ZFS On-Disk Specification

Draft

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A

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".

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 secion 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

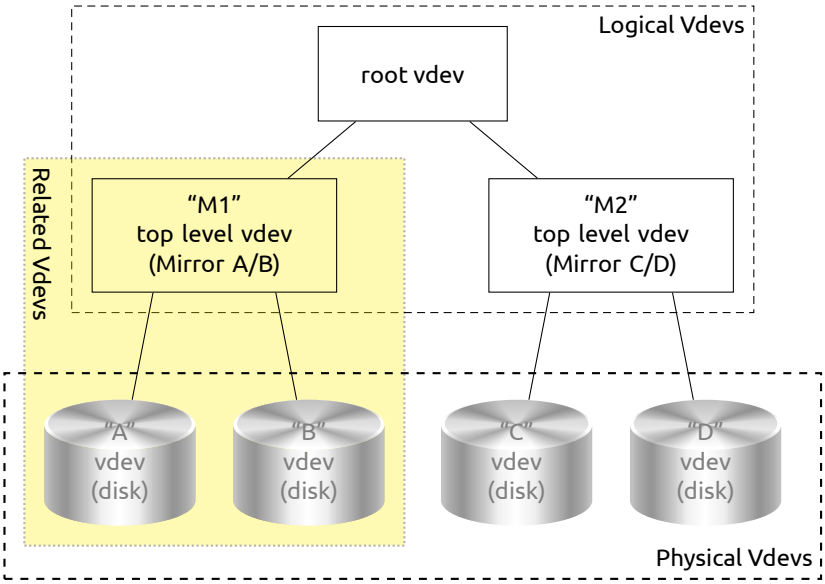

Illustration 2.1: Vdev Tree Sample Configuration

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.

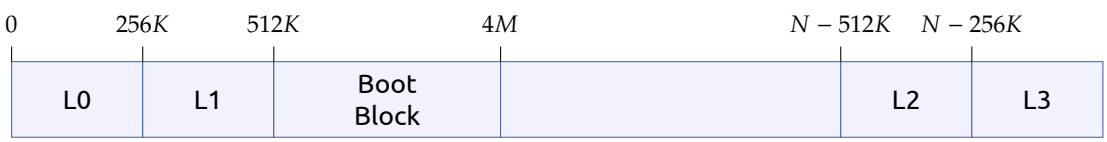

Illustration 2.2: Vdev Label layout on a block device of size N

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 (secion 2.3.1), boot block header (secion 2.3.2), name/value pair list (secion 2.3.3), and uberblock array array (secion 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.

[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.

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.
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.

	Name	Value	Description
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.

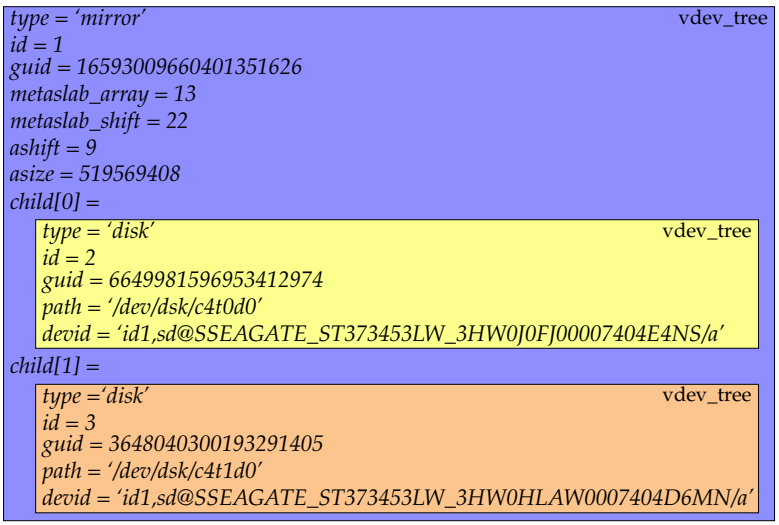

Illustration 2.3: Vdev Tree Nvlist Entries

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

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

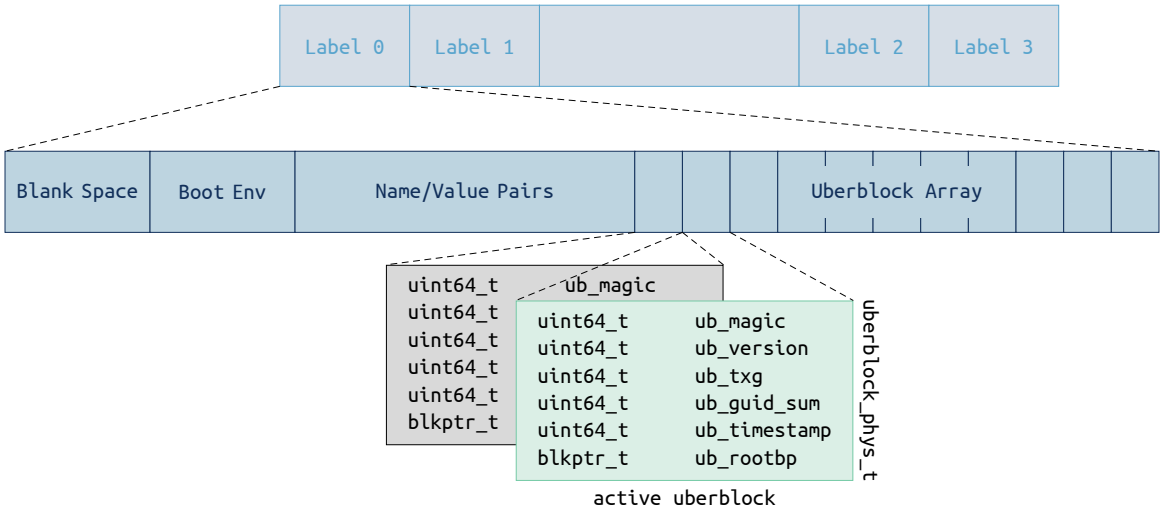

Illustration 2.4: Uberblock array showing uberblock contents

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.

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 secion 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 secion 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 5 and 3 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.

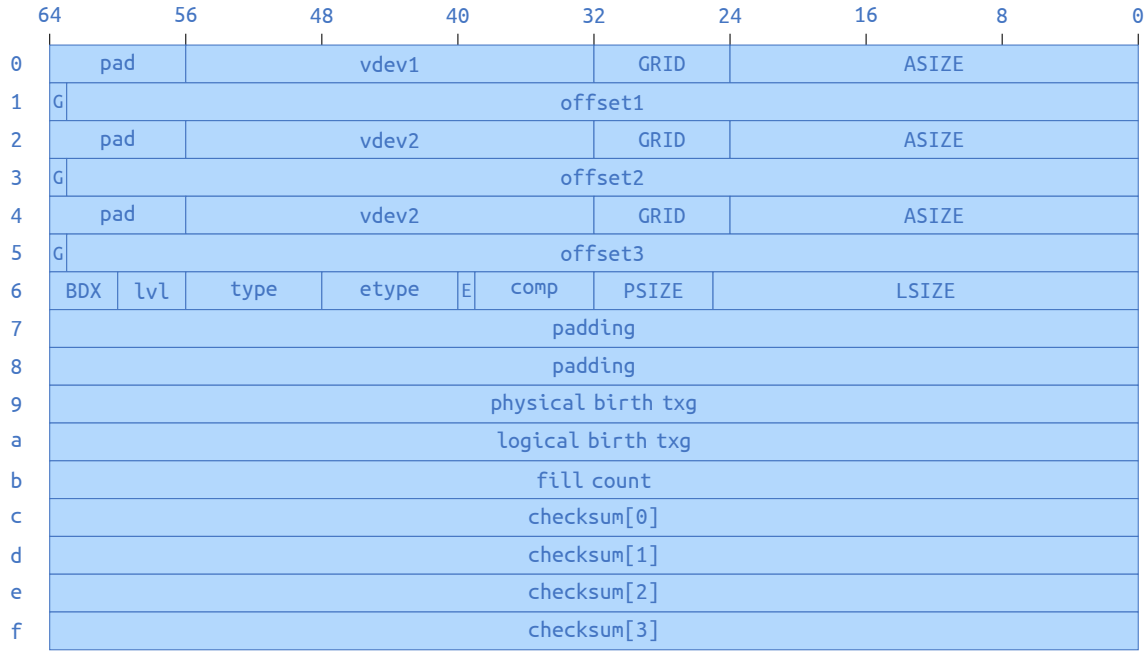

Illustration 3.1: Block pointer structure showing byte by byte usage.

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.

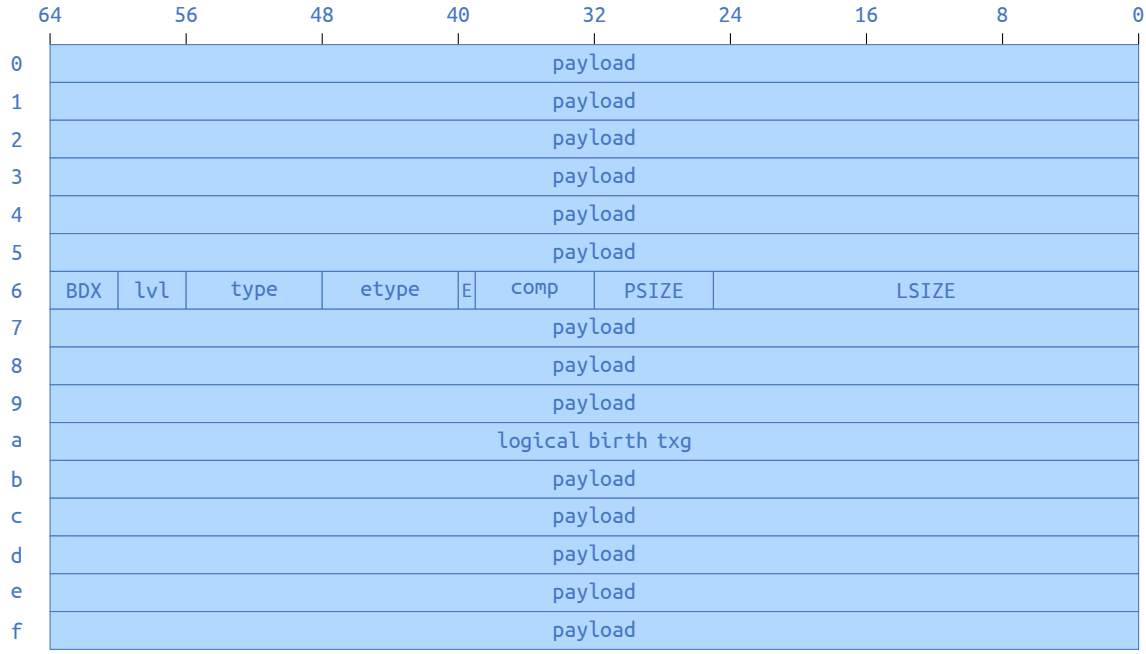

Illustration 3.2: Embedded Block Pointer Layout

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.

Gang blocks are identified by the "G" bit.

Table 3.1: Gang Block Values

"G" bit value	**Description**
0	non-gang block
1	gang block

Gang blocks are 512 byte sized, self checksumming blocks. A gang block contains up to 3 block pointers followed by a 32 byte checksum. The format of the gang block is described by the following structures.

```
typedef struct zio_gbh {
    blkptr_t        zg_blkptr[SPA_GBH_NBLKPTRS];
    uint64_t        zg_filler[SPA_GBH_FILLER];
    zio_eck_t       zg_tail;
} zio_gbh_phys_t;
```

zg_blkptr : Array of block pointers. Each 512 byte gang block can hold up to 3 block pointers.

zg_filler : The filler fields pads out the gang block so that it is nicely byte aligned.

```
typedef struct zio_eck {
    uint64_t    zec_magic;
    zio_cksum_t zec_cksum;
} zio_eck_t;
```

zec_magic: ZIO block tail magic number. The value is `0x210da7ab10c7a11` (zio-data-bloc-tail).

```
typedef struct zio_cksum {
    uint64_t    zc_word[4];
} zio_cksum_t;
```

zc_word: Four 8 byte words containing the checksum for this gang block.

3.4 Checksum

By default ZFS checksums all of its data and metadata. ZFS supports several algorithms for checksumming including fletcher2, fletcher4, and SHA-256 (256-bit Secure Hash Algorithm in FIPS 180-2, available at http://csrc.nist.gov/cryptval). The algorithm used to checksum this block is identified by the 8 bit integer stored in the cksum portion of the block pointer. The following table pairs each integer with a description and algorithm used to checksum this block's contents.

Table 3.2: Checksum Values and associated algorithms

Description	Value	Algorithm
on	1	fletcher2
off	2	none
label	3	SHA-256
gang header	4	SHA-256
zilog	5	fletcher2
fletcher2	6	fletcher2
fletcher4	7	fletcher4
SHA-256	8	SHA-256
zilog2	9	
noparity	10	
SHA-512	11	

Description	Value	Algorithm
skein	12	

A 256 bit checksum of the data is computed for each block using the algorithm identified in cksum. If the cksum value is 2 (off), a checksum will not be computed and checksum[0], checksum[1], checksum[2], and checksum[3] will be zero. Otherwise, the 256 bit checksum computed for this block is stored in the checksum[0], checksum[1], checksum[2], and checksum[3] fields.

Note: The computed checksum is always of the data, even if this is a gang block. Gang blocks (see above) and zilog blocks (see Chapter 8) are self checksumming.

3.5 Compression

ZFS supports several algorithms for compression. The type of compression used to compress this block is stored in the comp portion of the block pointer.

Table 3.3: Compression Values and associated algorithms

Description	Value	Algorithm
on	1	lz4
off	2	none
lzjb	3	lzjb
empty	4	empty
gzip level 1~9	5~13	gzip1~9
zle	14	zle
lz4	15	lz4
zstd	16	zstd

3.6 Block Size

The size of a block is described by three different fields in the block pointer; *psize*, *lsize*, and *asize*.

lsize: Logical size. The size of the data without compression, raidz or gang overhead.

psize: Physical size of the block on disk after compression.

asize: Allocated size, total size of all blocks allocated to hold this data including any gang headers or raid-Z parity information

If compression is turned off and ZFS is not on Raid-Z storage, lsize, asize, and psize will all be equal.

All sizes are stored as the number of 512 byte sectors (minus one) needed to represent the size of this block.

3.7 Endian

ZFS is an adaptive-endian filesystem (providing the restrictions described in Chapter One) that allows for moving pools across machines with different architectures: little endian vs. big endian. The "E" portion of the block pointer indicates which format this block has been written out in. Block are always written out in the machine's native endian format.

Table 3.4: Endian Values

Endian	Value
Little Endian	1
Big Endian	2

If a pool is moved to a machine with a different endian format, the contents of the block are byte swapped on read.

3.8 Type

The *type* portion of the block pointer indicates what type of data this block holds. The type can be the following values. More detail is provided in chapter 4 regarding object types.

Table 3.5: Object Types

Type	Value
DMU_OT_NONE	0
DMU_OT_OBJECT_DIRECTORY	1
DMU_OT_OBJECT_ARRAY	2
DMU_OT_PACKED_NVLIST	3
DMU_OT_NVLIST_SIZE	4
DMU_OT_BPLIST	5
DMU_OT_BPLIST_HDR	6
DMU_OT_SPACE_MAP_HEADER	7
DMU_OT_SPACE_MAP	8

Type	Value
DMU_OT_INTENT_LOG	9
DMU_OT_DNODE	10
DMU_OT_OBJSET	11
DMU_OT_DSL_DATASET	12
DMU_OT_DSL_DATASET_CHILD_MAP	13
DMU_OT_OBJSET_SNAP_MAP	14
DMU_OT_DSL_PROPS	15
DMU_OT_DSL_OBJSET	16
DMU_OT_ZNODE	17
DMU_OT_ACL	18
DMU_OT_PLAIN_FILE_CONTENTS	19
DMU_OT_DIRECTORY_CONTENTS	20
DMU_OT_MASTER_NODE	21
DMU_OT_DELETE_QUEUE	22
DMU_OT_ZVOL	23
DMU_OT_ZVOL_PROP	24

3.9 Level

The *level* portion of the block pointer is the number of levels (number of block pointers which need to be traversed to arrive at this data.) See Chapter 4 for a more complete definition of level.

3.10 Fill

The *fill* count describes the number of non-zero block pointers under this block pointer. The fill count for a data block pointer is 1, as it does not have any block pointers beneath it. The fill count is used slightly differently for block pointers of type DMU_OT_DNODE. For block pointers of this type, the fill count contains the number of free dnodes beneath this block pointer. For more information on dnodes see Chapter 4.

3.11 Birth Transaction

The birth transaction stored in the "physical birth txg" and "logical birth txg" block pointer field is a 64 bit integer containing the transaction group number in which this block pointer was allocated.

3.12 Padding

The two padding fields in the block pointer are space reserved for future use.

Chapter 4

Data Management Unit

The *Data Management Unit* (*DMU*) consumes blocks and groups them into logical units called objects. Objects can be further grouped by the DMU into object sets. Both objects and object sets are described in this chapter.

4.1 Objects

With the exception of a small amount of infrastructure, described in chapters 2 and 3, everything in ZFS is an object. The following table lists existing ZFS object types; many of these types are described in greater detail in future chapters of this document.

Table 4.1: DMU Ojbect Types

Type	Description
DMU_OT_NONE	Unallocated object
DMU_OT_OBJECT_DIRECTORY	DSL object directory ZAP object
DMU_OT_OBJECT_ARRAY	Object used to store an array of object numbers.
DMU_OT_PACKED_NVLIST	Packed nvlist object.
DMU_OT_SPACE_MAP	SPA disk block usage list.
DMU_OT_INTENT_LOG	Intent Log
DMU_OT_DNODE	Object of dnodes (metadnode)
DMU_OT_OBJSET	Collection of objects.
DMU_OT_DSL_DATASET_CHILD_MAP	DSL ZAP object containing child DSL directory information.
DMU_OT_DSL_OBJSET_SNAP_MAP	DSL ZAP object containing snapshot information for a dataset.

Type	Description
DMU_OT_DSL_PROPS	DSL ZAP properties object containing properties for a DSL dir object.
DMU_OT_BPLIST	Block pointer list – used to store the "deadlist": list of block pointers deleted since the last snapshot, and the "deferred free list" used for sync to convergence.
DMU_OT_BPLIST_HDR	BPLIST header: stores the bplist_phys_t structure.
DMU_OT_ACL	ACL (Access Control List) object
DMU_OT_PLAIN_FILE	ZPL Plain file
DMU_OT_DIRECTORY_CONTENTS	ZPL Directory ZAP Object
DMU_OT_MASTER_NODE	ZPL Master Node ZAP object: head object used to identify root directory, delete queue, and version for a filesystem.
DMU_OT_DELETE_QUEUE	The delete queue provides a list of deletes that were in-progress when the filesystem was force unmounted or as a result of a system failure such as a power outage. Upon the next mount of the filesystem, the delete queue is processed to remove the files/dirs that are in the delete queue. This mechanism is used to avoid leaking files and directories in the filesystem.
DMU_OT_ZVOL	ZFS volume (ZVOL)
DMU_OT_ZVOL_PROP	ZVOL properties

Objects are defined by 512 bytes structures called dnodes[1]. A dnode describes and organizes a collection of blocks making up an object. The dnode (`dnode_phys_t` structure), seen in the illustration below, contains several fixed length fields and two variable length fields. Each of these fields are described in detail below.

dn_type: An 8-bit numeric value indicating an object's type. See Table 8???TBD for a list of valid object types and their associated 8 bit identifiers.

dn_indblkshift and dn_datablkszsec: ZFS supports variable data and indirect (see `dn_nlevels` below for a description of indirect blocks) block sizes ranging from 512 bytes to 128 Kbytes.

[1] A dnode is similar to an *inode* in UFS.

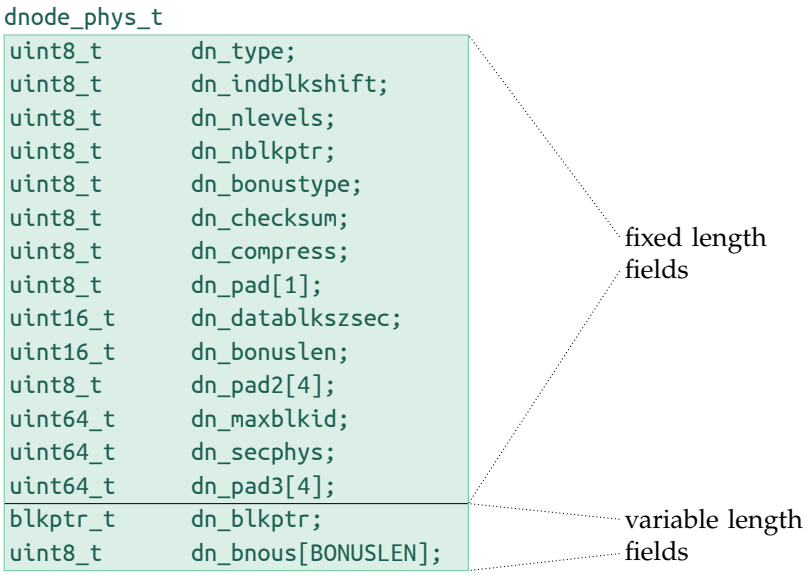

Illustration 4.1: dnode_phys_t structure

dn_indblkshift: 8-bit numeric value containing the log (base 2) of the size (in bytes) of an indirect block for this object.

dn_datablkszsec: 16-bit numeric value containing the data block size (in bytes) divided by 512 (size of a disk sector). This value can range between 1 (for a 512 byte block) and 256 (for a 128 Kbyte block).

dn_nblkptr and dn_blkptr: dn_blkptr is a variable length field that can contains between one and three block pointers. The number of block pointers that the dnode contains is set at object allocation time and remains constant throughout the life of the dnode.

dn_nblkptr: 8-bit numeric value containing the number of block pointers in this dnode.

dn_blkptr: block pointer array containing dn_nblkptr block pointers

dn_nlevels: dn_nlevels is an 8-bit numeric value containing the number of levels that make up this object. These levels are often referred to as levels of indirection.

Indirection

A dnode has a limited number (`dn_nblkptr`, see above) of block pointers to describe an object's data. For a dnode using the largest data block size (128KB) and containing the maximum number of block pointers (3), the largest object size it can represent (without indirection) is 384 KB: $3 \times 128\text{KB} = 384\text{KB}$.

To allow for larger objects, indirect blocks are used. An indirect block is a block containing block pointers. The number of block pointers that an indirect block can hold is dependent on the indirect block size (represented by `dn_indblkshift`) and can be calculated by dividing the indirect block size by the size of a blkptr (128 bytes). The largest indirect block (128KB) can hold up to 1024 block pointers. As an object's size increases, more indirect blocks and levels of indirection are created. A new level of indirection is created once an object grows so large that it exceeds the capacity of the current level. ZFS provides up to six levels of indirection to support files up to 2^{64} bytes long.

The illustration below shows an object with 3 levels of blocks (level 0, level 1, and level 2). This object has triple wide block pointers (dva1, dva2, and dva3) for metadata and single wide block pointers for its data (see Chapter 3 for a description of block pointer wideness). The blocks at level 0 are data blocks.

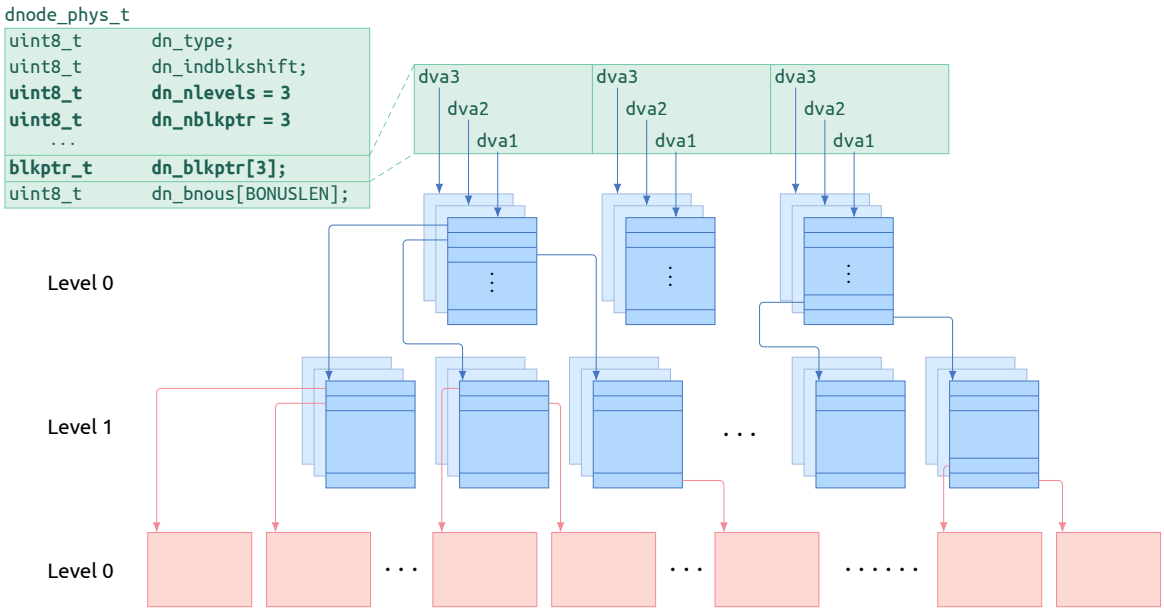

Illustration 4.2: Object with 3 levels. Triple wide block pointers used for metadata; single wide block pointers used for data

dn_maxblkid An object's blocks are identified by block ids. The blocks in each level of indirection are numbered from 0 to N, where the first block at a given level is given an id of 0, the second an id of 1, and so forth.

The dn_maxblkid field in the dnode is set to the value of the largest data (level zero) block id for this object.

Note on Block Ids: Given a block id and level, ZFS can determine the exact branch of indirect blocks which contain the block. This calculation is done using the block id, block level, and number of block pointers in an indirect block. For example, take an object which has 128KB sized indirect blocks. An indirect block of this size can hold 1024 block pointers. Given a level 0 block id of 16360, it can be determined that block 15 (block id 15) of level 1 contains the block pointer for level 0 blkid 16360.

$$level\ 1\ blkid = 16360\%1024 = 15$$

This calculation can be performed recursively up the tree of indirect blocks until the top level of indirection has been reached.

dn_secphys: The sum of all asize values for all block pointers (data and indirect) for this object.

dn_bonus, dn_bonuslen, and dn_bonustype: The bonus buffer (dn_bonus) is defined as the space following a dnode's block pointer array (dn_blkptr). The amount of space is dependent on object type and can range between 64 and 320 bytes.

dn_bonus_len: Length (in bytes) of the bonus buffer dn_bonus:

`dn_bonuslen` sized chunk of data. The format of this data is defined by dn_bonustype. dn_bonustype:

8-bit numeric value identifying the type of data contained within the bonus buffer. The following table shows valid bonus buffer types and the structures which are stored in the bonus buffer. The contents of each of these structures will be discussed later in this specification.

Table 4.3: Bonus Buffer Types and associated structures

Bonus Type	Description	Metadata Structure	Value
DMU_OT_PACKED_NVLIST_SIZE	Bonus buffer type containing size in bytes of a DMU_OT_PACKED_NVLIST object	uint64_t	4

Bonus Type	Description	Metadata Structure	Value
DMU_OT_SPACE_MAP_HEADER	Spa space map header	space_map_obj_t	7
DMU_OT_DSL_DIR	DSL Directory object used to define relationships and properties between related datasets	dsl_dir_phys_t	12
DMU_OT_DSL_DATASET	DSL dataset object used to organize snapshot and usage static information for objects of type DMU_OT_OBJSET.	dsl_dataset_phys_t	16
DMU_OT_ZNODE	ZPL metadata	znode_phys_t	17

4.2 Object Sets

The DMU organizes objects into groups called object sets. Object sets are used in ZFS to group related objects, such as objects in a filesystem, snapshot, clone, or volume.

Object sets are represented by a 1K byte `objset_phys_t` structure. Each member of this structure is defined in detail below.

TBD

Chapter 5

Dataset and Snapshot Layer

The *DSL* (*Dataset and Snapshot Layer*) provides a mechanism for describing and managing relationships-between and properties-of object sets. Before describing the DSL and the relationships it describes, a brief overview of the various flavors of object sets is necessary.

5.1 Object Set Overview

ZFS provides the ability to create four kinds of object sets: *filesystems*, *clones*, *snapshots*, and *volumes*.

- ZFS filesystems: A filesystem stores and organizes objects in an easily accessible, POSIX compliant manner.
- ZFS clone: A clone is identical to a filesystem with the exception of its origin. Clones originate from snapshots and their initial contents are identical to that of the snapshot from which it originated.
- ZFS snapshot: A snapshot is a read-only version of a filesystem, clone, or volume at a particular point in time.
- ZFS volume: A volume is a logical volume that is exported by ZFS as a block device.

ZFS supports several operations and/or configurations which cause interdependencies amongst object sets. The purpose of the DSL is to manage these relationships. The following is a list of such relationships.

- Clones: A clone is related to the snapshot from which it originated. Once a clone is created, the snapshot in which it originated can not be deleted unless the clone is also deleted.

- Snapshots: A snapshot is a point-in-time image of the data in the object set in which it was created. A filesystem, clone, or volume can not be destroyed unless its snapshots are also destroyed.
- Children: ZFS support hierarchically structured object sets; object sets within object sets. A child is dependent on the existence of its parent. A parent can not be destroyed without first destroying all children.

5.2 DSL Infrastructure

Each object set is represented in the DSL as a dataset. A dataset manages space consumption statistics for an object set, contains object set location information, and keeps track of any snapshots inter-dependencies.

Datasets are grouped together hierarchically into collections called Dataset Directories. Dataset Directories manage a related grouping of datasets and the properties associated with that grouping. A DSL directory always has exactly one "active dataset". All other datasets under the DSL directory are related to the "active" dataset through snapshots, clones, or child/parent dependencies.

The following picture shows the DSL infrastructure including a pictorial view of how object set relationships are described via the DSL datasets and DSL directories. The top level DSL Directory can be seen at the top/center of this figure. Directly below the DSL Directory is the "active dataset". The active dataset represents the live filesystem. Originating from the active dataset is a linked list of snapshots which have been taken at different points in time. Each dataset structure points to a DMU Object Set which is the actual object set containing object data. To the left of the top level DSL Directory is a child ZAP object containing a listing of all child/parent dependencies. To the right of the DSL directory is a properties ZAP object containing properties for the datasets within this DSL directory. A listing of all properties can be seen in Table 12? TBD below.

A detailed description of Datasets and DSL Directories are described in the Dataset Internals (section 5.4) and DSL Directory Internals. (section 5.5) sections below.

5.3 DSL Implementation Details

TBD

5.4 Dataset Internals

TBD

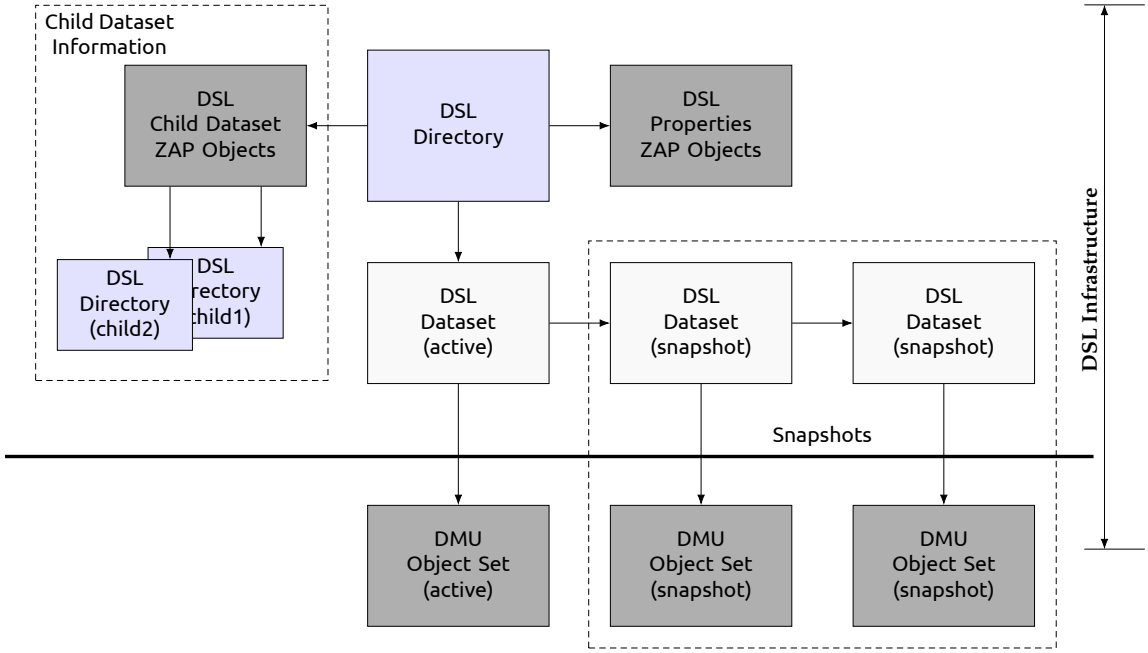

Illustration 5.1: DSL Infrastructure

5.5 DSL Directory Internals

TBD