NAME

zpoolconcepts - overview of ZFS storage pools

DESCRIPTION

Virtual Devices (vdevs)

A "virtual device" describes a single device or a collection of devices, organized according to certain performance and fault characteristics. The following virtual devices are supported:

disk A block device, typically located under */dev*. ZFS can use individual slices or partitions, though the recommended mode of operation is to use whole disks. A disk can be specified by a full path, or it can be a shorthand name (the relative portion of the path under */dev*). A whole disk can be specified by omitting the slice or partition designation. For example, *sda* is equivalent to */dev/sda*. When given a whole disk, ZFS automatically labels the disk, if necessary.

file A regular file. The use of files as a backing store is strongly discouraged. It is designed primarily for experimental purposes, as the fault tolerance of a file is only as good as the file system on which it resides. A file must be specified by a full path.

mirror A mirror of two or more devices. Data is replicated in an identical fashion across all components of a mirror. A mirror with *N* disks of size *X* can hold *X* bytes and can withstand *N-1* devices failing, without losing data.

raidz, **raidz1**, **raidz2**, **raidz3**

A distributed-parity layout, similar to RAID-5/6, with improved distribution of parity, and which does not suffer from the RAID-5/6 "write hole", (in which data and parity become inconsistent after a power loss). Data and parity is striped across all disks within a raidz group, though not necessarily in a consistent stripe width.

A raidz group can have single, double, or triple parity, meaning that the raidz group can sustain one, two, or three failures, respectively, without losing any data. The **raidz1** vdev type specifies a single-parity raidz group; the **raidz2** vdev type specifies a double-parity raidz group; and the **raidz3** vdev type specifies a triple-parity raidz group. The **raidz** vdev type is an alias for **raidz1**.

A raidz group with *N* disks of size *X* with *P* parity disks can hold approximately *(N-P)*X* bytes and can withstand *P* devices failing without losing data. The minimum number of devices in a raidz group is one more than the number of parity disks. The recommended number is between 3 and 9 to help increase performance.

draid, **draid1**, **draid2**, **draid3**

A variant of raidz that provides integrated distributed hot spares, allowing for faster resilvering,

while retaining the benefits of raidz. A dRAID vdev is constructed from multiple internal raidz groups, each with *D* data devices and *P* parity devices. These groups are distributed over all of the children in order to fully utilize the available disk performance.

Unlike raidz, dRAID uses a fixed stripe width (padding as necessary with zeros) to allow fully sequential resilvering. This fixed stripe width significantly affects both usable capacity and IOPS. For example, with the default *D=8* and *4 KiB* disk sectors the minimum allocation size is *32 KiB*. If using compression, this relatively large allocation size can reduce the effective compression ratio. When using ZFS volumes (zvols) and dRAID, the default of the **volblocksize** property is increased to account for the allocation size. If a dRAID pool will hold a significant amount of small blocks, it is recommended to also add a mirrored **special** vdev to store those blocks.

In regards to I/O, performance is similar to raidz since, for any read, all *D* data disks must be accessed. Delivered random IOPS can be reasonably approximated as **floor((N-S)/(D+P))*single_drive_IOPS**.

Like raidz, a dRAID can have single-, double-, or triple-parity. The **draid1**, **draid2**, and **draid3** types can be used to specify the parity level. The **draid** vdev type is an alias for **draid1**.

A dRAID with *N* disks of size *X*, *D* data disks per redundancy group, *P* parity level, and *S* distributed hot spares can hold approximately *(N-S)*(D/(D+P))*X* bytes and can withstand *P* devices failing without losing data.

draid[*parity*][**:***data***d**][**:***children***c**][**:***spares***s**]

A non-default dRAID configuration can be specified by appending one or more of the following optional arguments to the **draid** keyword:

parity The parity level (1-3).

data The number of data devices per redundancy group. In general, a smaller value of *D* will increase IOPS, improve the compression ratio, and speed up resilvering at the expense of total usable capacity. Defaults to *8*, unless *N-P-S* is less than *8*.

children The expected number of children. Useful as a cross-check when listing a large number of devices. An error is returned when the provided number of children differs.

spares The number of distributed hot spares. Defaults to zero.

spare A pseudo-vdev which keeps track of available hot spares for a pool. For more information, see the *Hot Spares* section.

log A separate intent log device. If more than one log device is specified, then writes are load-balanced between devices. Log devices can be mirrored. However, raidz vdev types are not

supported for the intent log. For more information, see the *Intent Log* section.

dedup A device solely dedicated for deduplication tables. The redundancy of this device should match the redundancy of the other normal devices in the pool. If more than one dedup device is specified, then allocations are load-balanced between those devices.

special A device dedicated solely for allocating various kinds of internal metadata, and optionally small file blocks. The redundancy of this device should match the redundancy of the other normal devices in the pool. If more than one special device is specified, then allocations are load-balanced between those devices.

For more information on special allocations, see the *Special Allocation Class* section.

cache A device used to cache storage pool data. A cache device cannot be configured as a mirror or raidz group. For more information, see the *Cache Devices* section.

Virtual devices cannot be nested arbitrarily. A mirror, raidz or draid virtual device can only be created with files or disks. Mirrors of mirrors or other such combinations are not allowed.

A pool can have any number of virtual devices at the top of the configuration (known as "root vdevs"). Data is dynamically distributed across all top-level devices to balance data among devices. As new virtual devices are added, ZFS automatically places data on the newly available devices.

Virtual devices are specified one at a time on the command line, separated by whitespace. Keywords like **mirror** and **raidz** are used to distinguish where a group ends and another begins. For example, the following creates a pool with two root vdevs, each a mirror of two disks:

zpool create *mypool* **mirror** *sda sdb* **mirror** *sdc sdd*

Device Failure and Recovery

ZFS supports a rich set of mechanisms for handling device failure and data corruption. All metadata and data is checksummed, and ZFS automatically repairs bad data from a good copy, when corruption is detected.

In order to take advantage of these features, a pool must make use of some form of redundancy, using either mirrored or raidz groups. While ZFS supports running in a non-redundant configuration, where each root vdev is simply a disk or file, this is strongly discouraged. A single case of bit corruption can render some or all of your data unavailable.

A pool’s health status is described by one of three states: **online**, **degraded**, or **faulted**. An online pool has all devices operating normally. A degraded pool is one in which one or more devices have failed,

but the data is still available due to a redundant configuration. A faulted pool has corrupted metadata, or one or more faulted devices, and insufficient replicas to continue functioning.

The health of the top-level vdev, such as a mirror or raidz device, is potentially impacted by the state of its associated vdevs or component devices. A top-level vdev or component device is in one of the following states:

DEGRADED One or more top-level vdevs is in the degraded state because one or more component devices are offline. Sufficient replicas exist to continue functioning.

One or more component devices is in the degraded or faulted state, but sufficient replicas exist to continue functioning. The underlying conditions are as follows:

- The number of checksum errors exceeds acceptable levels and the device is degraded as an indication that something may be wrong. ZFS continues to use the device as necessary.
- The number of I/O errors exceeds acceptable levels. The device could not be marked as faulted because there are insufficient replicas to continue functioning.

FAULTED One or more top-level vdevs is in the faulted state because one or more component devices are offline. Insufficient replicas exist to continue functioning.

One or more component devices is in the faulted state, and insufficient replicas exist to continue functioning. The underlying conditions are as follows:

- The device could be opened, but the contents did not match expected values.
- The number of I/O errors exceeds acceptable levels and the device is faulted to prevent further use of the device.

OFFLINE The device was explicitly taken offline by the **zpool offline** command.

ONLINE The device is online and functioning.

REMOVED The device was physically removed while the system was running. Device removal detection is hardware-dependent and may not be supported on all platforms.

UNAVAIL The device could not be opened. If a pool is imported when a device was unavailable, then the device will be identified by a unique identifier instead of its path since the path was never correct in the first place.

Checksum errors represent events where a disk returned data that was expected to be correct, but was not. In other words, these are instances of silent data corruption. The checksum errors are reported in

zpool status and **zpool events**. When a block is stored redundantly, a damaged block may be reconstructed (e.g. from raidz parity or a mirrored copy). In this case, ZFS reports the checksum error against the disks that contained damaged data. If a block is unable to be reconstructed (e.g. due to 3 disks being damaged in a raidz2 group), it is not possible to determine which disks were silently corrupted. In this case, checksum errors are reported for all disks on which the block is stored.

If a device is removed and later re-attached to the system, ZFS attempts to bring the device online automatically. Device attachment detection is hardware-dependent and might not be supported on all platforms.

Hot Spares

ZFS allows devices to be associated with pools as "hot spares". These devices are not actively used in the pool. But, when an active device fails, it is automatically replaced by a hot spare. To create a pool with hot spares, specify a **spare** vdev with any number of devices. For example,

zpool create *pool* **mirror** *sda sdb* **spare** *sdc sdd*

Spares can be shared across multiple pools, and can be added with the **zpool add** command and removed with the **zpool remove** command. Once a spare replacement is initiated, a new **spare** vdev is created within the configuration that will remain there until the original device is replaced. At this point, the hot spare becomes available again, if another device fails.

If a pool has a shared spare that is currently being used, the pool cannot be exported, since other pools may use this shared spare, which may lead to potential data corruption.

Shared spares add some risk. If the pools are imported on different hosts, and both pools suffer a device failure at the same time, both could attempt to use the spare at the same time. This may not be detected, resulting in data corruption.

An in-progress spare replacement can be cancelled by detaching the hot spare. If the original faulted device is detached, then the hot spare assumes its place in the configuration, and is removed from the spare list of all active pools.

The **draid** vdev type provides distributed hot spares. These hot spares are named after the dRAID vdev they're a part of (**draid1**-*2*-*3* specifies spare *3* of vdev *2*, which is a single parity dRAID) and may only be used by that dRAID vdev. Otherwise, they behave the same as normal hot spares.

Spares cannot replace log devices.

Intent Log

The ZFS Intent Log (ZIL) satisfies POSIX requirements for synchronous transactions. For instance,

databases often require their transactions to be on stable storage devices when returning from a system call. NFS and other applications can also use fsync(2) to ensure data stability. By default, the intent log is allocated from blocks within the main pool. However, it might be possible to get better performance using separate intent log devices such as NVRAM or a dedicated disk. For example:

zpool create *pool sda sdb* **log** *sdc*

Multiple log devices can also be specified, and they can be mirrored. See the *EXAMPLES* section for an example of mirroring multiple log devices.

Log devices can be added, replaced, attached, detached, and removed. In addition, log devices are imported and exported as part of the pool that contains them. Mirrored devices can be removed by specifying the top-level mirror vdev.

Cache Devices

Devices can be added to a storage pool as "cache devices". These devices provide an additional layer of caching between main memory and disk. For read-heavy workloads, where the working set size is much larger than what can be cached in main memory, using cache devices allows much more of this working set to be served from low latency media. Using cache devices provides the greatest performance improvement for random read-workloads of mostly static content.

To create a pool with cache devices, specify a **cache** vdev with any number of devices. For example:

zpool create *pool sda sdb* **cache** *sdc sdd*

Cache devices cannot be mirrored or part of a raidz configuration. If a read error is encountered on a cache device, that read I/O is reissued to the original storage pool device, which might be part of a mirrored or raidz configuration.

The content of the cache devices is persistent across reboots and restored asynchronously when importing the pool in L2ARC (persistent L2ARC). This can be disabled by setting **l2arc_rebuild_enabled=0**. For cache devices smaller than *1 GiB*, ZFS does not write the metadata structures required for rebuilding the L2ARC, to conserve space. This can be changed with **l2arc_rebuild_blocks_min_l2size**. The cache device header (*512 B*) is updated even if no metadata structures are written. Setting **l2arc_headroom=0** will result in scanning the full-length ARC lists for cacheable content to be written in L2ARC (persistent ARC). If a cache device is added with **zpool add**, its label and header will be overwritten and its contents will not be restored in L2ARC, even if the device was previously part of the pool. If a cache device is onlined with **zpool online**, its contents will be restored in L2ARC. This is useful in case of memory pressure, where the contents of the cache device are not fully restored in L2ARC. The user can off- and online the cache device when there is less memory pressure, to fully restore its contents to L2ARC.

Pool checkpoint

Before starting critical procedures that include destructive actions (like **zfs destroy**), an administrator can checkpoint the pool's state and, in the case of a mistake or failure, rewind the entire pool back to the checkpoint. Otherwise, the checkpoint can be discarded when the procedure has completed successfully.

A pool checkpoint can be thought of as a pool-wide snapshot and should be used with care as it contains every part of the pool's state, from properties to vdev configuration. Thus, certain operations are not allowed while a pool has a checkpoint. Specifically, vdev removal/attach/detach, mirror splitting, and changing the pool's GUID. Adding a new vdev is supported, but in the case of a rewind it will have to be added again. Finally, users of this feature should keep in mind that scrubs in a pool that has a checkpoint do not repair checkpointed data.

To create a checkpoint for a pool:

zpool checkpoint *pool*

To later rewind to its checkpointed state, you need to first export it and then rewind it during import:

zpool export *pool*

zpool import --rewind-to-checkpoint *pool*

To discard the checkpoint from a pool:

zpool checkpoint -d *pool*

Dataset reservations (controlled by the **reservation** and **refreservation** properties) may be unenforceable while a checkpoint exists, because the checkpoint is allowed to consume the dataset's reservation. Finally, data that is part of the checkpoint but has been freed in the current state of the pool won't be scanned during a scrub.

Special Allocation Class

Allocations in the special class are dedicated to specific block types. By default, this includes all metadata, the indirect blocks of user data, and any deduplication tables. The class can also be provisioned to accept small file blocks.

A pool must always have at least one normal (non-**dedup**/-**special**) vdev before other devices can be assigned to the special class. If the **special** class becomes full, then allocations intended for it will spill back into the normal class.

Deduplication tables can be excluded from the special class by unsetting the **zfs_ddt_data_is_special** ZFS module parameter.

Inclusion of small file blocks in the special class is opt-in. Each dataset can control the size of small file blocks allowed in the special class by setting the **special_small_blocks** property to nonzero. See zfsprops(7) for more info on this property.