

Chapter 19. The Z File System (ZFS)

Written by Tom Rhodes, Allan Jude, Benedict Reuschling and Warren Block.

The *Z File System*, or ZFS, is an advanced file system designed to overcome many of the major problems found in previous designs.

Originally developed at Sun™, ongoing open source ZFS development has moved to the [OpenZFS Project](#).

ZFS has three major design goals:

- **Data integrity:** All data includes a [checksum](#) of the data. When data is written, the checksum is calculated and written along with it. When that data is later read back, the checksum is calculated again. If the checksums do not match, a data error has been detected. ZFS will attempt to automatically correct errors when data redundancy is available.
- **Pooled storage:** physical storage devices are added to a pool, and storage space is allocated from that shared pool. Space is available to all file systems, and can be increased by adding new storage devices to the pool.
- **Performance:** multiple caching mechanisms provide increased performance. [ARC](#) is an advanced memory-based read cache. A second level of disk-based read cache can be added with [L2ARC](#), and disk-based synchronous write cache is available with [ZIL](#).

A complete list of features and terminology is shown in [Section 19.8, “ZFS Features and Terminology”](#).

19.1. What Makes ZFS Different

ZFS is significantly different from any previous file system because it is more than just a file system. Combining the traditionally separate roles of volume manager and file system provides ZFS with unique advantages. The file system is now aware of the underlying structure of the disks. Traditional file systems could only be created on a single disk at a time. If there were two disks then two separate file systems would have to be created. In a traditional hardware RAID configuration, this problem was avoided by presenting the operating system with a single logical disk made up of the space provided by a number of physical disks, on top of which the operating system placed a file system. Even in the case of software RAID solutions like those provided by GEOM, the UFS file system living on top of the RAID transform believed that it was dealing with a single device. ZFS's combination of the volume manager and the file system solves this and allows the creation of many file systems all sharing a pool of available storage. One of the biggest advantages to ZFS's awareness of the physical layout of the disks is that existing file systems can be grown automatically when additional disks are added to the pool. This new space is then made available to all of the file systems. ZFS also has a number of different properties that can be applied to each file system, giving many advantages to creating a number of different file systems and datasets rather than a single monolithic file system.

19.2. Quick Start Guide

There is a startup mechanism that allows FreeBSD to mount ZFS pools during system initialization. To enable it, add this line to `/etc/rc.conf` :

```
zfs_enable="YES"
```

Then start the service:

```
# service zfs start
```

The examples in this section assume three SCSI disks with the device names `da0`, `da1`, and `da2`. Users of SATA hardware should instead use `ada` device names.

19.2.1. Single Disk Pool

To create a simple, non-redundant pool using a single disk device:

```
# zpool create example /dev/da0
```

To view the new pool, review the output of `df`:

```
# df
Filesystem 1K-blocks    Used   Avail Capacity  Mounted on
/dev/ad0s1a 2026030 235230 1628718    13%    /
devfs        1         1         0   100%  /dev
/dev/ad0s1d 54098308 1032846 48737598     2%   /usr
example     17547136         0 17547136     0%  /example
```

This output shows that the `example` pool has been created and mounted. It is now accessible as a file system. Files can be created on it and users can browse it:

```
# cd /example
# ls
# touch testfile
# ls -al
total 4
drwxr-xr-x  2 root  wheel   3 Aug 29 23:15 .
drwxr-xr-x 21 root  wheel 512 Aug 29 23:12 ..
-rw-r--r--  1 root  wheel   0 Aug 29 23:15 testfile
```

However, this pool is not taking advantage of any ZFS features. To create a dataset on this pool with compression enabled:

```
# zfs create example/compressed
# zfs set compression=gzip example/compressed
```

The `example/compressed` dataset is now a ZFS compressed file system. Try copying some large files to `/example/compressed`.

Compression can be disabled with:

```
# zfs set compression=off example/compressed
```

To unmount a file system, use `zfs unmount` and then verify with `df`:

```
# zfs unmount example/compressed
# df
Filesystem 1K-blocks    Used   Avail Capacity  Mounted on
/dev/ad0s1a 2026030 235232 1628716    13%    /
devfs        1         1         0   100%  /dev
/dev/ad0s1d 54098308 1032864 48737580     2%   /usr
example     17547008         0 17547008     0%  /example
```

To re-mount the file system to make it accessible again, use `zfs mount` and verify with `df`:

```
# zfs mount example/compressed
# df
Filesystem 1K-blocks    Used   Avail Capacity  Mounted on
/dev/ad0s1a 2026030 235234 1628714    13%    /
devfs        1         1         0   100%  /dev
/dev/ad0s1d 54098308 1032864 48737580     2%   /usr
example     17547008         0 17547008     0%  /example
example/compressed 17547008         0 17547008     0%  /example/compressed
```

The pool and file system may also be observed by viewing the output from `mount`:

```
# mount
/dev/ad0s1a on / (ufs, local)
devfs on /dev (devfs, local)
/dev/ad0s1d on /usr (ufs, local, soft-updates)
example on /example (zfs, local)
example/compressed on /example/compressed (zfs, local)
```

After creation, ZFS datasets can be used like any file systems. However, many other features are available which can be set on a per-dataset basis. In the example below, a new file system called `data` is created. Important files will be stored here, so it is configured to keep two copies of each data block:

```
# zfs create example/data
# zfs set copies=2 example/data
```

It is now possible to see the data and space utilization by issuing `df`:

```
# df
Filesystem      1K-blocks    Used   Avail Capacity  Mounted on
/dev/ad0s1a      2026030  235234  1628714    13%     /
devfs              1         1         0   100%   /dev
/dev/ad0s1d    54098308 1032864 48737580     2%   /usr
example        17547008      0 17547008     0%   /example
example/compressed 17547008      0 17547008     0% /example/compressed
example/data    17547008      0 17547008     0% /example/data
```

Notice that each file system on the pool has the same amount of available space. This is the reason for using `df` in these examples, to show that the file systems use only the amount of space they need and all draw from the same pool. ZFS eliminates concepts such as volumes and partitions, and allows multiple file systems to occupy the same pool.

To destroy the file systems and then destroy the pool as it is no longer needed:

```
# zfs destroy example/compressed
# zfs destroy example/data
# zpool destroy example
```

19.2.2. RAID-Z

Disks fail. One method of avoiding data loss from disk failure is to implement RAID. ZFS supports this feature in its pool design. RAID-Z pools require three or more disks but provide more usable space than mirrored pools.

This example creates a RAID-Z pool, specifying the disks to add to the pool:

```
# zpool create storage raidz da0 da1 da2
```



Note

Sun™ recommends that the number of devices used in a RAID-Z configuration be between three and nine. For environments requiring a single pool consisting of 10 disks or more, consider breaking it up into smaller RAID-Z groups. If only two disks are available and redundancy is a requirement, consider using a ZFS mirror. Refer to [zpool\(8\)](#) for more details.

The previous example created the storage zpool. This example makes a new file system called `home` in that pool:

```
# zfs create storage/home
```

Compression and keeping extra copies of directories and files can be enabled:

```
# zfs set copies=2 storage/home
# zfs set compression=gzip storage/home
```

To make this the new home directory for users, copy the user data to this directory and create the appropriate symbolic links:

```
# cp -rp /home/* /storage/home
# rm -rf /home /usr/home
```

```
# ln -s /storage/home /home
# ln -s /storage/home /usr/home
```

Users data is now stored on the freshly-created `/storage/home`. Test by adding a new user and logging in as that user.

Try creating a file system snapshot which can be rolled back later:

```
# zfs snapshot storage/home@08-30-08
```

Snapshots can only be made of a full file system, not a single directory or file.

The `@` character is a delimiter between the file system name or the volume name. If an important directory has been accidentally deleted, the file system can be backed up, then rolled back to an earlier snapshot when the directory still existed:

```
# zfs rollback storage/home@08-30-08
```

To list all available snapshots, run `ls` in the file system's `.zfs/snapshot` directory. For example, to see the previously taken snapshot:

```
# ls /storage/home/.zfs/snapshot
```

It is possible to write a script to perform regular snapshots on user data. However, over time, snapshots can consume a great deal of disk space. The previous snapshot can be removed using the command:

```
# zfs destroy storage/home@08-30-08
```

After testing, `/storage/home` can be made the real `/home` using this command:

```
# zfs set mountpoint=/home storage/home
```

Run `df` and `mount` to confirm that the system now treats the file system as the real `/home`:

```
# mount
/dev/ad0s1a on / (ufs, local)
devfs on /dev (devfs, local)
/dev/ad0s1d on /usr (ufs, local, soft-updates)
storage on /storage (zfs, local)
storage/home on /home (zfs, local)
# df
Filesystem      1K-blocks    Used   Avail Capacity  Mounted on
/dev/ad0s1a      2026030  235240  1628708    13%      /
devfs              1         1         0    100%    /dev
/dev/ad0s1d    54098308 1032826 48737618     2%    /usr
storage         26320512      0  26320512     0%    /storage
storage/home    26320512      0  26320512     0%    /home
```

This completes the RAID-Z configuration. Daily status updates about the file systems created can be generated as part of the nightly [periodic\(8\)](#) runs. Add this line to `/etc/periodic.conf`:

```
daily_status_zfs_enable="YES"
```

19.2.3. Recovering RAID-Z

Every software RAID has a method of monitoring its state. The status of RAID-Z devices may be viewed with this command:

```
# zpool status -x
```

If all pools are [Online](#) and everything is normal, the message shows:

```
all pools are healthy
```

If there is an issue, perhaps a disk is in the [Offline](#) state, the pool state will look similar to:

```
pool: storage
state: DEGRADED
status: One or more devices has been taken offline by the administrator.
Sufficient replicas exist for the pool to continue functioning in a
degraded state.
action: Online the device using 'zpool online' or replace the device with
'zpool replace'.
scrub: none requested
config:

NAME      STATE      READ WRITE CKSUM
storage   DEGRADED   0     0     0
raidz1    DEGRADED   0     0     0
da0       ONLINE    0     0     0
da1       OFFLINE    0     0     0
da2       ONLINE    0     0     0

errors: No known data errors
```

This indicates that the device was previously taken offline by the administrator with this command:

```
# zpool offline storage da1
```

Now the system can be powered down to replace da1. When the system is back online, the failed disk can be replaced in the pool:

```
# zpool replace storage da1
```

From here, the status may be checked again, this time without -x so that all pools are shown:

```
# zpool status storage
pool: storage
state: ONLINE
scrub: resilver completed with 0 errors on Sat Aug 30 19:44:11 2008
config:

NAME      STATE      READ WRITE CKSUM
storage   ONLINE    0     0     0
raidz1    ONLINE    0     0     0
da0       ONLINE    0     0     0
da1       ONLINE    0     0     0
da2       ONLINE    0     0     0

errors: No known data errors
```

In this example, everything is normal.

19.2.4. Data Verification

ZFS uses checksums to verify the integrity of stored data. These are enabled automatically upon creation of file systems.



Warning

Checksums can be disabled, but it is *not* recommended! Checksums take very little storage space and provide data integrity. Many ZFS features will not work properly with checksums disabled. There is no noticeable performance gain from disabling these checksums.

Checksum verification is known as *scrubbing*. Verify the data integrity of the storage pool with this command:

```
# zpool scrub storage
```

The duration of a scrub depends on the amount of data stored. Larger amounts of data will take proportionally longer to verify. Scrubs are very I/O intensive, and only one scrub is allowed to run at a time. After the scrub completes, the status can be viewed with `status`:

```
# zpool status storage
pool: storage
state: ONLINE
scrub: scrub completed with 0 errors on Sat Jan 26 19:57:37 2013
config:

NAME        STATE      READ WRITE CKSUM
storage     ONLINE     0     0     0
  raidz1    ONLINE     0     0     0
    da0     ONLINE     0     0     0
    da1     ONLINE     0     0     0
    da2     ONLINE     0     0     0

errors: No known data errors
```

The completion date of the last scrub operation is displayed to help track when another scrub is required. Routine scrubs help protect data from silent corruption and ensure the integrity of the pool.

Refer to [zfs\(8\)](#) and [zpool\(8\)](#) for other ZFS options.

19.3. zpool Administration

ZFS administration is divided between two main utilities. The `zpool` utility controls the operation of the pool and deals with adding, removing, replacing, and managing disks. The `zfs` utility deals with creating, destroying, and managing datasets, both [file systems](#) and [volumes](#).

19.3.1. Creating and Destroying Storage Pools

Creating a ZFS storage pool (*zpool*) involves making a number of decisions that are relatively permanent because the structure of the pool cannot be changed after the pool has been created. The most important decision is what types of vdevs into which to group the physical disks. See the list of [vdev types](#) for details about the possible options. After the pool has been created, most vdev types do not allow additional disks to be added to the vdev. The exceptions are mirrors, which allow additional disks to be added to the vdev, and stripes, which can be upgraded to mirrors by attaching an additional disk to the vdev. Although additional vdevs can be added to expand a pool, the layout of the pool cannot be changed after pool creation. Instead, the data must be backed up and the pool destroyed and recreated.

Create a simple mirror pool:

```
# zpool create mypool mirror /dev/ada1 /dev/ada2
# zpool status
pool: mypool
state: ONLINE
scan: none requested
config:

NAME        STATE      READ WRITE CKSUM
mypool      ONLINE     0     0     0
  mirror-0  ONLINE     0     0     0
    ada1    ONLINE     0     0     0
    ada2    ONLINE     0     0     0

errors: No known data errors
```

Multiple vdevs can be created at once. Specify multiple groups of disks separated by the vdev type keyword, `mirror` in this example:

```
# zpool create mypool mirror /dev/ada1 /dev/ada2 mirror /dev/ada3 /dev/ada4
pool: mypool
state: ONLINE
scan: none requested
config:

    NAME      STATE    READ WRITE CKSUM
    mypool    ONLINE      0     0     0
      mirror-0 ONLINE      0     0     0
        ada1  ONLINE      0     0     0
        ada2  ONLINE      0     0     0
      mirror-1 ONLINE      0     0     0
        ada3  ONLINE      0     0     0
        ada4  ONLINE      0     0     0

errors: No known data errors
```

Pools can also be constructed using partitions rather than whole disks. Putting ZFS in a separate partition allows the same disk to have other partitions for other purposes. In particular, partitions with bootcode and file systems needed for booting can be added. This allows booting from disks that are also members of a pool. There is no performance penalty on FreeBSD when using a partition rather than a whole disk. Using partitions also allows the administrator to *under-provision* the disks, using less than the full capacity. If a future replacement disk of the same nominal size as the original actually has a slightly smaller capacity, the smaller partition will still fit, and the replacement disk can still be used.

Create a [RAID-Z2 \[395\]](#) pool using partitions:

```
# zpool create mypool raidz2 /dev/ada0p3 /dev/ada1p3 /dev/ada2p3 /dev/ada3p3 /dev/ada4p3 /dev/ada5p3
# zpool status
pool: mypool
state: ONLINE
scan: none requested
config:

    NAME      STATE    READ WRITE CKSUM
    mypool    ONLINE      0     0     0
      raidz2-0 ONLINE      0     0     0
        ada0p3 ONLINE      0     0     0
        ada1p3 ONLINE      0     0     0
        ada2p3 ONLINE      0     0     0
        ada3p3 ONLINE      0     0     0
        ada4p3 ONLINE      0     0     0
        ada5p3 ONLINE      0     0     0

errors: No known data errors
```

A pool that is no longer needed can be destroyed so that the disks can be reused. Destroying a pool involves first unmounting all of the datasets in that pool. If the datasets are in use, the unmount operation will fail and the pool will not be destroyed. The destruction of the pool can be forced with `-f`, but this can cause undefined behavior in applications which had open files on those datasets.

19.3.2. Adding and Removing Devices

There are two cases for adding disks to a zpool: attaching a disk to an existing vdev with `zpool attach`, or adding vdevs to the pool with `zpool add`. Only some [vdev types](#) allow disks to be added to the vdev after creation.

A pool created with a single disk lacks redundancy. Corruption can be detected but not repaired, because there is no other copy of the data. The [copies](#) property may be able to recover from a small failure such as a bad sector, but does not provide the same level of protection as mirroring or RAID-Z. Starting with a pool consisting of a single

disk vdev, `zpool attach` can be used to add an additional disk to the vdev, creating a mirror. `zpool attach` can also be used to add additional disks to a mirror group, increasing redundancy and read performance. If the disks being used for the pool are partitioned, replicate the layout of the first disk on to the second, `gpart backup` and `gpart restore` can be used to make this process easier.

Upgrade the single disk (stripe) vdev `ada0p3` to a mirror by attaching `ada1p3`:

```
# zpool status
pool: mypool
state: ONLINE
scan: none requested
config:

    NAME      STATE    READ WRITE CKSUM
    mypool    ONLINE      0     0     0
      ada0p3  ONLINE      0     0     0

errors: No known data errors
# zpool attach mypool ada0p3 ada1p3
Make sure to wait until resilver is done before rebooting.

If you boot from pool 'mypool', you may need to update
boot code on newly attached disk 'ada1p3'.

Assuming you use GPT partitioning and 'da0' is your new boot disk
you may use the following command:

    gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1 da0
# gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1 ada1
bootcode written to ada1
# zpool status
pool: mypool
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.
scan: resilver in progress since Fri May 30 08:19:19 2014
      527M scanned out of 781M at 47.9M/s, 0h0m to go
      527M resilvered, 67.53% done
config:

    NAME      STATE    READ WRITE CKSUM
    mypool    ONLINE      0     0     0
      mirror-0 ONLINE      0     0     0
        ada0p3 ONLINE      0     0     0
        ada1p3 ONLINE      0     0     0 (resilvering)

errors: No known data errors
# zpool status
pool: mypool
state: ONLINE
scan: resilvered 781M in 0h0m with 0 errors on Fri May 30 08:15:58 2014
config:

    NAME      STATE    READ WRITE CKSUM
    mypool    ONLINE      0     0     0
      mirror-0 ONLINE      0     0     0
        ada0p3 ONLINE      0     0     0
        ada1p3 ONLINE      0     0     0

errors: No known data errors
```

When adding disks to the existing vdev is not an option, as for RAID-Z, an alternative method is to add another vdev to the pool. Additional vdevs provide higher performance, distributing writes across the vdevs. Each vdev is responsible for providing its own redundancy. It is possible, but discouraged, to mix vdev types, like mirror and

RAID-Z. Adding a non-redundant vdev to a pool containing mirror or RAID-Z vdevs risks the data on the entire pool. Writes are distributed, so the failure of the non-redundant disk will result in the loss of a fraction of every block that has been written to the pool.

Data is striped across each of the vdevs. For example, with two mirror vdevs, this is effectively a RAID 10 that stripes writes across two sets of mirrors. Space is allocated so that each vdev reaches 100% full at the same time. There is a performance penalty if the vdevs have different amounts of free space, as a disproportionate amount of the data is written to the less full vdev.

When attaching additional devices to a boot pool, remember to update the bootcode.

Attach a second mirror group (*ada2p3* and *ada3p3*) to the existing mirror:

```
# zpool status
pool: mypool
state: ONLINE
scan: resilvered 781M in 0h0m with 0 errors on Fri May 30 08:19:35 2014
config:

    NAME        STATE        READ WRITE CKSUM
    mypool      ONLINE         0     0     0
      mirror-0  ONLINE         0     0     0
        ada0p3  ONLINE         0     0     0
        ada1p3  ONLINE         0     0     0

errors: No known data errors
# zpool add mypool mirror ada2p3 ada3p3
# gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1  ada2
bootcode written to ada2
# gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1  ada3
bootcode written to ada3
# zpool status
pool: mypool
state: ONLINE
scan: scrub repaired 0 in 0h0m with 0 errors on Fri May 30 08:29:51 2014
config:

    NAME        STATE        READ WRITE CKSUM
    mypool      ONLINE         0     0     0
      mirror-0  ONLINE         0     0     0
        ada0p3  ONLINE         0     0     0
        ada1p3  ONLINE         0     0     0
      mirror-1  ONLINE         0     0     0
        ada2p3  ONLINE         0     0     0
        ada3p3  ONLINE         0     0     0

errors: No known data errors
```

Currently, vdevs cannot be removed from a pool, and disks can only be removed from a mirror if there is enough remaining redundancy. If only one disk in a mirror group remains, it ceases to be a mirror and reverts to being a stripe, risking the entire pool if that remaining disk fails.

Remove a disk from a three-way mirror group:

```
# zpool status
pool: mypool
state: ONLINE
scan: scrub repaired 0 in 0h0m with 0 errors on Fri May 30 08:29:51 2014
config:

    NAME        STATE        READ WRITE CKSUM
    mypool      ONLINE         0     0     0
      mirror-0  ONLINE         0     0     0
        ada0p3  ONLINE         0     0     0
        ada1p3  ONLINE         0     0     0
```

```

      ada2p3  ONLINE          0      0      0
errors: No known data errors
# zpool detach mypool ada2p3
# zpool status
  pool: mypool
  state: ONLINE
    scan: scrub repaired 0 in 0h0m with 0 errors on Fri May 30 08:29:51 2014
config:

    NAME      STATE    READ WRITE CKSUM
    mypool     ONLINE      0     0     0
      mirror-0  ONLINE      0     0     0
        ada0p3  ONLINE      0     0     0
        ada1p3  ONLINE      0     0     0
errors: No known data errors

```

19.3.3. Checking the Status of a Pool

Pool status is important. If a drive goes offline or a read, write, or checksum error is detected, the corresponding error count increases. The status output shows the configuration and status of each device in the pool and the status of the entire pool. Actions that need to be taken and details about the last [scrub](#) are also shown.

```

# zpool status
  pool: mypool
  state: ONLINE
    scan: scrub repaired 0 in 2h25m with 0 errors on Sat Sep 14 04:25:50 2013
config:

    NAME      STATE    READ WRITE CKSUM
    mypool     ONLINE      0     0     0
      raidz2-0  ONLINE      0     0     0
        ada0p3  ONLINE      0     0     0
        ada1p3  ONLINE      0     0     0
        ada2p3  ONLINE      0     0     0
        ada3p3  ONLINE      0     0     0
        ada4p3  ONLINE      0     0     0
        ada5p3  ONLINE      0     0     0
errors: No known data errors

```

19.3.4. Clearing Errors

When an error is detected, the read, write, or checksum counts are incremented. The error message can be cleared and the counts reset with `zpool clear mypool`. Clearing the error state can be important for automated scripts that alert the administrator when the pool encounters an error. Further errors may not be reported if the old errors are not cleared.

19.3.5. Replacing a Functioning Device

There are a number of situations where it may be desirable to replace one disk with a different disk. When replacing a working disk, the process keeps the old disk online during the replacement. The pool never enters a [degraded](#) state, reducing the risk of data loss. `zpool replace` copies all of the data from the old disk to the new one. After the operation completes, the old disk is disconnected from the vdev. If the new disk is larger than the old disk, it may be possible to grow the zpool, using the new space. See [Growing a Pool](#).

Replace a functioning device in the pool:

```

# zpool status
  pool: mypool
  state: ONLINE
    scan: none requested

```

```

config:

NAME          STATE      READ WRITE CKSUM
mypool        ONLINE        0     0     0
  mirror-0    ONLINE        0     0     0
    ada0p3    ONLINE        0     0     0
    ada1p3    ONLINE        0     0     0

errors: No known data errors
# zpool replace mypool ada1p3 ada2p3
Make sure to wait until resilver is done before rebooting.

If you boot from pool 'zroot', you may need to update
boot code on newly attached disk 'ada2p3'.

Assuming you use GPT partitioning and 'da0' is your new boot disk
you may use the following command:

    gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1 da0
# gpart bootcode -b /boot/pmbr -p /boot/gptzfsboot -i 1 ada2
# zpool status
pool: mypool
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.
scan: resilver in progress since Mon Jun  2 14:21:35 2014
      604M scanned out of 781M at 46.5M/s, 0h0m to go
      604M resilvered, 77.39% done
config:

NAME          STATE      READ WRITE CKSUM
mypool        ONLINE        0     0     0
  mirror-0    ONLINE        0     0     0
    ada0p3    ONLINE        0     0     0
    replacing-1 ONLINE        0     0     0
      ada1p3    ONLINE        0     0     0
      ada2p3    ONLINE        0     0     0 (resilvering)

errors: No known data errors
# zpool status
pool: mypool
state: ONLINE
scan: resilvered 781M in 0h0m with 0 errors on Mon Jun  2 14:21:52 2014
config:

NAME          STATE      READ WRITE CKSUM
mypool        ONLINE        0     0     0
  mirror-0    ONLINE        0     0     0
    ada0p3    ONLINE        0     0     0
    ada2p3    ONLINE        0     0     0

errors: No known data errors

```

19.3.6. Dealing with Failed Devices

When a disk in a pool fails, the vdev to which the disk belongs enters the [degraded](#) state. All of the data is still available, but performance may be reduced because missing data must be calculated from the available redundancy. To restore the vdev to a fully functional state, the failed physical device must be replaced. ZFS is then instructed to begin the [resilver](#) operation. Data that was on the failed device is recalculated from available redundancy and written to the replacement device. After completion, the vdev returns to [online](#) status.

If the vdev does not have any redundancy, or if multiple devices have failed and there is not enough redundancy to compensate, the pool enters the [faulted](#) state. If a sufficient number of devices cannot be reconnected to the pool, the pool becomes inoperative and data must be restored from backups.

When replacing a failed disk, the name of the failed disk is replaced with the GUID of the device. A new device name parameter for `zpool replace` is not required if the replacement device has the same device name.

Replace a failed disk using `zpool replace`:

```
# zpool status
pool: mypool
state: DEGRADED
status: One or more devices could not be opened. Sufficient replicas exist for
the pool to continue functioning in a degraded state.
action: Attach the missing device and online it using 'zpool online'.
see: http://illumos.org/msg/ZFS-8000-2Q
scan: none requested
config:

    NAME                STATE      READ WRITE CKSUM
    mypool               DEGRADED   0     0     0
      mirror-0          DEGRADED   0     0     0
        ada0p3          ONLINE     0     0     0
        316502962686821739 UNAVAIL     0     0     0 was /dev/ada1p3

errors: No known data errors
# zpool replace mypool 316502962686821739 ada2p3
# zpool status
pool: mypool
state: DEGRADED
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.
scan: resilver in progress since Mon Jun  2 14:52:21 2014
      641M scanned out of 781M at 49.3M/s, 0h0m to go
      640M resilvered, 82.04% done
config:

    NAME                STATE      READ WRITE CKSUM
    mypool               DEGRADED   0     0     0
      mirror-0          DEGRADED   0     0     0
        ada0p3          ONLINE     0     0     0
        replacing-1     UNAVAIL     0     0     0
        15732067398082357289 UNAVAIL     0     0     0 was /dev/ada1p3/old
        ada2p3          ONLINE     0     0     0 (resilvering)

errors: No known data errors
# zpool status
pool: mypool
state: ONLINE
scan: resilvered 781M in 0h0m with 0 errors on Mon Jun  2 14:52:38 2014
config:

    NAME                STATE      READ WRITE CKSUM
    mypool               ONLINE     0     0     0
      mirror-0          ONLINE     0     0     0
        ada0p3          ONLINE     0     0     0
        ada2p3          ONLINE     0     0     0

errors: No known data errors
```

19.3.7. Scrubbing a Pool

It is recommended that pools be [scrubbed](#) regularly, ideally at least once every month. The scrub operation is very disk-intensive and will reduce performance while running. Avoid high-demand periods when scheduling scrub or use `vfs.zfs.scrub_delay` [\[392\]](#) to adjust the relative priority of the `scrub` to prevent it interfering with other workloads.

```
# zpool scrub mypool
```

```
# zpool status
pool: mypool
state: ONLINE
scan: scrub in progress since Wed Feb 19 20:52:54 2014
      116G scanned out of 8.60T at 649M/s, 3h48m to go
      0 repaired, 1.32% done
config:

    NAME      STATE    READ WRITE CKSUM
    mypool    ONLINE      0     0     0
      raidz2-0 ONLINE      0     0     0
        ada0p3 ONLINE      0     0     0
        ada1p3 ONLINE      0     0     0
        ada2p3 ONLINE      0     0     0
        ada3p3 ONLINE      0     0     0
        ada4p3 ONLINE      0     0     0
        ada5p3 ONLINE      0     0     0

errors: No known data errors
```

In the event that a scrub operation needs to be cancelled, issue `zpool scrub -s mypool`.

19.3.8. Self-Healing

The checksums stored with data blocks enable the file system to *self-heal*. This feature will automatically repair data whose checksum does not match the one recorded on another device that is part of the storage pool. For example, a mirror with two disks where one drive is starting to malfunction and cannot properly store the data any more. This is even worse when the data has not been accessed for a long time, as with long term archive storage. Traditional file systems need to run algorithms that check and repair the data like `fsck(8)`. These commands take time, and in severe cases, an administrator has to manually decide which repair operation must be performed. When ZFS detects a data block with a checksum that does not match, it tries to read the data from the mirror disk. If that disk can provide the correct data, it will not only give that data to the application requesting it, but also correct the wrong data on the disk that had the bad checksum. This happens without any interaction from a system administrator during normal pool operation.

The next example demonstrates this self-healing behavior. A mirrored pool of disks `/dev/ada0` and `/dev/ada1` is created.

```
# zpool create healer mirror /dev/ada0 /dev/ada1
# zpool status healer
pool: healer
state: ONLINE
scan: none requested
config:

    NAME      STATE    READ WRITE CKSUM
    healer    ONLINE      0     0     0
      mirror-0 ONLINE      0     0     0
        ada0   ONLINE      0     0     0
        ada1   ONLINE      0     0     0

errors: No known data errors
# zpool list
NAME      SIZE  ALLOC   FREE   CAP  DEDUP  HEALTH  ALROOT
healer    960M  92.5K   960M    0%  1.00x  ONLINE  -
```

Some important data that to be protected from data errors using the self-healing feature is copied to the pool. A checksum of the pool is created for later comparison.

```
# cp /some/important/data /healer
# zfs list
NAME      SIZE  ALLOC   FREE   CAP  DEDUP  HEALTH  ALROOT
healer    960M  67.7M   892M    7%  1.00x  ONLINE  -
```

```
# sha1 /healer > checksum.txt
# cat checksum.txt
SHA1 (/healer) = 2753eff56d77d9a536ece6694bf0a82740344d1f
```

Data corruption is simulated by writing random data to the beginning of one of the disks in the mirror. To prevent ZFS from healing the data as soon as it is detected, the pool is exported before the corruption and imported again afterwards.



Warning

This is a dangerous operation that can destroy vital data. It is shown here for demonstrational purposes only and should not be attempted during normal operation of a storage pool. Nor should this intentional corruption example be run on any disk with a different file system on it. Do not use any other disk device names other than the ones that are part of the pool. Make certain that proper backups of the pool are created before running the command!

```
# zpool export healer
# dd if=/dev/random of=/dev/ada1 bs=1m count=200
200+0 records in
200+0 records out
209715200 bytes transferred in 62.992162 secs (3329227 bytes/sec)
# zpool import healer
```

The pool status shows that one device has experienced an error. Note that applications reading data from the pool did not receive any incorrect data. ZFS provided data from the `ada0` device with the correct checksums. The device with the wrong checksum can be found easily as the `CKSUM` column contains a nonzero value.

```
# zpool status healer
pool: healer
state: ONLINE
status: One or more devices has experienced an unrecoverable error. An
attempt was made to correct the error. Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
using 'zpool clear' or replace the device with 'zpool replace'.
see: http://www.sun.com/msg/ZFS-8000-9P
scan: none requested
config:

    NAME      STATE    READ WRITE CKSUM
    healer    ONLINE    0     0     0
      mirror-0 ONLINE    0     0     0
        ada0   ONLINE    0     0     0
        ada1   ONLINE    0     0     1

errors: No known data errors
```

The error was detected and handled by using the redundancy present in the unaffected `ada0` mirror disk. A checksum comparison with the original one will reveal whether the pool is consistent again.

```
# sha1 /healer >> checksum.txt
# cat checksum.txt
SHA1 (/healer) = 2753eff56d77d9a536ece6694bf0a82740344d1f
SHA1 (/healer) = 2753eff56d77d9a536ece6694bf0a82740344d1f
```

The two checksums that were generated before and after the intentional tampering with the pool data still match. This shows how ZFS is capable of detecting and correcting any errors automatically when the checksums differ. Note that this is only possible when there is enough redundancy present in the pool. A pool consisting of a single device has no self-healing capabilities. That is also the reason why checksums are so important in ZFS and should not be disabled for any reason. No `fsck(8)` or similar file system consistency check program is required to detect

and correct this and the pool was still available during the time there was a problem. A scrub operation is now required to overwrite the corrupted data on `ada1`.

```
# zpool scrub healer
# zpool status healer
pool: healer
state: ONLINE
status: One or more devices has experienced an unrecoverable error. An
       attempt was made to correct the error. Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
       using 'zpool clear' or replace the device with 'zpool replace'.
       see: http://www.sun.com/msg/ZFS-8000-9P
scan: scrub in progress since Mon Dec 10 12:23:30 2012
      10.4M scanned out of 67.0M at 267K/s, 0h3m to go
      9.63M repaired, 15.56% done
config:

    NAME      STATE    READ WRITE CKSUM
    healer    ONLINE      0     0     0
    mirror-0   ONLINE      0     0     0
      ada0    ONLINE      0     0     0
      ada1    ONLINE      0     0    627  (repairing)

errors: No known data errors
```

The scrub operation reads data from `ada0` and rewrites any data with an incorrect checksum on `ada1`. This is indicated by the (repairing) output from `zpool status`. After the operation is complete, the pool status changes to:

```
# zpool status healer
pool: healer
state: ONLINE
status: One or more devices has experienced an unrecoverable error. An
       attempt was made to correct the error. Applications are unaffected.
action: Determine if the device needs to be replaced, and clear the errors
       using 'zpool clear' or replace the device with 'zpool replace'.
       see: http://www.sun.com/msg/ZFS-8000-9P
scan: scrub repaired 66.5M in 0h2m with 0 errors on Mon Dec 10 12:26:25 2012
config:

    NAME      STATE    READ WRITE CKSUM
    healer    ONLINE      0     0     0
    mirror-0   ONLINE      0     0     0
      ada0    ONLINE      0     0     0
      ada1    ONLINE      0     0  2.72K

errors: No known data errors
```

After the scrub operation completes and all the data has been synchronized from `ada0` to `ada1`, the error messages can be **cleared** from the pool status by running `zpool clear`.

```
# zpool clear healer
# zpool status healer
pool: healer
state: ONLINE
scan: scrub repaired 66.5M in 0h2m with 0 errors on Mon Dec 10 12:26:25 2012
config:

    NAME      STATE    READ WRITE CKSUM
    healer    ONLINE      0     0     0
    mirror-0   ONLINE      0     0     0
      ada0    ONLINE      0     0     0
      ada1    ONLINE      0     0     0

errors: No known data errors
```

The pool is now back to a fully working state and all the errors have been cleared.

19.3.9. Growing a Pool

The usable size of a redundant pool is limited by the capacity of the smallest device in each vdev. The smallest device can be replaced with a larger device. After completing a [replace](#) or [resilver](#) operation, the pool can grow to use the capacity of the new device. For example, consider a mirror of a 1 TB drive and a 2 TB drive. The usable space is 1 TB. When the 1 TB drive is replaced with another 2 TB drive, the resilvering process copies the existing data onto the new drive. Because both of the devices now have 2 TB capacity, the mirror's available space can be grown to 2 TB.

Expansion is triggered by using `zpool online -e` on each device. After expansion of all devices, the additional space becomes available to the pool.

19.3.10. Importing and Exporting Pools

Pools are *exported* before moving them to another system. All datasets are unmounted, and each device is marked as exported but still locked so it cannot be used by other disk subsystems. This allows pools to be *imported* on other machines, other operating systems that support ZFS, and even different hardware architectures (with some caveats, see [zpool\(8\)](#)). When a dataset has open files, `zpool export -f` can be used to force the export of a pool. Use this with caution. The datasets are forcibly unmounted, potentially resulting in unexpected behavior by the applications which had open files on those datasets.

Export a pool that is not in use:

```
# zpool export mypool
```

Importing a pool automatically mounts the datasets. This may not be the desired behavior, and can be prevented with `zpool import -N`. `zpool import -o` sets temporary properties for this import only. `zpool import altroot=` allows importing a pool with a base mount point instead of the root of the file system. If the pool was last used on a different system and was not properly exported, an import might have to be forced with `zpool import -f`. `zpool import -a` imports all pools that do not appear to be in use by another system.

List all available pools for import:

```
# zpool import
pool: mypool
id: 9930174748043525076
state: ONLINE
action: The pool can be imported using its name or numeric identifier.
config:

    mypool      ONLINE
    ada2p3      ONLINE
```

Import the pool with an alternative root directory:

```
# zpool import -o altroot= /mnt mypool
# zfs list
zfs list
NAME                USED  AVAIL  REFER  MOUNTPOINT
mypool              110K  47.0G   31K    /mnt/mypool
```

19.3.11. Upgrading a Storage Pool

After upgrading FreeBSD, or if a pool has been imported from a system using an older version of ZFS, the pool can be manually upgraded to the latest version of ZFS to support newer features. Consider whether the pool may ever need to be imported on an older system before upgrading. Upgrading is a one-way process. Older pools can be upgraded, but pools with newer features cannot be downgraded.

Upgrade a v28 pool to support Feature Flags:

```
# zpool status
pool: mypool
state: ONLINE
status: The pool is formatted using a legacy on-disk format. The pool can
still be used, but some features are unavailable.
action: Upgrade the pool using 'zpool upgrade'. Once this is done, the
pool will no longer be accessible on software that does not support feat
flags.
scan: none requested
config:

    NAME      STATE    READ WRITE CKSUM
    mypool     ONLINE   0     0     0
      mirror-0 ONLINE   0     0     0
    ada0      ONLINE   0     0     0
    ada1      ONLINE   0     0     0

errors: No known data errors
# zpool upgrade
This system supports ZFS pool feature flags.

The following pools are formatted with legacy version numbers and can
be upgraded to use feature flags. After being upgraded, these pools
will no longer be accessible by software that does not support feature
flags.

VER  POOL
---  -----
28   mypool

Use 'zpool upgrade -v' for a list of available legacy versions.
Every feature flags pool has all supported features enabled.
# zpool upgrade mypool
This system supports ZFS pool feature flags.

Successfully upgraded 'mypool' from version 28 to feature flags.
Enabled the following features on 'mypool':
  async_destroy
  empty_bpobj
  lz4_compress
  multi_vdev_crash_dump
```

The newer features of ZFS will not be available until `zpool upgrade` has completed. `zpool upgrade -v` can be used to see what new features will be provided by upgrading, as well as which features are already supported.

Upgrade a pool to support additional feature flags:

```
# zpool status
pool: mypool
state: ONLINE
status: Some supported features are not enabled on the pool. The pool can
still be used, but some features are unavailable.
action: Enable all features using 'zpool upgrade'. Once this is done,
the pool may no longer be accessible by software that does not support
the features. See zpool-features(7) for details.
scan: none requested
config:

    NAME      STATE    READ WRITE CKSUM
    mypool     ONLINE   0     0     0
      mirror-0 ONLINE   0     0     0
    ada0      ONLINE   0     0     0
    ada1      ONLINE   0     0     0
```

```

errors: No known data errors
# zpool upgrade
This system supports ZFS pool feature flags.

All pools are formatted using feature flags.

Some supported features are not enabled on the following pools. Once a
feature is enabled the pool may become incompatible with software
that does not support the feature. See zpool-features(7) for details.

POOL  FEATURE
-----
zstore
    multi_vdev_crash_dump
    spacemap_histogram
    enabled_txg
    hole_birth
    extensible_dataset
    bookmarks
    filesystem_limits
# zpool upgrade mypool
This system supports ZFS pool feature flags.

Enabled the following features on 'mypool':
    spacemap_histogram
    enabled_txg
    hole_birth
    extensible_dataset
    bookmarks
    filesystem_limits

```



Warning

The boot code on systems that boot from a pool must be updated to support the new pool version. Use `gpart bootcode` on the partition that contains the boot code. See [gpart\(8\)](#) for more information.

19.3.12. Displaying Recorded Pool History

Commands that modify the pool are recorded. Recorded actions include the creation of datasets, changing properties, or replacement of a disk. This history is useful for reviewing how a pool was created and which user performed a specific action and when. History is not kept in a log file, but is part of the pool itself. The command to review this history is aptly named `zpool history`:

```

# zpool history
History for 'tank':
2013-02-26.23:02:35 zpool create tank mirror /dev/ada0 /dev/ada1
2013-02-27.18:50:58 zfs set atime=off tank
2013-02-27.18:51:09 zfs set checksum=fletcher4 tank
2013-02-27.18:51:18 zfs create tank/backup

```

The output shows `zpool` and `zfs` commands that were executed on the pool along with a timestamp. Only commands that alter the pool in some way are recorded. Commands like `zfs list` are not included. When no pool name is specified, the history of all pools is displayed.

`zpool history` can show even more information when the options `-i` or `-l` are provided. `-i` displays user-initiated events as well as internally logged ZFS events.

```
# zpool history -i
```

```
History for 'tank':
2013-02-26.23:02:35 [internal pool create txg:5] pool spa 28; zfs spa 28; zpl 5;uts 9.1-
RELEASE 901000 amd64
2013-02-27.18:50:53 [internal property set txg:50] atime=0 dataset = 21
2013-02-27.18:50:58 zfs set atime=off tank
2013-02-27.18:51:04 [internal property set txg:53] checksum=7 dataset = 21
2013-02-27.18:51:09 zfs set checksum=fletcher4 tank
2013-02-27.18:51:13 [internal create txg:55] dataset = 39
2013-02-27.18:51:18 zfs create tank/backup
```

More details can be shown by adding `-l`. History records are shown in a long format, including information like the name of the user who issued the command and the hostname on which the change was made.

```
# zpool history -l
History for 'tank':
2013-02-26.23:02:35 zpool create tank mirror /dev/ada0 /dev/ada1 [user 0 (root) on
myzfsbox:global]
2013-02-27.18:50:58 zfs set atime=off tank [user 0 (root) on myzfsbox:global]
2013-02-27.18:51:09 zfs set checksum=fletcher4 tank [user 0 (root) on myzfsbox:global]
2013-02-27.18:51:18 zfs create tank/backup [user 0 (root) on myzfsbox:global]
```

The output shows that the `root` user created the mirrored pool with disks `/dev/ada0` and `/dev/ada1`. The hostname `myzfsbox` is also shown in the commands after the pool's creation. The hostname display becomes important when the pool is exported from one system and imported on another. The commands that are issued on the other system can clearly be distinguished by the hostname that is recorded for each command.

Both options to `zpool history` can be combined to give the most detailed information possible for any given pool. Pool history provides valuable information when tracking down the actions that were performed or when more detailed output is needed for debugging.

19.3.13. Performance Monitoring

A built-in monitoring system can display pool I/O statistics in real time. It shows the amount of free and used space on the pool, how many read and write operations are being performed per second, and how much I/O bandwidth is currently being utilized. By default, all pools in the system are monitored and displayed. A pool name can be provided to limit monitoring to just that pool. A basic example:

```
# zpool iostat
pool          capacity      operations      bandwidth
  alloc  free   read  write   read  write
-----
data         288G  1.53T      2    11   11.3K   57.1K
```

To continuously monitor I/O activity, a number can be specified as the last parameter, indicating a interval in seconds to wait between updates. The next statistic line is printed after each interval. Press `Ctrl+C` to stop this continuous monitoring. Alternatively, give a second number on the command line after the interval to specify the total number of statistics to display.

Even more detailed I/O statistics can be displayed with `-v`. Each device in the pool is shown with a statistics line. This is useful in seeing how many read and write operations are being performed on each device, and can help determine if any individual device is slowing down the pool. This example shows a mirrored pool with two devices:

```
# zpool iostat -v
pool          capacity      operations      bandwidth
  alloc  free   read  write   read  write
-----
data         288G  1.53T      2    12    9.23K   61.5K
  mirror         288G  1.53T      2    12    9.23K   61.5K
    ada1           -      -      0     4    5.61K   61.7K
    ada2           -      -      1     4    5.04K   61.7K
```

19.3.14. Splitting a Storage Pool

A pool consisting of one or more mirror vdevs can be split into two pools. Unless otherwise specified, the last member of each mirror is detached and used to create a new pool containing the same data. The operation should first be attempted with `-n`. The details of the proposed operation are displayed without it actually being performed. This helps confirm that the operation will do what the user intends.

19.4. `zfs` Administration

The `zfs` utility is responsible for creating, destroying, and managing all ZFS datasets that exist within a pool. The pool is managed using `zpool`.

19.4.1. Creating and Destroying Datasets

Unlike traditional disks and volume managers, space in ZFS is *not* preallocated. With traditional file systems, after all of the space is partitioned and assigned, there is no way to add an additional file system without adding a new disk. With ZFS, new file systems can be created at any time. Each [dataset](#) has properties including features like compression, deduplication, caching, and quotas, as well as other useful properties like readonly, case sensitivity, network file sharing, and a mount point. Datasets can be nested inside each other, and child datasets will inherit properties from their parents. Each dataset can be administered, [delegated](#), [replicated](#), [snapshotted](#), [jailed](#), and destroyed as a unit. There are many advantages to creating a separate dataset for each different type or set of files. The only drawbacks to having an extremely large number of datasets is that some commands like `zfs list` will be slower, and the mounting of hundreds or even thousands of datasets can slow the FreeBSD boot process.

Create a new dataset and enable [LZ4 compression](#) on it:

```
# zfs list
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool                              781M  93.2G  144K   none
mypool/R00T                         777M  93.2G  144K   none
mypool/R00T/default                 777M  93.2G  777M   /
mypool/tmp                          176K  93.2G  176K  /tmp
mypool/usr                          616K  93.2G  144K  /usr
mypool/usr/home                     184K  93.2G  184K  /usr/home
mypool/usr/ports                     144K  93.2G  144K  /usr/ports
mypool/usr/src                       144K  93.2G  144K  /usr/src
mypool/var                          1.20M  93.2G  608K  /var
mypool/var/crash                     148K  93.2G  148K  /var/crash
mypool/var/log                       178K  93.2G  178K  /var/log
mypool/var/mail                      144K  93.2G  144K  /var/mail
mypool/var/tmp                       152K  93.2G  152K  /var/tmp
# zfs create -o compress=lz4 mypool/usr/mydataset
# zfs list
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool                              781M  93.2G  144K   none
mypool/R00T                         777M  93.2G  144K   none
mypool/R00T/default                 777M  93.2G  777M   /
mypool/tmp                          176K  93.2G  176K  /tmp
mypool/usr                          704K  93.2G  144K  /usr
mypool/usr/home                     184K  93.2G  184K  /usr/home
mypool/usr/mydataset                 87.5K  93.2G  87.5K  /usr/mydataset
mypool/usr/ports                     144K  93.2G  144K  /usr/ports
mypool/usr/src                       144K  93.2G  144K  /usr/src
mypool/var                          1.20M  93.2G  610K  /var
mypool/var/crash                     148K  93.2G  148K  /var/crash
mypool/var/log                       178K  93.2G  178K  /var/log
mypool/var/mail                      144K  93.2G  144K  /var/mail
mypool/var/tmp                       152K  93.2G  152K  /var/tmp
```

Destroying a dataset is much quicker than deleting all of the files that reside on the dataset, as it does not involve scanning all of the files and updating all of the corresponding metadata.

Destroy the previously-created dataset:

```
# zfs list
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool                              880M  93.1G  144K   none
mypool/R00T                        777M  93.1G  144K   none
mypool/R00T/default                777M  93.1G  777M   /
mypool/tmp                         176K  93.1G  176K   /tmp
mypool/usr                         101M  93.1G  144K   /usr
mypool/usr/home                   184K  93.1G  184K   /usr/home
mypool/usr/mydataset              100M  93.1G  100M   /usr/mydataset
mypool/usr/ports                  144K  93.1G  144K   /usr/ports
mypool/usr/src                    144K  93.1G  144K   /usr/src
mypool/var                       1.20M  93.1G  610K   /var
mypool/var/crash                  148K  93.1G  148K   /var/crash
mypool/var/log                    178K  93.1G  178K   /var/log
mypool/var/mail                   144K  93.1G  144K   /var/mail
mypool/var/tmp                    152K  93.1G  152K   /var/tmp
# zfs destroy mypool/usr/mydataset
# zfs list
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool                              781M  93.2G  144K   none
mypool/R00T                        777M  93.2G  144K   none
mypool/R00T/default                777M  93.2G  777M   /
mypool/tmp                         176K  93.2G  176K   /tmp
mypool/usr                         616K  93.2G  144K   /usr
mypool/usr/home                   184K  93.2G  184K   /usr/home
mypool/usr/ports                  144K  93.2G  144K   /usr/ports
mypool/usr/src                    144K  93.2G  144K   /usr/src
mypool/var                       1.21M  93.2G  612K   /var
mypool/var/crash                  148K  93.2G  148K   /var/crash
mypool/var/log                    178K  93.2G  178K   /var/log
mypool/var/mail                   144K  93.2G  144K   /var/mail
mypool/var/tmp                    152K  93.2G  152K   /var/tmp
```

In modern versions of ZFS, `zfs destroy` is asynchronous, and the free space might take several minutes to appear in the pool. Use `zpool get freeing poolname` to see the `freeing` property, indicating how many datasets are having their blocks freed in the background. If there are child datasets, like [snapshots](#) or other datasets, then the parent cannot be destroyed. To destroy a dataset and all of its children, use `-r` to recursively destroy the dataset and all of its children. Use `-n -v` to list datasets and snapshots that would be destroyed by this operation, but do not actually destroy anything. Space that would be reclaimed by destruction of snapshots is also shown.

19.4.2. Creating and Destroying Volumes

A volume is a special type of dataset. Rather than being mounted as a file system, it is exposed as a block device under `/dev/zvol/ poolname/dataset`. This allows the volume to be used for other file systems, to back the disks of a virtual machine, or to be exported using protocols like iSCSI or HAST.

A volume can be formatted with any file system, or used without a file system to store raw data. To the user, a volume appears to be a regular disk. Putting ordinary file systems on these `zvols` provides features that ordinary disks or file systems do not normally have. For example, using the `compression` property on a 250 MB volume allows creation of a compressed FAT file system.

```
# zfs create -V 250m -o compression=on tank/fat32
# zfs list tank
NAME USED AVAIL REFER MOUNTPOINT
tank 258M 670M 31K /tank
# newfs_msdos -F32 /dev/zvol/tank/fat32
# mount -t msdosfs /dev/zvol/tank/fat32 /mnt
# df -h /mnt | grep fat32
Filesystem                Size Used Avail Capacity Mounted on
/dev/zvol/tank/fat32      249M  24k  249M      0%   /mnt
# mount | grep fat32
/dev/zvol/tank/fat32 on /mnt (msdosfs, local)
```

Destroying a volume is much the same as destroying a regular file system dataset. The operation is nearly instantaneous, but it may take several minutes for the free space to be reclaimed in the background.

19.4.3. Renaming a Dataset

The name of a dataset can be changed with `zfs rename`. The parent of a dataset can also be changed with this command. Renaming a dataset to be under a different parent dataset will change the value of those properties that are inherited from the parent dataset. When a dataset is renamed, it is unmounted and then remounted in the new location (which is inherited from the new parent dataset). This behavior can be prevented with `-u`.

Rename a dataset and move it to be under a different parent dataset:

```
# zfs list
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool                              780M  93.2G  144K   none
mypool/R00T                         777M  93.2G  144K   none
mypool/R00T/default                 777M  93.2G  777M   /
mypool/tmp                          176K  93.2G  176K   /tmp
mypool/usr                          704K  93.2G  144K   /usr
mypool/usr/home                     184K  93.2G  184K   /usr/home
mypool/usr/mydataset                87.5K  93.2G  87.5K   /usr/mydataset
mypool/usr/ports                     144K  93.2G  144K   /usr/ports
mypool/usr/src                       144K  93.2G  144K   /usr/src
mypool/var                          1.21M  93.2G  614K   /var
mypool/var/crash                     148K  93.2G  148K   /var/crash
mypool/var/log                       178K  93.2G  178K   /var/log
mypool/var/mail                      144K  93.2G  144K   /var/mail
mypool/var/tmp                       152K  93.2G  152K   /var/tmp

# zfs rename mypool/usr/mydataset mypool/var/newname
# zfs list
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool                              780M  93.2G  144K   none
mypool/R00T                         777M  93.2G  144K   none
mypool/R00T/default                 777M  93.2G  777M   /
mypool/tmp                          176K  93.2G  176K   /tmp
mypool/usr                          616K  93.2G  144K   /usr
mypool/usr/home                     184K  93.2G  184K   /usr/home
mypool/usr/ports                     144K  93.2G  144K   /usr/ports
mypool/usr/src                       144K  93.2G  144K   /usr/src
mypool/var                          1.29M  93.2G  614K   /var
mypool/var/crash                     148K  93.2G  148K   /var/crash
mypool/var/log                       178K  93.2G  178K   /var/log
mypool/var/mail                      144K  93.2G  144K   /var/mail
mypool/var/newname                   87.5K  93.2G  87.5K   /var/newname
mypool/var/tmp                       152K  93.2G  152K   /var/tmp
```

Snapshots can also be renamed like this. Due to the nature of snapshots, they cannot be renamed into a different parent dataset. To rename a recursive snapshot, specify `-r`, and all snapshots with the same name in child datasets will also be renamed.

```
# zfs list -t snapshot
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool/var/newname@first_snapshot    0     -   87.5K   -

# zfs rename mypool/var/newname@first_snapshot new_snapshot_name
# zfs list -t snapshot
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool/var/newname@new_snapshot_name  0     -   87.5K   -
```

19.4.4. Setting Dataset Properties

Each ZFS dataset has a number of properties that control its behavior. Most properties are automatically inherited from the parent dataset, but can be overridden locally. Set a property on a dataset with `zfs set property=value dataset`. Most properties have a limited set of valid values, `zfs get` will display each possible property and valid values. Most properties can be reverted to their inherited values using `zfs inherit`.

User-defined properties can also be set. They become part of the dataset configuration and can be used to provide additional information about the dataset or its contents. To distinguish these custom properties from the ones supplied as part of ZFS, a colon (:) is used to create a custom namespace for the property.

```
# zfs set custom:costcenter=1234 tank
# zfs get custom:costcenter tank
NAME PROPERTY          VALUE SOURCE
tank custom:costcenter 1234 local
```

To remove a custom property, use `zfs inherit` with `-r`. If the custom property is not defined in any of the parent datasets, it will be removed completely (although the changes are still recorded in the pool's history).

```
# zfs inherit -r custom:costcenter tank
# zfs get custom:costcenter tank
NAME PROPERTY          VALUE SOURCE
tank custom:costcenter - -
# zfs get all tank | grep custom:costcenter
#
```

19.4.4.1. Getting and Setting Share Properties

Two commonly used and useful dataset properties are the NFS and SMB share options. Setting these define if and how ZFS datasets may be shared on the network. At present, only setting sharing via NFS is supported on FreeBSD. To get the current status of a share, enter:

```
# zfs get sharenfs mypool/usr/home
NAME PROPERTY VALUE SOURCE
mypool/usr/home sharenfs on local
# zfs get sharesmb mypool/usr/home
NAME PROPERTY VALUE SOURCE
mypool/usr/home sharesmb off local
```

To enable sharing of a dataset, enter:

```
# zfs set sharenfs=on mypool/usr/home
```

It is also possible to set additional options for sharing datasets through NFS, such as `-alldirs`, `-maproot` and `-network`. To set additional options to a dataset shared through NFS, enter:

```
# zfs set sharenfs="-alldirs,-maproot= root,-network= 192.168.1.0/24 " mypool/usr/home
```

19.4.5. Managing Snapshots

[Snapshots](#) are one of the most powerful features of ZFS. A snapshot provides a read-only, point-in-time copy of the dataset. With Copy-On-Write (COW), snapshots can be created quickly by preserving the older version of the data on disk. If no snapshots exist, space is reclaimed for future use when data is rewritten or deleted. Snapshots preserve disk space by recording only the differences between the current dataset and a previous version. Snapshots are allowed only on whole datasets, not on individual files or directories. When a snapshot is created from a dataset, everything contained in it is duplicated. This includes the file system properties, files, directories, permissions, and so on. Snapshots use no additional space when they are first created, only consuming space as the blocks they reference are changed. Recursive snapshots taken with `-r` create a snapshot with the same name on the dataset and all of its children, providing a consistent moment-in-time snapshot of all of the file systems. This can be important when an application has files on multiple datasets that are related or dependent upon each other. Without snapshots, a backup would have copies of the files from different points in time.

Snapshots in ZFS provide a variety of features that even other file systems with snapshot functionality lack. A typical example of snapshot use is to have a quick way of backing up the current state of the file system when a risky action like a software installation or a system upgrade is performed. If the action fails, the snapshot can be rolled back and the system has the same state as when the snapshot was created. If the upgrade was successful, the snapshot can be deleted to free up space. Without snapshots, a failed upgrade often requires a restore from backup,

which is tedious, time consuming, and may require downtime during which the system cannot be used. Snapshots can be rolled back quickly, even while the system is running in normal operation, with little or no downtime. The time savings are enormous with multi-terabyte storage systems and the time required to copy the data from backup. Snapshots are not a replacement for a complete backup of a pool, but can be used as a quick and easy way to store a copy of the dataset at a specific point in time.

19.4.5.1. Creating Snapshots

Snapshots are created with `zfs snapshot dataset @snapshotname`. Adding `-r` creates a snapshot recursively, with the same name on all child datasets.

Create a recursive snapshot of the entire pool:

```
# zfs list -t all
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool                              780M  93.2G  144K   none
mypool/R00T                         777M  93.2G  144K   none
mypool/R00T/default                 777M  93.2G  777M   /
mypool/tmp                          176K  93.2G  176K   /tmp
mypool/usr                          616K  93.2G  144K   /usr
mypool/usr/home                     184K  93.2G  184K   /usr/home
mypool/usr/ports                     144K  93.2G  144K   /usr/ports
mypool/usr/src                       144K  93.2G  144K   /usr/src
mypool/var                          1.29M  93.2G  616K   /var
mypool/var/crash                     148K  93.2G  148K   /var/crash
mypool/var/log                       178K  93.2G  178K   /var/log
mypool/var/mail                      144K  93.2G  144K   /var/mail
mypool/var/newname                   87.5K  93.2G  87.5K   /var/newname
mypool/var/newname@new_snapshot_name 0      -    87.5K   -
mypool/var/tmp                       152K  93.2G  152K   /var/tmp
# zfs snapshot -r mypool@my_recursive_snapshot
# zfs list -t snapshot
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool@my_recursive_snapshot        0      -    144K   -
mypool/R00T@my_recursive_snapshot    0      -    144K   -
mypool/R00T/default@my_recursive_snapshot 0      -    777M   -
mypool/tmp@my_recursive_snapshot     0      -    176K   -
mypool/usr@my_recursive_snapshot     0      -    144K   -
mypool/usr/home@my_recursive_snapshot 0      -    184K   -
mypool/usr/ports@my_recursive_snapshot 0      -    144K   -
mypool/usr/src@my_recursive_snapshot 0      -    144K   -
mypool/var@my_recursive_snapshot     0      -    616K   -
mypool/var/crash@my_recursive_snapshot 0      -    148K   -
mypool/var/log@my_recursive_snapshot 0      -    178K   -
mypool/var/mail@my_recursive_snapshot 0      -    144K   -
mypool/var/newname@new_snapshot_name 0      -    87.5K   -
mypool/var/newname@my_recursive_snapshot 0      -    87.5K   -
mypool/var/tmp@my_recursive_snapshot 0      -    152K   -
```

Snapshots are not shown by a normal `zfs list` operation. To list snapshots, `-t snapshot` is appended to `zfs list`. `-t all` displays both file systems and snapshots.

Snapshots are not mounted directly, so path is shown in the `MOUNTPOINT` column. There is no mention of available disk space in the `AVAIL` column, as snapshots cannot be written to after they are created. Compare the snapshot to the original dataset from which it was created:

```
# zfs list -rt all mypool/usr/home
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool/usr/home                     184K  93.2G  184K   /usr/home
mypool/usr/home@my_recursive_snapshot 0      -    184K   -
```

Displaying both the dataset and the snapshot together reveals how snapshots work in [COW](#) fashion. They save only the changes (*delta*) that were made and not the complete file system contents all over again. This means that

snapshots take little space when few changes are made. Space usage can be made even more apparent by copying a file to the dataset, then making a second snapshot:

```
# cp /etc/passwd /var/tmp
# zfs snapshot mypool/var/tmp@after_cp
# zfs list -rt all mypool/var/tmp
```

NAME	USED	AVAIL	REFER	MOUNTPOINT
mypool/var/tmp	206K	93.2G	118K	/var/tmp
mypool/var/tmp@my_recursive_snapshot	88K	-	152K	-
mypool/var/tmp@after_cp	0	-	118K	-

The second snapshot contains only the changes to the dataset after the copy operation. This yields enormous space savings. Notice that the size of the snapshot *mypool/var/tmp@my_recursive_snapshot* also changed in the USED column to indicate the changes between itself and the snapshot taken afterwards.

19.4.5.2. Comparing Snapshots

ZFS provides a built-in command to compare the differences in content between two snapshots. This is helpful when many snapshots were taken over time and the user wants to see how the file system has changed over time. For example, *zfs diff* lets a user find the latest snapshot that still contains a file that was accidentally deleted. Doing this for the two snapshots that were created in the previous section yields this output:

```
# zfs list -rt all mypool/var/tmp
```

NAME	USED	AVAIL	REFER	MOUNTPOINT
mypool/var/tmp	206K	93.2G	118K	/var/tmp
mypool/var/tmp@my_recursive_snapshot	88K	-	152K	-
mypool/var/tmp@after_cp	0	-	118K	-

```
# zfs diff mypool/var/tmp@my_recursive_snapshot
```

M	/var/tmp/
+	/var/tmp/passwd

The command lists the changes between the specified snapshot (in this case *mypool/var/tmp@my_recursive_snapshot*) and the live file system. The first column shows the type of change:

+	The path or file was added.
-	The path or file was deleted.
M	The path or file was modified.
R	The path or file was renamed.

Comparing the output with the table, it becomes clear that *passwd* was added after the snapshot *mypool/var/tmp@my_recursive_snapshot* was created. This also resulted in a modification to the parent directory mounted at */var/tmp*.

Comparing two snapshots is helpful when using the ZFS replication feature to transfer a dataset to a different host for backup purposes.

Compare two snapshots by providing the full dataset name and snapshot name of both datasets:

```
# cp /var/tmp/passwd /var/tmp/passwd.copy
# zfs snapshot mypool/var/tmp@diff_snapshot
# zfs diff mypool/var/tmp@my_recursive_snapshot mypool/var/tmp@diff_snapshot
```

M	/var/tmp/
+	/var/tmp/passwd
+	/var/tmp/passwd.copy

```
# zfs diff mypool/var/tmp@my_recursive_snapshot mypool/var/tmp@after_cp
```

M	/var/tmp/
+	/var/tmp/passwd

A backup administrator can compare two snapshots received from the sending host and determine the actual changes in the dataset. See the [Replication](#) section for more information.

19.4.5.3. Snapshot Rollback

When at least one snapshot is available, it can be rolled back to at any time. Most of the time this is the case when the current state of the dataset is no longer required and an older version is preferred. Scenarios such as local development tests have gone wrong, botched system updates hampering the system's overall functionality, or the requirement to restore accidentally deleted files or directories are all too common occurrences. Luckily, rolling back a snapshot is just as easy as typing `zfs rollback snapshotname`. Depending on how many changes are involved, the operation will finish in a certain amount of time. During that time, the dataset always remains in a consistent state, much like a database that conforms to ACID principles is performing a rollback. This is happening while the dataset is live and accessible without requiring a downtime. Once the snapshot has been rolled back, the dataset has the same state as it had when the snapshot was originally taken. All other data in that dataset that was not part of the snapshot is discarded. Taking a snapshot of the current state of the dataset before rolling back to a previous one is a good idea when some data is required later. This way, the user can roll back and forth between snapshots without losing data that is still valuable.

In the first example, a snapshot is rolled back because of a careless `rm` operation that removes too much data than was intended.

```
# zfs list -rt all mypool/var/tmp
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool/var/tmp                      262K  93.2G  120K   /var/tmp
mypool/var/tmp@my_recursive_snapshot 88K   -      152K   -
mypool/var/tmp@after_cp             53.5K -      118K   -
mypool/var/tmp@diff_snapshot         0     -      120K   -
# ls /var/tmp
passwd      passwd.copy  vi.recover
# rm /var/tmp/passwd*
# ls /var/tmp
vi.recover
```

At this point, the user realized that too many files were deleted and wants them back. ZFS provides an easy way to get them back using rollbacks, but only when snapshots of important data are performed on a regular basis. To get the files back and start over from the last snapshot, issue the command:

```
# zfs rollback mypool/var/tmp@diff_snapshot
# ls /var/tmp
passwd      passwd.copy  vi.recover
```

The rollback operation restored the dataset to the state of the last snapshot. It is also possible to roll back to a snapshot that was taken much earlier and has other snapshots that were created after it. When trying to do this, ZFS will issue this warning:

```
# zfs list -rt snapshot mypool/var/tmp
AME                                USED  AVAIL  REFER  MOUNTPOINT
mypool/var/tmp@my_recursive_snapshot 88K   -      152K   -
mypool/var/tmp@after_cp             53.5K -      118K   -
mypool/var/tmp@diff_snapshot         0     -      120K   -
# zfs rollback mypool/var/tmp@my_recursive_snapshot
cannot rollback to 'mypool/var/tmp@my_recursive_snapshot': more recent snapshots exist
use '-r' to force deletion of the following snapshots:
mypool/var/tmp@after_cp
mypool/var/tmp@diff_snapshot
```

This warning means that snapshots exist between the current state of the dataset and the snapshot to which the user wants to roll back. To complete the rollback, these snapshots must be deleted. ZFS cannot track all the changes between different states of the dataset, because snapshots are read-only. ZFS will not delete the affected snapshots unless the user specifies `-r` to indicate that this is the desired action. If that is the intention, and the consequences of losing all intermediate snapshots is understood, the command can be issued:

```
# zfs rollback -r mypool/var/tmp@my_recursive_snapshot
# zfs list -rt snapshot mypool/var/tmp
NAME                                USED  AVAIL  REFER  MOUNTPOINT
```

```

mypool/var/tmp@my_recursive_snapshot    8K    -    152K    -
# ls /var/tmp
vi.recover

```

The output from `zfs list -t snapshot` confirms that the intermediate snapshots were removed as a result of `zfs rollback -r`.

19.4.5.4. Restoring Individual Files from Snapshots

Snapshots are mounted in a hidden directory under the parent dataset: `.zfs/snapshots/ snapshotname`. By default, these directories will not be displayed even when a standard `ls -a` is issued. Although the directory is not displayed, it is there nevertheless and can be accessed like any normal directory. The property named `snapdir` controls whether these hidden directories show up in a directory listing. Setting the property to `visible` allows them to appear in the output of `ls` and other commands that deal with directory contents.

```

# zfs get snapdir mypool/var/tmp
NAME          PROPERTY  VALUE    SOURCE
mypool/var/tmp snapdir   hidden   default
# ls -a /var/tmp
.      ..      passwd      vi.recover
# zfs set snapdir=visible mypool/var/tmp
# ls -a /var/tmp
.      ..      .zfs      passwd      vi.recover

```

Individual files can easily be restored to a previous state by copying them from the snapshot back to the parent dataset. The directory structure below `.zfs/snapshot` has a directory named exactly like the snapshots taken earlier to make it easier to identify them. In the next example, it is assumed that a file is to be restored from the hidden `.zfs` directory by copying it from the snapshot that contained the latest version of the file:

```

# rm /var/tmp/passwd
# ls -a /var/tmp
.      ..      .zfs      vi.recover
# ls /var/tmp/.zfs/snapshot
after_cp      my_recursive_snapshot
# ls /var/tmp/.zfs/snapshot/ after_cp
passwd      vi.recover
# cp /var/tmp/.zfs/snapshot/ after_cp/passwd /var/tmp

```

When `ls .zfs/snapshot` was issued, the `snapdir` property might have been set to `hidden`, but it would still be possible to list the contents of that directory. It is up to the administrator to decide whether these directories will be displayed. It is possible to display these for certain datasets and prevent it for others. Copying files or directories from this hidden `.zfs/snapshot` is simple enough. Trying it the other way around results in this error:

```

# cp /etc/rc.conf /var/tmp/.zfs/snapshot/ after_cp/
cp: /var/tmp/.zfs/snapshot/after_cp/rc.conf: Read-only file system

```

The error reminds the user that snapshots are read-only and cannot be changed after creation. Files cannot be copied into or removed from snapshot directories because that would change the state of the dataset they represent.

Snapshots consume space based on how much the parent file system has changed since the time of the snapshot. The written property of a snapshot tracks how much space is being used by the snapshot.

Snapshots are destroyed and the space reclaimed with `zfs destroy dataset@snapshot`. Adding `-r` recursively removes all snapshots with the same name under the parent dataset. Adding `-n -v` to the command displays a list of the snapshots that would be deleted and an estimate of how much space would be reclaimed without performing the actual destroy operation.

19.4.6. Managing Clones

A clone is a copy of a snapshot that is treated more like a regular dataset. Unlike a snapshot, a clone is not read only, is mounted, and can have its own properties. Once a clone has been created using `zfs clone`, the snapshot

it was created from cannot be destroyed. The child/parent relationship between the clone and the snapshot can be reversed using `zfs promote`. After a clone has been promoted, the snapshot becomes a child of the clone, rather than of the original parent dataset. This will change how the space is accounted, but not actually change the amount of space consumed. The clone can be mounted at any point within the ZFS file system hierarchy, not just below the original location of the snapshot.

To demonstrate the clone feature, this example dataset is used:

```
# zfs list -rt all camino/home/joe
```

NAME	USED	AVAIL	REFER	MOUNTPOINT
camino/home/joe	108K	1.3G	87K	/usr/home/joe
camino/home/joe@plans	21K	-	85.5K	-
camino/home/joe@backup	0K	-	87K	-

A typical use for clones is to experiment with a specific dataset while keeping the snapshot around to fall back to in case something goes wrong. Since snapshots cannot be changed, a read/write clone of a snapshot is created. After the desired result is achieved in the clone, the clone can be promoted to a dataset and the old file system removed. This is not strictly necessary, as the clone and dataset can coexist without problems.

```
# zfs clone camino/home/joe @backup camino/home/joeneu
# ls /usr/home/joe*
/usr/home/joe:
backup.txz    plans.txt

/usr/home/joeneu:
backup.txz    plans.txt
# df -h /usr/home
```

Filesystem	Size	Used	Avail	Capacity	Mounted on
usr/home/joe	1.3G	31k	1.3G	0%	/usr/home/joe
usr/home/joeneu	1.3G	31k	1.3G	0%	/usr/home/joeneu

After a clone is created it is an exact copy of the state the dataset was in when the snapshot was taken. The clone can now be changed independently from its originating dataset. The only connection between the two is the snapshot. ZFS records this connection in the property `origin`. Once the dependency between the snapshot and the clone has been removed by promoting the clone using `zfs promote`, the `origin` of the clone is removed as it is now an independent dataset. This example demonstrates it:

```
# zfs get origin camino/home/joeneu
```

NAME	PROPERTY	VALUE	SOURCE
camino/home/joeneu	origin	camino/home/joe@backup	-

```
# zfs promote camino/home/joeneu
# zfs get origin camino/home/joeneu
```

NAME	PROPERTY	VALUE	SOURCE
camino/home/joeneu	origin	-	-

After making some changes like copying `loader.conf` to the promoted clone, for example, the old directory becomes obsolete in this case. Instead, the promoted clone can replace it. This can be achieved by two consecutive commands: `zfs destroy` on the old dataset and `zfs rename` on the clone to name it like the old dataset (it could also get an entirely different name).

```
# cp /boot/defaults/loader.conf /usr/home/joeneu
# zfs destroy -f camino/home/joe
# zfs rename camino/home/joeneu camino/home/joe
# ls /usr/home/joe
backup.txz    loader.conf    plans.txt
# df -h /usr/home
```

Filesystem	Size	Used	Avail	Capacity	Mounted on
usr/home/joe	1.3G	128k	1.3G	0%	/usr/home/joe

The cloned snapshot is now handled like an ordinary dataset. It contains all the data from the original snapshot plus the files that were added to it like `loader.conf`. Clones can be used in different scenarios to provide useful features to ZFS users. For example, jails could be provided as snapshots containing different sets of installed applications. Users can clone these snapshots and add their own applications as they see fit. Once they are satisfied with the

changes, the clones can be promoted to full datasets and provided to end users to work with like they would with a real dataset. This saves time and administrative overhead when providing these jails.

19.4.7. Replication

Keeping data on a single pool in one location exposes it to risks like theft and natural or human disasters. Making regular backups of the entire pool is vital. ZFS provides a built-in serialization feature that can send a stream representation of the data to standard output. Using this technique, it is possible to not only store the data on another pool connected to the local system, but also to send it over a network to another system. Snapshots are the basis for this replication (see the section on [ZFS snapshots](#)). The commands used for replicating data are `zfs send` and `zfs receive`.

These examples demonstrate ZFS replication with these two pools:

```
# zpool list
NAME      SIZE  ALLOC   FREE      CAP  DEDUP  HEALTH  ALTROOT
backup    960M   77K    896M       0%  1.00x  ONLINE  -
mypool    984M  43.7M   940M       4%  1.00x  ONLINE  -
```

The pool named *mypool* is the primary pool where data is written to and read from on a regular basis. A second pool, *backup* is used as a standby in case the primary pool becomes unavailable. Note that this fail-over is not done automatically by ZFS, but must be manually done by a system administrator when needed. A snapshot is used to provide a consistent version of the file system to be replicated. Once a snapshot of *mypool* has been created, it can be copied to the *backup* pool. Only snapshots can be replicated. Changes made since the most recent snapshot will not be included.

```
# zfs snapshot mypool@backup1
# zfs list -t snapshot
NAME                                USED  AVAIL  REFER  MOUNTPOINT
mypool@backup1                      0      -   43.6M  -
```

Now that a snapshot exists, `zfs send` can be used to create a stream representing the contents of the snapshot. This stream can be stored as a file or received by another pool. The stream is written to standard output, but must be redirected to a file or pipe or an error is produced:

```
# zfs send mypool@backup1
Error: Stream can not be written to a terminal.
You must redirect standard output.
```

To back up a dataset with `zfs send`, redirect to a file located on the mounted backup pool. Ensure that the pool has enough free space to accommodate the size of the snapshot being sent, which means all of the data contained in the snapshot, not just the changes from the previous snapshot.

```
# zfs send mypool@backup1 > /backup/backup1
# zpool list
NAME      SIZE  ALLOC   FREE      CAP  DEDUP  HEALTH  ALTROOT
backup    960M  63.7M   896M       6%  1.00x  ONLINE  -
mypool    984M  43.7M   940M       4%  1.00x  ONLINE  -
```

The `zfs send` transferred all the data in the snapshot called *backup1* to the pool named *backup*. Creating and sending these snapshots can be done automatically with a [cron\(8\)](#) job.

Instead of storing the backups as archive files, ZFS can receive them as a live file system, allowing the backed up data to be accessed directly. To get to the actual data contained in those streams, `zfs receive` is used to transform the streams back into files and directories. The example below combines `zfs send` and `zfs receive` using a pipe to copy the data from one pool to another. The data can be used directly on the receiving pool after the transfer is complete. A dataset can only be replicated to an empty dataset.

```
# zfs snapshot mypool@replica1
# zfs send -v mypool@replica1 | zfs receive backup/mypool
send from @ to mypool@replica1 estimated size is 50.1M
total estimated size is 50.1M
```

```

TIME      SENT      SNAPSHOT

# zpool list
NAME      SIZE  ALLOC  FREE   CAP  DEDUP  HEALTH  ALTROOT
backup    960M  63.7M  896M   6%   1.00x  ONLINE  -
mypool    984M  43.7M  940M   4%   1.00x  ONLINE  -

```

19.4.7.1. Incremental Backups

`zfs send` can also determine the difference between two snapshots and send only the differences between the two. This saves disk space and transfer time. For example:

```

# zfs snapshot mypool@replica2
# zfs list -t snapshot
NAME                               USED  AVAIL  REFER  MOUNTPOINT
mypool@replica1                    5.72M      -   43.6M      -
mypool@replica2                     0         -   44.1M      -
# zpool list
NAME      SIZE  ALLOC  FREE   CAP  DEDUP  HEALTH  ALTROOT
backup    960M  61.7M  898M   6%   1.00x  ONLINE  -
mypool    960M  50.2M  910M   5%   1.00x  ONLINE  -

```

A second snapshot called *replica2* was created. This second snapshot contains only the changes that were made to the file system between now and the previous snapshot, *replica1*. Using `zfs send -i` and indicating the pair of snapshots generates an incremental replica stream containing only the data that has changed. This can only succeed if the initial snapshot already exists on the receiving side.

```

# zfs send -v -i mypool@replica1 mypool@replica2 | zfs receive /backup/mypool
send from @replica1 to mypool@replica2 estimated size is 5.02M
total estimated size is 5.02M
TIME      SENT      SNAPSHOT

# zpool list
NAME      SIZE  ALLOC  FREE   CAP  DEDUP  HEALTH  ALTROOT
backup    960M  80.8M  879M   8%   1.00x  ONLINE  -
mypool    960M  50.2M  910M   5%   1.00x  ONLINE  -

# zfs list
NAME                               USED  AVAIL  REFER  MOUNTPOINT
backup                    55.4M   240G   152K   /backup
backup/mypool             55.3M   240G   55.2M  /backup/mypool
mypool                    55.6M  11.6G   55.0M  /mypool

# zfs list -t snapshot
NAME                               USED  AVAIL  REFER  MOUNTPOINT
backup/mypool@replica1           104K      -   50.2M      -
backup/mypool@replica2            0         -   55.2M      -
mypool@replica1                 29.9K      -   50.0M      -
mypool@replica2                  0         -   55.0M      -

```

The incremental stream was successfully transferred. Only the data that had changed was replicated, rather than the entirety of *replica1*. Only the differences were sent, which took much less time to transfer and saved disk space by not copying the complete pool each time. This is useful when having to rely on slow networks or when costs per transferred byte must be considered.

A new file system, *backup/mypool*, is available with all of the files and data from the pool *mypool*. If `-P` is specified, the properties of the dataset will be copied, including compression settings, quotas, and mount points. When `-R` is specified, all child datasets of the indicated dataset will be copied, along with all of their properties. Sending and receiving can be automated so that regular backups are created on the second pool.

19.4.7.2. Sending Encrypted Backups over SSH

Sending streams over the network is a good way to keep a remote backup, but it does come with a drawback. Data sent over the network link is not encrypted, allowing anyone to intercept and transform the streams back into

data without the knowledge of the sending user. This is undesirable, especially when sending the streams over the internet to a remote host. SSH can be used to securely encrypt data send over a network connection. Since ZFS only requires the stream to be redirected from standard output, it is relatively easy to pipe it through SSH. To keep the contents of the file system encrypted in transit and on the remote system, consider using [PEFS](#).

A few settings and security precautions must be completed first. Only the necessary steps required for the `zfs` send operation are shown here. For more information on SSH, see [Section 13.8, “OpenSSH”](#).

This configuration is required:

- Passwordless SSH access between sending and receiving host using SSH keys
- Normally, the privileges of the `root` user are needed to send and receive streams. This requires logging in to the receiving system as `root`. However, logging in as `root` is disabled by default for security reasons. The [ZFS Delegation](#) system can be used to allow a non-root user on each system to perform the respective send and receive operations.
- On the sending system:

```
# zfs allow -u someuser send,snapshot mypool
```

- To mount the pool, the unprivileged user must own the directory, and regular users must be allowed to mount file systems. On the receiving system:

```
# sysctl vfs.usermount=1
vfs.usermount: 0 -> 1
# sysrc -f /etc/sysctl.conf vfs.usermount=1
# zfs create recvpool/backup
# zfs allow -u someuser create,mount,receive recvpool/backup
# chown someuser /recvpool/backup
```

The unprivileged user now has the ability to receive and mount datasets, and the `home` dataset can be replicated to the remote system:

```
% zfs snapshot -r mypool/home @monday
% zfs send -R mypool/home @monday | ssh someuser@backuphost zfs recv -dvv recvpool/backup
```

A recursive snapshot called `monday` is made of the file system dataset `home` that resides on the pool `mypool`. Then it is sent with `zfs send -R` to include the dataset, all child datasets, snapshots, clones, and settings in the stream. The output is piped to the waiting `zfs` receive on the remote host `backuphost` through SSH. Using a fully qualified domain name or IP address is recommended. The receiving machine writes the data to the `backup` dataset on the `recvpool` pool. Adding `-d` to `zfs recv` overwrites the name of the pool on the receiving side with the name of the snapshot. `-u` causes the file systems to not be mounted on the receiving side. When `-v` is included, more detail about the transfer is shown, including elapsed time and the amount of data transferred.

19.4.8. Dataset, User, and Group Quotas

[Dataset quotas](#) are used to restrict the amount of space that can be consumed by a particular dataset. [Reference Quotas](#) work in very much the same way, but only count the space used by the dataset itself, excluding snapshots and child datasets. Similarly, [user](#) and [group](#) quotas can be used to prevent users or groups from using all of the space in the pool or dataset.

To enforce a dataset quota of 10 GB for `storage/home/bob`:

```
# zfs set quota=10G storage/home/bob
```

To enforce a reference quota of 10 GB for `storage/home/bob`:

```
# zfs set refquota=10G storage/home/bob
```

To remove a quota of 10 GB for `storage/home/bob`:

```
# zfs set quota=none storage/home/bob
```

The general format is `userquota@user=size`, and the user's name must be in one of these formats:

- POSIX compatible name such as *joe*.
- POSIX numeric ID such as *789*.
- SID name such as *joe.bloggs@example.com*.
- SID numeric ID such as *S-1-123-456-789*.

For example, to enforce a user quota of 50 GB for the user named *joe*:

```
# zfs set userquota@joe=50G
```

To remove any quota:

```
# zfs set userquota@joe=none
```



Note

User quota properties are not displayed by `zfs get all`. Non-root users can only see their own quotas unless they have been granted the `userquota` privilege. Users with this privilege are able to view and set everyone's quota.

The general format for setting a group quota is: `groupquota@group=size`.

To set the quota for the group *firstgroup* to 50 GB, use:

```
# zfs set groupquota@firstgroup=50G
```

To remove the quota for the group *firstgroup*, or to make sure that one is not set, instead use:

```
# zfs set groupquota@firstgroup=none
```

As with the user quota property, non-root users can only see the quotas associated with the groups to which they belong. However, root or a user with the `groupquota` privilege can view and set all quotas for all groups.

To display the amount of space used by each user on a file system or snapshot along with any quotas, use `zfs userspace`. For group information, use `zfs groupspace`. For more information about supported options or how to display only specific options, refer to [zfs\(1\)](#).

Users with sufficient privileges, and root, can list the quota for `storage/home/bob` using:

```
# zfs get quota storage/home/bob
```

19.4.9. Reservations

[Reservations](#) guarantee a minimum amount of space will always be available on a dataset. The reserved space will not be available to any other dataset. This feature can be especially useful to ensure that free space is available for an important dataset or log files.

The general format of the reservation property is `reservation=size`, so to set a reservation of 10 GB on `storage/home/bob`, use:

```
# zfs set reservation=10G storage/home/bob
```

To clear any reservation:

```
# zfs set reservation=none storage/home/bob
```


The same principle can be applied to the `refreservation` property for setting a [Reference Reservation](#), with the general format `refreservation=size`.

This command shows any reservations or refreservations that exist on `storage/home/bob` :

```
# zfs get reservation storage/home/bob
# zfs get refreservation storage/home/bob
```

19.4.10. Compression

ZFS provides transparent compression. Compressing data at the block level as it is written not only saves space, but can also increase disk throughput. If data is compressed by 25%, but the compressed data is written to the disk at the same rate as the uncompressed version, resulting in an effective write speed of 125%. Compression can also be a great alternative to [Deduplication](#) because it does not require additional memory.

ZFS offers several different compression algorithms, each with different trade-offs. With the introduction of LZ4 compression in ZFS v5000, it is possible to enable compression for the entire pool without the large performance trade-off of other algorithms. The biggest advantage to LZ4 is the *early abort* feature. If LZ4 does not achieve at least 12.5% compression in the first part of the data, the block is written uncompressed to avoid wasting CPU cycles trying to compress data that is either already compressed or uncompressible. For details about the different compression algorithms available in ZFS, see the [Compression](#) entry in the terminology section.

The administrator can monitor the effectiveness of compression using a number of dataset properties.

# zfs get	used,compressratio,compression,logicalused		mypool/compressed_dataset
NAME	PROPERTY	VALUE	SOURCE
mypool/compressed_dataset	used	449G	-
mypool/compressed_dataset	compressratio	1.11x	-
mypool/compressed_dataset	compression	lz4	local
mypool/compressed_dataset	logicalused	496G	-

The dataset is currently using 449 GB of space (the `used` property). Without compression, it would have taken 496 GB of space (the `logicalused` property). This results in the 1.11:1 compression ratio.

Compression can have an unexpected side effect when combined with [User Quotas](#). User quotas restrict how much space a user can consume on a dataset, but the measurements are based on how much space is used *after compression*. So if a user has a quota of 10 GB, and writes 10 GB of compressible data, they will still be able to store additional data. If they later update a file, say a database, with more or less compressible data, the amount of space available to them will change. This can result in the odd situation where a user did not increase the actual amount of data (the `logicalused` property), but the change in compression caused them to reach their quota limit.

Compression can have a similar unexpected interaction with backups. Quotas are often used to limit how much data can be stored to ensure there is sufficient backup space available. However since quotas do not consider compression, more data may be written than would fit with uncompressed backups.

19.4.11. Deduplication

When enabled, [deduplication](#) uses the checksum of each block to detect duplicate blocks. When a new block is a duplicate of an existing block, ZFS writes an additional reference to the existing data instead of the whole duplicate block. Tremendous space savings are possible if the data contains many duplicated files or repeated information. Be warned: deduplication requires an extremely large amount of memory, and most of the space savings can be had without the extra cost by enabling compression instead.

To activate deduplication, set the `dedup` property on the target pool:

```
# zfs set dedup=on pool
```

Only new data being written to the pool will be deduplicated. Data that has already been written to the pool will not be deduplicated merely by activating this option. A pool with a freshly activated deduplication property will look like this example:

```
# zpool list
NAME  SIZE  ALLOC  FREE  CAP  DEDUP  HEALTH  ALTROOT
pool  2.84G  2.19M  2.83G   0%  1.00x  ONLINE  -
```

The `DEDUP` column shows the actual rate of deduplication for the pool. A value of `1.00x` shows that data has not been deduplicated yet. In the next example, the ports tree is copied three times into different directories on the deduplicated pool created above.

```
# for d in dir1 dir2 dir3; do
> mkdir $d && cp -R /usr/ports $d &
> done
```

Redundant data is detected and deduplicated:

```
# zpool list
NAME  SIZE  ALLOC  FREE  CAP  DEDUP  HEALTH  ALTROOT
pool  2.84G  20.9M  2.82G   0%  3.00x  ONLINE  -
```

The `DEDUP` column shows a factor of `3.00x`. Multiple copies of the ports tree data was detected and deduplicated, using only a third of the space. The potential for space savings can be enormous, but comes at the cost of having enough memory to keep track of the deduplicated blocks.

Deduplication is not always beneficial, especially when the data on a pool is not redundant. ZFS can show potential space savings by simulating deduplication on an existing pool:

```
# zdb -S pool
Simulated DDT histogram:
```

bucket	allocated				referenced			
refcnt	blocks	LSIZE	PSIZE	DSIZE	blocks	LSIZE	PSIZE	DSIZE
1	2.58M	289G	264G	264G	2.58M	289G	264G	264G
2	206K	12.6G	10.4G	10.4G	430K	26.4G	21.6G	21.6G
4	37.6K	692M	276M	276M	170K	3.04G	1.26G	1.26G
8	2.18K	45.2M	19.4M	19.4M	20.0K	425M	176M	176M
16	174	2.83M	1.20M	1.20M	3.33K	48.4M	20.4M	20.4M
32	40	2.17M	222K	222K	1.70K	97.2M	9.91M	9.91M
64	9	56K	10.5K	10.5K	865	4.96M	948K	948K
128	2	9.50K	2K	2K	419	2.11M	438K	438K
256	5	61.5K	12K	12K	1.90K	23.0M	4.47M	4.47M
1K	2	1K	1K	1K	2.98K	1.49M	1.49M	1.49M
Total	2.82M	303G	275G	275G	3.20M	319G	287G	287G

dedup = 1.05, compress = 1.11, copies = 1.00, dedup * compress / copies = 1.16

After `zdb -S` finishes analyzing the pool, it shows the space reduction ratio that would be achieved by activating deduplication. In this case, `1.16` is a very poor space saving ratio that is mostly provided by compression. Activating deduplication on this pool would not save any significant amount of space, and is not worth the amount of memory required to enable deduplication. Using the formula $ratio = dedup * compress / copies$, system administrators can plan the storage allocation, deciding whether the workload will contain enough duplicate blocks to justify the memory requirements. If the data is reasonably compressible, the space savings may be very good. Enabling compression first is recommended, and compression can also provide greatly increased performance. Only enable deduplication in cases where the additional savings will be considerable and there is sufficient memory for the DDT.

19.4.12. ZFS and Jails

`zfs jail` and the corresponding `jailed` property are used to delegate a ZFS dataset to a [Jail](#). `zfs jail jailid` attaches a dataset to the specified jail, and `zfs unjail` detaches it. For the dataset to be controlled from within a jail, the `jailed` property must be set. Once a dataset is jailed, it can no longer be mounted on the host because it may have mount points that would compromise the security of the host.

19.5. Delegated Administration

A comprehensive permission delegation system allows unprivileged users to perform ZFS administration functions. For example, if each user's home directory is a dataset, users can be given permission to create and destroy snapshots of their home directories. A backup user can be given permission to use replication features. A usage statistics script can be allowed to run with access only to the space utilization data for all users. It is even possible to delegate the ability to delegate permissions. Permission delegation is possible for each subcommand and most properties.

19.5.1. Delegating Dataset Creation

`zfs allow someuser create mydataset` gives the specified user permission to create child datasets under the selected parent dataset. There is a caveat: creating a new dataset involves mounting it. That requires setting the FreeBSD `vfs.usermount` [sysctl\(8\)](#) to 1 to allow non-root users to mount a file system. There is another restriction aimed at preventing abuse: non-root users must own the mountpoint where the file system is to be mounted.

19.5.2. Delegating Permission Delegation

`zfs allow someuser allow mydataset` gives the specified user the ability to assign any permission they have on the target dataset, or its children, to other users. If a user has the `snapshot` permission and the `allow` permission, that user can then grant the `snapshot` permission to other users.

19.6. Advanced Topics

19.6.1. Tuning

There are a number of tunables that can be adjusted to make ZFS perform best for different workloads.

- `vfs.zfs.arc_max` - Maximum size of the [ARC](#). The default is all RAM less 1 GB, or one half of RAM, whichever is more. However, a lower value should be used if the system will be running any other daemons or processes that may require memory. This value can only be adjusted at boot time, and is set in `/boot/loader.conf`.
- `vfs.zfs.arc_meta_limit` - Limit the portion of the [ARC](#) that can be used to store metadata. The default is one fourth of `vfs.zfs.arc_max`. Increasing this value will improve performance if the workload involves operations on a large number of files and directories, or frequent metadata operations, at the cost of less file data fitting in the [ARC](#). This value can only be adjusted at boot time, and is set in `/boot/loader.conf`.
- `vfs.zfs.arc_min` - Minimum size of the [ARC](#). The default is one half of `vfs.zfs.arc_meta_limit`. Adjust this value to prevent other applications from pressuring out the entire [ARC](#). This value can only be adjusted at boot time, and is set in `/boot/loader.conf`.
- `vfs.zfs.vdev.cache.size` - A preallocated amount of memory reserved as a cache for each device in the pool. The total amount of memory used will be this value multiplied by the number of devices. This value can only be adjusted at boot time, and is set in `/boot/loader.conf`.
- `vfs.zfs.min_auto_ashift` - Minimum `ashift` (sector size) that will be used automatically at pool creation time. The value is a power of two. The default value of 9 represents $2^9 = 512$, a sector size of 512 bytes. To avoid *write amplification* and get the best performance, set this value to the largest sector size used by a device in the pool.

Many drives have 4 KB sectors. Using the default `ashift` of 9 with these drives results in write amplification on these devices. Data that could be contained in a single 4 KB write must instead be written in eight 512-byte writes. ZFS tries to read the native sector size from all devices when creating a pool, but many drives with 4 KB sectors report that their sectors are 512 bytes for compatibility. Setting `vfs.zfs.min_auto_ashift` to 12 ($2^{12} = 4096$) before creating a pool forces ZFS to use 4 KB blocks for best performance on these drives.

Forcing 4 KB blocks is also useful on pools where disk upgrades are planned. Future disks are likely to use 4 KB sectors, and `ashift` values cannot be changed after a pool is created.

In some specific cases, the smaller 512-byte block size might be preferable. When used with 512-byte disks for databases, or as storage for virtual machines, less data is transferred during small random reads. This can provide better performance, especially when using a smaller ZFS record size.

- *vfs.zfs.prefetch_disable* - Disable prefetch. A value of 0 is enabled and 1 is disabled. The default is 0, unless the system has less than 4 GB of RAM. Prefetch works by reading larger blocks than were requested into the [ARC](#) in hopes that the data will be needed soon. If the workload has a large number of random reads, disabling prefetch may actually improve performance by reducing unnecessary reads. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.vdev.trim_on_init* - Control whether new devices added to the pool have the TRIM command run on them. This ensures the best performance and longevity for SSDs, but takes extra time. If the device has already been secure erased, disabling this setting will make the addition of the new device faster. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.vdev.max_pending* - Limit the number of pending I/O requests per device. A higher value will keep the device command queue full and may give higher throughput. A lower value will reduce latency. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.top_maxinflight* - Maximum number of outstanding I/Os per top-level [vdev](#). Limits the depth of the command queue to prevent high latency. The limit is per top-level vdev, meaning the limit applies to each [mirror](#) [394], [RAID-Z](#) [395], or other vdev independently. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.l2arc_write_max* - Limit the amount of data written to the [L2ARC](#) per second. This tunable is designed to extend the longevity of SSDs by limiting the amount of data written to the device. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.l2arc_write_boost* - The value of this tunable is added to [vfs.zfs.l2arc_write_max](#) [392] and increases the write speed to the SSD until the first block is evicted from the [L2ARC](#). This “Turbo Warmup Phase” is designed to reduce the performance loss from an empty [L2ARC](#) after a reboot. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.scrub_delay* - Number of ticks to delay between each I/O during a [scrub](#). To ensure that a scrub does not interfere with the normal operation of the pool, if any other I/O is happening the scrub will delay between each command. This value controls the limit on the total IOPS (I/Os Per Second) generated by the scrub. The granularity of the setting is determined by the value of `kern.hz` which defaults to 1000 ticks per second. This setting may be changed, resulting in a different effective IOPS limit. The default value is 4, resulting in a limit of: 1000 ticks/sec / 4 = 250 IOPS. Using a value of 20 would give a limit of: 1000 ticks/sec / 20 = 50 IOPS. The speed of scrub is only limited when there has been recent activity on the pool, as determined by [vfs.zfs.scan_idle](#) [392]. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.resilver_delay* - Number of milliseconds of delay inserted between each I/O during a [resilver](#). To ensure that a resilver does not interfere with the normal operation of the pool, if any other I/O is happening the resilver will delay between each command. This value controls the limit of total IOPS (I/Os Per Second) generated by the resilver. The granularity of the setting is determined by the value of `kern.hz` which defaults to 1000 ticks per second. This setting may be changed, resulting in a different effective IOPS limit. The default value is 2, resulting in a limit of: 1000 ticks/sec / 2 = 500 IOPS. Returning the pool to an [Online](#) state may be more important if another device failing could [Fault](#) the pool, causing data loss. A value of 0 will give the resilver operation the same priority as other operations, speeding the healing process. The speed of resilver is only limited when there has been other recent activity on the pool, as determined by [vfs.zfs.scan_idle](#) [392]. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.scan_idle* - Number of milliseconds since the last operation before the pool is considered idle. When the pool is idle the rate limiting for [scrub](#) and [resilver](#) are disabled. This value can be adjusted at any time with [sysctl\(8\)](#).
- *vfs.zfs.txg.timeout* - Maximum number of seconds between [transaction groups](#). The current transaction group will be written to the pool and a fresh transaction group started if this amount of time has elapsed since

the previous transaction group. A transaction group may be triggered earlier if enough data is written. The default value is 5 seconds. A larger value may improve read performance by delaying asynchronous writes, but this may cause uneven performance when the transaction group is written. This value can be adjusted at any time with `sysctl(8)`.

19.6.2. ZFS on i386

Some of the features provided by ZFS are memory intensive, and may require tuning for maximum efficiency on systems with limited RAM.

19.6.2.1. Memory

As a bare minimum, the total system memory should be at least one gigabyte. The amount of recommended RAM depends upon the size of the pool and which ZFS features are used. A general rule of thumb is 1 GB of RAM for every 1 TB of storage. If the deduplication feature is used, a general rule of thumb is 5 GB of RAM per TB of storage to be deduplicated. While some users successfully use ZFS with less RAM, systems under heavy load may panic due to memory exhaustion. Further tuning may be required for systems with less than the recommended RAM requirements.

19.6.2.2. Kernel Configuration

Due to the address space limitations of the i386™ platform, ZFS users on the i386™ architecture must add this option to a custom kernel configuration file, rebuild the kernel, and reboot:

```
options          KVA_PAGES=512
```

This expands the kernel address space, allowing the `vm.kvm_size` tunable to be pushed beyond the currently imposed limit of 1 GB, or the limit of 2 GB for PAE. To find the most suitable value for this option, divide the desired address space in megabytes by four. In this example, it is 512 for 2 GB.

19.6.2.3. Loader Tunables

The `kmem` address space can be increased on all FreeBSD architectures. On a test system with 1 GB of physical memory, success was achieved with these options added to `/boot/loader.conf`, and the system restarted:

```
vm.kmem_size="330M"
vm.kmem_size_max="330M"
vfs.zfs.arc_max="40M"
vfs.zfs.vdev.cache.size="5M"
```


For a more detailed list of recommendations for ZFS-related tuning, see <https://wiki.freebsd.org/ZFSTuningGuide>.

19.7. Additional Resources

- [FreeBSD Wiki - ZFS](#)
- [FreeBSD Wiki - ZFS Tuning](#)
- [Illumos Wiki - ZFS](#)
- [Oracle Solaris ZFS Administration Guide](#)
- [ZFS Evil Tuning Guide](#)
- [ZFS Best Practices Guide](#)
- [Calomel Blog - ZFS Raidz Performance, Capacity and Integrity](#)

19.8. ZFS Features and Terminology

ZFS is a fundamentally different file system because it is more than just a file system. ZFS combines the roles of file system and volume manager, enabling additional storage devices to be added to a live system and having the new space available on all of the existing file systems in that pool immediately. By combining the traditionally separate roles, ZFS is able to overcome previous limitations that prevented RAID groups being able to grow. Each top level device in a pool is called a *vdev*, which can be a simple disk or a RAID transformation such as a mirror or RAID-Z array. ZFS file systems (called *datasets*) each have access to the combined free space of the entire pool. As blocks are allocated from the pool, the space available to each file system decreases. This approach avoids the common pitfall with extensive partitioning where free space becomes fragmented across the partitions.

pool	<p>A storage <i>pool</i> is the most basic building block of ZFS. A pool is made up of one or more <i>vdevs</i>, the underlying devices that store the data. A pool is then used to create one or more file systems (<i>datasets</i>) or block devices (<i>volumes</i>). These <i>datasets</i> and <i>volumes</i> share the pool of remaining free space. Each pool is uniquely identified by a name and a GUID. The features available are determined by the ZFS version number on the pool.</p> <div><p>Note</p><p>FreeBSD 9.0 and 9.1 include support for ZFS version 28. Later versions use ZFS version 5000 with feature flags. The new feature flags system allows greater cross-compatibility with other implementations of ZFS.</p></div>
vdev Types	<p>A pool is made up of one or more <i>vdevs</i>, which themselves can be a single disk or a group of disks, in the case of a RAID transform. When multiple <i>vdevs</i> are used, ZFS spreads data across the <i>vdevs</i> to increase performance and maximize usable space.</p> <ul style="list-style-type: none">• <i>Disk</i> - The most basic type of <i>vdev</i> is a standard block device. This can be an entire disk (such as <code>/dev/ada0</code> or <code>/dev/da0</code>) or a partition (<code>/dev/ada0p3</code>). On FreeBSD, there is no performance penalty for using a partition rather than the entire disk. This differs from recommendations made by the Solaris documentation.• <i>File</i> - In addition to disks, ZFS pools can be backed by regular files, this is especially useful for testing and experimentation. Use the full path to the file as the device path in <code>zpool create</code> . All <i>vdevs</i> must be at least 128 MB in size.• <i>Mirror</i> - When creating a mirror, specify the <code>mirror</code> keyword followed by the list of member devices for the mirror. A mirror consists of two or more devices,

all data will be written to all member devices. A mirror vdev will only hold as much data as its smallest member. A mirror vdev can withstand the failure of all but one of its members without losing any data.



Note

A regular single disk vdev can be upgraded to a mirror vdev at any time with `zpool attach`.

- **RAID-Z** - ZFS implements RAID-Z, a variation on standard RAID-5 that offers better distribution of parity and eliminates the “RAID-5 write hole” in which the data and parity information become inconsistent after an unexpected restart. ZFS supports three levels of RAID-Z which provide varying levels of redundancy in exchange for decreasing levels of usable storage. The types are named RAID-Z1 through RAID-Z3 based on the number of parity devices in the array and the number of disks which can fail while the pool remains operational.

In a RAID-Z1 configuration with four disks, each 1 TB, usable storage is 3 TB and the pool will still be able to operate in degraded mode with one faulted disk. If an additional disk goes offline before the faulted disk is replaced and resilvered, all data in the pool can be lost.

In a RAID-Z3 configuration with eight disks of 1 TB, the volume will provide 5 TB of usable space and still be able to operate with three faulted disks. Sun™ recommends no more than nine disks in a single vdev. If the configuration has more disks, it is recommended to divide them into separate vdevs and the pool data will be striped across them.

A configuration of two RAID-Z2 vdevs consisting of 8 disks each would create something similar to a RAID-60 array. A RAID-Z group's storage capacity is approximately the size of the smallest disk multiplied by the number of non-parity disks. Four 1 TB disks in RAID-Z1 has an effective size of approximately 3 TB, and an array of eight 1 TB disks in RAID-Z3 will yield 5 TB of usable space.

- **Spare** - ZFS has a special pseudo-vdev type for keeping track of available hot spares. Note that installed hot spares are not deployed automatically; they must manually be configured to replace the failed device using `zfs replace`.

	<ul style="list-style-type: none"> • <i>Log</i> - ZFS Log Devices, also known as ZFS Intent Log (ZIL) move the intent log from the regular pool devices to a dedicated device, typically an SSD. Having a dedicated log device can significantly improve the performance of applications with a high volume of synchronous writes, especially databases. Log devices can be mirrored, but RAID-Z is not supported. If multiple log devices are used, writes will be load balanced across them. • <i>Cache</i> - Adding a cache vdev to a pool will add the storage of the cache to the L2ARC. Cache devices cannot be mirrored. Since a cache device only stores additional copies of existing data, there is no risk of data loss.
Transaction Group (TXG)	<p>Transaction Groups are the way changed blocks are grouped together and eventually written to the pool. Transaction groups are the atomic unit that ZFS uses to assert consistency. Each transaction group is assigned a unique 64-bit consecutive identifier. There can be up to three active transaction groups at a time, one in each of these three states:</p> <ul style="list-style-type: none"> • <i>Open</i> - When a new transaction group is created, it is in the open state, and accepts new writes. There is always a transaction group in the open state, however the transaction group may refuse new writes if it has reached a limit. Once the open transaction group has reached a limit, or the vdev limit vdev limit [392] has been reached, the transaction group advances to the next state. • <i>Quiescing</i> - A short state that allows any pending operations to finish while not blocking the creation of a new open transaction group. Once all of the transactions in the group have completed, the transaction group advances to the final state. • <i>Syncing</i> - All of the data in the transaction group is written to stable storage. This process will in turn modify other data, such as metadata and space maps, that will also need to be written to stable storage. The process of syncing involves multiple passes. The first, all of the changed data blocks, is the biggest, followed by the metadata, which may take multiple passes to complete. Since allocating space for the data blocks generates new metadata, the syncing state cannot finish until a pass completes that does not allocate any additional space. The syncing state is also where <i>synctasks</i> are completed. Synctasks are administrative operations, such as creating or destroying snapshots and datasets, that modify the uberblock are completed. Once the sync state is complete, the transaction group in the quiescing state is advanced to the syncing state.

	<p>All administrative functions, such as snapshot are written as part of the transaction group. When a synctask is created, it is added to the currently open transaction group, and that group is advanced as quickly as possible to the syncing state to reduce the latency of administrative commands.</p>
Adaptive Replacement Cache (ARC)	<p>ZFS uses an Adaptive Replacement Cache (ARC), rather than a more traditional Least Recently Used (LRU) cache. An LRU cache is a simple list of items in the cache, sorted by when each object was most recently used. New items are added to the top of the list. When the cache is full, items from the bottom of the list are evicted to make room for more active objects. An ARC consists of four lists; the Most Recently Used (MRU) and Most Frequently Used (MFU) objects, plus a ghost list for each. These ghost lists track recently evicted objects to prevent them from being added back to the cache. This increases the cache hit ratio by avoiding objects that have a history of only being used occasionally. Another advantage of using both an MRU and MFU is that scanning an entire file system would normally evict all data from an MRU or LRU cache in favor of this freshly accessed content. With ZFS, there is also an MFU that only tracks the most frequently used objects, and the cache of the most commonly accessed blocks remains.</p>
L2ARC	<p>L2ARC is the second level of the ZFS caching system. The primary ARC is stored in RAM. Since the amount of available RAM is often limited, ZFS can also use cache vdevs [396]. Solid State Disks (SSDs) are often used as these cache devices due to their higher speed and lower latency compared to traditional spinning disks. L2ARC is entirely optional, but having one will significantly increase read speeds for files that are cached on the SSD instead of having to be read from the regular disks. L2ARC can also speed up deduplication because a DDT that does not fit in RAM but does fit in the L2ARC will be much faster than a DDT that must be read from disk. The rate at which data is added to the cache devices is limited to prevent prematurely wearing out SSDs with too many writes. Until the cache is full (the first block has been evicted to make room), writing to the L2ARC is limited to the sum of the write limit and the boost limit, and afterwards limited to the write limit. A pair of <code>sysctl(8)</code> values control these rate limits. <code>vfs.zfs.l2arc_write_max</code> [392] controls how many bytes are written to the cache per second, while <code>vfs.zfs.l2arc_write_boost</code> [392] adds to this limit during the “Turbo Warmup Phase” (Write Boost).</p>
ZIL	<p>ZIL accelerates synchronous transactions by using storage devices like SSDs that are faster than those used in the main storage pool. When an application requests a synchronous write (a guarantee that the data has been safely stored to disk rather than merely cached to be written later), the data is written to the faster ZIL stor-</p>

	age, then later flushed out to the regular disks. This greatly reduces latency and improves performance. Only synchronous workloads like databases will benefit from a ZIL. Regular asynchronous writes such as copying files will not use the ZIL at all.
Copy-On-Write	Unlike a traditional file system, when data is overwritten on ZFS, the new data is written to a different block rather than overwriting the old data in place. Only when this write is complete is the metadata then updated to point to the new location. In the event of a shorn write (a system crash or power loss in the middle of writing a file), the entire original contents of the file are still available and the incomplete write is discarded. This also means that ZFS does not require a fsck(8) after an unexpected shutdown.
Dataset	<i>Dataset</i> is the generic term for a ZFS file system, volume, snapshot or clone. Each dataset has a unique name in the format <i>poolname/path@snapshot</i> . The root of the pool is technically a dataset as well. Child datasets are named hierarchically like directories. For example, <i>mypool/home</i> , the home dataset, is a child of <i>mypool</i> and inherits properties from it. This can be expanded further by creating <i>mypool/home/user</i> . This grandchild dataset will inherit properties from the parent and grandparent. Properties on a child can be set to override the defaults inherited from the parents and grandparents. Administration of datasets and their children can be delegated .
File system	A ZFS dataset is most often used as a file system. Like most other file systems, a ZFS file system is mounted somewhere in the systems directory hierarchy and contains files and directories of its own with permissions, flags, and other metadata.
Volume	In addition to regular file system datasets, ZFS can also create volumes, which are block devices. Volumes have many of the same features, including copy-on-write, snapshots, clones, and checksumming. Volumes can be useful for running other file system formats on top of ZFS, such as UFS virtualization, or exporting iSCSI extents.
Snapshot	The copy-on-write (COW) design of ZFS allows for nearly instantaneous, consistent snapshots with arbitrary names. After taking a snapshot of a dataset, or a recursive snapshot of a parent dataset that will include all child datasets, new data is written to new blocks, but the old blocks are not reclaimed as free space. The snapshot contains the original version of the file system, and the live file system contains any changes made since the snapshot was taken. No additional space is used. As new data is written to the live file system, new blocks are allocated to store this data. The apparent size of the snapshot will grow as the blocks are no longer used in the live file system, but only in the snapshot. These snapshots can be mounted read only to allow for the recov-

	<p>ery of previous versions of files. It is also possible to rollback a live file system to a specific snapshot, undoing any changes that took place after the snapshot was taken. Each block in the pool has a reference counter which keeps track of how many snapshots, clones, datasets, or volumes make use of that block. As files and snapshots are deleted, the reference count is decremented. When a block is no longer referenced, it is reclaimed as free space. Snapshots can also be marked with a hold. When a snapshot is held, any attempt to destroy it will return an EBUSY error. Each snapshot can have multiple holds, each with a unique name. The release command removes the hold so the snapshot can be deleted. Snapshots can be taken on volumes, but they can only be cloned or rolled back, not mounted independently.</p>
Clone	<p>Snapshots can also be cloned. A clone is a writable version of a snapshot, allowing the file system to be forked as a new dataset. As with a snapshot, a clone initially consumes no additional space. As new data is written to a clone and new blocks are allocated, the apparent size of the clone grows. When blocks are overwritten in the cloned file system or volume, the reference count on the previous block is decremented. The snapshot upon which a clone is based cannot be deleted because the clone depends on it. The snapshot is the parent, and the clone is the child. Clones can be <i>promoted</i>, reversing this dependency and making the clone the parent and the previous parent the child. This operation requires no additional space. Because the amount of space used by the parent and child is reversed, existing quotas and reservations might be affected.</p>
Checksum	<p>Every block that is allocated is also checksummed. The checksum algorithm used is a per-dataset property, see set. The checksum of each block is transparently validated as it is read, allowing ZFS to detect silent corruption. If the data that is read does not match the expected checksum, ZFS will attempt to recover the data from any available redundancy, like mirrors or RAID-Z). Validation of all checksums can be triggered with scrub. Checksum algorithms include:</p> <ul style="list-style-type: none"> • <code>fletcher2</code> • <code>fletcher4</code> • <code>sha256</code> <p>The <code>fletcher</code> algorithms are faster, but <code>sha256</code> is a strong cryptographic hash and has a much lower chance of collisions at the cost of some performance. Checksums can be disabled, but it is not recommended.</p>
Compression	<p>Each dataset has a compression property, which defaults to off. This property can be set to one of a number of compression algorithms. This will cause all new data that is written to the dataset to be compressed. Beyond a</p>

	<p>reduction in space used, read and write throughput often increases because fewer blocks are read or written.</p> <ul style="list-style-type: none"> • <i>LZ4</i> - Added in ZFS pool version 5000 (feature flags), LZ4 is now the recommended compression algorithm. LZ4 compresses approximately 50% faster than LZJB when operating on compressible data, and is over three times faster when operating on uncompressible data. LZ4 also decompresses approximately 80% faster than LZJB. On modern CPUs, LZ4 can often compress at over 500 MB/s, and decompress at over 1.5 GB/s (per single CPU core). <div style="border: 1px solid black; padding: 10px; margin: 10px 0;"> <div style="display: flex; align-items: center;">  <div style="margin-left: 10px;"> <p>Note</p> <p>LZ4 compression is only available after FreeBSD 9.2.</p> </div> </div> </div> <ul style="list-style-type: none"> • <i>LZJB</i> - The default compression algorithm. Created by Jeff Bonwick (one of the original creators of ZFS). LZJB offers good compression with less CPU overhead compared to GZIP. In the future, the default compression algorithm will likely change to LZ4. • <i>GZIP</i> - A popular stream compression algorithm available in ZFS. One of the main advantages of using GZIP is its configurable level of compression. When setting the <code>compress</code> property, the administrator can choose the level of compression, ranging from <code>gzip1</code>, the lowest level of compression, to <code>gzip9</code>, the highest level of compression. This gives the administrator control over how much CPU time to trade for saved disk space. • <i>ZLE</i> - Zero Length Encoding is a special compression algorithm that only compresses continuous runs of zeros. This compression algorithm is only useful when the dataset contains large blocks of zeros.
Copies	<p>When set to a value greater than 1, the <code>copies</code> property instructs ZFS to maintain multiple copies of each block in the File System or Volume. Setting this property on important datasets provides additional redundancy from which to recover a block that does not match its checksum. In pools without redundancy, the <code>copies</code> feature is the only form of redundancy. The <code>copies</code> feature can recover from a single bad sector or other forms of minor corruption, but it does not protect the pool from the loss of an entire disk.</p>
Deduplication	<p>Checksums make it possible to detect duplicate blocks of data as they are written. With deduplication, the reference count of an existing, identical block is increased, saving storage space. To detect duplicate blocks, a dedu-</p>

	<p>plication table (DDT) is kept in memory. The table contains a list of unique checksums, the location of those blocks, and a reference count. When new data is written, the checksum is calculated and compared to the list. If a match is found, the existing block is used. The SHA256 checksum algorithm is used with deduplication to provide a secure cryptographic hash. Deduplication is tunable. If <code>dedup</code> is on, then a matching checksum is assumed to mean that the data is identical. If <code>dedup</code> is set to <code>verify</code>, then the data in the two blocks will be checked byte-for-byte to ensure it is actually identical. If the data is not identical, the hash collision will be noted and the two blocks will be stored separately. Because DDT must store the hash of each unique block, it consumes a very large amount of memory. A general rule of thumb is 5-6 GB of ram per 1 TB of deduplicated data). In situations where it is not practical to have enough RAM to keep the entire DDT in memory, performance will suffer greatly as the DDT must be read from disk before each new block is written. Deduplication can use L2ARC to store the DDT, providing a middle ground between fast system memory and slower disks. Consider using compression instead, which often provides nearly as much space savings without the additional memory requirement.</p>
Scrub	<p>Instead of a consistency check like <code>fsck(8)</code>, ZFS has <code>scrub.scrub</code> reads all data blocks stored on the pool and verifies their checksums against the known good checksums stored in the metadata. A periodic check of all the data stored on the pool ensures the recovery of any corrupted blocks before they are needed. A scrub is not required after an unclean shutdown, but is recommended at least once every three months. The checksum of each block is verified as blocks are read during normal use, but a scrub makes certain that even infrequently used blocks are checked for silent corruption. Data security is improved, especially in archival storage situations. The relative priority of scrub can be adjusted with <code>vfs.zfs.scrub_delay</code> to prevent the scrub from degrading the performance of other workloads on the pool.</p>
Dataset Quota	<p>ZFS provides very fast and accurate dataset, user, and group space accounting in addition to quotas and space reservations. This gives the administrator fine grained control over how space is allocated and allows space to be reserved for critical file systems.</p> <p>ZFS supports different types of quotas: the dataset quota, the reference quota (refquota), the user quota, and the group quota.</p> <p>Quotas limit the amount of space that a dataset and all of its descendants, including snapshots of the dataset, child datasets, and the snapshots of those datasets, can consume.</p>

	<div>  <div> <p>Note</p> <p>Quotas cannot be set on volumes, as the <code>volsize</code> property acts as an implicit quota.</p> </div> </div>
Reference Quota	A reference quota limits the amount of space a dataset can consume by enforcing a hard limit. However, this hard limit includes only space that the dataset references and does not include space used by descendants, such as file systems or snapshots.
User Quota	User quotas are useful to limit the amount of space that can be used by the specified user.
Group Quota	The group quota limits the amount of space that a specified group can consume.
Dataset Reservation	<p>The <code>reservation</code> property makes it possible to guarantee a minimum amount of space for a specific dataset and its descendants. If a 10 GB reservation is set on <code>storage/home/bob</code>, and another dataset tries to use all of the free space, at least 10 GB of space is reserved for this dataset. If a snapshot is taken of <code>storage/home/bob</code>, the space used by that snapshot is counted against the reservation. The <code>refreservation</code> property works in a similar way, but it <i>excludes</i> descendants like snapshots.</p> <p>Reservations of any sort are useful in many situations, such as planning and testing the suitability of disk space allocation in a new system, or ensuring that enough space is available on file systems for audio logs or system recovery procedures and files.</p>
Reference Reservation	The <code>refreservation</code> property makes it possible to guarantee a minimum amount of space for the use of a specific dataset <i>excluding</i> its descendants. This means that if a 10 GB reservation is set on <code>storage/home/bob</code> , and another dataset tries to use all of the free space, at least 10 GB of space is reserved for this dataset. In contrast to a regular <code>reservation</code> , space used by snapshots and descendant datasets is not counted against the reservation. For example, if a snapshot is taken of <code>storage/home/bob</code> , enough disk space must exist outside of the <code>refreservation</code> amount for the operation to succeed. Descendants of the main data set are not counted in the <code>refreservation</code> amount and so do not encroach on the space set.
Resilver	When a disk fails and is replaced, the new disk must be filled with the data that was lost. The process of using the parity information distributed across the remaining drives to calculate and write the missing data to the new drive is called <i>resilvering</i> .

Online	A pool or vdev in the <code>Online</code> state has all of its member devices connected and fully operational. Individual devices in the <code>Online</code> state are functioning normally.
Offline	Individual devices can be put in an <code>Offline</code> state by the administrator if there is sufficient redundancy to avoid putting the pool or vdev into a <code>Faulted</code> state. An administrator may choose to offline a disk in preparation for replacing it, or to make it easier to identify.
Degraded	A pool or vdev in the <code>Degraded</code> state has one or more disks that have been disconnected or have failed. The pool is still usable, but if additional devices fail, the pool could become unrecoverable. Reconnecting the missing devices or replacing the failed disks will return the pool to an <code>Online</code> state after the reconnected or new device has completed the <code>Resilver</code> process.
Faulted	A pool or vdev in the <code>Faulted</code> state is no longer operational. The data on it can no longer be accessed. A pool or vdev enters the <code>Faulted</code> state when the number of missing or failed devices exceeds the level of redundancy in the vdev. If missing devices can be reconnected, the pool will return to a <code>Online</code> state. If there is insufficient redundancy to compensate for the number of failed disks, then the contents of the pool are lost and must be restored from backups.