PostgreSQL + ZFS

Best Practices and Standard Procedures

"If you are not using ZFS,
you are losing data*."

Clark's Three Laws

1. When a distinguished but elderly scientist states that something is possible, he is almost certainly right. When he states that something is impossible, he is very probably wrong.
2. The only way of discovering the limits of the possible is to venture a little way past them into the impossible.
3. Any sufficiently advanced technology is indistinguishable from magic.

ZFS is not magic, but it is an incredibly impressive piece of software.

PostgreSQL and ZFS

- Many bits
- Lots of bits
- Huge bits
- It's gunna be great
- Very excited
- We have the best filesystems
- People tell me this is true
- Except the fake media, they didn't tell me this

PostgreSQL and ZFS: It's about the bits and storage, stupid.

- ~~Many bits~~
- ~~Lots of bits~~
- ~~Huge bits~~
- ~~It's gunna be great~~
- ~~Very excited~~
- ~~We have the best filesystems~~
- ~~People tell me this is true~~
- ~~Except the fake media, they didn't tell me this~~

Too soon?

PostgreSQL and ZFS

1. Review PostgreSQL from a storage administrator's perspective
2. Learn what it takes to become a PostgreSQL "backup expert"
3. Dive through a naive block-based filesystem
4. Walk through the a high-level abstraction of ZFS
5. See some examples of how to use ZFS with PostgreSQL
    - Tips
    - Tunables
    - Anecdotes

Some FS minutiae may have been harmed in the making of this talk.
Nit-pick as necessary (preferably after).

PostgreSQL - A Storage Administrator's View

- User-land page cache maintained by PostgreSQL in shared memory
- 8K page size
- Each PostgreSQL table is backed by one or more files in `$PGDATA/`
- Tables larger than 1GB are automatically shared into individual 1GB files
- `pwrite(2)`'s to tables are:
  - append-only if no free pages in the table are available
  - in-place updated if free pages are available in the free-space map
- `pwrite(2)`'s are page-aligned
- Makes heavy use of a Write Ahead Log (WAL), aka an Intent Log

Storage Administration: WAL on Disk

- WAL files are written to sequentially
- append-only IO
- Still 8K page-aligned writes via `pwrite(2)`
- WAL logs are 16MB each, pre-allocated
- WAL logs are never `unlink(2)`'ed, only recycled via rename(2)
- Low-latency `pwrite(2)`'s and `fsync(2)` for WAL files is required for good write performance

PostgreSQL - Backups

Traditionally, only two SQL commands that you must know:

1. `pg_start_backup('my_backup')`
2. `${some_random_backup_utility} $PGDATA/`
3. `pg_stop_backup()`

Wait for `pg_start_backup()` to return before backing up `$PGDATA/` directory.

PostgreSQL - Backups

Only ~~two~~^Wthree SQL commands that you must know:

1. `CHECKPOINT`
2. `pg_start_backup('my_backup')`
3. `${some_random_backup_utility} $PGDATA/`
4. `pg_stop_backup()`

Manual `CHECKPOINT` if you can't twiddle the args to `pg_start_backup()`.

PostgreSQL - Backups

Only ~~two~~^W~~three~~^Wtwo commands that you must know:

1. ~~CHECKPOINT~~
2. `pg_start_backup('my_backup', true)`
3. `${some_random_backup_utility} $PGDATA/`
4. `pg_stop_backup()`

`pg_start_backup('my_backup', true)`
a.k.a. aggressive checkpointing (vs default perf hit of: `0.5 * checkpoint_completion_target`)

Achievement unlocked
PostgreSQL Backup Expert

Achievement Pending
ZFS Ninja

Quick ZFS Primer

Quick ZFS Primer

TIP: Look for parallels.

Quick ZFS Primer: Features (read: why you must use ZFS)

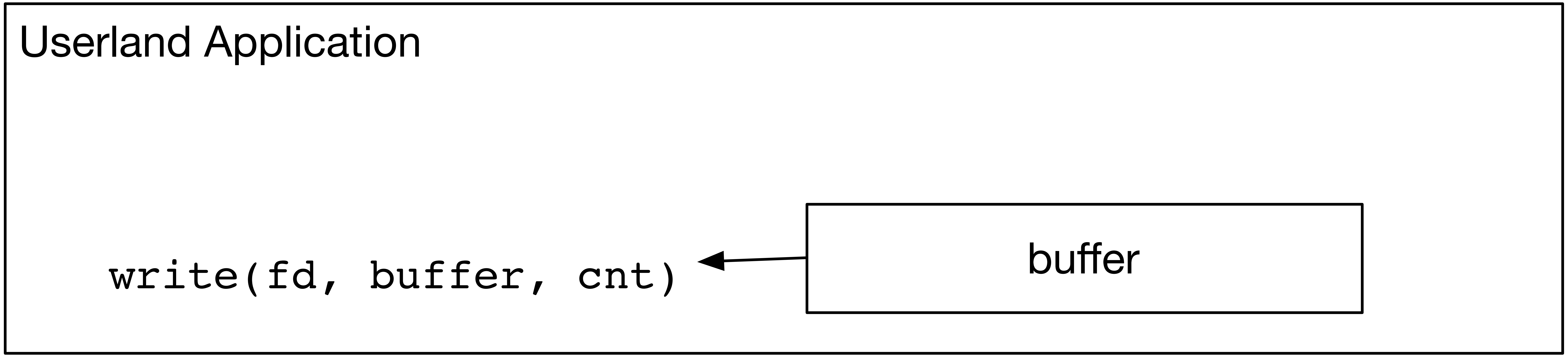
- Never inconsistent (no `fsck(8)`'s required, ever)
- Filesystem atomically moves from one consistent state to another consistent state
- All blocks are checksummed
- Compression builtin
- Snapshots are free and unlimited
- Clones are easy
- Changes accumulate in memory, flushed to disk in a transaction
- Redundant metadata (and optionally data)
- Filesystem management independent of physical storage management
- Log-Structured Filesystem
- Copy on Write (COW)

Feature Consequences (read: how your butt gets saved)

- bitrot detected and automatically corrected if possible
  - phantom writes
  - misdirected reads or writes by the drive heads
  - DMA parity errors
  - firmware or driver bugs
  - RAM capacitors aren't refreshed fast enough or with enough power
- Phenomenal sequential and random IO write performance
- Performance increase for sequential reads
- Cost of ownership goes down
- New tricks and tools to solve "data gravity" problems

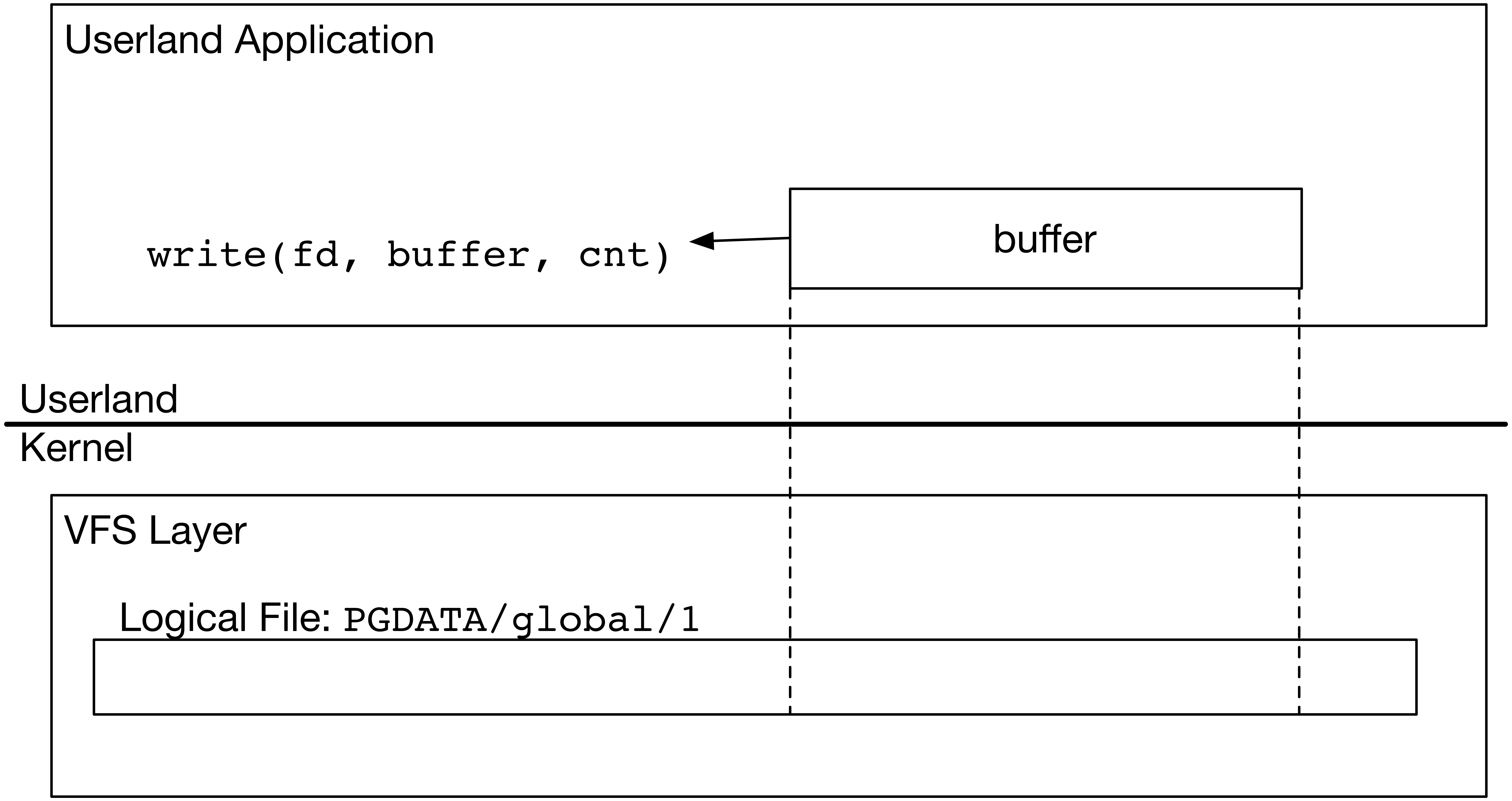
ELI5: Block Filesystems vs Log Structured Filesystems

Block Filesystems: Top-Down

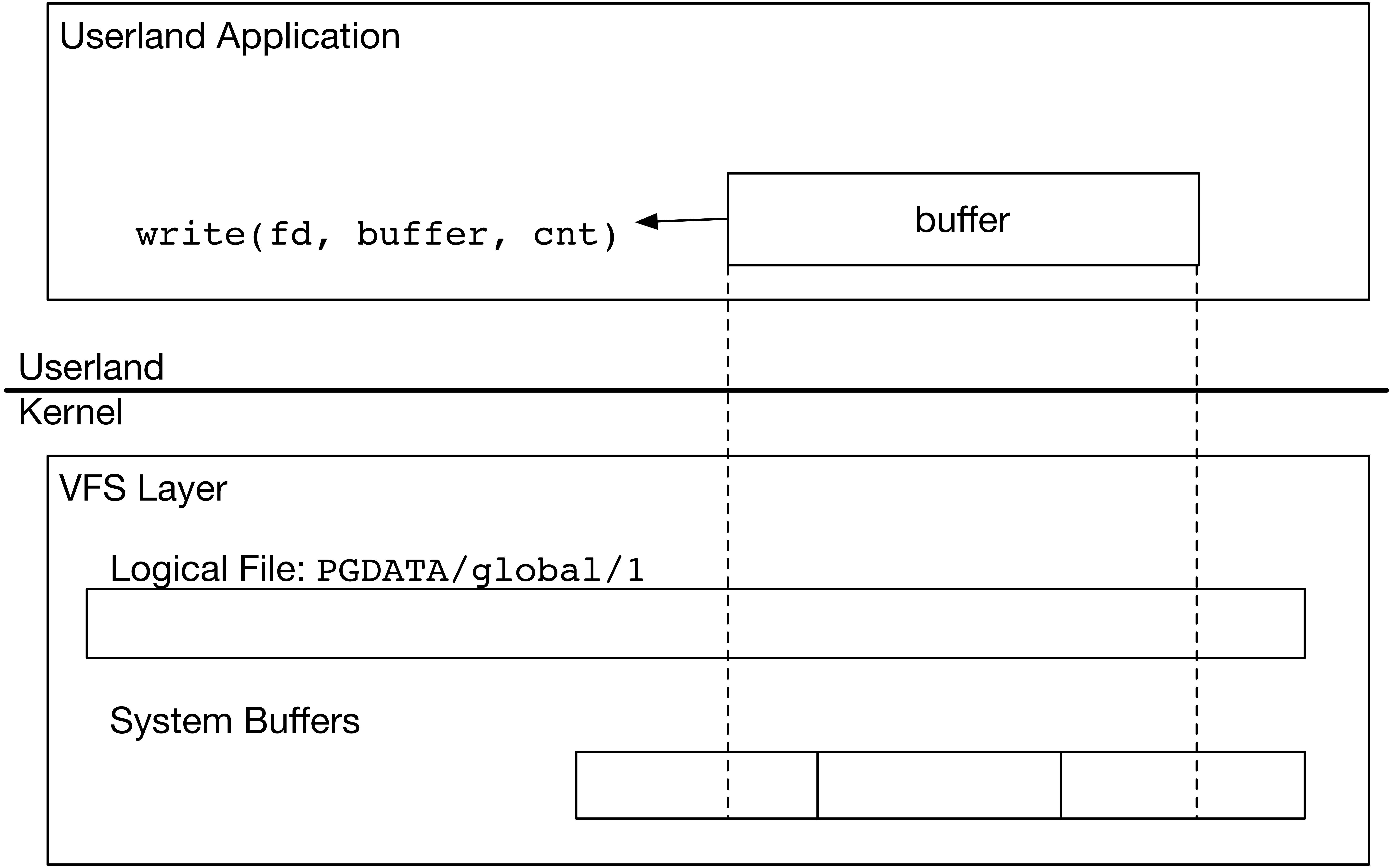


Userland

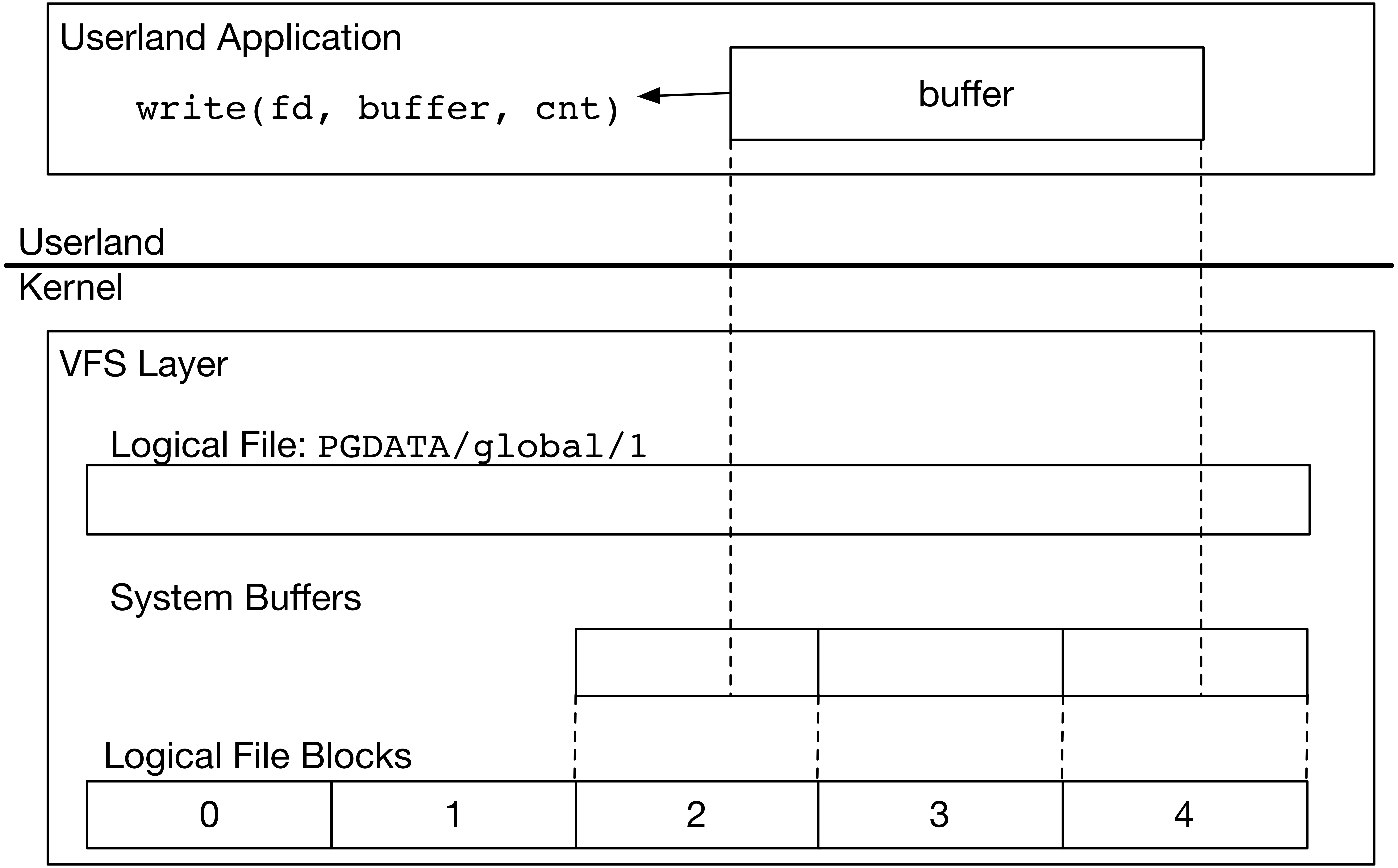
Block Filesystems: Top-Down



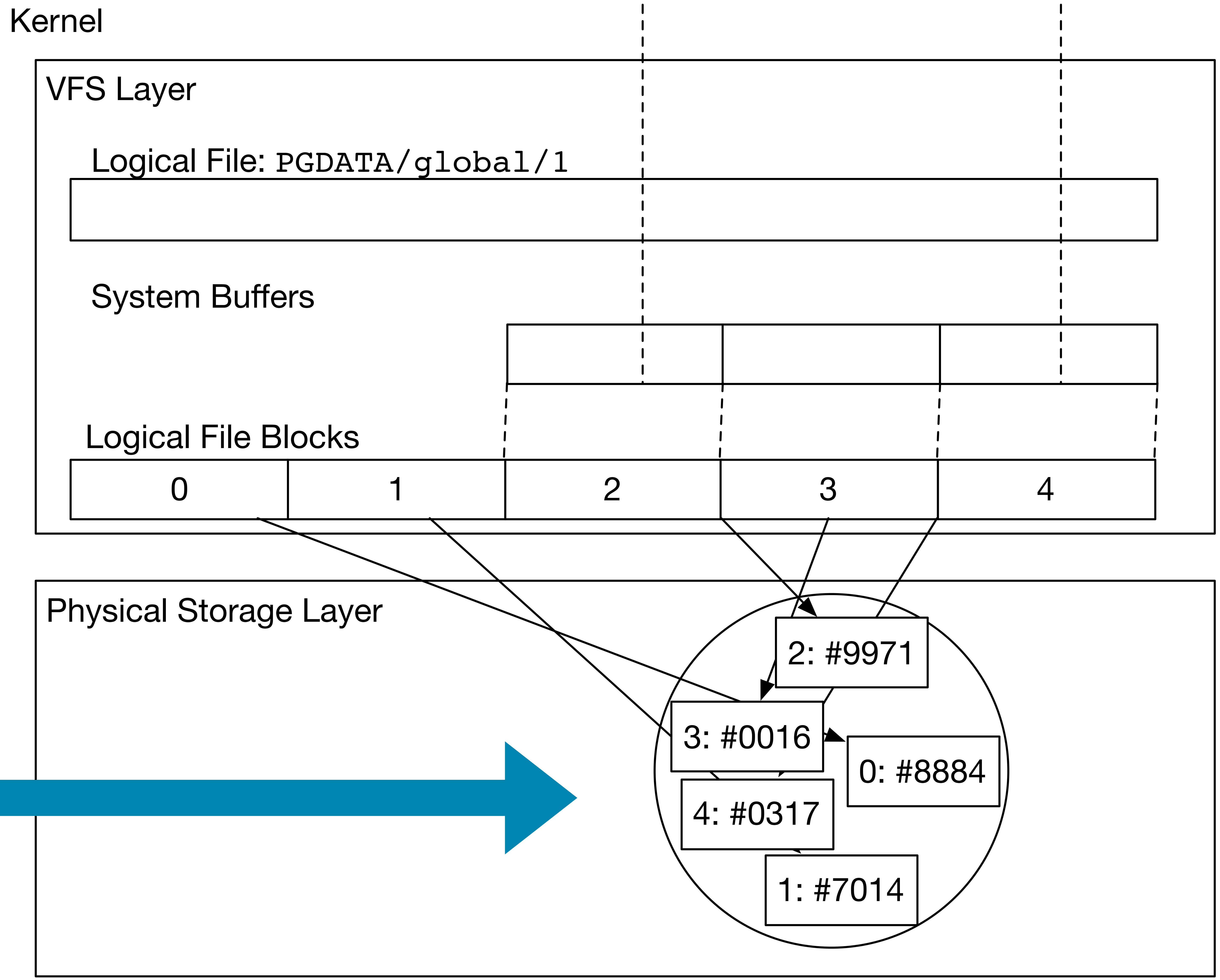
Block Filesystems: Top-Down



Block Filesystems: Top-Down

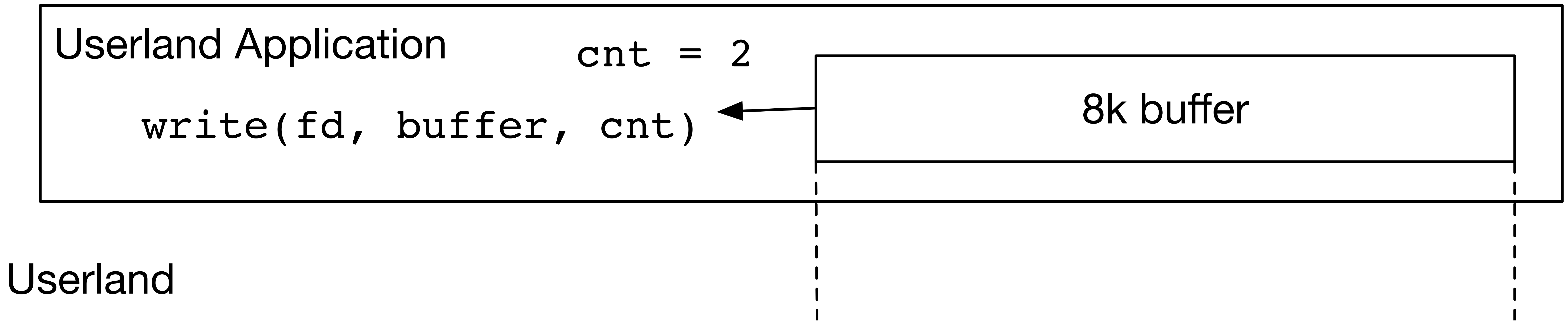


Block Filesystems: Top-Down

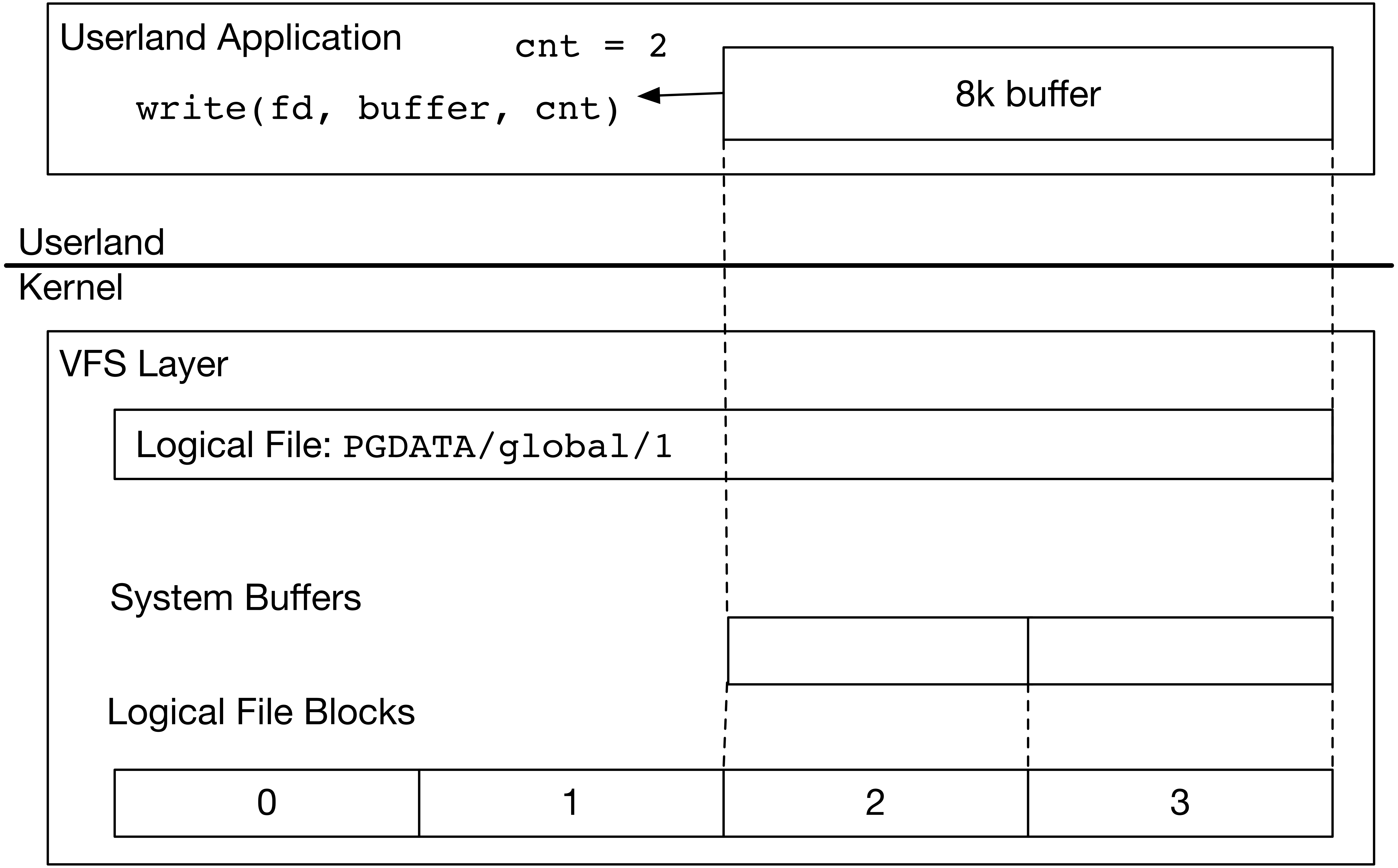


Pretend this is a spinning disk

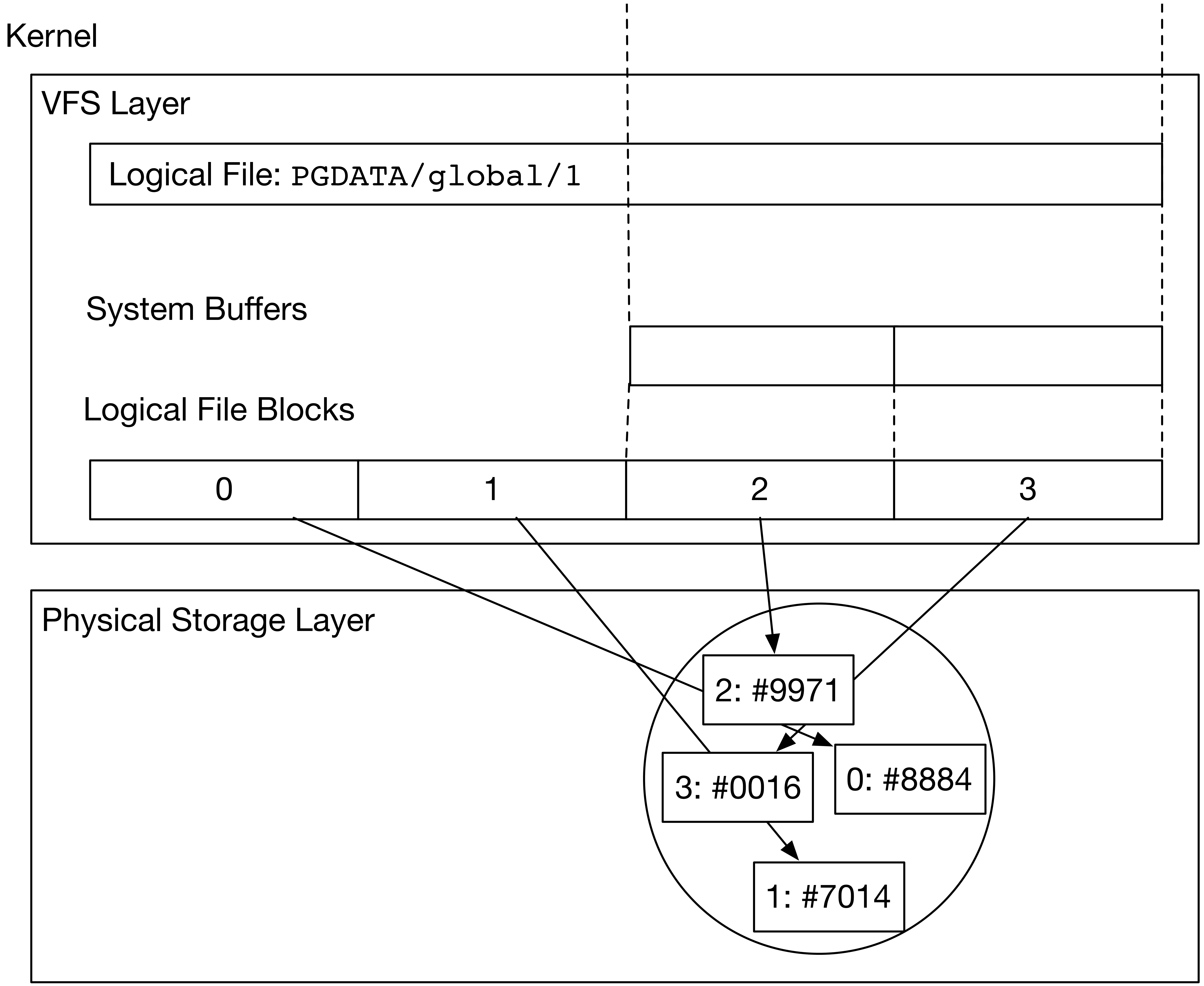
Block Filesystems: PostgreSQL Edition



Block Filesystems: PostgreSQL Edition



Block Filesystems: PostgreSQL Edition



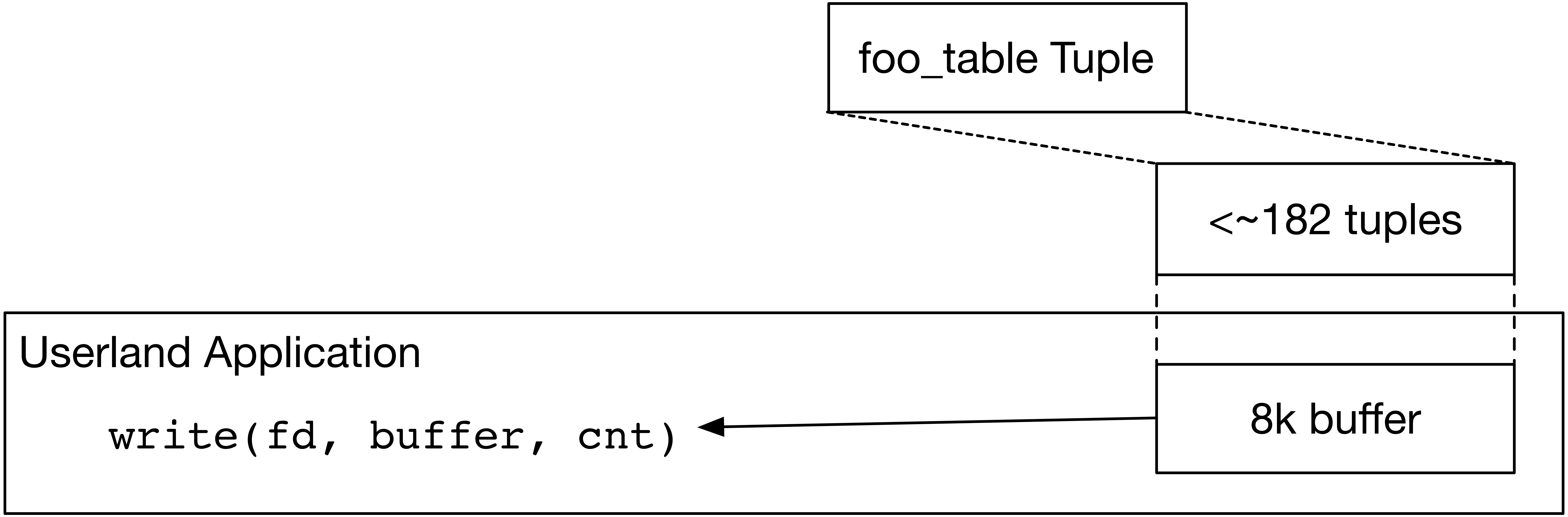
Quiz Time

What happens when you twiddle a bool in a row?

```
UPDATE foo_table SET enabled = FALSE WHERE id = 123;
```

Quiz Answer: Write Amplification

```
UPDATE foo_table SET enabled = FALSE WHERE id = 123;
```



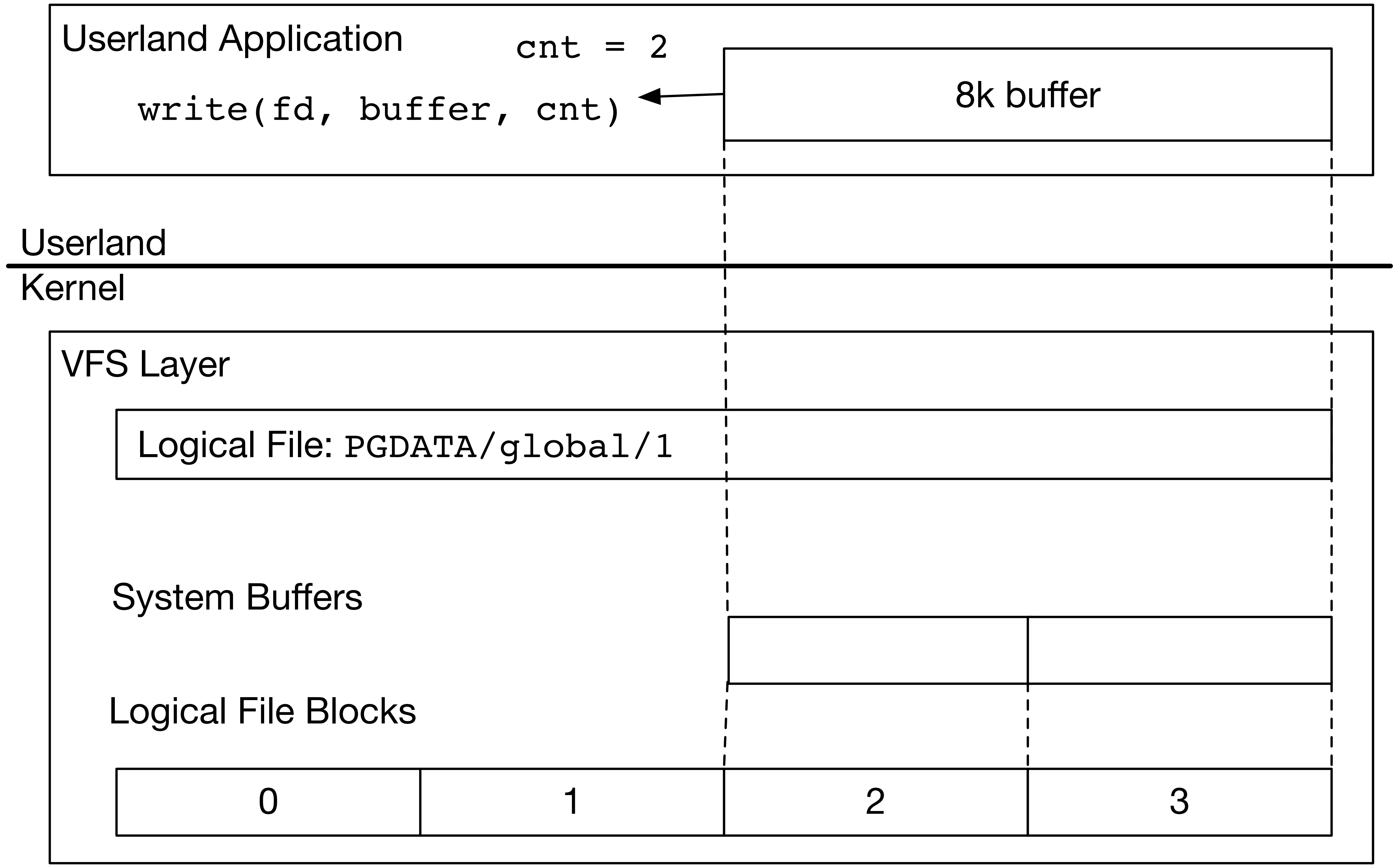
ZFS Tip: postgresql.conf: full_page_writes=off

```
ALTER SYSTEM SET full_page_writes=off;
CHECKPOINT;
-- Restart PostgreSQL
```

IMPORTANT NOTE: **`full_page_writes=off`** interferes with cascading replication

Block Filesystems: PostgreSQL Edition

- buffers can be 4K
- disk sectors are 512B - 4K
- ordering of writes is important
- consistency requires complete cooperation and coordination

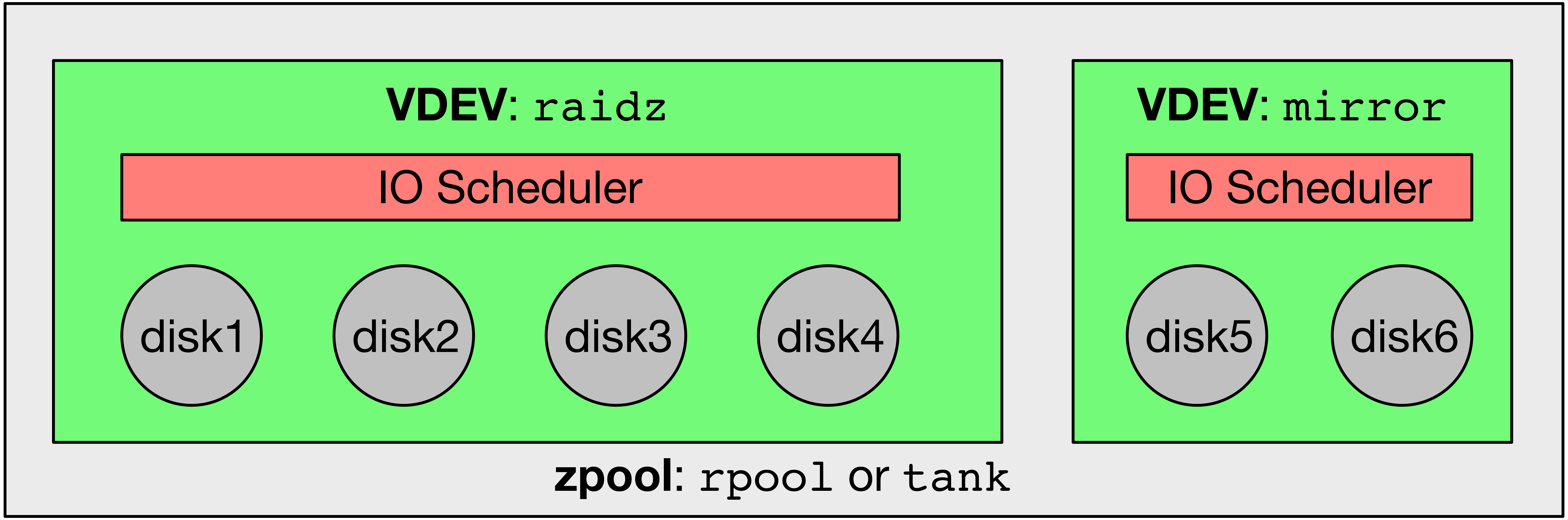


ZFS Filesystem Storage Abstraction

Physical Storage is decoupled from Filesystems.

If you remember one thing from this section, this is it.

VDEVs On the Bottom

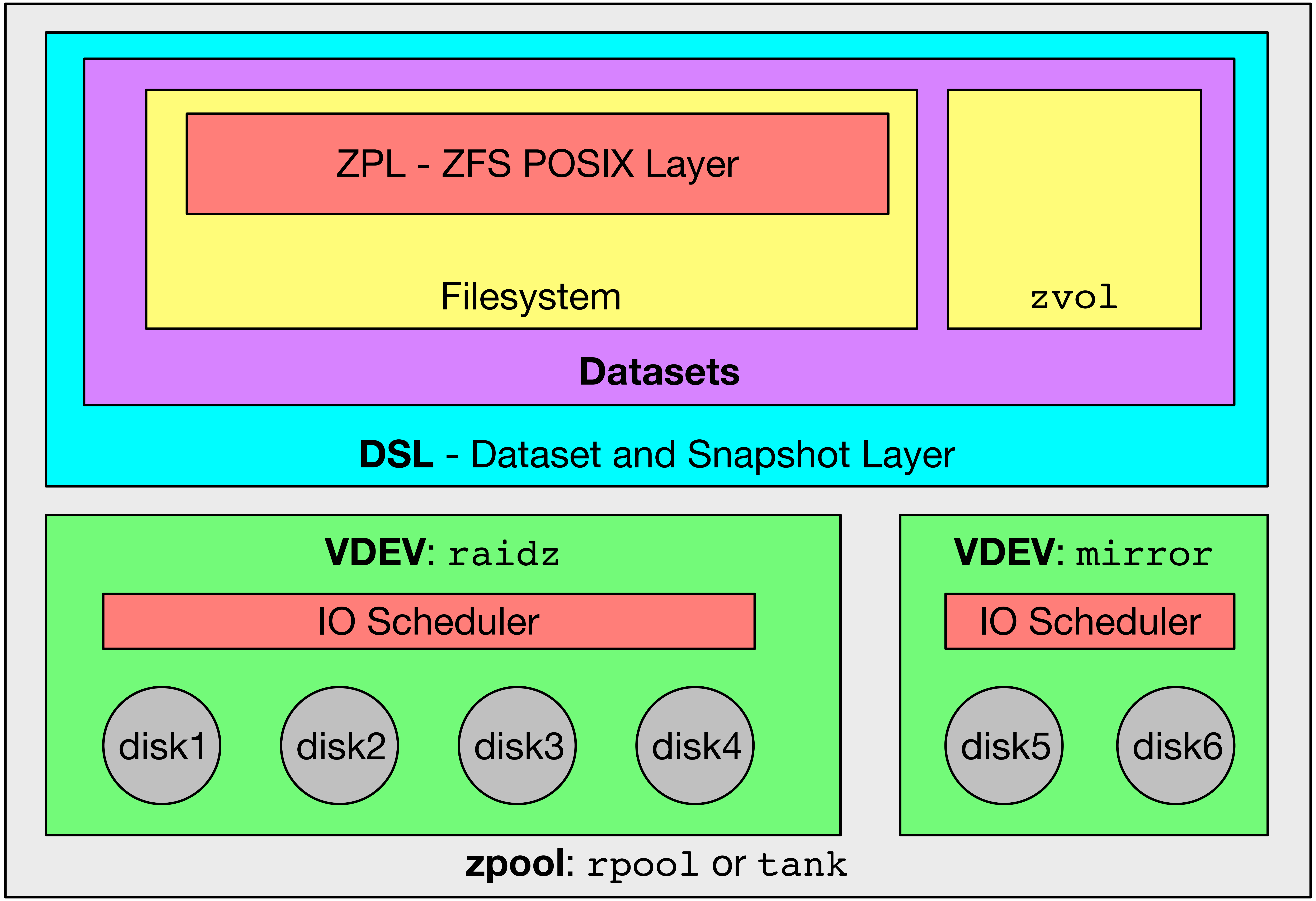


Filesystems On Top

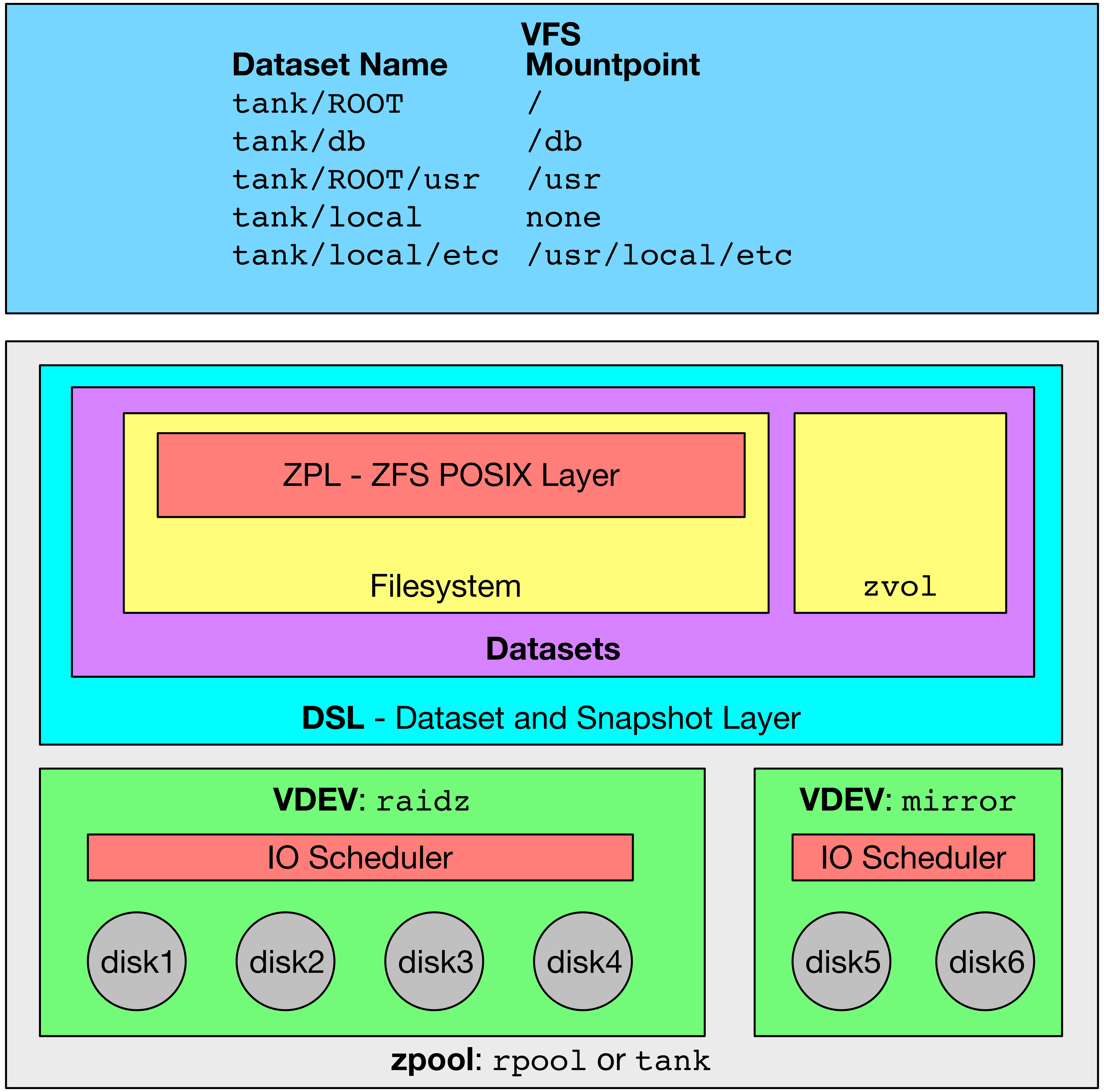
Dataset Name	VFS Mountpoint
tank/ROOT	/
tank/db	/db
tank/ROOT/usr	/usr
tank/local	none
tank/local/etc	/usr/local/etc

canmount=off

Offensively Over Simplified Architecture Diagram



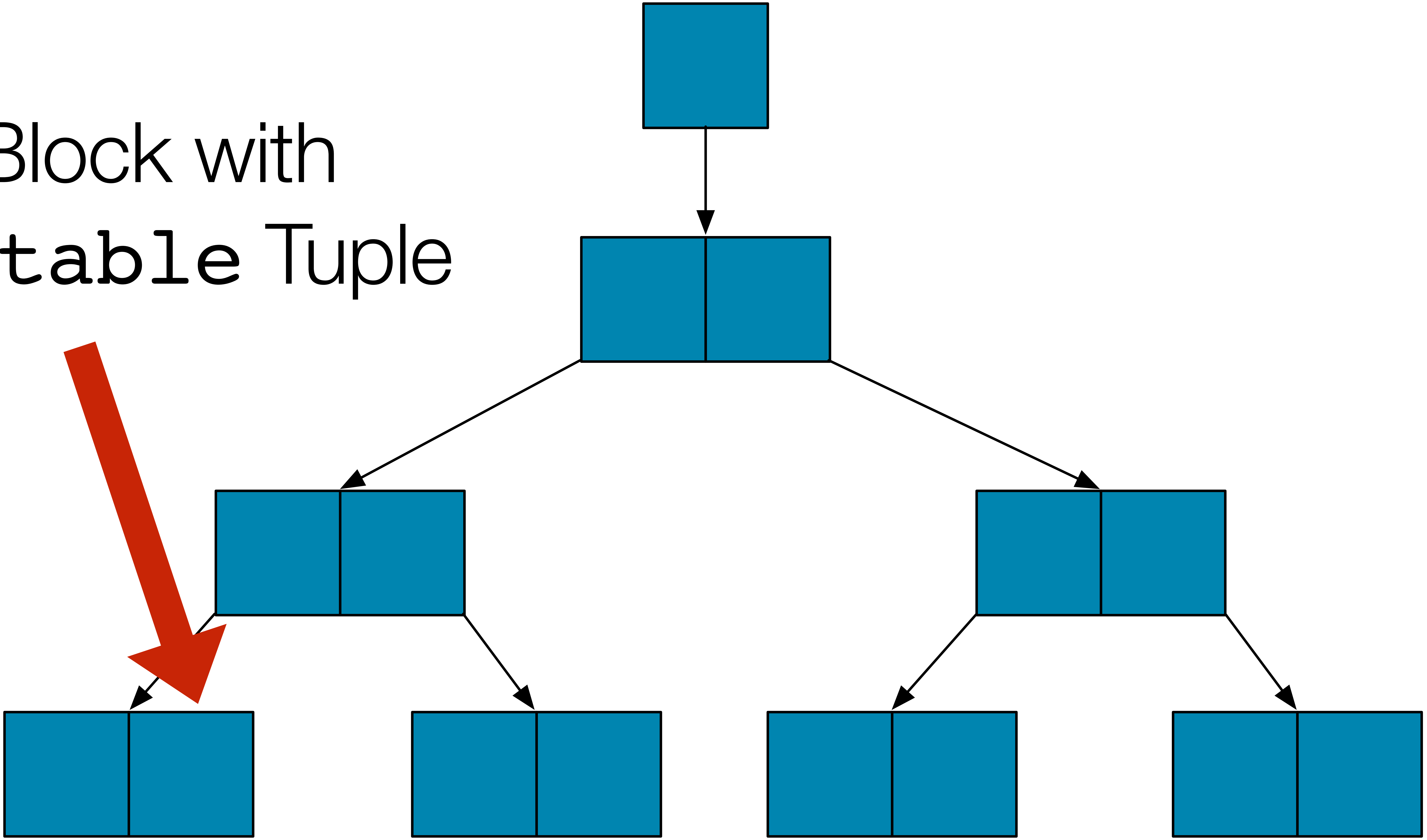
ZFS is magic until you know how it fits together



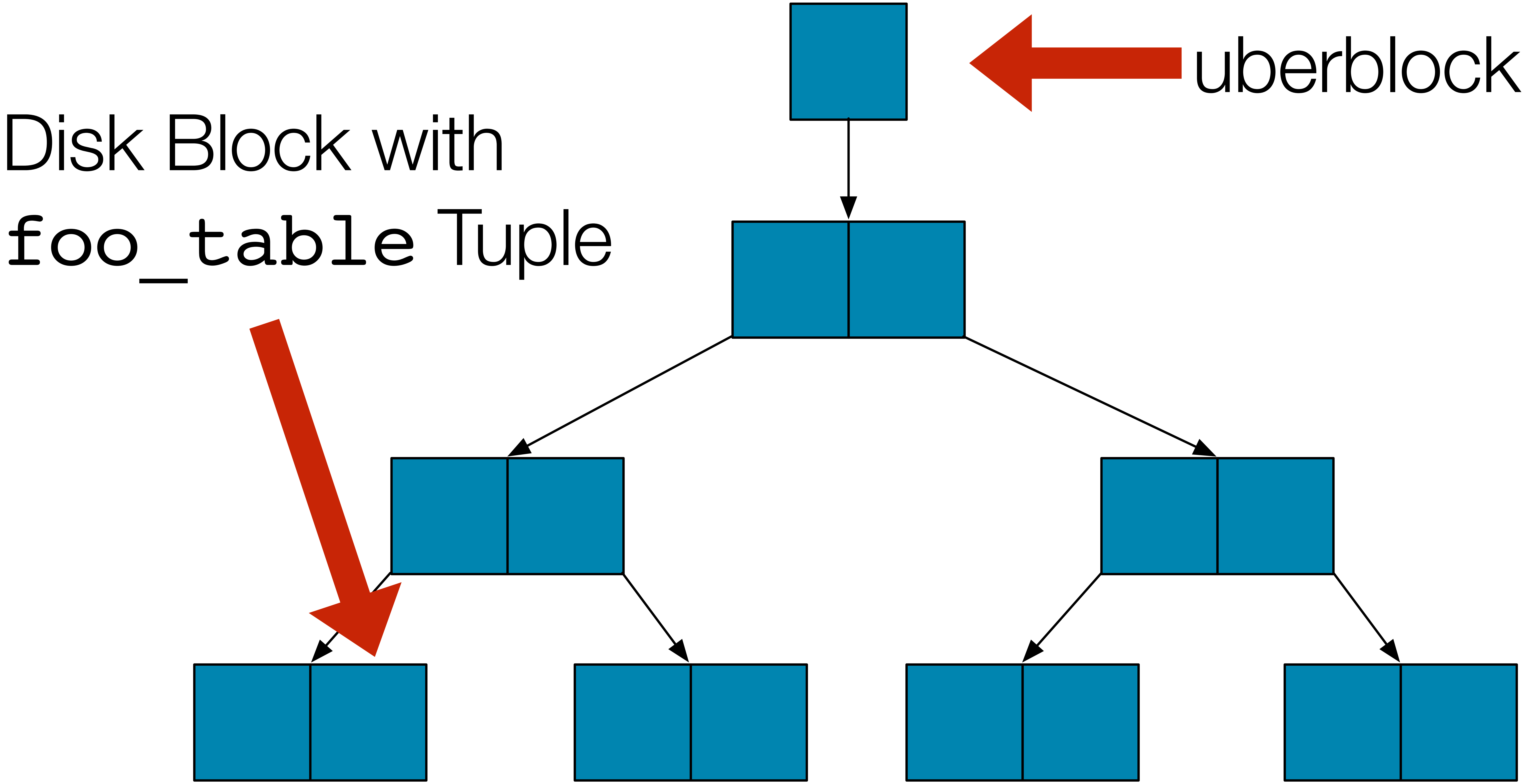
Log-Structured Filesystems: Top-Down

Log-Structured Filesystems: Top-Down

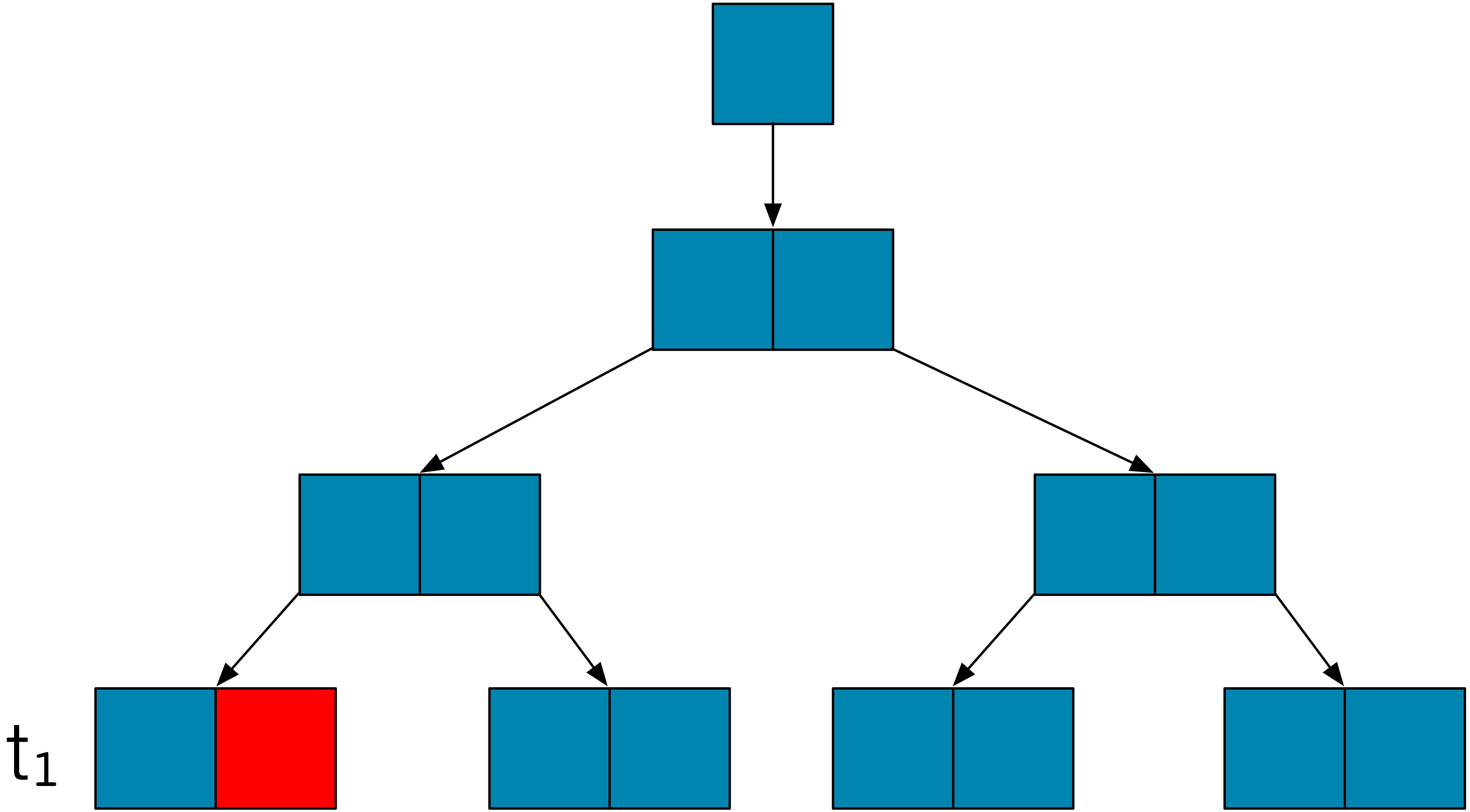
Disk Block with
`foo_table` Tuple



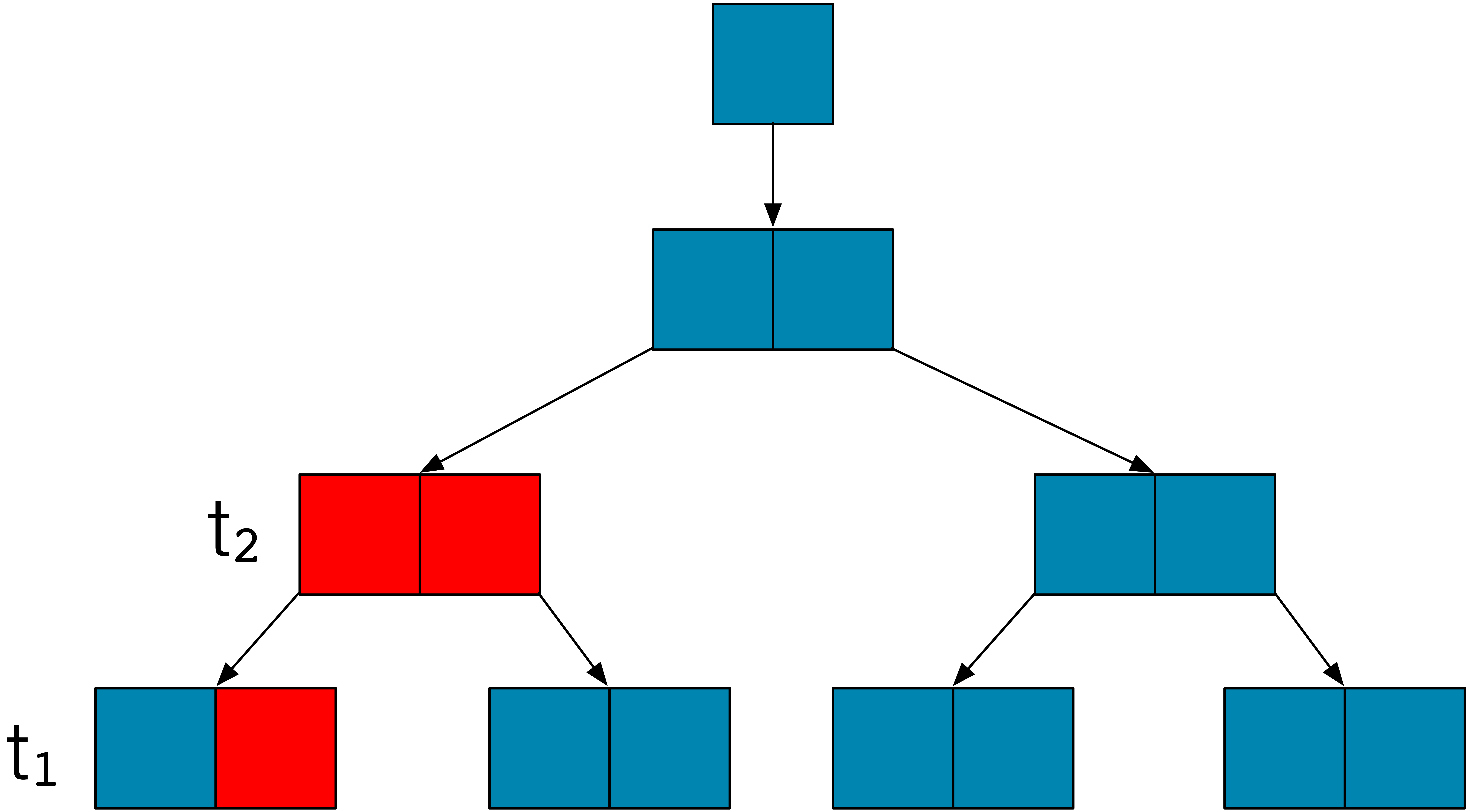
ZFS: User Data Block Lookup via ZFS Posix Layer



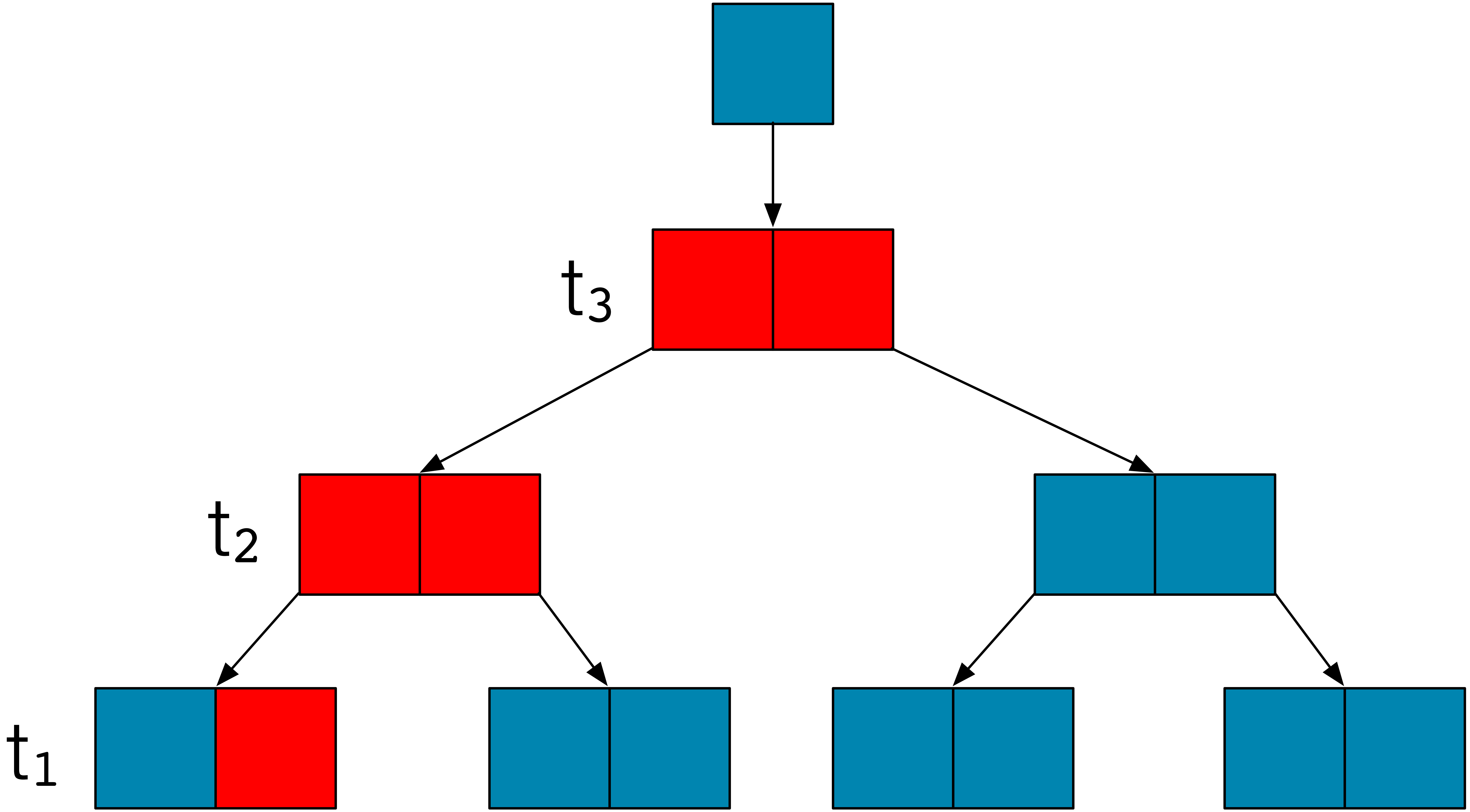
ZFS: User Data + File `dnode`



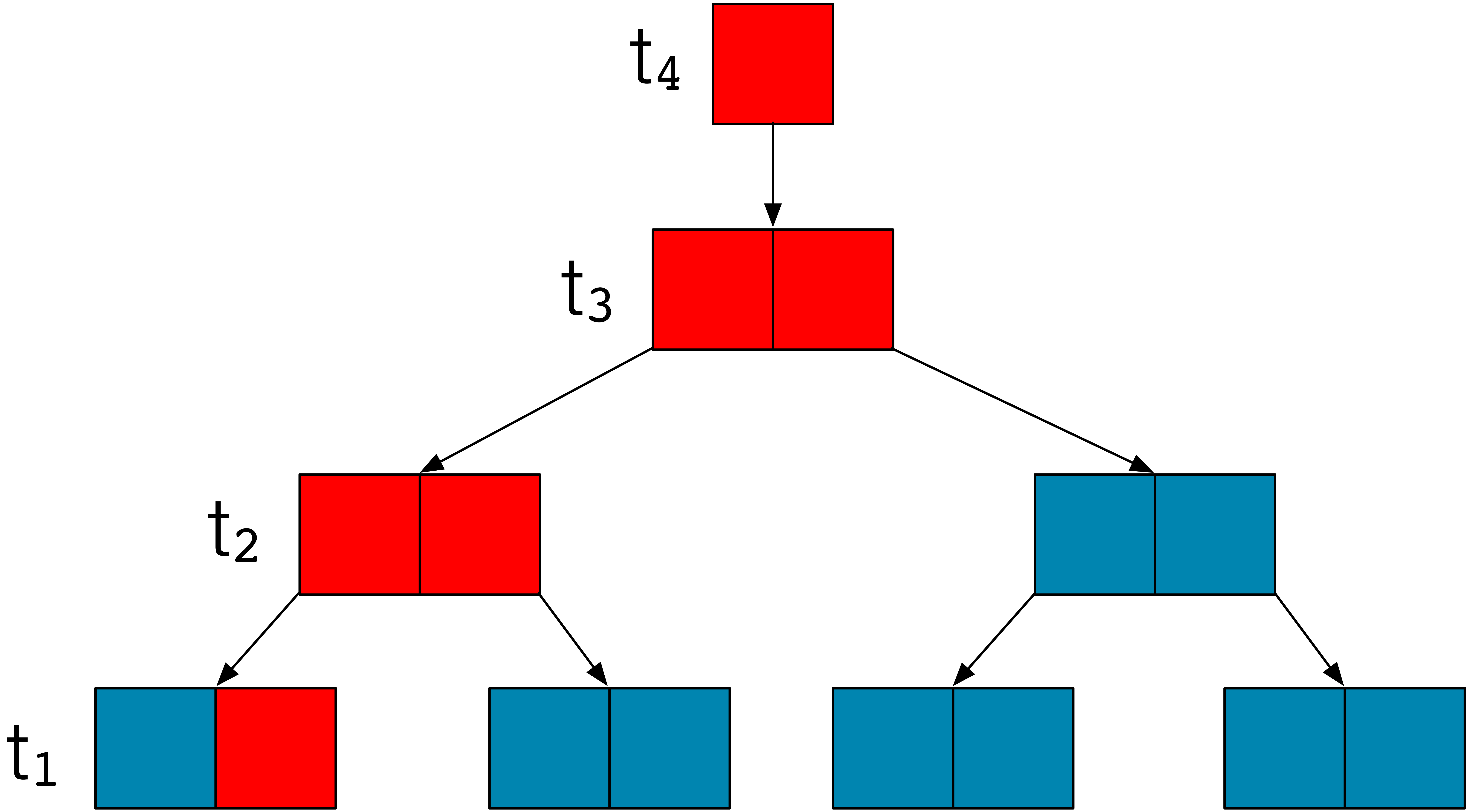
ZFS: Object Set



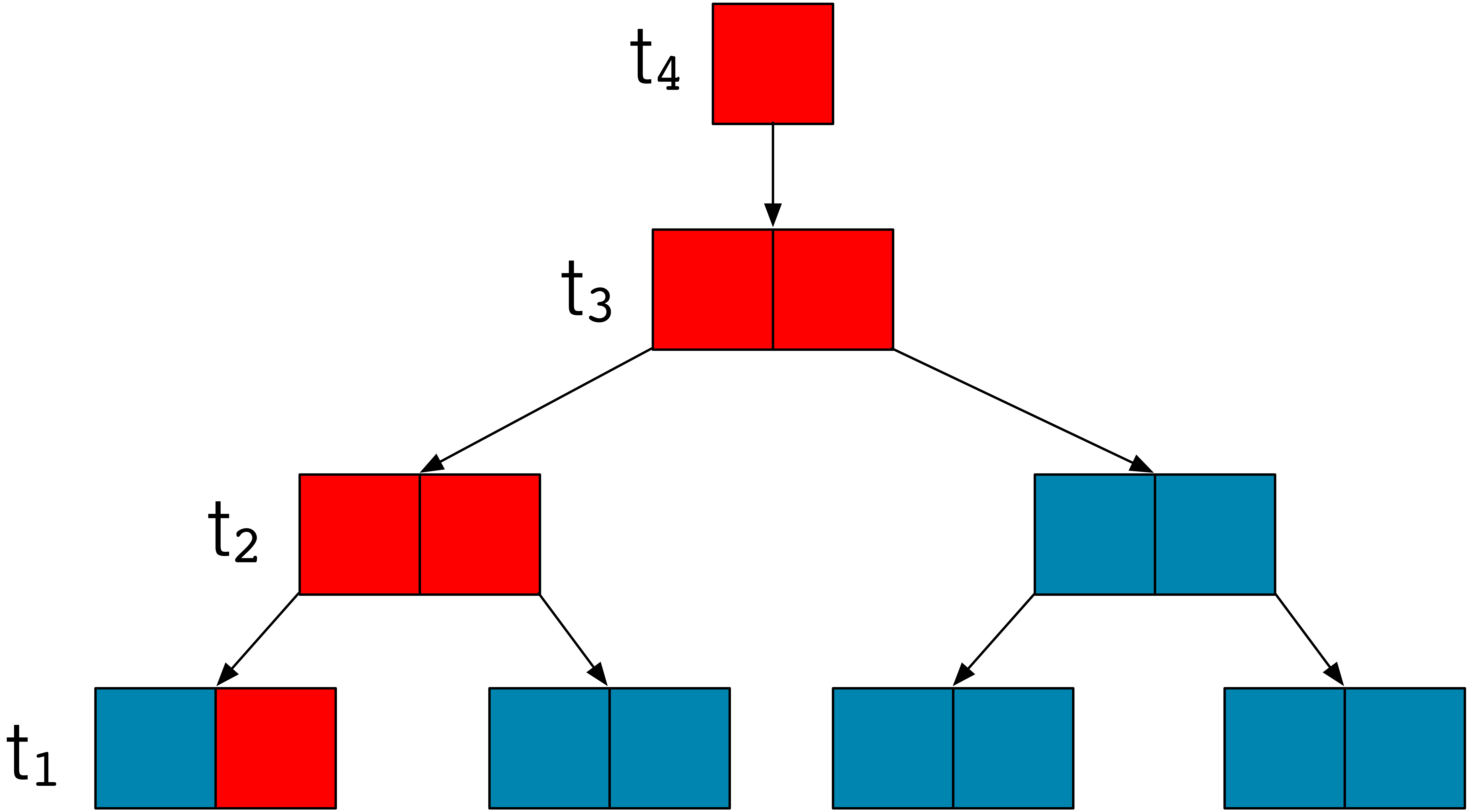
ZFS: Meta-Object Set Layer



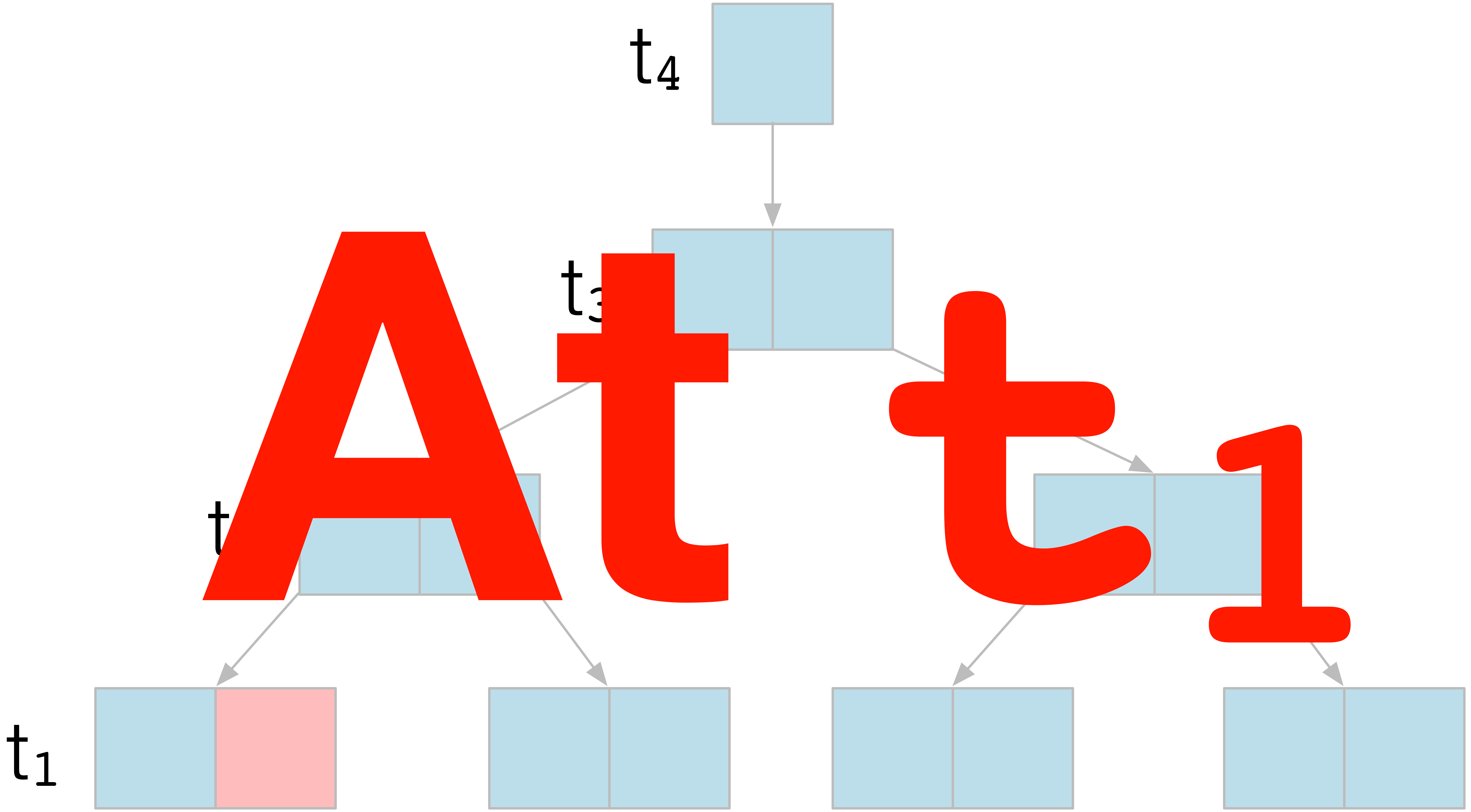
ZFS: Uberblock



At what point did the filesystem become inconsistent?



At what point could the filesystem become inconsistent?



How? I lied while explaining the situation. Alternate Truth.

Neglected to highlight **ZFS is Copy-On-Write** (read: knowingly committed perjury in front of a live audience)

How? I lied while explaining the situation. Alternate

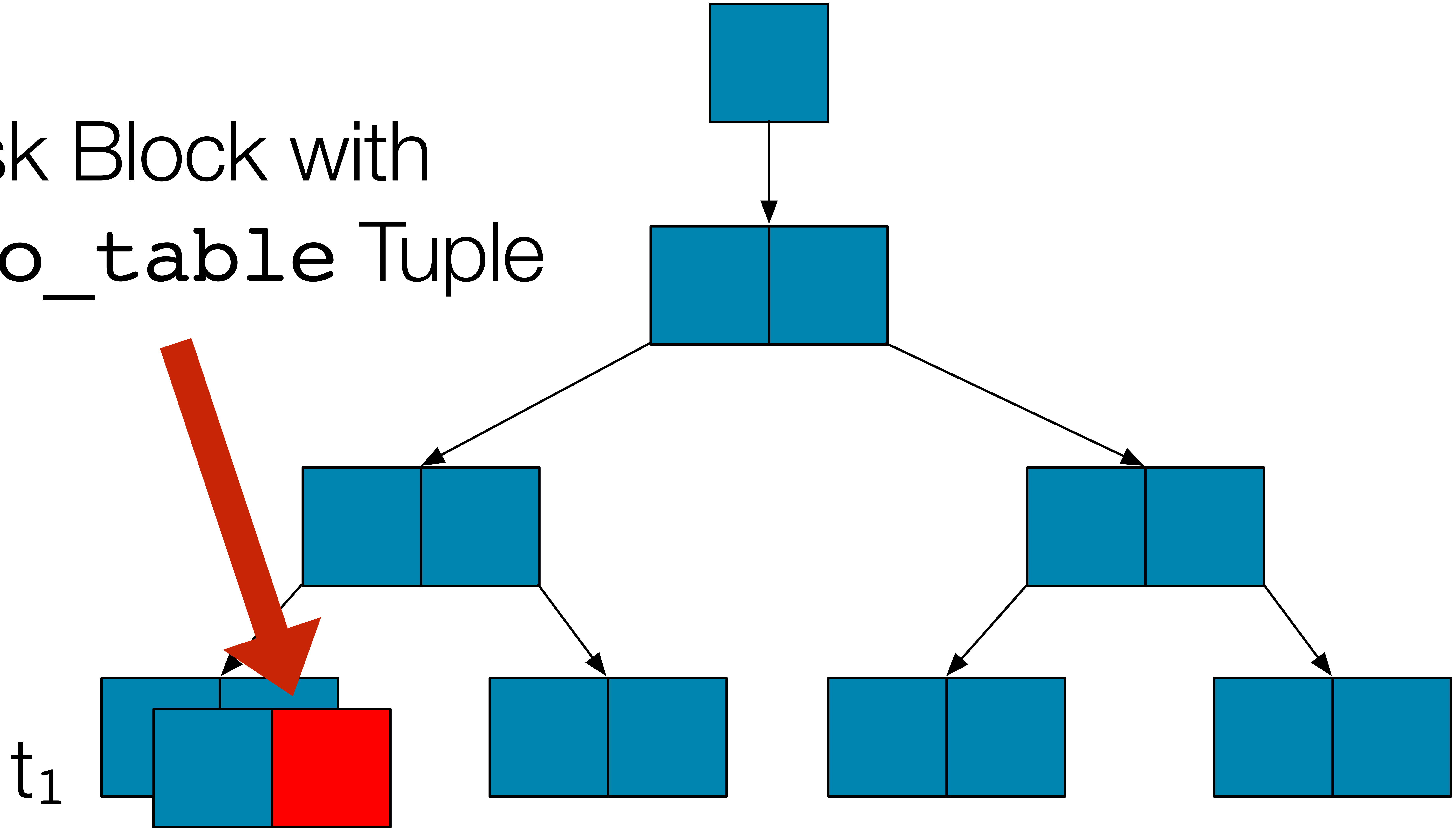
ZFS is Copy-On-Write

What what's not been deleted and on disk is immutable.

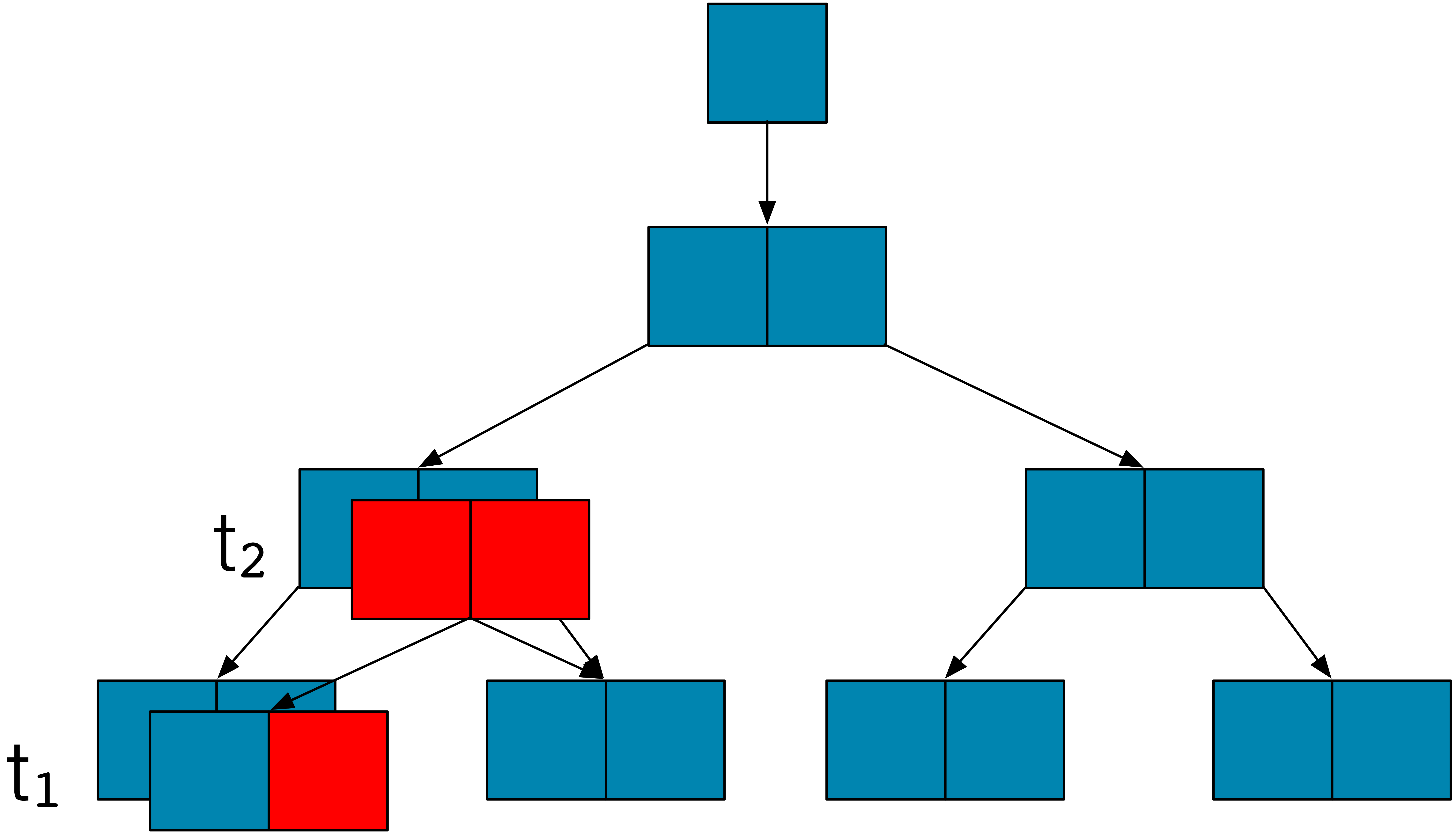
(read: I nearly committed perjury in front of a live audience by knowingly withholding vital information)

ZFS is Copy-On-Write

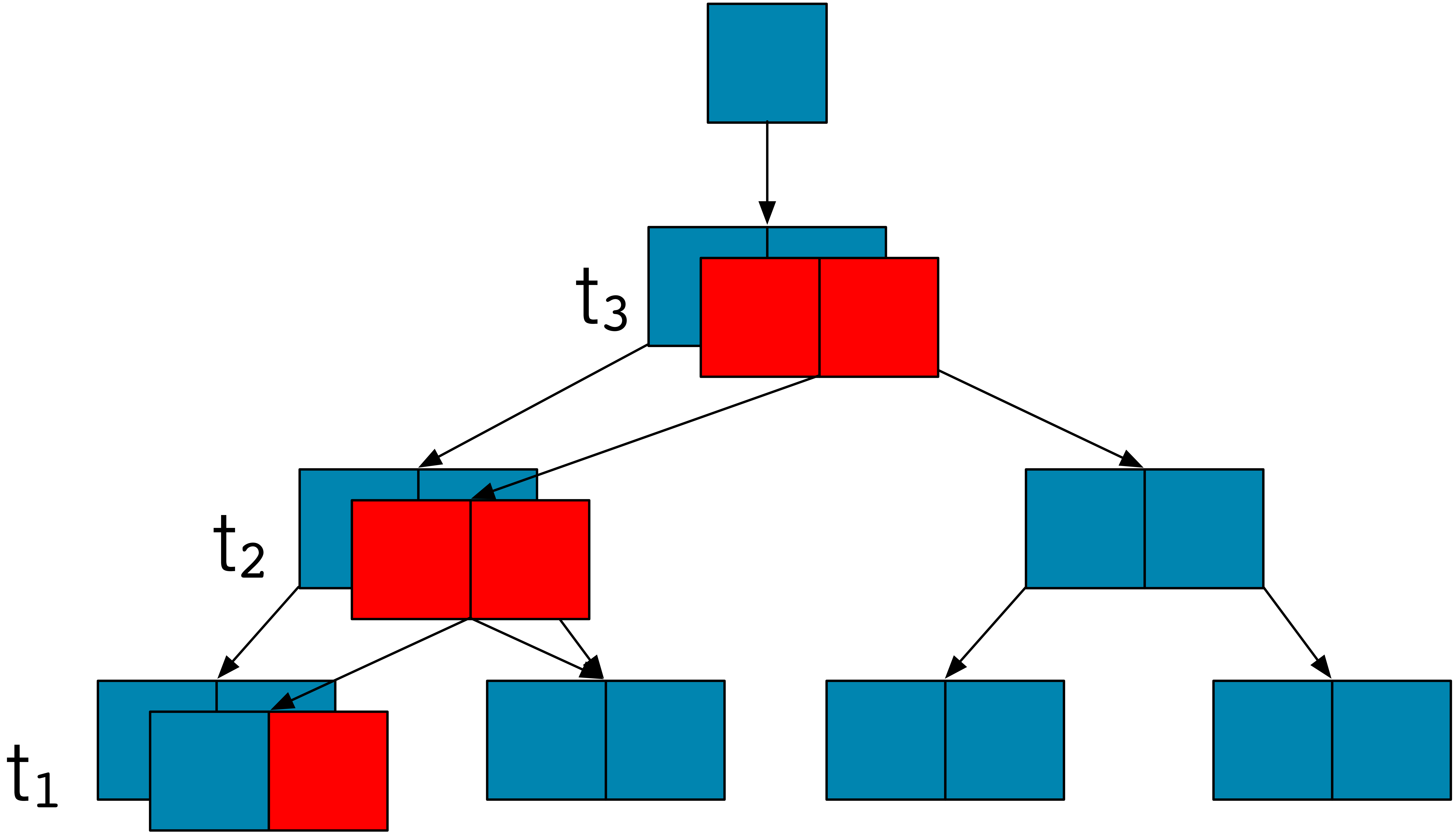
Disk Block with
`foo_table` Tuple



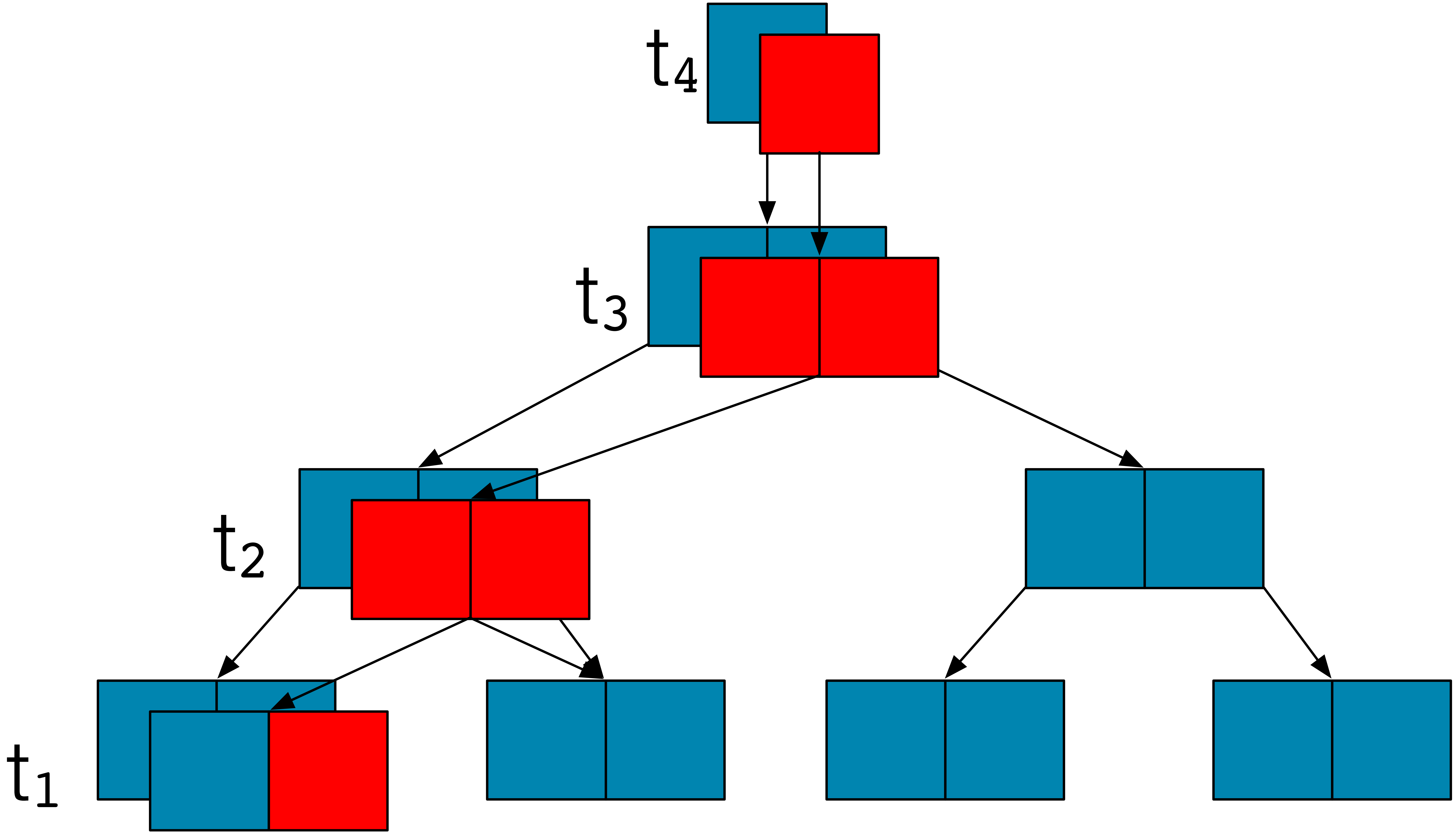
At what point did the filesystem become inconsistent?



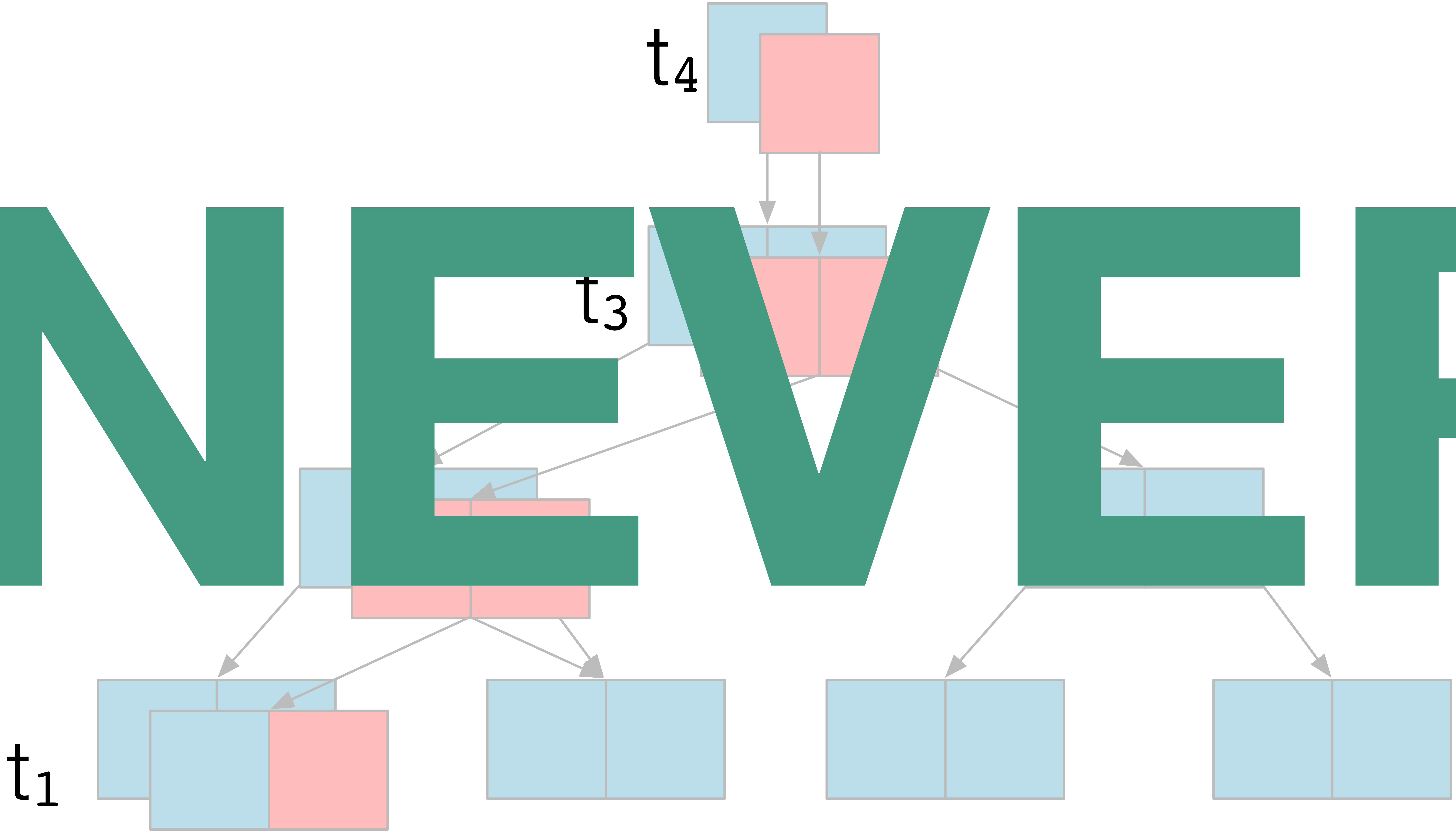
At what point did the filesystem become inconsistent?



At what point did the filesystem become inconsistent?



At what point could the filesystem become inconsistent?



NEVER

TIL about ZFS: Transactions and Disk Pages

- Transaction groups are flushed to disk ever `N` seconds (defaults to `5s`)
- A transaction group (`txg`) in ZFS is called a "checkpoint"
- User Data can be modified as its written to disk
- All data is checksummed
- Compression should be enabled by default

ZFS Tip: ALWAYS enable compression

```
$ zfs get compression
NAME         PROPERTY     VALUE     SOURCE
rpool        compression  off       default
rpool/root   compression  off       default
$ sudo zfs set compression=lz4 rpool
$ zfs get compression
NAME         PROPERTY     VALUE     SOURCE
rpool        compression  lz4       local
rpool/root   compression  lz4       inherited from rpool
```

- Across ~7PB of PostgreSQL and mixed workloads and applications: compression ratio of ~2.8:1 was the average.
- Have seen >100:1 compression on some databases (*cough* this data probably didn't belong in a database *cough*)
- Have seen as low as 1.01:1

ZFS Tip: ALWAYS enable compression

```
$ zfs get compression
NAME         PROPERTY     VALUE     SOURCE
rpool        compression  off       default
rpool/root   compression  off       default
$ sudo zfs set compression=lz4 rpool
$ zfs get compression
NAME         PROPERTY     VALUE     SOURCE
rpool        compression  lz4       local
rpool/root   compression  lz4       inherited from rpool
```

I have yet to see compression slow down benchmarking results or real world workloads. My experience is with:

- spinning rust (7.2K RPM, 10K RPM, and 15KRPM)
- fibre channel connected SANs
- SSDs
- NVME

ZFS Tip: ALWAYS enable compression

```
$ zfs get compressratio
NAME                 PROPERTY       VALUE  SOURCE
rpool                compressratio  1.64x  -
rpool/db             compressratio  2.58x  -
rpool/db/pgdb1-10    compressratio  2.61x  -
rpool/root           compressratio  1.62x  -
```

- Use `lz4` by default everywhere.
- Use `gzip-9` only for archive servers
- Never mix-and-match compression where you can't suffer the consequences of lowest-common-denominator performance
- Anxious to see `ZStandard` support (I'm looking at you Allan Jude)

ZFS Perk: Data Locality

- Data written at the same time is stored near each other because it's frequently part of the same record
- Data can now pre-fault into kernel cache (ZFS ARC) by virtue of the temporal adjacency of the related `pwrite(2)` calls
- Write locality + `compression=lz4` + `pg_repack` == PostgreSQL Dream Team

ZFS Perk: Data Locality

- Data written at the same time is stored near each other because it's frequently part of the same record
- Data can now pre-fault into kernel cache (ZFS ARC) by virtue of the temporal adjacency of the related `pwrite(2)` calls
- Write locality + `compression=lz4` + `pg_repack` == PostgreSQL Dream Team

If you don't know what `pg_repack` is, figure out how to move into a database environment that supports `pg_repack` and use it regularly.

https://reorg.github.io/pg_repack/ && https://github.com/reorg/pg_repack/

Extreme ZFS Warning: Purge all memory of `dedup`

- This is not just my recommendation, it's also from the community and author of the feature.
- These are not the droids you are looking for
- Do not pass Go
- Do not collect $200
- Go straight to system unavailability jail
- The feature works, but you run the risk of bricking your ZFS server.

Ask after if you are curious, but here's a teaser:

What do you do if the `dedup` hash tables don't fit in RAM?

Bitrot is a Studied Phenomena

A Large-Scale Study of Flash Memory Failures in the Field

Justin Meza
Carnegie Mellon University
meza@cmu.edu

Qiang Wu
Facebook, Inc.
qwu@fb.com

Sanjeev Kumar
Facebook, Inc.
skumar@fb.com

Onur Mutlu
Carnegie Mellon University
onur@cmu.edu

ABSTRACT

Servers use flash memory based solid state drives (SSDs) as a high-performance alternative to hard disk drives to store persistent data. Unfortunately, recent increases in flash density have also brought about decreases in chip-level reliability. In a data center environment, flash-based SSD failures can lead to downtime and, in the worst case, data loss. As a result, it is important to understand flash memory reliability characteristics over flash lifetime in a realistic production data center environment running modern applications and system software.

This paper presents the first large-scale study of flash-based SSD reliability in the field. We analyze data collected across a majority of flash-based solid state drives at Facebook data centers over nearly four years and many millions of operational hours in order to understand failure properties and trends of flash-based SSDs. Our study considers a variety of SSD characteristics, including: the amount of data written to and read from flash chips; how data is mapped within the SSD address space; the amount of data copied, erased, and discarded by the flash controller; and flash board temperature and bus power.

Based on our field analysis of how flash memory errors manifest when running modern workloads on modern SSDs, this paper is the first to make several major observations: (1) *SSD failure rates do not increase monotonically with flash chip wear*; instead they go through several distinct periods corresponding to how failures emerge and are subsequently detected, (2) the effects of read disturbance errors are *not* prevalent in the field, (3) sparse logical data layout across an SSD's physical address space (e.g., non-contiguous data), as measured by the amount of metadata required to track logical address translations stored in an SSD-internal DRAM buffer, can greatly affect SSD failure rate, (4) higher temperatures lead to higher failure rates, but techniques that throttle SSD operation appear to greatly *reduce* the negative reliability impact of higher temperatures, and (5) data written by the operating system to flash-based SSDs does *not* always accurately

Categories and Subject Descriptors

B.3.4. [**Hardware**]: Memory Structures—*Reliability, Testing, and Fault-Tolerance*

Keywords

flash memory; reliability; warehouse-scale data centers

1. INTRODUCTION

Servers use flash memory for persistent storage due to the low access latency of flash chips compared to hard disk drives. Historically, flash capacity has lagged behind hard disk drive capacity, limiting the use of flash memory. In the past decade, however, advances in NAND flash memory technology have increased flash capacity by more than 1000×. This rapid increase in flash capacity has brought both an increase in flash memory use and a decrease in flash memory reliability. For example, the number of times that a cell can be reliably programmed and erased before wearing out and failing dropped from 10,000 times for 50 nm cells to only 2,000 times for 20 nm cells [28]. This trend is expected to continue for newer generations of flash memory. Therefore, if we want to improve the operational lifetime and reliability of flash memory-based devices, we must first fully understand their failure characteristics.

In the past, a large body of prior work examined the failure characteristics of flash cells in controlled environments using small numbers e.g., tens) of raw flash chips (e.g., [36, 23, 21, 27, 22, 25, 16, 33, 14, 5, 18, 4, 24, 40, 41, 26, 31, 30, 37, 6, 11, 10, 7, 13, 9, 8, 12, 20]). This work quantified a variety of flash cell failure modes and formed the basis of the community's understanding of flash cell reliability. Yet prior work was limited in its analysis because these studies: (1) were conducted on small numbers of raw flash chips accessed in adversarial manners over short amounts of time, (2) did not examine failures when using real applications running on modern servers and

Bitrot is a Studied Phenomena

3.1 Bit Error Rate

The bit error rate (BER) of an SSD is the rate at which errors occur relative to the amount of information that is transmitted from/to the SSD. BER can be used to gauge the reliability of data transmission across a medium. We measure the uncorrectable bit error rate (UBER) from the SSD as:

$$UBER = \frac{Uncorrectable\ \ errors}{Bits\ \ accessed}$$

For flash-based SSDs, UBER is an important reliability metric that is related to the SSD lifetime. SSDs with high UBERs are expected to have more failed cells and encounter more (severe) errors that may potentially go undetected and corrupt

Bitrot is a Studied Phenomena

data than SSDs with low UBERs. Recent work by Grupp et al. examined the BER of *raw* MLC flash chips (without performing error correction in the flash controller) in a controlled environment [20]. They found the raw BER to vary from 1×10^{-1} for the least reliable flash chips down to 1×10^{-8} for the most reliable, with most chips having a BER in the 1×10^{-6} to 1×10^{-8} range. Their study did *not* analyze the effects of the use of chips *in SSDs under real workloads and system software.*

Table 1 shows the UBER of the platforms that we examine. We find that for flash-based SSDs used in servers, the UBER ranges from 2.6×10^{-9} to 5.1×10^{-11}. While we expect that the UBER of the SSDs that we measure (which correct small errors, perform wear leveling, and are not at the end of their rated life but still being used in a production environment) should be less than the raw chip BER in Grupp et al.'s study (which did not correct any errors in the flash controller, exercised flash chips until the end of their rated life, and accessed flash chips in an adversarial manner), we find that in some cases the BERs were within around one order of magnitude of each other. For example, the UBER of Platform B in our study, 2.6×10^{-9}, comes close to the lower end of the raw chip BER range reported in prior work, 1×10^{-8}.

Bitrot is a Studied Phenomena

Figure 2 (bottom) shows the average yearly uncorrectable error rate among SSDs within the different platforms – the sum of errors that occurred on all servers within a platform over 12 months ending in November 2014 divided by the number of servers in the platform. The yearly rates of uncorrectable errors on the SSDs we examined range from 15,128 for Platform D to 978,806 for Platform B. The older Platforms A and B have a higher error rate than the younger Platforms C through F, suggesting that the incidence of uncorrectable errors increases as SSDs are utilized more. We will examine this relationship further in Section 4.

Platform B has a much higher average yearly rate of uncorrectable errors (978,806) compared to the other platforms (the second highest amount, for Platform A, is 106,678). We find that this is due to a small number of SSDs having a much

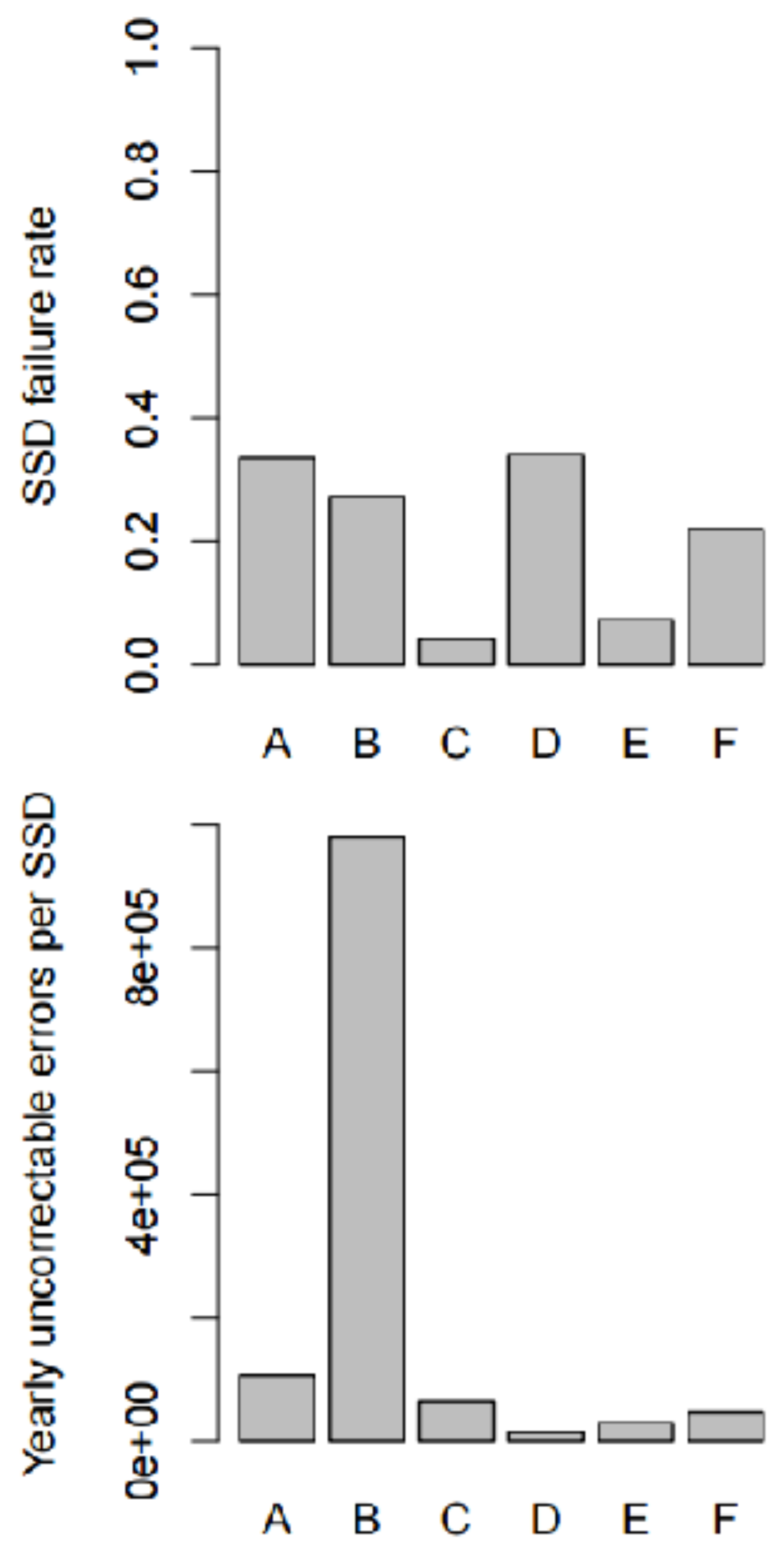


Figure 2: The failure rate (top) and average yearly uncorrectable errors per SSD (bottom), among SSDs within each platform.

TIL: Bitrot is here

- TL;DR: 4.2% -> 34% of SSDs have one UBER per year

TIL: Bitrot Roulette

`(1-(1-uberRate)^(numDisks))` = Probability of UBER/server/year

`(1-(1-0.042)^(20))` = 58%

`(1-(1-0.34)^(20))` = 99.975%

Highest quality SSD drives on the market

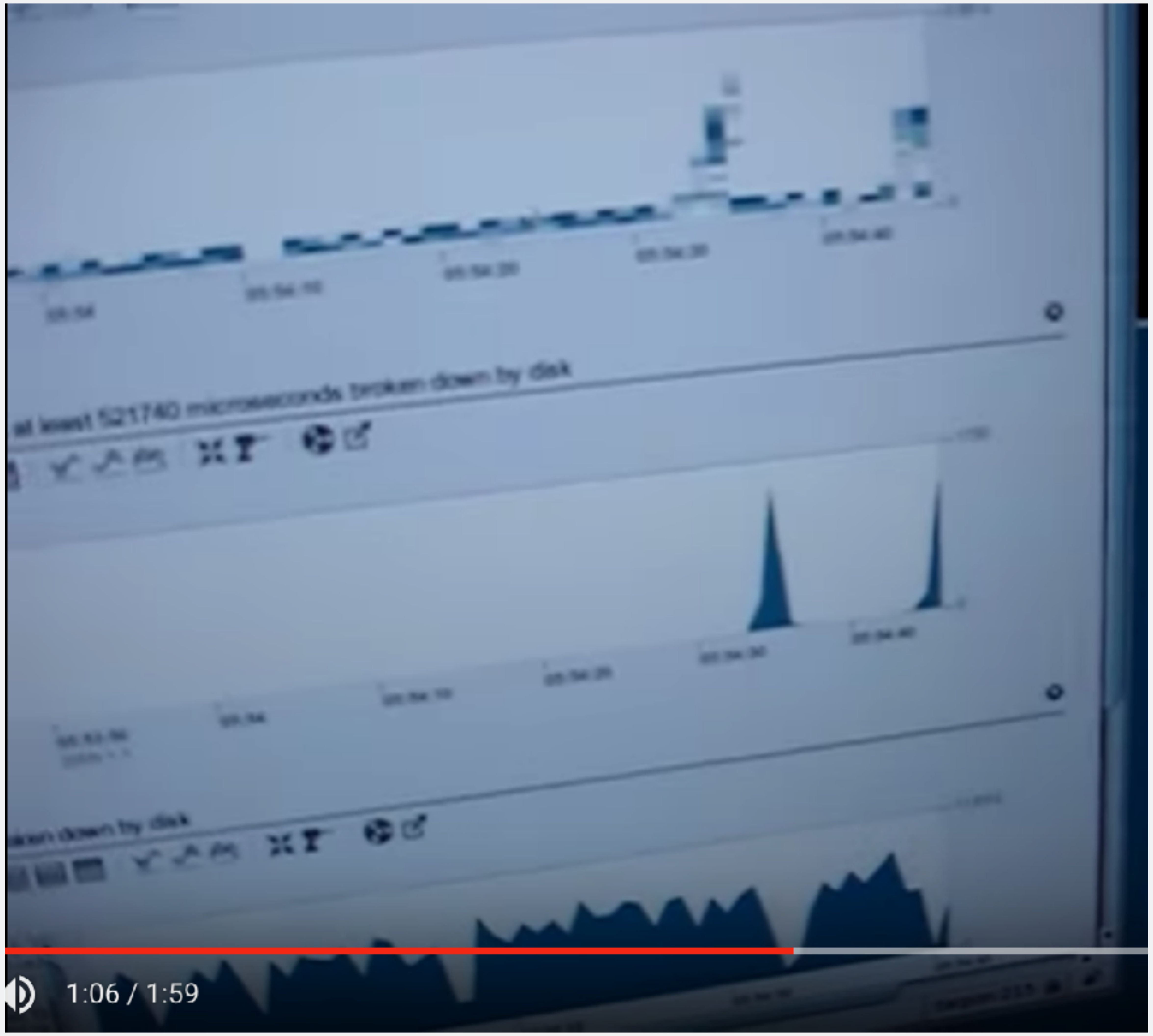
Lowest quality commercially viable SSD drives on the market

Causes of bitrot are Internal and External

External Factors for UBER on SSDs:

- Temperature
- Bus Power Consumption
- Data Written by the System Software
- Workload changes due to SSD failure

...except maybe they can.



Take Care of your bits

```
$ zpool status tank | head -n 3
  pool: tank
 state: ONLINE
  scan: scrub repaired 4.50K in 53h44m with 0 errors on Tue May 26 21:36:26 2015
```

Answer their cry for help.

Take Care of your bits

Similar studies and research exist for:

- Fibre Channel
- SAS
- SATA
- Tape
- SANs
- Cloud Object Stores

So what about PostgreSQL?

"...I told you all of that, so I can tell you this..."

ZFS Terminology: VDEV

`VDEV` | vē-dēv

noun

a virtual device

- Physical drive redundancy is handled at the VDEV level
- Zero or more physical disks arranged like a RAID set:
  - `mirror`
  - `stripe`
  - `raidz`
  - `raidz2`
  - `raidz3`

ZFS Terminology: `zpool`

`zpool` | zē-pōōl

noun

an abstraction of physical storage made up of a set of VDEVs

Loose a VDEV, loose the zpool.

ZFS Terminology: `ZPL`

`ZPL` | zē-pē-el

noun

ZFS POSIX Layer

- Layer that handles the impedance mismatch between POSIX filesystem semantics and the ZFS "object database."

ZFS Terminology: `ZIL`

`ZIL` | zil

noun

ZFS Intent Log

- The ZFS analog of PostgreSQL's WAL
- If you use a ZIL:
  - Use disks that have low-latency writes
  - Mirror your ZIL
  - If you loose your ZIL, whatever data had not made it to the main data disks will be lost. The PostgreSQL equivalent of: `rm -rf pg_xlog/`

ZFS Terminology: `ARC`

`ARC` | ärk

noun

Adaptive Replacement Cache

- ZFS's page cache
- ARC will grow or shrink to match use up all of the available memory

TIP: Limit ARC's max size to a percentage of physical memory minus the `shared_buffer` cache for PostgreSQL minus the kernel's memory overhead.

ZFS Terminology: Datasets

`dataset` | dædə ˌsɛt

noun

A filesystem or volume ("`zvol`")

- A ZFS filesystem dataset uses the underlying `zpool`
- A dataset belongs to one and only one `zpool`
- Misc tunables, including compression and quotas are set on the dataset level

ZFS Terminology: The Missing Bits

`ZAP`	ZFS Attribute Processor
`DMU`	Data Management Unit
`DSL`	Dataset and Snapshot Layer
`SPA`	Storage Pool Allocator
`ZVOL`	ZFS Volume
`ZIO`	ZFS I/O
`RAIDZ`	RAID with variable-size stripes
`L2ARC`	Level 2 Adaptive Replacement Cache
`record`	unit of user data, think RAID stripe size

Storage Management

```
$ sudo zfs list
NAME           USED  AVAIL  REFER  MOUNTPOINT
rpool          818M  56.8G    96K  none
rpool/root     817M  56.8G   817M  /
$ ls -lA -d /db
ls: cannot access '/db': No such file or directory
$ sudo zfs create rpool/db -o mountpoint=/db
$ sudo zfs list
NAME           USED  AVAIL  REFER  MOUNTPOINT
rpool          818M  56.8G    96K  none
rpool/db        96K  56.8G    96K  /db
rpool/root     817M  56.8G   817M  /
$ ls -lA /db
total 9
drwxr-xr-x  2 root root  2 Mar  2 18:06 ./
drwxr-xr-x 22 root root 24 Mar  2 18:06 ../
```

Storage Management

- Running out of disk space is bad, m'kay?
- Block file systems reserve ~8% of the disk space above 100%
- At ~92% capacity the performance of block allocators change from "performance optimized" to "space optimized" (read: performance "drops").

Storage Management

- Running out of disk space is bad, m'kay?
- Block file systems reserve ~8% of the disk space above 100%
- At ~92% capacity the performance of block allocators change from "performance optimized" to "space optimized" (read: performance "drops").

ZFS doesn't have an artificial pool of free space: you have to manage that yourself.

Storage Management

```
$ sudo zpool list -H -o size
59.6G
$ sudo zpool list
```

The pool should never consume more than 80% of the available space

Storage Management

```
$ sudo zfs set quota=48G rpool/db
$ sudo zfs get quota rpool/db
NAME      PROPERTY  VALUE  SOURCE
rpool/db  quota      48G   local
$ sudo zfs list
NAME          USED  AVAIL  REFER  MOUNTPOINT
rpool         818M  56.8G    96K  none
rpool/db       96K  48.0G    96K  /db
rpool/root    817M  56.8G   817M  /
```

Dataset Tuning Tips

- Disable `atime`
- Enable `compression`
- Tune the `recordsize`
- Consider tweaking the `primarycache`

ZFS Dataset Tuning

```
# zfs get atime,compression,primarycache,recordsize rpool/db
NAME      PROPERTY      VALUE          SOURCE
rpool/db  atime         on             inherited from rpool
rpool/db  compression   lz4            inherited from rpool
rpool/db  primarycache  all            default
rpool/db  recordsize    128K           default
# zfs set atime=off rpool/db
# zfs set compression=lz4 rpool/db
# zfs set recordsize=16K rpool/db
# zfs set primarycache=metadata rpool/db
# zfs get atime,compression,primarycache,recordsize rpool/db
NAME      PROPERTY      VALUE          SOURCE
rpool/db  atime         off            local
rpool/db  compression   lz4            local
rpool/db  primarycache  metadata       local
rpool/db  recordsize    16K            local
```

Discuss: `recordsize=16K`

- Pre-fault next page: useful for sequential scans
- With **`compression=lz4`**, reasonable to expect ~3-4x pages worth of data in a single ZFS record

Anecdotes and Recommendations:

- Performed better in most workloads vs ZFS's prefetch
- Disabling prefetch isn't necessary, tends to still be a net win
- Monitor arc cache usage

Discuss: `primarycache=metadata`

- metadata instructs ZFS's ARC to only cache metadata (e.g. `dnode` entries), not page data itself
- Default: cache all data

Two different recommendations based on benchmark workloads:

- Enable **`primarycache=all`** where working set exceeds RAM
- Enable **`primarycache=metadata`** where working set fits in RAM

Discuss: `primarycache=metadata`

- metadata instructs ZFS's ARC to only cache metadata (e.g. **`dnode`** entries), not page data itself
- Default: cache all data
- Double-caching happens

Two different recommendations based on benchmark workloads:

- Enable **`primarycache=all`** where working set exceeds RAM
- Enable **`primarycache=metadata`** where working set fits in RAM

Reasonable Default anecdote: Cap max ARC size ~15%-25% physical RAM + ~50% RAM **`shared_buffers`**

Performance Wins

2-4µs/`pwrite(2)`!!

```
# dtrace -s vfs-io-postgres.d
Latencies (ns)
postgres Write

           value  ------------- Distribution ------------- count
            1024 |                                         0
            2048 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@           1325
            4096 |@@@@@@                                   267
            8192 |@@                                       72
           16384 |                                         0
           32768 |                                         0
           65536 |                                         19
          131072 |                                         2
          262144 |                                         0
```

Performance Wins

```
# zpool iostat tank 1
                     capacity     operations    bandwidth
pool     alloc    free    read   write    read    write
-----  -----   -----  -----   -----   -----    -----
tank     958G   9.94T      0    210K    1022     330M
tank     958G   9.94T      1    207K   4.99K     326M
tank     958G   9.94T     32   30.5K   79.9K    46.9M
tank     958G   9.94T     22   9.62K    202K    15.9M
tank     958G   9.94T     15   10.2K    169K    16.5M
tank     958G   9.94T     36   10.5K    198K    14.9M
tank     958G   9.94T      6   10.8K   39.4K    17.4M
tank     958G   9.94T     12    189K    209K     298M
tank     958G   9.94T      1    210K   7.96K     340M
tank     958G   9.94T     10    218K   23.0K     355M
tank     958G   9.94T      2    224K   4.49K     359M
tank     958G   9.94T      6    228K   12.5K     367M
tank     958G   9.94T      7    140K   53.4K     225M
tank     958G   9.94T      9   26.9K   40.9K    44.0M
tank     958G   9.94T      0   9.43K       0    13.9M
tank     958G   9.94T      0   9.69K       0    16.3M
tank     958G   9.94T      1   74.0K   3.49K     120M
tank     958G   9.94T      6    226K   17.0K     366M
tank     958G   9.94T      0    225K       0     385M
tank     958G   9.94T      0    176K       0     515M
tank     958G   9.94T      0   84.7K       0     382M
tank     958G   9.94T      0   39.6K       0     163M
```

Performance Wins

```
# zpool iostat tank 1
                         capacity       operations    bandwidth
 pool    alloc      free     read    write      read       write
 -----   -----     -----    -----    -----     -----       -----
 tank     958G     9.94T        0     210K      1022        330M
 tank     958G     9.94T        1     207K     4.99K        326M
 tank     958G     9.94T       32    30.5K     79.9K       46.9M
 tank     958G     9.94T       22    9.62K      202K       15.9M
 tank     958G     9.94T       15    10.2K      169K       16.5M
 tank     958G     9.94T       36    10.5K      198K       14.9M
 tank     958G     9.94T        6    10.8K     39.4K       17.4M
 tank     958G     9.94T       12     189K      209K        298M
 tank     958G     9.94T        1     210K     7.96K        340M
 tank     958G     9.94T       10     218K     23.0K        355M
 tank     958G     9.94T        2     224K     4.49K        359M
 tank     958G     9.94T        6     228K     12.5K        367M
 tank     958G     9.94T        7     140K     53.4K        225M
 tank     958G     9.94T        9    26.9K     40.9K       44.0M
 tank     958G     9.94T        0    9.43K         0       13.9M
 tank     958G     9.94T        0    9.69K         0       16.3M
 tank     958G     9.94T        1    74.0K     3.49K        120M
 tank     958G     9.94T        6     226K     17.0K        366M
 tank     958G     9.94T        0     225K         0        385M
 tank     958G     9.94T        0     176K         0        515M
 tank     958G     9.94T        0    84.7K         0        382M
 tank     958G     9.94T        0    39.6K         0        163M
```

Performance Wins

```
# zpool iostat tank 1
                        capacity     operations    bandwidth
  pool     alloc     free   read    write    read    write
  -----    -----    -----  -----    -----   -----    -----
  tank      958G    9.94T      0     210K    1022     330M
  tank      958G    9.94T      1     207K   4.99K     326M
  tank      958G    9.94T     32    30.5K   79.9K    46.9M
  ta[illegible]
  ta[illegible]
  tank      958G    9.94T     36    10.5K    198K    14.9M
  tank      958G    9.94T      6    10.8K   39.4K    17.4M
  tank      958G    9.94T     12     189K    209K     298M
  tank      958G    9.94T      1     210K   7.96K     340M
  tank      958G    9.94T     10     218K   23.0K     355M
  tank      958G    9.94T      2     224K   4.49K     359M
  tank      958G    9.94T      6     228K   12.5K     367M
  tank      958G    9.94T      7     140K   53.4K     225M
  tank      958G    9.94T      9    26.9K   40.9K    44.0M
  tank      958G    9.94T      0    9.43K       0    13.9M
  tank      958G    9.94T      0    9.69K       0    16.3M
  tank      958G    9.94T      1    74.0K   3.49K     120M
  tank      958G    9.94T      6     226K   17.0K     366M
  tank      958G    9.94T      0     225K       0     385M
  tank      958G    9.94T      0     176K       0     515M
  tank      958G    9.94T      0    84.7K       0     382M
  tank      958G    9.94T      0    39.6K       0     163M
```

P.S. This was observed on 10K RPM spinning rust.

ZFS Always has your back

- ZFS will checksum every read from disk
- A failed checksum will result in a fault and automatic data reconstruction
- Scrubs do background check of every record
- Schedule periodic scrubs
  - Frequently for new and old devices
  - Infrequently for devices in service between 6mo and 2.5yr

PSA: The "Compressed ARC" feature was added to catch checksum errors in **RAM**

Checksum errors are an early indicator of failing disks

Schedule Periodic Scrubs

```
# zpool status
  pool: rpool
 state: ONLINE
  scan: none requested
config:

    NAME        STATE     READ WRITE CKSUM
    rpool       ONLINE       0     0     0
      sda1      ONLINE       0     0     0

errors: No known data errors
# zpool scrub rpool
# zpool status
  pool: rpool
 state: ONLINE
  scan: scrub in progress since Fri Mar  3 20:41:44 2017
    753M scanned out of 819M at 151M/s, 0h0m to go
    0 repaired, 91.97% done
config:

    NAME        STATE     READ WRITE CKSUM
    rpool       ONLINE       0     0     0
      sda1      ONLINE       0     0     0

errors: No known data errors
# zpool status
  pool: rpool
 state: ONLINE
  scan: scrub repaired 0 in 0h0m with 0 errors on Fri Mar  3 20:41:49 2017
```

Non-zero on any of these values is bad™

One dataset per database

- Create one ZFS dataset per database instance
- General rules of thumb:
  - Use the <u>same</u> dataset for **`$PGDATA/`** and **`pg_xlogs/`**
  - Set a reasonable quota
  - Optional: reserve space to guarantee minimal available space

Checksum errors are an early indicator of failing disks

One dataset per database

```
# zfs list
NAME            USED  AVAIL  REFER  MOUNTPOINT
rpool           819M  56.8G    96K  none
rpool/db        160K  48.0G    96K  /db
rpool/root      818M  56.8G   818M  /
# zfs create rpool/db/pgdb1
# chown postgres:postgres /db/pgdb1
# zfs list
NAME                USED  AVAIL  REFER  MOUNTPOINT
rpool               819M  56.8G    96K  none
rpool/db            256K  48.0G    96K  /db
rpool/db/pgdb1       96K  48.0G    96K  /db/pgdb1
rpool/root          818M  56.8G   818M  /
# zfs set reservation=1G rpool/db/pgdb1
# zfs list
NAME                USED  AVAIL  REFER  MOUNTPOINT
rpool              1.80G  55.8G    96K  none
rpool/db           1.00G  47.0G    96K  /db
rpool/db/pgdb1       96K  48.0G  12.0M  /db/pgdb1
rpool/root          818M  55.8G   818M  /
```

initdb like a boss

```
# su postgres -c 'initdb --no-locale -E=UTF8 -n -N -D /db/pgdb1'
Running in noclean mode.  Mistakes will not be cleaned up.
The files belonging to this database system will be owned by user "postgres".
This user must also own the server process.

The database cluster will be initialized with locale "C".
The default text search configuration will be set to "english".

Data page checksums are disabled.

fixing permissions on existing directory /db/pgdb1 ... ok
creating subdirectories ... ok
```

- Encode using UTF8, sort using C
- Only enable locale when you know you need it
  - ~2x perf bump by avoiding calls to `iconv(3)` to figure out sort order
  - **DO NOT** use PostgreSQL checksums or compression

Backups

```
# zfs list -t snapshot
no datasets available
# pwd
/db/pgdb1
# find . | wc -l
895
# head -1 postmaster.pid
25114
# zfs snapshot rpool/db/pgdb1@pre-rm
# zfs list -t snapshot
NAME                    USED  AVAIL  REFER  MOUNTPOINT
rpool/db/pgdb1@pre-rm      0      -  12.0M  -
# psql -U postgres
psql (9.6.2)
Type "help" for help.

postgres=# \q
# rm -rf *
# ls -1 | wc -l
0
# psql -U postgres
psql: FATAL:  could not open relation mapping file "global/pg_filenode.map": No such file or directory
```

Guilty Pleasure During Demos

Backups: Has Them

```
$ psql
psql: FATAL:  could not open relation mapping file "global/pg_filenode.map": No such file or directory
# cat postgres.log
LOG:  database system was shut down at 2017-03-03 21:08:05 UTC
LOG:  MultiXact member wraparound protections are now enabled
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
FATAL:  could not open relation mapping file "global/pg_filenode.map": No such file or directory
LOG:  could not open temporary statistics file "pg_stat_tmp/global.tmp": No such file or directory
LOG:  could not open temporary statistics file "pg_stat_tmp/global.tmp": No such file or directory
...
LOG:  could not open temporary statistics file "pg_stat_tmp/global.tmp": No such file or directory
LOG:  could not open file "postmaster.pid": No such file or directory
LOG:  performing immediate shutdown because data directory lock file is invalid
LOG:  received immediate shutdown request
LOG:  could not open temporary statistics file "pg_stat/global.tmp": No such file or directory
WARNING:  terminating connection because of crash of another server process
DETAIL:  The postmaster has commanded this server process to roll back the current transaction and exit,
because another server process exited abnormally and possibly corrupted shared memory.
HINT:  In a moment you should be able to reconnect to the database and repeat your command.
LOG:  database system is shut down
# ll
total 1
drwx------ 2 postgres postgres 2 Mar  3 21:40 ./
drwxr-xr-x 3 root     root     3 Mar  3 21:03 ../
```

Restores: As Important as Backups

```
# zfs list -t snapshot
NAME                       USED  AVAIL  REFER  MOUNTPOINT
rpool/db/pgdb1@pre-rm     12.0M      -  12.0M  -
# zfs rollback rpool/db/pgdb1@pre-rm
# su postgres -c '/usr/lib/postgresql/9.6/bin/postgres -D /db/pgdb1'
LOG:  database system was interrupted; last known up at 2017-03-03 21:50:57 UTC
LOG:  database system was not properly shut down; automatic recovery in progress
LOG:  redo starts at 0/14EE7B8
LOG:  invalid record length at 0/1504150: wanted 24, got 0
LOG:  redo done at 0/1504128
LOG:  last completed transaction was at log time 2017-03-03 21:51:15.340442+00
LOG:  MultiXact member wraparound protections are now enabled
LOG:  database system is ready to accept connections
LOG:  autovacuum launcher started
```

Works all the time, every time, even with `kill -9`

(possible dataloss from ungraceful shutdown and IPC cleanup not withstanding)

Clone: Test and Upgrade with Impunity

```
# zfs clone rpool/db/pgdb1@pre-rm rpool/db/pgdb1-upgrade-test
# zfs list -r rpool/db
NAME                              USED  AVAIL  REFER  MOUNTPOINT
rpool/db                         1.00G  47.0G    96K  /db
rpool/db/pgdb1                   15.6M  48.0G  15.1M  /db/pgdb1
rpool/db/pgdb1-upgrade-test         8K  47.0G  15.2M  /db/pgdb1-upgrade-test
# echo "Test pg_upgrade"
# zfs destroy rpool/db/pgdb1-clone
# zfs clone rpool/db/pgdb1@pre-rm rpool/db/pgdb1-10
# echo "Run pg_upgrade for real"
# zfs promote rpool/db/pgdb1-10
# zfs destroy rpool/db/pgdb1
```

Works all the time, every time, even with `kill -9`

(possible dataloss from ungraceful shutdown and IPC cleanup not withstanding)

Tip: Naming Conventions

- Use a short prefix not on the root filesystem (e.g. `/db`)
- Encode the PostgreSQL major version into the dataset name
- Give each PostgreSQL cluster its own dataset (e.g. `pgdb01`)
- Optional but recommended:
  - one database per cluster
  - one app per database
  - encode environment into DB name
  - encode environment into DB username

Suboptimal	Good
`rpool/db/pgdb1`	`rpool/db/prod-db01-pg94`
`rpool/db/myapp-shard1`	`rpool/db/prod-myapp-shard001-pg95`
`rpool/db/dbN`	`rpool/db/prod-dbN-pg10`

Be explicit: codify the tight coupling between PostgreSQL versions and `$PGDATA/`.

Defy Gravity

- Take and send snapshots to remote servers
- `zfs send` emits a snapshot to stdout: treat as a file or stream
- `zfs receive` reads a snapshot from stdin
- TIP: If available:
  - Use the `-s` argument to `zfs receive`
  - Use `zfs get receive_resume_token` on the receiving end to get the required token to resume an interrupted send: `zfs send -t <token>`

Unlimited flexibility. Compress, encrypt, checksum, and offsite to your heart's content.

Defy Gravity

```
# zfs send -v -L -p -e rpool/db/pgdb1@pre-rm > /dev/null
send from @ to rpool/db/pgdb1-10@pre-rm estimated size is 36.8M
total estimated size is 36.8M
TIME            SENT   SNAPSHOT
# zfs send -v -L -p -e \
    rpool/db/pgdb1-10@pre-rm | \
  zfs receive -v \
    rpool/db/pgdb1-10-receive
send from @ to rpool/db/pgdb1-10@pre-rm estimated size is 36.8M
total estimated size is 36.8M
TIME            SENT   SNAPSHOT
received 33.8MB stream in 1 seconds (33.8MB/sec)
# zfs list -t snapshot
NAME                                USED  AVAIL  REFER
MOUNTPOINT
rpool/db/pgdb1-10@pre-rm              8K      -  15.2M  -
rpool/db/pgdb1-10-receive@pre-rm      0       -  15.2M  -
```

Defy Gravity: Incrementally

- Use a predictable snapshot naming scheme
- Send snapshots incrementally
- Clean up old snapshots
- Use a monotonic snapshot number (a.k.a. "vector clock")

Remember to remove old snapshots.
Distributed systems bingo!

Defy Gravity: Incremental

```
# echo "Change PostgreSQL's data"
# zfs snapshot rpool/db/pgdb1-10@example-incremental-001
# zfs send -v -L -p -e \
    -i rpool/db/pgdb1-10@pre-rm \
       rpool/db/pgdb1-10@example-incremental-001 \
  > /dev/null
send from @pre-rm to rpool/db/pgdb1-10@example-incremental-001
estimated size is 2K
total estimated size is 2K
# zfs send -v -L -p -e \
    -i rpool/db/pgdb1-10@pre-rm \
       rpool/db/pgdb1-10@example-incremental-001 | \
  zfs receive -v \
    rpool/db/pgdb1-10-receive
send from @pre-rm to rpool/db/pgdb1-10@example-incremental-001
estimated size is 2K
total estimated size is 2K
receiving incremental stream of rpool/db/pgdb1-10@example-
incremental-001 into rpool/db/pgdb1-10-receive@example-incremental-001
received 312B stream in 1 seconds (312B/sec)
```

Defy Gravity: Vector Clock

```
# echo "Change more PostgreSQL's data: VACUUM FULL FREEZE"
# zfs snapshot rpool/db/pgdb1-10@example-incremental-002
# zfs send -v -L -p -e \
    -i rpool/db/pgdb1-10@example-incremental-001 \
       rpool/db/pgdb1-10@example-incremental-002 \
  > /dev/null
send from @example-incremental-001 to rpool/db/pgdb1-10@example-
incremental-002 estimated size is 7.60M
total estimated size is 7.60M
TIME        SENT   SNAPSHOT
# zfs send -v -L -p -e \
    -i rpool/db/pgdb1-10@example-incremental-001 \
       rpool/db/pgdb1-10@example-incremental-002 | \
  zfs receive -v \
    rpool/db/pgdb1-10-receive
send from @example-incremental-001 to rpool/db/pgdb1-10@example-
incremental-002 estimated size is 7.60M
total estimated size is 7.60M
receiving incremental stream of rpool/db/pgdb1-10@example-incremental-002
into rpool/db/pgdb1-10-receive@example-incremental-002
TIME        SENT   SNAPSHOT
received 7.52MB stream in 1 seconds (7.52MB/sec)
```

Defy Gravity: Cleanup

```
# zfs list -t snapshot -o name,used,refer
NAME                                                 USED  REFER
rpool/db/pgdb1-10@example-incremental-001             8K  15.2M
rpool/db/pgdb1-10@example-incremental-002           848K  15.1M
rpool/db/pgdb1-10-receive@pre-rm                      8K  15.2M
rpool/db/pgdb1-10-receive@example-incremental-001     8K  15.2M
rpool/db/pgdb1-10-receive@example-incremental-002      0  15.1M
# zfs destroy rpool/db/pgdb1-10-receive@pre-rm
# zfs destroy rpool/db/pgdb1-10@example-incremental-001
# zfs destroy rpool/db/pgdb1-10-receive@example-incremental-001
# zfs list -t snapshot -o name,used,refer
NAME                                                 USED  REFER
rpool/db/pgdb1-10@example-incremental-002           848K  15.1M
rpool/db/pgdb1-10-receive@example-incremental-002      0  15.1M
```

Controversial: `logbias=throughput`

- Measure tps/qps
- Time duration of an outage (OS restart plus WAL replay, e.g. 10-20min)
- Measure cost of back pressure from the DB to the rest of the application
- Use a **`txg`** timeout of 1 second

Position: since ZFS will never be inconsistent and therefore PostgreSQL will never loose integrity, 1s of actual data loss is a worthwhile tradeoff for a ~10x performance boost in write-heavy applications.

Rationale: loss aversion costs organizations more than potentially loosing 1s of data. Back pressure is a constant cost the rest of the application needs to absorb due to continual **`fsync(2)`**'ing of WAL data. Architectural cost and premature engineering costs need to be factored in. Penny-wise, pound foolish.

Controversial: `logbias=throughput`

```
# cat /sys/module/zfs/parameters/zfs_txg_timeout
5
# echo 1 > /sys/module/zfs/parameters/zfs_txg_timeout
# echo 'options zfs zfs_txg_timeout=1' >> /etc/modprobe.d/zfs.conf
# psql -c 'ALTER SYSTEM SET synchronous_commit=off'
ALTER SYSTEM
# zfs set logbias=throughput rpool/db
```

Thank You

Thank you to people who helped make this happen:

- Percona:
  - Peter Boros
  - Robert Barabas
- Groupon:
  - Sergio Murilo
  - Chris Schneider
  - Filippos Kalamidas
  - Jose Finotto

QUESTIONS?

Email sean@chittenden.org
seanc@joyent.com

Twitter: @SeanChittenden

We're hiring! Ask me what we're up to!