ZNSLSM: Design and Implementation of NVMe Zoned Namespace (ZNS) Optimised LSM KV Store

Krijn Doekemeijer
Contact: k.doekemeijer@student.vu.nl
Github: https://github.com/Krien
Supervisor: Animesh Trivedi

UNIVERSITY OF AMSTERDAM

The amount of data is exploding

The world demands more every year:

- IDC expects **175 zettabytes** of data by 2025!
- Performance requirements increase

A lot of challenges for data:

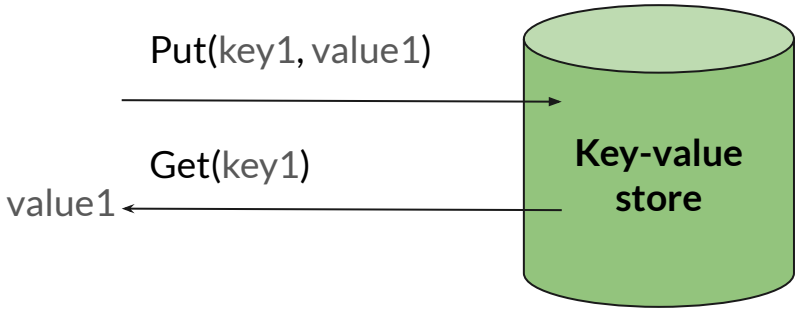
- **How**?
- **Where**?
- **Fast**, yet **green…**?
- …

This thesis is about one common type of data storage:
key-value stores + SSDs

https://en.wikipedia.org/wiki/Container_ship#/media/File:CSCL_Globe_arriving_at_Felixstowe,_United_Kingdom.jpg

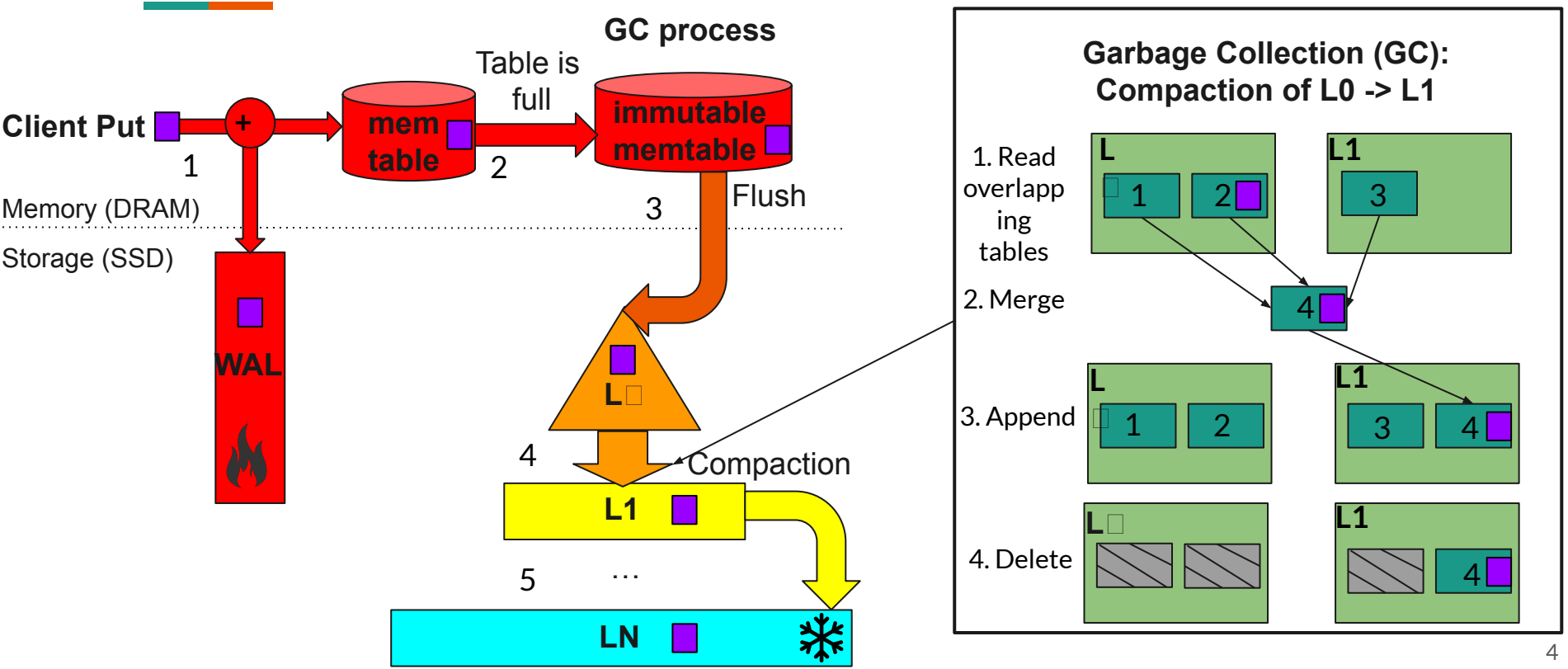
How to store? - Key-value stores

- Commonly used
- Flexible
- Few operations needed
  - Put
  - Get
- Optimisable for many use-cases
  - Also as relational database backend!



Used by:

Log-structured merge-tree key-value stores



Where to store? - Solid State Drives (SSD)

Why?

- Common 💾
- Fast I/O
- Affordable 💰
- More green than HDDs 🍃

Problem! Wear Leveling:

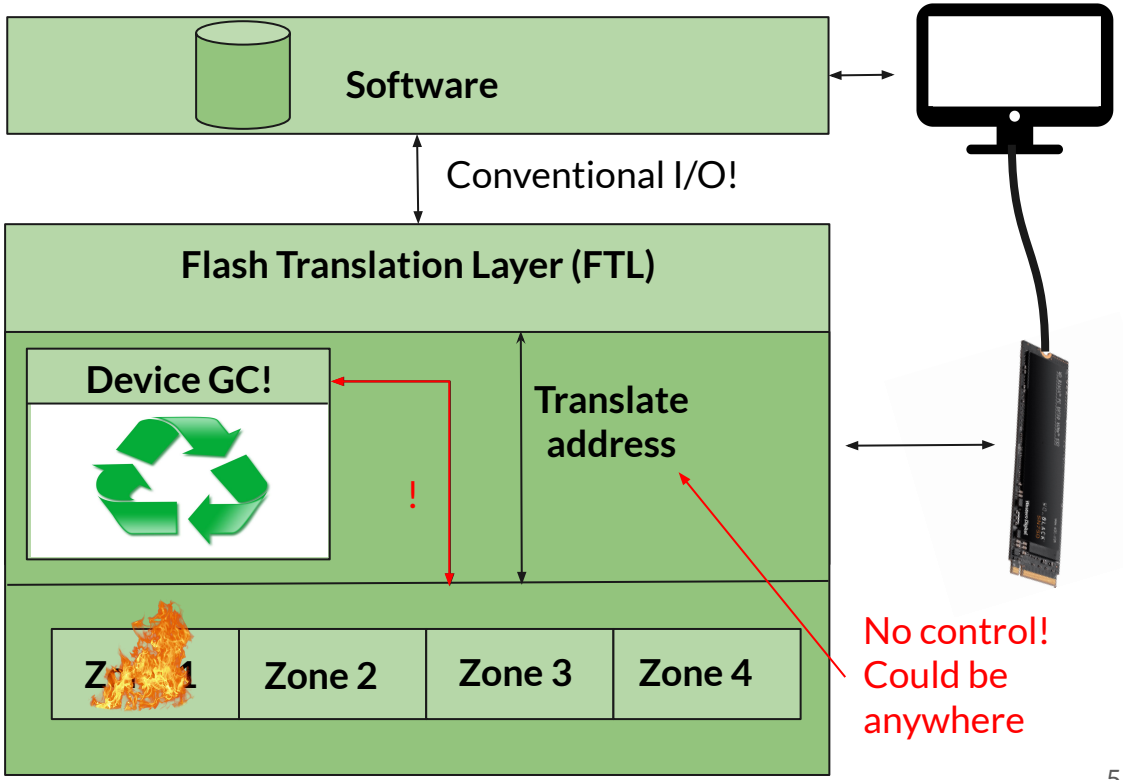
- I/O Harms device!
- So using is paying 💰🍃

Flash idiosyncrasies…:

- Three types of internal operations:
  - Read
  - **Expensive** write (but no overwrite!)
  - Even more **expensive** erase
- Divided into “erase blocks”/“zones”

FTL simplifies logic

- Conventional I/O



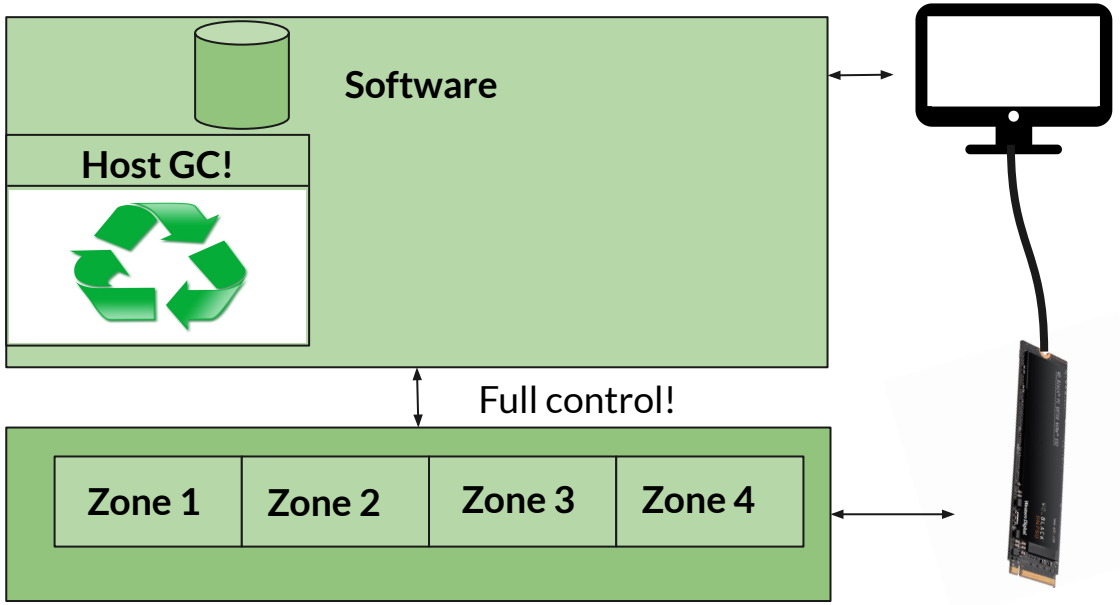
Meet Zoned Namespace (ZNS) SSDs

Full control:

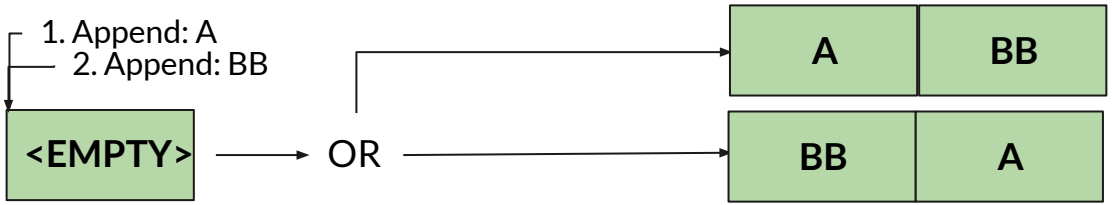
- All I/O is now be done by the host
- GC must be done manually
- All idiosyncrasies must be dealt with

Advantage:

- Use-case specific garbage collection (**key-value stores!**)
- Limits wear leveling
- Optimisations generalisable

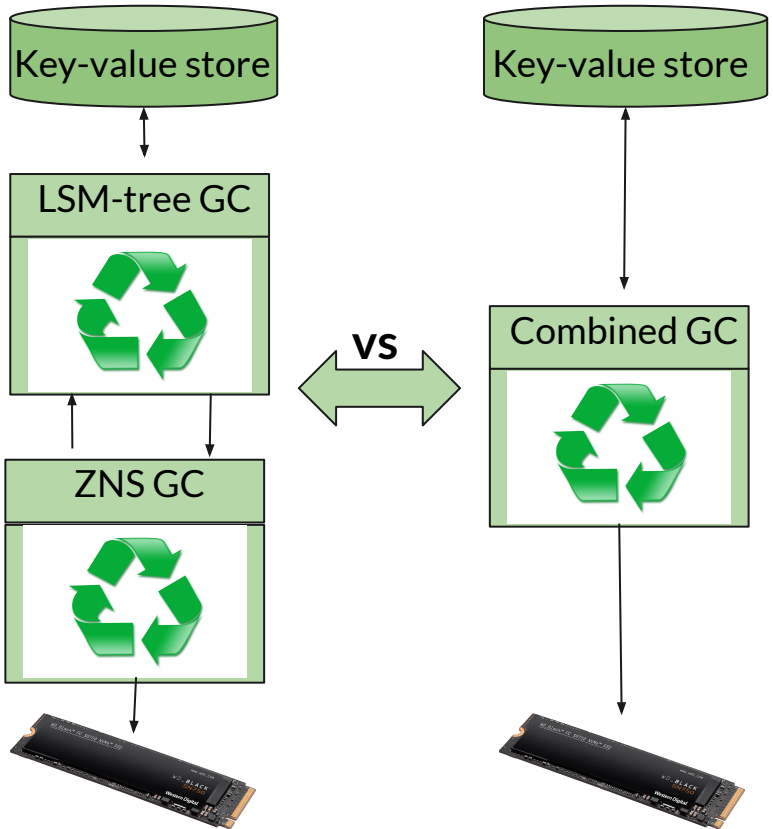


New I/O operation is now available: Append



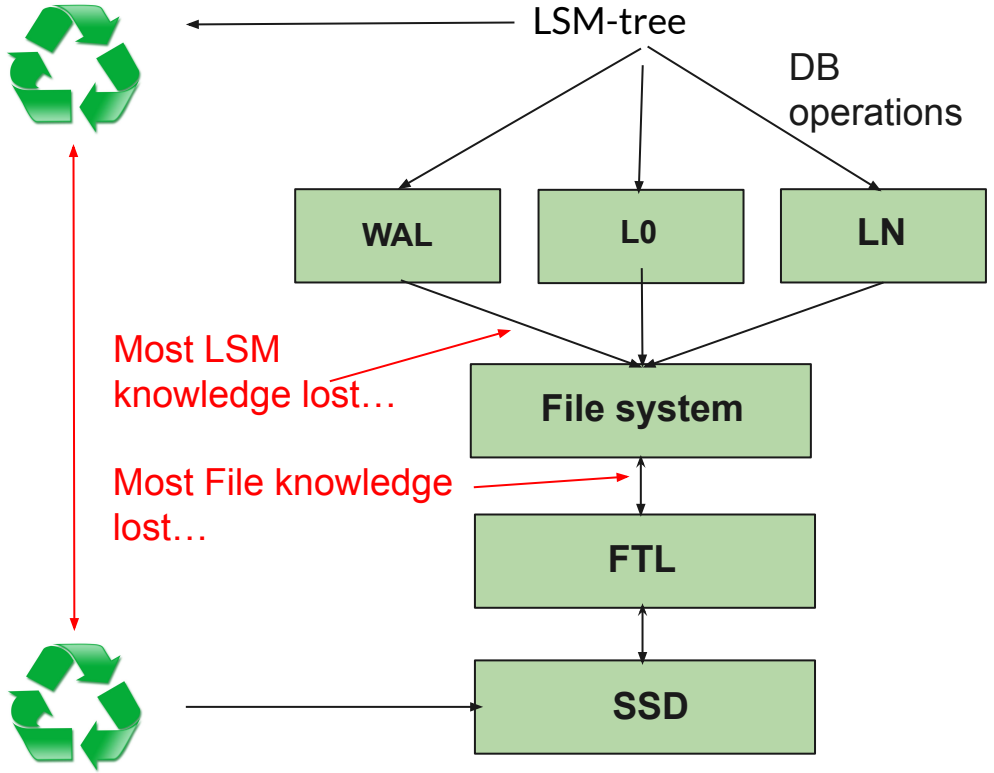
Collaborative Garbage Collection (GC)

- **Both** LSM-trees and ZNS need a GC!
- Relating GC of LSM-tree to GC of ZNS
- What is a natural relation?
  - Two Independent GCs?
  - Just one GC?
- That is what we intend to find out!

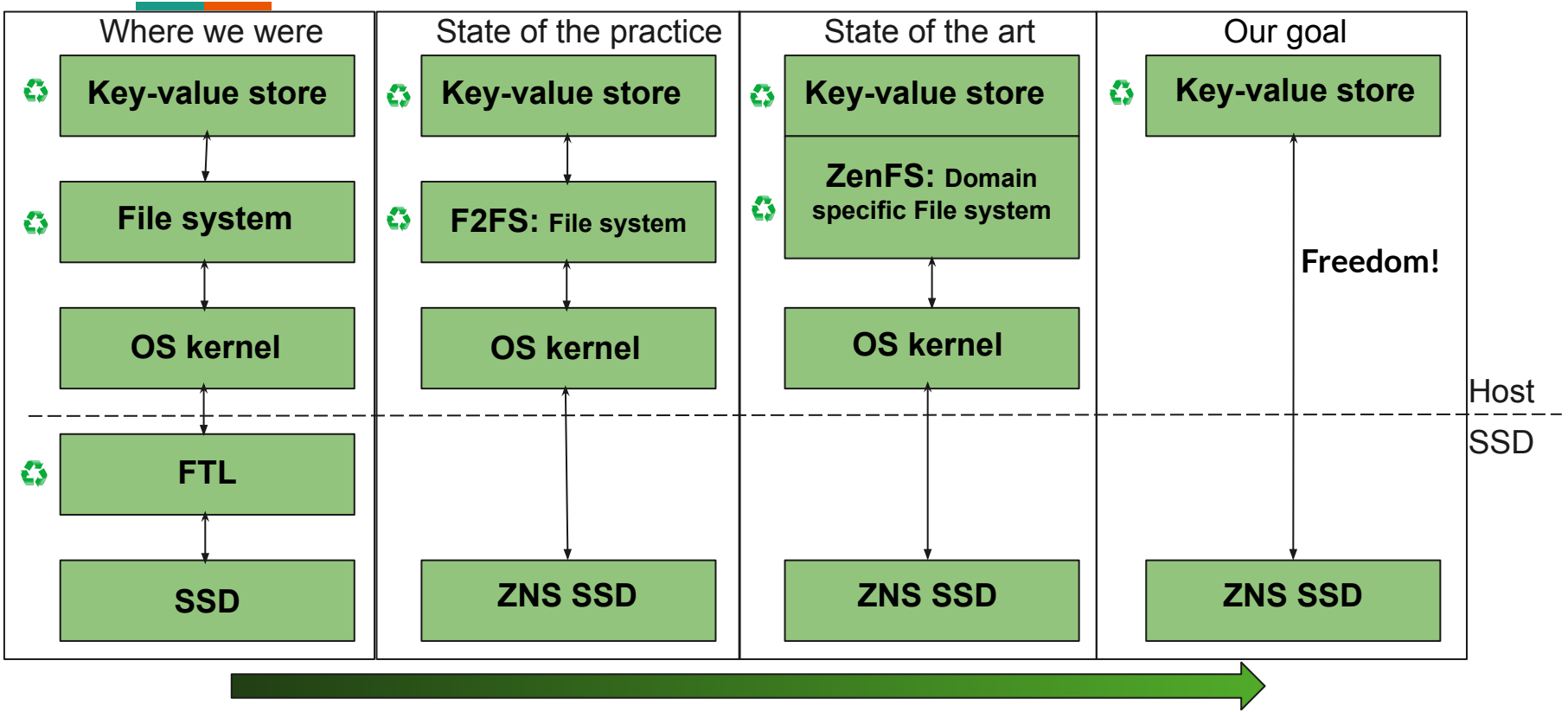


FTL is not the only “layer”

- File system layer
- Each layer leads to another **semantic gap**
- Each layer knows little about the next…
- No clear separation between hot and cold data
  - Even when LSM assumes there is!
- On the picture on the right:
  - WAL and LN data might be put in the same zone!
  - Hot files might be put in same zones as cold files!



Meet the layered world…



Our research goal

How can ZNS be used to co-optimise the garbage collection process of a LSM-tree-based key-value store?

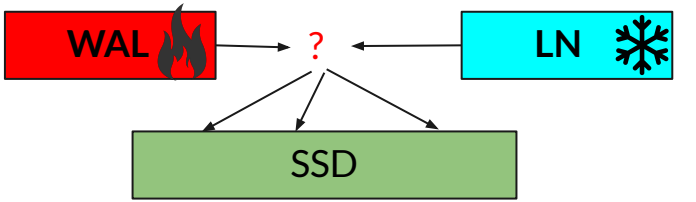
- RQ1: How can ZNS be used to reduce the **write amplification** of LSM-tree garbage collection operations?
- RQ2: How can ZNS be used to reduce the **number of erasures** needed because of garbage collection operations?
- RQ3: Does directly interfacing with storage help with **performance and its predictability**?

Approach

- Modify the state of the art: reuse parts of RocksDB and LevelDB
  - Do **not** reinvent the wheel
- Copy and reuse benchmarking tool of RocksDB: db_bench
- Create all I/O structures ourselves
  - No filesystem!
- Manage the GC process of the store **AND** I/O.
- Get full control of device with: SPDK
- Work entirely in user space for storage, no kernel
  - No layers, no handholding
- Programming languages: C++ and a little C

Three major problems to consider

1. How to properly separate hot and cold data?
   Needed for proper garbage collection



2. How should WALs be designed?
   Also true for L0, LN…


3. What erasure policy to pick?


when to erase ?


ZNSLSM: How do we separate on temperature?

- Each DB component gets its own **“region”**
- The device is split on temperature
- Each **“region”** gets a predefined **“size”**
- No multitenancy by design, the device is ours to use!
- No files, each **“region”** is designed differently
- **Hot** components get **hot** data structures, **cold** components **cold**
  - Internally the components separate again on temperature

Imagine the entire SSD as one big strip

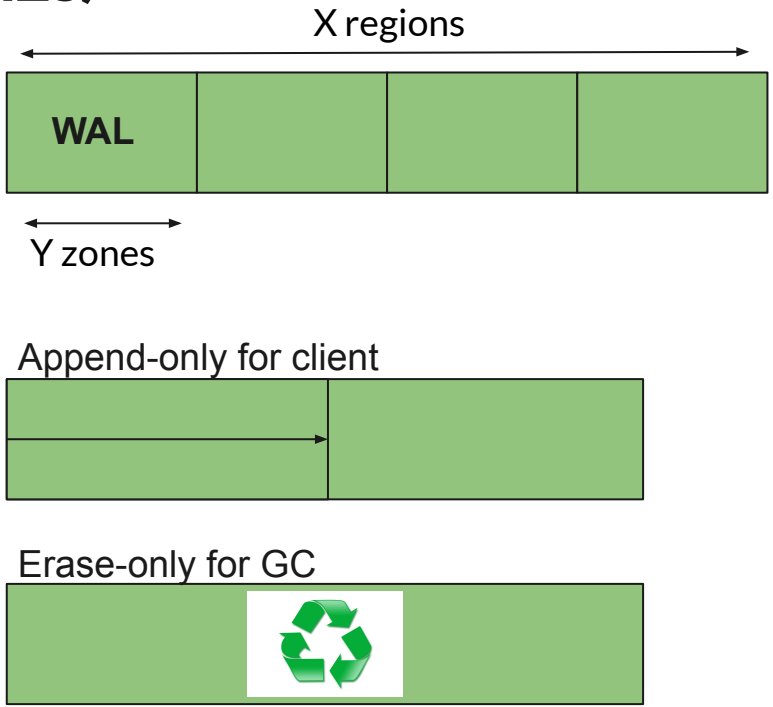
Designing Write-ahead Logs (WALs)

What do we know about a WAL?

- Clone on if in DRAM structure
- Only “used” for redos on restart
- During an active session it is write-only
- Order does not matter
- WALs are small
- Very HOT in number of writes

Design:

- X small WAL regions of each Y zones
- Clients write
- GC erases old WALs lazily
- Parallel writes with append (async)
    - Append allows doing multiple concurrent I/Os to same zone!



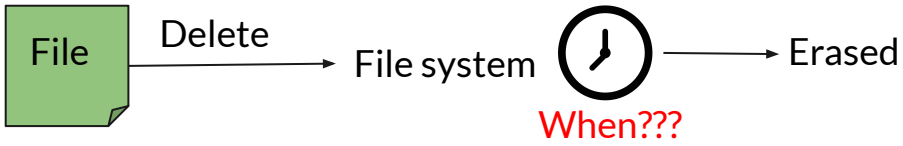
How does our zone erasure strategy work?

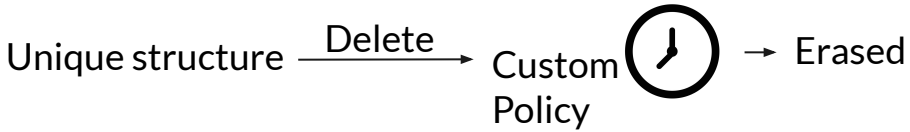
With a standard file system approach:

1. **Key-value store**: “deletes file” (this is just a hint)
2. **File system**: marks file to be ready to deleted
3. **File system**: decides if/when it wants to use the dead file and resets parts of the file.

With the ZNSLSM approach:

1. **Key-value store:** marks zones as “dirty” and ready for reuse.
2. **Key-value store**: decides that it wants to reuse zone for a component and resets zones itself.





Benchmarking

We need to force the **extreme**:

- Ensure that the key-value store is tested on actual **I/O**
- We do not want to test other parts of the key-value store
- Fill the entire device, forcing the **GC** to kick in at least once
- Cause overwrites! Do mostly **updates** of key-value pairs. The stuff that GC hates…

Compare with the alternatives!

- Normal filesystem: ZNS + F2FS + RocksDB,
- Domain specific filesystem: ZNS + ZenFS + RocksDB,
- No filesystem: ZNS + ZNSLSM <- **ours!**

Measure **GC effects (as best as we can)**:

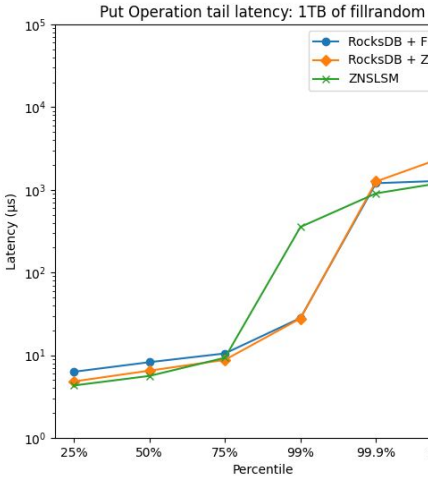
- Relates to RQ1: bytes written
- Relates to RQ2: resets issued
- Relates to RQ3: latency stability

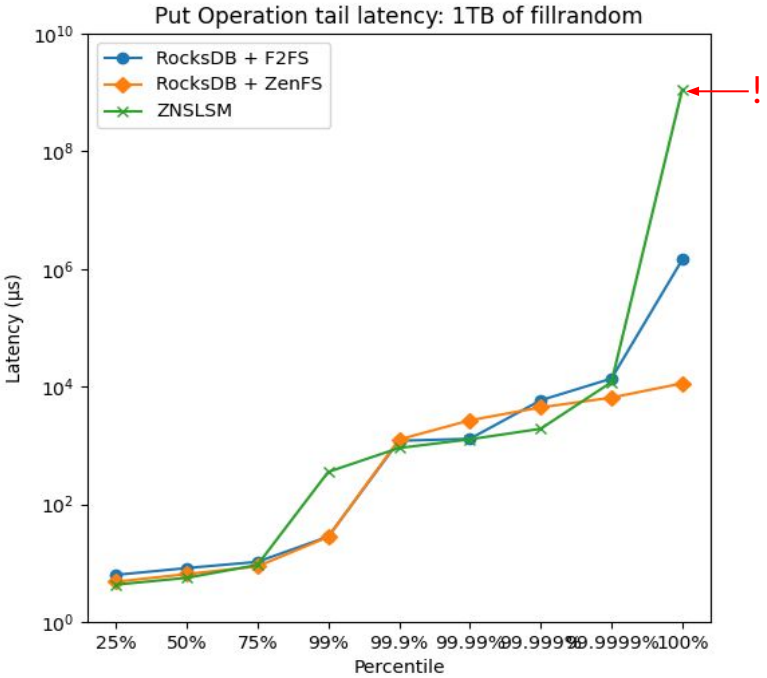
We run in 3 steps (3TB is available):

Benchmark Workload	**Finished when?**
Fill random key-value pairs	1 TB of pairs are written
Overwrite key-value pairs	1 TB of pairs are overwritten
Read while writing to test latency	1 extra hour of I/O has completed

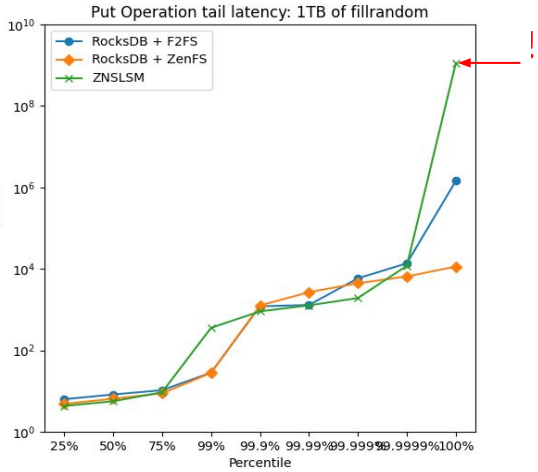
Results: Put operation Tail latencies (1TB of fillrandom)

IN PROGRESS





What happens after 99.9999%?

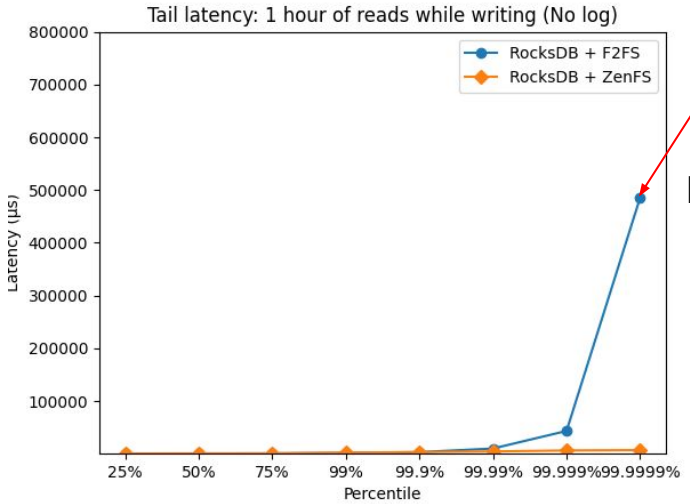


ZNSLSM does > 34 hours over 1 TB. F2FS and ZenFS less than 5. A few operations are everything…

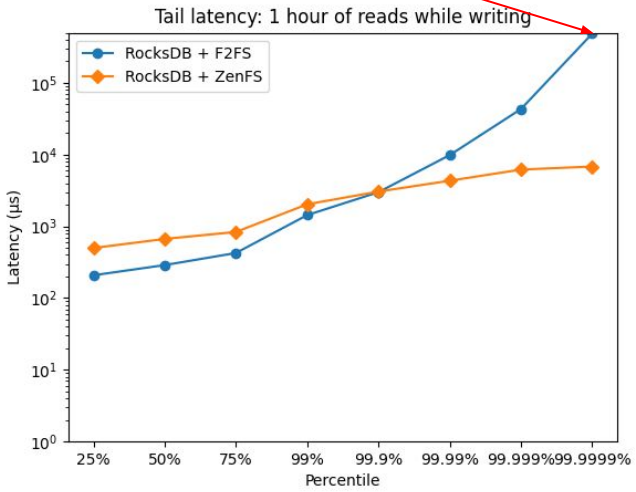
Results 2: Tail latencies (1 hour of reads during writes)

Acceptable?

IN PROGRESS

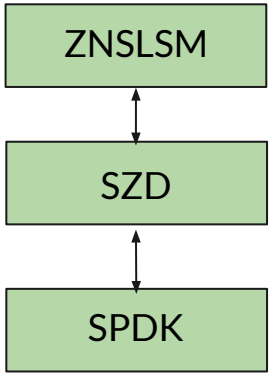


Lets move to log scale…



End product

- We have created:
  - A simple ZNS interface on top of SPDK (**SZD**)
    - Reusable for other SPDK ZNS projects
    - https://github.com/Krien/SimpleZNSDevice
  - A database based on RocksDB and LevelDB using this interface (ZNSLSM)
    - https://github.com/Krien/znslsm (still private for now...)
  - A set of benchmarking scripts based on ZenFS benchmarking scripts
  - A report addressing our experience with creating such a database (in progress!)
- What this project means for key-value stores:
  - With a few month we are able to come close to the state of the art
  - Time to consider building a full key-value store in production with this method as well
  - Reduction of wear-levelling and unstable latency is achievable



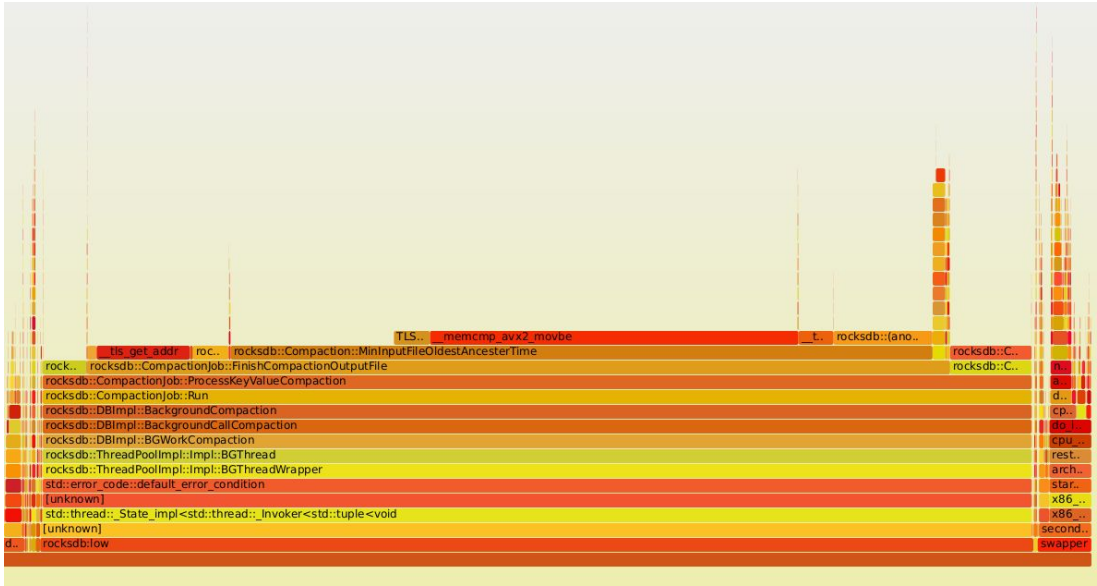
Revisiting the research question

How can ZNS be used to co-optimise the garbage collection process of a LSM-tree-based key-value store?

- RQ1: How can ZNS be used to reduce the write amplification of LSM-tree garbage collection operations?
  - ZenFS has shown to be able to reduce write amplification, therefore we expect similar results for ZNSLSM. Especially because of proper hot-cold separation.
- RQ2: How can ZNS be used to reduce the number of erasures needed because of garbage collection operations?
  - ZenFS mentioned that their design is able to reduce the number of erasures, therefore we expect similar results should be achievable for ZNSLSM.
- RQ3: Does directly interfacing with storage help with performance and its predictability?
  - ZenFS showcases that it achieves more stable latency than when F2FS, especially tail latency is more stable. Therefore, directly interfacing with the storage has an effect. An effect that we also expect to see in our design after we make some further changes.

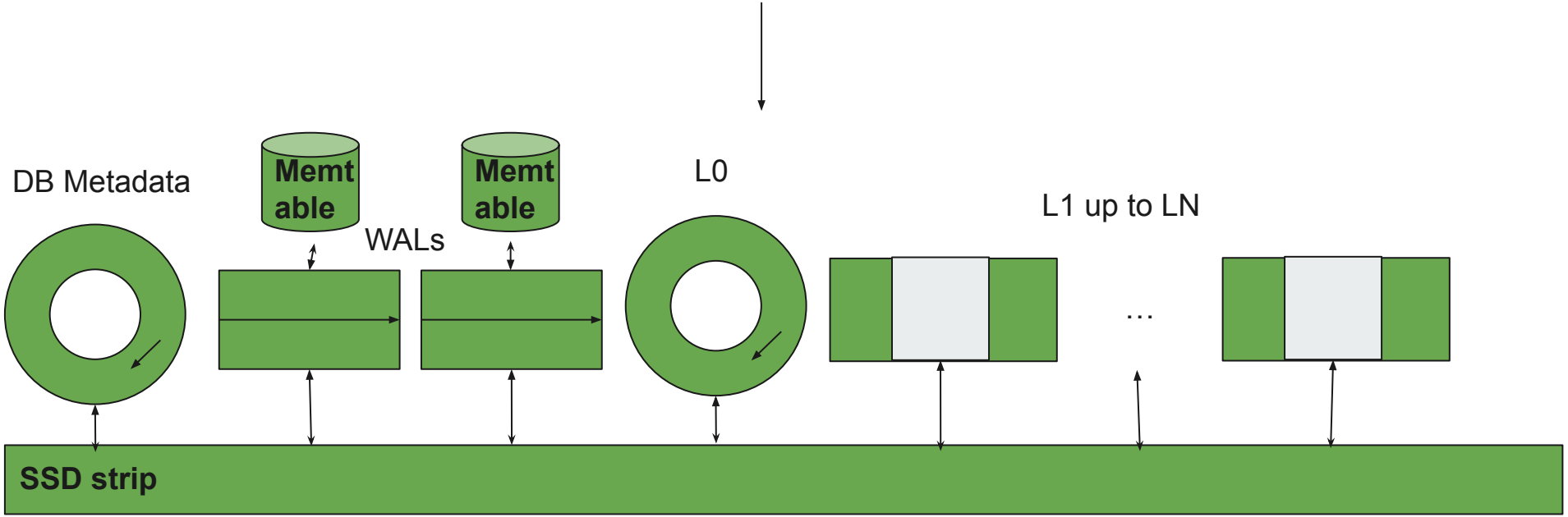
Experience of building a key-value store

- Work in a lot of places…
  - New interfaces,
    requires recreating everything yourself
    - Debug tooling…
    - CLIs…
  - Setting up the benchmarking environment is painful
    - Consistency
    - Getting modern tooling
    - Benchmarking, benchmarking, benchmarking…
    - Created tooling for ZNS that can be reused
  - Running 2 TB benchmarks themselves
    - Takes days…
    - Our DB needs to survive a lot!
      We can not create a simple PoC…
    - Need to use VALID parameters for
      F2FS and ZenFS as well
  - Guaranteeing stability
    - Persistence?
    - Deletes?
    - Persistent deletes!?



Any questions?

For example: “How should ZNSLSM be named?”



Related work

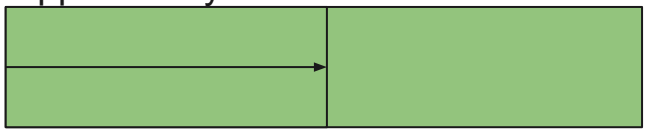
- As far as we know this is the **first** direct key-value store on top of ZNS.
- Filesystems do exist in some regard:
  - ZenFS, a file system made specifically for RocksDB and ZNS SSDs.
  - ZoneFS, is a general file system made for ZNS SSDs. This can also be used on RocksDB. NOT TESTED.
- There does exist an LSM-tree for ZNS SSD, but it is made for general GC policies, not for key-value stores.
- There exist various works that describe that building a key-value store is viable, such as “Don’t be a Zonehead” by Stavrinos et al., “ZNS: Avoiding the Block Interface Tax for Flash-based SSDs” by Bjørling et al.
- Purandara et al. also come with various hints and design decisions for ZNS SSDs, also for key-value stores. This is a good source for design decisions. The paper is “Append is Near: Log-based Data Management on ZNS SSDs”.

DB parts get unique I/O structures, not generic files

For example, consider a WAL (*not necessary yet to know what it is!*)

Generic file support for appends, random writes/reads,...

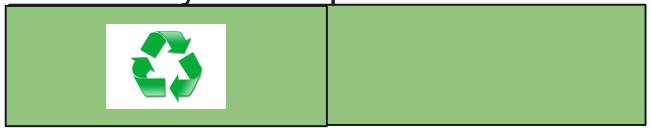
append-only for client



No control on when the file is physically deleted…

Marked for deletion?

Erase-only for GC process



External meta data structures. 2 writes…

Inode → **Data**

No external meta data, just 1 write!

metadata entry 1	**data entry 1**

What can a key-value store be used for

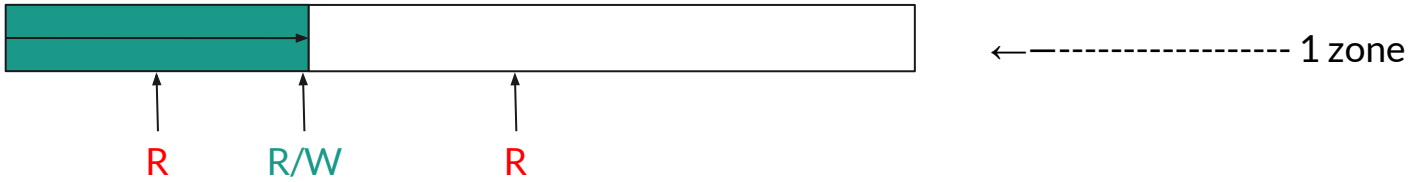
- Gameservices such as Xbox live
- Distributed databases such as ZippyDB at Meta
- Redis, Memcached, Riak…
- Various services such as AWS: DynamoDB
  - Might scale better than relational databases
- Graph query engines such as Dragon
- Low latency-queries, such as at Netflix

When to use key-value stores

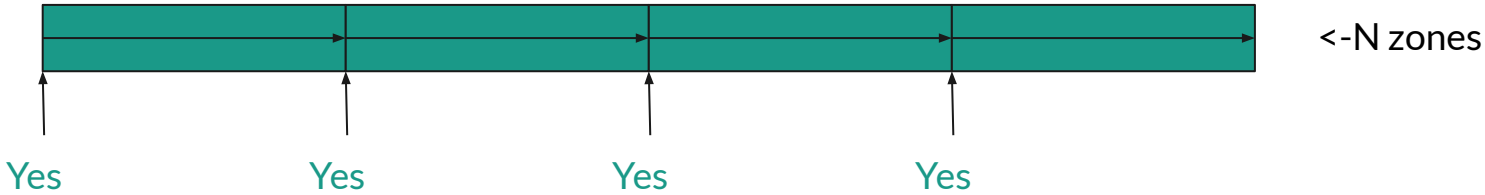
- I refer to my literature study
- Latency matters, fast reads and writes
  - Especially for in-memory key-value stores!
- The consistency model differs
  - Some key-value stores have a weaker consistency model, this might be all that is needed
- Data is highly dynamic, many optional parts of a value “JSON”
  - Think about rich documents, instead of predefined objects
- When the workload fits simple key-value queries?
  - We do not always need filters
- OLTP is a good use case
- Simple, allowing other applications/DBs to add what they want themselves. Minimalism!

ZNS (more detailed explanation)

Can only append to a zone, so no random writes, but read everywhere



However can append to multiple zones (limited to a number of active zones)

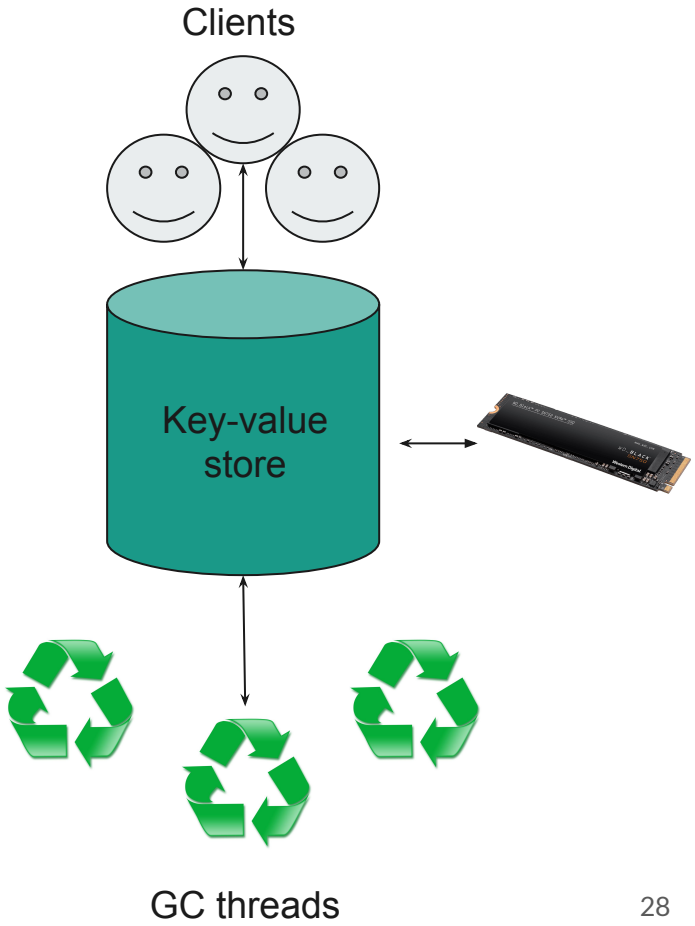


Garbage Collection

- Used by the **Flash Translation Layer (FTL)** to support overwrites…
- Process of managing “garbage”
- Removing stale data
- Moving data to new locations

The issue:

- Write Amplification!
  - Writing more physical data than logically requested
  - Remember last example…
- Consumes I/O, CPU and memory
  - typically by a few background threads
- Hampers latency of clients
  - Contention issue



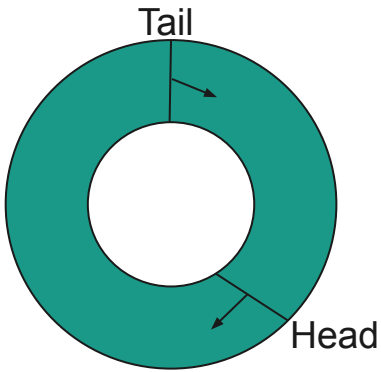
Designing of L0

L0 properties:

- Essentially an append-only queue
- Clients only read
- Background-thread writes to head and erases from tail
- Data is never rewritten!

Design:

- Circular log like an Ouroboros
- Data is written to head and erased from tail
- Data is read from the body
- The head wraps around



https://en.wikipedia.org/wiki/Ouroboros#/media/File:Serpiente_alquimica.jpg

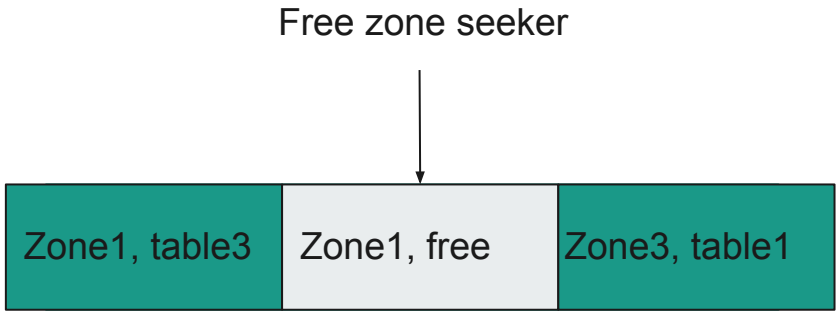
Designing LN

LN properties:

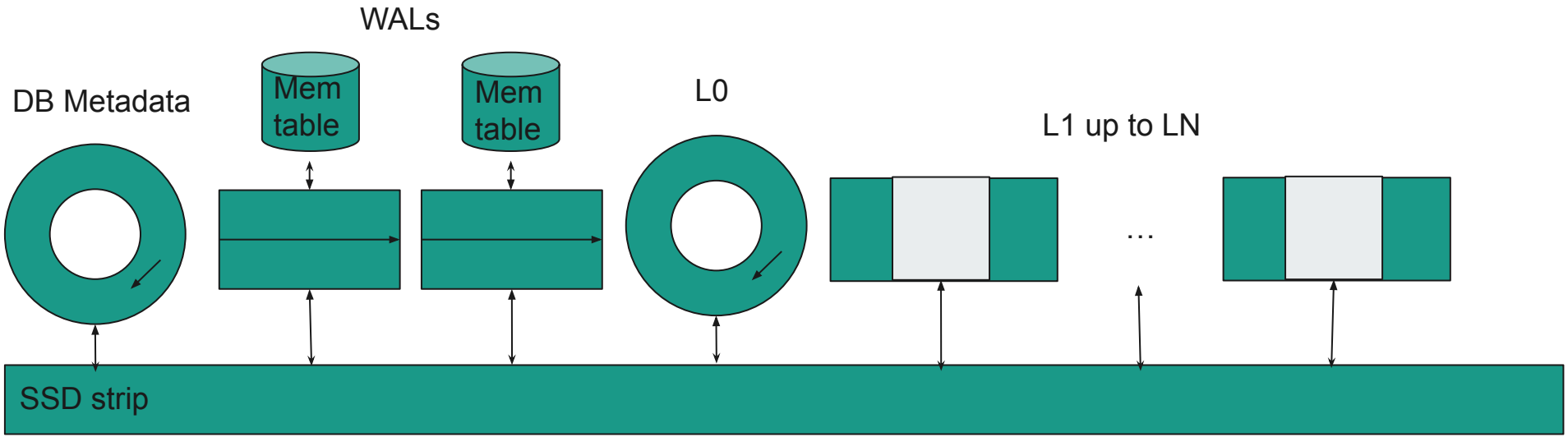
- Written to infrequently
- Large tables
- Tables need to be overwritten/merged

Fragmented log:

- Each table has its own zones (no unnecessary resets)
- As tables are large, unused data should be little…
- Metadata of zone collections is stored separately
- Read from collection, write to collection and reset collection

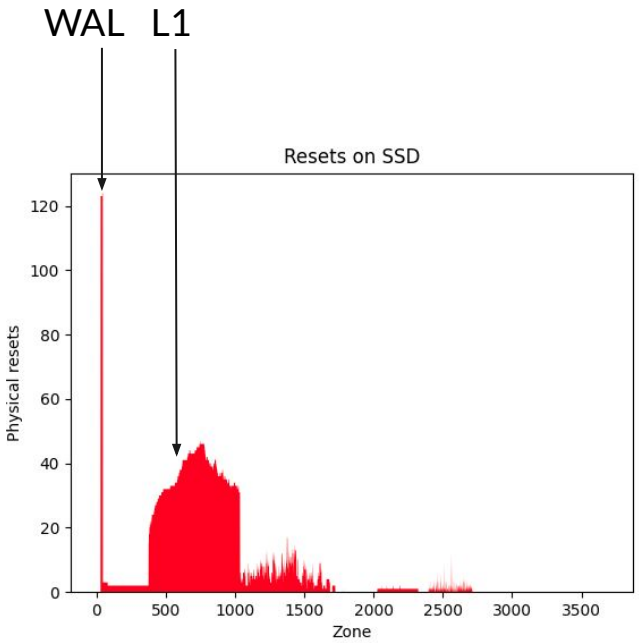


Final design



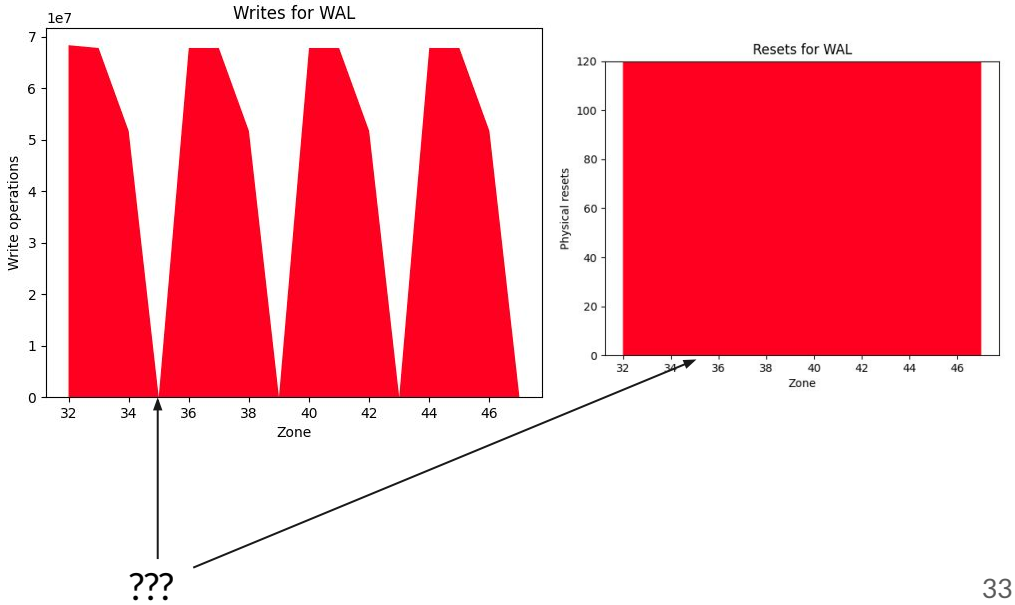
Results: Zone metrics

- Our design is a PoC,
  but we can immediately see the issue here…
- It is ok to separate on Hotness, but if a structure is too hot, it will burn up the SSD in one spot.
- WALs **NEED** to rotate
- LN **should** even out, not stay in the beginning.
- Solution:
  - Add L1 .. LN and WAL to same handout routine of zones.



Raw metrics: Diagnostics

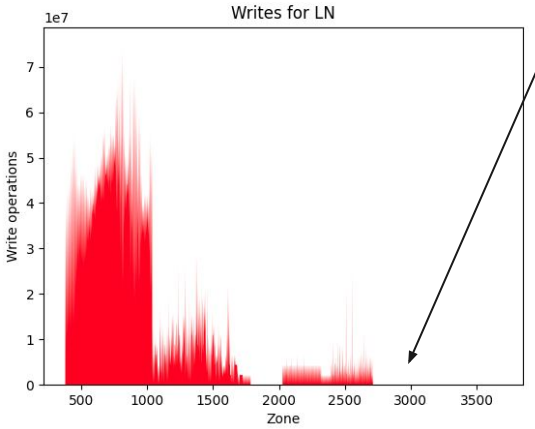
- WALS are filled up very soon.
- Why?
  - See picture on right
  - Memtable size =/= WAL size
- WALS are also very slow…
- Why?
  - Only 3 concurrent writes…
  - Thus every 4th write needs syncing.
  - 3 asyncs are too little compared to what the memtable can satisfy
  - Solution: increase async to > 3

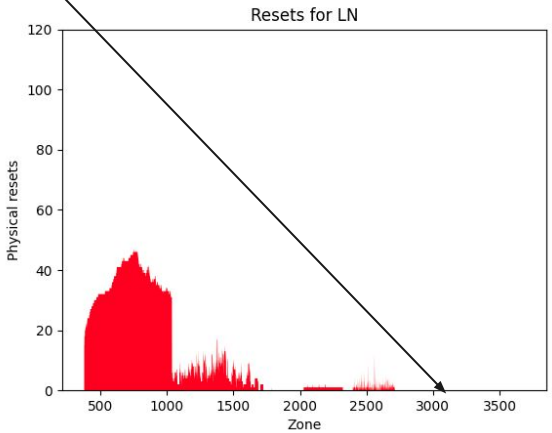


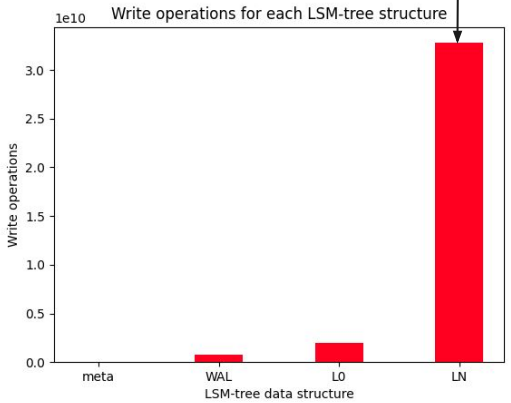
LN is a real problem

Not properly used (gaps)

Unacceptable







LN is a real problem 2

SSTable size on LN
is smaller on LN
than on L0???

L1 is almost ALL I/O!

```
==== Summary ====
Background operations:
        Flushes:492
        Compaction to 1:112
        Compaction to 2:594
        Compaction to 3:383
SSTable layout:
        Level 0: 44 tables
        Level 1: 136 tables
        Level 2: 208 tables
        Level 3: 341 tables
```

==== raw IO metrics ====

Metric	Append (ops)	Written (Bytes)	Read (ops)	Read (Bytes)	Reset (zones)
Manifest	5342	120596480	0	0	0
WAL0	250542745	384833656320	0	0	492
WAL1	249818989	383721967104	0	0	492
WAL2	249819135	383722191360	0	0	492
WAL3	249819131	383722185216	0	0	496
L0	496840	1021676317696	452408	930306827264	695
L1	6612849	13581578121216	13054671	26854013534208	23779
L2	1181198	2426096148992	2112552	4345609322496	3848
L3	377288	775035513856	380457	782616231936	693
Total	1008673517	19340506698240	16000088	32912545915904	30987

Solution

- Increase LN table size. Must be bigger than L0 and must be big enough to not require too much merges.
- Do not separate L1 from LN, use same space. Allows data to remain in the same zones :).
- Compaction to separate thread from flushes. Compaction is too expensive.