BTRFS Declustered Parity RAID For Zoned Devices

14 September, 2022

System Software Group, WD Research

Johannes Thumshirn

Outline

- Background
  - Btrfs Overview
  - Zoned Devices
  - ZONE APPEND Write Operations
  - Btrfs On Zoned Devices
- Problem Statement
  - Lessons Learned From RAID5/6

- Proposed Changes
  - Distribute Data Placement
  - Journaling
  - Configurable Parity Algorithm

- Design Background
  - Distributed Data Placement
  - RAID Stripe Tree
- Current Status
  - Outlook
  - Screenshots

R

Background

Btrfs Overview

What's btrfs?

- Copy-on-Write Filesystem
  - Based on CoW B-Trees
  - Snapshots
  - Subvolumes
- Additional Features
  - Transparent data compression
    - lzo, zlib or zstd
  - Checksums for data and metadata
    - crc32c, xxhash64, sha256, blake2b
  - Built-in multi device support (RAID)
    - RAID 0, RAID 1, RAID 10, RAID 5, RAID 6
  - Incremental backups with send/receive
    - Send stream of changes between two subvolume snapshots

Zoned Block Devices

What's ZBC, ZAC And ZNS?

- Most commonly found today in the form of SMR hard-disks (Shingled Magnetic Recording) or ZNS SSDs
  - Defined in SCSI ZBC, ATA ZAC and NVMe ZNS
- LBA range divided into zones
  - Conventional zones
    - Accept random writes
  - Sequential write required zones
    - Writes must be issued sequentially starting from the "write pointer"
    - Zones must be reset before rewriting
- Users of zoned devices must be aware of the sequential write rule
  - Device fails write command not starting at the zone write pointer

ZONE APPEND Write Operations

Introduced with NVMe Zoned Namespace (ZNS) SSDs

- ZONE APPEND write operation only specifies the target zone
  - The device automatically write at the current write pointer position of the zone
  - The first written LBA number is returned to the host with the command completion notification

- ZONE APPEND command is not defined in the ZBC (SCSI) and ZAC (ATA) standards
  - Emulated in the SCSI disk driver since kernel version 5.8
- With zone append, writes to a zone can be delivered in any order without failing
  - User must however be ready to handle out-of-order completions

Btrfs On Zoned Block Devices

What we've done

- Basic support merged with kernel v5.11
  - Log structured super block
    - Superblock is the only fixed location data structure in btrfs
  - Align block groups to zones
  - Zoned extent allocator
    - Append only allocation to avoid random writes

- Fully functional since kernel v5.12
  - Use ZONE APPEND for data writes
  - Not yet completely on par with regular BTRFS features
    - No NOCOW
    - No fallocate(2)
    - No RAID yet

- NVMe ZNS support since kernel v5.16
  - Zone capacity smaller than zone size
  - Respecting `queue_max_active_zones()` limits
- Currently in stabilization phase
  - Automatic zone reclaim merged in v5.13
  - Bug fixes for corner cases

Problem Statement

Problem Statement

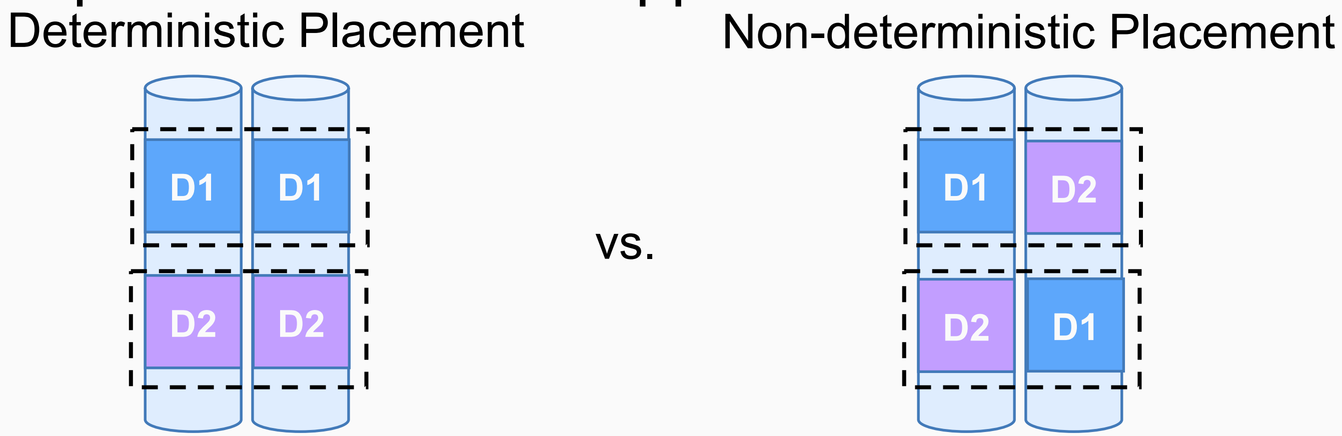
Lessons Learned From Btrfs RAID5/6

- Disconnection of ”File-Extent-Layer” and “RAID-Layer”
  - Sub stripe length updates in place
    - RAID Write Hole
    - Not possible on a zoned btrfs
  - CoW needs to know about RAID and vice versa
  - Needs to work with “nocow” files/filesystem as well

Problem Statement

Lessons Learned From Btrfs RAID

- Implicit data placement
  - Each per disk sub-stripe has same offset from chunk start
- Doesn’t work with a zoned filesystem (even for RAID 1)
  - Multiple writes to different drives can race
    - No explicit write position with zone append command: the drives decides



Problem Statement

Lessons Learned From RAID

- RAID Rebuild Stress
  - RAID5 can only tolerate one missing drive, two for RAID 6
  - High stress on remaining drives for rebuild
  - Increased chance of disk dying during rebuild

- Inflexible Encoding Scheme
  - XOR for RAID 5 (P-Stripe)
  - XOR and Shift for RAID 6 (Q-Stripe)

H

Proposed Changes

Proposed Changes

How to fix these problems

- Distribute Data Placement
  - Similar to what BTRFS RAID 1 already does
  - Less pressure on single disks in recovery

- Copy-on-Write to circumvent write hole
  - Introduce RAID Stripe Tree
  - Write data first, then meta-data describing the stripe
  - Allows us to use REQ_OP_ZONE_APPEND for zoned data writes

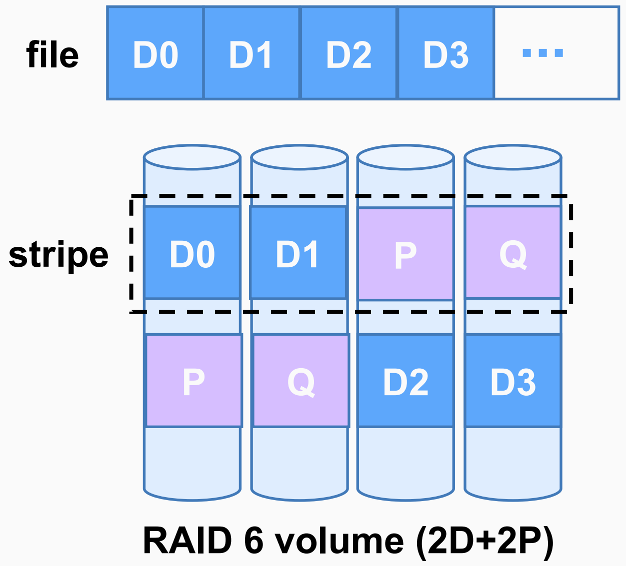
- Configurable Parity Algorithm
  - None (RAID 0/1)
  - XOR/P-Q Stripe (RAID 5/6)
  - Erasure-Codes: Reed Solomon or MDS Codes (more than 2 blocks of parity)

Design Background

Design Background

Distributed Data Placement

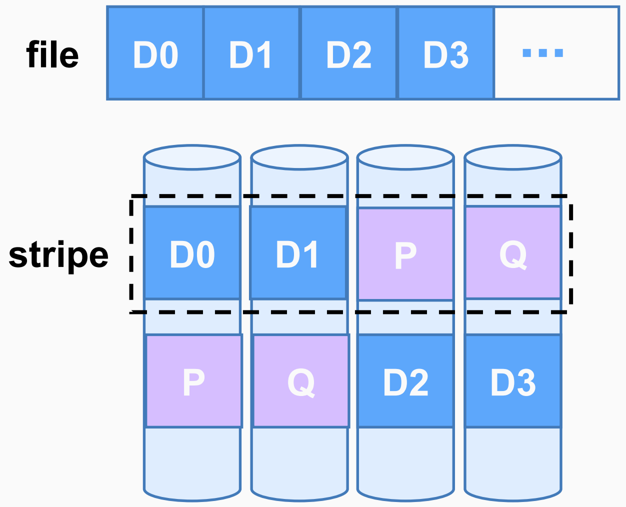
- Traditional RAID6 (2D+2P)
  - Dataset + parity is striped across all disks



Design Background

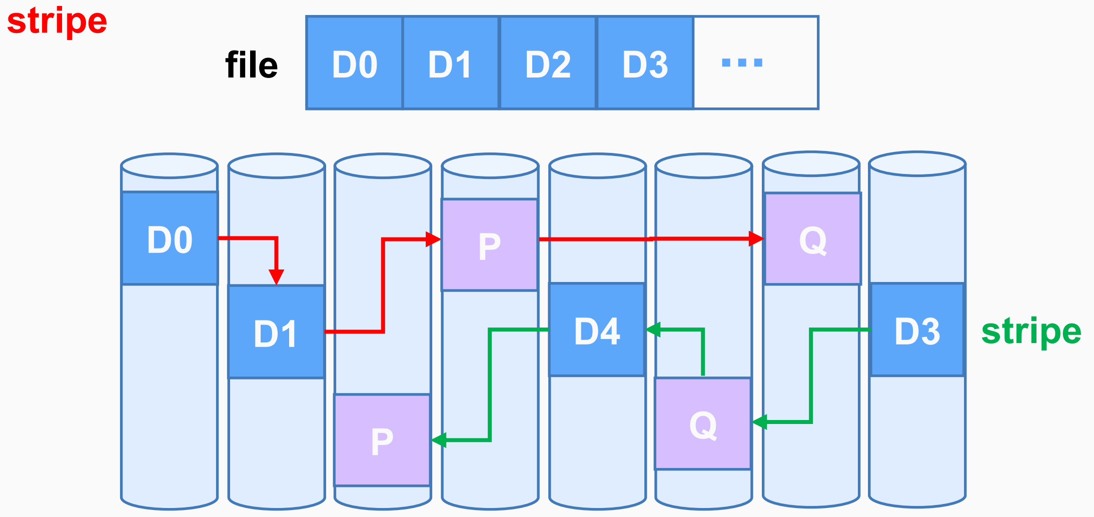
Distributed Data Placement

- Traditional RAID6 (2D+2P)
  - Dataset + parity is striped across all disks



RAID 6 volume (2D+2P)

- Declustered RAID (2D+2P)
  - Dataset + parity is distributed among a subset of disks



DP volume (2D+2P over 8 disks)

Design Background

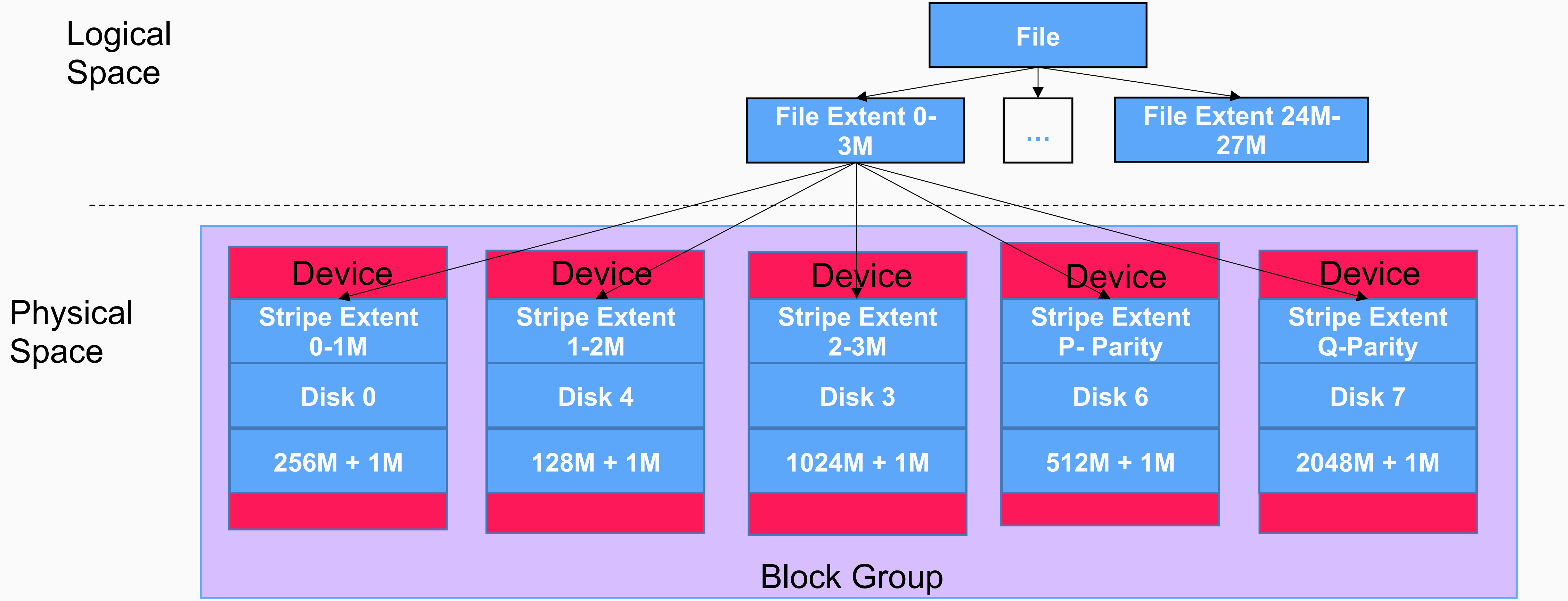
RAID Stripe Tree

- Can be seen as an inverse of the free space tree
  - Written after the data has reached the disks
  - Records the location (disk, LBA) of each sub-stripe
- Kind of RAID “journal”
  - Removes write hole (CoW)
  - Can be use for “nocow” as well
- Logical to physical addresses translation
  - Logical (start, length) tuple maps to N (disk, start) tuples

Design Background

RAID Stripe Tree

- Translate logical to physical addresses (3D + 2P)



Design Background

RAID Stripe Tree

- Keyed by logical, length
- Additional per file extent space consumption
  - N*16 Bytes
- Example 3D + 2P RAID
  - 5 * 16 Bytes = 80 Bytes stripe tree nodes
  - 51 Nodes per 4k sector

```
struct btrfs_key {
              .objectid = file_extent_logical,
              .type = BTRFS_RAID_STRIPE_EXTENT,
              .offset = file_extent_length,
};
```

```
struct btrfs_dp_stripe {
        /* array of RAID stripe extents this stripe is
         * comprised of
         */
        struct btrfs_stripe_extent extents[];
} __attribute__ ((__packed__));
```

```
struct btrfs_stripe_extent {
        /* btrfs device-id this raid extent lives on */
        __le64 devid;
        /* physical start address on the device */
        __le64 physical;
} __attribute__ ((__packed__));
```

Design Background

RAID Stripe Tree

```
struct btrfs_file_extent_item {
            __le64 generation;
            __le64 ram_bytes;
            __u8 compression;
            __u8 encryption;
            __le16 other_encoding;
            __u8 type;
            __le64 disk_bytenr;
            __le64 disk_num_bytes;
            __le64 offset;
            __le64 num_bytes;
} __attribute__ ((__packed__));
```

```
struct btrfs_key {
            .objectid = file_extent_logical,
            .type = BTRFS_RAID_STRIPE_EXTENT,
            .offset = file_extent_length,
};

struct btrfs_dp_stripe {
        /* array of RAID stripe extents this stripe is
         * comprised of
         */
        struct btrfs_stripe_extent extents[];
} __attribute__ ((__packed__));

struct btrfs_stripe_extent {
        /* btrfs device-id this raid extent lives on */
        __le64 devid;
        /* physical start address on the device */
        __le64 physical;
} __attribute__ ((__packed__));
```

Design Background

RAID Stripe Tree

```
struct btrfs_file_extent_item {
        __le64 generation;
        __le64 ram_bytes;
        __u8 compression;
        __u8 encryption;
        __le16 other_encoding;
        __u8 type;
        __le64 disk_bytenr;
        __le64 disk_num_bytes;
        __le64 offset;
        __le64 num_bytes;
} __attribute__ ((__packed__));
```

```
struct btrfs_key {
        .objectid = file_extent_logical,
        .type = BTRFS_RAID_STRIPE_EXTENT,
        .offset = file_extent_length,
};

struct btrfs_dp_stripe {
        /* array of RAID stripe extents this stripe is
         * comprised of
         */
        struct btrfs_stripe_extent extents[];
} __attribute__ ((__packed__));

struct btrfs_stripe_extent {
        /* btrfs device-id this raid extent lives on */
        __le64 devid;
        /* physical start address on the device */
        __le64 physical;
} __attribute__ ((__packed__));
```

Design Background

RAID Stripe Tree

```
struct btrfs_file_extent_item {
        __le64 generation;
        __le64 ram_bytes;
        __u8 compression;
        __u8 encryption;
        __le16 other_encoding;
        __u8 type;
        __le64 disk_bytenr;
        __le64 disk_num_bytes;
        __le64 offset;
        __le64 num_bytes;
} __attribute__ ((__packed__));
```

```
struct btrfs_key {
        .objectid = file_extent_logical,
        .type = BTRFS_RAID_STRIPE_EXTENT,
        .offset = file_extent_length,
};

struct btrfs_dp_stripe {
        /* array of RAID stripe extents this stripe is
         * comprised of
         */
        struct btrfs_stripe_extent extents[];
} __attribute__ ((__packed__));


struct btrfs_stripe_extent {
        /* btrfs device-id this raid extent lives on */
        __le64 devid;
        /* physical start address on the device */
        __le64 physical;
} __attribute__ ((__packed__));
```

Design Background

RAID Stripe Tree

```
struct btrfs_file_extent_item {
        __le64 generation;
        __le64 ram_bytes;
        __u8 compression;
        __u8 encryption;
        __le16 other_encoding;
        __u8 type;
        __le64 disk_bytenr;
        __le64 disk_num_bytes;
        __le64 offset;
        __le64 num_bytes;
} __attribute__ ((__packed__));
```

```
struct btrfs_key {
        .objectid = file_extent_logical,
        .type = BTRFS_RAID_STRIPE_EXTENT,
        .offset = file_extent_length,
};

struct btrfs_dp_stripe {
        /* array of RAID stripe extents this stripe is
         * comprised of
         */
        struct btrfs_stripe_extent extents[];
} __attribute__ ((__packed__));

struct btrfs_stripe_extent {
        /* btrfs device-id this raid extent lives on */
        __le64 devid;
        /* physical start address on the device */
        __le64 physical;
} __attribute__ ((__packed__));
```

Design Background

RAID Stripe Tree

Advantages

- Address translation
  - Scrub friendly
- RAID Journal
  - Ordered updates
  - Similar to how checksums are handled

Advantages

- No implicit connection needed
  - REQ_OP_ZONE_APPEND compatible
- Stronger reliability against device faults
  - M+K erasure code can be high

Design Background

RAID Stripe Tree

Disadvantages

- Additional Metadata
  - Especially if we also must do stripe tree entries for metadata
  - Merge consecutive and sequential on-disk stripe extents?

Design Background

Configurable Parity Algorithm

- Generates Parity or EC information
- Like how we handle compression
  - Do the math on data read/write
- But different to how we handle compression
  - Doesn’t modify the actual data but adds data

Current Status

9/14/22

Current Status

Where are at the moment?

- Data RAID1 and RAID1 implemented
  - Metadata doesn’t use REQ_OP_ZONE_APPEND
    - Already working out-of-the-box
  - Data writes are recorded in raid-stripe-tree

Current Status

- Boilerplate mkfs creating an FS with empty RAID stripe tree

```
rapido1:/# mkfs.btrfs -R raid-stripe-tree -d raid1 -m raid1 /dev/nullb0 /dev/nullb1
btrfs-progs v5.16.1
See http://btrfs.wiki.kernel.org for more information.

Zoned: /dev/nullb0: host-managed device detected, setting zoned feature
Resetting device zones /dev/nullb0 (100 zones) ...
NOTE: several default settings have changed in version 5.15, please make sure
      this does not affect your deployments:
      - DUP for metadata (-m dup)
      - enabled no-holes (-O no-holes)
      - enabled free-space-tree (-R free-space-tree)

Resetting device zones /dev/nullb1 (100 zones) ...
Label:              (null)
UUID:               566527dd-7a0f-49f3-9e94-a21f6e9f9132
Node size:          16384
Sector size:        4096
Filesystem size:    25.00GiB
Block group profiles:
  Data:             RAID1           128.00MiB
  Metadata:         RAID1           128.00MiB
  System:           RAID1           128.00MiB
SSD detected:       yes
Zoned device:       yes
  Zone size:        128.00MiB
Incompat features:  extref, skinny-metadata, no-holes, zoned
Runtime features:   free-space-tree, raid-stripe-tree
Checksum:           crc32c
Number of devices:  2
Devices:
   ID        SIZE  PATH
    1    12.50GiB  /dev/nullb0
    2    12.50GiB  /dev/nullb1

[   74.248386] BTRFS: device fsid 566527dd-7a0f-49f3-9e94-a21f6e9f9132 devid 1 transid 7 /dev/nullb0 scanned)
[   74.252388] BTRFS: device fsid 566527dd-7a0f-49f3-9e94-a21f6e9f9132 devid 2 transid 7 /dev/nullb1 scanned)
rapido1:/#
```

Current Status

- Tree-dump (on RAID1)

```
rapido1:/# mount /dev/nullb0 /mnt/test/
[   55.901581] BTRFS info (device nullb0): flagging fs with big metadata feature
[   55.902835] BTRFS info (device nullb0): using free space tree
[   55.903831] BTRFS info (device nullb0): has skinny extents
[   55.941664] BTRFS info (device nullb0): host-managed zoned block device /dev/nullb0, 100 zones of 1342177s
[   55.943480] BTRFS info (device nullb0): host-managed zoned block device /dev/nullb1, 100 zones of 1342177s
[   55.945249] BTRFS info (device nullb0): zoned mode enabled with zone size 134217728
[   55.946949] BTRFS info (device nullb0): enabling ssd optimizations
[   55.949492] BTRFS info (device nullb0): checking UUID tree
rapido1:/# xfs_io -fc "pwrite 0 1M" -c fsync /mnt/test/test
wrote 1048576/1048576 bytes at offset 0
1 MiB, 256 ops; 0.0009 sec (1.058 GiB/sec and 277356.4464 ops/sec)
rapido1:/# btrfs inspect-internal dump-tree -t raid_stripe /dev/nullb1
btrfs-progs v5.16.1
raid stripe tree key (RAID_STRIPE_TREE ROOT_ITEM 0)
leaf 805699584 items 9 free space 15770 generation 8 owner RAID_STRIPE_TREE
leaf 805699584 flags 0x1(WRITTEN) backref revision 1
checksum stored 3a0f6a6500000000000000000000000000000000000000000000000000000000
checksum calced 3a0f6a6500000000000000000000000000000000000000000000000000000000
fs uuid 0a922710-c894-4772-984c-5e90dbe334e2
chunk uuid 807c6a1b-25a9-441a-8f67-52a14d0aebdf
        item 0 key (939524096 RAID_STRIPE_KEY 126976) itemoff 16251 itemsize 32
                        stripe 0 devid 1 offset 939524096
                        stripe 1 devid 2 offset 536870912
        item 1 key (939651072 RAID_STRIPE_KEY 126976) itemoff 16219 itemsize 32
                        stripe 0 devid 1 offset 939651072
                        stripe 1 devid 2 offset 536997888
        item 2 key (939778048 RAID_STRIPE_KEY 126976) itemoff 16187 itemsize 32
                        stripe 0 devid 1 offset 939778048
                        stripe 1 devid 2 offset 537124864
        item 3 key (939905024 RAID_STRIPE_KEY 126976) itemoff 16155 itemsize 32
                        stripe 0 devid 1 offset 939905024
                        stripe 1 devid 2 offset 537251840
        item 4 key (940032000 RAID_STRIPE_KEY 126976) itemoff 16123 itemsize 32
                        stripe 0 devid 1 offset 940032000
                        stripe 1 devid 2 offset 537378816
        item 5 key (940158976 RAID_STRIPE_KEY 126976) itemoff 16091 itemsize 32
                        stripe 0 devid 1 offset 940158976
                        stripe 1 devid 2 offset 537505792
        item 6 key (940285952 RAID_STRIPE_KEY 126976) itemoff 16059 itemsize 32
                        stripe 0 devid 1 offset 940285952
                        stripe 1 devid 2 offset 537632768
        item 7 key (940412928 RAID_STRIPE_KEY 126976) itemoff 16027 itemsize 32
                        stripe 0 devid 1 offset 940412928
                        stripe 1 devid 2 offset 537759744
        item 8 key (940539904 RAID_STRIPE_KEY 32768) itemoff 15995 itemsize 32
                        stripe 0 devid 1 offset 940539904
                        stripe 1 devid 2 offset 537886720
total bytes 26843545600
bytes used 1245184
uuid 0a922710-c894-4772-984c-5e90dbe334e2
rapido1:/#
```

Current Status

- Fsck On An Non-Empty RAID Filesystem

```
rapido1:/# btrfs check --check-data-csum --progress /dev/nullb0
Opening filesystem to check...
Checking filesystem on /dev/nullb0
UUID: 0a922710-c894-4772-984c-5e90dbe334e2
[1/7] checking root items                      (0:00:00 elapsed, 26 items checked)
[2/7] checking extents                         (0:00:00 elapsed, 22 items checked)
[3/7] checking free space tree                 (0:00:00 elapsed, 5 items checked)
[4/7] checking fs roots                        (0:00:00 elapsed, 2 items checked)
[5/7] checking csums against data              (0:00:00 elapsed, 2 items checked)
[6/7] checking root refs                       (0:00:00 elapsed, 3 items checked)
[7/7] checking quota groups skipped (not enabled on this FS)
found 1245184 bytes used, no error found
total csum bytes: 1024
total tree bytes: 196608
total fs tree bytes: 32768
total extent tree bytes: 16384
btree space waste bytes: 166678
file data blocks allocated: 1048576
 referenced 1048576
rapido1:/#
```

Thanks

Questions?

TM