NAME

zpool-features - description of ZFS pool features

DESCRIPTION

ZFS pool on-disk format versions are specified via "features" which replace the old on-disk format numbers (the last supported on-disk format number is 28). To enable a feature on a pool use the **zpool upgrade**, or set the **feature**@*feature-name* property to **enabled**. Please also see the *Compatibility feature sets* section for information on how sets of features may be enabled together.

The pool format does not affect file system version compatibility or the ability to send file systems between pools.

Since most features can be enabled independently of each other, the on-disk format of the pool is specified by the set of all features marked as **active** on the pool. If the pool was created by another software version this set may include unsupported features.

Identifying features

Every feature has a GUID of the form *com.example*:*feature-name*. The reversed DNS name ensures that the feature's GUID is unique across all ZFS implementations. When unsupported features are encountered on a pool they will be identified by their GUIDs. Refer to the documentation for the ZFS implementation that created the pool for information about those features.

Each supported feature also has a short name. By convention a feature's short name is the portion of its GUID which follows the ':' (i.e. *com.example*:*feature-name* would have the short name *feature-name*), however a feature's short name may differ across ZFS implementations if following the convention would result in name conflicts.

Feature states

Features can be in one of three states:

active This feature's on-disk format changes are in effect on the pool. Support for this feature is required to import the pool in read-write mode. If this feature is not read-only compatible, support is also required to import the pool in read-only mode (see *Read-only compatibility*).

enabled An administrator has marked this feature as enabled on the pool, but the feature's on-disk format changes have not been made yet. The pool can still be imported by software that does not support this feature, but changes may be made to the on-disk format at any time which will move the feature to the **active** state. Some features may support returning to the **enabled** state after becoming **active**. See feature-specific documentation for details.

disabled This feature's on-disk format changes have not been made and will not be made unless an administrator moves the feature to the **enabled** state. Features cannot be disabled once they have been enabled.

The state of supported features is exposed through pool properties of the form **feature**@*short-name*.

Read-only compatibility

Some features may make on-disk format changes that do not interfere with other software's ability to read from the pool. These features are referred to as "read-only compatible". If all unsupported features on a pool are read-only compatible, the pool can be imported in read-only mode by setting the **readonly** property during import (see zpool-import(8) for details on importing pools).

Unsupported features

For each unsupported feature enabled on an imported pool, a pool property named **unsupported**@*feature-name* will indicate why the import was allowed despite the unsupported feature. Possible values for this property are:

inactive The feature is in the **enabled** state and therefore the pool's on-disk format is still compatible with software that does not support this feature.

readonly The feature is read-only compatible and the pool has been imported in read-only mode.

Feature dependencies

Some features depend on other features being enabled in order to function. Enabling a feature will automatically enable any features it depends on.

Compatibility feature sets

It is sometimes necessary for a pool to maintain compatibility with a specific on-disk format, by enabling and disabling particular features. The **compatibility** feature facilitates this by allowing feature sets to be read from text files. When set to **off** (the default), compatibility feature sets are disabled (i.e. all features are enabled); when set to **legacy**, no features are enabled. When set to a comma-separated list of filenames (each filename may either be an absolute path, or relative to */etc/zfs/compatibility.d* or */usr/share/zfs/compatibility.d*), the lists of requested features are read from those files, separated by whitespace and/or commas. Only features present in all files are enabled.

Simple sanity checks are applied to the files: they must be between 1 B and 16 KiB in size, and must end with a newline character.

The requested features are applied when a pool is created using **zpool create -o compatibility**=<?> and controls which features are enabled when using **zpool upgrade**. **zpool status** will not show a warning

about disabled features which are not part of the requested feature set.

The special value **legacy** prevents any features from being enabled, either via **zpool upgrade** or **zpool set feature**@*feature-name*=**enabled**. This setting also prevents pools from being upgraded to newer on-disk versions. This is a safety measure to prevent new features from being accidentally enabled, breaking compatibility.

By convention, compatibility files in */usr/share/zfs/compatibility.d* are provided by the distribution, and include feature sets supported by important versions of popular distributions, and feature sets commonly supported at the start of each year. Compatibility files in */etc/zfs/compatibility.d*, if present, will take precedence over files with the same name in */usr/share/zfs/compatibility.d*.

If an unrecognized feature is found in these files, an error message will be shown. If the unrecognized feature is in a file in */etc/zfs/compatibility.d*, this is treated as an error and processing will stop. If the unrecognized feature is under */usr/share/zfs/compatibility.d*, this is treated as a warning and processing will continue. This difference is to allow distributions to include features which might not be recognized by the currently-installed binaries.

Compatibility files may include comments: any text from '#' to the end of the line is ignored.

Example:

example# **cat** */usr/share/zfs/compatibility.d/grub2*
Features which are supported by GRUB2
async_destroy
bookmarks
embedded_data
empty_bpobj
enabled_txg
extensible_dataset
filesystem_limits
hole_birth
large_blocks
livelist
lz4_compress
spacemap_histogram
zpool_checkpoint

example# **zpool create -o compatibility**=*grub2 bootpool vdev*

See zpool-create(8) and zpool-upgrade(8) for more information on how these commands are affected by

feature sets.

FEATURES

The following features are supported on this system:

allocation_classes

GUID	**org.zfsonlinux:allocation_classes**
READ-ONLY COMPATIBLE	yes

This feature enables support for separate allocation classes.

This feature becomes **active** when a dedicated allocation class vdev (dedup or special) is created with the **zpool create** or **zpool add** commands. With device removal, it can be returned to the **enabled** state if all the dedicated allocation class vdevs are removed.

async_destroy

GUID	**com.delphix:async_destroy**
READ-ONLY COMPATIBLE	yes

Destroying a file system requires traversing all of its data in order to return its used space to the pool. Without **async_destroy**, the file system is not fully removed until all space has been reclaimed. If the destroy operation is interrupted by a reboot or power outage, the next attempt to open the pool will need to complete the destroy operation synchronously.

When **async_destroy** is enabled, the file system's data will be reclaimed by a background process, allowing the destroy operation to complete without traversing the entire file system. The background process is able to resume interrupted destroys after the pool has been opened, eliminating the need to finish interrupted destroys as part of the open operation. The amount of space remaining to be reclaimed by the background process is available through the **freeing** property.

This feature is only **active** while **freeing** is non-zero.

blake3

GUID	**org.openzfs:blake3**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the use of the BLAKE3 hash algorithm for checksum and dedup. BLAKE3 is a secure hash algorithm focused on high performance.

When the **blake3** feature is set to **enabled**, the administrator can turn on the **blake3** checksum on any dataset using **zfs set checksum**=**blake3** *dset* (see zfs-set(8)). This feature becomes **active** once a **checksum** property has been set to **blake3**, and will return to being **enabled** once all filesystems that have ever had their checksum set to **blake3** are destroyed.

block_cloning

GUID	**com.fudosecurity:block_cloning**
READ-ONLY COMPATIBLE	yes

When this feature is enabled ZFS will use block cloning for operations like **copy_file_range**(*2*). Block cloning allows to create multiple references to a single block. It is much faster than copying the data (as the actual data is neither read nor written) and takes no additional space. Blocks can be cloned across datasets under some conditions (like disabled encryption and equal **recordsize**).

This feature becomes **active** when first block is cloned. When the last cloned block is freed, it goes back to the enabled state.

bookmarks

GUID	**com.delphix:bookmarks**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	yes

This feature enables use of the **zfs bookmark** command.

This feature is **active** while any bookmarks exist in the pool. All bookmarks in the pool can be listed by running **zfs list -t bookmark -r** *poolname*.

bookmark_v2

GUID	**com.datto:bookmark_v2**
DEPENDENCIES	**bookmark**, **extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the creation and management of larger bookmarks which are needed for other features in ZFS.

This feature becomes **active** when a v2 bookmark is created and will be returned to the **enabled** state when all v2 bookmarks are destroyed.

bookmark_written

GUID **com.delphix:bookmark_written**
DEPENDENCIES **bookmark**, **extensible_dataset**, **bookmark_v2**
READ-ONLY COMPATIBLE no

This feature enables additional bookmark accounting fields, enabling the **written**#*bookmark* property (space written since a bookmark) and estimates of send stream sizes for incrementals from bookmarks.

This feature becomes **active** when a bookmark is created and will be returned to the **enabled** state when all bookmarks with these fields are destroyed.

device_rebuild

GUID **org.openzfs:device_rebuild**
READ-ONLY COMPATIBLE yes

This feature enables the ability for the **zpool attach** and **zpool replace** commands to perform sequential reconstruction (instead of healing reconstruction) when resilvering.

Sequential reconstruction resilvers a device in LBA order without immediately verifying the checksums. Once complete, a scrub is started, which then verifies the checksums. This approach allows full redundancy to be restored to the pool in the minimum amount of time. This two-phase approach will take longer than a healing resilver when the time to verify the checksums is included. However, unless there is additional pool damage, no checksum errors should be reported by the scrub. This feature is incompatible with raidz configurations. This feature becomes **active** while a sequential resilver is in progress, and returns to **enabled** when the resilver completes.

device_removal

GUID **com.delphix:device_removal**
READ-ONLY COMPATIBLE no

This feature enables the **zpool remove** command to remove top-level vdevs, evacuating them to reduce the total size of the pool.

This feature becomes **active** when the **zpool remove** command is used on a top-level vdev, and will never return to being **enabled**.

draid

GUID **org.openzfs:draid**
READ-ONLY COMPATIBLE no

This feature enables use of the **draid** vdev type. dRAID is a variant of RAID-Z which provides integrated distributed hot spares that allow faster resilvering while retaining the benefits of RAID-Z. Data, parity, and spare space are organized in redundancy groups and distributed evenly over all of the devices.

This feature becomes **active** when creating a pool which uses the **draid** vdev type, or when adding a new **draid** vdev to an existing pool.

edonr

GUID	**org.illumos:edonr**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the use of the Edon-R hash algorithm for checksum, including for nopwrite (if compression is also enabled, an overwrite of a block whose checksum matches the data being written will be ignored). In an abundance of caution, Edon-R requires verification when used with dedup: **zfs set dedup**=**edonr**,**verify** (see zfs-set(8)).

Edon-R is a very high-performance hash algorithm that was part of the NIST SHA-3 competition. It provides extremely high hash performance (over 350% faster than SHA-256), but was not selected because of its unsuitability as a general purpose secure hash algorithm. This implementation utilizes the new salted checksumming functionality in ZFS, which means that the checksum is pre-seeded with a secret 256-bit random key (stored on the pool) before being fed the data block to be checksummed. Thus the produced checksums are unique to a given pool, preventing hash collision attacks on systems with dedup.

When the **edonr** feature is set to **enabled**, the administrator can turn on the **edonr** checksum on any dataset using **zfs set checksum**=**edonr** *dset* (see zfs-set(8)). This feature becomes **active** once a **checksum** property has been set to **edonr**, and will return to being **enabled** once all filesystems that have ever had their checksum set to **edonr** are destroyed.

embedded_data

GUID	**com.delphix:embedded_data**
READ-ONLY COMPATIBLE	no

This feature improves the performance and compression ratio of highly-compressible blocks. Blocks whose contents can compress to 112 bytes or smaller can take advantage of this feature.

When this feature is enabled, the contents of highly-compressible blocks are stored in the block "pointer" itself (a misnomer in this case, as it contains the compressed data, rather than a pointer

to its location on disk). Thus the space of the block (one sector, typically 512 B or 4 KiB) is saved, and no additional I/O is needed to read and write the data block. This feature becomes **active** as soon as it is enabled and will never return to being **enabled**.

empty_bpobj

GUID	**com.delphix:empty_bpobj**
READ-ONLY COMPATIBLE	yes

This feature increases the performance of creating and using a large number of snapshots of a single filesystem or volume, and also reduces the disk space required.

When there are many snapshots, each snapshot uses many Block Pointer Objects (bpobjs) to track blocks associated with that snapshot. However, in common use cases, most of these bpobjs are empty. This feature allows us to create each bpobj on-demand, thus eliminating the empty bpobjs.

This feature is **active** while there are any filesystems, volumes, or snapshots which were created after enabling this feature.

enabled_txg

GUID	**com.delphix:enabled_txg**
READ-ONLY COMPATIBLE	yes

Once this feature is enabled, ZFS records the transaction group number in which new features are enabled. This has no user-visible impact, but other features may depend on this feature.

This feature becomes **active** as soon as it is enabled and will never return to being **enabled**.

encryption

GUID	**com.datto:encryption**
DEPENDENCIES	**bookmark_v2**, **extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the creation and management of natively encrypted datasets.

This feature becomes **active** when an encrypted dataset is created and will be returned to the **enabled** state when all datasets that use this feature are destroyed.

extensible_dataset

GUID	**com.delphix:extensible_dataset**

READ-ONLY COMPATIBLE no

This feature allows more flexible use of internal ZFS data structures, and exists for other features to depend on.

This feature will be **active** when the first dependent feature uses it, and will be returned to the **enabled** state when all datasets that use this feature are destroyed.

filesystem_limits

GUID **com.joyent:filesystem_limits**
DEPENDENCIES **extensible_dataset**
READ-ONLY COMPATIBLE yes

This feature enables filesystem and snapshot limits. These limits can be used to control how many filesystems and/or snapshots can be created at the point in the tree on which the limits are set.

This feature is **active** once either of the limit properties has been set on a dataset and will never return to being **enabled**.

head_errlog

GUID **com.delphix:head_errlog**
READ-ONLY COMPATIBLE no

This feature enables the upgraded version of errlog, which required an on-disk error log format change. Now the error log of each head dataset is stored separately in the zap object and keyed by the head id. With this feature enabled, every dataset affected by an error block is listed in the output of **zpool status**. In case of encrypted filesystems with unloaded keys we are unable to check their snapshots or clones for errors and these will not be reported. An "access denied" error will be reported.

This feature becomes **active** as soon as it is enabled and will never return to being **enabled**.

hole_birth

GUID **com.delphix:hole_birth**
DEPENDENCIES **enabled_txg**
READ-ONLY COMPATIBLE no

This feature has/had bugs, the result of which is that, if you do a **zfs send -i** (or **-R**, since it uses **-i**) from an affected dataset, the receiving party will not see any checksum or other errors, but

the resulting destination snapshot will not match the source. Its use by **zfs send -i** has been disabled by default (see **send_holes_without_birth_time** in zfs(4)).

This feature improves performance of incremental sends (**zfs send -i**) and receives for objects with many holes. The most common case of hole-filled objects is zvols.

An incremental send stream from snapshot **A** to snapshot **B** contains information about every block that changed between **A** and **B**. Blocks which did not change between those snapshots can be identified and omitted from the stream using a piece of metadata called the "block birth time", but birth times are not recorded for holes (blocks filled only with zeroes). Since holes created after **A** cannot be distinguished from holes created before **A**, information about every hole in the entire filesystem or zvol is included in the send stream.

For workloads where holes are rare this is not a problem. However, when incrementally replicating filesystems or zvols with many holes (for example a zvol formatted with another filesystem) a lot of time will be spent sending and receiving unnecessary information about holes that already exist on the receiving side.

Once the **hole_birth** feature has been enabled the block birth times of all new holes will be recorded. Incremental sends between snapshots created after this feature is enabled will use this new metadata to avoid sending information about holes that already exist on the receiving side.

This feature becomes **active** as soon as it is enabled and will never return to being **enabled**.

large_blocks

GUID	**org.open-zfs:large_blocks**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature allows the record size on a dataset to be set larger than 128 KiB.

This feature becomes **active** once a dataset contains a file with a block size larger than 128 KiB, and will return to being **enabled** once all filesystems that have ever had their recordsize larger than 128 KiB are destroyed.

large_dnode

GUID	**org.zfsonlinux:large_dnode**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature allows the size of dnodes in a dataset to be set larger than 512 B. This feature becomes **active** once a dataset contains an object with a dnode larger than 512 B, which occurs as a result of setting the **dnodesize** dataset property to a value other than **legacy**. The feature will return to being **enabled** once all filesystems that have ever contained a dnode larger than 512 B are destroyed. Large dnodes allow more data to be stored in the bonus buffer, thus potentially improving performance by avoiding the use of spill blocks.

livelist

GUID	**com.delphix:livelist**
READ-ONLY COMPATIBLE	yes

This feature allows clones to be deleted faster than the traditional method when a large number of random/sparse writes have been made to the clone. All blocks allocated and freed after a clone is created are tracked by the the clone's livelist which is referenced during the deletion of the clone. The feature is activated when a clone is created and remains **active** until all clones have been destroyed.

log_spacemap

GUID	**com.delphix:log_spacemap**
DEPENDENCIES	**com.delphix:spacemap_v2**
READ-ONLY COMPATIBLE	yes

This feature improves performance for heavily-fragmented pools, especially when workloads are heavy in random-writes. It does so by logging all the metaslab changes on a single spacemap every TXG instead of scattering multiple writes to all the metaslab spacemaps.

This feature becomes **active** as soon as it is enabled and will never return to being **enabled**.

lz4_compress

GUID	**org.illumos:lz4_compress**
READ-ONLY COMPATIBLE	no

lz4 is a high-performance real-time compression algorithm that features significantly faster compression and decompression as well as a higher compression ratio than the older **lzjb** compression. Typically, **lz4** compression is approximately 50% faster on compressible data and 200% faster on incompressible data than **lzjb**. It is also approximately 80% faster on decompression, while giving approximately a 10% better compression ratio.

When the **lz4_compress** feature is set to **enabled**, the administrator can turn on **lz4** compression on any dataset on the pool using the zfs-set(8) command. All newly written metadata will be

compressed with the **lz4** algorithm.

This feature becomes **active** as soon as it is enabled and will never return to being **enabled**.

multi_vdev_crash_dump

GUID	**com.joyent:multi_vdev_crash_dump**
READ-ONLY COMPATIBLE	no

This feature allows a dump device to be configured with a pool comprised of multiple vdevs. Those vdevs may be arranged in any mirrored or raidz configuration.

When the **multi_vdev_crash_dump** feature is set to **enabled**, the administrator can use dumpadm(8) to configure a dump device on a pool comprised of multiple vdevs.

Under FreeBSD and Linux this feature is unused, but registered for compatibility. New pools created on these systems will have the feature **enabled** but will never transition to **active**, as this functionality is not required for crash dump support. Existing pools where this feature is **active** can be imported.

obsolete_counts

GUID	**com.delphix:obsolete_counts**
DEPENDENCIES	**device_removal**
READ-ONLY COMPATIBLE	yes

This feature is an enhancement of **device_removal**, which will over time reduce the memory used to track removed devices. When indirect blocks are freed or remapped, we note that their part of the indirect mapping is "obsolete" - no longer needed.

This feature becomes **active** when the **zpool remove** command is used on a top-level vdev, and will never return to being **enabled**.

project_quota

GUID	**org.zfsonlinux:project_quota**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	yes

This feature allows administrators to account the spaces and objects usage information against the project identifier (ID).

The project ID is an object-based attribute. When upgrading an existing filesystem, objects

without a project ID will be assigned a zero project ID. When this feature is enabled, newly created objects inherit their parent directories' project ID if the parent's inherit flag is set (via **chattr [+-]P** or **zfs project -s|-C**). Otherwise, the new object's project ID will be zero. An object's project ID can be changed at any time by the owner (or privileged user) via **chattr -p** *prjid* or **zfs project -p** *prjid*.

This feature will become **active** as soon as it is enabled and will never return to being **disabled**. Each filesystem will be upgraded automatically when remounted, or when a new file is created under that filesystem. The upgrade can also be triggered on filesystems via **zfs set version=current** *fs*. The upgrade process runs in the background and may take a while to complete for filesystems containing large amounts of files.

redaction_bookmarks

GUID	**com.delphix:redaction_bookmarks**
DEPENDENCIES	**bookmarks**, **extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the use of redacted **zfs send**s, which create redaction bookmarks storing the list of blocks redacted by the send that created them. For more information about redacted sends, see zfs-send(8).

redacted_datasets

GUID	**com.delphix:redacted_datasets**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the receiving of redacted **zfs send** streams, which create redacted datasets when received. These datasets are missing some of their blocks, and so cannot be safely mounted, and their contents cannot be safely read. For more information about redacted receives, see zfs-send(8).

resilver_defer

GUID	**com.datto:resilver_defer**
READ-ONLY COMPATIBLE	yes

This feature allows ZFS to postpone new resilvers if an existing one is already in progress. Without this feature, any new resilvers will cause the currently running one to be immediately restarted from the beginning.

This feature becomes **active** once a resilver has been deferred, and returns to being **enabled**

when the deferred resilver begins.

sha512

GUID	**org.illumos:sha512**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the use of the SHA-512/256 truncated hash algorithm (FIPS 180-4) for checksum and dedup. The native 64-bit arithmetic of SHA-512 provides an approximate 50% performance boost over SHA-256 on 64-bit hardware and is thus a good minimum-change replacement candidate for systems where hash performance is important, but these systems cannot for whatever reason utilize the faster **skein** and **edonr** algorithms.

When the **sha512** feature is set to **enabled**, the administrator can turn on the **sha512** checksum on any dataset using **zfs set checksum=sha512** *dset* (see zfs-set(8)). This feature becomes **active** once a **checksum** property has been set to **sha512**, and will return to being **enabled** once all filesystems that have ever had their checksum set to **sha512** are destroyed.

skein

GUID	**org.illumos:skein**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	no

This feature enables the use of the Skein hash algorithm for checksum and dedup. Skein is a high-performance secure hash algorithm that was a finalist in the NIST SHA-3 competition. It provides a very high security margin and high performance on 64-bit hardware (80% faster than SHA-256). This implementation also utilizes the new salted checksumming functionality in ZFS, which means that the checksum is pre-seeded with a secret 256-bit random key (stored on the pool) before being fed the data block to be checksummed. Thus the produced checksums are unique to a given pool, preventing hash collision attacks on systems with dedup.

When the **skein** feature is set to **enabled**, the administrator can turn on the **skein** checksum on any dataset using **zfs set checksum=skein** *dset* (see zfs-set(8)). This feature becomes **active** once a **checksum** property has been set to **skein**, and will return to being **enabled** once all filesystems that have ever had their checksum set to **skein** are destroyed.

spacemap_histogram

GUID	**com.delphix:spacemap_histogram**
READ-ONLY COMPATIBLE	yes

This features allows ZFS to maintain more information about how free space is organized within the pool. If this feature is **enabled**, it will be activated when a new space map object is created, or an existing space map is upgraded to the new format, and never returns back to being **enabled**.

spacemap_v2

GUID	**com.delphix:spacemap_v2**
READ-ONLY COMPATIBLE	yes

This feature enables the use of the new space map encoding which consists of two words (instead of one) whenever it is advantageous. The new encoding allows space maps to represent large regions of space more efficiently on-disk while also increasing their maximum addressable offset.

This feature becomes **active** once it is **enabled**, and never returns back to being **enabled**.

userobj_accounting

GUID	**org.zfsonlinux:userobj_accounting**
DEPENDENCIES	**extensible_dataset**
READ-ONLY COMPATIBLE	yes

This feature allows administrators to account the object usage information by user and group.

This feature becomes **active** as soon as it is enabled and will never return to being **enabled**. Each filesystem will be upgraded automatically when remounted, or when a new file is created under that filesystem. The upgrade can also be triggered on filesystems via **zfs set version=current** *fs*. The upgrade process runs in the background and may take a while to complete for filesystems containing large amounts of files.

vdev_zaps_v2

GUID	**com.klarasystems:vdev_zaps_v2**
READ-ONLY COMPATIBLE	no

This feature creates a ZAP object for the root vdev.

This feature becomes active after the next **zpool import** or **zpool reguid**. Properties can be retrieved or set on the root vdev using **zpool get** and **zpool set** with **root** as the vdev name which is an alias for **root-0**.

zilsaxattr

GUID **org.openzfs:zilsaxattr**
DEPENDENCIES **extensible_dataset**
READ-ONLY COMPATIBLE yes

This feature enables **xattr**=**sa** extended attribute logging in the ZIL. If enabled, extended attribute changes (both **xattrdir**=**dir** and **xattr**=**sa**) are guaranteed to be durable if either the dataset had **sync**=**always** set at the time the changes were made, or sync(2) is called on the dataset after the changes were made.

This feature becomes **active** when a ZIL is created for at least one dataset and will be returned to the **enabled** state when it is destroyed for all datasets that use this feature.

zpool_checkpoint

GUID **com.delphix:zpool_checkpoint**
READ-ONLY COMPATIBLE yes

This feature enables the **zpool checkpoint** command that can checkpoint the state of the pool at the time it was issued and later rewind back to it or discard it.

This feature becomes **active** when the **zpool checkpoint** command is used to checkpoint the pool. The feature will only return back to being **enabled** when the pool is rewound or the checkpoint has been discarded.

zstd_compress

GUID **org.freebsd:zstd_compress**
DEPENDENCIES **extensible_dataset**
READ-ONLY COMPATIBLE no

zstd is a high-performance compression algorithm that features a combination of high compression ratios and high speed. Compared to **gzip**, **zstd** offers slightly better compression at much higher speeds. Compared to **lz4**, **zstd** offers much better compression while being only modestly slower. Typically, **zstd** compression speed ranges from 250 to 500 MB/s per thread and decompression speed is over 1 GB/s per thread.

When the **zstd** feature is set to **enabled**, the administrator can turn on **zstd** compression of any dataset using **zfs set compress**=**zstd** *dset* (see zfs-set(8)). This feature becomes **active** once a **compress** property has been set to **zstd**, and will return to being **enabled** once all filesystems that have ever had their **compress** property set to **zstd** are destroyed.

SEE ALSO

zfs(8), zpool(8)