How ZFS Snapshots Really Work

Matt Ahrens

BSDCAN 2019

DELPHIX

What are snapshots?

- Store an old “copy” of the data
- “Oops” recovery
- Malware recovery
- Replication with zfs send/receive

How to use snapshots

```
zfs snapshot pool/fs@snap
zfs snapshot -r pool@snap
zfs destroy pool/fs@snap
zfs send -i @oldsnap pool/fs@newsnap | \
    ssh … zfs receive …
zfs get … pool/fs@snap
```

How to use snapshots

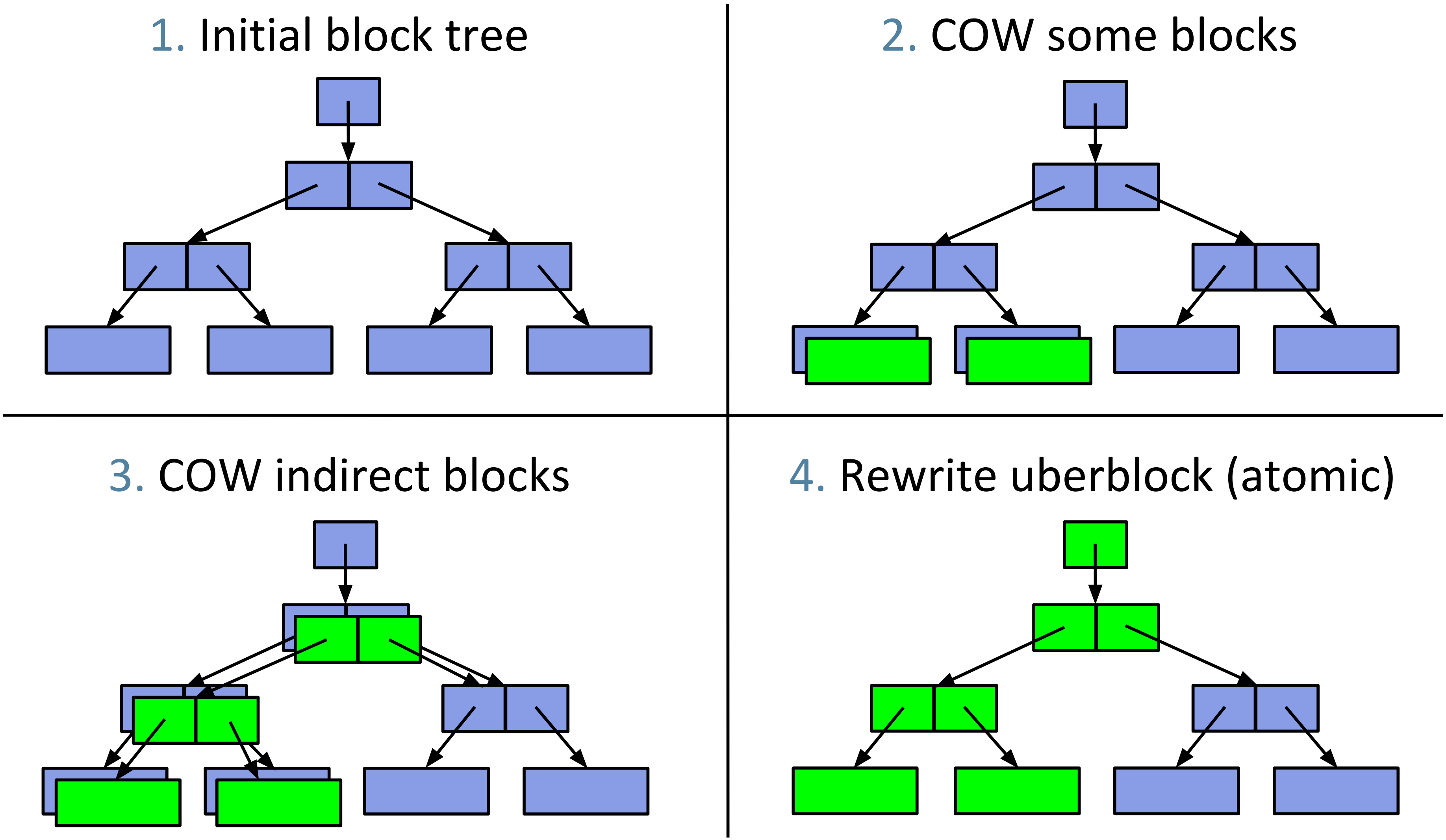
1. Take a snapshot of every filesystem every hour

   (8700 snapshots per filesystem per year)

2. …
3. Wonder where all your space went

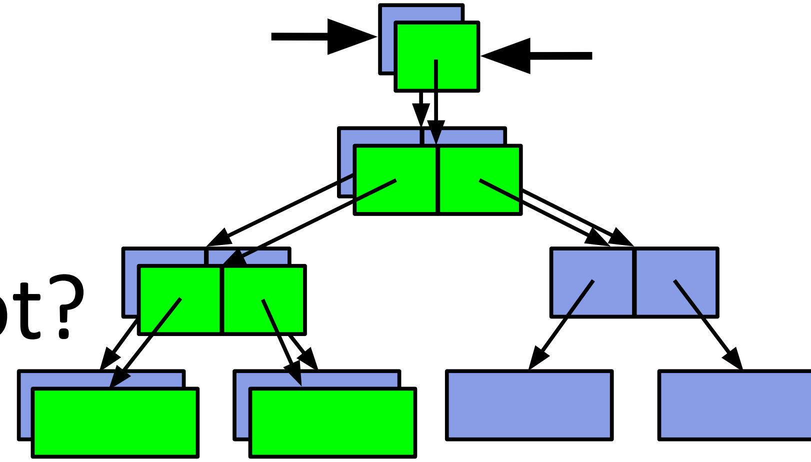
How do snapshots work?

Copy-On-Write Transaction Groups (TXG's)

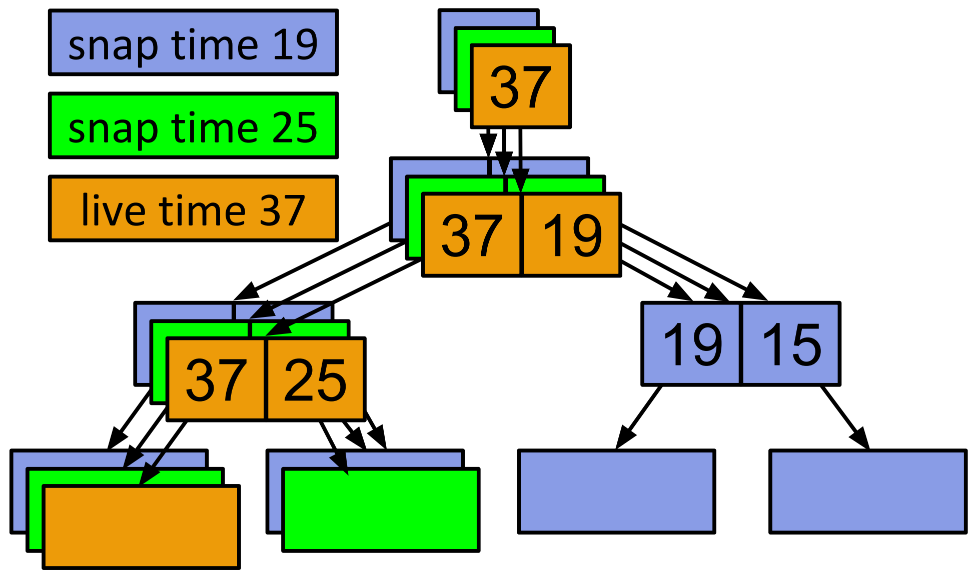


ZFS Snapshots

- How to create snapshot?
  - Save the root block



- When block is removed, can we free it?
  - Use BP’s birth time
  - If birth > prevsnap
    - Free it

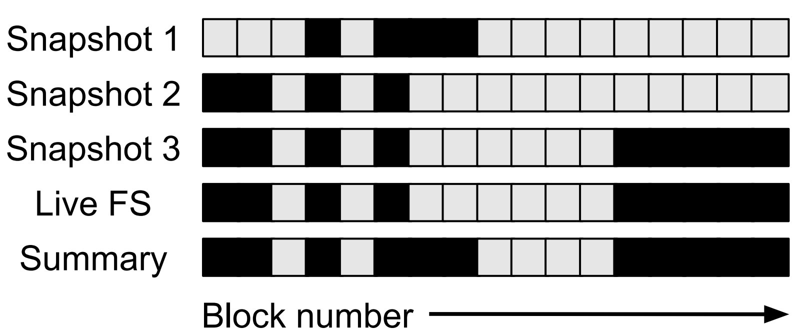


- When delete snapshot, what to free?
  - Find unique blocks - Tricky!

Trickiness will be worth it!

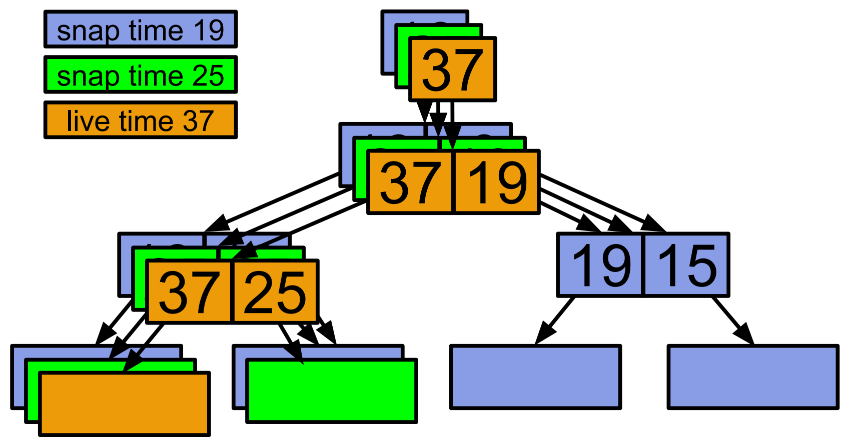
Per-Snapshot Bitmaps

- Block allocation bitmap for every snapshot
  - O(N) per-snapshot space overhead
  - Limits number of snapshots
- O(N) create, O(N) delete, O(N) incremental
  - Snapshot bitmap comparison is O(N)
  - Generates unstructured block delta
  - Requires some prior snapshot to exist



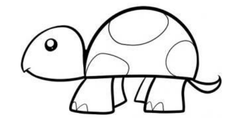
ZFS Birth Times

- Each block pointer contains child's birth time
  - O(1) per-snapshot space overhead
  - Unlimited snapshots
- O(1) create, O(Δ) delete, O(Δ) incremental
  - Birth-time-pruned tree walk is O(Δ)
  - Generates semantically rich object delta
  - Can generate delta since any point in time

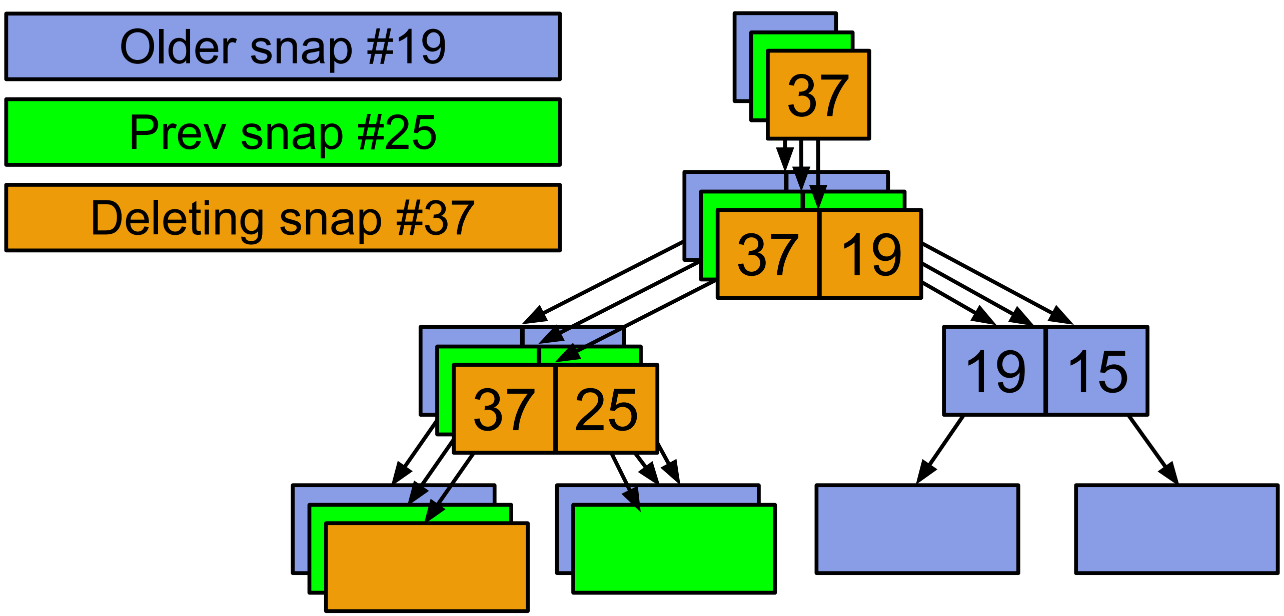


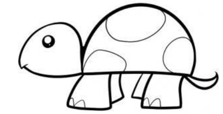
Snapshot Deletion

- Free unique blocks (ref'd only by this snap)
- Optimal algo: O(# blocks to free)
  - And # blocks to read from disk << # blocks to free
- Block lifetimes are contiguous
  - AKA "there is no afterlife"
  - Unique = not ref'd by prev or next (ignore others)

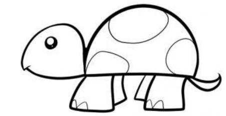
Snapshot Deletion ()

- Traverse tree of blocks
- Birth time <= prev snap?
  - Ref'd by prev snap; do not free.
  - Do not examine children; they are also <= prev



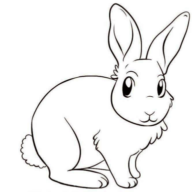
Snapshot Deletion ()

- Traverse tree of blocks
- Birth time <= prev snap?
  - Ref'd by prev snap; do not free.
  - Do not examine children; they are also <= prev
- Find BP of same file/offset in next snap
  - If same, ref'd by next snap; do not free.
- O(# blocks written since prev snap)
- How many blocks to read?
  - Could be 2x # blocks written since prev snap

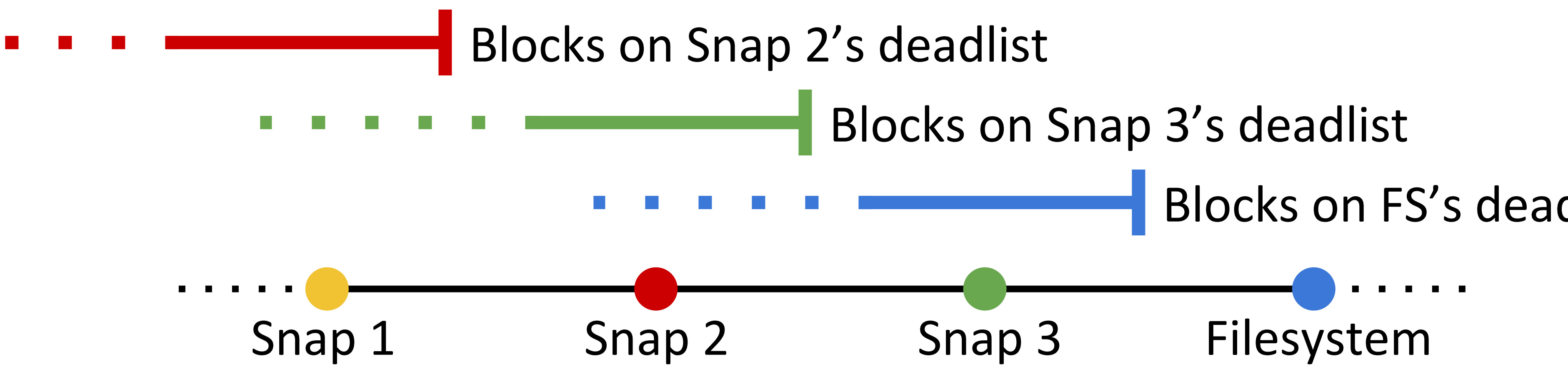
Snapshot Deletion ()

- Read Up to 2x # blocks written since prev snap
- Maybe you read a million blocks and free nothing
  - (next snap is identical to this one)
- Maybe you have to read 2 blocks to free one
  - (only one block modified under each indirect)
- RANDOM READS!
  - 200 IOPS, 8K block size -> free 0.8 MB/s
  - Can write at ~200MB/s

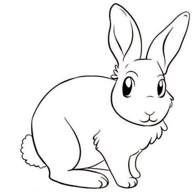
FIGURE 131. Hourglass

Snapshot Deletion ()

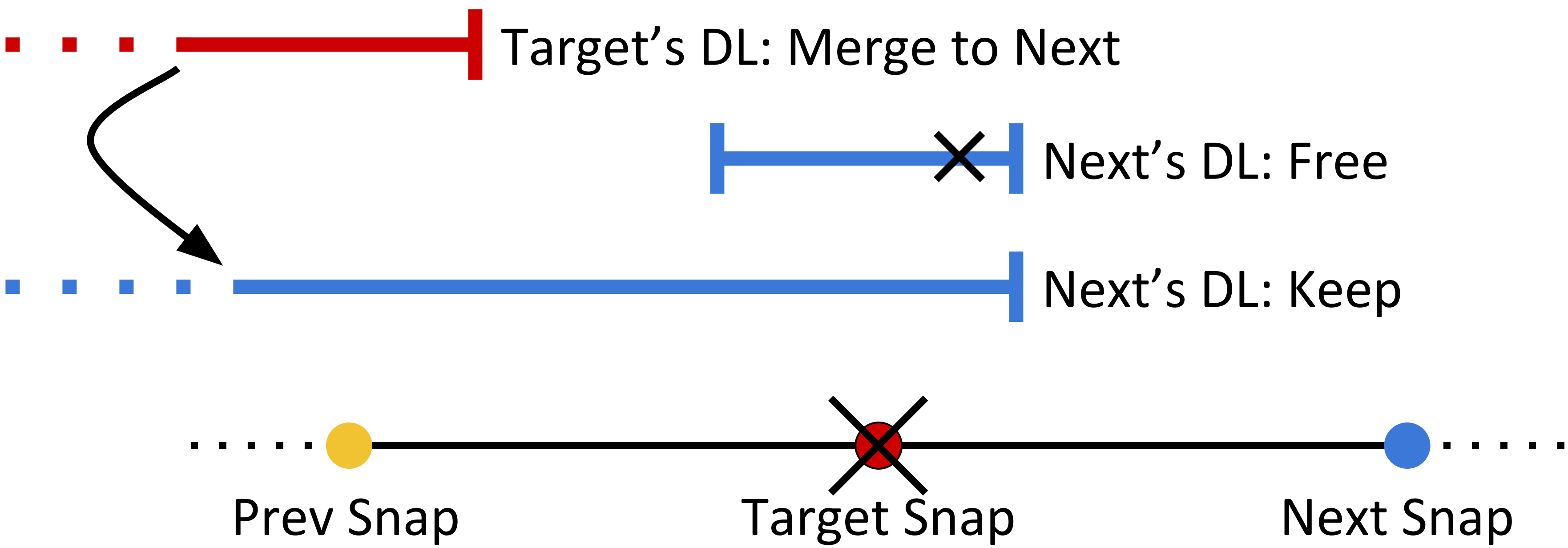
- Keep track of no-longer-referenced (“dead”) blocks
- Each dataset (snapshot & filesystem) has “dead list”
  - On-disk array of block pointers (BP’s)
  - blocks ref’d by prev snap, not ref’d by me

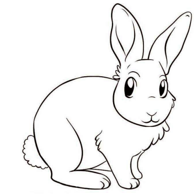


-> Snapshot Timeline ->

Snapshot Deletion ()

- Traverse next snap’s deadlist
- Free blocks with birth > prev snap



Snapshot Deletion ()

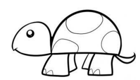
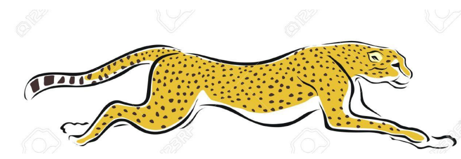
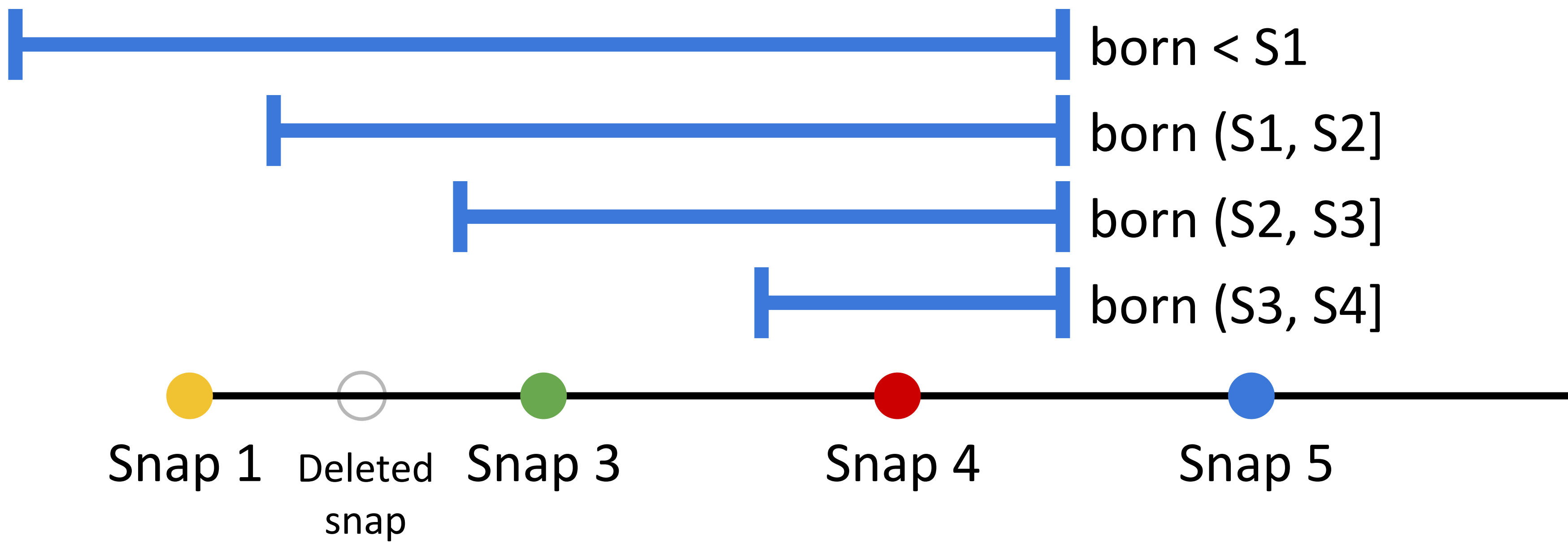
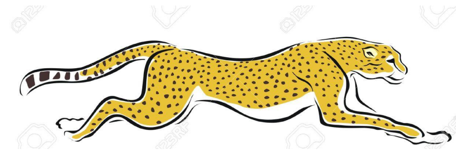
- O(size of next's deadlist)
  - = O(# blocks deleted before next snap)
  - Similar to  (# deleted ~= # created)
- Deadlist is compact!
  - 1 read = process 1024 BP's
  - Up to 2048x faster than Algo 1!
- Could still take a long time to free nothing

FIGURE 131. Hourglass

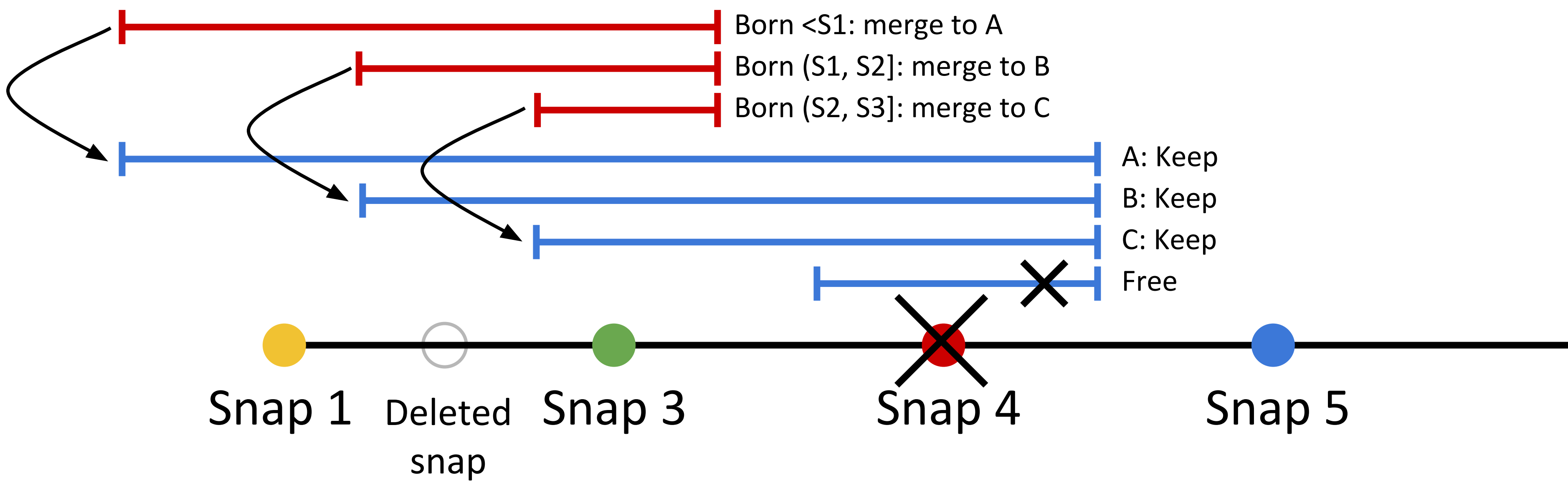
Snapshot Deletion ()

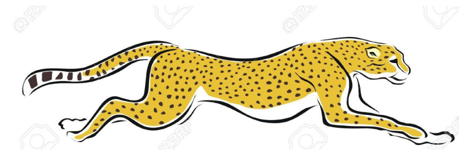
- Divide deadlist into sub-lists based on birth time
- One sub-list per earlier snapshot
    - Delete snapshot: merge FS’s sublists



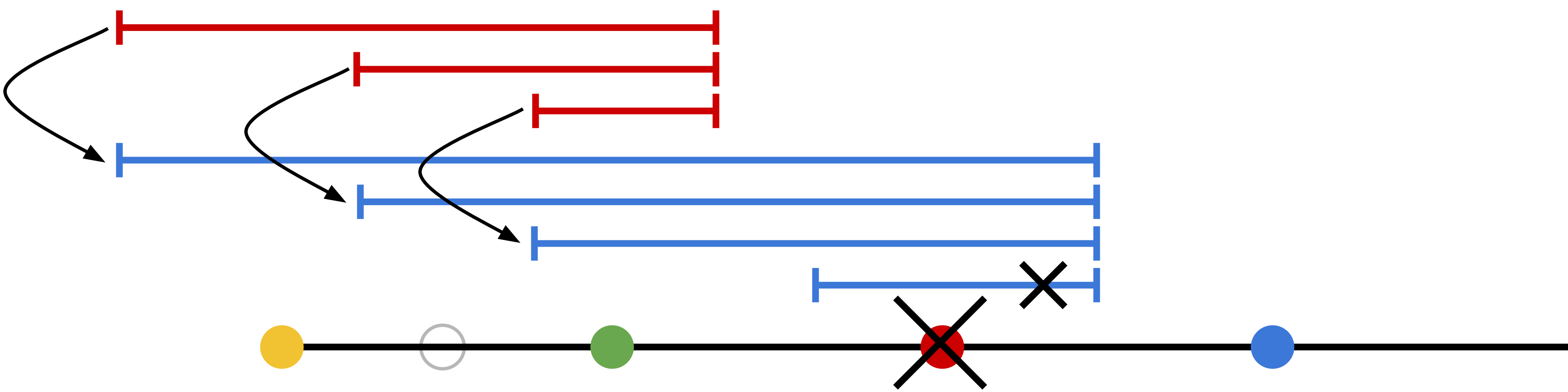
Snapshot Deletion ()

- Iterate over sublists
- If mintxg > prev, free all BP's in sublist
- Merge target's deadlist into next's
  - Append sublist by reference -> O(1)



Snapshot Deletion ()

- Deletion: O(# sublists + # blocks to free)
  - 200 IOPS, 8K block size -> free 1500MB/sec
- Optimal: O(# blocks to free)
- # sublists = # snapshots present when snap created
- # sublists << # blocks to free



Where did all the space go?

How much space are the snapshots using?

```
$ zfs list
NAME                    USED  AVAIL  REFER  MOUNTPOINT
rpool                   1000G  100G    50K  /rpool
rpool/fs                1000G  100G   700G  /rpool/fs

$ zfs get usedbysnapshots pool/fs
300G
```

How much space would be recovered if all of this fs's snapshots were destroyed.

I.e. How much storage am I paying for all these snapshots?

How much space are the snapshots using?

```
$ zfs list -t all
NAME                    USED  AVAIL  REFER  MOUNTPOINT
rpool                   1000G  100G    50K  /rpool
rpool/fs                1000G  100G   700G  /rpool/fs
rpool/fs@snap1             1G     -   699G  -
rpool/fs@snap2             2G     -   699G  -
rpool/fs@snap3             1G     -   700G  -
rpool/fs@snap4             3G     -   700G  -
```

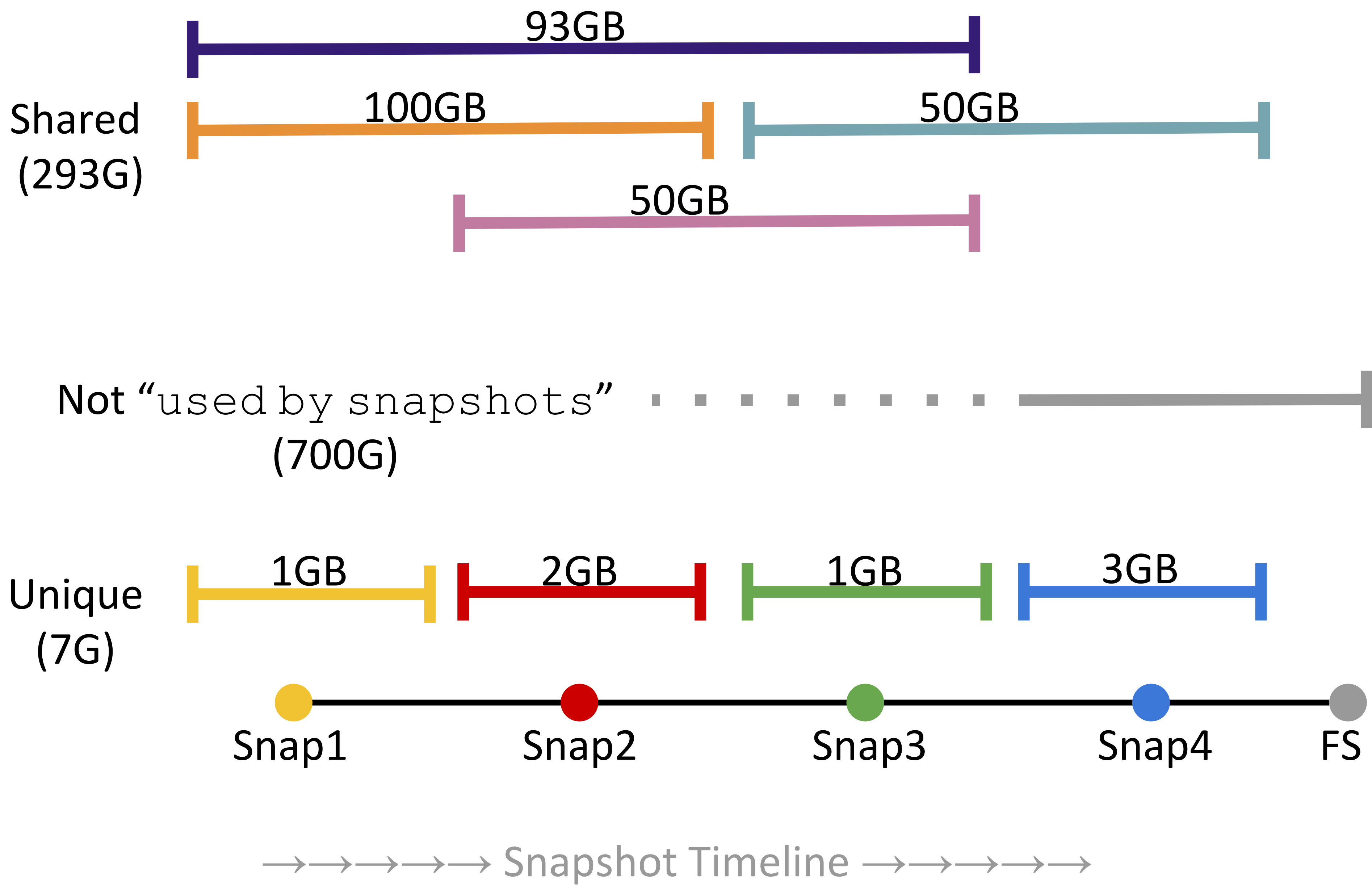
```
$ zfs get usedbysnapshots pool/fs
300G
```

How much space would be recovered if each snapshot was destroyed?

$$1+2+1+3 = \mathbf{7G} \neq \mathbf{300G}$$

What about the other 293GB?

Snapshots' "`used`" is "`unique`"



93GB
Shared
(293G)
100GB
50GB
50GB
Not "used by snapshots"
(700G)
Unique
(7G)
1GB
2GB
1GB
3GB
Snap1
Snap2
Snap3
Snap4
FS
→→→→→ Snapshot Timeline →→→→→

How much space is being used?

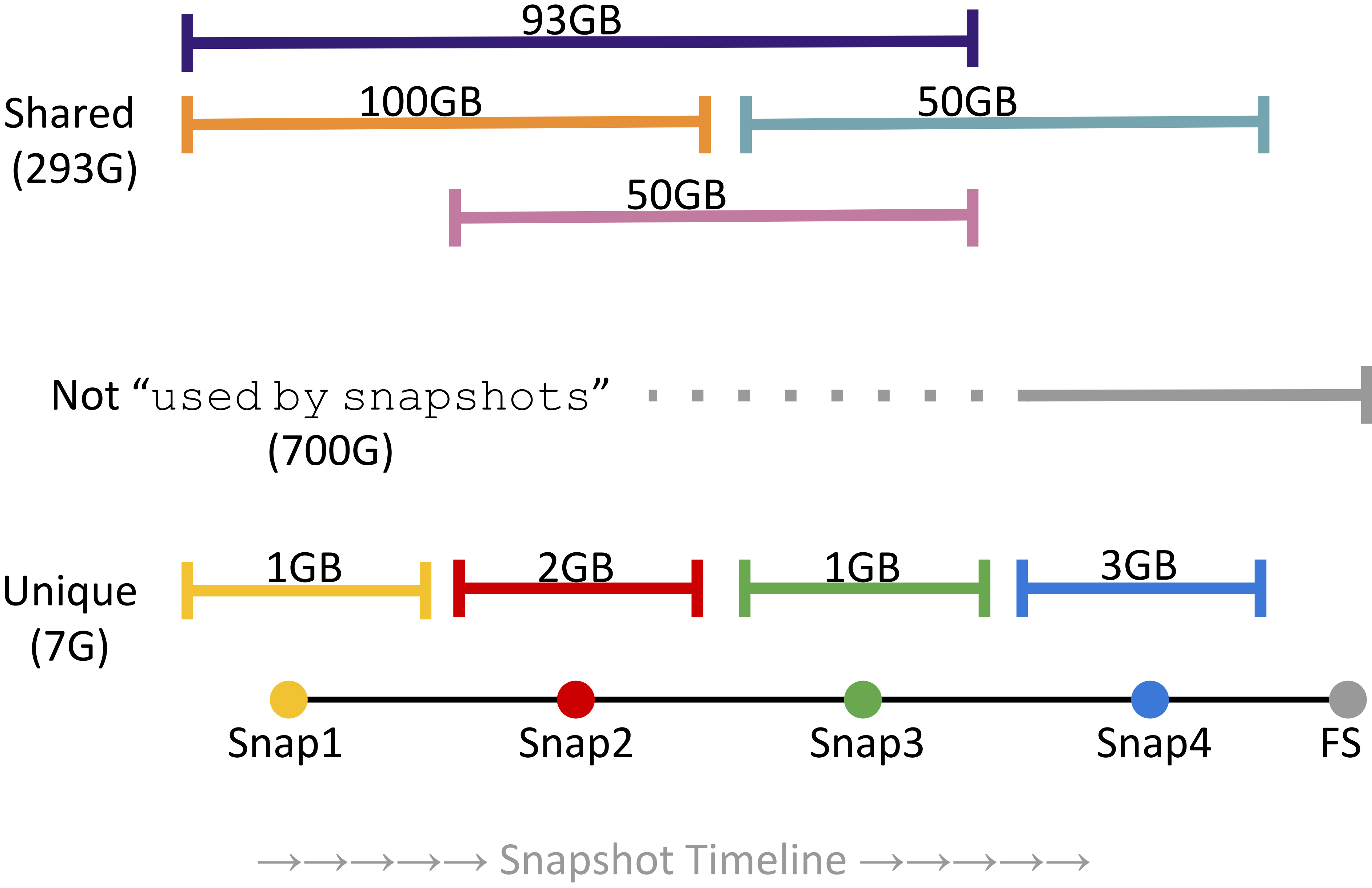
```
$ zfs list -t all -o name,written,used,refer rpool/fs
NAME                       WRITTEN   USED   REFER
rpool/fs                         0  1000G    700G
rpool/fs@snap1                894G     1G    699G
rpool/fs@snap2                 52G     2G    699G
rpool/fs@snap3                 51G     1G    700G
rpool/fs@snap4                  3G     3G    700G

$ zfs get usedbysnapshots pool/fs
300G
```

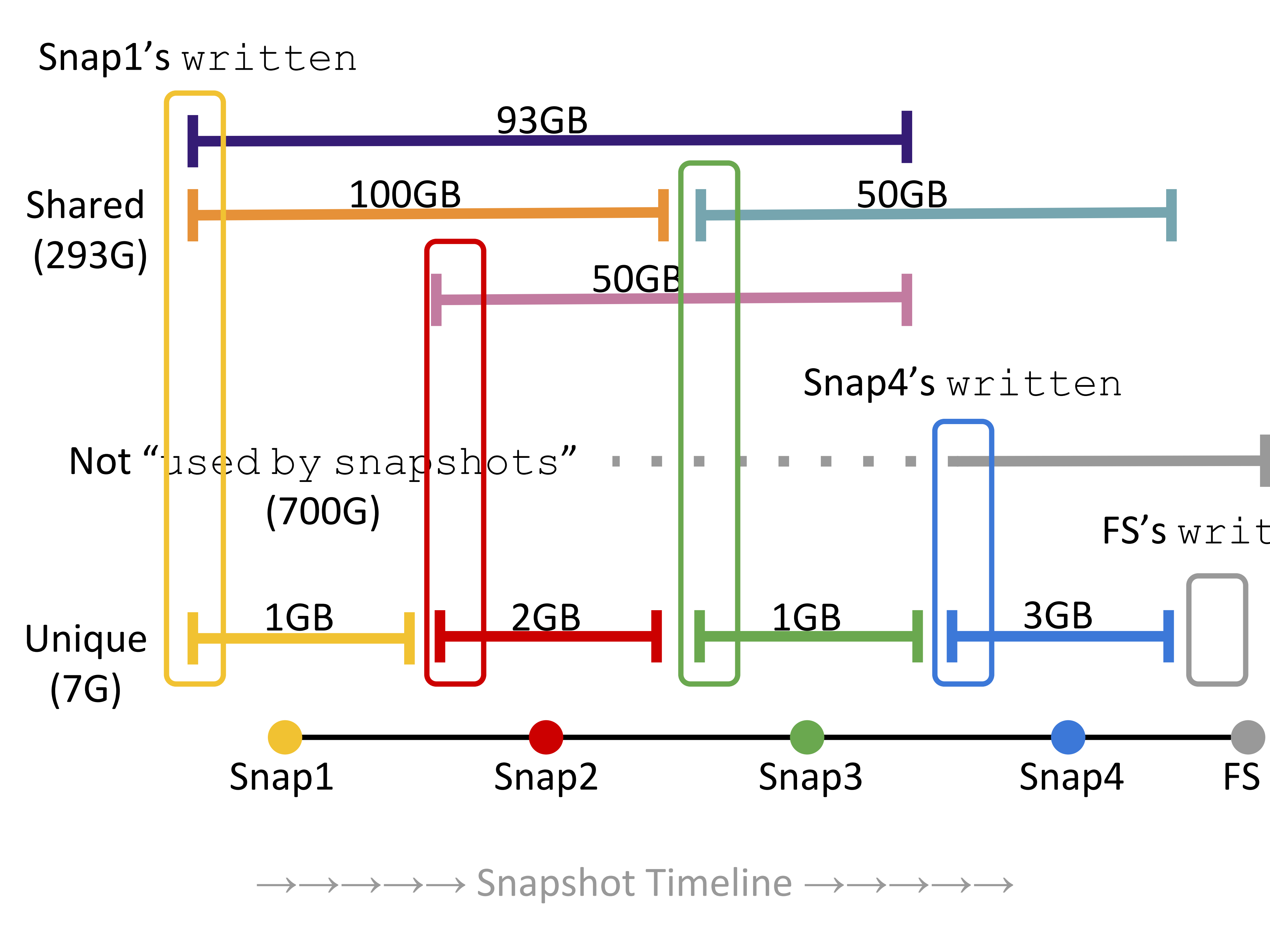
Sum of `written` = FS's `used`

0 + 894 + 52 + 51 + 3 = **1000G**

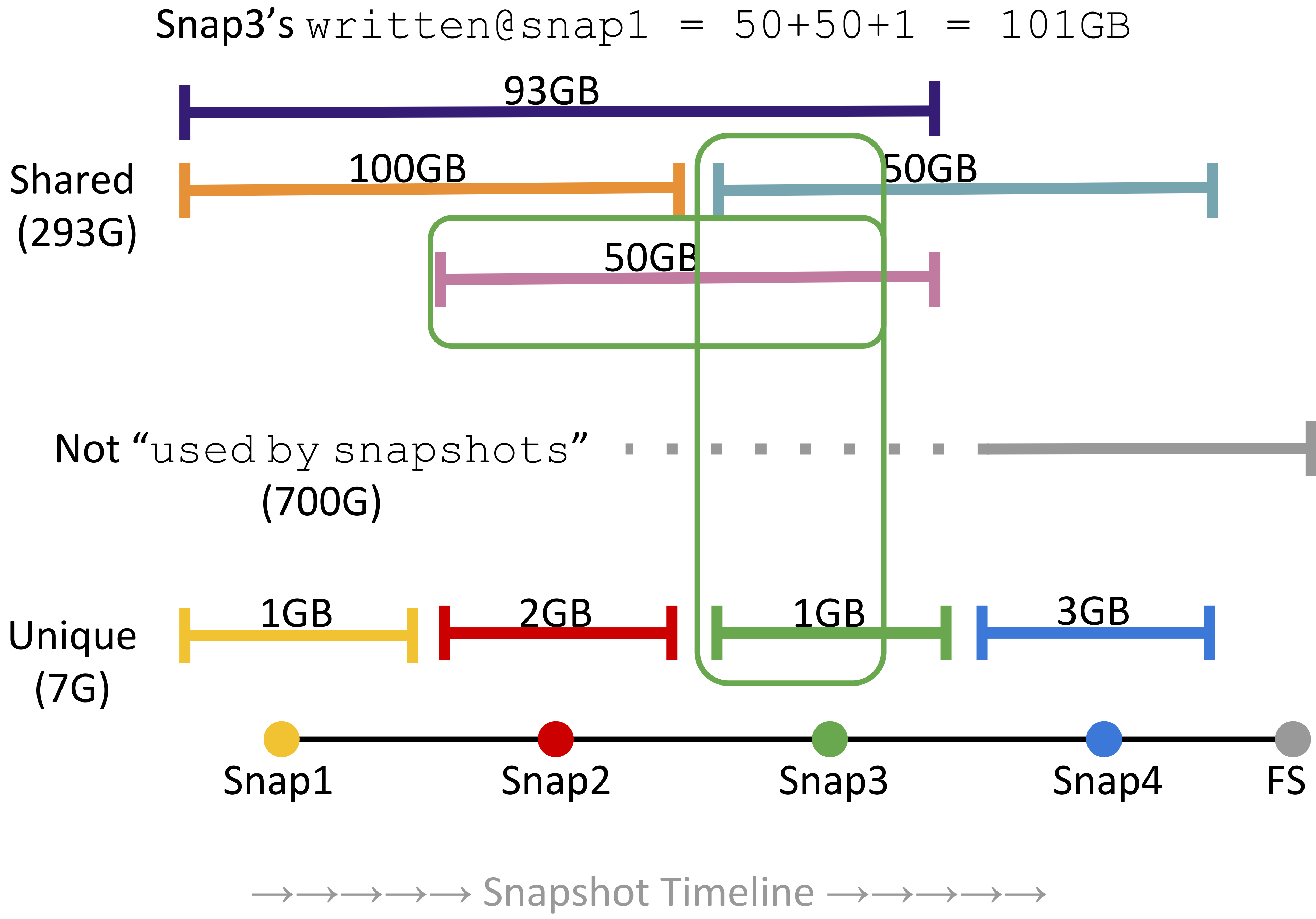
FS's `referenced` + `usedbysnapshots` = `used`

700 + 300 = **1000G**


93GB
Shared
(293G)
100GB
50GB
50GB
Not "used by snapshots"
(700G)
Unique
(7G)
1GB
2GB
1GB
3GB
Snap1
Snap2
Snap3
Snap4
FS
Snapshot Timeline



Snap1's written
93GB
Shared
(293G)
100GB
50GB
50GB
Snap4's written
Not "used by snapshots"
(700G)
FS's writ
Unique
(7G)
1GB
2GB
1GB
3GB
Snap1
Snap2
Snap3
Snap4
FS
→→→→→ Snapshot Timeline →→→→→



Snap3's written@snap1 = 50+50+1 = 101GB
93GB
Shared
(293G)
100GB
50GB
50GB
Not "used by snapshots"
(700G)
Unique
(7G)
1GB
2GB
1GB
3GB
Snap1
Snap2
Snap3
Snap4
FS
→→→→→ Snapshot Timeline →→→→→

How does `written@old` work?

- Can't quickly find "blocks born in this txg range that exist in this snapshot"
  - Deadlists store blocks that were killed
  - We are interested in some blocks that are still alive
- New's `refer` - old's `refer` + space freed in between
- Deadlists tell us what was freed
- Written
  - Examine one sublist
  - O(1)
- written@...
  - Examine all snapshots in between
    - Examine their sublists for births < old
  - O(num_snaps_between_old_and_new * num_snaps_before_old)

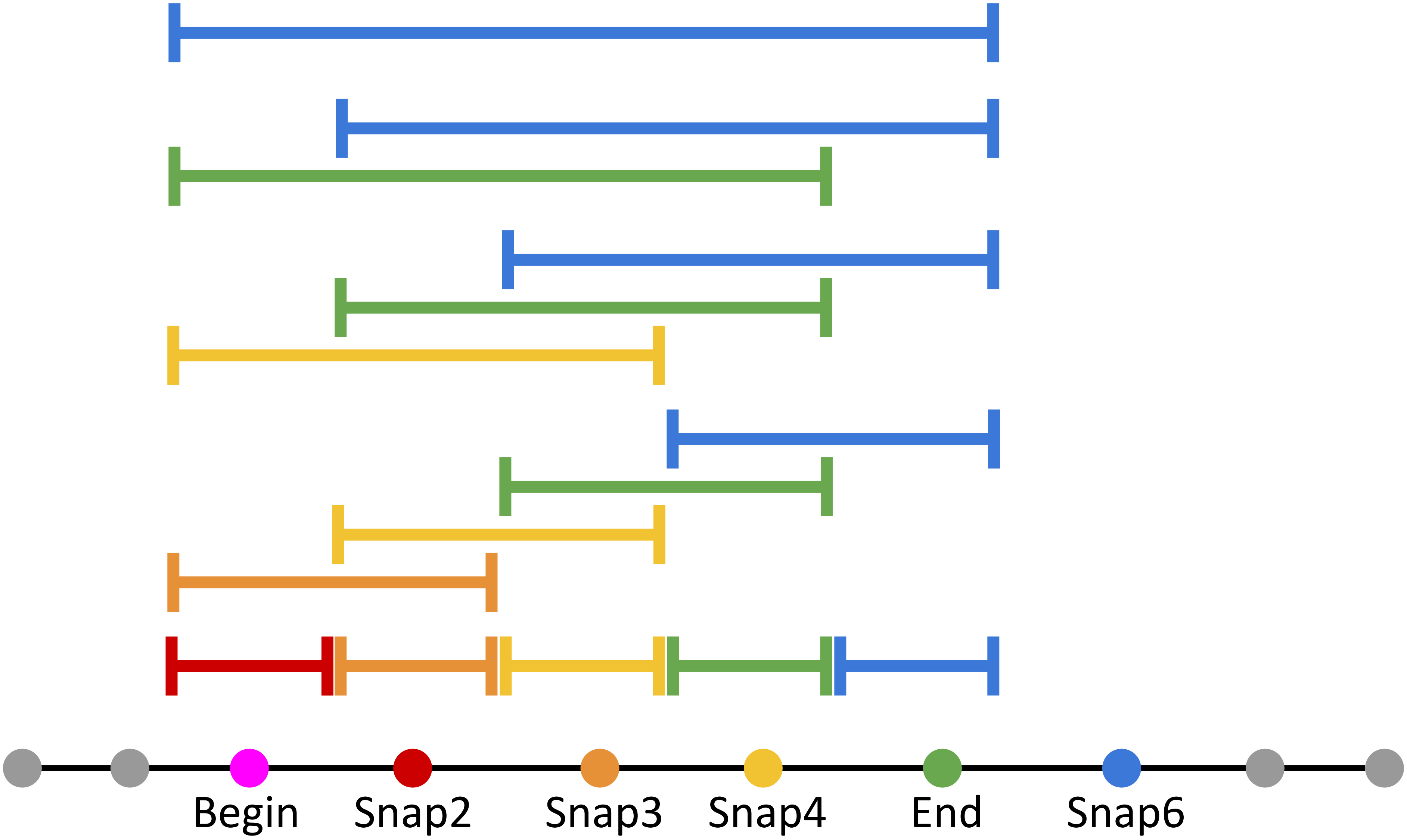
How to understand shared snapshot space?

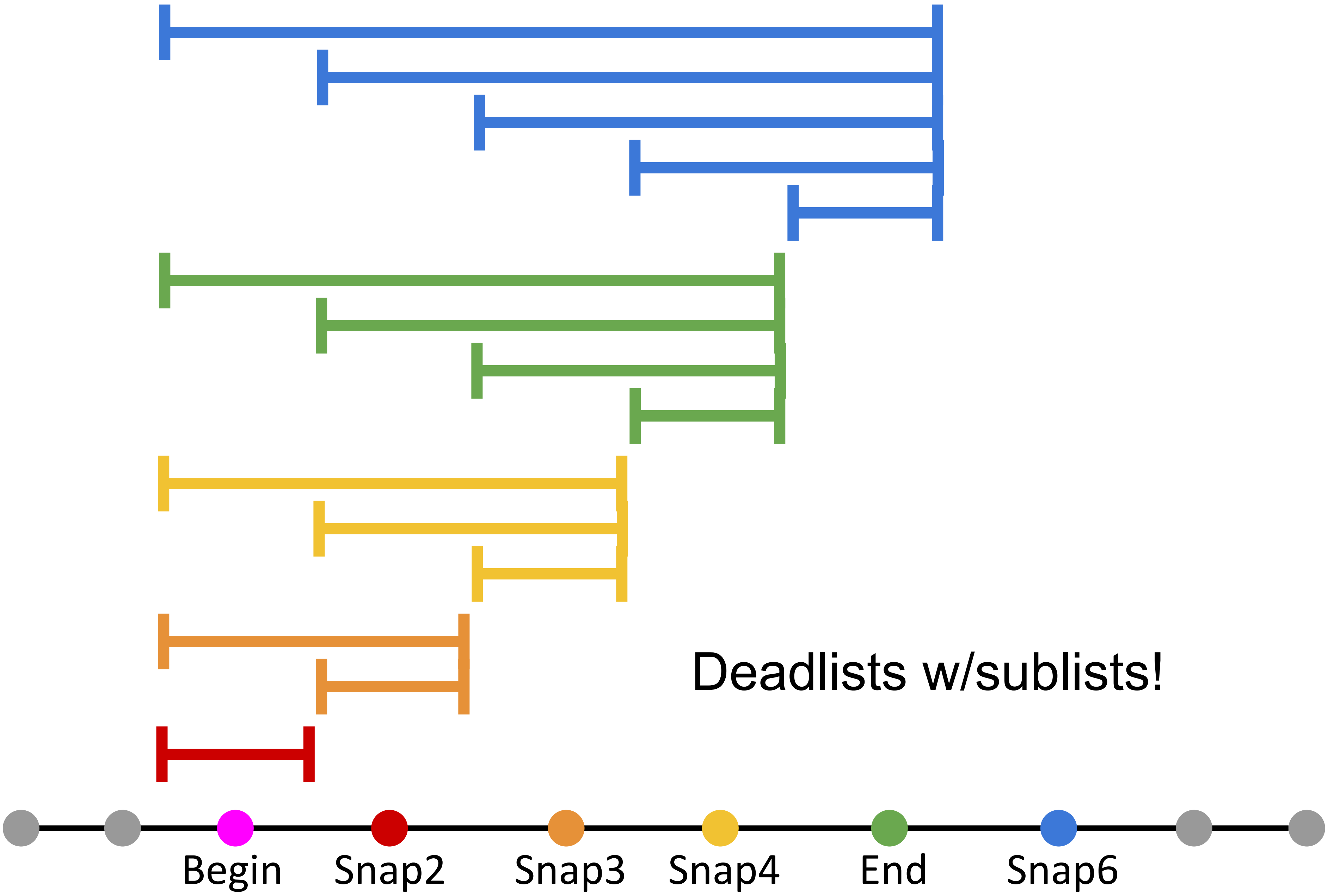
- What if we delete some of the snapshots?
  - `zfs destroy -nv pool/fs@begin%end`
  - `zfs destroy -nv pool/fs@a,b,j,k,z`
- How to use
  - Categorize snap space into different (application-defined) classes
  - E.g. space for periodic snapshots vs user-requested snaps (but some space will be shared between classes too)

How to implement shared snapshot space?

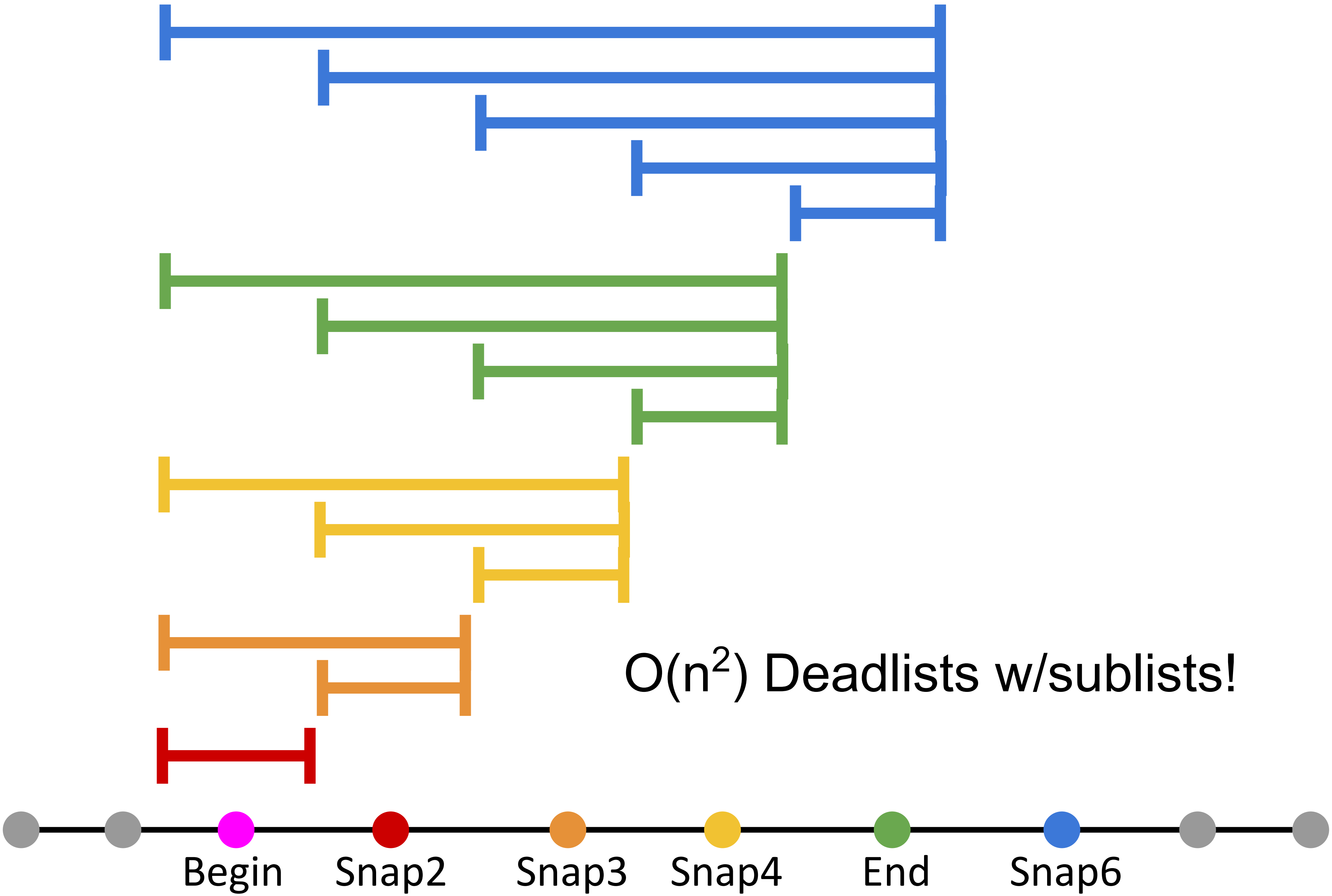
- Corner cases:
  - One snapshot: same as `used` and `unique` properties
  - All snapshots: same as `used by snapshots` property
- General case:
  - Blocks born after begin->prev, died before end->next
  - Deadlist breakdown

What if we delete Begin...End (5 snaps)?



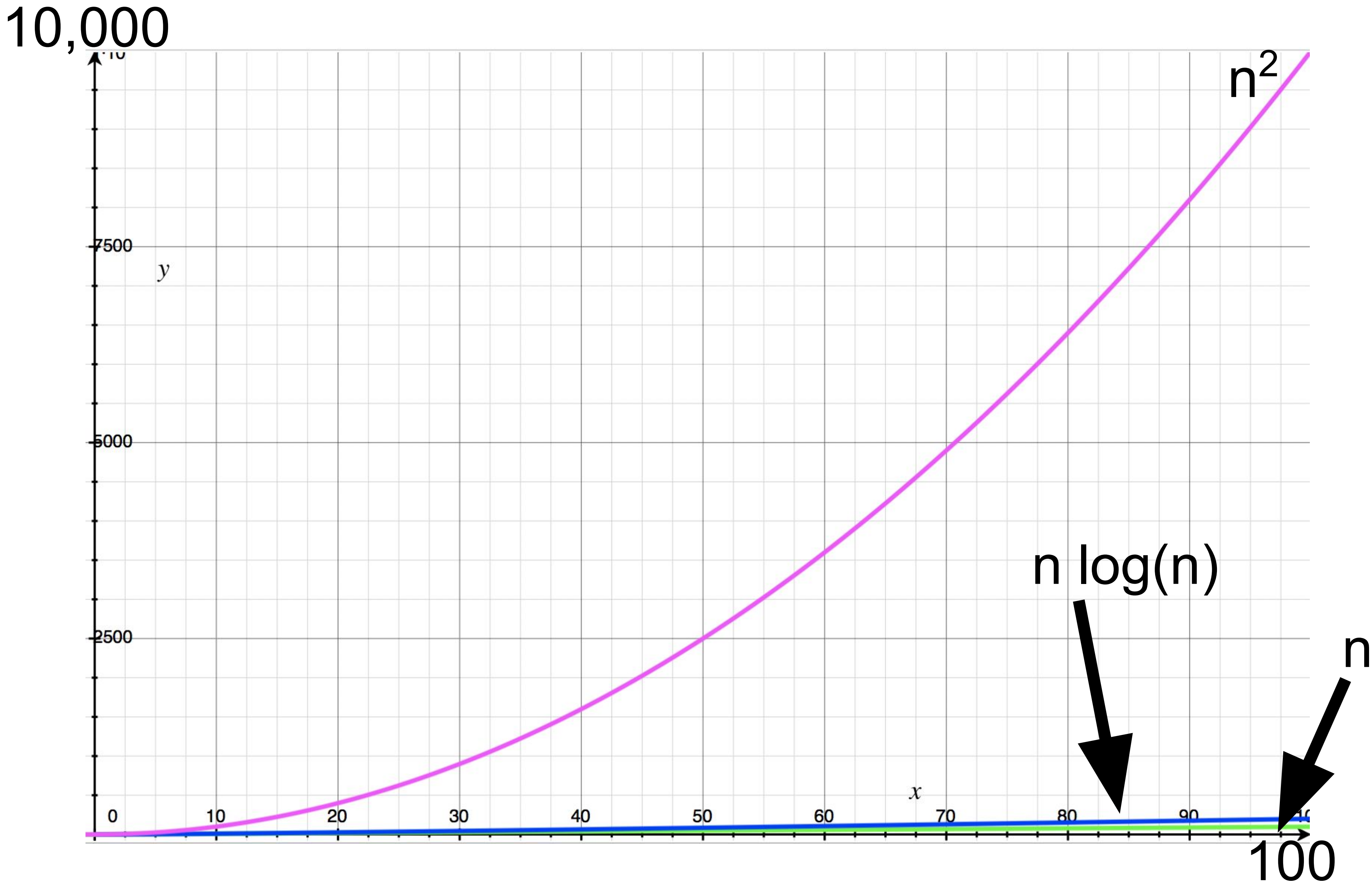


Deadlists w/sublists!
Begin
Snap2
Snap3
Snap4
End
Snap6



O(n2) Deadlists w/sublists!
Begin
Snap2
Snap3
Snap4
End
Snap6

Fear O(n^2)



Fear $O(n^2)$

- About those 8700 snaps per year (per fs)...
- 75 Million lists!
  - Imagine each one is 1 sector (4K)
  - 288GB on disk (per fs)
- `zfs destroy -nv pool/fs@snap10%snap8690`
  - Read them all (at 10,000 iops) in 2 hours
  - While holding locks that prevent TXG sync

Fear $O(n^2)$?

Nearly all lists are empty

- Don’t store them on disk (`empty_bpobj` feature, 2012)
  - 60 seconds (when ARC-cached)
- Partial deadlist load (ignore empty bpobj’s)
  - 5x speed up → 12 sec
  - Review out
- Cache (partial) deadlist
  - Additional 70x speed up (350x from base) → 0.2 sec
  - Prototyped
- Still $O(n^2)$!

Confused by snapshot space usage? You’re not alone :-)

1. Look at `usedbysnapshots` first
2. Ignore snapshots’ `used` (it’s really `unique`)
3. `written` can help understand space growth
4. “What if” with `zfs destroy -nv pool/fs@<snaps>`

7th annual OZDS!
November 4-5, 2019
Talk proposals due Aug 19
Sponsorship opportunities