The ZFS File System

Agenda

- Introduction to ZFS
- Vdevs and ZPOOL Organization
- The Distribution of Data and Metadata
- ZFS Operations
- Measured Behavior and General Performance
- Conclusions

What is ZFS ?

- ZFS is an open source storage technology developed by Sun Microsystems
- It includes support for volume management and file system management
  - ZFS can aggregate disjoint storage into a ZPOOL using certain RAID options or as a simple catenation (RAID0) infrastructure
  - ZFS is a 128 bit storage system, allowing ZPOOLS to reach sizes that are unprecedented in the industry
  - A ZPOOL is a source of communal space for multiple file systems, snapshots, clones or exportable volumes/devices (ZVOLS)
  - ZFS is an object based file system that supports multiple arbitrary attributes per object (file, directory, file system, ZVOL, etc.)

ZFS Limits

- ZFS is a 128-bit storage system, so it can store 18 billion billion (18.4×10^{18}) times more data than current 64-bit systems
  - 2^{48} — Number of files in any individual file system ($\sim 2 \times 10^{14}$)
  - 2^{48} — Number of snapshots (clones) in any file system
  - 16 exabytes (2^{64} byte) — Maximum size of a file system
  - 16 exabytes (2^{64} byte) — Maximum size of a single file (or zvol)
  - 16 exabytes (2^{64} byte) — Maximum size of any attribute
  - 3×10^{23} petabytes (2^{128} bytes) — Maximum size of any zpool
  - 2^{48} — Number of attributes of a file
  - 2^{48} — Number of files in a directory
  - 2^{64} — Number of devices in any zpool
  - 2^{64} — Number of zpools in a system
  - 2^{64} — Number of file systems in a zpool
- If 1,000 files were created every second, it would take about 9,000 years to fill just one ZFS file system (a zpool can support 2^{64} of these).

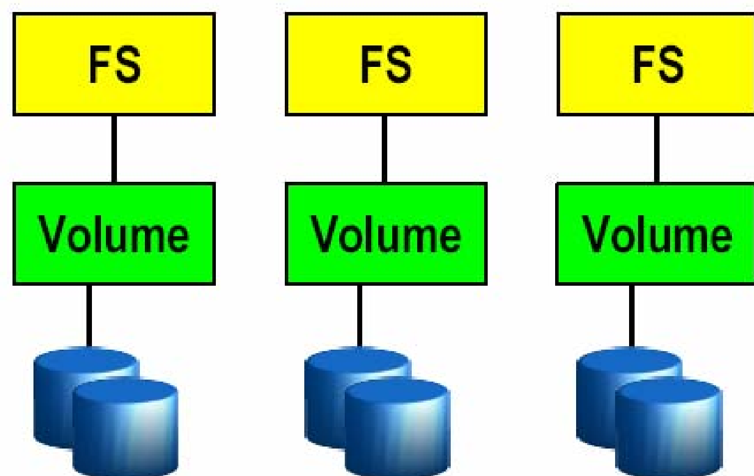
ZFS Design Principles

- **Pooled storage**
  - **Completely eliminates the antique notion of volumes**
  - **Does for storage what VM did for memory**
- **End-to-end data integrity**
  - **Historically considered “too expensive”**
  - **Current implementation shows that is can be manageable**
  - **For “simple storage” the alternative is unacceptable**
- **Transactional operation**
  - **Keeps things always consistent on disk**
  - **Removes almost all constraints on I/O order**

FS/Volume Model vs. ZFS

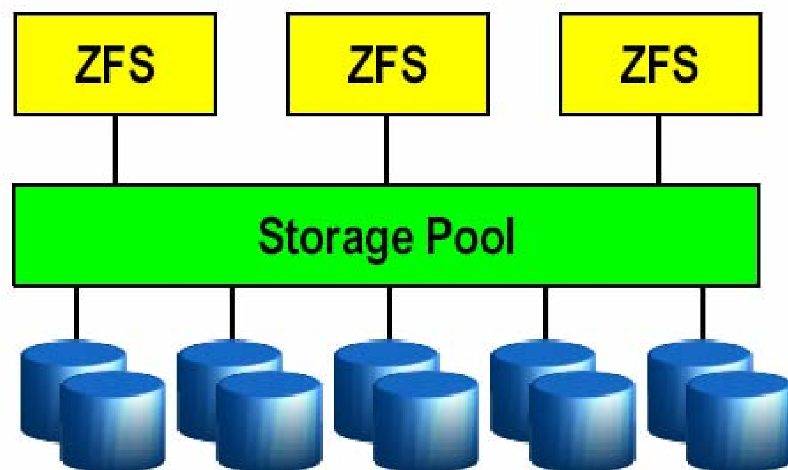
Traditional Volumes

- Abstraction: virtual disk
- Partition/volume for each FS
- Grow/shrink by hand
- Each FS has limited bandwidth
- Storage is fragmented, stranded



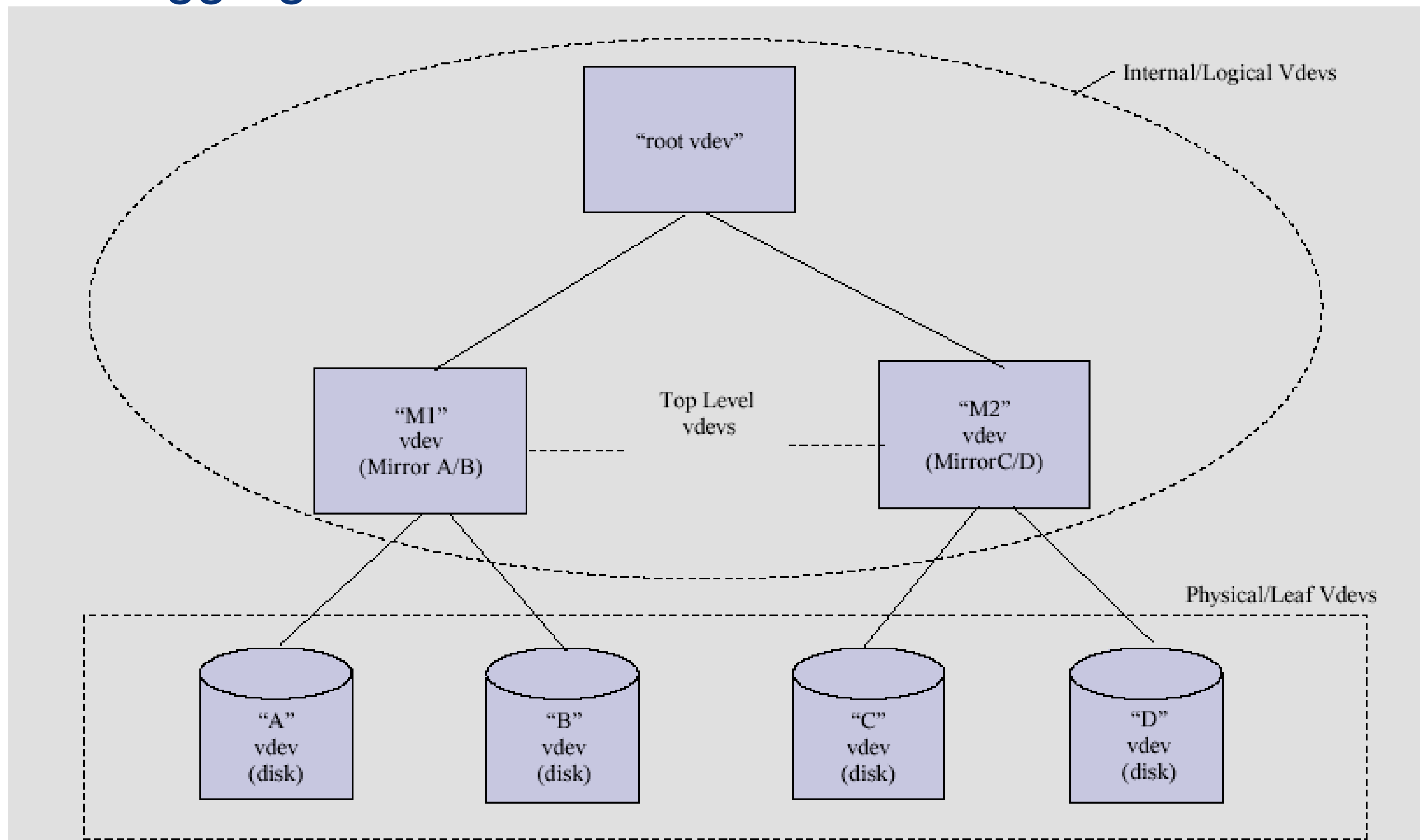
ZFS Pooled Storage

- Abstraction: malloc/free
- No partitions to manage
- Grow/shrink automatically
- All bandwidth always available
- All storage in the pool is shared



Sun microsystems

ZFS Aggregation



ZFS Data Integrity Model

- **Everything is copy-on-write**
  - **Never overwrite live data**
  - **On-disk state always valid – no “windows of vulnerability”**
  - **No need for fsck(1M)**
- **Everything is transactional**
  - **Related changes succeed or fail as a whole**
  - **No need for journaling (except for fsync() requirements)**
- **Everything is checksummed**
  - **No silent data corruption**
  - **No panics due to silently corrupted metadata**

FS/Volume Model vs. ZFS

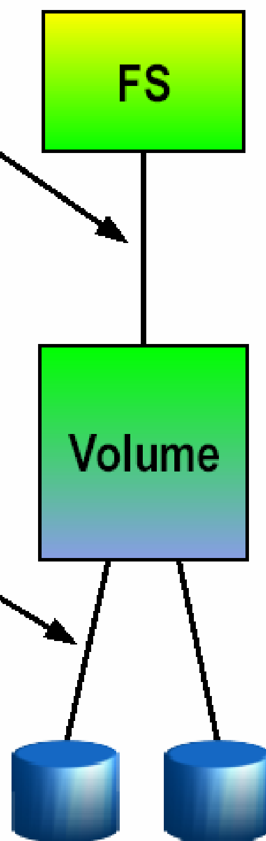
FS/Volume I/O Stack

Block Device Interface

- "Write this block, then that block, ..."
- Loss of power = loss of on-disk consistency
- Workaround: journaling, which is slow & complex

Block Device Interface

- Write each block to each disk immediately to keep mirrors in sync
- Loss of power = resync
- Synchronous and slow



ZFS I/O Stack

Object-Based Transactions

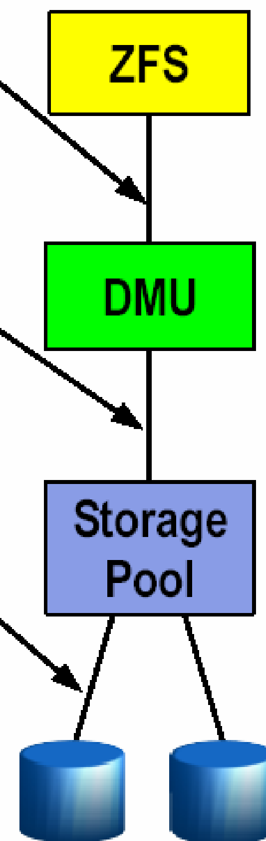
- "Make these 7 changes to these 3 objects"
- All-or-nothing

Transaction Group Commit

- Again, all-or-nothing
- Always consistent on disk
- No journal – not needed

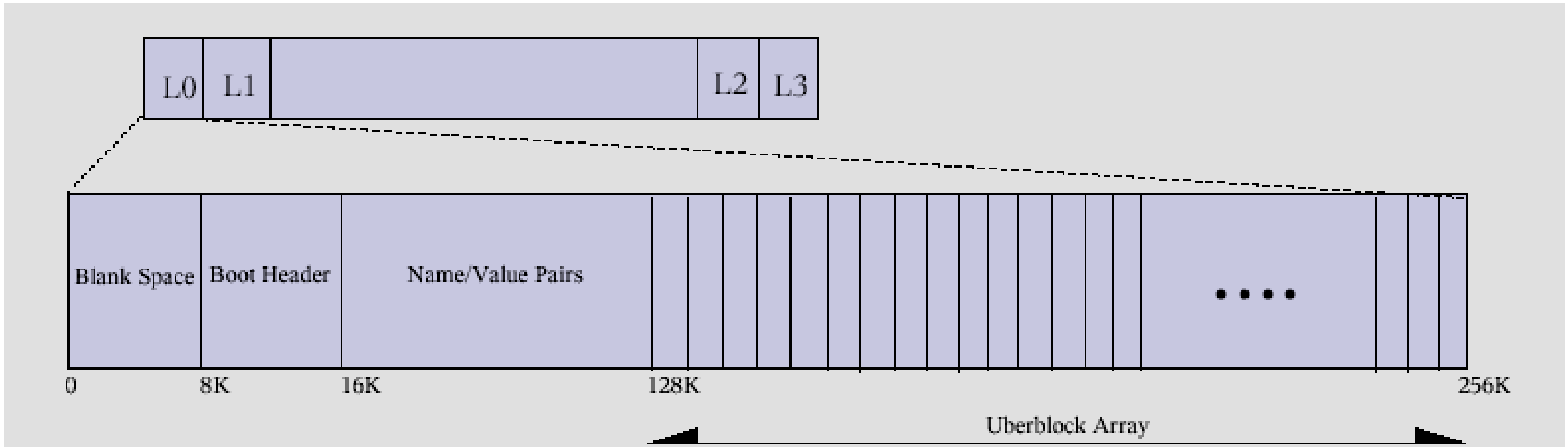
Transaction Group Batch I/O

- Schedule, aggregate, and issue I/O at will
- No resync if power lost
- Runs at platter speed



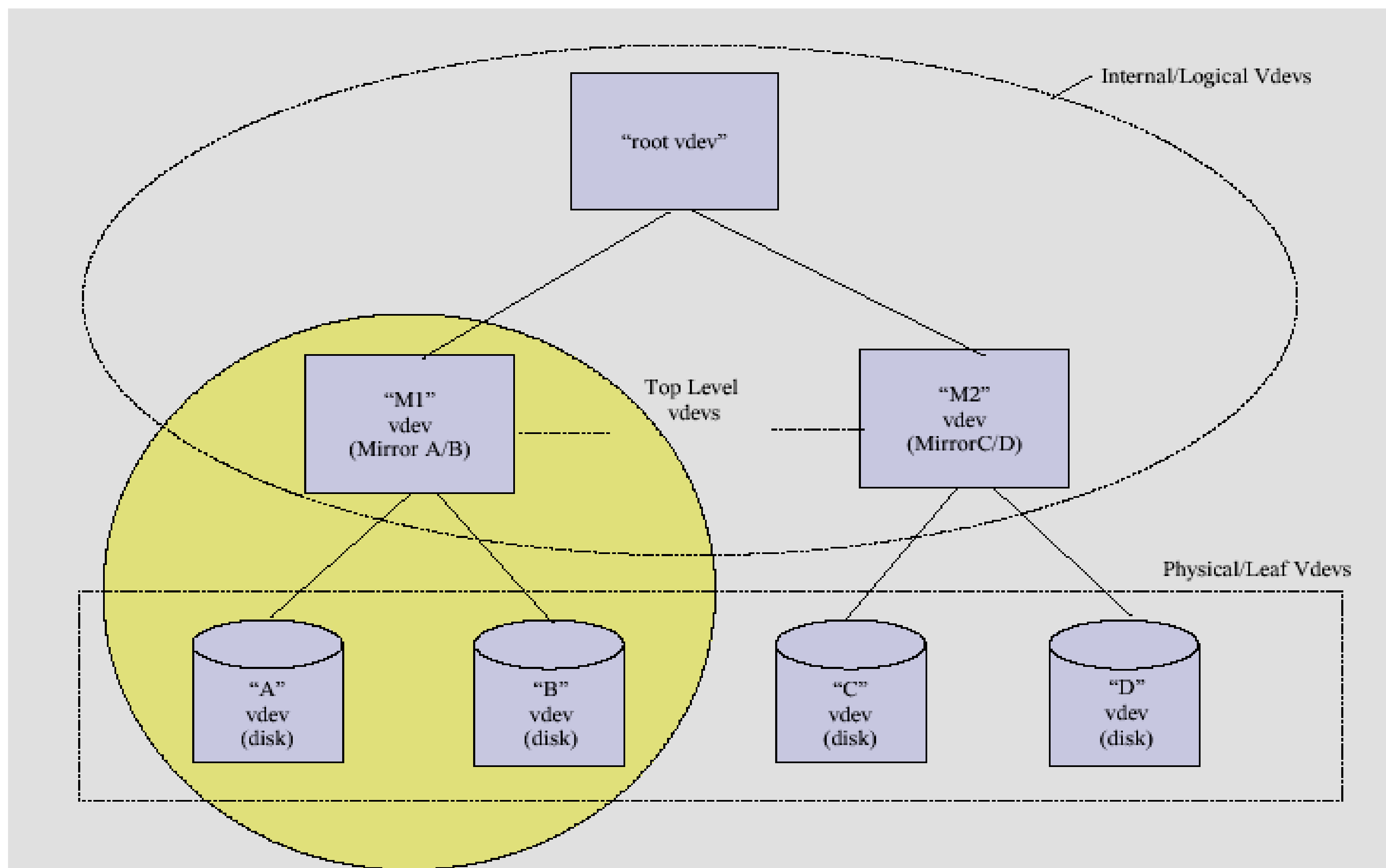
Sun microsystems

Physical VDEV Label Organization

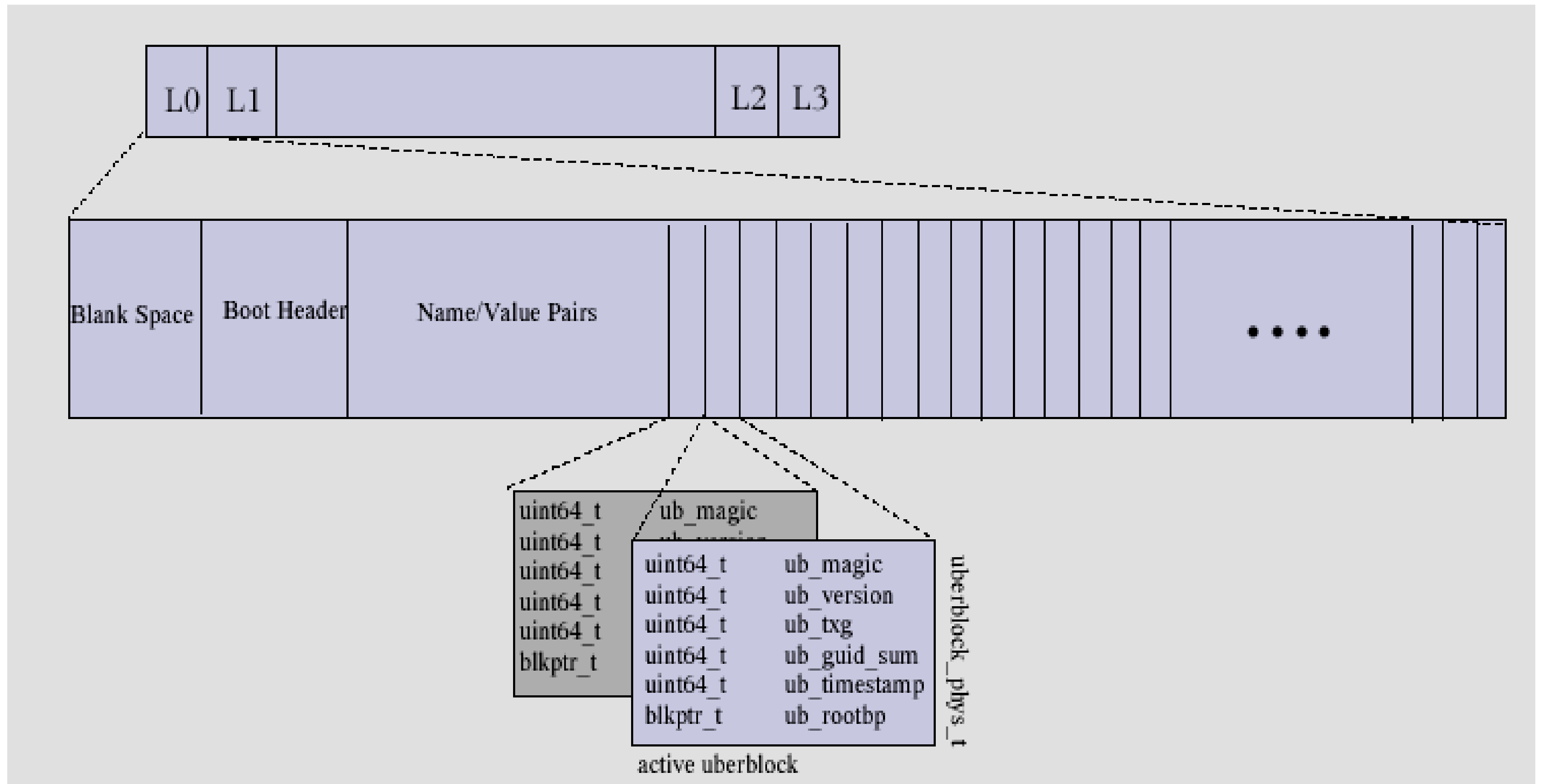


- Name/Value Pairs provide information about all of the devices that make up the sub-tree that this vdev is a part of
- Elements of the Uberblock Array are used to point to the top of the active meta data tree in the ZPOOL
  - Name/Value Pair information determines if this label's Uberblock Array has the currently active Uberblock

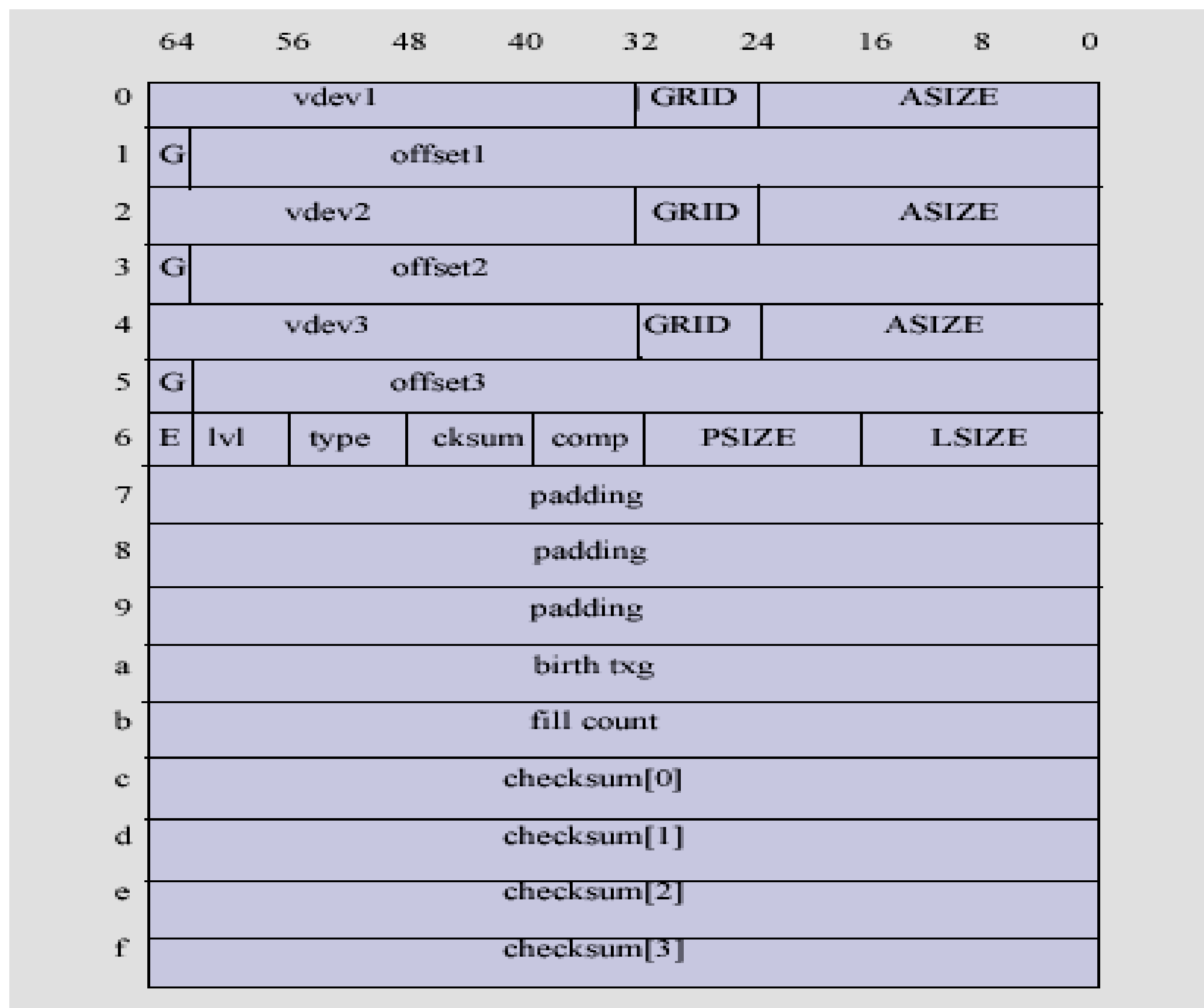
A VDEV Tree: Defined in Name /Value Pairs



Uberblock Layout: Uberblocks Allocated in RR Fashion



The ZFS Block Pointer Structure



Objects Are Managed by dnodes

- The dnode data structure is analogous to the inode data structure used in UFS type implementations
- Each object in a ZFS system is represented by a dnode
- There are about 2 dozen different object types
  - Many are meta-data objects (organized by the MOS)
  - Others are user objects (files, directories)
- User objects are organized into object sets
  - Filesystems
  - Snapshots
  - Clones
  - Volumes

The dnode Structure

dnode_phys_t

```
uint8_t     dn_type;
uint8_t     dn_indblkshift;
uint8_t     dn_nlevels
uint8_t     dn_nblkptr;
uint8_t     dn_bonustype;
uint8_t     dn_checksum;
uint8_t     dn_compress;
uint8_t     dn_pad[1];
uint16_t    dn_datablkszsec;
uint16_t    dn_bonuslen;
uint8_t     dn_pad2[4];
uint64_t    dn_maxblkid;
uint64_t    dn_secphys;
uint64_t    dn_pad3[4];
blkptr_t    dn_blkptr[N];
uint8_t     dn_bonus[BONUSLEN]
```

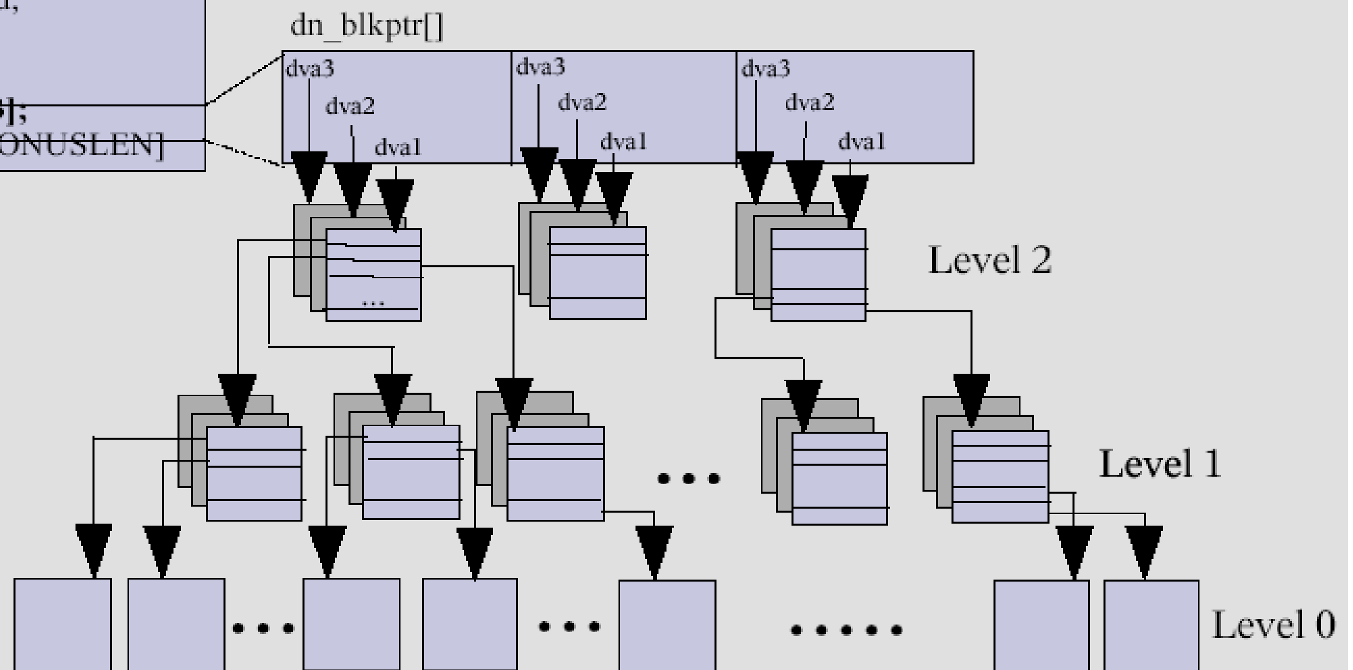
fixed length fields

variable length fields

Objects Support Multiple Copies of Meta Data and Data

dnode_phys_t

```
uint8_t     dn_type;
uint8_t     dn_indblkshift;
uint8_t     dn_nlevels = 3
uint8_t     dn_nblkptr = 3
uint8_t     dn_bonustype;
uint8_t     dn_checksum;
uint8_t     dn_compress;
uint8_t     dn_pad[1];
uint16_t    dn_datablkszsec;
uint16_t    dn_bonuslen;
uint8_t     dn_pad2[4];
uint64_t    dn_maxblkid;
uint64_t    dn_secphys;
uint64_t    dn_pad3[4];
blkptr_t    dn_blkptr[3];
uint8_t     dn_bonus[BONUSLEN]
```

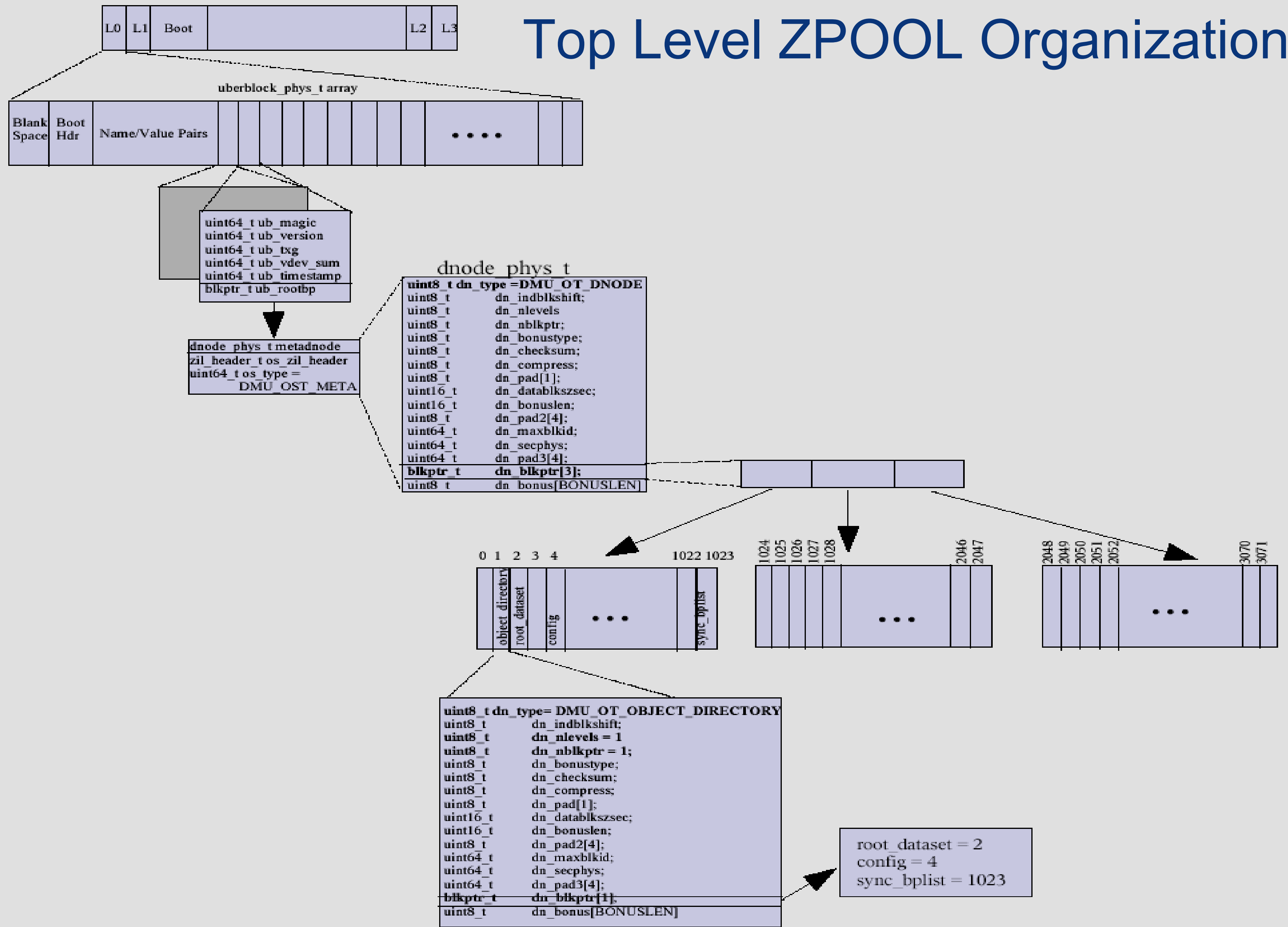


ZPOOL Data Structures

- A ZPOOL is logically a tree, rooted at the location pointed to by the currently active Uberblock
- The Uberblock uses a Block Pointer to point to an Object Set structure for the pool's Meta Data Object Set (MOS)
- Object Set structures include the dnode of the Top Level Object of the set and the set's type

```
objset_phys_t
    dnode_phys_t metadnode
    zil_header_t os_zil_header
    uint64_t os_type
```

  - Like all objects, the Top Level Object is represented by a dnode, and is the first object in the Meta Data Object Set (MOS)
  - This object provides a source of dnodes for other objects in the MOS

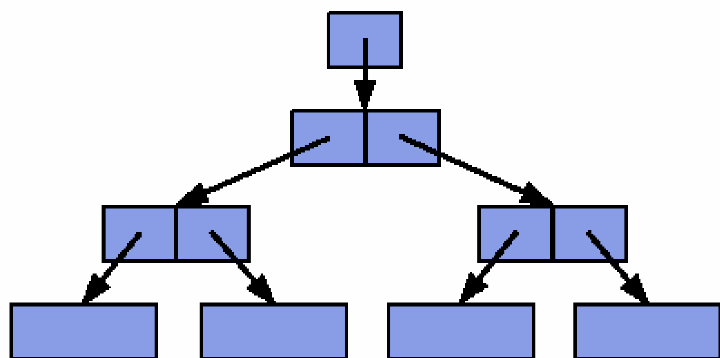

Top Level ZPOOL Organization
L0
L1
Boot
L2
L3
uberblock_phys_t array
Blank Space
Boot Hdr
Name/Value Pairs
uint64_t ub_magic
uint64_t ub_version
uint64_t ub_txg
uint64_t ub_vdev_sum
uint64_t ub_timestamp
blkptr_t ub_rootbp
dnode_phys_t metadnode
zil_header_t os_zil_header
uint64_t os_type = DMU_OST_META
dnode_phys_t
uint8_t dn_type =DMU_OT_DNODE
uint8_t dn_indblkshift;
uint8_t dn_nlevels
uint8_t dn_nblkptr;
uint8_t dn_bonustype;
uint8_t dn_checksum;
uint8_t dn_compress;
uint8_t dn_pad[1];
uint16_t dn_datablkszsec;
uint16_t dn_bonuslen;
uint8_t dn_pad2[4];
uint64_t dn_maxblkid;
uint64_t dn_secphys;
uint64_t dn_pad3[4];
blkptr_t dn_blkptr[3];
uint8_t dn_bonus[BONUSLEN]
0 1 2 3 4
1022 1023
object directory
root_dataset
config
sync_bplist
1024 1025 1026 1027 1028
2046 2047
2048 2049 2050 2051 2052
3070 3071
uint8_t dn_type= DMU_OT_OBJECT_DIRECTORY
uint8_t dn_indblkshift;
uint8_t dn_nlevels = 1
uint8_t dn_nblkptr = 1;
uint8_t dn_bonustype;
uint8_t dn_checksum;
uint8_t dn_compress;
uint8_t dn_pad[1];
uint16_t dn_datablkszsec;
uint16_t dn_bonuslen;
uint8_t dn_pad2[4];
uint64_t dn_maxblkid;
uint64_t dn_secphys;
uint64_t dn_pad3[4];
blkptr_t dn_blkptr[1];
uint8_t dn_bonus[BONUSLEN]
root_dataset = 2
config = 4
sync_bplist = 1023

The Transactional Update Model

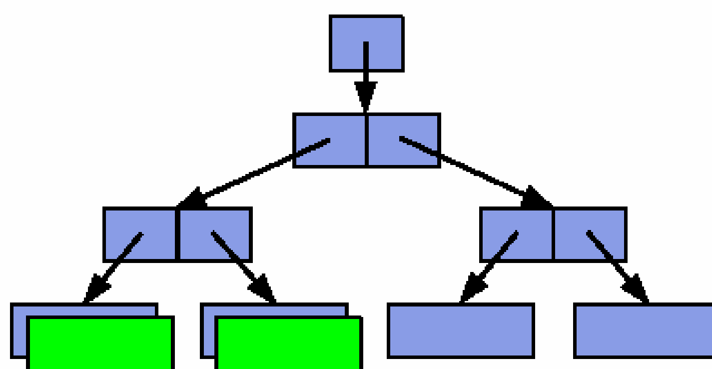
- ZFS uses a Copy On Write (COW) strategy for all update operations
  - Persistent storage is never overwritten (except for vdev labels)
  - Updates are collected in the ZFS cache for a period of time (about 5 seconds)
  - A collection of Updates are marked as belonging to the same Transaction Group
  - The entire Transaction Group is always written to new storage
- Whenever a Transaction Group is successfully written, a new Uberblock entry is written to point to the new state of the ZPOOL
  - This last step is an atomic update that either changes the state of the ZPOOL or leaves it in it's current state if it fails; a partial change is not possible

Copy-On-Write Transactions

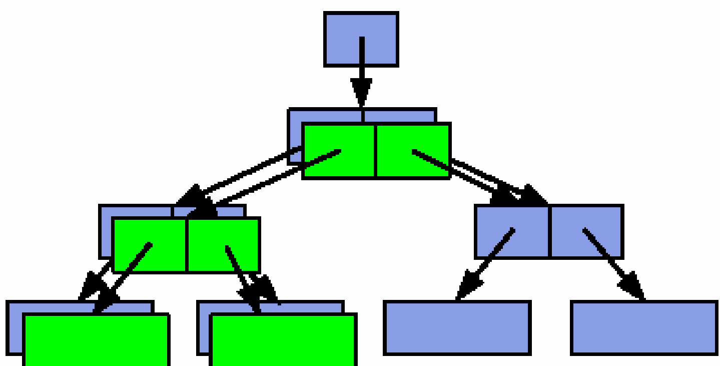
1. Initial block tree



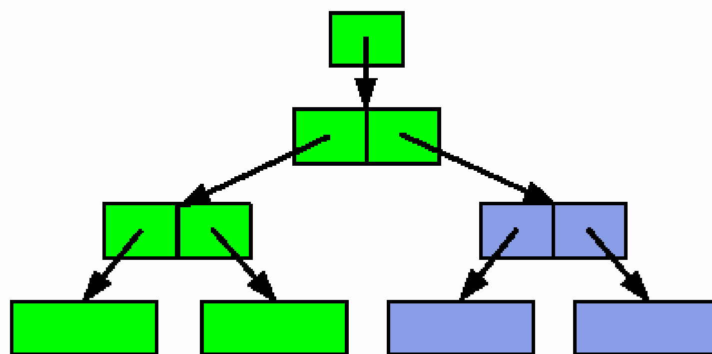
2. COW some blocks



3. COW indirect blocks



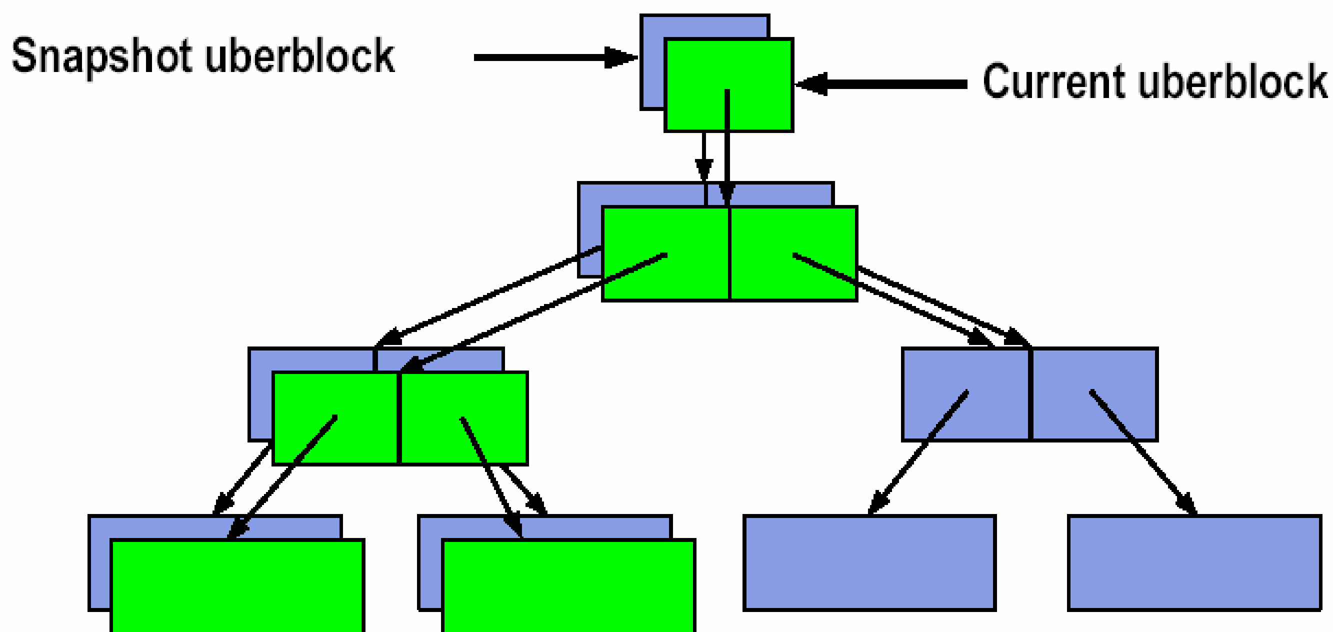
4. Rewrite uberblock (atomic)



Sun microsystems

Bonus: Constant-Time Snapshots

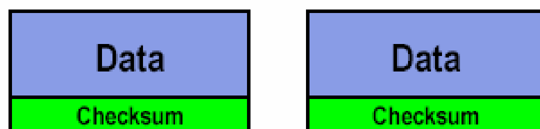
- At end of TX group, don't free COWed blocks
  - Actually cheaper to take a snapshot than not!



End-to-End Checksums

Disk Block Checksums

- Checksum stored with data block
- Any self-consistent block will pass
- Can't even detect stray writes
- Inherent FS/volume interface limitation

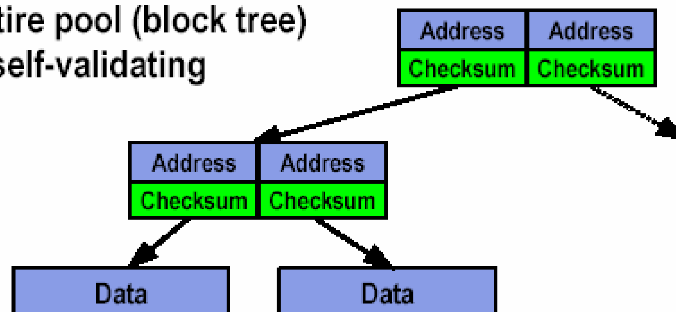


Disk checksum only validates media

✔	Bit rot
✗	Phantom writes
✗	Misdirected reads and writes
✗	DMA parity errors
✗	Driver bugs
✗	Accidental overwrite

ZFS Checksum Trees

- Checksum stored in parent block pointer
- Fault isolation between data and checksum
- Entire pool (block tree) is self-validating

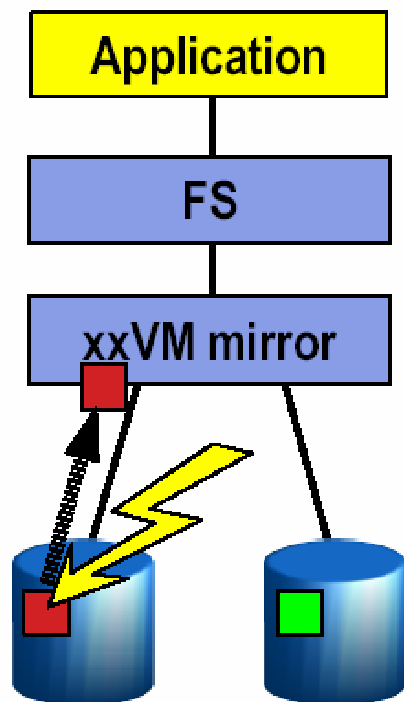


ZFS validates the entire I/O path

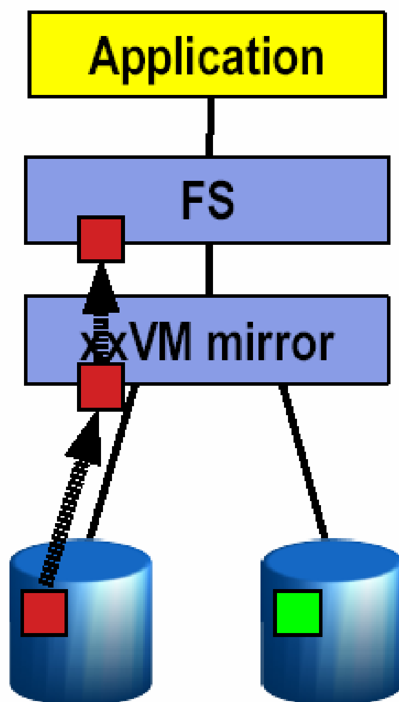
✔	Bit rot
✔	Phantom writes
✔	Misdirected reads and writes
✔	DMA parity errors
✔	Driver bugs
✔	Accidental overwrite

Traditional Mirroring

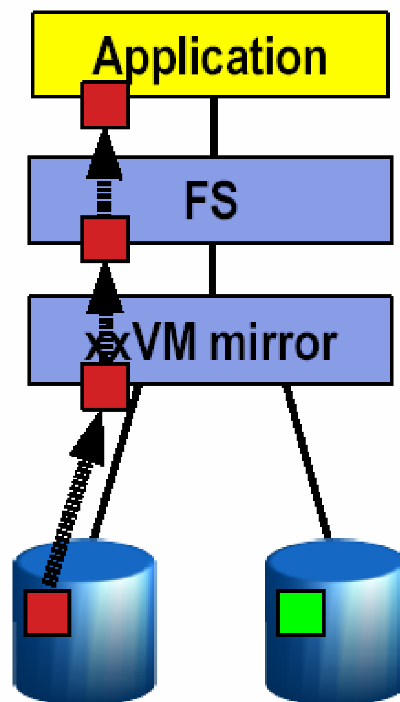
1. Application issues a read. Mirror reads the first disk, which has a corrupt block. It can't tell.



2. Volume manager passes bad block up to filesystem. If it's a metadata block, the filesystem panics. If not...

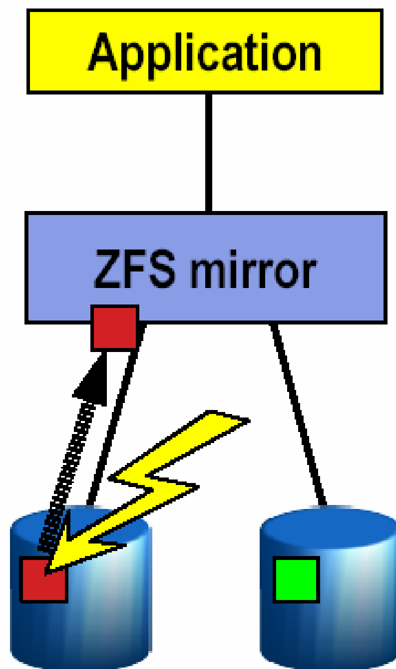


3. Filesystem returns bad data to the application.

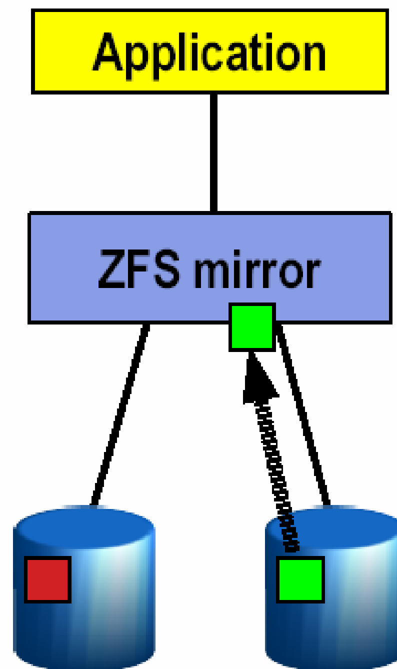


Self-Healing Data in ZFS

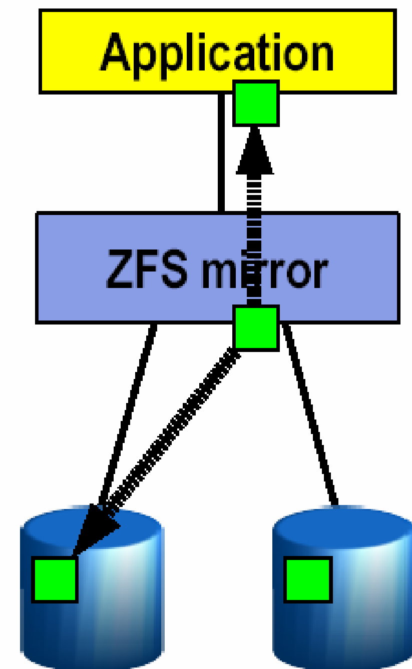
1. Application issues a read. ZFS mirror tries the first disk. Checksum reveals that the block is corrupt on disk.



2. ZFS tries the second disk. Checksum indicates that the block is good.



3. ZFS returns good data to the application and repairs the damaged block.



Traditional RAID-4 and RAID-5

- Several data disks plus one parity disk

- Fatal flaw: partial stripe writes
  - Parity update requires read-modify-write (slow)
    - Read old data and old parity (two synchronous disk reads)
    - Compute new parity = new data ^ old data ^ old parity
    - Write new data and new parity
  - Suffers from *write hole:*

    - Loss of power between data and parity writes will corrupt data
    - Workaround: $$$ NVRAM in hardware (i.e., don't lose power!)
- Can't detect or correct silent data corruption

Sun microsystems

RAID-Z

- Dynamic stripe width
  - Each logical block is its own stripe
    - 3 sectors (logical) = 3 data blocks + 1 parity block, etc.
    - Integrated stack is key: metadata drives reconstruction
    - Single-parity (RAID_Z) or double-parity (RAID-Z2) versions
- All writes are full-stripe writes
  - Eliminates read-modify-write (it's fast)
  - Eliminates the RAID-5 write hole (you don't need NVRAM)
- Detects and corrects silent data corruption
  - Checksum-driven combinatorial reconstruction
- No special hardware – ZFS loves cheap disks

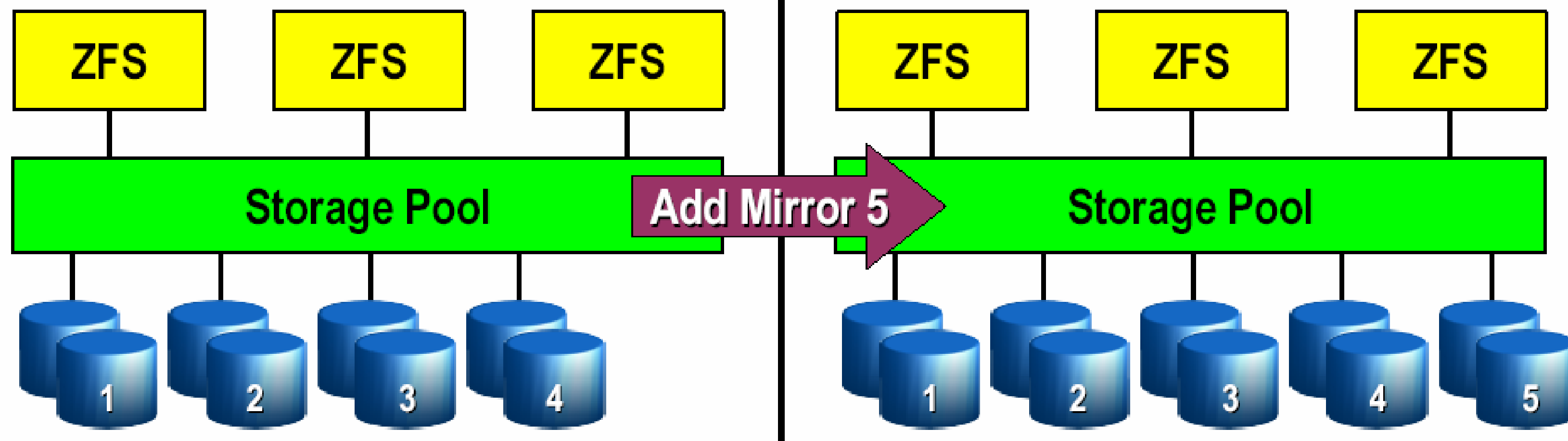
Sun microsystems

Dynamic Striping

- **Automatically distributes load across all devices**

- Writes: striped across all four mirrors
- Reads: wherever the data was written
- Block allocation policy considers:
  - Capacity
  - Performance (latency, BW)
  - Health (degraded mirrors)

- Writes: striped across all five mirrors
- Reads: wherever the data was written
- No need to migrate existing data
  - Old data striped across 1-4
  - New data striped across 1-5
  - COW gently reallocates old data



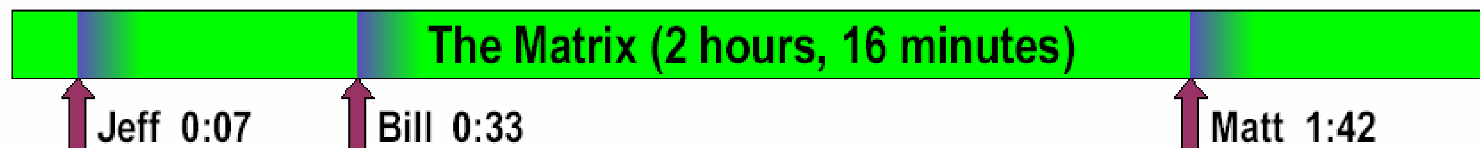
Disk Scrubbing

- Finds latent errors while they're still correctable
  - ECC memory scrubbing for disks
- Verifies the integrity of all data
  - Traverses pool metadata to read every copy of every block
  - Verifies each copy against its 256-bit checksum
  - Self-healing as it goes
- Provides fast and reliable resilvering
  - Traditional resilver: whole-disk copy, no validity check
  - ZFS resilver: live-data copy, everything checksummed
  - All data-repair code uses the same reliable mechanism
    - Mirror resilver, RAID-Z resilver, attach, replace, scrub

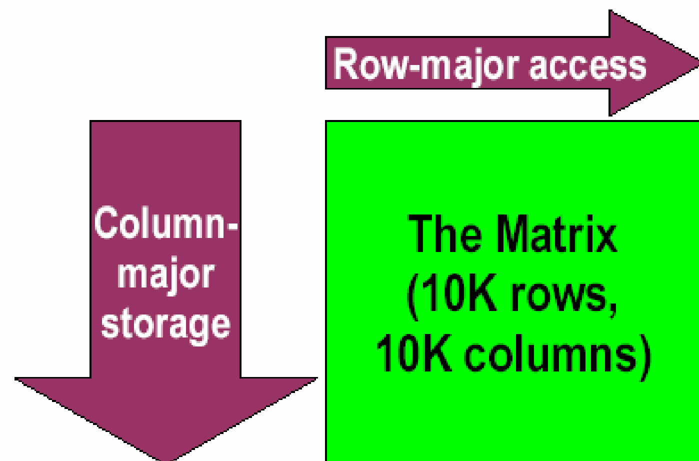
Sun microsystems

Intelligent Prefetch

- Multiple independent prefetch streams
  - Crucial for any streaming service provider

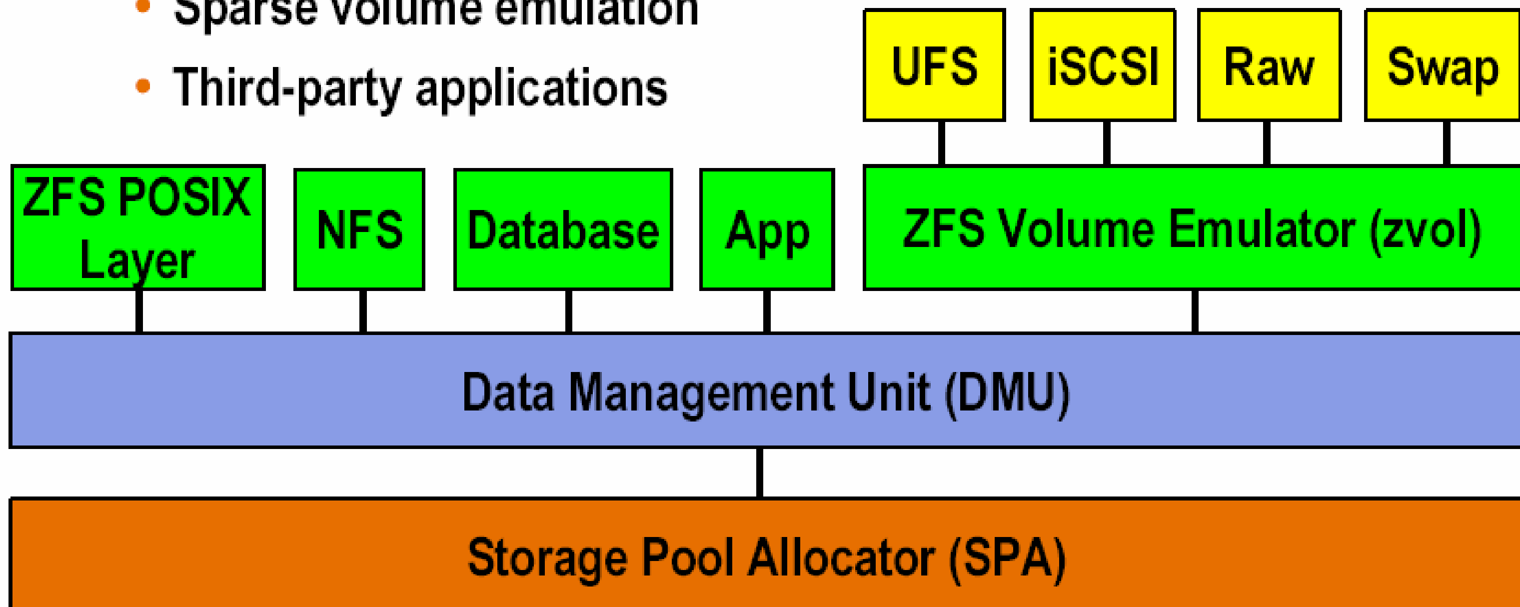


- Automatic length and stride detection
  - Great for HPC applications
  - ZFS understands the matrix multiply problem
    - Detects any linear access pattern
    - Forward or backward



Object-Based Storage

- DMU is a general-purpose transactional object store
  - Filesystems
  - Databases
  - Swap space
  - Sparse volume emulation
  - Third-party applications



Sun microsystems

Working With ZFS:

- ZFS is bundled as part of Sun's Open Solaris x86 distribution
  - The distribution is open source
  - The latest builds are available as binary downloads from Sun, and are updated every few weeks
  - The implementation includes a collection of management tools (currently command line only) for working with ZFS
- A partial implementation of ZFS over the Linux FUSE software is available
  - This is ZFS running in user space
  - ZPOOL and file system functionality is supported, but ZVOLs are not currently included

Considerations and Conclusions

- ZFS provides a robust and efficient storage solution for inexpensive JBOD configurations
- It provides end to end data integrity with selectable checksum options
  - Fletcher 2 and 4 options show little CPU impact, but use of SHA-256 at least doubles the CPU demands of a system under test
- Its transactional update implementation guarantees a consistent on-disk version at all times
  - Recovery after a system failure, for example, is quick and easy
  - ZFS, however, does not protect against data loss in this situation
- ZFS meta data requirements for a specific data load are between 10 and 20 times that of traditional UFS systems

Considerations and Conclusions (cont'd)

- ZFS boot support is not officially available in Solaris, but has been incorporated into the latest released development binary and is targeted for release in Solaris in late 2007
- New vdev's can be added to a storage pool, but they cannot be removed
  - A vdev can be exchanged for using a bigger new one
  - The ability to shrink a zpool is a work in progress, currently targeted for a Solaris 10 update in late 2007
- ZFS encourages creation of many filesystems inside the pool (for example, quota control is done on a file-system, not user basis), but importing a pool with thousands of filesystems is a slow operation (can take minutes)

Considerations and Conclusions (cont'd)

- ZFS eats a lot of CPU when doing small writes (for example, a single byte)
- ZFS Copy-on-Write operation can degrade on-disk file layout (file fragmentation) when files are modified, decreasing performance
- ZFS blocksize is configurable per filesystem, currently 128KB by default
  - If your workload reads/writes data in fixed sizes (blocks), for example a database, you should (manually) configure ZFS blocksize equal to the application blocksize, for better performance and to conserve cache memory and disk bandwidth
- ZFS only offlines a faulty harddisk if it can't be opened. Read/write errors or slow/timed-out operations are not currently used in the faulty/spare logic (work in progress)

Considerations and Conclusions (cont'd)

- When listing ZFS space usage, the "used" column only shows non-shared usage. So if some of your data is shared (for example, between snapshots), you don't know how much is there. You don't know, for example, which snapshot deletion would give you more free space.
- There is work in progress to provide automatic and periodic disk scrubbing, in order to provide corruption detection and early disk-rotting detection. Currently the data scrubbing must be done manually with "zpool scrub" command.
- When taking or destroying a snapshot while the zpool is scrubbing/resilvering, the process will be restarted from the beginning.