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;
}
```

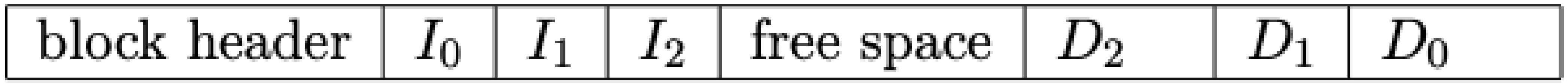
- Internal tree nodes hold only `[key, block-pointer]` pairs

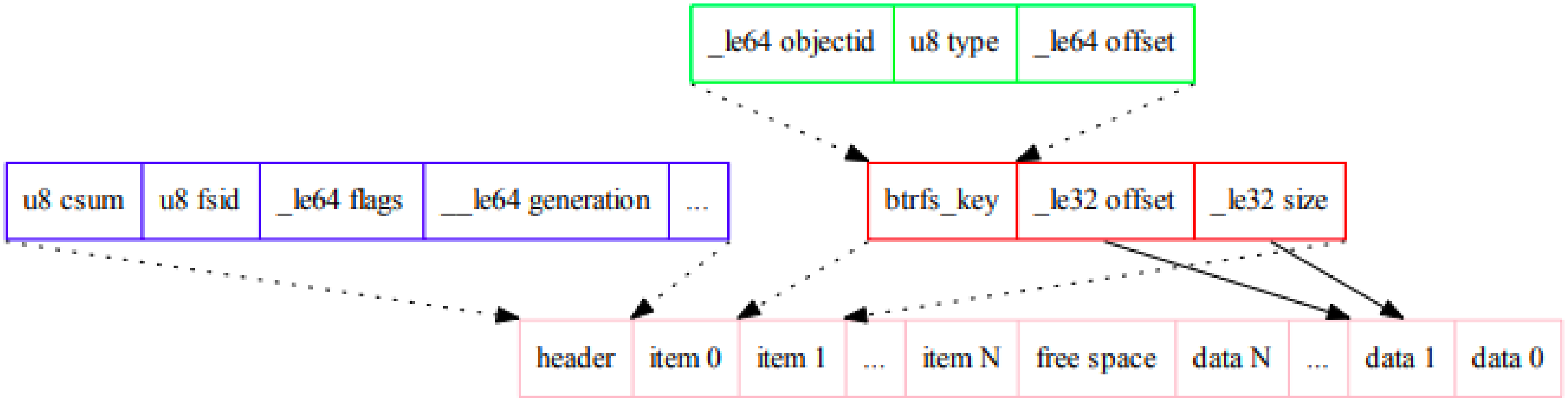
- Leaf nodes hold arrays of `[item, data]` pairs.

- 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.

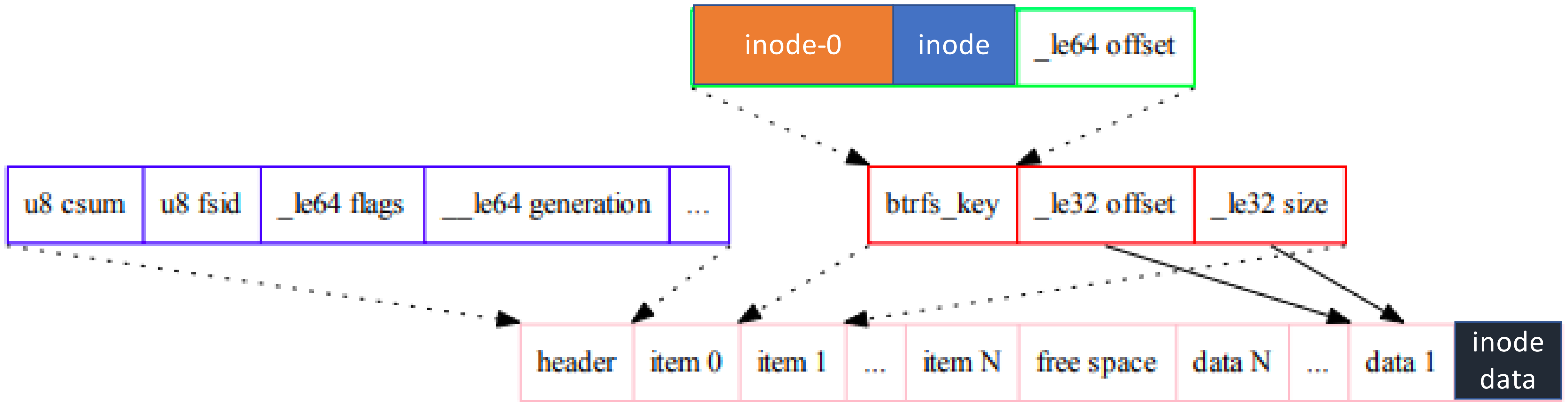




inode

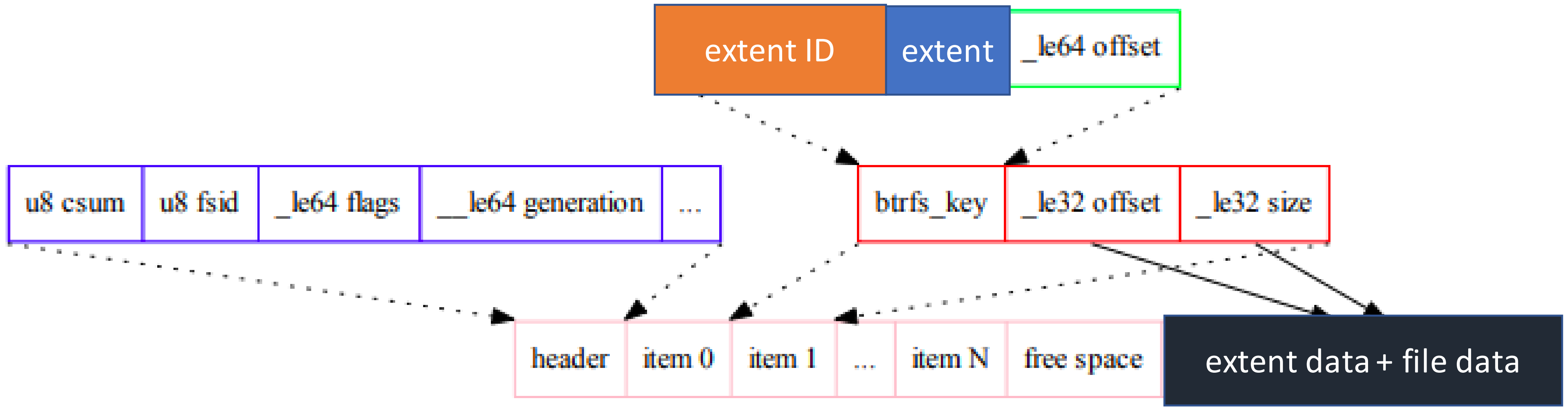
- Inodes are storaed in a inode item at offset zero in the key
  - Have a type value of one
- The lowest valued key for a given object
  - Store the traditional stat data for files and directories
- The inode structure is relatively small
  - Does not contain embedded file data or extended attribute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

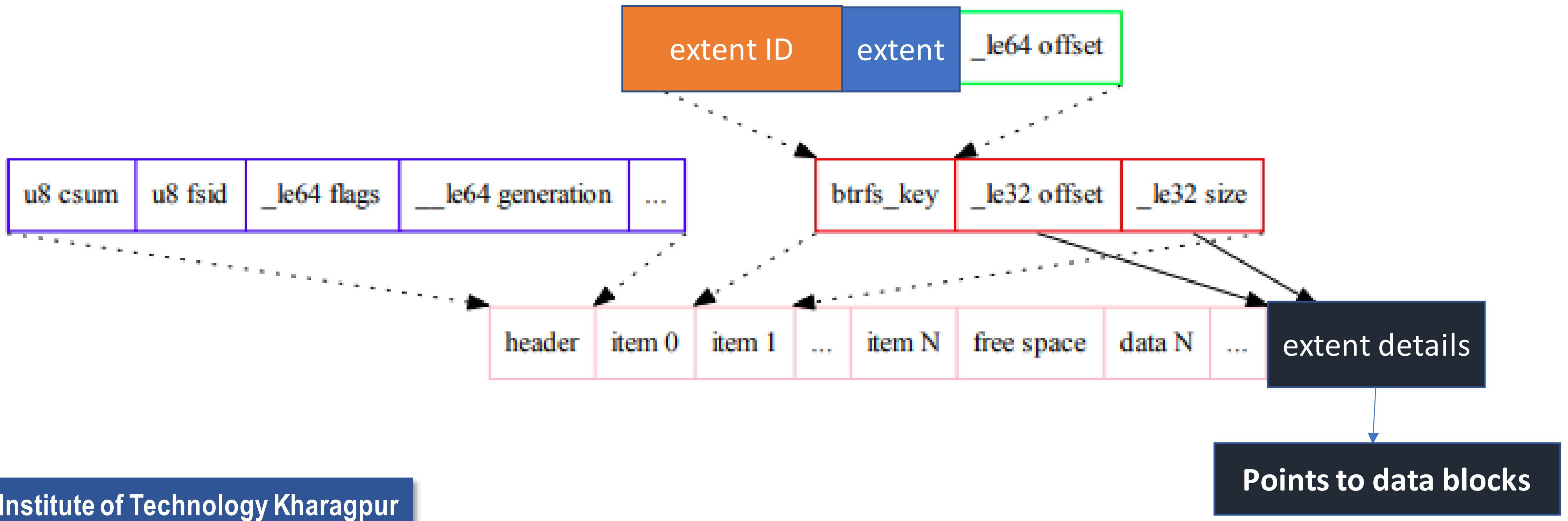
Small Files in the Leaf

- 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



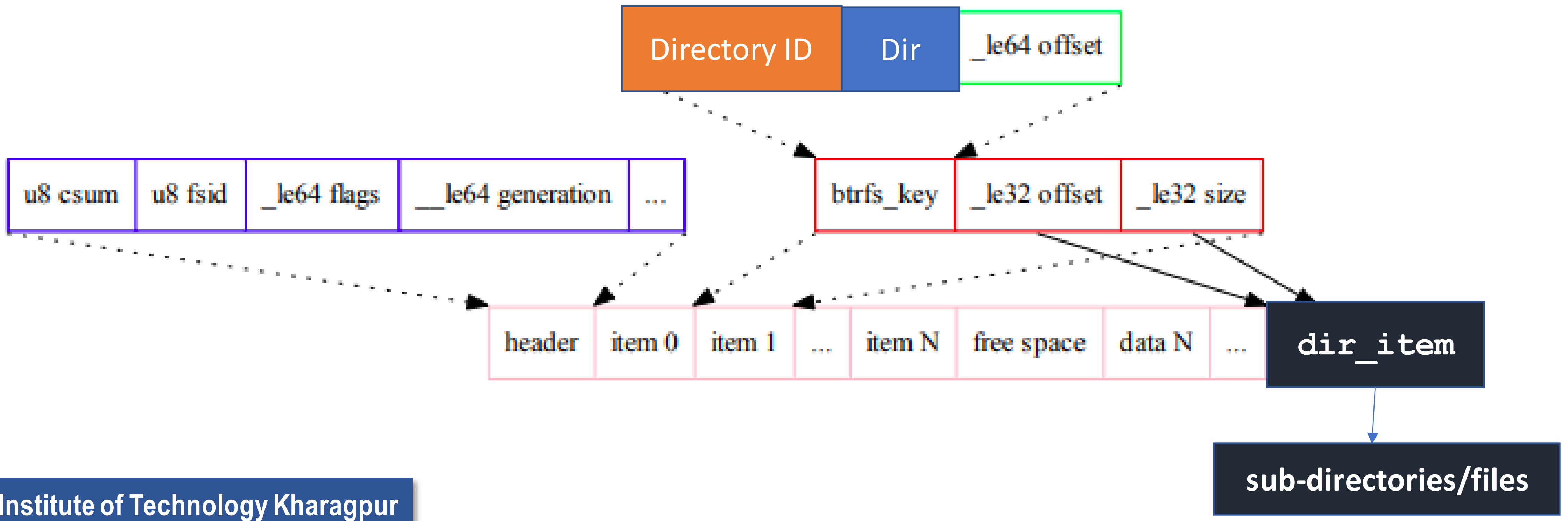
Large Files

- 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.

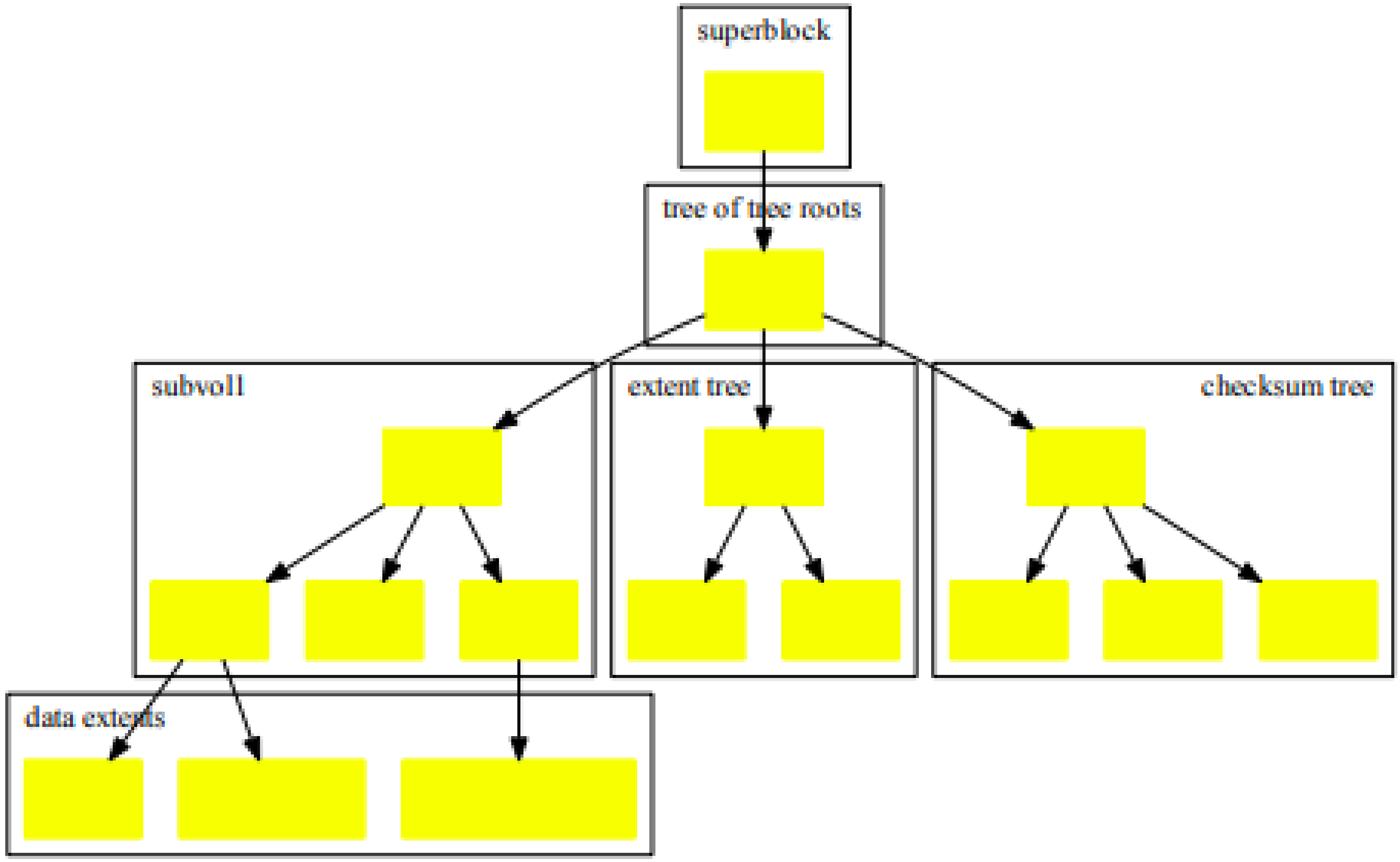


Directory

- 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

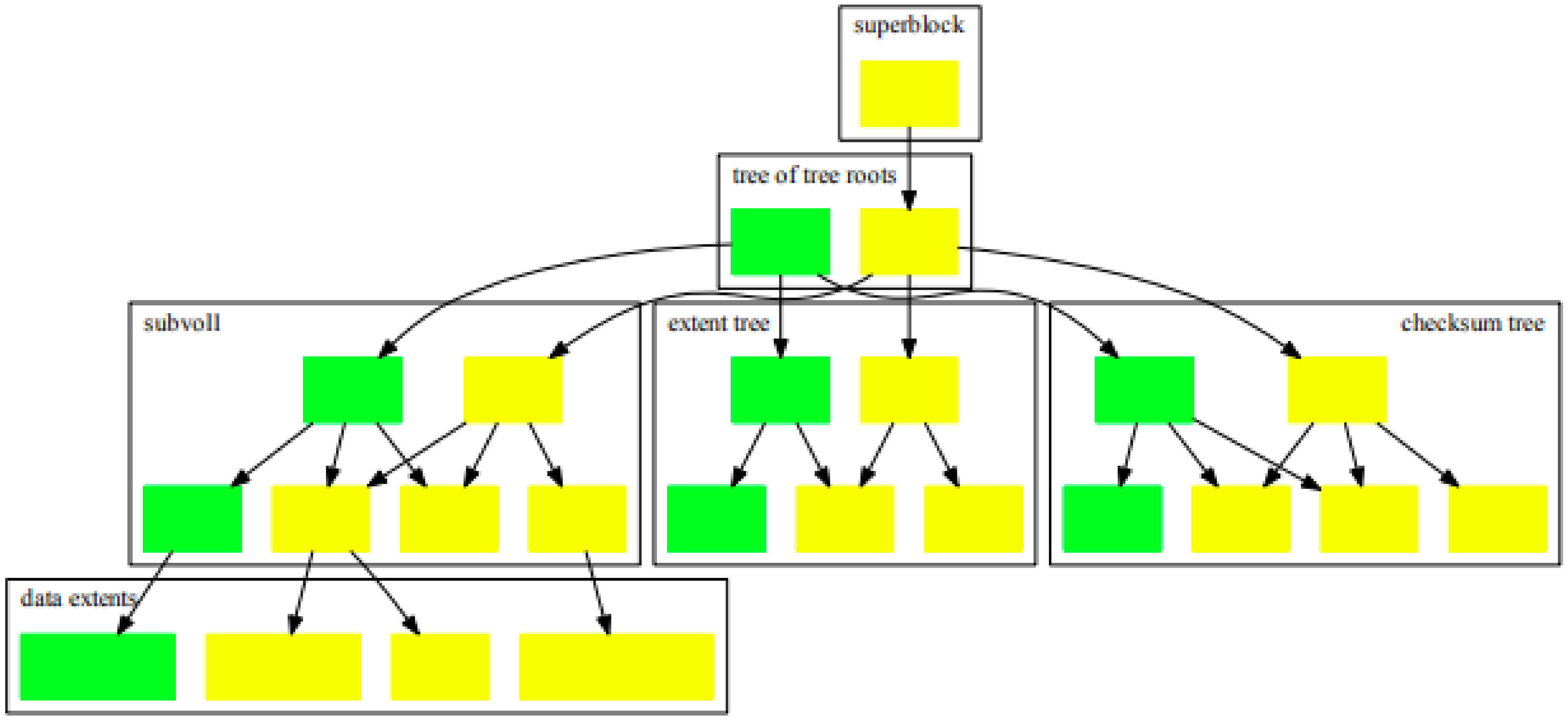
- Tracks allocated extents in **extent items**
  - Serves as an on-disk free-space map

- 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.

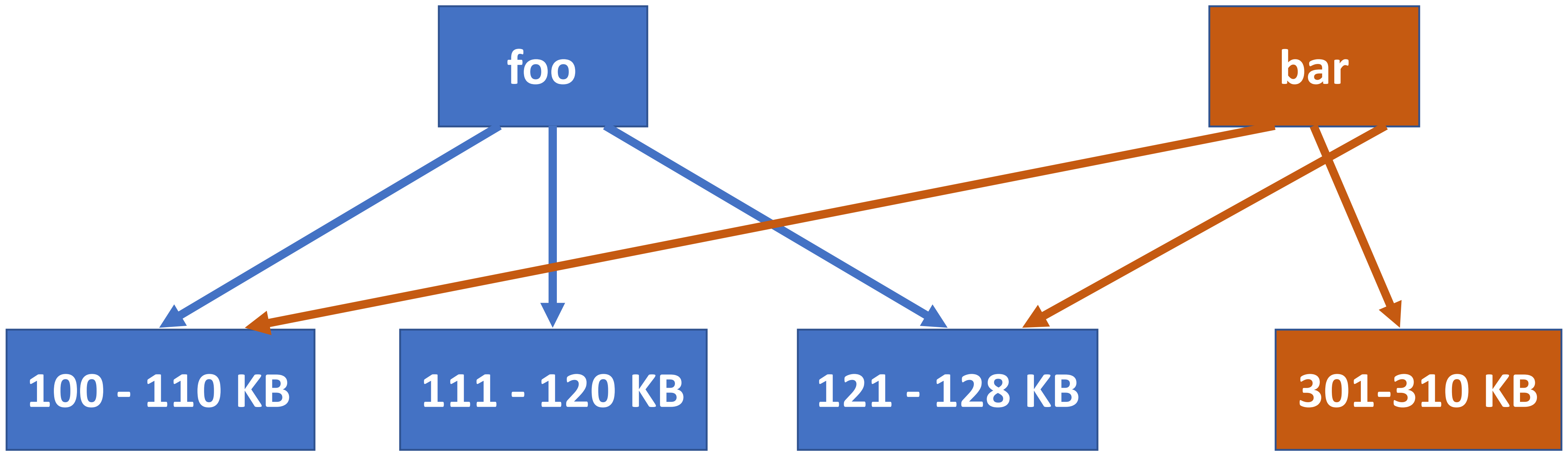
Extent Allocation Tree and Back References

- 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`

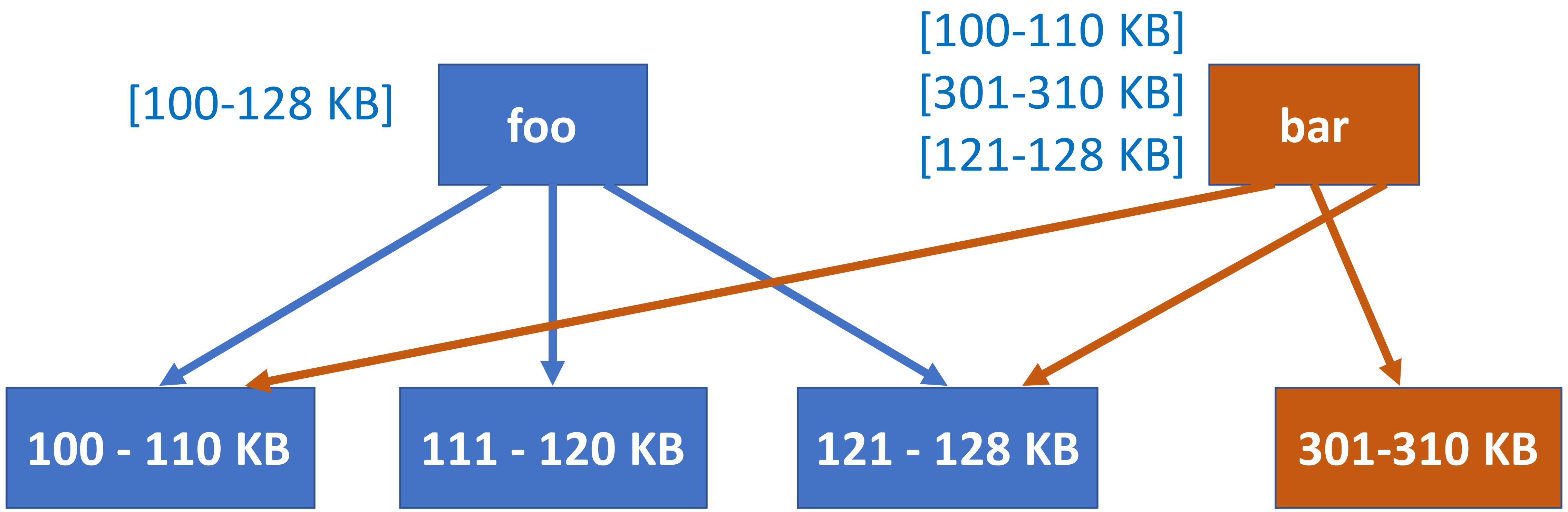
File	**On disk extents**
foo	**100-128KB**
bar	**100-110KB, $\underbrace{\text{301-310KB,}}$ 121-128KB**

This is located much further in the disk

Extent Allocation Tree and Back References

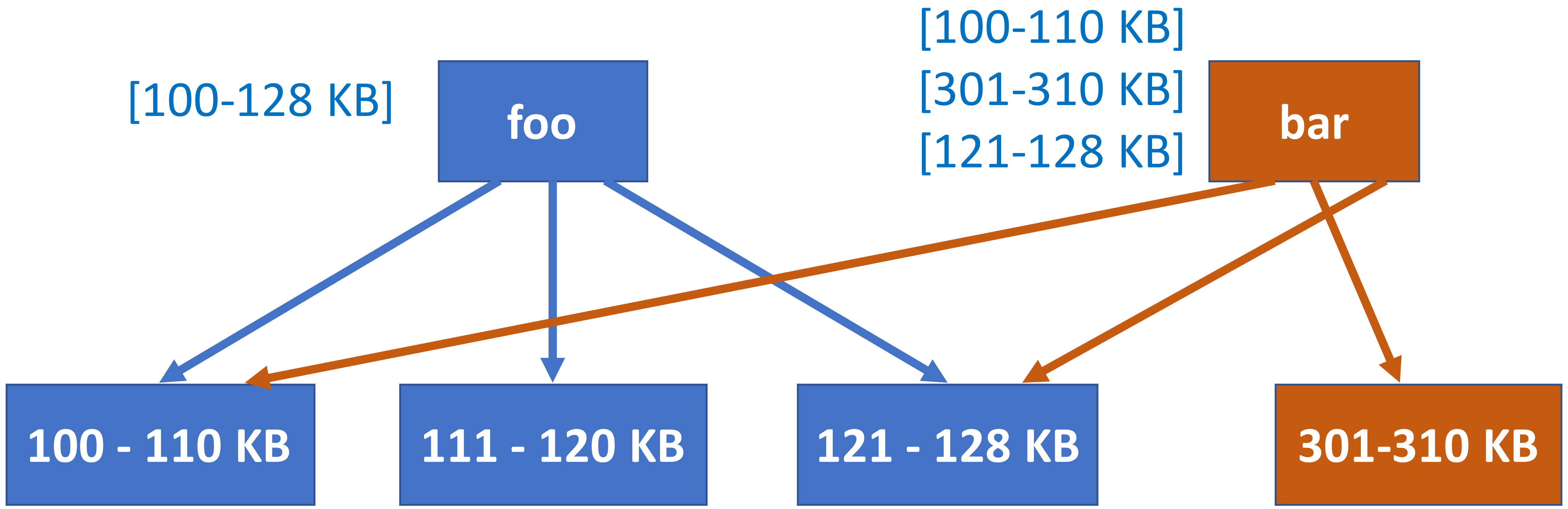


Extent Allocation Tree and Back References

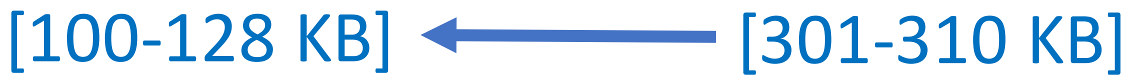


- The extent-item keeps track of all the references, to allow moving the entire extent at a later time

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



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

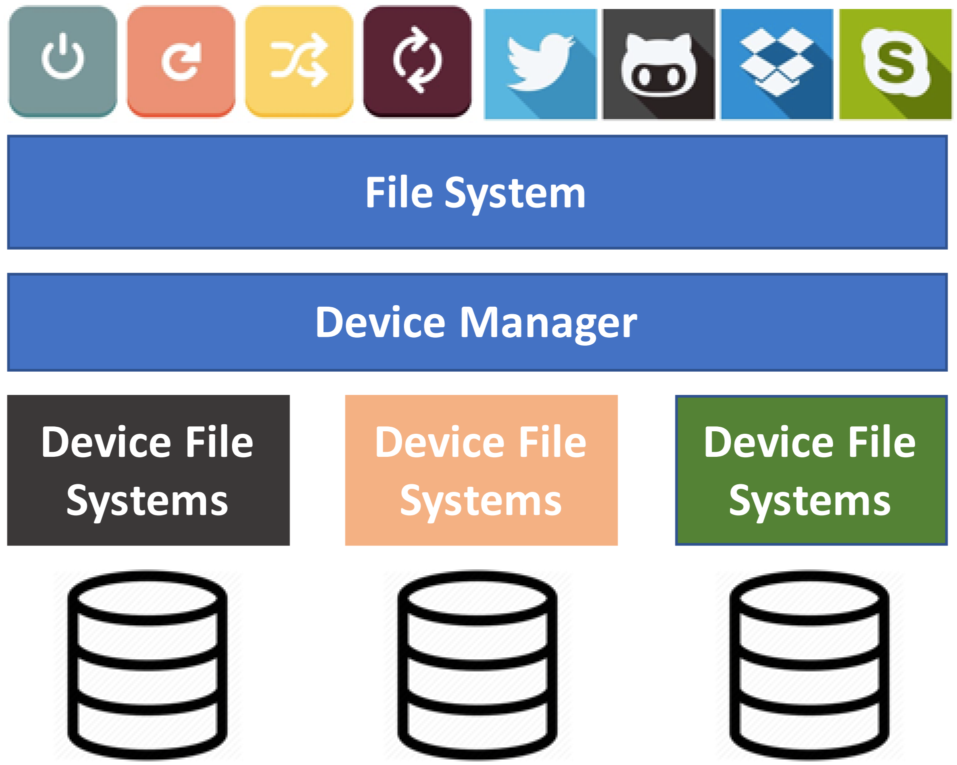
Multiple Device Support in BTRFS

- 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

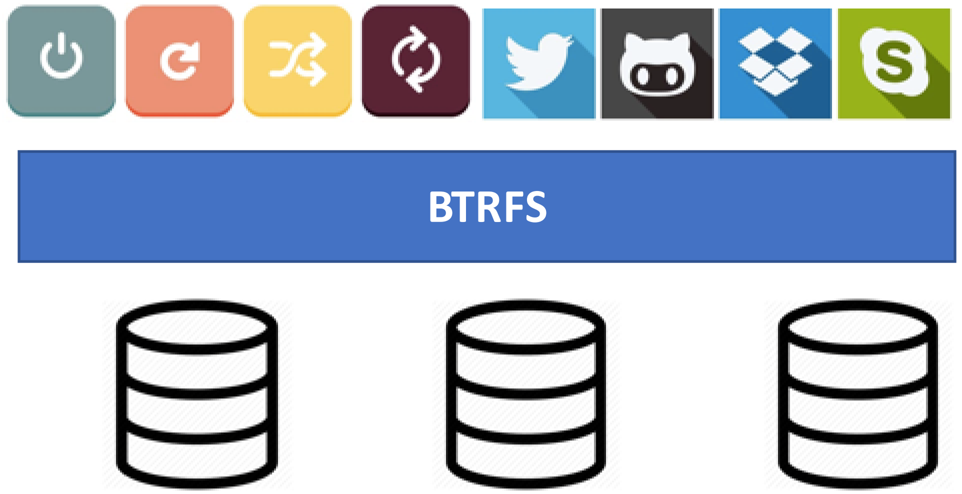
- The chunk and device trees are small - can typically be cached in memory
  - Reduces the performance cost of an added indirection layer

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

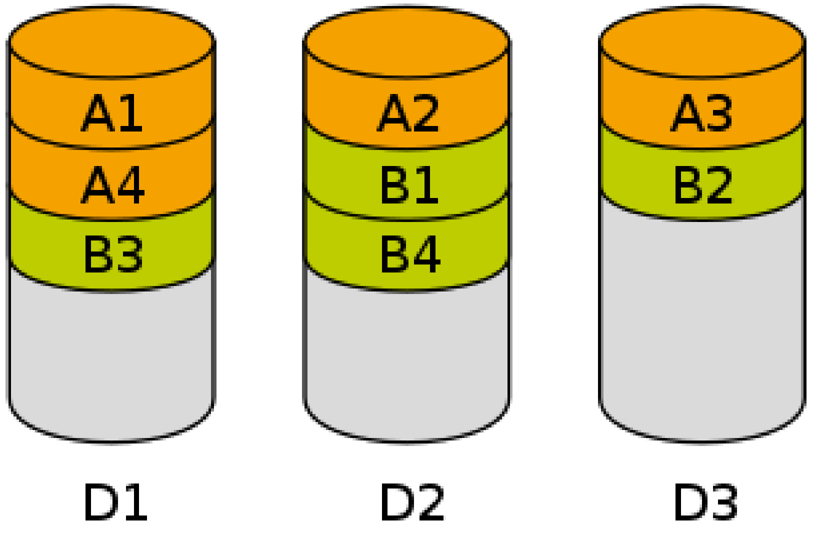
- Along with data in mirroring, keep the parity bit and a Q function defined by Reed-Solomon codes (double chunk failure is recoverable)

	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

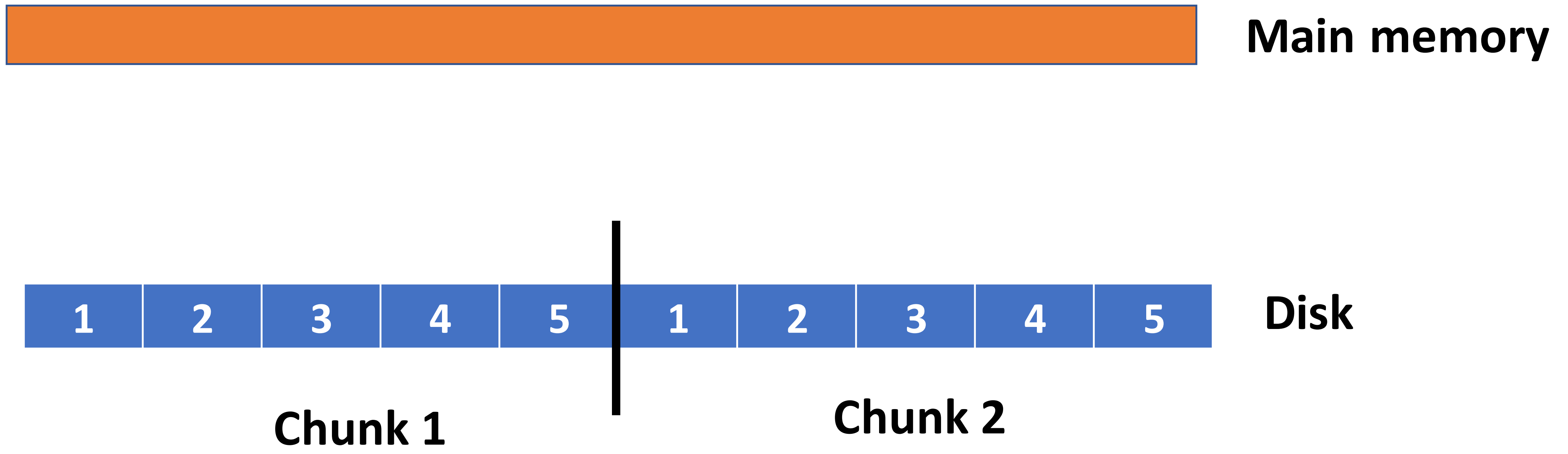
Defragmentation - Relocator

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

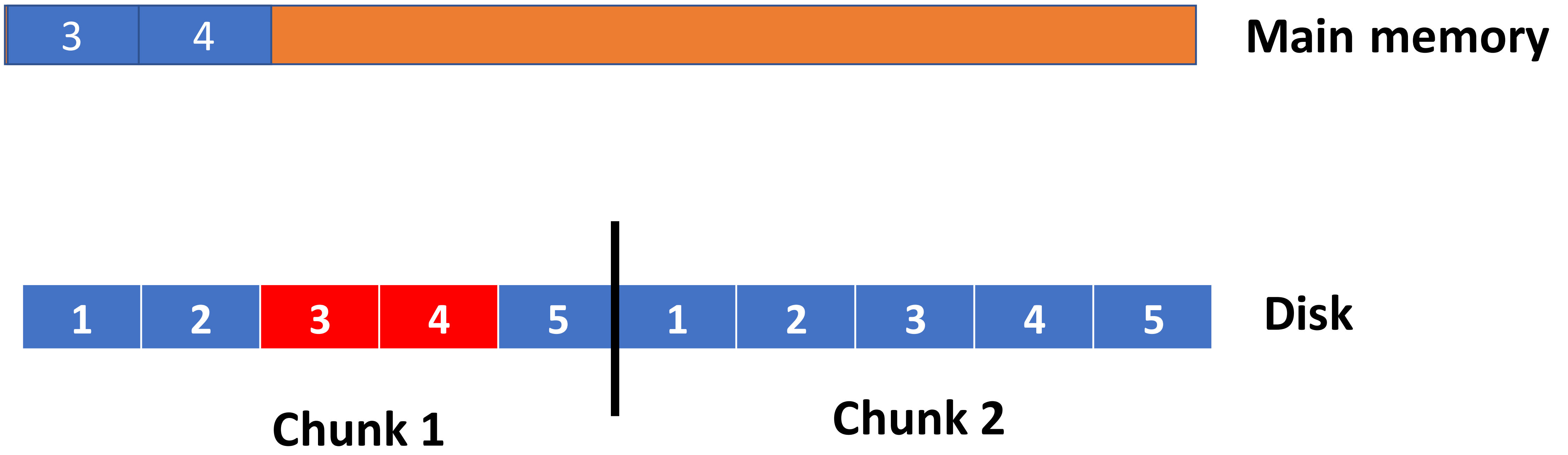
- BTRFS uses a *relocator* - works on a chunk by chunk basis
  1. Move out all the live extents in the chunk
  2. Find all references into a chunk
  3. Fix the references while maintaining sharing

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

Memory Shrinking - Broad Idea

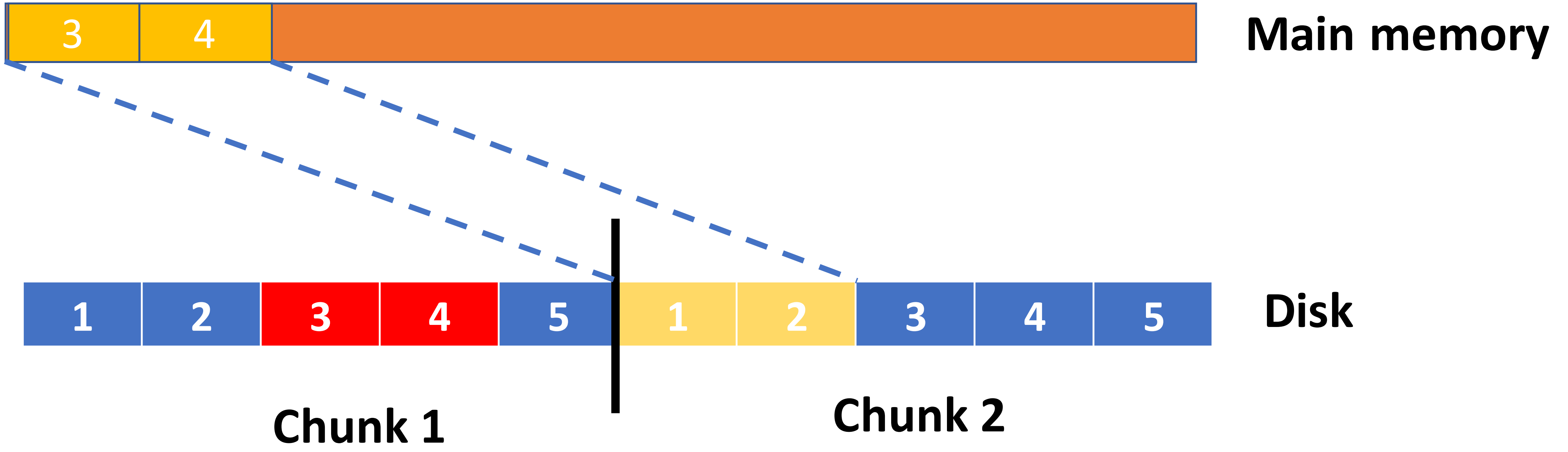


Memory Shrinking - Broad Idea



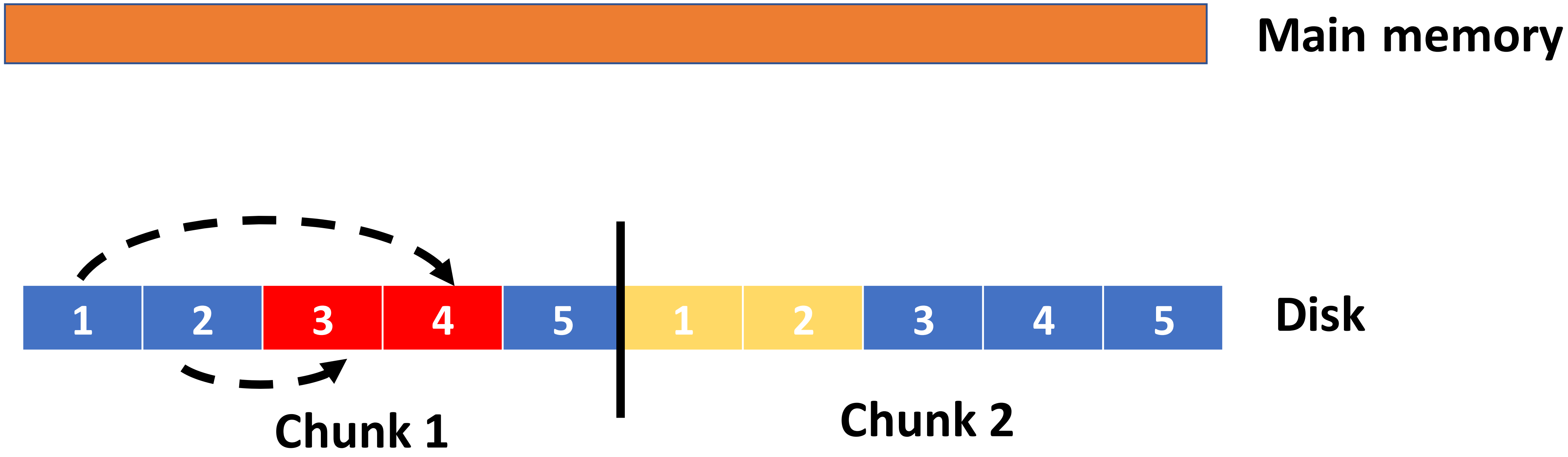
- Frames 3 and 4 are going to get updated -- needs to write back to the disk

Memory Shrinking - Broad Idea



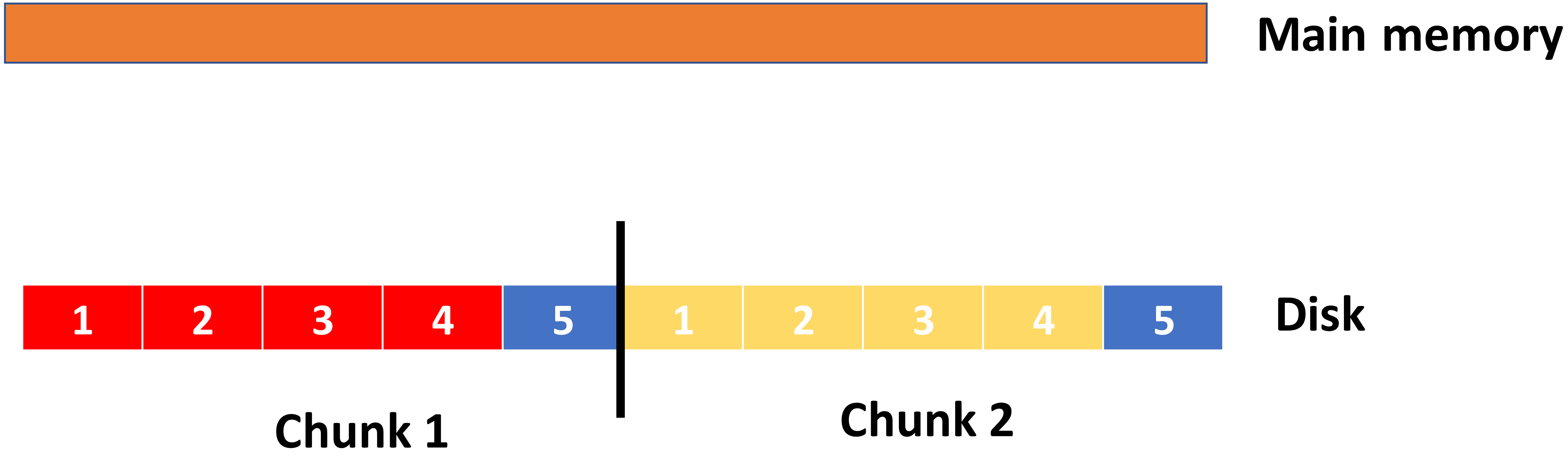
- The frames would be written back to a new location - may be at frames 1 and 2 under chunk 2

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

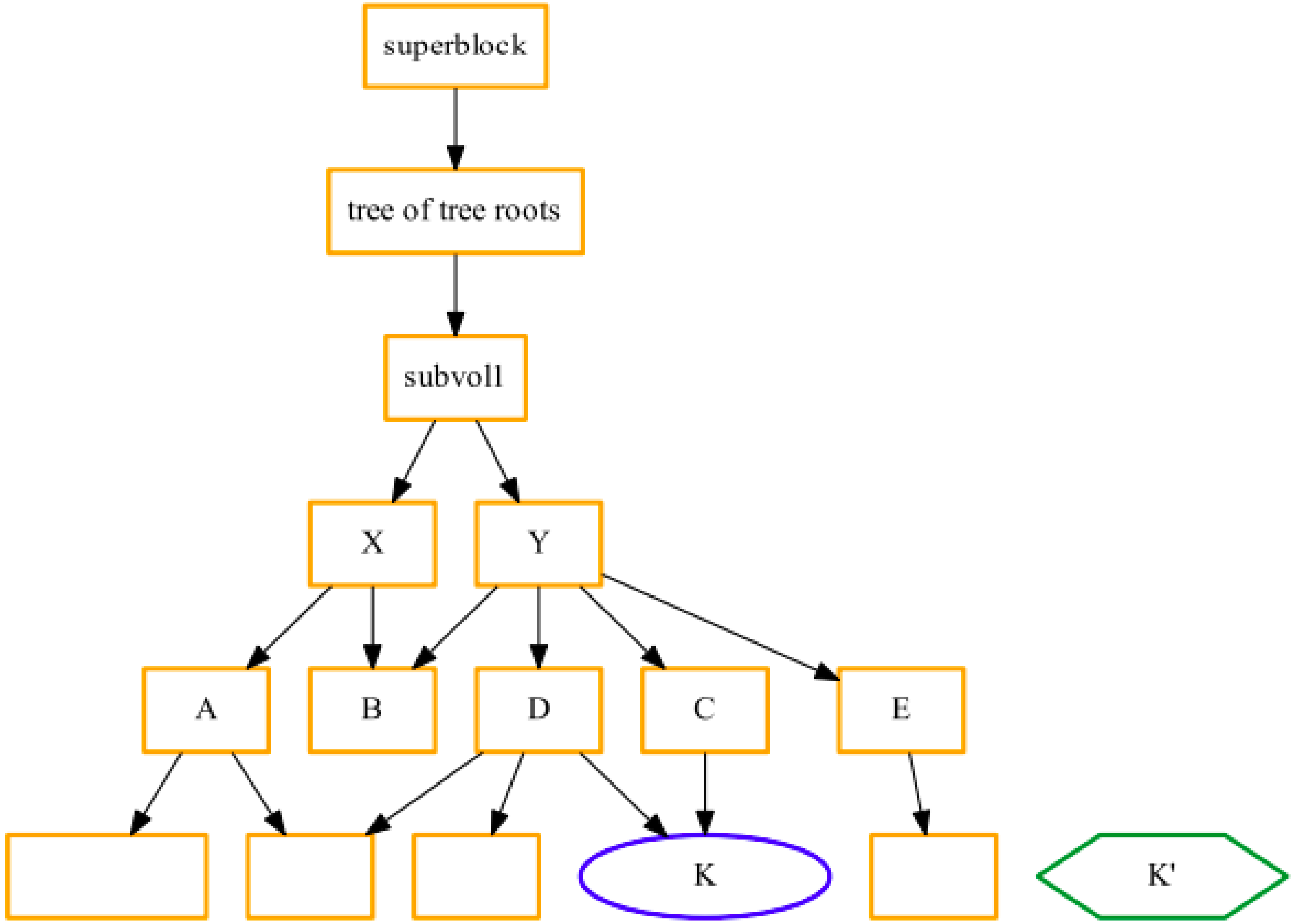
Memory Shrinking - Broad Idea



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

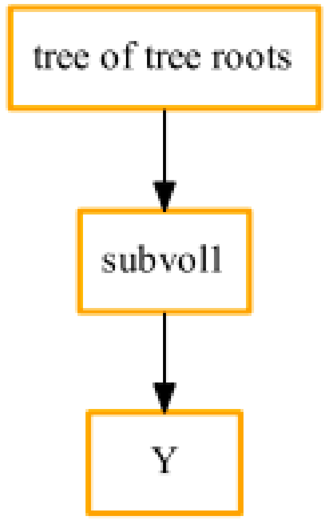
Memory Shrinking in BTRFS

- Let the extent K needs to be relocated.



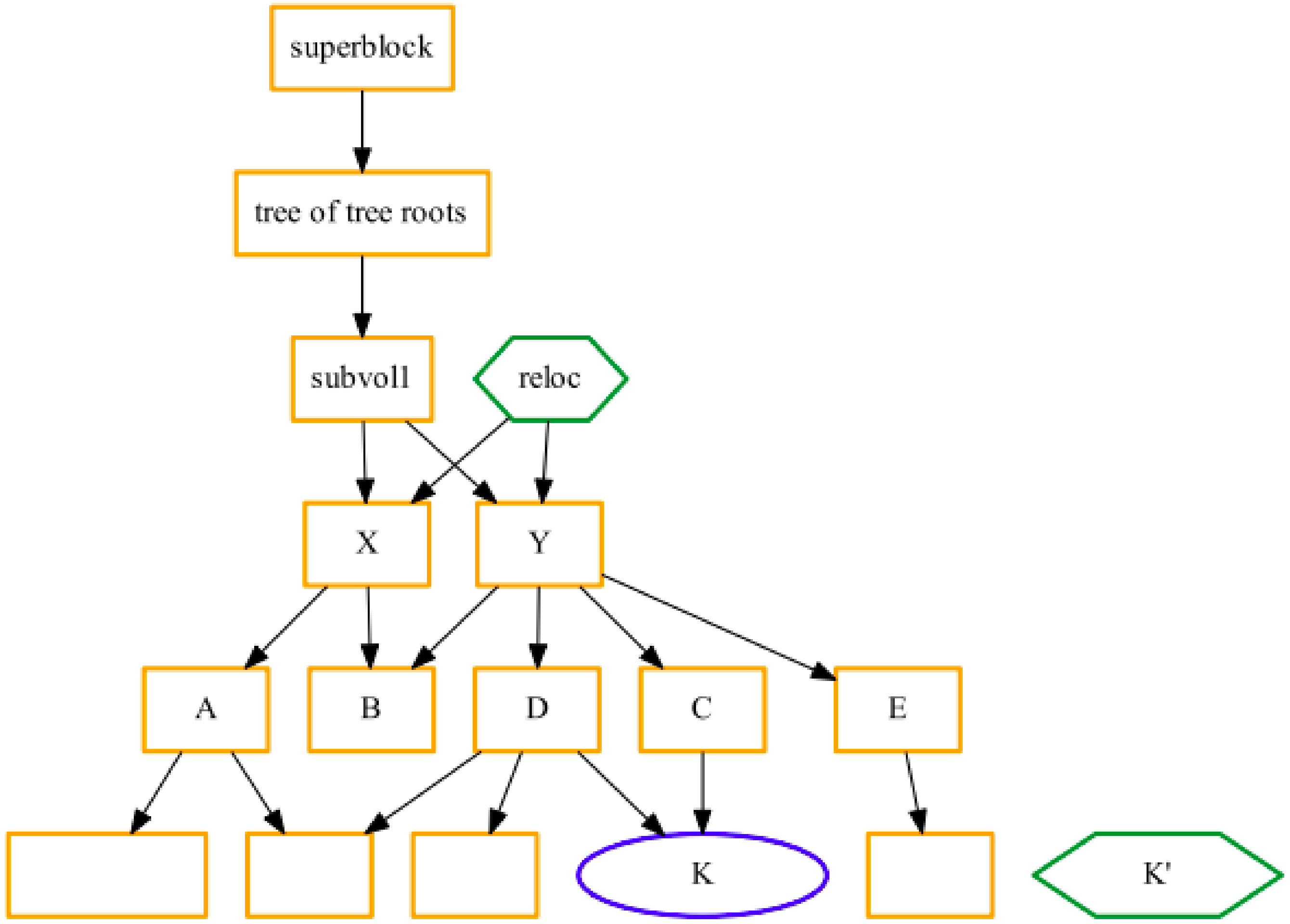
Memory Shrinking in BTRFS

- 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.



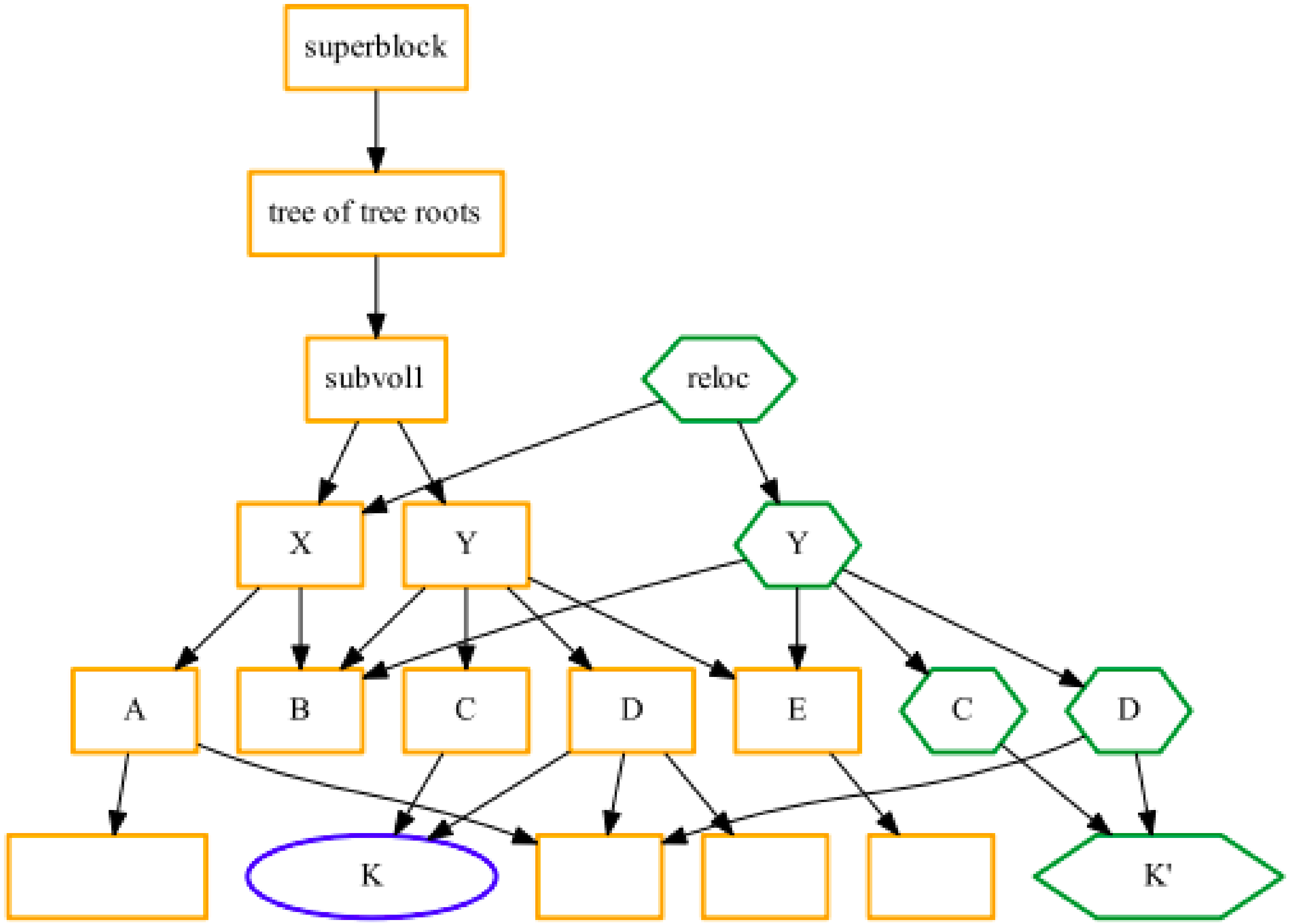
Memory Shrinking in BTRFS

- 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**



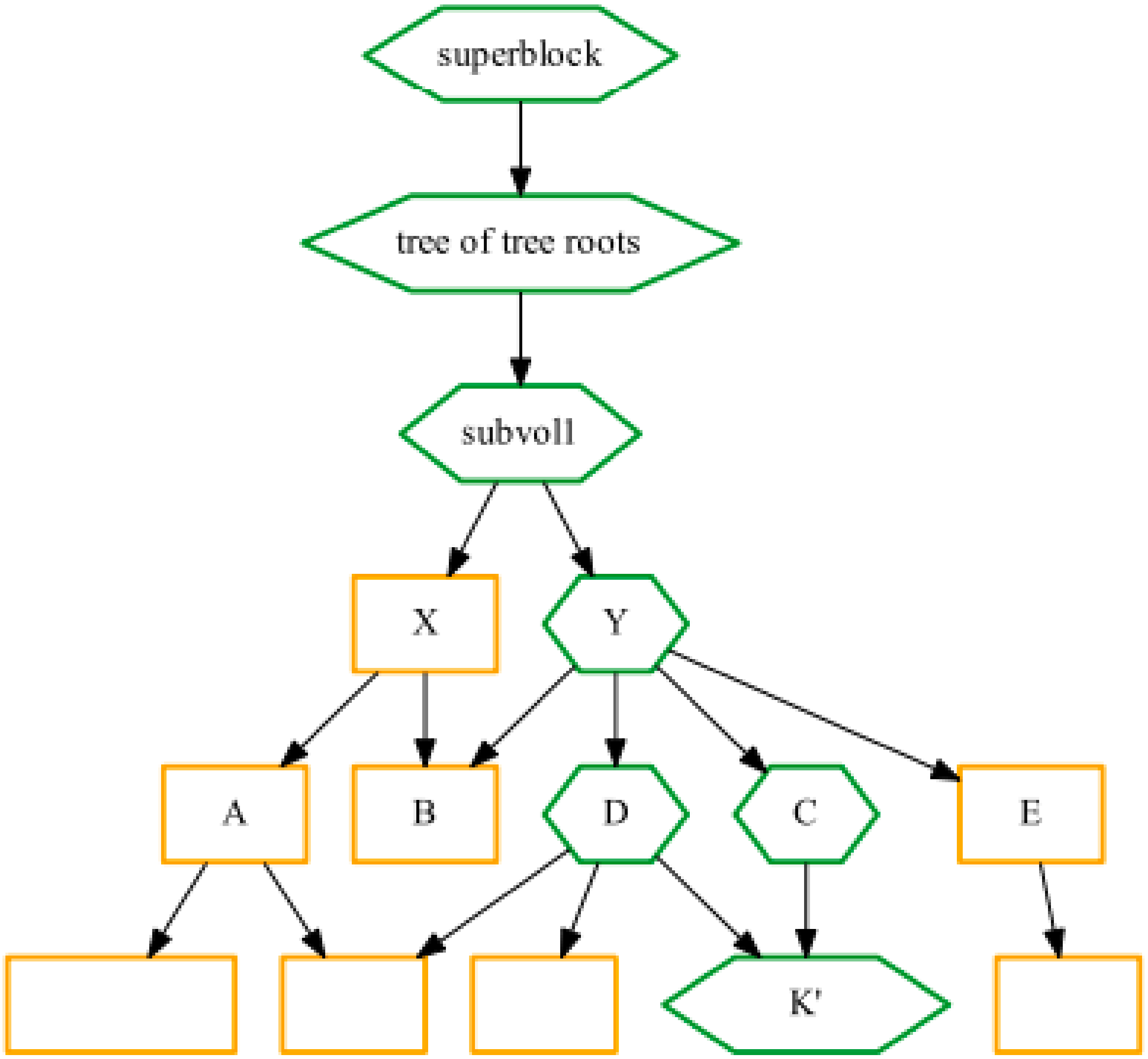
Memory Shrinking in BTRFS

- 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



Happy
Learning