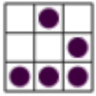


Aaron Toponce

{ 2012.12.14 }

ZFS Administration, Part IX- Copy-on-write

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

Before we can get into the practical administration of ZFS datasets, we need to understand exactly how ZFS is storing our data. So, this post will be theoretical, and cover a couple concepts that you will want to understand. Namely the the Merkle tree and copy-on-write. I'll try and keep it at a higher level that is easier to understand, without getting into C code system calls and memory allocations.

Merkle Trees

Merkle trees are nothing more than cryptographic hash trees, invented by Ralph Merkle. In order to understand a Merkle tree, we will start from the bottom, and work our way up the tree. Suppose you have 4 data blocks, block 0, block 1, block 2 and block 3. Each block is cryptographically hashed, and their hash is stored in a node, or "hash block". Each data block has a one-to-one relationship with their hash block. Let's assume the SHA-256 cryptographic hashing algorithm is the hash used. Suppose further that our four blocks hash to the following:

- Block 0- 888b19a43b151683c87895f6211d9f8640f97bdc8ef32f03dbe057c8f5e56d32 (hash block 0-0)
- Block 1- 4fac6dbe26e823ed6edf999c63fab3507119cf3cbfb56036511aa62e258c35b4 (hash block 0-1)
- Block 2- 446e21f212ab200933c4c9a0802e1ff0c410bbd75fca10168746fc49883096db (hash block 1-0)
- Block 3- 0591b59c1bdd9acd2847a202ddd02c3f14f9b5a049a5707c3279c1e967745ed4 (hash block 1-1)

The reason we cryptographically hash each block, is to ensure data integrity. If the block intentionally changes, then its SHA-256 hash should also change. If the the block was corrupted, then the hash would not change. Thus, we can cryptographically hash the block with the SHA-256 algorithm, and check to see if it matches its parent hash block. If it does, then we can be certain with a high degree of probability that the block was not corrupted. However, if the hash does not match, then under the same premise, the block is likely corrupted.

Hash trees are typically binary trees (one node has at most 2 children), although there is no requirement to make them so. Suppose in our example, our hash tree is a binary tree. In this case, hash blocks 0-0 and 0-1 would have a single parent, hash block 0. And hash blocks 1-0 and 1-1 would have a single parent, hash block 1 (as shown below). Hash block 0 is a SHA-256 hash of concatenating hash block 0-0 and hash block 0-1 together. Similarly for hash block 1. Thus, we would have the following output:

- Hash block 0- 8a127ef29e3eb8079aca9aa5fc0649e60edcd0a609dd0285d1f8b7ad9e49c74d

- Hash block 1- 69f1a2768dd44d11700ef08c1e4ece72c3b56382f678e6f20a1fe0f8783b12cf

We continue in a like manner, climbing the Merkle tree until we get to the super hash block, super node or uber block. This is the parent node for the entire tree, and it is nothing more than a SHA-256 hash of its concatenated children. Thus, for our uber block, we would have the following output by concatenating hash blocks 0 and 1 together:

- Uber block- 6b6fb7c2a8b73d24989e0f14ee9cf2706b4f72a24f240f4076f234fa361db084

This uber block is responsible for verifying the integrity of the entire Merkle tree. If a data block changes, all the parent hash blocks should change as well, including the uber block. If at any point, a hash does not match its corresponding children, we have inconsistencies in the tree or data corruption.

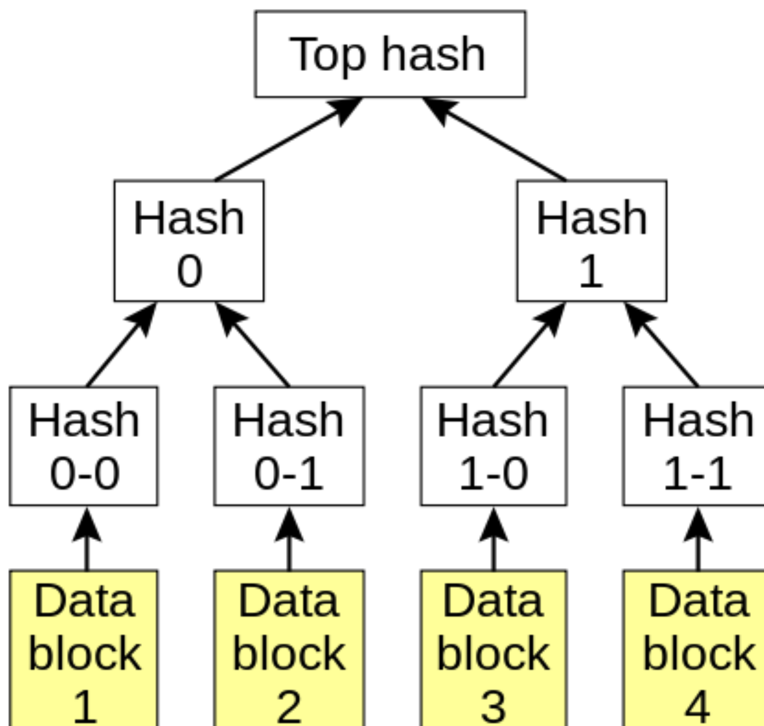


Image showing our binary hash tree example

ZFS uses a Merkle tree to verify the integrity of the entire filesystem and all of the data stored in it. When you scrub your storage pool, ZFS is verifying every SHA-256 hash in the Merkle tree to make sure there is no corrupted data. If there is redundancy in the storage pool, and a corrupted data block is found, ZFS will look elsewhere in the pool for a good data block at the same location by using these hashes. If one is found, it will use that block to fix the corrupted one, then reverify the SHA-256 hash in the Merkle tree.

Copy-on-write (COW)

Copy-on-write (COW) is a data storage technique in which you make a copy of the data block that is going to be modified, rather than modify the data block directly. You then update your pointers to look at the new block location, rather than the old. You also free up the old block, so it can be available to the application. Thus, you don't use any more disk space than if you were to modify the original block. However, you do severely fragment the underlying data. But, the COW model of data storage opens up new features for our data that were previously either impossible or very difficult to implement.

The biggest feature of COW is taking snapshots of your data. Because we've made a copy of the block elsewhere on the filesystem, the old block still remains, even though it's been marked as free by the filesystem. With COW, the filesystem is working its way slowly to the end of the disk. It may take a long time before the old freed up blocks are rewritten to. If a snapshot has been taken, it's treated as a first class filesystem. If a block gets overwritten after it has been snapshotted, it gets copied to the snapshot filesystem. This is possible, because the snapshot is a copy of the hash tree at that exact

moment. As such, snapshots are super cheap, and unless the snapshotted blocks are overwritten, they take up barely any space.

In the following image, when a data block is updated, the hash tree must also be updated. All hashes starting with the child block, and all its parent nodes must be updated with the new hash. The yellow blocks are a copy of the data, elsewhere on the filesystem, and the parent hash nodes are updated to point to the new block locations.

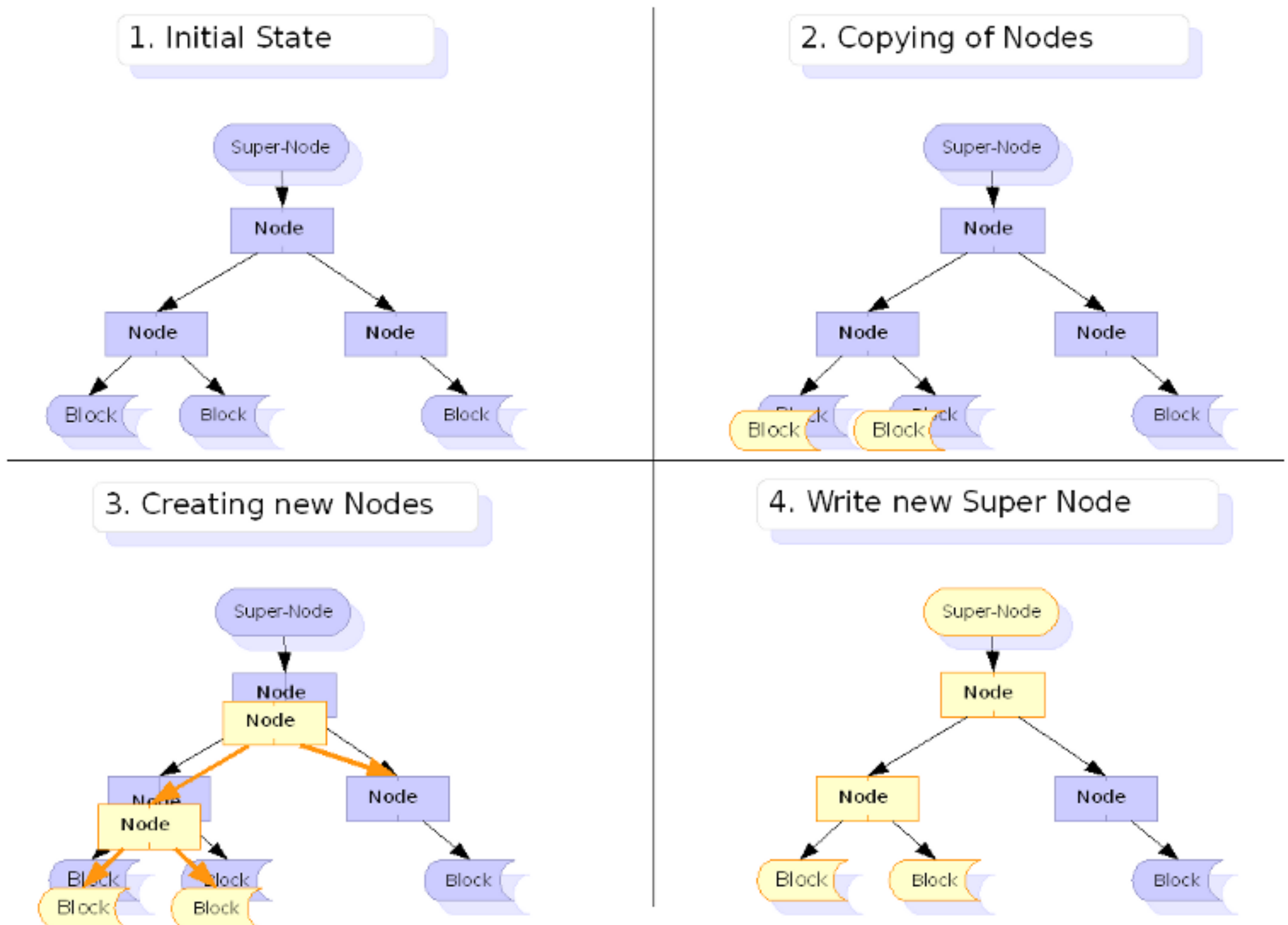


Image showing the COW model when a data block is created, and the hash tree is updated. Courtesy of dest-unreachable.net.

As mentioned, COW heavily fragments the disk. This can have massive performance impacts. So, some work needs to be taken to allocate blocks in advance to minimize fragmentation. There are two basic approaches: using a b-tree to pre-allocate extents, or using a slab approach, marking slabs of disk for the copy. ZFS uses the slab approach, where Btrfs uses the b-tree approach.

Normally, filesystems write data in 4 KB blocks. ZFS writes data in 128 KB blocks. This minimizes the fragmentation by an order of 32. Second, the slab allocater will allocate a slab, then chop up the slab into 128 KB blocks. Thirdly, ZFS delays syncing data to disk every 5 seconds. All data remaining is flushed after 30 seconds. That makes a lot of data flushed to disk at once, in the slab. As a result, this highly increases the probability that similar data will be in the same slab. So, in practice, even though COW is fragmenting the filesystem, there are a few things we can do to greatly minimize that fragmentation.

Not only does ZFS use the COW model, so does Btrfs, NILFS, WAFL and the new Microsoft filesystem ReFS. Many virtualization technologies use COW for storing VM images, such as Qemu. COW filesystems are the future of data storage. We'll discuss snapshots in much more intimate detail at a later post, and how the COW model plays into that.

Posted by Aaron Toponce on Friday, December 14, 2012, at 6:00 am.

Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 4 } Comments

1. Jono Bacon | December 14, 2012 at 9:23 am | Permalink

   Awesome series, Aaron! Thanks for all the work you put into it!

2. Aaron Toponce | December 14, 2012 at 11:56 am | Permalink

   No problem. Glad you're enjoying it. I'm only about 1/2 way through. Should finish some time around the end of the month. 🙂

3. Chris | December 7, 2014 at 2:53 am | Permalink

   Thanks for this great tutorial.

   My question is: When you scrub your storage pool and there is NO redundancy in the storage pool, what happens next?

4. Aaron Toponce | December 9, 2014 at 10:46 am | Permalink

   If you have a block that does not hash to the stored SHA-256 digest, and you do not have redundancy, then you may have a corrupted file, and you will need to restore it from backup.

{ 5 } Trackbacks

1. Links 16/12/2012: Wrapping Up 2012, Many Leftover Links | Techrights | December 16, 2012 at 10:18 am | Permalink

   [...] ZFS Administration, Part IX- Copy-on-write [...]

   [WORDPRESS HASHCASH] The comment's server IP (216.105.40.114) doesn't match the comment's URL host IP (216.105.40.123) and so is spam.

2. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | December 18, 2012 at 12:36 pm | Permalink

   [...] Copy-on-write [...]

3. Aaron Toponce : ZFS Administration, Part XII- Snapshots and Clones | December 19, 2012 at 6:01 am | Permalink

   [...] Copy-on-write [...]

4. Aaron Toponce : ZFS Administration, Part IV- The Adjustable Replacement Cache | January 7, 2013 at 9:26 pm | Permalink

   [...] Copy-on-write [...]

5. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:03 am | Permalink

   [...] Copy-on-write [...]

Aaron Toponce

{ 2012.12.17 }

ZFS Administration, Part X- Creating Filesystems

Table of Contents

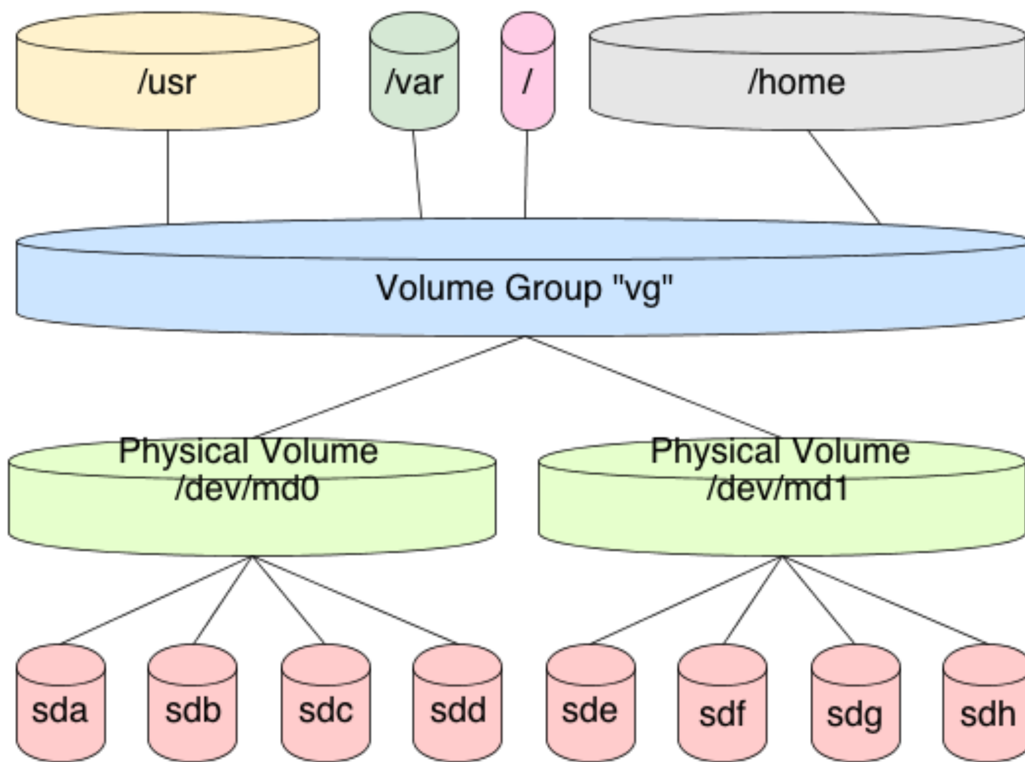
Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

We now begin down the path that is the "bread and butter" of ZFS, known as "ZFS datasets", or filesystems. Previously, up to this point, we've been discussing how to manage our storage pools. But storage pools are not meant to store data directly. Instead, we should create filesystems that share the same storage system. We'll refer to these filesystems from now as datasets.

Background

First, we need to understand how traditional filesystems and volume management work in GNU/Linux before we can get a thorough understanding of ZFS datasets. To treat this fairly, we need to assemble Linux software RAID, LVM, and ext4 or another Linux kernel supported filesystem together.

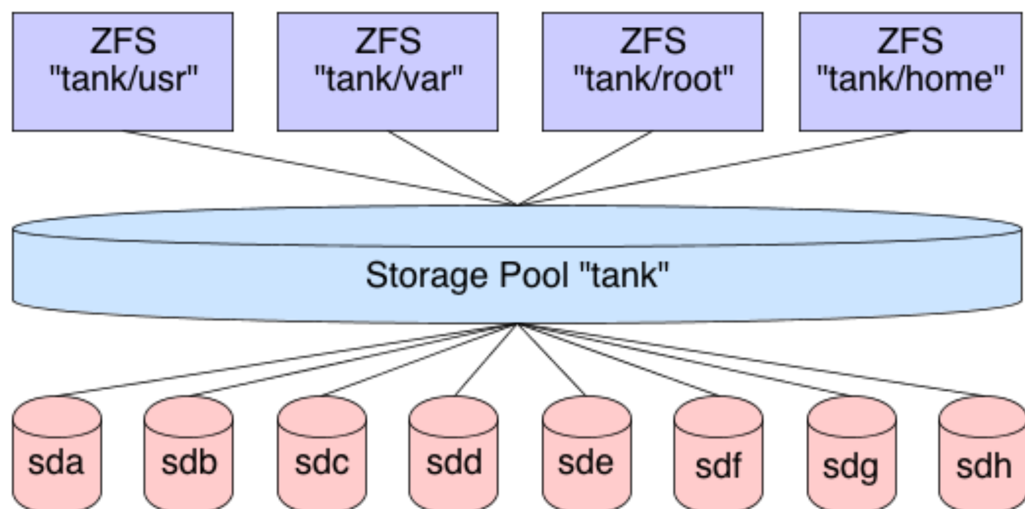
This is done by creating a redundant array of disks, and exporting a block device to represent that array. Then, we format that exported block device using LVM. If we have multiple RAID arrays, we format each of those as well. We then add all these exported block devices to a "volume group" which represents my pooled storage. If I had five exported RAID arrays, of 1 TB each, then I would have 5 TB of pooled storage in this volume group. Now, I need to decide how to divide up the volume, to create logical volumes of a specific size. If this was for an Ubuntu or Debian installation, maybe I would give 100 GB to one logical volume for the root filesystem. That 100 GB is now marked as occupied by the volume group. I then give 500 GB to my home directory, and so forth. Each operation exports a block device, representing my logical volume. It's these block devices that I format with ex4 or a filesystem of my choosing.



Linux RAID, LVM, and filesystem stack. Each filesystem is limited in size.

In this scenario, each logical volume is a fixed size in the volume group. It cannot address the full pool. So, when formatting the logical volume block device, the filesystem is a fixed size. When that device fills, you must resize the logical volume and the filesystem together. This typically requires a myriad of commands, and it's tricky to get just right without losing data.

ZFS handles filesystems a bit differently. First, there is no need to create this stacked approach to storage. We've already covered how to pool the storage, now we well cover how to use it. This is done by creating a dataset in the filesystem. By default, this dataset will have full access to the entire storage pool. If our storage pool is 5 TB in size, as previously mentioned, then our first dataset will have access to all 5 TB in the pool. If I create a second dataset, it too will have full access to all 5 TB in the pool. And so on and so forth.



Each ZFS dataset can use the full underlying storage.

Now, as files are placed in the dataset, the pool marks that storage as unavailable to all datasets. This means that each dataset is aware of what is available in the pool and what is not by all other datasets in the pool. There is no need to create logical volumes of limited size. Each dataset will continue to place files in the pool, until the pool is filled. As the cards fall, they fall. You can, of course, put quotas on datasets, limiting their size, or export ZVOLs, topics we'll cover later.

So, let's create some datasets.

Basic Creation

In these examples, we will assume our ZFS shared storage is named "tank". Further, we will assume that the pool is created with 4 preallocated files of 1 GB in size each, in a RAIDZ-1 array. Let's create some datasets.

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

Notice that the dataset "tank/test" is mounted to "/tank/test" by default, and that it has full access to the entire pool. Also notice that it is occupying only 41.9 KB of the pool. Let's create 4 more datasets, then look at the output:

```
# zfs create tank/test2
# zfs create tank/test3
# zfs create tank/test4
# zfs create tank/test5
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         392K  2.92G  47.9K  /tank
tank/test   41.9K  2.92G  41.9K  /tank/test
tank/test2  41.9K  2.92G  41.9K  /tank/test2
tank/test3  41.9K  2.92G  41.9K  /tank/test3
tank/test4  41.9K  2.92G  41.9K  /tank/test4
tank/test5  41.9K  2.92G  41.9K  /tank/test5
```

Each dataset is automatically mounted to its respective mount point, and each dataset has full unfettered access to the storage pool. Let's fill up some data in one of the datasets, and see how that affects the underlying storage:

```
# cd /tank/test3
# for i in {1..10}; do dd if=/dev/urandom of=file$i.img bs=1024 count=$RANDOM &> /dev/null; done
# zfs list
NAME         USED  AVAIL  REFER  MOUNTPOINT
tank         159M  2.77G  49.4K  /tank
tank/test   41.9K  2.77G  41.9K  /tank/test
tank/test2  41.9K  2.77G  41.9K  /tank/test2
tank/test3   158M  2.77G   158M  /tank/test3
tank/test4  41.9K  2.77G  41.9K  /tank/test4
tank/test5  41.9K  2.77G  41.9K  /tank/test5
```

Notice that in my case, "tank/test3" is occupying 158 MB of disk, so according to the rest of the datasets, there is only 2.77 GB available in the pool, where previously there was 2.92 GB. So as you can see, the big advantage here is that I do not need to worry about preallocated block devices, as I would with LVM. Instead, ZFS manages the entire stack, so it understands how much data has been occupied, and how much is available.

Mounting Datasets

It's important to understand that when creating datasets, you aren't creating exportable block devices by default. This means you don't have something directly to mount. In conclusion, there is nothing to add to your /etc/fstab file for persistence across reboots.

So, if there is nothing to add do the /etc/fstab file, how do the filesystems get mounted? This is done by importing the pool, if necessary, then running the "zfs mount" command. Similarly, we have a "zfs unmount" command to unmount datasets, or we can use the standard "umount" utility:

```
# umount /tank/test5
# mount | grep tank
tank/test on /tank/test type zfs (rw,relatime,xattr)
tank/test2 on /tank/test2 type zfs (rw,relatime,xattr)
tank/test3 on /tank/test3 type zfs (rw,relatime,xattr)
tank/test4 on /tank/test4 type zfs (rw,relatime,xattr)
# zfs mount tank/test5
# mount | grep tank
tank/test on /tank/test type zfs (rw,relatime,xattr)
tank/test2 on /tank/test2 type zfs (rw,relatime,xattr)
tank/test3 on /tank/test3 type zfs (rw,relatime,xattr)
tank/test4 on /tank/test4 type zfs (rw,relatime,xattr)
tank/test5 on /tank/test5 type zfs (rw,relatime,xattr)
```

By default, the mount point for the dataset is "/<pool-name>/<dataset-name>". This can be changed, by changing the dataset property. Just as storage pools have properties that can be tuned, so do datasets. We'll dedicate a full post to dataset properties later. We only need to change the "mountpoint" property, as follows:

```
# zfs set mountpoint=/mnt/test tank/test
# mount | grep tank
tank on /tank type zfs (rw,relatime,xattr)
tank/test2 on /tank/test2 type zfs (rw,relatime,xattr)
tank/test3 on /tank/test3 type zfs (rw,relatime,xattr)
tank/test4 on /tank/test4 type zfs (rw,relatime,xattr)
tank/test5 on /tank/test5 type zfs (rw,relatime,xattr)
tank/test on /mnt/test type zfs (rw,relatime,xattr)
```

Nested Datasets

Datasets don't need to be isolated. You can create nested datasets within each other. This allows you to create namespaces, while tuning a nested directory structure, without affecting the other. For example, maybe you want compression on /var/log, but not on the parent /var. there are other benefits as well, with some caveats that we will look at later.

To create a nested dataset, create it like you would any other, by providing the parent storage pool *and* dataset. In this case we will create a nested log dataset in the test dataset:

```
# zfs create tank/test/log
# zfs list
NAME            USED  AVAIL  REFER  MOUNTPOINT
tank             159M  2.77G  47.9K  /tank
tank/test       85.3K  2.77G  43.4K  /mnt/test
tank/test/log   41.9K  2.77G  41.9K  /mnt/test/log
tank/test2      41.9K  2.77G  41.9K  /tank/test2
tank/test3       158M  2.77G   158M  /tank/test3
tank/test4      41.9K  2.77G  41.9K  /tank/test4
tank/test5      41.9K  2.77G  41.9K  /tank/test5
```

Additional Dataset Administration

Along with creating datasets, when you no longer need them, you can destroy them. This frees up the blocks for use by other datasets, and cannot be reverted without a previous snapshot, which we'll cover later. To destroy a dataset:

```
# zfs destroy tank/test5
# zfs list
NAME            USED  AVAIL  REFER  MOUNTPOINT
tank             159M  2.77G  49.4K  /tank
tank/test       41.9K  2.77G  41.9K  /mnt/test
tank/test/log   41.9K  2.77G  41.9K  /mnt/test/log
```

```
tank/test2     41.9K  2.77G  41.9K  /tank/test2
tank/test3      158M  2.77G   158M  /tank/test3
tank/test4     41.9K  2.77G  41.9K  /tank/test4
```

We can also rename a dataset if needed. This is handy when the purpose of the dataset changes, and you want the name to reflect that purpose. The arguments take a dataset source as the first argument and the new name as the last argument. To rename the tank/test3 dataset to music:

```
# zfs rename tank/test3 tank/music
# zfs list
NAME            USED  AVAIL  REFER  MOUNTPOINT
tank            159M  2.77G  49.4K  /tank
tank/music      158M  2.77G   158M  /tank/music
tank/test      41.9K  2.77G  41.9K  /mnt/test
tank/test/log  41.9K  2.77G  41.9K  /mnt/test/log
tank/test2     41.9K  2.77G  41.9K  /tank/test2
tank/test4     41.9K  2.77G  41.9K  /tank/test4
```

Conclusion

This will get you started with understanding ZFS datasets. There are many more subcommands with the "zfs" command that are available, with a number of different switches. Check the manpage for the full listing. However, even though this isn't a deeply thorough examination of datasets, many more principles and concepts will surface as we work through the series. By the end, you should be familiar enough with datasets that you will be able to manage your entire storage infrastructure with minimal effort.

Posted by Aaron Toponce on Monday, December 17, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 15 } Comments

1. Michael | March 20, 2013 at 12:09 pm | Permalink

   Great articles. I may have found a minor typo you may want to fix (unless I read it wrong)... In the part X post I think "my" should be "by" or "by my"... My brain filled it in as "by" and I almost missed the typo... For reference, the portion I am referring to is "... how do the filesystems get mounted? This is done my importing the pool, ..."

   Thanks again !!!

2. Aaron Toponce | March 20, 2013 at 12:37 pm | Permalink

   Fxed! Thanks!

3. Warren Downs | July 1, 2013 at 4:17 pm | Permalink

   ZFS sharing the pool seems like it could be at least partially emulated by using bind mounts on a traditional Linux system. Great feature but not really unique, in that sense...

4. Ryan | July 25, 2013 at 11:10 pm | Permalink

Each dataset has full access to the pool and takes the space it needs when writing files to the dataset. Does the opposite apply as well? That is, when erasing files in a dataset, is the space returned to the pool?

5. Aaron Toponce | August 7, 2013 at 10:13 am | Permalink

   Correct. Every dataset is available of used blocks. Free blocks are defined as those that are not used. So, when a dataset frees blocks back to the pool, the other datasets are aware of that change.

6. Ryan | September 21, 2013 at 1:20 pm | Permalink

   I continue to be amazed at how well zfs is laid out, but maybe a little shocked at how little protection it has from keeping you from doing something stupid. Prior to committing to zfs, I did a little bit of experimenting.

   I was a little uneasy about how instantaneous a "zpool destroy tank" command is. But looking into it a little more, you can recover with "zpool import -Df". Good. Not sure if it's something to be relied on, but I don't expect to be playing with the zpool command much after the initial setup.

   I created a zpool and a dataset "tank/Backup". The dataset is mounted at /tank/Backup, and everything was great. I copied a file to /tank/Backup, then deleted it. I don't know why, but I went to the /tank directory and did a "rmdir Backup". Backup then disappeared. I would have thought it would complain and prohibit this, but it removed it as would happen with any other directory. I rebooted to see if it zfs would automatically create a mount point for the dataset, but that didn't happen. I then re-created the mount point (mkdir /tank/Backup), or so I thought. When I copy a gigabyte sized file into that directory, "zfs list" still shows the size as 272K. That file is still there, and I can modify or delete it, but it doesn't show up in the dataset size or even in the total for the zpool. When I reboot the system after doing all of this, all of these files are still there. So where are those files? It seems like they're in the zpool, just not officially. This all seems like a thing to avoid, and something to be aware of.

   So, if I could create a mount point by just going to the pool mount point "/tank", what else could I do? I tried copying a file to "/tank" and it was successful. But "/tank" isn't a dataset, so where is the file actually? Again, I don't intend to do these things, but what are the consequences if it's done by accident?

7. Ryan | September 21, 2013 at 3:53 pm | Permalink

   Looking at it again, I discovered that the directory I created and the files I copied over to "/tank" were actually created/copied to the root filesystem. So, the mount point /tank consisted of a mix of the ext4 root filesystem and the zfs datasets.

8. Tim Milstead | March 4, 2014 at 9:16 am | Permalink

   Thanks for all your hard work putting this together. I have found it very useful.

   I think I might have found a small typo. Should the line:

   # umount /tank/test5

   say:

   # umount tank/test5

   ?

9. Roger Qiu | June 27, 2014 at 2:02 am | Permalink

   Is there a way to set ARC sizes for specific datasets?

In some situations, some applications already have their own RAM cache, and so would it be possible to reduce ARC on a particular dataset or folder? That way we don't have double caching of the same data.

10. Ildefonso Camargo | August 12, 2014 at 10:41 am | Permalink

    "When that device fills, you must resize the logical volume and the filesystem together. This typically requires a myriad of commands, and it's tricky to get just right without losing data."

    Sorry, but, I do this all the time, never lost data, and it is not tricky. Furthermore, it only requires two commands:

    lvexpand
    resize2fs

    Now what *is* tricky is to *shrink* the filesystem, and there is real data-loss risk if you don't do it correctly. This is why almost everyone using LVM leaves "spare" disk space on the volume group, to allow for grow.

11. NM | February 4, 2015 at 5:17 pm | Permalink

    "In this scenario, each logical volume is a fixed size in the volume group."

    The LVs aren't even listed in the diagram and they're not fixed, they can be expanded (assuming the underlying filesystem can too). The bitch is shrinking the filesystem.

    An advantage of being restrictive (i.e. using partition-like things) is that if your webserver gets DDoSed or something only your /var/log/web gets maxed out, not your whole system.

    Not having a limit with everyone pissing in the pool seems easy but, as you mention, causes fragmentation.

    I assume you cover these caveats in the next chapters, especially with quotas and ZVOLs, but i haven't gotten there yet. 🙂

    Don't get me wrong, i think ZFS is great. Keep it up.

12. Aaron Toponce | February 17, 2015 at 1:40 pm | Permalink

    > "In this scenario, each logical volume is a fixed size in the volume group."
    >
    > The LVs aren't even listed in the diagram and they're not fixed, they can be expanded (assuming the underlying filesystem can too). The bitch is shrinking the filesystem.

    The LVs are listed in the digram (/usr, /var, /, /home). When referring to the logical volumes, I am referring to LVM2, not ZFS. Logical volumes in LVM are indeed fixed in size. If you have an LVM volume group of 2 TB, and you create a logical volume of 1 TB in size, then there is only 1 TB of volume group space left to create additional LVM logical volumes.

    > An advantage of being restrictive (i.e. using partition-like things) is that if your webserver gets DDoSed or something only your /var/log/web gets maxed out, not your whole system. Not having a limit with everyone pissing in the pool seems easy but, as you mention, causes fragmentation.

    That is true. ZFS has dataset quotas that work well here, or ZVOLs of fixed size, if you would prefer to guard against that in this situation.

    > I assume you cover these caveats in the next chapters, especially with quotas and ZVOLs, but i haven't gotten there yet. 🙂 Don't get me wrong, i think ZFS is great. Keep it up.

No problem. I'll answer your questions as I come across them.

13. Eric Pedersen | March 26, 2015 at 11:59 am | Permalink

    This is a fantastic source of information that I should have read before doing something dumb!

    I created a pool called /data and a dataset called /data in the same location. ie: 'zpool list' and 'zfs list' come back with 'data'. (That's me - worst practices...)

    Now I've got a whack of backups sitting in /data. Is it to possible to a) create a new dataset called data/backup, b) move the stuff in data to data/backup, and c) do a 'zfs destroy data' without losing all my backups? Or should I even bother at this point?

14. Eric Pedersen | March 26, 2015 at 2:20 pm | Permalink

    I think you can safely ignore my question, because I think I've figured it out.

15. AS | July 1, 2017 at 8:13 am | Permalink

    Hello, I'd like to use ZFS for my desktop / workstation box.

    I is there a way to tell ZFS where to place a dataset on the disks in a pool? AFAIK reading and writing is faster on the outer parts than on inner parts of a disk.

    There is a tool that arranges files in a sequential order so that data is accessed "in a row". That speeds up the boot process or other scenarios like starting X loading the binaries, fonts, desktop theme, icons….
    There is a tool that does this but it only works for EXT* file systems. It's named e4rat
    http://e4rat.sourceforge.net/ . Years ago I used a tool on windows which did both.

    I am almost sure that both is possible with ZFS. Am I wrong?
    Is there anything speaking against positioning a DS on the disks and making ZFS store data in a sequential order?

{ 9 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part XI- Compression and Deduplication | December 18, 2012 at 6:01 am | Permalink

    [...] Creating Filesystems [...]

2. Aaron Toponce : ZFS Administration, Part IX- Copy-on-write | December 18, 2012 at 12:35 pm | Permalink

    [...] Creating Filesystems [...]

3. Aaron Toponce : ZFS Administration, Part XIII- Sending and Receiving Filesystems | December 20, 2012 at 6:01 am | Permalink

    [...] Creating Filesystems [...]

4. Aaron Toponce : ZFS Administration, Part XIV- ZVOLS | December 21, 2012 at 6:01 am | Permalink

    [...] Creating Filesystems [...]

5. Aaron Toponce : ZFS Administration, Part XV- iSCSI, NFS and Samba | December 31, 2012 at 6:01 am | Permalink

    [...] Creating Filesystems [...]

6. Aaron Toponce : ZFS Administration, Part XVI- Getting and Setting Properties | January 2, 2013 at 6:01 am | Permalink

   [...] Creating Filesystems [...]

7. Aaron Toponce : ZFS Administration, Part XVII- Best Practices and Caveats | January 3, 2013 at 6:02 am | Permalink

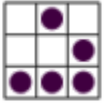
   [...] Creating Filesystems [...]

8. Aaron Toponce : ZFS Administration, Part IV- The Adjustable Replacement Cache | April 19, 2013 at 4:56 am | Permalink

   [...] Creating Filesystems [...]

9. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:09 am | Permalink

   [...] Creating Filesystems [...]



[Aaron Toponce]

{ 2012.12.18 }

ZFS Administration, Part XI- Compression and Deduplication

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

Compression

Compression is transparent with ZFS if you enable it. This means that every file you store in your pool can be compressed. From your point of view as an application, the file does not appear to be compressed, but appears to be stored uncompressed. In other words, if you run the "file" command on your plain text configuration file, it will report it as such. Instead, underneath the file layer, ZFS is compressing and decompressing the data on disk on the fly. And because compression is so cheap on the CPU, and exceptionally fast with some algorithms, it should not be noticeable.

Compression is enabled and disabled per dataset. Further, the supported compression algorithms are LZJB, LZ4, ZLE, and Gzip. With Gzip, the standards levels of 1 through 9 are supported, where 1 is as fast as possible, with the least compression, and 9 is as compressed as possible, taking as much time as necessary. The default is 6, as is standard in GNU/Linux and other Unix operating systems. LZJB, on the other hand, was invented by Jeff Bonwick, who is also the author of ZFS. LZJB was designed to be fast with tight compression ratios, which is standard with most Lempel-Ziv algorithms. LZJB is the default. ZLE is a speed demon, with very light compression ratios. LZJB seems to provide the best all around results it terms of performance and compression.

UPDATE: Since the writing of this post, LZ4 has been introduced to ZFS on Linux, and is now the preferred way to do compression with ZFS. Not only is it fast, but it also offers tighter compression ratios than LZJB- on average about 0.23%

Obviously, compression can vary on the disk space saved. If the dataset is storing mostly uncompressed data, such as plain text log files, or configuration files, the compression ratios can be massive. If the dataset is storing mostly compressed images and video, then you won't see much if anything in the way of disk savings. With that

said, compression is disabled by default, and enabling LZJB or LZ4 doesn't seem to yield any performance impact. So even if you're storing largely compressed data, for the data files that are not compressed, you can get those compression savings, without impacting the performance of the storage server. So, IMO, I would recommend enabling compression for all of your datasets.

WARNING: Enabling compression on a dataset is not retroactive! It will only apply to newly committed or modified data. Any previous data in the dataset will remain uncompressed. So, if you want to use compression, you should enable it before you begin committing data.

To enable compression on a dataset, we just need to modify the "compression" property. The valid values for that property are: "on", "off", "lzjb", "lz4", "gzip", "gzip[1-9]", and "zle".

```
# zfs create tank/log
# zfs set compression=lz4 tank/log
```

Now that we've enabled compression on this dataset, let's copy over some uncompressed data, and see what sort of savings we would see. A great source of uncompressed data would be the /etc/ and /var/log/ directories. Let's create a tarball of these directories, see it's raw size and see what sort of space savings we achieved:

```
# tar -cf /tank/test/text.tar /var/log/ /etc/
# ls -lh /tank/test/text.tar
-rw-rw-r-- 1 root root 24M Dec 17 21:24 /tank/test/text.tar
# zfs list tank/test
NAME        USED  AVAIL  REFER  MOUNTPOINT
tank/test  11.1M  2.91G  11.1M  /tank/test
# zfs get compressratio tank/test
NAME       PROPERTY       VALUE  SOURCE
tank/test  compressratio  2.14x  -
```

So, in my case, I created a 24 MB uncompressed tarball. After copying it to the dataset that had compression enabled, it only occupied 11.1 MB. This is less than half the size (text compresses very well)! We can read the "compressratio" property on the dataset to see what sort of space savings we are achieving. In my case, the output is telling me that the compressed data would occupy 2.14 times the amount of disk space, if uncompressed. Very nice.

Deduplication

We have another way to save disk in conjunction with compression, and that is deduplication. Now, there are three main types of deduplication: file, block, and byte. File deduplication is the most performant and least costly on system resources. Each file is hashed with a cryptographic hashing algorithm, such as SHA-256. If the hash matches for multiple files, rather than storing the new file on disk, we reference the original file in the metadata. This can have significant savings, but has a serious drawback. If a single byte changes in the file, the hashes will no longer match. This means we can no longer reference the whole file in the filesystem metadata. As such, we must make a copy of all the blocks to disk. For large files, this has massive performance impacts.

On the extreme other side of the spectrum, we have byte deduplication. This deduplication method is the most expensive, because you must keep "anchor points" to determine where regions of deduplicated and unique bytes start and end. After all, bytes are bytes, and without knowing which files need them, it's nothing more than a sea of data. This sort of deduplication works well for storage where a file may be stored multiple times, even if it's not aligned under the same blocks, such as mail attachments.

In the middle, we have block deduplication. ZFS uses block deduplication only. Block deduplication shares all the same blocks in a file, minus the blocks that are different. This allows us to store only the unique blocks on disk, and reference the shared blocks in RAM. It's more efficient than byte deduplication, and more flexible than file deduplication. However, it has a drawback- it requires a great deal of memory to keep track of which blocks

are shared, and which are not. However, because filesystems read and write data in block segments, it makes the most sense to use block deduplication for a modern filesystem.

The shared blocks are stored in what's called a "deduplication table". The more duplicated blocks on the filesystem, the larger this table will grow. Every time data is written or read, the deduplication table is referenced. This means you want to keep the ENTIRE deduplication table in fast RAM. If you do not have enough RAM, then the table will spill over to disk. This can have massive performance impacts on your storage, both for reading and writing data.

The Cost of Deduplication

So the question remains: how much RAM do you need to store your deduplication table? There isn't an easy answer to this question, but we can get a good general idea on how to approach the problem. First, is to look at the number of blocks in your storage pool. You can see this information as follows (be patient- it may take a while to scan all the blocks in your filesystem before it gives the report):

```
# zdb -b rpool

Traversing all blocks to verify nothing leaked ...

        No leaks (block sum matches space maps exactly)

        bp count:           288674
        bp logical:    34801465856      avg: 120556
        bp physical:   30886096384      avg: 106992     compression:   1.13
        bp allocated:  31092428800      avg: 107707     compression:   1.12
        bp deduped:              0    ref>1:      0   deduplication:   1.00
        SPA allocated: 31092244480     used: 13.53%
```

In this case, there are 288674 used blocks in the storage pool "rpool" (look at "bp count"). It requires about 320 bytes of RAM for each deduplicated block in the pool. So, for 288674 blocks multiplied by 320 bytes per block gives us about 92 MB. The filesystem is about 200 GB in size, so we can assume that the deduplication could only grow to about 670 MB seeing as though it is only 13.53% filled. That's 3.35 MB of deduplicated data for every 1 GB of filesystem, or 3.35 GB of RAM per 1 TB of disk.

If you are planning your storage in advance, and want to know the size before committing data, then you need to figure out what your average block size would be.
In this case, you need to be intimately familiar with the data. ZFS reads and writes data in 128 KB blocks. However, if you're storing a great deal of configuration files, home directories, etc., then your files will be smaller than 128 KB. Let us assume, for this example, that the average block size would be 100 KB, as in our example above. If my total storage was 1 TB in size, then 1 TB divided by 100 KB per block is about 10737418 blocks. Multiplied by 320 bytes per block, leaves us with 3.2 GB of RAM, which is close to the previous number we got.

A good rule of thumb, would be to plan 5 GB of RAM for every 1 TB of disk. This can get very expensive quickly. A 12 TB pool, small in many enterprises, would require 60 GB RAM to make sure your dedupe table is stored, and quickly accessible. Remember, once it spills to disk, it causes severe performance impacts.

Total Deduplication Ram Cost

ZFS stores more than just the deduplication table in RAM. It also stores the ARC as well as other ZFS metadata. And, guess what? **The deduplication table is capped at 25% the size of the ARC**. This means, you don't need 60 GB of RAM for a 12 TB storage array. You need 240 GB of RAM to ensure that your deduplication table fits. In other words, if you plan on doing deduplication, make sure you quadruple your RAM footprint, or you'll be hurting.

Deduplication in the L2ARC

The deduplication table however can spill over to the L2ARC, rather than to slow platter disk. If your L2ARC consists of fast SSDs or RAM drives, then pulling up the deduplication table on every read and write won't impact performance quite as bad as if it spilled over to platter disk. Still, it will have an impact, however, as SSDs don't have the latency speeds that system RAM does. So for storage servers where performance is not critical, such as nightly or weekly backup servers, the deduplication table on the L2ARC can be perfectly acceptable

Enabling Deduplication

To enable deduplication for a dataset, you change the "dedup" property. However, realize that even though the "dedup" property is enabled on a dataset, it deduplicates against ALL data in the entire storage pool. Only data committed to that dataset will be checked for duplicate blocks. As with compression, deduplication is not retroactive on previously committed data. It is only applied to newly committed or modified data. Further, deduplicated data is not flushed to disk as an atomic transaction. Instead, the blocks are written to disk serially, one block at a time. Thus, this does open you up for corruption in the event of a power failure before the blocks have been written.

Let's enable deduplication on our "tank/test" dataset, then copy over the same tarball, but this time, giving it a different name in the storage, and see how that affects our deduplication ratios. Notice that the deduplication ratio is found from the pool using the "zpool" command, and not the "zfs" command. First, we need to enable deduplication on the dataset:

```
# zfs set dedup=on tank/test
# cp /tank/test/text.tar{,.2}
# tar -cf /tank/test/boot.tar /boot
# zfs get compressratio tank/test
NAME       PROPERTY       VALUE  SOURCE
tank/test  compressratio  1.74x  -
# zpool get dedupratio tank
NAME  PROPERTY    VALUE  SOURCE
tank  dedupratio  1.42x  -
# ls -lh /tank/test
total 38M
-rw-rw-r-- 1 root root 18M Dec 17 22:31 boot.tar
-rw-rw-r-- 1 root root 24M Dec 17 22:27 text.tar
-rw-rw-r-- 1 root root 24M Dec 17 22:29 text.tar.2
# zfs list tank/test
NAME        USED  AVAIL  REFER  MOUNTPOINT
tank/test  37.1M  2.90G  37.1M  /tank/test
```

In this case, the data is being compressed first, then deduplicated. The raw data would normally occupy about 66 MB of disk, however it's only occupying 37 MB, due to compression and deduplication. Significant savings.

Conclusion and Recommendation

Compression and deduplication can provide massive storage benefits, no doubt. For live running production data, compression offers great storage savings with negligible performance impacts. For mixed data, it's been common for me to see 1.15x savings. For the cost, it's well worth it. However, for deduplication, I have found it's not worth the trouble, unless performance is not of a concern at all. The weight it puts on RAM and the L2ARC is immense. When it spills to slower platter, you can kiss performance goodbye. And for mixed data, I rarely see it go north of 1.10x savings, which isn't worth it IMO. The risk of data corruption with deduplication is also not worth it, IMO. So, as a recommendation, I would encourage you to enable compression on all your datasets by default, and not worry about deduplication unless you know you have the RAM to accommodate the table. If you

can afford that purchase, then the space savings can be pretty significant, which is something ext4, XFS and other filesystems just can't achieve.

Posted by Aaron Toponce on Tuesday, December 18, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 10 } Comments

1. Michael | March 20, 2013 at 1:07 pm | Permalink

   Aaron, Thanks for the series. I am also using zfsonlinux but am using rhel6 (actually oel6) instead of debian.

   I was at first getting nothing on the following command and thought maybe it was because of a difference in the distributions or something:

   "zdb -b rpool"

   I then thought maybe I should replace rpool with my pools name (tank) and that worked... Just a thought but in your earlier posts I thought you were using tank and other sites use tank so maybe it would be better for consistency's sake to change the example to:

   "zdb -b tank"

   It may not be necessary but that would have made it more clear to me and maybe will help someone else...

   Thanks again. Great writing & analysis !

2. Dzezik | September 12, 2013 at 2:06 pm | Permalink

   but if my files on filesystem are in megabytes (i.e MP3, JPG...) then I can increase block size from 128kb to 1mb and then I need about 8 time less memory for the same deduplication. 640MB for 1TB, for about 12TB storage is then less than 8GB.
   other documents on separate dataset without dedup but with compression. compression is useless for jpg mp3, flac, avi, mpg, mkv. 8GB is not a problem, You can build 32GB RAM machine on any CPU/motherboard.
   I decided to use single xeon lga 2011 and get 128GB to build some VM machines with databases on the same hardware, ZFS will be storage appliance for databases.

3. Magni | September 27, 2013 at 6:24 pm | Permalink

   And how you can set blocksize to 1MB? As far as I understand ZFS for Linux you can only set size in range 512B - 128KB and not more?
   Do I miss something here (I'm pretty new to ZFS so it's possible 🙂

4. ianjo | September 29, 2013 at 11:40 am | Permalink

Just two small notes on compression:
- the option for gzip is gzip-[1-9] (for example 'gzip-9')
- there is a new compression algorithm in most open-source zfs implementations, 'lz4', that is intended as a faster and smarter replacement for lzjb in exactly your use case of "a little compression by default never hurt anyone"

5. grin | December 16, 2013 at 2:04 pm | Permalink

   Here's a simple test about compression since I was curious:

   Original file is a pretty verbose log form an ip phone:
   -rw-r----- 1 root adm 551M Dec 16 22:01 /var/log/gxp2000.log

   The filesystem:
   NAME USED AVAIL REFER MOUNTPOINT
   tank/watch/gzip 55.5M 16.8G 55.5M /tank/pleigh/gzip
   tank/watch/gzip-9 51.2M 16.8G 51.2M /tank/pleigh/gzip-9
   tank/watch/lz4 76.1M 16.8G 76.1M /tank/pleigh/lz4
   tank/watch/lzjb 134M 16.8G 134M /tank/pleigh/lzjb
   tank/watch/on 134M 16.8G 134M /tank/pleigh/on
   tank/watch/zle 576M 16.8G 576M /tank/pleigh/zle

   And compressing with gzip -9 the file:
   -rw-r----- 1 root root 28M Dec 16 21:56 zle/gxp2000.log.gz

   Funny thing is about 'zfs list' is that after compressing the file in the zle dir (which compressed none) the USED stayed at 576M while 'du' sees through the veil:
   # du -h zle/
   28M zle/

6. James Mills | December 10, 2014 at 5:06 am | Permalink

   I get the following error with zdb:

   # zdb -b tank
   zdb: can't open 'tank': No such file or directory

7. Cliff Lawton | March 2, 2015 at 12:22 am | Permalink

   Hi, I am confused how enabling dedupe on a zfs dataset effects the zpool. Are you able to explain more clearly the difference of these two sentences? Thanks
   "realize that even though the "dedup" property is enabled on a dataset, it deduplicates against ALL data in the entire storage pool. Only data committed to that dataset will be checked for duplicate blocks."

8. Aaron Toponce | March 3, 2015 at 12:07 pm | Permalink

   Data in the dataset is deduplicated. The data is matched against all the data in the pool, which includes data outside of that dataset. However, data in other datasets is not deduplicated. In other words, deduplication is handled per dataset, but the data that it's being deduplicated against can be any block on the pool, in or out of the dataset itself.

9. James Lin | February 17, 2016 at 2:34 pm | Permalink

   In your example of compression, you turned on compression for tank/log:

```
# zfs create tank/log
# zfs set compression=lz4 tank/log
```

and later you use another destination path as compression example /tank/test:

```
# tar -cf /tank/test/text.tar /var/log/ /etc/
# ls -lh /tank/test/text.tar
-rw-rw-r-- 1 root root 24M Dec 17 21:24 /tank/test/text.tar
# zfs list tank/test
NAME USED AVAIL REFER MOUNTPOINT
tank/test 11.1M 2.91G 11.1M /tank/test
# zfs get compressratio tank/test
NAME PROPERTY VALUE SOURCE
tank/test compressratio 2.14x -
```

Why is that the case? Is it typo?

10. Richard Geary | September 10, 2016 at 8:06 am | Permalink

    > "In this case, the data is being compressed first, then deduplicated. The raw data would normally occupy about 66 MB of disk, however it's only occupying 37 MB, due to compression and deduplication."

    You're only displaying the compressed size, not the dedup + compressed size. Compression ratio is 1.74, so 37Mb = 66Mb / 1.74.
    Since dedup is applied to the pool, I presume you can't see the on-disk size unless you look at zpool status (otherwise you'd get duplicate counting)

{ 12 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | December 18, 2012 at 12:35 pm | Permalink

   [...] Compression and Deduplication [...]

2. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | December 18, 2012 at 12:35 pm | Permalink

   [...] Compression and Deduplication [...]

3. Aaron Toponce: ZFS Administration, Part XI- Compression and Deduplication - Bartle Doo | December 18, 2012 at 4:40 pm | Permalink

   [...] Copy-on-write Creating Filesystems Compression and Deduplication [...]

4. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | December 20, 2012 at 8:07 am | Permalink

   [...] Compression and Deduplication [...]

5. Aaron Toponce: ZFS Administration, Part XIII- Sending and Receiving Filesystems - Bartle Doo | December 20, 2012 at 2:32 pm | Permalink

   [...] Creating Filesystems Compression and Deduplication Snapshots and Clones Sending and Receiving [...]

6. Aaron Toponce: ZFS Administration, Part XIV- ZVOLS - Bartle Doo | December 21, 2012 at 11:02 am | Permalink

   [...] Creating Filesystems Compression and Deduplication Snapshots and Clones Sending and Receiving Filesystems ZVOLs What is a [...]

7. Aaron Toponce : ZFS Administration, Part VI- Scrub and Resilver | January 7, 2013 at 9:25 pm | Permalink

   [...] Compression and Deduplication [...]

8. Aaron Toponce : ZFS Administration, Part III- The ZFS Intent Log | January 7, 2013 at 9:26 pm | Permalink

   [...] Compression and Deduplication [...]

9. Aaron Toponce : ZFS Administration, Part I- VDEVs | January 7, 2013 at 9:27 pm | Permalink

   [...] Compression and Deduplication [...]

10. Aaron Toponce : Install ZFS on Debian GNU/Linux | April 19, 2013 at 4:55 am | Permalink

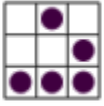
    [...] Compression and Deduplication [...]

11. Aaron Toponce : ZFS Administration, Part XIV- ZVOLS | April 19, 2013 at 4:59 am | Permalink

    [...] Compression and Deduplication [...]

12. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | July 8, 2013 at 10:08 pm | Permalink

    […] Compression and Deduplication […]



[Aaron Toponce](#)

{ 2012.12.19 }

ZFS Administration, Part XII- Snapshots and Clones

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

Snapshots with ZFS are similar to snapshots with Linux LVM. A snapshot is a first class read-only filesystem. It is a mirrored copy of the state of the filesystem at the time you took the snapshot. Think of it like a digital photograph of the outside world. Even though the world is changing, you have an image of what the world was like at the exact moment you took that photograph. Snapshots behave in a similar manner, except when data changes that was part of the dataset, you keep the original copy in the snapshot itself. This way, you can maintain persistence of that filesystem.

You can keep up to 2^64 snapshots in your pool, ZFS snapshots are persistent across reboots, and they don't require any additional backing store; they use the same storage pool as the rest of your data. If you remember our post about the nature of copy-on-write filesystems, you will remember our discussion about Merkle trees. A ZFS snapshot is a copy of the Merkle tree in that state, except we make sure that the snapshot of that Merkle tree is never modified.

Creating snapshots is near instantaneous, and they are cheap. However, once the data begins to change, the snapshot will begin storing data. If you have multiple snapshots, then multiple deltas will be tracked across all the snapshots. However, depending on your needs, snapshots can still be exceptionally cheap.

Creating Snapshots

You can create two types of snapshots: pool snapshots and dataset snapshots. Which type of snapshot you want to take is up to you. You must give the snapshot a name, however. The syntax for the snapshot name is:

- *pool/dataset@snapshot-name*
- *pool@snapshot-name*

To create a snapshot, we use the "zfs snapshot" command. For example, to take a snapshot of the "tank/test" dataset, we would issue:

```
# zfs snapshot tank/test@tuesday
```

Even though a snapshot is a first class filesystem, it does not contain modifiable properties like standard ZFS datasets or pools. In fact, everything about a snapshot is read-only. For example, if you wished to enable compression on a snapshot, here is what would happen:

```
# zfs set compression=lzjb tank/test@friday
cannot set property for 'tank/test@friday': this property can not be modified for snapshots
```

Listing Snapshots

Snapshots can be displayed two ways: by accessing a hidden ".zfs" directory in the root of the dataset, or by using the "zfs list" command. First, let's discuss the hidden directory. Check out this madness:

```
# ls -a /tank/test
./  ../  boot.tar  text.tar  text.tar.2
# cd /tank/test/.zfs/
# ls -a
./  ../  shares/  snapshot/
```

Even though the ".zfs" directory was not visible, even with "ls -a", we could still change directory to it. If you wish to have the ".zfs" directory visible, you can change the "snapdir" property on the dataset. The valid values are "hidden" and "visible". By default, it's hidden. Let's change it:

```
# zfs set snapdir=visible tank/test
# ls -a /tank/test
./  ../  boot.tar  text.tar  text.tar.2  .zfs/
```

The other way to display snapshots is by using the "zfs list" command, and passing the "-t snapshot" argument, as follows:

```
# zfs list -t snapshot
NAME                              USED  AVAIL  REFER  MOUNTPOINT
pool/cache@2012:12:18:51:2:19:00     0      -   525M  -
pool/cache@2012:12:18:51:2:19:15     0      -   525M  -
pool/home@2012:12:18:51:2:19:00  18.8M      -  28.6G  -
pool/home@2012:12:18:51:2:19:15  18.3M      -  28.6G  -
pool/log@2012:12:18:51:2:19:00    184K      -  10.4M  -
pool/log@2012:12:18:51:2:19:15    184K      -  10.4M  -
pool/swap@2012:12:18:51:2:19:00      0      -    76K  -
pool/swap@2012:12:18:51:2:19:15      0      -    76K  -
pool/vmsa@2012:12:18:51:2:19:00      0      -  1.12M  -
pool/vmsa@2012:12:18:51:2:19:15      0      -  1.12M  -
pool/vmsb@2012:12:18:51:2:19:00      0      -  1.31M  -
pool/vmsb@2012:12:18:51:2:19:15      0      -  1.31M  -
tank@2012:12:18:51:2:19:00           0      -  43.4K  -
tank@2012:12:18:51:2:19:15           0      -  43.4K  -
tank/test@2012:12:18:51:2:19:00      0      -  37.1M  -
tank/test@2012:12:18:51:2:19:15      0      -  37.1M  -
```

Notice that by default, it will show all snapshots for all pools.

If you want to be more specific with the output, you can see all snapshots of a given parent, whether it be a dataset, or a storage pool. You only need to pass the "-r" switch for recursion, then provide the parent. In this case, I'll see only the snapshots of the storage pool "tank", and ignore those in "pool":

```
# zfs list -r -t snapshot tank
NAME                              USED  AVAIL  REFER  MOUNTPOINT
tank@2012:12:18:51:2:19:00           0      -  43.4K  -
tank@2012:12:18:51:2:19:15           0      -  43.4K  -
tank/test@2012:12:18:51:2:19:00      0      -  37.1M  -
tank/test@2012:12:18:51:2:19:15      0      -  37.1M  -
```

Destroying Snapshots

Just as you would destroy a storage pool, or a ZFS dataset, you use a similar method for destroying snapshots. To destroy a snapshot, use the "zfs destroy" command, and supply the snapshot as an argument that you want to destroy:

```
# zfs destroy tank/test@2012:12:18:51:2:19:15
```

An important thing to know, is if a snapshot exists, it's considered a child filesystem to the dataset. As such, you cannot remove a dataset until all snapshots, and nested datasets have been destroyed.

```
# zfs destroy tank/test
cannot destroy 'tank/test': filesystem has children
use '-r' to destroy the following datasets:
tank/test@2012:12:18:51:2:19:15
tank/test@2012:12:18:51:2:19:00
```

Destroying snapshots can free up additional space that other snapshots may be holding onto, because they are unique to those snapshots.

Renaming Snapshots

You can rename snapshots, however, they must be renamed in the storage pool and ZFS dataset from which they were created. Other than that, renaming snapshots is pretty straight forward:

```
# zfs rename tank/test@2012:12:18:51:2:19:15 tank/test@tuesday-19:15
```

Rolling Back to a Snapshot

A discussion about snapshots would not be complete without a discussion about rolling back your filesystem to a previous snapshot.

Rolling back to a previous snapshot will discard any data changes between that snapshot and the current time. Further, by default, you can only rollback to the most recent snapshot. In order to rollback to an earlier snapshot, you must destroy all snapshots between the current time and that snapshot you wish to rollback to. If that's not enough, the filesystem must be unmounted before the rollback can begin. This means downtime.

To rollback the "tank/test" dataset to the "tuesday" snapshot, we would issue:

```
# zfs rollback tank/test@tuesday
cannot rollback to 'tank/test@tuesday': more recent snapshots exist
use '-r' to force deletion of the following snapshots:
tank/test@wednesday
tank/test@thursday
```

As expected, we must remove the "@wednesday" and "@thursday" snapshots before we can rollback to the "@tuesday" snapshot.

ZFS Clones

A ZFS clone is a writeable filesystem that was "upgraded" from a snapshot. Clones can only be created from snapshots, and a dependency on the snapshot will remain as long as the clone exists. This means that you cannot destroy a snapshot, if you cloned it. The clone relies on the data that the snapshot gives it, to exist. You must destroy the clone before you can destroy the snapshot.

Creating clones is nearly instantaneous, just like snapshots, and initially does not take up any additional space. Instead, it occupies all the initial space of the snapshot. As data is modified in the clone, it begins to take up space separate from the snapshot.

Creating ZFS Clones

Creating a clone is done with the "zfs clone" command, the snapshot to clone, and the name of the new filesystem. The clone does not need to reside in the same dataset as the clone, but it does need to reside in the same storage pool. For example, if I wanted to clone the "tank/test@tuesday" snapshot, and give it the name of "tank/tuesday", I would run the following command:

```
# zfs clone tank/test@tuesday tank/tuesday
# dd if=/dev/zero of=/tank/tuesday/random.img bs=1M count=100
# zfs list -r tank
NAME            USED  AVAIL  REFER  MOUNTPOINT
tank            161M  2.78G  44.9K  /tank
tank/test      37.1M  2.78G  37.1M  /tank/test
tank/tuesday    124M  2.78G   161M  /tank/tuesday
```

Destroying Clones

As with destroying datasets or snapshots, we use the "zfs destroy" command. Again, you cannot destroy a snapshot until you destroy the clones. So, if we wanted to destroy the "tank/tuesday" clone:

```
# zfs destroy tank/tuesday
```

Just like you would with any other ZFS dataset.

Some Final Thoughts

Because keeping snapshots is very cheap, it's recommended to snapshot your datasets frequently. Sun Microsystems provided a Time Slider that was part of the GNOME Nautilus file manager. Time Slider keeps snapshots in the following manner:

- frequent- snapshots every 15 mins, keeping 4 snapshots
- hourly- snapshots every hour, keeping 24 snapshots
- daily- snapshots every day, keeping 31 snapshots
- weekly- snapshots every week, keeping 7 snapshots
- monthly- snapshots every month, keeping 12 snapshots

Unfortunately, Time Slider is not part of the standard GNOME desktop, so it's not available for GNU/Linux. However, the ZFS on Linux developers have created a "zfs-auto-snapshot" package that you can install from the project's PPA if running Ubuntu. If running another GNU/Linux operating system, you could easily write a Bash or Python script that mimics that functionality, and place it on your root's crontab.

Because both snapshots and clones are cheap, it's recommended that you take advantage of them. Clones can be useful to test deploying virtual machines, or development environments that are cloned from production environments. When finished, they can easily be destroyed, without affecting the parent dataset from which the snapshot was created.

Posted by Aaron Toponce on Wednesday, December 19, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 12 } Comments

1. Ahmed Kamal | July 22, 2013 at 2:51 pm | Permalink

   Worth noting that if you clone a snapshot, that snapshot cannot be deleted afterwards. If you really want to delete it, you can "zfs promote" the clone to a full file-system, and reverse the parent-child relationship, enabling you to delete the origin snapshot

2. Chris | December 8, 2013 at 9:48 am | Permalink

   one question, just to clarify: assume I create a snapshot, clone it and then write files to the clone. Will these files be written to the snapshot and therefore change the snapshot or will these files be stored at another position. Basically, the question is if cloning a snapshot makes the snapshot writable or if cloning makes a writable snapshot of the snapshot. (in the later the original snapshot should be restored after deleting the clone).

3. Truxton Fulton | January 17, 2014 at 12:11 am | Permalink

   Thank you Aaron, for these ZFS articles. I have started using ZFS for linux in the last few weeks, and your articles are the most succinct source of information that searching could find. I'm still experimenting with my 12 TB raidz3 pool, but after learning not to use the "sda", "sdb" disk identifiers, I'm rapidly getting comfortable with the idea of trusting my new filesystem. I appreciate you sharing your knowledge.

4. Steve Yuroff | January 20, 2014 at 12:04 pm | Permalink

   A small correction: you say "However, once the data begins to change, the snapshot will begin storing data."
   No, the production filesystem starts storing data. I know you know this, but as this series is a great benefit to those wrapping their heads around ZFS concepts, this is a point worth clarifying.

   I think this section would also benefit from documenting how the .zfs directory can be used to navigate to a file or folder and duplicating that content vs rolling back the entire filesystem to the snapshot state. I experience needing a file or directory restore from previous version much more often than an entire filesystem.

5. RvdK | May 13, 2014 at 8:02 am | Permalink

   Wow...this is easily the best explanation of snapshots/clones I've found yet!

6. Martin Colello | October 22, 2014 at 3:27 pm | Permalink

If I snap the volume, and then roll back the snap, does that mean all filesystems on that volume are rolled back also?

Please email me if you know the answer.

7. Aaron Toponce | December 9, 2014 at 10:52 am | Permalink

   I'm not sure what you mean by "volume". You snapshot datasets. The snapshot then keeps a delta of any deltas after the snapshot. If you rollback the snapshot, then the data on the dataset will be restored to the time you took the snapshot.

8. HSN | March 30, 2015 at 9:48 am | Permalink

   When you destroy a snapshot you are not actually destroying user data from that snapshot, correct? I mean the most recent data in the dataset will still have all your up-to-the instant changes. If that is true then destroying a snapshot effectively merges that with the previous one in the chain.

9. Aaron Toponce | April 2, 2015 at 9:48 am | Permalink

   Correct. When destroying the snapshot, you are only destroying the snapshot itself, not the data it points to.

10. Wade Fitzpatrick | July 12, 2015 at 11:01 pm | Permalink

    It might be worth mentioning that a snapshot is really just a reference to the merkle tree (see the Copy-on-write section) at the point in time that the snapshot was created. Therefore destroying the snapshot is only destroying the reference to the data in the snapshot. It is also the reason why creating and deleting snapshots is instantaneous.

    If you have a file in a dataset, snapshot the dataset then remove the file from the original dataset, you can still access it via the snapshot. The original dataset no longer has a reference to the file but the snapshot does. So when you destroy the dataset, you remove the last remaining reference to it and ZFS can then re-use that space.

11. Pritchard Musonda | April 25, 2016 at 12:40 pm | Permalink

    Just wanted to say this is the most comprehensive set of tutorials on ZFS for linux available on the internet. Thank you!

12. Simone Baglioni | May 4, 2017 at 8:49 am | Permalink

    Hi, excellent work. I always go back to this pages about ZFS. I think there's a typo here: " The clone does not need to reside in the same dataset as the clone, but it does need to reside in the same storage pool."... one of the two "clone" should be "snapshot".

{ 6 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part IV- The Adjustable Replacement Cache | December 20, 2012 at 8:08 am | Permalink

   [...] Snapshots and Clones [...]

2. Aaron Toponce : ZFS Administration, Part XV- iSCSI, NFS and Samba | January 1, 2013 at 11:04 am | Permalink

   [...] Snapshots and Clones [...]

3. Aaron Toponce : ZFS Administration, Part XVII- Best Practices and Caveats | January 7, 2013 at 9:18 pm | Permalink

   [...] Snapshots and Clones [...]

4. Aaron Toponce : ZFS Administration, Part XVI- Getting and Setting Properties | January 7, 2013 at 9:18 pm | Permalink

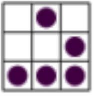
   [...] Snapshots and Clones [...]

5. Aaron Toponce : ZFS Administration, Part XIII- Sending and Receiving Filesystems | April 19, 2013 at 4:59 am | Permalink

   [...] Snapshots and Clones [...]

6. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 8:15 am | Permalink

   [...] Snapshots and Clones [...]



Aaron Toponce

{ 2012.12.20 }

ZFS Administration, Part XIII- Sending and Receiving Filesystems

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

Now that you're a pro at snapshots, we can move to one of the crown jewels of ZFS- the ability to send and receive full filesystems from one host to another. This is epic, and I am not aware of any other filesystem that can do this without the help of 3rd party tools, such as "dd" and "nc".

ZFS Send

Sending a ZFS filesystem means taking a snapshot of a dataset, and sending the snapshot. This ensures that while sending the data, it will always remain consistent, which is crux for all things ZFS. By default, we send the data to a file. We then can move that single file to an offsite backup, another storage server, or whatever. The advantage a ZFS send has over "dd", is the fact that you do not need to take the filesystem offilne to get at the data. This is a Big Win IMO.

To send a filesystem to a file, you first must make a snapshot of the dataset. After the snapshot has been made, you send the snapshot. This produces an output stream, that must be redirected. As such, you would issue something like the following:

```
# zfs snapshot tank/test@tuesday
# zfs send tank/test@tuesday > /backup/test-tuesday.img
```

Now, your brain should be thinking. You have at your disposal a whole suite of Unix utilities to manipulate data. So, rather than storing the raw data, how about we compress it with the "xz" utility?

```
# zfs send tank/test@tuesday | xz > /backup/test-tuesday.img.xz
```

Want to encrypt the backup? You could use OpenSSL or GnuPG:

```
# zfs send tank/test@tuesday | xz | openssl enc -aes-256-cbc -a -salt > /backup/test-tuesday.img.xz.asc
```

ZFS Receive

Receiving ZFS filesystems is the other side of the coin. Where you have a data stream, you can import that data into a full writable filesystem. It wouldn't make much sense to send the filesystem to an image file, if you can't really do anything with the data in the file.

Just as "zfs send" operates on streams, "zfs receive" does the same. So, suppose we want to receive the "/backup/test-tuesday.img" filesystem. We can receive it into any storage pool, and it will create the necessary dataset.

```
# zfs receive tank/test2 < /backup/test-tuesday.img
```

Of course, in our sending example, I compressed and encrypted a sent filesystem. So, to reverse that process, I do the commands in the reverse order:

```
# openssl enc -d -aes-256-cbc -a -in /storage/temp/testzone.gz.ssl | unxz | zfs receive tank/test2
```

The "zfs recv" command can be used as a shortcut.

Combining Send and Receive

Both "zfs send" and "zfs receive" operate on streams of input and output. So, it would make sense that we can send a filesystem into another. Of course we can do this locally:

```
# zfs send tank/test@tuesday | zfs receive pool/test
```

This is perfectly acceptable, but it doesn't make a lot of sense to keep multiple copies of the filesystem on the same storage server. Instead, it would make better sense to send the filesystem to a remote box. You can do this trivially with OpenSSH:

```
# zfs send tank/test@tuesday | ssh user@server.example.com "zfs receive pool/test"
```

Check out the simplicity of that command. You're taking live, running and consistent data from a snapshot, and sending that data to another box. This is epic for offsite storage backups. On your ZFS storage servers, you would run frequent snapshots of the datasets. Then, as a nightly cron job, you would "zfs send" the latest snapshot to an offsite storage server using "zfs receive". And because you are running a secure, tight ship, you encrypt the data with OpenSSL and XZ. Win.

Conclusion

Again, I can't stress the simplicity of sending and receiving ZFS filesystems. This is one of the biggest features in my book that makes ZFS a serious contender in the storage market. Put it in your nightly cron, and make offsite backups of your data with ZFS sending and receiving. You can send filesystems without unmounting them. You can change dataset properties on the receiving end. All your data remains consistent. You can combine it with other Unix utilities.

It's just pure win.

Posted by Aaron Toponce on Thursday, December 20, 2012, at 6:00 am.

Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 12 } Comments

1. Niek Bergboer | February 12, 2013 at 12:26 pm | Permalink

   Snapshots, and the ability to send and receive them, makes for automatic incremental backups: http://freecode.com/projects/zrep uses this, and you can trivially do hourly (or even more often) filesystem replication, keeping all snapshots.

2. max | April 18, 2013 at 8:32 am | Permalink

   Thanks for zfs guide. Its very helpfull.
   But I have a trouble with send-receive.
   I have two pools in my system. I tried to send snapshot from first pool to second. First dataset had compression

property.
I tried to execute command such as:
zfs send pool750/data@01 | zfs receive -o compression=gzip pool250

But system said:

invalid option 'o'
usage:
receive [-vnFu]
receive [-vnFu] [-d | -e]

For the property list, run: zfs set|get

For the delegated permission list, run: zfs allow|unallow

I have last version of ubuntu-zfs package.
What is wrong?

P.S. Sorry for my english

3. Aaron Toponce | April 18, 2013 at 9:31 am | Permalink

   According to the zfs(8) manpage, "zfs receive -o" is not a valid option. "zfs receive | recv [-vnFu] filesystem|volume|snapshot" or "zfs receive | recv [-vnFu] [-d|-e] filesystem".

   By default, ZFS saves dataset settings when sending them to another location. If you want to change a setting, then you will need to run a separate "zfs set" command after the sending of the dataset has finished.

4. max | April 18, 2013 at 11:25 pm | Permalink

   >>According to the zfs(8) manpage, “zfs receive -o” is not a valid option.

   Yes, but in your post was written about this feature.

   >> By default, ZFS saves dataset settings when sending them to another location.

   Hm... I tried to send about 280GB from gziped dataset to another pool yesterday. The pool was about 320GB free space. This was epic fail for me because compression wasn't automatically set on destination dataset.

5. Aaron Toponce | April 19, 2013 at 2:27 am | Permalink

   "zfs create" has the -o switch for setting dataset parameters. "zfs send" and "zfs receive" do not have this switch, and no where in this post do I state that "-o" is valid for "zfs send" or "zfs receive". If you run "zfs get compress pool/dataset" on your dataset, and "gzip" is returned, then when using "zfs send" you need to use the " -p" switch:

   > -p
   >
   > Include the dataset's properties in the stream. This flag is implicit when -R is specified. The receiving system must also support this feature.

6. max | April 19, 2013 at 3:03 am | Permalink

   Thanks a lot.

7. Warren Downs | July 1, 2013 at 5:40 pm | Permalink

   Regarding 'no where in this post do I state that “-o” is valid for “zfs send” or “zfs receive”', you may wish to change the above post to make that statement true (quoting original post):

   Further, you can change dataset properties on the receiving end. If the sending filesystem was not compressed, but you wish to compress the data on the receiving side, you can enable that as follows:

zfs send tank/test@tuesday | ssh user@server.example.com "zfs receive -o compression=lzjb pool/test"

8. syed | October 28, 2013 at 12:57 pm | Permalink

   Thanks, really appreciate it. Was very helpful.

9. Broen van Besien | January 11, 2015 at 11:47 am | Permalink

   Sent and receive of entire snapshots is a big win indeed.
   And it's even more powerful:
   You can send incremental updates out-of-the-box by using the -i option, which will only send the difference between a current and a previous snapshot.

10. NM | February 4, 2015 at 5:48 pm | Permalink

    The incremental updates suggested by Broen van Besien are a big plus. This could enable a portable/carryable system: send to laptop in the morning, work on it, send back to desktop when arriving home.
    Feasible?

11. Aaron Toponce | February 17, 2015 at 1:35 pm | Permalink

    > The incremental updates suggested by Broen van Besien are a big plus. This could enable a portable/carryable system: send to laptop in the morning, work on it, send back to desktop when arriving home. Feasible?

    Sure. I send incremental snapshots from my workstation to my external backup drive every 15 minutes. Works like a charm. I don't see any problems with this for a laptop going to a backup server.

12. Alan Galitz MD | December 8, 2015 at 9:27 pm | Permalink

    While it has been over 2 years, perhaps a how to on migrating a large pool from one array to another might be worth including.

    This link https://pthree.org/2012/12/20/zfs-administration-part-xiii-sending-and-receiving-filesystems/ covers the basic steps, but I and others, judging by all the posts I found, hit several problems whose solutions are not well listed and thought since so many people use your site as a reference this might help. Perhaps this issue is cropping up with the addition and changes to the auto-snapshot implementation in zfs on linux.

    I have several home servers with 4-5TB if data each. They store identical copies of media, crashplan backup files, desktop/laptop files (yes the crashplan goes off site too, but the media is too large so 3 separate machines with redundant zfs is the backup plan for some the the large media). The send operation is the better part of a day.

    A few years ago I followed your excellent advice -THANKS- and setup Raidz with 3 x4TB on 2 machines and 4x3TB on one. Well now it turns out that this no longer best practice so after some drives began failing from age - no data loss- yea ZFS scrub! - time to covert to mirrors and stripes. So with some dual USB3 toasters, I build a 7TB array with mirrors 2x2 to temp store the data. So time to send the data to the temp array:

    # zfs snapshot -r zsrv@01
    # zfs send -R zsrv@01 | zfs receive -Fdvu zsrv-old

    Note, don't keep your terminal open while doing this as a sleep/hibenation might end the operation. Detach from the session to resume later. I use byobu since it comes with xubuntu.

    OK, to avoid massive fail, I found it necessary to turn OFF auto-shapshots. The new dataset started doing frequent snapshots and their presence killed the send/recv randomly. Perhaps this is a bug, the the first send/recv (no "-i") doesn't like a pre-exiisting data set)

    The important error message was just before a long string of "...broken pipe..." in SSH session. I found it at the machine's display since the volume blew past the SSH buffer. The 'v' switch on the recv produces alot of output if you have hundreds of old snapshots.

Also, since I was sending nfs shares, both the old and new datasets used the same folder, causing mounting errors, but that's later. I set the the top level dataset with the pool name to not mount or send shares but since the send -R causes properties to be copied the new pools switch them back on. The -u option is supposed to stop local mounting, but the nfs shares mounted locally anyway it seems. I'm not a pro so do not have the time to test much or know.

It did not help that without the nfs share mounted by zfs, crashplan periodically would create a local folder, blocking the creation of the backup target dataset. So before the first full stream, check the usualy mounting place to ensure they really are empty and

zfs list -t snapshot | grep new # or a unique piece of the new pool

OK, so for the final incremental send, ensure the disk isn't in use (shutdown iSCSI, NFS, syslog, crashplan,etc) see above
Make another snapshot of 'zsrv' and incrementally send it to 'zsrv-new' (this should take a short time)

zfs snapshot -r zsrv@02
zfs send -R -i zsrv@01 zsrv@02 | zfs recv -dvu zsrv-new

Now the instructions included a way to bring back the old pool and do some comparison. for me looked at data set sizes and opened many folders to see how things worked.
So if you want to have both pools, pay attention to exported (nfs, smb etc) mounts and

Set 'srv' as readonly
Export the two zpools
Rename and import the original zpool as'zsrv-old'
Rename an import the new zpool as 'zsrv'

Once all is well, then zpool to rebuild using the old drives and required replacements. Again, I went from Raidz to mirrored stripes.

As you likely know, automatic snapshots were added to ZFS on linux as an external module and looks like now part of the main build. (Please pardon my terminology as I do this for my home servers not professionally). So, in summary, it looks like this has been leading or errors or perhaps a bug with at times the newly created pool starting to make auto-snapshots and cause the receive to fail. It may be that since I was sending the top level dataset, with its children with one command, that brought this out. I have seen with the "broken pipe" people saying it worked better to send datasets by themselves and that might prevent some of these problem at the cost if extra steps.

I share this with you as adding this information would be helpful to many others judging by the many posts and frustration I've seen. I hope this helps others know to shut off any auto-snapshots while doing the initial send with large datasets which seem to bring this out.

cheers

{ 6 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part XI- Compression and Deduplication | December 20, 2012 at 8:06 am | Permalink

   [...] Sending and Receiving Filesystems [...]

2. Aaron Toponce : ZFS Administration, Part V- Exporting and Importing zpools | December 20, 2012 at 8:08 am | Permalink

   [...] Sending and Receiving Filesystems [...]

3. Aaron Toponce : Install ZFS on Debian GNU/Linux | December 20, 2012 at 8:10 am | Permalink

   [...] Sending and Receiving Filesystems [...]

4. Aaron Toponce : ZFS Administration, Part I- VDEVs | December 29, 2012 at 6:19 am | Permalink

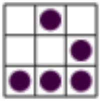
[...] Sending and Receiving Filesystems [...]

5. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | February 20, 2013 at 8:34 am | Permalink

   [...] Sending and Receiving Filesystems [...]

6. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:03 am | Permalink

   [...] Sending and Receiving Filesystems [...]



Aaron Toponce

{ 2012.12.21 }

ZFS Administration, Part XIV- ZVOLS

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

What is a ZVOL?

A ZVOL is a "ZFS volume" that has been exported to the system as a block device. So far, when dealing with the ZFS filesystem, other than creating our pool, we haven't dealt with block devices at all, even when mounting the datasets. It's almost like ZFS is behaving like a userspace application more than a filesystem. I mean, on GNU/Linux, when working with filesystems, you're constantly working with block devices, whether they be full disks, partitions, RAID arrays or logical volumes. Yet somehow, we've managed to escape all that with ZFS. Well, not any longer. Now we get our hands dirty with ZVOLs.

A ZVOL is a ZFS block device that resides in your storage pool. This means that the single block device gets to take advantage of your underlying RAID array, such as mirrors or RAID-Z. It gets to take advantage of the copy-on-write benefits, such as snapshots. It gets to take advantage of online scrubbing, compression and data deduplication. It gets to take advantage of the ZIL and ARC. Because it's a legitimate block device, you can do some very interesting things with your ZVOL. We'll look at three of them here- swap, ext4, and VM storage. First, we need to learn how to create a ZVOL.

Creating a ZVOL

To create a ZVOL, we use the "-V" switch with our "zfs create" command, and give it a size. For example, if I wanted to create a 1 GB ZVOL, I could issue the following command. Notice further that there are a couple new symlinks that exist in /dev/zvol/tank/ and /dev/tank/ which points to a new block device in /dev/:

```
# zfs create -V 1G tank/disk1
# ls -l /dev/zvol/tank/disk1
lrwxrwxrwx 1 root root 11 Dec 20 22:10 /dev/zvol/tank/disk1 -> ../../zd144
# ls -l /dev/tank/disk1
lrwxrwxrwx 1 root root 8 Dec 20 22:10 /dev/tank/disk1 -> ../zd144
```

Because this is a full fledged, 100% bona fide block device that is 1 GB in size, we can do anything with it that we would do with any other block device, and we get all the benefits of ZFS underneath. Plus, creating a ZVOL is near instantaneous, regardless of size. Now, I could create a block device with GNU/Linux from a file on the filesystem. For example, if running ext4, I can create a 1 GB file, then make a block device out of it as follows:

```
# fallocate -l 1G /tmp/file.img
# losetup /dev/loop0 /tmp/file.img
```

I now have the block device /dev/loop0 that represents my 1 GB file. Just as with any other block device, I can format it, add it to swap, etc. But it's not as elegant, and it has severe limitations. First off, by default you only have 8 loopback devices for your exported block devices. You can change this number, however. With ZFS, you can create 2^64 ZVOLs by default. Also, it requires a preallocated image, on top of your filesystem. So, you are managing three layers of data: the block device, the file, and the blocks on the filesystem. With ZVOLs, the block device is exported right off the storage pool, just like any other dataset.

Let's look at some things we can do with this ZVOL.

Swap on a ZVOL

Personally, I'm not a big fan of swap. I understand that it's a physical extension of RAM, but swap is only used when RAM fills, spilling the cache. If this is happening regularly and consistently, then you should probably look into getting more RAM. It can act as part of a healthy system, keeping RAM dedicated to what the kernel actively needs. But, when active RAM starts spilling over to swap, then you have "the swap of death", as your disks thrash, trying to keep up with the demands of the kernel. So, depending on your system and needs, you may or may not need swap.

First, let's create 1 GB block device for our swap. We'll call the dataset "tank/swap" to make it easy to identify its intention. Before we begin, let's check out how much swap we currently have on our system with the "free" command:

```
# free
             total       used       free     shared    buffers     cached
Mem:      12327288    8637124    3690164          0     175264    1276812
-/+ buffers/cache:    7185048    5142240
Swap:            0          0          0
```

In this case, we do not have any swap enabled. So, let's create 1 GB of swap on a ZVOL, and add it to the kernel:

```
# zfs create -V 1G tank/swap
# mkswap /dev/zvol/tank/swap
# swapon /dev/zvol/tank/swap
# free
             total       used       free     shared    buffers     cached
Mem:      12327288    8667492    3659796          0     175268    1276804
-/+ buffers/cache:    7215420    5111868
Swap:      1048572          0    1048572
```

It worked! We have a legitimate Linux kernel swap device on top of ZFS. Sweet. As is typical with swap devices, they don't have a mountpoint. They are either enabled, or disabled, and this swap device is no different.

Ext4 on a ZVOL

This may sound wacky, but you could put another filesystem, and mount it, on top of a ZVOL. In other words, you could have an ext4 formatted ZVOL and mounted to /mnt. You could even partition your ZVOL, and put multiple filesystems on it. Let's do that!

```
# zfs create -V 100G tank/ext4
# fdisk /dev/tank/ext4
( follow the prompts to create 2 partitions- the first 1 GB in size, the second to fill the rest )
# fdisk -l /dev/tank/ext4
```

```
Disk /dev/tank/ext4: 107.4 GB, 107374182400 bytes
16 heads, 63 sectors/track, 208050 cylinders, total 209715200 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 8192 bytes
I/O size (minimum/optimal): 8192 bytes / 8192 bytes
Disk identifier: 0x000a0d54

          Device Boot      Start         End      Blocks   Id  System
/dev/tank/ext4p1            2048     2099199     1048576   83  Linux
/dev/tank/ext4p2         2099200   209715199   103808000   83  Linux
```

Let's create some filesystems, and mount them:

```
# mkfs.ext4 /dev/zd0p1
# mkfs.ext4 /dev/zd0p2
# mkdir /mnt/zd0p{1,2}
# mount /dev/zd0p1 /mnt/zd0p1
# mount /dev/zd0p2 /mnt/zd0p2
```

Enable compression on the ZVOL, copy over some data, then take a snapshot:

```
# zfs set compression=lzjb pool/ext4
# tar -cf /mnt/zd0p1/files.tar /etc/
# tar -cf /mnt/zd0p2/files.tar /etc /var/log/
# zfs snapshot tank/ext4@001
```

You probably didn't notice, but you just enabled transparent compression and took a snapshot of your ext4 filesystem. These are two things you can't do with ext4 natively. You also have all the benefits of ZFS that ext4 normally couldn't give you. So, now you regularly snapshot your data, you perform online scrubs, and send it offsite for backup. Most importantly, your data is consistent.

ZVOL storage for VMs

Lastly, you can use these block devices as the backend storage for VMs. It's not uncommon to create logical volume block devices as the backend for VM storage. After having the block device available for Qemu, you attach the block device to the virtual machine, and from its perspective, you have a "/dev/vda" or "/dev/sda" depending on the setup.

If using libvirt, you would have a /etc/libvirt/qemu/vm.xml file. In that file, you could have the following, where "/dev/zd0" is the ZVOL block device:

```
<disk type='block' device='disk'>
  <driver name='qemu' type='raw' cache='none'/>
  <source dev='/dev/zd0'/>
  <target dev='vda' bus='virtio'/>
  <alias name='virtio-disk0'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0'/>
</disk>
```

At this point, your VM gets all the ZFS benefits underneath, such as snapshots, compression, deduplication, data integrity, drive redundancy, etc.

Conclusion

ZVOLs are a great way to get to block devices quickly while taking advantage of all of the underlying ZFS features. Using the ZVOLs as the VM backing storage is especially attractive. However, I should note that when using ZVOLs, you cannot replicate them across a cluster. ZFS is not a clustered filesystem. If you want data replication across a cluster, then you should not use ZVOLs, and use file images for your VM backing storage instead. Other than that, you get all of the amazing benefits of ZFS that we have been blogging about up to this point, and beyond, for whatever data resides on your ZVOL.

Posted by Aaron Toponce on Friday, December 21, 2012, at 6:00 am.

Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 13 } Comments

1. Ahmed Kamal | July 22, 2013 at 2:56 pm | Permalink

   When you put ext4 on top of a ZVOL, and snapshot it .. You say it's "consistent" I guess it's only crash-consistent .. There is no FS/ZVOL integration to ensure better consistency, right

2. Aaron Toponce | August 7, 2013 at 10:08 am | Permalink

   Not sure what you're saying. When you put ext4 on top of a ZVOL, ext4 is just a standard run-of-the-mill application wishing to stare data on ZFS just as much as anything else. So, the data is pooled into a TXG just as anything else. TXGs are flushed in sequential order to the ZIL. The contents of the ZIL are flushed to disk synchronously. So, the data is always consistent.

   Suppose you have a VM that is using that ZVOL for its storage. Suppose further that your VM crashes. At worst case, the ext4 journel is not closed. So, at next boot, you will be forced to fsck(8) the disk. What's important to know, is that the data on ZFS is still consistent, even if ext4 may have lost some data as a result of the crash. In other words, closing the journal did not happen before the rest of the data blocks were flushed to disk.

3. Jack Relish | August 26, 2013 at 3:25 pm | Permalink

   Excellent guide. I was wondering if you had any insights on mounting partitions that exist on a ZVOL?

   For example, lets say that I have a ZVOL /dev/tank/vm0, which was used as the root device for a VM. At some point, the VM breaks or for whatever other reason I want to be able to access the contents of it's filysystem. Is it possible to expose the internals of the ZVOL? I'm sure that it could be done manually an tediously by getting the start offset of the partition and then mounting it in the same way you would a raw image file, but if there is a slicker way to do so that would be incredible.

4. Zyon | March 10, 2014 at 7:17 pm | Permalink

   When you say "you cannot replicate them across a cluster", that means I cannot use DRBD over ZVOL?

5. Andy | April 12, 2014 at 6:58 am | Permalink

   @Jack

   kpartx is probably what you'd need for getting at partitions within a ZVOL holding a VM's disk. Certainly it works for images taken from entire real disks using dd.

   Andy

6. sammand | June 3, 2014 at 9:26 pm | Permalink

   @Zyon

Yes ZVOLs can be replicated using DRBD and we support it in ZBOSS Linux distribution.
You are free to check it out http://www.zettalane.com

7. cbu | October 20, 2014 at 10:31 am | Permalink

   Hi,
   First thank for this incredible tutorial.
   I created a zvol and I am trying to attach it to a VM but I could not. Every time that I try to create a new Block device Storage Pool from virt-manager I get:

   RuntimeError: Could not start storage pool: internal error: Child process (/usr/bin/mount -t auto /dev/zvol/tank/rhel/disk1 /var/lib/libvirt/images/sss) unexpected exit status 32: mount: /dev/zd0 is write-protected, mounting read-only
   mount: unknown filesystem type '(null)'

   How can I get a block device available for Qemu? Thanks!

   P.S: I am using centOS 7 and zfs-0.6.3-1.1

8. Mark | February 17, 2015 at 1:43 pm | Permalink

   Since ZFS is so vulnerable to fragmentation, would BTRFS on ZVOLs be a working combination? You'd miss the shared space usage between datasets, but you'd gain the advantages of the now stable BTRFS filesystem, while maintaining the reliability of RAID using ZFS. BTRFS RAID is still unstable, but it can defragment. And one can grow a ZVOL and BTRFS when needed. BTRFS is still evolving, while ZFS is stable, but Oracle will likely never release the new code (goodbye encryption) and the v5000 code isn't evolving much either. Wonder where compression would be more effective though. What's your opinion?

9. Luca | September 8, 2015 at 5:49 am | Permalink

   Thanks for these very interesting articles on zfs!
   I want to set up some VM with Xen on ZFS and it is not clear to me which is the best solution for guest disk images: on LVM I use one LV and, when it fills up, I have to extend it. On ZFS I suppose I must create a ZVOL of some extent but what do to when is full? What is the smartest way to manage the VMs on ZFS?

10. John Naggets | October 10, 2015 at 5:57 am | Permalink

    How do you create a ZVOL using 100% of the available size with its -V option?

11. Ekkehard | April 25, 2016 at 2:55 am | Permalink

    Thanks for your awesome series, it was the only resource I studied before diving into ZFS, and I feel like I have a quite deep understanding now; creating and deploying my first (home) NAS with ZFS (published over Samba and rsync) was a snap.

    I have tried iSCSI as well (I use a FreeBSD-based NAS), it works beautifully to provide partitions to Win7 as "native" NTFS drives.

    Obviously, some of the benefits of datasets do not apply to ZVOLS (i.e., ZFS cannot know which blocks are actually in use and which are not, when the file system has been "cycled" a few times), I wonder whether you have practical experience with how ZVOLs evolve over time.

    For example, say I use a 100GB NTFS-formatted ZVOL to backup a windows partition (using robocopy, some Windows imaging software, or whatever other tool), to be able to keep all the windows permissions that do not survive with rsync- or Samba/CIFS-based copying. I don't know much about NTFS internals, but I assume that NTFS will, sooner than later, have touched every block at least once; at this point, ZFS will see the 100GB as active. Every further block change will directly lead to more use (at least when there are snapshots around). When I delete or "overwrite" files (in the NTFS world), ZFS will not notice, etc..

Compared to a dataset, where ZFS knows about actual files, the ZVOL will thus have a drawback as long as the dataset is smaller. But if I compare a 100G ZVOL with a dataset that actually *uses* 100G of data, it should pretty much be the same (in terms of using up storage in the pool), no matter what file operations I do, right? (All with deduplication off, of course).

Regarding compression, there should not be a noticable difference between a ZVOL and a dataset, right?

I have a hourly-daily-weekly-monthly snapshot scheme; would you say ZVOLs will eat up the space quicker than a comparable dataset when many snapshots are in use?

12. Mael Strom | February 7, 2017 at 1:20 am | Permalink

    It maybe necroposting, but it can make a difference...

    2Ekkehard: in your case you need to use sparse zvols (-s key) and enable iscsi export to acts like ssd, with rpm=1 and unmap=on, and use windows 8 or above (XP, Vista and 7 are unable to send unmap command through iscsi). So just used (or touched by snapshot) blocks will be keep, others will discard.

13. Anonimous | February 20, 2017 at 8:26 am | Permalink

    Just as a point, not to cause any discussion.

    I had seen some apps that refuse to work is there is no swap area defined (some als refuse to even start, no swap error message shown, etc).

    I had seen some apps that causes swap area to be used among there is plenty free ram at the same time (they are not so common, thanks who know why, etc); by the way, how an app can force data go to swap when there is free ram ata same time?

    But the worst case is when you can not add more ram to your motherboard (when i personally by motherboards i also buy the top most ram they can support), some motherboards (quite old, or not so much) only allows 2GiB of RAM (talking about PC, not laptops, etc).

    And there is also the other part counting, what if adding RAM is multiply the cost of the entire PC by four or five times? Example: A laptop with touch screen that can be turned (TabletPC) with 3GiB of ram, with a max of 4GiB said by vendor, but having some people tested it with 8GiB and with 16GiB (really max, there are no bigger ram than 8GiB and it only has two modules), now the costs, the 4GiB module (2x4GiB=8GiB) cost arround 300 euros each (the tablet cost 300 euros with 3GiB of ram), so putting 8GiB is making the tabletPC cost the triple, but the 8GiB module (2x8GiB=16GiB) cost more than one thusand euros each, both arround 2500 euros, so cost would be more than eight times the cost of the tabletpc... and much much more than a new computer.

    Sometimes adding more ram is not an option, some can not hold more than 2GiB, others is too much expensive.

    So could you explain a little how ZFS would go in a system with 2GiB RAM and 4GiB on SWAP with only one disk (laptop) of 500GiB? Understanding no dedup is being used, of course; and the top most important point: how to configure it to not be a pain in terms of speed!

    I mean: Ext4 is great (i had no loose i had noticed), but i can not trust it for silent file changes... i do not mind if HDD breaks, i have OffLine backUPs...

    I better explain it a little: If i use Ext4 for BackUPs on external media, silent changes can occur, if i use ZFS they will be detected (at leas most of them); since i use 3 to 7 external HDDs, only one powered at same time because i am a really paranoid on loosing my data (tutorials maid by me), all with Ext4, i can suffer from silent corruption (never seen it yet, but not impossible), ZFS would be great to detect them if they occur.

Till i can use ZFS i may think my method to avoid silent corruption is great, i use 7-Zip to compress LZMA2 one directory or file, then i put such 7z file on one external disk, then unplug it, then on another, ... up to 7 disks... 7-Zip has an internal checksum, but how can i be sure all 7 copies had not have a silent corruption ata the same time? so i can not recover data from inside 7z files (all copies are bad)... to avoid at most that, i check 7z integrity prior to copy it on the 7 external disks... but it does not warranty at all co corruption can occur.

If i just can put ZFS on each of that 7 external DISKs i would have another level of trust.

By the way... a lot of times the Ext4 has been powered of (freeze) at brute force... but i am really lucky, i never lost anything, neither seen any of such silent changes... but i am paranoid, they can happen, so better to be safe.

Resuming: How would you configure ZFS for rootfs (i do not like to create partitions for /home, etc, since i am so paranoid i make periodically full clones of all system on external media) for a laptop (only one hdd) with only 2GiB (3GiB at most) of RAM, with 500GiB HDD, but only 64GiB for rootfa and 64GiB for data partition, the rest is used by other OSs... better is you can explain it for SolidXK distro, thanks; thinking of having a similar response as having a Ext4 over a LUKs over a LUKs over a LUKs over a logical partition (i hate primary partitions)... and of course, having encryption enabled (better if ZFS encryption with cascade of TwoFish and Serpent algoriths, since i collaborate on coding the break of AES-128 up to AES-8192).

Thanks in advance for any help, and also thanks for your great turorial i am reading with pleasure.

{ 3 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part XI- Compression and Deduplication | January 7, 2013 at 9:24 pm | Permalink

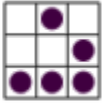
   [...] ZVOLs [...]

2. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | May 9, 2013 at 6:00 am | Permalink

   [...] ZVOLs [...]

3. Thoughts and feelings on data storage implementations after four years of immersion. | EpiJunkie | March 22, 2014 at 11:56 am | Permalink

   […] finally booted I would have to reconfigure the iSCSI LUNs due to the encryption. After the LUNs/zvols were reconfigured they were presented to the ESXi machine via iSCSI as a datastore which contained […]



Aaron Toponce

{ 2012.12.31 }

ZFS Administration, Part XV- iSCSI, NFS and Samba

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

I spent the previous week celebrating the Christmas holiday with family and friends, and as a result, took a break from blogging. However, other than the New Year, I'm finished with holidays for a while, and eager to get back to blogging, and finishing off this series. Only handful of posts left to go. So, let's continue our discussion with ZFS administration on GNU/Linux, by discussing sharing datasets.

Disclaimer

I have been trying to keep these ZFS posts as operating system agnostic as much as possible. Even though they have had a slant towards Linux kernels, you should be able to take much of this to BSD or any of the Solaris derivatives, such as OpenIndiana or Nexenta. With this post, however, it's going to be Linux-kernel specific, and even Ubuntu and Debian specific at that. The reason being is iSCSI support is not compiled into ZFS on Linux as of the writing of this post, but sharing via NFS and SMB is. Further, the implementation details for sharing via NFS and SMB will be specific to Debian and Ubuntu in this post. So, you may need to make adjustments if using Fedora, openSUSE, Gentoo, Arch, et cetera.

Motivation

You are probably asking why you would want to use ZFS specific sharing of datasets rather than using the "tried and true" methods with standard software. The reason is simple. When the system boots up, and goes through its service initialization process (typically by executing the shell scripts found in /etc/init.d/), it has a method to the madness on which service starts first. Loosely speaking, filesystems are mounted first, networking is enabled, then services are started at last. Some of these are tied together, such as NFS exports, which requires the filesystem to be mounted, a firewall in place, networking started, and the NFS daemon running. But, what

happens when the filesystem is not mounted? If the directory is still accessible, it will be exported via NFS, and applications could begin dumping data into the export. This could lead to all sorts of issues, such as data inconsistencies. As such, administrators have put checks into place, such as exporting only nested directories in the mount point, which would not be available if the filesystem fails to mount. These are clever hacks, but certainly not elegant.

When tying the export directly into the filesystem, you can solve this beautifully, which ZFS does. In the case of ZFS, you can share a specific dataset via NFS, for example. However, if the dataset does not mount, then the export will not be available to the application, and the NFS client will block. Because the network share is inherent to the filesystem, there is no concern for data inconsistencies, and no need for silly check hacks or scripts. As a result, ZFS from Oracle has the ability to share a dataset via NFS, SMB (CIFS or Samba) and iSCSI. ZFS on Linux only supports NFS and SMB currently, with iSCSI support on the way.

In each case, you still must install the necessary daemon software to make the share available. For example, if you wish to share a dataset via NFS, then you need to install the NFS server software, and it must be running. Then, all you need to do is flip the sharing NFS switch on the dataset, and it will be immediately available.

Sharing via NFS

To share a dataset via NFS, you first need to make sure the NFS daemon is running. On Debian and Ubuntu, this is the "nfs-kernel-server" package. Further, with Debian and Ubuntu, the NFS daemon will not start unless there is an export in the /etc/exports file. So, you have two options: you can create a dummy export, only available to localhost, or you can edit the init script to start without checking for a current export. I prefer the former. Let's get that setup:

```
$ sudo aptitude install -R nfs-kernel-server
$ echo '/mnt localhost(ro)' >> /etc/exports
$ sudo /etc/init.d/nfs-kernel-server start
$ showmount -e hostname.example.com
Export list for hostname.example.com:
/mnt localhost
```

With our NFS daemon running, we can now start sharing ZFS datasets. I'll assume already that you have created your dataset, it's mounted, and you're ready to start committing data to it. You'll notice in the zfs(8) manpage, that for the "sharenfs" property, it can be "on", "off" or "opts", where "opts" are valid NFS export options. So, if I wanted to share my "pool/srv" dataset, which is mounted to "/srv" to the 10.80.86.0/24 network, I could do something like:

```
# zfs set sharenfs="rw=@10.80.86.0/24" pool/srv
# zfs share pool/srv
# showmount -e hostname.example.com
Export list for hostname.example.com:
/srv 10.80.86.0/24
/mnt localhost
```

If you want your ZFS datasets to be shared on boot, then you need to install the /etc/default/zfs config file. If using the Ubuntu PPA, this will be installed by default for you. If compiling from source, this will not be provided. Here are the contents of that file. I've added emphasis to the two lines that should be modified for persistence across boots, if you want to enable sharing via NFS. Default is 'no':

```
$ cat /etc/default/zfs
# /etc/default/zfs
#
# Instead of changing these default ZFS options, Debian systems should install
# the zfs-mount package, and Ubuntu systems should install the zfs-mountall
# package. The debian-zfs and ubuntu-zfs metapackages ensure a correct system
# configuration.
```

```
#
# If the system runs parallel init jobs, like upstart or systemd, then the
# `zfs mount -a` command races in a way that causes sporadic mount failures.

# Automatically run `zfs mount -a` at system start. Disabled by default.
ZFS_MOUNT='yes'
ZFS_UNMOUNT='no'

# Automatically run `zfs share -a` at system start. Disabled by default.
# Requires nfsd and/or smbd. Incompletely implemented for Linux.
ZFS_SHARE='yes'
ZFS_UNSHARE='no'
```

As mentioned in the comments, running a parallel init system creates problems for ZFS. This is something I recently banged my head against, as my /var/log/ and /var/cache/ datasets were not mounting on boot. To fix the problem, and run a serialized boot, thus ensuring that everything gets executed in the proper order, you need to touch a file:

```
# touch /etc/init.d/.legacy-bootordering
```

This will add time to your bootup, but given the fact that my system is up months at a time, I'm not worried about the extra 5 seconds this puts on my boot. This is documented in the /etc/init.d/rc script, setting the "CONCURRENCY=none" variable.

You should now be able to mount the NFS export from an NFS client:

```
(client)# mount -t nfs hostname.example.com:/srv /mnt
```

Sharing via SMB

Currently, SMB integration is not working 100%. See bug [#1170 I reported on Github](). However, when things get working, this will likely be the way.

As with NFS, to share a ZFS dataset via SMB/CIFS, you need to have the daemon installed and running. Recently, the [Samba development team released Samba version 4](). This release gives the Free Software world a Free Software implementation of Active Directory running on GNU/Linux systems, SMB 2.1 file sharing support, clustered file servers, and much more. Currently, Debian testing has the beta 2 packages. Debian experimental has the stable release, and it may make its way up the chain for the next stable release. One can hope. Samba v4 is not needed to share ZFS datasets via SMB/CIFS, but it's worth mentioning. We'll stick with version 3 of the Samba packages, until version 4 stabilizes.

```
# aptitude install -R aptitude install samba samba-client samba-doc samba-tools samba-doc-pdf
# ps -ef | grep smb
root     22413     1  0 09:05 ?        00:00:00 /usr/sbin/smbd -D
root     22423 22413  0 09:05 ?        00:00:00 /usr/sbin/smbd -D
root     22451 21308  0 09:06 pts/1    00:00:00 grep smb
```

At this point, all we need to do is share the dataset, and verify that it's been shared. It is also worth noting that Microsoft Windows machines are not case sensitive, as things are in Unix. As such, if you are in a heterogenious environment, it may be worth disabling case sensitivity on the ZFS dataset. Setting this value can only be done on creation time. So, you may wish to issue the following when creating that dataset:

```
# zfs create -o casesensitivity=mixed pool/srv
```

Now you can continue with configuring the rest of the dataset:

```
# zfs set sharesmb=on pool/srv
# zfs share pool/srv
```

```
# smbclient -U guest -N -L localhost
Domain=[WORKGROUP] OS=[Unix] Server=[Samba 3.6.6]

        Sharename       Type      Comment
        ---------       ----      -------
        print$          Disk      Printer Drivers
        sysvol          Disk
        netlogon        Disk
        IPC$            IPC       IPC Service (eightyeight server)
        Canon-imageRunner-3300 Printer   Canon imageRunner 3300
        HP-Color-LaserJet-3600 Printer   HP Color LaserJet 3600
        salesprinter    Printer   Canon ImageClass MF7460
        pool_srv        Disk      Comment: /srv
Domain=[WORKGROUP] OS=[Unix] Server=[Samba 3.6.6]

        Server               Comment
        ---------            -------
        EIGHTYEIGHT          eightyeight server

        Workgroup            Master
        ---------            -------
        WORKGROUP            EIGHTYEIGHT
```

You can see that in this environment (my workstation hostname is 'eightyeight'), there are some printers being shared, and a couple disks. I've emphasized the disk that we are sharing in my output, to verify that it is working correctly. So, we should be able to mount that share as a CIFS mount, and access the data:

```
# aptitude install -R cifs-utils
# mount -t cifs -o username=USERNAME //localhost/srv /mnt
# ls /mnt
foo
```

Sharing via iSCSI

Unfortunately, sharing ZFS datasets via iSCSI is not yet supported with ZFS on Linux. However, it is available in the Illumos source code upstream, and work is being done to get it working in GNU/Linux. As with SMB and NFS, you will need the iSCSI daemon installed and running. When support is enabled, I'll finish writing up this post on demonstrating how you can access iSCSI targets that are ZFS datasets. In the meantime, you would do something like the following:

```
# aptitude install -R openiscsi
# zfs set shareiscsi=on pool/srv
```

At which point, from the iSCSI client, you would access the target, format it, mount it, and start working with the data..

Posted by Aaron Toponce on Monday, December 31, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 12 } Comments

1. David E. Anderson | December 31, 2012 at 8:29 am | Permalink

   thanks again for the series of docs and in particular the boot-spanning and serial tips!

2. Roger Hunwicks | January 16, 2013 at 3:42 am | Permalink

   Thanks for the great series of posts - you've encouraged me to finally set up a ZFS server for experimenting with.

   On my Ubuntu 12.04.1 server I didn't need to create a dummy export in order to get NFS shares working - the supplied /etc/init.d/nfs-kernel-server checks for the existing of /etc/exports, but doesn't care what is in. The file exists by default but contains only comments, which is enough.

   Similarly, I am using the mountall from the Ubuntu ZFS PPA and that seems be working fine without changing ZFS_MOUNT in /etc/default/zfs

   Therefore the only change I needed was to change ZFS_SHARE to yes in /etc/default/zfs

3. Roger Hunwicks | January 16, 2013 at 3:52 am | Permalink

   FWIW I am also not using .legacy-bootordering and I haven't had any problems. That may be because the zfs pool I am using is all shares (NFS and SMB) and virtual machines (run using libvirt/KVM on the same server), and I am not using ZFS for the OS files (on a separate ext4 mdadm mirror)

4. Aaron Toponce | January 16, 2013 at 8:51 am | Permalink

   That's good to know about just the existence of the /etc/exports file, instead of its contents. Thanks for the heads up. Also, with regards to legacy booting, it's only an issue if using it for filesystem mounts, such as /var/, /home/, etc. Things outside of that, such as /kvm/ may or may not be affected, all depending on when your starting services need the mounts.

5. Lloyd | June 21, 2013 at 9:29 pm | Permalink

   Great work!

   For Ubuntu 13.04 it looks like you do need a dummy export. Without it this happens.

   lloyd@tank:/var/log$ sudo /etc/init.d/nfs-kernel-server start
   * Not starting NFS kernel daemon: no exports.

6. Fernando | February 20, 2014 at 1:17 pm | Permalink

   Hi there!

   I would like to share a ZFS dataset via SMB.

   At the end of the SMB section, you mounting the SMB share with: "mount -t cifs -o username=USERNAME //localhost/srv /mnt"

   Which username should I use here? My linux username? I'm asking because with my linux username I could not authenticate. Do I need to edit smb.conf in order to authenticate?

   Thanks!

   Fernando

7. Fernando | February 21, 2014 at 8:23 am | Permalink

   Hi guys!

   Nevermind my last question.. I solved my problem with:

   sudo smbpasswd -L -a your_username
   sudo smbpasswd -L -e your_username

   Thanks!

   Fernando

8. Mark | March 12, 2014 at 4:42 am | Permalink

   "Unfortunately, sharing ZFS datasets via iSCSI is not yet supported with ZFS on Linux"

   Doesn't this just refer to the meta data on the volume? If you create a zvol and hence a block device in the kernel, can't you just tell the IET (the iscsi target) to use that block device.

   Other than convenience, what other benefits are there for using the zfs command to create iscsi (and for that matter smb) shares?

9. Chris Scott | August 9, 2014 at 8:09 am | Permalink

   How can I remove a NFS share I've created? I shared my entire ZFS drive (as per http://www.latentexistence.me.uk/zfs-and-ubuntu-home-server-howto/) but now I think I'd prefer to share out only the folders I create within the ZFS pool.

   Can't figure out how to remove my ZFS shares altogether, though (apart from setting them to "off")

10. Mike | September 29, 2014 at 12:58 pm | Permalink

    Thanks for this series. It's been really helpful.

    Can someone point me to the zfs config file in CentOS7? I've installed zfs from the EPEL repository, but I don't have /etc/default/zfs. I don't see anything similar in /etc/zfs/ either.

11. NM | May 12, 2015 at 5:40 am | Permalink

    You closed bug #1170

12. Doug | December 21, 2015 at 2:06 pm | Permalink

    Re: SMB sahres

    I can't get this to work with the instructions. SMB is running. When I go to set the smbshare=on, I get :

    cannot share 'mypool/test2': smb add share failed

    But when I check it, it does turn it on:

    zfs get sharesmb mypool/test2
    NAME PROPERTY VALUE SOURCE
    mypool/test2 sharesmb on local

    But then telling it to share it:

zfs share mypool/test2
cannot share 'mypool/test2': smb add share failed

But running smbclient it does not show up as a share and therefore trying to mount with the smbclient fails.

{ 7 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | January 7, 2013 at 9:25 pm | Permalink

   […] iSCSI, NFS and Samba […]

2. Aaron Toponce : ZFS Administration, Part V- Exporting and Importing zpools | January 7, 2013 at 9:26 pm | Permalink

   […] iSCSI, NFS and Samba […]

3. Aaron Toponce : Install ZFS on Debian GNU/Linux | January 7, 2013 at 9:27 pm | Permalink

   […] iSCSI, NFS and Samba […]

4. Aaron Toponce : ZFS Administration, Part I- VDEVs | April 19, 2013 at 4:55 am | Permalink

   […] iSCSI, NFS and Samba […]

5. Aaron Toponce : ZFS Administration, Part III- The ZFS Intent Log | April 19, 2013 at 4:56 am | Permalink

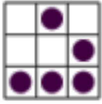
   […] iSCSI, NFS and Samba […]

6. Aaron Toponce : ZFS Administration, Part XII- Snapshots and Clones | April 19, 2013 at 4:58 am | Permalink

   […] iSCSI, NFS and Samba […]

7. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 23, 2013 at 7:45 am | Permalink

   […] iSCSI, NFS and Samba […]



Aaron Toponce

{ 2013.01.02 }

ZFS Administration, Part XVI- Getting and Setting Properties

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

Motivation

Just as with Zpool properties, datasets also contain properties that can be changed. Because datasets are where you actually store your data, there are quite a bit more than with storage pools. Further, properties can be inherited from parent datasets. Again, not every property is tunable. Many are read-only. But, this again gives us the ability to tune our filesystem based on our storage needs. One aspect with ZFS datasets also, is the ability to set your own custom properties. These are known as "user properties" and differ from "native properties".

Because there are just so many properties, I've decided to put the administration "above the fold", and put the properties, along with some final thoughts at the end of the post.

Getting and Setting Properties

As with getting and setting storage pool properties, there are a few ways that you can get at dataset properties as well- you can get all properties at once, only one property, or more than one, comma-separated. For example, suppose I wanted to get just the compression ration of the dataset. I could issue the following command:

```
# zfs get compressratio tank/test
NAME       PROPERTY       VALUE  SOURCE
tank/test  compressratio  1.00x  -
```

If I wanted to get multiple settings, say the amount of disk used by the dataset, as well as how much is available, and the compression ratio, I could issue this command instead:

```
# zfs get used,available,compressratio tank/test
tank/test  used                  1.00G                 -
tank/test  available             975M                  -
tank/test  compressratio         1.00x                 -
```

And of course, if I wanted to get all the settings available, I could run:

```
# zfs get all tank/test
NAME       PROPERTY              VALUE                 SOURCE
tank/test  type                  filesystem            -
tank/test  creation              Tue Jan  1  6:07 2013  -
tank/test  used                  1.00G                 -
tank/test  available             975M                  -
tank/test  referenced            1.00G                 -
tank/test  compressratio         1.00x                 -
tank/test  mounted               yes                   -
tank/test  quota                 none                  default
tank/test  reservation           none                  default
tank/test  recordsize            128K                  default
tank/test  mountpoint            /tank/test            default
tank/test  sharenfs              off                   default
tank/test  checksum              on                    default
tank/test  compression           lzjb                  inherited from tank
tank/test  atime                 on                    default
tank/test  devices               on                    default
tank/test  exec                  on                    default
tank/test  setuid                on                    default
tank/test  readonly              off                   default
tank/test  zoned                 off                   default
tank/test  snapdir               hidden                default
tank/test  aclinherit            restricted            default
tank/test  canmount              on                    default
tank/test  xattr                 on                    default
tank/test  copies                1                     default
tank/test  version               5                     -
tank/test  utf8only              off                   -
tank/test  normalization         none                  -
tank/test  casesensitivity       sensitive             -
tank/test  vscan                 off                   default
tank/test  nbmand                off                   default
tank/test  sharesmb              off                   default
tank/test  refquota              none                  default
tank/test  refreservation        none                  default
tank/test  primarycache          all                   default
tank/test  secondarycache        all                   default
tank/test  usedbysnapshots       0                     -
tank/test  usedbydataset         1.00G                 -
tank/test  usedbychildren        0                     -
tank/test  usedbyrefreservation  0                     -
tank/test  logbias               latency               default
tank/test  dedup                 off                   default
tank/test  mlslabel              none                  default
tank/test  sync                  standard              default
tank/test  refcompressratio      1.00x                 -
tank/test  written               0                     -
```

Inheritance

As you may have noticed in the output above, properties can be inherited from their parents. In that case, I set the compression algorithm to "lzjb" on the storage pool filesystem "tank" ("tank" is more than just a storage pool- it is a valid ZFS dataset). As such, any datasets created under the "tank" dataset will inherit that property. Let's create a nested dataset, and see how this comes into play:

```
# zfs create -o compression=gzip tank/test/one
# zfs get -r compression tank
NAME           PROPERTY     VALUE     SOURCE
tank           compression  lzjb      local
tank/test      compression  lzjb      inherited from tank
tank/test/one  compression  gzip      local
```

Notice that the "tank" and "tank/test" datasets are using the "lzjb" compression algorithm, where "tank/test" inherited it from its parent "tank". Whereas with the "tank/test/one" dataset, we chose a different compression algorithm. Let's now inherit the parent compression algorithm from "tank", and see what happens to "tank/test/one":

```
# zfs inherit compression tank/test/one
# zfs get -r compression tank
NAME           PROPERTY     VALUE     SOURCE
tank           compression  lzjb      local
tank/test      compression  lzjb      inherited from tank
tank/test/one  compression  lzjb      inherited from tank
```

In this case, we made the change from the "gzip" algorithm to the "lzjb" algorithm, by inheriting from its parent. Now, the "zfs inherit" command also supports recursion. I can set the "tank" dataset to be "gzip", and apply the property recursively to all children datasets:

```
# zfs set compression=gzip tank
# zfs inherit -r compression tank/test
# zfs get -r compression tank
NAME           PROPERTY     VALUE     SOURCE
tank           compression  gzip      local
tank/test      compression  gzip      inherited from tank
tank/test/one  compression  gzip      inherited from tank
```

Be very careful when using the "-r" switch. Suppose you quickly typed the command, and gave the "tank" dataset as your argument, rather than "tank/test":

```
# zfs inherit -r compression tank
# zfs get -r compression tank
NAME           PROPERTY     VALUE     SOURCE
tank           compression  off       default
tank/test      compression  off       default
tank/test/one  compression  off       default
```

What happened? All compression algorithms got reset to their defaults of "off". As a result, be very fearful of the "-r" recursive switch with the "zfs inherit" command. As you can see here, this is a way that you can clear dataset properties back to their defaults, and apply it to all children. This applies to datasets, volumes and snapshots.

User Dataset Properties

Now that you understand about inheritance, you can understand setting custom user properties on your datasets. The goal of user properties is for applications designed around ZFS specifically, to take advantage of those settings. For example, poudriere is a tool for FreeBSD designed to test package production, and to build FreeBSD packages in bulk. If using ZFS with FreeBSD, you can create a dataset for poudriere, and then create some custom properties for it to take advantage of.

Custom user dataset properties have no effect on ZFS performance. Think of them merely as "annotation" for administrators and developers. User properties must use a colon ":" in the property name to distinguish them from native dataset properties. They may contain lowercase letters, numbers, the colon ":", dash "-", period "." and underscore "_". They can be at ost 256 characters, and must not begin with a dash "-".

To create a custom property, just use the "module:property" syntax. This is not enforced by ZFS, but is probably the cleanest approach:

```
# zfs set poudriere:type=ports tank/test/one
# zfs set poudriere:name=my_ports_tree tank/test/one
# zfs get all tank/test/one | grep poudriere
tank/test/one  poudriere:name        my_ports_tree          local
tank/test/one  poudriere:type        ports                  local
```

I am not aware of a way to remove user properties from a ZFS filesystem. As such, if it bothers you, and is cluttering up your property list, the only way to remove the user property is to create another dataset with the properties you want, copy over the data, then destroy the old cluttered dataset. Of course, you can inherit user properties with "zfs inherit" as well. And all the standard utilities, such as "zfs set", "zfs get", "zfs list", et cetera will work with user properties.

With that said, let's get to the native properties.

Native ZFS Dataset Properties

- aclinherit: Controls how ACL entries are inherited when files and directories are created. Currently, ACLs are not functioning in ZFS on Linux as of 0.6.0-rc13. Default is "restricted". Valid values for this property are:
  - discard: do not inherit any ACL properties
  - noallow: only inherit ACL entries that specify "deny" permissions
  - restricted: remove the "write_acl" and "write_owner" permissions when the ACL entry is inherited
  - passthrough: inherit all inheritable ACL entries without any modifications made to the ACL entries when they are inherited
  - passthrough-x: has the same meaning as passthrough, except that the owner@, group@, and everyone@ ACEs inherit the execute permission only if the file creation mode also requests the execute bit.
- aclmode: Controls how the ACL is modified using the "chmod" command. The value "groupmask" is default, which reduces user or group permissions. The permissions are reduced, such that they are no greater than the group permission bits, unless it is a user entry that has the same UID as the owner of the file or directory. Valid values are "discard", "groupmask", and "passthrough".
- acltype: Controls whether ACLs are enabled and if so what type of ACL to use. When a file system has the acltype property set to noacl (the default) then ACLs are disabled. Setting the acltype property to posixacl indicates Posix ACLs should be used. Posix ACLs are specific to Linux and are not functional on other platforms. Posix ACLs are stored as an xattr and therefore will not overwrite any existing ZFS/NFSv4 ACLs which may be set. Currently only posixacls are supported on Linux.
- atime: Controls whether or not the access time of files is updated when the file is read. Default is "on". Valid values are "on" and "off".
- available: Read-only property displaying the available space to that dataset and all of its children, assuming no other activity on the pool. Can be referenced by its shortened name "avail". Availability can be limited by a number of factors, including physical space in the storage pool, quotas, reservations and other datasets in the pool.
- canmount: Controls whether the filesystem is able to be mounted when using the "zfs mount" command. Default is "on". Valid values can be "on", "off", or "noauto". When the noauto option is set, a dataset can only be mounted and unmounted explicitly. The dataset is not mounted automatically when the dataset is created or imported, nor is it mounted by the "zfs mount" command or unmounted with the "zfs unmount" command. This property is not inherited.
- casesensitivity: Indicates whether the file name matching algorithm used by the file system should be case-sensitive, case-insensitive, or allow a combination of both styles of matching. Default value is "sensitive". Valid values are "sensitive", "insensitive", and "mixed". Using the "mixed" value would be beneficial in

heterogenous environments where Unix POSIX and CIFS filenames are deployed. Can only be set during dataset creation.

- checksum: Controls the checksum used to verify data integrity. The default value is "on", which automatically selects an appropriate algorithm. Currently, that algorithm is "fletcher2". Valid values is "on", "off", "fletcher2", "fletcher4", or "sha256". Changing this property will only affect newly written data, and will not apply retroactively.
- clones: Read-only property for snapshot datasets. Displays in a comma-separated list datasets which are clones of this snapshot. If this property is not empty, then this snapshot cannot be destroyed (not even with the "-r" or "-f" options). Destroy the clone first.
- compression: Controls the compression algorithm for this dataset. Default is "off". Valid values are "on", "off", "lzjb", "gzip", "gzip-N", and "zle". The "lzjb" algorithm is optimized for speed, while provide good compression ratios. The setting of "on" defaults to "lzjb". It is recommended that you use "lzjb", "gzip", "gzip-N", or "zle" rather than "on", as the ZFS developers or package maintainers may change the algorithm "on" uses. The gzip compression algorithm uses the same compression as the "gzip" command. You can specify the gzip level by using "gzip-N" where "N" is a valid number of 1 through 9. "zle" compresses runs of binary zeroes, and is very fast. Changing this property will only affect newly written data, and will not apply retroactively.
- compressratio: Read-only property that displays the compression ratio achieved by the compression algorithm set on the "compression" property. Expressed as a multiplier. Does not take into account snapshots; see "refcompressratio". Compression is not enabled by default.
- copies: Controls the number of copies to store in this dataset. Default value is "1". Valid values are "1", "2", and "3". These copies are in addition to any redundancy provided by the pool. The copies are stored on different disks, if possible. The space used by multiple copies is charged to the associated file and dataset. Changing this property only affects newly written data, and does not apply retroactively.
- creation: Read-only property that displays the time the dataset was created.
- defer_destroy: Read-only property for snapshots. This property is "on" if the snapshot has been marked for deferred destruction by using the "zfs destroy -d" command. Otherwise, the property is "off".
- dedup: Controls whether or not data deduplication is in effect for this dataset. Default is "off". Valid values are "off", "on", "verify", and "sha256[,verify]". The default checksum used for deduplication is SHA256, which is subject to change. When the "dedup" property is enabled, it overrides the "checksum" property. If the property is set to "verify", then if two blocks have the same checksum, ZFS will do a byte-by-byte comparison with the existing block to ensure the blocks are identical. Changing this property only affects newly written data, and is not applied retroactively. Enabling deduplication in the dataset will dedupe data in that dataset against all data in the storage pool. Disabling this property does not destroy the deduplication table. Data will continue to remain deduped.
- devices: Controls whether device nodes can be opened on this file system. The default value is "on". Valid values are "on" and "off".
- exec: Controls whether processes can be executed from within this file system. The default value is "on". Valid values are "on" and "off".
- groupquota@<group>: Limits the amount of space consumed by the specified group. Group space consumption is identified by the "userquota@<user>" property. Default value is "none". Valid values are "none", and a size in bytes.
- groupsused@<group>: Read-only property displaying the amount of space consumed by the specified group in this dataset. Space is charged to the group of each file, as displayed by "ls -l". See the userused@<user> property for more information.
- logbias: Controls how to use the SLOG, if one exists. Provides a hint to ZFS on how to handle synchronous requests. Default value is "latency", which will use a SLOG in the pool if present. The other valid value is "throughput" which will not use the SLOG on synchronous requests, and go straight to platter disk.
- mlslabel: The mlslabel property is a sensitivity label that determines if a dataset can be mounted in a zone on a system with Trusted Extensions enabled. Default value is "none". Valid values are a Solaris Zones label or "none". Note, Zones are a Solaris feature, and not relevant to GNU/Linux. However, this may be something that could be implemented with SELinux an Linux containers in the future.

- mounted: Read-only property that indicates whether the dataset is mounted. This property will display either "yes" or "no".
- mountpoint: Controls the mount point used for this file system. Default value is "<pool>/<dataset>". Valid values are an absolute path on the filesystem, "none", or "legacy". When the "mountpoint" property is changed, the new destination must not contain any child files. The dataset will be unmounted and re-mounted to the new destination.
- nbmand: Controls whether the file system should be mounted with non-blocking mandatory locks. This is used for CIFS clients. Default value is "on". Valid values are "on" and "off". Changing the property will only take effect after the dataset has ben unmounted then re-mounted.
- normalization: Indicates whether the file system should perform a unicode normalization of file names whenever two file names are compared and which normalization algorithm should be used. Default value is "none". Valid values are "formC", "formD", "formKC", and "formKD". This property cannot be changed after the dataset is created.
- origin: Read-only property for clones or volumes, which displays the snapshot from whence the clone was created.
- primarycache: Controls what is cached in the primary cache (ARC). If this property is set to "all", then both user data and metadata is cached. If set to "none", then neither are cached. If set to "metadata", then only metadata is cached. Default is "all".
- quota: Limits the amount of space a dataset and its descendents can consume. This property enforces a hard limit on the amount of space used. There is no soft limit. This includes all space consumed by descendents, including file systems and snapshots. Setting a quota on a descendant of a dataset that already has a quota does not override the ancestor's quota, but rather imposes an additional limit. Quotas cannot be set on volumes, as the volsize property acts as an implicit quota. Default value is "none" Valid values are a size in bytes or "none".
- readonly: Controls whether this dataset can be modified. The default value is off. Valid values are "on" and "off". This property can also be referred to by its shortened column name, "rdonly".
- recordsize: Specifies a suggested block size for files in the file system. This property is designed solely for use with database workloads that access files in fixed-size records. ZFS automatically tunes block sizes according to internal algorithms optimized for typical access patterns. The size specified must be a power of two greater than or equal to 512 and less than or equal to 128 KB. Changing the file system's recordsize affects only files created afterward; existing files are unaffected. This property can also be referred to by its shortened column name, "recsize".
- refcompressratio: Read only property displaying the compression ratio achieved by the space occupied in the "referenced" property.
- referenced: Read-only property displaying the amount of data that the dataset can access. Initially, this will be the same number as the "used" property. As snapshots are created, and data is modified however, those numbers will diverge. This property can be reference by its shortened name "refer".
- refquota: Limits the amount of space a dataset can consume. This property enforces a hard limit on the amount of space used. This hard limit does not include space used by descendents, including file systems and snapshots. Default value is "none". Valid values are "none", and a size in bytes.
- refreservation: The minimum amount of space guaranteed to a dataset, not including its descendents. When the amount of space used is below this value, the dataset is treated as if it were taking up the amount of space specified by refreservation. Default value is "none". Valid values are "none" and a size in bytes. This property can also be referred to by its shortened column name, "refreserv".
- reservation: The minimum amount of space guaranteed to a dataset and its descendents. When the amount of space used is below this value, the dataset is treated as if it were taking up the amount of space specified by its reservation. Reservations are accounted for in the parent datasets' space used, and count against the parent datasets' quotas and reservations. This property can also be referred to by its shortened column name, reserv. Default value is "none". Valid values are "none" and a size in bytes.
- secondarycache: Controls what is cached in the secondary cache (L2ARC). If this property is set to "all", then both user data and metadata is cached. If this property is set to "none", then neither user data nor metadata is cached. If this property is set to "metadata", then only metadata is cached. The default value is "all".

- setuid: Controls whether the set-UID bit is respected for the file system. The default value is on. Valid values are "on" and "off".
- shareiscsi: Indicates whether a ZFS volume is exported as an iSCSI target. Currently, this is not implemented in ZFS on Linux, but is pending. Valid values will be "on", "off", and "type=disk". Other disk types may also be supported. Default value will be "off".
- sharenfs: Indicates whether a ZFS dataset is exported as an NFS export, and what options are used. Default value is "off". Valid values are "on", "off", and a list of valid NFS export options. If set to "on", the export can then be shared with the "zfs share" command, and unshared with the "zfs unshare" command. An NFS daemon must be running on the host before the export can be used. Debian and Ubuntu require a valid export in the /etc/exports file before the daemon will start.
- sharesmb: Indicates whether a ZFS dataset is export as a SMB share. Default value is "off". Valid values are "on" and "off". Currently, a bug exists preventing this from being used. When fixed, it will require a running Samba daemon, just like with NFS, and will be shared and unshared with the "zfs share" and "zfs unshare" commands.
- snapdir: Controls whether the ".zfs" directory is hidden or visible in the root of the file system. Default value is "hidden". Valid values are "hidden" and "visible". Even though the "hidden" value might be set, it is still possible to change directories into the ".zfs" directory, to access the shares and snapshots.
- sync: Controls the behavior of synchronous requests (e.g. fsync, O_DSYNC). Default value is "default", which is POSIX behavior to ensure all synchronous requests are written to stable storage and all devices are flushed to ensure data is not cached by device controllers. Valid values are "default", "always", and "disabled". The value of "always" causes every file system transaction to be written and flushed before its system call returns. The value of "disabled" does not honor synchronous requests, which will give the highest performance.
- type: Read-only property that displays the type of filesystem, whether it be a "dataset", "volume" or "snapshot".
- used: Read-only property that displays the amount of space consumed by this dataset and all its children. When snapshots are created, the space is initially shared between the parent dataset and its snapshot. As data is modified in the dataset, space that was previously shared becomes unique to the snapshot, and is only counted in the "used" property for that snapshot. Further, deleting snapshots can free up space unique to other snapshots.
- usedbychildren: Read-only property that displays the amount of space used by children of this dataset, which is freed if all of the children are destroyed.
- usedbydataset: Read-only property that displays the amount of space used by this dataset itself., which would then be freed if this dataset is destroyed.
- usedbyrefreservation: Read-only property that displays the amount of space used by a refreservation set on this dataset, which would be freed if the refreservation was removed.
- usedbysnapshots: Read-only property that displays the amount of space consumed by snapshots of this dataset. In other words, this is the data that is unique to the snapshots. Note, this is not a sum of each snapshot's "used" property, as data can be shared across snapshots.
- userquota@<user>: Limits the amount of space consumed by the specified user. Similar to the "refquota" property, the userquota space calculation does not include space that is used by descendent datasets, such as snapshots and clones. Enforcement of user quotas may be delayed by several seconds. This delay means that a user might exceed their quota before the system notices that they are over quota and begins to refuse additional writes with the EDQUOT error message. This property is not available on volumes, on file systems before version 4, or on pools before version 15. Default value is "none". Valid values are "none" and a size in bytes.
- userrefs: Read-only property on snapshots that displays the number of user holds on this snapshot. User holds are set by using the zfs hold command.
- userused@<user>: Read-only property that displays the amount of space consumed by the specified user in this dataset. Space is charged to the owner of each file, as displayed by "ls -l". The amount of space charged is displayed by du and ls -s. See the zfs userspace subcommand for more information. The "userused@<user>" properties are not displayed with "zfs get all". The user's name must be appended after the @ symbol, using one of the following forms:
    - POSIX name (for example, joe)

- POSIX numeric ID (for example, 789)
  - SID name (for example, joe.smith@mydomain)
  - SID numeric ID (for example, S-1-123-456-789)
- utf8only: Indicates whether the file system should reject file names that include characters that are not present in the UTF-8 character set. Default value is "off". Valid values are "on" and "off". This property cannot be changed after the dataset has been created.
- version: The on-disk version of this file system, which is independent of the pool version. This property can only be set to later supported versions. Valid values are "current", "1", "2", "3", "4", or "5".
- volblocksize: Read-only property for volumes that specifies the block size of the volume. The blocksize cannot be changed once the volume has been written, so it should be set at volume creation time. The default blocksize for volumes is 8 KB. Any power of 2 from 512 bytes to 128 KB is valid.
- vscan: Controls whether regular files should be scanned for viruses when a file is opened and closed. In addition to enabling this property, the virus scan service must also be enabled for virus scanning to occur. The default value is "off". Valid values are "on" and "off".
- written: Read-only property that displays the amount of referenced space written to this dataset since the previous snapshot.
- written@<snapshot>: Read-only property on a snapshot that displays the amount of referenced space written to this dataset since the specified snapshot. This is the space that is referenced by this dataset but was not referenced by the specified snapshot.
- xattr: Controls whether extended attributes are enabled for this file system. The default value is "on". Valid values are "on" and "off".
- zoned: Controls whether the dataset is managed from a non-global zone. Zones are a Solaris feature and are not relevant on Linux. Default value is "off". Valid values are "on" and "off".

Final Thoughts

As you have probably noticed, some ZFS dataset properties are not fully implemented with ZFS on Linux, such as sharing a volume via iSCSI. Other dataset properties apply to the whole pool, such as the case with deduplication, even though they are applied to specific datasets. Many properties only apply to newly written data, and are not retroactive. As such, be aware of each property, and the pros/cons of what it provides. Because the parent storage pool is also a valid ZFS dataset, any child datasets will inherit non-default properties, as seen. And, the same is true for nested datasets, snapshots and volumes.

With ZFS dataset properties, you now have all the tuning at your fingertips to setup a solid ZFS storage backend. And everything has been handled with the "zfs" command, and its necessary subcommands. In fact, up to this point, we've only learned two commands: "zpool" and "zfs", yet we've been able to build and configure powerful, large, redundant, consistent, fast and tuned ZFS filesystems. This is unprecedented in the storage world, especially with GNU/Linux. The only thing left to discuss is some best practices and caveats, and then a brief post on the "zdb" command (which you should never need), and we'll be done with this series. Hell, if you've made it this far, I commend you. This has been no small series (believe me, my fingers hate me).

Posted by Aaron Toponce on Wednesday, January 2, 2013, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 15 } Comments

1. Sim | January 2, 2013 at 9:51 am | Permalink

   A really excellent series of articles!

2. Michael | March 20, 2013 at 1:56 pm | Permalink

   Very nice. Just finished reading to the end. I didn't see the zdb post you mentioned at the end of this post (XVI). If you don't still plan to add a post on zdb at the end maybe delete the reference to zdb from the end of XVI.

   Thanks again !

3. Aaron Toponce | March 20, 2013 at 2:37 pm | Permalink

   Yeah. I decided against zdb(8) last minute. Sorry about that.

4. Anonymous | April 3, 2014 at 3:59 pm | Permalink

   zfs inherit seemed to work to get rid of user properties:

   root@fs01# zfs set jim:test="testing user properties" tank/stor/jim
   root@fs01# zfs get jim:test tank/stor/jim
   NAME PROPERTY VALUE SOURCE
   tank/stor/jim jim:test testing user properties local
   root@fs01# zfs inherit jim:test tank/stor/jim
   root@fs01# zfs get jim:test tank/stor/jim
   NAME PROPERTY VALUE SOURCE
   tank/stor/jim jim:test - -

5. John Naggets | January 17, 2015 at 11:09 am | Permalink

   What about xattr? I read it should be set to xattr=sa but I can't find any mention of "sa" option in your documentation

6. NM | February 5, 2015 at 3:28 pm | Permalink

   So what's the difference between a VDEV with quotas and a ZVOL?
   And how to minimize fragmentation?

7. John Naggets | February 6, 2015 at 8:47 am | Permalink

   By setting sync=disabled on a ZFS volume, does this mean that there is no ZIL nor any L2ARC, is my interpretation correct here?

8. Aaron Toponce | February 17, 2015 at 1:32 pm | Permalink

   > By setting sync=disabled on a ZFS volume, does this mean that there is no ZIL nor any L2ARC, is my interpretation correct here?

   By setting "sync=disabled", the ZIL is completely bypassed, and data is written directly to spinning platter. If the ZIL resides on a SLOG, then the SLOG will also be equally bypassed. However, if you have an L2ARC (a cache), writes aren't taken into account here, except to evict pages from RAM to the L2ARC. The L2ARC (should be) is read-heavy. Setting "sync=disabled" has no effect here.

9. Aaron Toponce | February 17, 2015 at 1:34 pm | Permalink

> So what's the difference between a VDEV with quotas and a ZVOL? And how to minimize fragmentation?

I think you mean a "dataset with quotas" rather than a VDEV. A dataset with quotas just has an upper limit on how much storage can be set. A ZVOL actually creates a block device that can be partitioned, formatted, etc. in /dev/ just like any other block device.

10. John Naggets | March 6, 2015 at 4:10 am | Permalink

    Just noticed that you are missing the "acltype" property in your documentation. You might want to add it, as for example I need to change it to posixacl in order to use GlusterFS on top of ZFS.

11. Aaron Toponce | March 6, 2015 at 9:42 am | Permalink

    Looks like some properties have been added since I've created this post. I'll update. Thanks for catching that!

12. Wade | September 5, 2015 at 12:23 pm | Permalink

    Dam, my studies of zfs is getting very very complicated. I think I've seen anywhere from 11-15 different dataset size/capacity related properties, depending on the quality of the documentation provided for any given implementation. FYI logicalused and a number of other properties are missing above.

    Here's a list I thought was comprehensive but now I know is not:
    available
    quota
    referenced
    refquota
    refreservation
    reservation
    used
    usedbychildren
    usedbydataset
    usedbyrefreservation
    usedbysnapshots
    volsize

    Firstly, let me explain my task. I need at least one simple bar graph for each ZFS dataset, however, what the bar graph should represent I think is completely different depending on dataset type and a number of properties. Like if you have a volsize than that should be the 100% mark, the quota could be shown at the appropriate mark and then used space could show the utilization in this single diagram. There are just too many decisions to be made based off these unnormalized variables. It's really difficult and I need some input.

    One, big decision. When I show this bar graph for each dataset, do I use the quantities that include the children... and where is a comprehensive definition of child? Can a snapshot have a child? I think I need some drawings, to understand this.

    My instinct tells me every single size based variable needs it's place as a bar graph marker point, total or % fill... but man it's proving very difficult to determine which variables to show when and how I can use the variables to actually calculate anything. Like, on most normal file systems Freespace = Total space - minus used space. But on ZFS it's like these calculations are so incredibly complicated they provide each discrete variable you may need, but if the variables were normalized it would be a lot easier! Can anyone help? I'd like to see a list of quick sketches, for each dataset type, of all the bar graph possibilities and/or

recommendations depending on all these incredibly complicated factors. Or am I complicating it too much?

Thanks if you got this far! lol

13. Anonymous | February 6, 2016 at 2:55 pm | Permalink

    You can clear user defined properties using "zfs inherit", as described in zfs man-page:
    "Use the zfs inherit command to clear a user property . If the property is not defined in any parent dataset, it is removed entirely."

14. Kal | November 22, 2016 at 7:56 pm | Permalink

    I have a few questions regarding the sharenfs property and the zfs share/unshare commands.

    Under the explanation for the sharenfs property, you write:
    Debian and Ubuntu require a valid export in the /etc/exports file before the daemon will start.

    Is that still true?

    What's considered *valid*? For instance, if I have a dataset tank/foo, do I need to create an export rule for /tank/foo in /etc/exports, with export options matching the sharenfs property (rw, no_root_squash, etc)?

    What happens if the export options don't match between /etc/exports and the sharenfs property?

    What's so special about ZFS that they included the sharenfs property and share/unshare commands? Why not let people manage the sharing of the dataset just like any other file system?

15. Mike Holden | April 5, 2017 at 9:35 pm | Permalink

    Hi Aaron, love these references. Just starting with zfs, and this is the best guide by a mile!

    Just to note that the groupquota item in the list above references userquota in the notes. Copy&paste error no doubt!

{ 6 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | January 14, 2013 at 9:16 am | Permalink

    [...] Getting and Setting Properties [...]

2. Aaron Toponce : ZFS Administration, Part V- Exporting and Importing zpools | April 19, 2013 at 4:56 am | Permalink

    [...] Getting and Setting Properties [...]

3. Aaron Toponce : ZFS Administration, Part XVII- Best Practices and Caveats | April 19, 2013 at 5:00 am | Permalink

    [...] Getting and Setting Properties [...]

4. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:03 am | Permalink

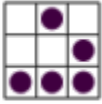
    [...] Getting and Setting Properties [...]

5. Aaron Toponce : Install ZFS on Debian GNU/Linux | May 5, 2013 at 6:02 pm | Permalink

   [...] Getting and Setting Properties [...]

6. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | July 8, 2013 at 10:08 pm | Permalink

   […] Getting and Setting Properties […]



Aaron Toponce

{ 2013.01.03 }

ZFS Administration, Part XVII- Best Practices and Caveats

Table of Contents

Zpool Administration	ZFS Administration	Appendices
0. Install ZFS on Debian GNU/Linux	9. Copy-on-write	A. Visualizing The ZFS Intent Log (ZIL)
1. VDEVs	10. Creating Filesystems	B. Using USB Drives
2. RAIDZ	11. Compression and Deduplication	C. Why You Should Use ECC RAM
3. The ZFS Intent Log (ZIL)	12. Snapshots and Clones	D. The True Cost Of Deduplication
4. The Adjustable Replacement Cache (ARC)	13. Sending and Receiving Filesystems	
5. Exporting and Importing Storage Pools	14. ZVOLs	
6. Scrub and Resilver	15. iSCSI, NFS and Samba	
7. Getting and Setting Properties	16. Getting and Setting Properties	
8. Best Practices and Caveats	17. Best Practices and Caveats	

Best Practices

As with all recommendations, some of these guidelines carry a great amount of weight, while others might not. You may not even be able to follow them as rigidly as you would like. Regardless, you should be aware of them. I'll try to provide a reason why for each. They're listed in no specific order. The idea of "best practices" is to optimize space efficiency, performance and ensure maximum data integrity.

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

- Consider using "zfs send" over "rsync", as the "zfs send" command can preserve dataset properties.

Caveats

The point of the caveat list is by no means to discourage you from using ZFS. Instead, as a storage administrator planning out your ZFS storage server, these are things that you should be aware of, so as not to catch you with your pants down, and without your data. If you don't head these warnings, you could end up with corrupted data. The line may be blurred with the "best practices" list above. I've tried making this list all about data corruption if not headed. Read and head the caveats, and you should be good.

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

Posted by Aaron Toponce on Thursday, January 3, 2013, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 14 } Comments

1. Asif | January 8, 2013 at 12:09 am | Permalink

   Awesome series! You just helped me to learn and plan deployment of ZFS for my Home NAS in a day. I have gone through the whole series and it's been easy to follow while also providing details on necessary parts! Thank you!

2. aasche | January 30, 2013 at 3:57 pm | Permalink

   18 Parts - enough stuff for a small book. Thank you very much for your efforts 🙂

3. ovigia | February 25, 2013 at 12:27 pm | Permalink

   great tips...

thank you very much!

4. Michael | March 20, 2013 at 2:03 pm | Permalink

   Nice series !!!

   I looked and my rhel6/oel6 box doesn't have a "/etc/modprobe.d/zfs.conf " file anywhere. Is that something you added & just put that one command in (options zfs zfs_arc_max=2147483648)?

   I was also curious how you came up with 2GB as your limit & how much RAM your storage box has and whether you are using the box for anything else?

   My box is currently dedicated to just ZFS & currently has 16GB and I was considering expanding to 32GB. If that scenario any idea what a good arc max is?

   Thanks again !!!

5. Aaron Toponce | March 20, 2013 at 2:39 pm | Permalink

   Yeah. the /etc/modprobe.d/zfs.conf is manually created. The 2 GB is just an example. It's up to how much RAM you have in your system. You should keep it under 1/4 your RAM size, IMO.

6. Mike | April 9, 2013 at 10:48 am | Permalink

   Just want to add my thanks for a great series and all the obvious effort that went into it. While I have enough desktop experience, I am a complete newbie to servers in general and ZFS in particular. You've given me the confidence to proceed.

7. Graham Perrin | December 2, 2013 at 11:48 am | Permalink

   Please: is the ' zfs-auto-snapshot script' link correct? Unless I'm missing something, it doesn't lead to the script.

8. Aaron Toponce | December 4, 2013 at 4:23 pm | Permalink

   Fixed. Sorry about that. I don't know what caused it to change.

9. Joshua Zitting | January 7, 2014 at 9:32 pm | Permalink

   This is an AWESOME Tutorial!!! I read every word and added it to bookmarks for safe keeping! Great work!!! My next project is Postgres... You havent done a Tutorial on it have you?? if so you should start charging!

10. Scott Burson | August 7, 2014 at 10:28 am | Permalink

    I think you're understating the importance of regular snapshots. They're not just a good idea; they're a critical part of ZFS's redundancy strategy. Data loss with ZFS is rare, but it can be caused by faulty hardware. Regular snapshots significantly decrease the likelihood of data loss in such cases. This is especially important for home and small business users, who tend to back up irregularly at best.

11. Paul | October 16, 2014 at 5:55 pm | Permalink

    Thank you very much for this informative and well-written series. I used Freenas for ~2 years before deciding to re-roll my home NAS using Linux, but I knew I wanted to use ZFS. I have learned more about ZFS from reading this guide than I did from using it over the past 2 years.

Some confusion remains regarding Freenas's ZFS implementation in that it used 2Gb from each drive as "swap". Do I still need to do this and if so, could I just use a small (8-10Gb) partition on a separate platter disk instead of multiple partitions across all drives in the zpool? Would I assign this to the zpool as L2ARC or something else?

Thank you!

12. John | February 17, 2015 at 10:47 am | Permalink

    I'm new to ZFS so your series is awesome! I still have a couple questions though if you have time:

    I want to use ZFS for a home NAS, I currently use Unraid (visualized) but I'm getting hit with silent corruption.

    I have 15 2TB disks (media/pc backups/ office files) and a 240GB SSD (running ESXi with VMs (no redundancy - just backed up to unraid)

    I like unraid as it will spin down disks when not in use. I have primarily media that gets accessed maybe a couple times a day in the evening, so my disks are down most of the day.

    I'm thinking of moving to an all linux based server (ditch esxi) with all content in zfs (inc the VMs - I'll move to host them in kvm/xen). I would expect to use the SSD as a ZIL/ARC drive.

    Topics...

    Spin down:
    1) Does zfs ever spin down disks?
    2) If yes, at what level is it managed? pool, dataset, vol, vdev?
    3) Therefore, is there a way to arrange the configuration to ensure those holding media spin down when no demand for them?

    VM:
    4) What performance will I expect to get from my VM's when moving from SSD to ZFS (spinners for content, SSD for ZIL/ARC) - e.g. will it be a noticeable degradation or will the SSD ZIL/ARC mask the slower performance of the spinners? I'll take 'a' hit if I have to, as long as these busy VM's don't grind to a halt!

    Pool config:
    5) depending on the spin down, I had been considering either multiple raidz1 or z2 vdevs, but as I understand it any data written to a dataset or zvol in the pool will spread across all vdevs in a pool for performance - What happens if I lose a vdev (i.e never to return)? do I lose the whole pool or just the vdev
    6) Any suggestions on how to config my 15 2TB disks taking into account the spin down question with 'always on VMs' and 'nearly always off media'? I had considered 2 x raidz1 (for media, PC backups etc.) 1 x raidz2 for VMs and Important office docs but if I can't separate data in a pool is there any benefit!? Would I need multiple pools to achieve this?
    7) you mention a revo drive presenting as two disks/partitions and then striping them for the ARC, any benefit in creating two partitions on my regular SATA SSD for the arc stripe?

    Prioritization:
    8) If I use zVOLS for each VM, can I assign any level of IO weighting in ZFS to ensure my high priority VM gets first dibs at ZFS access? Or is this negated by using SSD ZIL/ARC?

    Thank you in advance!!
    John

13. Aaron Toponce | February 17, 2015 at 11:44 am | Permalink

To answer your questions directly:

> 1) Does zfs ever spin down disks?

Not to my knowledge. ZFS was designed with datacenter storage in mind, and not home use.

> 3) Therefore, is there a way to arrange the configuration to ensure those holding media spin down when no demand for them?

From just brief testing on my workstation, trying to spin down my 250GB drives, ZFS immediately spins them right back up. Also, doing a quick search for spinning down ZFS disks shows up this posts on the FreeBSD forums: https://forums.freebsd.org/threads/spinning-disks-down-with-zfs.23973/#post-135480. So, I don't know if it's FreeBSD specific, and not working with Linux as a result, or something else. Long story short, I just don't know.

> 4) What performance will I expect to get from my VM's when moving from SSD to ZFS (spinners for content, SSD for ZIL/ARC) - e.g. will it be a noticeable degradation or will the SSD ZIL/ARC mask the slower performance of the spinners? I'll take 'a' hit if I have to, as long as these busy VM's don't grind to a halt!

Well, if you are migrating from SSDs to spinning rust, then you will most definitely notice a degredation in throughput. If the SLOG is fast disk, such as modern SSDs, and it is also partitioned for the L2ARC, you will notice that read/write access latencies are the same, but raw read/write throughput is victim to the throughput of the spinning rust. I don't know anything about your VM architecture, but I wouldn't expect your "VM's [to] grind to a halt". I am running two KVM hypervisors sharing disk via GlusterFS for live migrations. The underlying ZFS pool consists for 4x1TB drives in a RAID-1+0 with OCZ SSDs for the L2ARC. I am not using my SSDs for a SLOG, but instead, writing data ascyncronously (because GlusterFS is already fully synchronous). While performance isn't "AMAZING", it's satisfactory. This blog is running in one of the VMs in this 2-node cluster.

> 5) depending on the spin down, I had been considering either multiple raidz1 or z2 vdevs, but as I understand it any data written to a dataset or zvol in the pool will spread across all vdevs in a pool for performance - What happens if I lose a vdev (i.e never to return)? do I lose the whole pool or just the vdev?

Unless you know you need the space, I would advise against parity based RAIDZ. Parity RAID is always slower than RAID-1+0. In terms of performance, RAID-1+0 > RAIDZ1 > RAIDZ2 > RAIDZ3. It's expensive, but I would highly recommend it. However, if you have enough disks, you could probably get away with a RAIDZ1+0 or a RAIDZ2+0, to help keep performance up. Regardless, to answer your question, if you lose a disk in a redundant pool, the pool will continue operating, although in "degraded" mode. Whether or not you can suffer another disk failure is entirely dependent on the RAID array.

> 6) Any suggestions on how to config my 15 2TB disks taking into account the spin down question with 'always on VMs' and 'nearly always off media'? I had considered 2 x raidz1 (for media, PC backups etc.) 1 x raidz2 for VMs and Important office docs but if I can't separate data in a pool is there any benefit!? Would I need multiple pools to achieve this?

For 15 disks, I would recommend setting up 5xRAIDZ1 VDEVs of 3 disks eeach. IE:

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

        NAME                 STATE     READ WRITE CKSUM
        pthree               ONLINE       0     0     0
```

```
    raidz1-0       ONLINE       0     0     0
      /tmp/file1   ONLINE       0     0     0
      /tmp/file2   ONLINE       0     0     0
      /tmp/file3   ONLINE       0     0     0
    raidz1-1       ONLINE       0     0     0
      /tmp/file4   ONLINE       0     0     0
      /tmp/file5   ONLINE       0     0     0
      /tmp/file6   ONLINE       0     0     0
    raidz1-2       ONLINE       0     0     0
      /tmp/file7   ONLINE       0     0     0
      /tmp/file8   ONLINE       0     0     0
      /tmp/file9   ONLINE       0     0     0
    raidz1-3       ONLINE       0     0     0
      /tmp/file10  ONLINE       0     0     0
      /tmp/file11  ONLINE       0     0     0
      /tmp/file12  ONLINE       0     0     0
    raidz1-4       ONLINE       0     0     0
      /tmp/file13  ONLINE       0     0     0
      /tmp/file14  ONLINE       0     0     0
      /tmp/file15  ONLINE       0     0     0

errors: No known data errors
```

You lose 5 disks of storage space (one-third of the raw size), but then you have 5xRAIDZ VDEVs striped. This will help keep performance up, at minimal cost, and you could suffer a drive failure in each VDEV (5 failures total), and still have an operational pool.

> 7) you mention a revo drive presenting as two disks/partitions and then striping them for the ARC, any benefit in creating two partitions on my regular SATA SSD for the arc stripe?

Striping across a singe disk will be limited to the throughput of the SATA controller connected to the drive. It will certainly improve performance, but only up to what the controller can sustain as the upper bottlneck. Also, I should modify this post. When adding drives to the cache (L2ARC), they aren't actually "striped" in the true sense of the word. The drives are balanced evenly, of course, but there actually isn't any striping going on. IE- the pages aren't split into multiple chunks, and placed on each drive in the cache.

> 8) If I use zVOLS for each VM, can I assign any level of IO weighting in ZFS to ensure my high priority VM gets first dibs at ZFS access? Or is this negated by using SSD ZIL/ARC?

I'm not aware of any such prioritization for ZVOLs. If it exists, I'll certainly add it to the series, but I'm not aware of it.

14. John | February 18, 2015 at 2:23 am | Permalink

    Aaron - a big thanks for the speedy response, some food for thought!

{ 4 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part II- RAIDZ | January 7, 2013 at 9:27 pm | Permalink

   [...] Best Practices and Caveats [...]

2. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | March 20, 2013 at 12:38 pm | Permalink

   [...] Best Practices and Caveats [...]

3. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | April 19, 2013 at 4:57 am | Permalink

[...] Best Practices and Caveats [...]

4. Aaron Toponce : ZFS Administration, Part XIV- ZVOLS | April 19, 2013 at 4:59 am | Permalink

   [...] Best Practices and Caveats [...]