the wiz book

Torben Schinke

Version 1.0, 2017-12-05

Table of Contents

Dedication

For the family.

Preface

The idea of a robust, simple and scalable storage format superseeding the lowest denomiator filesystems, fascinated me already 15 years ago, however I never had the opportunity to actually start implementing such a project. When the time came, I started to design a paper based specification in 2015 which performs well for deduplicating large files, nested directory trees and continues snapshots. To solve the typical problems of a *multi file based document format* at work, I created a proprietary java based implementation from it, called wiz - which is just the opposite of a git, similarities are purely coincidental. For the original intention, it worked pretty well. But as requirements changed, the performance for a lot of additional use cases was disappointing. The main performance issues are caused by both, inherent format decisions and the necessity of a complex virtual machine. In practice, the latter caused also penalties on the probably most successful mobile platform of our time. To solve all of these issues I started to design an entirely new specification which addresses all of the new additional scenarios (and even more). Hereafter this new specification is actually *wiz version 3* or simply *wiz*. Therefore the proprietary existing wiz implementation is called *legacy wiz* and is not only implemented in a different language but also does a lot of things differently to improve performance, storage usage, reliability and system complexity. Today, the market for closed source commercial software libraries is nearly dead and gaining money or finding acceptance is not easy. Usually large companies dominate the market with a lot (but definity not all) high quality products.

Format specification

Wiz is both, an implementation and a specification. In this chapter only the specification matters and is described in a way that it can be implemented in any language or ecosystem.

A node always starts with a byte identifier and is otherwise undefined. Most lengths uses a varuint so that it has an adaptive overhead which increases dynamically as the payload size increases. Overhead also depends on the used compression algorithm, if a node supports that at all. The payload of a node should not exceed something reasonable, e.g. ZFS (see also [zfs-spec]) uses 128KiB but you may even go into the range of MiB to increase efficiency. The UTF8 type is always prefixed with a varuint length to increase efficiency for short strings but allowing also more than the typical 64k bytes. All numbers are treated as big endian to match network byte order. As a side note, even if most operating systems are little endian today, one cannot ignore BE systems, so any code must be endian independent anyway. It is not expected that wiz may profit from a system specific endianess, like ZFS does.

Magic node

Marks a container and must be always the first node of a file and should not occur once again. If it does (e.g. for recovery purposes), it is not allowed to be contradictory. Wiz containers can simply be identified using the magic bytes *[0x00 0x77 0x69 0x7a 0x63]*.

Table 1. on-disk format of the magic node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x00	type *header*
0x01	4	[]uint8	magic	[77 69 7a 63]	the magic header value
0x05	4	[]uint8	sub magic	[* * * *]	the user defined sub magic header value
0x09	4	uint32	version	0x03	this is the third version of the wiz format
0x13	1	uint8	encryption type	*	the kind of encryption algorithm

The *version* indicates which nodes and how they are defined. A node format may be changed in future revisions but should be extended in a backwards compatible manner. If such a thing is not possible (e.g. also by adding new kinds) the number increases. Because the format depends on the node kind (and therefore the sizes to parse) an outdated reader can actually only use it's recovery options to continue reading.

Some notes to the version flag: Actually this is the third generation of the wiz format. The first only existed on paper, the second was implemented largely based on the paper based specification but is

proprietary. So this is the first which is now open source. It is not only implemented in a different language but also does a lot of things differently to improve performance, storage usage, reliability and system complexity.

The known sub format identifiers of all known publicly available sub format identifiers.

Table 2. sub format identifiers

Value	Description
[77 69 7a 61]	*wiza* the standard archive format of the command line tool
[77 69 7a 62]	*wizb* the format of the backup tool

The encryption formats are defined as follows:

Table 3. encryption format identifiers

Value	Description
0x00	no encryption, all nodes are written as they are, just in plain bytes
0x01	AES-256 CTR mode

See the encryption chapter for the detailed specification of each encryption mode.

Configuration node

The wiz repository (as defined by the file) may include different properties. These properties are important to open the repository properly, e.g. picking the correct hash algorithm. Also it may contain persistent optional settings for tweaking. This node must always be located at file offset 0x1000. It is not intended to be modified on a regular basis.

Table 4. on-disk format of the configuration node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x01	type *configuration*
0x01	*	kvobj	key value	*	key value properties in kvobj format

TBD define kvobj format (xdr like zfs?), better keep that small and put it into it's own kvobj-node? (same as for nosql data nodes?)

Super node

The super node is a ring buffer having 128 transaction entries which are written in a round-robin manner. The transaction node with the highest transaction id and a valid checksum is the transaction node to use. If something went wrong, older transactions may be used for recovery, but the usefulness depends on the kind of damage. Usually one would expect that if the transaction is

written to the ring buffer and the underlying file system crashes, it hopefully will loose the data in the same order (the transaction node is always the last thing written), however there is no guarantee on that. Also fsync cannot protect us from that, because it is broken on many filesystems, even by design (see also [btrfs-fsync]). Today, I don't know how to solve that properly.

The super node is always located at file offset 0x2000 (TBD) and is defined as follows.

Table 5. on-disk format of the super node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x02	type *super*
0x01	128 * ?	[]tx-node	array	*	ring buffer of 128 transaction nodes

Transaction node

The transaction node is the entry point which defines an applied transaction and all references to nodes which describe the valid state of the storage. This includes references to the root nodes for snapshots (equivalent to tags and branches) and also to additional trees, holding information about reference counts and deleted nodes. The *transaction id* is found in all other written nodes (TBD) to easily identify which modifications belong a specific transaction (TBD, does not make sense when overwriting! snapshots). The id is strict monotonic increasing.

Table 6. on-disk format of the transaction node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x03	type *transaction*
0x01	8	uint64	transaction id	*	increasing number

Chapters can contain sub-sections nested up to three deep. [1: An example footnote.]

Chapters can have their own bibliography, glossary and index.

And now for something completely different: monkeys, lions and tigers (Bengal and Siberian) using the alternative syntax index entries. Note that multi-entry terms generate separate index entries.

Here are a couple of image examples: an [smallnew] example inline image followed by an example block image:

[Tiger image] | *images/tiger.png*

Figure 1. Tiger block image

Followed by an example table:

Table 7. An example table

Option	Description
-a *USER GROUP*	Add *USER* to *GROUP*.
-R *GROUP*	Disables access to *GROUP*.

Example 1. An example example

> Lorum ipum…

Sub-section with Anchor

Sub-section at level 2.

Chapter Sub-section

Sub-section at level 3.

Chapter Sub-section

Sub-section at level 4.

This is the maximum sub-section depth supported by the distributed AsciiDoc configuration. [2: A second example footnote.]

The Second Chapter

An example link to anchor at start of the first sub-section.

An example link to a bibliography entry [taoup].

The Third Chapter

Book chapters are at level 1 and can contain sub-sections.

Appendix A: Example Appendix

One or more optional appendixes go here at section level 1.

Appendix Sub-section

Sub-section body.

Example Bibliography

The bibliography list is a style of AsciiDoc bulleted list.

Books

- [taoup] Eric Steven Raymond. *The Art of Unix Programming*. Addison-Wesley. ISBN 0-13-142901-9.
- [walsh-muellner] Norman Walsh & Leonard Muellner. *DocBook - The Definitive Guide*. O'Reilly & Associates. 1999. ISBN 1-56592-580-7.
- [zfs-spec] http://www.giis.co.in/Zfs_ondiskformat.pdf
- [btrfs-fsync] https://btrfs.wiki.kernel.org/index.php/FAQ#Does_Btrfs_have_data.3Dordered_mode_like_Ext3.3F

Articles

- [abc2003] Gall Anonim. *An article*, Whatever. 2003.

Example Glossary

Glossaries are optional. Glossaries entries are an example of a style of AsciiDoc labeled lists.

A glossary term

The corresponding (indented) definition.

A second glossary term

The corresponding (indented) definition.

Example Colophon

Text at the end of a book describing facts about its production.

Example Index