Getting the most out of the ‘btrfs’ filesystem

TUT91782

Thorsten Kukuk, Senior Architect SLES & Common Code Base
<kukuk@suse.com>
Jeff Mahoney, Team Lead, Kernel File Systems, SUSE Labs
<jeffm@suse.com>

Agenda

- Overview
- Copy on Write
- Disk Uage
- Chunk Allocator
- Useful Commands
- Compression
- Send / Receive
- Deduplication

Btrfs – What is it?

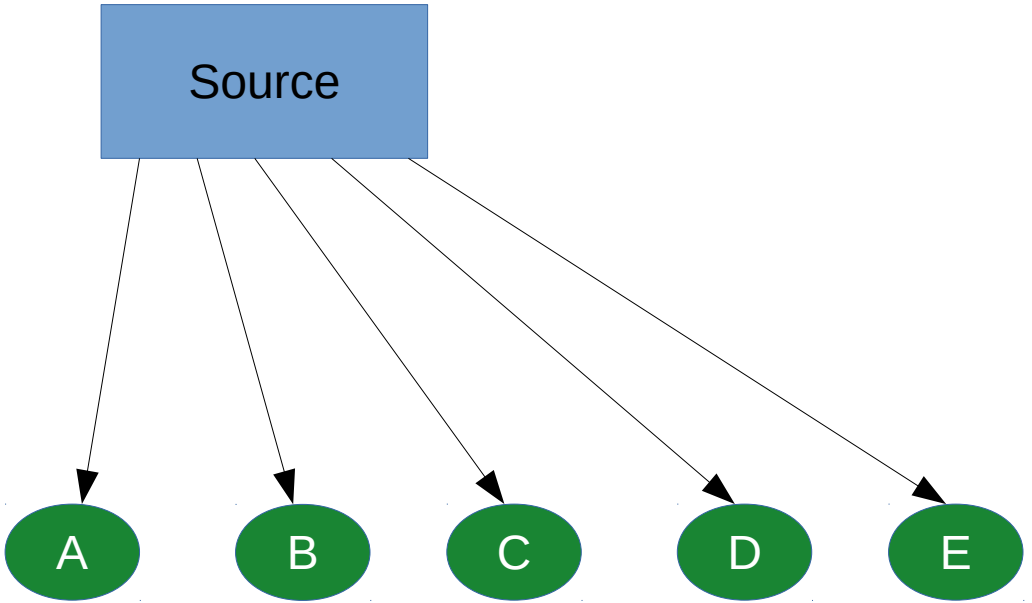
- **Copy on Write (CoW) general purpose file system**
- **Different Trees for**
  - Data
  - Metadata
- **CRCs maintained for all data and metadata**
- **Subvolumes**
- **Snapshots**

Copy on Write

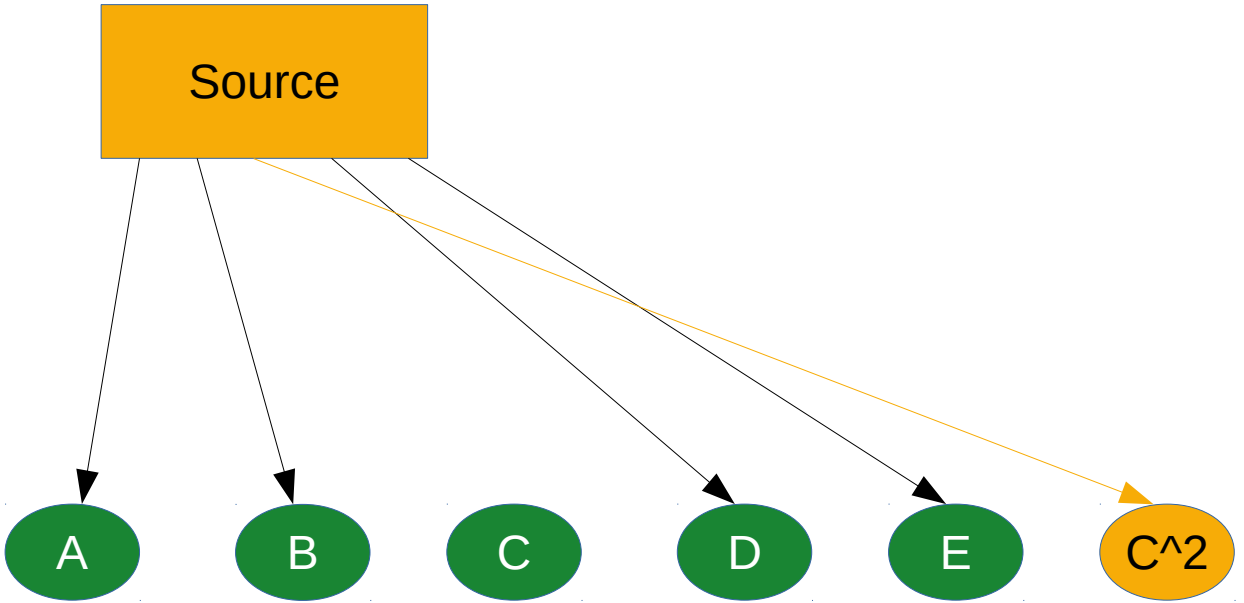
Copy on Write

- **If data will be modified:**
  - A copy of the original data is made
  - The new copy of the data is modified
  - References are changed to the new data (metadata)
  - If no references to the original data exist, the data is freed in the background
- **CoW operation is used on all writes (including metadata)**
- **Allows lazy copies**
- **cp –reflink=always**
- **To disable: mount with nodatacow, or use “**`chattr +C`**”**

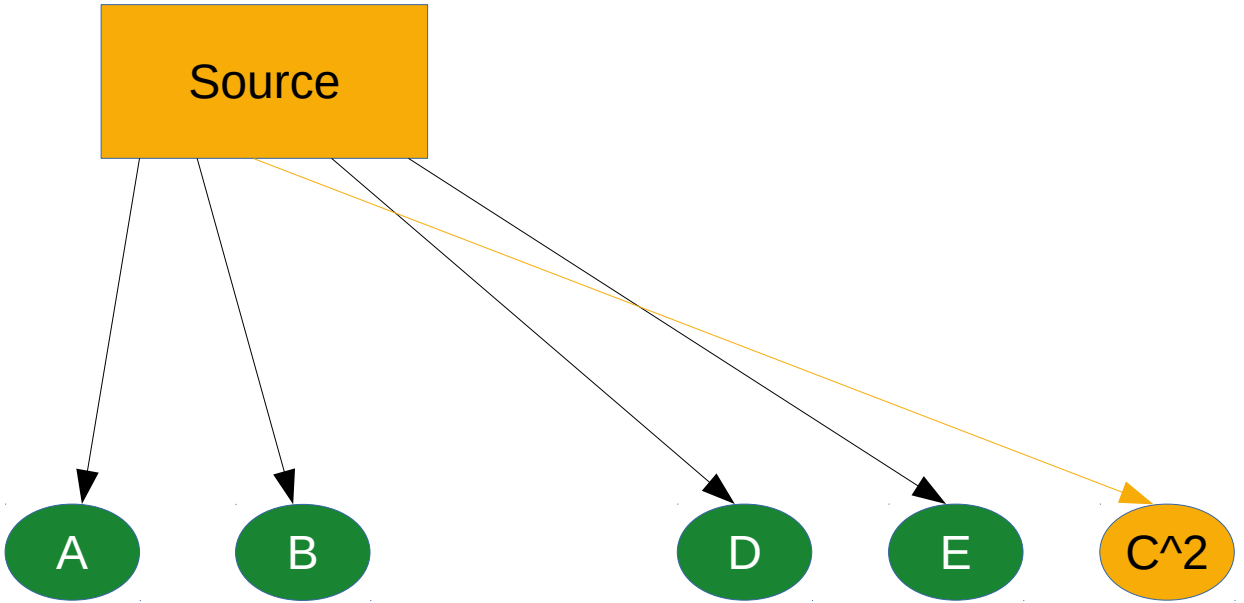
Btrfs / Copy-on-Write (1/3)



Btrfs / Copy-on-Write (2/3)



Btrfs / Copy-on-Write (3/3)

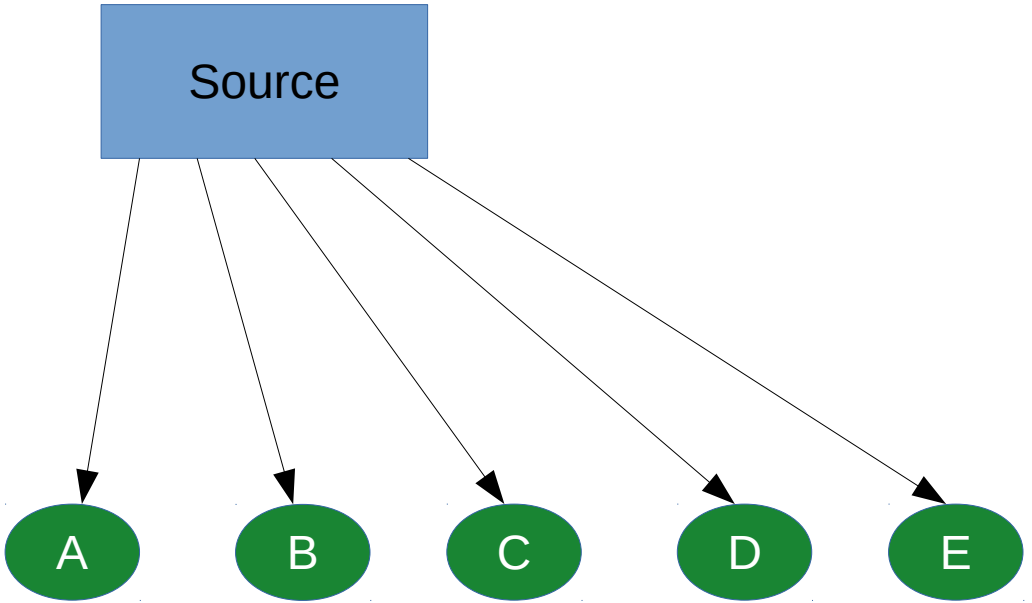


Snapshots

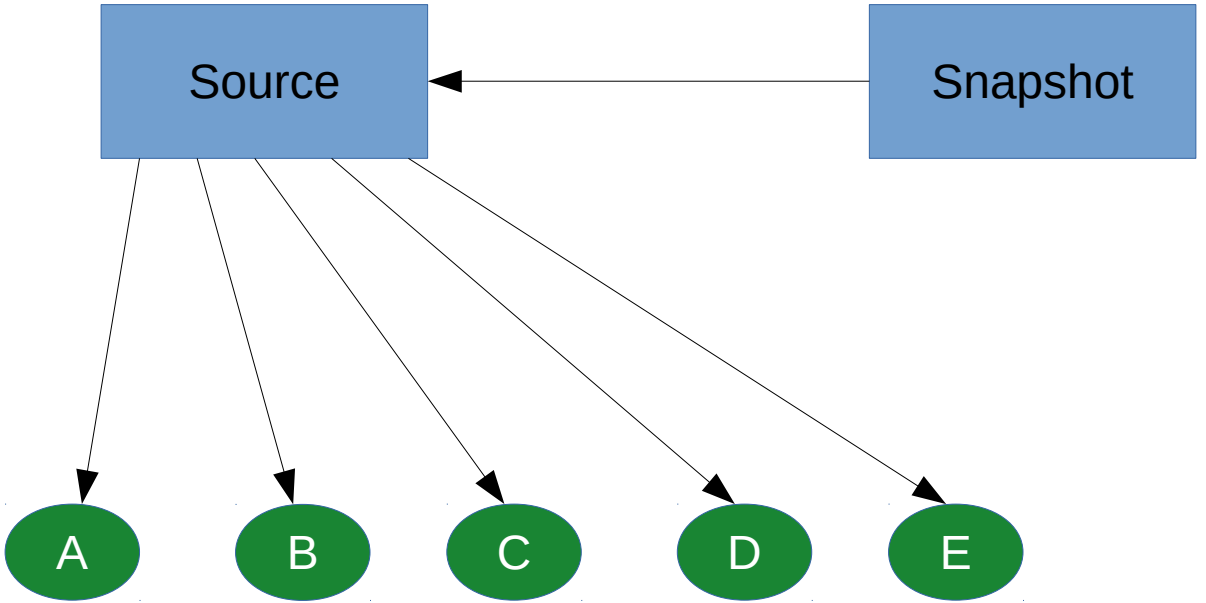
Snapshots

- **Simply a subvolume**
- **Shares data (and metadata) with some other subvolume**
- **Snapshots of snapshots are possible**
  - In the end these are all only subvolumes
- **Files in a snapshot may still be accessible to users, even if the permissions on the original files has changed!**
- btrfs subvolume list /

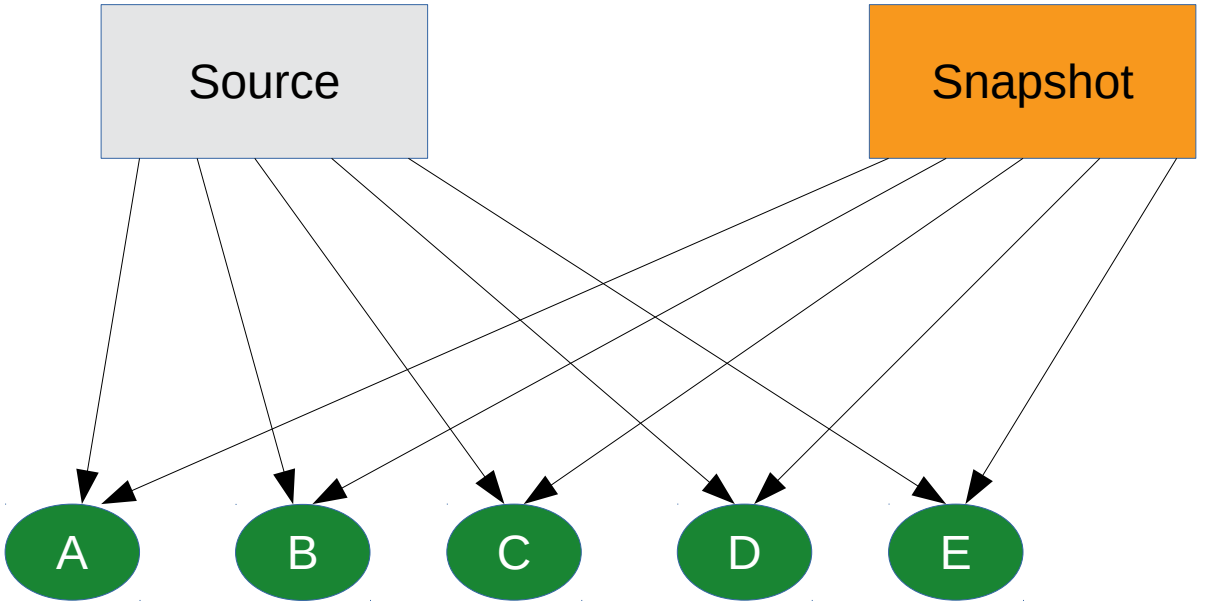
Btrfs / Snapshots (1/4)



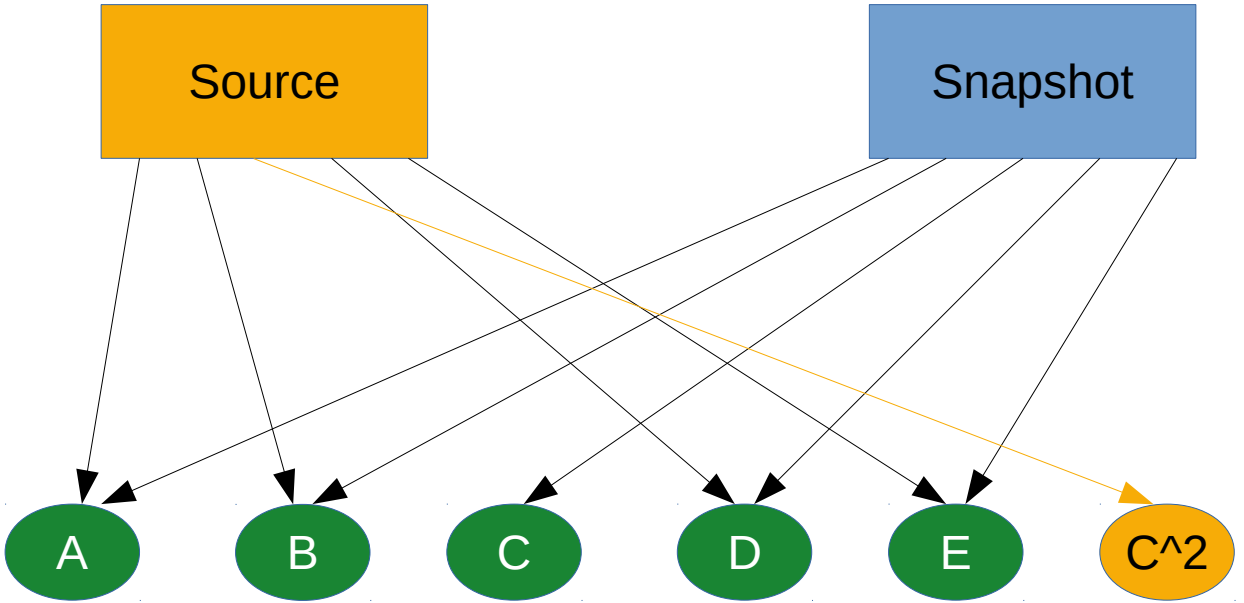
Btrfs / Snapshots (2/4)



Btrfs / Snapshots (3/4)



Btrfs / Snapshots (4/4)



Disk Usage

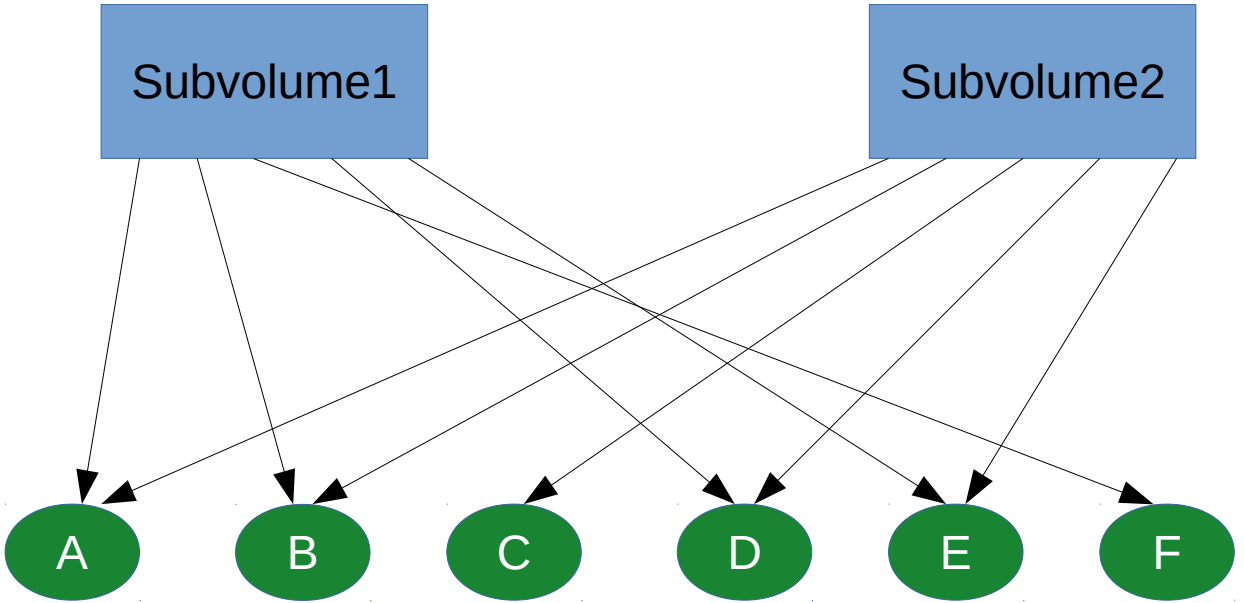
Btrfs Snapshots and Disk Usage

- **How much disk space does a snapshot need?**

  The answer nobody likes: It depends!

- **Initial snapshot: few Bytes for Metadata**
- **Growing over time when original data changes**
- **At the end: same amount as original data**
- **Worst case: Lot of snapshots and no common blocks between them.**

Btrfs - Disk Usage



Btrfs – Disk Usage

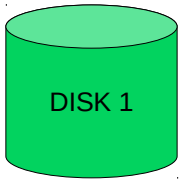
- **du: 2xA + 2xB + C + 2xD + 2xE + F => Could be bigger as your disk**
- **df: A + B + C + D + E + F => Allocated space missing**
- **Reality:** ‘btrfs filesystem usage /’
- **‘rm -rf subvolume1’**
  - Only sizeof (F) is free’d
- **‘rm -rf subvolume2’**
  - Only sizeof (C) is free’d

Chunk Allocation

Chunk Tree

- Chunks
  - Unit of logical volume space used by the extent tree
  - Typically 1 GiB in size, one per device required by the allocation policy
    - SINGLE – One stripe from one device, logically 1 GiB
    - DUP – Two stripes from one device, logically 1 GiB
    - RAID0 – Two stripes from two devices, logically 2 GiB
    - RAID1 – Two stripes from two devices, logically 1 GiB
- Can be used for either DATA or METADATA
  - Not mixed except in very small file systems
  - METADATA changes much more frequently than the DATA it describes
  - CoW means fragmentation everywhere – keep it separate!
- Can be balanced to relocate extents in order to release chunks for reuse
- No direct mapping from file block number to chunk-stripe block number from userspace using generic tools

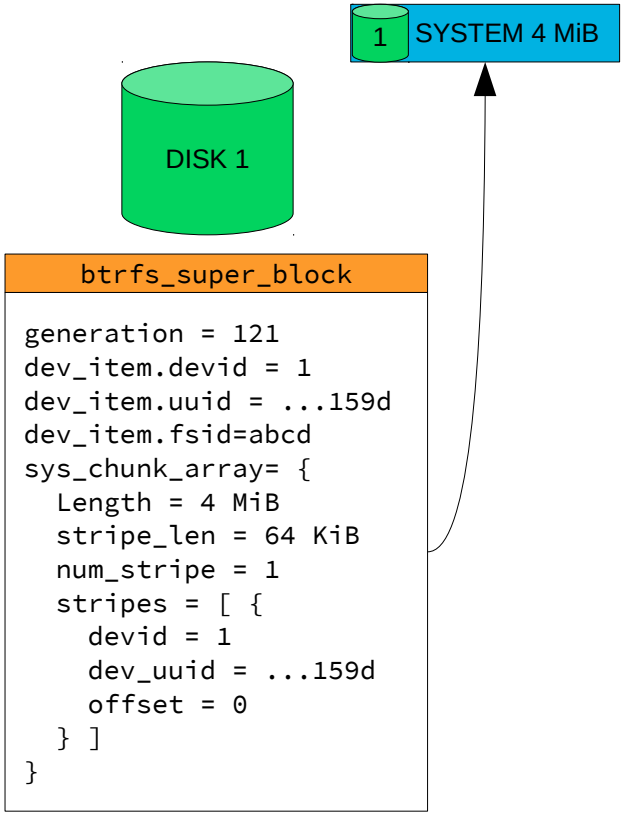
Chunk Allocation (SINGLE)



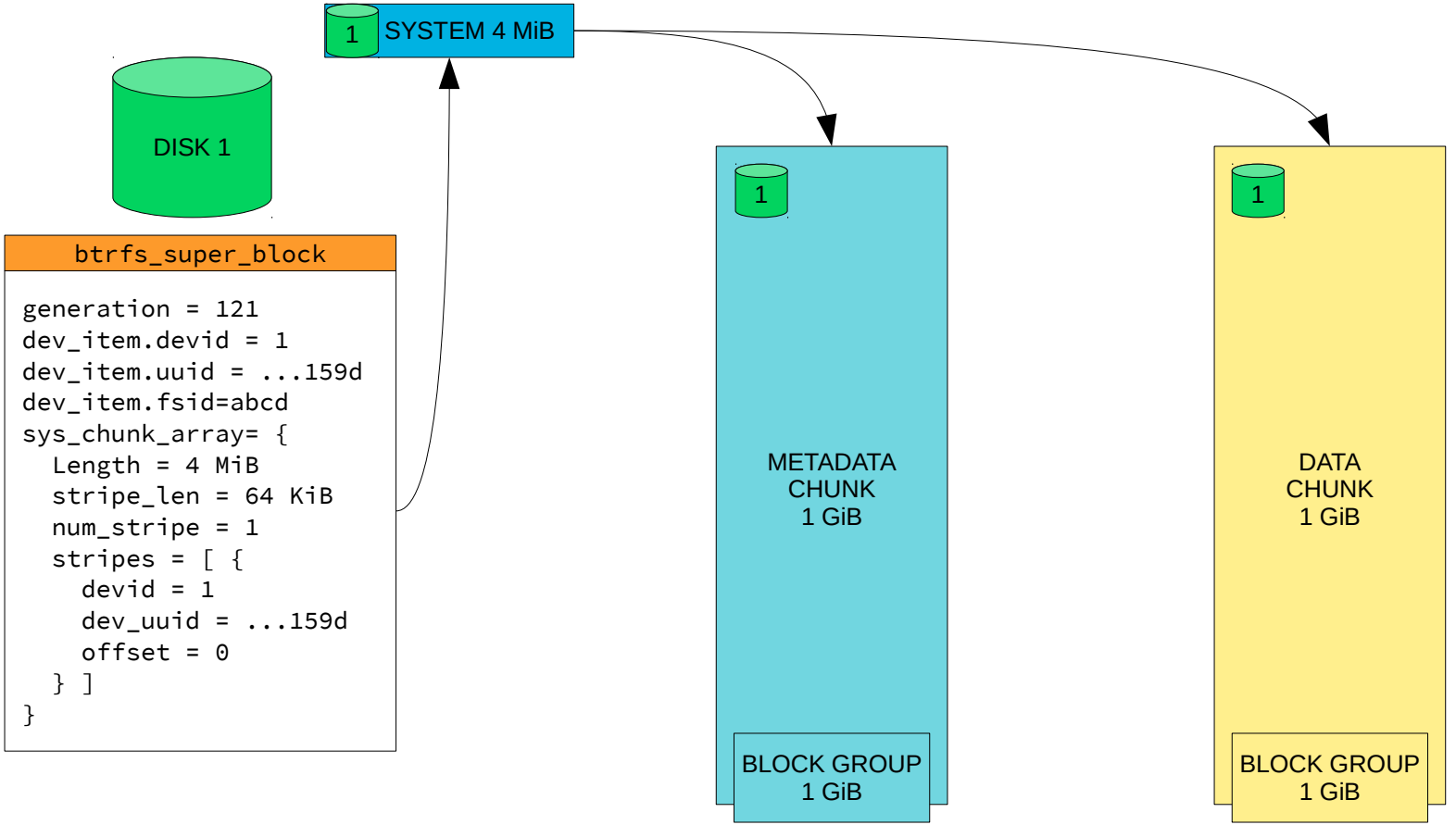
btrfs_super_block

```
generation = 121
dev_item.devid = 1
dev_item.uuid = ...159d
dev_item.fsid=abcd
sys_chunk_array= {
  Length = 4 MiB
  stripe_len = 64 KiB
  num_stripe = 1
  stripes = [ {
    devid = 1
    dev_uuid = ...159d
    offset = 0
  } ]
}
```

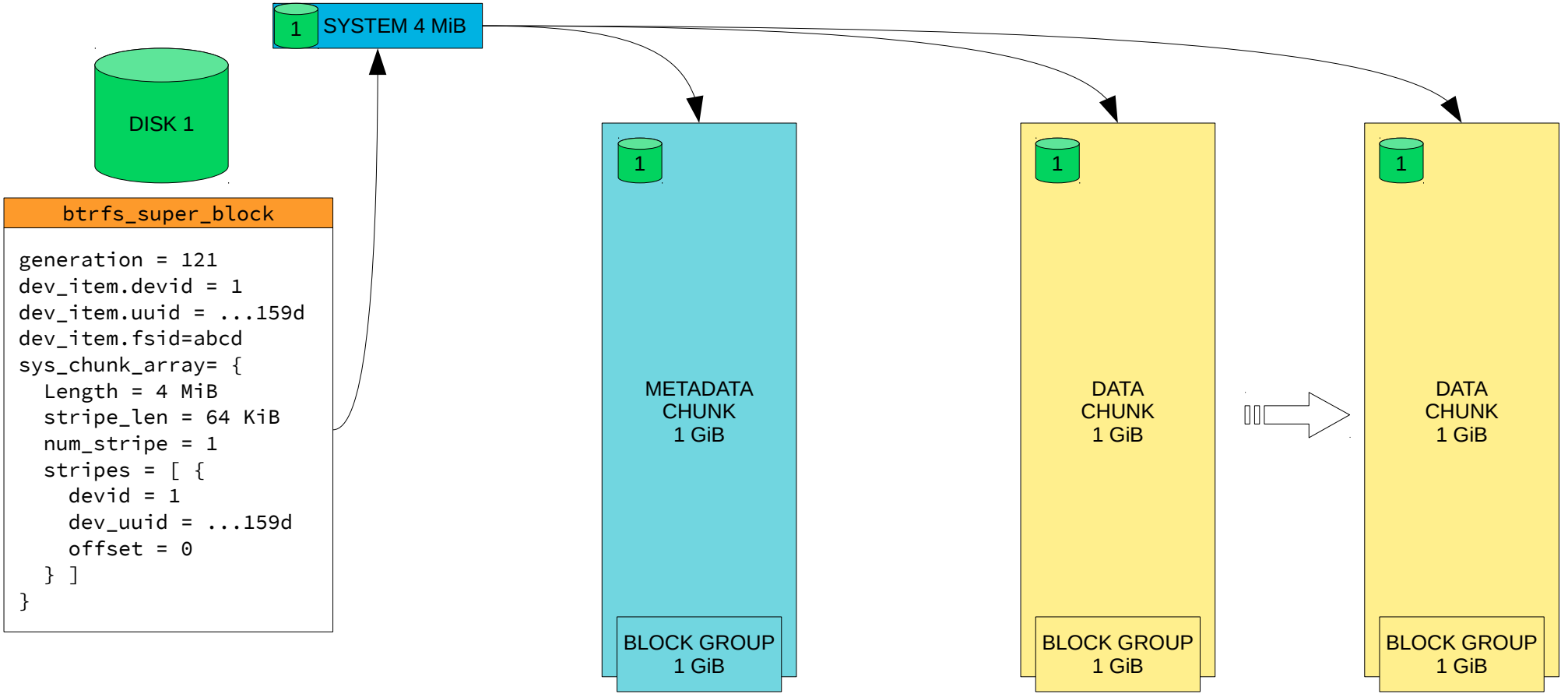
Chunk Allocation (SINGLE)



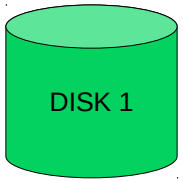
Chunk Allocation (SINGLE)



Chunk Allocation (SINGLE)



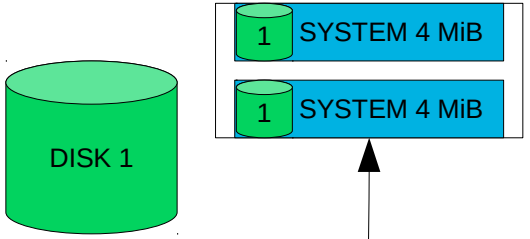
Chunk Allocation (DUP METADATA)



btrfs_super_block

```
generation = 121
dev_item.devid = 1
dev_item.uuid = ...159d
dev_item.fsid=abcd
sys_chunk_array= {
  length = 8 MiB
  stripe_len = 64 KiB
  num_stripe = 2
  stripes = [ {
    devid = 1
    dev_uuid = ...159d
    offset = 20971520
  }, {
    devid = 1
    dev_uuid = ...159d
    offset = 29360128
  } ]
}
```

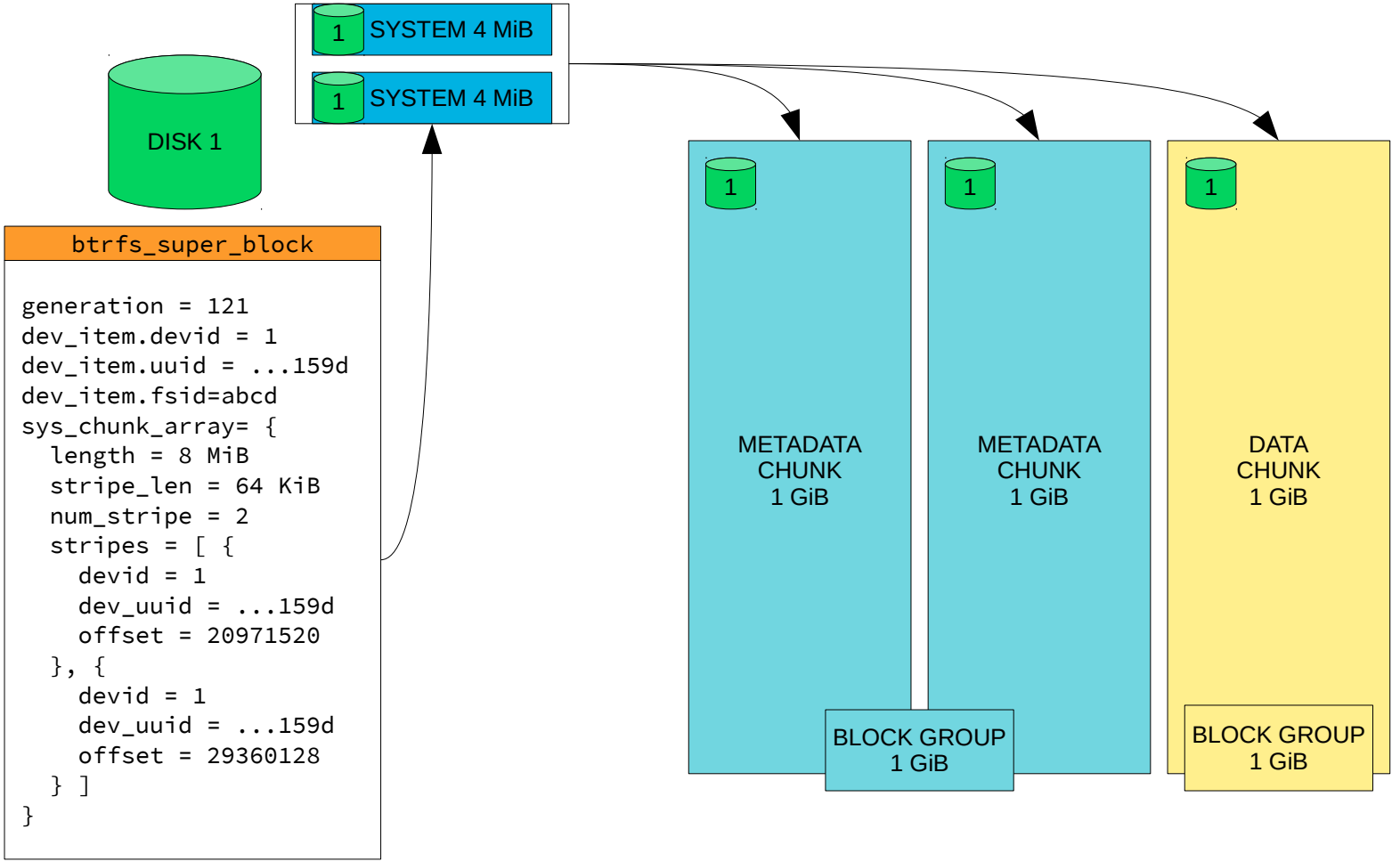
Chunk Allocation (DUP METADATA)



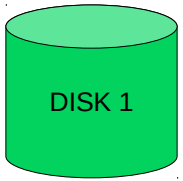
btrfs_super_block

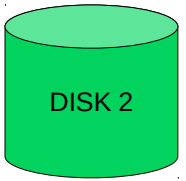
```
generation = 121
dev_item.devid = 1
dev_item.uuid = ...159d
dev_item.fsid=abcd
sys_chunk_array= {
  length = 8 MiB
  stripe_len = 64 KiB
  num_stripe = 2
  stripes = [ {
    devid = 1
    dev_uuid = ...159d
    offset = 20971520
  }, {
    devid = 1
    dev_uuid = ...159d
    offset = 29360128
  } ]
}
```

Chunk Allocation (DUP METADATA)



Chunk Allocation (RAID1 METADATA, 2 Disks)





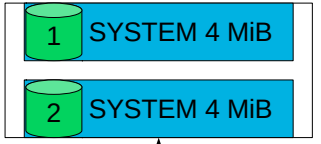
btrfs_super_block

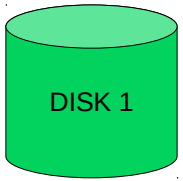
```
generation = 121
dev_item.devid = 1
dev_item.uuid = ...159d
dev_item.fsid=abcd
sys_chunk_array= {
  length = 8 MiB
  stripe_len = 64 KiB
  num_stripe = 2
  stripes = [ {
    devid = 1
    dev_uuid = ...159d
    offset = 1048576
  }, {
    devid = 2
    dev_uuid = ...26ae
    offset = 20971520
  } ]
}
```

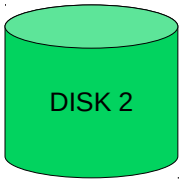
btrfs_super_block

```
generation = 122
dev_item.devid = 2
dev_item.uuid = ...26ae
dev_item.fsid=abcd
sys_chunk_array= {
  length = 8 MiB
  stripe_len = 64 KiB
  num_stripe = 2
  stripes = [ {
    devid = 1
    dev_uuid = ...159d
    offset = 1048576
  }, {
    devid = 2
    dev_uuid = ...26ae
    offset = 20971520
  } ]
}
```

Chunk Allocation (RAID1 METADATA, 2 Disks)







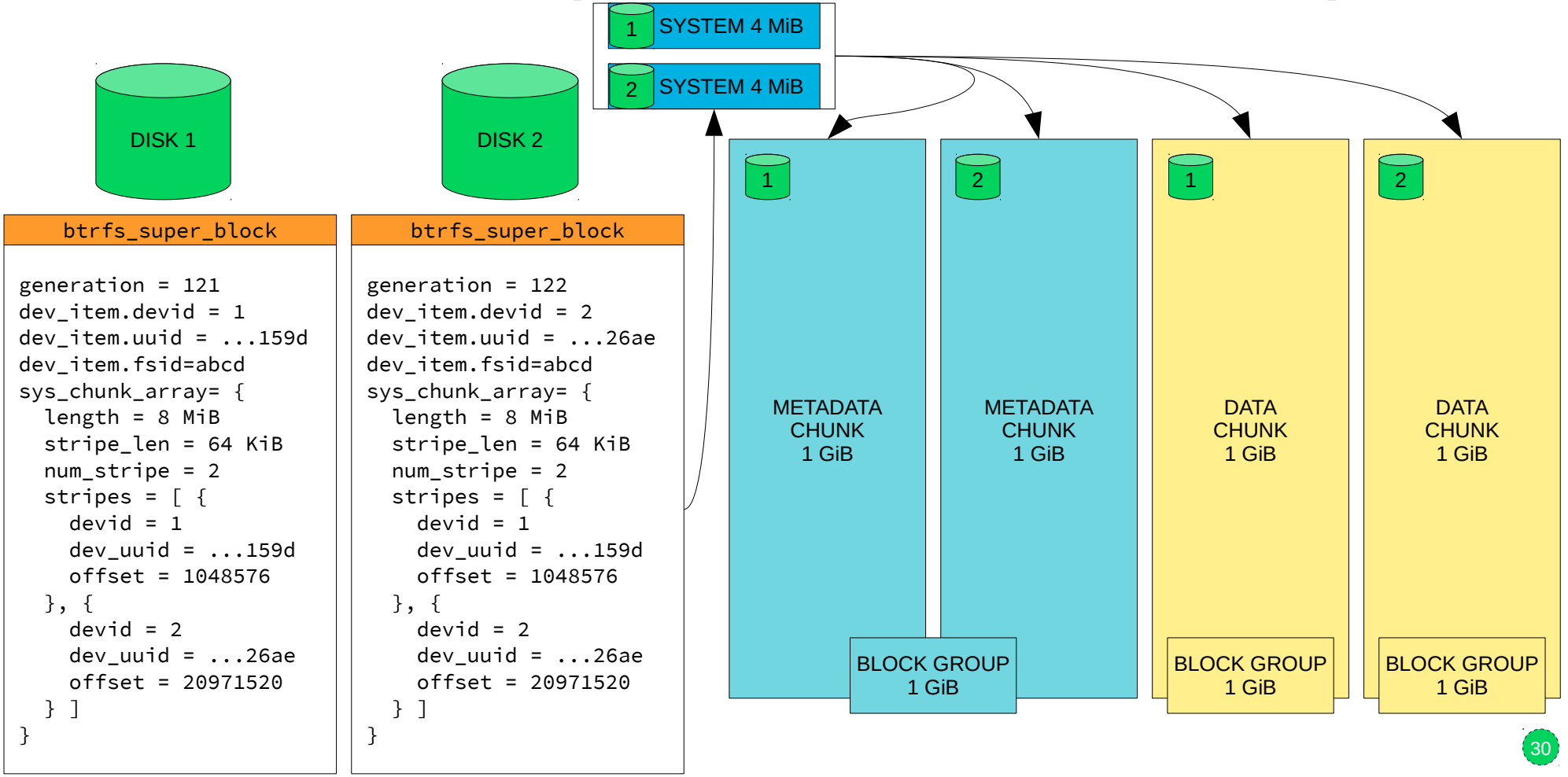
btrfs_super_block

```
generation = 121
dev_item.devid = 1
dev_item.uuid = ...159d
dev_item.fsid=abcd
sys_chunk_array= {
  length = 8 MiB
  stripe_len = 64 KiB
  num_stripe = 2
  stripes = [ {
    devid = 1
    dev_uuid = ...159d
    offset = 1048576
  }, {
    devid = 2
    dev_uuid = ...26ae
    offset = 20971520
  } ]
}
```

btrfs_super_block

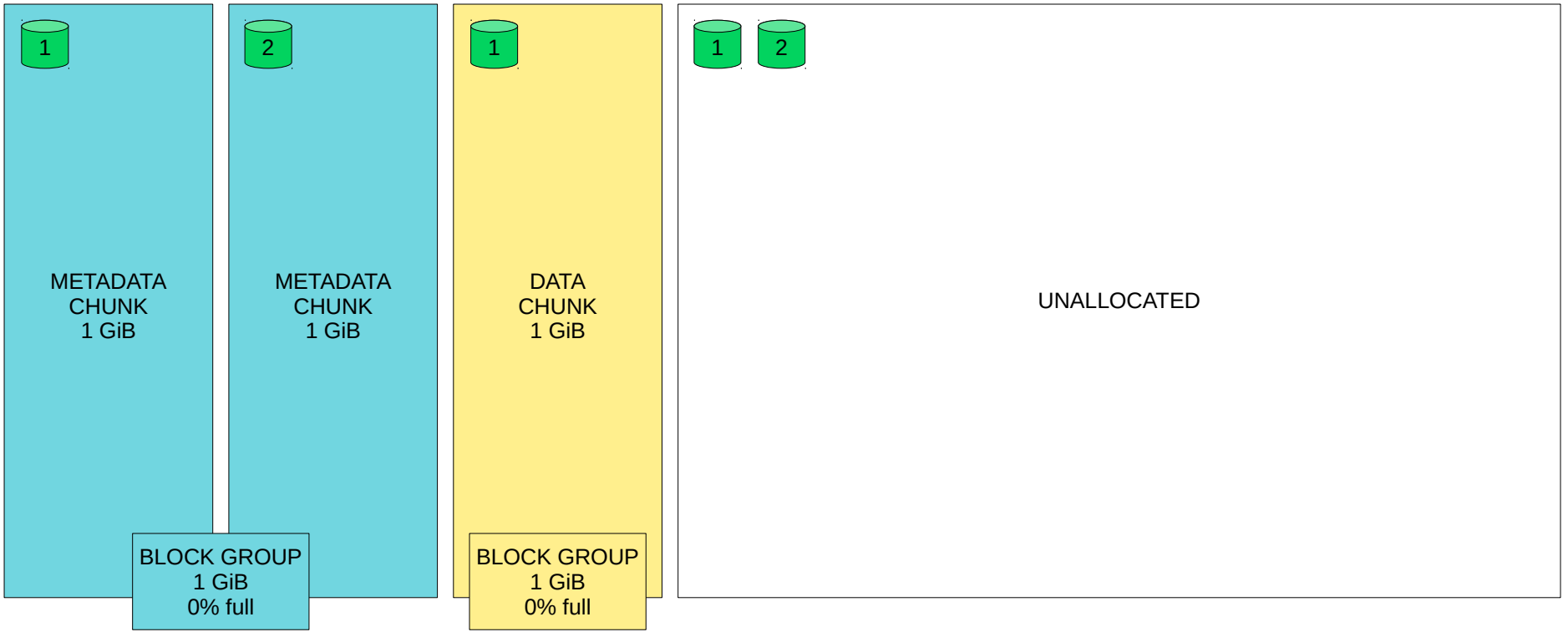
```
generation = 122
dev_item.devid = 2
dev_item.uuid = ...26ae
dev_item.fsid=abcd
sys_chunk_array= {
  length = 8 MiB
  stripe_len = 64 KiB
  num_stripe = 2
  stripes = [ {
    devid = 1
    dev_uuid = ...159d
    offset = 1048576
  }, {
    devid = 2
    dev_uuid = ...26ae
    offset = 20971520
  } ]
}
```

Chunk Allocation (RAID1 METADATA, 2 Disks)



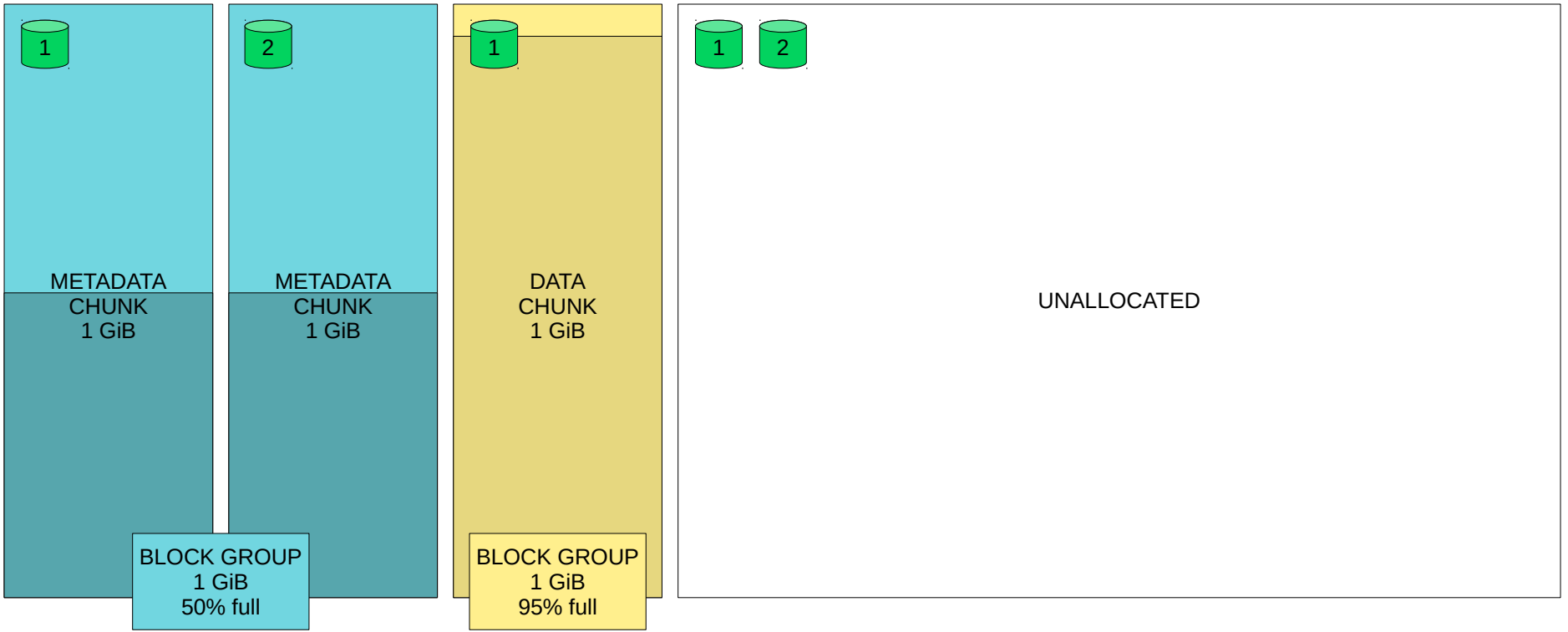
Chunk Allocation and Balancing

File system is initially empty.



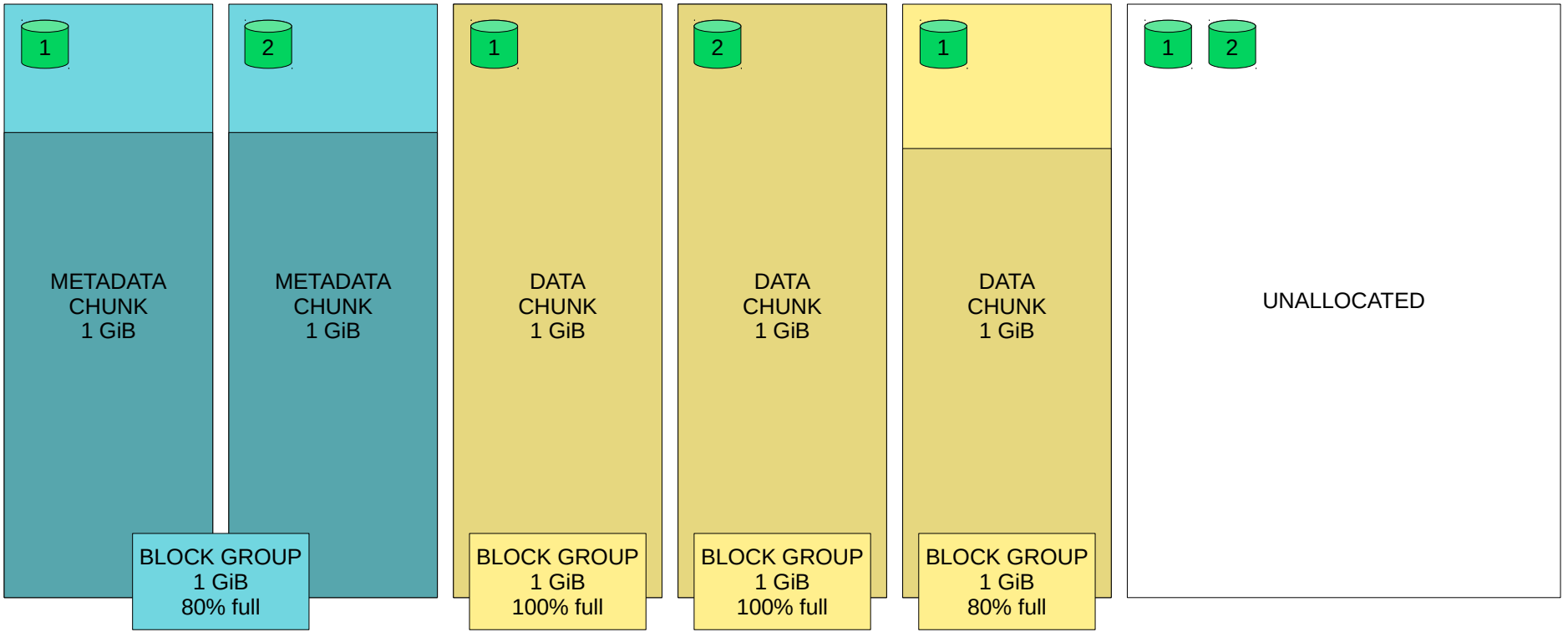
Chunk Allocation and Balancing

Creation of many medium-sized files fills the DATA CHUNK to 95% and the RAID1 METADATA CHUNKs to 50%.



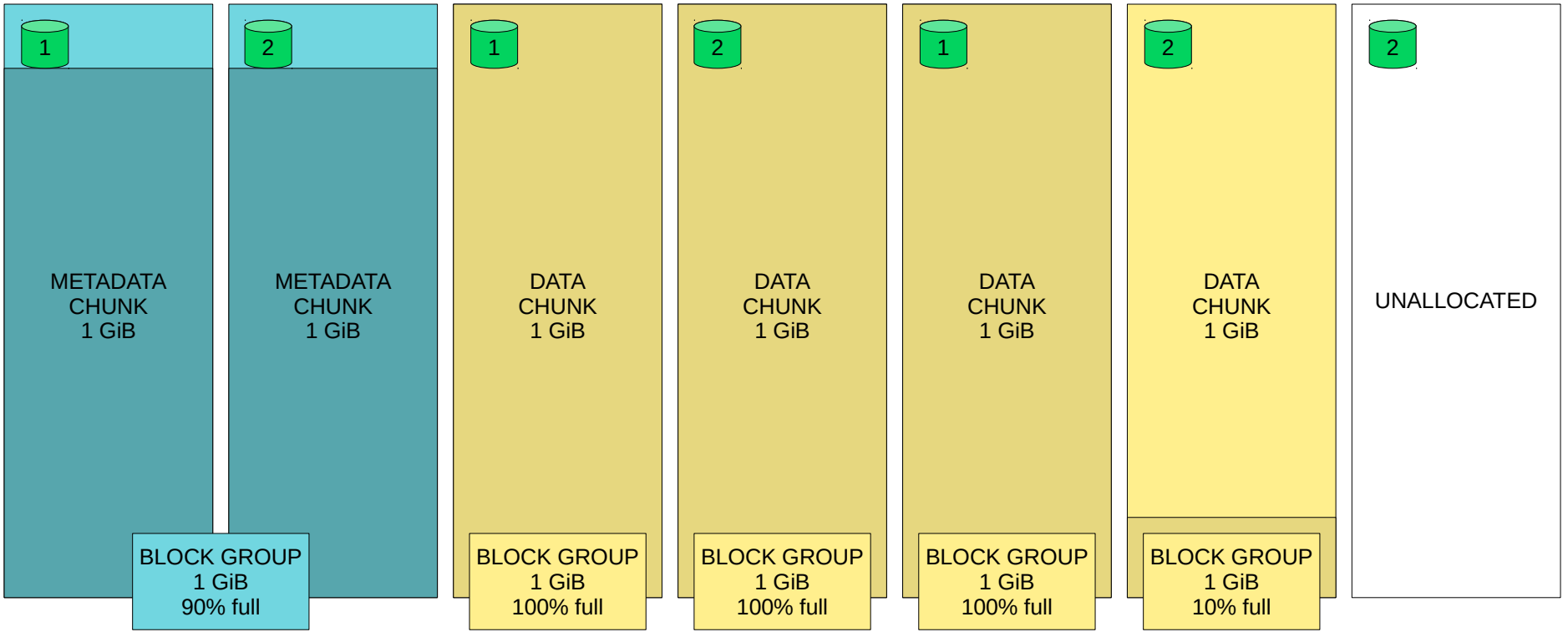
Chunk Allocation and Balancing

Continued activity raises METADATA CHUNK consumption to 80% and causes Btrfs to allocate two new DATA CHUNKs.



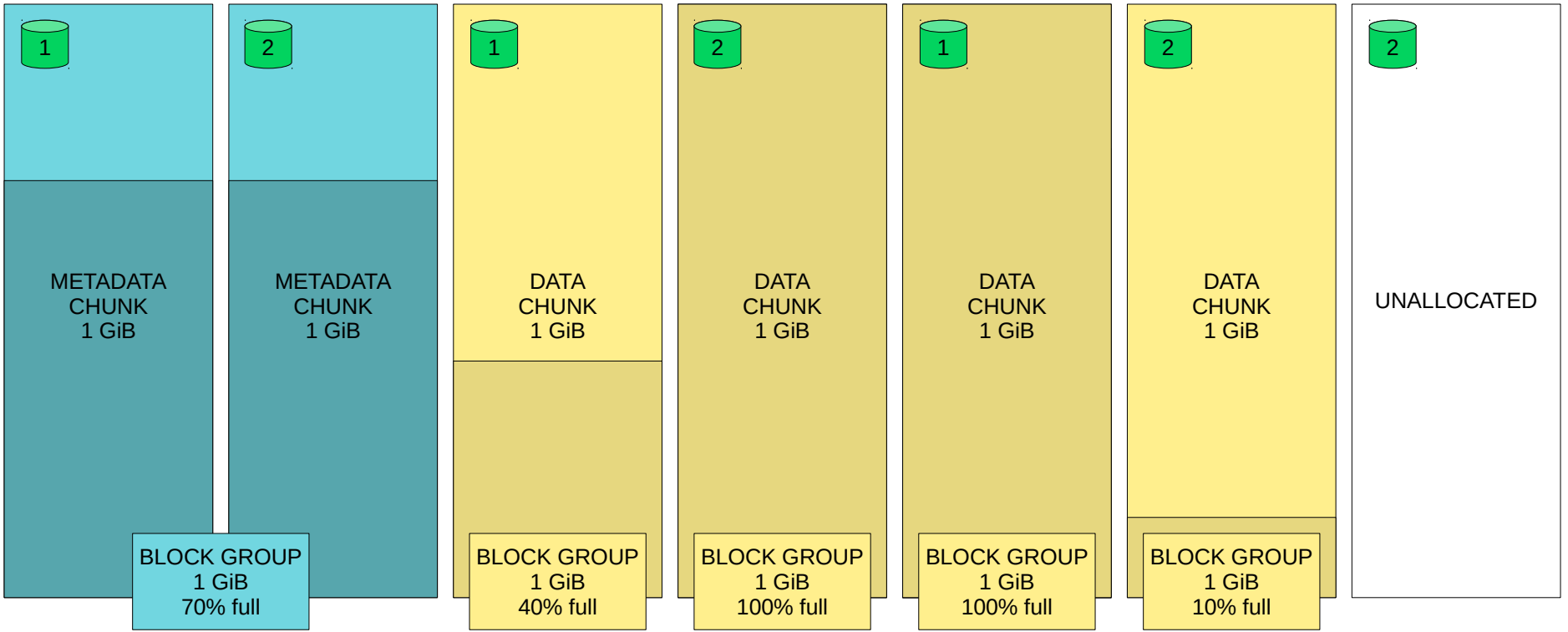
Chunk Allocation and Balancing

Continued activity raises METADATA CHUNK consumption to 90% and causes Btrfs to allocate a another new DATA CHUNK.



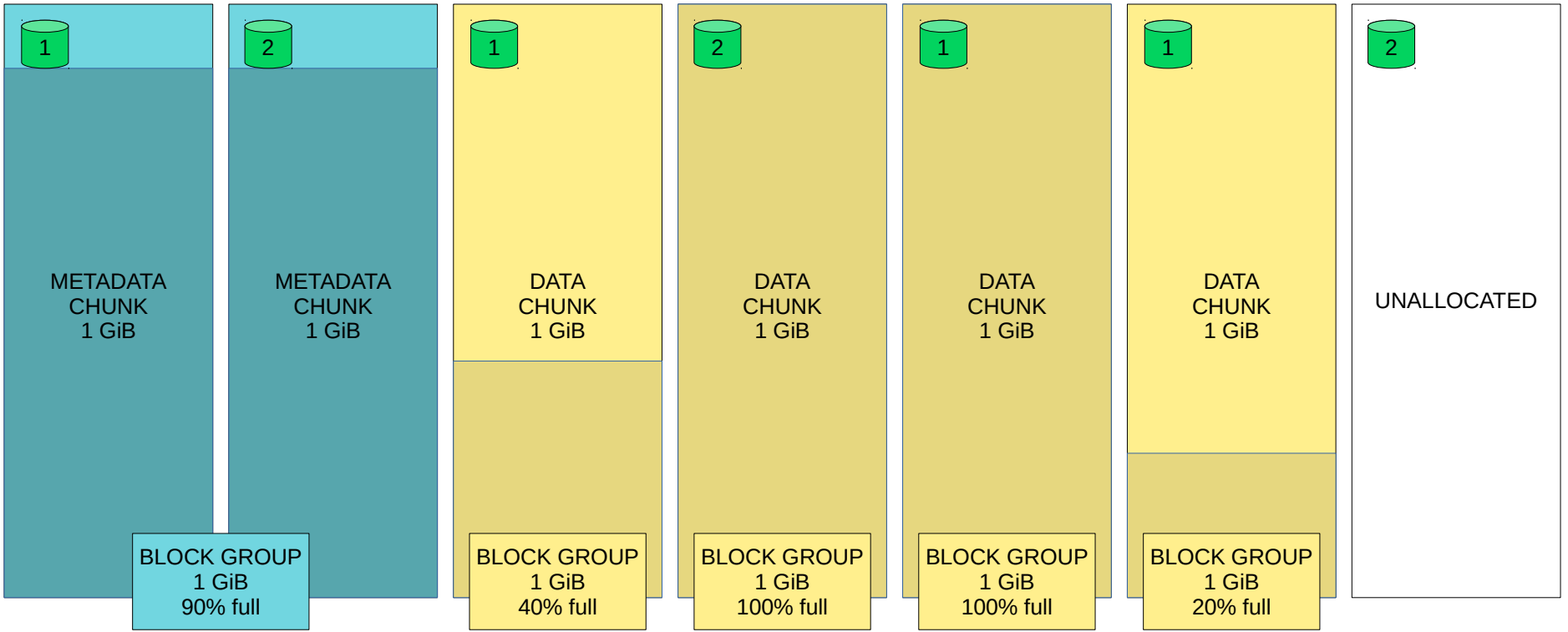
Chunk Allocation and Balancing

Deletion of files results in reducing METADATA CHUNK consumption to 70% and usage of a DATA CHUNK drops to 40%.



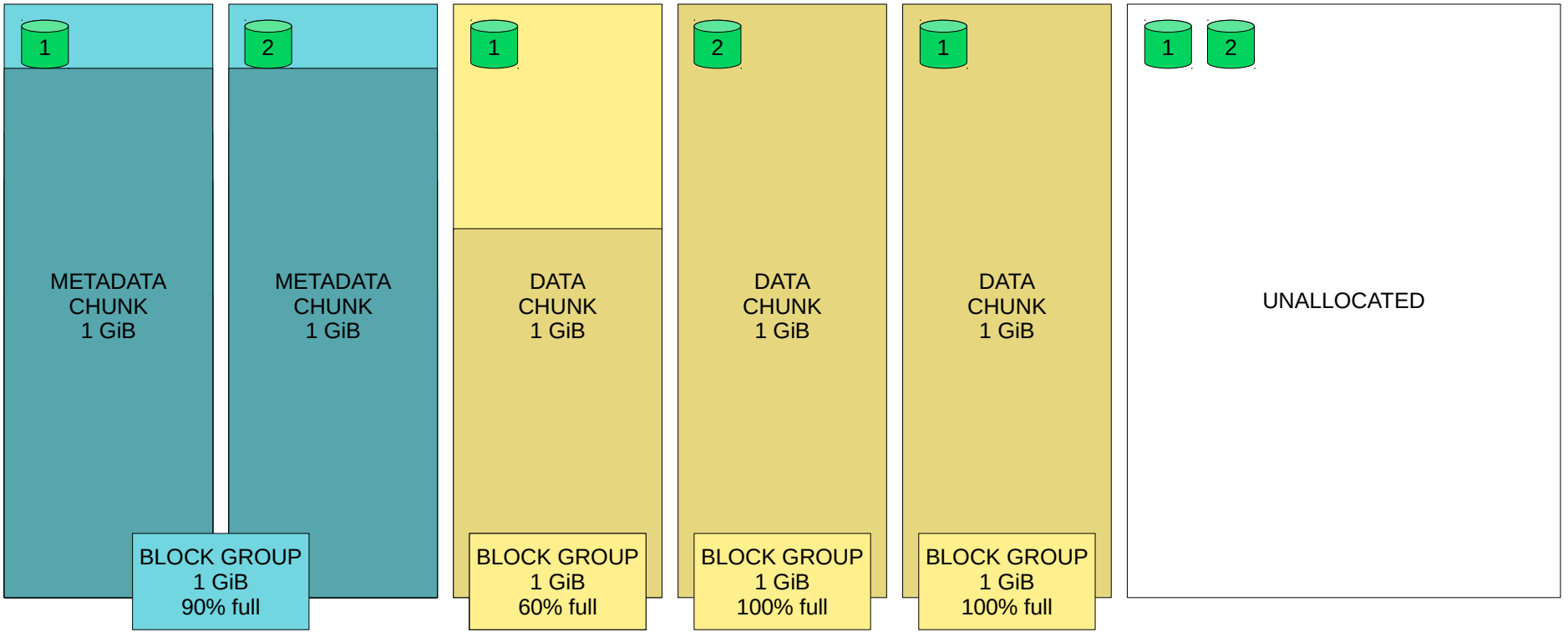
Chunk Allocation and Balancing

Growing some already-existing files raises METADATA CHUNK consumption to 90% and one of the DATA CHUNKs to 20% full.



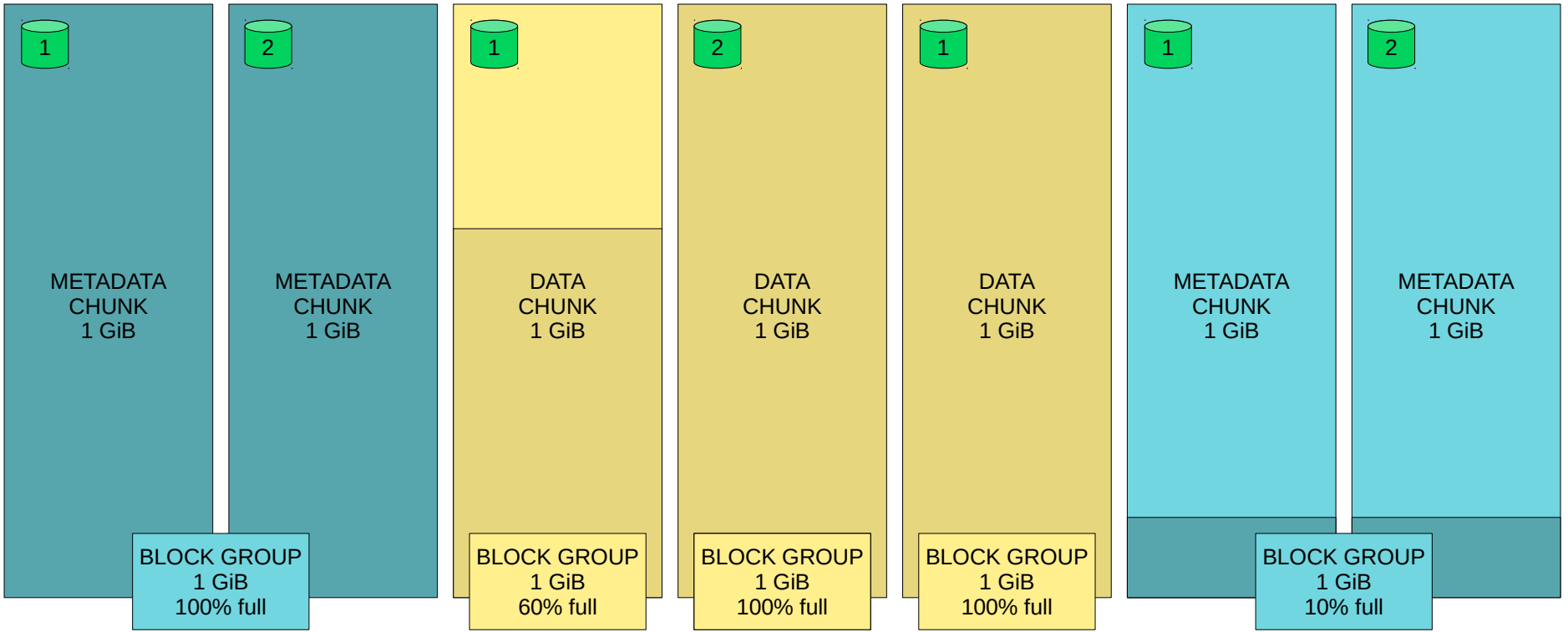
Chunk Allocation and Balancing

A BALANCE operation occurs, relocating extents from the 20% full DATA CHUNK to the 40% full DATA CHUNK, raising consumption to 60% and freeing the now-unused DATA CHUNK for reuse.

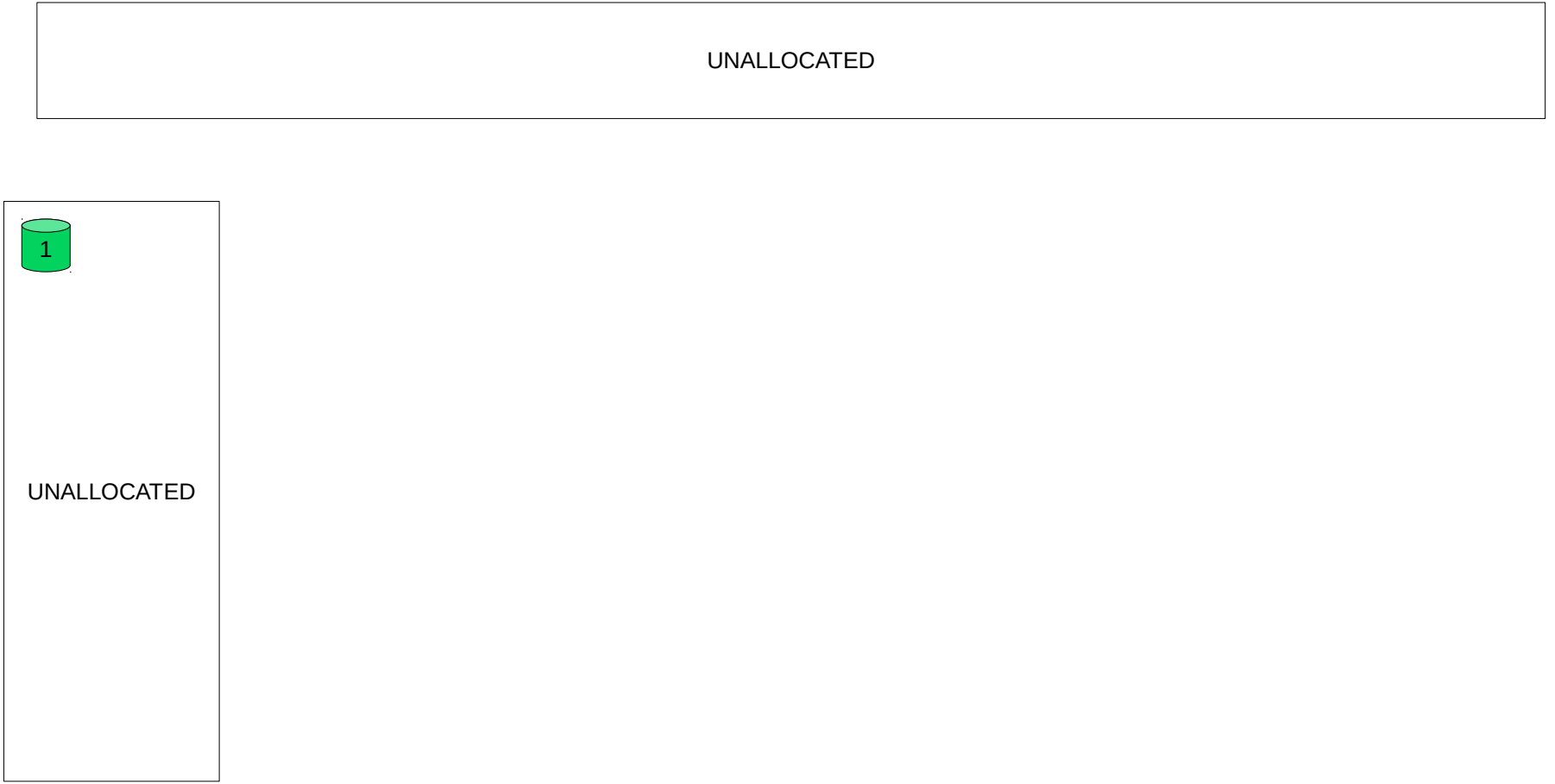


Chunk Allocation and Balancing

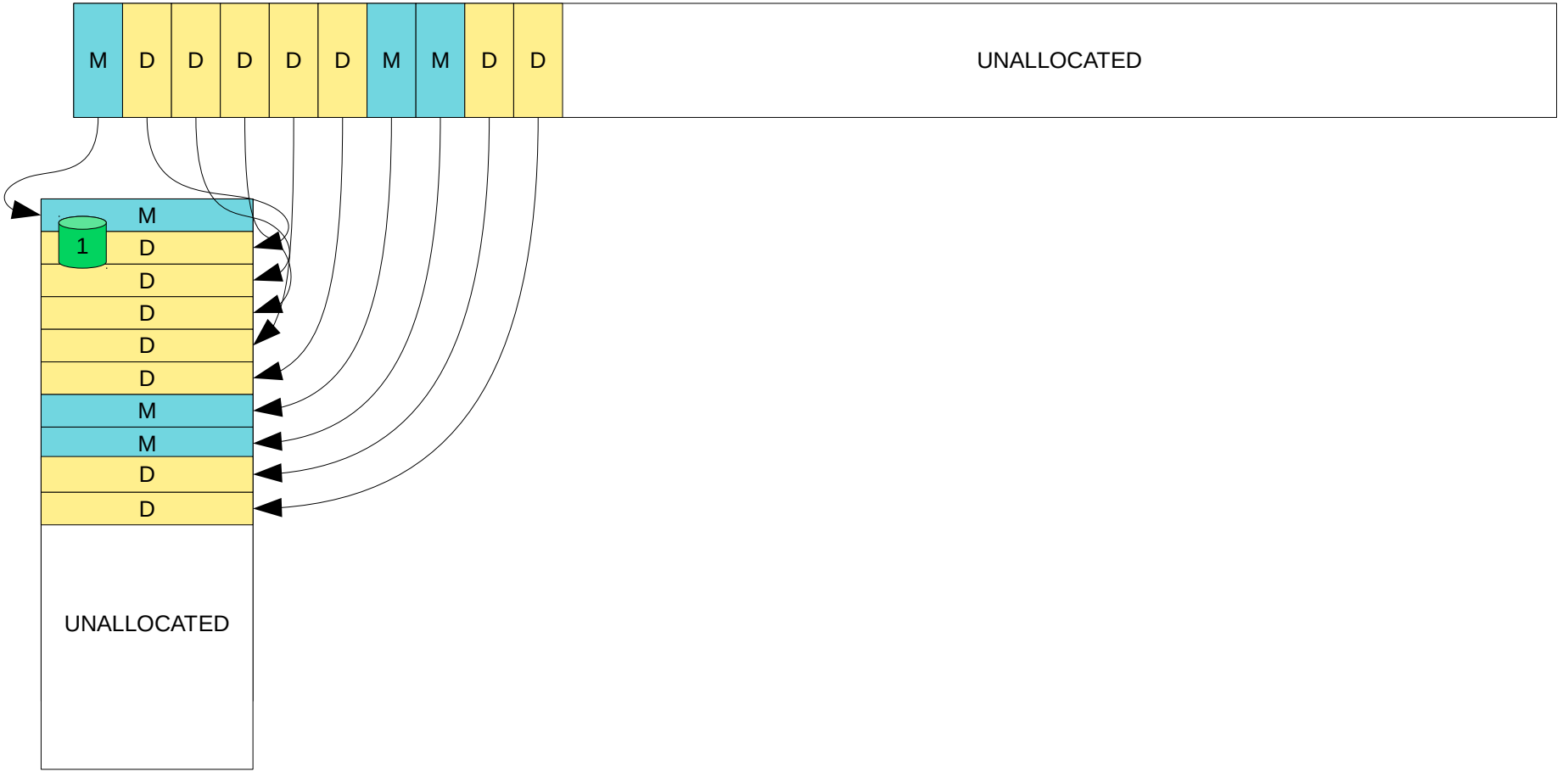
Creation of many very small files, which are created as INLINE EXTENTS in the METADATA CHUNKs, raise consumption to 100% and require allocation of two new RAID1 METADATA CHUNKs.



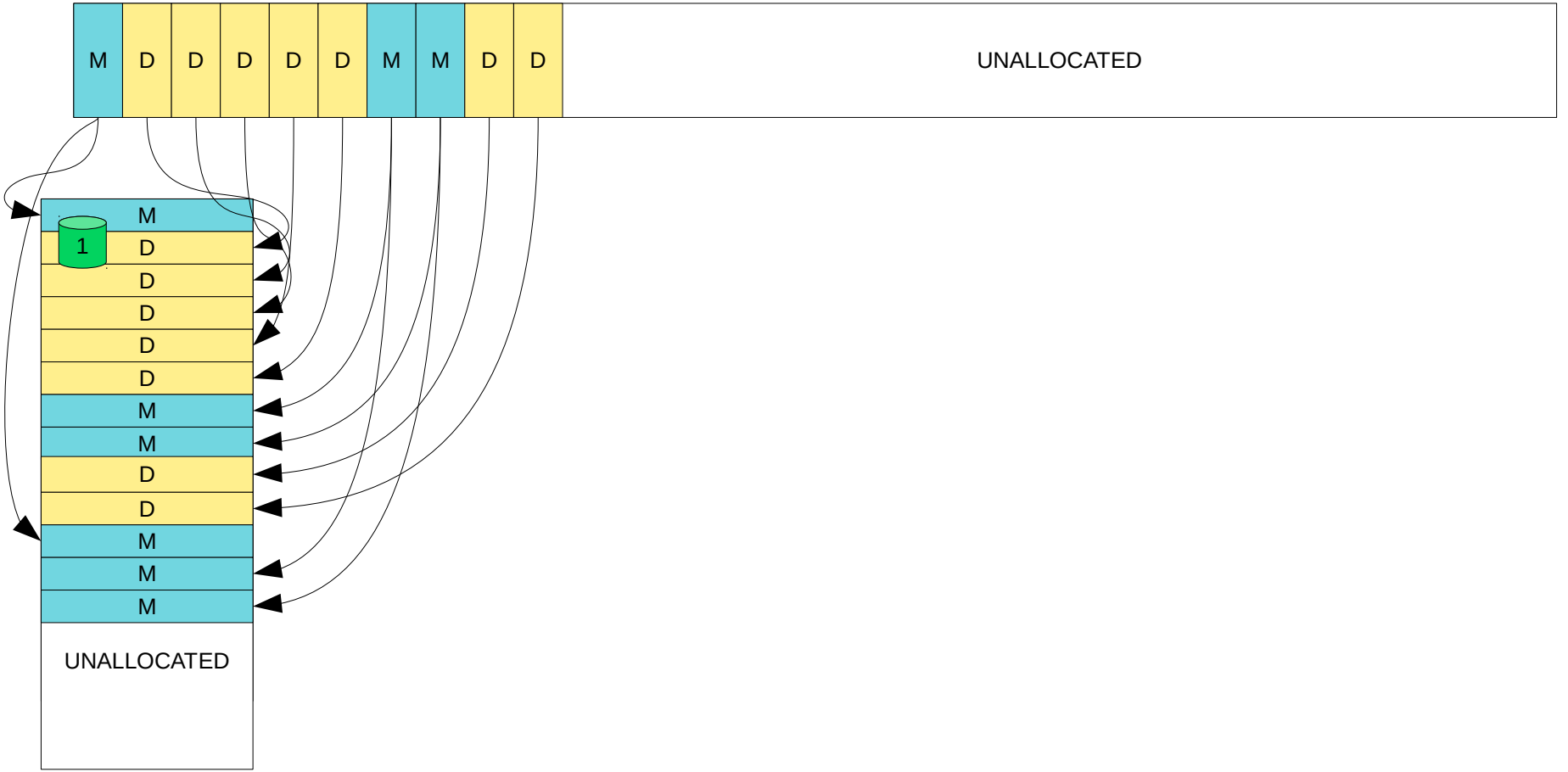
Logical Volume



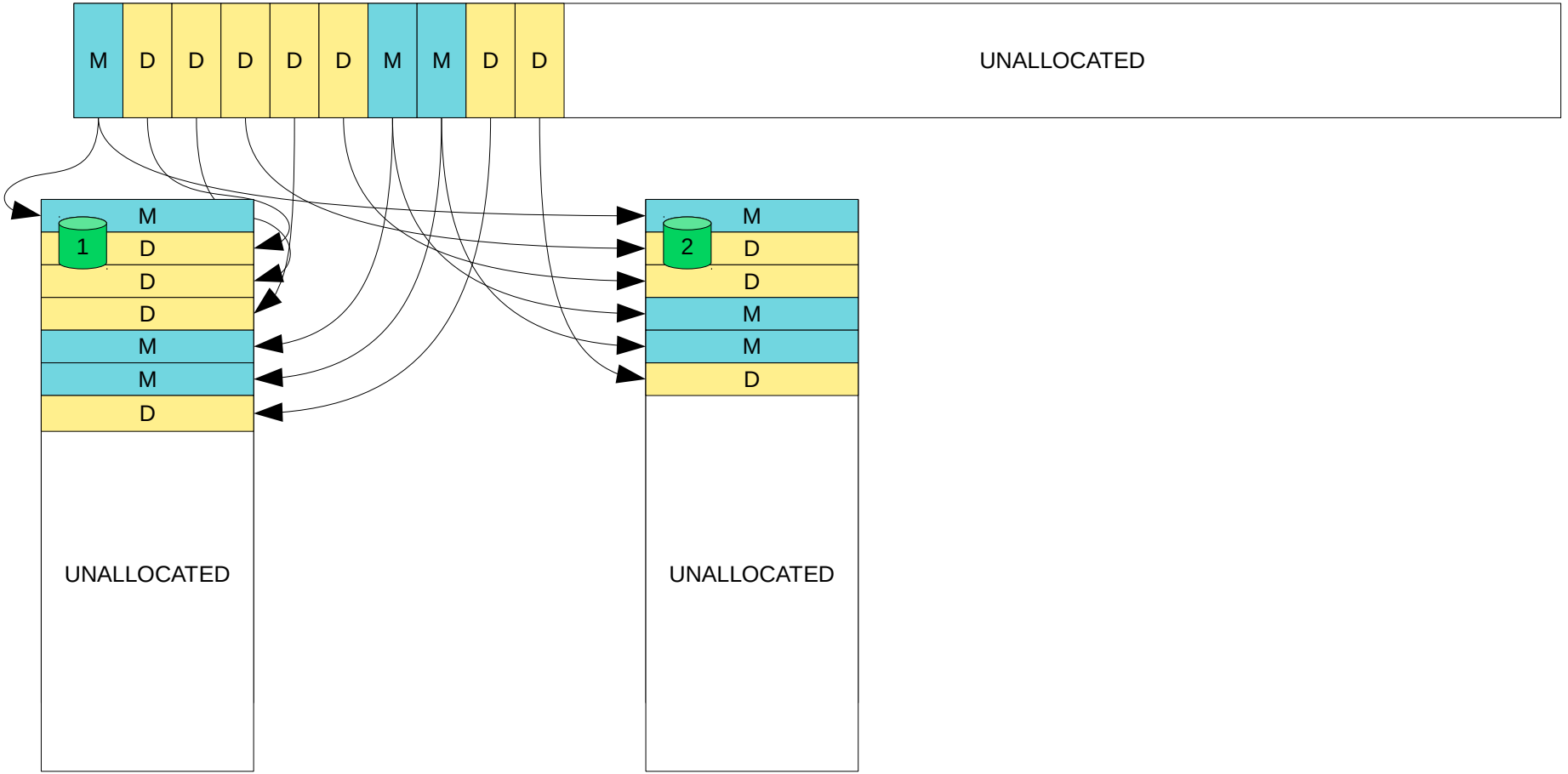
Logical Volume (SINGLE)



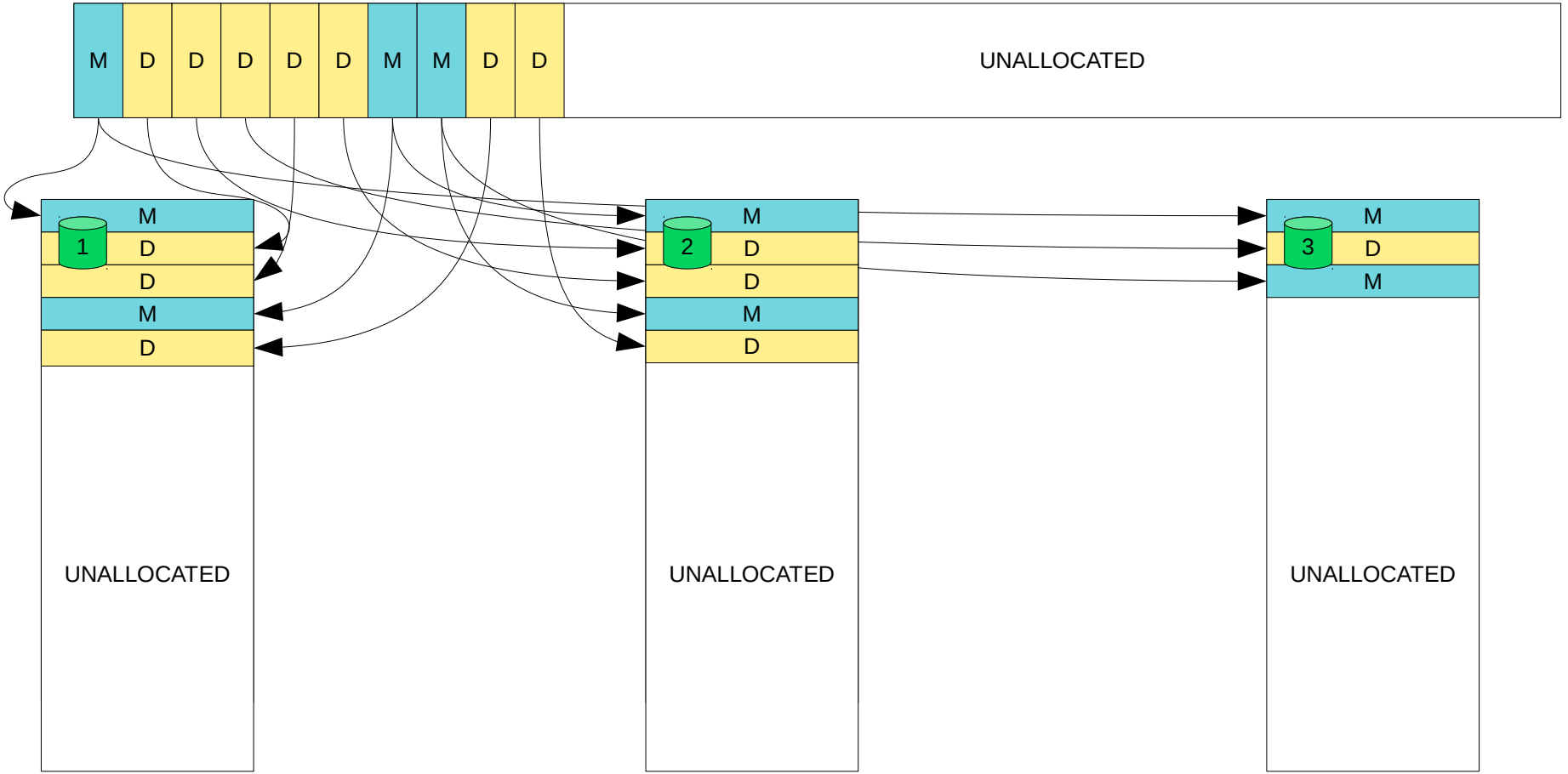
Logical Volume (DUP METADATA)



Logical Volume (RAID1 METADATA, 2 Disks)



Logical Volume (RAID1 METADATA, 3 Disks)



Useful Commands

Btrfs – df /

```
# df -h /
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda2        50G   38G   11G  78% /
```

Btrfs – btrfs filesystem usage /

```
# btrfs filesystem usage /
Overall:
    Device size:                  50.00GiB
    Device allocated:             40.84GiB
    Device unallocated:            9.16GiB
    Device missing:                0.00B
    Used:                         37.27GiB
    Free (estimated):             10.92GiB      (min: 6.34GiB)
    Data ratio:                    1.00
    Metadata ratio:                2.00
    Global reserve:              512.00MiB      (used: 0.00B)
```

Btrfs commands

```
# btrfs filesystem df /
Data, single: total=35.52GiB, used=33.76GiB
System, DUP: total=32.00MiB, used=16.00KiB
Metadata, DUP: total=2.62GiB, used=1.75GiB
GlobalReserve, single: total=512.00MiB, used=0.00B
```

Disk full

```
# df -h
Filesystem            Size  Used  Avail  Use%  Mounted on
/dev/sda2              38G    12G  13M  100%   /

# btrfs filesystem df /
Data, single: total=9.47GiB, used=9.46GiB
System, DUP: total=8.00MiB, used=16.00KiB
System, single: total=4.00MiB, used=0.00
Metadata, DUP: total=13.88GiB, used=1.13GiB
Metadata, single: total=8.00MiB, used=0.00
```

btrfs balance

btrfs balance start -m /

Running out of metadata space:
btrfs balance start -dusage=5 /

Compression

Compression

- File data extents can be compressed using one of several algorithms
  - Currently zlib and lzo (default since v3.6).
  - Metadata cannot be compressed.
- Multiple algorithms can be in use on the same file system and even in the same file if new extents are written.
- Compression can be enabled or disabled on a filesystem-wide, directory, or file basis. Subvolume compression is not implemented but is on the project roadmap.

Controlling Compression

- Can be controlled in several ways:
  - All options apply only to newly written extents. Already written extents are not rewritten
  - Mount the file system with `-o compress[=algo]`
    - Every new extent will be compressed if the compressed size is smaller than the uncompressed size and the `force_compress` flag is not set on the inode. If a uncompressed extent is written, future extents will not be compressed.
  - Mount the file system with `-o compress-force[=algo]`
    - Every new extent will be compressed regardless of whether the compressed size is smaller.
  - `chattr +c <file|directory>`
    - New extents written to this file will be compressed similarly to `-o compress`. If the flag is set on a directory, it will be inherited by any newly created files or subdirectories. Mounting with either of the compression mount options does not affect this flag on any file.

Controlling Compression

- `btrfs property set <file|directory> compression “”`
  - New extents written to this file will not be compressed regardless of the mount options.
- `btrfs property set <file|directory> compression “<algo>”`
  - New extents written to this file will be compressed with the specified algorithm, regardless of the algorithm used with `-o compress`.

- Disabling compression on a file will *not* result in the already-written extents to be decompressed on disk.
  - Rewriting the extents is the only way to accomplish this. The extents must be rewritten without compression enabled, either at the file level via disabling compression, or by clearing +c and mounting without the compression options enabled.
  - The most obvious way to rewrite the extents is to copy the file and move it back into place.
  - Alternatively, `btrfs filesystem defrag <path>` will accomplish the same thing without disturbing access to the file.

Things to Consider

- Interaction with subvolume quotas (qgroups):
  - The subvolume quotas track the amount of space occupied on disk. The *compressed* size is accounted against the limit.
- Drawbacks:
  - Extents are compressed in segments of up to 128 KiB in size. Accessing a single block in the middle requires that the entire segment be decompressed.
  - No hardware assist for compression (yet).
- Compatibility:
  - ✔ CoW
  - ✗ NoDataCoW
    - The entire segment must be decompressed to be accessed and may span multiple blocks.
  - ✗ DirectIO
    - The entire extent must be buffered in order to perform (de-)compression. Users of DirectIO will fall back to buffered I/O when accessing compressed extents.

Send / Receive

Send

- Exports a read-only subvolume as a serial stream of commands
  - Can also export differences between two read-only, related subvolumes
- Introduced in SLES 11 SP4, restricted to send metadata only until SLE 12 SP4.
  - Used by snapper diff to retrieve a list of changed files between snapshots very quickly

- `btrfs send /.snapshots/4/snapshot`
  - Sends the entire snapshot's data and metadata

- Commands are simple operations like "mkdir", "create snapshot", "unlink", "write", etc.

Send parent vs clone

- `btrfs send -p /.snapshots/3/snapshot /.snapshots/4/snapshot`
  - Sends the differences between snapshot 3 and 4, including all metadata and data
  - Assumes only that the remote file system has a shared subvolume

- `btrfs send -c /.snapshots/3/snapshot /.snapshots/4/snapshot`
  - Sends the differences between snapshot 3 and 4, including all metadata and data not already on the remote
  - Assumes that some extents already exist in the remote because they exist in a shared subvolume – uses CLONE_RANGE on remote side
  - Faster but less resilient

Receive

- Receives a stream of changes and replicates a remote subvolume in the local file system
  - Command processing in userspace, translates to POSIX system calls and btrfs ioctl calls
  - Does not modify file system metadata directly
- Introduced in SLE 12 SP2
  - In line with the SLE process of rolling out new btrfs features as SUSE considers them mature.
- Will safely fail if:
  - The the receiving subvolume already exists
  - A previously received subvolume has been modified
  - The default subvolume has changed or it is not mounted at the top-level subvolume

- `btrfs receive /mountpoint`

Deduplication

Challenges

- Identifying identical data
  - Offset
  - Chunk size
  - Cost / benefit for “perfect” identification
  - File system contents may vary broadly

Challenges

- Identifying identical data
  - Offset
  - Chunk size
  - Cost / benefit for “perfect” identification
  - File system contents may vary broadly
- Performance
  - Chunk size: trade-off between space savings, overhead, and fragmentation
  - Method: trade-off between “perfect” dedupe and write overhead
  - Anything examining data carries an access cost

Challenges

- Identifying identical data
  - Offset
  - Chunk size
  - Cost / benefit for “perfect” identification
  - File system contents may vary broadly
- Performance
  - Chunk size: trade-off between space savings, overhead, and fragmentation
  - Method: trade-off between “perfect” dedupe and write overhead
  - Anything examining data carries an access cost
- Limitations
  - May not always be possible to release deduplicated space

“In-Band” deduplication

- Could be implemented by the file system or the storage system
- Data is examined for duplicates before writing
- Benefits:
  - No temporary storage of duplicate data until deduplicator runs

“In-Band” deduplication

- Could be implemented by the file system or the storage system
- Data is examined for duplicates before writing
- Benefits:
  - No temporary storage of duplicate data until deduplicator runs
  - Data is processed once, at the time of write
- Drawbacks:
  - May slow down writing substantially with little benefit for many workloads
  - It’s effectively a read/write cache; If the hit rate is high, the trade-off may be worthwhile
  - Difficult to define policy beyond simple heuristics
- No file system support in Btrfs (yet)

“Out-of-band” Deduplication

- User executed tool
- Examines data at rest
- Benefits:
  - No impact on write performance
  - Can be scheduled for periods of low write activity
  - Different rules for different files:
    - Dedupe several VM images with the same operating system installed
    - Skip files for which no fragmentation cost is acceptable entirely
    - Apply “expensive” duplicate identification rules to files that may benefit more
    - Little benefit in comparing files with little probability of duplicated data
- Drawbacks:
  - Temporary storage is still used by duplicated data
  - Needs to be scheduled for use
  - May process the same data many times

BTRFS Mechanics

- In either case, the duplicate extent will be CLONED from an extent that is already on disk.
- This is the same mechanism that snapshots and `cp --reflink` use.
- Has the same caveats that snapshots and reflinks do: in addition to new writes causing a CoW in the middle of an extent, operations like `btrfs filesystem defrag <path>` will also cause the extent to be copied back out into a separate extent.

“Out-of-band” Deduplication: duperemove

- “duperemove”
  - Developed by SUSE as an open source project
  - Supports a hash database to cache previous runs. On the next run, only changed files will be examined if run with the same parameters.
  - https://github.com/markfasheh/duperemove
  - Interruptible; Restartable
  - Introduced in SLES 11 SP4
  - Getting faster and using fewer resources
  - Uses FIEMAP ioctl to map already-shared files

Duperemove: Simple usage

- Deduplicate several VM images
  - `duperemove -d <image1> <image2> <image3> ...`
- Deduplicate files recursively
  - `duperemove -dr /path`
  - Only use for small data sets – memory consumption can grow quite large
- Use duperemove with fdupes to dedupe completely identical files
  - `fdupes -r /path | duperemove --fdupes`

Duperemove: Hash files

- When the hash database is in use, duperemove will stream the calculated hashes to the database instead of caching them all in memory
  - Less memory usage
  - Can reuse the hashes that are still valid (mtime) next time
- Same recursive dedupe as previous slide, but safe for larger data sets
  - `duperemove -dr –hashfile=duperemove.hash /path`
- Same run but skip new files
  - `duperemove -dr –hashfile=duperemove.hash`
- Add a new directory to scan; Will reuse old hashes and add newly discovered files
  - `duperemove -dr –hashfile=duperemove.hash /path /newpath`
- List tracked files
  - `Duperemove -L –hashfile=duperemove.hash`

Questions?

Thank you.

File System Internals

Extent Tree

Extent Tree

- Block groups:
  - Refer to chunks as a flat representation of the underlying allocation policy
  - Tracks usage as a simple counter
- Extent items:
  - Describes space used by file data, which are outside of the btree
  - Contains back references to each file or shared owner
  - Tracks references to each extent
- Metadata items
  - Describes space used by metadata, which is a part of the btree
  - Backrefs to shared blocks or roots
  - Tracks references to each extent

FS Tree

FS Tree

- “Global” FS root
  - / in a traditional file system
  - Btrfs can use any subvolume as / via “btrfs subvolume set-default” command
  - Can’t be removed; Otherwise identical to any other subvolume
- Contains traditional file system objects
  - Inodes
  - Directories
  - References to other subvolumes
  - Extent references for files
    - File data extents may point to segments of extents rather than the entire extent
  - Inline extents
- Slightly less traditional: Backreferences for everything

Other Trees

Other Trees

- Root Tree
  - Contains location information for the root of every other tree
- Quota Tree
  - Contains quota objects for subvolume quotas (qgroups)
- Dev Tree
  - Contains information on constituent devices
- UUID Tree
  - Used for mapping UUIDs to other objects (e.g. subvolumes with UUIDs)
- Csum Tree
  - Contains the crc32c checksums for data extents.
- Free Space Tree (new)
  - Faster free space tracking for the extent tree


SUSE®
We adapt. You succeed.