openzfs Documentation

Release latest

Mar 02, 2020

OpenZFS

This guide is intended to replace the various bits of ZFS documentation found online with something more comprehensive for OpenZFS. The closest thing I've been able to find for a comprehensive guide to ZFS is targeted specifically at Solaris ZFS, which always comes with caveats of not being quite accurate for OpenZFS.

My experience is definitely with ZFS on Linux, and that's where the first draft of this guide will focus; but contributions specific to BSD, illumos, or other OpenZFS distributions are welcome and encouraged. (Where behavior or recommendations differ between platforms, these differences should be called out with no preference for any specific platform.)

CHAPTER 1

Introduction to ZFS

1.1 History

ZFS was originally developed by Sun Microsystems for the Solaris operating system. The source code for ZFS was released under the CDDL as part of the OpenSolaris operating system and was subsequently ported to other platforms.

OpenSolaris was discontinued in 2010, and it is from this last release that illumos was forked. Official announcement of The OpenZFS project was announced in 2013 to serve as an upstream for illumos, BSD, Linux, and other ports of ZFS. Further development of ZFS on Solaris is not open source.

OpenZFS is the truly open source successor to the ZFS project. Development thrives as a part of illumos, which has added many features and performance improvements. New OpenZFS features and fixes are regularly pulled in from illumos, to all ports to other platforms, and vice versa.

1.2 Architecture

ZFS itself is composed of three principle layers. The bottom layer is the Storage Pool Allocator, which handles organizing the physical disks into storage. The middle layer is the Data Management Unit, which uses the storage provided by the SPA by reading and writing to it transactionally in an atomic manner. The top layer is the dataset layer, which translates between operations on the filesystems and block devices (zvols) provided by the pool into operations in the DMU.

1.3 Storage pools

The basic unit of storage in ZFS is the pool and from it, we obtain datasets that can be either mountpoints (a mountable filesystem) or block devices. The ZFS pool is a full storage stack capable of replacing RAID, partitioning, volume management, fstab/exports files and traditional single-disk file systems such as UFS and XFS. This allows the same tasks to be accomplished with less code, greater reliability and simplified administration.

The creation of a usable filesystem with redundancy from a set of disks can be accomplished with 1 command and this will be persistent upon reboots. This is because a ZFS pool will always have a mountable filesystem called the root dataset, which is mounted at pool creation. At creation, a pool is imported into the system, such that an entry in the `zpool.cache` file is created. At time of import or creation, the pool stores the system's unique hostid and for the purposes of supporting multipath, import into other systems will fail unless forced.

Administration of ZFS is performed through the zpool and zfs commands. To create a pool, you can use `zpool create poolname ...`. The part following the pool name is the vdev tree specification. The root dataset will be mounted at `/poolname` by default unless `zpool create -m none` is used. Other mountpoint locations can be specified by writing them in place of `none`.

Any file or disk vdevs before a top level vdev keyword is specified will be a top level vdev. Any after a top level vdev keyword will be a child of that vdev. Non-equal sized disks or files inside a top level vdev will restrict its storage to the smaller of the two.

Additional information can be found in the zpool man page on your platform. This is section 1m on Illumos and section 8 on Linux.

1.4 Virtual devices (vdevs)

The organization of disks in a pool by the SPA is a tree of vdevs or virtual devices. At the top level of the tree is the root vdev. Its immediate children can be any vdev type other than itself. The main types of vdevs are:

- mirror (n-way mirrors supported)
- raidz - raidz1 (1-disk parity, similar to RAID 5) - raidz2 (2-disk parity, similar to RAID 6) - raidz3 (3-disk parity, no RAID analog)
- disk
- file (not recommended for production due to another filesystem adding unnecessary layering)

Any number of these can be children of the root vdev, which are called top-level vdevs. Furthermore, some of these may also have children, such as mirror vdevs and raidz vdevs. The commandline tools do not support making mirrors of raidz or raidz of mirrors, although such configurations are used in developer testing.

The use of multiple top level vdevs will affect IOPS in an additive manner where total IOPS will be the sum of the top level vdevs. Consequently, the loss of any main top level vdev will result in the loss of the entire pool, such that proper redundancy must be used on all top level vdevs.

The smallest supported file vdev or disk vdev size is 64MB (2^16 bytes) while the largest depends on the platform, but all platforms should support vdevs of at least 16EB (2^64 bytes).

There are also three special device types.

- spare
- cache
- log

The spare devices are used for replacement when a drive fails, provided that the pool's autoreplace property is enabled and your platform supports that functionality. It will not replace a cache device or log device.

The cache devices are used for extending ZFS's in-memory data cache, which replaces the page cache with the exception of mmap(), which still uses the page cache on most platforms. The algorithm used by ZFS is the Adaptive Replacement Cache algorithm, which has a higher hit rate than the Last Recently Used algorithm used by the page cache. The cache devices are intended to be used with flash devices. The data stored in them is non-persistent, so cheap devices can be used.

The log devices allow ZFS Intent Log records to be written to different devices, such as flash devices, to increase performance of synchronous write operations, before they are written to main storage. These records are used unless the sync=disabled dataset property is set. In either case, the synchronous operations' changes are held in memory until they are written out on the next transaction group commit. ZFS can survive the loss of a log device as long as the next transaction group commit finishes successfully. If the system crashes at the time that the log default is lost, the pool will be faulted. While it can be recovered, whatever synchronous changes made in the current transaction group will be lost to the datasets stored on it.

1.5 Datasets

file system

volume

snapshot

bookmark

1.6 Data integrity

ZFS has multiple mechanisms by which it attempts to provide data integrity:

- Committed data is stored in a merkle tree that is updated atomically on each transaction group commit
- The merkle tree uses 256-bit checksums stored in the block pointers to protect against misdirected writes, including those that would be likely to collide for weaker checksums. sha256 is a supported checksum for cryptographically strong guarantees, although the default is fletcher4.
- Each disk/file vdev contains four disk labels (two on each end) so that the loss of data at either end from a head drop does not wipe the labels.
- The transaction group commit uses two stages to ensure that all data is written to storage before the transaction group is considered committed. This is why ZFS has two labels on each end of each disk. A full head sweep is required on mechanical storage to perform the transaction group commit and flushes are used to ensure that the latter half does not occur before anything else.
- ZIL records storing changes to be made for synchronous IO are self checksumming blocks that are read only on pool import if the system made changes before the last transaction group commit was made.
- All metadata is stored twice by default, with the object containing the pool's state at a given transaction group to which the labels point being written three times. An effort is made to store the metadata at least 1/8 of a disk apart so that head drops do not result in irrecoverable damage.
- The labels contain an uberblock history, which allows rollback of the entire pool to a point in the near past in the event of a worst case scenario. The use of this recovery mechanism requires special commands because it should not be needed.
- The uberblocks contain a sum of all vdev GUIDs. Uberblocks are only considered valid if the sum matches. This prevents uberblocks from destroyed old pools from being be mistaken as being valid uberblocks.
- N-way mirroring and up to 3 levels of parity on raidz are supported so that increasingly common 2-disk failures[1] that kill RAID 5 and double mirrors during recovery do not kill a ZFS pool when proper redundancy is used.

Misinformation has been circulated that ZFS data integrity features are somehow worse than those of other filesystems when ECC RAM is not used. This is not the case: all software needs ECC RAM for reliable operation and ZFS is no different from any other filesystem in that regard.

1.7 Example

A simple example here creates a new single-device pool "tank" using the single disk `sda` and mounts this pool at `/mnt/tank`.

```
zpool create -o mountpoint=/mnt/tank tank sda
zfs mount tank
```

CHAPTER 2

ZFS storage pools

2.1 The zpool command

2.2 Virtual devices

disk

file

mirror

raidz, raidz1, raidz2, raidz3

spare

log

dedup

special

cache

```
zpool create tank mirror sda sdb mirror sdc sdd
```

2.2.1 Device sanity checks

2.3 Creating storage pools

2.4 Adding devices to an existing pool

2.4.1 Display options

2.5 Attaching a mirror device

2.6 Importing and exporting

2.7 Pool properties

allocated *read-only*

altroot *set at creation time and import time only*

ashift=ashift

autoexpand=on|off

autoreplace=on|off

autotrim=on|off

bootfs=(unset)|pool/dataset

cachefile=path|none

capacity *read-only*

comment=text

dedupditto=number *deprecated.*

delegation=on|off

expandsize *read-only*

feature@feature_name=enabled

fragmentation *read-only*

free *read-only*

freeing *read-only*

health *read-only*

guid *read-only*

listsnapshots=on|off

load_guid *read-only*

multihost=on|off

readonly=on|off *set only at import time*

size *read-only*

unsupported@feature_guid *read-only*

version=version

2.8 Device failure and recovery

DEGRADED

FAULTED

OFFLINE

ONLINE

REMOVED

UNAVAIL

2.8.1 Hot spares

```
zpool create tank mirror sda sdb spare sdc sdd
```

2.8.2 Clearing errors

```
zpool clear pool [device]
```

2.9 Scrubs

2.10 The intent log

```
zpool create tank sda sdb log sdc
```

2.11 Cache devices

```
zpool create tank sda sdb cache sdc sdd
```

2.12 Checkpoints

```
zpool checkpoint [-d, --discard] pool
```

```
zpool checkpoint pool
```

```
zpool export pool
zpool import --rewind-to-checkpoint pool
```

```
zpool checkpoint --discard pool
```

2.13 The special allocation class

2.14 Pool features

CHAPTER 3

Redundancy

Two-disk failures are common enough that raidz1 vdevs should not be used for data storage in production. 2-disk mirrors are also at risk, but not quite as much as raidz1 vdevs unless only two disks are used. This is because the failure of all disks in a 2-disk mirror is statistically less likely than the failure of only 2 disks in a parity configuration such as raidz1, unless only 2 disks are used, where the failure risk is the same. The same risks also apply to RAID 1 and RAID 5.

Single disk vdevs (excluding log and cache devices) should never be used in production because they are not redundant. However, pools containing single disk vdevs can be made redundant by attaching disks via `zpool attach` to convert the disks into mirrors. For production pools, it is important not to create top level vdevs that are not raidz, mirror, log, cache or spare vdevs.

CHAPTER 4

The ZFS on-disk format

The last numbered version of the ZFS on-disk format is v28. Use of this version in OpenZFS guarantees compatibility between Solaris ZFS and OpenZFS. As Oracle's code is no longer open source, OpenZFS is not compatible with Solaris ZFS pools beyond v28.

4.1 Features

Version numbering was unsuitable for the distributed development of OpenZFS: any change to the number would have required agreement, across all implementations, of each change to the on-disk format.

OpenZFS feature flags–-an alternative to traditional version numbering-–allow a uniquely named pool property for each change to the on-disk format. This approach supports both independent and inter-dependent format changes.

Feature flags are implemented by artificially rolling the pool version to v5000 to avoid any conflicts with Oracle's version.

The zgrep.org project tracks and documents the availability of each known OpenZFS feature by parsing each project's man pages.

CHAPTER 5

Performance tuning

5.1 Basic concepts

5.1.1 Adaptive Replacement Cache

Operating systems traditionally use RAM as a cache to avoid the necessity of waiting on disk IO, which is comparatively extremely slow. This concept is called page replacement. Until ZFS, virtually all filesystems used the Least Recently Used (LRU) page replacement algorithm in which the least recently used pages are the first to be replaced. Unfortunately, the LRU algorithm is vulnerable to cache flushes, where a brief change in workload that occurs occasionally removes all frequently used data from cache. The Adaptive Replacement Cache (ARC) algorithm was implemented in ZFS to replace LRU. It solves this problem by maintaining four lists:

- A list for recently cached entries.
- A list for recently cached entries that have been accessed more than once.
- A list for entries evicted from #1.
- A list of entries evicited from #2.

Data is evicted from the first list while an effort is made to keep data in the second list. In this way, ARC is able to outperform LRU by providing a superior hit rate.

In addition, a dedicated cache device (typically an SSD) can be added to the pool, with `zpool add poolname cache devicename`. The cache device is managed by the Level 2 ARC (L2ARC) which scans entries that are next to be evicted and writes them to the cache device. The data stored in ARC and L2ARC can be controlled via the `primarycache` and `secondarycache` ZFS properties respectively, which can be set on both zvols and datasets. Possible settings are `all`, `none`, and `metadata`. It is possible to improve performance when a zvol or dataset hosts an application that does its own caching by caching only metadata. One example is PostgreSQL. Another would be a virtual machine using ZFS.

5.1.2 Alignment shift (ashift)

Top-level vdevs contain an internal property called ashift, which stands for alignment shift. It is set at vdev creation and is otherwise immutable. It can be read using the `zdb` command. It is calculated as the maximum base 2 logarithm of the physical sector size of any child vdev and it alters the disk format such that writes are always done according to it. This makes 2^ashift the smallest possible IO on a vdev. Configuring ashift correctly is important because partial sector writes incur a penalty where the sector must be read into a buffer before it can be written.

ZFS will attempt to ensure proper alignment by extracting the physical sector sizes from the disks. ZFS makes the implicit assumption that the sector size reported by drives is correct and calculates ashift based on that. The largest sector size will be used per top-level vdev to avoid partial sector modification overhead is eliminated. This will not be correct when drives misreport their physical sector sizes.

In an ideal world, physical sector size is always reported correctly and therefore, this requires no attention. Unfortunately, this is not the case. The sector size on all storage devices was 512-bytes prior to the creation of flash-based solid state drives. Some operating systems, such as Windows XP, were written under this assumption and will not function when drives report a different sector size.

Flash-based solid state drives came to market around 2007. These devices report 512-byte sectors, but the actual flash pages, which roughly correspond to sectors, are never 512-bytes. The early models used 4096-byte pages while the newer models have moved to an 8192-byte page. In addition, "Advanced Format" hard drives have been created which also use a 4096-byte sector size. Partial page writes suffer from similar performance degradation as partial sector writes. In some cases, the design of NAND-flash makes the performance degradation even worse, but that is beyond the scope of this description.

Reporting the correct sector sizes is the responsibility the block device layer. This unfortunately has made proper handling of devices that misreport drives different across different platforms. The respective methods are as follows:

- `sd.conf` on illumos
- `gnop` on FreeBSD
- `-o ashift=` on Linux and potentially other OpenZFS implementations

"-o ashift=' is convenient, but it is flawed in that the creation of pools containing top level vdevs that have multiple optimal sector sizes require the use of multiple commands. A newer syntax that will rely on the actual sector sizes has been discussed as a cross platform replacement and will likely be implemented in the future.

In addition, a database of drives known to misreport sector sizes is used to automatically adjust ashift without the assistance of the system administrator. This approach is unable to fully compensate for misreported sector sizes whenever drive identifiers are used ambiguously (e.g. virtual machines, iSCSI LUNs, some rare SSDs), but it does a great amount of good. The format is roughly compatible with illumos' `sd.conf`, and it is expected that other implementations will integrate the database in future releases. Strictly speaking, this database does not belong in ZFS, but the difficulty of patching the Linux kernel (especially older ones) necessitated that this be implemented in ZFS itself for Linux. The same is true for MacZFS. However, FreeBSD and illumos are both able to implement this in the correct layer.

5.1.3 Compression

Internally, ZFS allocates data using multiples of the device's sector size, typically either 512 bytes or 4KB (see above). When compression is enabled, a smaller number of sectors can be allocated for each block. The uncompressed block size is set by the `recordsize` file system property (defaults to 128KB) or the `volblocksize` volume property (defaults to 8KB).

The following compression algorithms are available:

LZ4 New algorithm added after feature flags were created. It is significantly superior to LZJB in all metrics tested. It is new default compression algorithm (compression=on) in OpenZFS[1], but not all platforms have adopted the commit changing it yet.

LZJB Original default compression algorithm (compression=on) for ZFS. It was created to satisfy the desire for a compression algorithm suitable for use in filesystems. Specifically, that it provides fair compression, has a high compression speed, has a high decompression speed and detects incompressible data detection quickly.

GZIP (1 through 9) Classic Lempel-Ziv implementation. It provides high compression, but it often makes IO CPU-bound.

ZLE (Zero Length Encoding) A very simple algorithm that only compresses zeroes.

If you want to use compression and are uncertain which to use, use LZ4. It averages a 2.1:1 compression ratio while gzip-1 averages 2.7:1, but gzip is much slower. Both figures are obtained from testing by the LZ4 project on the Silesia corpus. The greater compression ratio of gzip is usually only worthwhile for rarely accessed data.

5.1.4 RAID-Z stripe width

Choose a RAID-Z stripe width based on your IOPS needs and the amount of space you are willing to devote to parity information. If you need more IOPS, use fewer disks per stripe. If you need more usable space, use more disks per stripe. Trying to optimize your RAID-Z stripe width based on exact numbers is irrelevant in nearly all cases.

5.1.5 Dataset recordsize

ZFS datasets use a default internal recordsize of 128KB. The dataset recordsize is the basic unit of data used for internal copy-on-write on files. Partial record writes require that data be read from either ARC (cheap) or disk (expensive). recordsize can be set to any power of 2 from 512 bytes to 128 kilobytes. Software that writes in fixed record sizes (e.g. databases) will benefit from the use of a matching recordsize.

Changing the recordsize on a dataset will only take effect for new files. If you change the recordsize because your application should perform better with a different one, you will need to recreate its files. A `cp` followed by a `mv` on each file is sufficient. Alternatively, `send` / `recv` should recreate the files with the correct recordsize when a full receive is done.

5.1.6 zvol volblocksize

Zvols have a volblocksize property that is analogous to record size. The default size is 8KB, which is the size of a page on the SPARC architecture. Workloads that use smaller sized IOs (such as swap on x86 which use 4096-byte pages) will benefit from a smaller volblocksize.

5.1.7 Deduplication

Deduplication uses an on-disk hash table, using extensible hashing as implemented in the ZAP (ZFS Attribute Processor). Each cached entry uses slightly more than 320 bytes of memory. The DDT code relies on ARC for caching the DDT entries, such that there is no double caching or internal fragmentation from the kernel memory allocator. Each pool has a global deduplication table shared across all datasets and zvols on which deduplication is enabled. Each entry in the hash table is a record of a unique block in the pool. (Where the block size is set by the `recordsize` or `volblocksize` properties.)

The hash table (also known as the deduplication table, or DDT) must be accessed for every dedup-able block that is written or freed (regardless of whether it has multiple references). If there is insufficient memory for the DDT to be cached in memory, each cache miss will require reading a random block from disk, resulting in poor performance. For example, if operating on a single 7200RPM drive that can do 100 io/s, uncached DDT reads would limit overall write throughput to 100 blocks per second, or 400KB/s with 4KB blocks.

The consequence is that sufficient memory to store deduplication data is required for good performance. The deduplication data is considered metadata and therefore can be cached if the `primarycache` or `secondarycache`

properties are set to `metadata`. In addition, the deduplication table will compete with other metadata for metadata storage, which can have a negative effect on performance. Simulation of the number of deduplication table entries needed for a given pool can be done using the -D option to zdb. Then a simple multiplication by 320-bytes can be done to get the approximate memory requirements. Alternatively, you can estimate an upper bound on the number of unique blocks by dividing the amount of storage you plan to use on each dataset (taking into account that partial records each count as a full recordsize for the purposes of deduplication) by the recordsize and each zvol by the volblocksize, summing and then multiplying by 320-bytes.

5.1.8 Metaslab allocator

ZFS top level vdevs are divided into metaslabs from which blocks can be independently allocated so allow for concurrent IOs to perform allocations without blocking one another.

By default, the selection of a metaslab is biased toward lower LBAs to improve performance of spinning disks, but this does not make sense on solid state media. This behavior can be adjusted globally by setting the ZFS module's global `metaslab_lba_weighting_enabled` tuanble to `0`. This tunable is only advisable on systems that only use solid state media for pools.

The metaslab allocator will allocate blocks on a first-fit basis when a metaslab has more than or equal to 4 percent free space and a best-fit basis when a metaslab has less than 4 percent free space. The former is much faster than the latter, but it is not possible to tell when this behavior occurs from the pool's free space. However, the command `zdb -mmm $POOLNAME` will provide this information.

5.1.9 Pool geometry

If small random IOPS are of primary importance, mirrored vdevs will outperform raidz vdevs. Read IOPS on mirrors will scale with the number of drives in each mirror while raidz vdevs will each be limited to the IOPS of the slowest drive.

If sequential writes are of primary importance, raidz will outperform mirrored vdevs. Sequential write throughput increases linearly with the number of data disks in raidz while writes are limited to the slowest drive in mirrored vdevs. Sequential read performance should be roughly the same on each.

Both IOPS and throughput will increase by the respective sums of the IOPS and throughput of each top level vdev, regardless of whether they are raidz or mirrors.

5.1.10 Whole disks vs partitions

ZFS will behave differently on different platforms when given a whole disk.

ZFS will also attempt minor tweaks on various platforms when whole disks are provided. On Illumos, ZFS will enable the disk cache for performance. It will not do this when given partitions to protect other filesystems sharing the disks that might not be tolerant of the disk cache, such as UFS. On Linux, the IO elevator will be set to noop to reduce CPU overhead. ZFS has its own internal IO elevator, which renders the Linux elevator redundant. The Performance Tuning page explains this behavior in more detail.

On illumos, ZFS attempts to enable the write cache on a whole disk. The illumos UFS driver cannot ensure integrity with the write cache enabled, so by default Sun/Solaris systems using UFS file system for boot were shipped with drive write cache disabled (long ago, when Sun was still an independent company). For safety on illumos, if ZFS is not given the whole disk, it could be shared with UFS and thus it is not appropriate for ZFS to enable write cache. In this case, the write cache setting is not changed and will remain as-is. Today, most vendors ship drives with write cache enabled by default.

On Linux, the Linux IO elevator is largely redundant given that ZFS has its own IO elevator, so ZFS sets the IO elevator to noop to avoid unnecessary CPU overhead.

ZFS also creates a GPT partition table own partitions when given a whole disk under illumos on x86/amd64 and on Linux. This is mainly to make booting through UEFI possible because UEFI requires a small FAT partition to be able to boot the system. The ZFS driver will be able to tell the difference between whether the pool had been given the entire disk or not via the whole_disk field in the label.

This is not done on FreeBSD. Pools created by FreeBSD will always have the whole_disk field set to true, such that a pool imported on another platform that was created on FreeBSD will always be treated as the whole disks were given to ZFS.

5.2 General recommendations

5.2.1 Alignment shift

Make sure that you create your pools such that the vdevs have the correct alignment shift for your storage device's size. if dealing with flash media, this is going to be either 12 (4K sectors) or 13 (8K sectors). For SSD ephemeral storage on Amazon EC2, the proper setting is 12.

5.2.2 Free space

Keep pool free space above 10% to avoid many metaslabs from reaching the 4% free space threshold to switch from first-fit to best-fit allocation strategies. When the threshold is hit, the metaslab allocator becomes very CPU intensive in an attempt to protect itself from fragmentation. This reduces IOPS, especially as more metaslabs reach the 4% threshold.

The recommendation is 10% rather than 5% because metaslabs selection considers both location and free space unless the global `metaslab_lba_weighting_enabled` tunable is set to `0`. When that tunable is 0, ZFS will consider only free space, so the the expense of the best-fit allocator can be avoided by keeping free space above 5%. That setting should only be used on systems with pools that consist of solid state drives because it will reduce sequential IO performance on mechanical disks.

5.2.3 LZ4 compression

Set `compression=lz4` on your pools' root datasets so that all datasets inherit it unless you have a reason not to enable it. Userland tests of LZ4 compression of incompressible data in a single thread has shown that it can process 10GB/sec, so it is unlikely to be a bottleneck even on incompressible data. The reduction in IO from LZ4 will typically be a performance win.

5.2.4 Pool geometry

Do not put more than ~16 disks in raidz. The rebuild times on mechanical disks will be excessive when the pool is full.

5.2.5 Synchronous I/O

If your workload involves fsync or O_SYNC and your pool is backed by mechanical storage, consider adding one or more SLOG devices. Pools that have multiple SLOG devices will distribute ZIL operations across them.

To ensure maximum ZIL performance on NAND flash SSD-based SLOG devices, you should also overprovison spare area to increase IOPS. You can do this with a mix of a secure erase and a partition table trick, such as the following:

1. Run a secure erase on the NAND-flash SSD.

2. Create a partition table on the NAND-flash SSD.
3. Create a 4GB partition.
4. Give the partition to ZFS to use as a log device.

If using the secure erase and partition table trick, do not use the unpartitioned space for other things, even temporarily. That will reduce or eliminate the overprovisioning by marking pages as dirty.

Alternatively, some devices allow you to change the sizes that they report.This would also work, although a secure erase should be done prior to changing the reported size to ensure that the SSD recognizes the additional spare area. Changing the reported size can be done on drives that support it with `hdparm -N <sectors>` on systems that have laptop-mode-tools.

The choice of 4GB is somewhat arbitrary. Most systems do not write anything close to 4GB to ZIL between transaction group commits, so overprovisioning all storage beyond the 4GB partition should be alright. If a workload needs more, then make it no more than the maximum ARC size. Even under extreme workloads, ZFS will not benefit from more SLOG storage than the maximum ARC size. That is half of system memory on Linux and 3/4 of system memory on illumos.

5.2.6 Whole disks

Whole disks should be given to ZFS rather than partitions. If you must use a partition, make certain that the partition is properly aligned to avoid read-modify-write overhead. See the section on Alignment Shift for a description of proper alignment. Also, see the section on Whole Disks versus Partitions for a description of changes in ZFS behavior when operating on a partition.

Single disk RAID 0 arrays from RAID controllers are not equivalent to whole disks.

5.3 Bit Torrent

Bit torrent performs 16KB random reads/writes. The 16KB writes cause read-modify-write overhead. The read-modify-write overhead can reduce performance by a factor of 16 with 128KB record sizes when the amount of data written exceeds system memory. This can be avoided by using a dedicated dataset for bit torrent downloads with recordsize=16KB.

When the files are read sequentially through a HTTP server, the random nature in which the files were generated creates fragmentation that has been observed to reduce sequential read performance by a factor of two on 7200RPM hard disks. If performance is a problem, fragmentation can be eliminated by rewriting the files sequentially in either of two ways:

The first method is to configure your client to download the files to a temporary directory and then copy them into their final location when the downloads are finished, provided that your client supports this.

The second method is to use send/recv to recreate a dataset sequentially.

In practice, defragmenting files obtained through bit torrent should only improve performance when the files are stored on magnetic storage and are subject to significant sequential read workloads after creation.

5.4 InnoDB (MySQL)

Make separate datasets for InnoDB's data files and log files. Set recordsize=16K on InnoDB's data files to avoid expensive partial record writes and leave `recordsize=128K` on the log files. Set primarycache=metadata on both to prefer InnoDB's caching. Set `logbias=throughput` on the data to stop ZIL from writing twice.

Set `skip-innodb_doublewrite` in `my.cnf` to prevent innodb from writing twice. The double writes are a data integrity feature meant to protect against corruption from partially-written records, but those are not possible on ZFS. It should be noted that Percona's blog had advocated using an ext4 configuration where double writes were turned off for a performance gain, but later recanted it because it caused data corruption. Following a well timed power failure, an in place filesystem such as ext4 can have half of a 8KB record be old while the other half would be new. This would be the corruption that caused Percona to recant its advice. However, ZFS' copy on write design would cause it to return the old correct data following a power failure (no matter what the timing is). That prevents the corruption that the double write feature is intended to prevent from ever happening. The double write feature is therefore unnecessary on ZFS and can be safely turned off for better performance.

On Linux, the driver's AIO implementation is a compatibility shim that just barely passes the POSIX standard. InnoDB performance suffers when using its default AIO codepath. Set `innodb_use_native_aio=0` and `innodb_use_atomic_writes=0` in `my.cnf` to disable AIO. Both of these settings must be disabled to disable AIO.

5.5 PostgreSQL

Make separate datasets for PostgreSQL's data and WAL. Set `recordsize=8K` on both to avoid expensive partial record writes. Set `logbias=throughput` on PostgreSQL's data to avoid writing twice.

5.6 Virtual machines

Virtual machine images on ZFS should be stored using either zvols or raw files to avoid unnecessary overhead. The recordsize/volblocksize and guest filesystem should be configured to match to avoid overhead from partial record modification. This would typically be 4K. If raw files are used, a separate dataset should be used to make it easy to configure recordsize independently of other things stored on ZFS.

5.6.1 QEMU / KVM / Xen

AIO should be used to maximize IOPS when using files for guest storage.

CHAPTER 6

Boot process

On a traditional POSIX system, the boot process is as follows:

1. The CPU starts executing the BIOS.
2. The BIOS will do basic hardware initialization and load the bootloader.
3. The bootloader will load the kernel and pass information about the drive that contains the rootfs.
4. The kernel will do additional initialization, mount the rootfs and start /sbin/init.
5. `init` will run scripts that start everything else. This includes mounting filesystems from fstab and exporting NFS shares from exports.

There are some variations on this. For instance, the bootloaders will also load kernel modules on Illumos (via a boot_archive) and FreeBSD (individually). Some Linux systems will load an initramfs, which is an temporary rootfs that contains modules to load and moves some logic of the boot process into userland, mounts the real rootfs and switches into it via an operation called pivot_root. Also, EFI systems are capable of loading operating system kernels directly, which eliminates the need for the bootloader stage.

ZFS makes the following changes to the boot process:

- When the rootfs is on ZFS, the pool must be imported before the kernel can mount it. The bootloader on Illumos and FreeBSD will pass the pool informaton to the kernel for it to import the root pool and mount the rootfs. On Linux, an initramfs must be used until bootloader support for creating the initramfs dynamically is written.
- Regardless of whether there is a root pool, imported pools must appear. This is done by reading the list of imported pools from the zpool.cache file, which is at `/etc/zfs/zpool.cache` on most platforms. It is at `/boot/zfs/zpool.cache` on FreeBSD. This is stored as a XDR-encoded nvlist and is readable by executing the `zdb` command without arguments.
- After the pool(s) are imported, the filesystems must be mounted and any filesystem exports or iSCSI LUNs must be made. If the mountpoint property is set to legacy on a dataset, fstab can be used. Otherwise, the boot scripts will mount the datasets by running `zfs mount -a` after pool import. Similarly, any datasets being shared via NFS or SMB for filesystems and iSCSI for zvols will be exported or shared via `zfs share -a` after the mounts are done. Not all platforms support `zfs share -a` on all share types. Legacy methods may always be used and must be used on platforms that do not support automation via `zfs share -a`.

CHAPTER 7

License