ZFS 1010

Author: Aaron Toponce
Email: aaron.toponce@gmail.com
Date: February 23, 2013

Contact and Details

You can find the source code, PDF, compressed tarball and HTML presentation at http://aarontoponce.org/presents/zfs.

My email address: aaron.toponce@gmail.com.

License

What is ZFS?

- 128-bit filesystem (actually, not really)
- Copy-on-write filesystem

- Combined filesystem, volume and RAID manager
- Native support for SSDs
- Deduplication
- Compression
- Encryption (not Free Software)
- Checksums from top to bottom
- Developed from the ground up

Calling ZFS a 128-bit filesystem is a bit of false advertising. In practice, it's really only a 78-bit filesystem. Still, this can handle 256 zettabytes. However, this is a far cry from what would be advertized as 288,230,376,151,711,744 zettabytes. The current sizes are as follows:

- 2^48 — Number of entries in any individual directory
- 2^64 bytes — Maximum size of a single file
- 2^64 bytes — Maximum size of any attribute
- 2^78 bytes — Maximum size of any pool
- 2^56 — Number of attributes of a file (actually constrained to 2^48 for the number of files in a ZFS file system)
- 2^64 — Number of devices in any pool
- 2^64 — Number of pools in a system
- 2^64 — Number of file systems in a pool

Jeff Bonwick, the developer of ZFS makes an interesting observation about what it would take to store 2^128 bits worth of data:

> Although we'd all like Moore's Law to continue forever, quantum mechanics imposes some fundamental limits on the computation rate and information capacity of any physical device. In particular, it has been shown that 1 kilogram of matter confined to 1 liter of space can perform at most 10^51 operations per second on at most 1031 bits of information [see Seth Lloyd, "Ultimate physical limits to computation." Nature 406, 1047-1054 (2000)]. A fully-populated 128-bit storage pool would contain 2^128 blocks = 2^137 bytes = 2140 bits; therefore the minimum mass required to hold the bits would be (2^140 bits) / (10^31 bits/kg) = 136 billion kg.
>
> That's a lot of gear.
>
> To operate at the 10^31 bits/kg limit, however, the entire mass of the computer must be in the form of pure energy. By E=mc^2, the rest energy of 136 billion kg is 1.2x10^28 J. The mass of the oceans is about 1.4x10^21 kg. It takes about 4,000 J to raise the temperature of 1 kg of water by 1 degree Celcius, and thus about 400,000 J to heat 1 kg of water from freezing to boiling. The latent heat of vaporization adds another 2 million J/kg. Thus the energy required to boil the oceans is about 2.4x10^6 J/kg * 1.4x10^21 kg = 3.4x1^027 J. Thus, fully populating a 128-bit storage pool would, literally, require more energy than boiling the oceans.
>
> https://blogs.oracle.com/bonwick/entry/128_bit_storage_are_you

ZFS uses a a data storage technique called "copy-on-write". This is covered more in the next section.

ZFS combines the ideas of a traditional filesystem with volmue and RAID management. You don't need to continue exporting blocks of data in layers like traditional GNU/Linux filesystems. This simplifies the administration by quite a bit, and the training for future storage administrators is easier to handle.

Due to the rise of SSDs in the production storage market, ZFS was designed to take advantage of them. ZFS supports the wear leveling algorithms of SSDs, and as such, is suitable to use SSDs for external log devices, and/or read-only caches.

Transparent deduplication and compression can be enabled to get more out of your storage. Deduplication does require a deduplication table to be maintained in RAM. As such, it's recommended that you have sufficient amounts of RAM if using deduplication. If the deduplication table spills over to platter disk, this can drastically slow the filesystem down. Deduplication is handled at the block level.

However, compression does not incure such a performance penalty. ZFS supports both the gzip and lzjb compression algorithms, with the latter being the default. With gzip, levels 1 through 9 are supported, as are standard on Unix.

After Oracle purchased Sun Microsystems, and the code for ZFS went proprietary, Oracle released encryption support for ZFS. This is native AES 256-bit encryption, without the need for any external utilities. ZFS will take advantage of hardware AES instruction chips, should they be on the CPU die.

The most compelling feature of ZFS is that it was designed with ultimate data integrity in mind. Every block of the filesystem is checksummed with SHA-256 by default. When a block is read, it is hashed with the SHA-256 algorithm, then compared against the checksum in the metadata. This does incur a performance penalty, but as you'll see later, this is less of a concern. Further, all writes are done in transactions, where the pointers, blocks and log are written in one simultaneous write. As such, an `fsck(8)` is not needed with ZFS. As such, ZFS detects, and can fix, silent data errors imposed by hardware are any step of the write. Due to the atomic nature of transaction, the write is either all-or-nothing. So, at worst, you have old data. You never have corrupted data.

Lastly, ZFS was written from the ground up by Jeff Bonwick. It does not contain any legacy UFS or other filesystem code. It's modern, and behaves much more like a transactional relational database than a filesystem. Many of the old ideas were challenged, rethought, and implemented in new ways.

Copy-on-write

- Copy of block created, pointers updated
- Atomic- all or nothing
- Transactions- pointers, blocks, logs updated simultanously
- Uses a Merkle (hash) tree
- No need for `fsck(8)`
- ZFS "scrubs" data

http://pthree.org/?p=2832

Copy-on-write is a standardized data storage technique where rather than modifying the blocks in place, you copy the block to another location on disk. This does incure a performance penalty through disk fragmentation, but opens up a world of features that are more difficult to use.

Because you have a copy of the block, with the pointers updated to point to the new block instead of the old, you can have filesystem revisions. This allows you to "rollback" to a previous version of the filesystem. This is typically done through snapshots, which ZFS fully supports. You can then clone filesystems, and create forks for whatever reason.

ZFS uses transactions to commit the data to disk rather than a journal. Suppose a file is 100 blocks in size, and you nede to modify 10 of them. With typical journaling filesystems, you create a journal, update the inode, modify the first block, then close the journal. This is done for all 10 blocks. Thus, 40 writes to disk happen in this scenario. Instead, ZFS will pool all 10 blocks, as well as metadata and log updates, into a single simultaneous transactional write flush. As such, there is no need for a journal, and the chances of a power outage leading to data corruption is greatly minimized. Lastly, all transactions are atomic in nature- you either get it all, or you get nothing. You never only get part.

As a result, there is no need for a `fsck(8)` utility, as there is no journal to verify data consistency. Instead, ZFS uses a scrubbing technique that verifies (and corrects in the event of redundancy) the data integrity. It's recommended that you scrub your data weekly.

Installing ZFS on GNU/Linux

- http://zfsonlinux.org
- Ubuntu PPA
- DEB and RPM supported
- `aptitude install build-essential gawk alien fakeroot linux-headers-$(uname -r) zlib1g-dev uuid-dev libblkid-dev libselinux-dev parted lsscsi`
- `tar -xf spl*.tar.gz && cd spl*`
- `./configure && make deb && dpkg -i \*.deb`
- `tar -xf zfs*.tar.gz && cd zfs*`
- `./configure && make deb && dpkg -i \*.deb`

http://pthree.org/?p=2357

ZFS Commands

- 3 commands, one of which you will never use
- `zpool`- Configures storage pools
- `zfs`- Configures filesystems
- `zdb`- Display pool debugging and consistency information

Unlike traditional storage administration with GNU/Linux, there are only 3 commands that you need to be familiar with, the latter of which you will never use: `zpool(8)`, `zfs(8)` and `zdb(8)`.

The `zpool(8)` command is responsible for creating, configuring, displaying and destroying your ZFS storage pools. This including adding drives, configuring hot spares, setting properties, such as advanced format drive detection, and a number of other things. If you need to do anything with your physical disks, the `zpool(8)` is the command you reach for.

The `zfs(8)` command is responsible for creating, configuring, displaying and destroying your ZFS datasets. What you would typically call a "filelystem", ZFS refers to as a dataset. The reason being, is the filesystem stack is everything from your disks up to your data, of which a dataset is part of. Things like setting compression, sharing the dataset via NFS, SMB or iSCSI, creating snapshots, and sending and receiving datasets over the wire.

The `zdb(8)` command is a debugging utility for ZFS that can display and configure your ZFS filesystem by tuning various parameters. WARNING: There be dragons ahead! The `zdb(8)` command is a powerful utility that allows you get get into the bowels of ZFS and make some very majoyr changes to how the filesystem operates. You can completely screw up your data if you are not careful. It's likely that you will never need to use this command.

Beginner pool

- `zpool create tank sda sdb sdc`
- `zpool status tank`

To start off, let's create a simple pool. In this example, we have four disks: `/dev/sda`, `/dev/sdb` and `/dev/sdc` that we wish to put into our storage pool. We'll call the storage pool `tank` for lack of a better word. Notice, that I don't have to put the absolute path to the block devices. Any block device discoverable under `/dev/` can be referenced by its file name when using disks. Thus, referring to `/dev/sda as` just `sda` is perfectly acceptable.

After creating our storage pool, you can see the status with the `zpool status` command. Our output would look something like this:

```
  pool: tank
 state: ONLINE
 scan: none requested
config:

        NAME   STATE     READ WRITE CKSUM
         tank  ONLINE       0     0     0
          sda  ONLINE       0     0     0
          sdb  ONLINE       0     0     0
          sdc  ONLINE       0     0     0
```

VDEVs

- Virtual device managed by ZFS
- Always striped
- disk (default)
- file
- mirror
- raidz1/2/3
- spare
- log
- cache

http://pthree.org/?p=2584

VDEVs are virtual devices that are managed internally by ZFS. Similar to how Linux software RAID keeps a virtual block device `/dev/md0` to represent multiple disks, ZFS does the same, although it does not export a block device. There are seven VDEVs in ZFS, and VDEVs are always striped across each other.

Disk VDEV

- Dynamic striping (RAID 0)
- Disks can be full path, or those listed in `/dev/`
- Best practice to use the disks in `/dev/disk/by-id/`
- `zpool create tank sda sdb sdc`

This is the default VDEV, and can be any block device discoverable in `/dev/`. You can provide the full path, such as `/dev/sda`, or you can provide just the file `sda`. Because each disk provided as an argument is an individual VDEV, the data will be striped across all disks. An example would be:

```
# zpool create tank sda sdb sdc sdd
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:
```

```
        NAME   STATE     READ WRITE CKSUM
         tank  ONLINE       0     0     0
           sda  ONLINE       0     0     0
           sdb  ONLINE       0     0     0
           sdc  ONLINE       0     0     0
           sdd  ONLINE       0     0     0
```

A best practice would be to use the block devices found in `/dev/disk/by-id/` instead of `/dev/sda`, etc. The reason being, is some motherboards do not present the drives in the same order to Linux kernel on every boot. A drive discovered as `/dev/sda` on one boot might be discovered as `/dev/sdb` on another boot, whereas the `/dev/disk/by-id/ata-Maxtor_7L300S0_L60HZ9MH` will always be the same, even though its symlink may be pointing to the newly assigned drive letter.

For the main storage pool, using devices `/dev/sda` and `/dev/sdb` generally isn't a problem, as ZFS can read the metadata on the disks, and rebuild the pool as necessary. As such, the reason for this best practice advice may not be clear now, but will become more clear when we discuss the `log` and `cache` VDEVs.

File VDEV

- Really only useful for testing configurations
- Requires the absolute path
- Must be preallocated
- Compression, encryption or deduplication
- Resizing pool
- Sandbox
- NOT RECOMMENDED FOR PRODUCTION!!!
- `zpool create tank /tmp/file1 /tmp/file2 /tmp/file3`

Preallocated files can be used as storage for a pool. Sparse files will not work. You must provide the absolute path to every file when creating the pool. File VDEVs are not to be used for storing production data, but rather for testing scripts, setting properties, and understanding the internals of ZFS. An example would be:

```
# for i in {1..4}; do fallocate -l 1G /tmp/file$i; done
# zpool create tank /tmp/file1 /tmp/file2 /tmp/file3 /tmp/file4
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:

        NAME           STATE     READ WRITE CKSUM
         tank          ONLINE       0     0     0
           /tmp/file1  ONLINE       0     0     0
           /tmp/file2  ONLINE       0     0     0
           /tmp/file3  ONLINE       0     0     0
           /tmp/file4  ONLINE       0     0     0

errors: No known data errors
```

When working with files in your storage pool, you should not create them from files that belong to an existing pool. Race conditions will occur, and you could end up with corrupted data. Rather, create the files on another filesystem, such as ext4.

Mirror VDEV

- Standard RAID 1
- Fastest redundancy for reads and writes
- Two or more disks
- `zpool create tank mirror sda sdb sdc sdd`

The standard RAID-1 mirror. All disks contain the same data as every other disk in the mirrored VDEV. This is the most expensive VDEV in terms of storage, but in terms of sequential reads and writes, will also be the fastest. An example would be:

```
# zpool create tank mirror sda sdb sdc sdd
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:

    NAME        STATE     READ WRITE CKSUM
    tank        ONLINE       0     0     0
      mirror-0  ONLINE       0     0     0
        sda     ONLINE       0     0     0
        sdb     ONLINE       0     0     0
        sdc     ONLINE       0     0     0
        sdd     ONLINE       0     0     0

errors: No known data errors
```

As mentioned previously, VDEVs are striped across each other. This makes it easy to create "nested" VDEVs. In this example, instead of mirroring 4 disks, would could create two mirror VDEV:

```
# zpool create tank mirror sda sdb mirror sdc sdd
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:

    NAME        STATE     READ WRITE CKSUM
    tank        ONLINE       0     0     0
      mirror-0  ONLINE       0     0     0
        sda     ONLINE       0     0     0
        sdb     ONLINE       0     0     0
      mirror-1  ONLINE       0     0     0
        sdc     ONLINE       0     0     0
        sdd     ONLINE       0     0     0

errors: No known data errors
```

Notice that the first mirrored VDEV is labeled as `mirror-0` and is assigned to `sda` and `sdb` as we provided in our command. The second mirror VDEV is labeled as `mirror-1`, and is assigned to `sdc` and `sdd` as we also provided in our command. Because VDEVs are always striped, the data is striped across the `mirror-0` and `mirror-1` VDEVs, thus creating our RAID-1+0. Note, this is not `RAID-10`. The Linux kernel has a non-standard RAID-10 level, where it is not clear where the mirrors exist. In ZFS, this is using two standardized RAID levels- a stripe of mirrors. Thus, it is RAID-1+0.

This helps regain space back from your mirror, while also greatly improving sequential performance. Nested VDEVs will always outperform non-nested VDEVs in terms of sequential reads and writes.

You can create nested VDEVs with mirrors or RAIDZ, as we'll discover here.

RAID-Z VDEVs

- RAID-Z1- Single distributed parity (RAID 5 (slow))
- RAID-Z2- Dual distributed parity (RAID 6 (slower))
- RAID-Z3- Triple distributed parity (slowest)
- `zpool create tank raidz1 sda sdb sdc`
- `zpool create tank raidz2 sda sdb sdc sdd`
- `zpool create tank raidz3 sda sdb sdc sdd sde`

http://pthree.org/?p=2590

To understand RAIDZ, you must first understand standards-based parity RAID, such as RAID-5 and RAID-6. With RAID-5, a single parity bit is distributed across the disks in a striped array, rather than using a dedicated parity disk, such as you would find in RAID-4. This balances reads, writes and disk wear. Further, it allows you to lose any one disk in the array, as the missing data can be re-calculated by the remaining data in the stripe, and the parity bit. RAID-6 uses a dual-distributed parity, and as a result, can lose any two disks in the array before there is data loss.

The problem with RAID-5 and RAID-6 is two-fold. First, the stripe is written to disk before the parity bit. This means that if there is a power outage after writing the stripe, and before writing the parity, you have a problem. This problem is referred to in the ZFS community as the "RAID-5 write hole". When power is restored, even though the data in the stripe might be consistent, the parity may not be the write bit to represent the XOR in the stripe. As such, if a disk failure occurs in the array, the parity will not be able to help reconstruct the data, and you have bad data being sent to the application.

The second problem is the stripe width. In standards-based parity arrays, the stripe width is exactly n-1 disks in the array if using RAID-5, and n-2 disks if using RAID-6. To determine your exact stripe width, use the following calculation:

- Divide the chunk size by the block size for one spindle/drive only. This gives you your stride size.
- Then you take the stride size, and multiply it by the number of data-bearing disks in the RAID array. This gives you the stripe width to use when formatting the volume.

If your chunk size is 512 KB (default for Linux software RAID), and your filesystem block size is 4 KB (default for ext4), then your stride size is 128 KB. If this is a 4-disk RAID-5, then your stripe width is a static 384 KB.

Not all data you commit to disk will be in multiples of 384 KB. Most likely, you will spend a lot of time writing data that is less than 384 KB and you will spend a lot of time writing data is that more than 384 KB. So, if you commit data to a RAID-5 array that is only a few kilobytes, then when calculating the parity bit, you must XOR data on disks in the stripe that does not contain live, actual data. As a result, you spend a great deal of time calculating the parity for "dead data".

Now, let's take a look at ZFS RAIDZ.

Other VDEVs

- Spare- Hot spare to replace failed disks
- Log- Write-intensive separate LOG (SLOG) called ZFS Intent Log (ZIL)
- Cache- Read-intensive L2ARC cache
- Log and cache should and be on fast SSDs
- `zpool add tank spare sda`
- `zpool add tank log mirror sda sdb`
- `zpool add tank cache sdb sdc`

The spare VDEV allows us to create hot spares in the event of a disk failure. This will ensure that in the event of drive failure, a new disk will replace the failed disk, and you won't have a degraded pool. However, even though you can create a spare VDEV, it does not replace failed disk by default. You must enable this after setting up your pool. This is done with the following command:

```
# zpool set autoreplace=on tank
```

Two additional VDEVs allow us to create hybrid ZFS Storage pools. These VDEVs enhance the functionality of your pool, and can greatly decrease latencies, and response times if fast SSDs or RAM drives are used. These are explained in the next slide.

The Separate Intent Log (SLOG)

- SSD, NVRAM or 15k RPM SAS
- Stores the ZFS Intent Log (ZIL)
- Write intensive
- Consider life expectency
- Can be very small
- Should be mirrored

http://pthree.org/?p=2564 and http://pthree.org/?p=2592

The `log` VDEV should be setup with a fast SSD or RAM drive. It's known as a "separate log device", or "slog" for short. The ZFS intent log, or "ZIL" is typically stored on the same drives as your ZFS pool. However, if you add a SLOG to the pool, then the ZIL will be transferred to the SLOG. The ZIL acts very much like a journal, but it's not a journal. All synchronous write transactions will be written to the SLOG first, then flushed to slower platter later. ZFS flushes the SLOG to platter disk every 5 seconds by default. The SLOG is not used with asynchronous transactions, and not all synchronous writes use the SLOG. Basically, only those transactions thas call O_SYNC or F_SYNC.

As an entertaining sidenote, the "SLOG" device is typically referred to as the "slogzilla". Your SLOG likely does not need to be very big. Of course, this all depends on your data needs, but in practice, it's pretty difficult to get the SLOG to see more than 1GB of data. Further, your SLOG needs to maintain consistency when no power is present. AN SSD is good for this, but battery-backed RAM drives can work as well. The SLOG should be mirrored to maintain integrity.

The Adjustable Replacement Cache

- Traditional Linux cache: MRU/LRU

- ZFS ARC is a modification of IBM ARC
- ARC in main RAM
- ARC stores an MRU and MFU pages
- Level 2 ARC (L2ARC) exists on fast disk
- L2ARC stores ghost MRU and MFU pages
- Read intensive
- Should be very large
- Can be volatile and striped

http://pthree.org/?p=2659

The `cache` VDEV should also be setup with a fast SSD or RAM drive. The cache is actually a level adjustable replacement cache, or "ARC". The ARC is stored in main system memory. The ARC stores two caches- the most recently used pages, and the most frequently used pages. When these pages fill the ARC, any additional request will push pages to a level two ARC, or "L2ARC". This is your cache VDEV specified when building the pool.

The L2ARC is a read-only cache. Active written data is not stored in the L2ARC. Actively synchronous written data is stored only in the SLOG and main platter drives. Because the L2ARC is a fast device, such as an SSD or RAM drive, this maintains that discarded MRU and MFU pages remain snappy to the application. The L2ARC should be as large as possible, and to increase throughput, can be striped. In the event of a power outage, the L2ARC data will be discarded, which is okay, because the data already exists on slower platter disk, and can be rebuilt when power resumes.

Hybrid Storage Pools

- See handout

To illustrate setting up a hybrid storage pool, suppose we have two 40 GB Intel SSDs. Each SSD is partitioned the same, where the first partition is 1 GB in size for the SLOG, and the second partition is the remaining 39 GB for the L2ARC. Here is how you could set it up, with 4 disks in a RAID-1+0:

```
# zpool create tank mirror sda sdb mirror sdc sdd log sde1 sdf1 cache sde2 sdf2
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:

    NAME        STATE     READ WRITE CKSUM
    tank        ONLINE       0     0     0
      mirror-0  ONLINE       0     0     0
        sda     ONLINE       0     0     0
        sdb     ONLINE       0     0     0
      mirror-1  ONLINE       0     0     0
        sdc     ONLINE       0     0     0
        sdd     ONLINE       0     0     0
    logs
      mirror-2  ONLINE       0     0     0
        sde1    ONLINE       0     0     0
        sdf1    ONLINE       0     0     0
    cache
      sde2      ONLINE       0     0     0
```

```
      sdf2      ONLINE       0     0     0

errors: No known data errors
```

Export and Import pools

- Offlines all disks
- Creates `/etc/zfs/zpool.cache`
- Useful for migrating disks

http://pthree.org/?p=2594

Exporting a ZFS pool allows you to migrate disks from one storage server to another. Exporting the pool will create a cache file found in /etc/zfs/zpool.cache, which can then be used on the new server, if need be. This is done with the `zpool export` command.

Just as you can export a ZFS pool, you can use the `zfs import` command to import a pool on the new server. However, if the pool was not properly exported, then ZFS will complain, and will not let you import the pool on the new server.

If you destroy a ZFS pool, you can still import the pool- not all is lost. Just used the `zpool import -D` switch with the command to import a destroyed pool. ZFS will then do a sanity check on the metadata, making sure everything looks good, before bringing the pool ONLINE. This may also mean scrubbing and resilvering the data as needed.

WARNING: You should only import ZFS pools onto servers that have the same or a newer version of ZFS. ZFS will not allow you to import pools that have a newer version on the source, than on the destination.

Scrub and Resilver pools

- Every block (data and meta) SHA-256 checksummed
- Self healing with redundancy
- `zpool scrub tank`
- Resilver rebuilds missing data on new disk
- `zpool replace tank sde sdh`

http://pthree.org/?p=2630

Most filesystem utilities have a standardized way to do data validation through an `fsck(8)` utility. This utility is typically for journaled filesystems, and reads the journal, along with the inodes and metadata, to determine data consistency. Unfortunately, these utilities don't provide protection against silent disk errors, nor can they be run while the disk or partition is online.

With ZFS, every block, both metadata and data, is checksummed using the SHA-256 algorithm. This means that if data corruption were to occur, it is practically impossible that the corrupted block has the same SHA-256 checksum as what it was originally intended to be. The checksum itself resides in the matadata.

There are two ways in which ZFS knows that a block has been corrupted: through manual scrubbing, and through automatic detection by application requests. Scrubbing is similar in nature to a typical `fsck(8)`, except that the scrub can happen whele the data is online. If corruption is detected in the scrub, then the block can be corrected if good data exists elsewhere in a redundant pool. It is recommended to manually scrub your data once per week for consumer drives, and once per month for enterprise drives.

As disks die, they need to be replaced. This means that when replacing a disk in ZFS, the data missing needs to be repopulated onto the new disk. However, in ZFS our rebuild of the data is smarter than standard utilities, such as MD RAID in the Linux kernel. Most rebuilds will read the entire disks in the pool, and make the necessary changes on the new disk. With ZFS, we know where live data resides, and where it doesn't. So, we don't need to read all the blocks on the all the disks, we only need to read the live data. As such, ZFS refers to this as a resilver.

To replace a disk in an array, use the `zfs replace` command:

```
# zpool replace tank sde sde
# zpool status tank
  pool: tank
 state: ONLINE
status: One or more devices is currently being resilvered.  The pool will
        continue to function, possibly in a degraded state.
action: Wait for the resilver to complete.
 scrub: resilver in progress for 0h2m, 16.43% done, 0h13m to go
config:

        NAME          STATE       READ WRITE CKSUM
        tank          DEGRADED       0     0     0
          mirror-0    DEGRADED       0     0     0
            replacing DEGRADED       0     0     0
            sde       ONLINE         0     0     0
            sdf       ONLINE         0     0     0
          mirror-1    ONLINE         0     0     0
            sdg       ONLINE         0     0     0
            sdh       ONLINE         0     0     0
          mirror-2    ONLINE         0     0     0
            sdi       ONLINE         0     0     0
            sdj       ONLINE         0     0     0
```

Zpool Properties

- `zpool get all tank`
- `ashift=9` or `ashift=12`
- `autoexpand` off by default
- `autoreplace` off by default
- `health` shows status of pool

http://pthree.org/?p=2632

The ZFS pool has a number of different properties that you can apply that affect the operation of the pool differently. Think of this as tuning parameters for the filesystem at the pool level. The properties are listed as follows:

To get a property from the pool, use the `zpool get` command. You can provide the `all` option to list all properties of the pool, or you can list properties that you want to view, comma-separated:

```
# zpool get health tank
NAME  PROPERTY  VALUE   SOURCE
tank  health    ONLINE  -

# zpool get health,free,allocated tank
```

```
NAME  PROPERTY   VALUE   SOURCE
tank  health     ONLINE  -
tank  free       176G    -
tank  allocated  32.2G   -

# zpool get all tank
NAME  PROPERTY       VALUE       SOURCE
tank  size           208G        -
tank  capacity       15%         -
tank  altroot        -           default
tank  health         ONLINE      -
tank  guid           16951123779703469 70  default
tank  version        28          default
tank  bootfs         -           default
tank  delegation     on          default
tank  autoreplace    off         default
tank  cachefile      -           default
tank  failmode       wait        default
tank  listsnapshots  off         default
tank  autoexpand     off         default
tank  dedupditto     0           default
tank  dedupratio     1.00x       -
tank  free           176G        -
tank  allocated      32.2G       -
tank  readonly       off         -
tank  ashift         0           default
tank  comment        -           default
tank  expandsize     0           -
```

All of the properties can be found at my blog post above, in the `zpool(8)` manual, or at the official Solaris documentation.

To set a property, use the `zfs set` command. However, there is a catch: for properties that require a string argument, there is no way to get back to default without know what the default is beforehand. If I wished to set the comment property for the pool, I could do the following:

```
# zfs set comment="Contact admins@example.com" tank
# zpool get comment tank
NAME  PROPERTY  VALUE                       SOURCE
tank  comment   Contact admins@example.com  local
```

Zpool Best Practices and Caveats

- 64-bit
- Gobs of ECC RAM
- Use whole disks
- Keep pools separate
- Speed: RAID-1+0 > RAIDZ1 > RAIDZ2 > RAIDZ3
- Power of 2 plus parity
- Fast disk for SLOG & L2ARC
- Regular scrubs

- Enable compression

http://pthree.org/?p=2782

As with most "best practices", these guidelines do carry a great amount of weight, but they also may not work for your environment. Heed the advice, but do as you must:

- Only run ZFS on 64-bit kernels. It has 64-bit specific code that 32-bit kernels cannot do anything with.
- Install ZFS only on a system with lots of RAM. 1 GB is a bare minimum, 2 GB is better, 4 GB would be preferred to start. Remember, ZFS will use 7/8 of the available RAM for the ARC.
- Use ECC RAM when possible for scrubbing data in registers and maintaining data consistency. The ARC is an actual read-only data cache of valuable data in RAM.
- Use whole disks rather than partitions. ZFS can make better use of the on-disk cache as a result. If you must use partitions, backup the partition table, and take care when reinstalling data into the other partitions, so you don't corrupt the data in your pool.
- Keep each VDEV in a storage pool the same size. If VDEVs vary in size, ZFS will favor the larger VDEV, which could lead to performance bottlenecks.
- Use redundancy when possible, as ZFS can and will want to correct data errors that exist in the pool. You cannot fix these errors if you do not have a redundant good copy elsewhere in the pool. Mirrors and RAID-Z levels accomplish this.
- For the number of disks in the storage pool, use the "power of two plus parity" recommendation. This is for storage space efficiency and hitting the "sweet spot" in performance. So, for a RAIDZ-1 VDEV, use three (2+1), five (4+1), or nine (8+1) disks. For a RAIDZ-2 VDEV, use four (2+2), six (4+2), ten (8+2), or eighteen (16+2) disks. For a RAIDZ-3 VDEV, use five (2+3), seven (4+3), eleven (8+3), or nineteen (16+3) disks. For pools larger than this, consider striping across mirrored VDEVs.
- Consider using RAIDZ-2 or RAIDZ-3 over RAIDZ-1. You've heard the phrase "when it rains, it pours". This is true for disk failures. If a disk fails in a RAIDZ-1, and the hot spare is getting resilvered, until the data is fully copied, you cannot afford another disk failure during the resilver, or you will suffer data loss. With RAIDZ-2, you can suffer two disk failures, instead of one, increasing the probability you have fully resilvered the necessary data before the second, and even third disk fails.
- Perform regular (at least weekly) backups of the full storage pool. It's not a backup, unless you have multiple copies. Just because you have redundant disk, does not ensure live running data in the event of a power failure, hardware failure or disconnected cables.
- Use hot spares to quickly recover from a damaged device. Set the `autoreplace` property to on for the pool.
- Consider using a hybrid storage pool with fast SSDs or NVRAM drives. Using a fast SLOG and L2ARC can greatly improve performance.
- If using a hybrid storage pool with multiple devices, mirror the SLOG and stripe the L2ARC.
- If using a hybrid storage pool, and partitioning the fast SSD or NVRAM drive, unless you know you will need it, 1 GB is likely sufficient for your SLOG. Use the rest of the SSD or NVRAM drive for the L2ARC. The more storage for the L2ARC, the better.
- Keep pool capacity under 80% for best performance. Due to the copy-on-write nature of ZFS, the filesystem gets heavily fragmented.
- Email reports of capacity at least monthly.
- Scrub consumer-grade SATA and SCSI disks weekly and enterprise-grade SAS and FC disks monthly.

- Email reports of the storage pool health weekly for redundant arrays, and bi-weekly for non-redundant arrays.
- When using advanced format disks that read and write data in 4 KB sectors, set the `ashift` value to 12 on pool creation for maximum performance. Default is 9 for 512-byte sectors.
- Set `autoexpand` to on, so you can expand the storage pool automatically after all disks in the pool have been replaced with larger ones. Default is off.
- Always export your storage pool when moving the disks from one physical system to another.
- When considering performance, know that for sequential writes, mirrors will always outperform RAID-Z levels. For sequential reads, RAID-Z levels will perform more slowly than mirrors on smaller data blocks and faster on larger data blocks. For random reads and writes, mirrors and RAID-Z seem to perform in similar manners. Striped mirrors will outperform mirrors and RAID-Z in both sequential, and random reads and writes.
- Compression is disabled by default. This doesn't make much sense with today's hardware. ZFS compression is extremely cheap, extremely fast, and barely adds any latency to the reads and writes. In fact, in some scenarios, your disks will respond faster with compression enabled than disabled. A further benefit is the massive space benefits.

As with any "best practices" list, there is also warnings and caveats you should be aware of. As with the list above, this is by no means exhaustive, but following it should help you in most situations:

- Your VDEVs determine the IOPS of the storage, and the slowest disk in that VDEV will determine the IOPS for the entire VDEV.
- ZFS uses 1/64 of the available raw storage for metadata. So, if you purchased a 1 TB drive, the actual raw size is 976 GiB. After ZFS uses it, you will have 961 GiB of available space. The `zfs list` command will show an accurate representation of your available storage. Plan your storage keeping this in mind.
- ZFS wants to control the whole block stack. It checksums, resilvers live data instead of full disks, self-heals corrupted blocks, and a number of other unique features. If using a RAID card, make sure to configure it as a true JBOD (or "passthrough mode"), so ZFS can control the disks. If you can't do this with your RAID card, don't use it. Best to use a real HBA.
- Do not use other volume management software beneath ZFS. ZFS will perform better, and ensure greater data integrity, if it has control of the whole block device stack. As such, avoid using dm-crypt, mdadm or LVM beneath ZFS.
- Do not share a SLOG or L2ARC DEVICE across pools. Each pool should have its own physical DEVICE, not logical drive, as is the case with some PCI-Express SSD cards. Use the full card for one pool, and a different physical card for another pool. If you share a physical device, you will create race conditions, and could end up with corrupted data.
- Do not share a single storage pool across different servers. ZFS is not a clustered filesystem. Use GlusterFS, Ceph, Lustre or some other clustered filesystem on top of the pool if you wish to have a shared storage backend.
- Other than a spare, SLOG and L2ARC in your hybrid pool, do not mix VDEVs in a single pool. If one VDEV is a mirror, all VDEVs should be mirrors. If one VDEV is a RAIDZ-1, all VDEVs should be RAIDZ-1. Unless of course, you know what you are doing, and are willing to accept the consequences. ZFS attempts to balance the data across VDEVs. Having a VDEV of a different redundancy can lead to performance issues and space efficiency concerns, and make it very difficult to recover in the event of a failure.
- Do not mix disk sizes or speeds in a single VDEV. Do mix fabrication dates, however, to prevent mass drive failure.
- In fact, do not mix disk sizes or speeds in your storage pool at all. Do not mix disk counts across VDEVs. If one VDEV uses 4 drives, all VDEVs should use 4 drives.

- Do not put all the drives from a single controller in one VDEV. Plan your storage, such that if a controller fails, it affects only the number of disks necessary to keep the data online.
- When using advanced format disks, you must set the ashift value to 12 at pool creation. It cannot be changed after the fact. Use `zpool create -o ashift=12 tank mirror sda sdb` as an example.
- Hot spare disks will not be added to the VDEV to replace a failed drive by default. You MUST enable this feature. Set the autoreplace feature to on. Use `zpool set autoreplace=on tank` as an example.
- The storage pool will not auto resize itself when all smaller drives in the pool have been replaced by larger ones. You MUST enable this feature, and you MUST enable it before replacing the first disk. Use `zpool set autoexpand=on tank` as an example.
- ZFS does not restripe data in a VDEV nor across multiple VDEVs. Typically, when adding a new device to a RAID array, the RAID controller will rebuild the data, by creating a new stripe width. This will free up some space on the drives in the pool, as it copies data to the new disk. ZFS has no such mechanism. Eventually, over time, the disks will balance out due to the writes, but even a scrub will not rebuild the stripe width.
- You cannot shrink a pool, only grow it. This means you cannot remove VDEVs from a storage pool.
- You can only remove drives from mirrored VDEV using the `zpool detach` command. You can replace drives with another drive in RAIDZ and mirror VDEVs however.
- Do not create a storage pool of files or ZVOLs from an existing pool. Race conditions will be present, and you will end up with corrupted data. Always keep multiple pools separate.
- The Linux kernel may not assign a drive the same drive letter at every boot. Thus, you should use the /dev/disk/by-id/ convention for your SLOG and L2ARC. If you don't, your pool devices could end up as a SLOG device, which would in turn clobber your ZFS data.
- Don't create massive storage pools "just because you can". Even though ZFS can create 78-bit storage pool sizes, that doesn't mean you need to create one.
- Don't put production directly into the pool. Use ZFS datasets instead.
- Don't commit production data to file VDEVs. Only use file VDEVs for testing scripts or learning the ins and outs of ZFS.

Creating ZFS Datasets

- Combined volume manager & filesystem
- Filesystem == Dataset
- Filesystems have all pool storage
- Filesystems aware of each other's storage
- Nested datasets possible
- `zfs create tank/store1`
- `zfs list`

http://pthree.org/?p=2849

ZFS datasets are a bit different than standard filesystems and volume management tools in GNU/Linux. With the standard LVM approach, each container is limited in size, until the full volume group is filled. This means that if one logical volume needs more space, you must reduce the space in another volume. While this is highly simplified from the partition approach, it's still not optimal. Further, this problem is complicated by the myriad of tools that you must use to do the task. If that's not enough, aligning the filesystem exactly with the container can be problematic as well.

With ZFS, each filesystem is called a "dataset". Each dataset by default has full 100% access to the entire pool. Thus, if your pool is 5 TB in size, each dataset created will have access to all 5 TB. This means that each dataset must be aware of how much space the other datasets are storing. Also, it's important to note, that while every dataset shares the same pool, you cannot share the pool across multiple operating systems. ZFS is not a clustered filesystem.

Datasets that reside inside other datasets are also possible. For example, maybe you have a dataset for /var. You can also have a dataset for /var/log/ and /var/cache/. These are known as nested datasets, and have their own dataset properties, and also have full 100% access to the entire storae pool by default.

To create a dataset, use the `zfs create <pool>/<dataset>` command, where `<pool>` is the name of the pool you want to create the dataset from, and `<dataset>` is the dataset you wish to create. Like so:

```
# zfs create tank/store1
```

To see the created datasets, use the `zfs list` command:

```
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         175K  2.92G  43.4K  /tank
tank/store1  41.9K 2.92G  41.9K  /tank/store1
```

Should the need arise to destroy a dataset, use the `zfs destroy` command:

```
# zfs destroy tank/store1
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         175K  2.92G  43.4K  /tank
```

Compression

- LZJB invented by Jeff Bonwick
- `lzjb`, `gzip` 1-9 and `zle` supported
- ZOL rc14 supports LZ4 (fast)
- `zfs set compression=lzjb tank/store1`

http://pthree.org/?p=2878

Compression in 100% transparent to the user and applications. Jeff Bonwick invented "LZJB" which is a variant of the Lempel-Ziv algorithm. It's fast and lightweight, and because computer processors have gotten multiple cores, and exceptionally fast, it's almost a no-brainer to enable compression. In my tests, I've seen less than a 1% performance hit.

ZFS supports also supports the gzip(1) and ZLE algorithms. ZLE will provide the best performance, but won't give you outstanding algorithms. ZFS supports all 9 levels of gzip(1), with level 1 being the most fast and less tight level to level 9 being the slowest and most tight level. As of 0.6.0-rc14 with ZFS on Linux, the LZ4 algorithm is also supported, which provides amazing performance, and amazing compression ratios.

To set compression on your datasets, use the `zfs set` command:

```
# zfs set compression=lzjb tank/store1
```

It's probably best practice to specify the algorithm rather than specifying `on` as the argument. The reason being, is if for any reason, the default compression algorithm is changed, the underlying blocks will remain standardized on a single algorithm. LZJB is also probably recommended.

Deduplication

- Block level
- Deduplication table stored in RAM
- Occupies 25% of the ARC
- Rule of thumb: 5 GB of RAM for every 1 TB of disk
- Set on datasets, applied pool-wide
- `zfs set dedup=on tank/store1`

http://pthree.org/?p=2878

Deduplication is another way to save storage space. ZFS deduplicates data on the block level, rather than the file or byte levels. Deduplication is set per-dataset, but the data is deduplicated against all the data in the pool. To know what data has been deduplicated, a deduplication table is stored in the ARC in RAM.

The deduplication table can be a bit unwieldly. It occupies 25% of the ARC by default, and it's a good "rule of thumb" to assume that the deduplication table needs 5 GB of ARC for every 1 TB of deduplicated disk. So, for 1 TB of disk, you should have at least 20 GB of RAM. If you don't the deduplication table will spill out to disk. This may not be a problem with a fast SSD or NVRAM drive, but with platter, can be horrendous.

Because deduplication is checking against every block in the pool, this means that if you disable deduplication on a dataset, the table still must exist to manage existing deduplicated data, until all data blocks are no longer deduplicated. So, if you find that deduplication is causing problems, disabling it will not provide immediate relief.

To set deduplication on a dataset, use the `zfs set` command as well:

```
# zfs set dedup=on tank/store1
```

Snapshots and Clones

- First-class read-only filesystems
- Store only deltas
- `pool@snapshot-name`
- `pool/dataset@snapshot-name`
- `zfs snapshot tank/store1@001`
- `zfs list -t snapshot`
- `ls /pool/dataset/.zfs/snapshot/`

http://pthree.org/?p=2900

Think of snapshots as a digital photograph in time. When you walk out to a busy intersection, and take a photo with your camera, you get a snashot of what was going on at that intersection at that specific amount of time. Even though the intersection is constantly changing, you will always have a physical representation of that the data looked like at that intersection at that specific moment. Filesystem snapshots are similar in nature.

A snapshot is a first-class filesystem. All of the data is fully accessible, and can be copied, archived, or sent to other computers. However, snapshots are read-only. You cannot modify data in the snopshot itself. You must copy the data out of the snapshot, before modification. The reason being, is snapshots store only delta changes from the time that the snapshot was taken. This means that snapshots are cheap in terms of storage. You colud store millions of snapshots, so long as the deltas remain small, and you have the storage.

To create a snapshot of a dataset, you must give the snapshot a name. Use the `zfs snapshot <pool>/<dataset>@<name>` syntax for creating your snapshot, as follows:

```
# zfs snapshot tank/store1@001
```

To see our snapshot, we have two ways of viewing it. We can pass the `-t snapshot` argument to the `zfs list` command, or we can cd(1) into the directory. Let's look at the `zfs list -t snapshot` command first:

```
# zfs list -t snapshot -r tank/store1
NAME              USED  AVAIL  REFER  MOUNTPOINT
tank/store1@001      0      -   345K  -
```

We can also see tho contents of the snapshot by looking in the hidden .zfs directory under the dataset mountpoint:

```
# ls /tank/store1/.zfs/snapshot/
001/
# ls /tank/store1/.zfs/snapshot/001/
file10.img  file1.img  file2.img  file3.img  file4.img
file5.img   file6.img  file7.img  file8.img  file9.img
```

By default, the `.zfs` directory is hidden, even from ls(1). To show the directory as a standard hidden directory, you must set the `snapdir` dataset property to "visible":

```
# zfs set snapdir=visible tank/store1
```

Now, if we were to change some data, our snapshot should reflect the changes:

> # dd if=/dev/random of=/tank/store1/file11.img bs=1 count=$RANDOM # dd if=/dev/random of=/tank/store1/file5.img bs=1 count=$RANDOM # zfs list -t snapshot -r tank/store1 NAME USED AVAIL REFER MOUNTPOINT tank/store1@001 55.9K - 345K - # ls /tank/store1/.zfs/snapshot/001/ file10.img file1.img file2.img file3.img file4.img file 5.img file6.img file7.img file8.img file9.img

Notice that the contents of file5.img changed, so the snapshot ended up getting a copy of the original data. However, file11.img did not exist at the time of the snapshot, so it is not part of the snapshot itself.

Because snapshots are first-class filesystems, they are treated as standard datasets for the most part, and can be destroyed just like a standard ZFS dataset.

Sending and Receiving Snapshots

- Full filesystems send to STDOUT
- Redirect to image file
- Combine with OpenSSL/GnuPG
- Redirect to STDIN
- Receive filesystems

- Combine with OpenSSH
- `zfs send tank/store1@001 | zfs recv backup/store1`

http://pthree.org/?p=2910

ZFS has the capability to send snapshots to other ZFS pools. This gives us the ability to send full, complete filesystems to another destination, while keeping the original source filesystem intact. To send a ZFS snapshot, we use the `zfs send` command.

By default, it will send all of the data to STDOUT. This allows us to use the standard Unix utilities to redirect and manipulate the data. For example, we could send the filesystem to a single monolithic image:

```
# zfs send tank/store1@001 > /tank/store1/001.img
```

We could encrypt and/or compress the data using gzip(1), openssl(1) and gpg(1):

```
# zfs send tank/store1@001 | gzip > /tank/store1/001.img.gz
# zfs send tank/store1@001 | gzip | openssl enc -aes-256-cbc -a -salt > /tank/store1/001.img.gz.asc
```

ZFS also provides a utility to catch the stream on STDIN, and send it to a dataset in another pool with the `zfs receive` command. For example, if I had a pool named `backup`, I could send my snapshot from tank to backup as follows:

```
# zfs send tank/store1@001 | zfs recv backup/store1
```

Note that the `recv` subcommand is an alias to `receive`.

ZVOLs

- Exported block device
- Can be formatted with any filesystem
- Useful for swap and VM storage
- `zfs create -V 1G tank/vol1`
- `mkfs -t ext4 /dev/zvol/tank/vol1`

http://pthree.org/?p=2933

ZFS allows you to create exported block devices, as you may already be used to with `mdadm(8)` and LVM2. With a block device, you can do anything with it that you could do with standard block devices. For example, it could be used as a swap device. It could be used as the block device for a virtual machine. You could even format it with ext4, and mount it. These block devices are referred to as ZVOLs.

To create a ZVOL, just pass the `-V` switch with an argument to its size. For example, if I wished to create a 1 GB ZVOL to use for swap, I could issue the following command:

```
# zfs create -V 1G tank/swap
```

To verify that I have created the block device, a file was created under /dev, with symlinks created elsewhere:

```
# find /dev -ls | grep zd
476149    0 lrwxrwxrwx   1 root     root            9 Feb 22 08:17 /dev/zvol/tank/swap -> ../../zd0
476144    0 lrwxrwxrwx   1 root     root            6 Feb 22 08:17 /dev/tank/swap -> ../zd0
476138    0 brw-rw---T   1 root     disk              Feb 22 08:17 /dev/zd0
476142    0 lrwxrwxrwx   1 root     root            6 Feb 22 08:17 /dev/block/230:0 -> ../zd0
```

Now that we have our swap device, let's set it up. First we must format it as a swap device, so the kernel knows it's available, then we must "turn it on":

```
# mkswap /dev/zd0
# swapon /dev/zd0
```

We can now verify that we have an extra 1 GB of swap for the kernel to use. When using swap devices, it's best practice to disable synchronous writes, to make "the swap of death" as painless as possible. Use `zfs set` to enable that:

```
# zfs set sync=disabled tank/swap
```

Unfortunately with ZFS on Linux, as of rc14, there is a bug with VZOLs where they are not created on boot. There is a work around until the bug is fixed, which is to rename the ZVOL twice. Put the following into your /etc/rc.local:

zfs rename tank/swap tank/foo zfs rename tank/foo tank/swap swapon tank/swap

Now your swap ZVOL will be ready for use.

This is only one example of how to use a ZVOL. As mentioned, you could use them for storage for virtual machines, you could format them using any standard filesystem utility, such as `mkfs(8)`.

Sharing via NFS, SMB and iSCSI

- Native support
- Requires running daemon
- Benefit: dataset mounted, before shared
- Full NFS option support
- SMB support spotty for GNU/Linux
- No iSCSI support for GNU/Linux... yet
- `zfs set sharenfs|sharesmb|shareiscsi tank/store1`

http://pthree.org/?p=2943

Native NFS, Samba (CIFS) and iSCSI support are built into ZFS. The largest reason for this, is to enable an order of operational consistency. For example, you want to make sure your dataset is mounted before your export your directory over NFS. You don't want NFS clients sending data to the export prematurely. Thus, the ZFS dataset will be mounted before the export is available on the network, this NFS clients can safely send data to it.

For NFS, full NFS option support is builtin. However, it does require a running NFS daemon on the server. With Debian and Ubuntu GNU/Linux, the exports(5) file must be created before you share the ZFS datasets via NFS.

To enable NFS support for your dataset, just use the `zfs set` command. Because all of the full NFS v3 options are supported, you can pass them as arguments to the command. For example, if I wanted to make the /tank/share/ directory available only to the 10.80.86.0/24 network, then I could use the following command:

```
# zfs set sharenfs="rw=10.80.86.0/24" tank/share
```

SMB support is also available, although I've had trouble getting it working correctly. As with NFS, it requires a running SMB daemon on the server. To enable SMB sharing on your dataset, use the following command:

```
# zfs set sharesmb=on tank/share
```

iSCSI support is also available with Solaris and the Illumos builds. It is currently not supported for ZFS on Linux, although it is pending, and we will likely see it shortly. To enable iSCSI support, use the following command:

```
# zfs set shareiscsi=on tank/share
```

Dataset Properties

- `compression` and `compressratio`
- `mountpoint`
- `snapdir`
- `sync`

http://pthree.org/?p=2950

ZFS datasets also contain properties, as ZFS pools do. In fact, they have a great deal more. And, there are properties that are available for snapshots that are not available for datasets, and vice versa. A similar command for `zfs(8)` is used to list the properties like with `zpool(8)`. Just use the `zfs get` command. As with `zpool(8)`, you can list all properties, or list them one at a time, comma separated:

```
# zfs get all tank/store1
NAME         PROPERTY              VALUE                  SOURCE
tank/store1  type                  filesystem             -
tank/store1  creation              Fri Feb 22  7:57 2013  -
tank/store1  used                  352K                   -
tank/store1  available             69.7G                  -
tank/store1  referenced            279K                   -
tank/store1  compressratio         1.00x                  -
tank/store1  mounted               yes                    -
tank/store1  quota                 none                   default
tank/store1  reservation           none                   default
tank/store1  recordsize            128K                   default
tank/store1  mountpoint            /tank/store1           default
tank/store1  sharenfs              off                    default
tank/store1  checksum              on                     default
tank/store1  compression           off                    default
tank/store1  atime                 on                     default
tank/store1  devices               on                     default
tank/store1  exec                  on                     default
tank/store1  setuid                on                     default
tank/store1  readonly              off                    default
tank/store1  zoned                 off                    default
tank/store1  snapdir               hidden                 default
tank/store1  aclinherit            restricted             default
tank/store1  canmount              on                     default
tank/store1  xattr                 on                     default
tank/store1  copies                1                      default
tank/store1  version               5                      -
tank/store1  utf8only              off                    -
tank/store1  normalization         none                   -
tank/store1  casesensitivity       sensitive              -
```

```
tank/store1  vscan                off                    default
tank/store1  nbmand               off                    default
tank/store1  sharesmb             off                    default
tank/store1  refquota             none                   default
tank/store1  refreservation       none                   default
tank/store1  primarycache         all                    default
tank/store1  secondarycache       all                    default
tank/store1  usedbysnapshots      72.5K                  -
tank/store1  usedbydataset        279K                   -
tank/store1  usedbychildren       0                      -
tank/store1  usedbyrefreservation 0                      -
tank/store1  logbias              latency                default
tank/store1  dedup                off                    default
tank/store1  mlslabel             none                   default
tank/store1  sync                 standard               default
tank/store1  refcompressratio     1.00x                  -
tank/store1  written              86.4K                  -
```

All of the properties can be found on my blog post above, in the `zfs(8)` manual, or at the official Solaris documentation.

If you wish to see only the `compressratio` and `mountpoint` properties, then you could issue the following command:

```
# zfs get compressratio,mountpoint tank/store1
tank/store1  compressratio  1.00x         -
tank/store1  mountpoint     /tank/store1  default
```

As with the `zpool(8)` command, you can set properties using the `zfs set` command. For example, if I wished to set the compression algorithm to LZJB for the tank/store1 dataset, then I could issue the following command:

```
# zfs set compression=lzjb tank/store1
```

A note about nested filesystems. Technically speaking, `tank/store1` is a nested dataset to `tank`. As such, any properties set on the parent `tank` dataset will get inherited onto the `tank/store1` dataset. Thus, if the LZJB compression algorithm was set on `tank`, then when creating `tank/store1` and `tank/store1/cache`, these datasets would also inherit the LZJB compression algorithm.

Further, if you make a change on a parent dataset, after the nested datasets have already been created, they will automatically get ineherited down as follows:

> # zfs set compression=lzjb tank # zfs get compression tank/store1 NAME PROPERTY VALUE SOURCE tank/store1 compression lzjb inherited from tank # zfs set compression=gzip tank # zfs get compression tank/store1 NAME PROPERTY VALUE SOURCE tank/store1 compression gzip inherited from tank

If you wish to clear a nested dataset to it's parent value, then you can use the `zfs inherit` command.

> # zfs inherit compression tank/store1

Dataset Best Practices and Caveats

- Enable compression by default
- Snapshot frequently and regularly

- With sending snapshots, use incremental sends
- Set `options zfs zfs_arc_max=2147483648` for GNU/Linux

http://pthree.org/?p=2963

Again, as with the ZFS pool best practices and caveats, take this with some weight, but realize that not everything may fit your specific situation. Try to adhere to them as best you can.

- Always enable compression. There is almost certainly no reason to keep it disabled. It hardly touches the CPU and hardly touches throughput to the drive, yet the benefits are amazing.
- Unless you have the RAM, avoid using deduplication. Unlike compression, deduplication is very costly on the system. The deduplication table consumes massive amounts of RAM.
- Avoid running a ZFS root filesystem on GNU/Linux for the time being. It's a bit too experimental for /boot and GRUB. However, do create datasets for /home/, /var/log/ and /var/cache/.
- Snapshot frequently and regularly. Snapshots are cheap, and can keep a plethora of file versions over time. Consider using something like the zfs-auto-snapshot script.
- Snapqhots are not a backup. Use `zfs send` and `zfs receive` to send your ZFS snapshots to an external storage.
- If using NFS, use ZFS NFS rather than your native exports. This can ensure that the dataset is mounted and online before NFS clients begin sending data to the mountpoint.
- Don't mix NFS kernel exports and ZFS NFS exports. This is difficult to administer and maintain.
- For /home/ ZFS installations, setting up nested datasets for each user. For example, pool/home/atoponce and pool/home/dobbs. Consider using quotas on the datasets.
- When using `zfs send` and `zfs receive`, send incremental streams with the `zfs send -i` switch. This can be an exceptional time saver.
- Consider using `zfs send` over `rsync(1)`, as the `zfs send` command can preserve dataset properties.

And the caveats that you should be aware of:

- A `zfs destroy` can cause downtime for other datasets. A `zfs destroy` will touch every file in the dataset that resides in the storage pool. The larger the dataset, the longer this will take, and it will use all the possible IOPS out of your drives to make it happen. Thus, if it take 2 hours to destroy the dataset, that's 2 hours of potential downtime for the other datasets in the pool.
- Debian and Ubuntu will not start the NFS daemon without a valid export in the `/etc/exports` file. You must either modify the `/etc/init.d/nfs` init script to start without an export, or create a local dummy export.
- Debian and Ubuntu, and probably other systems use a parallized boot. As such, init script execution order is no longer prioritized. This creates problems for mounting ZFS datasets on boot. For Debian and Ubuntu, touch the `/etc/init.d/.legacy-bootordering` file, and make sure that the `/etc/init.d/zfs` init script is the first to start, before all other services in that runlevel.
- Do not create ZFS storage pools from files in other ZFS datasets. This will cause all sorts of headaches and problems.
- When creating ZVOLs, make sure to set the block size as the same, or a multiple, of the block size that you will be formatting the ZVOL with. If the block sizes do not align, performance issues could arise.
- When loading the `zfs` kernel module, make sure to set a maximum number for the ARC. Doing a lot of `zfs send` or snapshot operations will cache the data. If not set, RAM will slowly fill until the kernel invokes OOM killer, and the system becomes responsive. I have set in my /etc/modprobe.d/zfs.conf file `options zfs zfs_arc_max=2147483648`, which is a 2 GB limit for

the ARC.