Tuning OpenZFS

ALLAN JUDE AND MICHAEL W. LUCAS

Allan Jude is VP of Operations at ScaleEngine Inc., a global HTTP and Video Streaming CDN, where he makes extensive use of ZFS on FreeBSD. He is also the host of the video podcasts BSDNow.tv (with Kris Moore) and TechSNAP.tv. He is a FreeBSD src and doc committer, and was elected to the FreeBSD Core team in the summer of 2016. allanjude@freebsd.org

Michael W. Lucas has used UNIX since the late '80s and started his sysadmin career in 1995. He's the author of over 20 technology books, including *Absolute OpenBSD, PAM Mastery,* and *SSH Mastery.* Lucas lives with his wife in Detroit, Michigan, has pet rats, and practices martial arts. mwlucas@michaelwlucas.com

OpenZFS is such a powerful file system that it has found its way into illumos, Linux, FreeBSD, and other operating systems. Its flexibility requires whole new ways of thinking, however. If you're using OpenZFS for a special purpose, such as database storage or retaining particular sizes of files, you'll want to tune the file system for those purposes.

This article uses FreeBSD as a reference platform, as it's one of the biggest OpenZFS consumers. You will need to change paths and such for other operating systems, but all the ZFS information is consistent across platforms.

Recordsize

While many ZFS properties impact performance, start with `recordsize`.

The `recordsize` property gives the maximum size of a logical block in a ZFS dataset. The default `recordsize` is 128 KB, which comes to 32 sectors on a disk with 4 KB sectors, or 256 sectors on a disk with 512-byte sectors. The maximum `recordsize` was increased to 1 MB with the introduction of the `large_blocks` feature flag in 2015. Many database engines prefer smaller blocks, such as 4 KB or 8 KB. It makes sense to change the `recordsize` on datasets dedicated to such files. Even if you don't change the `recordsize`, ZFS automatically sizes records as needed. Writing a 16 KB file should take up only 16 KB of space (plus metadata and redundancy space), not waste an entire 128 KB record.

The most important tuning you can perform for an application is the dataset block size. If an application consistently writes blocks of a certain size, `recordsize` should match the block size used by the application. This becomes really important with databases.

Databases and ZFS

Many ZFS features are highly advantageous for databases. Every DBA wants fast, easy, and efficient replication, snapshots, clones, tunable caches, and pooled storage. While ZFS is designed as a general-purpose file system, you can tune it to make your databases fly.

Databases usually consist of more than one type of file, and since each has different characteristics and usage patterns, each requires different tuning. We'll discuss MySQL and PostgreSQL in particular, but the principles apply to any database software.

Tuning the block size avoids write amplification. Write amplification happens when changing a small amount of data requires writing a large amount of data. Suppose you must change 8 KB in the middle of a 128 KB block. ZFS must read the 128 KB, modify 8 KB somewhere in it, calculate a new checksum, and write the new 128 KB block. ZFS is a copy-on-write file system, so it would wind up writing a whole new 128 KB block just to change that 8 KB. You don't want that. Now multiply this by the number of writes your database makes. Write amplification eviscerates performance.

Low-load databases might not need this sort of optimization, but on a high-performance system it is invaluable. Write amplification reduces the life of SSDs and other flash-based storage that can handle a limited volume of writes over their lifetime.

The different database engines don't make `recordsize` tuning easy. Each database server has different needs. Journals, binary replication logs, error and query logs, and other miscellaneous files also require different tuning.

Before creating a dataset with a small `recordsize`, be sure you understand the interaction between VDEV type and space utilization. In some situations, disks with the smaller 512-byte sector size can provide better storage efficiency. It is entirely possible you may be better off with a separate pool specifically for your database, with the main pool for your other files.

For high-performance systems, use mirrors rather than any type of RAID-Z. Yes, for resiliency you probably want RAID-Z. Hard choices are what makes system administration fun!

All Databases

Enabling lz4 compression on a database can, unintuitively, decrease latency. Compressed data can be read more quickly from the physical media, as there is less to read, which can result in shorter transfer times. With lz4's early abort feature, the worst case is only a few milliseconds slower than opting out of compression, but the benefits are usually quite significant. This is why ZFS uses lz4 compression for all of its own metadata and for the L2ARC (level 2 adaptive replacement cache).

The Compressed ARC feature recently landed in OpenZFS and is slowly trickling out to OpenZFS consumers. Enabling cache compression on the dataset allows more data to be kept in the ARC, the fastest ZFS cache. In a production case study done by Delphix, a database server with 768 GB of RAM went from using more than 90 percent of its memory to cache a database to using only 446 GB to cache 1.2 TB of compressed data. Compressing the in-memory cache resulted in a significant performance improvement. As the machine could not support any more RAM, compression was the only way to improve. When your operating system gets compressed ARC, definitely check it out.

ZFS metadata can also affect databases. When a database is rapidly changing, writing out two or three copies of the metadata for each change can take up a significant number of the available IOPS of the backing storage. Normally, the quantity of metadata is relatively small compared to the default 128 KB record size. Databases work better with small record sizes, though. Keeping three copies of the metadata can cause as much disk activity, or more, than writing actual data to the pool.

Newer versions of OpenZFS also contain a `redundant_metadata` property, which defaults to *all*. This is the original behavior from previous versions of ZFS. However, this property can also be set to *most*, which causes ZFS to reduce the number of copies of some types of metadata that it keeps.

Depending on your needs and workload, allowing the database engine to manage caching might be better. ZFS defaults to caching much or all of the data from your database in the ARC, while the database engine keeps its own cache, resulting in wasteful double caching. Setting the `primarycache` property to *metadata* rather than the default *all* tells ZFS to avoid caching actual data in the ARC. The `secondarycache` property similarly controls the L2ARC.

Depending on the access pattern and the database engine, ZFS may already be more efficient. Use a tool like `zfsmon` from the `zfs-tools` package to monitor the ARC cache hit ratio, and compare it to that of the database's internal cache.

Once the Compressed ARC feature is available, it might be wise to consider reducing the size of the database's internal cache and let ZFS handle the caching instead. The ARC might be able to fit significantly more data in the same amount of RAM than your database can.

Now let's talk about some specific databases.

MySQL—InnoDB/XtraDB

InnoDB became the default storage engine in MySQL 5.5 and has significantly different characteristics than the previously used MyISAM engine. Percona's XtraDB, also used by MariaDB, is similar to InnoDB. Both InnoDB and XtraDB use a 16 KB block size, so the ZFS dataset that contains the actual data files should have its `recordsize` property set to match. We also recommend using MySQL's `innodb_one_file_per_table` setting to keep the InnoDB data for each table in a separate file, rather than grouping it all into a single `ibdata` file. This makes snapshots more useful and allows more selective restoration or rollback.

Store different types of files on different datasets. The data files need 16 KB block size, lz4 compression, and reduced metadata. You might see performance gains from caching only metadata, but this also disables prefetch. Experiment and see how your environment behaves.

```
# zfs create -o recordsize=16k -o compress=lz4 -o redundant_
metadata=most -o primarycache=metadata mypool/var/db/mysql
```

The primary MySQL logs compress best with gzip, and don't need caching in memory.

```
# zfs create -o compress=gzip1 -o primarycache=none mysql/var/
log/mysql
```

The replication log works best with lz4 compression.

```
# zfs create -o compress=lz4 mypool/var/log/mysql/replication
```

Tell MySQL to use these datasets with these my.cnf settings.

```
data_path=/var/db/mysql
log_path=/var/log/mysql
binlog_path=/var/log/mysql/replication
```

You can now initialize your database and start loading data.

MySQL—MyISAM

Many MySQL applications still use the older MyISAM storage engine, either because of its simplicity or just because they have not been converted to using InnoDB.

MyISAM uses an 8 KB block size. The dataset record size should be set to match. The dataset layout should otherwise be the same as for InnoDB.

PostgreSQL

ZFS can support very large and fast PostgreSQL systems, if tuned properly. Don't initialize your database until you've created the needed datasets.

PostgreSQL defaults to using 8 KB storage blocks for everything. If you change PostgreSQL's block size, you must change the dataset size to match.

The examples here use FreeBSD. Other operating systems will use different paths and have their own database initialization scripts. Substitute your preferred operating system commands and paths as needed.

PostgreSQL data goes in `/usr/local/pgsql/data`. For a big install, you probably have a separate pool for that data. Here I'm using the pool `pgsql` for PostgreSQL.

```
# zfs set mountpoint=/usr/local/pgsql pgsql
# zfs create pgsql/data
```

Now we have a chicken-and-egg problem. PostgreSQL's database initialization routine expects to create its own directory tree, but we want particular subdirectories to have their own datasets. The easiest way to do this is to let PostgreSQL initialize, and then create datasets and move the files. Here's how FreeBSD initializes a PostgreSQL database.

```
# /usr/local/etc/rc.d/postgresql oneinitdb
```

The initialization routine creates databases, views, schemas, configuration files, and all the other components of a high-end database. Now you can create datasets for the special parts.

Our test system's PostgreSQL install stores databases in /usr/local/pgsql/data/base. The Write Ahead Log, or WAL, lives in `/usr/local/pgsql/data/pg_xlog`. Move both of these out of the way.

```
# cd /usr/local/pgsql/data
# mv base base-old
# mv pg_xlog pg_xlog-old
```

Both of these parts of PostgreSQL use an 8 KB block size, and you would want to snapshot them separately, so create a dataset for each. As with MySQL, tell the ARC to cache only the metadata. Also tell these datasets to bias throughput over latency with the `logbias` property.

```
# zfs create -o recordsize=8k -o redundant_metadata=most -o
primarycache=metadata logbias=throughput pgsql/data/pg_xlog
# zfs create -o recordsize=8k -o redundant_metadata=most -o
primarycache=metadata logbias=throughput pgsql/data/base
```

Copy the contents of the original directories into the new datasets.

```
# cp -Rp base-old/* base
# cp -Rp pg_xlog-old/* pg_xlog
```

You can now start PostgreSQL.

Tuning for File Size

ZFS is designed to be a good general-purpose file system. If you have a ZFS system serving as file server for a typical office, you don't really have to tune for file size. If you know what size of files you're going to have, though, you can make changes to improve performance.

Small Files

When creating many small files at high speed in a system without a SLOG (Separate (ZFS-Intent) Log), ZFS spends a significant amount of time waiting for the files and metadata to finish flushing to stable storage.

If you are willing to risk the loss of any new files created in the last five seconds (or more if your vfs.zfs.txg.timeout is higher), setting the sync property to disabled tells ZFS to treat all writes as asynchronous. Even if an application asks that it not be told that the write is complete until the file is safe, ZFS returns immediately and writes the file along with the next regularly scheduled txg.

A high-speed SLOG lets you store those tiny files both synchronously and quickly.

Big Files

ZFS recently added support for blocks larger than 128 KB via the `large_block` feature. If you're storing many large files, certainly consider this. The default maximum block size is 1 MB.

Theoretically, you can use block sizes larger than 1 MB. Very few systems have extensively tested this, however, and the interaction with the kernel memory allocation subsystem has not been tested under prolonged use. You can try really large record sizes, but be sure to file a bug report when everything goes sideways.

On FreeBSD, the `sysctl vfs.zfs.max_recordsize` controls the maximum block size. On Linux, `zfs_max_recordsize` is a module parameter.

Once you activate `large_blocks` (or any other feature), the pool can no longer be used by hosts that do not support the feature. Deactivate the feature by destroying any datasets that have ever had their `recordsize` set to larger than 128 KB.

Storage systems struggle to balance latency and throughput. ZFS uses the `logbias` property to decide which way it should lean. ZFS uses a `logbias` of *latency* by default, so that data is quickly synched to disk, allowing databases and other applications to continue working. When dealing with large files, changing the `logbias` property to *throughput* might result in better performance. You must do your own testing and decide which setting is right for your workload.

With a few adjustments, you can make your database's file system fly...leaving you capacity to cope with your next headache.

This article was adapted from Allan Jude and Michael W Lucas, *FreeBSD Mastery: Advanced ZFS* (Tilted Windmill Press, 2016).