LVM & ZFS: Linux Storage Management

Designing, Managing, and Protecting Data on Modern Linux Systems

Preface

Every system administrator eventually faces a moment of reckoning: a disk fails, a partition runs out of space at 2 AM, or a critical database needs to be restored from a snapshot that was never taken. In those moments, the difference between a routine recovery and a catastrophic loss often comes down to one thing — how well the underlying storage was designed and managed.

This book exists because **ZFS** changed the way I think about storage, and I believe it will change the way you think about it, too.

Why This Book?

LVM & ZFS: Linux Storage Management was written to give Linux professionals a comprehensive, practical guide to modern storage management — with **ZFS as its central focus**. While LVM remains a foundational technology in the Linux ecosystem and receives thorough coverage in the first half of this book, the heart of what follows is an in-depth exploration of ZFS: its architecture, its philosophy, and the extraordinary capabilities it brings to data protection, integrity, and administration.

ZFS is more than a filesystem. It is a fundamentally different approach to storage — one that collapses the traditional boundaries between volume management, filesystem logic, and data integrity into a single, coherent system. Understanding ZFS deeply means understanding *why* storage has historically been so fragile, and how ZFS was designed from the ground up to eliminate entire categories of failure.

What You Will Learn

This book takes you on a deliberate journey. We begin with the **Linux storage stack** – the layers of abstraction that sit between your applications and your physical disks – and establish the design principles that should guide every storage decision you make. From there, we build a solid understanding of **LVM**, not only because it remains widely deployed, but because understanding LVM's strengths and limitations provides essential context for appreciating what **ZFS does differently**.

The core of the book dives deep into **ZFS architecture and philosophy**, covering pool creation and management, datasets and properties, snapshots and clones, data integrity through checksumming and scrubbing, and performance tuning. You will learn not just *how* to run ZFS commands, but *why* ZFS behaves the way it does – knowledge that proves invaluable when designing systems, diagnosing problems, or making architectural decisions under pressure.

The final chapters bring everything together: **production deployment strategies**, a direct comparison of LVM and ZFS, and a forward-looking discussion on evolving from storage administrator to infrastructure architect – a transition that ZFS knowledge uniquely enables.

How This Book Is Structured

- **Chapters 1–2** establish foundational concepts in Linux storage and design thinking.
- **Chapters 3–6** provide comprehensive LVM coverage as both a practical skill and a comparative baseline.
- **Chapters 7–11** form the heart of the book, delivering deep, hands-on ZFS expertise.

- **Chapters 12–16** address performance, monitoring, production operations, and professional growth.
- **Appendices A–E** offer quick-reference materials, including a **ZFS command cheat sheet**, storage design templates, failure recovery procedures, and a learning roadmap.

Who This Book Is For

Whether you are a junior administrator encountering ZFS for the first time, a seasoned engineer evaluating ZFS for production workloads, or an architect designing resilient infrastructure, this book meets you where you are and takes you further.

Acknowledgments

This book would not exist without the brilliant engineers who created and continue to develop **OpenZFS**, nor without the vibrant community that has kept ZFS thriving on Linux. I am also grateful to the system administrators, colleagues, and mentors whose hard-won lessons — often learned during those 2 AM emergencies — shaped the practical wisdom found in these pages.

Finally, to every reader who has chosen to invest their time in truly understanding storage rather than merely configuring it: *thank you*. The systems you build will be better for it, and the data entrusted to your care will be safer.

Let's begin.

Miles Everhart

Table of Contents

Chapter 1: Understanding Linux Storage Layers

The journey into modern Linux storage management begins with a fundamental understanding of how data moves from an application down to the physical disk and back again. For decades, Linux administrators have relied on a layered approach to storage, where each layer handles a specific responsibility. This chapter explores those layers in depth, with a particular emphasis on how ZFS reimagines and consolidates them into a single, cohesive system. By the end of this chapter, you will have a solid mental model of traditional Linux storage architecture and a clear picture of why ZFS represents a paradigm shift in how we think about managing data on Linux systems.

The Traditional Linux Storage Stack

In a conventional Linux system, data passes through several distinct layers before it reaches the physical media. Each of these layers was designed independently, and over the years, system administrators have learned to stitch them together to build reliable storage solutions. Understanding this traditional stack is essential because it provides the context against which ZFS was designed and the problems it was built to solve.

At the very bottom of the stack sit the physical storage devices themselves. These can be traditional spinning hard disk drives, solid state drives, NVMe devices, or even remote storage accessed over a network. The Linux kernel communicates with these devices through device drivers, which present each device as a block device, typically found under the `/dev` directory. For example, a first SATA disk appears as `/dev/sda`, while an NVMe drive might appear as `/dev/nvme0n1`.

Above the physical devices sits the partitioning layer. Tools like `fdisk`, `gdisk`, and `parted` allow administrators to divide a single physical disk into multiple partitions. Each partition is itself presented as a block device. For instance, the first partition on `/dev/sda` becomes `/dev/sda1`. The partition table, whether it uses the older MBR format or the newer GPT format, is stored on the disk itself and tells the kernel how to interpret the layout.

On top of partitions, administrators can optionally deploy a volume management layer. The most common implementation on Linux is the Logical Volume Manager, known as LVM. LVM introduces three key abstractions: Physical Volumes (PVs), Volume Groups (VGs), and Logical Volumes (LVs). Physical Volumes are initialized from partitions or whole disks. Volume Groups pool one or more Physical Volumes together into a single storage resource. Logical Volumes are then carved out of a Volume Group and presented to the system as block devices that can be formatted and mounted. LVM provides flexibility by allowing administrators to resize volumes, create snapshots, and span storage across multiple disks, but it operates entirely at the block level and has no awareness of the data stored within those blocks.

Finally, at the top of the traditional stack sits the filesystem layer. This is where the actual organization of files and directories takes place. Common Linux filesystems include ext4, XFS, and Btrfs. The filesystem is created on top of a block device, whether that device is a raw partition, an LVM logical volume, or a software RAID device. The filesystem manages inodes, directory entries, free space allocation, journaling, and all the metadata required to turn raw blocks into a meaningful hierarchy of files.

If an administrator wants redundancy, there is yet another layer to consider: software RAID, typically managed by the `mdadm` utility. Software RAID combines multiple block devices into a single array that provides mirroring, striping, or parity-based protection. This RAID layer sits between the partitioning layer and either LVM or the filesystem, adding yet another component to configure and manage.

The following table summarizes the traditional Linux storage layers:

Layer	**Responsibility**	**Common Tools**	**Example Devices**
Physical Devices	Raw block storage	lsblk, smartctl	/dev/sda, /dev/nvme0n1
Partitioning	Dividing disks into regions	fdisk, gdisk, parted	/dev/sda1, /dev/nvme0n1p1
Software RAID	Redundancy and performance	mdadm	/dev/md0, /dev/md1
Volume Management	Pooling and flexible allocation	pvcreate, vgcreate, lvcreate	/dev/vg0/lv_data
Filesystem	File and directory organization	mkfs.ext4, mkfs.xfs	Mounted at /home, /var

Each of these layers must be configured, monitored, and maintained independently. When something goes wrong, the administrator must determine which layer is responsible, a task that can be surprisingly difficult in complex environments. This is the world into which ZFS was introduced, and understanding this complexity is the key to appreciating what ZFS brings to the table.

How ZFS Reimagines the Storage Stack

ZFS was originally developed at Sun Microsystems in the early 2000s by a team led by Jeff Bonwick and Matthew Ahrens. It was first released as part of OpenSolaris in 2005. The design philosophy behind ZFS was radical in its simplicity: instead of layering independent tools on top of one another, why not build a single integrated system that handles volume management, RAID, filesystem organization, and data integrity all in one place?

This is exactly what ZFS does. When you use ZFS, you do not separately configure partitions, then RAID arrays, then volume groups, then logical volumes, then filesystems. Instead, you create a storage pool, known as a zpool, from one or more physical devices, and then you create datasets within that pool. A dataset in ZFS is analogous to a filesystem, but it is far more flexible and feature-rich than a traditional filesystem. The pool handles all the underlying details of how data is distributed across disks, how redundancy is maintained, and how free space is allocated.

To illustrate this concretely, consider the task of setting up a mirrored storage system with two disks on a traditional Linux system. You would need to perform the following steps:

1. Partition both disks using `fdisk` or `gdisk`.
2. Create a RAID 1 array using `mdadm --create /dev/md0 --level=1 --raid-devices=2 /dev/sda1 /dev/sdb1`.
3. Optionally create a Physical Volume on the RAID array using `pvcreate /dev/md0`.
4. Create a Volume Group using `vgcreate vg0 /dev/md0`.

5. Create a Logical Volume using `lvcreate -L 100G -n lv_data vg0`.
6. Create a filesystem using `mkfs.ext4 /dev/vg0/lv_data`.
7. Mount the filesystem using `mount /dev/vg0/lv_data /data`.

With ZFS, the equivalent operation is dramatically simpler:

```
zpool create datapool mirror /dev/sda /dev/sdb
```

That single command creates a mirrored storage pool using both disks. ZFS automatically handles the partitioning, the RAID configuration, the volume management, and the creation of a root dataset that can be mounted and used immediately. The pool is mounted by default at `/datapool`, and you can begin storing data right away.

You can then create additional datasets within the pool:

```
zfs create datapool/documents
zfs create datapool/backups
```

Each dataset behaves like an independent filesystem with its own mount point, its own properties (such as compression, quota, and reservation settings), and its own snapshot history. Yet all datasets share the same underlying pool of storage, and ZFS manages the allocation of space dynamically.

The following table compares the traditional layered approach with the ZFS integrated approach:

Capability	**Traditional Stack**	**ZFS Approach**
Volume Management	LVM (pvcreate, vgcreate, lvcreate)	Built into zpool
RAID / Redundancy	mdadm (separate configuration)	Built into zpool (mirror, raidz, raidz2, raidz3)
Filesystem	ext4, XFS (mkfs, mount)	Built into ZFS datasets

Snapshots	LVM snapshots (limited, performance impact)	Native ZFS snapshots (copy-on-write, no performance penalty)
Data Integrity	Not guaranteed (silent corruption possible)	End-to-end checksumming of all data and metadata
Compression	Filesystem-level or application-level	Native, per-dataset, transparent compression
Administration	Multiple tools, multiple configuration files	Single toolset (zpool, zfs commands)

This consolidation is not merely a convenience. It has profound implications for data integrity, performance, and administrative simplicity, all of which we will explore in greater detail throughout this book.

The Copy-on-Write Foundation

One of the most important architectural decisions in ZFS is its use of a copy-on-write (COW) transactional model. In a traditional filesystem, when you modify a block of data, the filesystem writes the new data over the old data in place. If the system loses power during this write, the block can be left in an inconsistent state, containing neither the old data nor the complete new data. Traditional filesystems mitigate this risk through journaling, which adds complexity and overhead.

ZFS takes a fundamentally different approach. When data is modified, ZFS never overwrites the existing data. Instead, it writes the new data to a new location on disk, updates the metadata pointers to reference the new location, and then frees the old blocks. This means that the on-disk state is always consistent. Either the old data is intact (if the metadata update has not yet been committed) or the new data is intact (if the metadata update has been committed). There is no intermediate state where data is partially written.

This copy-on-write model is what makes ZFS snapshots so efficient. A snapshot is simply a record of the metadata pointers at a particular point in time. Because ZFS never overwrites data in place, the snapshot remains valid as long as the old blocks are not freed. Creating a snapshot is nearly instantaneous and consumes no additional disk space at the moment of creation. Space is only consumed as the active dataset diverges from the snapshot over time.

To create a snapshot in ZFS:

```
zfs snapshot datapool/documents@monday_backup
```

To list all snapshots:

```
zfs list -t snapshot
```

To roll back to a snapshot:

```
zfs rollback datapool/documents@monday_backup
```

These operations are fast, reliable, and deeply integrated into the storage system. There is no need for a separate snapshot mechanism, no performance penalty during normal operations, and no risk of the snapshot becoming inconsistent.

End-to-End Data Integrity

Perhaps the most compelling feature of ZFS, and one that has no equivalent in the traditional Linux storage stack, is its end-to-end data integrity verification. ZFS computes a cryptographic checksum for every block of data and metadata that it writes to disk. These checksums are stored in the parent block's metadata, not alongside the data itself. This separation means that a corruption event that damages a data block cannot also damage the checksum that verifies it.

Every time ZFS reads a block from disk, it recomputes the checksum and compares it to the stored value. If the checksums do not match, ZFS knows that the data has been corrupted. If the pool has redundancy (for example, a mirror or raidz configuration), ZFS can automatically repair the corrupted block by reading the correct copy from another disk and rewriting the damaged block. This process is called self-healing, and it happens transparently without any administrator intervention.

This is a critical capability because silent data corruption, sometimes called bit rot, is a real and well-documented phenomenon. Traditional filesystems have no mechanism to detect or repair this kind of corruption. A file can become corrupted on disk, and the filesystem will happily serve the corrupted data to applications without any indication that something is wrong. ZFS eliminates this risk entirely.

You can verify the integrity of an entire pool at any time using the `scrub` command:

```
zpool scrub datapool
```

A scrub reads every block in the pool, verifies its checksum, and repairs any corruption it finds (assuming redundancy is available). It is recommended to run scrubs on a regular schedule, typically weekly or monthly, to proactively detect and repair any issues.

To check the status of a scrub or the overall health of a pool:

```
zpool status datapool
```

The output of this command provides detailed information about the state of each device in the pool, any errors that have been detected, and the progress of any ongoing scrub operations.

Integrity Feature	**Traditional Filesystems**	**ZFS**
Checksum of data blocks	Not available	SHA-256, fletcher4, or other algorithms

Checksum of metadata	Journal-based protection only	Full checksumming of all metadata
Silent corruption detection	Not possible	Automatic on every read
Automatic repair	Not possible	Self-healing with redundant configurations
Scheduled verification	Not available (fsck is offline only)	Online scrub with no downtime

Practical Exercise: Exploring the Storage Layers

To solidify your understanding of the concepts presented in this chapter, the following exercise walks you through examining the storage layers on a Linux system and then creating a basic ZFS pool.

Note: This exercise assumes you have a Linux system with ZFS installed. On Ubuntu, you can install ZFS with `sudo apt install zfsutils-linux`. On other distributions, consult the OpenZFS documentation for installation instructions.

Step 1: Examine existing block devices.

```
lsblk
```

This command displays all block devices on the system, including their partitions and mount points. Take note of any unused disks that you can use for experimentation.

Step 2: Examine the current storage configuration in detail.

```
lsblk -f
```

The `-f` flag adds filesystem type and label information to the output. This helps you see which devices already have filesystems and which are available.

Step 3: Create a simple ZFS pool using a single disk or partition.

```
sudo zpool create testpool /dev/sdb
```

Replace `/dev/sdb` with an appropriate unused device on your system. This command creates a new ZFS pool named `testpool` using the specified device.

Note: Be extremely careful to specify the correct device. Creating a ZFS pool on a device will destroy any existing data on that device.

Step 4: Verify the pool was created.

```
zpool status testpool
zpool list testpool
```

The `status` command shows the detailed state of the pool, including the devices it contains and their health. The `list` command shows a summary including total size, used space, and free space.

Step 5: Create a dataset within the pool.

```
sudo zfs create testpool/mydata
```

Step 6: Verify the dataset and its mount point.

```
zfs list
df -h /testpool/mydata
```

Notice that ZFS automatically mounted the dataset at `/testpool/mydata` without any entry in `/etc/fstab`. ZFS manages its own mount points.

Step 7: Create a snapshot of the dataset.

```
sudo zfs snapshot testpool/mydata@initial
zfs list -t snapshot
```

Step 8: Clean up when finished experimenting.

```
sudo zpool destroy testpool
```

This removes the pool and all its datasets. In a production environment, you would obviously not destroy pools casually, but for learning purposes this allows you to start fresh.

Summary of Key Concepts

This chapter has established the foundation for everything that follows in this book. The traditional Linux storage stack is composed of multiple independent layers: physical devices, partitions, software RAID, volume management, and filesystems. Each layer requires its own tools, its own configuration, and its own monitoring. ZFS collapses all of these layers into a single integrated system that provides volume management, redundancy, filesystem services, data integrity verification, and advanced features like snapshots and compression, all managed through a unified set of commands.

The copy-on-write transactional model ensures that the on-disk state is always consistent, eliminating the need for filesystem journals and enabling instantaneous, space-efficient snapshots. End-to-end checksumming provides protection against silent data corruption that no traditional filesystem can match. And the administrative simplicity of ZFS, where a single command can replace a dozen steps in the traditional stack, reduces the likelihood of human error and makes storage management more accessible.

As you progress through the remaining chapters, you will build on this foundation to design, deploy, and manage sophisticated ZFS storage systems. But everything starts here, with a clear understanding of the layers that ZFS replaces and the principles that guide its design.

Chapter 2: Storage Planning and Design Principles

Storage planning is the foundation upon which every reliable system is built. Before a single command is typed, before a pool is created, and before data ever touches a disk, the decisions made during the planning phase will determine whether a storage system thrives under pressure or crumbles when it matters most. This chapter is dedicated to the art and science of planning storage with ZFS in mind. We will explore how to assess requirements, choose appropriate hardware, design pool layouts, plan for growth, and think critically about the trade-offs that every storage architect must navigate. By the end of this chapter, you will have a comprehensive framework for designing ZFS storage systems that are resilient, performant, and aligned with real-world workloads.

Understanding Storage Requirements Before You Begin

Every storage design begins with a fundamental question: what is this storage system expected to do? The answer to that question shapes every decision that follows. ZFS is a remarkably flexible filesystem, but its flexibility means that there are many possible configurations, and not all of them are appropriate for every situation.

The first step in storage planning is to conduct a thorough requirements analysis. This means sitting down and carefully documenting the workload characteristics, performance expectations, capacity needs, data protection requirements, and growth projections for the system you are building. Without this analysis, you are essentially guessing, and guessing with storage systems leads to costly mistakes.

Consider the following dimensions of storage requirements:

Requirement Dimension	Key Questions to Answer	Impact on ZFS Design
Capacity	How much data will be stored now? In one year? In five years?	Determines the number and size of disks, pool layout, and expansion strategy
Performance	What are the expected IOPS? What is the read/write ratio? Are workloads sequential or random?	Influences vdev type, use of SLOG and L2ARC, recordsize tuning, and disk selection
Availability	How much downtime is acceptable? What is the recovery time objective (RTO)?	Drives redundancy level (mirror, RAIDZ1, RAIDZ2, RAIDZ3) and spare disk strategy
Data Integrity	How critical is the data? What is the tolerance for silent corruption?	Affects checksum policy, copies property, and scrub scheduling
Compliance	Are there regulatory requirements for data retention or encryption?	Influences encryption settings, snapshot policies, and audit logging
Budget	What is the total budget for hardware and ongoing maintenance?	Constrains disk type (SSD vs HDD), redundancy level, and expansion plans

Let us walk through a concrete example. Suppose you are tasked with designing a storage system for a small media production company. They have approximately 20 terabytes of video footage today, expect to grow by 10 terabytes per year, need fast sequential read performance for editing workflows, and cannot afford to lose any data. They have a moderate budget and a small IT team. This profile immediately suggests certain ZFS design choices: large-capacity HDDs for bulk storage, a

RAIDZ2 configuration for strong data protection, possibly an SSD-based special vdev or L2ARC for caching frequently accessed project files, and a clear expansion plan that accounts for adding new vdevs over time.

The point is that storage planning is not about memorizing a single "best" configuration. It is about understanding the specific needs of your environment and mapping those needs to the capabilities that ZFS provides.

Choosing the Right Hardware for ZFS

ZFS was designed with the understanding that hardware fails. Disks develop bad sectors. Controllers introduce errors. Memory can flip bits. ZFS addresses all of these concerns through its architecture, but the hardware you choose still matters enormously. Poor hardware choices can undermine even the best ZFS configuration.

Disks: The Foundation of Your Pool

The single most important hardware decision is which disks to use. ZFS works with both HDDs and SSDs, and the choice between them depends entirely on your workload profile.

Hard disk drives remain the most cost-effective option for large-capacity storage. For workloads that are primarily sequential, such as video streaming, backup repositories, or archival storage, HDDs provide excellent value. When selecting HDDs for ZFS, prefer enterprise-grade drives that are designed for continuous operation. Consumer drives often have firmware-level error recovery behaviors (such as extended retry loops) that can conflict with ZFS's own error handling, potentially causing drives to be ejected from a pool unnecessarily.

Solid-state drives are the clear choice when random I/O performance is critical. Database workloads, virtual machine storage, and any application that demands low latency will benefit enormously from SSDs. NVMe drives, which connect directly to the PCIe bus, offer the highest performance tier and are increasingly common in modern ZFS deployments.

A hybrid approach is often the most practical design. ZFS supports special device classes that allow you to combine HDDs and SSDs within the same pool. For example, you might use HDDs for bulk data storage while dedicating SSDs to metadata storage (using the special vdev), write log acceleration (using the SLOG), or read caching (using the L2ARC).

Memory: More Is Almost Always Better

ZFS uses RAM extensively for its Adaptive Replacement Cache (ARC), which caches both data and metadata in memory. The general guideline is to provide at least 1 GB of RAM per terabyte of storage, but more is better, especially for workloads with large working sets. For systems using deduplication, the memory requirements increase dramatically because ZFS must keep the deduplication table (DDT) in memory for acceptable performance. A rough estimate is 5 GB of RAM per terabyte of deduplicated data, though this varies with block size and data characteristics.

ECC Memory: A Strong Recommendation

ZFS checksums all data and metadata, which allows it to detect corruption. However, if corruption occurs in RAM before data is written to disk, ZFS will faithfully checksum and store the corrupted data. ECC (Error-Correcting Code) memory detects and corrects single-bit errors in RAM, providing an additional layer of protection. While ZFS will function without ECC memory, using ECC memory is strongly recommended for any system where data integrity is a priority.

Host Bus Adapters Over Hardware RAID Controllers

This is a critical point that catches many newcomers off guard. ZFS needs direct access to the physical disks. Hardware RAID controllers that present virtual disks to the operating system interfere with ZFS's ability to manage redundancy, detect errors, and perform self-healing. The correct approach is to use a Host Bus Adapter (HBA) in IT mode (also called JBOD mode or passthrough mode), which presents each physical disk directly to the operating system. The LSI SAS 9207-8i and its successors are among the most commonly recommended HBAs for ZFS deployments.

Hardware Component	**Recommended Approach**	**What to Avoid**
Disks	Enterprise HDDs or SSDs matched to workload	Consumer drives with aggressive error recovery
Memory	ECC RAM, minimum 1 GB per TB of storage	Non-ECC RAM in production environments
Disk Controller	HBA in IT/JBOD mode	Hardware RAID controllers
SLOG Device	High-endurance, low-latency NVMe SSD with power-loss protection	Consumer SSDs without power-loss protection
L2ARC Device	Fast SSD, sized appropriately for working set	Oversized L2ARC that consumes too much RAM for indexing
Boot Device	Mirrored SSDs separate from data pool	Single boot device with no redundancy

Designing Pool Layouts and Vdev Topologies

The pool layout is the heart of ZFS storage design. A ZFS pool is composed of one or more virtual devices (vdevs), and the type and arrangement of these vdevs determine the pool's performance, capacity, and fault tolerance characteristics.

Understanding vdev types is essential. The primary vdev types used for data storage are:

Mirror vdevs consist of two or more disks that contain identical copies of data. A mirror of two disks can survive one disk failure. A three-way mirror can survive two simultaneous failures. Mirrors offer the best random I/O performance because reads can be distributed across all mirror members, and write performance is limited only by the slowest member. The trade-off is capacity efficiency: a two-way mirror provides only 50% of the raw disk capacity.

RAIDZ1 vdevs distribute data and a single parity block across multiple disks. A RAIDZ1 vdev can survive exactly one disk failure. The capacity overhead is one disk's worth of parity per vdev. For example, a five-disk RAIDZ1 vdev provides four disks' worth of usable capacity. RAIDZ1 is suitable for non-critical data or situations where the risk of a second disk failing during a resilver is acceptably low.

RAIDZ2 vdevs use double parity, allowing the vdev to survive up to two simultaneous disk failures. This is the most commonly recommended configuration for production systems because it provides a comfortable safety margin during the resilver process, which can take many hours with large disks. A six-disk RAIDZ2 vdev provides four disks' worth of usable capacity.

RAIDZ3 vdevs use triple parity and can survive three simultaneous disk failures. This level of protection is typically reserved for very large vdevs or environments with extremely high data protection requirements.

The following table summarizes the key characteristics of each vdev type:

Vdev Type	Minimum Disks	Fault Tolerance	Usable Capacity (n disks)	Read Performance	Write Performance	Best Use Case
Mirror	2	n-1 disk failures	n/2 (for 2-way mirror)	Excellent	Good	Databases, VMs, high-IOPS workloads

RAIDZ1	3	1 disk failure	n-1	Good	Moderate	Non-critical bulk storage
RAIDZ2	4	2 disk failures	n-2	Good	Moderate	General production storage
RAIDZ3	5	3 disk failures	n-3	Good	Moderate	High-value archival storage

Stripe Width and Performance Considerations

The number of disks in a RAIDZ vdev (the stripe width) has a significant impact on both performance and space efficiency. Wider stripes (more disks per vdev) improve space efficiency because the parity overhead is amortized across more data disks. However, wider stripes also increase the resilver time when a disk fails, because more data must be reconstructed.

A common and well-balanced design for RAIDZ2 is to use vdevs of six to eight disks. This provides a good balance between space efficiency, performance, and resilver time. If you have 24 disks, for example, you might create four RAIDZ2 vdevs of six disks each rather than two RAIDZ2 vdevs of twelve disks each. The four-vdev configuration will deliver significantly better IOPS because ZFS stripes data across vdevs, and each vdev can handle I/O operations independently.

A Practical Example: Designing a 12-Disk Pool

Suppose you have twelve identical 8 TB HDDs and need to design a pool for a file server. Here are three possible designs:

Configuration A: Two RAIDZ2 vdevs of six disks each. This provides 32 TB of usable capacity (8 data disks times 8 TB), can tolerate two disk failures per vdev, and delivers good IOPS because two vdevs handle I/O in parallel.

Configuration B: Six mirror vdevs of two disks each. This provides 48 TB of raw capacity but only 24 TB usable (six mirrors times 8 TB per mirror). However, IOPS performance is excellent because six vdevs operate independently, and each mirror can serve reads from either member.

Configuration C: One RAIDZ3 vdev of twelve disks. This provides 72 TB usable (nine data disks times 8 TB), but IOPS performance is poor because there is only a single vdev, and resilver times will be extremely long.

For a general-purpose file server, Configuration A is typically the best balance. For a database server, Configuration B would be preferred despite the lower capacity. Configuration C is rarely recommended due to its poor performance characteristics and long resilver times.

Planning for Growth and Expansion

One of the most important aspects of storage planning is thinking about the future. Data almost always grows, and a storage system that cannot accommodate growth becomes a liability.

ZFS provides several mechanisms for expanding storage, but each has constraints that must be understood during the planning phase.

Adding New Vdevs to an Existing Pool

The most straightforward way to expand a ZFS pool is to add new vdevs. When you add a vdev to a pool, the total capacity and performance of the pool increase immediately. ZFS will begin writing new data across all vdevs, balancing the load over time.

The critical constraint here is that once a vdev is added to a pool, it cannot be removed (with limited exceptions for mirror and special vdevs in recent ZFS versions). This means you must plan your vdev additions carefully. Adding a single disk as a stripe vdev, for example, would create a pool with no redundancy for the data on that disk, which is almost never acceptable.

```
# Adding a new RAIDZ2 vdev to an existing pool
zpool add tank raidz2 /dev/sde /dev/sdf /dev/sdg /dev/sdh /dev/
sdi /dev/sdj

# Verify the new pool layout
zpool status tank
```

Note: The new vdev should ideally match the configuration of existing vdevs in the pool. If your pool consists of RAIDZ2 vdevs with six disks each, your expansion vdevs should also be RAIDZ2 with six disks. This ensures consistent performance and fault tolerance characteristics across the pool.

Replacing Disks with Larger Ones

Another expansion strategy is to replace each disk in a vdev with a larger disk, one at a time, allowing the vdev to resilver after each replacement. Once all disks in the vdev have been replaced, ZFS can use the additional capacity through the `autoexpand` property.

```
# Enable autoexpand on the pool
zpool set autoexpand=on tank

# Replace a disk with a larger one
zpool replace tank /dev/sda /dev/new_larger_disk

# Monitor the resilver progress
zpool status tank
```

This approach is time-consuming because each resilver must complete before the next disk can be replaced, but it allows capacity expansion without adding new physical disk slots.

RAIDZ Expansion (OpenZFS 2.3 and Later)

A long-awaited feature in ZFS is the ability to expand an existing RAIDZ vdev by adding individual disks to it. This feature, introduced in OpenZFS 2.3, allows you to add a single disk to an existing RAIDZ vdev, and ZFS will redistribute the data across the wider stripe.