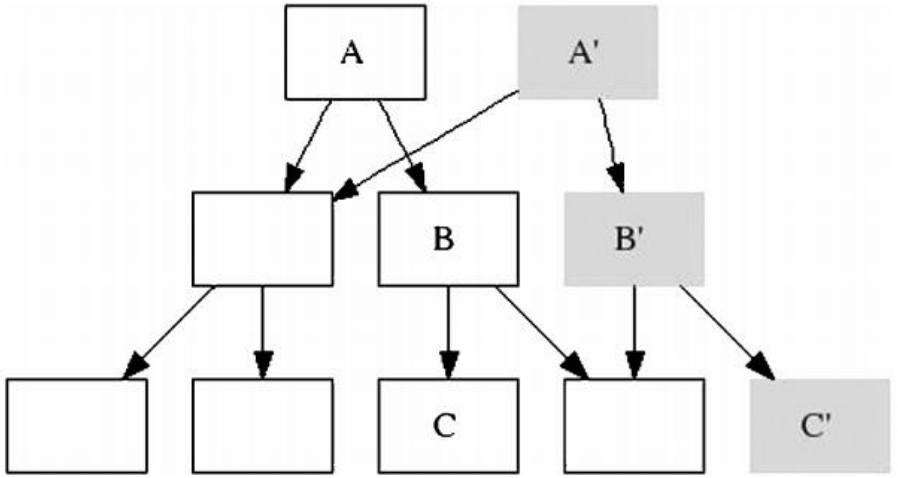
Shadowing

- To update an on-disk page (the page is in the disk, not available in the memory)
  - Read the entire page in the memory
  - Modify the page
  - Write in an alternate location



Shadowing

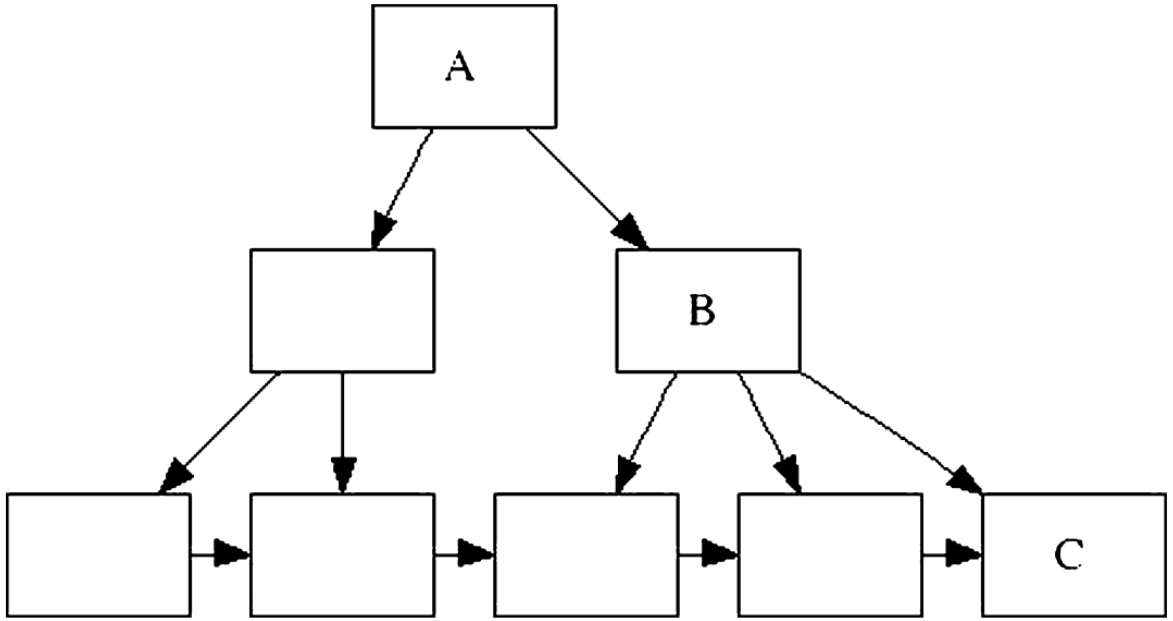
- To update an on-disk page (the page is in the disk, not available in the memory)
  - Read the entire page in the memory
  - Modify the page
  - Write in an alternate location



- When a page is shadowed, its location on the disk changes
  - Update (and shadow) the immediate ancestor of the page with the new address
  - Propagates up to the file system root

Shadowing over B Trees

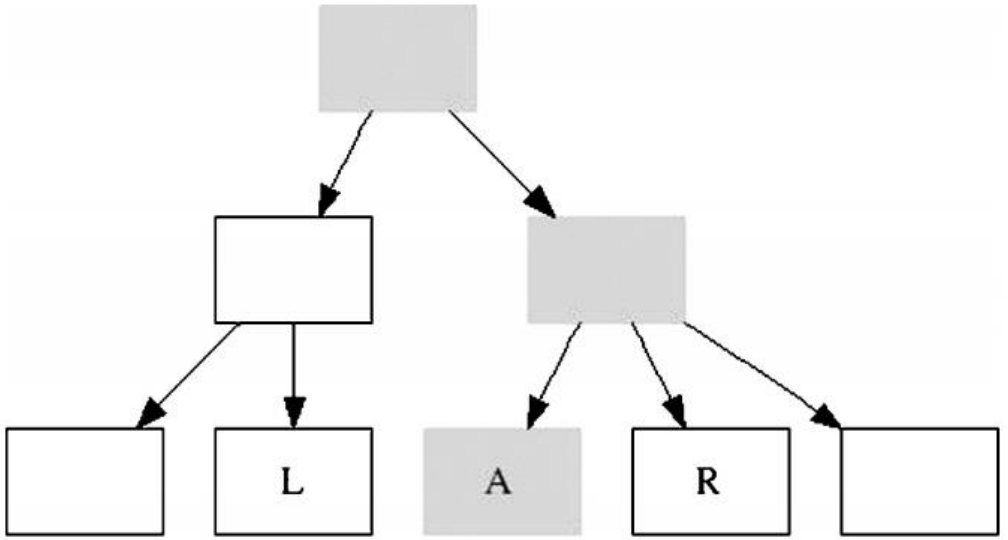
- **Leaf Chaining**
  - Used in B-trees for tree rebalancing and range lookup
  - COW needs entire B Tree to be modified



Shadowing over B Trees

- **Concurrency**
  - Only leaf changes in a regular B Tree updates
  - COW needs locks up to the root

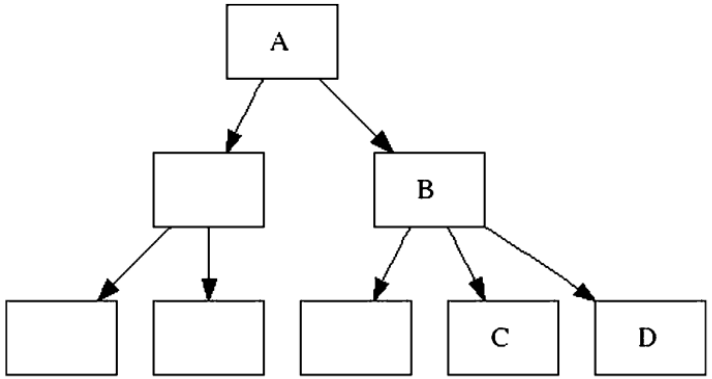
- **Modifying a Single Path**
  - Regular B Trees shuffle keys between neighbors for rebalancing after a "remove key"
  - COW makes this expensive



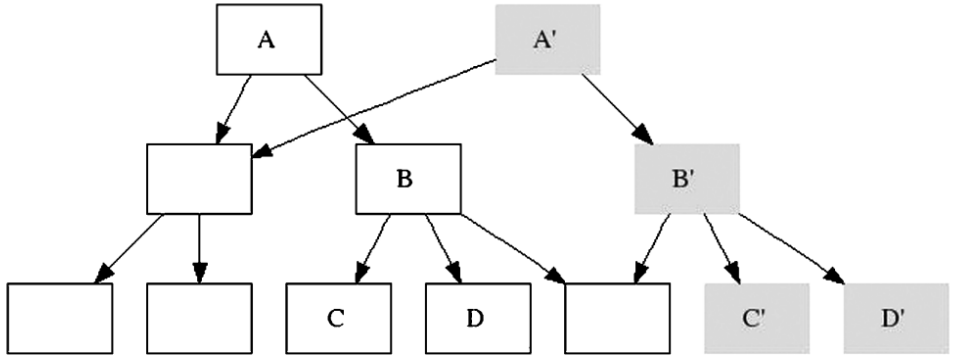
Checkpoint and Recoverability

- Shadowing file systems ensure recoverability by taking periodic checkpoints, and logging commands in between

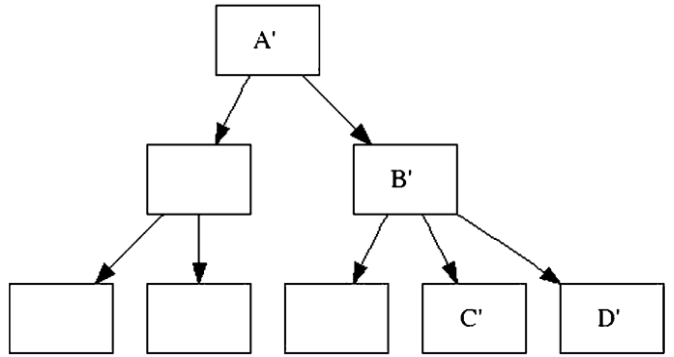
- Checkpoint includes the entire file system tree
  - Once a checkpoint is taken on the disk, the previous checkpoint can be removed
  - In case of a crash, go back to the last saved checkpoint and replay the log



(a) initial file system tree



(b) modifications



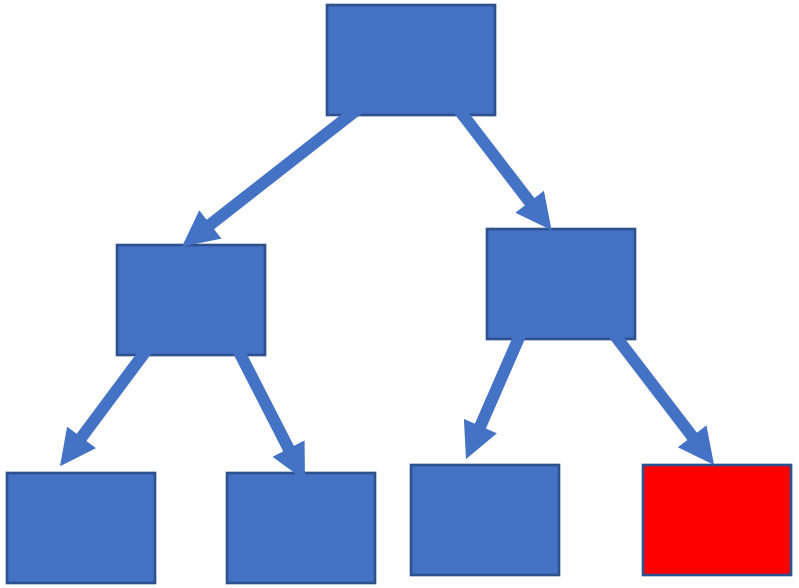
(c) new checkpoint

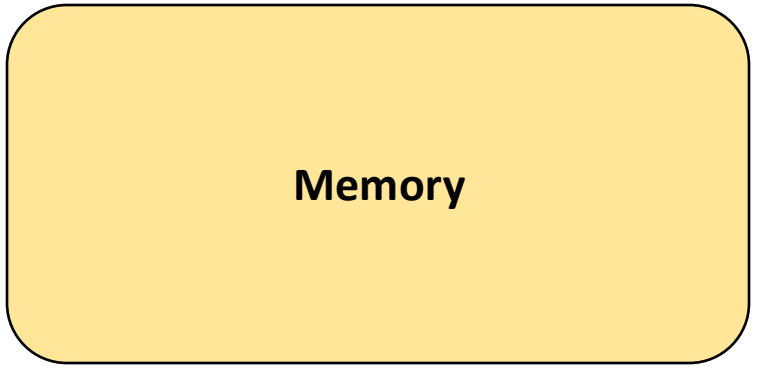
COW with Checkpoints

- Checkpoint makes COW efficient – modifications can be batched and written sequentially in the disk
- Command logging combines multiple operations on the file systems on a single log entry
- When a page belonging to a checkpoint is first shadowed, a cached copy of it is created and held in memory
  - All modifications to the page can be done on the cached shadow copy
  - The dirty pages can be held in the memory until the next checkpoint (assuming that there is sufficient memory)
  - For swap out of pages, write the pages to the shadow location of the disk

Shadowing and Batch Update

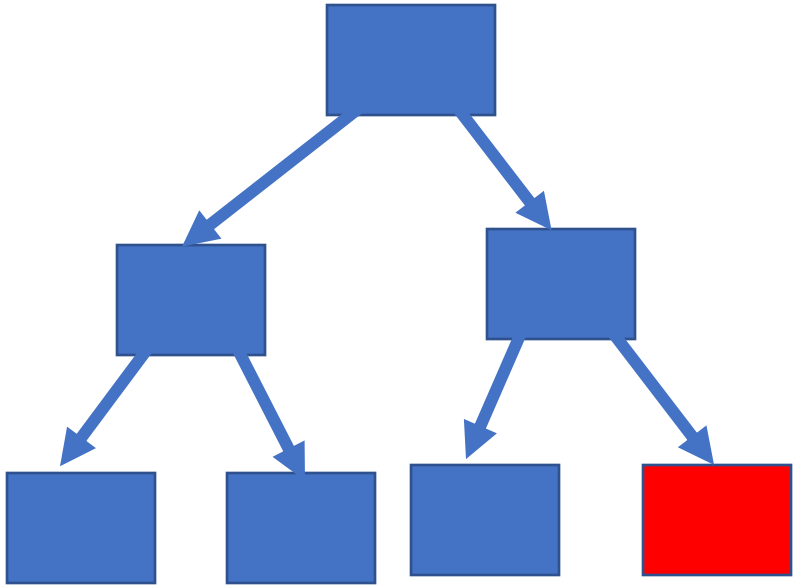
- A page is loaded in the memory and all the updates are performed on it

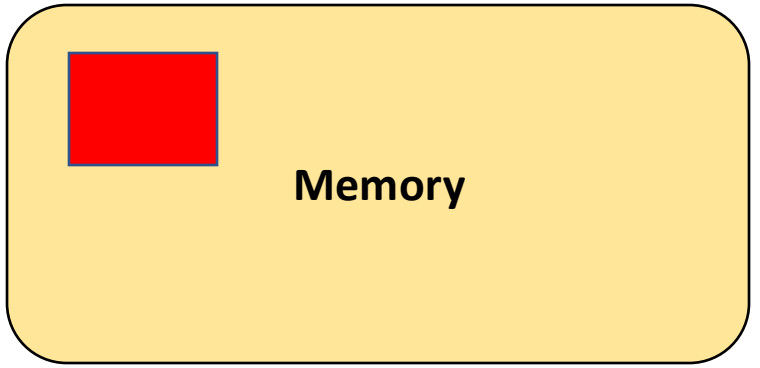




Shadowing and Batch Update

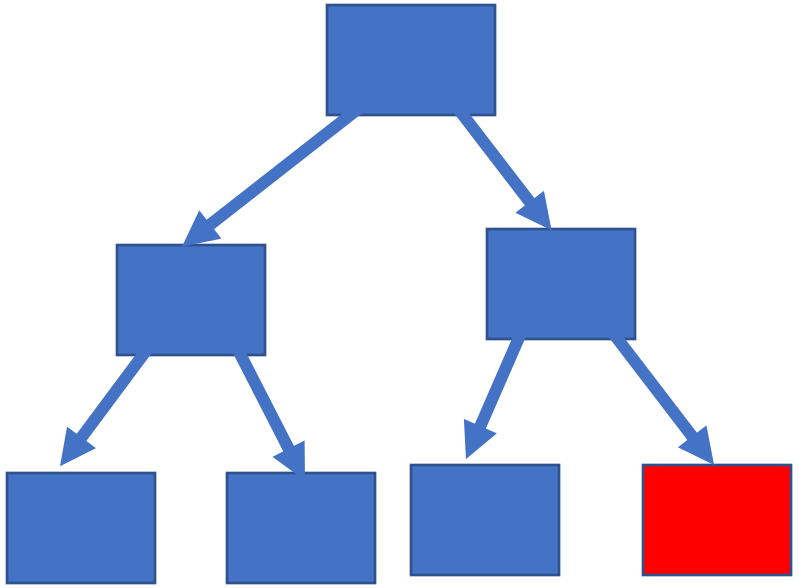
- A page is loaded in the memory and all the updates are performed on it

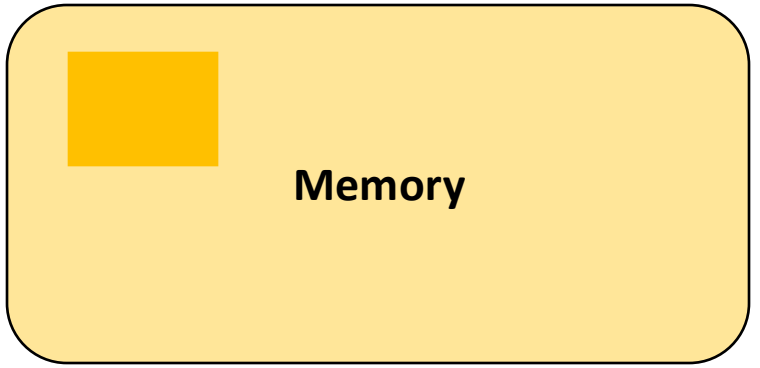




Shadowing and Batch Update

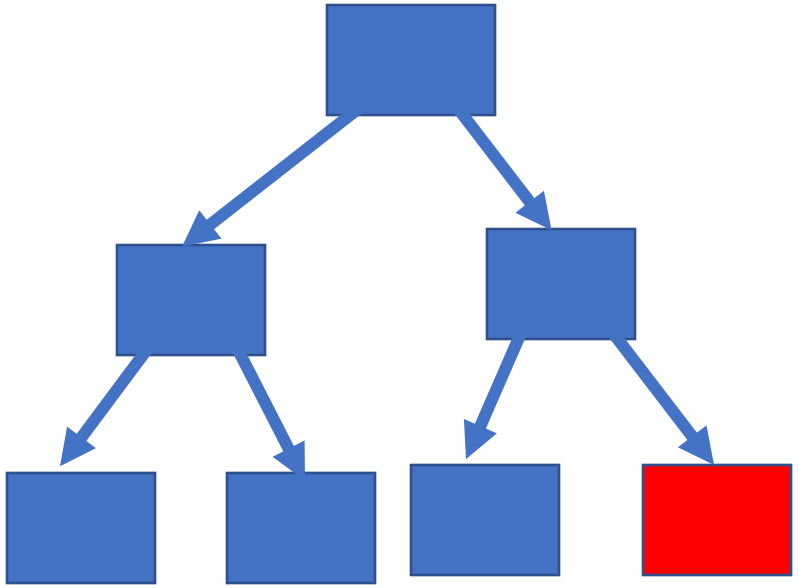
- A page is loaded in the memory and all the updates are performed on it

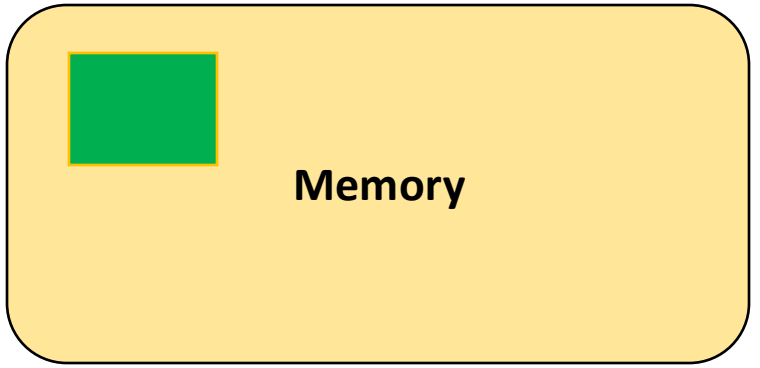




Shadowing and Batch Update

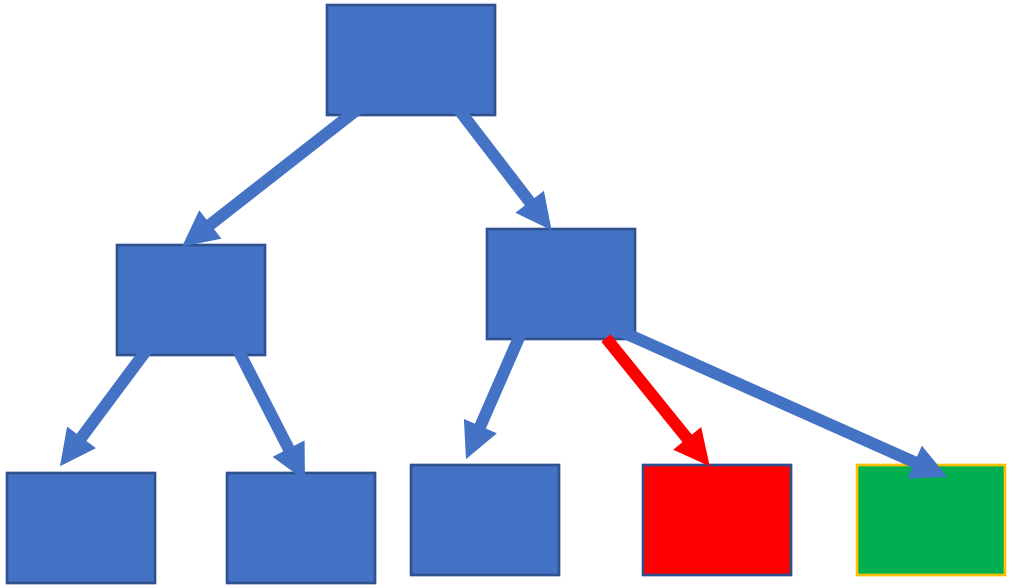
- A page is loaded in the memory and all the updates are performed on it

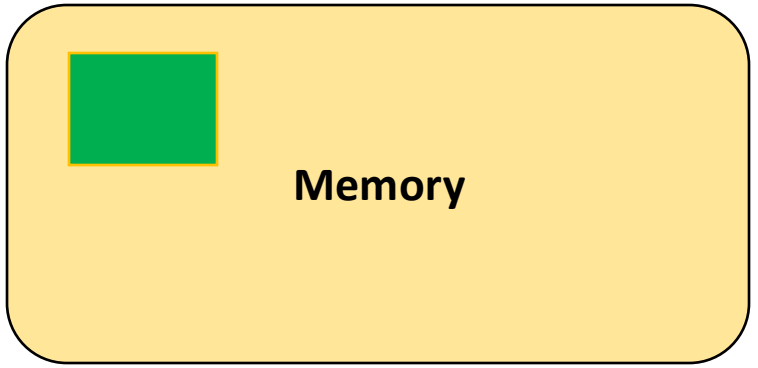




Shadowing and Batch Update

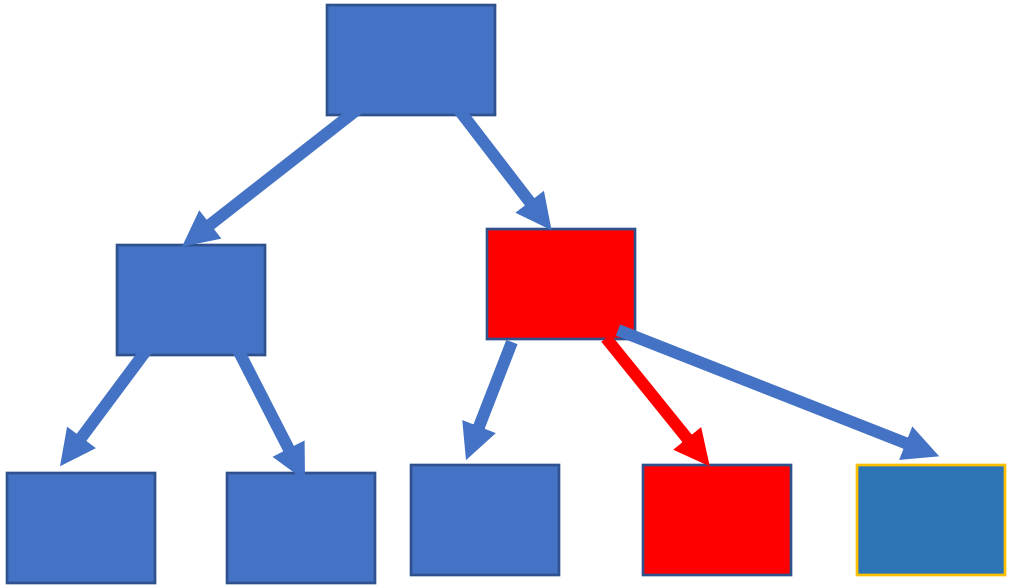
- A page is loaded in the memory and all the updates are performed on it
- During the checkpoint, load its parent page for the update of the address.

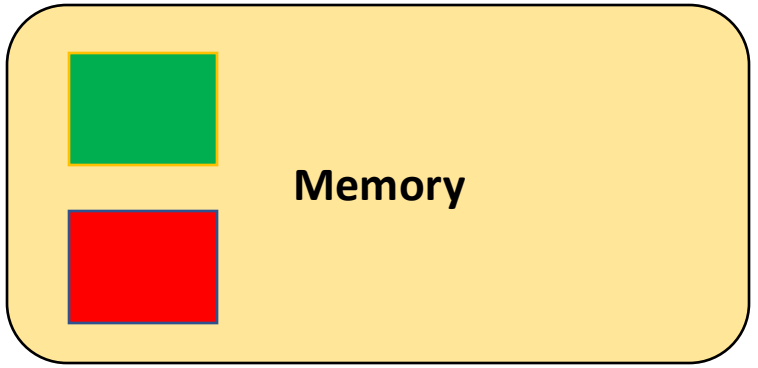




Shadowing and Batch Update

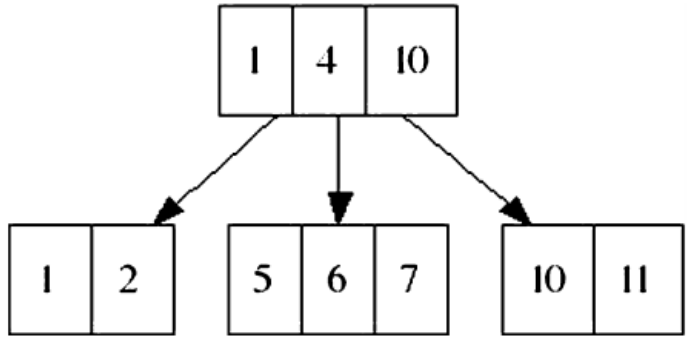
- A page is loaded in the memory and all the updates are performed on it
- During the checkpoint, load its parent page for the update of the address.





B (Typically B+) Tree Insertion

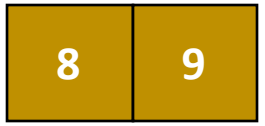
- Normally between *b* and *2b-1* entries per node
- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two
- Assume *b=2*

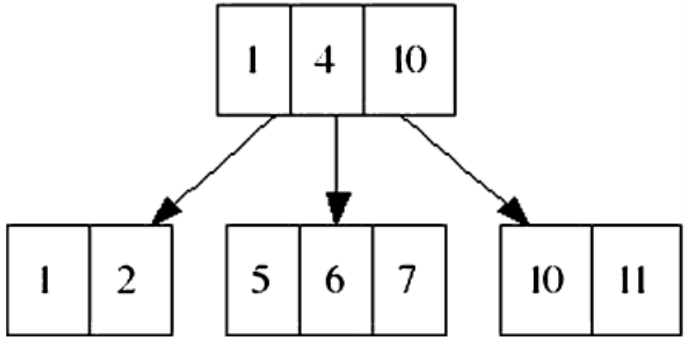


8

B (Typically B+) Tree Insertion

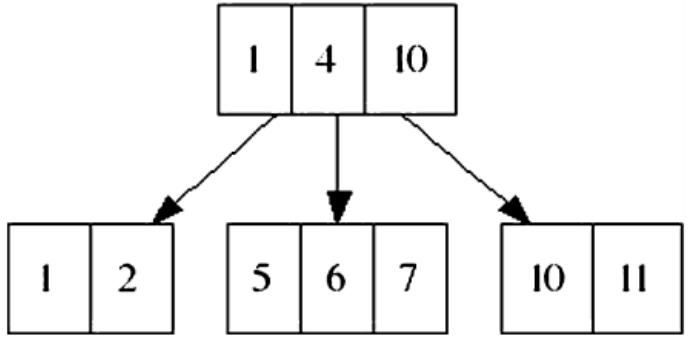
- Normally between *b* and *2b-1* entries per node
- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two
- Assume *b=2*





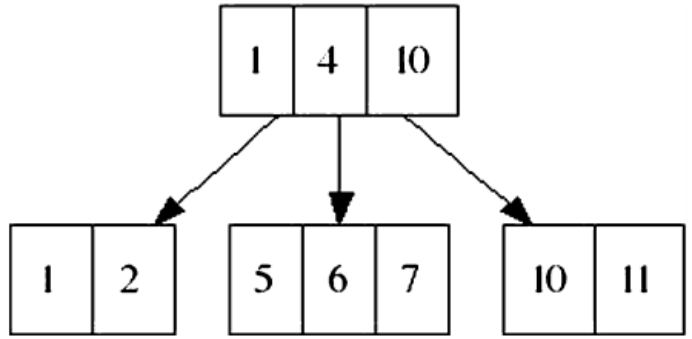
B (Typically B+) Tree Insertion

- Normally between *b* and *2b-1* entries per node
- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two
- Assume *b=2*



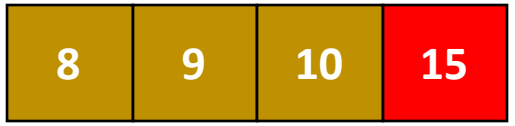
B (Typically B+) Tree Insertion

- Normally between *b* and *2b-1* entries per node



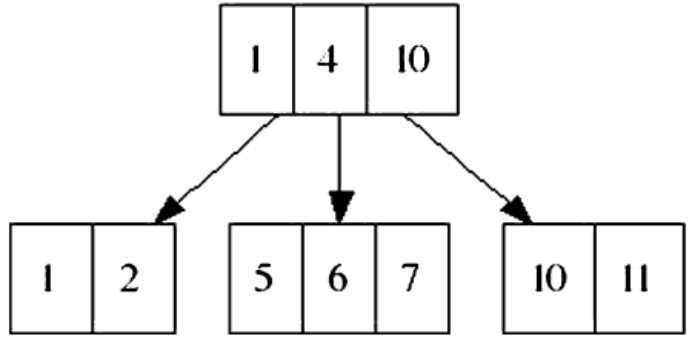
- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two

- Assume *b=2*



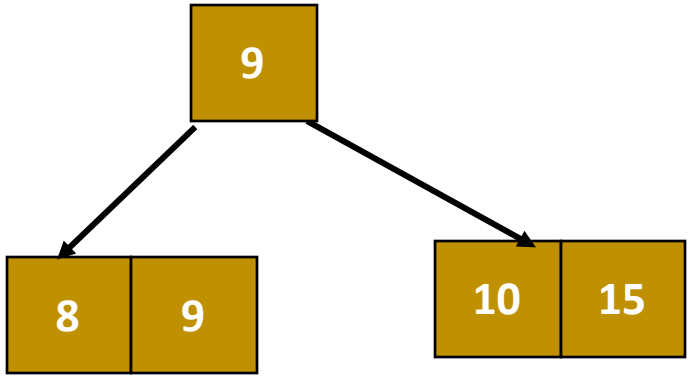
B Tree Insertion

- Normally between *b* and *2b-1* entries per node



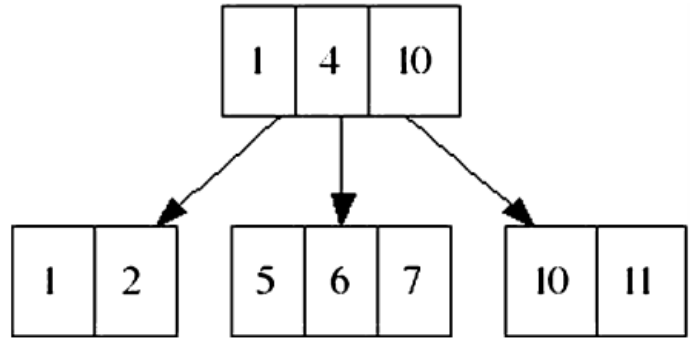
- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two

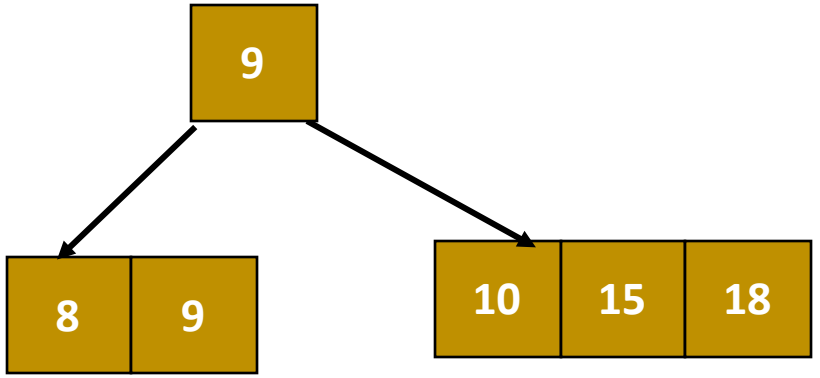
- Assume *b=2*



B Tree Insertion

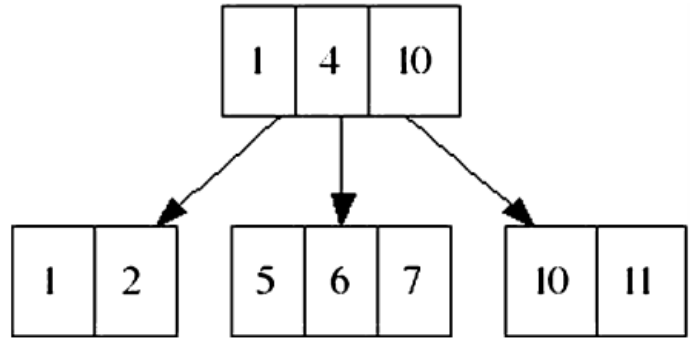
- Normally between *b* and *2b-1* entries per node
- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two
- Assume *b=2*





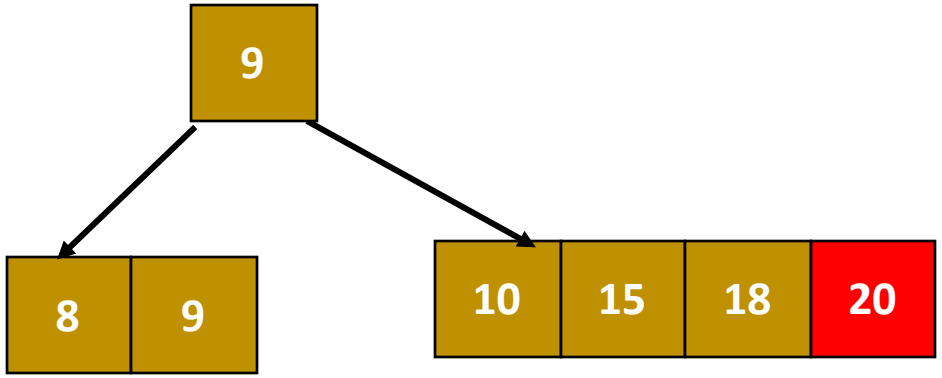
B Tree Insertion

- Normally between *b* and *2b-1* entries per node



- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two

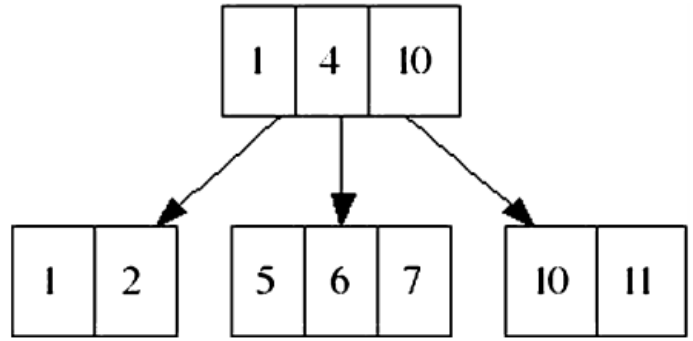
- Assume *b=2*



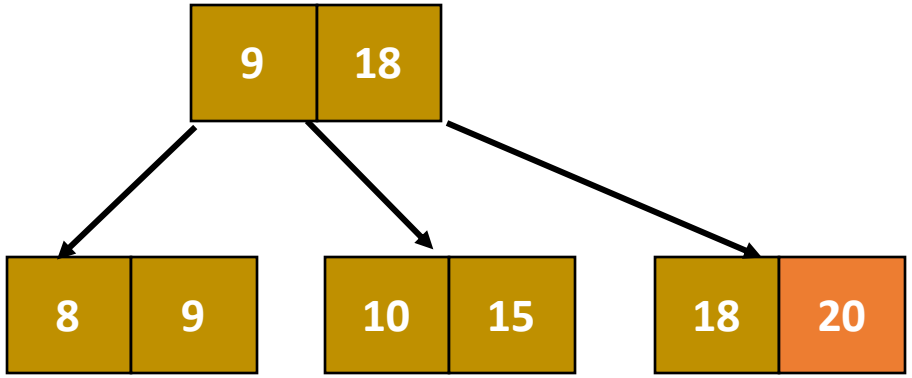
B Tree Insertion

- Normally between *b* and *2b-1* entries per node

- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two

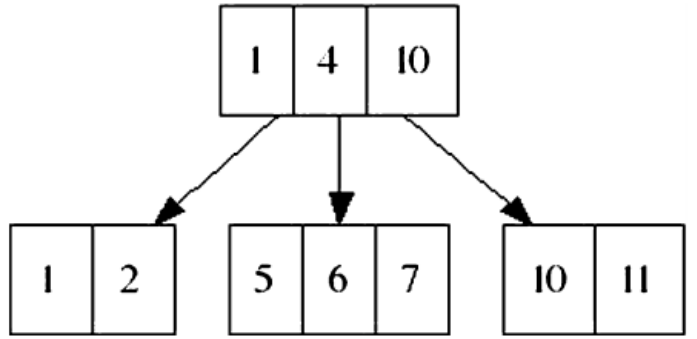


- Assume *b=2*



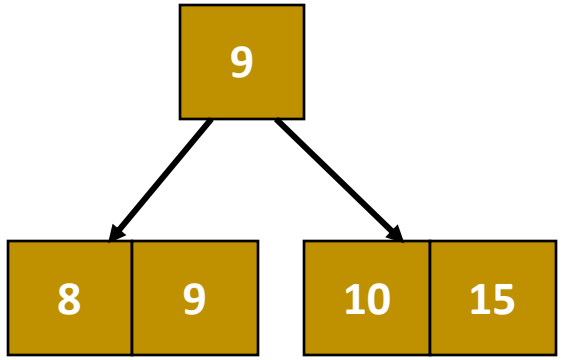
B Tree Insertion

- Normally between *b* and *2b-1* entries per node



- During the insertion, if the entries to a node becomes more than *2b-1,* then the node is split into two

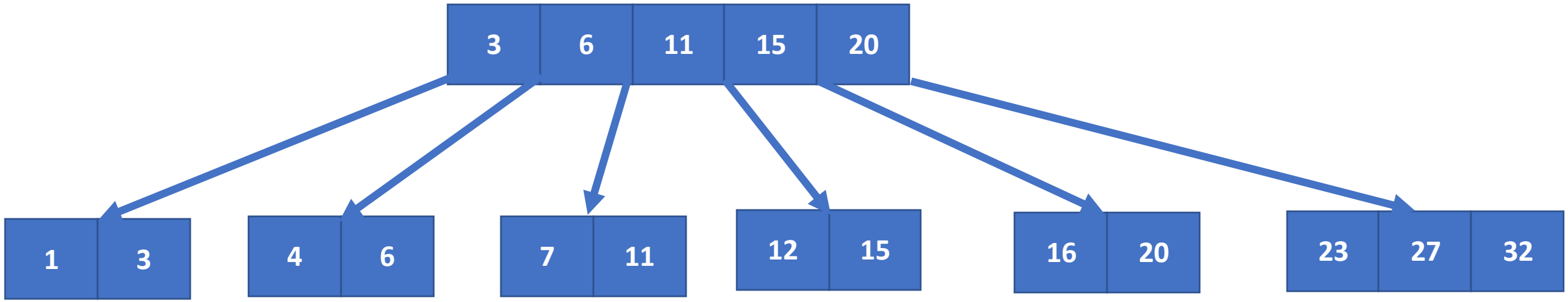
- **Problems with COW**
  - The updates may get back-propagated up to the root
  - All the intermediate pages need to be read to the memory (affect the batch update)
  - Further, the intermediate nodes might be locked due to another concurrent shadowing – the update gets delayed until the previous lock is released



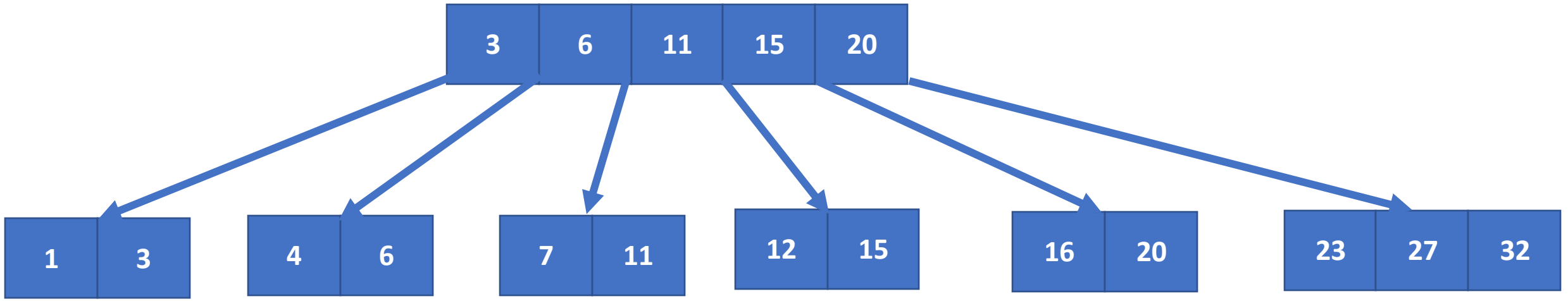
B Tree Operations (COW Friendly)

- Use a proactive approach for tree rebalancing
  - Take a top-down approach instead of a bottom-up approach
- Relax the constraints
  - A node may contain between b and $2b+1$ *keys* for $b \geq 2$
- During the insert-key operation
  - When a node with $2b+1$ entries is encountered, it is split
- During the remove-key operation
  - When a node with b entries is encountered, it is merged

A B-Tree with *b* and *2b+1* keys (*b=2*)

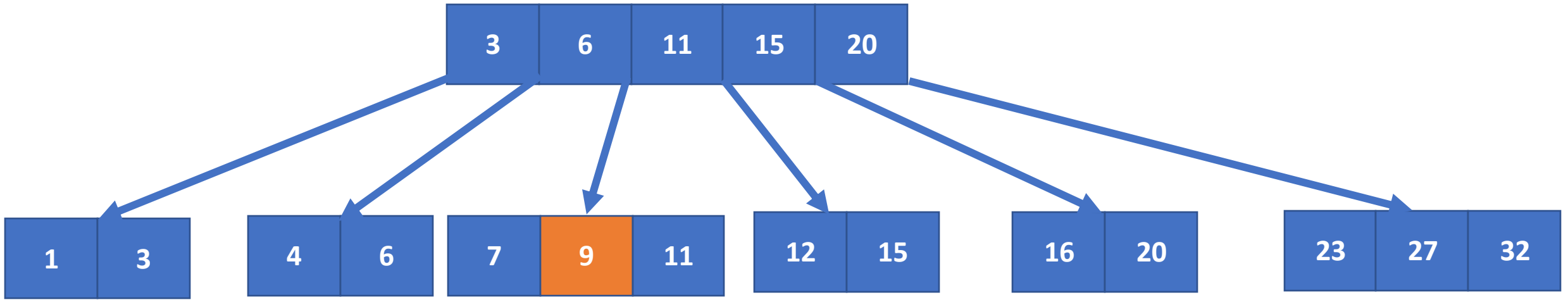


A B-Tree with *b* and *2b+1* keys (*b*=2)

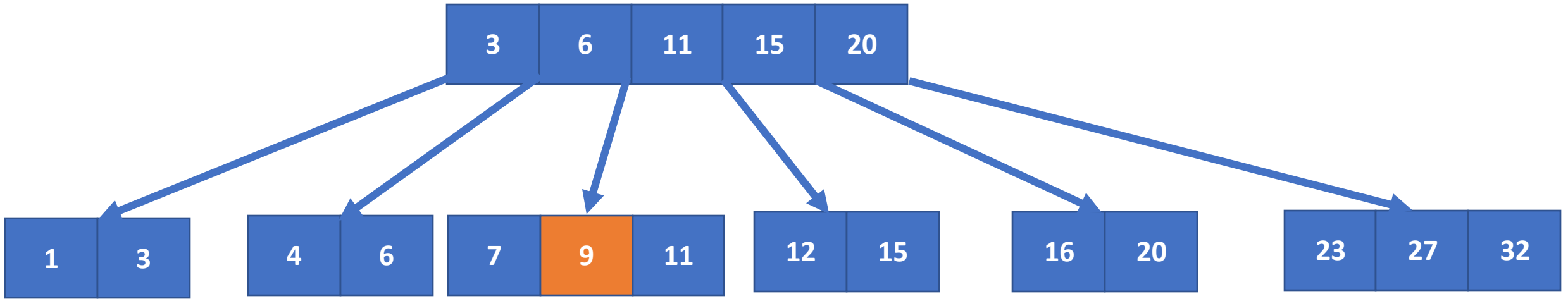


Insert 9 to this tree – what will be the normal operation?

Insert 9 (with normal operations)

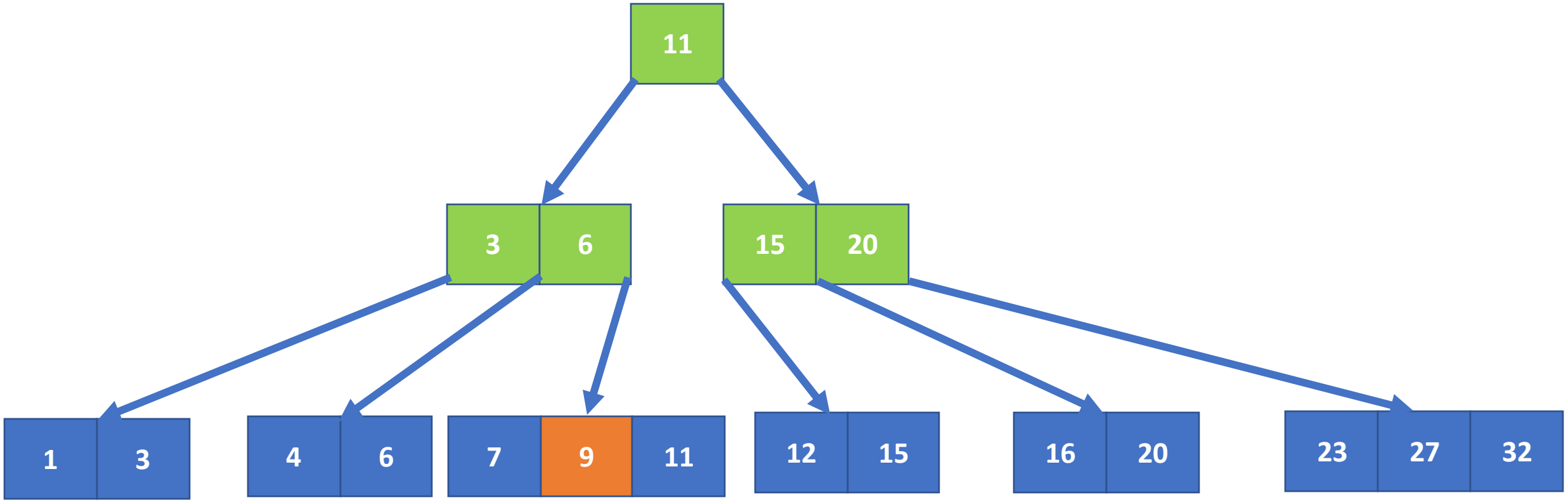


Insert 9 (with normal operations)



What will be the tree structure with proactive split?

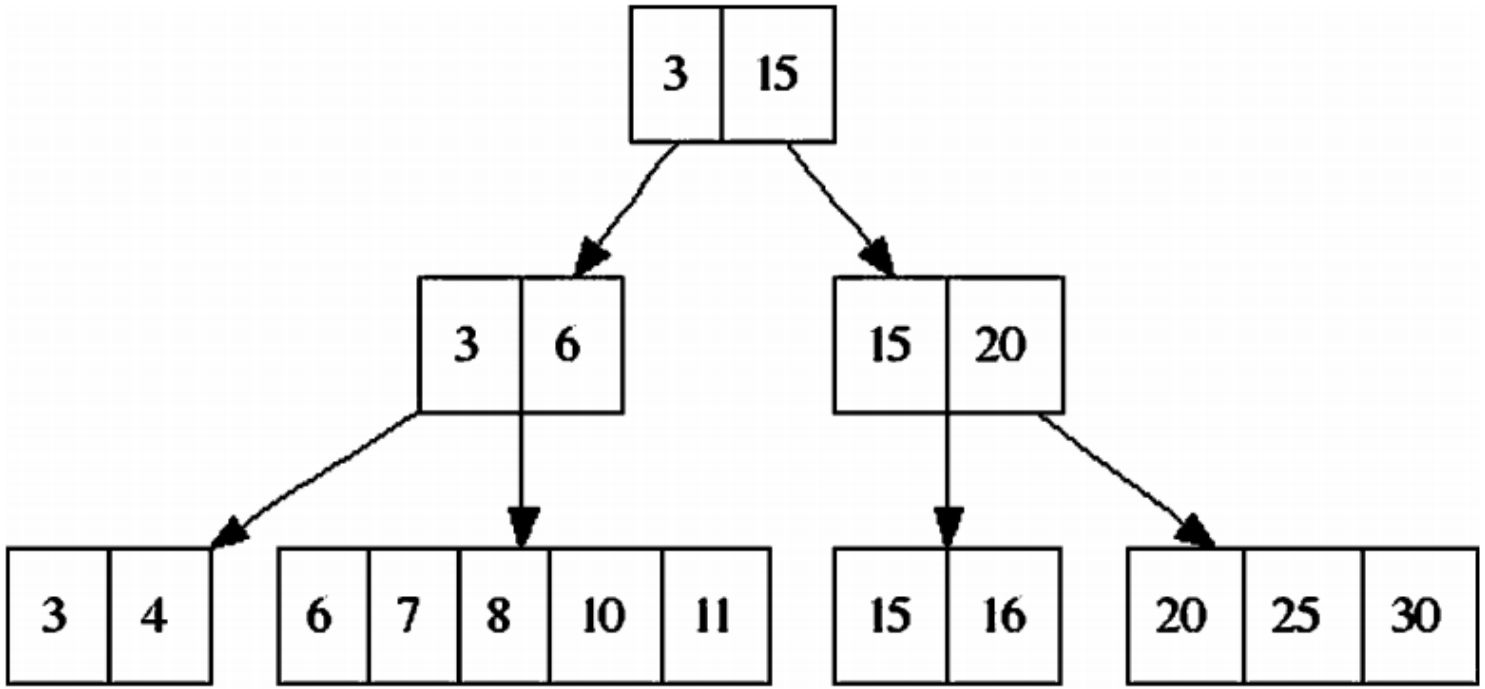
Insert 9 (with Proactive Split)



Remove-key

- Proactive merge is used during the remove-key operations

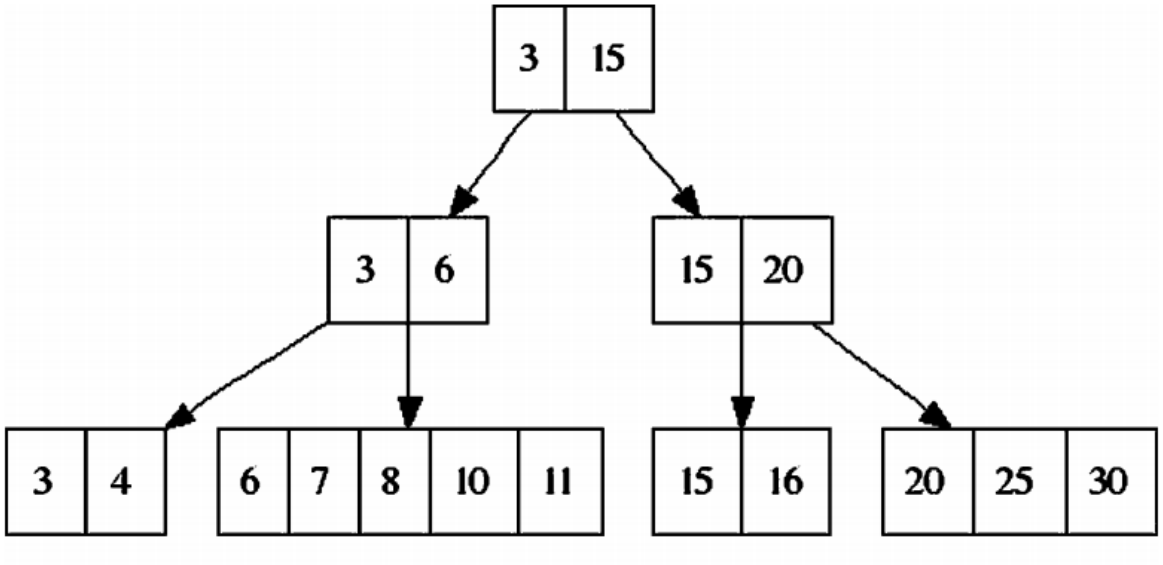
Remove 10 fom the tree

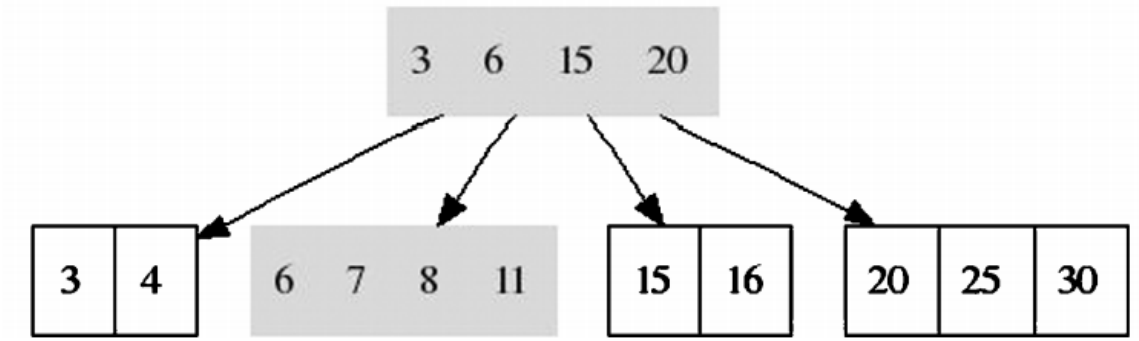


Remove-key

- Proactive merge is used during the remove-key operations

- Grey nodes are shadowed





Cloning a File System

- Create another instance of the file system preserving its structure

- Let Tp be a B tree and Tq is the clone of Tp
  - **Space Efficiency:** Tp and Tq should share the common pages as much as possible
  - **Speed:** Constructing Tq from Tp should take minimum time
  - **Number of clones:** It should be possible to clone Tp as much times as needed
  - **Clones as First-class Citizens:** It should be possible to clone Tq

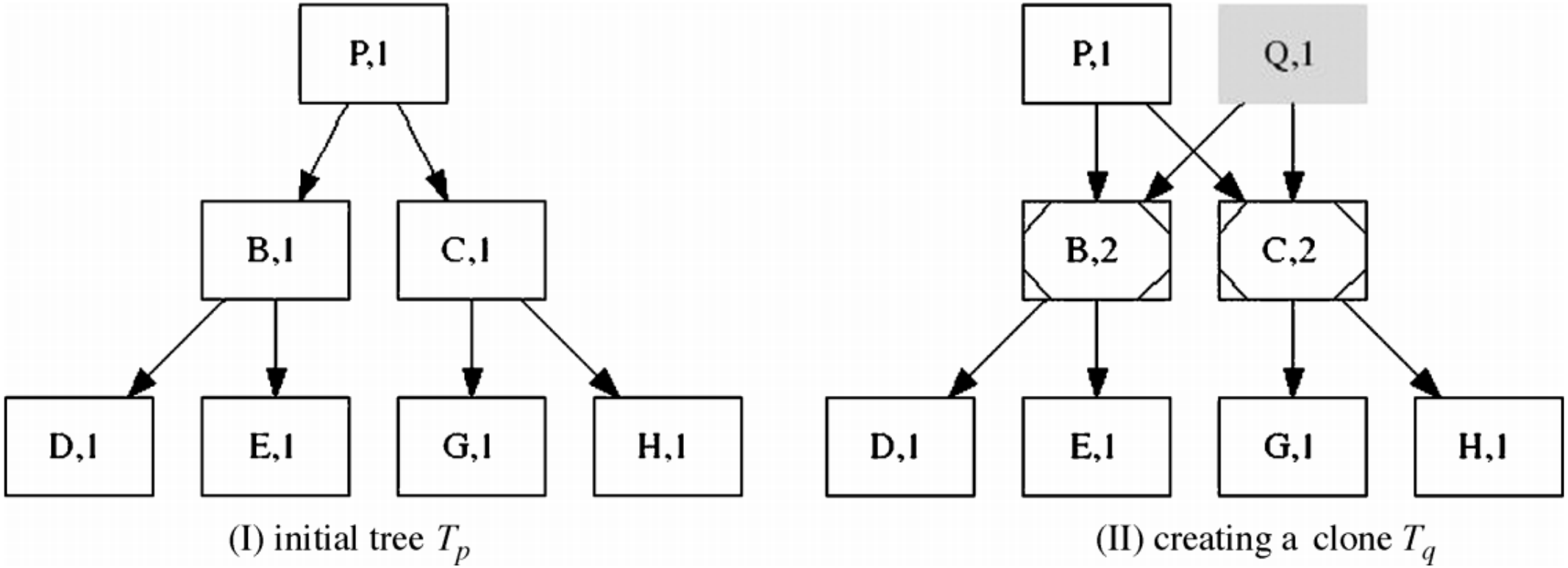
Cloning using Shadowing

- Use a free space map that maintains a reference count for each block
  - Records how many times a page is pointed to
  - Zero reference count -> the block is free

- Instead of copying a tree, the ref-counts of all its nodes are incremented by one
  - Indicates that the nodes belong to two trees instead of one; the nodes are shared pages

- **Lazy updates of reference counts –** Instead of updating the reference counts for all the nodes in the tree, just update the immediate children's; rest follow

Cloning using Shadowing

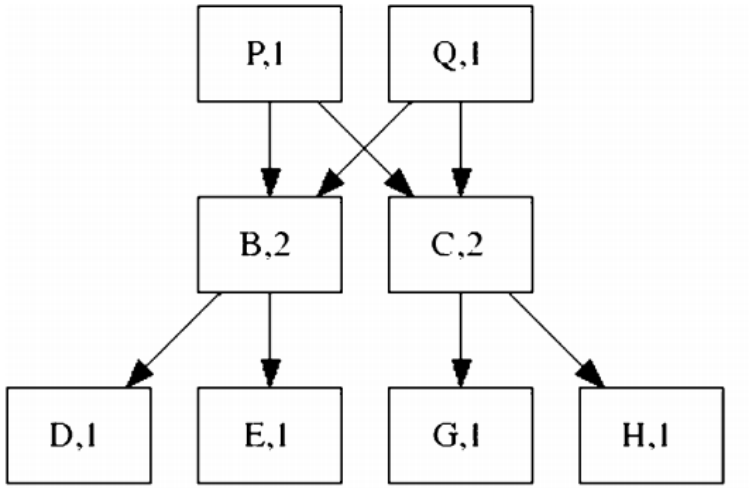


Insert-key and Remove-key on a Cloned Tree

- Before modifying a page, it is "marked-dirty"
  - The runtime system knows that the page is about to be modified
  - Gives it a chance to shadow the page if necessary

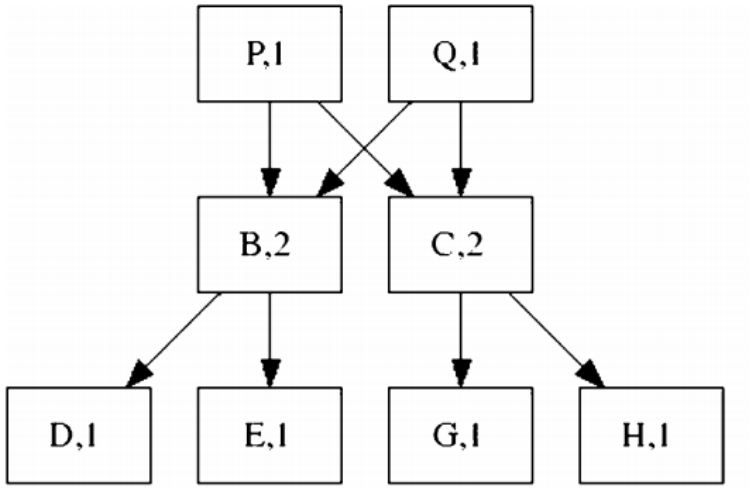
- When marking dirty a clean page N
  - If the reference count is 1, nothing special is needed; same as the tree without cloning
  - If the reference count is more than 1, and page N is relocated from address L1 to address L2
    - The reference count of L1 is decremented
    - The reference count of L2 is made 1
    - The reference count of L's children is incremented by 1

Insert-key and Remove-key on a Cloned Tree

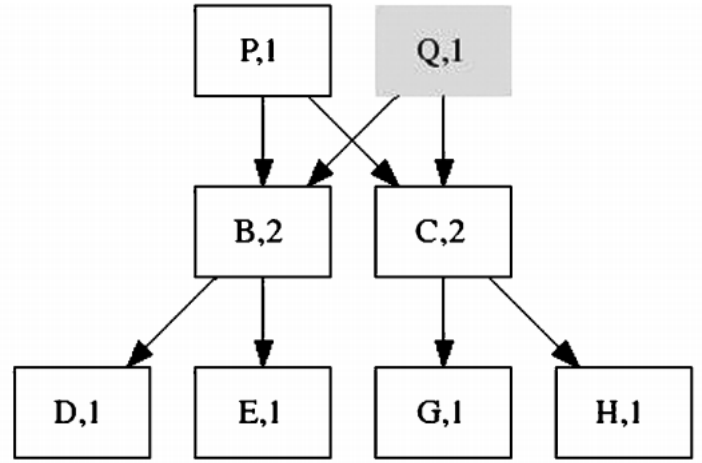


(I) initial trees T_p and t_q

Insert-key and Remove-key on a Cloned Tree

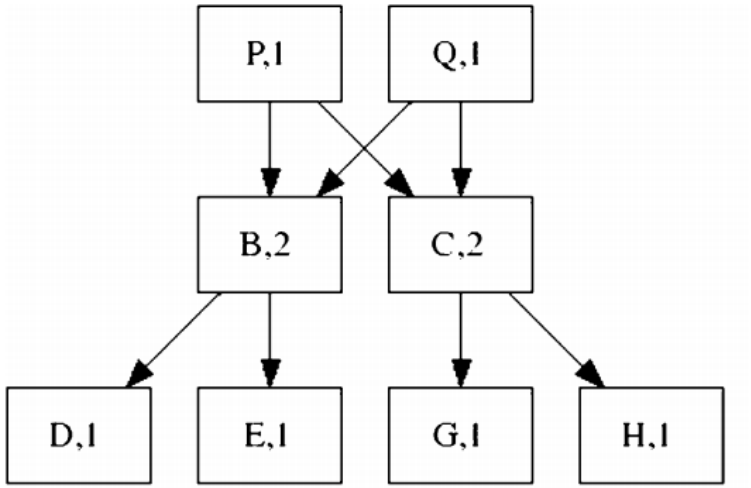


(I) initial trees T_p and t_q

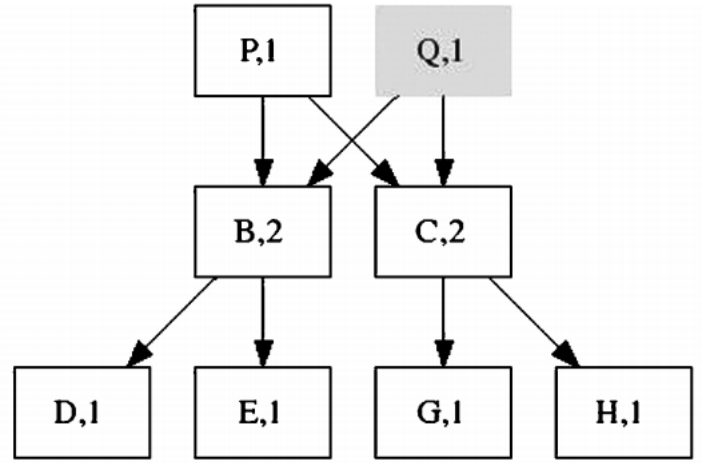


(II) shadow Q

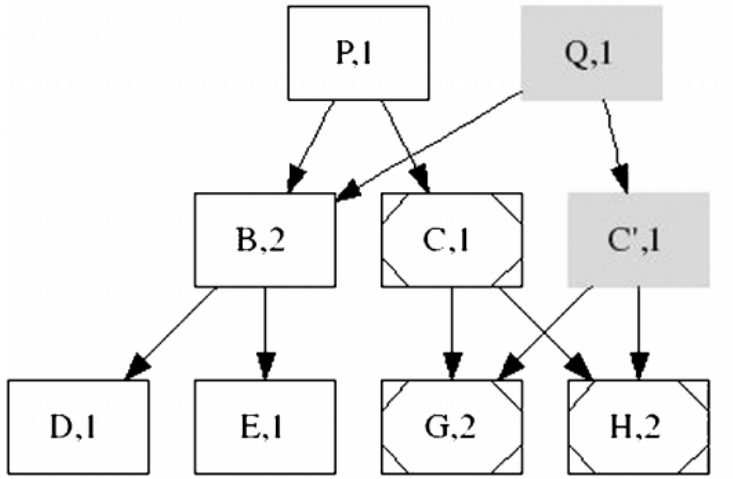
Insert-key and Remove-key on a Cloned Tree



(I) initial trees T_p and t_q

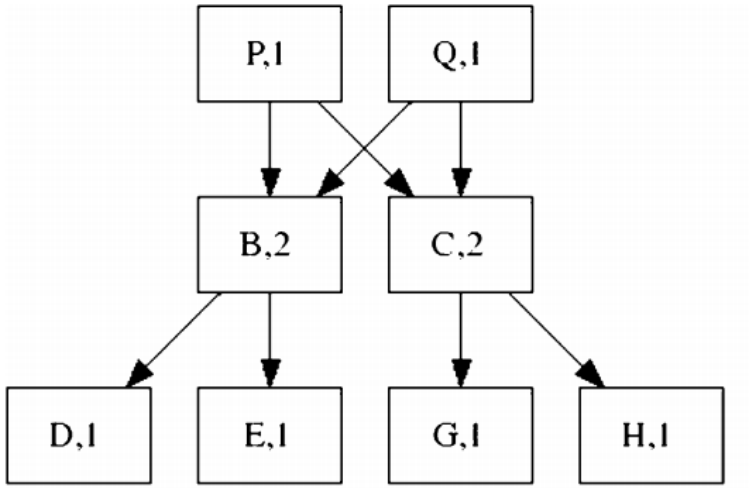


(II) shadow Q

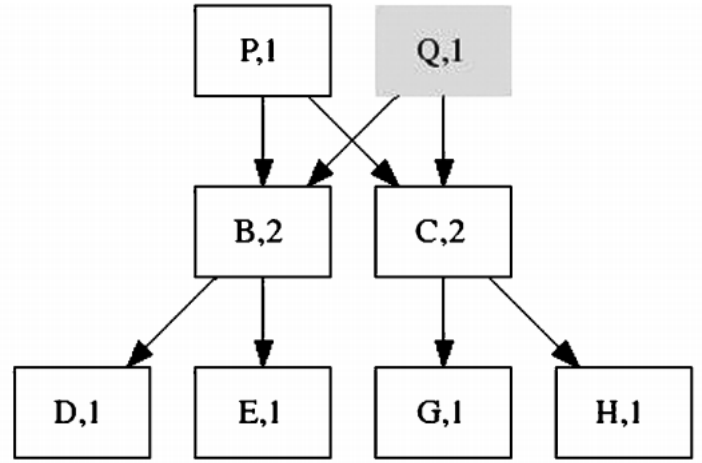


(III) shadow C

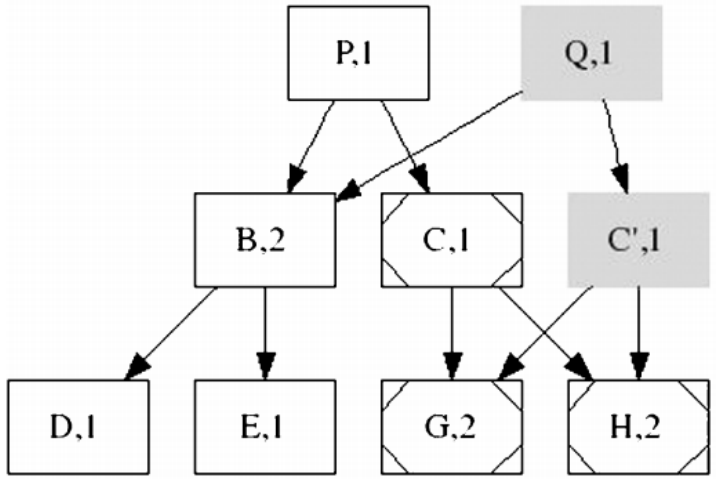
Insert-key and Remove-key on a Cloned Tree



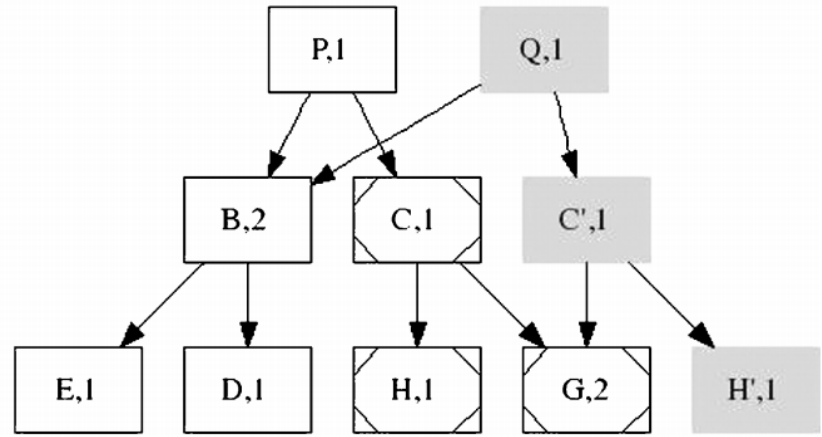
(I) initial trees T_p and t_q



(II) shadow Q



(III) shadow C



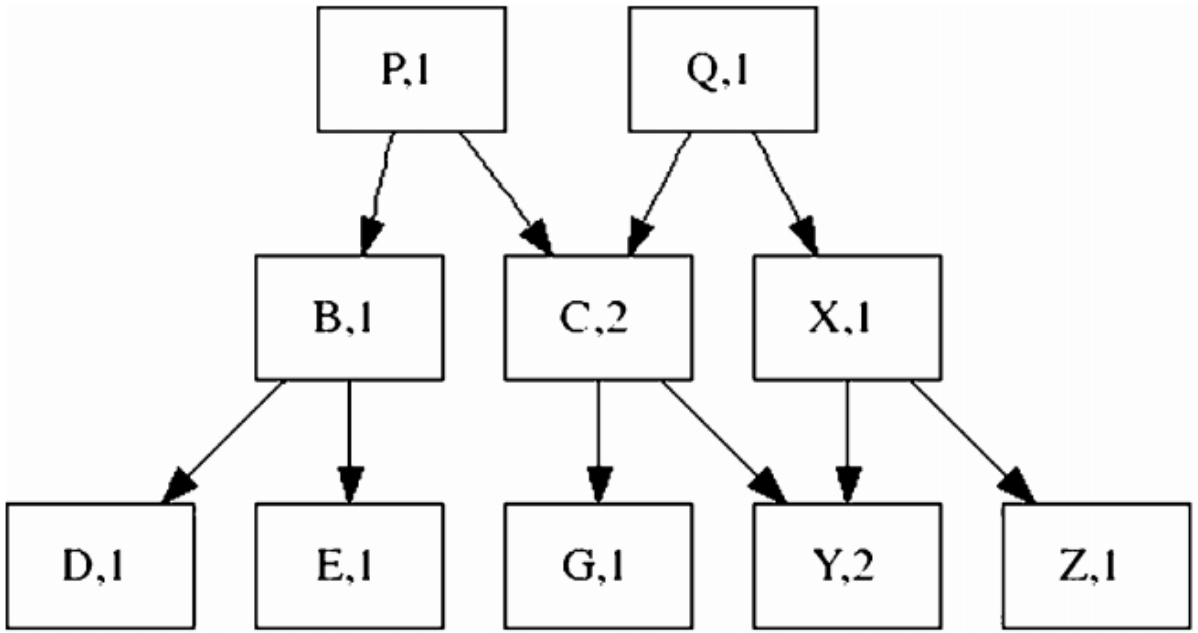
(IV) shadow H

Delete a Cloned Tree

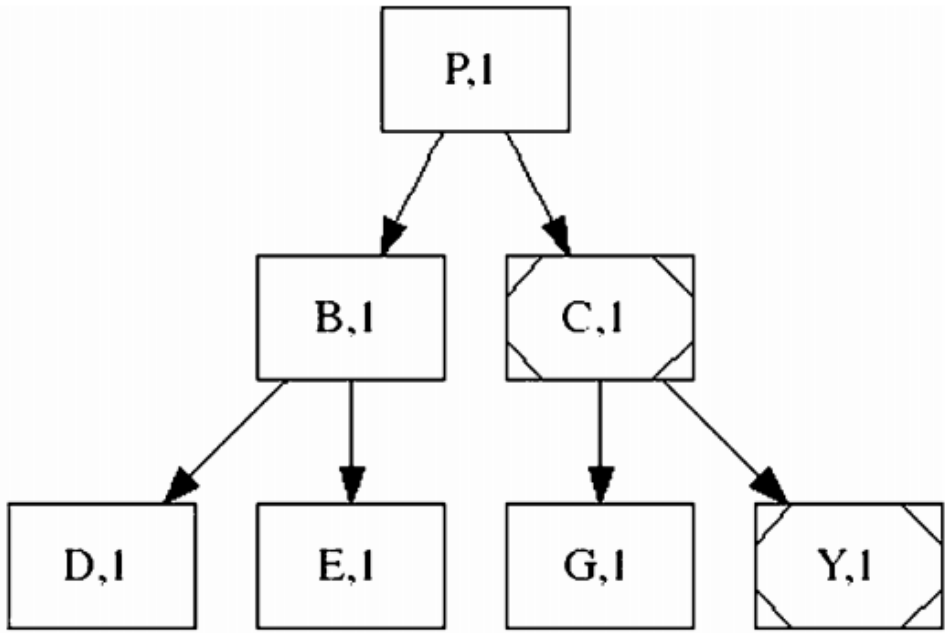
- All nodes are deallocated based on a post-order traversal of the tree
  - Reference counts need to be updated

- Tree Tp is being deleted; during the downward part of the post-order traversal, node N is reached
  - If the ref-count of N > 1, decrement the ref-count and stop downward traversal
  - If the ref-count of N = 1, then it belongs only to Tp. Continue the downward traversal, and on the way back, deallocate N

Delete a Cloned Tree

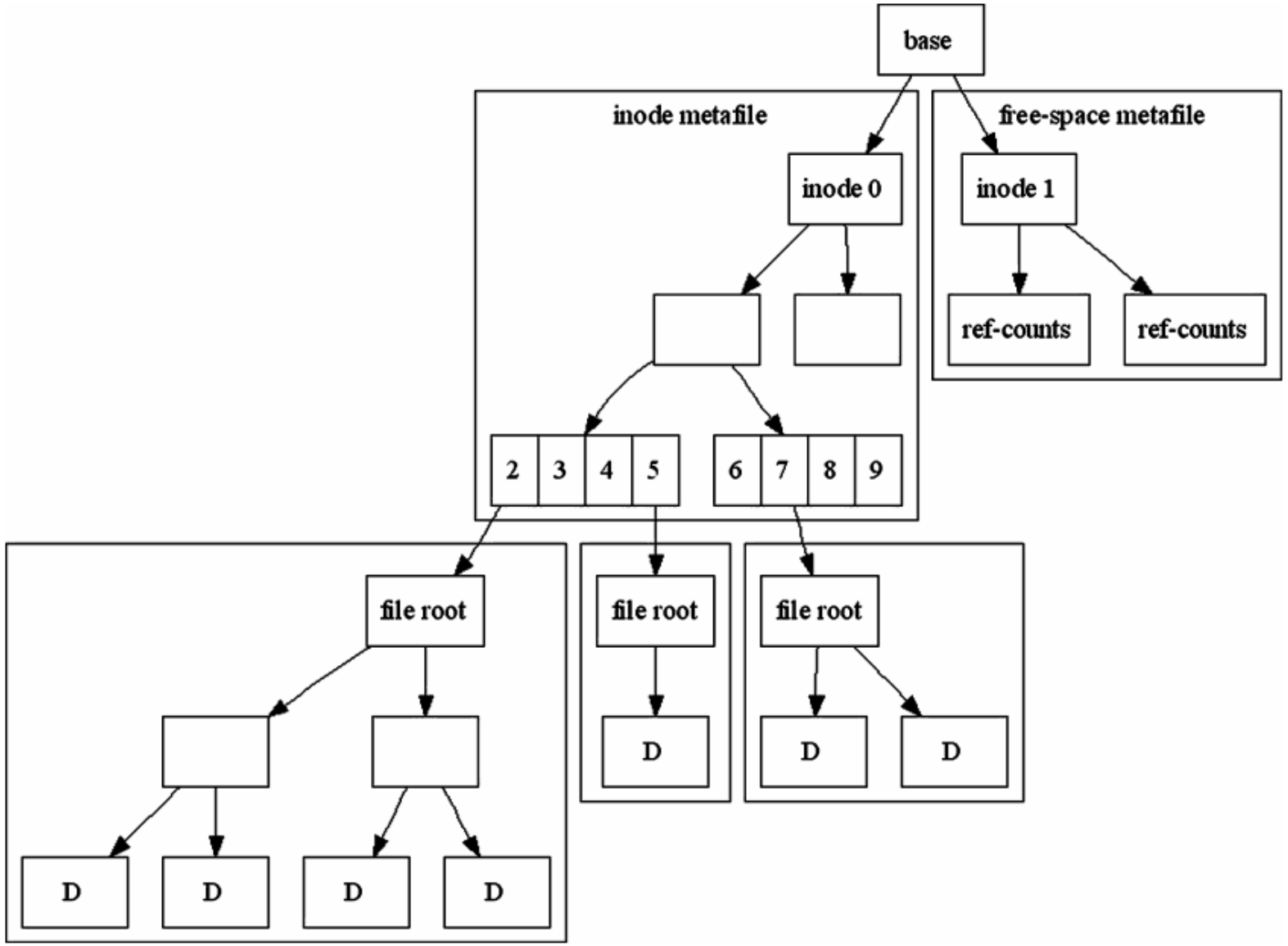


(I) initial tree T_p and T_q

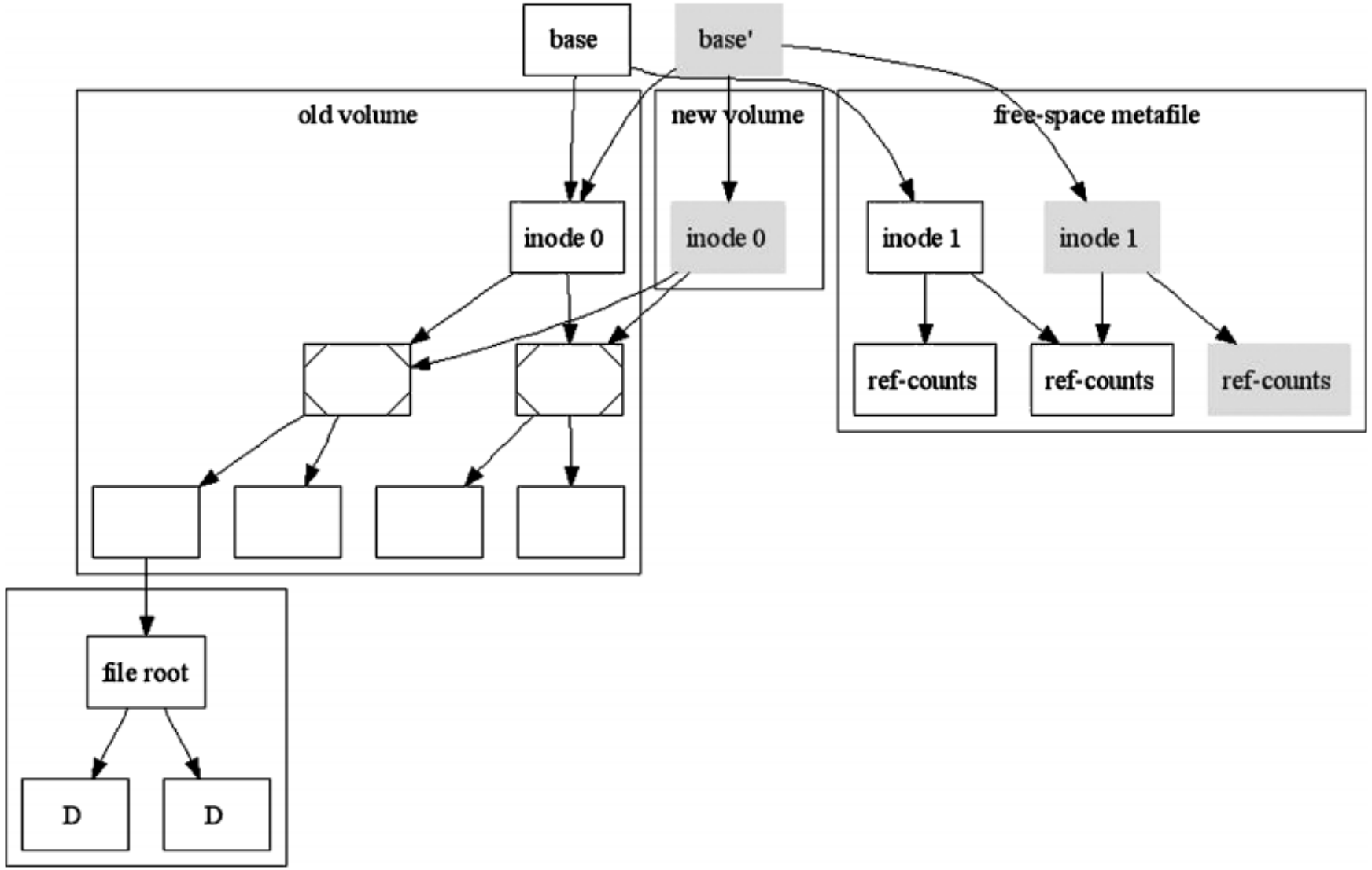


(II) After deleting T_q

A Basic B Tree based File System

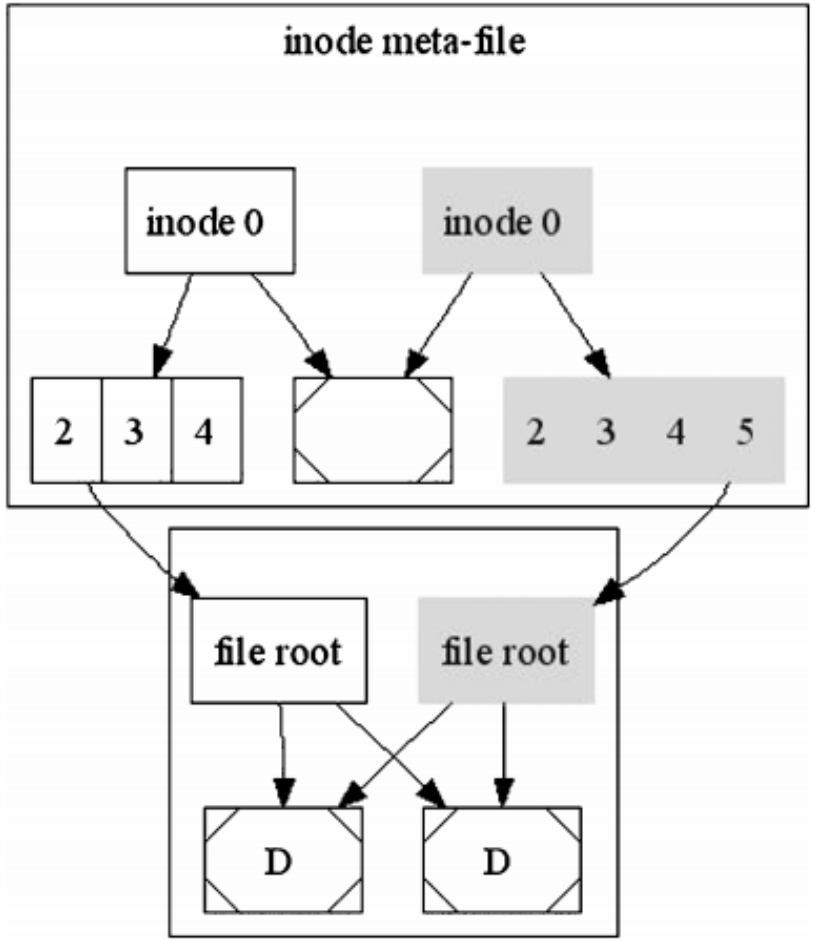


Cloning a Volume



Cloning a File

- Create a new inode with a new file name, clone the B tree holding the file data
  - Inode 2 points the original file
  - Create a new inode (inode 5)
    - Note that the root (inode 0) needs to be shadowed as well
  - Create a new file root; inode 5 points to the new file root



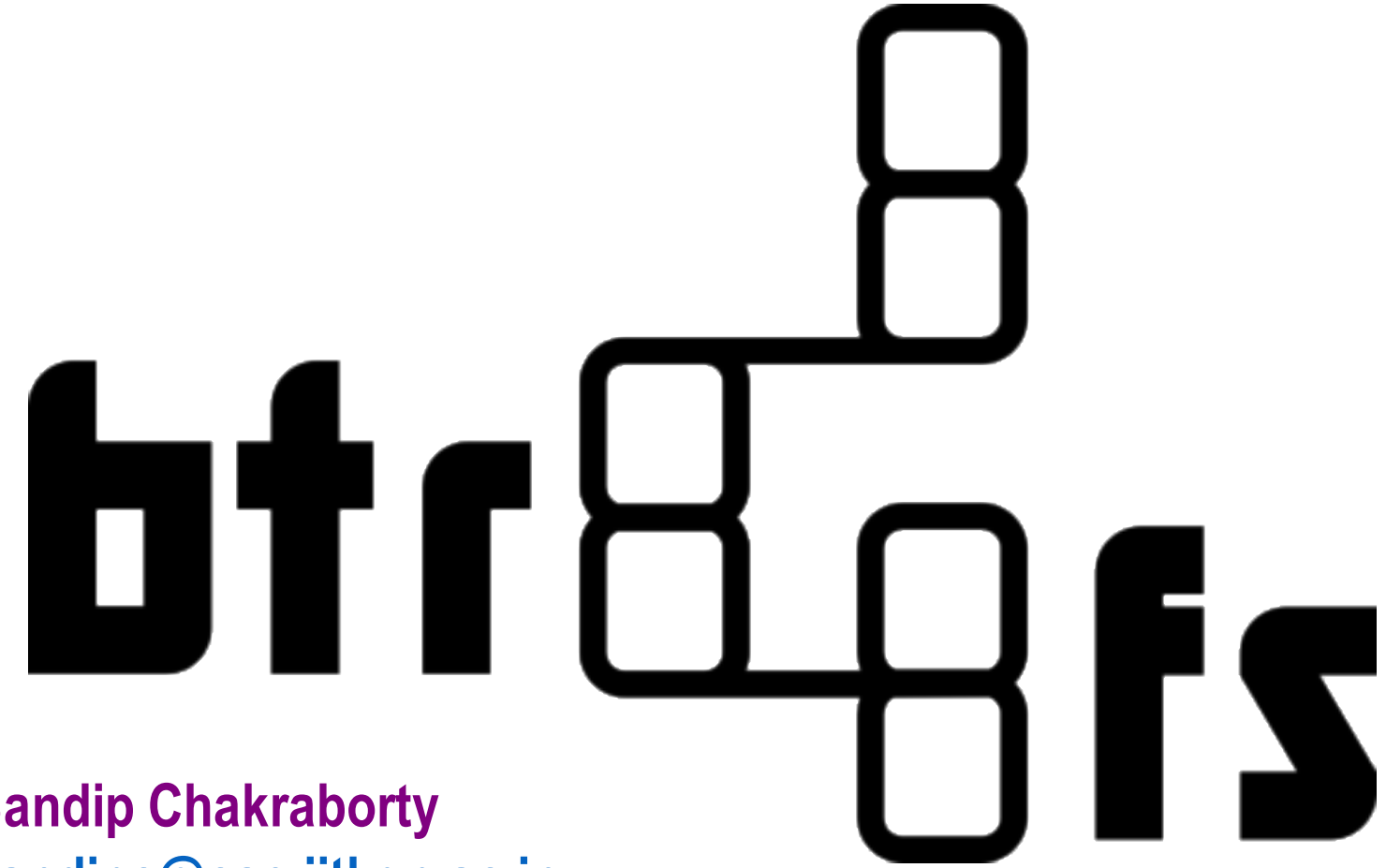
Reference

Rodeh, Ohad. "B-trees, shadowing, and clones." *ACM Transactions on Storage (TOS)* 3.4 (2008): 1-27.

BTRFS: A B-Tree Based File System

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR



Sandip Chakraborty

sandipc@cse.iitkgp.ac.in

Some History

- A file system based on COW principle -- initially designed at Oracle Corporation for use in Linux

- The development began in 2007, since November 2013 it has been marked as stable

- Principal Btrfs author: Chris Mason

"to let Linux scale for the storage that will be available. Scaling is not just about addressing the storage but also means being able to administer and to manage it with a clean interface."

Fundamentals

- **Page, block:** A 4KB contiguous region on disk and in memory. This is the standard Linux page size.

- **Extent:** A contiguous on-disk area. It is page aligned, and its length is a multiple of pages.

- **Copy-on-write (COW):** Creating a new version of an extent or a page at a different location
  - The data is loaded from disk to memory, modified, and then written elsewhere
  - Do not update the original location in place, risking a power failure and partial update

BTRFS B-tree

- A generic structure with three types of data structures: *keys, items,* and *block headers*

- **Block header:** A fixed size data structure, holds fields like checksums, flags, filesystem ids, generation number, etc.

- **Key:** describes an object address,

```
struct key {
    u64: objectid;  u8: type;  u64 offset;
}
```

BTRFS B-Tree

- **Item:** is a key with additional offset and size fields.

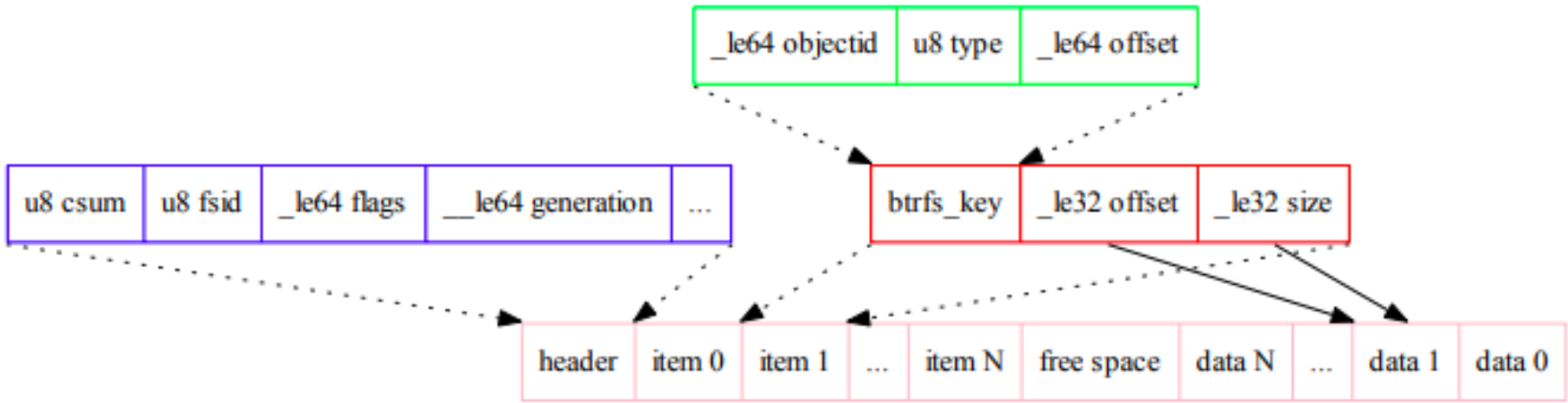
```
struct item {
    struct key key; u32 offset; u32 size;
}
```

- Internal tree nodes hold only `[key, block-pointer]` pairs
- Leaf nodes hold arrays of `[item, data]` pairs.
- The offset field describes data held in an extent.

BTRFS B-Tree

- A leaf stores
  - an array of items in the beginning
  - a reverse sorted data array at the ends
  - These arrays grow towards each other.

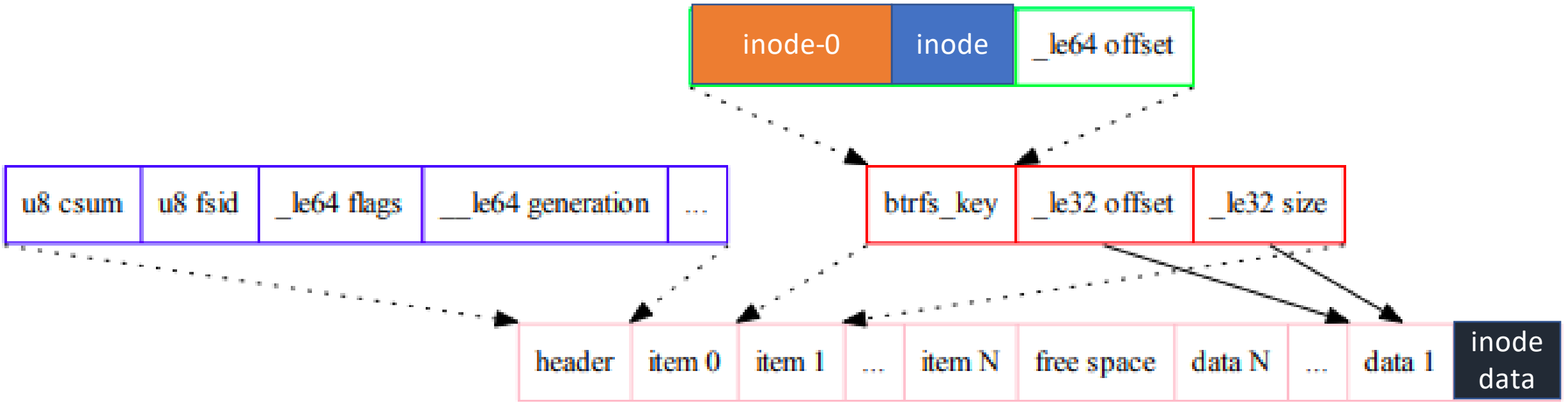
block header	I_0	I_1	I_2	free space	D_2	D_1	D_0



inode

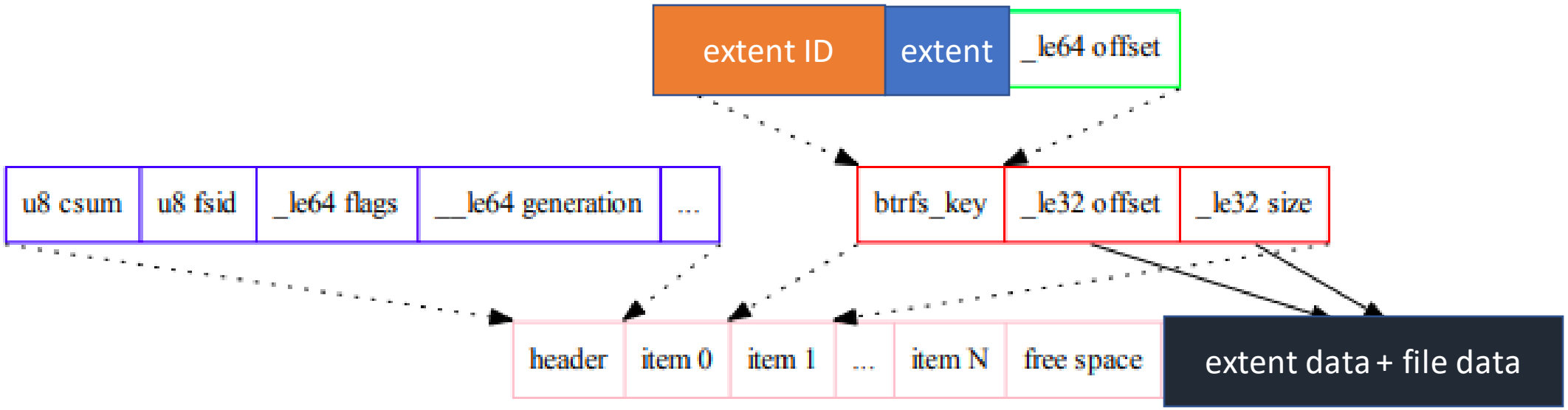
- Inodes are storaed in a inode item at offset zero in the key
  - Have a type value of one
- The lowest valued key for a given object
  - Store the traditional stat data for files and directories
- The inode structure is relatively small
  - Does not contain embedded file data or extended attribute data

inode


inode-0
inode
_le64 offset
u8 csum
u8 fsid
_le64 flags
__le64 generation
...
btrfs_key
_le32 offset
_le32 size
header
item 0
item 1
...
item N
free space
data N
...
data 1
inode data

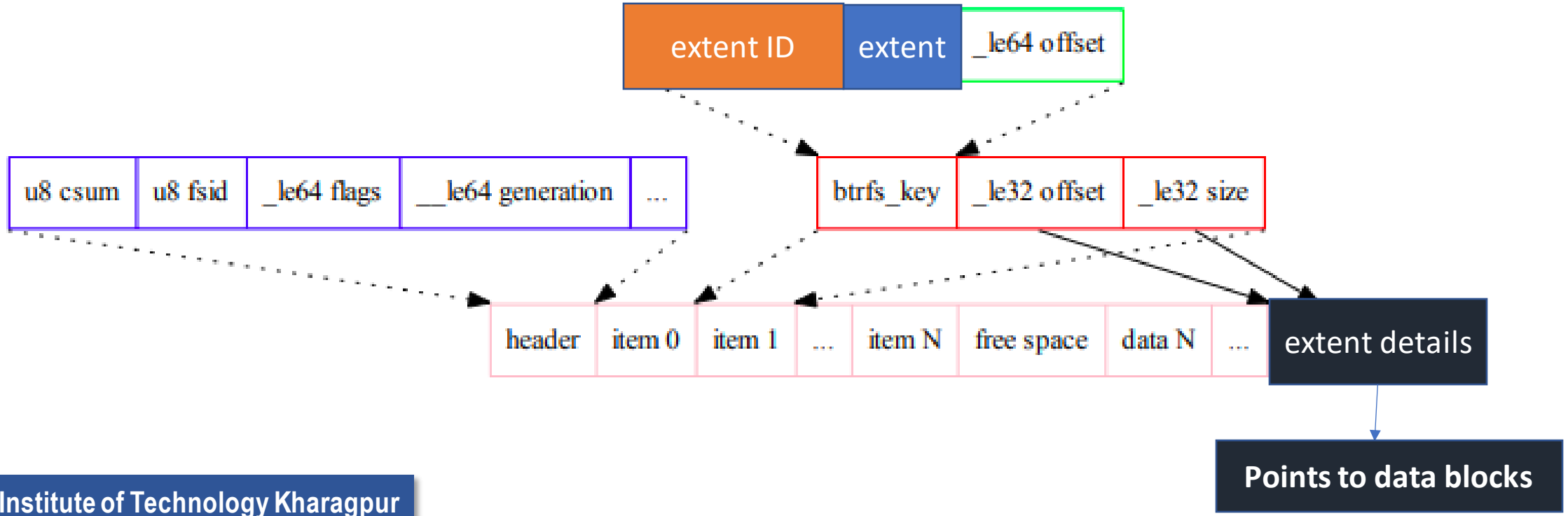
Small Files in the Leaf

- Small files that occupy less than one leaf block are packed into the b-tree inside the extent item
  - Key offset if the byte offset of the data in the file
  - The size field indicates how much data is stored



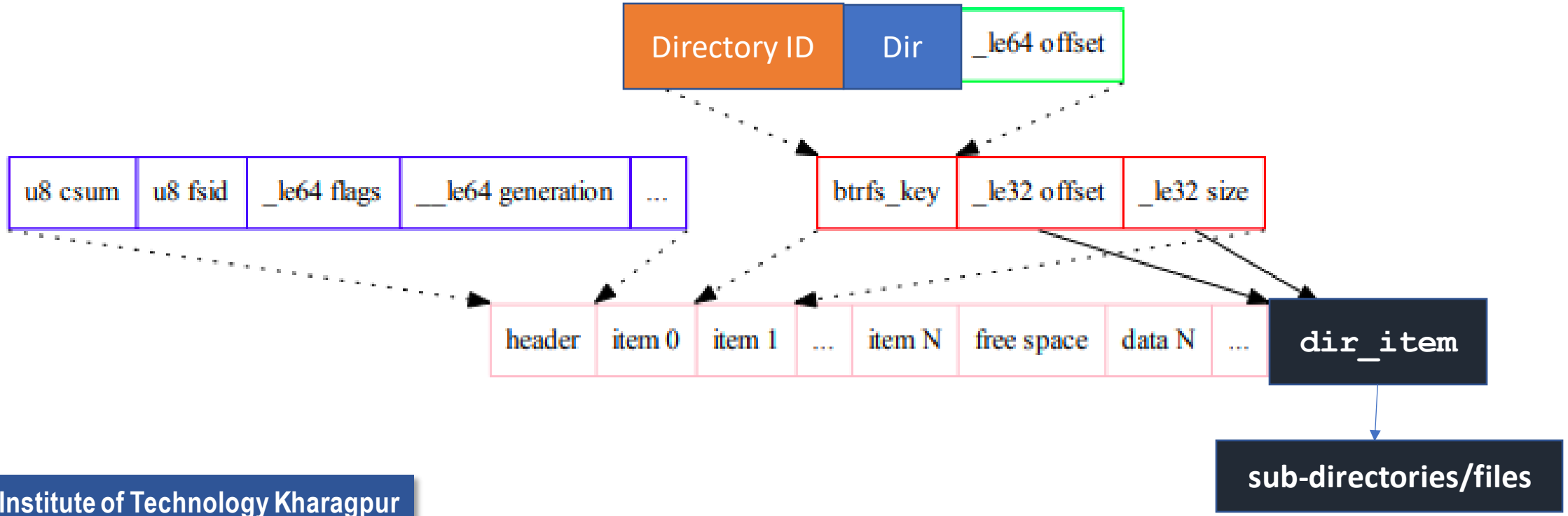
Large Files

- Large files are stored in extents -- contiguous on-disk areas that hold user-data without additional headers or formatting
- Extent maintains a `[disk block, disk num blocks]` pair to record the area of disk corresponding to the file.

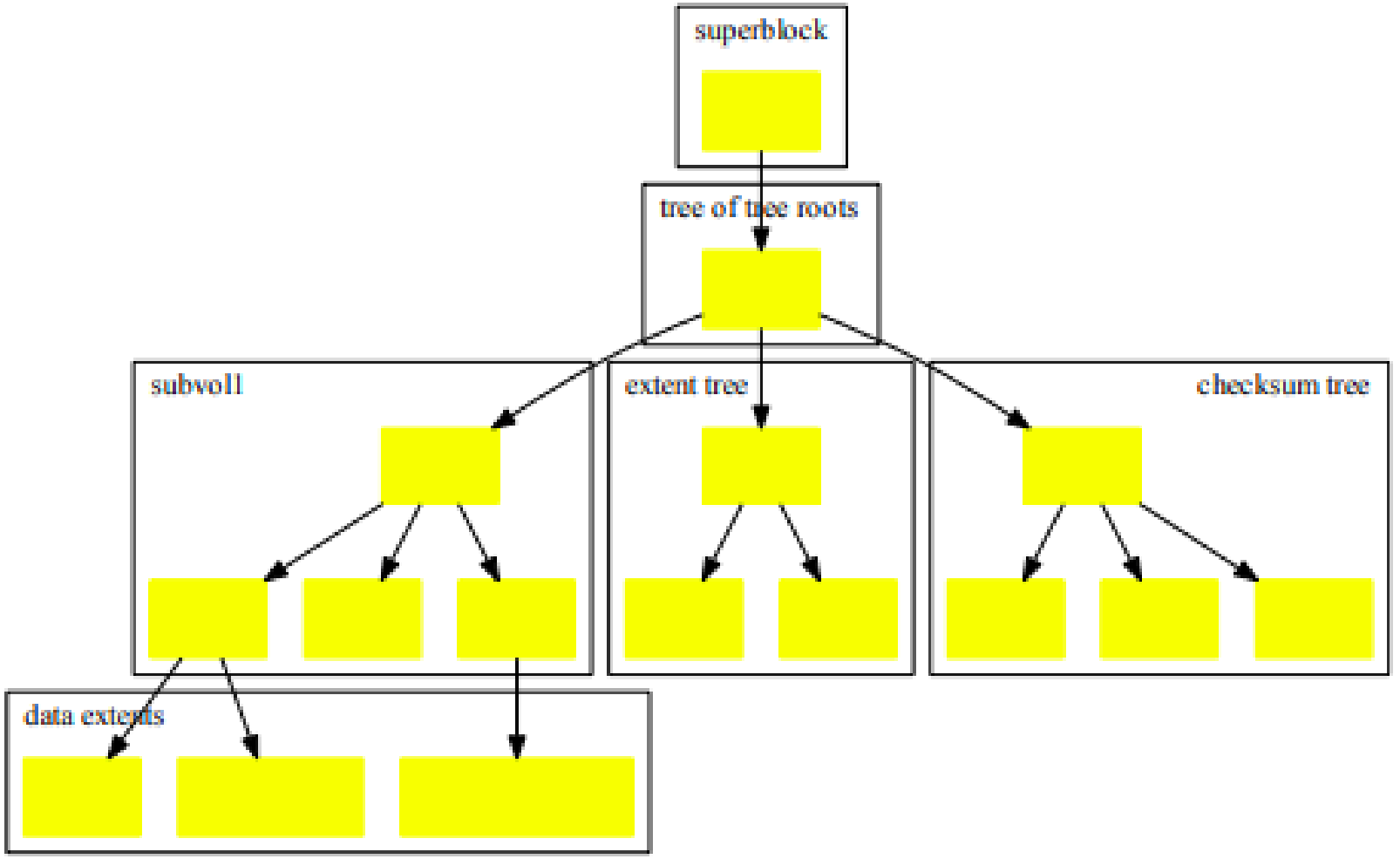


Directory

- A directory holds an array of `dir_item` elements
  - Maps a file name (string) to a 64bit `object_id`
- Directory lookup index - `[dir_item_key, filename 64bit hash]`



Filesystem Forest



Filesystem Forest - Sub-volumes

- Store user visible files and directories
- Each sub-volume is implemented by a separate tree
- Sub-volumes can be snapshotted and cloned, creating additional b-trees
- The roots of all sub-volumes are indexed by the tree of tree roots

Extent Allocation Tree

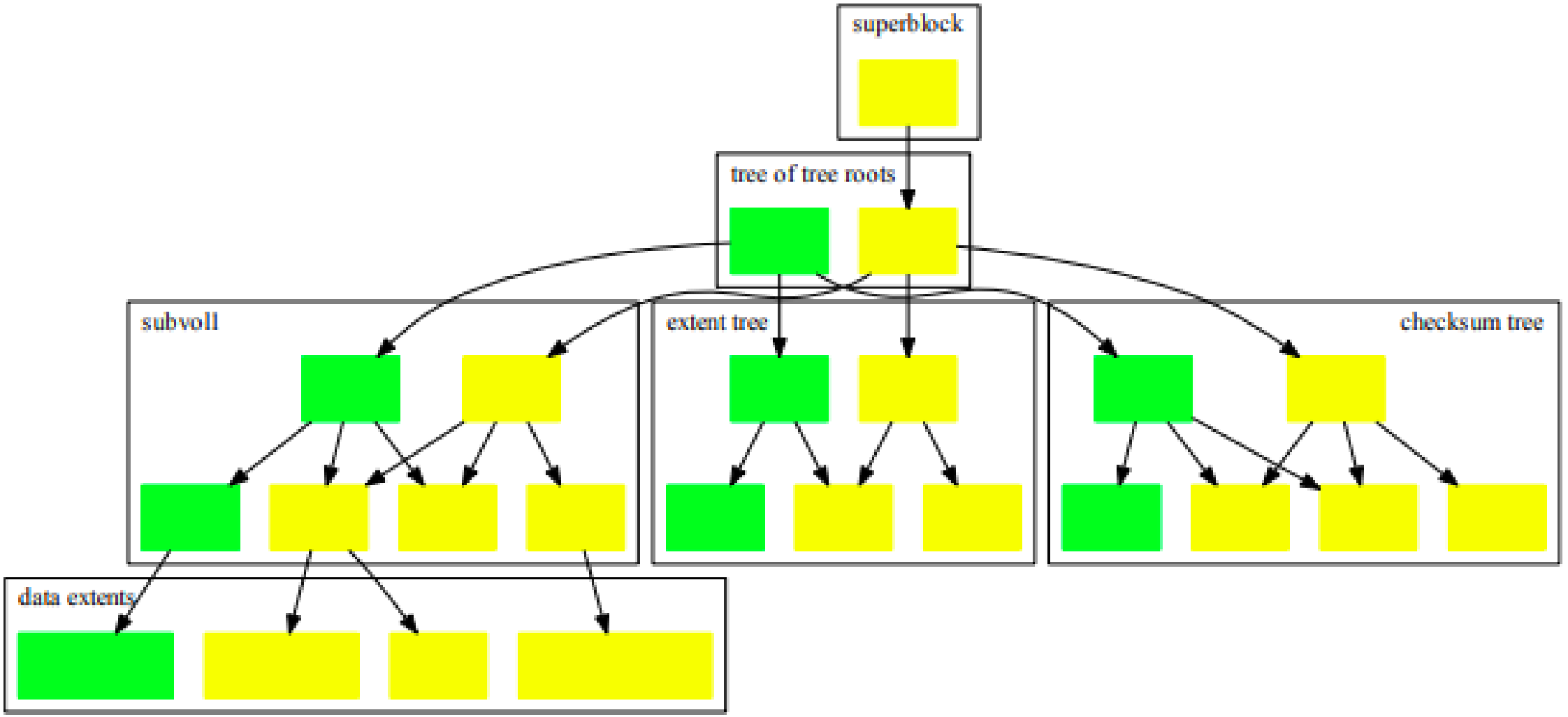
- Tracks allocated extents in **extent items**
  - Serves as an on-disk free-space map

- All *back-references* to an extent are recorded in the extent item -- allows moving an extent if needed, or recovering from a damaged disk block

Other Trees

- **Checksum tree**: holds a checksum item per allocated extent
  - The item contains a list of checksums per page in the extent
- **Chunk and device trees:** indirection layer for handling physical devices
  - Allows mirroring/stripping and RAID -- implements multiple device support
- **Reloc tree:** for special operations involving moving extents
  - Used for defragmentation

COW over the Filesystem Forest



COW Semantics

- All the tree modifications are captured at the top most level as a new root (shadow copy) in the tree of tree roots.
- Modifications are accumulated in memory, are written in batch (forms a checkpoint) to a new disk location after
  - a timeout
  - enough pages have changed
- Default timeout is 30 seconds
- Once the checkpoint has been written, the superblock is modified to point the new checkpoint

COW Semantics - Recovery after a Crash

- The filesystem recovers by reading the superblock, and following the pointers to the last valid on-disk checkpoint.
  - **This is the reason that the superblock is pointed to the new checkpoint after the modifications are written on the disk**
  - **If the system crashes before the superblock is updated, the recovery points to the old checkpoint, new updates are lost but the file system remains consistent**

- When a checkpoint is initiated, all dirty memory pages that are part of it are marked immutable - **why?**

COW Semantics - Recovery after a Crash

- The filesystem recovers by reading the superblock, and following the pointers to the last valid on-disk checkpoint.
  - **This is the reason that the superblock is pointed to the new checkpoint after the modifications are written on the disk**
  - **If the system crashes before the superblock is updated, the recovery points to the old checkpoint, new updates are lost but the file system remains consistent**

- When a checkpoint is initiated, all dirty memory pages that are part of it are marked immutable
  - Immutable pages need to be re-COWed for user updates
  - Allows user visible file system operations to proceed without damaging the checkpoint integrity, while the checkpoint is in flight

Efficiency

- A filesystem update affects many on-disk structures
  - A 4KB write into a file changes the file i-node, the file extents, checksums, and back-references
- Each of these changes causes an entire path to change in its respective tree
- This is very expensive if user performs entirely random updates
- Btrfs assumes the locality of user access; however, worst cases are taken care of through COW friendly B-trees.

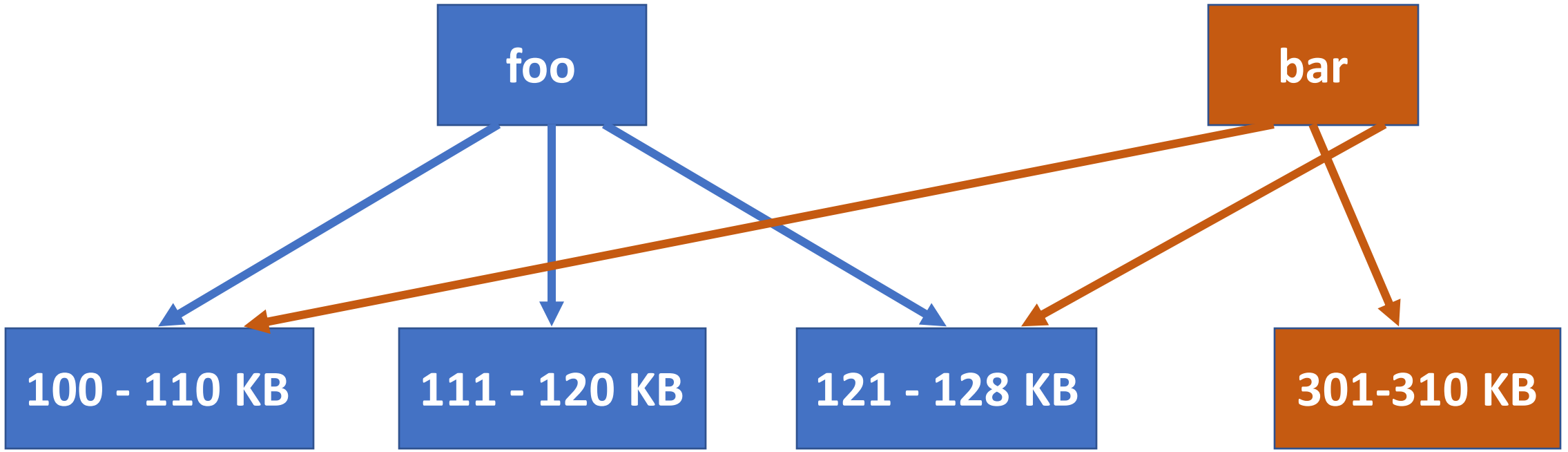
Extent Allocation Tree and Back References

- Consider the file `foo` that has an on-disk extent 100KB - 128KB
- File `foo` is cloned creating file `bar`
- Later on, a range of 10KB is overwritten in `bar`

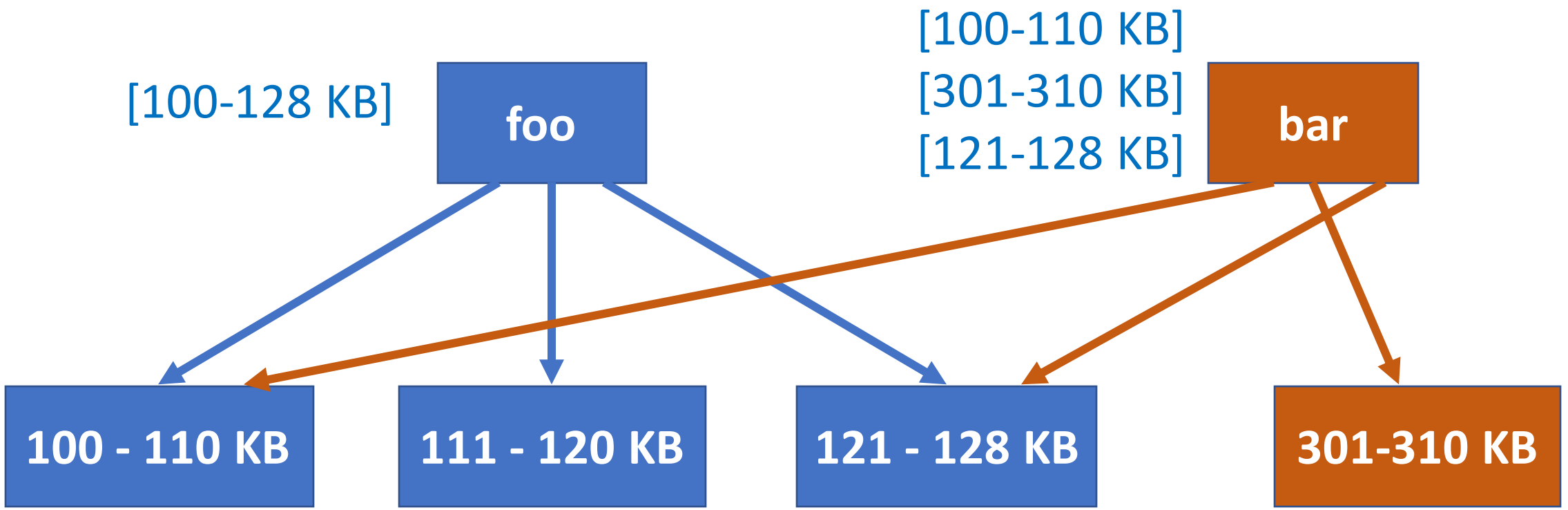
File	On disk extents
foo	**100-128KB**
bar	**100-110KB, 301-310KB, 121-128KB**

This is located much further in the disk

Extent Allocation Tree and Back References

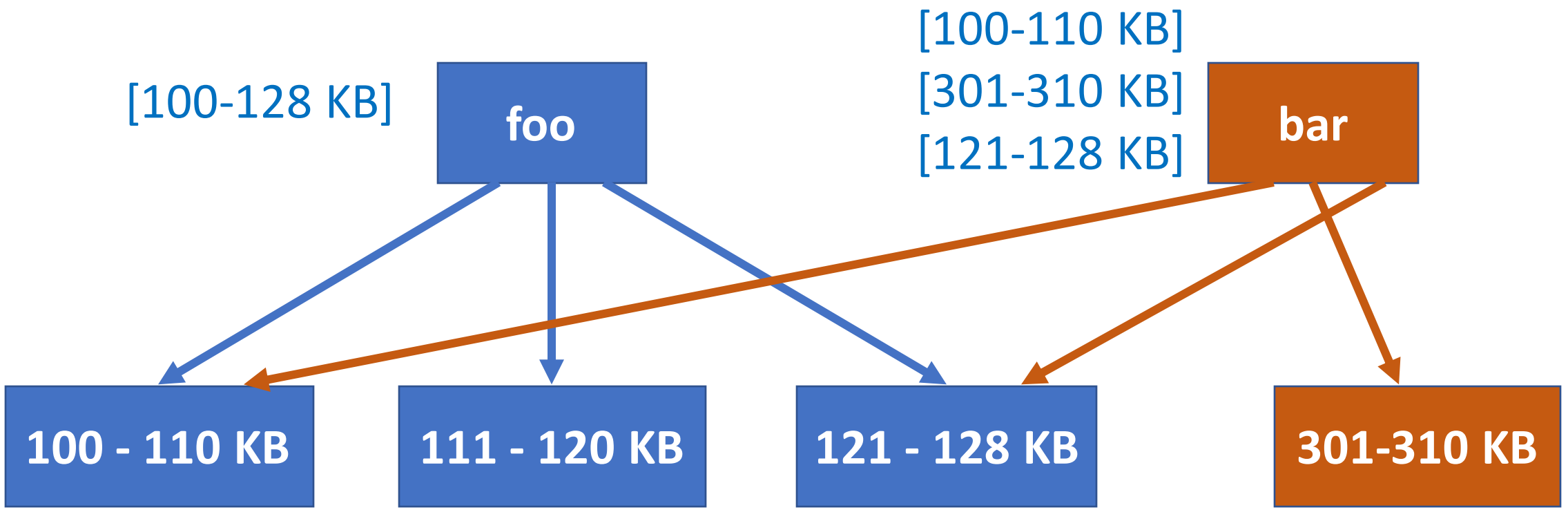


Extent Allocation Tree and Back References



- The extent-item keeps track of all the references, to allow moving the entire extent at a later time

Extent Allocation Tree and Back References



- The extent-item keeps track of all the references, to allow moving the entire extent at a later time

[100-128 KB] ← [301-310 KB]

Extent Allocation Tree and Back References

- An extent could potentially have a large number of back references -- extent-time may not fit in a single b-tree leaf node
  - Item spills and takes up more than one leaf

- A back reference is logical, not physical -- this is computed from the extent allocation tree from the parameters root_object_id, generation_id, tree level and lowest object-id in the pointing block.

Multiple Device Support

- Linux has *device-mapper* (DMs) subsystems that manage storage device
  - `LVM, mdadm`
- DMs are software modules
  - Take raw disks
  - Merge them into a logically virtually contiguous block-address space
  - Export that abstract to higher level kernel layers
- Supports mirroring, striping, and RAID5/6
- Checksums are not supported - problem for BTRFS

Problem with Checksum and DMs

- Consider the following case - data is stored in RAID-1 form on disk, each 4KB block has an additional copy
- The filesystem detects a checksum error on one copy - it needs to recover from the other copy
- DMs hide that information behind the virtual address space abstraction, and return one of the copies

Multiple Device Support in BTRFS

- BTRFS does its own device management
  - Calculates checksums
  - Stores them in a separate tree
  - Recover data by matching the checksums

- BTRFS splits each device into large *chunks*
  - Chunk should be about 1% of the device size.

- **Chunk Tree**: Maintains a mapping from logical chunks to physical chunks
- **Device Tree**: Maintains the reverse mapping

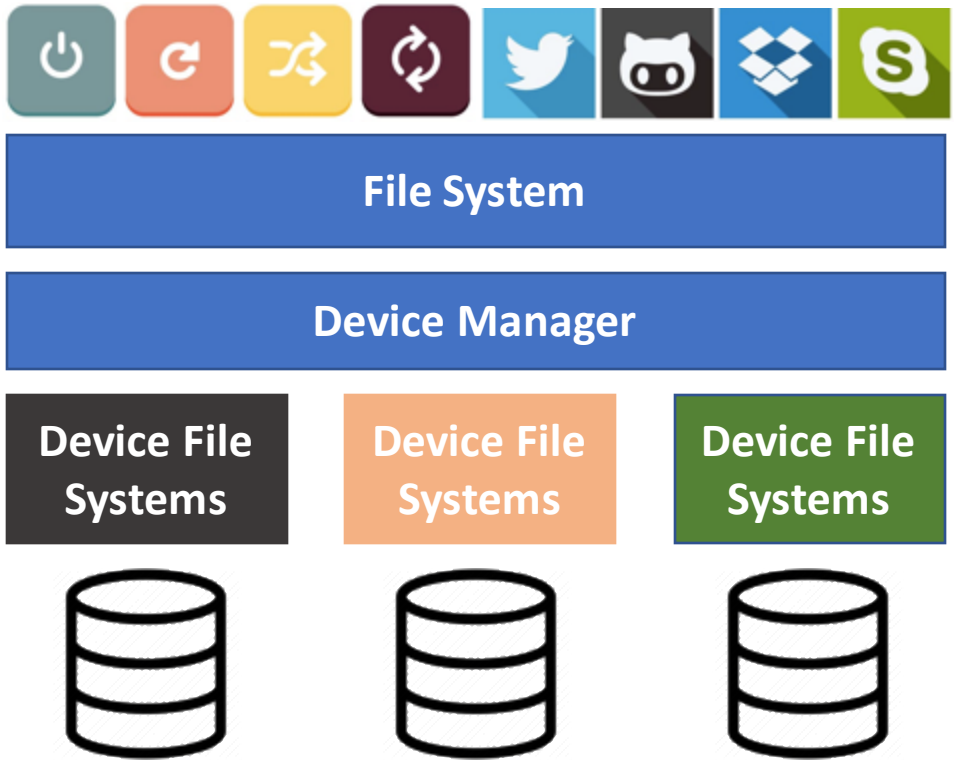
Multiple Device Support in BTRFS

- Rest of the filesystem sees logical chunks -- all extent references address logical chunks
  - Allows moving physical chunks under the covers without the need to backtrace and fix references

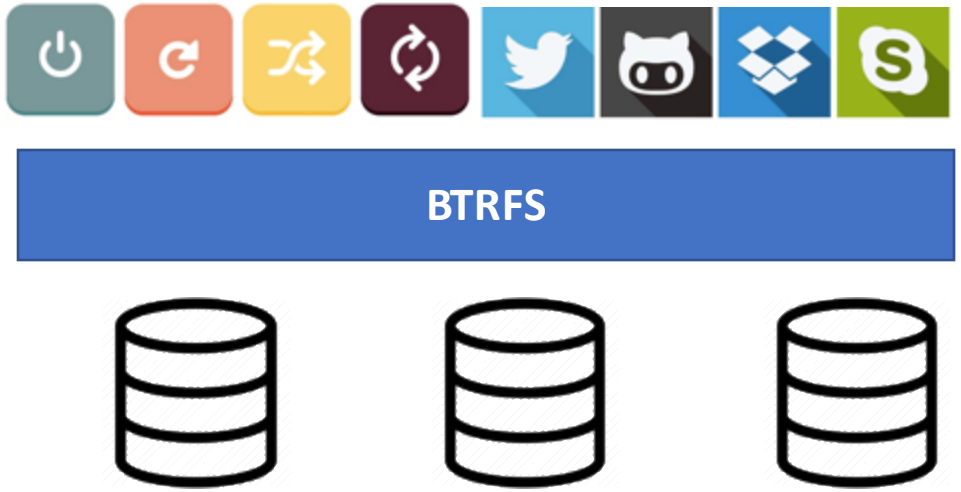
- The chunk and device trees are small - can typically be cached in memory
  - Reduces the performance cost of an added indirection layer

DMs versus BTRFS

Traditional File Systems with DMs



BTRFS



Physical Chunk Management

- Physical chunks are divided into groups according to the required RAID level of the logical chunk

- For Mirroring, chunks are divided into pairs

logical chunks	disk 1	disk 2	disk 3
L_1	C_{11}	C_{21}	
L_2		C_{22}	C_{31}
L_3	C_{12}		C_{32}

RAID-1 Logical Chunks

L: Logical Chunks

C: Physical Chunks

Physical Chunk Management

- Physical chunks are divided into groups according to the required RAID level of the logical chunk

- For Mirroring, chunks are divided into pairs

logical chunks	disk 1	disk 2	disk 3
L_1	C_{11}	C_{21}	
L_2		C_{22}	C_{31}
L_3	C_{12}	C_{23}	
L_4		C_{24}	C_{32}

RAID-1 Logical Chunks - One Large disk and two small disks

L: Logical Chunks

C: Physical Chunks

Physical Chunk Management

- For **stripping**, groups on *n* chunks are used, where each physical chunk is on a different disk

logical chunks	disk 1	disk 2	disk 3	disk 4
L_1	C_{11}	C_{21}	C_{31}	C_{41}
L_2	C_{12}	C_{22}	C_{32}	C_{42}
L_3	C_{13}	C_{23}	C_{33}	C_{43}

Stripping with strip width = 4

L: Logical Chunks

C: Physical Chunks

Complex RAID Levels

- RAID 6 configurations - Logical chunks are constructed from doubly protected physical chunks

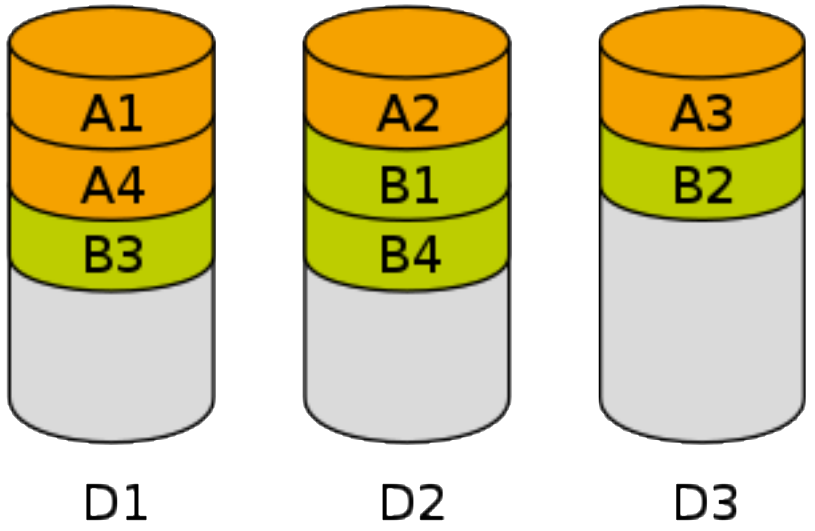
- Along with data in mirroring, keep the parity bit and a Q function defined by Reed-Solomon codes (double chunk failure is recoverable)

	physical disks			
logical chunks	D_1	D_2	P	Q
L_1	C_{11}	C_{21}	C_{31}	C_{41}
L_2	C_{12}	C_{22}	C_{32}	C_{42}
L_3	C_{13}	C_{23}	C_{33}	C_{43}

Supporting Flexibility in RAID Levels

- A single BTRFS storage pool can have various logical chunks at different RAID levels
  - Decouples the top level logical structure from the low-level reliability and striping mechanisms

- Useful for many operations
  - Changing the RAID levels on the fly
  - Changing stripe width
  - Giving different sub-volumes different RAID levels



Data Striping

Image source: Wikipedia

Device Addition and Removal

(a) 2 disks	logical chunks	disk 1	disk 2	
	L_1	C_{11}	C_{21}	
	L_2	C_{12}	C_{22}	
	L_3	C_{13}	C_{23}	
(b) disk added				disk 3
	L_1	C_{11}	C_{21}	
	L_2	C_{12}	C_{22}	
	L_3	C_{13}	C_{23}	
(c) rebalance				
	L_1	C_{11}	C_{21}	
	L_2		C_{22}	C_{12}
	L_3	C_{13}		C_{23}

Defragmentation

- **Simple Approach**:
  1. Read the file
  2. COW
  3. Write to the disk in the next checkpoint

- Likely to make it much more sequential
  - The allocator will try to write it out in as few extents as possible

- **Cons:** Sharing with older snapshot is lost

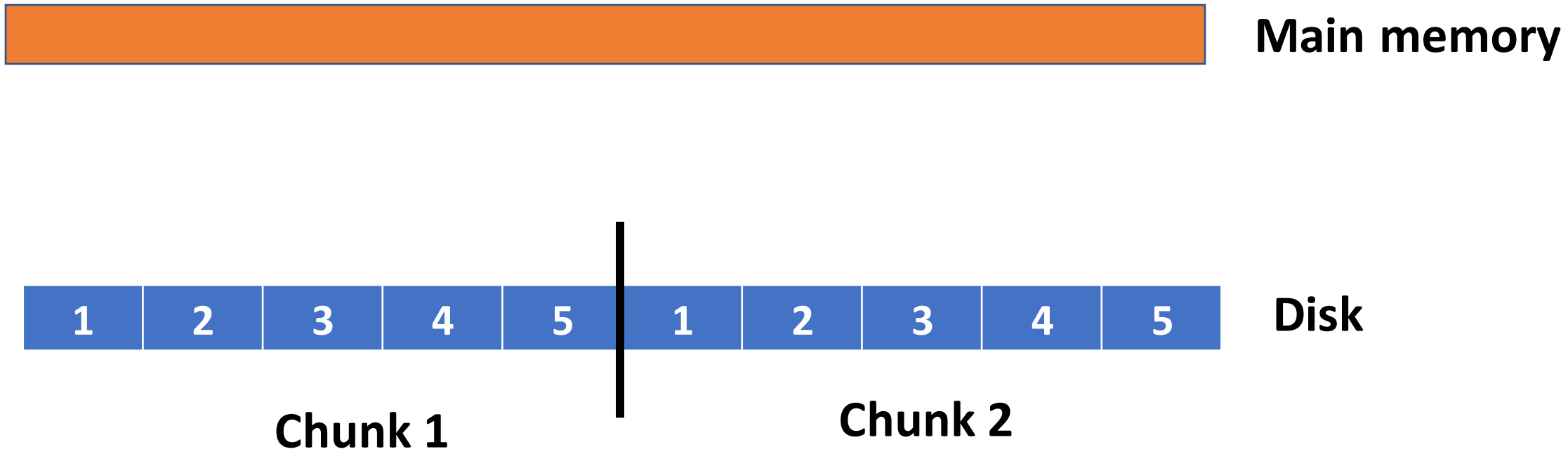
Defragmentation - Relocator

- Need a sophisticated approach for cases
  - Shrinking a filesystem
  - Evicting data from a disk

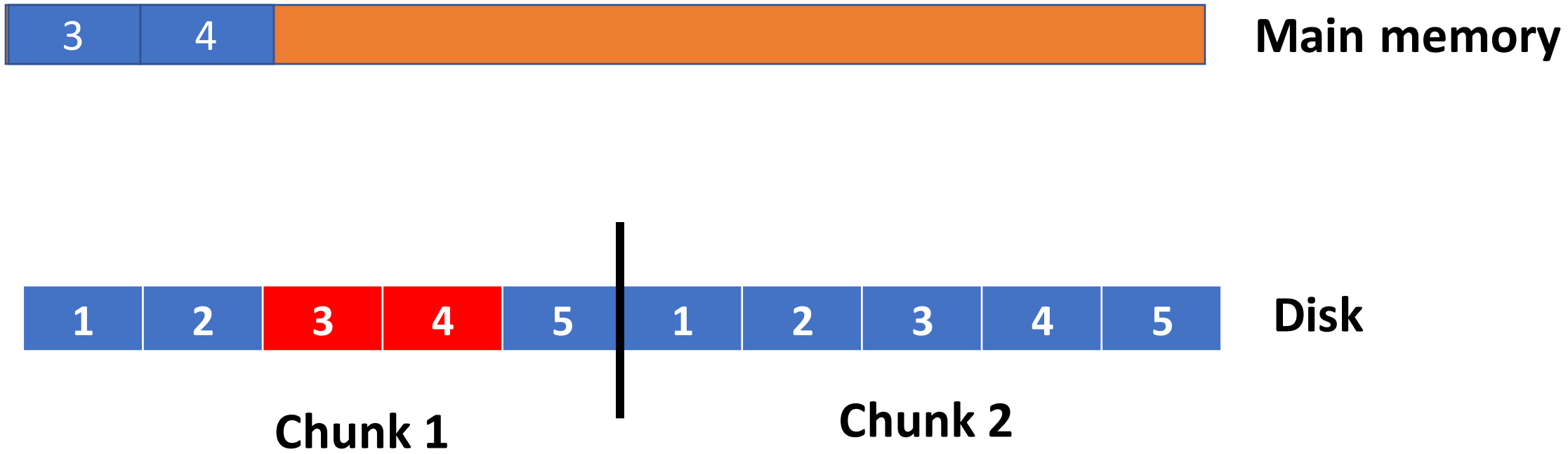
- BTRFS uses a *relocator* - works on a chunk by chunk basis
  1. Move out all the live extents in the chunk
  2. Find all references into a chunk
  3. Fix the references while maintaining sharing

- References are not updated in-place, COW is used.

Memory Shrinking - Broad Idea

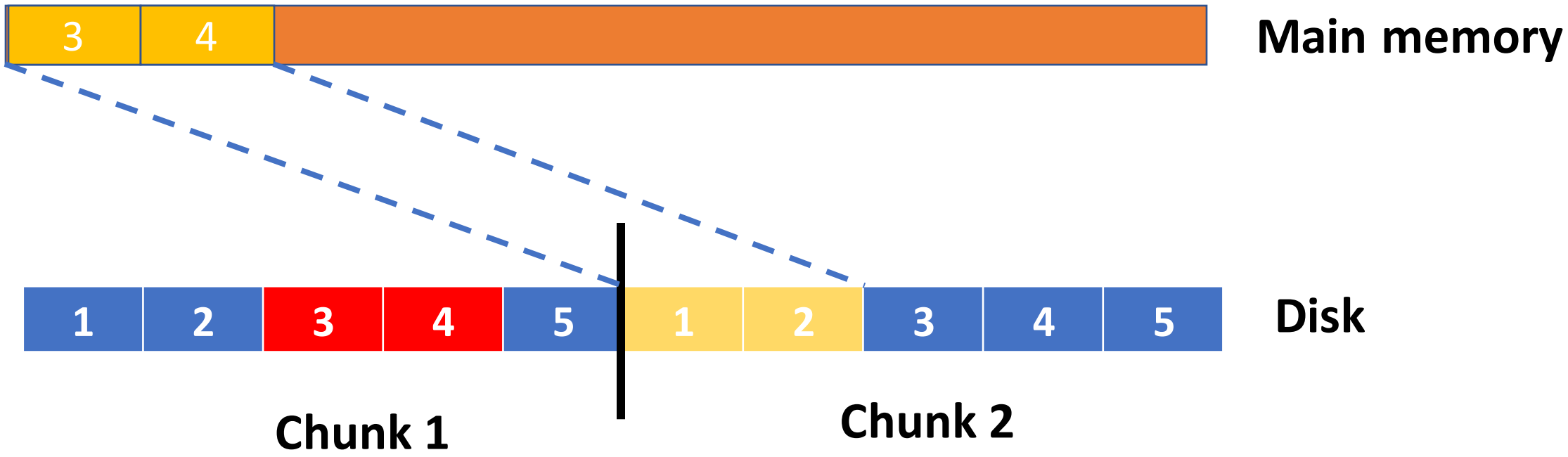


Memory Shrinking - Broad Idea



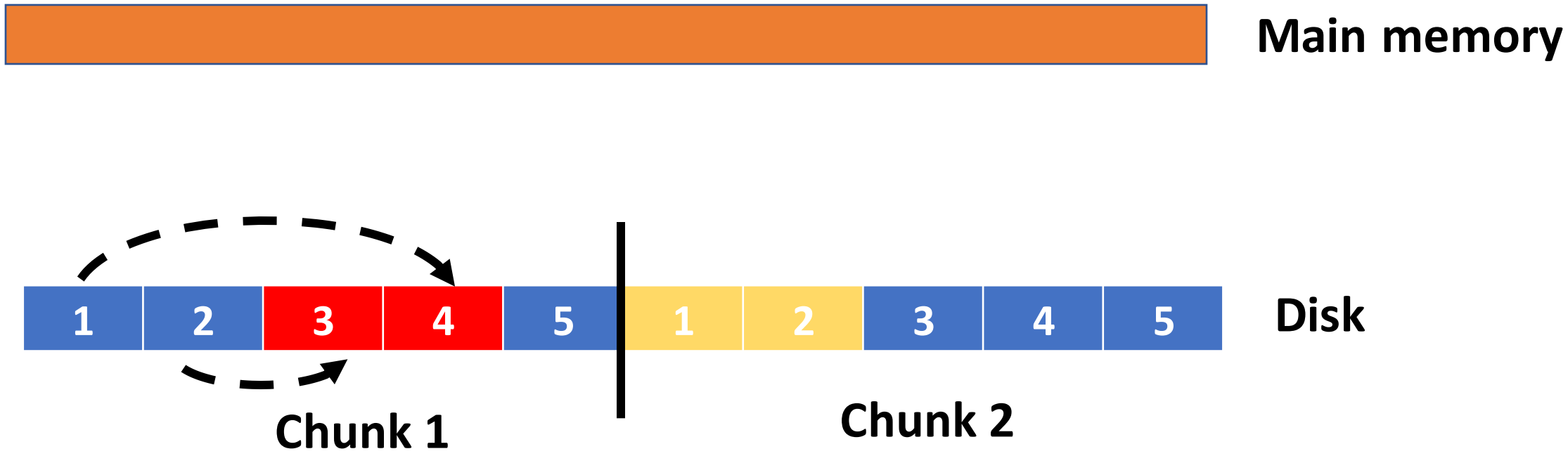
- Frames 3 and 4 are going to get updated -- needs to write back to the disk

Memory Shrinking - Broad Idea



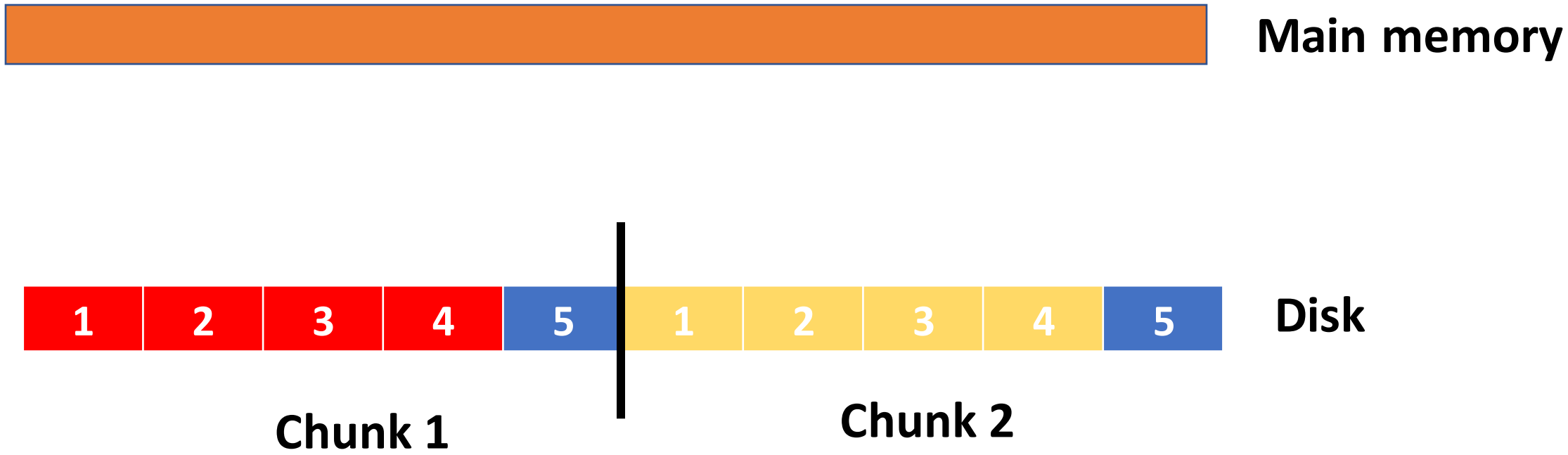
- The frames would be written back to a new location - may be at frames 1 and 2 under chunk 2

Memory Shrinking - Broad Idea



- Say in Chunk 1, Frame 1 has a reference to Frame 4 and Frame 2 has a reference to Frame 3

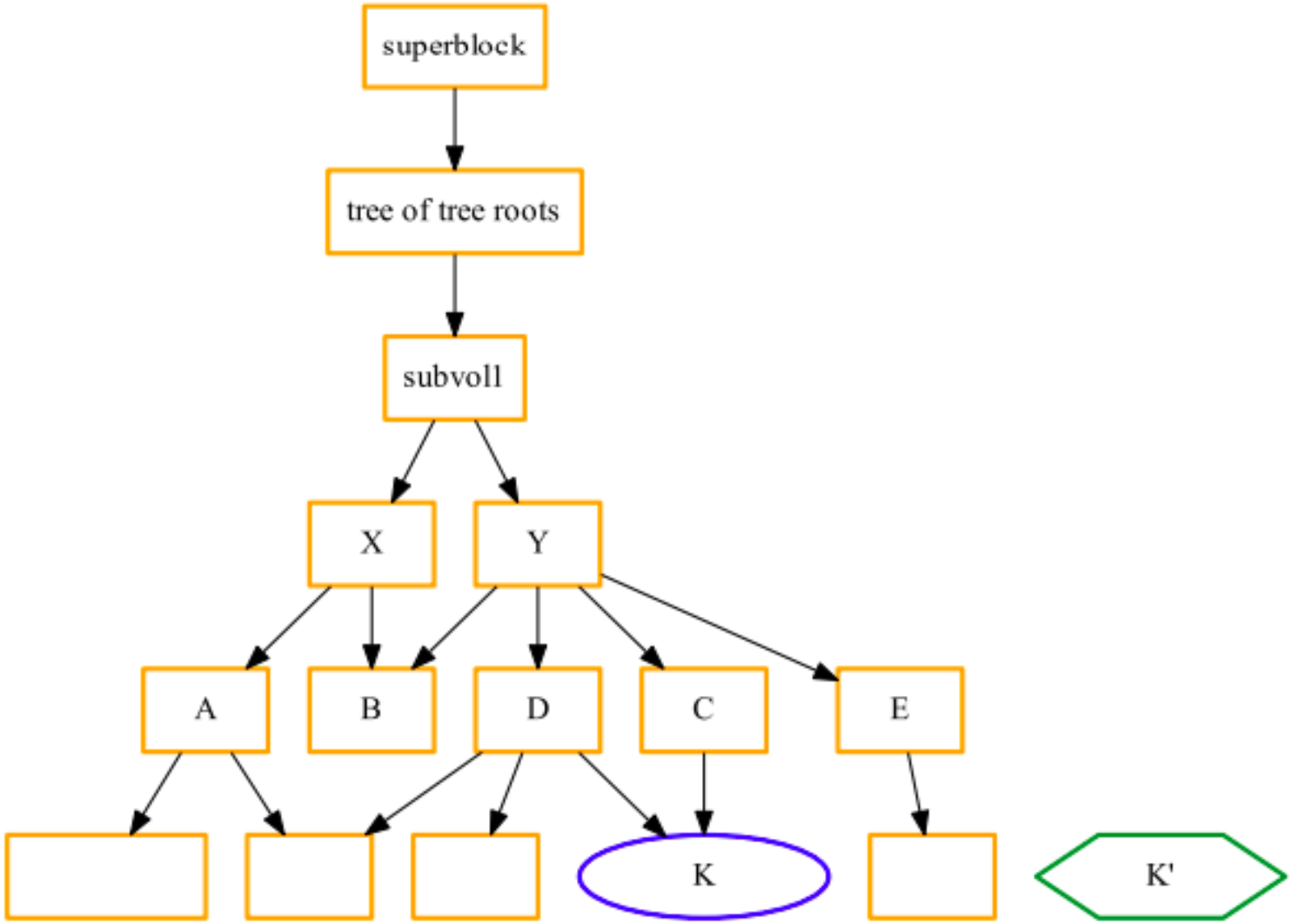
Memory Shrinking - Broad Idea



- Relocate all the four frames in Chunk 2, and make the disk space free in Chunk 1

Memory Shrinking in BTRFS

- Let the extent K needs to be relocated.

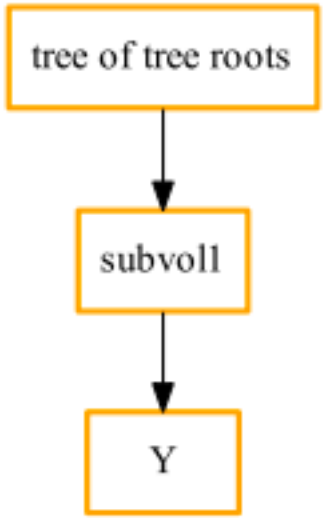


Memory Shrinking in BTRFS

- BTRFS maintains a backreference cache in the extent tree to find out the upper level tree blocks that directly or indirectly reference the chunk.

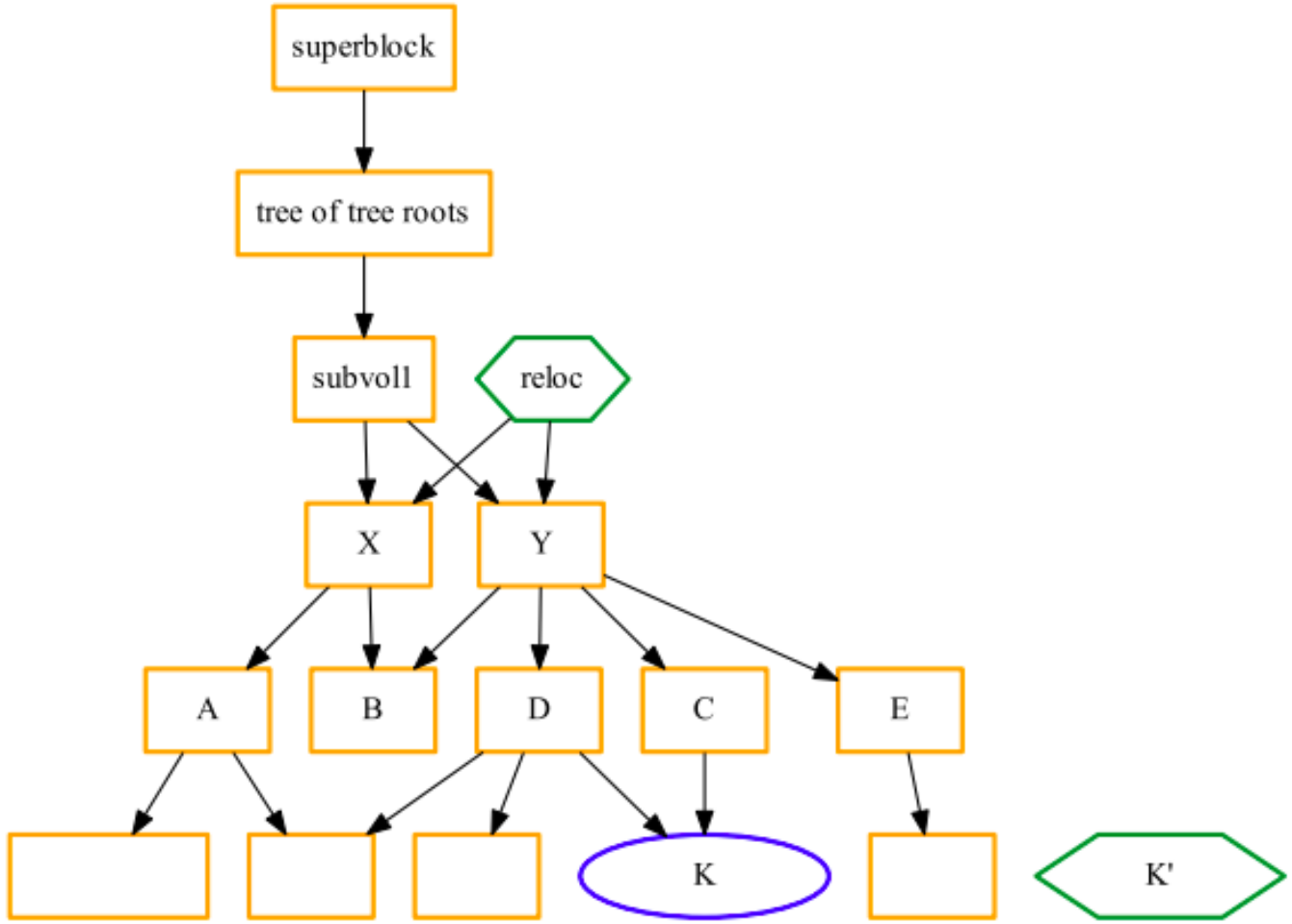
- tree of tree roots and subvol1 are stored as the back reference for node Y

- The backreference cache speeds up searching for the extents that needs to be relocated while relocating a leaf extent.



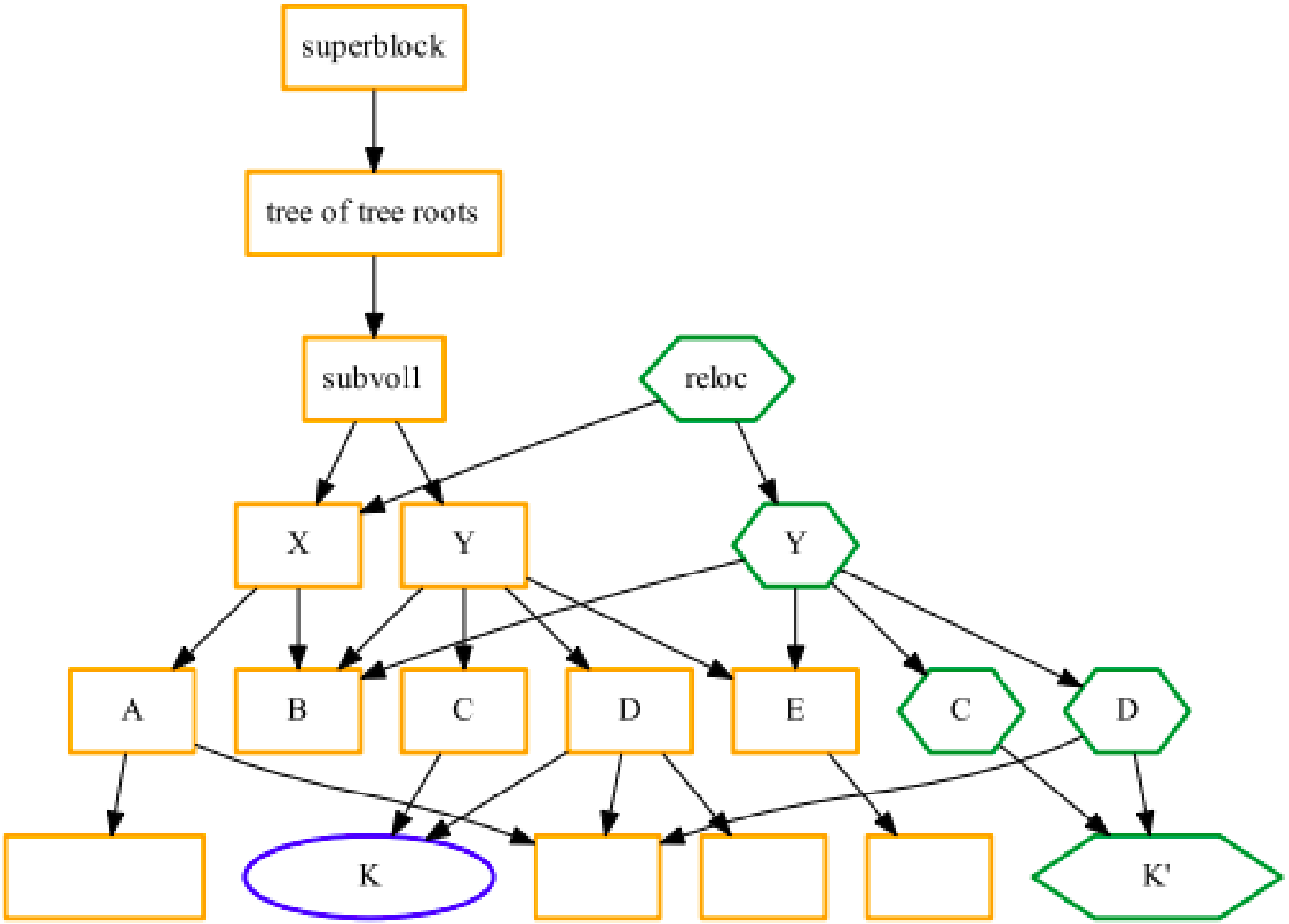
Memory Shrinking in BTRFS

- A list of sub-volume trees that reference the chunk from the backreference cache is calculated; these trees are subsequently cloned - the cloned trees are called **relocation (reloc) trees**



Memory Shrinking in BTRFS

- COW is used to fix the references in the reloc trees.



Memory Shrinking in BTRFS

- Merge the reloc tree with the original file system tree and drop the reloc tree from the main memory