Contents

Chapter 1

Introduction

ZFS is a new filesystem technology that provides immense capacity (128-bit), provable data integrity, always-consistent on-disk format, self-optimizing performance, and real-time remote replication.

ZFS departs from traditional filesystems by eliminating the concept of volumes. Instead, ZFS filesystems share a common storage pool consisting of writeable storage media. Media can be added or removed from the pool as filesystem capacity requirements change. Filesystems dynamically grow and shrink as needed without the need to re-partition underlying storage.

ZFS provides a truly consistent on-disk format, but using a *copy on write* (*COW*) transaction model. This model ensures that on disk data is never overwritten and all on disk updates are done atomically.

The ZFS software is comprised of seven distinct pieces: the *SPA* (*Storage Pool Allocator*), the *DSL* (*Dataset and Snapshot Layer*), the *DMU* (*Data Management Layer*), the *ZAP* (*ZFS Attribute Processor*), the _ ZPL_ (*ZFS Posix layer*), the *ZIL* (*ZFS Intent Log*), and *ZVOL* (*ZFS Volume*). The on-disk structures associated with each of these pieces are explained in the following chapters: SPA (Chapters 1 and 2), DSL (Chapter 5), DMU (Chapter 3), ZAP (Chapter 4), ZPL (Chapter 6), ZIL (Chapter 7), ZVOL (Chapter 8).

Chapter 2

Vdevs, Labels, and Boot Block

2.1 Virtual Devices

ZFS storage pools are made up of a collection of virtual devices. There are two types of virtual devices: physical virtual devices (sometimes called leaf vdevs) and logical virtual devices (sometimes called interior vdevs). A physical vdev, is a writeable media block device (a disk, for example). A logical vdev is a conceptual grouping of physical vdevs.

Vdevs are arranged in a tree with physical vdev existing as leaves of the tree. All pools have a special logical vdev called the "root" vdev which roots the tree. All direct children of the "root" vdev (physical or logical) are called top-level vdevs. Illustration. 2.1 shows a tree of vdevs representing a sample pool configuration containing two mirrors. The first mirror (labeled "M1") contains two disk, represented by "vdev A" and "vdev B". Likewise, the second mirror "M2" contains two disks represented by "vdev C" and "vdev D". Vdevs A, B, C, and D are all physical vdevs. "M1" and M2" are logical vdevs; they are also top-level vdevs since they originate from the "root vdev".

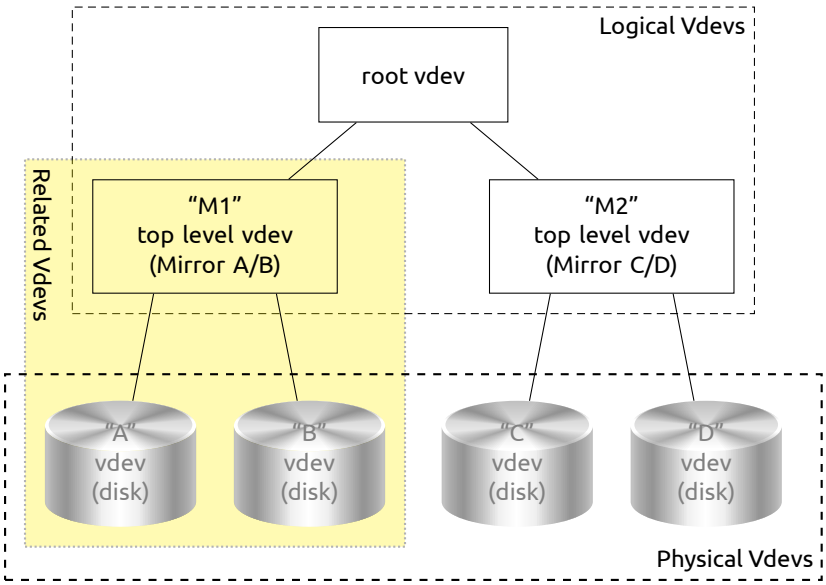

Illustration 2.1: Vdev Tree Sample Configuration

2.2 Vdev Labels

Each physical vdev within a storage pool contains a 256KB structure called a *vdev label*. The vdev label contains information describing this particular physical vdev and all other vdevs which share a common top-level vdev as an ancestor. For example, the vdev label structure contained on vdev "C", in the previous illustration, would contain information describing the following vdevs: "C", "D", and "M2". The contents of the vdev label are described in greater detail in Section. 2.3, Vdev Technical Details.

The vdev label serves two purposes: it provides access to a pool's contents and it is used to verify a pool's integrity and availability. To ensure that the vdev label is always available and always valid, redundancy and a staged update model are used. To provide redundancy, four copies of the label are written to each physical vdev within the pool. The four copies are identical within a vdev, but are not identical across vdevs in the pool. During label updates, a two staged transactional approach is used to ensure that a valid vdev label is always available on disk. Vdev label redundancy and the transactional update model are described in more detail below.

2.2.1 Label Redundancy

Four copies of the vdev label are written to each physical vdev within a ZFS storage pool. Aside from the small time frame during label update (described below), these four labels are identical and any copy can be used to access and verify the contents of the pool. When a device is added to the pool, ZFS places two labels at the front of the device and two labels at the back of the device. Illustration. 2.2 shows the layout of these labels on a device of size N; L0 and L1 represent the front two labels, L2 and L3 represent the back two labels.

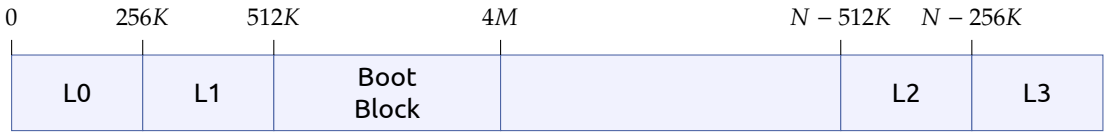

Illustration 2.2: Vdev Label layout on a block device of size N

Based on the assumption that corruption (or accidental disk overwrites) typically occurs in contiguous chunks, placing the labels in non-contiguous locations (front and back) provides ZFS with a better probability that some label will remain accessible in the case of media failure or accidental overwrite (e.g. using the disk as a swap device while it is still part of a ZFS storage pool).

2.2.2 Transactional Two Staged Label Update

The location of the vdev labels are fixed at the time the device is added to the pool. Thus, the vdev label does not have copy-on-write semantics like everything else in ZFS. Consequently, when a vdev label is updated, the contents of the label are overwritten. Any time on-disk data is overwritten, there is a potential for error. To ensure that ZFS always has access to its labels, a staged approach is used during update. The first stage of the update writes the even labels (L0 and L2) to disk. If, at any point in time, the system comes down or faults during this update, the odd labels will still be valid. Once the even labels have made it out to stable storage, the odd labels (L1 and L3) are

updated and written to disk. This approach has been carefully designed to ensure that a valid copy of the label remains on disk at all times.

2.3 Vdev Technical Details

The contents of a vdev label are broken up into four pieces: 8KB of blank space, 8K of boot header information, 112KB of name-value pairs, and 128KB of 1K sized uberblock structures. The drawing below shows an expanded view of the L0 label. A detailed description of each components follows: blank space (Section. 2.3.1), boot block header (Section. 2.3.2), name/value pair list (Section. 2.3.3), and uberblock array array (Section. 2.3.4).

2.3.1 Blank Space

ZFS supports both VTOC (Volume Table of Contents) and EFI disk labels as valid methods of describing disk layout.[1] While EFI labels are not written as part of a slice (they have their own reserved space), VTOC labels must be written to the first 8K of slice 0. Thus, to support VTOC labels, the first 8k of the vdev_label is left empty to prevent potentially overwriting a VTOC disk label.

2.3.2 Boot Block Header

The boot block header is an 8K structure that is reserved for future use. The contents of this block will be described in a future appendix of this paper.

2.3.3 Name-Value Pair List

The next 112KB of the label holds a collection of name-value pairs describing this vdev and all of it's *related vdevs*. Related vdevs are defined as all vdevs within the subtree rooted at this vdev's top-level vdev. For example, the vdev label on device "A" (seen in Illustration. 2.1) would contain information describing the subtree highlighted: including vdevs "A", "B", and "M1" (top-level vdev).

All name-value pairs are stored in *XDR* encoded nvlists. For more information on XDR encoding or nvlists, see the `libnvpair` (3LIB) and `nvlist_free` (3NVPAIR) man pages. The following name-value pairs (Table. 2.1) are contained within this 112KB portion of the vdev_label.

Table 2.1: Name-Value Pairs within vdev_label

	Name	Value	Description
Version	"version"	DATA_TYPE_UINT64	On disk format version. Current value is "1" (5000?).
Name	"name"	DATA_TYPE_STRING	Name of the pool in which this vdev belongs.
State	"state"	DATA_TYPE_UINT64	State of this pool. The following table shows all existing pool states. The Table. 2.2 shows all existing pool states.

[1]Disk labels describe disk partition and slice information. See `fdisk`(1M) and/or `format`(1M) for more information on disk partitions and slices. It should be noted that disk labels are a completely separate entity from vdev labels and while their naming is similar, they should not be confused as being similar.

	Name	Value	Description
Tranaction	"txg"	DATA_TYPE_UINT64	Transaction group number in which this label was written to disk.
Pool Guid	"pool_guid"	DATA_TYPE_UINT64	Global unique identifier (guid) for the pool.
Top Guid	"top_guid"	DATA_TYPE_UINT64	Global unique identifier for the top-level vdev of this subtree.
Vdev Tree	"vdev_tree"	DATA_TYPE_NVLIST	The vdev_tree is a nvlist structure which is used recursively to describe the hierarchical nature of the vdev tree as seen in illustrations one and four. The vdev_tree recursively describes each "related" vdev witin this vdev's subtree.

Table 2.2: Pool States

State	Value
POOL_STATE_ACTIVE	0
POOL_STATE_EXPORTED	1
POOL_STATE_DESTROYED	2

Each vdev_tree nvlist contains the elements as described in the Table. 2.3. Note that not all nvlist elements are applicable to all vdevs types. Therefore, a vdev_tree nvlist may contain only a subset of the elements described below. The Illustration. 2.3 below shows what the "vdev_tree" entry might look like for "vdev A" as shown in Illustration. 2.2 earlier in this document.

Illustration 2.3: Vdev Tree Nvlist Entries

Table 2.3: Vdev Tree NV List Entries

Name	Value	Description
"type"	DATA_TYPE_UINT64	The id is the index of this vdev in its parent's children array.
"guid"	DATA_TYPE_UINT64	Global Unique Identifier for this vdev_tree element.
"path"	DATA_TYPE_STRING	Device path. Only used for leaf vdevs.
"devid"	DATA_TYPE_STRING	Device ID for this vdev_tree element. Only used for vdevs of type disk.
"metaslab_array"	DATA_TYPE_UINT64	Object number of an object containing an array of object numbers. Each element of this array (ma[i]) is, in turn, an object number of a space map for metaslab 'i'.
"metaslab_shift"	DATA_TYPE_UINT64	Log base 2 of the metaslab size.
"ashift"	DATA_TYPE_UINT64	Log base 2 of the minimum allocatable unit for this top level vdev. This is currently '10' for a RAIDz configuration, '9' otherwise.
"asize"	DATA_TYPE_UINT64	Amount of space that can be allocated from this top level vdev
"children"	DATA_TYPE_NVLIST_ARRAY	Array of vdev_tree nvlists for each child of this vdev_tree element.

2.3.4 The Uberblock

Immediately following the nvpair lists in the vdev label is an array of *uberblocks*. The uberblock is the portion of the label containing information necessary to access the contents of the pool[2]. Only one uberblock in the pool is active at any point in time. The uberblock with the highest transaction group number and valid SHA-256 checksum is the active uberblock.

To ensure constant access to the active uberblock, the active uberblock is never overwritten. Instead, all updates to an uberblock are done by writing a modified uberblock to another element of the uberblock array. Upon writing the new uberblock, the transaction group number and timestamps are incremented thereby making it the new active uberblock in a single atomic action. Uberblocks are written in a round robin fashion across the various vdevs with the pool. The Illustration. 2.4 has an expanded view of two uberblocks within an uberblock array.

Uberblock Technical Details

The uberblock is stored in the machine's native endian format and has the following contents:

- **ub_magic:** The uberblock magic number is a 64 bit integer used to identify a device as containing ZFS data. The value of the ub_magic is $0x00bab10c$ (oo-ba-block). The Table. 2.4 shows the ub_magic number as seen on disk.

[2]The uberblock is similar to the superblock in UFS.

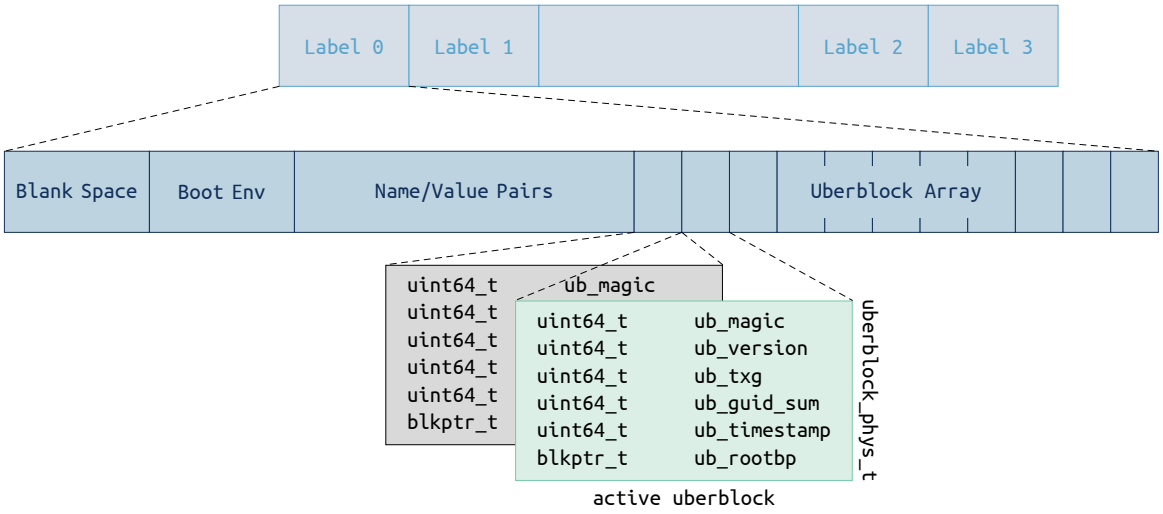

Illustration 2.4: Uberblock array showing uberblock contents

Table 2.4: Uberblock values per machine endian type

Machine Endianess	Uberblock Value
Big Endian	0x00bab10c
Little Endian	0x0cb1ba00

- **ub_version:** The version field is used to identify the on-disk format in which this data is laid out. The current on-disk format version number is 0x1 (5000?). This field contains the same value as the "version" element of the name/value pairs described in Section. 2.3.3.
- **ub_txg:** All writes in ZFS are done in transaction groups. Each group has an associated transaction group number. The ub_txg value reflects the transaction group in which this uberblock was written. The ub_txg number must be greater than or equal to the "txg" number stored in the nvlist for this label to be valid.
- **ub_guid_sum:** The ub_guid_sum is used to verify the availability of vdevs within a pool. When a pool is opened, ZFS traverses all leaf vdevs within the pool and totals a running sum of all the GUIDs (a vdev's guid is stored in the guid nvpair entry, see Section. 2.3.3) it encounters. This computed sum is checked against the ub_guid_sum to verify the availability of all vdevs within this pool.
- **ub_timestamp:** Coordinated Universal Time (UTC) when this uberblock wasf written in seconds since January 1st 1970 (GMT).
- **ub_rootbp:** The ub_rootbp is a blkptr structure containing the location of the MOS. Both the MOS and blkptr structures are described in later chapters of this document: Chapters 4 and 2 respectively.

2.4 Boot Block

Immediately following the L0 and L1 labels is a 3.5MB chunk reserved for future use (see Illustration. 2.2). The contents of this block will be described in a future appendix of this paper.

Chapter 3

Block Pointers and Indirect Blocks

Data is transferred between disk and main memory in units called blocks. A block pointer (`blkptr_t`) is a 128 byte ZFS structure used to physically locate, verify, and describe blocks of data on disk.

The 128 byte blkptr_t structure layout is shown in the Illustration. 3.1.

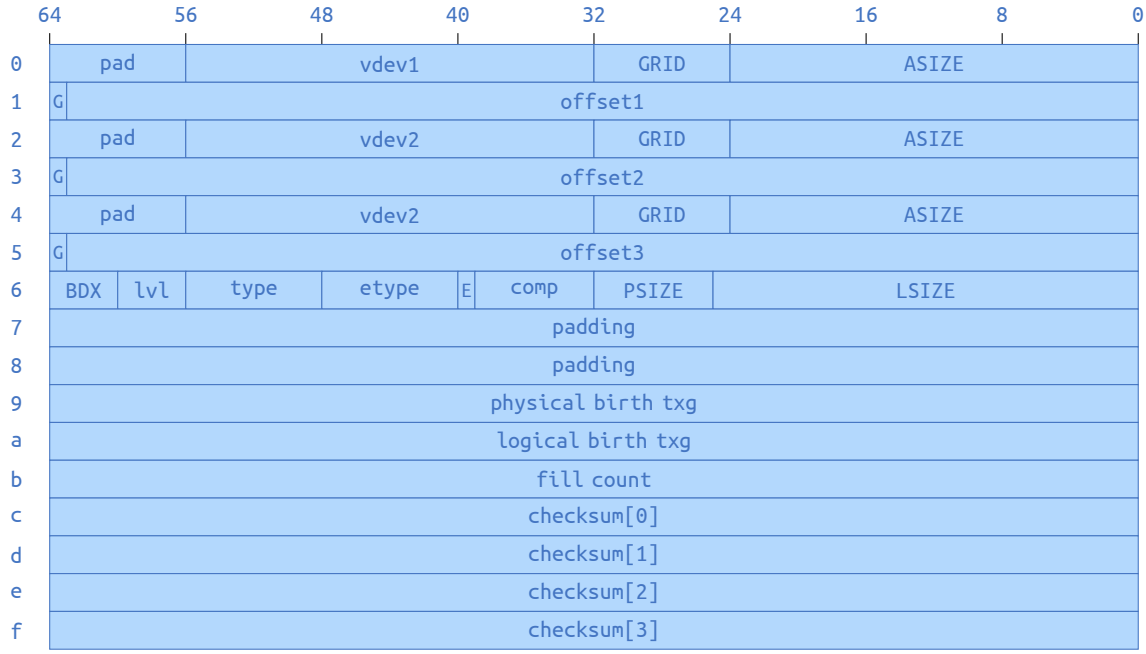

Illustration 3.1: Block pointer structure showing byte by byte usage.

Normally, block pointers point (via their DVAs) to a block which holds data. If the data that we need to store is very small, this is an inefficient use of space, Additionally, reading these small blocks tends to generate more random reads. Embedded-data Block Pointers was introduced. It allows small pieces of data (the "payload", upto 112 bytes) embedded in the block pointer, the block pointer doesn't point to anything then. The layout of an embedded block pointer is as Illustration. 3.2.

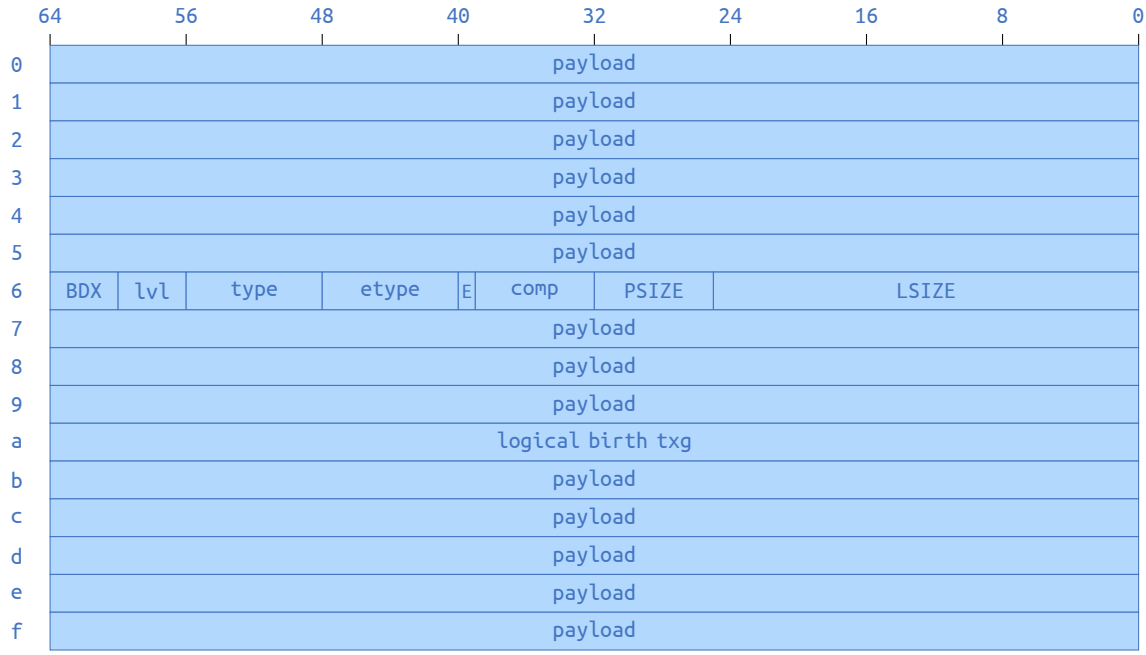

Illustration 3.2: Embedded Block Pointer Layout

3.1 Data Virtual Address

The *data virtual address*, or *DVA* is the name given to the combination of the vdev and offset portions of the block pointer, for example the combination of vdev1 and offset1 make up a DVA (dva1). ZFS provides the capability of storing up to three copies of the data pointed to by the block pointer, each pointed to by a unique DVA (dva1, dva2, or dva3). The data stored in each of these copies is identical. The number of DVAs used per block pointer is purely a policy decision and is called the "wideness" of the block pointer: single wide block pointer (1 DVA), double wide block pointer (2 DVAs), and triple wide block pointer (3 DVAs).

The *vdev* portion of each DVA is a 32 bit integer which uniquely identifies the vdev ID containing this block. The offset portion of the DVA is a 63 bit integer value holding the offset (starting after the vdev labels (L0 and L1) and boot block) within that device where the data lives. Together, the vdev and offset uniquely identify the block address of the data it points to.

The value stored in offset is the offset in terms of sectors (512 byte blocks). To find the physical block byte offset from the beginning of a slice, the value inside offset must be shifted over ($\ll$) by 9 ($2^9 = 512$) and this value must be added to $0x400000$ (size of two vdev_labels and boot block).

$$physical\ block\ address = (offset \ll 9) + 0x400000\ (4MB)$$

3.2 GRID

Raid-Z layout information, reserved for future use.

3.3 GANG

A *gang block* is a block whose contents contain block pointers. Gang blocks are used when the amount of space requested is not available in a contiguous block. In a situation of this kind, several smaller blocks will be allocated (totaling up to the size requested) and a gang block will be created to contain the block pointers for the allocated blocks. A pointer to this gang block is returned to the requester, giving the requester the perception of a single block.