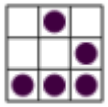

Aaron Toponce

{ 2012 12 18 }

ZFS Administration, Part XI- Compression and Deduplication

Table of Contents

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