Wodan

a pure OCaml, flash-aware filesystem library

Gabriel de Perthuis – OCaml Labs

Problems and motivation

Motivation: MirageOS

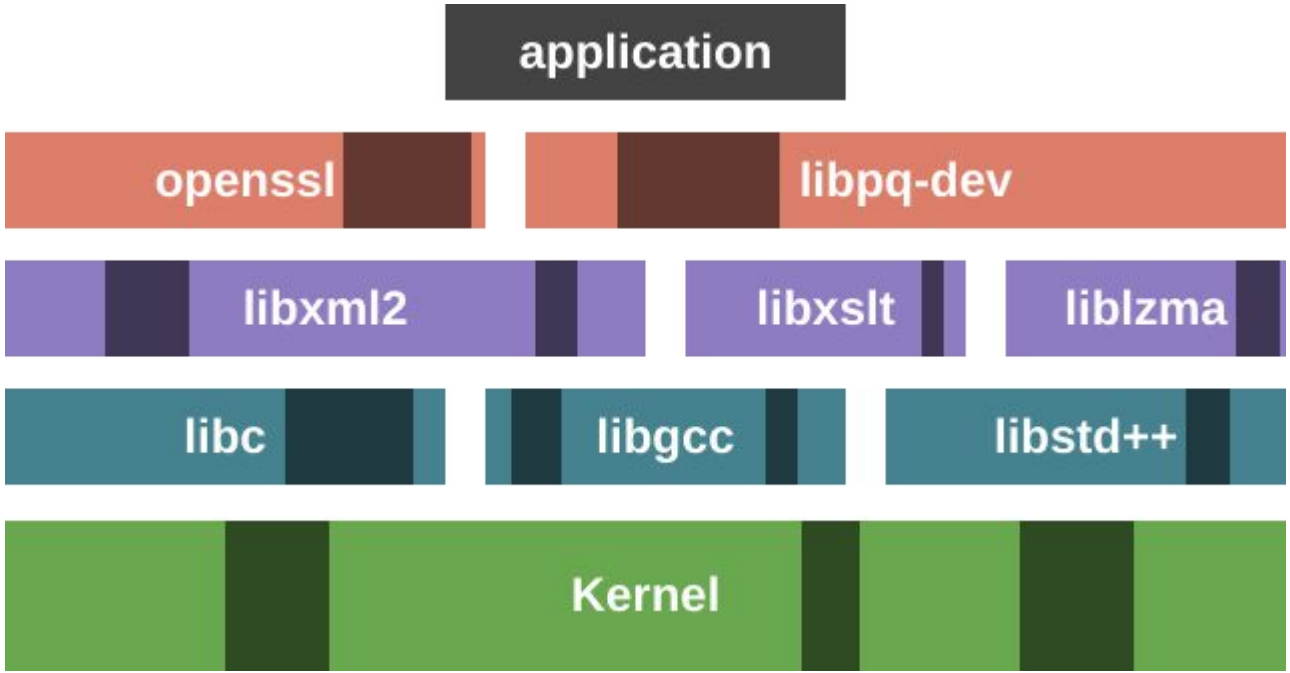
MirageOS is a library operating system written in OCaml. Mirage builds unikernels.

Operating system components are linked as libraries.

More tuned, more secure.

Put control: layout details, performance trade-offs in the library user’s hands

Unikernels: slicing things up



Storage challenges

Present a usable API, on top of hardware with complex characteristics

Usable: sufficiently expressive but also efficient.

Hardware: disks, flash, SSDs, persistent memory / NVM, hybrid devices…

Storage challenges: general

Atomicity

Reordering

Bit rot (durability)

Storage challenges: specific

A hardware primer

	Pros	Cons
Disks	Cheap, large	Seeks
Flash	Fast	Erase cycles, endurance
SSDs	Fast	FTL complexity, write amplification

Design and solutions

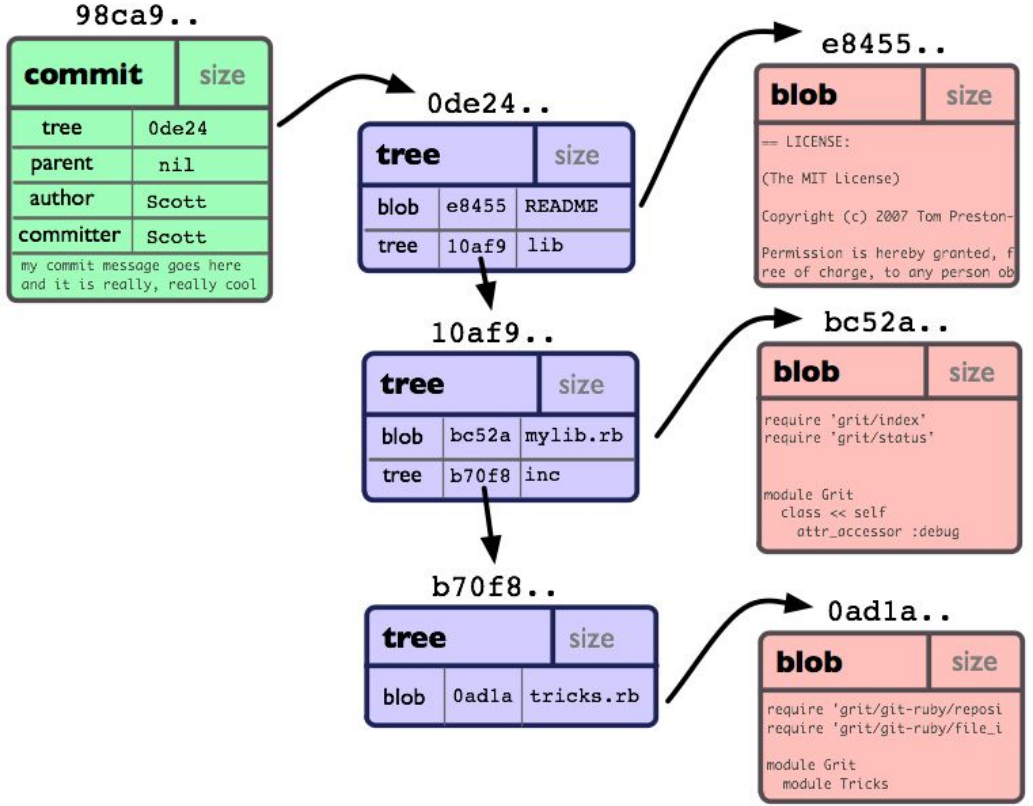
Usability and expressiveness

Irmin can provide a high-level, expressive API

Irmin is Git-like: objects, trees, commits and references. Functional updates.

Wodan provides fixed-length keys and bounded-length values.
On the low end of the expressiveness scale, but sufficiently expressive to layer a chunking layer and then Irmin on top without affecting performance. With some tricks, this is also sufficient to express a variable-length key API (used by some of the Irmin internals).

Irmin’s model



Layering Irmin

<table>
<tr><td colspan="2">Irmin blobs</td><td>Irmin meta</td><td>Irmin refs</td></tr>
<tr><td colspan="3">Chunking</td><td>Hashing (+other tricks)</td></tr>
<tr><td colspan="4">Wodan</td></tr>
</table>

The key-value layer

Present a key-value API on top of an efficient, functional, wide-branching tree structure.

Wide-branching trees minimize seeks, and a functional structure won’t need rewrites, while being atomic and non-amplifying. This addresses the characteristics of disks and flash.

Concretely, use hitchhiker trees, an optimal implementation of a functional wide-branching tree.

Hitchhiker trees; B^{ε}-trees

Hitchhiker trees are the functional version of B^{ε}-trees

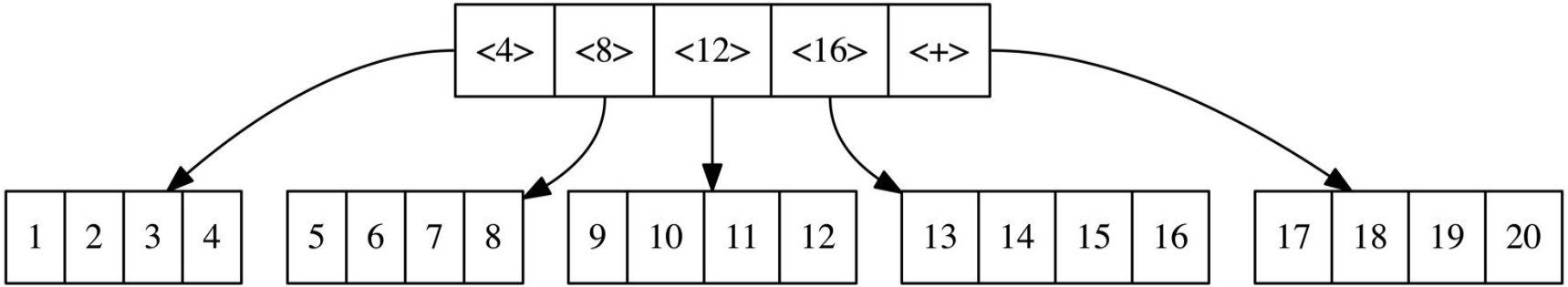
B^{ε}-trees derive from B+trees

Instead of containing just children pointers (B+tree parent nodes) or data (B+tree leaves), each node contains both. A node contains reserved space for buffered data and children pointers.

A new tree operation: spilling

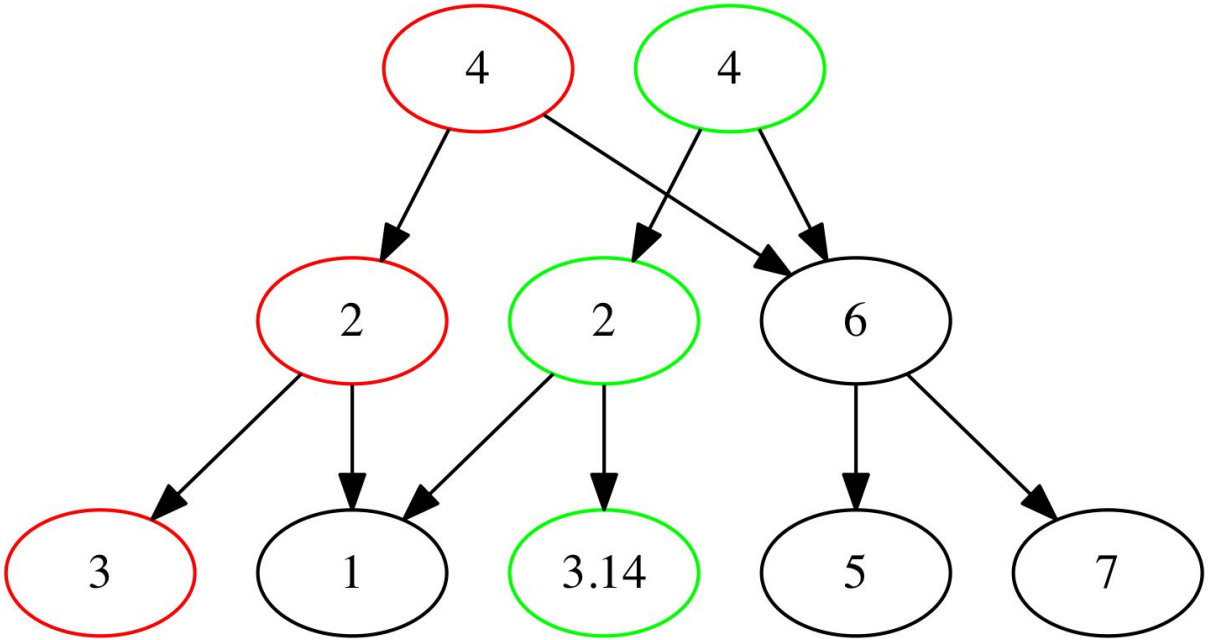
Advantage: writes become much cheaper and closer to the root. In the functional version, no path dirtying. Cost: depth hit of at most a constant factor ε.

B+trees



Functional trees

Path dirtying

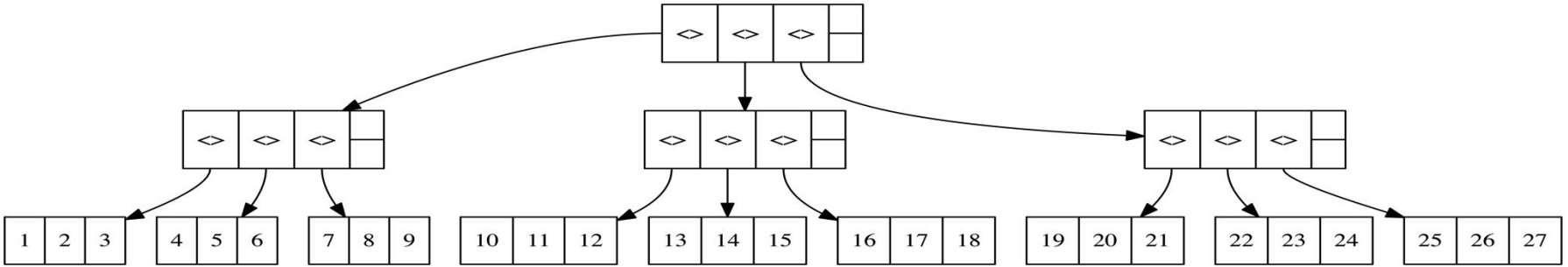


Hitchhiker tree node layout

Meta: type, gen	Key, data len, data	Key, data len, data	…	Key, child loc	Key, child loc	CRC32C

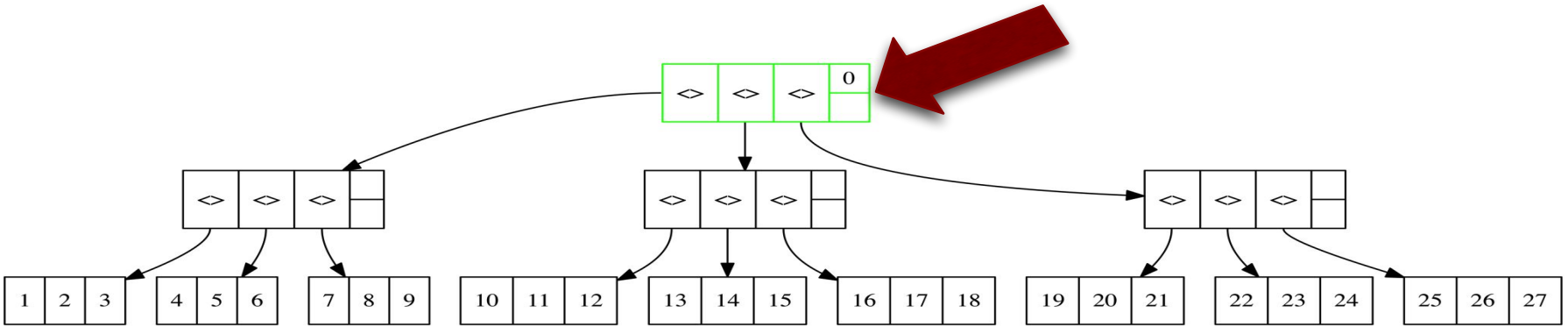
A red zone is used to separate the data and child arrays.

B^{ε}-tree example



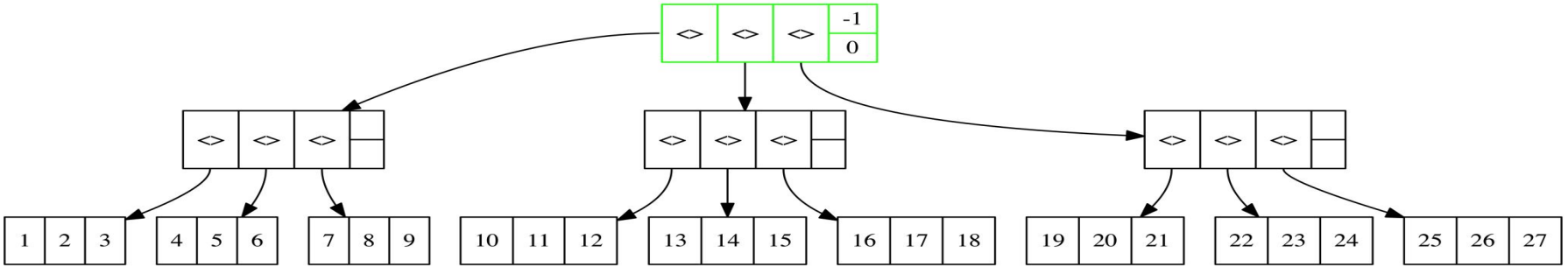
B^{ε}-tree example: insertion

Inserting 0



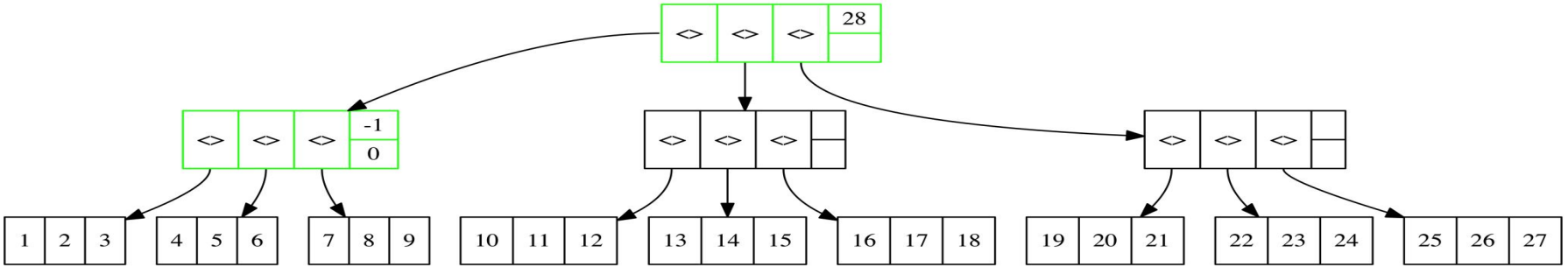
B^{ε}-tree example: insertion

Inserting -1



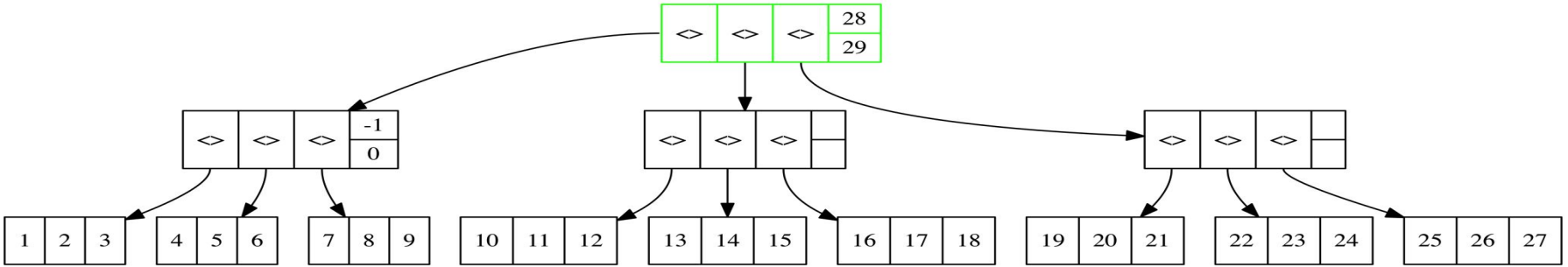
B^{ε}-tree example: insertion

Inserting 28
(spill)

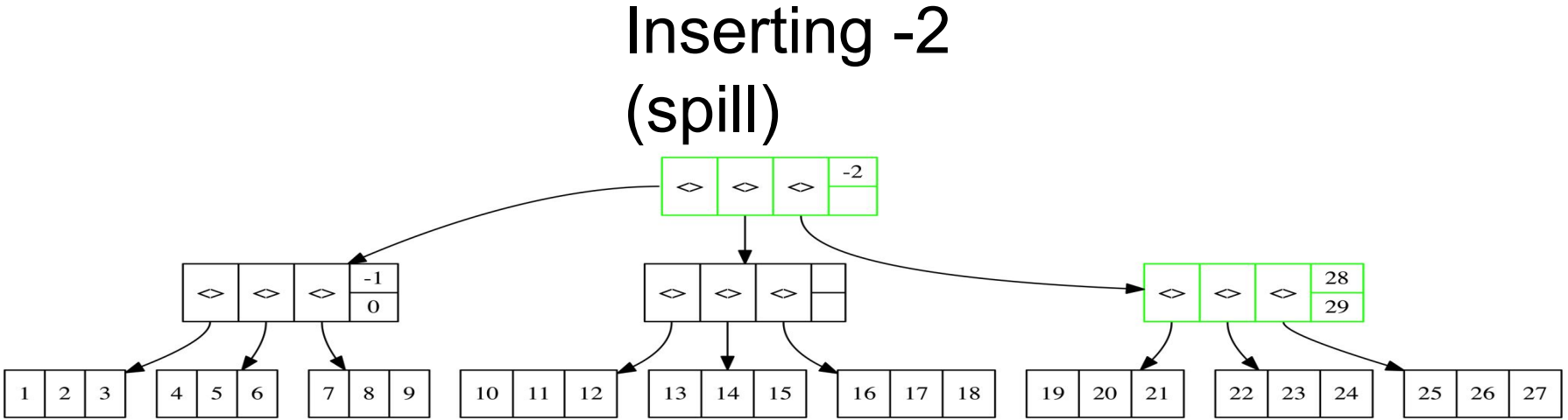


B^{ε}-tree example: insertion

Inserting 29

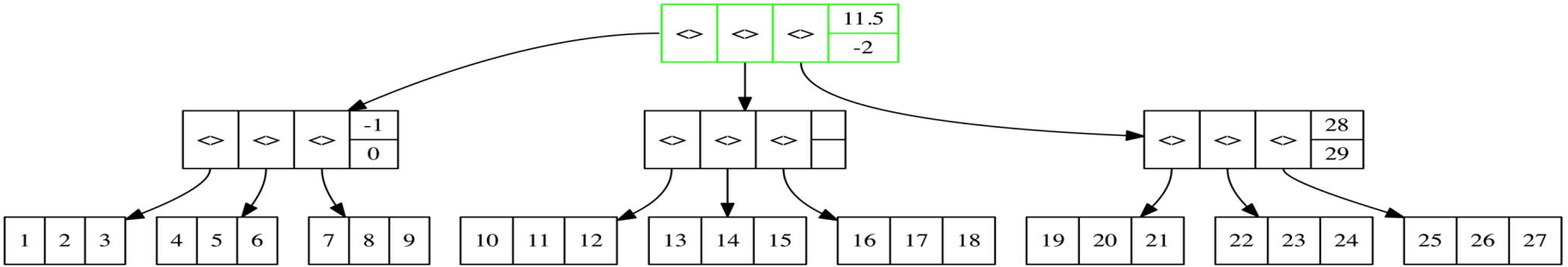


B^{ε}-tree example: insertion



B^{ε}-tree example: insertion

Inserting 11.5



The root block challenge

If the root node is at a fixed place, it gets high-frequency updates. The flash wears out. Updates are not functional.

So don’t require a fixed location.

Store generation numbers on nodes.

Locate the root block with a bisection.

General layout; bisection

Super-block	non-root	2	non-root	4	8	1	non-root

Layout control

The block size should match the size of erase blocks (between 256k and 4M).

Prevents write amplification

- smaller blocks would be write amplified by the flash translation layer
- larger blocks would amplify writes at the filesystem level by writing more data than necessary each time the data is checkpointed.

Key size is application controlled as well.

A custom allocator will also be added, for sequential file-backed devices.

These settings appear on an OCaml functor parameter.

Semantics control

Tombstones for empty data

In the future: upserts?

Operation control

Manually triggered at the KV layer:

- Flushing
- Garbage collection

Consistency

The backing device could disappear at any point.

Prevent torn writes and detect corruption in the backing device with a per-block CRC32C.

Prevent out of order writes: send a barrier prior to writing a root block, so that child block references are always valid.

Resiliency

Optionally, everything is scanned at mount time. Bit rot is detected early. Also builds a free space map for the allocator.

Optimised default: build the free space map without reading leaf nodes, by tracking tree depth at the root.

Testing

Insert random data.

Periodically close and reopen.

Run under AFL, American Fuzzy Lop. Maximises branch coverage.

Prepared image + fuzzing + CRC fixup + single insert.

Second reference implementation.

Use cases

Unikernels in more places

- Short-lived stateful servers
- Data-intensive servers

Irmin

- Tezos, Datakit (Docker), CueKeeper, Jitsu, Canopy

Git storage

Redis like

Future work

Stdlib Map pull requests: start_iter_at

Better in-memory indexing of a single node

Upserts

Concurrency

Javascript port

References

What every programmer should know about solid state drives — http://codecapsule.com/2014/02/12/coding-for-ssds-part-6-a-summary-what-every-programmer-should-know-about-solid-state-drives/

B^{ε}-trees — Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries.

Hitchhiker trees — http://www.slideshare.net/DavidGreenberg7/hitchhiker-trees-strangeloop-2016

American Fuzzy Lop — http://lcamtuf.coredump.cx/afl/

Questions?

Thank you for your attention.

Go use it! https://github.com/g2p/wodan

Future work

Upserts — take advantage of the functorial interface

CLI tool

wodan format

wodan dump

wodan restore

PRs (for spilling). Explain the reason for the next Map PR.

Compare with the Rust impl

Example with Irmin, Git example?

Or just code snippets