Log Space Maps

An optimization for ZFS Allocation Performance

Serapheim Dimitropoulos | Delphix

Background

Background - ZFS Writes

- ZFS **records dirty data in memory** that come from the write() syscall
- **Periodically syncs** dirty data to disk
- We refer to each **sync cycle** as a **TXG** (aka **Transaction Group**)

TXGs are used as **logical timestamps** to represent time within ZFS

Background - Syncing Data

ZFS Copy-On-Write

- Never Overwrite! Just mark old segment as free and allocate elsewhere!
- True even for metadata

Syncing Passes

- 1st sync pass: we sync the dirty user data
- Subsequent ones: we update our ZFS metadata on-disk

Background - Space Allocation

Each **vdev** is divided into ~200 equal regions we call ***metaslabs***

Each metaslab keeps track of free space in its assigned region

We represent metaslabs by mainly using two structures:

- ***Range Trees*** (in-memory)
- ***Space maps*** (on-disk)

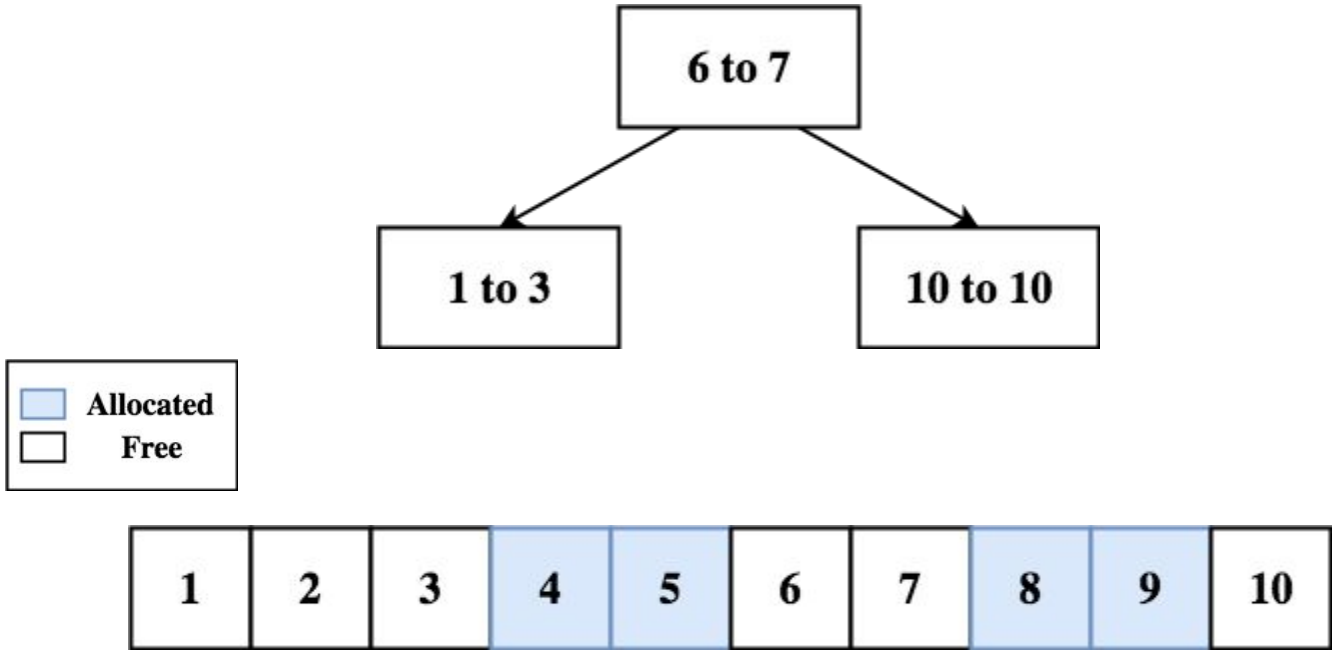
Every time we ***load a metaslab***, we read its space map from disk and load its entries into a range tree in memory

We load metaslabs to allocate space from them

Background - Range Tree

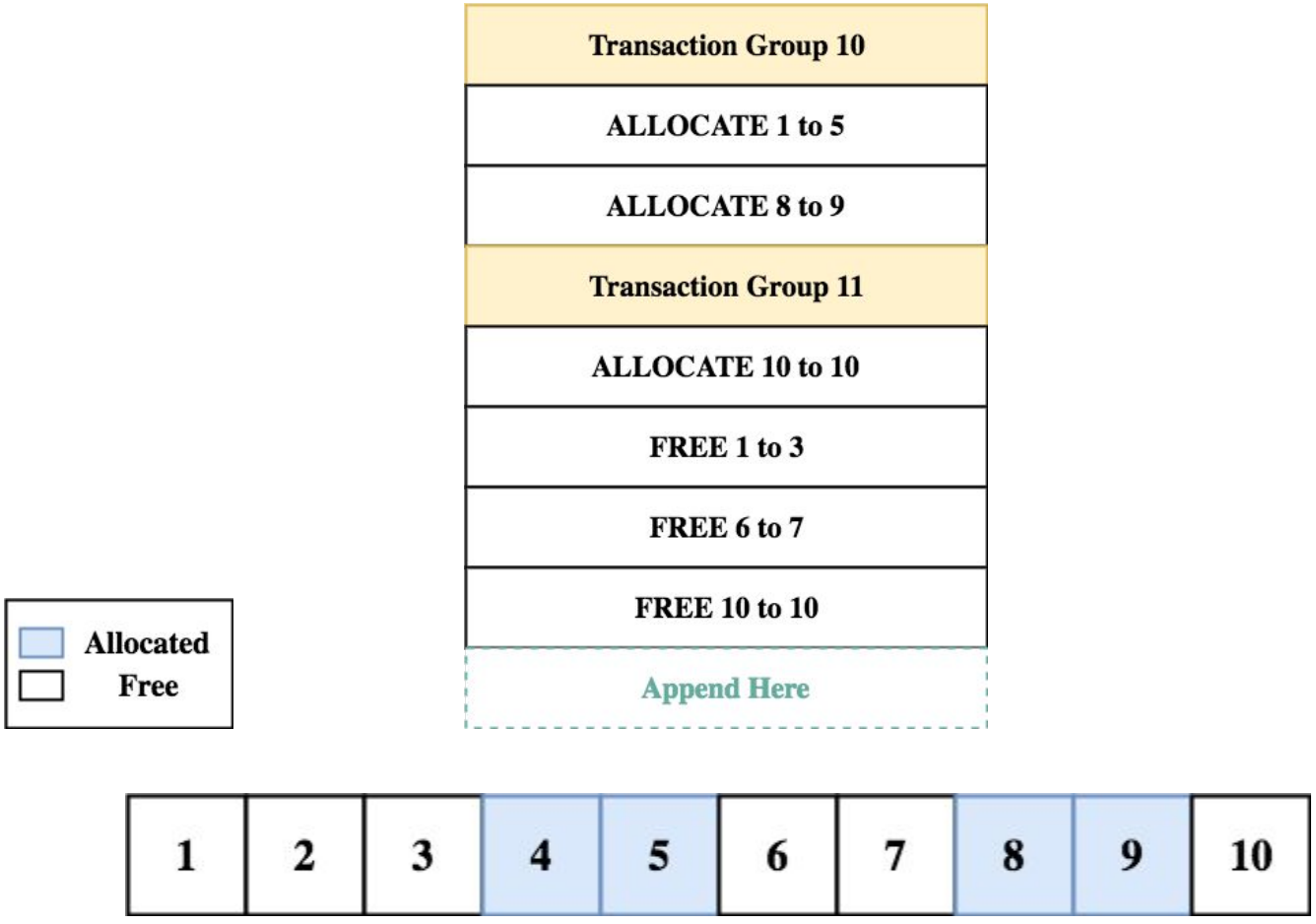
An **AVL tree** where each node represents a **free segment**

Its segments are sorted by offset or size



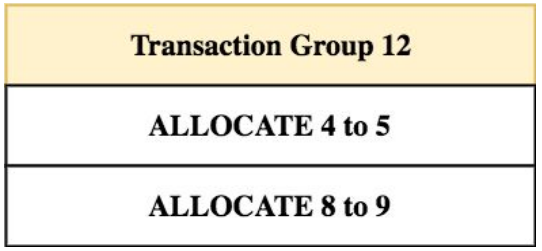
Background - Space Maps

An **on-disk append-only log** of allocations and frees

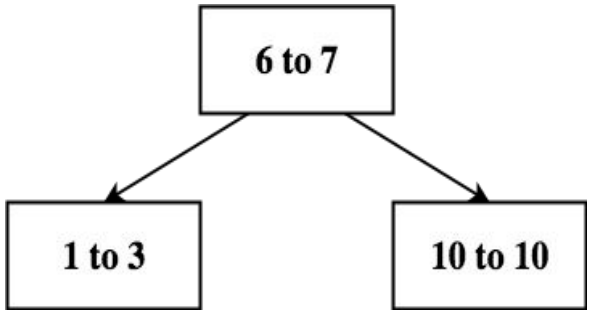


Background - Space Map Condensing

Reduces the metaslab's space overhead on-disk and thus its loading time



*Any segment not mentioned in the spacemap is assumed to be free



Background - Recap

- We sync writes and other changes to disk in **TXG**s
- Within a TXG we have multiple **sync passes**
  - **1st sync pass**: syncs **user data** and dataset changes
  - **Subsequent passes**: update ZFS **metadata**

- We track space using **metaslabs**
- A metaslab's free space is represented by:
  - A **Range Tree** in memory
  - A **Space Map** on disk

- We can **free space from any** metaslab
- but we can **allocate space only from loaded** ones

So what is the problem?

Dynamic Allocation Behavior

- After some time we've touched the whole pool
- Allocate from a few loaded metaslabs in each vdev
- Frees are scattered across all parts of the pool
  - Freeing blocks from most metaslabs
  - Thus, appending to most metaslab space maps

Space Map I/Os

Each TXG per VDEV:

- Append to almost all metaslab space maps (~200 I/Os)
  - [assume data fits block size = 4K]
- Update space map indirect blocks (~400 additional I/Os)
  - [assume 2 levels of indirection]
- 2 extra copies for metadata redundancy (3x the I/Os)

That is **~1800 I/Os per VDEV per TXG**

Is this actually a problem?

DTrace output for a single TXG

```
2017 Oct 18 16:37:24 txg 71463672 is next
time since last sync
      |  written by sync
      |       | syncing time (% pass 1)
      |       |          |          |     write rate
      |       |          |          |    while syncing
      |       |          |          |       |
      v       v          v          v       v
    0ms  3205MB in  4479ms (71% p1) 715MB/s
```

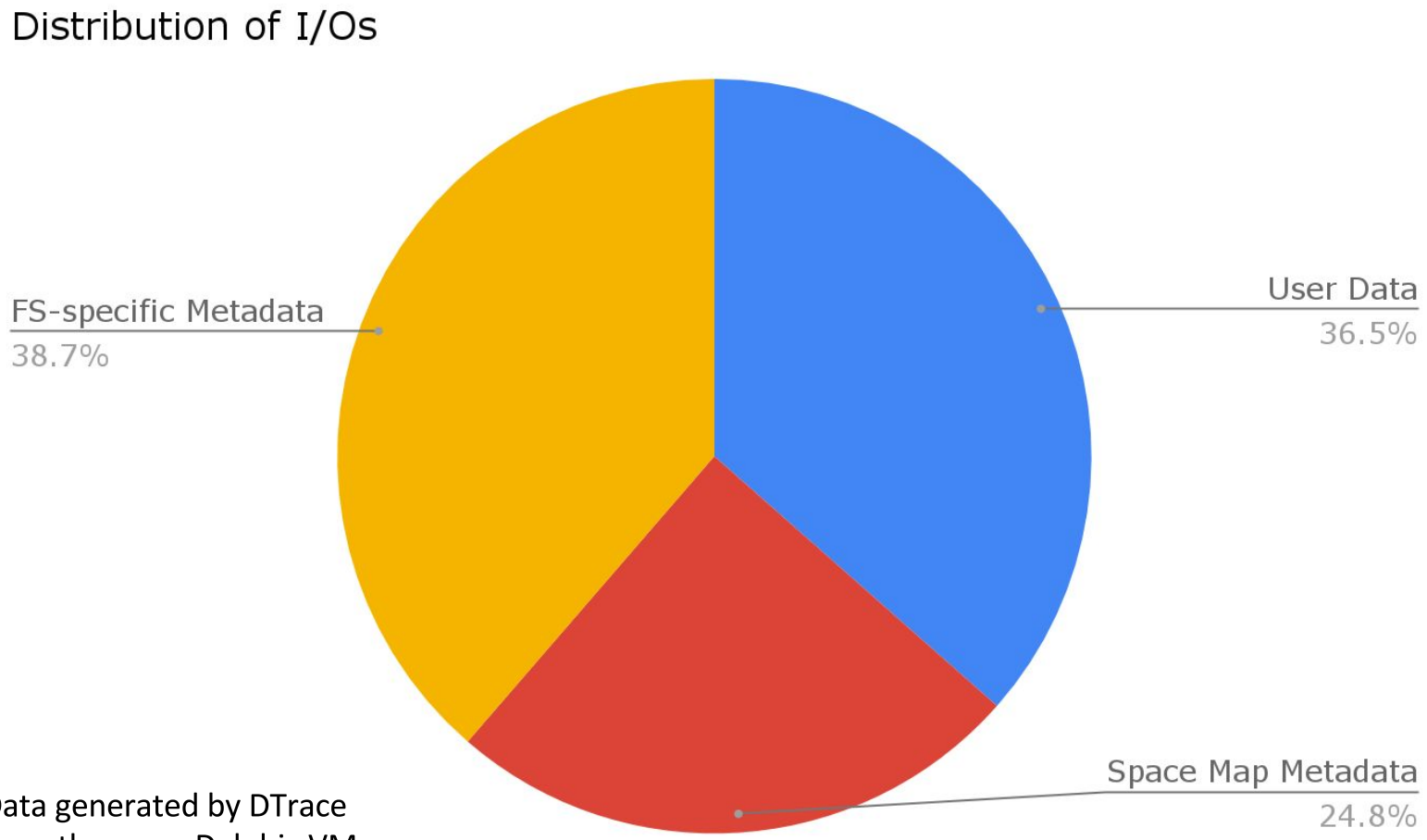
From a Delphix VM with a 30TB pool

Is this actually a problem?

```
2017 Oct 18 16:37:24 txg 71463672 is next
time since last sync
      |  written by sync
      |       | syncing time (% pass 1)
      |       |           |         |     write rate
      |       |           |         |    while syncing
      |       |           |         |       |
      v       v           v         v       v
     0ms   3205MB in   4479ms (71% p1) 715MB/s
     0ms   2623MB in   4010ms (66% p1) 654MB/s
     0ms   2138MB in   4090ms (77% p1) 522MB/s
     0ms   2447MB in   3579ms (69% p1) 683MB/s
     0ms   2204MB in   3705ms (72% p1) 594MB/s
     0ms   2210MB in   3621ms (66% p1) 610MB/s
     0ms   2274MB in   4049ms (66% p1) 561MB/s
     0ms   2677MB in   3932ms (74% p1) 681MB/s
     0ms   2358MB in   4414ms (75% p1) 534MB/s
     0ms   2793MB in   5826ms (71% p1) 479MB/s
     0ms   3461MB in   5810ms (62% p1) 595MB/s
     0ms   3427MB in   6037ms (84% p1) 567MB/s
     0ms   3834MB in   7225ms (78% p1) 530MB/s
     0ms   3644MB in   5505ms (79% p1) 661MB/s
     0ms   2810MB in   5363ms (66% p1) 523MB/s
```

20% to 40% of sync time spent updating space accounting after sync pass 1 ?!?

Is this actually a problem?



Data generated by DTrace
from the same Delphix VM

So what can we do about this?

Decrease # of Metaslabs?

Less metaslabs means fewer space maps and therefore **fewer I/Os**

It also means that each metaslab will represent a larger region on disk. This could lead to two issues:

1. Each metaslab **takes longer to be loaded** in memory from disk
2. Loaded metaslabs can take up **too much memory**

Increase Block Size?

Increasing block size and fitting more entries within an I/O means **fewer I/Os**
It should also **speed up the loading time**

Unfortunately, we tend to **write a lot to a few metaslabs**
and **a little to the rest of them**

Thus, increasing the block size would mean a lot of **wasted bandwidth**

Idea!

Don’t flush every modified metaslab every txg!

Keep changes in memory until flushed in 2 range trees

- unflushed allocs and unflushed frees

Flush 1 metaslab per TXG (round-robin fashion)

- Flush many TXGs worth of changes to 1 space map per TXG

To **load a metaslab**:

1. read its space map from disk
2. Apply unflushed allocs and frees

What if we crash..

… **and the unflushed changes in memory are lost?**

Every TXG we create a new space map which logs all allocs & frees pool-wide [aka a **Log Space Map**]

After a crash:

- Read log space maps and reconstruct unflushed allocs & frees

Recap of Main Idea

- Before -

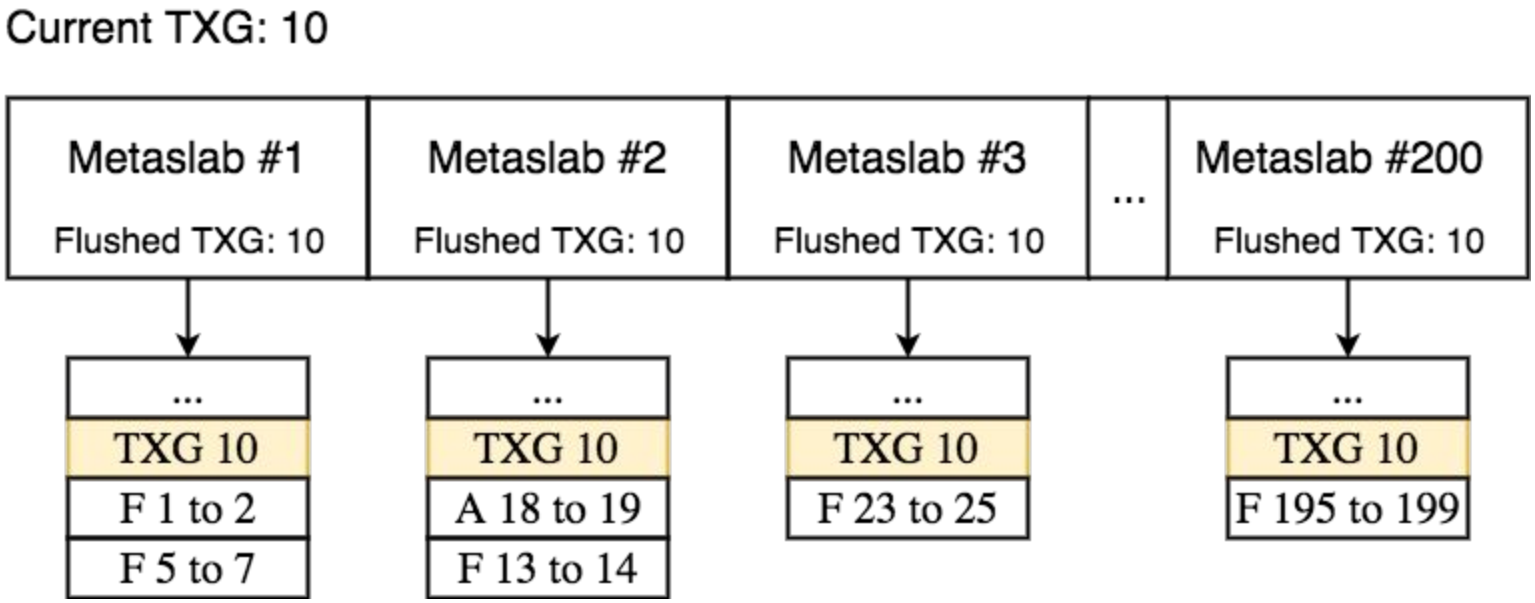
All allocations and frees are **written to the metaslab's space map** every TXG

- After -

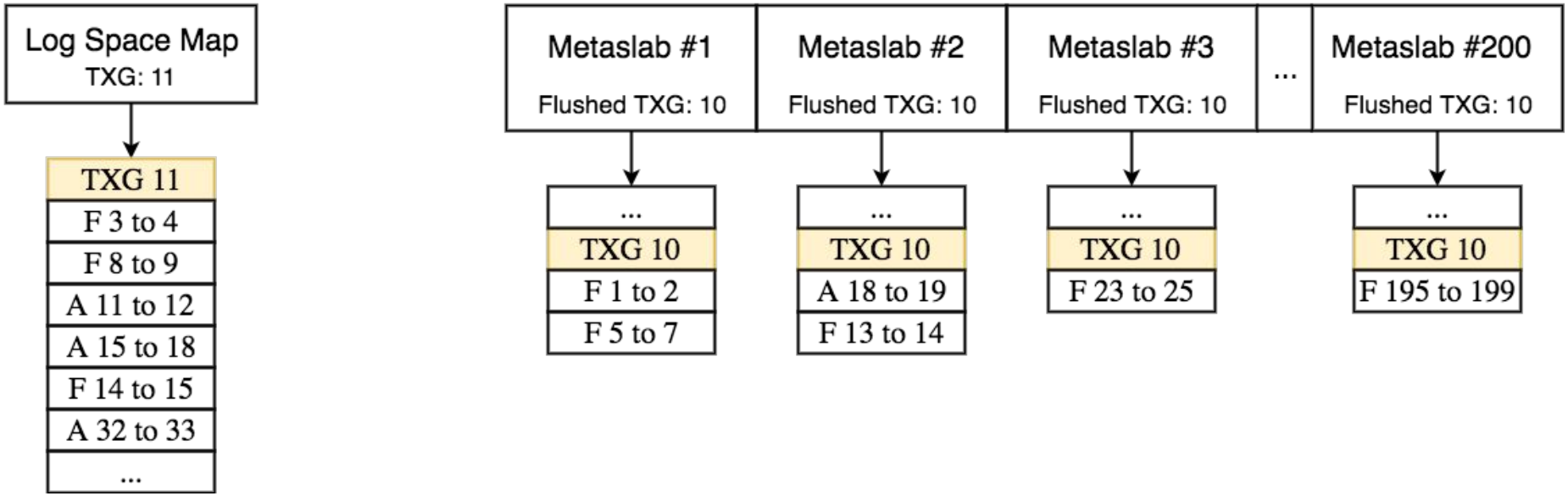
All allocations and frees are **written to the log space map** and then transferred to their respective **unflushed trees**

When the metaslab is later **flushed**, we **write the unflushed trees** to the **metaslab's space map**

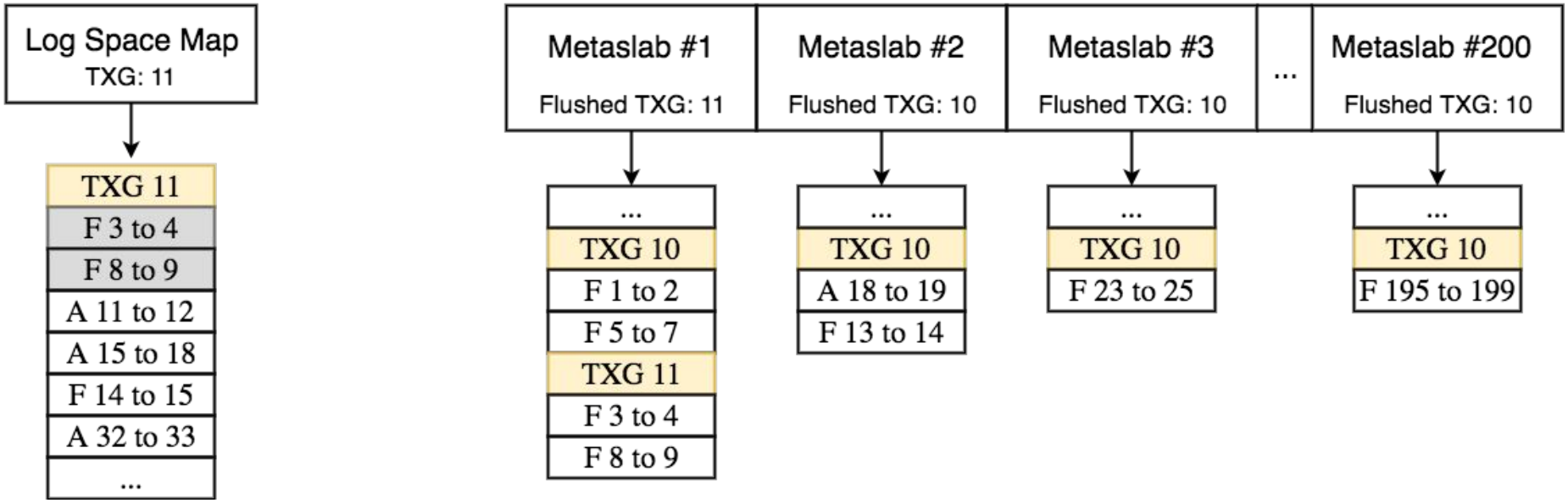
Example

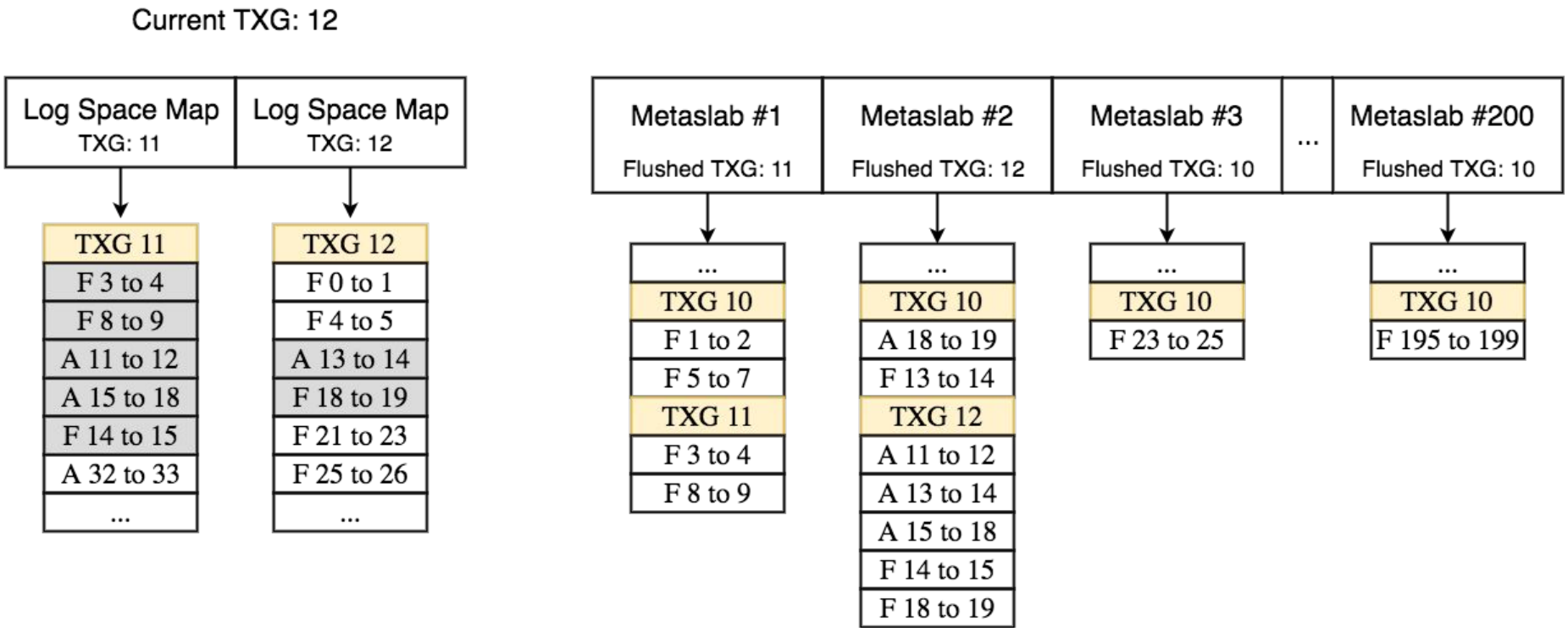

Current TXG: 10
Metaslab #1
Flushed TXG: 10
Metaslab #2
Flushed TXG: 10
Metaslab #3
Flushed TXG: 10
...
Metaslab #200
Flushed TXG: 10
...
TXG 10
F 1 to 2
F 5 to 7
...
TXG 10
A 18 to 19
F 13 to 14
...
TXG 10
F 23 to 25
...
TXG 10
F 195 to 199

Current TXG: 11 (Before Flush)

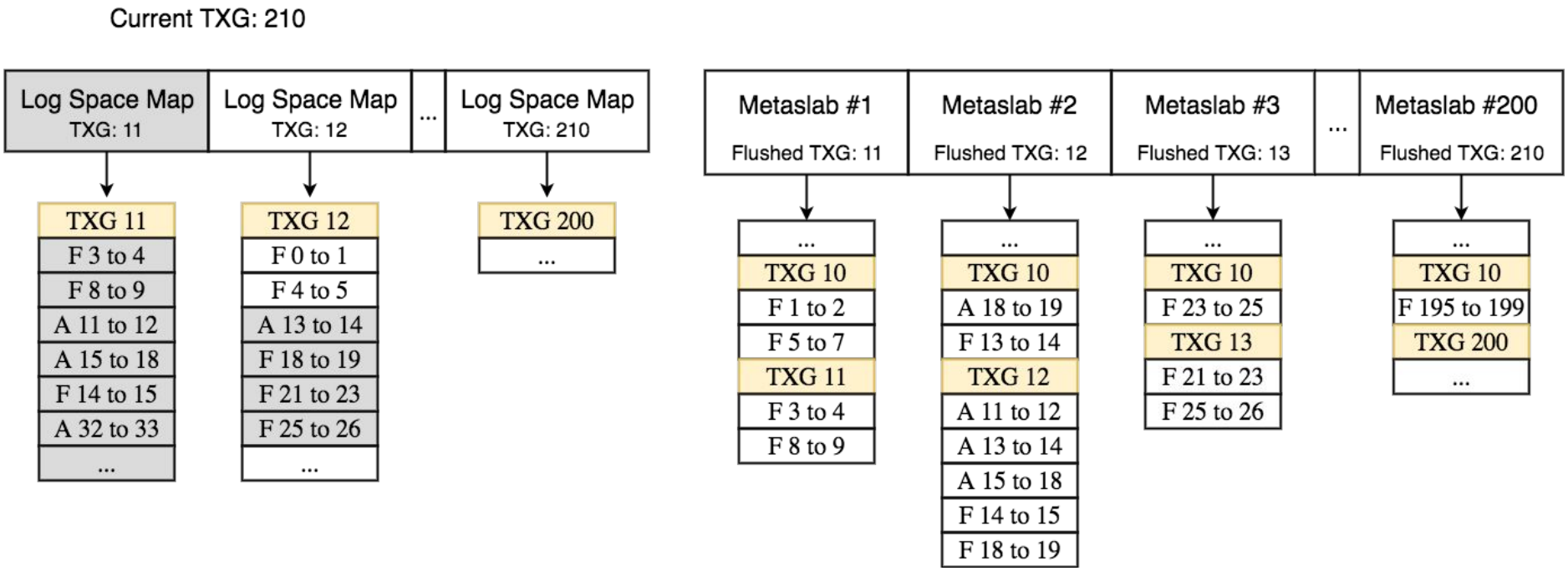


Current TXG: 11 (After Flush)




Current TXG: 12
Log Space Map
TXG: 11
Log Space Map
TXG: 12
TXG 11
F 3 to 4
F 8 to 9
A 11 to 12
A 15 to 18
F 14 to 15
A 32 to 33
...
TXG 12
F 0 to 1
F 4 to 5
A 13 to 14
F 18 to 19
F 21 to 23
F 25 to 26
...
Metaslab #1
Flushed TXG: 11
Metaslab #2
Flushed TXG: 12
Metaslab #3
Flushed TXG: 10
...
Metaslab #200
Flushed TXG: 10
...
TXG 10
F 1 to 2
F 5 to 7
TXG 11
F 3 to 4
F 8 to 9
...
TXG 10
A 18 to 19
F 13 to 14
TXG 12
A 11 to 12
A 13 to 14
A 15 to 18
F 14 to 15
F 18 to 19
...
TXG 10
F 23 to 25
...
TXG 10
F 195 to 199

Some TXGs later...



Problem - Obsolete Logs

Log Space Maps whose entries have been flushed to the metaslab space maps

- No longer needed
- Take up space
- Increase crash recovery time

Dealing With Obsolete Logs

Keep a list of the metaslabs sorted by the TXG that they were last flushed

Keep a list of the logs sorted by the TXG of the changes they contain

Each TXG:

- Flush oldest flushed metaslab (first in metaslab list)
- Put it back in the list (at the end - most recently flushed)
- Look at flushed TXG of the new oldest flushed metaslab
- Destroy all logs older than that

Space Map Blocksize

4KB (currently)

- Made sense for a lot of small appends
- Bad~OK for loading metaslab (depending on metaslab size)
- Bad for reading log space maps (crash recovery)

128KB (proposed)

- Log space maps project - few large appends
- Better for metaslab loading
- Great for reading log space maps (crash recovery)

Pool Export/Import

Flush all metaslabs at export (and thus destroy all logs)

- so we don’t have to read them during pool import

This saves time as:

- Writes are faster than reads in this context
- We skip reading obsolete entries during import
- Just flush all unflushed range trees

Performance Evaluation

Work In Progress

Reconsidering Flushing Algorithm

Problems with flushing 1 metaslab per TXG

- Unflushed changes may take up too much memory (range trees)
- Long reconstruction time due to many long log space maps

How many metaslabs to flush each TXG?

Things to consider:

- Reconstruction time during crash
- Memory usage of unflushed changes
- Smooth and consistent behavior
- Bandwidth utilization when flushing
  - Want to append big chunks to each space map

Resources

Watch/Read:

- Matt Ahrens on OpenZFS Space Allocation (link)
- Jeff Bonwick on Space Maps (link)

Ask:

- Slack (#OpenZFS)
- Tweet @OpenZFS or @AmazingDim
- Email me!

Questions?

Thank you for your time

Serapheim Dimitropoulos

serapheim@delphix.com

sdimitro.github.io

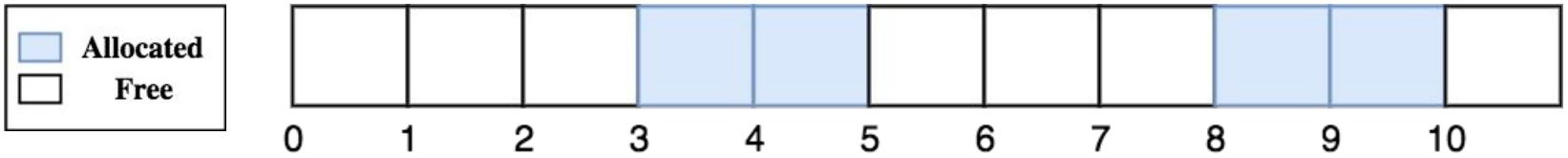
Appendix - Space Map Scalability


SPACE MAPS
HD
HISTORY.COM
imgflip.com

Space Map Encoding

```
/*
 * debug entry
 *
 *      1      3          10                      50
 *    ,---+--------+------------+-------------------------------.
 *    | 1 | action |  syncpass  |          txg (lower bits)     |
 *    `---+--------+------------+-------------------------------'
 *     63  62     60 59        50 49                            0
 *
 *
 * non-debug entry
 *
 *      1               47                      1           15
 *    ,-----------------------------------------------------------.
 *    | 0 |   offset (sm_shift units)     | type |       run      |
 *    `-----------------------------------------------------------'
 *     63  62                             17   16   15            0
 */
```

ALLOC - TXG 10 - sync pass 1
ALLOC | offset = 0 | run = 5
ALLOC | offset = 10 | run = 1
ALLOC - TXG 11 - sync pass 1
ALLOC | offset = 8 | run = 2
FREE - TXG 11 - sync pass 1
FREE | offset = 0 | run = 3
FREE | offset = 10 | run = 1



Space Map Encoding

```
/*
 * debug entry
 *
 *    1      3          10                      50
 *  .---+--------+------------+---------------------------------.
 *  | 1 | action |  syncpass  |        txg (lower bits)         |
 *  '---+--------+------------+---------------------------------'
 *   63  62    60 59        50 49                               0
 *
 *
 * non-debug entry
 *
 *    1               47                    1           15
 *  .-------------------------------------------------------------.
 *  | 0 |   offset (sm_shift units)     | type |       run       |
 *  '-------------------------------------------------------------'
 *   63  62                            17   16   15              0
 */
```

Using 512-byte addressable storage each space map entry:

1. can only represent a **16 MB region** [512 << 15 = 16MB]
2. can address a **64 PB offset** [512 << 47 = 64PB].

[1] makes storing large regions of space very inefficient.

[2] limits our ability to address very large address spaces/vdevs.

Space Map Encoding (cont.)

This matters more now as we want to use space maps to address whole vdevs besides individual metaslabs.

Examples of doing this already exist:

- Device Removal
- Storage Pool Checkpoint

In addition, in the Log Space Map project a log space map entry can represent a segment anywhere in the whole pool.

New Space Map Encoding!

```
/*
 * debug entry
 *
 *     2     2        10                        50
 *  +-----+-----+------------+-----------------------------------+
 *  | 1 0 | act |  syncpass  |        txg (lower bits)           |
 *  +-----+-----+------------+-----------------------------------+
 *   63 62 61 60 59        50 49                                 0
 *
 *
 * one-word entry
 *
 *    1               47                    1           15
 *  +-------------------------------------------------------------+
 *  | 0 |   offset (sm_shift units)     | type |       run       |
 *  +-------------------------------------------------------------+
 *   63  62                            16   15   14               0
 *
 *
 * two-word entry
 *
 *     2     2               36                        24
 *  +-----+-----+---------------------------+-------------------+
 *  | 1 1 | pad |            run            |       vdev        |
 *  +-----+-----+---------------------------+-------------------+
 *   63 62 61 60 59                         24 23                0
 *
 *     1                            63
 *  +------+-------------------------------------------------------+
 *  | type |                       offset                          |
 *  +------+-------------------------------------------------------+
 *     63   62                                                     0
 */
```

New Space Map Encoding!

It is compatible with the old one.

It includes a new field that specifies the VDEV that the segment belongs to.

Using 512-byte addressable storage each space map entry:

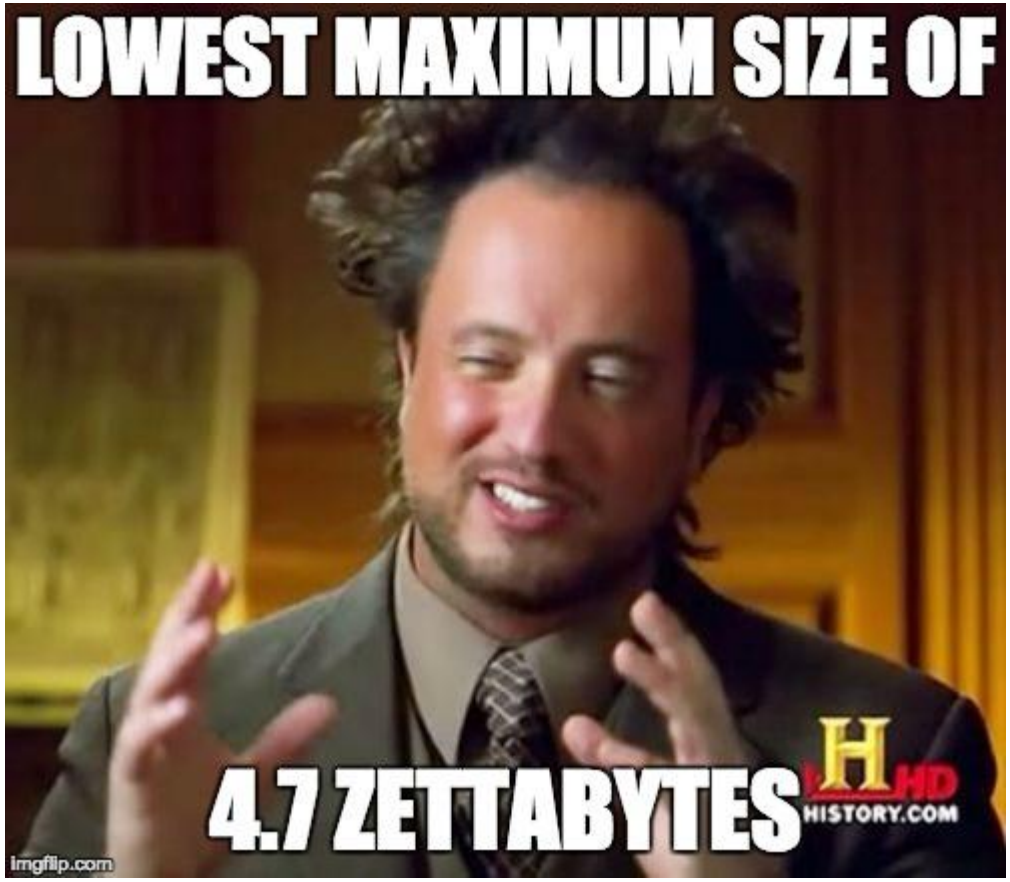
1. can represent a **35 TB region** [512 << 36 = 35 TB]
2. can address a **~4.7 ZB offset** [512 << 63 = 4.722… ZB].

It includes padding for any future changes.

In metaslab and vdev-wide space maps:

- Use single-word entry for segments less than 16 MB
- Use double-word for anything else

Next time you hear about Space Maps



Questions?

Thank you for your time

Serapheim Dimitropoulos

serapheim@delphix.com

sdimitro.github.io

Appendix - Condensing Unloaded Metaslabs

Problem - Long Metaslab Space Maps

We **only condense** space maps of **loaded metaslabs**.

What if a **metaslab** has been **flushed** (i.e. we've appended to its space map) **many times** but it **hasn't been loaded** for a while?

Its **space map may get too long** thus **increasing the metaslab's loading time**.

Dealing With Long Metaslab Space Maps

When we flush a metaslab and we see that it is not loaded we **check if**:

- The metaslab **has at least ~4 blocks**
- **Condensing will actually shrink** the space map

If both of the above are satisfied then we kick of **a thread that preloads the metaslab** so it can get condensed in a future TXG.

The **thread runs asynchronously** so the loading time won't affect any other operations like syncing changes.

Questions?

Thank you for your time

Serapheim Dimitropoulos

serapheim@delphix.com

sdimitro.github.io