# 第十八讲：文件系统实例

## 第 3 节：Zettabyte File System (ZFS)

向勇、陈渝

清华大学计算机系

*xyong,yuchen@tsinghua.edu.cn*

2020 年 4 月 19 日

# 提纲

1. 第 3 节：Zettabyte File System (ZFS)
   - ZFS overview
   - ZFS I/O Stack
   - ZFS Data Integrity Model

Ref:

- Richard McDougall, Jim Mauro, Solaris Internals:Solaris 10 and OpenSolaris Kernel Architecture, 2nd Edition, Prentice Hall, July 10, 2006, ISBN 0-13-148209-2
- ZFS: The Last Word in File Systems

# What is ZFS?

ZFS is a new kind of filesystem that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability .

# What is ZFS?

ZFS is a new kind of filesystem that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability .

- Pooled storage
  - Completely eliminates the antique notion of volumes
  - Does for storage what VM did for memory

# What is ZFS?

ZFS is a new kind of filesystem that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability .

- Pooled storage
  - Completely eliminates the antique notion of volumes
  - Does for storage what VM did for memory
- Transactional object system
  - Always consistent on disk –no fsck, ever
  - Universal –file, block, iSCSI, swap ...

# What is ZFS?

ZFS is a new kind of filesystem that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability .

- Pooled storage
  - Completely eliminates the antique notion of volumes
  - Does for storage what VM did for memory
- Transactional object system
  - Always consistent on disk –no fsck, ever
  - Universal –file, block, iSCSI, swap ...
- Provable end-to-end data integrity
  - Detects and corrects silent data corruption
  - Historically considered "too expensive" –no longer true

# What is ZFS?

ZFS is a new kind of filesystem that provides simple administration, transactional semantics, end-to-end data integrity, and immense scalability .

- Pooled storage
  - Completely eliminates the antique notion of volumes
  - Does for storage what VM did for memory
- Transactional object system
  - Always consistent on disk –no fsck, ever
  - Universal –file, block, iSCSI, swap ...
- Provable end-to-end data integrity
  - Detects and corrects silent data corruption
  - Historically considered “too expensive” –no longer true
- Simple administration
  - Concisely express your intent

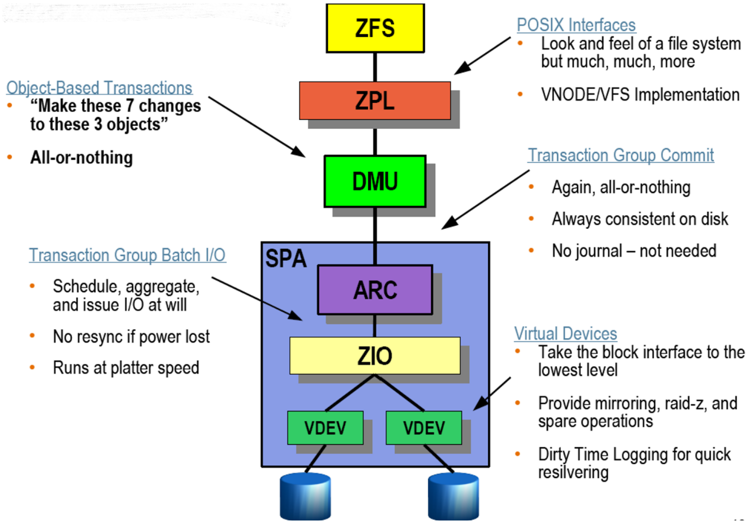
# ZFS Features

- Immense capacity
  - 128bit
- Provable data integrity
  - Detects and corrects silent data corruption
- Simple administration
  - a pleasure to use

# Pooled storage

- No volume
- Pooled storage
- Many file systems share pool
- And share all I/O channel in the pool

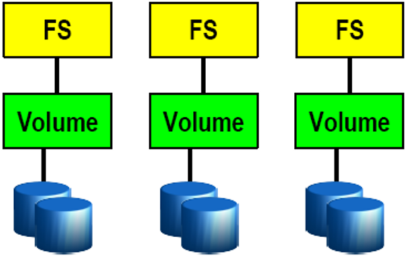
# ZFS I/O Stack


ZFS
ZPL
DMU
SPA
ARC
ZIO
VDEV
VDEV
POSIX Interfaces
Look and feel of a file system but much, much, more
VNODE/VFS Implementation
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
Virtual Devices
Take the block interface to the lowest level
Provide mirroring, raid-z, and spare operations
Dirty Time Logging for quick resilvering

# FS/Volume Model vs. ZFS

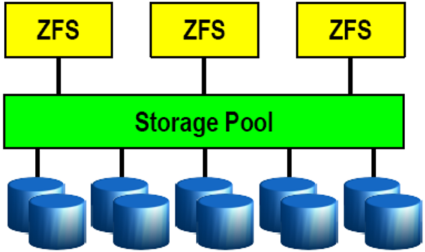
## Traditional Volumes

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

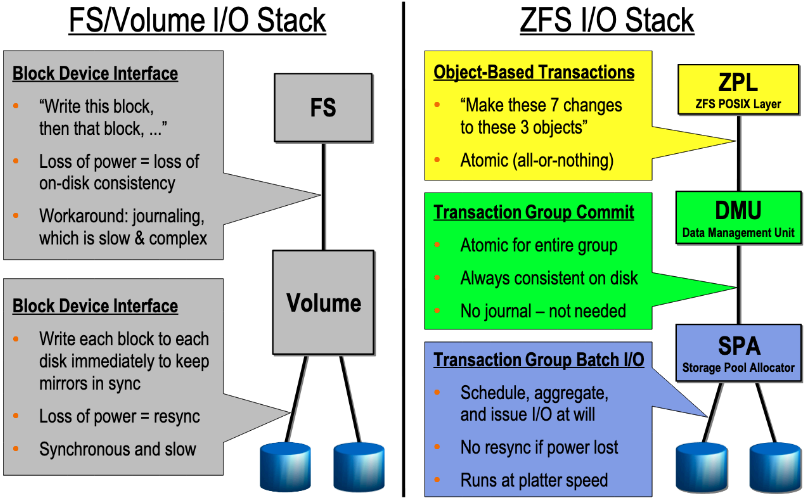



## ZFS Pooled Storage

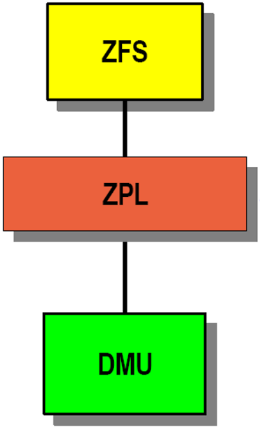
- Abstraction: malloc/free
- No partitions to manage
- Grow/shrink automatically
- All bandwidth always available
- All storage in the pool is shared



# FS/Volume Interfaces vs. ZFS

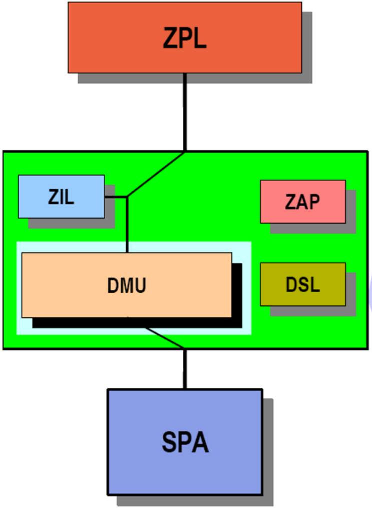


# ZFL (ZFS POSIX Layer)



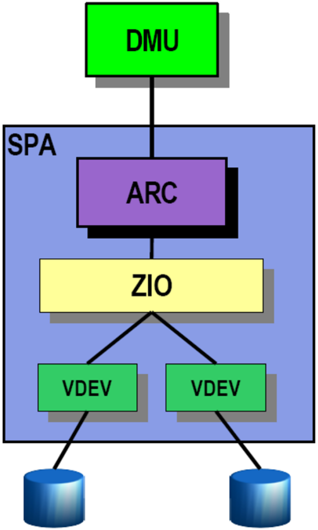

- The ZPL is the primary interface for interacting with ZFS as a filesystem.
- It is a layer that sits atop the DMU and presents a filesystem abstraction of files and directories.
- It is responsible for bridging the gap between the VFS interfaces and the underlying DMU interfaces.

# DMU (Data Management Unit)



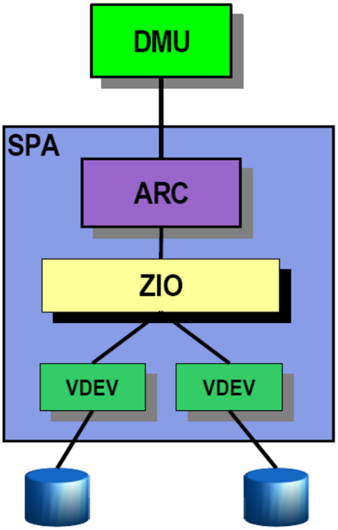

- Responsible for presenting a transactional object model, built atop the flat address space presented by the SPA.
- Consumers interact with the DMU via objsets, objects, and transactions.
- An objset is a collection of objects, where each object are pieces of storage from the SPA (i.e. a collection of blocks).
- Each transaction is a series of operations that must be committed to disk as a group; it is central to the on-disk consistency for ZFS.

# ARC (Adaptive Replacement Cache)



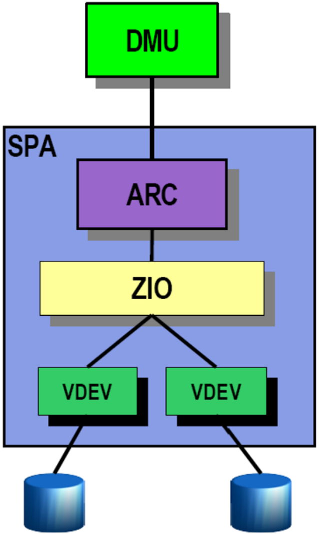

- DVA (Data Virtual Address) based cache used by DMU
- Self-tuning cache will adjust based on I/O workload
  - Replaces the page cache
- Central point for memory management for the SPA
  - Ability to evict buffers as a result of memory pressure

# ZIO (ZFS I/O Pipeline)




- Centralized I/O framework
  - > I/Os follow a structured pipeline
- Translates DVAs to logical locations on vdevs
- Drives dynamic striping and I/O retries across all active vdevs
- Drives compression, checksumming, and data redundancy

# VDEV (Virtual Devices)




- Abstraction of devices
  - Physical devices (leaf vdevs)
  - Logical devices (internal vdevs)
- Implementation of data replication algorithms
  - Mirroring, RAID-Z, and RAID-Z2
- Interfaces with the block level devices
- Provides I/O scheduling and caching

# 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)

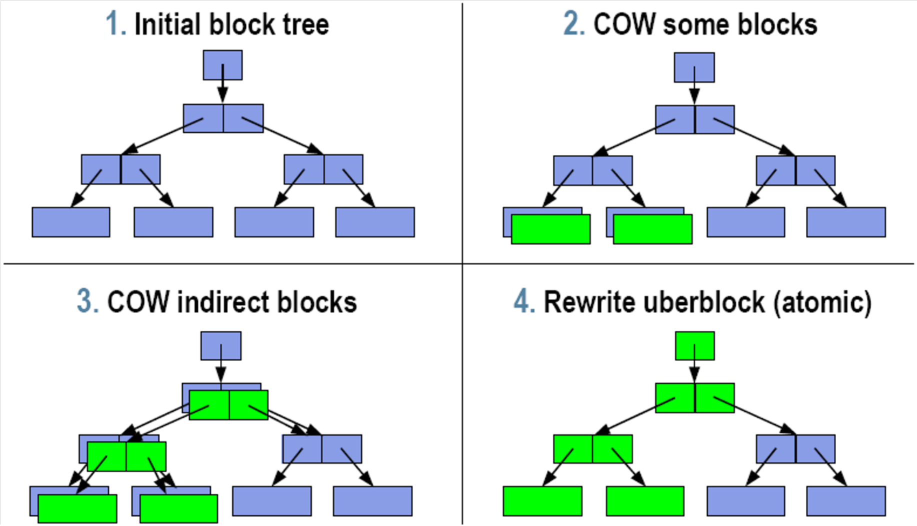
# ZFS Data Integrity Model

- 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

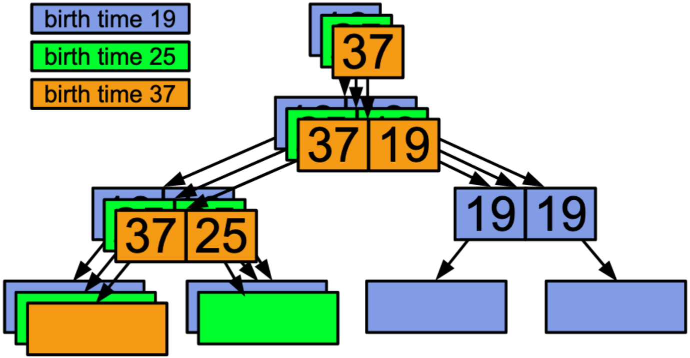
# ZFS Data Integrity Model

- 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
- Everything is checksummed
  - No silent data corruption

# Copy-On-Write Transactions



# Constant-Time Snapshots

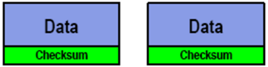



- At end of TX group, don't free COWed blocks
  - Actually cheaper to take a snapshot than not!
- The tricky part: how do you know when a block is free?

# 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

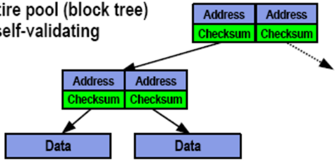



**Disk checksum only validates media**

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

## ZFS Checksum Trees

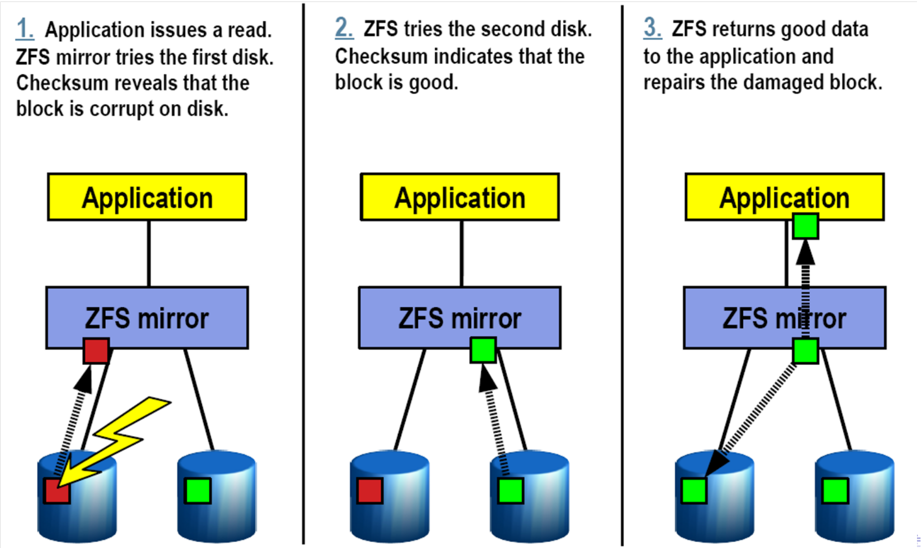
- Checksum stored in parent block pointer
- Fault isolation between data and checksum
- Entire pool (block tree) is self-validating




**ZFS validates the entire I/O path**

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

# Self-Healing Data in ZFS



# RAID-Z

| LBA \ Disk | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | $P_0$ | $D_0$ | $D_2$ | $D_4$ | $D_6$ |
| 1 | $P_1$ | $D_1$ | $D_3$ | $D_5$ | $D_7$ |
| 2 | $P_0$ | $D_0$ | $D_1$ | $D_2$ | $P_0$ |
| 3 | $D_0$ | $D_1$ | $D_2$ | $P_0$ | $D_0$ |
| 4 | $P_0$ | $D_0$ | $D_4$ | $D_8$ | $D_{11}$ |
| 5 | $P_1$ | $D_1$ | $D_5$ | $D_9$ | $D_{12}$ |
| 6 | $P_2$ | $D_2$ | $D_6$ | $D_{10}$ | $D_{13}$ |
| 7 | $P_3$ | $D_3$ | $D_7$ | $P_0$ | $D_0$ |
| 8 | $D_1$ | $D_2$ | $D_3$ | X | $P_0$ |
| 9 | $D_0$ | $D_1$ | X | $P_0$ | $D_0$ |
| 10 | $D_3$ | $D_6$ | $D_9$ | $P_1$ | $D_1$ |
| 11 | $D_4$ | $D_7$ | $D_{10}$ | $P_2$ | $D_2$ |
| 12 | $D_5$ | $D_8$ | • | • | • |

- Dynamic stripe width
  - Variable block size: 512–128K
  - Each logical block is its own stripe

# RAID-Z

| LBA \ Disk | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | $P_0$ | $D_0$ | $D_2$ | $D_4$ | $D_6$ |
| 1 | $P_1$ | $D_1$ | $D_3$ | $D_5$ | $D_7$ |
| 2 | $P_0$ | $D_0$ | $D_1$ | $D_2$ | $P_0$ |
| 3 | $D_0$ | $D_1$ | $D_2$ | $P_0$ | $D_0$ |
| 4 | $P_0$ | $D_0$ | $D_4$ | $D_8$ | $D_{11}$ |
| 5 | $P_1$ | $D_1$ | $D_5$ | $D_9$ | $D_{12}$ |
| 6 | $P_2$ | $D_2$ | $D_6$ | $D_{10}$ | $D_{13}$ |
| 7 | $P_3$ | $D_3$ | $D_7$ | $P_0$ | $D_0$ |
| 8 | $D_1$ | $D_2$ | $D_3$ | X | $P_0$ |
| 9 | $D_0$ | $D_1$ | X | $P_0$ | $D_0$ |
| 10 | $D_3$ | $D_6$ | $D_9$ | $P_1$ | $D_1$ |
| 11 | $D_4$ | $D_7$ | $D_{10}$ | $P_2$ | $D_2$ |
| 12 | $D_5$ | $D_8$ | • | • | • |

- Dynamic stripe width
  - Variable block size: 512–128K
  - Each logical block is its own stripe
- All writes are full-stripe writes
  - Eliminates read-modify-write (it's fast)
  - Eliminates the RAID-5 write hole(no need for NVRAM)

# RAID-Z

| LBA \ Disk | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | $P_0$ | $D_0$ | $D_2$ | $D_4$ | $D_6$ |
| 1 | $P_1$ | $D_1$ | $D_3$ | $D_5$ | $D_7$ |
| 2 | $P_0$ | $D_0$ | $D_1$ | $D_2$ | $P_0$ |
| 3 | $D_0$ | $D_1$ | $D_2$ | $P_0$ | $D_0$ |
| 4 | $P_0$ | $D_0$ | $D_4$ | $D_8$ | $D_{11}$ |
| 5 | $P_1$ | $D_1$ | $D_5$ | $D_9$ | $D_{12}$ |
| 6 | $P_2$ | $D_2$ | $D_6$ | $D_{10}$ | $D_{13}$ |
| 7 | $P_3$ | $D_3$ | $D_7$ | $P_0$ | $D_0$ |
| 8 | $D_1$ | $D_2$ | $D_3$ | X | $P_0$ |
| 9 | $D_0$ | $D_1$ | X | $P_0$ | $D_0$ |
| 10 | $D_3$ | $D_6$ | $D_9$ | $P_1$ | $D_1$ |
| 11 | $D_4$ | $D_7$ | $D_{10}$ | $P_2$ | $D_2$ |
| 12 | $D_5$ | $D_8$ | • | • | • |

- Dynamic stripe width
  - Variable block size: 512–128K
  - Each logical block is its own stripe
- All writes are full-stripe writes
  - Eliminates read-modify-write (it's fast)
  - Eliminates the RAID-5 write hole(no need for NVRAM)
- Both single and double parity
- Detects and corrects silent data corruption
  - Checksum-driven combinatorial reconstruction

# RAID-Z

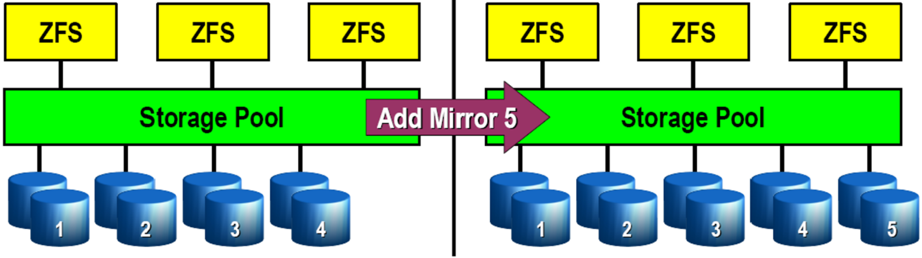
| LBA \ Disk | A | B | C | D | E |
|---|---|---|---|---|---|
| 0 | $P_0$ | $D_0$ | $D_2$ | $D_4$ | $D_6$ |
| 1 | $P_1$ | $D_1$ | $D_3$ | $D_5$ | $D_7$ |
| 2 | $P_0$ | $D_0$ | $D_1$ | $D_2$ | $P_0$ |
| 3 | $D_0$ | $D_1$ | $D_2$ | $P_0$ | $D_0$ |
| 4 | $P_0$ | $D_0$ | $D_4$ | $D_8$ | $D_{11}$ |
| 5 | $P_1$ | $D_1$ | $D_5$ | $D_9$ | $D_{12}$ |
| 6 | $P_2$ | $D_2$ | $D_6$ | $D_{10}$ | $D_{13}$ |
| 7 | $P_3$ | $D_3$ | $D_7$ | $P_0$ | $D_0$ |
| 8 | $D_1$ | $D_2$ | $D_3$ | X | $P_0$ |
| 9 | $D_0$ | $D_1$ | X | $P_0$ | $D_0$ |
| 10 | $D_3$ | $D_6$ | $D_9$ | $P_1$ | $D_1$ |
| 11 | $D_4$ | $D_7$ | $D_{10}$ | $P_2$ | $D_2$ |
| 12 | $D_5$ | $D_8$ | • | • | • |

- Dynamic stripe width
  - Variable block size: 512–128K
  - Each logical block is its own stripe
- All writes are full-stripe writes
  - Eliminates read-modify-write (it's fast)
  - Eliminates the RAID-5 write hole(no need for NVRAM)
- Both single and double parity
- Detects and corrects silent data corruption
  - Checksum-driven combinatorial reconstruction
- No special hardware–ZFS loves cheap disks

# Dynamic Striping

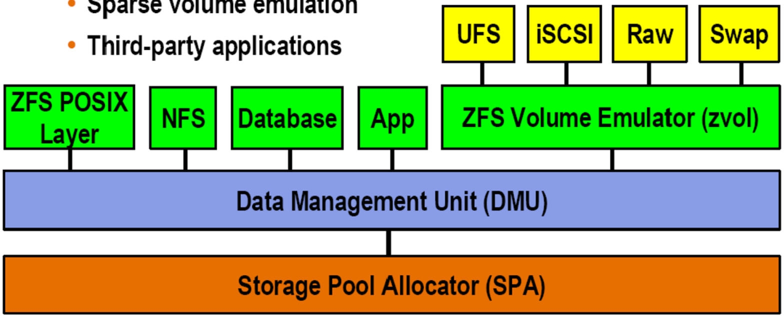
- Writes: striped across all four mirrors
- Reads: wherever the data was written
- Block allocation policy considers:
  - Capacity
  - Performance (latency, BW)
  - Health (degraded mirrors)

- 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



# Object-Based Storage

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



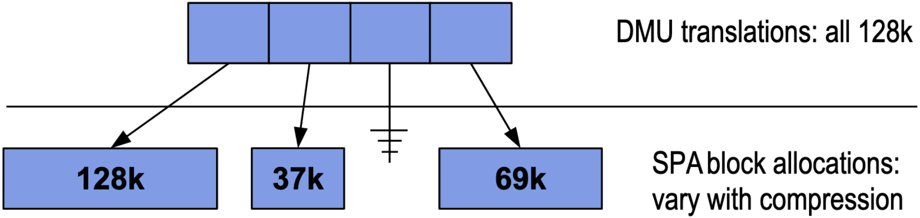
# Variable Block Size

- No single block size is optimal for everything
  - Large blocks: less metadata, higher bandwidth
  - Small blocks: more space-efficient for small objects
  - Record-structured files (e.g. databases) have natural granularity; filesystem must match it to avoid read/modify/write

# Variable Block Size

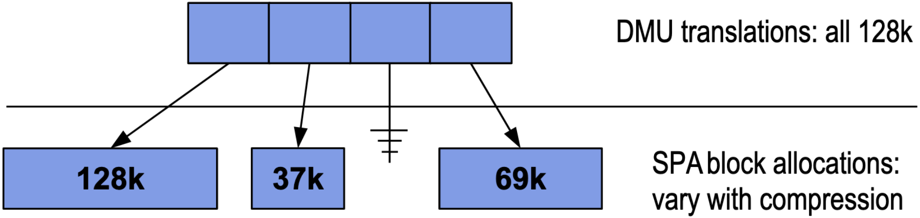
- No single block size is optimal for everything
    - Large blocks: less metadata, higher bandwidth
    - Small blocks: more space-efficient for small objects
    - Record-structured files (e.g. databases) have natural granularity; filesystem must match it to avoid read/modify/write
- Per-object granularity
    - A 37k file consumes 37k –no wasted space
- Enables transparent block-based compression

# Built-in Compression




- Block-level compression in SPA
- Transparent to all other layers

# Built-in Compression




- Block-level compression in SPA
- Transparent to all other layers
- Each block compressed independently
- All-zero blocks converted into file holes
- LZJB and GZIP available today; more on the way