the wiz book

Torben Schinke

Version 1.0, 2017-12-05

Table of Contents

Dedication

-

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 and ecosystem.

Everything in the world of wiz is represented by a node, which always starts with a byte identifier. Besides that, there are no other common properties among node types.

Some nodes support compression for their payload, but it is not a generic feature. In contrast to that, encryption is a property of the pageing infrastructure itself. I believe that providing unencrypted insight into meta information is already an absolute security flaw. Examples for this are stacked filesystems like EncFs or eCryptfs, which provides plain information about the folder structure, amount of files and the file sizes. Keeping facts about nodes unencrypted and just encrypt payload would be the same kind of security flaw.

Pointers to nodes have a variadic size from 2 bytes upto 20 bytes, which boils down to a theoretical 128 bit addressing, in which the first 10 bytes refer to the id of a virtual device (vdev) and the last 10 bytes to a physical offset. The additonal two bytes are caused by the overhead of the varint encoding. The vdev is usually a simple file in your host's filesystem. Using this technique, the pointer adapts itself to different use cases, like many small vdevs or a single large file. The usage of a varint favors storage costs over performance. Addressing millions of nodes, especially in independent vdev sets - I think of an online mirror storage - would waste a lot of space. In this scenario, one could argue that the performance penalties are also on the client side and will not stress the server at all.

All texts should be encoded in UTF-8, however there is no simple answer for filepaths. When taking a look at Git and how it handles filenames, it works perfectly on Linux filesystems, which just treat a filename as a byte sequence but fails miserably when mixing platforms like MacOS and Windows. Some users expect an UTF-8 normalization and others excoriate that. So we also keep that decision to a preprocess. Moreover, instead of following the errornous c approach of zero termination, we also use a varint as a length encoding, which has the same overhead for most expected strings (up to 127 byte, runes may vary). We define a text payload always as UTF-8 (a varint prefixed array of uint8) and undefined text payload as a varint prefixed array of uint8.

In general, byte order is the common network order - big endian. Most elements are already invariant to endianess, like varint or arrays. There is no expectation that a specialization to the host endianess will actually result in any performance gain, for an implementation of this specification.

In the following, offsets are always defined in hexadecimal values prefixed by 0x. Fixed array values are also declared using hex values, but with 0x omitted.

Table 1. specification of types

Size	Type	Description
1-10	varuint	unsigned, as defined in [varint]
1-10	varint	signed, as defined in [varint]
1-10	vdev id	varint id, must be unique within a vdev set

Size	Type	Description
2-20	ndptr	a node pointer is a double varint 128 bit address, where the first 10 byte determine the vdev id and the last 10 byte determine the physical offset in the vdev.
1-*	utf8	UTF-8 sequences are encoded with a prefix of the type *varuint* followed by an arbritary amount of bytes
2-*	wbo	a wbo serialized object

WBO specification

The *wiz binary object* serialization format is specified by the following BNF like declaration. It is somewhat comparable to the BSON format (see [bson]) but uses the packed varint format from above to improve space efficency. Due to the copy-on-write approach, we do not plan to update a distinct data field within a written structure. BSON cannot guarantee that either when increasing the length of a string.

Table 2. Pseudo BNF, types as uint8 in quotes

object ::= varuint varuint field_list	a WBO starts with the total object length in bytes (including nested objects), followed by the amount of field entries and the actual field_list
field_list ::= field field_list	the recursive definition
field_name ::= varuint (uint8*)	a varuint declares the number of (UTF-8) bytes to follow
field ::=	
"0x00" field_name uint8	byte / uint8
"0x01" field_name uint16	uint16
"0x02" field_name uint32	uint32
"0x03" field_name uint64	uint64
"0x04" field_name int8	int8
"0x05" field_name int16	int16
"0x06" field_name int32	int32
"0x07" field_name int64	int64
"0x08" field_name float32	float32
"0x09" field_name float64	float64
"0x0A" field_name complex64	complex64
"0x0B" field_name complex128	complex128

"0x0C" field_name varuint (uint8*)	a varuint declares the number of UTF-8 bytes to follow
"0x0D" field_name varuint (uint8*)	a varuint declares the number of bytes to follow
"0x0E" field_name varuint	a variable length unsigned integer in LEB 128 format (1 - 10 bytes)
"0x10" field_name varint	a variable length signed integer in LEB 128 format (1 - 10 bytes) with zigzag encoding
"0x11" field_name varuint varuint	the vdev id of two variable unsigned length integers
"0x13" field_name varuint type (type content bytes*)	an array with the bytes of the according type to follow. E.g. could be a list of float32 or object.
"0x14" field_name object	a field containing another (recursive) object definition

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 *[00 03 77 69 7a 63]*.

Table 3. on-disk format of the magic node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x00	type *header*
0x01	4	uint32	version	0x03	this is the third version of the wiz format
0x05	4	[]uint8	magic	[77 69 7a 63]	the magic header value *wizc* for the container
0x06	1	uint8	encryption type	*	the kind of encryption algorithm for the pages
0x07	*	utf8	sub magic	*	the user defined sub magic header value as varuint prefixed UTF-8

Offset	Size	Type	Name	Value	Description
#5	16	UUID	wiz file set identifier	*	the UUID of this wiz storage. Any vdev id and therefore ndptr is only valid within the same set of wiz files sharing the same UUID.
#6	1-10	varuint	vdev id	*	The unique vdev id of this wiz file within the file set. Should start with 0.

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.

One of the basic ideas of wiz is to replace custom *on disk formats* with something better. Today, probably the most widespreaded format is the zip file format from pkware. Amongst others, it is used by the entire Microsoft Office suite for their **x files*. To easily identify such subformats, the wiz header defines an UTF-8 subformat specifier. In the following table one can see a list of known sub format identifiers. If you create your own identifier, use your reversed company or product internet domain, e.g. *com.mycompany.myproduct* to minimize collisions. You may also invent your own file extension, but as a rule of thumb, you should never rely on it and check the magic node instead.

Table 4. known sub format identifiers

Value	Description
0x04 [77 69 7a 61]	*wiza* the standard archive format of the command line tool
0x04 [77 69 7a 62]	*wizb* the format of the backup tool

The encryption formats are defined as follows:

Table 5. 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.

A wiz storage may consist of multiple files or devices, which have each their own magic node but a unique vdev id. Any *ndptr* contains also that id, so referred nodes can be spreaded across vdevs. Use cases for this may be to improve performance, to create append-only / WORM (write once read many) storages or simply to attach additional storage volumes. To detect which vdevs belong to the same vdev set, a unique UUID is assigned to each set. You should not rely on a file name to identify a set, if the user has access to the files.

TIP	Choose wisely your trade-of when considering (large) file sets, especially when dealing with end users. A common expectation is that an application stores a document always in a single file.

It is a hard descision where to write and update the *super node*. Depending on the use case it is either unrealistic (linear growing amount of vdevs) or even impossible (WORM) to update existing vdevs, hence there is no definitive rule here.

CAUTION	Each application has to define where to write or update the *super node*.

In order to alleviate the situation, there are some well defined use cases. If a type matches your use case, apply one of the following rules.

Type 1

For single file formats (ever a single vdev) always update the ringbuffer.

Type 2

A performance optimized stripe vdev set (like RAID 0) only updates the ring buffer in the vdev with the lowest number (typical 0). Stripe sets are wobbly anyway. So actually *Type 1* is only a special case of a stripe set with a single vdev.

Type 3

For redundant vdevs (like mirrors / RAID 1 / RAID 5) always update the ringbuffer in every vdev.

Type 4

For WORM / append-only formats only write a new super node to the added vdev and never change an already written file.

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. It has a reserved maximum size of 128KiB(?).

Table 6. on-disk format of the configuration node

Offset	Size	Type	Name	Value	Description
0x00	1	uint8	node type	0x01	type *configuration*
0x01	*	wbo	configuration	*	key value properties in wbo 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 7. 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 8. 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 9. 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
- [varint] https://developers.google.com/protocol-buffers/docs/encoding
- [bson] http://bsonspec.org/spec.html

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