ZFS On-Disk Specification

Draft

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

Contents

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 4), DMU (Chapter 3), ZAP (Chapter 5), ZPL (Chapter 6), ZIL (Chapter 7), ZVOL (Chapter 8).

Chapter 1

Vdev, Labels, and Boot Block

1.1 Virutal 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 1.1 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.

1.2 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 1.3, 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

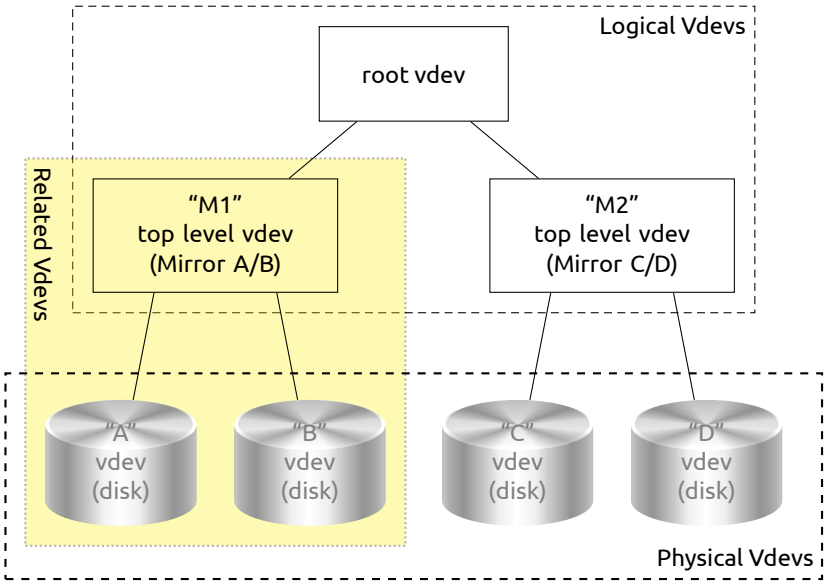

Illustration 1.1: Vdev Tree Sample Configuration

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.

1.2.1 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 1.2 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.

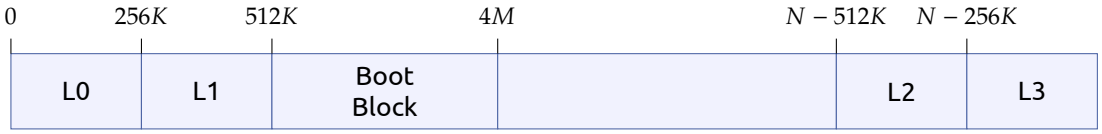

Illustration 1.2: Vdev Label layout on a block device of size N

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).

1.2.2 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.

1.3 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 1.3.1), boot block header (section 1.3.2), name/value pair list (section 1.3.3), and uberblock array (section 1.3.4).

1.3.1 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.

1.3.2 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.

1.3.3 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

[1] Disk labels describe disk partition and slice information. See `fdisk`(1 M) and/or `format`(1 M) 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.

device A (seen in Illustration 1.1) 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`(3 LIB) and `nvlist_free`(3 NVPAIR) man pages. The following name-value pairs (Table 1.1) are contained within this 112KB portion of the vdev_label.

Table 1.1: Name-Value Pairs within vdev_label

Name	Value	Description
"version"	DATA_TYPE_UINT64	On disk format version. Current value is "5000".
"name"	DATA_TYPE_STRING	Name of the pool in which this vdev belongs.
"state"	DATA_TYPE_UINT64	State of this pool. The following table shows all existing pool states. The Table 1.2 shows all existing pool states.
"txg"	DATA_TYPE_UINT64	Transaction group number in which this label was written to disk.
"pool_guid"	DATA_TYPE_UINT64	Global unique identifier (guid) for the pool.
"top_guid"	DATA_TYPE_UINT64	Global unique identifier for the top-level vdev of this subtree.
"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 1.2: 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 1.3. 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 below shows what the "vdev_tree" entry might look like for "vdev A" as shown in Illustration 1.2 earlier in this document.

Illustration 1.3: Vdev Tree Nvlist Entries

Table 1.3: 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.

Continued on next page

Table 1.3 – continued from previous page

Name	Value	Description
“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.

1.3.4 The Uberblock

Immediately following the nvpair lists in the vdev label is an array of *uberblocks*.

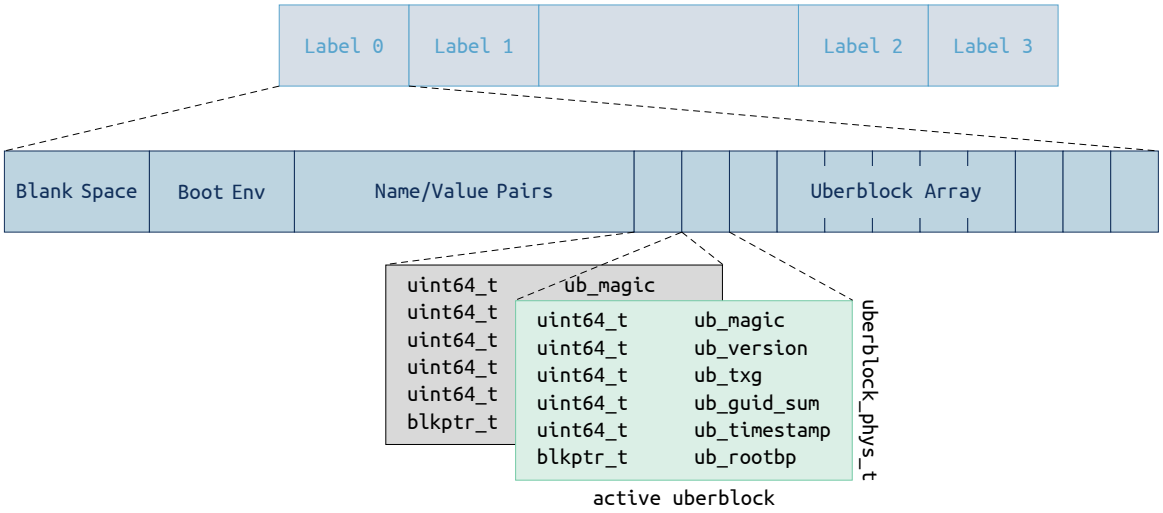

Illustration 1.4: Uberblock array showing uberblock contents

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 1.4 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 1.4 shows the ub_magic number as seen on disk.

Table 1.4: 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 "5000". This field contains the same value as the version element of the name/value pairs described in Section 1.3.3.

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.

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

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 1.3.3) 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.

1.4 Bool Block

Immediately following the L0 and L1 labels is a 3.5MB chunk reserved for future use (see Illustration 1.2). The contents of this block will be described in a future appendix of this paper.

Chapter 2

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 Figure 2.1.

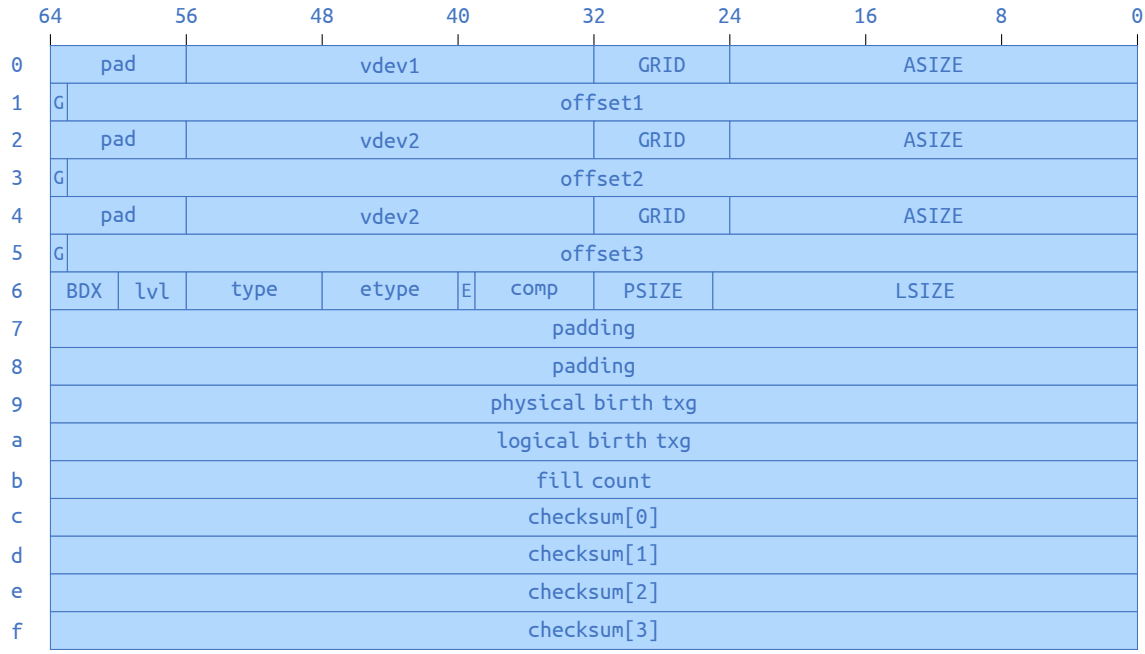

Illustration 2.1: Block pointer structure showing byte by byte usage

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 Figure 2.2.

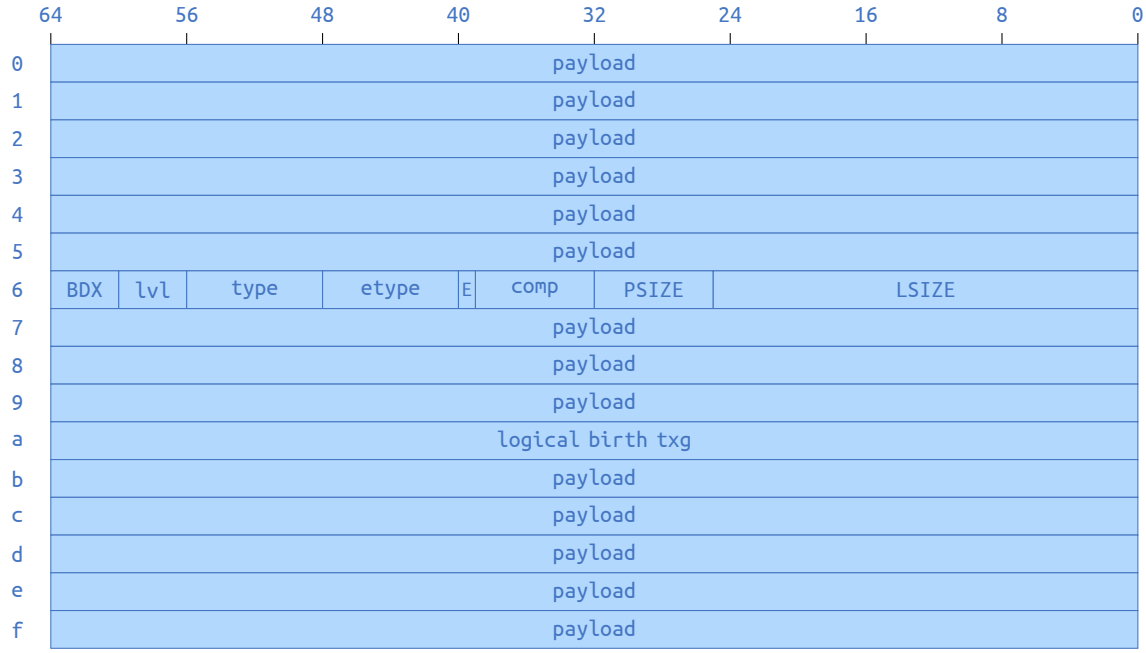

Illustration 2.2: Embedded Block Pointer Layout

2.1 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).

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

2.2 GRID

Raid-Z layout information, reserved for future use.

2.3 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 2.1: 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.

2.4 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.

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 7) are self checksumming.

Table 2.2: Checksum Values and associated algorithms

Name	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

Continued on next page

Table 2.2 – continued from previous page

Name	Value	Algorithm
SHA-256	8	SHA-256
zilog2	9	fletcher2
noparity	10	???
SHA-512	11	SHA-512
skein	12	???

2.5 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 2.3: 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

2.6 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.

2.7 Endian

ZFS is an adaptive-endian filesystem (providing the restrictions described in Chapter 1) 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 2.4: 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.

2.8 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 3 regarding object types.

Table 2.5: Object Types

Type	Value
DMU_OT_NONE	0
DMU_OT_OBJECT_DIRECTORY	1
DMU_OT_OBJECT_ARRAY	2

Continued on next page

Table 2.5 – continued from previous page

Type	Value
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
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

2.9 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 3 for a more complete definition of level.

2.10 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 3.

2.11 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.

2.12 Padding

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

Chapter 3

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.

3.1 Objects

With the exception of a small amount of infrastructure, described in Chapters 1 and 2, 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 3.1: Object 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.

Continued on next page

Table 3.1 – continued from previous page

Type	Description
DMU_OT_DSL_OBJSET_SNAP_MAP	DSL ZAP object containing snapshot information for a dataset.
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.

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

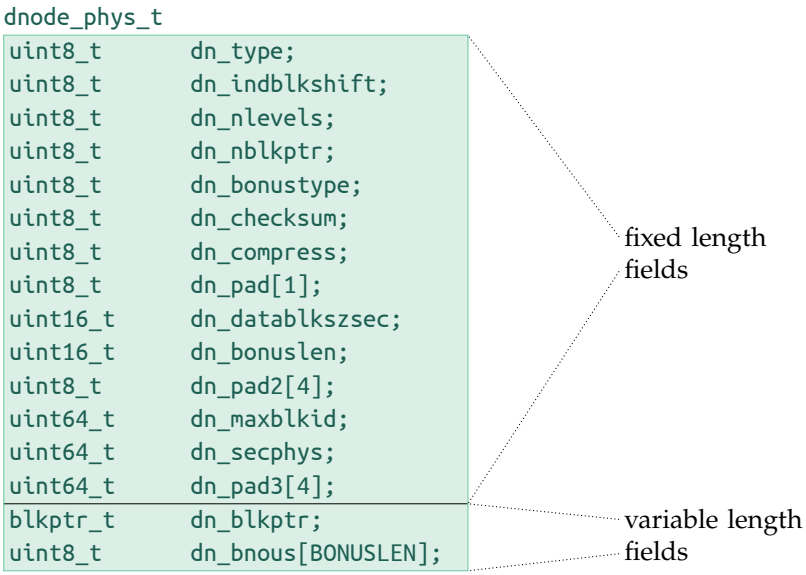

Illustration 3.1: `dnode_phys_t` structure

dn_type
An 8-bit numeric value indicating an object's type. See Table 2.5 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.

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 2 for a description of block pointer wideness). The blocks at level 0 are data blocks.

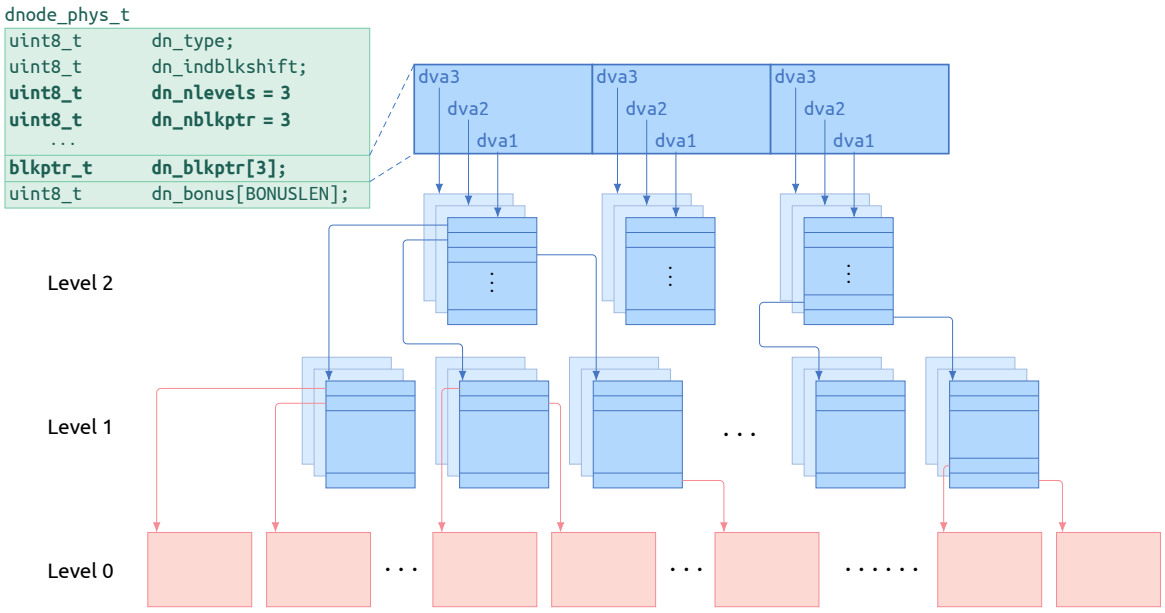

Illustration 3.2: Object with 3 levels. Triple wide block pointers used for metadata; single wide block pointers used for data

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. Table 3.2 below 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 3.2: 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
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

3.2 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.

```
objset_phys_t
dnode_phys_t       os_metadnode;
zil_header_t       os_zil_header;
uint64_t           os_type;
uint64_t           os_flags;
uint8_t            os_portable_mac[];
uint8_t            os_local_mac[];
char               os_pad0[];
dnode_phys_t       os_userused_dnode;
dnode_phys_t       os_groupused_dnode;
dnode_phys_t       os_projectused_dnode;
char               os_pad1[];
```

Illustration 3.3: `objset_phys_t` structure

os_type

The DMU supports several types of object sets, where each object set type has it's own well defined format/layout for its objects. The object set's type is identified

by a 64 bit integer, `os_type`. The table below lists available DMU object set types and their associated `os_type` integer value.

Table 3.3: DMU Object Set Types

Object Set Type	Description	Value
DMU_OST_NONE	Uninitialized Object Set	0
DMU_OST_META	DSL Object Set, see Chapter 4	1
DMU_OST_ZFS	ZPL Object Set, see Chapter 6	2
DMU_OST_ZVOL	ZVOL Object Set, see Chapter 8	3

os_zil_header
The ZIL header is described in detail in Chapter 7 of this document.

os_flags
TBD

os_portable_mac
TBD

os_local_mac
TBD

os_userused_dnode
TBD

os_groupused_dnode
TBD

os_projectused_dnode
TBD

os_metadnode
As described earlier in this chapter, each object is described by a `dnode_phys_t`. The collection of `dnode_phys_t` structures describing the objects in this object set are stored as an object pointed to by the metadnode. The data contained within this object is formatted as an array of `dnode_phys_t` structures (one for each object within the object set).

Each object within an object set is uniquely identified by a 64 bit integer called an object number. An object's "object number" identifies the array element, in the dnode array, containing this object's `dnode_phys_t`.

The illustration below shows an object set with the metadnode expanded. The metadnode contains three block pointers, each of which have been expanded to

show their contents. Object number 4 has been further expanded to show the details of the dnode_phys_t and the block structure referenced by this dnode.

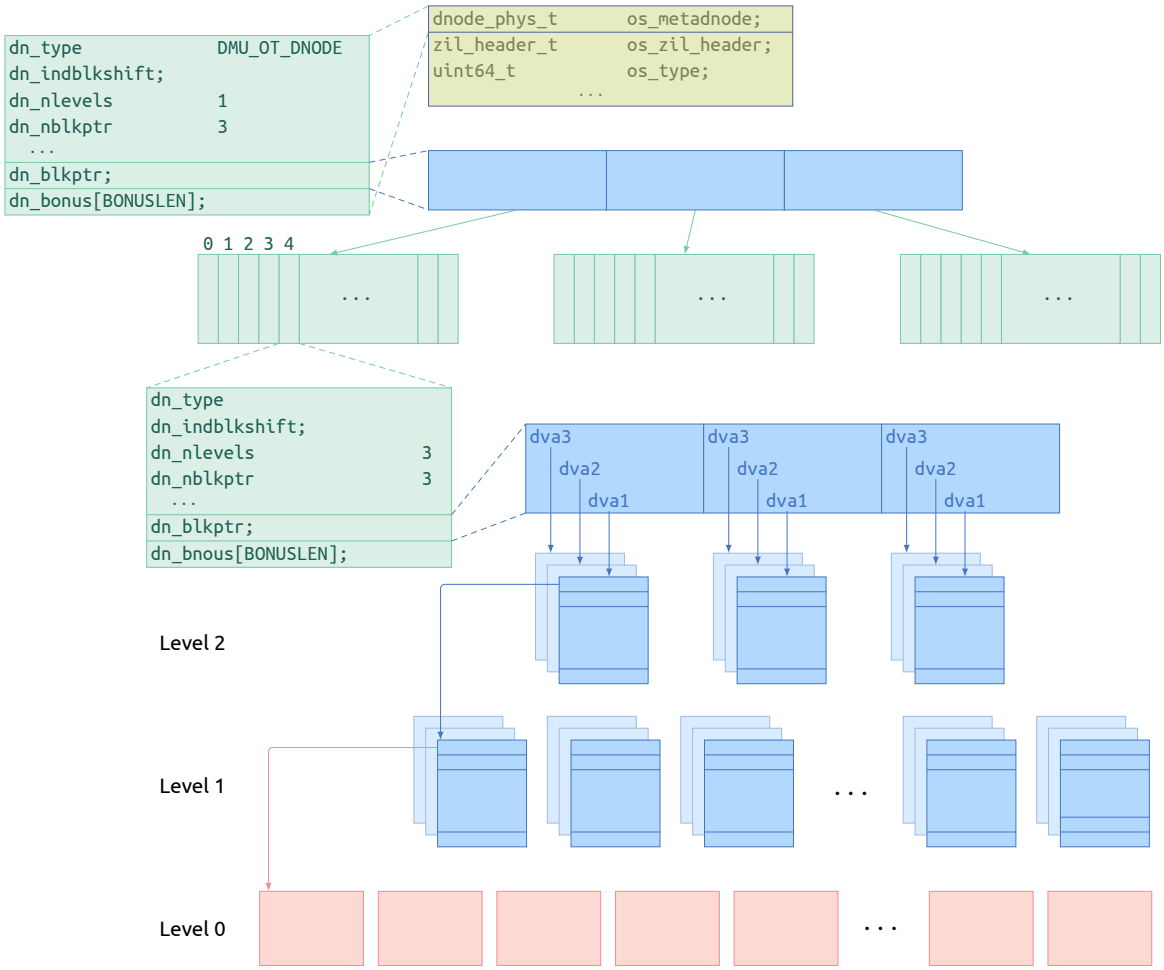

Illustration 3.4: Object Set

3.2.1 zap_phys_t

The first block of a fatzap object contains a 128KB zap_phys_t structure. Depending on the size of the pointer table, this structure may contain the pointer table. If the pointer table is too large to fit in the space provided by the zap_phys_t, some information about where it can be found is stored in the zap_table_phys portion of this structure. The definitions of the zap_phys_t contents are as follows:

Chapter 4

Dataset and Snapshot Layer

The *Dataset and Snapshot Layer*, or *DSL* 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.

4.0.1 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.

4.1 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.

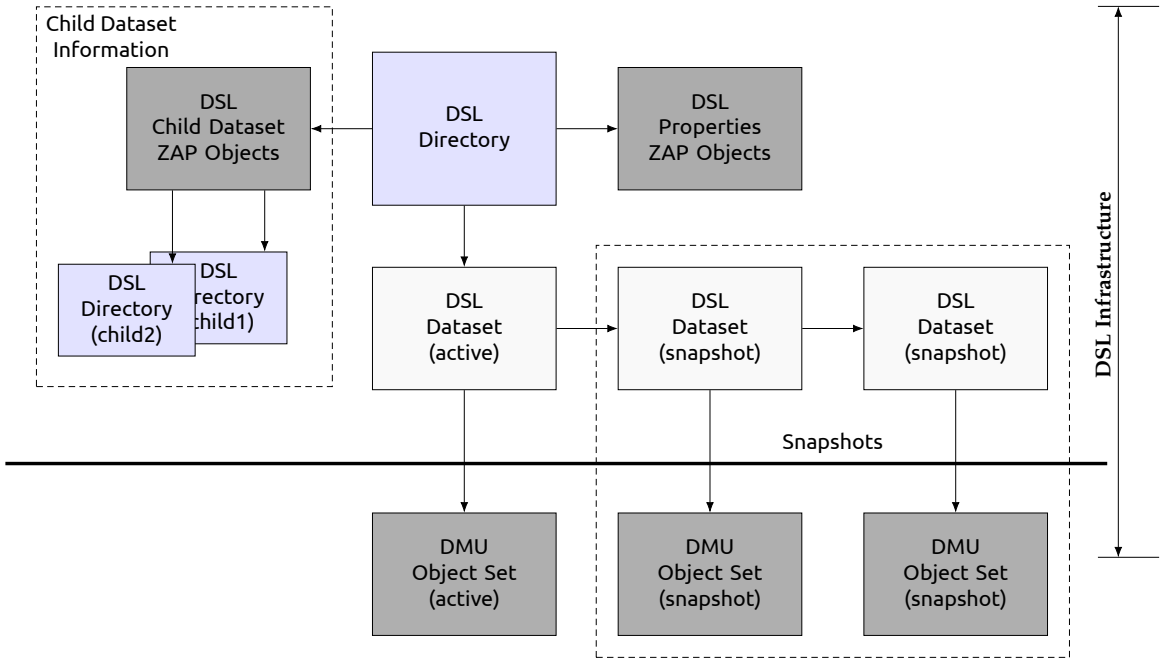

Illustration 4.1: DSL Infrastructure

The Illustration 4.1 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 4.1 below.

A detailed description of Datasets and DSL Directories are described in the Dataset Internals (section 4.3) and DSL Directory Internals (section 4.4) sections below.

4.2 DSL Implementation Details

The DSL is implemented as an object set of type `DMU_OST_META`. This object set is often called the *Meta Object Set*, or *MOS*. There is only one MOS per pool and the uberblock (see Chapter 1) points to it directly.

There is a single distinguished object in the Meta Object Set. This object is called the object directory and is always located in the second element of the dnode array (index 1). All objects, with the exception of the object directory, can be located by traversing through a set of object references starting at this object.

4.2.1 The object directory

The object directory is a ZAP object (an object containing name/value pairs – see Chapter 5 for a description of ZAP objects) containing three attribute pairs (name/-value) named: root_dataset, config, and sync_bplist.

root_dataset:
The "root_dataset" attribute contains a 64 bit integer value identifying the object number of the root DSL directory for the pool. The root DSL directory is a special object whose contents reference all top level datasets within the pool. The "root_dataset" directory, is an object of type `DMU_OT_DSL_DIR` and will be explained in greater detail in Section 4.4: DSL Directory Internals

config:
The config attribute contains a 64 bit integer value identifying the object number for an object of type `DMU_OT_PACKED_NVLIST`. This object contains XDR_ENCODED name value pairs describing this pools vdev configuration. Its contents are similar to those described in Section 1.3.3: Name-Value Pair List.

sync_bplist:
The "sync_bplist" attribute contains a 64 bit integer value identifying the object number for an object of type `DMU_OT_SYNC_BPLIST`. This object contains a list of block pointers which need to be freed during the next transaction.

The illustration below shows the meta object set (MOS) in relation to the uberblock and label structures discussed in Chapter 1.

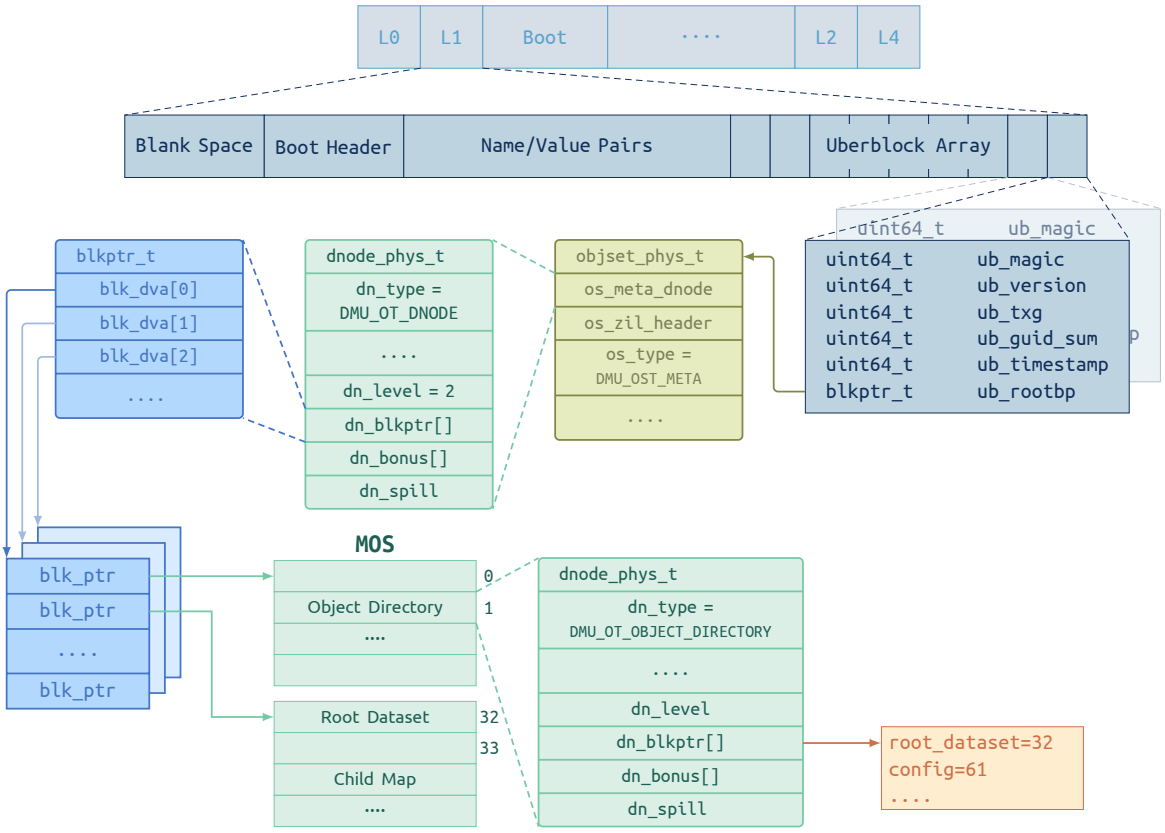

Illustration 4.2: Meta Object Set

4.3 Dataset Internals

Datasets are stored as an object of type `DMU_OT_DSL_DATASET`. This object type uses the bonus buffer in the `dnode_phys_t` to hold a `dsl_dataset_phys_t` structure. The contents of the `dsl_dataset_phys_t` structure are shown below.

uint64_t ds_dir_obj
: Object number of the DSL directory referencing this dataset.

uint64_t ds_prev_snap_obj
: If this dataset represents a filesystem, volume, or clone, this field contains the 64 bit object number for the most recent snapshot taken; this field is zero if no snapshots have been taken.

  If this dataset represents a snapshot, this field contains the 64 bit object number for the snapshot taken prior to this snapshot. This field is zero if there are no previous snapshots.

uint64_t ds_prev_snap_txg
The transaction group number when the previous snapshot (pointed to by `ds_prev_snap_obj`) was taken.

uint64_t ds_next_snap_ob
This field is only used for datasets representing snapshots. It contains the object number of the dataset which is the most recent snapshot. This field is always zero for datasets representing clones, volumes, or filesystems.

uint64_t ds_snapnames_zapobj
Object number of a ZAP object (see Chapter 5) containing name value pairs for each snapshot of this dataset. Each pair contains the name of the snapshot and the object number associated with it's DSL dataset structure.

uint64_t ds_num_children
Always zero if not a snapshot. For snapshots, this is the number of references to this snapshot: 1 (from the next snapshot taken, or from the active dataset if no snapshots have been taken) + the number of clones originating from this snapshot.

uint64_t ds_creation_time
Seconds since January 1^{st} 1970 (GMT) when this dataset was created.

uint64_t ds_creation_txg
The transaction group number in which this dataset was created.

uint64_t ds_deadlist_obj
The object number of the deadlist (an array of blkptr's deleted since the last snapshot).

uint64_t ds_used_bytes
unique bytes used by the object set represented by this dataset

uint64_t ds_compressed_bytes
number of compressed bytes in the object set represented by this dataset

uint64_t ds_uncompressed_bytes
number of uncompressed bytes in the object set represented by this dataset

uint64_t ds_unique_bytes
When a snapshot is taken, its initial contents are identical to that of the active copy of the data. As the data changes in the active copy, more and more data becomes unique to the snapshot (the data diverges from the snapshot). As that happens, the amount of data unique to the snapshot increases. The amount of unique snapshot data is stored in this field it is zero for clones, volumes, and filesystems.

uint64_t ds_fsid_guid
64 bit ID that is guaranteed to be unique amongst all currently open datasets. Note, this ID could change between successive dataset opens.

uint64_t ds_guid
64 bit global id for this dataset. This value never changes during the lifetime of the object set.

uint64_t ds_restoring
The field is set to "1" if ZFS is in the process of restoring to this dataset through "'zfs restore" [1]

blkptr_t ds_bp
Block pointer containing the location of the object set that this dataset represents.

4.4 DSL Directory Internals

The DSL Directory object contains a `dsl_dir_phys_t` structure in its bonus buffer. The contents of this structure are described in detail below.

uint64_t dd_creation_time
Seconds since January 1st, 1970 (GMT) when this DSL directory was created.

uint64_t dd_head_dataset_obj
64 bit object number of the active dataset object

uint64_t dd_parent_obj
64 bit object number of the parent DSL directory

uint64_t dd_clone_parent_obj
For cloned object sets, this field contains the object number of snapshot used to create this clone.

uint64_t dd_child_dir_zapobj
Object number of a ZAP object containing name-value pairs for each child of this DSL directory.

uint64_t dd_used_bytes
Number of bytes used by all datasets within this directory: includes any snapshot and child dataset used bytes.

uint64_t dd_compressed_bytes
Number of compressed bytes for all datasets within this DSL directory.

uint64_t dd_uncompressed_bytes
Number of uncompressed bytes for all datasets within this DSL directory.

[1] See the ZFS Admin Guide for information about the zfs command.

uint64_t dd_quota
Designated quota, if any, which can not be exceeded by the datasets within this DSL directory.

uint64_t dd_reserved
The amount of space reserved for consumption by the datasets within this DSL directory.

uint64_t dd_props_zapobj
64 bit object number of a ZAP object containing the properties for all datasets within this DSL directory. Only the non-inherited/locally set values are represented in this ZAP object. Default, inherited values are inferred when there is an absence of an entry.

Table 4.1: Editable Property Values stored in the dd_props_zabobj

Property	Description	Values
aclinherit	Controls inheritance behavior for datasets.	discard = 0 noallow = 1 passthrough = 3 secure = 4 (default)
aclmode	Controls chmod and file/dir creation behavior for datasets.	discard = 0 groupmask = 2 (default) passthrough = 3
atime	Controls whether atime is updated on objects within a dataset.	off = 0 on = 1 (default)
checksum	Checksum algorithm for all datasets within this DSL Directory.	off = 0 on = 1 (default)
compression	Compression algorithm for all datasets within this DSL Directory.	off = 0 (default) on = 1
devices	Controls whether device nodes can be opened on datasets.	devices = 0 nodevices = 1 (default)
exec	Controls whether files can be executed on a dataset.	noexec = 0 exec = 1 (default)

Continued on next page

Table 4.1 – continued from previous page

Property	Description	Values
mountpoint	Mountpoint path for datasets within this DSL Directory.	string
quota	Limits the amount of space all datasets within a DSL directory can consume.	quota size in bytes or zero for no quota (default)
readonly	Controls whether objects can be modified on a dataset.	readwrite = 0 (default) readonly = 1
recordsize	Block Size for all objects within the datasets contained in this DSL Directory	record size in bytes
reservation	Amount of space reserved for this DSL Directory, including all child datasets and child DSL Directories.	reservation size in bytes
setuid	Controls whether the set-UID bit is respected on a dataset.	nosetuid = 0 setuid = 1 (default)
sharenfs	Controls whether the datasets in a DSL Directory are shared by NFS.	string – any valid nfs share options
snapdir	Controls whether `.zfs` is hidden or visible in the root filesystem.	hidden = 0 visible = 1 (default)
volblocksize	For volumes, specifies the block size of the volume. The blocksize cannot be changed once the volume has been written, so it should be set at volume creation time	between 512 to 128K, powers of two. Defaults to 8K.
volsize	Volume size, only applicable to volumes.	volume size in bytes
zoned	Controls whether a dataset is managed through a local zone.	off = 0 (default) on = 1

Chapter 5

ZFS Attribute Processor

The *ZFS Attribute Processor*, or *ZAP* is a module which sits on top of the DMU and operates on objects called ZAP objects. A ZAP object is a DMU object used to store attributes in the form of name-value pairs. The name portion of the attribute is a zero-terminated string of up to 256 bytes (including terminating NULL). The value portion of the attribute is an array of integers whose size is only limited by the size of a ZAP data block.

ZAP objects are used to store properties for a dataset, navigate filesystem objects, store pool properties and more. The following table contains a list of ZAP object types.

Table 5.1: ZAP Object Types

ZAP Object Type
DMU_OT_OBJECT_DIRECTORY
DMU_OT_DSL_DIR_CHILD_MAP
DMU_OT_DSL_DS_SNAP_MAP
DMU_OT_DSL_PROPS
DMU_OT_DIRECTORY_CONTENTS
DMU_OT_MASTER_NODE
DMU_OT_DELETE_QUEUE
DMU_OT_ZVOL_PROP

ZAP objects come in two forms; *microzap* objects and *fatzap* objects. Microzap objects are a lightweight version of the fatzap and provide a simple and fast lookup mechanism for a small number of attribute entries. The fatzap is better suited for ZAP objects containing large numbers of attributes.

The following guidelines are used by ZFS to decide whether or not to use a fatzap or a microzap object.

A microzap object is used if all three conditions below are met:

- all name-value pair entries fit into one block. The maximum data block size in ZFS is 128KB and this size block can fit up to 2047 microzap entries.
- The value portion of all attributes are of type `uint64_t`.
- The name portion of each attribute is less than or equal to 50 characters in length (including NULL terminating character).

If any of the above conditions are not met, a fatzap object is used.

The first 64 bit word in each block of a ZAP object is used to identify the type of ZAP contents contained within this block. The table below shows these values.

Table 5.2: ZAP Object Block Types

Identifier	Description	Value
ZBT_MICRO	This block contains microzap entries	(1ULL « 63) + 3
ZBT_HEADER	This block is used for the fatzap. This identifier is only used for the first block in a fatzap object.	(1ULL « 63) + 1
ZBT_LEAF	This block is used for the fatzap. This identifier is used for all blocks in the fatzap with the exception of the first.	(1ULL « 63) + 0

5.1 The Micro Zap

The microzap implements a simple mechanism for storing a small number of attributes. A microzap object consists of a single block containing an array of microzap entries (`mzap_ent_phys_t` structures). Each attribute stored in a microzap object is represented by one of these microzap entry structures.

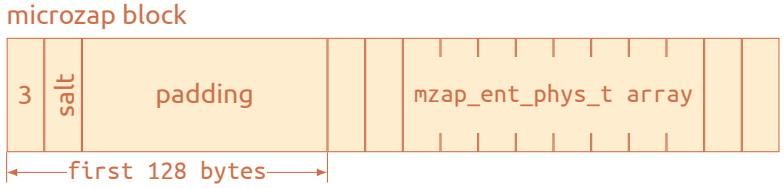

Illustration 5.1: Microzap Block Layout

A microzap block is laid out as follows: the first 128 bytes of the block contain a microzap header structure called the `mzap_phys_t`. This structure contains a 64

bit ZBT_MICRO value indicating that this block is used to store microzap entries. Following this value is a 64 bit salt value that is stirred into the hash so that the hash function is different for each ZAP object. The next 42 bytes of this header is intentionally left blank and the last 64 bytes contain the first microzap entry (a structure of type mzap_ent_phys_t). The remaining bytes in this block are used to store an array of mzap_ent_phys_t structures. The Illustration 5.1 above shows the layout of this block.

The mzap_ent_phys_t structure and associated #defines are shown below.

```
#define MZAP_ENT_LEN            64
#define MZAP_NAME_LEN           (MZAP_ENT_LEN - 8  4 - 2)
typedef struct mzap_ent_phys {
        uint64_t mze_value;
        uint32_t mze_cd;
        uint16_t  mze_pad;
        char mze_name[MZAP_NAME_LEN];
} mzap_ent_phys_t;
```

mze_value: 64 bit integer

mze_cd: 32 bit collision differentiator ("CD"): associated with an entry whose hash value is the same as another entry within this ZAP object. When an entry is inserted into the ZAP object, the lowest CD which is not already used by an entry with the same hash value is assigned. In the absence of hash collisions, the CD value will be zero.

mze_pad: reserved for future use

mze_name: NULL terminated string less than or equal to 50 characters in length

5.2 The Fat Zap

The fatzap implements a flexible architecture for storing large numbers of attributes, and/or attributes with long names or complex values (not uint64_t). This section begins with an explanation of the basic structure of a fatzap object and is followed by a detailed explanation of each component of a fatzap object.

All entries in a fatzap object are arranged based on a 64 bit hash of the attribute's name. The hash is used to index into a pointer table (as can be seen on the left side of the illustration below). The number of bits used to index into this table (sometimes called the prefix) is dependent on the number of entries in the table. The number of entries in the table can change over time. As policy stands today, the pointer table will grow if the number of entries hashing to a particular bucket

exceeds the capacity of one leaf block (explained in detail below). The pointer table entries reference a chain of fatzap blocks called leaf blocks, represented by the `zap_leaf_phys` structure. Each leaf block is broken up into some number of chunks (zap_leaf_chunks) and each attribute is stored in one or more of these leaf chunks. The illustration below shows the basic fatzap structures, each component is explained in detail in the following sections.

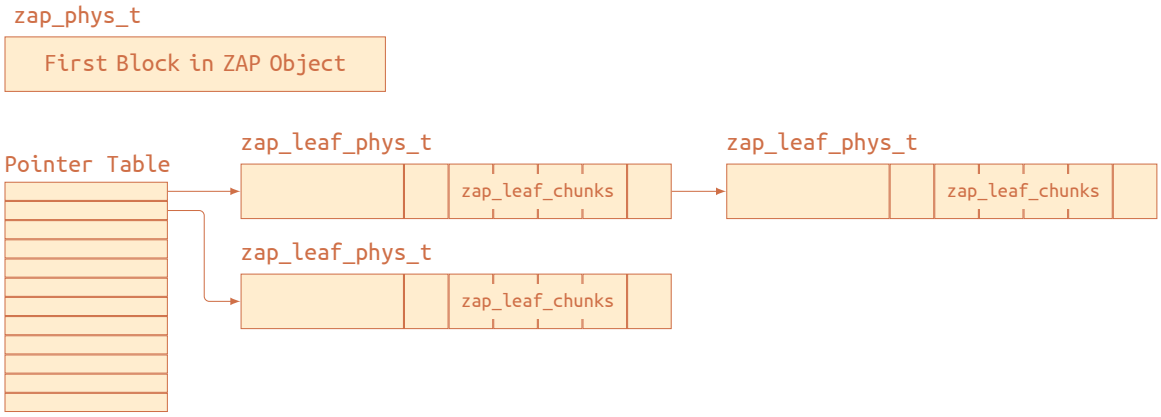

Illustration 5.2: Fatzap Structure Overview

5.2.1 zap_phys_t

```
zap_phys_t
uint64_t        zap_block_type;
uint64_t        zap_magic;
struct zap_table_phys {
        uint64_t  zt_blk;
        uint64_t  zt_numblks;
        uint64_t  zt_shift;
        uint64_t  zt_nextblk;
        uint64_t  zt_blk_copied;
}
uint64_t        zap_freeblk;
uint64_t        zap_num_leafs;
uint64_t        zap_num_entries;
uint64_t        zap_salf;
uint64_t        zap_pad[8181];
uint64_t        zap_leaf[8192];
```

Illustration 5.3: zap_phys_t Structure

The first block of a fatzap object contains a 128KB `zap_phys_t` structure. Depending on the size of the pointer table, this structure may contain the pointer table. If the pointer table is too large to fit in the space provided by the `zap_phys_t`, some

information about where it can be found is stored in the zap_table_phys portion of this structure. The definitions of the zap_phys_t contents are as follows:

zap_block_type:
64 bit integer identifying type of ZAP block. Always set to ZBT_HEADER (see Table 14) for the first block in the fatzap.

zap_magic:
64 bit integer containing the ZAP magic number 0x2F52AB2AB (zfs-zap-zap)

zap_table_phys:
structure whose contents are used to describe the pointer table

zt_blk:
Blkid for the first block of the pointer table. This field is only used when the pointer table is external to the zap_phys_t structure; zero otherwise.

zt_numblks:
Number of blocks used to hold the pointer table. This field is only used when the pointer table is external to the zap_phys_t structure; zero otherwise.

zt_shift:
Number of bits used from the hash value to index into the pointer table. If the pointer table is contained within the zap_phys_t, this value will be 13.

uint64_t zt_nextblk: uint64_t zt_blks_copied:
The above two fields are used when the pointer table changes sizes.

zap_freeblk:
64 bit integer containing the first available ZAP block that can be used to allocate a new zap_leaf.

zap_num_leafs:
Number of zap_leaf_phys_t structures (described below) contained within this ZAP object.

zap_salt:
The salt value is a 64 bit integer that is stirred into the hash function, so that the hash function is different for each ZAP object.

zap_num_entries:
Number of attributes stored in this ZAP object.

zap_leafs[8192]:
The zap_leaf array contains 213 (8192) slots. If the pointer table has fewer than 213 entries, the pointer table will be stored here. If not, this field is unused.

5.2.2 Pointer Table

The pointer table is a hash table which uses a chaining method to handle collisions. Each hash bucket contains a 64 bit integer which describes the level zero block id (see Chapter 3 for a description of block ids) of the first element in the chain of entries hashed here. An entries hash bucket is determined by using the first few bits of the 64 bit ZAP entry hash computed from the attribute's name. The value used to index into the pointer table is called the *prefix* and is the `zt_shift` high order bits of the 64 bit computed hash.

5.2.3 zap_leaf_phys_t

The `zap_leaf_phys_t` is the structure referenced by the pointer table. Collisions in the pointer table result in `zap_leaf_phys_t` structures being strung together in a link list fashion. The `zap_leaf_phys_t` structure contains a header, a hash table, and some number of chunks.

```
typedef struct zap_leaf_phys {
        struct zap_leaf_header {
                uint64_t lh_block_type;
                uint64_t lh_pad1;
                uint64_t lh_prefix;
                uint32_t lh_magic;
                uint16_t lh_nfree;
                uint16_t lh_nentries;
                uint16_t lh_prefix_len;
                uint16_t lh_freelist;
                uint8_t lh_flags;
                uint8_t lh_pad2[11];
        } l_hdr;
        /*
         * The header is followed by a hash table with
         * ZAP_LEAF_HASH_NUMENTRIES(zap) entries.
         */
        uint16_t l_hash[1];
        /*
         * The hash table is followed by an array of
         * ZAP_LEAF_NUMCHUNKS(zap) zap_leaf_chunk structures.
         * These structures are accessed with the
         * ZAP_LEAF_CHUNK() macro.
         */
        union zap_leaf_chunk {
                struct zap_leaf_entry {
                        uint8_t le_type;
```

```
                        uint8_t le_value_intlen;
                        uint16_t le_next;
                        uint16_t le_name_chunk;
                        uint16_t le_name_numints;
                        uint16_t le_value_chunk;
                        uint16_t le_value_numints;
                        uint32_t le_cd;
                        uint64_t le_hash;
                } l_entry;
                struct zap_leaf_array {
                        uint8_t la_type;
                        uint8_t la_array[ZAP_LEAF_ARRAY_BYTES];
                        uint16_t la_next;
                } l_array;
                struct zap_leaf_free {
                        uint8_t lf_type;
                        uint8_t lf_pad[ZAP_LEAF_ARRAY_BYTES];
                        uint16_t lf_next;
                } l_free;
        } l_chunk[ZAP_LEAF_NUMCHUNKS];
} zap_leaf_phys_t;
```

Header

The header for the ZAP leaf is stored in a `zap_leaf_header` structure. It's description is as follows:

lhr_block_type:
always `ZBT_LEAF` (see Table 5.2 for values)

lhr_next:
64 bit integer block id for the next leaf in a block chain.

lhr_prefix and lhr_prefix_len:
Each leaf (or chain of leafs) stores the ZAP entries whose first `lhr_prefixlen` bits of their hash value equals `lhr_prefix`. `lhr_prefixlen` can be equal to or less than `zt_shift` (the number of bits used to index into the pointer table) in which case multiple pointer table buckets reference the same leaf.

lhr_magic:
leaf magic number == `0x2AB1EAF` (zap-leaf)

lhr_nfree:
number of free chunks in this leaf (chunks described below)

lhr_nentries:
number of ZAP entries stored in this leaf

lhr_freelist:
head of a list of free chunks, 16 bit integer used to index into the zap_leaf_chunk array

Leaf Hash

The next 8KB of the zap_leaf_phys_t is the zap leaf hash table. The entries in the has table reference chunks of type zap_leaf_entry. Twelve bits (the twelve following the lhr_prefix_len used to uniquely identify this block) of the attribute's hash value are used to index into the this table. Hash table collisions are handled by chaining entries. Each bucket in the table contains a 16 bit integer which is the index into the zap_leaf_chunk array.

5.2.4 zap_leaf_chunk

Each leaf contains an array of chunks. There are three types of chunks: zap_leaf_entry , zap_leaf_array, and zap_leaf_free. Each attribute is represented by some number of these chunks: one zap_leaf_entry and some number of zap_leaf_array chunks. The illustration below shows how these chunks are arranged. A detailed description of each chunk type follows the illustration.

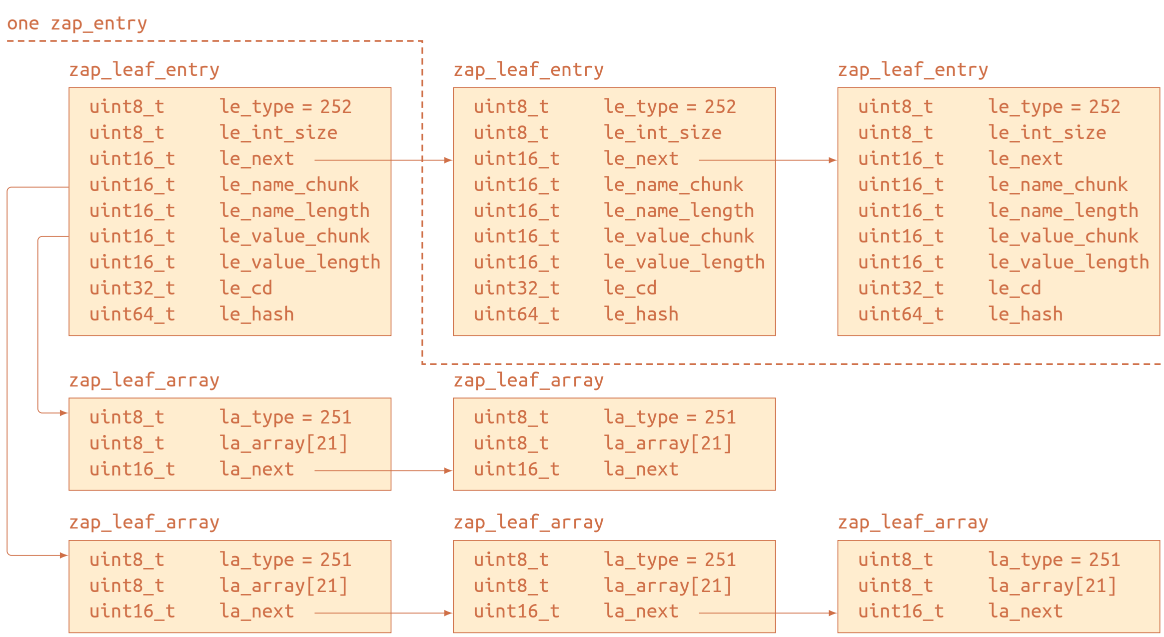

Illustration 5.4: Zap Leaf Structure

zap_leaf_entry:
The leaf hash table (described above) points to chucks of this type. This entry contains pointers to chunks of type `zap_leaf_array` which hold the name and value for the attributes being stored here.

le_type:
`ZAP_LEAF_ENTRY` == 252

le_int_size:
Size of integers in bytes for this entry.

le_next:
Next entry in the `zap_leaf_chunk` chain. Chains occur when there are collisions in the hash table. The end of the chain is designated by a `le_next` value of `0xffff`.

le_name_chunk:
16 bit integer identifying the chunk of type `zap_leaf_array` which contains the first 21 characters of this attribute's name.

le_name_length:
The length of the attribute's name, including the NULL character.

le_value_chunk:
16 bit integer identifying the first chunk (type `zap_leaf_array`) containing the first 21 bytes of the attribute's value.

le_value_length:
The length, in integer increments (`le_int_size`)

le_cd:
The collision differentiator ("CD") is a value associated with an entry whose hash value is the same as another entry within this ZAP object. When an entry is inserted into the ZAP object, the lowest CD which is not already used by an entry with the same hash value is assigned. In the absence of hash collisions, the CD value will be zero.

le_hash:
64 bit hash of this attribute's name.

zap_leaf_array:
Chunks of the `zap_leaf_array` hold either the name or the value of the ZAP attribute. These chunks can be strung together to provide for long names or large values. `zap_leaf_array` chunks are pointed to by a `zap_leaf_entry` chunk.

la_type:
`ZAP_LEAF_ARRAY` == 251

la_array:
21 byte array containing the name or value's value. Values of type "integer" are always stored in bigendian format, regardless of the machine's native endianness.

la_next:
16 bit integer used to index into the `zap_leaf_chunk` array and references the next `zap_leaf_array` chunk for this attribute; a value of `0xffff` (`CHAIN_END`) is used to designate the end of the chain

zap_leaf_free:
Unused chunks are kept in a chained free list. The root of the free list is stored in the leaf header.

lf_type:
`ZAP_LEAF_FREE` == 253

lf_next:
16 bit integer pointing to the next free chunk.

Chapter 6

ZFS Posix Layer

The *ZFS POSIX Layer*, or *ZPL* makes DMU objects look like a POSIX filesystem. POSIX is a standard defining the set of services a filesystem must provide. ZFS filesystems provide all of these required services.

The ZPL represents filesystems as an object set of type `DMU_OST_ZFS`. All snapshots, clones and filesystems are implemented as an object set of this type. The ZPL uses a well defined format for organizing objects in its object set. The section below describes this layout.

6.1 ZPL Filesystem Layout

A ZPL object set has one object with a fixed location and fixed object number. This object is called the "master node" and always has an object number of 1. The master node is a ZAP object containing three attributes: `DELETE_QUEUE`, `VERSION`, and `ROOT`.

Table 6.1: ZPL Objects

Name	Value	Description
DELECT_QUEUE	64 bit object number for the delete queue object	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 inthe delete queue. This mechanism is used to avoid leaking files and directories in the filesystem.

Continued on next page

Table 6.1 – continued from previous page

Name	Value	Description
VERSION	Currently a value of "1" (really???)	ZPL version used to lay out this filesystem.
ROOT	64bit object number	This attribute's value contains the object number for the top level directory in this filesystem, the root directory.

6.2 Directories and Directory Traversal

Filesystem directories are implemented as ZAP objects (object type `DMU_OT_DIRECTORY`). Each directory holds a set of name-value pairs which contain the names and object numbers for each directory entry. Traversing through a directory tree is as simple as looking up the value for an entry and reading that object number.

All filesystem objects contain a `znode_phys_t` structure in the bonus buffer of it's dnode. This structure stores the attributes for the filesystem object. The `znode_phys_t` structure is shown below.

```
typedef struct znode_phys {
        uint64_t zp_atime[2];
        uint64_t zp_mtime[2];
        uint64_t zp_ctime[2];
        uint64_t zp_crtime[2];
        uint64_t zp_gen;
        uint64_t zp_mode;
        uint64_t zp_size;
        uint64_t zp_parent;
        uint64_t zp_links;
        uint64_t zp_xattr;
        uint64_t zp_rdev;
        uint64_t zp_flags;
        uint64_t zp_uid;
        uint64_t zp_gid;
        uint64_t zp_zap;
        uint64_t zp_pad[3];
        fs_acl_phys_t zp_acl;
} znode_phys_t;
```

zp_atime: Two 64 bit integers containing the last file access time in seconds (`zp_atime`

[0]) and nanoseconds (zp_atime[1]) since January 1st 1970 (GMT).

zp_mtime: Two 64 bit integers containing the last file modification time in seconds (zp_mtime[0]) and nanoseconds (zp_mtime[0]) since January 1st 1970 (GMT).

zp_ctime: Two 64 bit integers containing the last file change time in seconds (zp_ctime [0]) and nanoseconds (zp_ctime[1]) since January 1st 1970 (GMT).

zp_crtime: Two 64 bit integers containing the file's creation time in seconds (zp_crtime [0]) and nanoseconds (zp_crtime[1]) since January 1st 1970 (GMT).

zp_gen: 64 bit generation number, contains the transaction group number of the creation of this file.

zp_mode: 64 bit integer containing file mode bits and file type. The lower 8 bits of the mode contain the access mode bits, for example 755. The 9th bit is the sticky bit and can be a value of zero or one. Bits 13-16 are used to designate the file type. The file types can be seen in the table below.

Table 6.2: File types and their associated mode bits

Type	Description	Value in bits 13-16
S_IFIFO	Fifo	0x1
S_IFCHR	Character Special Device	0x2
S_IFDIR	Directory	0x4
S_IFBLK	Block special device	0x6
S_IFREG	Regular file	0x8
S_IFLNK	Symbolic Link	0xA
S_IFSOCK	Socket	0xC
S_IFDOOR	Door	0xD
S_IFPORT	Event Port	0xE

zp_size: size of file in bytes

zp_parent: object id of the parent directory containing this file

zp_links: number of hard links to this file

zp_xattr: object ID of a ZAP object which is the hidden attribute directory. It is treated like a normal directory in ZFS, except that its hidden and an application will need to "tunnel" into the file via openat() to get to it.

zp_rdev: dev_t for files of type S_IFCHR or S_IFBLK

zp_flags: Persistent flags set on the file. The following are valid flag values.

Table 6.3: zp_flag values

Flag	Value
ZFS_XATTR	0x1
ZFS_INHERIT_ACE	0x2

zp_uid: 64 bit integer (uid_t) of the files owner.

zp_gid: 64 bit integer (gid_t) owning group of the file.

zp_acl: zfs_znode_acl structure containing any ACL entries set on this file. The zfs_acl_phys structure is defined below.

6.3 ZFS Access Control Lists

Access control lists (*ACL*) serve as a mechanism to allow or restrict user access privileges on a ZFS object. ACLs are implemented in ZFS as a table containing ACEs (Access Control Entries).

The znode_phys_t contains a zfs_acl_phys structure. This structure is shown below.

```
#define ACE_SLOT_CNT 6

typedef struct zfs_acl_phys_v0 {
        uint64_t  z_acl_extern_obj;
        uint32_t  z_acl_count;
        uint16_t  z_acl_version;
        uint16_t  z_acl_pad;
        zfs_oldace_t  z_ace_data[ACE_SLOT_CNT];
} zfs_acl_phys_v0_t;
```

z_acl_extern_obj: Used for holding ACLs that won't fit in the znode. In other words, its for ACLs great than 6 ACEs. The object type of an extern ACL is DMU_OT_ACL.

z_acl_count: number of ACE entries that make up an ACL.

z_acl_version: reserved for future use.

z_acl_pad: reserved for future use.

z_ace_data: Array of up to 6 ACEs.

An ACE specifies an access right to an individual user or group for a specific object.

```
typedef struct zfs_oldace {
        uint32_t  z_fuid;
        uint32_t  z_access_mask;
        uint16_t  z_flags;
        uint16_t  z_type;
} zfs_oldace_t;
```

z_fuid: This field is only meaningful when the `ACE_OWNER`, `ACE_GROUP` and `ACE_EVERYONE` flags (set in `z_flags`, described below) are not asserted. The `z_fuid` field contains a UID or GID. If the `ACE_IDENTIFIER_GROUP` flag is set in `z_flags` (see below), the `z_fuid` field will contain a GID. Otherwise, this field will contain a UID.

a_access_mask : 32 bit access mask. The table below shows the access attribute associated with each bit.

Table 6.4: Access Mask Values

Attribute	Value
ACE_READ_DATA	0x00000001
ACE_LIST_DIRECTORY	0x00000001
ACE_WRITE_DATA	0x00000002
ACE_ADD_FILE	0x00000002
ACE_APPEND_DATA	0x00000004
ACE_ADD_SUBDIRECTORY	0x00000004
ACE_READ_NAMED_ATTRS	0x00000008
ACE_WRITE_NAMED_ATTRS	0x00000010
ACE_EXECUTE	0x00000020
ACE_DELETE_CHILD	0x00000040
ACE_READ_ATTRIBUTES	0x00000080
ACE_WRITE_ATTRIBUTES	0x00000100
ACE_DELETE	0x00010000
ACE_READ_ACL	0x00020000

Continued on next page

Table 6.4 – continued from previous page

Attribute	Value
ACE_WRITE_ACL	0x00040000
ACE_WRITE_OWNER	0x00080000
ACE_SYNCHRONIZE	0x00100000

z_flags: 16 bit integer whose value describes the ACL entry type and inheritance flags.

Table 6.5: Entry Type and Inheritance Flag Values

ACE flag	Value
ACE_FILE_INHERIT_ACE	0x0001
ACE_DIRECTORY_INHERIT_ACE	0x0002
ACE_NO_PROPAGATE_INHERIT_ACE	0x0004
ACE_INHERIT_ONLY_ACE	0x0008
ACE_SUCCESSFUL_ACCESS_ACE_FLAG	0x0010
ACE_FAILED_ACCESS_ACE_FLAG	0x0020
ACE_IDENTIFIER_GROUP	0x0040
ACE_OWNER	0x1000
ACE_GROUP	0x2000
ACE_EVERYONE	0x4000

z_type: The type of this ace. The following types are listed in the table below.

Table 6.6: ACE Types and Values

Type	Value	Description
ACE_ACCESS_ALLOWED_ACE_TYPE	0x0000	Grants access as described in `z_access_mask`.

Continued on next page

Table 6.6 – continued from previous page

Type	Value	Description
ACE_ACCESS_DENIED_ACE_TYPE	0x0001	Denies access as described in `z_access_mask`.
ACE_SYSTEM_AUDIT_ACE_TYPE	0x0002	Audit the successful or failed accesses (depending on the presence of the successful/failed access flags) as defined in the `z_access_mask`. [1]
ACE_SYSTEM_ALARM_ACE_TYPE	0x0003	Alarm the successful of failed accesses as defined in the `z_access_mask` [2]

Chapter 7

ZFS Intent Log

The *ZFS intent log* (*ZIL*) saves transaction records of system calls that change the file system in memory with enough information to be able to replay them. These are stored in memory until either the DMU transaction group (txg) commits them to the stable pool and they can be discarded, or they are flushed to the stable log (also in the pool) due to a fsync, `O_DSYNC` or other synchronous requirement. In the event of a panic or power failure, the log records (transactions) are replayed.

There is one ZIL per file system. Its on-disk (pool) format consists of 3 parts:

- ZIL header
- ZIL blocks
- ZIL records

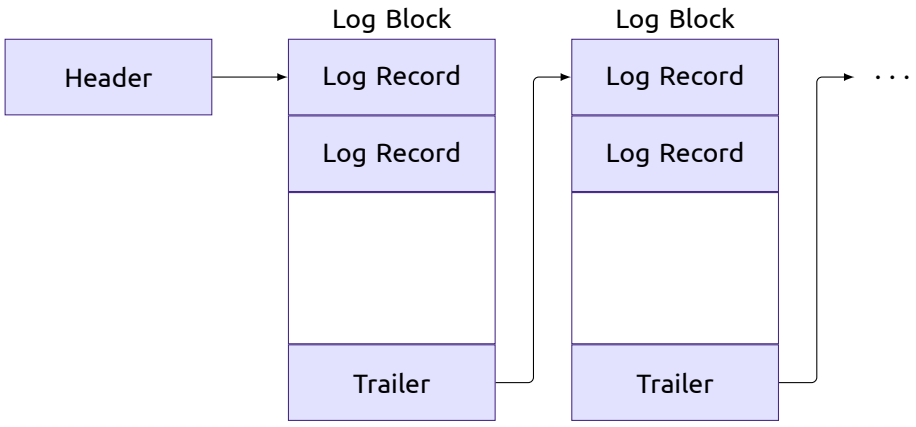

Illustration 7.1: Overview of ZIL Structure

A log record holds a system call transaction. Log blocks can hold many log records and the blocks are chained together. Each ZIL block contains a block pointer in the trailer(`blkptr_t`) to the next ZIL block in the chain. Log blocks can be different

sizes. The ZIL header points to the first block in the chain. Note there is not a fixed place in the pool to hold blocks. They are dynamically allocated and freed as needed from the blocks available. The illustration above shows the ZIL structure showing log blocks and log records of different sizes:

More details of the current ZIL on disk structures are given below.

7.1 ZIL Header

There is one of these per ZIL and it has a simple structure:

```
typedef struct zil_header {
        uint64_t zh_claim_txg;
        uint64_t zh_replay_seq;
        blkptr_t zh_log;
        uint64_t zh_claim_blk_seq;
        uint64_t zh_flags;
        uint64_t zh_claim_lr_seq;
        uint64_t zh_pad[3];
} zil_header_t;
```

7.2 ZIL Blocks

ZIL blocks contain ZIL records. The blocks are allocated on demand and are of a variable size according to need. The size field is part of the `blkptr_t` which points to a log block. Each block is filled with records and contains a `zil_chain_t` at the end of the block:

ZIL Trailer

```
typedef struct zil_chain {
        uint64_t zc_pad;
        blkptr_t zc_next_blk;
        uint64_t zc_nused;
        zio_eck_t zc_eck;
} zil_chain_t;
```

ZIL Records

ZIL Record Common Structure

ZIL records all start with a common section followed by a record (transac-

tion) specific structure. The common log record structure and record types (values for `lrc_txtype`) are:

```
typedef struct {
        uint64_t  lrc_txtype;
        uint64_t  lrc_reclen;
        uint64_t  lrc_txg;
        uint64_t  lrc_seq;
} lr_t;
#define TX_COMMIT   0
#define TX_CREATE   1
#define TX_MKDIR    2
#define TX_MKXATTR    3
#define TX_SYMLINK    4
#define TX_REMOVE   5
#define TX_RMDIR    6
#define TX_LINK     7
#define TX_RENAME   8
#define TX_WRITE    9
#define TX_TRUNCATE   10
#define TX_SETATTR    11
#define TX_ACL_V0   12
#define TX_ACL      13
#define TX_CREATE_ACL   14
#define TX_CREATE_ATTR    15
#define TX_CREATE_ACL_ATTR  16
#define TX_MKDIR_ACL    17
#define TX_MKDIR_ATTR   18
#define TX_MKDIR_ACL_ATTR 19
#define TX_WRITE2   20
#define TX_MAX_TYPE   21
```

ZIL Record Specific Strucgures

For each of the record (transaction) types listed above there is a specific structure which embeds the common structure. Within each record enough information is saved in order to be able to replay the transaction (usually one VOP call). The VOP layer will pass in-memory pointers to vnodes. These have to be converted to stable pool object identifiers (oids). When replaying the transaction the VOP layer is called again. To do this we reopen the object and pass it's vnode. Some of the record specific structures are used for more than one transaction type. The `lr_create_t` record specific structure is used for: `TX_CREATE`, `TX_MKDIR`, `TX_MKXATTR` and `TX_SYMLINK`, and `lr_remove_t` is used for both

TX_REMOVE and TX_RMDIR. All fields (other than strings and user data) are 64 bits wide. This provides for a well defined alignment which allows for easy compatibility between different architectures, and easy endianness conversion if necessary. Here's the definition of the record specific structures:

```
typedef struct {
        lr_t  lr_common;
        uint64_t  lr_doid;
        uint64_t  lr_foid;
        uint64_t  lr_mode;
        uint64_t  lr_uid;
        uint64_t  lr_gid;
        uint64_t  lr_gen;
        uint64_t  lr_crtime[2];
        uint64_t  lr_rdev;
} lr_create_t;
typedef struct {
        lr_create_t lr_create;
        uint64_t  lr_aclcnt;
        uint64_t  lr_domcnt;
        uint64_t  lr_fuidcnt;
        uint64_t  lr_acl_bytes;
        uint64_t  lr_acl_flags;
} lr_acl_create_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_doid;
} lr_remove_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_doid;
        uint64_t  lr_link_obj;
} lr_link_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_sdoid;
        uint64_t  lr_tdoid;
} lr_rename_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_foid;
        uint64_t  lr_offset;
```

```
        uint64_t  lr_length;
        uint64_t  lr_blkoff;
        blkptr_t  lr_blkptr;
} lr_write_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_foid;
        uint64_t  lr_offset;
        uint64_t  lr_length;
} lr_truncate_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_foid;
        uint64_t  lr_mask;
        uint64_t  lr_mode;
        uint64_t  lr_uid;
        uint64_t  lr_gid;
        uint64_t  lr_size;
        uint64_t  lr_atime[2];
        uint64_t  lr_mtime[2];
} lr_setattr_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_foid;
        uint64_t  lr_aclcnt;
} lr_acl_v0_t;
typedef struct {
        lr_t    lr_common;
        uint64_t  lr_foid;
        uint64_t  lr_aclcnt;
        uint64_t  lr_domcnt;
        uint64_t  lr_fuidcnt;
        uint64_t  lr_acl_bytes;
        uint64_t  lr_acl_flags;
} lr_acl_t;
```

Chapter 8

ZFS Volume

ZFS Volumes, or *ZVOL* provides a mechanism for creating logical volumes. ZFS volumes are exported as block devices and can be used like any other block device. ZVOLs are represented in ZFS as an object set of type DMU_OST_ZVOL (see Table 8.1). A ZVOL object set has a very simple format consisting of two objects: a properties object and a data object, object type DMU_OT_ZVOL_PROP and DMU_OT_ZVOL respectively. Both objects have statically assigned object Ids. Each object is described below.

Table 8.1: ZVOL Object Types

Object	Type	Number	Description
Properties	DMU_OT_ZVOL_PROP	2	The ZVOL property object is a ZAP object containing attributes associated with this volume. A particular attribute of interest is the "volsize" attribute. This attribute contains the size, in bytes, of the volume.
Data	DMU_OT_ZVOL	1	This object stores the contents of this virtual block device.