the wiz book -
a complete guide to the wiz storage

Torben Schinke

December 27, 2018

Contents

1 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.

2 Requirements

In software development one distinguishes functional and non-functional needs. A functional requirement (FR) describes what a system is supposed to do on a certain input and involves typically some sort of calculation and processing. In contrast to that, non-functional requirements (NFR) define how a system behaves, e.g. in terms of response speed, memory consumption or security. Another important aspect of NFRs is software quality and maintainability.

2.1 functional requirements

When following a comprehensive requirements analysis, one has to interview the target group or rather the customer. The result is a list of rated must- and should-be criteria. The following list is an opionated view of the requirements as we have identified them and only include the must-have needs.

2.1.1 FR-01: library

The wiz storage format is an embedded database and must be includable into existing or new programs. The lowest common denomiator is a c-based ABI. Even if there is nothing like a *standard* ABI, each relevant operating system provides a c toolchain. Therefore the library must support the c calling conventions for static or shared libraries for at least the following architecture and operating system combinations:

Operating System	**Architecture**
Linux	x86_32, x86_64, armv7, armv8
Windows	x86_32, x86_64
MacOS	x86_64
Android	armv8
iOS	armv8

Table 2.1: At least supported platforms

2.1.2 FR-02: tooling

The *wiz* command line tool is available for all architecture and operating system combinations as defined in table 2.1. This tool allows at least to create, modify and inspect *wiz files* on a file level, just like the unix *tar* or *zip* programs.

The man page and command line interface.

tbd

```
$ wiz create storage.wiz
```

2.1.3 FR-03: format

The wiz storage format can also be called a *repository*, a *filesystem* or a *database*. These terms can be used interchangeably. It supports the following modes of operation:

- A single repository file on a conventional filesystem.
- A single repository file with one or multiple external log files.
- Multiple repository files with none, one or multiple external log files.
- A raw disk file, with a maximum fixed size.
- A simple remote storage for repository files like a ftp server. These remotes do not necessarily support random access options or have other limitations like maximum file sizes, limited file name lengths, limited amount of entries per folder, no folder at all etc.
- Read-only variants of all noted above.

tbd

A cluster mode with a consensus algorithm. Note: RAFT is not a good one, because it fails to scale. Probably something simple, like sharding with delayed replication. IMHO better to have something without guarantees here, than a system which is irrecoverable broken after e.g. a split brain error.

2.1.4 FR-04: transaction

Everything in wiz is a transaction. These transactions are always isolated and provide a high read throughput by using MVCC (Multiversion Concurrency Control) implementations. A repository supports an arbitrary amount of sub volumes, sharing the available space of the entire storage. A sub volume may optionally support infinite snapshots, an infinite and cryptographically verifiable history and deduplication.

tbd

Deduplication can be a hard thing and has multiple trade ofs. ZFS has only online-deduplication and requires 1GiB RAM per 1TiB of storage to perform well. BTRFS only has offline-dedup but is hard to use and probably scales also badly memory wise.

2.1.5 FR-05: record size

The default record size for WIZ is 4096 bytes but a configurable record size is allowed. The size must be a power of 2 and the minimal size is 512 bytes. A small record size increases fragmentation but a larger record sizes has less overhead and less fragmentation at the cost of more memory and larger i/o operations. For example, if you request 16 KiB but your record size is 1MiB you will saturate the bandwidth very early, however if you know that your payload is always larger than 1MiB you use a record size of 512, you will

probably saturate your IOPS earlier. Also records are loaded into memory, especially when dealing with concurrency, and therefore increase the memory consumption. Please note that the record size is fixed per pool, which means that the record size cannot be changed after the creation of a repository.

2.1.6 FR-06: resilience

Resilience is a major design goal. The legacy WIZ has a packed format, using a variable node size up to a configurable maximum size. This is basically a good thing but has also the major drawbacks of an added overhead for length parsing, efficient buffer recycling and most importantly when recovering from bit failures. As soon as a damage in the length header occurs, it is impossible to calculate the beginning of the next node. To mitigate this issue one can insert boundary restart markers and checksums to skip broken parts and find valid points. However all of these workarounds increase complexity and decrease performance. However, as in legacy WIZ a node may have a variable size but it always fits and aligns with a record. In a potential streaming scenario with an unknown offset, a boundary node can be inserted to detect the correct record offset.

The definition of a fixed record size and boundary nodes allows efficient recovery mechanisms. When using nodes with checksums, it will be very simple to restore nodes from an arbitrarily damaged storage. This certainly does not mean that actual stored data can be read, especially if the data is fragmented accross multiple records. This can be improved a lot when using nodes with a block chain data structure instead of using offset pointers. In block chain data structures, the role of a pointer is replaced with a unique hash of the referenced content. Using this technique and saving redundant nodes, makes it an easy task to recover a damaged storage.

2.2 non-functional requirements

2.2.1 NFR-01: mount speed

The time to open a storage must be constant, independent of how many entries are contained or how large those entries are. This rule does not apply if the repository is located on a storage system without random access, like a simple ftp server.

2.2.2 NFR-02: open files

Open files are a very limited resource and are usually restricted to a few hundred per process. The library and command line must be capable of handling an arbitrary amount of open storage files by using a custom file handle pooling.

2.2.3 NFR-03: memory consumption

Repositories can grow very large, right into the range of hundreds of terabytes. Also a process may open thousands of repositories at once. So the memory consumption

must be defacto constant independent of how many repositories are open or how large a repository is or how large a single entry is. However there must be options to increase the memory consumption to trade of resources against performance, to avoid hitting the I/O subsystem when required.

2.3 On-disk format specification

2.3.1 Node

Fields

Id	Discriminator	

2.3.2 VDevLabel

A VDevLabel is the first node in the first block. It contains all information related to the entire vdev, like the block size. A VDevLabel is usually not updated and not part of any copy-on-write semantics. When updated, it is entirely overwritten. For recovery purposes the VDevLabel is repeated at block 16, if there are more than 16 blocks available. Fields

Magic	[8]byte'w','i','z','b','l','o','c','k'	The 8 magic bytes identifying a vdev block file.
Version	uint321	The version is a 4 byte integer. Future versions
CustomMagic	[32]byte	The custom magic flag is an arbitrary defined 3
PoolId	[32]byte	The unique 32 byte id of the pool in which this
Id	uint32	There may be 2^32 vdevs in a pool and each or
Blocksize	uint32	An unsigned 4 byte integer denoting the block s
HashAlgorithm	HashAlgorithm	Within a vdev the HashAlgorithm is a one-size-

2.3.3 HashAlgorithm

A HashAlgorithm is always 32 byte long but may be calculated by different algorithms. You should always choose a cryptographically strong algorithm. Fields

2.3.4 Hash

A Hash is always 32 byte long but may be calculated by different algorithms. Fields

2.3.5 TxReference

A TxReference is used in the TxRing and refers to a transaction node. Fields

Ptr	NPtr	
Checksum	Hash('nptr'+Ptr)	The checksum of this node.

2.3.6 NPtr

A NPtr is a pointer to a node within a vdev. The block number is Offset Fields

VDev	uint32	vdev denotes the id of the referenced vdev.
Offset	uint64	The absolute offset of the node within a vdev. We do not address blocks directly.

2.3.7 TxRing

The TxRing data structure is a ring buffer which refers to the latest valid transaction roots. The entry with the highest transaction number and a valid checksum points to the root of the valid state of the storage. Fields

Transactions	[16]TxReference	