UBIFS file system

NOKIA

Adrian Hunter (Адриан Хантер)
Artem Bityutskiy (Битюцкий Артём)

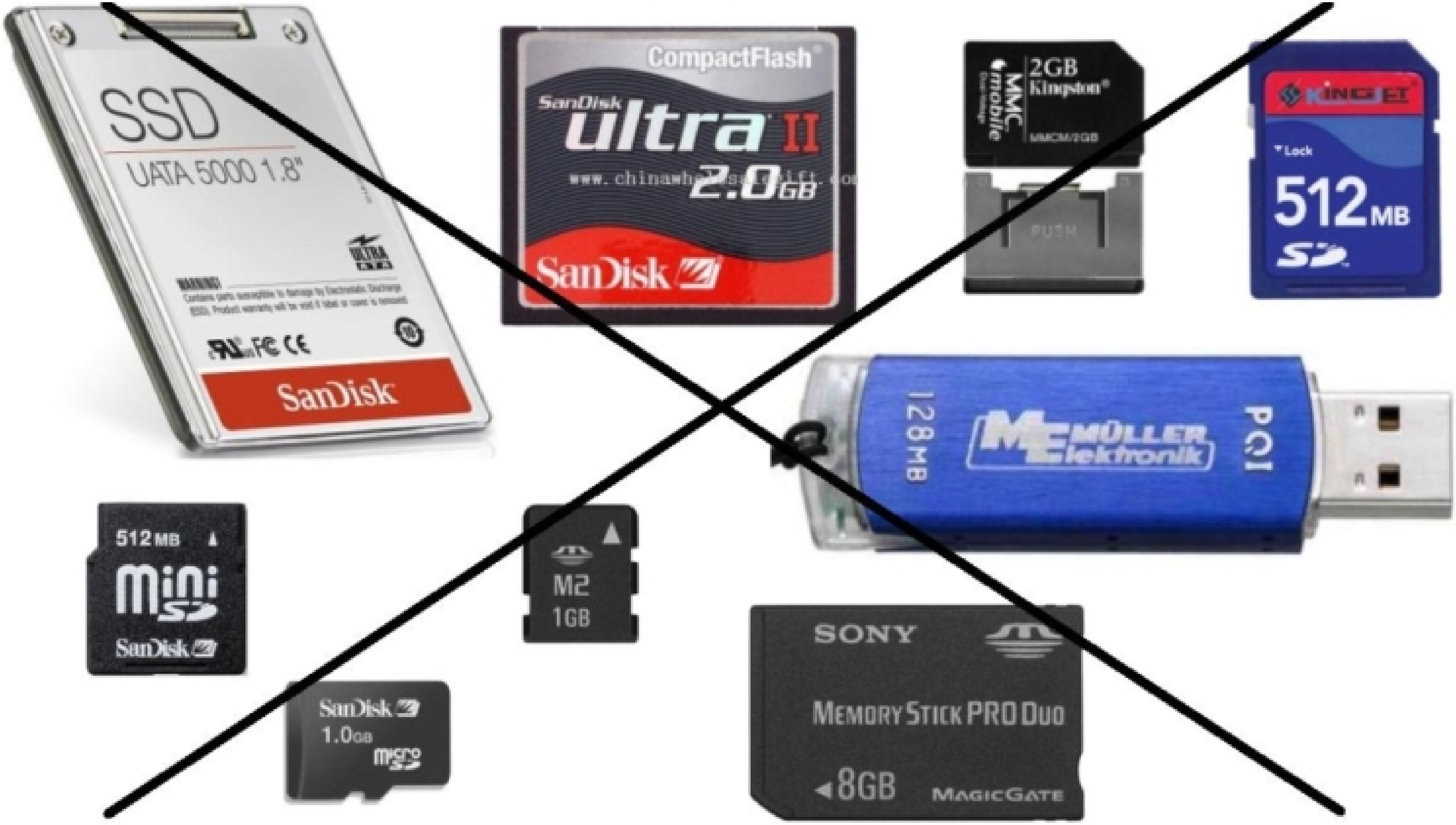
Plan

- **Introduction (Artem)**
- MTD and UBI (Artem)
- UBIFS (Adrian)

UBIFS scope

- UBIFS stands for UBI file system (argh...)
- UBIFS is designed for **raw** flash devices
- UBIFS is **not** designed for SSD, MMC, SD, Compact Flash, USB sticks, and so on
  - I call them “FTL devices”
  - They have raw flash inside, but they are block devices
  - They are very different to raw flash devices

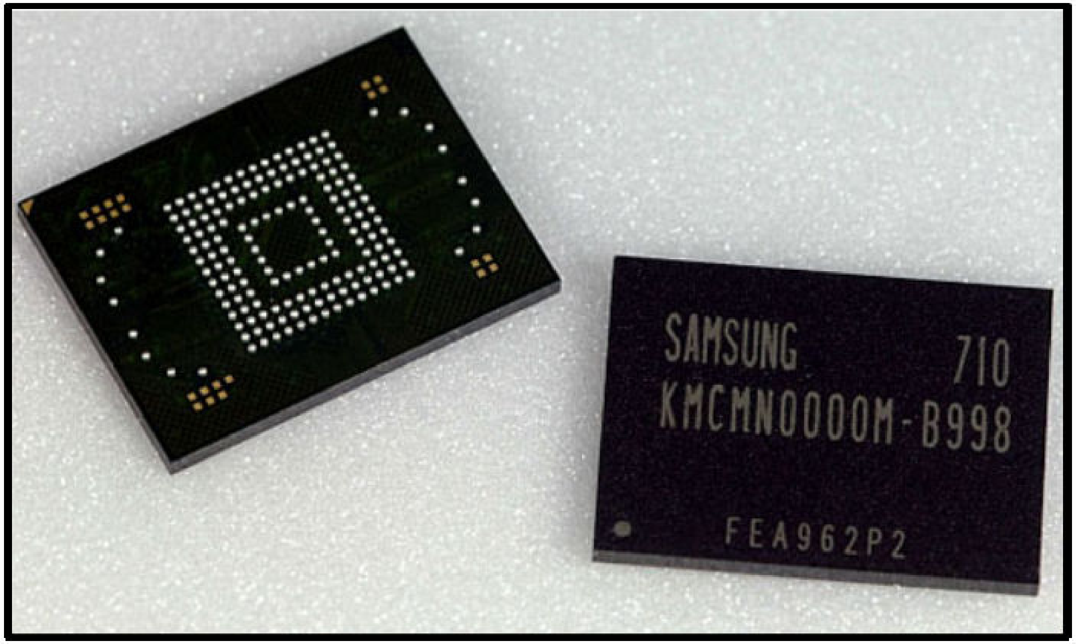
UBIFS scope



SSD
UATA 5000 1.8"
SanDisk
CompactFlash
SanDisk
ultra II
2.0GB
SanDisk
2GB
Kingston
MMC mobile
512MB
SD
128MB
PQI
512MB
mini SD
SanDisk
M2
1GB
SanDisk
1.0GB
micro SD
SONY
MEMORY STICK PRO Duo
8GB
MAGICGATE

UBIFS scope

- UBIFS is designed for raw flash devices
- E.g. NAND, NOR, OneNAND, etc



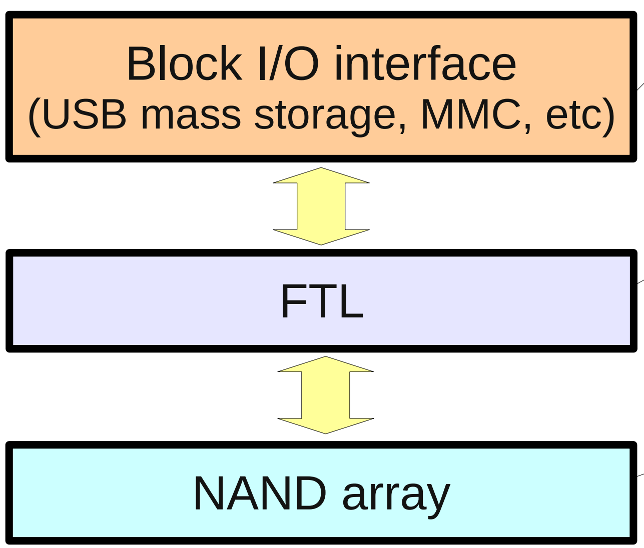
FTL device vs. Raw flash

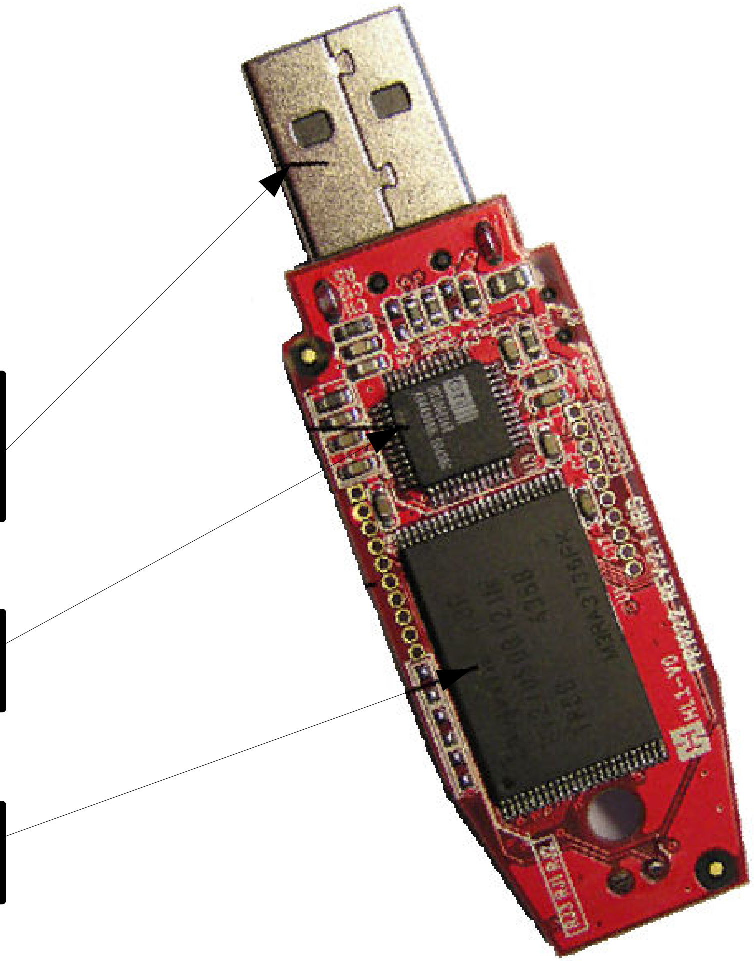
FTL device	Raw Flash
Consists of sectors, typically 512 bytes	Consists of eraseblocks, typically 128KiB
Has 2 main operations: 1. read sector 2. write sector	Has 3 main operations: 1. read from eraseblock 2. write to eraseblock 3. erase the eraseblock
Bad sectors are hidden and re-mapped by hardware	Hardware does not manage bad eraseblocks
Sectors do not wear-out	Eraseblocks wear-out after 10^3-10^5 erase operations

Raw flash and block device are completely different

FTL devices

- FTL stands for "Flash Translation Layer"
- FTL devices have raw flash plus a controller
- Controller runs FTL firmware
- FTL firmware emulates block device





FTL devices – cons and pros

- One may run trusted traditional software (e.g., ext3)
- Standardized
- Black box, FTL algorithms are trade secrets
- Fast wear-out and death reports
- Data corruption reports
- Historically designed for FAT FS
- Optimized for FAT

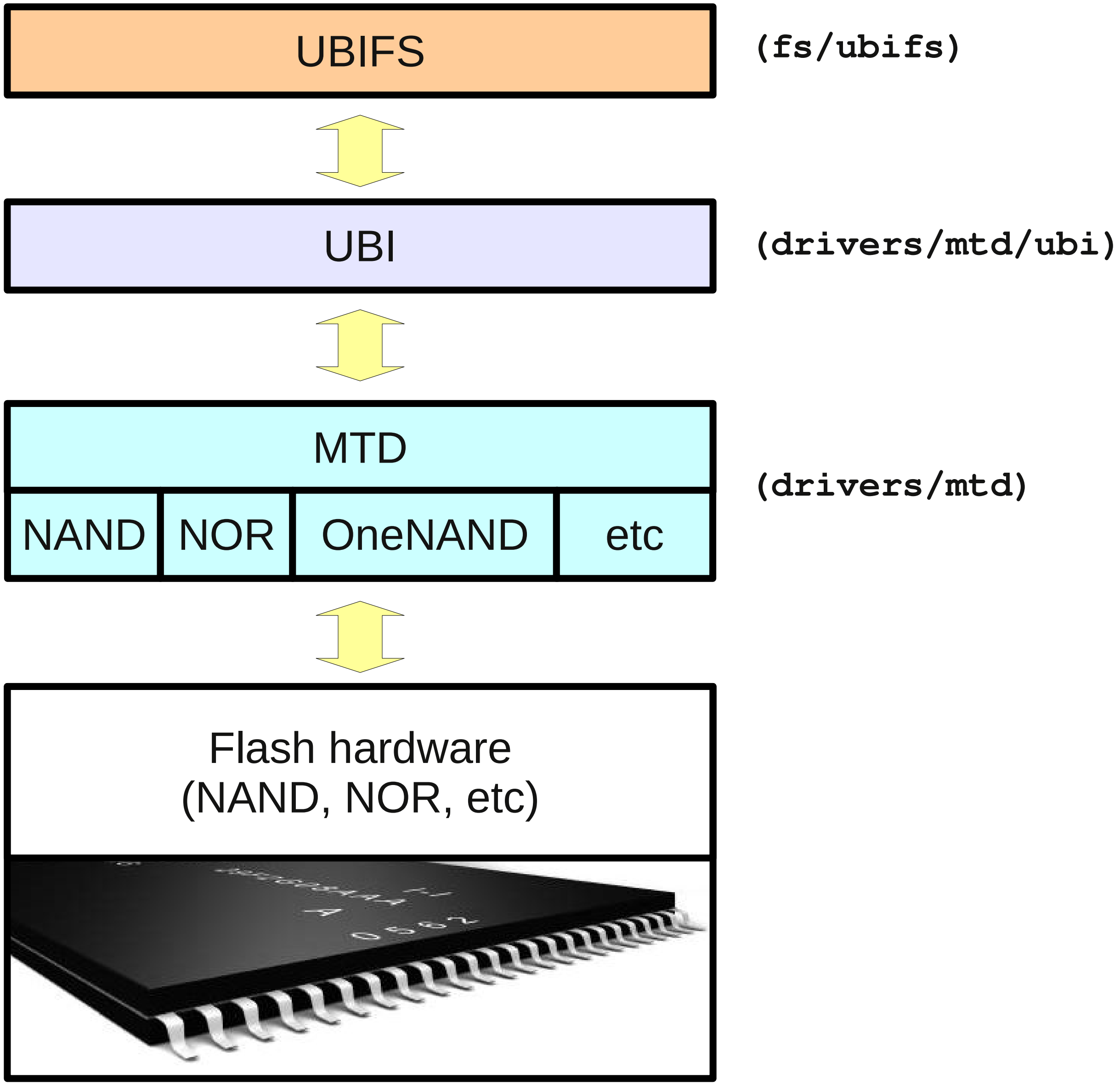
UBIFS goals

- Fix JFFS2 scalability issues
  - Faster mount
  - Fast opening of big files
  - Faster write speed
- But preserve cool JFFS2 features
  - On-the-flight compression
  - Tolerance to power cuts
  - Recoverability

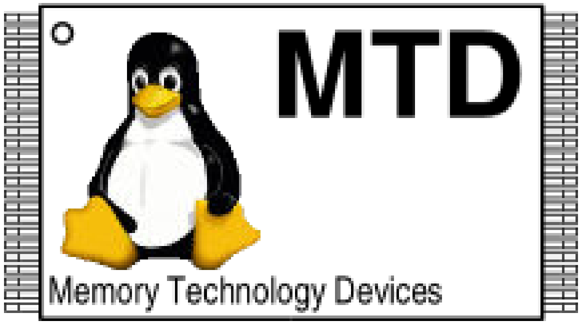
Plan

- Introduction (Artem)
- **MTD and UBI (Artem)**
- UBIFS (Adrian)

UBI/UBIFS stack



MTD



- MTD stands for “Memory Technology Devices”
- Provides an abstraction of MTD device
- Hides many aspects specific to particular flash
- Provides uniform API to access various types of flashes
- E.g., MTD supports NAND, NOR, ECC-ed NOR, DataFlash, OneNAND, etc
- Provides partitioning capabilities

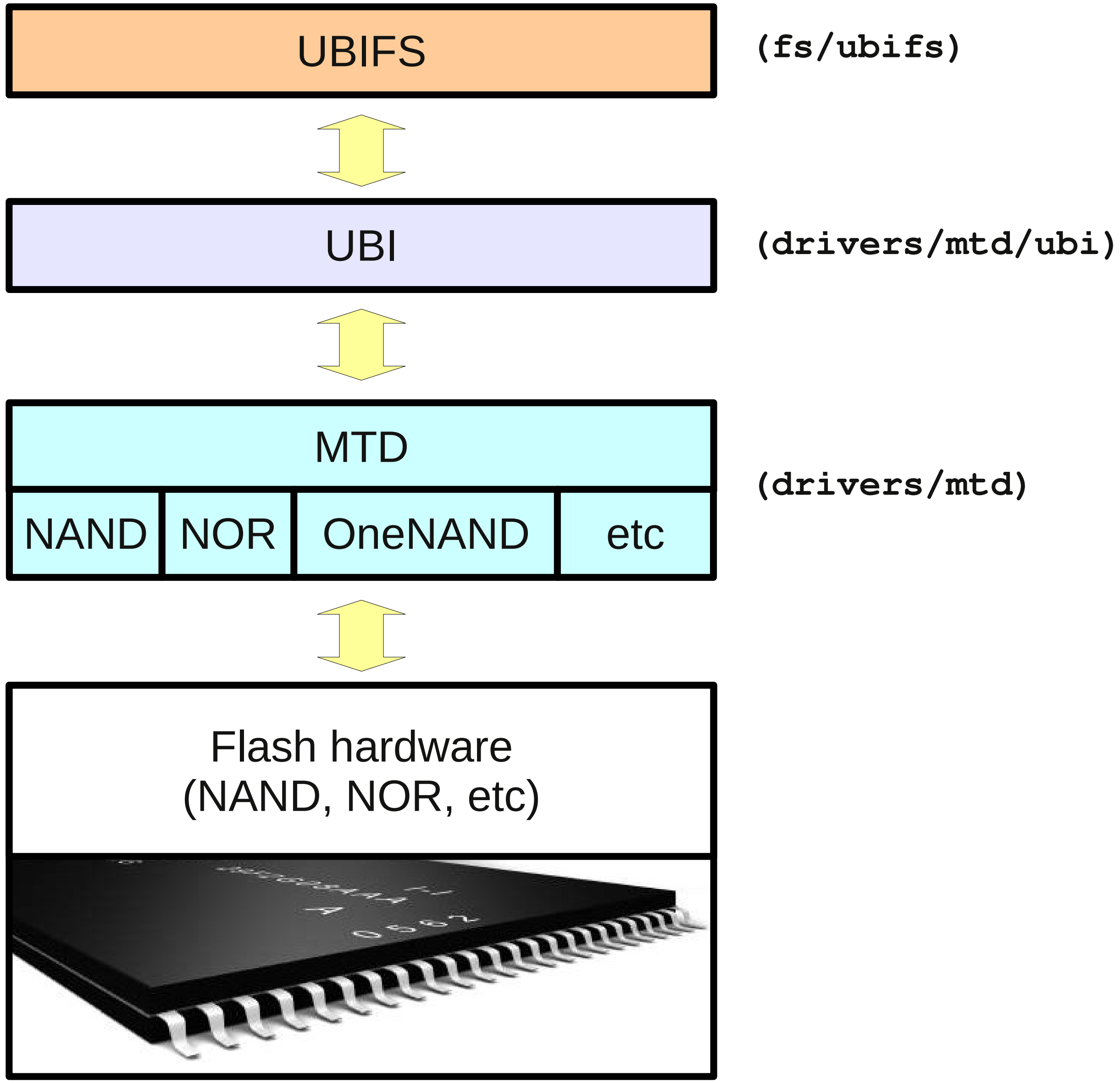
MTD API

- In-kernel API (`struct mdt_device`) and user-space API (`/dev/mtd0`)
  - Information (device size, min. I/O unit size, etc)
  - Read from and write to eraseblocks
  - Erase an eraseblock
  - Mark an eraseblock as bad
  - Check if an eraseblock is bad
- Does not hide bad eraseblocks
- Does not do any wear-leveling

UBI

- Stands for "Unsorted Block Images"
- Provides an abstraction of "UBI volume"
- Has kernel API (`include/mtd/ubi-user.h`) and user-space API (`/dev/ubi0`)
- Provides wear-leveling
- Hides bad eraseblocks
- Allows run-time volume creation, deletion, and re-size
- Is somewhat similar to LVM, but for MTD devices

UBI/UBIFS stack

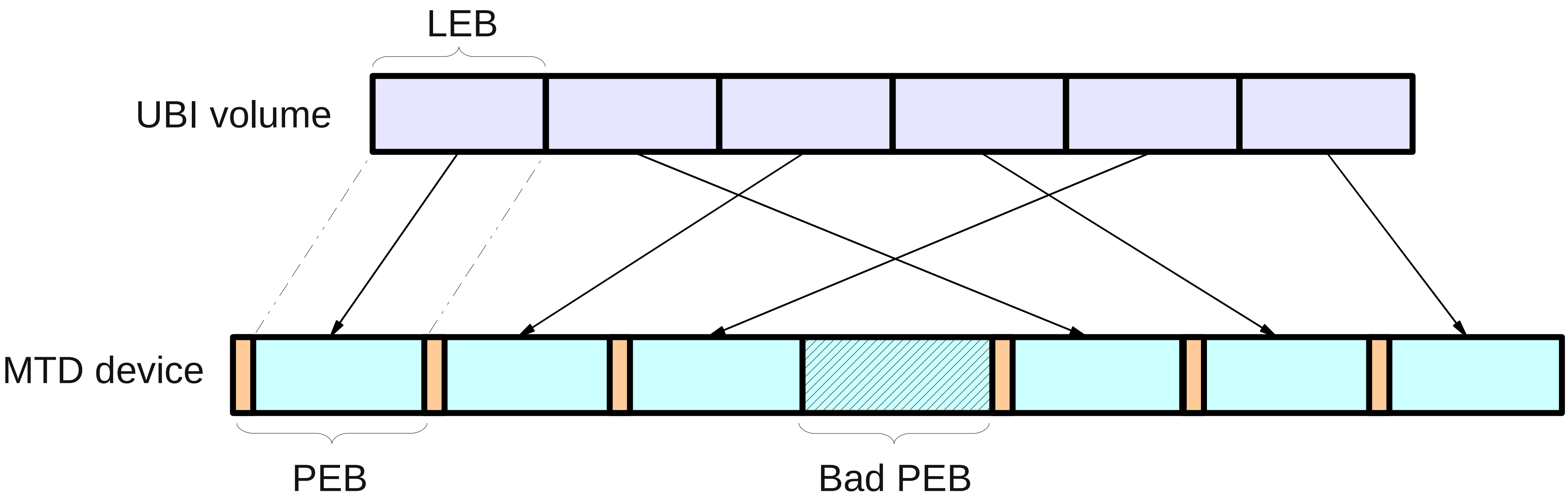


UBI volume vs. MTD device

MTD device	UBI volume
Consists of physical eraseblocks (PEB), typically 128KiB	Consists of logical eraseblocks (LEB), slightly smaller than PEB (e.g., 126KiB)
Has 3 main operations: 1. read from PEB 2. write to PEB 3. erase PEB	Has 3 main operations: 1. read from LEB 2. write to LEB 3. erase LEB
May have bad PEBs	Does not have bad LEBs - UBI transparently handles bad PEBs
PEBs wear out	LEBs do not wear out - UBI spreads the I/O load evenly across whole flash device (transparent wear-leveling)
MTD devices are static: cannot be created/deleted/re-sized run-time	UBI volumes are dynamic – can be created, deleted and re-sized run-time

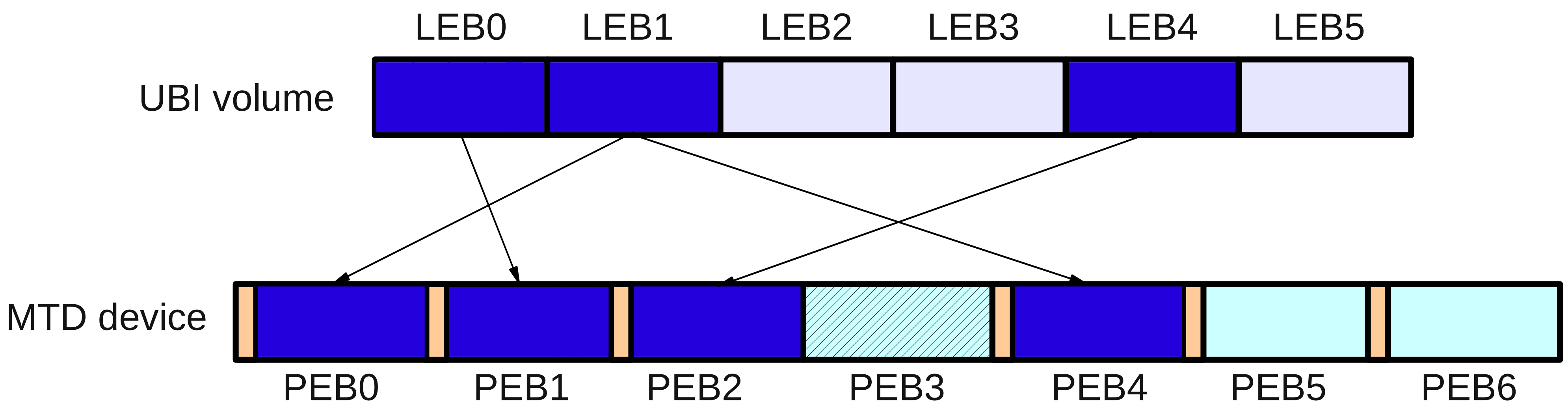
Main idea behind UBI

- Maps LEBs to PEBs
- Any LEB may be mapped to any PEB
- Eraseblock headers store mapping information and erase count



UBI operation example

1. Write data to LEB0
   a) Map LEB0 to PEB1
   b) Write the data
2. Write data to LEB1, LEB4
3. Erase LEB1
   a) Un-map LEB1 ... return
   b) Erase PEB4 in background
4. Write data to LEB1

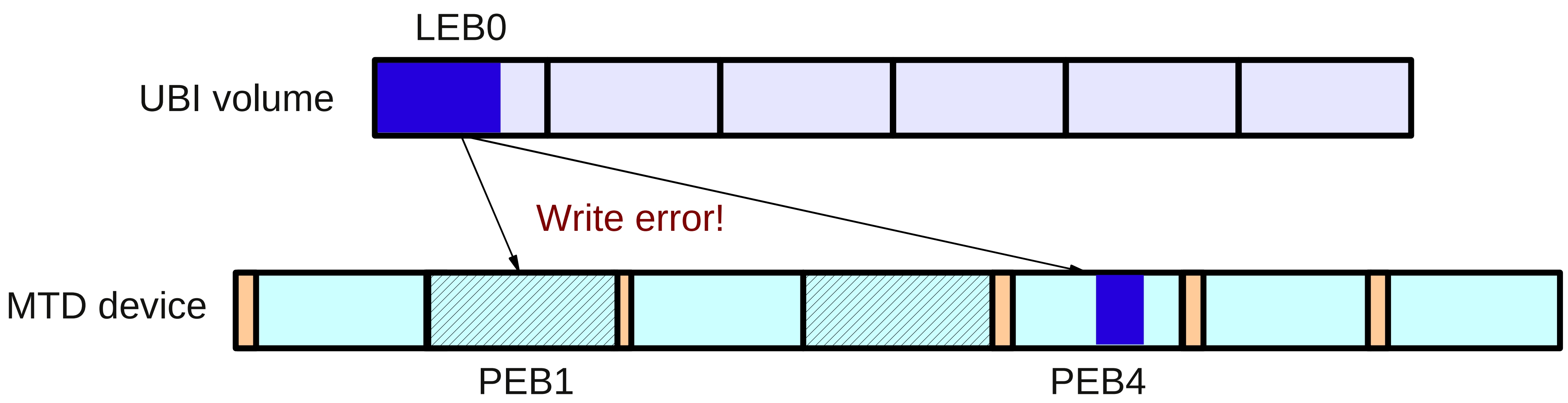


UBI bad eraseblock handling

- 1% of PEBs are reserved for bad eraseblock handling
- If a PEB becomes bad, corresponding LEB is re-mapped to a good PEB
- I/O errors are handled transparently

Write error handling example

1. User writes data to LEB0 ...
   a) Select a good PEB to recover the data to ... PEB4
   b) Recover the data by copying it to PEB4
   c) Re-map LEB0 to PEB4
   d) Write new data again
   e) Recovery is done! Return from UBI
   f) Erase, torture and check PEB1 in background ... and mark it as bad



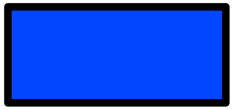
Other

- Handle bit-flips by moving data to a different PEB
- Configurable wear-leveling threshold
- Volume update operation
- Volume rename operation
- Suitable for MLC NAND
- Performs operations in background
- Works on NAND, NOR and other flash types
- Tolerant to power cuts
- Simple and robust design

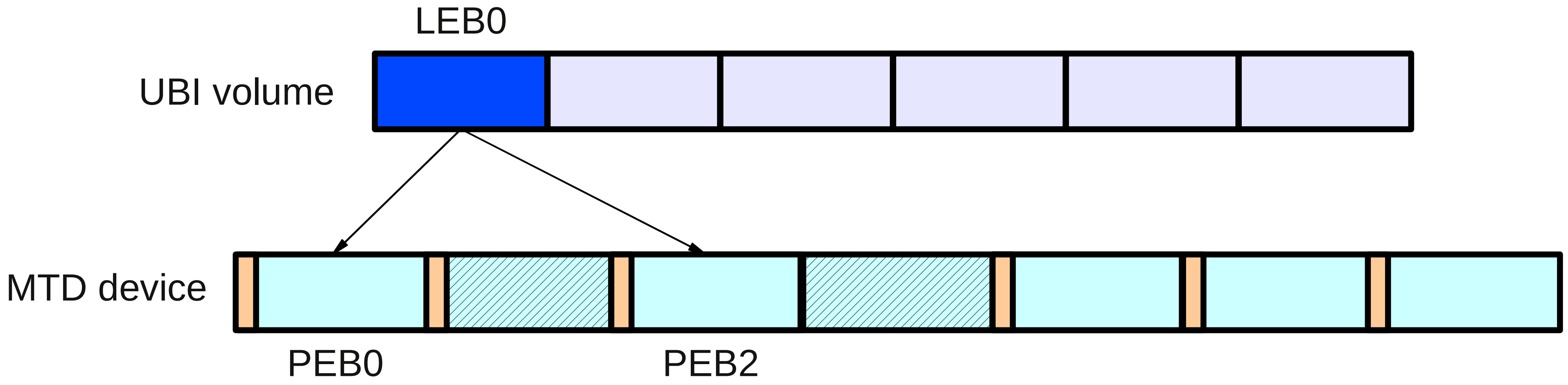
Atomic LEB change

- Very important for UBIFS

Suppose LEB0 has to be atomically updated



a) Select a PEB ... PEB0
b) Write new data to PEB0
c) Re-map LEB0 to PEB0
d) Done! Return from UBI
e) Erase PEB2 in background



UBI Scalability issue

- Unfortunately, UBI scales linearly
- UBI reads all eraseblock headers on initialization
- Initialization time grows with growing flash size
- But it scales considerably better than JFFS2
- May be improved
- UBI2 may be created, UBIFS would not change
- Suitable for 1-16GiB flashes, depending on I/O speed and requirements

Plan

- Introduction (Artem)
- MTD and UBI (Artem)
- UBIFS (Adrian)

UBIFS relies on UBI

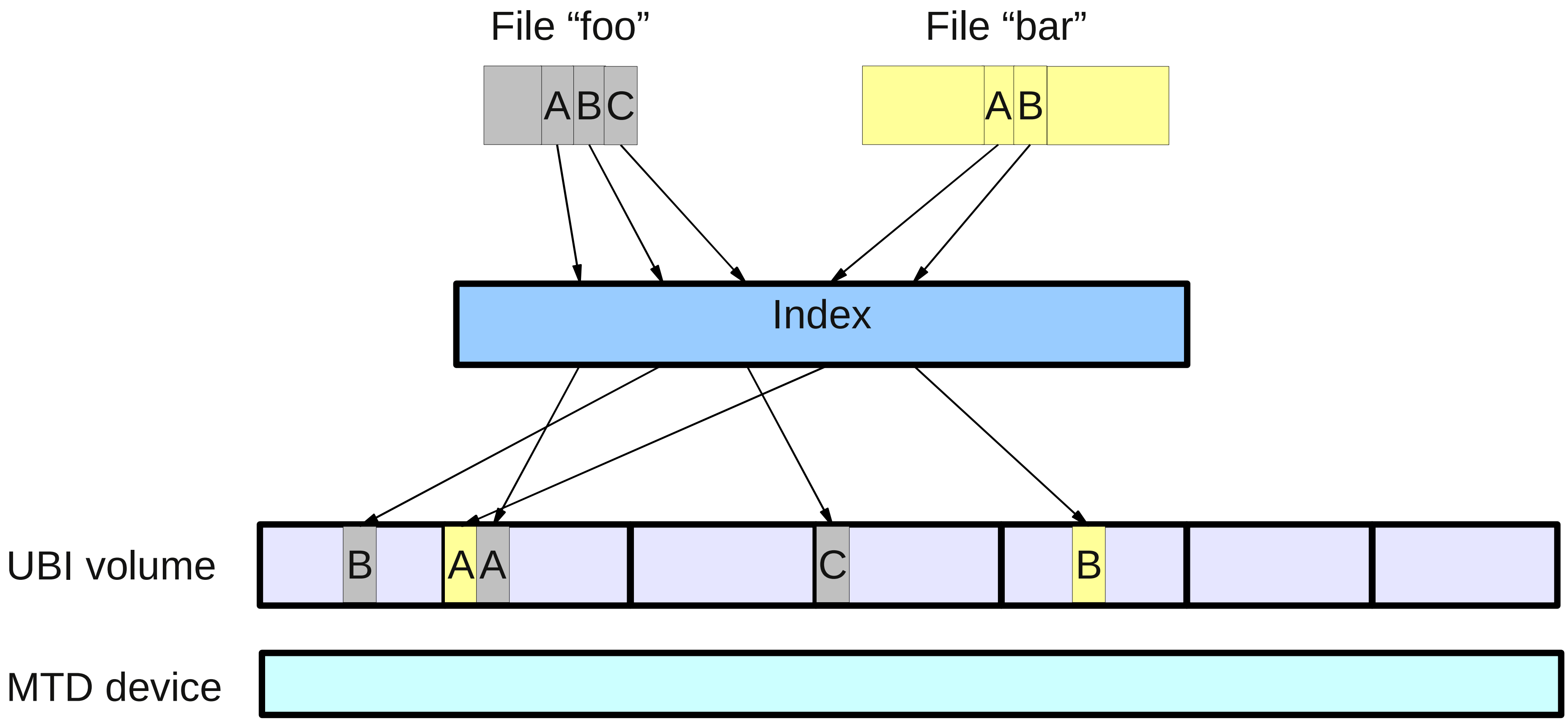
- UBIFS does not care about bad eraseblocks and relies on UBI
- UBIFS does not care about wear-leveling and relies on UBI
- UBIFS exploits the atomic LEB change feature

Requirements

- Good scalability
- High performance
- On-the-flight compression
- Power-cut tolerance
- High reliability
- Recoverability

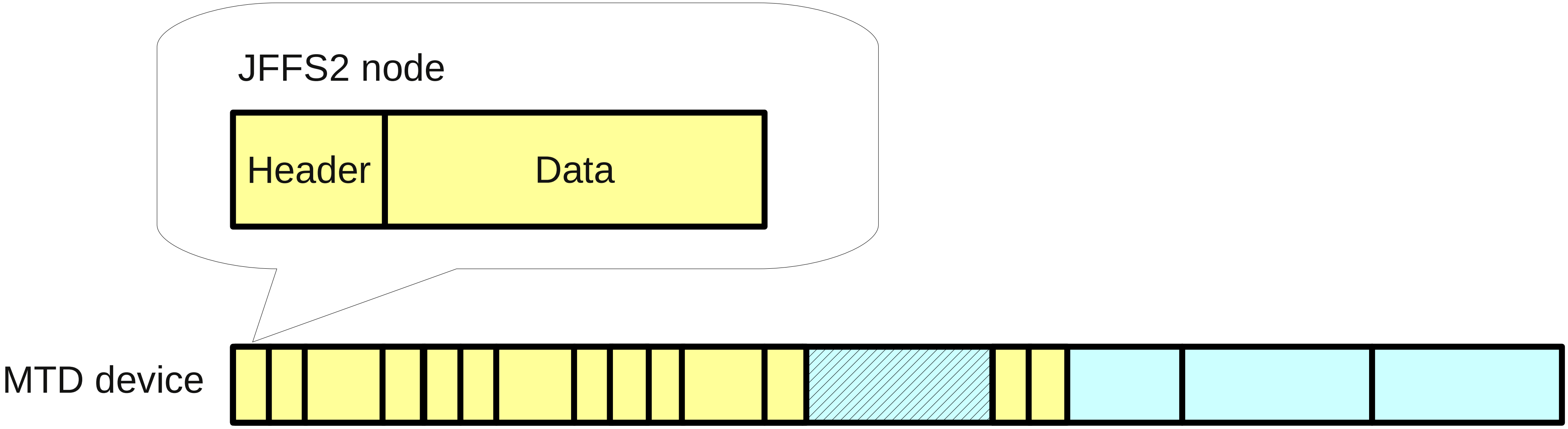
File System Index

- Index allows to look-up physical address for any piece of FS data



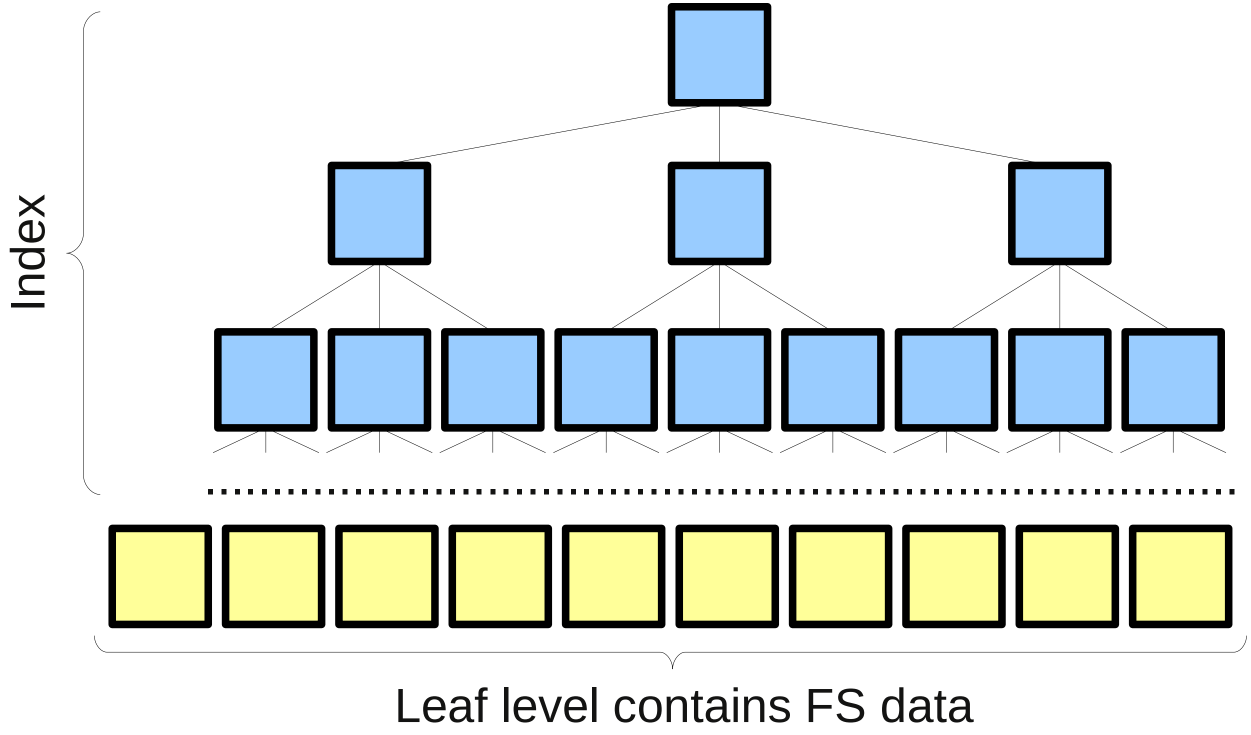
JFFS2 index

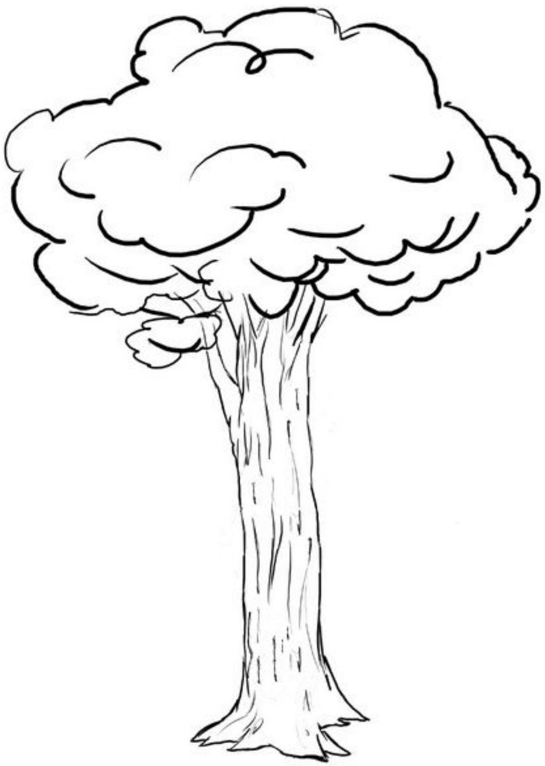
- JFFS2 does not store the index on flash
- On mount JFFS2 fully scans the flash media
- Node headers are read to build the index in RAM



UBIFS index

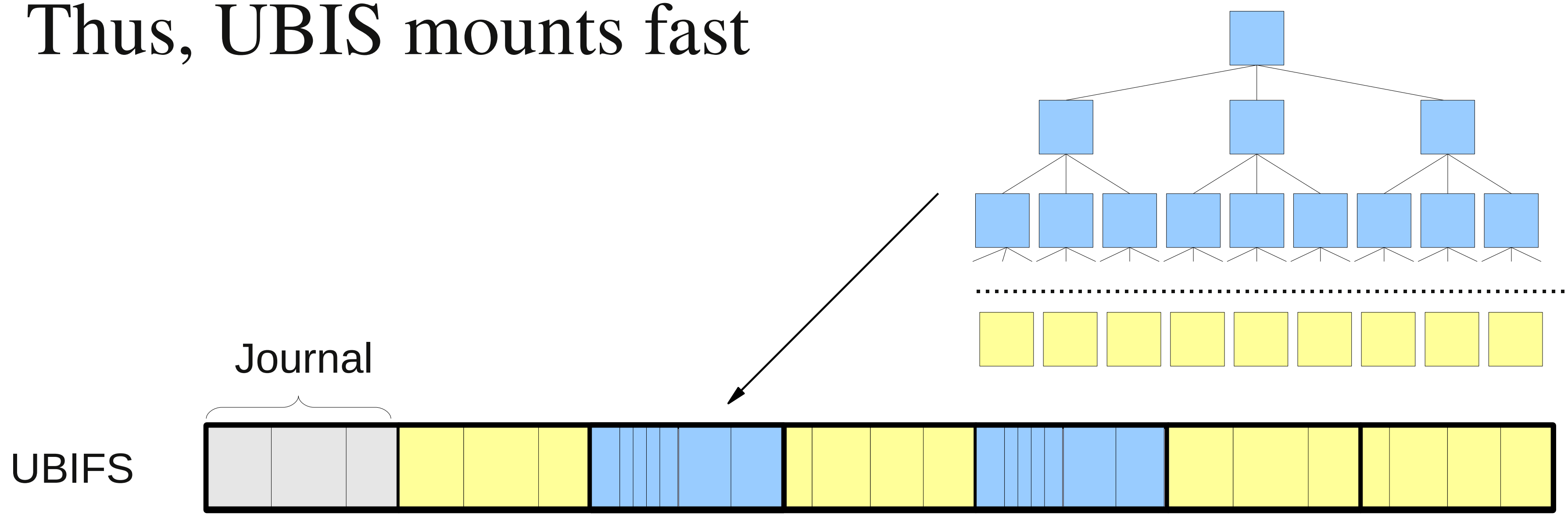
- UBIFS index is a B+ tree
- Leaf level contains data
- Tree fanout is configurable, default is 8





UBIFS Index

- UBIFS index is stored and maintained on flash
- Full flash media scanning is not needed
- Only the journal is scanned in case of power cut
- Journal is small, has fixed and configurable size
- Thus, UBIS mounts fast



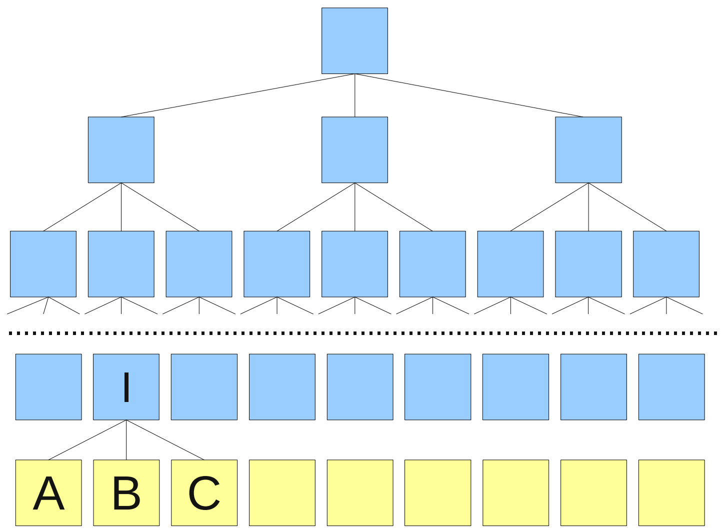
Out-of-place updates

- Flash technology and power-cut-tolerance require out-of-place updates

Change "foo" (overwrite A, B, C)

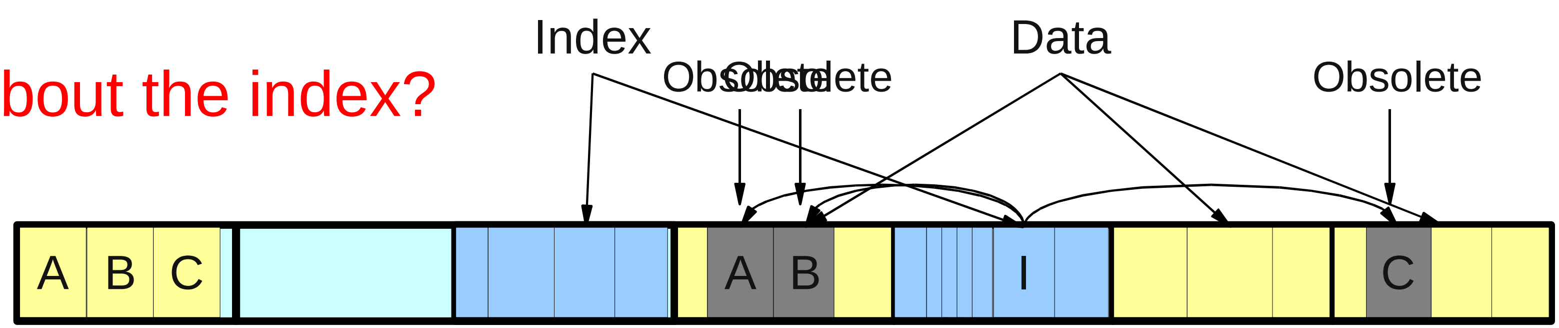
File "/foo": A | B | C

1. Write "A" to a free space
2. Old "A" becomes obsolete
3. Write "B" to free space
4. Old "B" becomes obsolete
5. Write "C" to free space
6. Old "C" becomes obsolete



But how about the index?

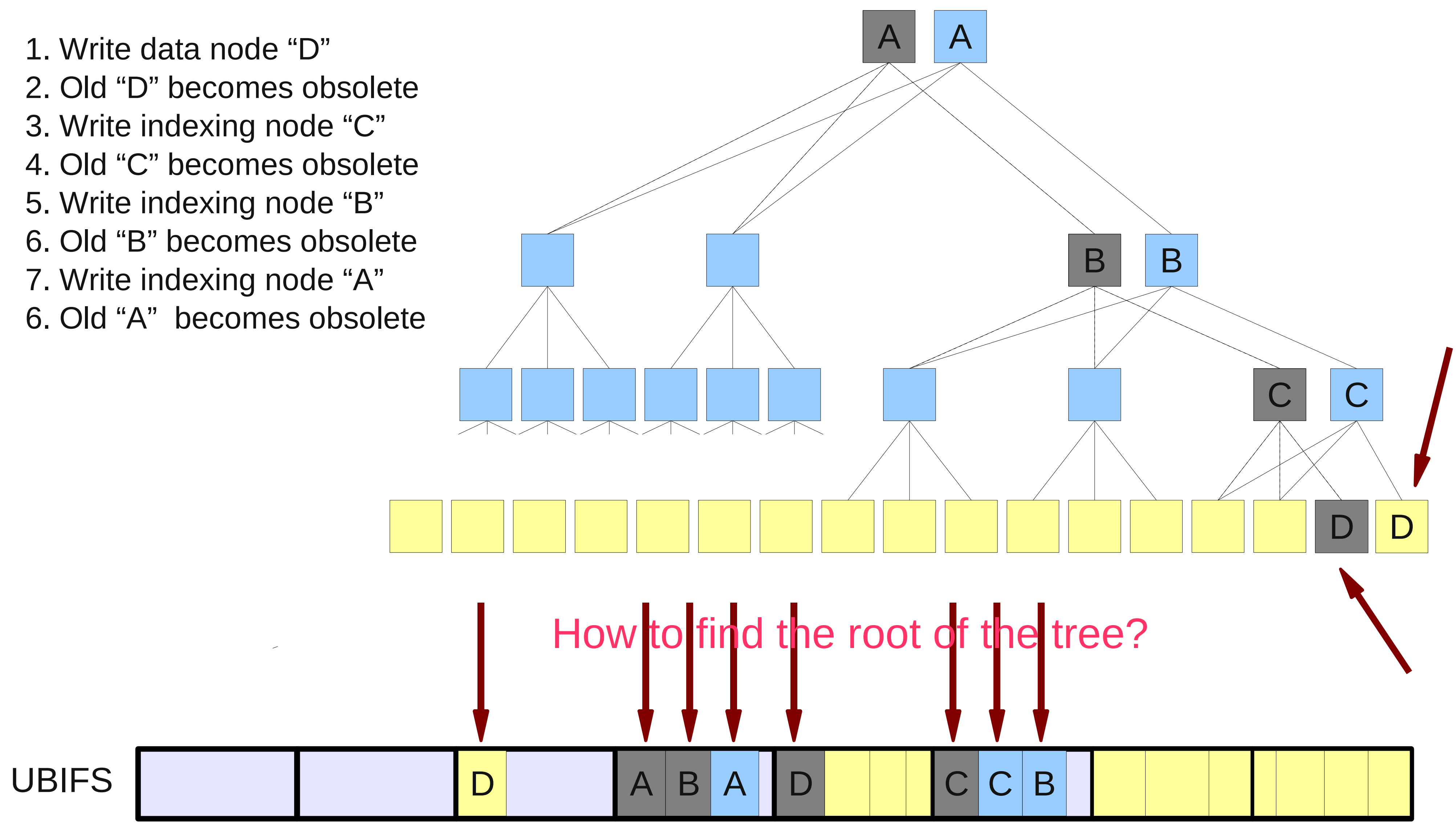
UBIFS



Wandering trees

Wandering trees

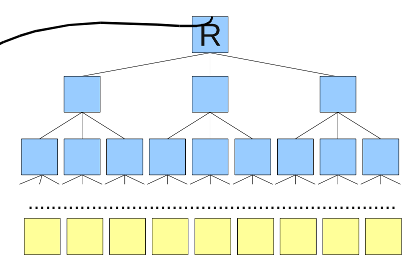
1. Write data node “D”
2. Old “D” becomes obsolete
3. Write indexing node “C”
4. Old “C” becomes obsolete
5. Write indexing node “B”
6. Old “B” becomes obsolete
7. Write indexing node “A”
6. Old “A” becomes obsolete

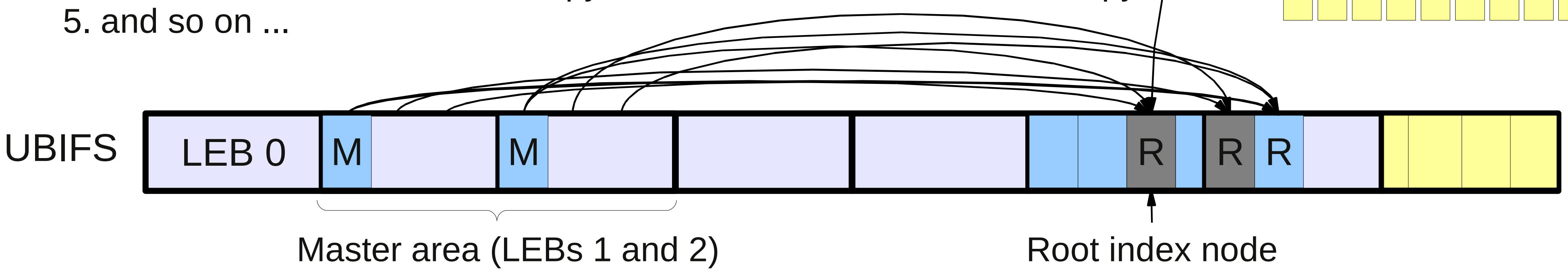


Master node

- Stored at the master area (LEBs 1 and 2)
- Points to the root index node
- 2 copies of master node exist for recoverability
- Master area may be quickly found on mount
- Valid master node is found by scanning master area

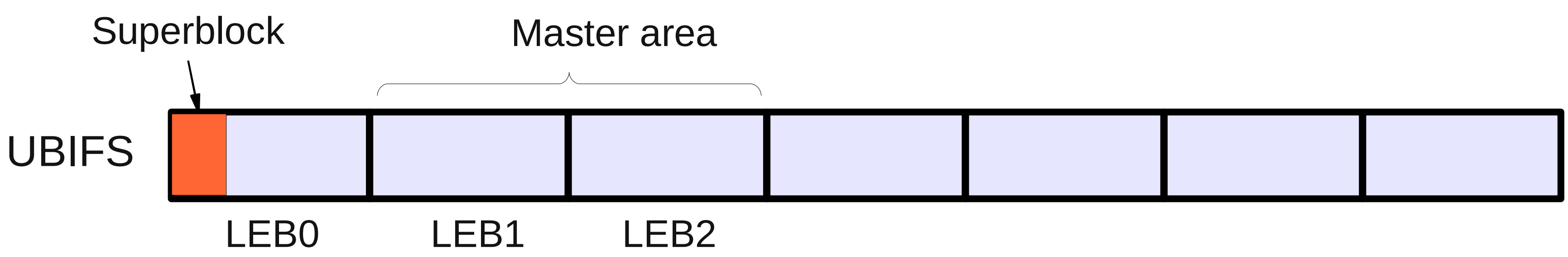
1. Suppose “R” is changed
2. Then “M” is updated
3. Old “M” becomes obsolete
4. The same is done to the 2nd copy
5. and so on ...
6. LEBs 1 and 2 become full
7. LEB 1 is erased
8. “M” is written
9. The same for the 2nd copy





Superblock

- Situated at LEB0
- Read-only for UBIFS
- May be changed by user-space tools
- Stores configuration information like indexing tree fanout, default compression type (zlib or LZO), etc
- Superblock is read on mount



Journal

- All FS changes go to the journal
- Indexing information is changed in RAM, but not on the flash
- Journal greatly increases FS write performance
- When mounting, journal is scanned and re-played
- Journal is roughly like a small JFFS2 inside UBIFS
- Journal size is configurable and is stored in superblock

TNC

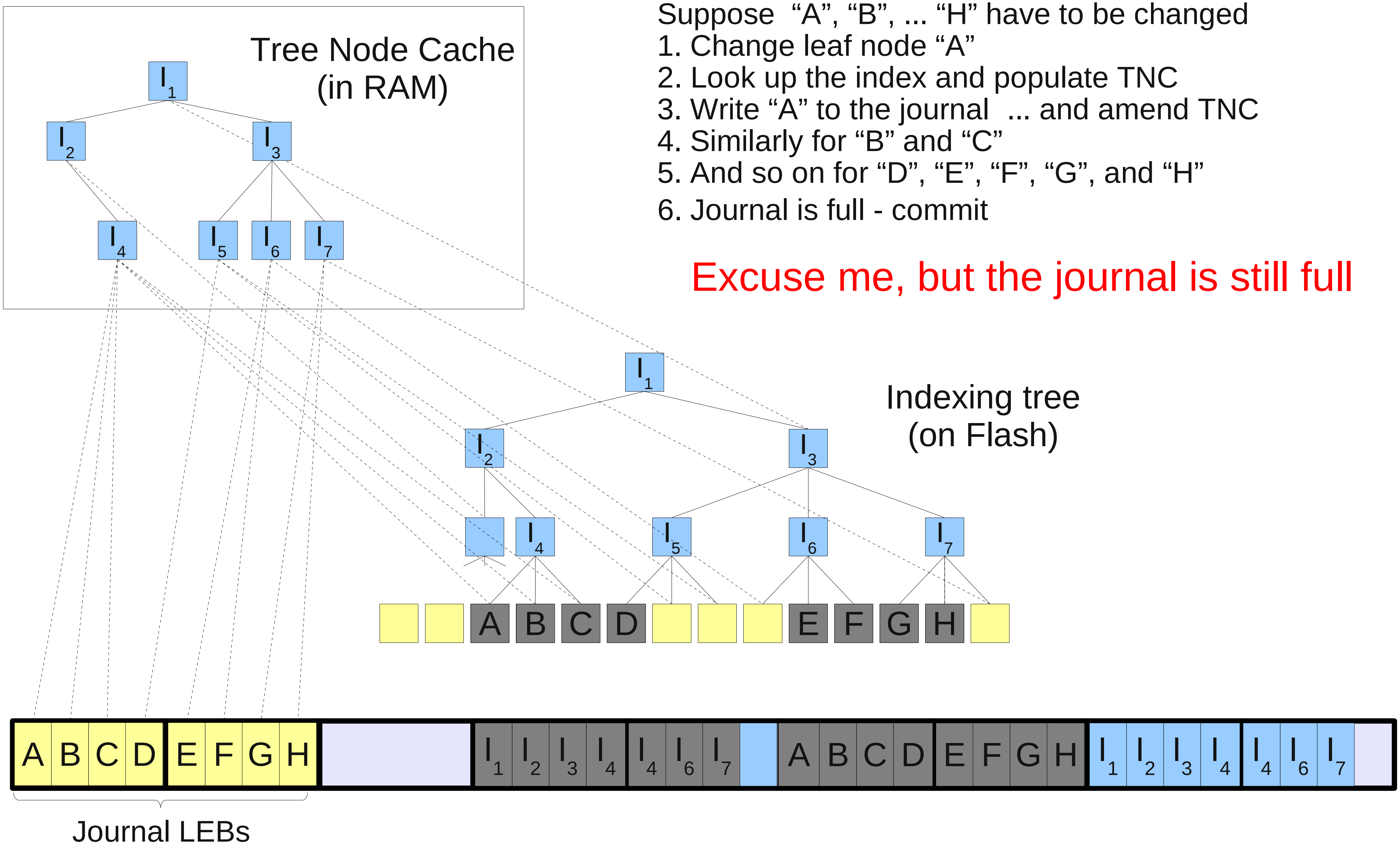
- Stands for Tree Node Cache
- Caches indexing nodes
- Is also a B^+-tree, but in RAM
- Speeds up indexing tree lookup
- May be shrunk in case of memory pressure

Journal and TNC

Suppose "A", "B", ... "H" have to be changed

1. Change leaf node "A"
2. Look up the index and populate TNC
3. Write "A" to the journal ... and amend TNC
4. Similarly for "B" and "C"
5. And so on for "D", "E", "F", "G", and "H"
6. Journal is full - commit

Excuse me, but the journal is still full



Journal is also wandering

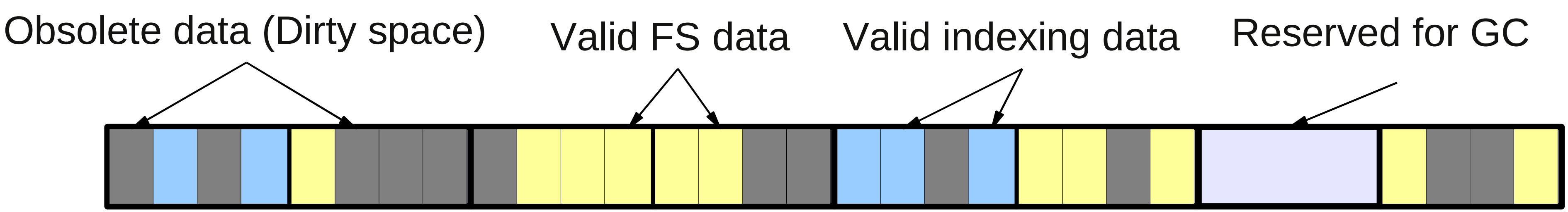
- After the commit we pick different LEBs for the journal
- Do not move data out of the journal
- Instead, we move the journal
- Journal changes the position all the time

More about the Journal

- Journal has multiple heads
  - This improves performance
- Journal does not have to be continuous
  - Journal LEBs may have random addresses
  - LEBs do not have to be empty to be used for journal
  - This makes journal very flexible and efficient

Garbage collection

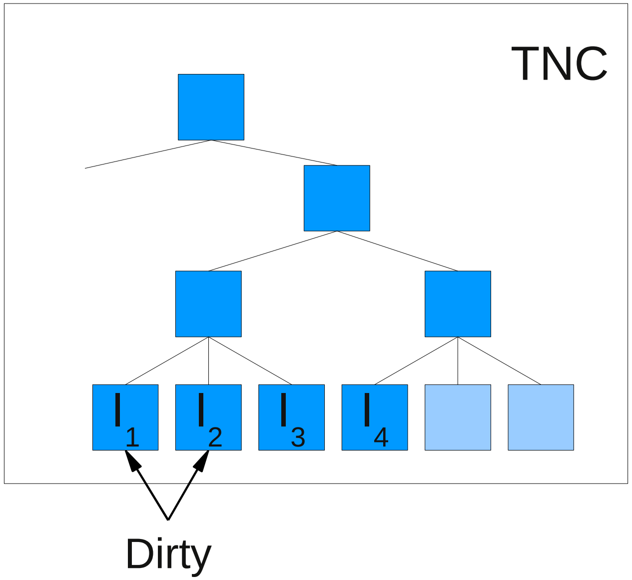
- At some point UBIFS runs out of free space
- Garbage Collector (GC) is responsible to turn dirty space to free space
- One empty LEB is always reserved for GC

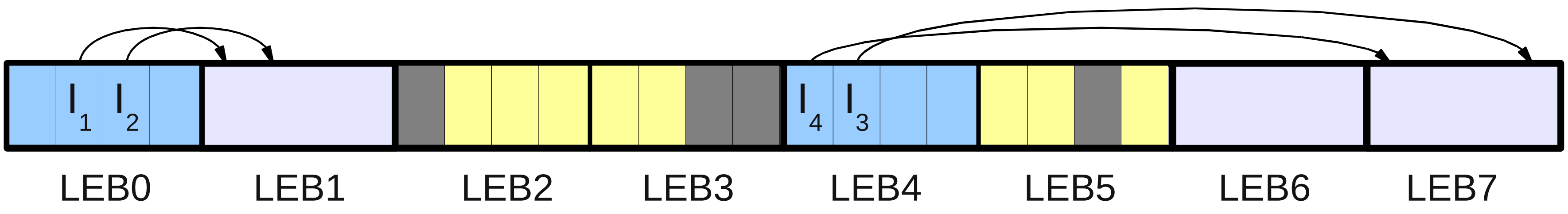


How GC works

- GC copies valid data to the journal (GC head)

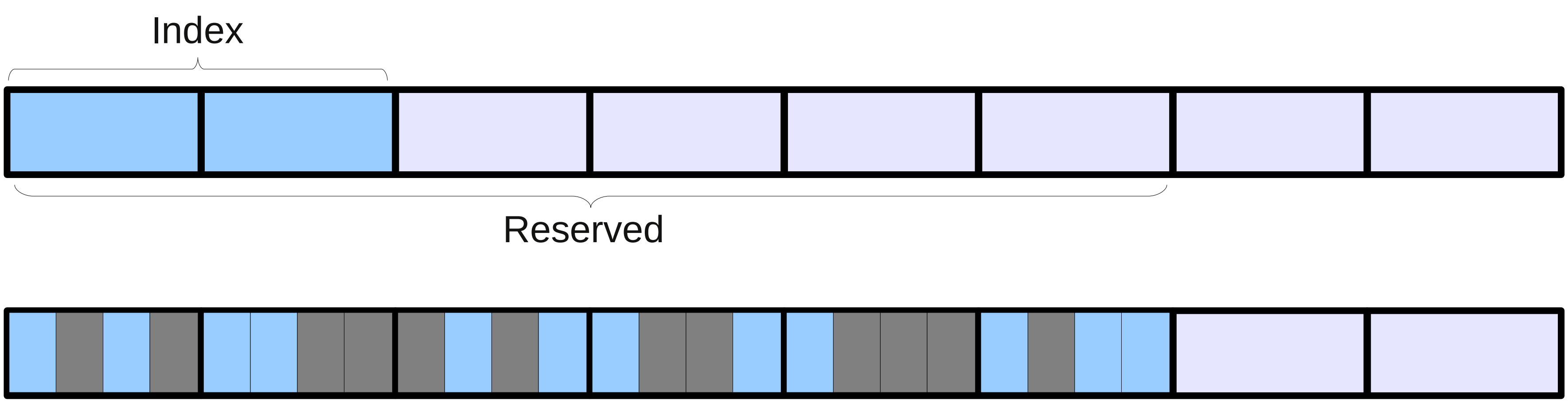
1. Pick a dirty LEB ... LEB1
2. Copy valid data to LEB6
3. LEB1 may now be erased
4. Pick another dirty LEB ... LEB7
5. Copy valid data to LEB 6
6. LEB 7 now may be erased
8. LEB 1 is reserved for GC, LEB7 is available
9. How about the index?
10. Indexing nodes are just marked as dirty in TNC
11. But what if there is no free space for commit?





Commit

- Commit operation is always guaranteed to succeed
- For the index UBIFS reserves 3x as much space
- In-the-gaps commit method is used
- Atomic LEB change UBI feature is used



LEB properties

- UBIFS stores per LEB-information of flash
  - LEB type (indexing or data)
  - Amount of free and dirty space
- Overall space accounting information is maintained on the media
  - Total amount of free space
  - Total amount of dirty space
  - Etc
- Used e.g., when
  - A free LEB should be found for new data
  - A dirty LEB should be found for GC

LPT

- Stands for LEB Properties Tree
- Is a B^+-tree
- Has fixed size
- Is much smaller than the main indexing tree
- Managed similarly to the main indexing tree

Requirements

- Good scalability
  - Data structures are trees
  - Only journal has to be replayed
- High performance
  - Write-back
  - Background commit
  - Read/write is allowed during commit
  - Multi-head journal minimizes amount of GC
  - TNC makes look-ups fast
  - LPT caches make LEB searches fast

Requirements

- On-the-flight compression
- Power-cut tolerance
  - All updates are out-of-place
  - Was extensively tested
- High reliability
  - All data is protected by CRC32 checksum
  - Checksum may be disabled for data
- Recoverability
  - All nodes have a header which fully describes the node
  - The index may be fully re-constructed from the headers