NexentaStor: ZFS Copy-on-Write, Checksums, and Consistency

A Common Whiteboard Session Captured on Paper

Chris Nelson, Sales Engineer, Nexenta Systems

May 2012

NexentaStor: ZFS Copy-on-Write, Checksums, and Consistency

Introduction

This paper started as an explanation of the term “copy on write” as it applies to ZFS, but the foundational concepts relevant to that discussion are equally applicable to many other great features of ZFS as well, so this paper was expanded and then broken apart to cover a broader selection of ZFS highlights often described in a whiteboard session. This first paper in a series starts with three necessary concepts, and the subsequent papers build upon this beginning.

Three Key Features of ZFS

Since it was first introduced, almost every slide deck describing ZFS has contained a diagram much like Figure 1. Unfortunately, as accurate as that diagram is, it leaves much to be explained by the presenter, so those reviewing the slides without the benefit of the presentation audio or video are left to make their own conclusions. To understand what the slide describes, it is necessary to know a few key things about ZFS:

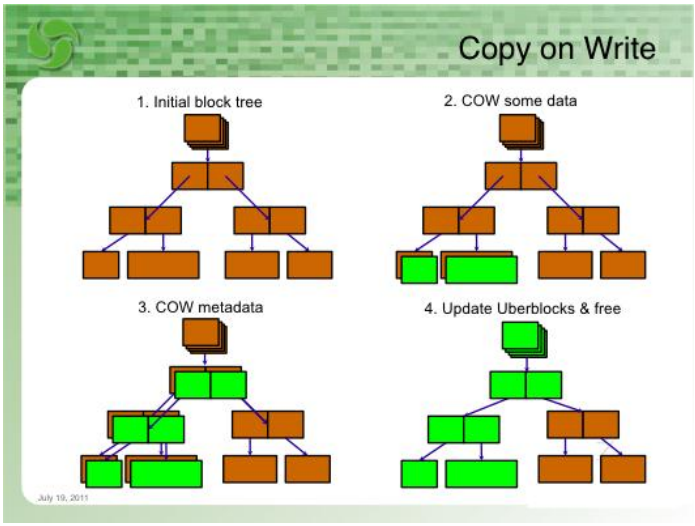
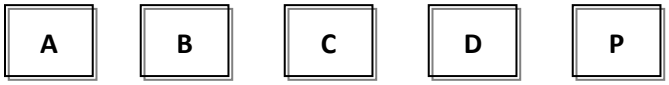
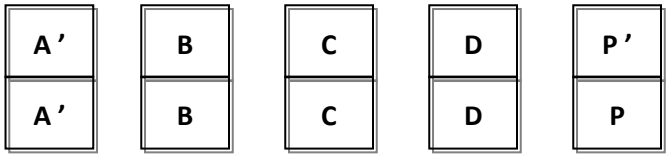


Figure 1. A common ZFS "Copy on Write" diagram

1) ZFS Never Modifies Data in Place: Doing so would expose the possibility of data corruption common to other file systems when a write is not completed properly (known as the "RAID-5 Write Hole"). Consider the following example, where pieces of data A, B, C, and D result in parity P written in a traditional RAID-5 file system:



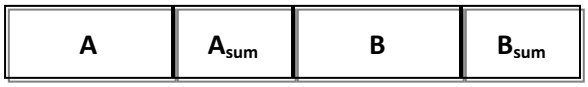
If A, then, is to be modified to A', of course the parity must be recalculated to P' and updated as well—as shown here:



If, however, the storage system is interrupted between the modification of A to A' and P to P', it is possible to end up with a situation as follows—where P is the old and incorrect parity value for A', B, C, and D.

This, then, is corrupted data. The traditional RAID-5 file system is unaware of the problem, and, in fact, most traditional file systems blindly trust parity data. It is possible that this will go undetected for a period of time, but if P is ever called upon (e.g., during the reconstruction of a drive that contained A'), it would result in entirely incorrect data in the file system. By never modifying data in place, ZFS avoids this problem.

2) ZFS Checksums Every Data and Metadata Block in the File System, and the Checksums are Stored with the Parent: Traditional file systems store a data block's checksum physically next to the data. If the checksum algorithm is strong (and not all file systems use a strong one), reading the checksum and comparing it to the data will detect an error within the data block (often called bit rot or digital decay).



This does not, however, protect against several other very real problems—things like phantom writes (where a drive claimed to have written data but never did), misdirected reads or writes (where a drive read or wrote a data block with a matching checksum but in the wrong location), and other problems (like drive firmware bugs). **ZFS, however, can detect all of these conditions (and correct them in mirrored or RAIDZ configurations!)** because the checksums for each data and metadata block are stored in physically separate locations with pointer information from "parent" to "child"—often depicted as a tree as in the following diagram. These checksums are calculated and compared every time ZFS walks through this tree to retrieve a piece of data, and if data is found to be in error, ZFS will use parity to return the correct data. It then will fix the bad data block as well! Note that even the highest parent metadata block is itself protected by a checksum stored in an "uberblock" (the entry point of a ZFS file system)—of which there are multiple copies stored in strategic places on disk.

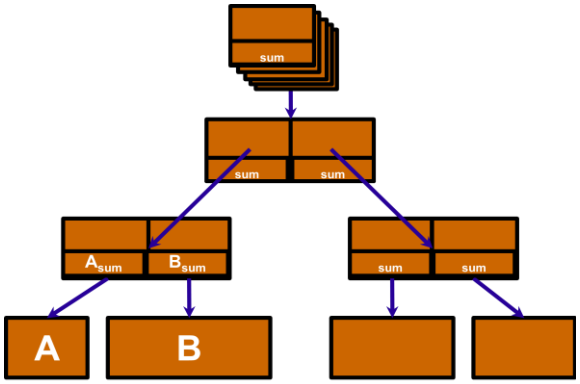


Figure 2: ZFS stores checksums separate from data

3) ZFS Always is Consistent on Disk: There is never a need for a file system check (fsck) in ZFS. Since ZFS never modifies data in place, new data is written to new space—starting from the bottom of the tree (data) and working up (metadata), but only when the entry point to the file system (known as the "uberblock" in ZFS parlance) is updated is the state of the file system actually changed. Consider the first diagram again—shown again here:

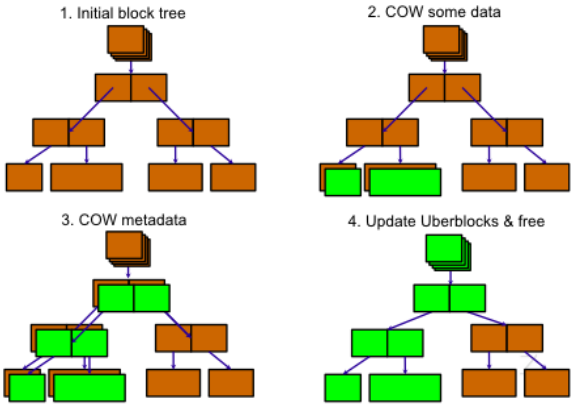


Figure 3. ZFS updates are atomic; the file system state is always consistent.

In the first state (#1 in Figure 3, above), the top-center uberblock is the entry state to the file system. To retrieve any data block at the bottom, ZFS starts with the uberblock and navigates through the appropriate metadata block pointers (comparing checksums along the way) to retrieve the appropriate data and be sure of its integrity.

In the next state (#2, above), the process of modifying the bottom-left two data blocks has begun. The green blocks depict the new data written in a new place, but that means that new parent block pointers, with checksums for the new data, are required as well.

In the third state (#3, above), most of the necessary new metadata block pointers have been written, but the uberblock has not yet been updated. Note that the orange state of the file system still is perfectly consistent, because all of the metadata and data still is intact. If power was lost at this point, when the system came up again, it could still navigate the orange file system state without error.

Finally (step #4, above), the uberblock is updated in what is known as an "atomic" operation—meaning it either happens successfully or it does not happen at all. At this point, the new state of the file system—containing all of the green blocks and the unchanged orange blocks from the right-hand side—also is perfectly consistent and updated with the new bottom-left data. Note that ZFS moves from one perfectly consistent file system state to another. The old orange data and metadata blocks now can be free for future use.

This process is where the notion of "copy on write" in ZFS comes into play. There are times when multiple parent block pointers can refer to a single physical block of data on disk, and this efficiency poses no problem for ZFS. When one of the users of that data needs to modify it, the new data is written to a new place, and the appropriate changes to the metadata are made. Some have described that as "redirect on write," but understanding the process is more important than settling on a name. (Note that not all of the updates depicted in the diagrams above are necessarily writes to disk; many of these changes happen simultaneously when ZFS flushes its transaction groups from memory to disk. There are a lot of additional optimizations not described here.)

So where is the confusion on Copy-on-Write with ZFS? The idea of Copy-on-Write—often abbreviated as "COW"—has been around for years. The original concept is an optimization strategy used in computing. The idea is that multiple users unknowingly share a resource (e.g., a piece of memory or disk space) as long as the resource is exactly the same; only when one of the users needs to modify the resource does that user get its own copy of the resource to use however it desires. Since the users are completely unaware of this happening, the strategy is great for minimizing necessary resources and is used in many places in the computing industry today.

The trouble is that this strategy can be implemented in various ways, and some are more efficient than others. Specifically, in the world of snapshots on storage devices, there are some strikingly different implementations from different vendors that all use the "copy on write" terminology. While this series of papers will not attempt to explain all of the other "copy on write" implementations in the storage industry, a couple of rather well-known contrasts will be drawn that should help alleviate confusion when people believe they know what "copy on write" means and apply that incorrectly to their understanding of ZFS.

Read Next: *"NexentaStor: ZFS Snapshots and Replication"*