RECORDSIZE=1M, XATTR=SA, ASHIFT=13, ATIME=OFF, COMPRESSION=LZ4 —

ZFS 101—Understanding ZFS storage and performance

Learn to get the most out of your ZFS filesystem in our new series on storage fundamentals.

JIM SALTER - 5/8/2020, 2:00 PM

Zpools, vdevs, and devices

To really understand ZFS, you need to pay real attention to its actual structure. ZFS merges the traditional volume management and filesystem layers, and it uses a copy-on-write transactional mechanism—both of these mean the system is very structurally different than conventional filesystems and RAID arrays. The first set of major building blocks to understand are `zpools`, `vdevs`, and `devices`.

zpool

The `zpool` is the uppermost ZFS structure. A zpool contains one or more `vdevs`, each of which in turn contains one or more `devices`. Zpools are self-contained units—one physical computer may have two or more separate zpools on it, but each is entirely independent of any others. Zpools cannot share `vdevs` with one another.

ZFS redundancy is at the `vdev` level, not the `zpool` level. There is absolutely *no* redundancy at the zpool level—if any storage `vdev` or `SPECIAL` vdev is lost, the entire `zpool` is lost with it.

Modern zpools can survive the loss of a `CACHE` or `LOG` vdev—though they may lose a small amount of dirty data, if they lose a `LOG` vdev during a power outage or system crash.

It is a common misconception that ZFS "stripes" writes across the pool—but this is inaccurate. A zpool is not a funny-looking RAID0—it's a funny-looking JBOD, with a complex distribution mechanism subject to change.

For the most part, writes are distributed across available vdevs in accordance to their available free space, so that all vdevs will theoretically become full at the same time. In more recent versions of ZFS, vdev utilization may also be taken into account—if one vdev is significantly busier than another (ex: due to read load), it may be skipped temporarily for write despite having the highest ratio of free space available.

The utilization awareness mechanism built into modern ZFS write distribution methods can decrease latency and increase throughput during periods of unusually high load—but it should not be mistaken for *carte blanche* to mix slow rust disks and fast SSDs willy-nilly in the same pool. Such a mismatched pool will still generally perform as though it were entirely composed of the slowest device present.

vdev

Each `zpool` consists of one or more `vdevs`(short for virtual device). Each vdev, in turn, consists of one or more real `devices`. Most vdevs are used for plain storage, but several special support classes of vdev exist as well—including `CACHE`, `LOG`, and `SPECIAL.` Each of these vdev types can offer one of five topologies —single-device, RAIDz1, RAIDz2, RAIDz3, or mirror.

RAIDz1, RAIDz2, and RAIDz3 are special varieties of what storage greybeards call "diagonal parity RAID." The 1, 2, and 3 refer to how many parity blocks are allocated to each data stripe. Rather than having entire disks dedicated to parity, RAIDz vdevs distribute that parity semi-evenly across the disks. A RAIDz array can lose as many disks as it has parity blocks; if it loses another, it fails, and takes the `zpool` down with it.

Mirror vdevs are precisely what they sound like—in a mirror vdev, each block is stored on every device in the vdev. Although two-wide mirrors are the most common, a mirror vdev can contain any arbitrary number of devices—three-way are common in larger setups for the higher read performance and fault resistance. A mirror vdev can survive any failure, so long as at least one device in the vdev remains healthy.

Single-device vdevs are also just what they sound like—and they're inherently dangerous. A single-device vdev cannot survive any failure—and if it's being used as a storage or `SPECIAL` vdev, its failure will take the entire `zpool` down with it. Be very, very careful here.

`CACHE`, `LOG`, and `SPECIAL` vdevs can be created using any of the above topologies—but remember, loss of a `SPECIAL` vdev means loss of the pool, so redundant topology is strongly encouraged.

device

This is probably the easiest ZFS related term to understand—it's literally just a random-access block device. Remember, `vdevs` are made of individual devices, and the `zpool` is made of `vdevs`.

Disks—either rust or solid-state—are the most common block devices used as `vdev` building blocks. Anything with a descriptor in `/dev` that allows random access will work, though—so entire hardware RAID arrays can be (and sometimes are) used as individual devices.

The simple raw file is one of the most important alternative block devices a `vdev` can be built from. Test pools made of sparse files are an incredibly convenient way to practice zpool commands, and see how much space is available on a pool or vdev of a given topology.

Let's say you're thinking of building an eight-bay server, and pretty sure you'll want to use 10TB (~9300 GiB) disks—but you aren't sure what topology best suits your needs. In the above example, we build a test pool out of sparse files in seconds—and now we know that a RAIDz2 vdev made of eight 10TB disks offers 50TiB of usable capacity.

There's one special class of `device`—the `SPARE`. Hotspare devices, unlike normal devices, belong to the entire pool, not a single `vdev`. If any `vdev` in the pool suffers a device failure, and a SPARE is attached to the pool and available, the SPARE will automatically attach itself to the degraded `vdev`.

Once attached to the degraded `vdev`, the SPARE begins receiving copies or reconstructions of the data that should be on the missing device. In traditional RAID, this would be called "rebuilding"—in ZFS, it's called "resilvering."

It's important to note that SPARE devices don't permanently replace failed devices. They're just placeholders, intended to minimize the window during which a `vdev` runs degraded. Once the admin has replaced the vdev's failed device and the new, permanent replacement device resilvers, the SPARE detaches itself from the vdev, and returns to pool-wide duty.

Datasets, blocks, and sectors

The next set of building blocks you'll need to understand on your ZFS journey relate not so much to the hardware, but how the data itself is organized and stored. We're skipping a few levels here—such as the metaslab—in the interests of keeping things as simple as possible, while still understanding the overall structure.

Datasets

A ZFS `dataset` is roughly analogous to a standard, mounted filesystem—like a conventional filesystem, it appears to casual inspection as though it's "just another folder." But also like conventional mounted filesystems, each ZFS `dataset` has its own set of underlying properties.

First and foremost, a `dataset` may have a quota assigned to it. If you `zfs set quota=100G poolname/datasetname`, you won't be able to put more than 100GiB of data into the system mounted folder `/poolname/datasetname`.

Notice the presence—and absence—of leading slashes in the above example? Each dataset has its place in both the ZFS hierarchy, and the system mount hierarchy. In the ZFS hierarchy, there is no leading slash—you begin with the name of the pool, and then the path from one dataset to the next—eg `pool/parent/child` for a dataset named `child` under the parent dataset `parent`, in a pool creatively named `pool`.

By default, the mountpoint of a `dataset` will be equivalent to its ZFS hierarchical name, with a leading slash—the pool named `pool` is mounted at `/pool`, dataset `parent` is mounted at `/pool/parent`, and child dataset `child` is mounted at `/pool/parent/child`. The system mountpoint of a dataset may be altered, however.

If we were to `zfs set mountpoint=/lol pool/parent/child`, the dataset `pool/parent/child` would actually be mounted on the system as `/lol`.

In addition to datasets, we should mention `zvols`. A `zvol` is roughly analogous to a `dataset`, except it doesn't actually have a filesystem in it—it's just a block device. You could, for example, create a `zvol` named `mypool/myzvol`, then format it with the ext4 filesystem, then mount that filesystem—you now have an ext4 filesystem, but backed with all of the safety features of ZFS! This might sound silly on a single computer—but it makes a lot more sense as the back end for an iSCSI export.

Blocks

In a ZFS pool, all data—including metadata—is stored in `blocks`. The maximum size of a `block` is defined for each `dataset` in the `recordsize` property. Recordsize is mutable, but changing `recordsize` won't change the size or layout of any `blocks` which have already been written to the dataset—only for new blocks as they are written.

If not otherwise defined, the current default `recordsize` is 128KiB. This represents a sort of uneasy compromise in which performance won't be ideal for much of anything, but won't be awful for much of anything either. `Recordsize` may be set to any value from 4K through 1M. (`Recordsize` may be set even larger with additional tuning and sufficient determination, but that's rarely a good idea.)

Any given `block` references data from only one file—you can't cram two separate files into the same `block`. Each file will be composed of one or more `blocks`, depending on size. If a file is smaller than `recordsize`, it will be stored in an undersized block—for example, a `block` holding a 2KiB file will only occupy a single 4KiB `sector` on disk.

If a file is large enough to require multiple `blocks`, all records containing that file will be `recordsize` in length—including the last record, which may be mostly slack space.

`Zvols` do not have the `recordsize` property—instead, they have `volblocksize`, which is roughly equivalent.

Sectors

The final building block to discuss is the lowly `sector`. A `sector` is the smallest physical unit that can be written to or read from its underlying `device`. For several decades, most disks used 512 byte `sectors`. More recently, most disks use 4KiB `sectors`, and some—particularly SSDs—use 8KiB `sectors`, or even larger.

ZFS has a property which allows you to manually set the `sector` size, called `ashift`. Somewhat confusingly, `ashift` is actually the binary exponent which represents sector size—for example, setting `ashift=9` means your `sector` size will be 2^9, or 512 bytes.

ZFS queries the operating system for details about each block `device` as it's added to a new `vdev`, and in theory will automatically set `ashift` properly based on that information. Unfortunately, there are many disks that lie through their teeth about what their `sector` size is, in order to remain compatible with Windows XP (which was incapable of understanding disks with any other `sector` size).

This means that a ZFS admin is strongly advised to be aware of the actual `sector` size of his or her `devices`, and manually set `ashift` accordingly. If `ashift` is set too low, an astronomical read/write amplification penalty is incurred—writing a 512 byte "sectors" to a 4KiB real `sector` means having to write the first "sector", then read the 4KiB `sector`, modify it with the second 512 byte "sector", write it back out to a *new* 4KiB `sector`, and so forth, for every single write.

In real world terms, this amplification penalty hits a Samsung EVO SSD—which should have `ashift=13`, but lies about its sector size and therefore defaults to `ashift=9` if not overridden by a savvy admin—hard enough to make it appear *slower* than a conventional rust disk.

By contrast, there is virtually no penalty to setting `ashift` too high. There is no real performance penalty, and slack space increases are infinitesimal (or zero, with compression enabled). We strongly recommend even disks that really *do* use 512 byte sectors should be set `ashift=12` or even `ashift=13` for future-proofing.

The `ashift` property is per-`vdev`—*not* per pool, as is commonly and mistakenly thought!—and is immutable, once set. If you accidentally flub `ashift` when adding a new vdev to a pool, you've irrevocably contaminated that pool with a drastically under-performing `vdev`, and generally have no recourse but to destroy the pool and start over. Even `vdev` removal can't save you from a flubbed `ashift` setting!

Copy-on-Write semantics

CoW—Copy on Write—is a fundamental underpinning beneath most of what makes ZFS awesome. The basic concept is simple—if you ask a traditional filesystem to modify a file in-place, it does precisely what you asked it to. If you ask a copy-on-write filesystem to do the same thing, it says "okay"—but it's lying to you.

Instead, the copy-on-write filesystem writes out a new version of the `block` you modified, then updates the file's metadata to unlink the old `block`, and link the new `block` you just wrote.

Unlinking the old `block` and linking in the new is accomplished in a single operation, so it can't be interrupted—if you dump the power after it happens, you have the new version of the file, and if you dump power before, then you have the old version. You're always filesystem-consistent, either way.

Copy-on-write in ZFS isn't only at the filesystem level, it's also at the disk management level. This means that the RAID hole—a condition in which a stripe is only partially written before the system crashes, making the array inconsistent and corrupt after a restart—doesn't affect ZFS. Stripe writes are atomic, the vdev is always consistent, and Bob's your uncle.

ZIL—the ZFS Intent Log

There are two major categories of write operations—synchronous (sync) and asynchronous (async). For most workloads, the vast majority of write operations are asynchronous—the filesystem is allowed to aggregate them and commit them in batches, reducing fragmentation and tremendously increasing throughput.

Sync writes are an entirely different animal—when an application requests a sync write, it's telling the filesystem "you need to commit this to non volatile storage *now*, and until you do so, I can't do anything else." Sync writes must therefore be committed to disk immediately—and if that increases fragmentation or decreases throughput, so be it.

ZFS handles sync writes differently from normal filesystems—instead of flushing out sync writes to normal storage immediately, ZFS commits them to a special storage area called the ZFS Intent Log, or ZIL. The trick here is, those writes *also* remain in memory, being aggregated along with normal asynchronous write requests, to later be flushed out to storage as perfectly normal TXGs (Transaction Groups).

In normal operation, the ZIL is written to and never read from again. When writes saved to the ZIL are committed to main storage from RAM in normal TXGs a few moments later, they're unlinked from the ZIL. The only time the ZIL is ever *read* from is upon pool import.

If ZFS crashes—or the operating system crashes, or there's an unhandled power outage—while there's data in the ZIL, that data will be read from during the next pool import (eg when a crashed system is restarted). Whatever is in the ZIL will be read in, aggregated into TXGs, committed to main storage, and then unlinked from the ZIL during the import process.

One of the classes of support `vdev` available is `LOG`—also known as SLOG, or Secondary LOG device. All the SLOG does is provide the pool with a separate—and hopefully far faster, with very high write endurance—`vdev` to store the `ZIL` in, instead of keeping the `ZIL` on the main storage `vdevs`. In all respects, the `ZIL` behaves the same whether it's on main storage, or on a `LOG` vdev—but if the `LOG` vdev has very high write performance, then sync write returns will happen very quickly.

Adding a `LOG` vdev to a pool absolutely **cannot** and **will not** directly improve asynchronous write performance—even if you force all writes into the ZIL using `zfs set sync=always`, they still get committed to main storage in TXGs in the same way and at the same pace they would have without the `LOG`. The only direct performance improvements are for synchronous write latency (since the `LOG`'s greater speed enables the `sync` call to return faster).

However, in an environment that already requires lots of sync writes, a `LOG` vdev can indirectly accelerate asynchronous writes and uncached reads as well. Offloading ZIL writes to a separate `LOG` vdev means less contention for IOPS on primary storage, thereby increasing performance for all reads and writes to some degree.

Snapshots

Copy-on-write semantics are also the necessary underpinning for ZFS's atomic snapshots and incremental asynchronous replication. The live filesystem has a tree of pointers marking all the `records` that contain current data—when you take a snapshot, you simply make a copy of that tree of pointers.

When a record is overwritten in the live filesystem, ZFS writes the new version of the `block` to unused space first. Then it unlinks the old version of the `block` from the current filesystem. But if any `snapshot` references the old `block`, it still remains immutable. The old `block` won't actually be reclaimed as free space until all `snapshots` referencing that `block` have been destroyed!

Replication

FURTHER READING
rsync.net: ZFS Replication to the cloud is finally here—and it's *fast*

Once you understand how snapshots work, you're in a good place to understand replication. Since a snapshot is simply a tree of pointers to `records`, it follows that if we `zfs send` a snapshot, we're sending both that tree and all of the associated records. When we pipe that `zfs send` to a `zfs receive` on the target, it writes both the actual `block` contents, and the tree of pointers referencing the `blocks`, into the target dataset.

Things get more interesting on your second `zfs send`. Now that you've got two systems, each containing the snapshot `poolname/datasetname@1`, you can take a new snapshot, `poolname/datasetname@2`. So on the source pool, you have `datasetname@1` and `datasetname@2`, and on the target pool, so far you just have the first snapshot—`datasetname@1`.

Since we have a common snapshot between source and destination—`datasetname@1`—we can build an *incremental* `zfs send` on top of it. When we ask the system to `zfs send -i poolname/datasetname@1 poolname/datasetname@2`, it compares the two pointer trees. Any pointers which exist only in `@2` obviously reference new `blocks`—so we'll need the contents of those `blocks` as well.

On the remote system, piping in the resulting incremental `send` is similarly easy. First, we write out all the new `records` included in the `send` stream, then we add in the pointers to those `blocks`. Presto, we've got `@2` on the new system!

ZFS asynchronous incremental replication is an enormous improvement over earlier, non-snapshot-based techniques like `rsync`. In both cases, only the changed data needs to get sent over the wire—but `rsync` must first *read* all the data from disk, on both sides, in order to checksum and compare it. By contrast, ZFS replication doesn't need to read anything but the pointer trees—and any `blocks` that pointer tree contains which *weren't* already present in the common snapshot.

Inline compression

Copy-on-write semantics also make it easier to offer inline compression. With a traditional filesystem offering in-place modification, compression is problematic—both the old version and the new version of the modified data must fit in exactly the same space.

If we consider a chunk of data in the middle of a file which begins life as 1MiB of zeroes—`0x00000000` ad nauseam—it would compress down to a single disk sector very easily. But what happens if we replace that 1MiB of zeroes with 1MiB of incompressible data, such as JPEG or pseudo-random noise? Suddenly, that 1MiB of data needs 256 4KiB sectors, not just one—and the hole in the middle of the file is only one sector wide.

ZFS doesn't have this problem, since modified records are always written to unused space—the original `block` only occupies a single 4KiB `sector`, and the new record occupies 256 of them, but that's not an issue—the newly modified chunk from the "middle" of the file would have been written to unused space whether its size changed or not, so for ZFS, this "problem" is just another day at the office.

ZFS's inline compression is off by default, and it offers pluggable algorithms—currently including LZ4, gzip (1-9), LZJB, and ZLE.

- **LZ4** is a stream algorithm offering extremely rapid compression and decompression, and is a performance win for the majority of use cases—even with very anemic CPUs.
- **GZIP** is the venerable algorithm all Unix-like users know and love. It can be implemented with compression levels 1-9, with increasing compression ratio and CPU usage as levels approach 9. Gzip

may be a win for all-text (or otherwise extremely compressible) use-cases, but frequently results in CPU bottlenecks otherwise—use with caution, particularly at higher levels.

- **LZJB** is the original algorithm used by ZFS. It is deprecated, and should no longer be used—LZ4 is superior in every metric.
- **ZLE** is Zero Level Encoding—it leaves normal data alone entirely, but will compress large sequences of zeroes. Useful for entirely incompressible datasets (eg JPEG, MP4, or other already-compressed formats), since it ignores the incompressible data, but compresses slack space on final records.

We recommend LZ4 compression for nearly any conceivable use-case; the performance penalty when it encounters incompressible data is very small, and the performance *gain* for typical data is significant. Copying a VM image for a new Windows operating system installation (just the installed Windows operating system, no data on it yet) went 27% faster with `compression=lz4` than `compression=none` in this 2015 test.

ARC—the Adaptive Replacement Cache

ZFS is the only modern filesystem we know of which uses its own read cache mechanism, rather than relying on its operating system's page cache to keep copies of recently-read blocks in RAM for it.

Although the separate cache mechanism has its problems—ZFS cannot react to new requests to allocate memory as immediately as the kernel can, and therefore a new `mallocate()` call may fail, if it would need RAM currently occupied by the ARC—there's good reason, at least for now, for putting up with it.

Every well-known modern operating system—including MacOS, Windows, Linux, and BSD—uses the LRU (Least Recently Used) algorithm for its page cache implementation. LRU is a naive algorithm that bumps a cached block up to the "top" of the queue each time it's read, and evicts blocks from the "bottom" of the queue as necessary to add new cache misses (blocks which had to be read from disk, rather than cache) at the "top."

This is fine so far as it goes, but in systems with large working data sets, the LRU can easily end up "thrashing"—evicting very frequently-needed blocks, to make room for blocks that will never get read from cache again.

The ARC is a much less naive algorithm, which can be thought of as a "weighted" cache. Each time a cached block is read, it becomes a little "heavier" and more difficult to evict—and even after an eviction, the evicted block is *tracked* for a period of time. A block which has been evicted but then must be read back into cache will also become "heavier" and more difficult to evict.

The end result of all this is a cache with typically far greater hit ratios—the ratio between cache hits (reads served from cache) and cache misses (reads served from disk). This is an extremely important statistic—not only are cache hits themselves served orders of magnitude more rapidly, the cache *misses* can also be served more rapidly, since more cache hits==fewer concurrent requests to disk==lower latency for those remaining misses which *must* be serviced from disk.

Conclusion

Now that we've covered the basic semantics of ZFS—how copy-on-write works, and the relationships between pools, vdevs, blocks, sectors and files—we're ready to talk actual performance, with real numbers.

Stay tuned for the next installment of our storage fundamentals series to see the actual performance seen in pools using mirror and RAIDz vdevs, as compared with both one another and the traditional Linux kernel RAID topologies we explored earlier.

Initially, we're just going to cover the basics—the ZFS topologies themselves—but after *that*, we'll be ready to talk about more advanced ZFS setup and tuning, including the use of support vdev types like L2ARC, SLOG, and Special Allocation.