Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

# TropoDB: Design, Implementation and Evaluation of an Optimised KV-Store for NVMe Zoned Namespace Devices

**Author:** Krijn Doekemeijer (2714085)

*1st supervisor:* dr. ir. Animesh Trivedi
*2nd reader:* prof dr. ir. Alexandru Iosup

*A thesis submitted in fulfillment of the requirements for*
*the joint UvA-VU Master of Science degree in Computer Science*

August 26, 2022

*"All problems in computer science can be solved by another level of indirection" "..but that usually will create another problem"*

*—David Wheeler(99)*

## Abstract

The amount of data that is processed each year keeps increasing, expected to reach approximately 180 zettabytes of data in the year 2025. Larger data volumes require applications and hardware solutions to scale along. A big part of the digital ecosystem makes use of key-value stores to store and load data. Key-value stores, therefore, become a prime target for optimisation efforts. Recently a new interface for storage devices, known as Zoned Namespaces (ZNS), has been proposed. This interface is able to achieve better results than the conventional block-based approach when used on flash-based SSDs, both in performance and durability. Therefore, there is an interest by both industry and academia to shift to the usage of ZNS SSDs. Key-value stores are also a target for this shift because key-value stores are I/O heavy, require high performance and are used extensively. Further on, key-value stores commonly make use of data structures known as Log-structured Merge-trees (LSM-trees), whose design already follows ZNS closely, making them a natural fit for ZNS optimisations. Especially, co-optimising the garbage collection process of both storage and a key-value store is promising. Therefore, we have designed and implemented an LSM-tree-based key-value store directly on top of ZNS, known as TropoDB. TropoDB directly interfaces with the storage and is designed using an userspace I/O stack within the RocksDB ecosystem.

We come with a number of novel designs for LSM-tree components on top of ZNS. The results of the implementation are both promising and competitive. TropoDB is able to come close to RocksDB with state-of-the-art file systems. However, GC-centric optimisations show limited gains in regards to tail latency, write amplification and numbers of erasures. To improve the performance gains, we present and discuss various optimisation avenues for TropoDB. Source code for TropoDB is available at `https://github.com/Krien/TropoDB`.

## Acknowledgements

I was helped by various people during the thesis. Their help allowed me to set up the resulting database TropoDB and test ZNS devices. I am grateful for their help and would like to express my gratitude. Matias Bjørling, Dennis Maisenbacher and Indraneel Mukherjee have helped tremendously with setting up benchmarking tooling and explaining them both for ZNS and ZenFS. Nick Tehrany has also aided with benchmarking ZNS, especially for fio and the plots for resets, and additionally has aided with setting up F2FS on ZNS. Maximilian Barger has aided with fio for read operations. Further on, I thank the AtLarge Team at the VU Amsterdam for the feedback that was received during the thesis defense. To continue, I would like to thank Alexandru Iosup for his feedback on both my thesis defense and writing. Lastly, I would be remiss in not mentioning the help that the supervisor, Animesh Trivedi, has given. Animesh has helped both in setting up the thesis, weekly discussions and aiding with debugging on GNU/Linux.

# Contents

# List of Figures

# List of Tables

# Glossary

| | |
|---|---|
| **CoW** | Copy-on-Write |
| **DB** | database |
| **DMA** | direct memory access |
| **FS** | File system |
| **FTL** | Flash Translation Layer |
| **GC** | garbage collection |
| **HDD** | hard disk drive |
| **L0** | LSM-tree level 0 |
| **L2P** | logical to physical |
| **LBA** | logical block address |
| **LN** | LSM-tree level N |
| **LSM-tree** | Log-structured Merge-tree |
| **MDTS** | maximum data transfer size |
| **MLC** | multi-level cells |
| **MVCC** | multiversion concurrency control |
| **NUMA** | non-uniform memory access |
| **NVMe** | Non-Volatile Memory Express |
| **OCSSD** | Open-channel SSD |
| **OP** | overprovisioning |
| **PBA** | physical block address |
| **RA** | read amplification |
| **RLE** | run-length encoding |
| **S.M.A.R.T.** | Self-Monitoring, Analysis, and Reporting Technology |
| **SLA** | Service Level Agreement |
| **SLC** | single-level cells |
| **SPDK** | Storage Performance Development Kit |
| **SSD** | solid state drive |
| **SSTable** | sorted string table |
| **WA** | write amplification |
| **WAL** | write-ahead Log |
| **wp** | write pointer |
| **ZASL** | zone append size limit |
| **zcap** | zone cap |
| **ZNS** | Zoned Namespaces |
| **zse** | zone size |

1

# Introduction

We live a data-driven society. Each and every one of us generates data on the daily. This data is collected from all sorts of media, platforms and devices. For example, data is generated by social media (44, 103, 122), IoT and edge computing (16, 21, 118), Large-Hadron Colliders (13, 52), bio-informatics and cancer research (53, 59, 132), and ecommerce (8, 49, 104). The amount of data this creates is enormous and ever-increasing. The amount of data is expected to reach 180 zettabytes in the year 2025 (105). All of this data needs to be stored somewhere and loaded from somewhere. Storing and loading data is not free, it requires storage devices to maintain all of the data. Additionally, it needs to be accessible in a timely manner. No one wants to wait minutes for their data to be processed. Therefore, there is a large investment in data centers and other ICT infrastructure. According to a recently published CompSys Manifesto for the Netherlands, the ICT infrastructure of the Netherlands enables at least 3.3 million jobs and contributes to 60% of its GDP (62).

To be able to facilitate access to data and fast storage, researchers and industry alike have done many innovations in the past decade. Innovations have been made in both hardware and software technologies to deliver high performance. For example, we have seen interest in storage media such as flash-based Solid State Drives (SSD) (41), Optane storage (128) and even DNA storage (85). In the last decade, flash-based SSDs have become the norm for fast storage. State-of-the-art SSDs are typically connected to machines with a PCIe connection and a protocol known as NVMe (80, 91, 129). Such SSDs can deliver throughput of up to 3GB/s and sub-ten microseconds of I/O latency (80). NVMe SSDs are not a technology that will become obsolete anytime soon. On the contrary, it is expected that the market share of NVMe SSDs will grow from 44.6 billion USD in 2022 to 163 billion USD in 2025 (4). However, it is not enough to just use a fast storage medium. In order to guarantee that data is accessible in a timely a manner, the software that is built on top of the hardware needs to be fast as well. Therefore, there have been innovations in software to store the data and communicate with the hardware as well. For example, there have been innovations in file systems (24, 79), object stores (61, 93), relational databases (27, 69, 95) and key-value stores (41) alike. Every one of these data stores has its own advantages and disadvantages. When data stores are properly combined with the underlying hardware, they can scale along with the demands of the data-driven society. In this research we also look at the challenges involved in properly combining hardware with software to scale along with the data-driven society. In particular, we will look at properly combining a database known as a *key-value store* with a type of hardware known as a *flash-based SSD*.

## 1.1 Key-value Stores with Flash Storage

Key-value stores are a type of database and a common way of storing data. One of the few requirements of a key-value store is to be able to store data as a collection of key-value pairs. Data can be stored by the client with simple *put* operations and retrieved by client with simple *get* operations. This abstraction can also be seen in Figure 1.1.

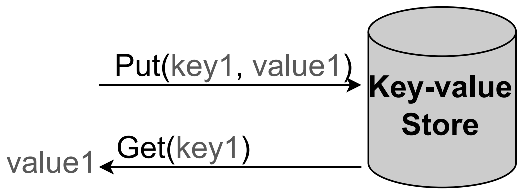


**Figure 1.1:** Key-value store: Put and get operation

Key-value stores are considered to be a part of the *NoSQL* movement and have since their inception been used in all sorts of applications and have become ubiquitous (41). Many large companies such as Google (30, 51), Facebook (43), Amazon (116) and Microsoft (38), are known to use them. Their usage has reached far and nowadays they can even be used as the backend for traditional SQL databases (95). Further on, key-value stores are also used heavily for latency-sensitive/high-throughput use cases (41, 83). Key-value stores are thus a prime optimisation target and a topic that attracts a lot of research efforts (41). A part of this research is focused on the storage medium that the data is stored upon, such as the research that we have conducted for this thesis.

Key-value stores are commonly stored on top of flash-based SSDs. However, flash storage does not come without its own challenges and idiosyncrasies. For example, flash storage lacks support for overwriting data. Instead, data must first be erased, before it can be

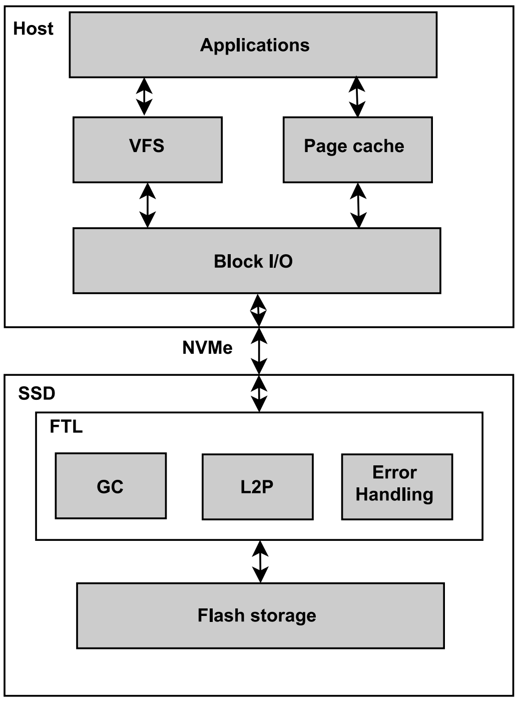


**Figure 1.2:** Linux kernel with a traditional SSD: Storage stack

overwritten. It thus comes with a write and an erasure operation, instead of just a write. Internally the SSD stores its data in *flash cells*. A property of flash storage that is challenging, is that these flash cells *degrade*. Each write or reset to a flash cell degrades the durability of a flash cell, which is commonly known as *wear levelling*. Similarly, reads also degrade the durability, which is known as *read disturbance* (88). Therefore, it is essential to reduce the amount of I/O an application needs when flash storage is used as it can increase monetary costs (5).

A typical approach to deal with SSDs up until now has been to hide away the idiosyncrasies and expose the device to the user as if it is a traditional block device; a device, that many developers are already familiar with. This is done through an interface, which is typically implemented directly in flash devices themselves as firmware. This interface is known as the *flash translation layer* (FTL) (34). The FTL does allow overwrites and does not require the earlier mentioned erasures. Clients can write and read anywhere. It is left up to the FTL to come up with an efficient solution for random writes. As data can only be erased at the erasure level, the FTL is as a result forced to move data around in order to reclaim old space. This is typically done through a process known as *garbage collection* (GC). When an FTL is used on the SSD, this can lead to a storage stack such as is visible in Figure 1.2.

This figure represents how applications can interface with flash storage within the GNU/Linux operating system. In this stack, applications need to go through various layers of kernel abstractions, before it can interface with the SSD. When the request has reached the SSD, the FTL processes the request and only after this has been completed, the request is sent to the flash storage.

**The Challenge of Garbage Collection**:
Garbage collection (GC) on flash storage is considered undesirable, because it causes interference in I/O latency and throughput. Interference in latency is causes because GC claims the resources that are available on the device. When the resources are claimed, they can not be used by client-issued operations. It is generally beyond the clients control to determine when GC should be done, which can lead to interference peaks. Such peaks increase latency of I/O operations. In other words, some operations will take longer to finish and clients have to wait. This is problematic as many companies make use of *Service Level Agreements* (SLA) that specify the maximum allowed latency of an operation. Such an agreement can be set to guarantees such as: 99% of all operations finish in less than 200 microseconds and 99.99% of all operations finish in 10 seconds (83). GC makes such SLA's troublesome as it is hard to guarantee a low latency. Additionally, longer latencies can also have significant influence over the behaviour of customers. To give some examples: Linden et al. reported that for Amazon every 100ms of latency costs them 1% in sales (60, 87), Mayer et al. report that for Google an extra latency of .5 second causes traffic to drop by 20% (60, 96) and brokers can lose millions of revenue if their trading platform is 5 miliseconds behind the competition (60). Therefore, there is a real incentive to keep tail latency to a minimum. Garbage collection hinders this incentive.

Garbage collection also causes *write amplification* (131), which means that the amount of physically written data is more than the logical amount of written data requested by the client. Garbage collection moves data around in the background to facilitate data overwrites. Each data movement caused by garbage collection adds extra write amplification. Earlier, we already established that flash storage has a problem known as wear levelling. Overwrites are, therefore, not only an issue for performance, but they can also degrade the life expectancy of flash storage significantly, which will cost money as new storage needs to be bought. Another problem that arises with the usage of GC is that it requires the SSD to maintain extra data in its DRAM (device memory). This extra data is needed because the FTL needs to maintain where all its data is stored to facilitate garbage collection.

To continue, the FTL and its GC suffer from a problem that is known as the *semantic gap* (135). The semantic gap means that the FTL has no knowledge of the application that is using its interface. This means that it can not accurately predict what data is either *hot* or *cold* or how data is even roughly related. Hot and cold data in this case refers to how frequently the data is updated. It is beneficial to separate frequently updated data from infrequently update data, as this allows minimal write amplification for cold data. Remember that the GC can move data around to support overwrites. If data is not frequently updated, and is thus cold, it should in general not move. The semantic gap causes the garbage collector to frequently put together data that is only written once (cold) with data that needs frequent overwrites(hot), which causes expensive overwrites, and essentially aggravates write amplification issues.

Lastly, as the FTL is generally left up to the SSD vendor, it means that there are various implementations, with each implementation behaving differently and requiring new optimisations from users of SSDs. GC is not something that can be avoided as it can happen at undesirable times, such as during peaks of database usage (15). This complicates optimisation efforts significantly for developers.

## 1.2 Emergence of Zoned-namespace Devices: an Opportunity

There is a solution to these issues. This solution is a new interface for SSDs, known as *Zoned Namespaces* (ZNS). ZNS is a thin interface that no longer resorts to a block interface and comes with no explicit garbage collector on the device itself (24). Instead, we use a common interface that naturally fits flash storage, a *zone interface*. In this interface, the SSD is divided into zones, with each zone corresponding to an individual *erasure unit*, which can be erased by the application itself when necessary, leaving optimisation and garbage collection up to the application. For GNU/Linux this leads to the stack visible in Figure 1.3 instead of Figure 1.2. The erasure operation can thus now be issued by the user of the SSD explicitly as the application is in charge of garbage collection. Further on, in ZNS data can only be appended to zones as overwrites are explicitly not allowed. Remember that data is not moved around in the background anymore by a garbage collector. However, reads can still be done everywhere on the device.

An advantage that ZNS brings is that ZNS is a common interface on top of the NVMe host interface specification (1), reducing the need to create a solution across different architectures and reducing the cost of optimisation efforts. This is different from traditional FTLs, where individual implementations can be very different. Further on, garbage collection is left entirely up to the host. Meaning that applications can remove the semantic gap and the entire storage logic can be optimised for one particular application, such as

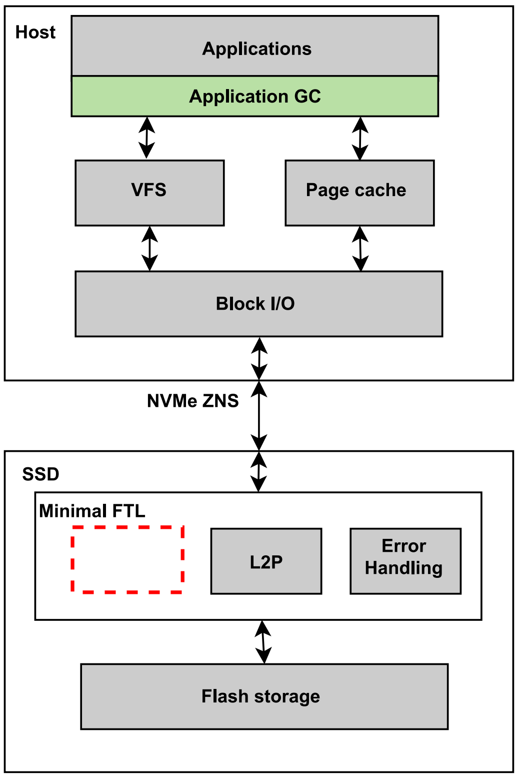


**Figure 1.3:** Linux kernel with a ZNS SSD: Storage stack

in our case key-value stores. It thus allows the key-value store designers to consider how the data should be laid out explicitly on the SSDs flash storage and how data should be garbage collected in the most efficient manner possible depending on the use-case.

A key-value store itself also comes with garbage collection (GC) procedures. These typically come in the form of background operations. Similar to the GC of ZNS, the GC of key-value stores moves around data to different locations and deletes old data. With ZNS, a key-value store is able to optimise ZNS garbage collection procedures for its own use-case. This can lead to a new garbage collection design, in which the two garbage collection procedures are integrated. We refer to this phenomenon as *co-optimising garbage collection*.

Our hypothesis is that if garbage collection is done correctly with the help of ZNS (and is properly co-optimised), it can reduce wear levelling, reduce read disturbance and increase write and read throughput of key-value stores. It can also aid in making latency of I/O more stable as it is more predictable what the SSD is currently doing. Lastly, as it uses the same hardware as ordinary SSDs, there is no additional monetary cost involved in using the interface.

As a co-optimisation it is also possible to reduce the number of abstractions on the host.

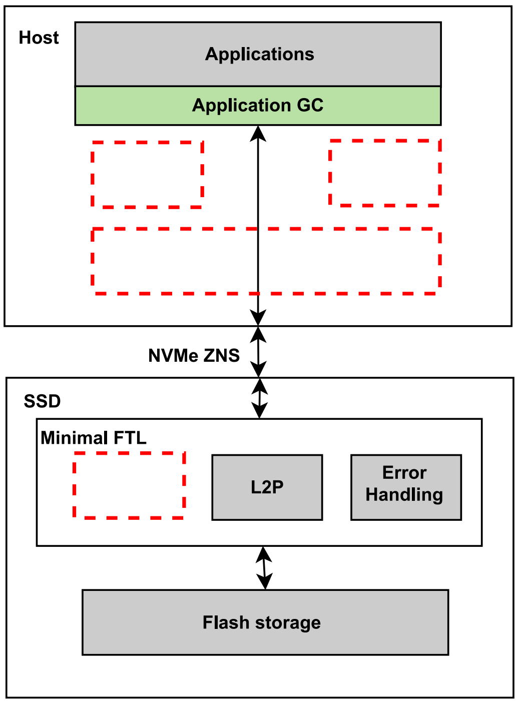


**Figure 1.4:** ZNS SSD when used in user-space: Storage stack

For example, removing most of the operating system logic and giving full control over the SSD to an application. In this case, the SSD is essentially run in user-space. This can increase latency stability and improve throughput. When such a stack is used within GNU/Linux, it would look like Figure 1.4. Our hypothesis is that this can also increase latency stability and throughput for applications like key-value stores.

## 1.3 TropoDB: Optimising KV-Store for NVMe Zoned Namespace Devices

In this thesis, we focus primarily on one of the main advantages of ZNS, co-optimising garbage collection of flash storage for key-value stores by reducing the semantic gap. To define this issue, we should first take a look at the current situation for key-value stores. Key-value store designs such as LSM-trees are already “aware” of the need of hot and cold separation and typically attempt to put together data that is related. For example, component “A" of key-value store “x” is colder than component “B” and should, therefore, be stored elsewhere in a different location. Unfortunately, key-value stores that use LSM-trees typically have no control over the internal logic of SSDs and can not force the SSD to store data separately. Key-value stores can only guess how data is stored on the device or in the best case give hints to the SSD (e.g. with multistream SSDs (43, 76, 130)).

The ZNS interface removes the lack of control over the storage (107). We should now be able to have full control over the layout of all key-value store data and be able to remove the semantic gap at the device level. As an extra co-optimisation it is also possible to remove the file system and most operating system logic as well when used with a key-value store. This is not ZNS-specific, but it removes yet another interface and semantic gap. This approach is taken for TropoDB. When all abstractions layers between the key-value store and storage removed, it gives the key-value store full access to the ZNS interface and full control over how the data is managed. If some layers remain between the key-value store and ZNS, the ZNS effects can no longer be properly be used to their fullest potential. Removing all layers is possible when the key-value store is used within user-space, with for instance the *Storage Performance Development Kit*(SPDK) on GNU/Linux (3).

If this lean stack is designed and implemented efficiently and correctly, such an approach can make the latency of key-value stores more stable; it will make the key-value store easier to predict and can stabilise latency. This is beneficial when the key-value store should respect SLAs set by clients. Further on, it is hypothesised that it can reduce wear levelling effects significantly and thus increase the life expectancy of flash storage in SSDs (24, 107).

In this thesis, we have investigated one particular type of key-value stores, *Log-structured Merge-tree* (LSM) key-value stores. The reason that we are looking at LSM-tree key-value store designs specifically is because LSM-trees are seen as a natural fit for ZNS and are considered to be a prime target for ZNS optimisations (107, 117). It is considered to be a prime target because the data in LSM-trees is generally organised in large contiguous chunks that are treated as append-only logs. The logs could then directly correspond to “N" zones and the concept of append-only fits nicely with the philosophy of ZNS focusing on appends instead of random writes.

## 1.4 Research Questions

Due to the timely nature of this work on newly standardised NVMe ZNS devices, the research space is under-explored, thus opening opportunities for multiple research avenues around a closer storage-application integration. To reduce the design space complexity, this research focuses on a few major ideas that can potentially aid in similar ZNS-integration efforts, not just key-value stores. The overarching research question in this thesis is “how to leverage the unique design properties of NVMe ZNS devices to optimise a key-value store". Under this broader question, we tackle the following subquestions:

- *RQ1: What are the key-value store integration choices available on top of ZNS SSDs?*
  Due to the open and complex nature of key-value store design space (for a more comprehensive treatment of the topic see a key-value store survey by Doekemeijer et al. (41)), we need to both synthesise and identify optimisation opportunities that can be practically tackled within the limited scope of this thesis. Specifically, we target a design space analysis around how ZNS devices have been integrated into an existing system and how key-value stores with the regards to the choice of software frameworks, data structures, garbage collection location, and effort makes use of such an integration. The design space exploration and the position of this thesis in comparison to the state-of-the-art and the state-of-the-practice is done in Chapter 3.

- *RQ2: What unique optimisation opportunities does a ZNS device offer for LSM-tree based key-value stores?*
  Both LSM-tree and NVMe storage are complex topics and there exists a large body of research into both of these topics (see Chapter 2.5). The aim of this question is to identify unique optimisation opportunities that our choice of integration offers (the integration identified in RQ1). In this research question we explore the LSM-tree data structure design space with its access, background operations, and concurrency properties to co-optimise them for the garbage collection on ZNS devices. This research question will be answered in Chapter 4.

- *RQ3: How to implement and evaluate the design of an LSM-based key-value store on ZNS devices?*
  The design of a key-value store with a co-optimised data structures, processes, and policies is complex and needs verification. An implementation of such a complex idea must reconcile practical challenges regarding the choice of hardware available, operating system abstractions, storage APIs, I/O stacks, the use of programming language, debugging and development challenges regarding building the first-of-its-kind artifact. We will present our experience and analysis of implementation related challenges in section Chapter 5.

- *RQ4: What is the impact of the optimisation process of TropoDB on the key-value store and ZNS device?*

There are limited publicly available guidelines and information available on the performance characteristics of ZNS devices and associated software ecosystems like key-value stores or file systems. Hence, to scientifically quantify and verify the claims that are made in the thesis, we must establish the best practices for experiment setup, design experiments (layered evaluation from the ZNS device to the key-value store itself), data collection, and analyse them to study the impact of various design decisions on the overall performance of the key-value store. We report on our findings in Chapter 6.

## 1.5 Research Method

This thesis project is a systems research in which we create a new key-value store. First, we will do an integration exploration on how to integrate ZNS and a key-value store (**RQ1**). Then, after this exploration has been completed, we will start on the key-value store itself. We do this by designing (**RQ2**), implementing (**RQ3**) and evaluating (**RQ4**) the key-value store ourselves.

The following methodologies were used to carry out the research. Firstly, a literature survey was carried out (separate from this thesis) to investigate key-value stores for flash storage (41) (**M1** Quantitive research (71, 84)). This also helped to get an understanding of key-value stores for ZNS SSDs and forms the background for this research.

The design and implementation of the key-value store was carried out with a back-and-forth process. Ideas were abstracted, designed and then prototyped and when prototypes did not work as expected, we went back to the the design and abstraction phases. This is an iterative process (**M2** design, abstraction, prototyping (55, 63, 106)).

The entire research is conducted in the open. The design, implementation and benchmarks are carried out with methods of experimental research (**M3** Experimental research, designing appropriate micro and workload-level benchmarks, quantifying a running system prototype (58, 65, 100)) and are all open-source (see Chapter 8 and Chapter 1.6) and are within the principles of open-science (**M4** Open-science, open-source software, community building, peer-reviewed scientific publications, reproducible experiments (17, 20, 121, 125)).

## 1.6 Thesis Contributions

The primary result of this thesis is the design and implementation of the key-value store TropoDB. TropoDB **is the first key-value store that is fully designed and im-**

**plemented on top of ZNS without a file system, kernel and block in layer in between the application and the ZNS device**. Additionally, it runs entirely in user-space. It is novel in its approaches to make use of unique data structures to store LSM-tree components. Furthermore, its exclusive use of *append* I/O operations instead of *write* I/O operations. The thesis addresses various issues caused by garbage collection and how those issues can be solved with ZNS. For example, how to separate hot and cold data and when and how to delete data. In addition, in the end product, we have explicitly separated the database code from the storage interfacing code, allowing the storage interfacing code to be reused for other (research) projects. The logic in this project can be of value for other ZNS projects. As TropoDB is the first of its kind, we also came across, and discussed, various design decisions for TropoDB that do not work properly or do not scale. These can be valuable lessons for future attempts at creating key-value stores or other applications built directly on top of (ZNS) SSD storage. Specifically, the contributions of the thesis are the following:

1. Design and integration considerations for building a key-value store on top of ZNS SSDs. Specifically how LSM-trees fit in zones and how garbage collection of LSM-trees can look for ZNS SSDs.

2. SimpleZNSDevice (SZD), a small API built on top of SPDK, meant to interface with ZNS. This API comes with a simple configuration and some data structures optimised for ZNS. It can be reused for any ZNS project.

3. Benchmarks for ZNS SSDs that showcase the raw device characteristics of reads, appends, writes and resets when SPDK is used.

4. TropoDB, a key-value store built directly on top of SZD in user-space with the help of SPDK. We explore design choices regarding the underlying ZNS storage with a focus on garbage collection.

5. An investigation of TropoDB, showcasing its tail latency, write amplification and zone reset characteristics compared to the state-of-the-art and the state-of-the-practice.

6. Publicly available source code, both for SZD, SPDK and the benchmarks. The source code can be found respectively for TropoDB, SZD and the benchmarks at `https://github.com/Krien/TropoDB`, `https://github.com/Krien/SimpleZNSDevice` and `https://github.com/Krien/ZNS_SPDK_Benchmarks`. The sources are described in more detail in Chapter 8.

## 1.7 Research Relevance

The research relevance of TropoDB lies in its novelty. TropoDB is the first key-value store built entirely on ZNS SSDs. This thesis has shown that such an approach is feasible. Additionally, by implementing the database in user-space, we have been able to investigate the viability for key-value stores in user-space with ZNS as well. During the design and implementation of TropoDB, we have come across various challenges that were not known before and are specific to ZNS. We have shown that a few designs are able to come close to what is achievable on ZNS, while others are not. The WAL design for example is able to closely match the raw performance of ZNS devices and is a design that has shown its merit for key-value stores. The other components have had less success, but by demonstrating the potential of dividing an SSD in various components, we have paved the way for future research. It should be possible to try out various data structures for different components and find designs that work better for their use cases. This thesis built a platform for future research in a direction of improving average latency, tail latency, throughput, wear levelling and concurrency capabilities for all applications that make use of storage, not just key-value stores. Many ideas, such as design of WAL, are more broadly applicable beyond key-value stores, such as in file systems or relational databases. It also shows that is possible to investigate each component separately in a key-value store. However, considering that without a file system everything needs to be crafted by hand, piece by piece, it requires a significant investment from the research community.

## 1.8 Societal Relevance

In this research we aim to design and implement a high performance key-value store. The goal is to tackle the influx of large amounts of data and the ever-increasing needs that go along with this data. In the end, the large amount of data has implications for monetary costs and and environmental costs(62). We address this challenge by coming with a key-value store that is optimised for the underlying SSD storage medium. Such a co-optimisation can reduce latency issues and increase device lifetime. The proposed key-value store can lead to storage systems that are better provisioned, more efficient and scale better, all at the same time. It is common to scale systems by adding multiple units/nodes(27, 45, 86). Such an approach leads to both monetary costs and environmental costs. The solution proposed in this thesis, allows for increasing the performance of a single unit instead. The result is that less units are needed to guarantee performance and both monetary costs and energy costs can be decreased. Additionally, the approach taken can increase device lifetime significantly. Therefore, there is less investment needed to replace broken storage devices, which again reduces monetary and environmental costs. In short, this research aids society by allowing time-sensitive applications to work in low latency, and by reducing economical and environmental costs of data.

## Plagiarism Declaration

I confirm that all material present in this report, unless explicitly stated, is the result of my own efforts. No parts of this report are copied from other sources unless credited and properly cited. The work has also not been submitted elsewhere for assessment. I understand that plagiarism is a serious issue and should be dealt with if found. Note that the background section contains parts from my survey on flash-based key-value stores, which is my own work (41).

# 2

# Background and Related Work on KV-Stores on Modern Storage Devices

In order to explain the design of our key-value store, we first have to explain what a key-value store is and what kind of technologies the key-value store will be interacting with. We need to understand their individual characteristics. We will discuss flash storage, the ZNS interface, key-value stores and LSM-trees, as these are essential to explain how the design of TropoDB came to be.

## 2.1 How do Flash-based SSDs work?

*Solid State Drives* (SSD) are storage devices that do not make use of moving parts. They internally consist of an integrated circuit and a storage medium. While storage media can differ between SSDs, in this research, we are only interested in the storage medium used most commonly within SSDs, *flash storage*. Whenever we refer to an SSD in this paper, we are talking about flash-based SSDs, unless explicitly stated. In this section, we will highlight the properties of SSDs that are most interesting for this research.

### 2.1.1 Why Flash-based SSDs are used

Flash storage is not a new technology, as it was invented in 1980 by Toshiba (72). Nevertheless, flash only saw increased usage in the past two decades. In these two decades flash storage became cheaper and saw an increased adoption rate by data centers amongst others (11, 82). Flash storage is considered promising because of a number of reasons, which led to its large rate of adoption. Most of these reasons relate to SSDs being (in most cases) superior to the most common alternative, *Hard Disk Drives* (HDD). SSDs can achieve lower latencies and higher throughput than conventional HDDs and come with high concurrency capabilities (6, 12, 110). Further on, unlike HDDs, SSD performance is affected less by random I/O. This makes them more suited for I/O heavy use-cases such as persistent databases. Yet at the same time, they are not exponentially more expensive than HDDs and are widely available (54). This can make moving to SSDs an affordable transition to increase application performance. Another advantage of SSDs is that they are considered more green *to use*, because SSDs consume less electrical power than HDDs, especially when idle (11, 37). Nevertheless, the exact difference in power usage depends on the SSD model and how it is used (37). Further on, the cost of manufacturing SSDs is not necessarily greener than the cost of manufacturing HDDs (119). One reason for this cost difference is that SSDs are not just flash storage and each part of the SSD can be made by a different manufacturer.

### 2.1.2 What is Flash Storage?

Flash storage is a non-volatile storage medium where data is stored as an electronic charge on a floating-gate between a control-gate and the channel of a CMOS transistor, in what is also known as a *memory cell.* This cell can then be programmed, read, and erased, but only a finite number of times. A cell can be written to a finite number of times because of a process known as *wear levelling.* Wear levelling means that after each write or erasure, the cell slowly degrades. This results in decreased performance and eventually leads to a dead cell. Further on, cell degradation is not just limited to writes and erasures, reads can also degrade the performance of cells, albeit to a lesser degree. This is known as *read disturbance* (88). It is, therefore, considered harmful to do excessive IO on flash storage (98).

The amount of bits a cell holds depends on the cell technology used. The cell technology can be defined by several *levels*, with higher levels holding more bits. For example, *single-level cells* (SLC) can hold just 1 bit and *multi-level cells* hold 2. At the moment, there exist in order of density: SLC, MLC, TLC, QLC and PLC (92). Higher levels allow us to store more data in each cell, which allows for denser storage. Yet at the same time, with such designs, it is only possible to write or read at the granularity of the cell technology. That is because multiple bits occupy the same cell. This can decrease throughput and increase wear levelling (10).

Cells are then packed together in a hierarchical configuration. Hierarchical configurations are for example *NAND flash* and *NOR flash*. In the case of SSDs, NAND flash is used. So we will only discuss NAND. NAND packs cells together densely, which is good for mass storage, but only allows addressing I/O at the level of *pages* (36). For example, a common page size is 4KB. This means that it is not possible to write to just 1 cell, but you always have to write to a multitude instead. Further on, it is not possible to overwrite a page. Once a page is written, it is written. Instead, a page must first be *erased*, before it can be overwritten. However, such erasures can only be done at the next hierarchical level, the level of *blocks*; for example, a block of 2MB. So, every time data needs to be erased, it might be necessary to move around multiple pages to a new block before the old block can be deleted without data loss. This makes it very beneficial to group data based on their lifetime, as this can significantly reduce the need to move around cold pages on erasures. This need to write more data is also known as *write amplification* (WA), writing more physical data than was logically requested. For example, a logical write of 1 page, might be rewritten physically multiple times because of erasures. To continue the hierarchical layout: blocks are then packed together in *planes*, which are packed together in *dies*, which are packed together in a *package*. Packages can be processed in parallel, increasing performance. Lastly, flash storage also comes with multiple *channels* that can be shared by multiple packages. Those channels allow managing the I/O operations.

Another important detail of flash is that the cost of operations is asymmetric. The latency of a read operation is lower than the latency of a write operation and in turn the latency of a write operation is lower than the latency of an erasure operation. Further on, flash storage does not make use of moving parts. This allows the device to get the performance of random I/O close to the performance of sequential I/O. For example, Xu et al. benchmarked a flash-based SSD and measured a difference in average device latency between 85$\mu$s for sequential reads and 90$\mu$s for random reads (129). Do note, that these results are mainly chosen to get an idea of how close random I/O performance is to sequential I/O performance. Results are highly dependent on the device and workload used. In this research, we will also benchmarks random and sequential I/O ourselves (see Chapter 6.3.2).

### 2.1.3 What is a Flash-based SSD?

A flash-based SSD is an SSD that contains flash storage. However, an SSD does not just contain a persistent storage medium such as flash. It typically also comes with a small processor complex, which allows managing the storage, and some RAM to buffer I/O operations. How such a design looks is visible in Figure 2.1. Therefore, to make SSDs truly persistent, it is necessary to force the SSD to flush its buffers. Further on, SSDs are also not just hardware. They typically also come with firmware installed, which defines how to interface with the storage and how to deal with storage errors.

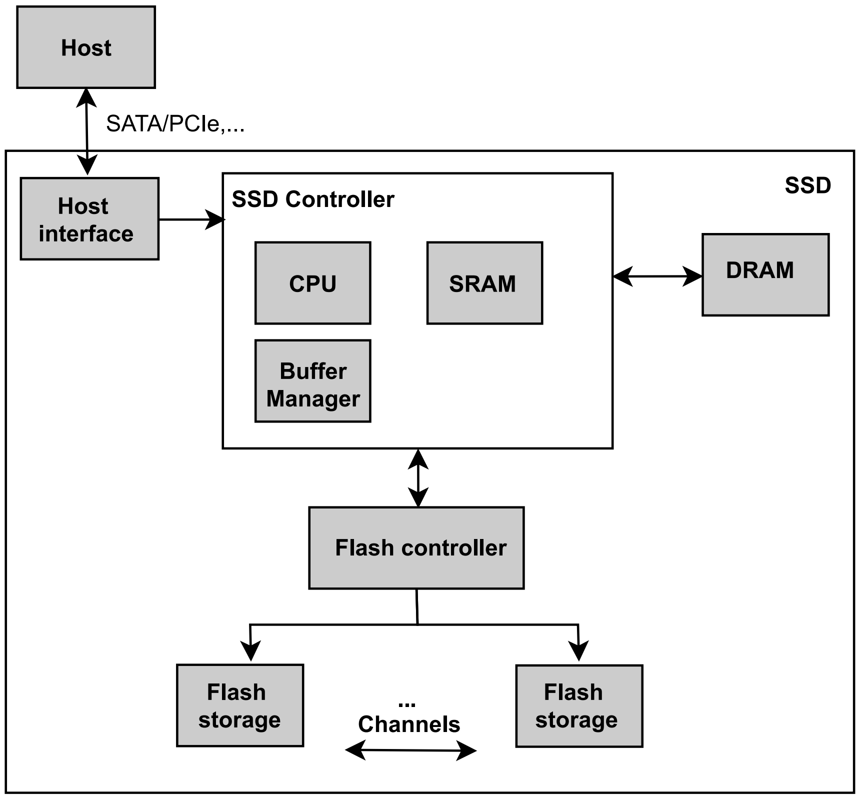


**Figure 2.1:** Flash-based SSD: Organisation

An SSD on itself does not do much, so it is generally attached to a more functional machine. There exist multiple methods for attaching SSDs. For example by making use of *SAS*, *SATA*, *FC* or *PCIe* (127). Each of these has different performance characteristics, requiring rethinking how to optimally use the attached storage. In addition, they can be connected with different interface protocols such as *AHCI* and *NVMe* (77). NVMe directly interfaces with the storage and allows achieving the maximum parallelisation that an SSD can achieve. AHCI on the other hand, does not. NVMe with PCIe is currently considered the state of the art and is also the attachment method is used in this research.

### 2.1.4 Flash Translation Layer

SSDs are typically interfaced with an interface known as the *Flash translation layer* (FTL) (34, 75). This can either be implemented directly as firmware on the device itself or implemented on the host (66). The FTL allows interacting with the SSD as if it is a conventional block device. In other words, it can be interfaced like other block devices such as HDDs. The device can then be used as a flat array of virtual addresses. This can also be seen in Figure 2.2. When using an FTL, only writes and reads have to be issued by the client and there is no need to think about how to perform overwrites or erasures anymore. For example, the client can write to any LBA presented in Figure 2.2. There is no need to force sequential writes on the host. Instead, this is left entirely up to the FTL. The FTL is in charge of translating virtual addresses to physical addresses and managing erasures in the background with a *logical to physical* table (L2P). This leads to situations as can be seen in Figure 2.2. The first LBA points to the second *physical block address* (PBA) and once pointed to the first PBA. When a logical address updates, the old physical address becomes invalid and the logical address is assigned a new physical address. It is up to the FTL to clean up the old addresses on the background.

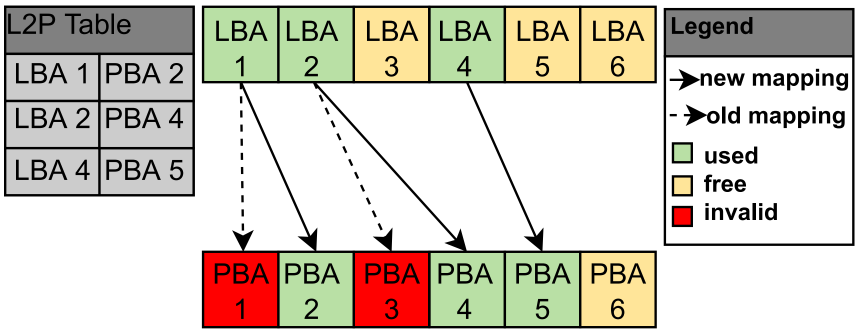


**Figure 2.2:** FTL: L2P table

Background operation are performed by a *garbage collector* (GC) present in the FTL. The GC is in charge of erasing old data and moving data to more *convenient* locations. Typically this means that when a host application issues an overwrite, the virtual address is mapped to a different physical address and the old physical address is marked dirty and at some point in time erased by the GC. This erasure can be done immediately or postponed and is left up to the erasure policy of the GC.

Using an FTL makes it trivial to add support for SSDs to an application that uses a standard block interface. It also makes it easier for software engineers that are not familiar with flash specifics to use such devices. Yet, a flash translation layer is not a free lunch. It comes with some significant disadvantages. Later on, we will discuss a different interface, ZNS, that will mitigate most of these disadvantages. The first disadvantage is that there is no control over the physical layout of data on flash (24). A virtual address can point to any erasure block, no matter what else is stored in this erasure block. This prevents hot and cold separation techniques, which is problematic as hot data can be stored together with cold data. In turn, this means that cold data might be moved/erased more often than it needs to be, leading to expensive GC. The second problem is that there is no control over this GC. The GC can be scheduled at sub-optimal times, for example, when a lot of writes are issued by the client. This can cause I/O to stall and the latency of client-issued I/O operations on the SSD to become unpredictable. Thirdly, an FTL requires *overprovisioning* (OP). Overprovisioning reserves a part of the flash storage on the SSD for garbage collection. This can for example be used as a sink for data when the garbage collector wants to reset a zone that still contains live data. This data must be stored somewhere. However, overprovisioning wastes storage by reducing the amount of usable space. Lastly, there is no restriction on FTL implementations, meaning that designs can vary between both vendors and SSD models. For example, there exist FTLs with page-level mapping, block-level mapping and even hybrid-level mapping (hybrid-log FTLs) (31). Therefore, it becomes a Herculean effort to optimise for SSDs as a different approach needs to be taken for different SSDs.

### 2.1.5 How do NVMe ZNS SSDs work?

*NVMe Zoned Namespace devices* (ZNS) for SSDs is a specification that defines an SSD in which the flash storage within an SSD is exposed to the user as an array of *zones* (1, 24). It is a standard declared in the NVMe specification and is the successor of *Open-channel SSDs* (22). It is an alternative to standard FTLs and comes with no garbage collector by itself. In ZNS each zone closely matches with the erasure unit (block) of the flash storage within the SSD. Using ZNS mainly revolves around interacting with these individual zones. This design is perpendicular to ordinary FTLs in which the SSD is exposed as one large flat array of addresses. Each individual zone has a state and a set of attributes that are maintained within the SSD, but can be requested by the host as seen fit. For a full explanation of ZNS, we recommend reading the NVMe specification (1) or the explanation at zonedstorage (2), we will only look at a subset of the logic. A major advantage of ZNS is that both garbage collection processes and access patterns can be altered to fit specific applications. This allows creating an optimal mapping for each individual use case. For example, optimising for low-latency applications or optimising for applications to reduce wear levelling effects.

**How Does a Zone Work?**:

To explain zones and their states we will follow the example given in Figure 2.3. Each zone is append-only and has exactly one write head, known as the *write pointer* (wp). The beginning and ending of each zone are fixed. Additionally, the wp of each zone initially starts at the zone's beginning. This zone is then in a state known as the *empty* state. In order to write to this zone, the zone must first be opened. This can be done *explicitly* with an *open* operation or *implicitly* by just writing to the zone (step 1 in the figure). It is only possible to write to exactly the wp of a zone and nowhere else. This can be done with either a *write* or an *append* operation. The difference between a write and an append operation is that for a write you need to know the wp as you write explicitly to the wp (need to specify its location). Whereas for an append, you only need to specify the beginning of the zone and do not need to maintain the wp itself on the host. The physical location of the data will be returned to the host on completion instead; essentially inverting the responsibilities. Additionally, ZNS can accept multiple append operations simultaneously to the same zone as appends are *nameless* (94). This can increase the performance, as no additional waits are necessary between "writes", allowing the SSD to determine the optimal order for appends. After each completed write, the wp increases. This is visible in step 1. Three pages are written in this step and the wp has increased three positions as well. The wp can continue up till the end of the zone is reached, at this point the state is transitioned to *full*. When the zone needs to be reused, it can be reset with the *reset* operation. For this it does not matter if the zone is empty, partially filled or full, the zone will be erased regardless. On a reset, the wp will be reset to the beginning of the zone and the zone will transition back to the empty state. An example of a reset is visible in step 3, the wp moves back to the beginning and all pages are erased. Reads can be performed on the zone regardless of the location of the wp. This means that is possible to read at any address of both a full zone, a partially filled or an empty zone.

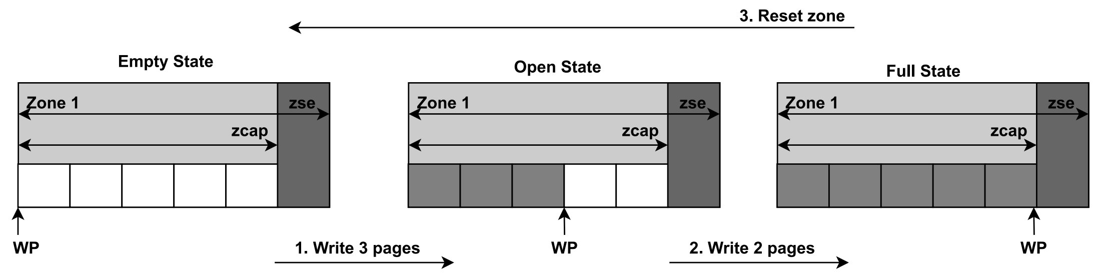


**Figure 2.3:** ZNS design: Example of ZNS state transitions

### Where Does a Zone End?:

We mentioned that writes can only continue until the end of the zone is reached. This requires some extra explanation. Each zone has a *zone size* (zse) and a *zone cap* (zcap).

The zse and zcap can also be seen in Figure 2.3. Zone sizes are the same for all zones and are equal to the total size of a zone. Zone capacities are less than or equal to the zone size. However, the zone capacity is the only part of the zone that can be used by the host. The part between the zcap and zse can not be used by either writes or reads by the host and is meant for the device only. That means the zone is full when the zcap is reached and reads after the zcap are considered illegal by the device.

**Size Constraints**:
Yet another property that must be dealt with has to do with I/O limits. Reads and writes must always be aligned on the block level. That means that it is not possible to write just one byte or read just one byte. Instead, I/O must always happen in units of *pages*, typically 512 bytes or 4kB. Writes, appends and reads are lastly limited by a maximum size. The maximum amount of bytes that can be written or read in one operation is limited by the *maximum data transfer size* (MDTS) and the maximum amount of bytes that can be appended in one operation is limited by the *zone append size limit* (ZASL).

**Constrains of Using Multiple Zones**:
Finally, there is one last idiosyncrasy of ZNS that we must elaborate on. It is possible to write to different zones at the same time. This can increase parallelism and thus performance. It also make it possible to separate hot and cold data (hot and cold can be written to different zones). Yet, the ZNS specification specifies that it is possible to set a limit on the maximum number of *open* and *active* zones. This means that there is a limit on the number of zones that can be written to simultaneously. Open zones are all zones where the wp of the zone is neither at the beginning nor at the ending of a zone. Once the host considers that a zone no longer needs writes for a while, the zone can be closed with the *close* operation. The wp will not be reset in this case, but the zone transitions to the *active* state. It can then be reopened when it needs to be used again. In other words, each open zone is always active, but not each active zone is open. However, as active zones also have a limit, this might not be enough. Therefore, if the host knows that the zone no longer needs any writes at all, the zone can also be *finished*. The finished command immediately sets the zone to the *full* state. In short, developers that make use of ZNS need to be aware of the number of zones that are open or active concurrently and act accordingly.

To give a short summary, ZNS is a specification that allows developers to interact with SSD storage on the level of zones. Developers have to consider the state of zones, have to know where zones begin and start, should adhere to size constraints of I/O and should be careful that they limit their usage of multiple concurrent zones.

## 2.2 What is a Key-value Store?

A key-value store is a type of database that is part of the NoSQL movement (41). The idea of key-value stores is not new as they were already used in the 20th century (113). The requirements to be called a key-value store are low and not set in stone. They are identified as a flat collection of key-value pairs. Data is stored as keys pointing to values. For example, a filename pointing to file contents or a human name pointing to a social media profile. In this research, we will stick to a minimal definition. In this definition a number of operations are defined. These are also shown in Table 2.1. data can be stored with a *put* operation and data can be retrieved with a *get* operation. A put operation requires setting a mapping from a key to a value. A get retrieves the value associated with a key, as long as there is one. Some key-value stores also come with a *delete* operation, but this can also be implemented by setting an *empty* value with a put. Further on, many key-value stores come with a *scan* operation, allowing iteration on a set of keys, but this is again not required.

| Operation | Explanation | Required |
|---|---|---|
| put | Stores a key-value pair | Yes |
| get | Retrieves a key-value pair | Yes |
| delete | Deletes a key-value pair | No |
| scan | Retrieves a range of key-value pairs | No |

**Table 2.1:** Key-value store: Available operations

The persistency level of key-value stores can differ. Some are stored entirely in memory and will be erased on shutdown (completely in-memory key-value stores), others are also stored on storage. In this research, we are only interested in persistent key-value stores, key-value stores that maintain data on storage and will remain persistent, even on server shutdown. The reliability model can differ between persistent key-value stores. They do not have to be *ACID* (see the ACID specification for more information (112)). In fact, using *BASE* or another less strict model is not uncommon. Therefore, picking a key-value store as a storage backend also requires verifying if the picked key-value store has the reliability guarantees needed for the end product.

Persistent key-value stores have become commonplace and are used by a large number of industries (28, 38, 43, 109). Key-value stores are considered flexible, which allows optimising them for various use-cases. This can range from web shops (109), to graph data bases (28) to game servers (38), to backends for relational data bases (95). Not all storage needs the functionalities that traditional relational databases provide or their persistency guarantees. For such storage needs, it might be better to use a lighter database and add extra requirements on top of the database only as they are needed. For example, ZippyDB uses key-value stores in a distributed setting and adds reliability itself on top (28). In this case, it is possible to use a key-value store for each node and to move reliability problems to the distributed setting itself. This is not (efficiently) possible if the database already includes all of this logic. This allows for a clear separation of problems. Many modern key-value stores are optimised for low-latency, large throughput, and terabytes of data and many of them are optimised for SSDs (41). These types of databases are the target of this research. As they are optimised for SSDs, they are a target for ZNS optimisation, and as they have to deal with large I/O and low latency, they have characteristics interesting to verify the effects that co-optimising GC can give.

Key-value stores can internally be implemented by almost any data structure; any structure that holds, manages and stores and loads key-value data from storage suffices. Some common implementations make use of *hash tables*, *B-trees* and *LSM-trees* (41). LSM-trees are the target for this research as they are favorable for flash storage, which we will explain in more detail in the next section.

### 2.2.1 How does an LSM-tree work?

An LSM-tree is a composite data structure, a data structure that itself consists of various smaller connected data structures. In the case of an LSM-tree all structures are connected together in a large sequential funnel. Each data structure in this funnel serves as a waiting hub for the next structure, which is always bigger than the previous structure. All changes always begin in the smallest data structure and progress step by step to the largest structure. This design allows amortising expensive I/O operations and creating individual data structures optimised for their underlying storage and size. For example, structures optimised for doing many small writes in memory and structures optimised for holding large, but infrequently updated quantities of data on storage (101). In order to properly describe this funnel, we will describe how written data moves through the tree step by step. How this data movement happens can also be seen in Figure 2.4, which showcases the design of the LSM-tree design in LevelDB (50, 51) and is based on an image by Lu et al. for the Wisckey paper (90). Then we will describe, again step by step, how data can be read from the tree.

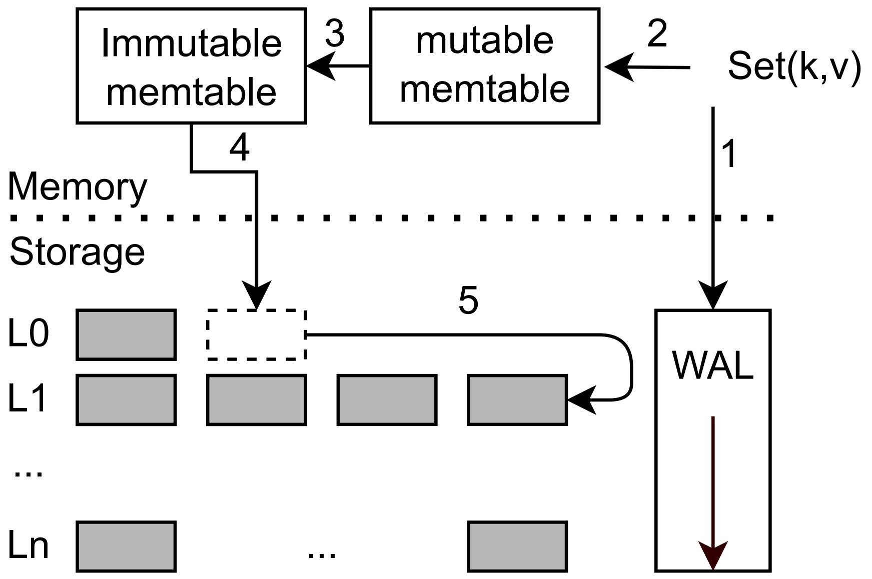


**Figure 2.4:** LSM-tree design of LevelDB

### 2.2.1.1 LSM-tree Write Operations

**(1,2)** All *put* operations first arrive at the most basic components of the tree, the *memtable* and optionally a *WAL*. A memtable is a completely in-memory structure that is used to store and get data. It can be characterised as a small in-memory key-value store. When a client *puts* data, it is always first applied to the memtable. The memtable is optimised for random-access memory and can thus overwrite, reorder and sort data without major performance problems. There are no constraints on the implementation of memtables, but a common implementation is a *skip list* (134). However, using just a memtable does not guarantee any persistence of data. For example, think about what would happen during a power loss or even a clean and correct database shutdown. In this case, all data written to the in-memory structure will be lost.

Therefore, most LSM-trees come with an extra option that allows making the database persistent. When this option is active each *put* is apart from being logged to the memtable **(2)**, also logged to another data structure, known as a *write-ahead log* (WAL)**(1)**. Unlike the memory table, this structure resides entirely in storage. The function of the WAL is to back the memory table. WALs are implemented as append-only logs and are similar in functionality to *redo logs* or *journals* used in traditional relational databases. WALs only record changes applied to the memtable and are thus essentially logs of deltas. Whenever a key-value store restarts, all changes can be read from the WAL and reapplied to a memtable. This recreates the state of the memtable to what it was before database shutdown. The combination of memtable and WAL is thus persistent.

**(3,4)** Once the memtable or the WAL becomes too large, the data of memtable should be moved to the next part of the funnel, which will now be completely written to storage. The memtable and WAL are in this case flagged as dirty and the flagged memtable will become *immutable* **(3)**, or in other words *read-only*. This is necessary as the immutable memtable will be written to a bigger data structure and should, therefore, no longer be altered. Writing the immutable memtable to storage is known as *flushing* and is done by a separate background thread **(4)**. During this flush the immutable memtable is converted to a *Sorted String Table* (SSTable), which is a collection of key-value pairs sorted on keys. This SSTable is then appended to the top-level of the tree, known as *L0*. Important to remember is that while all SSTables are stored on storage, a bit of the metadata such as the range of each SSTable are retained within memory. This reduces the cost of later read operations as it can already be seen beforehand if a key is present within the range of an SSTable. If it is not, no read will be necessary. Lastly, in order to not stall any updates issued by the client during the flush, a common optimisation is to create a new memtable and WAL so that the client can continue to issue puts. The client will work on these new structures, while the flush can safely work on the old structures.

**(5)** Eventually, L0 is considered to be too big as well. In this case, a few L0 SSTables move to the next level of the tree, which is known as *L1*, with a process known as *compaction*. A big difference between L0 and L1 is that L0 can have SSTables with overlapping key-ranges, but L1 does not; instead, all SSTables must contain a unique key range. Therefore, the chosen SSTables of L0 are merge sorted with all overlapping SSTables of L1. The tables that result from the merge are appended to L1 and the old SSTables (both the chosen L0 SSTables and the overlapping L1 SSTables) can be removed.

An LSM-tree can have more than two levels on storage. The number of levels is typically configurable. All levels after L1 are similar to L1; they all contain non-overlapping SSTables and they can all be moved to the next level with the compaction procedure. The only difference is that each level is bigger than the previous. The last level where the data might end up is called *LN*.

#### 2.2.1.2 LSM-tree Read Operations

On a *get* operation, we might have to look into multiple data structures as the data can reside in any of the layers of the tree. We should always ensure that only the most recent data is read, as the client does not want to read old data. This can easily be guaranteed if the writes work correctly and they indeed proceed in a funnel. That is because each next part in the funnel contains older data, so on a read we should move through the funnel from beginning to the end. It is important that a snapshot is used, to prevent a client from missing a data movement between levels. It needs to read L0 up to LN as they are at the moment of the get. Whenever an entry is found that matches the key, the value is immediately returned and we no longer have to look further in the funnel as we know it is the most recent entry. The following structures might need to be read in order: memtable, immutable memtable, L0, L1, L..., LN. Note that each of these structures might itself need multiple reads. Both of the memtables reside entirely in memory and only require 1 read each. L0 might require reading all of its SSTables from storage as the ranges can overlap. Lastly, L1 up to LN only requires 1 read from storage as the tables can no longer overlap.

#### 2.2.1.3 LSM-tree Garbage Collection

Write operations on the LSM-tree are not only issued by clients. Clients only put data into the memtable and the WAL. Separate background processes, usually implemented with threads, are in charge of operations 2,3 and 4 of the write process. These operations are also referred to as *garbage collection* operations. They move the data to new locations and remove the old data. For example, on a compaction, multiple tables are merged into a new table. The old tables need to be removed afterwards. This can lead to write amplification, as data will be rewritten for each layer and can also be rewritten to the same layer on a merge (data is copied). Similarly, it causes quite a few deletions of old data.

#### 2.2.1.4 Why a Funnel is used in the LSM-tree

The general idea of composite data structures is that each structure has properties that better fit the size, usage and storage medium of the data that it contains. For example, a large part of the data is generally only read, making it beneficial to store that part of the data in a structure that works well for reads only. The LSM-tree is no different in this regard. Only a small part remains in memory as memory is both limited in size and not persistent. This part is optimised for this use case with the use of memtables. Each higher level of the LSM-tree is optimised to handle more data than the previous level and each higher level is generally accessed less frequently. Thus each higher level in the tree has more to gain with efficient storage than with efficient write or update throughput/latency. Another benefit of a funnel design is that it allows delaying and amortising expensive operations. For example, if L0 did not allow overlaps, all flushes would need to merged in L0. This would significantly increase the cost of flushes and it might be better to delay such operations and do them in one go later on; say merging 10 L0 SSTables into L1 instead of merging 10 times into L0 and then one time into L1. This can also reduce write amplification as the number of overwrites is reduced. The same idea holds for levels above L1. Increasing the number of levels can reduce the size of most merges, decreasing write amplification. At the same time, increasing levels, increases the amount of data that might need to be read as more levels need to be read. This is known as *read amplification* (RA), reading more physical data than was logically requested. LSM-trees thus come with a trade-off between WA and RA.

#### 2.2.1.5 Why are LSM-trees Favorable for (ZNS) SSDs?

An LSM-tree is considered favorable for flash for a number of reasons (107, 117). Part of the explanation will be similar to the reasons why open-channel SSDs are favorable for LSM-trees as well (135), as the two technologies are very similar. Specifically, we will consider an LSM-tree directly on the storage. Otherwise, it depends on the layers between the LSM-tree and storage whether an LSM-tree is efficient. Firstly, LSM-trees mainly make use of appends. Changes to any structure, add new data, and delete old data. This is true for all major I/O components: updates to WALs, updates to metadata and adding new SSTables are all appends. Using appends fits naturally on SSDs as SSDs do not allow random I/O in the form of overwrites. Instead, adding new data is more convenient. In other words, data is never updated in place, which is the type of access pattern that works best on flash storage. Further on, LSM-trees already come with a clear separation of hot and cold data. Each step in the *funnel* is colder than the previous one. This can help the garbage collector to properly separate data based on temperature. Further on, LSM-trees are optimised to reduce write amplification, not read amplification (in most cases) (89, 90, 97). Many LSM-tree designs trade read performance for better write performance. This fits SSDs because of the asymmetric cost of I/O on SSDs, as writes are more expensive than reads in both latency and wear levelling costs. Normally there is an FTL between the SSD and the key-value store. An FTL might translate in-place updates to appends instead; removing benefits that append-only designs might have had before. Similarly, it can not be guaranteed that hot and cold data are stored separately. Lastly, the results can differ between FTL implementations. This makes it hard and not always justifiable to state that LSM-trees are more favorable than other structures. ZNS SSDs give full control over the SSD, which allows guaranteeing that the LSM-tree is used as it is designed.

### 2.2.2 RocksDB and LevelDB

RocksDB is a key-value store that is considered the state-of-the-art (41, 46). It is used by a large number of users and is a hot topic for research (as of this writing in August 2022). Many key-value store ideas originated from RocksDB or have been applied to RocksDB. For example, applying Wisckey (90) on RocksDB with BlobDB (43). RocksDB is built on top of an earlier key-value store, LevelDB (50, 51). LevelDB is a key-value store that originally sparked a large interest in key-value stores and saw a surge in their usage (41). However, LevelDB left many optimisations open and has, for example, many issues with large I/O and compactions. Further on, it is not explicitly built for Solid State Drives, but Hard Disk Drives. On the other hand, RocksDB is explicitly optimised for flash-based Solid State Drives (43). This in combination with its popularity, makes it an interesting target for both optimisations and comparisons. Internally both LevelDB and RocksDB make use of an LSM-tree. The figure, Figure 2.4, is also based on the LevelDB design. RocksDB comes with advancements such as multiple compaction threads to make better use of the internal parallelism of flash. TropoDB in turn continues on the work of both RocksDB and LevelDB. Lastly, RocksDB also comes with its own benchmarking tool, db_bench. A benchmark that is based on the workloads in which RocksDB is used.

## 2.3 Garbage Collection and the Semantic Gap

We have already introduced the notion of a *garbage collection* (GC) process for both LSM-trees and flash storage. LSM-trees need a GC process to manage flushes and compactions. Flash storage needs a GC process to erase blocks and move live pages (if any) to other blocks when blocks are reset. Both are not entirely disjoint, which can lead to duplicate work and coordination issues. Where possible, the two should coordinate and decide on the block(s) data will be written to and how and when data should be removed.

In the current situation, the GC of an LSM-tree can by default not delete or write data by itself. Instead, it relies on other software to do it, such as an FTL or a file system. In other words, whenever the LSM-trees orders an SSTable to be deleted on a compaction, the deletion itself is handled by other software. Eventually, it will arrive at the GC of the SSD, which then has to decide what to do with the deleted pages. The LSM-tree can not order the data to be physically deleted and can not determine when the data is deleted. This can happen at any time, which means it can also happen at peak activity of the key-value store. Ideally, it should happen at a time that is convenient. For example, when the key-value store is idle. This can only be done if all software layers between the store and storage can communicate about how and when deletion should be done. However, this is generally not possible as it requires all layers between the store and the SSD to support the needed communication features. This coordination issue is not limited to erasures, the same thing happens for writes. For example, when an FTL is used, the LSM-tree has no control over the physical location(s) of its data. It can be stored anywhere. The LSM-tree is designed in such a way that there is a clear hot and cold separation of data: WALs are hot, L0 is colder and LN is the coldest. Using such a hot and cold separation would be very beneficial for garbage collection on the device, as it can prevent storing hot and cold data together in the same blocks. Unfortunately, this again requires all layers in between the database and the SSD to be able to communicate this separation and adhere to this separation (some layers might intentionally ignore the conveyed message). The result is that WAL data and LN data can be stored together in the same erasure blocks. Such problems are known as *semantic gap* issues (24, 107, 114, 135). Each additional layer between the store and the SSD leads to another semantic gap.

The semantic gap can lead to a number of issues:

- No control over the physical locations of data. Hot data can be stored together with cold data, even when an application knows they should be stored separately, which leads to more expensive GC operations in both latency and wear levelling.
- No physical control over the scheduling of background operations. Background operations of each layer do not have to be issued at predefined times and each layer can have its own background scheduler logic. This can lead to latency instability as schedulers from different layers can schedule expensive operations at the same time (amplifying effect) or idle at the same time (reducing effect).
- No control over data deletions. Additionally, each layer can have a separate definition of what a deletion actually entails; think about deleting entire structures, files or erasing blocks. LSM-trees can delete a structure, but the layers below might only physically erase a few of the data blocks of the structure on storage. The other "deleted" blocks of the data can be deleted at a completely different time or not at all.

There are multiple solutions to solve the semantic gap. One is to add all semantics that are needed for proper communication between each layer. However, this might be infeasible. For example, it would require an FTL to be aware of all of its possible applications and support them all. An alternative is to simply remove the layers and collapse their features on top of each other. In this paper, we will take a look at this approach and study its effects. ZNS is already an example of such an approach. It leaves the GC up to the host, which allows optimising the GC for one specific application, such as an LSM-tree-based key-value store. Therefore, it can bridge the semantic gap.

However, there is one problem left. The FTL is not the only layer that can exist between key-value stores and storage. Many key-value stores make use of a *file system* as well (28, 51). Instead of storing the components of an LSM-tree directly on the SSD, they are built on top of files. For example, a WAL can be stored within a file, which is then itself stored on multiple erasure blocks/ZNS zones. This provides a higher abstraction on top of SSDs as key-value stores no longer have to deal with FTL/ZNS specifics. At the same time, it does require the file system to have enough functionalities to communicate the LSM-trees needs to the storage interface. Therefore, hot structures might not be differentiable from cold structures by the file system and be defined as generic files. The result is that they can still be put together in the same zones. Only once all layers between storage and key-value store are truly removed, can the GC of the database and storage be linked together. The main disadvantage is that the key-value store implementation will become storage implementation specific and will require more functionalities than before. It goes against the UNIX philosophy in this regard (48) and creates a more complicated design. In the end, it is a trade-off. In this thesis the effect on the GC will be examined when all layers are removed.

## 2.4 Storage in User-space

There exist multiple libraries and APIs that can be used to communicate with ZNS storage from an operating system (2, 3, 120, 133). As of 2022, the ZNS technology is still relatively new for SSDs, which means that not every storage engine supports all ZNS functionalities for SSDs yet. However, this might change soon. In the end, many ideas and optimisations proposed in this research should be applicable to most other storage engines/operating systems, as long as they adhere to the ZNS specifications as stated in the NVMe ZNS specification (1). That is because we focus on ZNS optimisations in this work only. Yet, there can be significant performance and scheduling differences between implementations, which is something to look out for. For example, in GNU/Linux it is possible to set a scheduler for NVMe SSDs. Some schedulers serialise I/O and can, therefore, not make full use of the parallel I/O capabilities of ZNS SSDs such as appends (120). Further on, some APIs might depend on such a scheduler to be present. Using a serialising scheduler such as *deadline* in GNU/Linux does not allow sending multiple outgoing append commands to the same zone and can, therefore, not reach the full performance benefits of appends. Therefore, in order to get full performance benefits, it is recommended to pick a storage engine solution that allows getting full concurrency capabilities and gives full ZNS control.

In this research, we want to get maximum control over the SSD, scheduling included. As one of the goals is to get latency stability, expensive kernel operations should preferably be minimised as well. Therefore, for TropoDB it was decided to run the SSD within user-space, to get maximum control (133). Running it entirely in user-space allows controlling the entire I/O path from beginning to end (minus a few minor steps) and allows having more influence on latency stability. This is possible with the *Storage Performance Development* (SPDK). The SPDK framework unbinds the SSD from the kernel and allows using the device as it is in user-space instead. This removes the kernel from the I/O path, which can reduce more latency issues related to the kernel and removes one additional layer. It also adds more control over the I/O scheduler and its latency. Internally, SPDK only has a basic scheduler that makes use of lightweight threads and revolves around polling from applications themselves. This works well with the latency characteristics of NVMe SSDs and can allow applications to decide for themselves when to poll. It also prevents excessive context switches and their unpredictability. Lastly, SPDK is also considered to be the state of the art and at the moment one of the fastest I/O solutions out there, both for synchronous and asynchronous I/O (40, 74). Lastly, as it is not bound to standard kernel development practices, it is more up-to-date and already has full ZNS support for SSDs. Using SPDK should be a co-optimisation for the key-value store design, but not required to reproduce most of the results presented. It can reduce latency and increase throughput, but it is mainly used to help with stabilising latency and GC effects.

## 2.5 Related Work

Key-value stores for flash storage are a widely researched topic as can be seen in "Key-Value Stores on Flash Storage Devices: A Survey" (41); a survey that was conducted as preliminary work for this thesis. Additionally, ZNS is a novel storage technology that attracts the attention of various research communities. It has been stated that ZNS storage is beneficial for key-value stores and that optimising key-value stores for ZNS can be beneficial. There exists concurrent work that looks in the potential of ZNS for key-value stores, but TropoDB is the first key-value store built entirely on ZNS. In this section, we will describe adjacent works.

### 2.5.1 Key-value Stores for ZNS SSDs

There already exists a full LSM-tree implementation for ZNS devices (33), but this LSM-tree is optimised for garbage collection of the storage itself and not for key-value stores. Purandare et al. come with some advice and design considerations on how to implement LSM-trees more efficiently on ZNS SSDs (107). It also comes with some general research directions for LSM-trees on ZNS SSDs. The proposed ideas can aid with the design of our solution for key-value stores and our implemented solution can in turn also aid the research by Purandare et al. by addressing some of the proposed research directions. For example, they mention the layout of LSM-trees, how to manage different data lifetimes, compaction procedures, indexing, filter and metadata, and how to alter internal data structures of LSM-trees when used on ZNS devices. However, none of these ideas are implemented yet. There also already exists a key-value store that can run on ZNS, but this key-value store itself has no knowledge of ZNS, nor is optimised for it. That is because this implementation runs the key-value store RocksDB (46) on a ZNS-friendly file system, known as ZenFS (24). This is possible because RocksDB can use file systems without knowing how they are implemented. Therefore, other ZNS-friendly file systems such as ZoneFS can also be used (78, 107). Stavrinos et al. discuss various design considerations for applications that make use of ZNS devices and some general research directions (117). They also mention that key-value stores could benefit from ZNS devices. There does exist some concurrent work that is looking at how parts of the LSM-tree can be improved for ZNS storage. Jung et al. introduce *lifetime-leveling compaction* (LL-compaction), a compaction tailored for ZNS storage that can reduce space amplification effects and GC effects by storing SSTables with similar lifetimes together (67). TropoDB also stores SSTables with similar lifetimes together, but takes a different approach to do so. Lee et al. propose a different zone allocation algorithm for ZenFS (81), *Compaction-Aware Zone Allocation algorithm* (CAZA). This work is more adjacent to ZenFS than it is to TropoDB, but comes with interesting strategies to deal with separating SSTables based on lifetime. It stores SSTables that are likely to be compacted in the near future together instead of storing SSTables based on their level, which is the strategy used by ZenFS and TropoDB to separate on lifetime. It also comes with a number of other strategies for allocations. These strategies and ideas are orthogonal to TropoDB and could have also been used in its design. TropoDB separates SSTables on lifetime similar to ZenFS, the CAZA algorithm could, therefore, have also been used in the design of TropoDB.

None of the aforementioned ideas fully addresses our issue and as far as we know there does not exist any LSM-tree key-value store that is specifically implemented for ZNS NVMe devices yet (as of August 2022). We will be the first to create one. All existing solutions make use of file systems in between the store and the storage or look at only a part of the LSM-tree.

### 2.5.2 Key-value Stores for Open-channel SSDs

ZNS can be seen as a successor to *Open-channel* SSDs (OCSSD) (24). Similar to ZNS, OCSSD gives full control over the SSD to the host (25). However, the model it uses to give this control is different. Nevertheless, key-value stores have been created for OCSSDs and many of their design decisions can apply to ZNS as well. A few examples of key-value stores that come with designs that are still partially applicable to ZNS are LOCS (124), FlashKV (135) and NoFTL (123). Those designs come with various solutions on how to map key-value stores to open-channel storage. For example, using as many memtables as there are channels to increase parallelism and match the full bandwidth that flash storage can leverage. Other techniques include scheduling operations based on their cost. For example, LOCS proposes assigning costs to reads, writes and erasures and scheduling these operations based on their cost. Perpendicular to this scheduling technique is scheduling as done by FlashKV, where operations can be prioritised based on the workload. For example, prioritising reads over writes if there is enough space left, but prioritising compactions if forcing compactions becomess critical. Scheduling these types of operations is a bit more problematic for ZNS, as you can not directly do the scheduling for the channels themselves. Nevertheless, it is still possible to schedule operations appropriately and the knowledge gained from this key-value store design is useful. For example, it is still known that scheduling multiple I/O operations at the same time, will lead to interference. Further on, ZNS gives control over all I/O operations issued, resets included. Therefore, it should be possible to schedule resets at appropriate times. For example, the design proposed by FlashKV could still be attempted by giving the compaction procedure more zones to use if compactions become critical or to allow resets only when they are critical.

### 2.5.3 Key-value Store LSM-tree Garbage Collection

A goal of TropoDB is to improve the garbage collection process of an LSM-tree-based key-value store. Garbage collection of LSM-trees is a known problem. Especially compaction procedures are problematic. TropoDB is definitely not the first to address this issue and many earlier concepts can be reused to aid in the design of key-value stores for ZNS SSDs as well. Various ideas have been proposed to reduce the effect of garbage collection procedures that are not necessarily limited to approaches that relate to the storage medium. We will look at a few interesting solutions. Only the first solution, will be tied to the storage medium and is most interesting for TropoDB.

The first solution that we like to discuss is improving the scheduling of compaction procedures. This idea does not aim to remove compactions, but to do compactions at the right time. It should be scheduled in such a manner that read and write operations receive as little hinder as possible from this background operation. For example, if compactions are performed too late, flushes have to wait before L0 is properly compacted to L1. On the other hand, if they are performed too early, compactions might not fully use the buffering capabilities available and compact duplicate data (hot entries that keep reoccurring). One such approach is proposed by Balmau et al. and is known as SILK (15). SILK properly links the compaction operations to the internal flash parallelism. It always prioritises compactions on L0 because these compactions are most relevant for client operations and it allows preempting compactions. This means that for example a compaction on L2 can be stopped, to perform a compaction on L0 instead. Compactions also only get a part of all available flash bandwidth, allowing clients to always make some progress, even when multiple compactions happen at the same time. This idea is shown to stabilise the latency of the key-value store and shows no latency peaks like would happen without such a scheduling procedure. The scheduling procedures described in Chapter 2.5.2 are orthogonal and such scheduling designs can also be used for ZNS optimised designs.

A common approach to reduce garbage collection cost is to move away from the standard levelled LSM-tree design, towards a *tiered* design, also known as a *LSM-forest* design (89, 97, 108). With a forest design, each level of the LSM-tree can contain overlapping SSTables or multiple logical LSM-trees. With such designs, it is not necessary to do full compactions and partial compactions are allowed. Partial compactions can reduce the effects of congestion in the LSM-tree, reducing stalls for the client. This might, therefore, also have been interesting to use for TropoDB as it can reduce the garbage collection effects. In the end, such solutions can reduce the write amplification effects of compactions, at the cost of increasing read amplification. Forest designs also suffer from other problems. For example, such designs can lead to more expensive garbage collection procedures later on. Additionally there exist partitioned forest designs (97), which come with yet again other trade-offs. The main point to make here is that LSM-trees do not have to be limited to levelled designs. The design can be altered if that makes sense for the use-case. There is a lot of research in this area. Such designs are not necessarily storage specific, but can aid in reducing the effects of garbage collection processes as a co-optimisation. This approach can use a garbage collection optimised tree design with a garbage collection optimised storage design. This has not been tried with TropoDB, but it might be worthwhile to also support LSM-forests on ZNS. Similar to how TropoDB has been optimised for a levelled LSM-tree, so can a different database be optimised for a LSM-forest or a different tree design altogether. There might be a tree design that fits ZNS more naturally (a zone tree?) that works better for reducing the effects of garbage collection.

Another common approach to deal with compactions is to delay compactions altogether. This is again not storage specific. A problem with delaying compactions is that while it might improve performance (latency, throughput, write amplification) in the short run, it is likely to get worse results in the long run. It can lead to significant spikes in long-tail latency (15, 29), which is the opposite of the goal of TropoDB. However, if done correctly delaying compactions can still be beneficial. TRIAD delays compactions based on heuristics (14). TRIAD only compacts from L0 to L1 if the overlap of tables in L0 is big enough, which can reduce write amplification effects of LSM-trees. However, it does generally lead to higher read amplification and can lead to a bigger more expensive compaction in the end. A larger more expensive compaction can cause tail latency to increase. Additionally, it comes with a novel strategy to reuse part of the WAL on a flush. Instead of creating an SSTable from the immutable memtable, it writes references to data in the WAL and keeps the old WAL around. Such a design reduces write amplification, but leads to read amplifications and fragmented data, as the WAL holds both live data and dead data. Such an approach will also make it non-trivial to do garbage collection on ZNS SSDs as it will eventually require compacting the WALs.

### 2.5.4 Key-value Stores in User-space

TropoDB runs its key-value store in user-space as a co-optimisation. It uses SPDK to achieve this. Removing the kernel from the I/O path can reduce interference from the kernel. TropoDB is not the first to do this and it is valuable to look at how other key-value stores implemented such an approach. Kourtis et al. proposed uDepot (74). This key-value store allows using SPDK to run in user-space as well. They note that it is able to achieve higher performance in both throughput and latency to run the key-value store in user-space. They also note that it is not always possible to have full control over the device, which makes user-space key-value stores unusable. They come with a custom framework that allows switching between I/O backends, some of which do not require running in user-space. This is not a target of TropoDB, but is in general advisable to do. Such an approach could also be used for TropoDB and is orthogonal.

### 2.5.5 ZNS SSD Research

Research on ZNS storage is not limited to key-value stores. There is also research on other applications for ZNS storage and research into the ZNS interface itself. Ideas proposed for other applications might be transferable to key-value stores and the other way around. Research into ZNS has implications for key-value store designs. If it turns out that a certain strategy for ZNS achieves low latency for reads, that should be used for key-value stores as well. Additionally, new changes to the ZNS interface can lead to new options becoming available for key-value stores as well. In this section, we will look at such ideas. There exist a few works that investigate the raw performance of ZNS. These works can be used together with our ZNS SPDK performance benchmarks to get a more accurate picture of ZNS performance. For example, Tehrany et al. come with some initial performance investigations for ZNS and file systems for ZNS (120). They benchmark with the libaio/psync storage engines and investigate the performance of various I/O schedulers as well. Whereas, for benchmarking ZNS we solely focus on SPDK benchmarks with SPDKs default I/O scheduler. We also do not benchmark raw file system performance, but solely benchmark file systems when used in conjunction with RocksDB. The benchmarks are, therefore, complementary. Nick et al. also state that larger I/O sizes are required to saturate the device bandwidth (120), which is important to know for key-value store design. Shin et al. also come to this conclusion (115). Additionally, Shin et al. come with other ZNS performance characteristics as well (115). For example, that performance of I/O operations differs based on the LBA, something that is not accounted for in the key-value store or measured for that matter. LSM-trees can address this characteristic by using LBAs that are able to achieve higher performance for more urgent matters. Bjørling et al. also come with various benchmarks for ZNS, RocksDB and file systems (24). They benchmarked with db_bench and measured among others throughput, average read latency, write amplification and workload runtime (time till completion of a large workload). Further on, they used the same workloads as were used in this research. Their research confirms the results shown in our work.

Key-value stores are not the sole target of ZNS research. For example, Bergman et al. investigate a swap subsystem optimised for ZNS, known as ZNSwap (18). This work comes with a garbage collector optimised for the use-case, that is able to achieve both better throughput and tail latency. Lessons learned from this research can be carried over to key-value store research. For example, ZNSwap allows for swapping between the custom data placement and zone reclamation policies (garbage collection) and comes with an API to add new policies. A similar approach can be taken for LSM-tree designs. For example, allowing the garbage collection policies to be altered dependent on the use-case of the key-value store.

There are also works that investigate modifications to the ZNS interface. For example, Han et al. propose ZNS+, an expanded ZNS interface (56). ZNS+ adds various modifications on top of ZNS, that make it more optimised for *log-structured file systems* (LFS). LFS and LSM-tree key-value stores have a lot in common, which means that ZNS+ also has advantages for key-value stores. In ZNS+ it is possible to offload data copy operations to the device and it is possible to overwrite data with sparse sequential writes. They state that this allows for an alternative reclamation strategy, known as *threaded logging-based block reclamation*. In the end, both optimisations allow for less communication overhead between host and device and both can also be used in LSM-trees. Therefore, it is valuable to also investigate building an LSM-tree on top of ZNS+. Maheshwari comes with an extension on top of ZNS that allows for variable-size pieces of data, known as *rocks* (94). Rocks can both be implemented on the host-level and the device-level. Rocks support an I/O granularity smaller than 512 bytes (a block size less than 512 bytes is not supported by NVMe) such as 16 bytes, which can aid LSM-tree design in a number of ways. For example, allowing small key-value pairs or small WAL updates to be written to ZNS with minimal padding. At the moment, designs like TropoDB are limited by size constraints, which hinder small I/O. Rocks prevent such issues. Lastly, Purandare et al. propose *group appends*, which also allow appends smaller or bigger than the page size (107). This leads to similar possible optimisations for LSM-trees as Rocks.

### 2.5.6 Summary

In this chapter we have looked at various related works. We have looked at key-value store designs for ZNS SSDs. While there are various partial implementations of LSM-trees on ZNS and designs for LSM-trees on ZNS, no full LSM-tree has been implemented on ZNS yet. TropoDB is the first one. We have also looked at key-value stores for Open-channel SSDs, and discussed various design decisions that are also applicable for ZNS.

We continued by looking at general work in garbage collection for LSM-trees. There, we described techniques to reduce garbage collection effects. We described techniques to schedule garbage collection operations, postpone garbage collections and reduce the cost of garbage collection. Such techniques are orthogonal and are also applicable to TropoDB.

We also shortly investigated key-value stores for user-space. The works presented here are orthogonal and can be applied to the design of TropoDB as well.

Lastly, we took a look at general research for ZNS SSDs. We have seen research in performance benchmarks, which showcase performance characteristics for ZNS, that have not yet been investigated for key-value stores. We have also seen designs for applications different than key-value stores. We finished by looking at alterations in the ZNS interface. If any of these alterations become part of the ZNS specification, they will have profound effects on LSM-tree key-value store design.

## 2.6 Summary

In short, flash-based SSDs are an affordable, fast and available storage medium, that are widely used. However, traditional SSDs come with an interface known as an FTL. An FTL is typically installed on the device and comes with a garbage collection process. The garbage collection process causes the QoS of the device to suffer and leads to latency and lifetime issues of the device. Additionally, it requires overprovisioning of storage. ZNS is an extension to the NVMe specification that addresses the issues of the garbage collection process and overprovisioning, by transferring control to the host software. This allows applications to decide for themselves, how to do garbage collection.

Key-value stores are a common type of database, that is also used in low-latency use-cases and use-cases that require latency stability. A common implementation makes use of LSM-trees. Internally, it also conducts garbage collection to move data around. A design that is said to fit SSD characteristics, but was not able to achieve optimal performance because of earlier-stated issues with the FTL and garbage collector. ZNS gives new opportunities and allows optimising a garbage collector specifically for both an LSM-tree and ZNS storage, effectively co-optimising both garbage collection processes.

Co-optimising garbage collection processes can be taken a step further by reducing additional layers in between the key-value store and storage. This can be done by reducing the kernel from the I/O path as well and moving the key-value store into user-space I/O. This effectively gives the key-value store full control over the storage.

We have also looked at various related works that have aided in the design in implementation TropoDB. There exist initial designs for LSM-trees on ZNS, but no implementations yet. These designs lay a foundation for TropoDBs design. While there are no LSM-tree key-value stores on ZNS, there are a few file systems for ZNS and general design guidelines for building applications on ZNS. Further on, there do exist key-value store implementations for Open-channel SSDs, the successor of ZNS storage. Additionally, there exist general research for ZNS to improve the interface and benchmark performance characteristics of ZNS. Lastly, there are various related works that have looked at reducing garbage collection effects of LSM-trees and building key-value stores in user-space.

# 3

# Analysis/Exploration of the Key-value Design Space

In this section we answer RQ1: *"What are the key-value store integration choices available on top of ZNS SSDs?"*. We will answer this question in a couple of steps. First, we will state the requirements needed by the thesis concerning the integration choice. This should limit the set of integration choices to the set of key-value stores that are able to achieve our goal. We will continue by investigating in-depth how ZNS devices and LSM-tree key-value stores can coordinate and what integration choices can achieve such a feat. This should also help in defining the problem space. Then we will investigate and look at the current solutions for key-value stores that already exist. The existing solutions will also function as the alternatives to evaluate against in Chapter 6. We will finish by stating all of the integration choices that we have considered for this thesis and what integration choice we have picked in the end. After having picked an integration choice, we will describe the design on top of this choice in Chapter 4 and the implementation of the design in Chapter 5.

## 3.1 Requirements of the Key-Value Store Integration Choice

The goal of this project is to measure the effects that the unique design properties of NVMe ZNS devices can give for the performance of LSM-tree-based key-value stores. For this the ZNS device and LSM-tree need to be able to collaborate. To achieve our goal we are only interested in the storage aspects of the LSM-tree, not in other parts of the key-value store. Though important for any practical key-value store implementation, in this thesis we do not focus on optimisations for memtable implementations, compression support or any other in-memory LSM-tree component, due to time constraints. An approach were we only focus on the storage aspects is also necessary, to accurately measure the effects of our approach. It should be possible to compare our ZNS approach with a default implementation not using this approach; to test the storage effect ceteris paribus. To be able to compare two approaches, it is required to support two databases (configurations) and a tool to measure the effect. To accurately match the effects of co-optimising GC, it is also required to take an approach that allows integrating the GC of ZNS with the GC of the key-value store. We thus define the following set of **requirements (REQ)**:

- **REQ1**: *Support for a common benchmarking tool*:
  We need a benchmarking tool that is able to benchmark both our solutions and the key-value stores that we intend to evaluate against. This requires a tool that is usable by both.

- **REQ2**: *Support co-optimising GC*:
  A goal in this thesis is to implement co-optimsising GC. Therefore, it is imperative to pick an integration choice that supports maximum control over garbage collection and allows integrating garbage collection of both the key-value store and ZNS.

- **REQ3 (soft requirement)**: *Limit the amount of work needed for in-memory components*:
  This is not a hard requirement, but something to prefer. We should not focus on in-memory components and limit our design as much as possible to storage solutions only. Therefore, we should pick an integration choice that adheres to REQ1 and REQ2 and is favorable for REQ3.

## 3.2 Key-value Design Problem Space

The goal in this experiment is to make a key-value store where the garbage collector of the key-value store and the SSD properly coordinate. This requires making a design that allows such coordination (REQ2). A design should either allow the GC of the key-value store and SSD to properly communicate or the two should integrate with each other as one garbage collector. This communication is generally hindered by one concept, the *semantic gap*. A concept we have explained in Chapter 2.3. In short, the semantic gap means that two different systems are not properly able to communicate as they do not fully understand this other. Additionally, we will now identify another problem that can occur because of the semantic gap. The semantic gap denies making specialised designs for storage components in applications such as key-value stores. A few components require certain performance characteristics, for example, WALs need excellent write performance, but read performance less so.

We now know what the problem in the problem space is, but there are multiple approach to solve the issue. There are two general approaches to improve the communication: add support for possible communication or remove layers. The first problem is problematic as it requires complex layers that have support for all applications that make use of it and is generally unfeasible. The second approach, requires a vertical integration of the layer into the application itself. This can improve garbage collection at the cost of development time and maintainability. It is a trade-off.

With key-value store designs we identify a number of layers: the FTL and in some cases file systems. Using ZNS removes the FTL as a layer, but it is not always the only the one between the key-value store and the SSD. Key-value stores typically also make use of a file system. Using a file system in between the store and storage, makes communicating garbage collection problematic. File systems do not come with explicit ZNS commands by default or specialised data structures for WALs. File systems can define for themselves when to erase data and support generic files. It is possible to create a file system specific for a certain domain. ZenFS, for example, is built as a plugin for the key-value store RocksDB (24). However, ideally for this research we want to have a more informed communication between the store and storage. For TropoDB we want to remove parts of the file system as well where applicable to get a real collaborative-GC in which the key-value store can direct control the GC of the SSD. This should address REQ2 in an optimal manner. *We thus propose a design without any semantic gaps.*

## 3.3 Current Solutions in the Problem Space

It is important to test against a key-value store that is already up to standard to make a fair comparison. It is not possible to test the effect of a collaborative-GC without also testing a solution without such an approach. Ideally, we should also be able to test various solutions with more semantic gaps. This would allow, measuring the difference that each gap can bring, rather than just looking at one gap. Therefore, we decided to test against the state-of-the-art that is still able to work on top of ZNS SSDs. This is possible with *RocksDB* (see Chapter 2.2.2). RocksDB is optimised for ordinary SSDs, but has no direct knowledge of ZNS. Nevertheless, it does not need to as it makes use of file systems to do the actual I/O (24). RocksDB can thus be used with ZNS-aware file systems.

File systems that are, as of May 2022, able to support ZNS include *ZoneFS* (78) and *F2FS* (120). F2FS is a file system that is optimised for flash (79) and already used frequently for conventional SSDs and can, therefore, be seen as the **state-of-the-practice** for RocksDB. It requires no further action for RocksDB to make use of this file system configuration, but this configuration is not optimised for ZNS devices and has a semantic gap in the file system level. This solution can thus suffice as the baseline performance for this project: a key-value store using a generic file system on top of ZNS.

However, RocksDB can also make use of *plugins* to circumvent relying on generic file systems. A plugin allows setting up a custom file system API and can thus be used to create a domain-specific file systems. This allows optimising a file system for key-value stores and reducing the semantic gap. One of such approaches is *ZenFS* (24). This is also a file system optimised specifically for ZNS storage. This can thus be used as the **state-of-the-art** for benchmarks. As both file systems make use of the same key-value store, it also allows for a more proper evaluation. The last advantage of RocksDB is that it already comes with a benchmarking tool, *db_bench* (28). This tool ensures that there already is a well-defined benchmarking tool that can be used for measuring I/O characteristics.

## 3.4 Key-value Store Design Integration Choices

The next decision to make is how to built a key-value store on top of ZNS with SPDK and design/implement TropoDB. We wanted to make use of the existing benchmarking tooling to allow for proper comparison against the alternatives. This requires a tool that can benchmark both the RocksDB solutions and the TropoDB solution. Therefore, the following four solutions were considered, respectively in order of effect and required work:

1. Create another domain-specific file system for RocksDB with a plugin. This allows leveraging all RocksDB functionalities, but it does mean that there is limit to the effect GC can have. It is not possible to properly differentiate between different LSM-tree data structures as they are all stored in generic files. It only allows specifying file systems hints indicating their *temperature*. It would also not be possible to truly combine the GC of the key-value store and the file system. In the end with this solution, there will always be a generic file and a semantic gap will remain.

2. Alter RocksDB to make use of specialised storage data structures instead of files. This would allow making custom and optimised structures for each individual database component. This would essentially nullify the semantic gap. At the same time,

RocksDB was made to make use of a file system API. RocksDB comes with many options, meaning these all need to be implemented or their support needs to be dropped. This approach requires an intrinsic knowledge of each RocksDB operation and requires reworking many parts of the database from the ground up, even if they are not relevant for the research.

3. Built a new key-value store on top of the RocksDB interface. RocksDB comes with a basic interface (defined in the C++ header file *db.h*). This allows leveraging the entire RocksDB ecosystem. Whenever, a function from RocksDB needs to be used, it can then be included. At the same time it gives full control over the SSD removing the gap. This is similar to altering RocksDB, apart from its approach. This approach is bottom-up instead of top-down. This allows beginning with storage aspects first and only adding what is necessary. As RocksDB is a continuation of LevelDB, it also allows using LevelDB components instead of RocksDB components in some cases. LevelDB components are generally simpler in design and come with less options. This makes them easier to add and modify for ZNS.

4. Built an entirely new key-value store and modify *db_ bench* or another benchmarking tool to support the key-value store. This fundamentally gives the most control. However, it does require reworking each part from scratch. This also includes the non-storage parts of the key-value store and the benchmark tooling. Such an approach would make it non-trivial to compare the solution against the alternatives and makes it easy to make an *apples to pear comparison* instead, which is generally not what we want.

In the end, approach (**3**) is taken for TropoDB, creating a new key-value store on top of the RocksDB interface with a bottom-up approach. This gives a lot of control and reduces the semantic gap (REQ2), while still retaining access to the RocksDB ecosystem(REQ1,REQ3), which is already researched by a large selection of academia and industry alike. Creating a file system, approach (**1**), is rejected as this would be very similar to ZenFS and not bring anything novel to the table. It would also complicate collaborative-GC as a semantic gap would remain (Fails REQ2). Approach (**2**), modifying RocksDB top-down, is rejected as well as it would require investigating all RocksDB I/O paths for each step. This would make it harder to test individual I/O structures and test the effect they would have on the key-value store. We still think that it would be a worthwhile approach to test out in future work. Approach (**4**), creating an entirely new key-value store, is rejected as it would make it hard to compare the key-value stores against alternatives (Fails REQ1). There is no other key-value store made purely for ZNS; so there would not be a clear baseline anymore. This in combination with the need to recreate everything from scratch, including the memory parts of the key-value store, would make it non-trivial to distinguish storage from other parts of the store and would in all likelihood lead to measuring interaction effects (Fails REQ3).

The resulting approach for TropoDB will lead to a minimal stack. As we also make use of SPDK, the kernel is also removed from the stack of layers, providing maximum control over the storage used. The amount of layers TropoDB has compared to the alternatives in the problem space can be seen in Figure 3.1. RocksDB with NVMe SSD has 3 storage layers and an OS layer, F2FS with ZNS removes one storage layer and ZenFS removes one layer and integrates another (domain-specific file system). TropoDB on the other hand removes all layers.

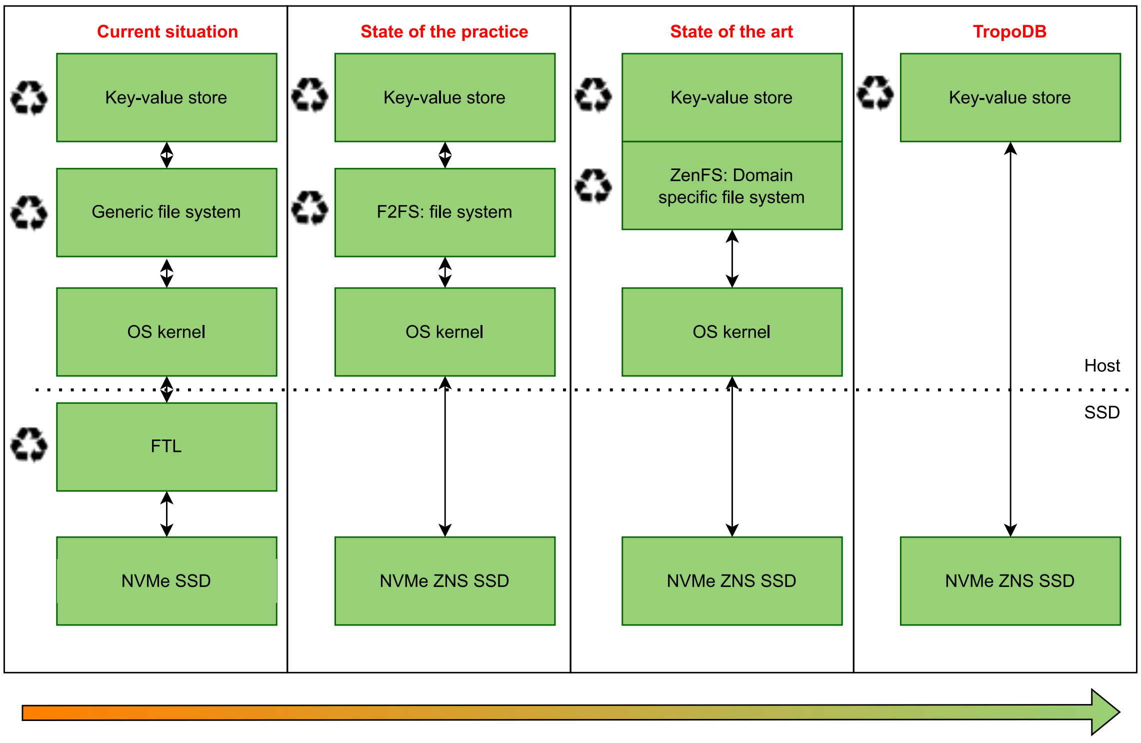


**Figure 3.1:** The semantic gap between key-value stores and flash storage

### 3.4.1 RocksDB for ZNS Design Choices

Following last section it has been decided to built TropoDB bottom-up on top of the RocksDB API This automatically gives full access to *db_bench* as well. We will now describe what else we need to do to implement the chosen integration choice and how we intend to achieve this.

As RocksDB is written in and has support for C++17 (64), it has been decided to make use C++17 as well. A part of the lower API stack of TropoDB still makes use of C code. TropoDB is thus a mix of C++ and C. By default RocksDB has no access to SPDK and especially not to ZNS. To allow calling raw ZNS SPDK functions, the application is linked to SPDK. To built the key-value store, logic has been added bottom-up. When a feature needs to be added, it is first investigated if this logic can be reused from RocksDB and LevelDB. Then if this feature makes use of storage, it is investigated how this structure can be optimised for ZNS storage and what needs to be altered to the already existing LevelDB/RocksDB logic. In many cases this requires creating new I/O structures as no solutions exist yet for ZNS. Not all LevelDB/RocksDB functionalities have been implemented, as they are not all necessary for this research. Instead, we only implement the minimum needed to get a functional key-value store. This means that there is logic for *put*, *get* and *delete* operations, but no support for *scan* operations or *column families*. Similarly, there is no support for non-ZNS specific optimisations. Such as database filters or memtable-only approaches (in-memory key-value store). The key-value store was in the end built in three steps: implementing the basic key-value store, allowing the data base to be used with benchmark tooling such as db_bench and optimising the individual key-value store components based on large I/O benchmarks. The resulting key-value store is known as *TropoDB*. In later sections, we will take a look at the individual components, how they are designed and how they are implemented, and then we will take a look at how the resulting key-value store behaves as a whole and how it can be configured.

### 3.4.2 Splitting Key-value Store Logic from Generic ZNS Structures

During development it was noticed that not all structures and storage logic that was created up until that point in time were ZNS specific. In fact, many parts could be reused by other ZNS projects. Therefore, it was decided to split the project. This resulted in a separate library, apart from TropoDB, that could be linked to statically and dynamically by the database, *SimpleZNSDevice* (SZD). This contains generic logic that can also be reused for other projects such as file systems. It comes with the additional benefit, that it allows changing the backend of the database. This would allow optimising the storage aspects at least partially independent of the key-value store at the cost of introducing a new layer.

### 3.4.3 How to Benchmark the Solution and the Effects of GC?

Since, we are benchmarking key-value stores on SSDs specifically and want to also verify write amplification among others, we have to make sure that we benchmark correctly (REQ1). Just executing db_bench, will not be enough because, db_bench, only measures the top-level effects of the key-value store. It measures latency of I/O operations, compaction cost in latency, write amplification of the LSM-tree itself and throughput. However, it does not test raw effects of I/O. This is necessary, as we are not testing just one *layer*, we are testing the entire stack; from key-value store to storage. Effects on storage need to be measured as well. Therefore, in order to properly benchmark, some additional custom tooling needs to be added on top of db_bench to measure the storage effects. All possible effects GC need to be measured. This includes latency of puts, tail latency of puts, number of writes to storage, number of reads from storage and number of erasures issued to storage. To benchmark storage correctly, guidelines in *the unwritten contract of SSDs* (57) and *Toward a better understanding and evaluation of tree structures on flash ssds* (39) are followed where applicable. However, guidelines concerning garbage collection and a steady state are harder to follow. That is mainly because the solutions used have inherently different designs that are not completely compatible with each other. For example, TropoDB will claim the entire SSD and clean it before it creates a database. Therefore, it is not possible to bring the SSD to a steady state with other operations than database operations. Instead, it is only possible to bring the database to a steady state and reuse this database for later tests. To make such comparisons fair with designs using file systems, the file systems should test with a clean slate as well and only reach the steady state by using database operations. In general, all candidates for the benchmarks should start out in the same state. In the end we identify the following **measurable GC effects** that need to be measured by all solutions:

- **Measuring write amplification (WA) of the entire key-value store**:
  We define total write amplification as the total amount data written to storage during a benchmark divided by the size of all key-value pairs written to storage (number of put operations and raw size of each pair). Thus all bytes written on storage divided by all data put by clients. We use this definition as the solutions live in a different design space. Both the garbage collection process of the key-value store (application-level) and the device (device-gc) can cause write amplification. TropoDB will not cause write amplification on the device level (or an amplification that approximates 1) as the key-value store has full control over the storage. All amplification is thus issued by

the application and can not be logically separated. Similarly, the RocksDB solutions only record write amplification at the application level and using this metric would thus be wrong. To properly compare the two solutions, it is necessary to incorporate all write amplification.

- **Measuring normalised reset count**:
  We define the normalised reset count as the total amount of zones reset during a benchmark divided by the size of all key-value pairs written to storage (number of put operations and raw size of each pair. Resets are an explicit garbage collection operation. There purpose revolves around it. The garbage collection process that has control over the storage, resets the zones. This should be measured.

- **Tail latency of client-issued key-value store operations**:
  Garbage collection processes do not just write and move data, they also claim available hardware resources. For example, they use part of the available bandwidth (and available parallelism) of the device. This reduces the amount of resources that are available for other resources. Additionally, some garbage collection operations are time-sensitive in the sense that clients have to wait for them to finish. For example, if there is no more space left, the client has to wait for space to become available. It is hypothesised that during expensive garbage collection processes, this will lead to fluctuations in latency for client-issued operations as clients might have fewer resources or are stalled. This metric will also contain various other latencies (latencies of in-memory operations and more) and means little on its own. However, when we compare comparable key-value store solutions, it should possible to recognise garbage collection effects (hypothesised).

### 3.4.4 Summary

In short, we have answered RQ1 by looking at various key-value store integration choices that are available on top of ZNS SSDs. In the end we picked an integration choice where we built TropoDB on top of RocksDB with a bottom-up approach. In the bottom-up approach we start with the database interface from RocksDB, but do not start with any additional RocksDB logic by default. Instead, we add functionalities on top as needed. Functionalities are taken from RocksDB, LevelDB or created ourselves. In particular, all storage components will be crafted by hand. Storage will be interfaced with the help of SPDK. The solution will be benchmarked with RocksDB's benchmarking tool, db_bench.

TropoDB will be compared against the state-of-the-art, RocksDB + ZenFS, and the state-of-the-practice, RocksDB + F2FS.

# 4

# Design of TropoDB

In this section we will answer RQ2: *"What unique optimisation opportunities does a ZNS device offer for LSM-tree based key-value stores?"*. We are thus mainly interested in the unique design properties that ZNS can give for LSM-tree key-value stores. For example, co-optimising garbage collection for key-value stores by reducing the semantic gap. Most design decisions that are thus discussed in this paper are only related to these issues and other optimisation efforts are left to future work or only discussed shortly. In this section, the major design decisions of TropoDB are discussed. We will first give an overview of what we need to design, along with their requirements. Then we will describe how each individual LSM-tree component is designed in TropoDB, followed by how all background operations are designed and we will finish by describing how the individual components interact with each other. Components that make use of storage will be described in more detail.

## 4.1 Overview and Requirements

Before we come up with a design, we need to get an understanding of what we need to design. To achieve this we look back at the LSM-tree as defined in Chapter 2.2.1. An LSM-tree consists of a couple of structures that need to be physically stored. Those structures are *Write-ahead logs* (WALs), *Sorted String Tables* (SSTables) for L0 to LN and *metadata for the index structure.* We need to create a different data structure for each different component.

We also define a number of background operations that we need to take care of. These background operations are vital as they are in charge of the garbage collection. For key-value stores these include *flushes* and *compactions.* Additionally, we define *deletion* operations. Operations that are in charge of removing old data. This is necessary because the key-value is in charge of physical deletion according to the design approach we have taken for TropoDB in Chapter 3.

We setup a few requirements that the designs of each the components and background operations should adhere to. These are based on features that make ZNS stand out. ZNS stands out because of its support for append operations, explicit parallelism with active zones and appends and the ability to create a custom garbage collector. To ensure that all of these properties are tested, so that we can answer the research question (RQ2), we want to force the design to use these unique properties. Therefore, we define the following **Design Requirements (DR)**:

- **DR1**: *The design should be append-only*:
  The design is not allowed to make use of the traditional *write* command to perform writes. It is only allowed to make use of the ZNS I/O operation *appends.*

- **DR2**: *Reduce the number of overwrites and erasures*:
  ZNS gives full control over the physical locations of data. Every write and reset has to be issued by the client. This is also true for garbage collection as this is in the hands of the client. This should allow for designs that minimise overwrites and resets of data (in-place I/O). For example, by using logs and hot and cold separation. Every design that uses overwrites in any way, needs justifications.

- **DR3**: *Make effective use of the available parallelism*:
  ZNS devices allow doing I/O operations in parallel. For example, multiple appends to the same zone and using multiple zones concurrently. Not using the available performance would waste what ZNS can give. Where possible, we should try to use what is given to us by the device.

- **DR4**: *Limit interference of garbage collection to aid latency stability*:
  With the co-optimised garbage collection approach taken in this research it is possible to decide when to do garbage collection operations for both the LSM-tree and storage. This can lead to better control over latency. Where possible, the design should take this into account and stabilise latency effects. For example, by not postponing deletes to the last moment.

For each of these components and background operations we will also come with a structured analysis and description. These details should help to get a better understanding of what each structure exactly is, how it is currently stored and how we can improve this with ZNS. For each component we will at least describe:

- how the component functions and is used (access patterns)
- how the component is traditionally stored in a file system
- how the component is stored in TropoDB
- how the component is cleaned (garbage collection)

Additionally, we will explicitly mention how each component tries to adhere to the design requirements and come with some disadvantages for each proposed design as well. We will also properly identify what the semantic gap for each LSM-tree component is, this is necessary to state as this is the main problem that TropoDB's approach can solve. We should design our components around that principle. For example, if we want to reduce the effects of the semantic gap, we need to make components that are aware of the temperature of their data and if we want to reduce the effects of garbage collection, we should use proper scheduling for cleaning operations.

Throughout all of our explanations, we will refer to RocksDB and its precursor LevelDB as the traditional approaches. We will also refer to the state-of-the-art file system ZenFS for some components, to explain how using ZenFS as a file system can alleviate the effects of a stated problem. Justifications for many design decisions are supported by experiments/evaluations described in Chapter 6. This includes both advantages and disadvantages of the designs chosen.

## 4.2 TropoDB as a Whole

TropoDB is a collection of individual components that together can be used to create an LSM-tree. We will described each of the individual components independently in later sections. In this section, we will describe how the components together make up the LSM-tree and what configuration designs should be considered. It thus gives an overview of the design philosophy of TropoDB as a whole. Each individual component in TropoDB reserves a contiguous number of zones. This results in a design where the entire SSD is split between components and there is no additional space left. Such a result can be seen in Figure 4.1. In such an approach, it must be approximated or known beforehand how much data each component is expected to use. As this can differ on the workload used for an application or the storage device used, it is to be made configurable, but only on initialisation. After initialisation, it should be trivial to find the location of each component on storage.

Clients of such a database must make a trade-off between the components. Reserving many zones for metadata might be wasteful, but too few zones will create problems for large key sizes or devices with small zones. Similarly, assigning to few zones for WALs might result in faster wear levelling on a few zones, but will leave more space for SSTables. Further on, each component requires I/O and the I/O each component uses reduces the amount available for others. In the case of active zones, it can even cause too little zones to be available. Another trade-off is, therefore, to prioritise what components should get the highest level of concurrency (addressing DR3). In short, TropoDB removes most abstractions and physically stores the key-value store components to exact locations. Users of TropoDB are thus left to make choices based on the available hardware, not on higher abstractions.

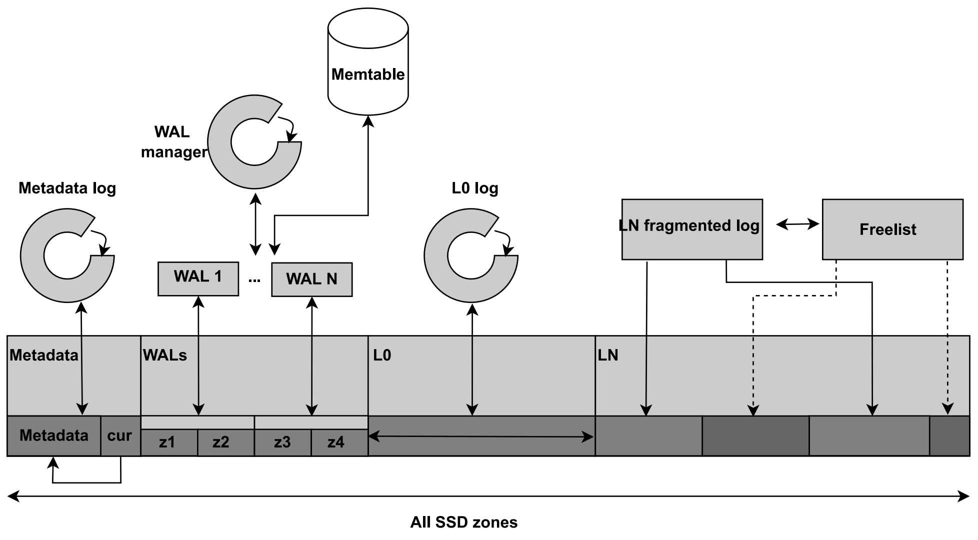


**Figure 4.1:** TropoDB design: Dividing an SSD among LSM-tree components

## 4.3 Write-ahead Logs (WALs)

*Write-ahead logs* are structures necessary to guarantee persistence of the database. In particular, they are the persistent part of memtables. They do this by functioning as append-only logs and recording all of the changes applied to the memtable. Examples of such changes are: insertions, modifications and deletions of key-value pairs. A put operations is only considered complete when the put data is written to both the memtable and the WAL. The result is that WALs are very hot and one of the most I/O heavy data structures of the key-value store. This makes them an important target for optimisations (it is on the common path). This is also true for garbage collection and latency stability issues. When looking at how WALs are used in Chapter 2.2.1.1, an important detail to notice is that for LSM-tree WALs, only the client appends to the structure. Further on, during the WALs lifetime it is generally never read. It would thus be accurate to say that the WAL is write-only, except on startup when it is only read. Therefore, when designing a WAL, we only have to take care of two situations: a situation with only writers and a situation with only readers. In addition, writes generally need to be fast as otherwise clients have to wait longer before their put requests are finished, while reads may be slower as they are only done once at the start or during a recovery (common and uncommon I/O path).

### 4.3.1 File System Approach for Designing WALs

To design WALs for ZNS, we first take a look at how WALs are currently implemented with a file system approach. We refer to the WAL implementation as used by LevelDB and RocksDB. In both implementations, there can be numerous WALs on storage. The maximum amount of WALs is configurable and each WAL has its own file, that can easily be identified with a filename like ".LOG" (46). However, only one WAL is active for writes at each point in time. Old WALs can be removed by a background thread when they are no longer needed, which happens when the data is also present in an SSTable and is, therefore, no longer necessary. We can at this point state that the data is backed to storage twice. Data only needs to be present on one layer of the LSM-tree.

By giving WALs unique files, they are already clearly separated from the rest of the data. In fact, individual WALs are also separated as they are stored in different files. Both key-value stores identify these files as *append-only files.* The problem with this approach is that the underlying file is still just a general file and not optimised for its use case, regardless of how it is used. Similarly, even though WALs are identified as unique files, there is no control over the physical location of the files. Lastly, when a WAL file is removed, it might be removed at a later point in time, providing little control over the actual removal process.

Using a domain-specific file system such as ZenFS can alleviate the issue. ZenFS tries to put different files in different zones (24), which should allow for better hot and cold separation. Yet, they still have to resort to using only one type of file, which is optimised for the general use case. ZenFS can not come with unique removal schemes for WALs or unique implementations for WALs because it needs to generalise. When it receives a file, it does not know whether it is a WAL or another file. ZenFS is a domain-specific file system, so this can be solved by adding semantics between the file system and the key-value store to indicate the file type (e.g. extended attributes).

Another property that LevelDB and RocksDB support are various levels of persistency. For many applications, persistency guarantees are less strict than performance guarantees. Trading persistency for performance can be a good trade-off in such situations. By default, persistency guarantees of both databases are not that strict. The resulting persistency model makes use of buffering and asynchronous I/O. Writes to the WAL are first buffered to a small application buffer. Only once the buffer is filled or the application is shutdown, will it be written to the file system. To get more reliability, this buffer must be disabled manually by the client. However, this might not be enough. By default, writes are only issued to the underlying file system. However, the file system can have a buffer of itself and is not forced to flush the contents of its buffers to storage immediately. Further on, depending on the storage engine used, it is also not guaranteed that all I/O sent to the SSD will be immediately written to flash storage. This can also lead to data loss on power loss of the machine, but data should generally still be written to storage on a graceful shutdown. More reliability can be ensured by forcing the databases to *sync*, which forces the underlying file systems to flush their contents and use direct I/O (e.g. fsync on GNU/Linux (26)).

### 4.3.2 TropoDB Approach for Designing WALs

In TropoDB we take an entirely different approach. We reserve a part of the zones to be used for WALs exclusively and each individual WAL gets its own distinct contiguous subset of zones. This separates hot and cold data by creating one cluster that spreads the temperature of WALs (addressing DR2). The WAL implementation is optimised for its use case only and uses a highly specialised data structure that we call a *once log*. Thus we specialise, instead of generalise. It is called a once log because its use case is to be used only once and then thrown away. The log is internally made up of a contiguous set of zones, where there are three possible types of operations: appends (DR1), random reads and complete erasures (throwing the log away). As there are only appends, the log can only grow in one direction until it is full. At that point in time, we have to claim a fresh WAL, or else we will lose data.

**Consistency models**

Similar to how RocksDB supports multiple consistency levels, we support multiple consistency levels. In TropoDBs design it is possible to configure a small application buffer to stall issuing writes to ZNS. When the buffer is made larger, more data can be lost, but I/O speed can increase. We also support the earlier mentioned *sync* option. When the option sync is disabled it is not necessary to immediately flush the content of a put to storage (direct I/O), as long as it is queued to be flushed at some point in time. In TropoDB, we do this by only queuing the contents of the put, but we do not wait for the write to finish. The request is only added to a submission queue, but no polling is done yet. RocksDB also guarantees that in case of a crash of RocksDB, the write will still persist, as long as the underlying file system does not crash. We make no such guarantees for TropoDB as we will need proof to make such claims, which makes our design a bit less reliable when sync is disabled. Future work can look into how to make sure that writes still complete on a crash.

**Effectively using the append command**

The once log also uses a unique concurrency model that is designed to match the performance that ZNS appends could give. The structure does not allow concurrent reads or writes, which reduces the need for unnecessary and expensive locking operations. Yet, it does allow for multiple concurrent appends (which aids DR3). This is made possible by using the append operation because, as stated in Chapter 2.1.5, it is possible to issue multiple appends in parallel to the same zone, at the risk of the individual appends being reordered. The WAL is allowed to be reordered as long as the order is known at the time of recovery. The ordering can be recovered by storing some additional metadata that specifies the ordering. After reading the WAL, the data can then be reordered/sorted and reapplied to the memory table in order. There are multiple types of metadata that can be used to infer the ordering. We will explain a few of them shortly. We will begin with the methods that we did not pick and explain why we did not pick them as well.

When using multiple appends asynchronously it is not possible to know where the data is stored, this information is only made available once the append has completed, which makes it impossible to store information about where other current appends are stored to WAL entries. However, it is possible to know where completed appends are stored. This allows storing a small metadata table containing the physical location of all previous appends in the order that they were issued (to allow sorting). Naively, this would be stored along with the appends at the beginning of each WAL append. This would not work. A major problem with this design is that it does not work for all current outstanding appends. For example, if there are 4 outstanding append requests, those 4 append requests will not have each other's location in their metadata tables. They might not even have knowledge of each other. This is always a problem for the last 'n' appends and it would not be possible to find out the order of the last ‘n‘ appends. Instead, such metadata must be stored separately *after* each append completed to guarantee correctness. In this new scenario, the last completed append will have accurate information about its ancestors. It requires an extra metadata append after each successful data append to the WAL. So each logical append require two device appends. This comes with the additional benefit, that only fully completed WAL appends are accepted (only WAL appends followed by a metadata update are accepted). However, this does requires an extra append operation (Not good for DR2). Further on, the metadata appends themselves can also be reordered. Therefore, it is not enough to pick the most *recent* metadata table. Instead, the one with the largest number of entries must be picked or we need to store additional metadata to guarantee the most recent change is picked. Such metadata can for example be a sequence number or timestamp. In short, during a recovery multiple pages need to be read to find out the most recent metadata table update. Then this table will be used to read all WAL changes in order and apply them to the memtable in order.

TropoDB uses an alternative approach. The approach of appends as done in TropoDB for WALs is visible in Figure 4.2. It does not bother with maintaining where physical data is stored. In fact, each WAL append has no knowledge at all about any other append. Instead, at the beginning of each WAL append a small sequence number is stored. This sequence number starts at 0 for an empty WAL. After each WAL append operation, no matter if they are synchronous or asynchronous, the in-memory sequence number is globally incremented. On recovery, the in-memory sequence number is set to the largest sequence number found in the WAL log. With this approach, the order of all WAL appends can be inferred from their sequence numbers. An alternative to sequence numbers would be to make use of timestamps. In this approach each WAL update is stored along with the epoch time the WAL append was issued at. This also guarantees ordering of append operations as long as the clock is properly synchronised. Using sequence numbers or timestamps does make recovery a bit more complicated. The recovery procedure is visible in Figure 4.3. On recovery, all appends and their sequence number are first read. The appends are then sorted based on their sequence number and then applied one by one to the memory table in order. This design thus requires 2 reads for each append. However, it is not required to do 2 reads to storage. WALs are generally small and can be stored completely in-memory. Therefore, the WAL is first read in steps of MDTS (see Chapter 2.1.5) and then the WAL ordering operations are done completely in fast memory. Further on, the sorting is done on sequence numbers pointing to pointers which itself point to the actual locations in the memory buffer (the WAL is now in-memory), making the sorting cost negligible. It does not sort on the WAL data itself. In the evaluation as done in Chapter 6.4.2, we will see that the overhead of reordering an asynchronous WAL versus recovery for a synchronous WAL is not significant.

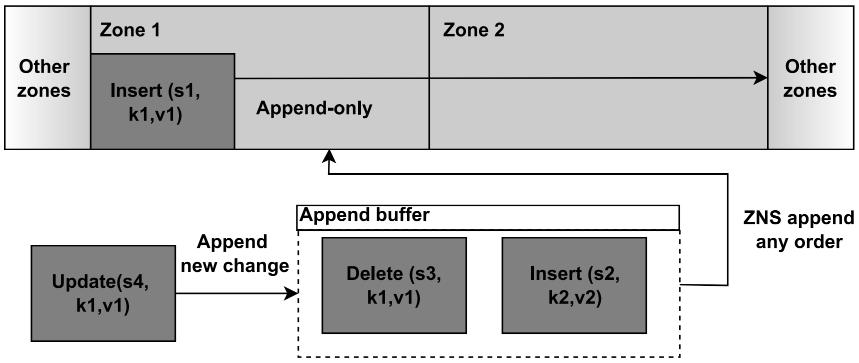


**Figure 4.2:** WAL design: Unordered appends to a WAL

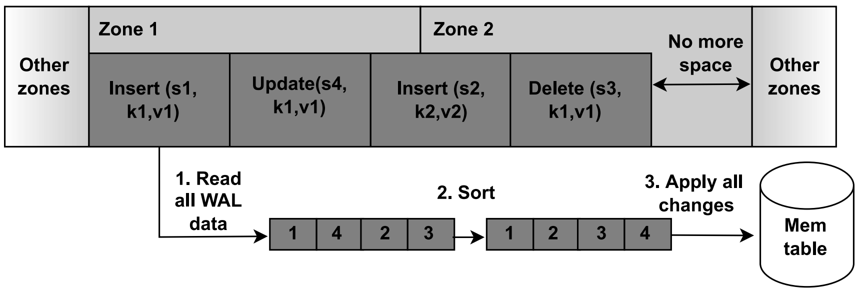


**Figure 4.3:** WAL design: Recovering an unordered WAL

Using versioning numbers to find out the ordering of “multiple” WALs is not new. This idea is based on ideas of earlier key-value stores that make use of multiple concurrent WALs. For example, SplinterDB (35). SplinterDB uses multiple concurrent per-thread WALs and increments and uses *cross-referenced logs* to achieve this feat. Each WAL operation increments a globally maintained generation number that is logged along with the data in the individual WALs, enabling a recoverable order. SpanDB takes a similar approach by using multiple parallel WAL write streams (32). Notice the similarity to how WALs in TropoDB store a generation number for each update as well to ensure ordering.

The proposed design makes using multiple appends a valid optimisation technique. It makes the hot path: appends, faster at the cost of making the cold path: recovery, slower. However, in some cases we do have to do some synchronisation (this is of course always the case when sync is enabled). This happens when the end of a zone is reached or a put requires more than one append. A put requires more than one append if its size is more than ZASL or the border of a zone is reached, which requires breaking up the append into multiple smaller appends. Synchronisation is needed here because the WAL does require individual WAL appends to be appended in order; all data of one unordered append is required be adjacent to each other. When I/O is split into multiple asynchronous appends, this can not be guaranteed. Alternatively, such big appends could have been further split into fragments with their own sequence numbers. Then apart from sorting writes, the fragments of each append have to be sorted as well. This approach has not been tried and left to future work (will aid for DR3). Instead in TropoDBs case, all outstanding appends in this case are synchronised until the write pointer is known and then only one append is requested for such I/O (forcing order for this append). Otherwise, we can safely enqueue multiple WAL updates asynchronously. Asynchronous appends are thus only possible at granularities less than ZASL. An outstanding asynchronous append request is reserved with the help of a *concurrent waiting list*. The concurrent waiting lists checks if the maximum number of appends is not reached before an append is issued and reserves a slot for this append, requiring minimal locking. When the maximum number of appends is reached, some minor synchronisation is needed as well. In this case the outstanding appends are polled until one append finishes and a slot becomes available. The newly available slot is then claimed and is used for a new asynchronous append.

Appends with an unordered order can be beneficial, but it might be better to use ordered solutions in some cases. For example, when reliability is required of the key-value store all WAL appends must complete before a put request is considered to be in a completed state. In this case, there is no asynchronicity to speak of and no queue depth. Therefore, unordered appends should be disabled in such a scenario. TropoDB disables unordered writes by default if the option sync is enabled. The option sync is used in RocksDB and TropoDB when I/O needs to be synchronised after each put operation. Disabling unordered appends is also used to test the performance gain that can be achieved with unordered appends compared to ordered appends. When unordered appends are disabled, it is not necessary to log sequence numbers anymore and a default implementation can be used. In this case, appends in the WAL will use a queue depth of 1 and look like the design seen in Figure 4.4. Notice that all I/O will happen in sequence. During recovery, all entries can be read in order and no sorting is needed. This recovery is more simplistic and visible in Figure 4.5.

**WAL zone circulation**

Another property that is interesting in TropoDBs WAL design is that it can lead to a few zones of the ZNS device getting most of the I/O. As mentioned earlier, WALs are the hottest I/O data structure, but are relatively small, only necessitating WALs to claim just

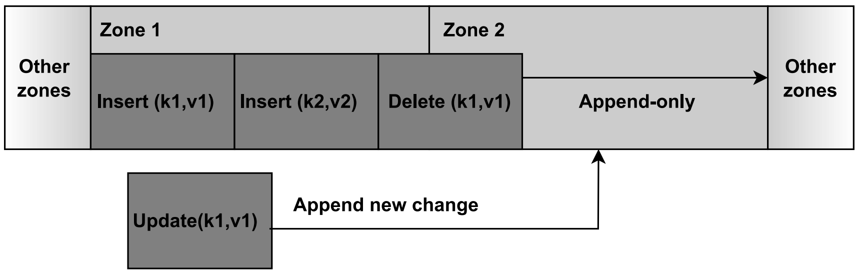


**Figure 4.4:** WAL design: Ordered appends to a WAL

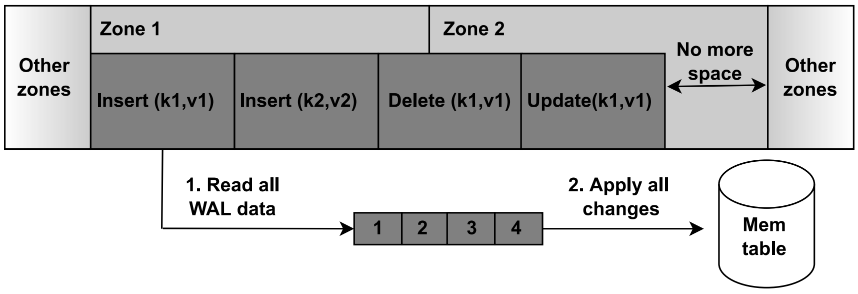


**Figure 4.5:** WAL design: Recovering an ordered WAL

a few zones. Nevertheless, this will lead to a few zones wearing out faster than other zones. Therefore, it might be better to rotate zones used for WALs to spread the load and prevent burning a few zones. Therefore, TropoDB allows assigning a larger region of WALs than just one WAL requires. The amount of zones reserved must always be a multiple of the amount of zones one WAL needs. Internally, WALs will then be circulated in this region of zones to spread the load. This is done with a *circular log* approach and is visible in Figure 4.6. In this example a WAL needs 2 zones and 4 zones are assigned to all WALs. The first WAL will use zone 1-2 and the next WAL uses zone 3-4. As all WALs have the same temperature, the temperature can be evenly spread. At the same time, it requires reserving more space for WALs and is thus a trade-off. This is, therefore, a good parameter to make configurable.

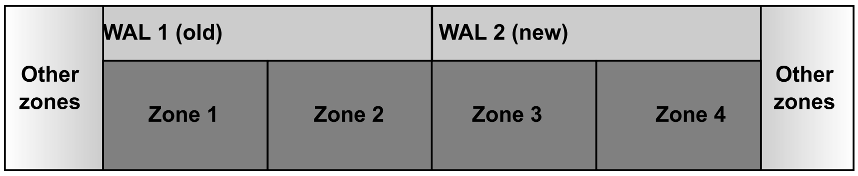


**Figure 4.6:** WAL manager design: Contiguous circular log of WALs

### WAL resets

The last property that is unique in TropoDB is that the key-value store is directly responsible for cleaning its own WALs (Control for DR4). Theoretically, multiple erasure policies can be used, depending on the use case. This allows the key-value store to time its resets and make the latency more predictable. Removing WALs in TropoDBs design is not a suggestion like it would frequently be in file systems. At the moment, TropoDB only supports an eager policy. Whenever WALs are removed and no longer necessary, the space is immediately reset for later reuse. This is done by background threads and never by clients. This ensures that clients should in most cases not have to wait for an erasure to complete, unless no more WALs are available (aids DR4). With the approach taken in TropoDB it is not possible to come in a situation where no more WALs are available, as all WALs are erased eagerly and there need to be at least 3 WAls (there is always one free). The reason to pick an eager policy is to prevent needing large erasures later on for WALs or needing to make the client wait for new WALs to become available. WALs are hot, which makes it likely that WAL zones will need to be reused, especially when the amount of WAL zones reserved is little. A goal of TropoDB is to get latency stability, therefore, it might be better to spread out the cost of erasing the zones, instead of issuing a few expensive erasures once in a while. Further on, postponing erasures can cause all WAL zones to become filled. This will cause issues when a new WAL needs to be reserved, in this case the client has to wait for a new WAL to become available, creating latency issues. Ideally, this should be confirmed by an evaluation. Therefore, future research could look into multiple erasures policies apart from eager, such as a lazy strategy or erasing WALs every once in a while periodically (addressing DR4 optimisations).

**Addressing the design requirements**
The WAL design addresses the design requirements (and RQ2) in the following way:

- DR1: The WAL implementation makes exclusive use of append operations.
- DR2: The WAL design of TropoDB can minimise overwrites and erasures because there are no in-place updates, either implicit or explicit. This is possible because only appends are used and data is never rewritten in the WAL by the LSM-tree itself. Further on, as WALs get a predefined set of zones, there is hot and cold separation, not requiring implicit rewrites for cold data by garbage collection processes. All WAL updates are in one big append-only log. While not necessarily an overwrite issue, some minor write amplification does occur because of the page size of the device. That is because WAL updates always need to be a multiple of the page size (by adding padding), but this can not be circumvented and happens regardless of ZNS

or FTLs. Note that there is also research into *group appends* (107) and *rocks* (94), solutions that come with ZNS interface modifications that allow for smaller page granularities. The need for padding might, therefore, change in the future. Erasures are minimised as WALs are only erased when they are no longer needed and they contain stale data. This is data what should be removed in all cases. Further reduction of erasures can only be reduced by changing the LSM-tree design itself on top (compression techniques for example).

- DR3: The WAL design makes use of the concurrency capabilities of the device by effective use of append operations. WALs make use of a novel operation known as unordered appends, that enqueue multiple appends to the same zone. This increase concurrency capabilities of the WAL and the key-value store as well. As an extra benefit, using multiple concurrent appends does not make use of extra active zones to achieve parallelism.

- DR4: The WAL design of TropoDB can aid in performance and latency stability with the help of appends and its erasure policy. Erasures are done eagerly, which should cause most clients to experience the same latency fluctuations because they are spread out. Similarly, clients should never have to wait for WALs to become available, which can be done by ensuring that more than one WAL region is available. Using a proper ratio of writes and appends can lead to more predictable and lower latency.

**Disadvantages of the proposed design**

The WAL design comes with a few disadvantages. The first issue with the WAL design is that the amount and maximum size of WALs are predefined and reserved from the start. This is not the most space-efficient design. It means that WALs always make use of the maximum amount of space that they are allowed to use. Future work can look into an allocator that assigns zones to WALs only as needed, preventing the need to pre-allocate. This can also aid with spreading temperature on the ZNS device. WALs are very hot, even when a rotating scheme is used as is done in TropoDB. It can, therefore, be beneficial to also append to a few other regions of the SSD once in a while. Another challenge of the WAL design is that the size of an individual WAL is fixed and that WALs are inherently tied to a memtable. This makes it non-trivial to configure the size of a WAL to match the buffer size of memtables. Once a WAL is full, it is also truly full. It is not possible to write a little bit more data to the WAL in this case. This forces the key-value store to do a flush. Similarly, if the WAL is larger than what a memtable can hold, a part of the WAL will never be used. This can lead to resets on partially filled zones, which makes these resets less efficient (more data could have been written to these zones). This is also not solved by the WAL rotation scheme and will lead to holes in the WAL that will never be written or reset (Some WALs will never become completely filled), Currently, this is mitigated by making the WAL just big enough to match the memory table, but it might be a better idea to allow a memory table to use more than one WAL if necessary as this is more flexible. Additionally, the size of WALs should be altered to be more flexible.

### 4.3.3 Group Logging

Both LevelDB and RocksDB support *group logging* (32). Group logging allows batching multiple write requests for one log data write. For example, if a put is issued to the database and another put operation is also issued and waiting, the two can be merged into a bigger put request. This is done even before the write is issued to the memtable and the WAL. While it is not a focus for TropoDB to optimise group logging, it does support group logging and uses a modified implementation of LevelDBs group logging. Therefore, multiple appends can be batched as one update to the WAL and the memtable. Updates are, therefore, not necessarily send one by one. It is known that for faster NVMe SSDs group logging can result in writer threads idling and an increased software overhead (32). It is not known if this is also true for ZNS and this has not been investigated in TropoDB. It is a good idea to further investigate this in future ZNS investigations.

### 4.3.4 Handling Data Corruption in WALs

TropoDB does not focus on how to become crash resistant or how to deal with corruption. However, it does come with a few features to make it a bit more tolerant of these issues. Appends can be done up to steps of ZASL, but there is no guarantee that all pages of the append are physically written. That is because updates up to ZASL do not have to be atomic, which can lead to issues on a system crash. However, appends at the page level do have to be atomic. Therefore, an append should come with some metadata to ensure that no corruption has occurred for each individual page. Each page written to the WAL (one append can have multiple) comes with a little bit of extra metadata. This includes the segment of the append (head, center, trailer), the amount of bytes used in this page and a CRC32 header to check for corruption of data. This was taken from the block implementation of LevelDB (51). Future work, could test the effectiveness of this approach for ZNS designs and come up with designs that are more optimised and made more reliable for ZNS.

### 4.3.5 Summary

In short, WALs in TropoDB are designed around the concept of once logs. Once logs are structures made uniquely for WALs. They only support appends, resets and reads and only type of operation can be conducted at the same time. WALs are for most of their lifetime append-only, only requiring reads on startup and recovery. Therefore, WALs trade read performance for more append performance (DR1). This is done with the help of unordered appends. Internally multiple asynchronous appends can be enqueued to the WAL in any order (DR4). On database recovery, the entries can then be sorted to get a consistent ordering. WALs support a novel circulation scheme to spread heat across multiple zones. WALs are reset eagerly in background processes, which resets in at most one active WAL and one dead WAL at most. Lastly, WALs have support for group appends and come with basic corruption checking.

## 4.4 Index Structure

LSM-trees make use of an index structure to maintain their metadata. This structure is necessary to maintain indexes for the tree components. That is because the LSM-tree contains various components, but there needs to be at least one structure that knows what each of these components is and where their data is stored. In particular, it needs to at least keep track of how many SSTables each level has and it needs some data to retrieve the data of each SSTable from storage. It can also be used to store additional data, such as metadata from deleted SSTables. The design of this index structure can differ between LSM-tree implementations, but generally resides in memory. Nevertheless, this structure needs to be stored on storage as well. That is because, when the key-value store is restarted, the index structure should still *remember* the state of all of its components. Otherwise, proper recovery will not be possible. In general, it holds all state of the key-value store that needs to be persisted apart from the raw data of the components themselves.

Similar to memtables, this research is not about optimal in-memory structures, so an existing index structure is reused. Nevertheless, this structure does, unlike the memtable, need to be altered significantly. Normally it would contain file system information, such as filenames of live- and dead files, but this is not possible when not using any file system. Instead, for TropoDB it needs to contain component-specific information for each individual component. Therefore, it was decided to pick the LevelDB implementation. This implementation does not contain a significant amount of file system or I/O-specific information, which makes such a transition achievable.

### 4.4.1 In-memory Design of the Index Structure

LevelDBs version structure is not very different from RocksDBs implementation, but contains less information and logic. In its most basic essence, it is a collection of metadata attributes and a *version number*. In later sections, we will see what data needs to be maintained/stored for each structure. Notably, it does not need to maintain any metadata for WALs, it only needs data for L0 and up. The implementation of LevelDB makes use of a concurrency control method known as *multiversion concurrency control* (MVCC). In MVCC multiple versions of the index structure can be alive at the same time. This method is used as an optimisation because the index structure can not be used safely while it is modified. That is because, if the index structure is modified during use, it will cause correctness issues for the reads. For example, an operation might read a part of the index structure in a different state than it reads the rest of the structure. On a read, all metadata must always be in the same state. Updates must, therefore, be atomic. A naive implementation would be to lock around all metadata updates to guarantee all changes are only done in one go. However, this requires waits for client-issued gets and is not ideal for operations that wait on I/O to complete (Will inevitable cause issues for DR4). The MVCC pattern can solve the correctness problem and still allows the client and updates to be done at the same time. We will explain how MVCC achieves this feat.

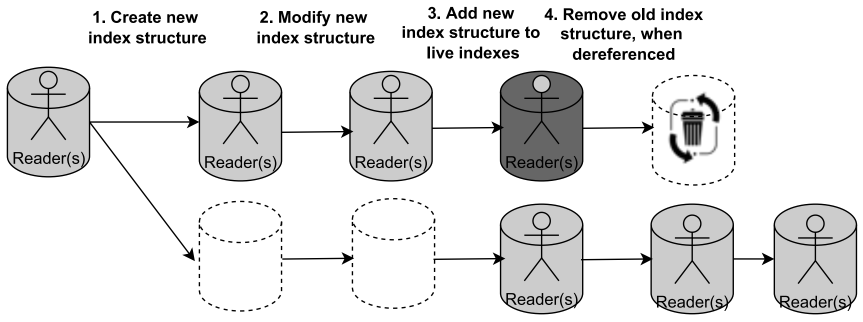

1. Create new index structure
2. Modify new index structure
3. Add new index structure to live indexes
4. Remove old index structure, when dereferenced
Reader(s)

**Figure 4.7:** Metadata design: MVCC pattern, the light gray version is the preferred version for clients

We will explain MVCC with the help of the explanation in Figure 4.7. **(1)** In MVCC, when the index needs to be updated, a new version of the index structure is created in the background.**(2)** This new index structure will be modified, not the original index structure. This allows both the update operation and the get operation to proceed as they use a different structure. After the modification is completed, the new version will then be serialised and written to storage. Only once the modifications to the new version are finished (atomic), serialisation to storage included, is it safe to read for clients. **(3)** It is then added as a *live* version to a *version list*. On the background a *versioning number* will be incremented globally and the new version will get this number. All get operations will read the version from the version list with the highest number. This effectively replaces the old version with the current version for all future reads. Such an approach, allows the old version to still be usable by gets of the client and should give minimal latency hindrance. The client always tries to claim the most recent available version and once it is finished, *releases* the version.**(4)** Once an older version is no longer referenced by any client, it can be safely deleted from memory and storage (reference counting). This means that at any point in time, it is possible for multiple versions of the index structure to be *active*.

The index structure is used to retrieve information about the LSM-tree. For example, on a get it is necessary to find an SSTable that contains the key. In order to get a consistent state (not information from multiple versions as data can move between levels as well) and ensure MVCC is obeyed, one version is referenced and pinned by the get operation (snapshotting an MVCC version). It then proceeds to iterate through L0 to LN until it finds the key-value pair that matches. The pinning operation is also done by both flushes and compactions to determine what SSTables they to merge and where to find those SSTables. Again a version needs to be referenced and pinned for these procedures to be safe. Versions are only released by an operation, once they are no longer used.

MVCC makes deleting data complicated. To illustrate this issue, assume an SSTable is no longer needed in a new version and can thus be deleted (at least according to this version). Therefore, a new version is made that no longer references this SSTable and marks the SSTable ready to delete. However, at the same time a client-issued get is still reading this old SSTable. In this case, we are essentially left in a state where some data is dead nor alive. To circumvent such issues, deletes of any type **must** be treated differently. This can be done by adding metadata for deleted data as well and only truly deleting it when no version contains this data in a *non-deleted* state. In other words, metadata of deleted objects needs to be maintained as well. The metadata of the index can in the end be divided into three types of data: static data that is never modified after database creation, live data that differs between versions and dead data that differs between versions.

On a shutdown of the key-value store, no versions are referenced anymore. As stated earlier, the newest version is already present on storage. Therefore, they can all be safely removed from memory. On a restart, the most recent version is loaded from storage and reapplied to memory. This returns the database to a consistent state on recovery.

### 4.4.2 File System Approach for Storing Metadata of the Index Structure

RocksDB and LevelDB make use of files to store their metadata. In particular, they make use of 2 different types of files, which we will both explain. These are known as *manifest* (42) files and *current* (46) files. A manifest file is a file that contains all metadata of the index structure. There can be multiple manifest files with each file holding (a) different version(s) (again a form of MVCC). The current file's sole purpose is to keep track of what manifest file is the most recent manifest file. The current file only contains the name of the most recent manifest file. The file system approach for storing data can be seen in Figure 4.8. In this figure, the current file references the current manifest file and will reference the new manifest file once the new manifest file is ready. Thus, the purpose of the current file is to determine what manifest file holds the most recent data. The current file only needs to be updated each time a new manifest file is created. Each manifest file that is not the *current* can be deleted once it is no longer referenced. When and how the manifest updates or a new manifest is created, differs between LevelBD and RocksDB and what options are picked for these databases.

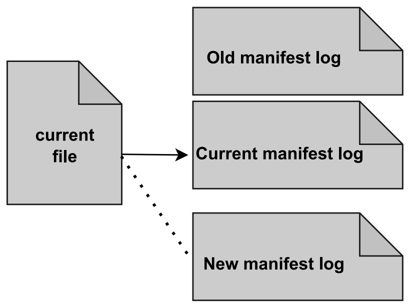


**Figure 4.8:** Metadata design: RocksDBs metadata file approach

For LevelDB, a new manifest file is created each time the database is restarted (51). On database startup, the index structure is serialised and written to this file as a snapshot. From this point onward, the manifest file becomes and append-only transactional log. Each update to the index structure, is serialised and appended as a transaction to the manifest file. Updates can range from adding new SSTables, adding new deleted SSTables, to updates to compaction metadata (we will tell more about this in Chapter 4.6.2). Important to note is that because the manifest file is used as a transactional log, the file does not hold just one version or just one state. Instead, it contains a snapshot of the index structure as it was on startup and all changes that were applied to the index structure after database startup. On a restart the database can be recovered in two steps. First, the snapshot is read and the resulting index structure is set to be the current index structure. Then all transactional changes are read from the transaction log and applied in order to the index structure. After all changes are applied and a new snapshot is created, LevelDB creates a new manifest file that contains the current *snapshot* of the current state of the index structure. The old manifest file (that were used to get to the snapshot) can then be deleted. In this design, static data will only be written once for each manifest file at the beginning. Added and deleted files are not differentiated and are added to the log on a change (files are added or removed). As each file has a unique name, it becomes trivial to determine what files are still present when the log is read in order.

In RocksDB it is also implemented as a transactional log. RocksDB by default creates a new manifest file once the size of the transactional log passes a certain size. When a new manifest file is opened, the current state is added to the manifest file as a snapshot, similar to LevelDB. RocksDB policy for creating new manifest files prevents creating a large file of metadata (space-efficiency) and reading large logs on a restart.

An advantage of the manifest file design is that it fits flash storage and its properties. It does not do any in-place updates and is append-only instead. As manifest data is stored in different files, it also separates manifest data from the rest of the key-value store data, such as WAL and SSTable data. This leads to hot and cold separation. An advantage of storing the name of the current manifest in a current file, is that it prevents needing to read multiple manifest files to determine the current version. Instead, only 2 files need to be read: the current file and one manifest file. It also results in reasonably low write amplification at the application level. Only changes applied to the structure need to be written to storage, instead of the whole index structure on each change. It does make reads a bit slower as all changes need to be read instead of just a snapshot. However, since the manifest is only read once on startup, this is justifiable.

The problem with using files for such a design is that it relies on the file system to treat the manifest as an append-only file. Manifest files can increase quite a bit in size throughout their lifetime, especially with LevelDBs design, and no space is reserved for manifest files beforehand by the database. Further on, manifest files are reasonably hot. They need to be updated on each flush or compaction, regardless of the SSTable in question. This results in the need to allocate more space for manifests on each background update. This in combination with the inability to physically force the file system to store manifest files in different zones than other files, can lead to issues. It prevents communicating to the underlying storage that it is better to store the manifest in a few separate zones and pre-allocate some space. For example allocating to the maximum size that a manifest can become in the case of RocksDB. Nevertheless, updates to metadata are generally small and are just a small part of the I/O issued by the key-value store. Therefore, the gain that can be achieved by reducing the semantic gap of manifests is less than for other parts of the database.

### 4.4.3 TropoDB Approach for Storing Metadata of the Index Structure

Index data is stored a bit differently for TropoDB. In TropoDB a few zones are reserved at the start to hold all metadata. Essentially, the maximum size that metadata might need, needs to be reserved beforehand. This allows hot and cold separation of metadata from the rest of the key-value store's data (Addressing DR2). All of the metadata zones will then have the same temperature. A big difference in TropoDBs design is that unlike how LevelDB and RocksDB try to separate *manifest* data and *current* data, TropoDB makes no such attempt. In TropoDB they are always stored physically together in the same zones. That is done, because there is no concept of files in TropoDB, so it is not possible to differentiate between the two types of data on that level. The only method to truly separate manifest and current data physically would be to store them to separate zones. As the current only needs to hold a few bytes that references the current manifest, it is wasteful to reserve entire zones just to hold a reference to the current. In order to guarantee persistence of the current data, at least 2 zones would always be needed (because of erasures and no support for overwrites). To illustrate this issue in more detail: the ZNS SSD used for this experiment has a zone size of $> 1$ GB. This would mean reserving a few GB for just a few bytes, as only one current entry is active at the same time. Instead, it was argued that current is inherently tied to the manifest and the caused write amplification is negligible. As ZNS does not allow random writes, this means that manifest data and current data needed to be added to the same write pointer. Essentially making it impossible to figure out in what page the current is stored exactly.

Therefore, it was decided to use the current in TropoDB as a *barrier* for updates. This makes it easy to spot the current data, as it *should* always be the last page written. After each update to the database, the current data is appended (DR1) in a separate page. This looks similar to Figure 4.9. Data can be added to the same zones or different zones, but the current always follows the metadata that it references. If on a restart no such page is found, it is assumed that the last update did not complete or is corrupt. Instead, the last change is picked that is followed by an explicit *current page*. A current page is a page that

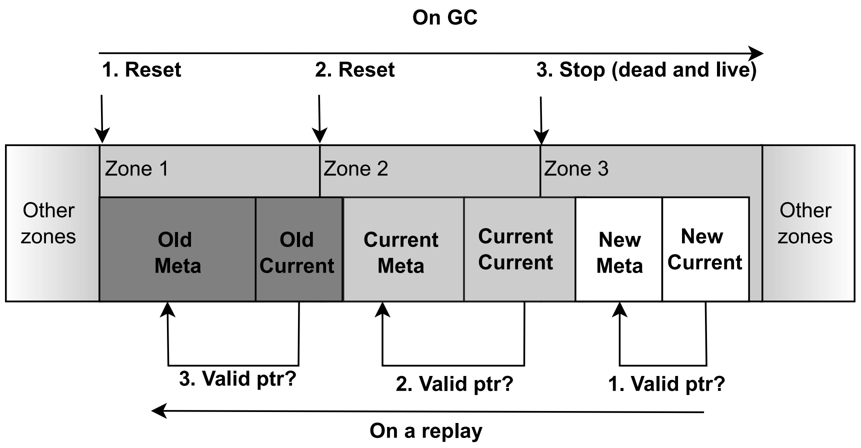


**Figure 4.9:** Metadata design: Metadata operations on zones

holds the current data, it thus replaces the concept of a current file used in LevelDB and RocksDB. Since the current data is never more than a few bytes, it should always fit in just one page. On a replay, the metadata zones are thus replayed from end to beginning until a valid current data page is found. This is also visible in the graph, where in order: the new, the current and the old current are read and verified. The first current that is valid is picked. This design also guarantees that updates to the index structure are atomic, which is an additional benefit that comes with this design. It is not possible to write a current page without writing all metadata first and the other way around.

Further on, the manifest in TropoDB is not implemented as an append-only log like LevelDB and RocksDB. It instead uses a *Copy-on-Write* (CoW) design, or perhaps more accurately Copy-on-Append, implementation and uses explicit snapshots. Snapshots mean that after each update to the index structure, which happens after a flush or a compaction, the entire index structure is serialised and appended to storage (DR2 no overwrites). Snapshots are not assigned to different zones than other snapshots. So, multiple snapshots can be stored together in the same zones, achieving higher space-efficiency. This leads to situations such as the one in Figure 4.9. Storing metadata as snapshots is also beneficial for the design of the earlier mentioned current data. When, a design different from snapshotting is used, we will have to deal with manifest data and current data being intertwined. Within the first page of a manifest and the last page of a manifest, there can be multiple pages holding current data. Therefore, without snapshotting a current should know where a manifest's changes begin, end and what pages in between the beginning and ending contain other current pages. These other current pages need to be ignored. This would require adding significantly more data about all other current data pages to the data of current page. When the amount of data passes a certain size, it will not even fit in one page anymore.

The CoW design used in the metadata design does require the *current page* data to hold a bit more data, because it does need to know where the latest snapshot begins and ends. This is done by adding a few bytes to the current, defining the first page of the latest snapshot and the number of pages of the version it references. Since old snapshots are not stored in files, it does make deletion (and GC) tricky. It is not possible to delete old manifest files based on their name as would be possible with file systems. We can also not delete all zones that hold old snapshots, as there can be a live snapshot in a zone along with a dead snapshot. This issue is in TropoDB solved by deleting all metadata zones that do contain data, but that do not contain the most recent snapshot and current. The resulting GC process is visible in Figure 4.9. In this graph, the GC moves from the first zone to the last zone that it is allowed to reset. Once the GC reaches a zone that it can no longer reset, it stops. Eventually, after enough updates to the index are completed, the last reserved metadata zone will be reached and the write pointer will have reached the end of the log. At this point in time, many earlier metadata zones should be safe to delete and reuse. Therefore, it should be possible to move back to the beginning of the metadata log. This requires a circular log design as visible in Figure 4.10. Since metadata is not restricted to one zone, in such a design it is possible and in fact likely for a snapshot to wrap around from the last zone to the first zone if it contains more than 1 page. This is also why the current data holds a reference to the first page and the number of pages of a snapshot. This should make it possible to also read the snapshots that wrap around from the end of the log to the beginning of the log (when first page + number of pages > last page, do a modulo from the start). The metadata zones are thus used as a *circular log* in TropoDB.

Deletions are done eagerly. After each update, all old data is removed (if any). This prevents doing one large removal operation once the circular log is filled (addresses DR4 issues). Instead, the cost of resets is spread amongst all updates. Alternative removal strategies have not been tried and are left for future work. For example, it would also be a possible design to do GC when no other background action is running, as removal of metadata zones is generally low-priority until all zones are filled.

On a restart, it must first be determined where the circular log starts and begins. The location and the number of metadata zones are fixed, making it trivial to find out where all metadata is stored. Therefore, the zone heads are retrieved for all of the metadata zones. A zone that is followed by an empty zone (with wrap around as it is a circular log), is

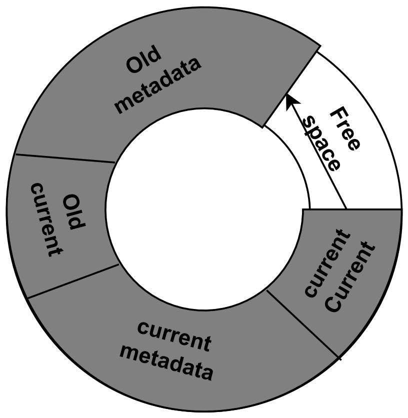


**Figure 4.10:** Metadata design: Metadata as a circular log

considered to be the *head zone*. If all zones are empty, the first zone is the head. The tail is the first zone that is followed by a non-empty zone or the head. This recovery would break when all zones are filled, therefore, there must always be at least one empty zone between head and tail. This is forced in TropoDB.

**Addressing the design requirements**

The metadata design addresses the design requirements (and RQ2) in the following way:

- DR1: The metadata log design only makes use of appends.
- DR2: While the design does not make use of any overwrites, conforming DR2, it does make use of many writes. The metadata log design is not designed specifically to reduce write amplification. The design is CoW, which means that the entire index structure needs to be serialised on each update. The design by RocksDB only logs the changes in most cases, which can lead to less written bytes in the end. However, unlike RocksDB, TropoDB's metadata log design never has to rewrite data because hot and cold data is stored together. It is not possible to store metadata in just any zone. Live metadata will only be stored in a few contiguous zones at most. Metadata is considered to be relatively little for a LSM-tree, which means that the write amplification gains that can be achieved for this component are not big.

  The amount of resets can be kept low with the metadata logs design. During no test on real hardware was their a necessity to reset zones in the metadata log. Nevertheless, the circular log design is already prepared for situations in which it would (and evaluated on small emulated ZNS SSDs). The metadata explicitly uses CoW, requiring no in-place updates. This prevents needing to move live data out of dead zones in case there is not enough space left in the log. The circular log combined with an eager reset policy that resets whenever possible, further ensures that it will only be necessary to delete what is already dead. If a zone contains both dead- and live data, it will be ignored, at the cost of a little space overhead (because of the circular design, a zone can only contain 1 live entry at most).

- DR3: The metadata log does not specifically try to increase parallelism. The metadata log is considered lower priority in TropoDBs design than other components. By not using extra I/O resources for the metadata log, more resources will be available for the other components.
- DR4: The circular log uses an eager reset policy that resets whenever possible, By eagerly resetting, clients never have to wait for space to become available in the metadata log. The space is either already there or there is too much data to fit in the metadata log in the first place. However, as resets can be issued after each metadata update, metadata updates can take a bit longer as they wait for the reset to finish. Using a CoW design can lead to more I/O being required for each change as all metadata needs to be serialised on each update, this has not been a problem in practice, but it is something to consider. The amount of data that is needed to be written on each change depends on the size of the index structure. For a bigger key-value store (more SSTables), this resulted in writing a few megabytes of data on each update instead of a few pages. A low-latency system should also reduce the cost of this operation.

**Disadvantages of the proposed design**

The proposed design works, but also has a bunch of disadvantages. Since it uses a CoW design, it leads to WA (DR2 issues). All metadata is rewritten on each change, even if it is not altered. This can be expensive and lead to wear levelling. Preferably, a log of changes should be used as well, similar to how RocksDB stores metadata or how the WAL stores its data. Nevertheless, during testing it was shown that the amount of data written to the metadata log was relatively small, needing less than 1GB (1 zone) after more than a TB of key-value pairs was written. The achievable gains might thus not be as high. Future work can still look into how to use a log with changes instead of snapshotting, while still appending the "current" data after each update. For example, by maintaining positions of all live *changes* in the current data instead of just a snapshot and merging all changes together into one big snapshot once in a while. Another problem with the design is the circular log. If an external force manages to erase one of the zones in the middle of the log, the implementation would no longer function. This can be mitigated, by storing a versioning number at the beginning of each zone. This should allow determining the ordering by something else than zone heads. Another persistency issue is that there is no error recovery in the case data is not followed by a current. Instead it tries to read an older version, if any, and discards all data in the corrupted version. This older version might reference data that is already deleted. All the data that was stored in the new version is lost and can not be recovered. This can lead to persistent storage leaks when new data was added to say L0 and is dangerous. To guarantee crash resistance, this needs to be improved.

### 4.4.4 Handling Data Corruption in Metadata of the Index Structure

RocksDB and LevelDB come with some checks for data corruption for writes to storage and reads from storage. Most writes done to files are split into the smallest denominator that a file system (and the underlying storage) can guarantee to be atomic to storage, such as a page. This is beneficial as the NVMe spec also specifies writes on the *sector size* (which is another word for the page size) to be atomic. Each of these pages will then get a small header, containing information about this page. It contains how much data this page contains, the segment part of the append (trailer, center, or end) and a small CRC32 (see "A painless guide to CRC error detection algorithms" (126)) header check to verify data corruption. On a read, these headers will be read and verified. This corruption handling is also done for manifest and current files in RocksDB and LevelDB. TropoDB does not deviate here and does exactly the same. Metadata updates are split into pages and each page starts with a small header section. How well this corruption checking works, was not tested, but it has caught quite a few errors throughout development. Future research could look more into how to recover from such corruptions and what other forms of corruption are not caught yet.

### 4.4.5 Summary

TropoDB uses a similar index structure to LevelDB and RocksDB. This index structure makes use of the MVCC pattern to circumvent locking issues. To ensure that deletes are done correctly, both in memory and on storage, the index structure maintains pointers to both alive and dead data. After each change applied to the index structure, the index structure is serialised and persisted on storage. This is done with a circular log implemented on top of ZNS that holds snapshots of the index structure. Multiple snapshots can be written to the same zones. Internally the circular log also maintains pointers to valid snapshots. These pointers are stored along with the data they point towards. On recovery, the most recent valid pointers along with their associated snapshot are read. This snapshot is then deserialised and becomes the current the index structure. In the background old zones are reset eagerly (addressing DR4). The zones reset procedure is called after each update to the circular log and it resets all zones that do not contain the currently-used index data. The circular log also makes use of basic corruption checks by using CRC checksums.

## 4.5 SSTables

*Sorted string tables* (SSTables) are designed to be stored on storage. They are immutable memtables converted to a storage-friendly format. They are always a collection of key-value pairs sorted by their keys. This allows iterating through them at a reasonable speed with for example *binary search*. It also allows making a minimal summary of the SSTable to be used in the index structure, containing information such as its smallest and largest key. When a key does not fall in between the smallest and largest key of the summary, the SSTable will not contain said key, reducing the need for many read operations to storage. Additionally, SSTables store a *versioning number*. This versioning number increments after each SSTable is generated and can be used to retrieve only the most recent change. For example, tables can overlap on L0. In that case only the most recent change needs to be picked. A common compression technique is to make use of *run-length encoding* (RLE (111)) to store the keys in SSTables. Run-length encoding is a lossless form of data compression, where consecutive overlapping items are compressed to one pair. This pair contains the overlapping data and the number of consecutive items that contain the overlap. For example if a part of a key is repeated multiple times in sequence, this part can be summarised as the overlapping part followed by the number of keys that have the overlap. This is an effective strategy for SSTables as the keys are stored sequentially. This allows reducing the space needed for each SSTable and can increase space-efficiency. As tables are also smaller it reduces the number of pages that need to be written and read for writes and reads of SSTables. RocskDB, LevelDB and TropoDB all do this. The effect with and without such encoding is not tested for TropoDB in detail, as it has already been shown to have had a great effect in LevelDB.

### 4.5.1 File System Approach for Storing SSTables

To design SSTables for ZNS, it is a good idea to first look at how SSTables are stored in conventional designs. After transforming the memtable to the RLE-encoded SSTable, it is stored in a file in LevelDB and RocksDB. Each SSTable is stored in a separate file with a unique name. Each SSTable is thus treated as a distinct file system object. On a successful write, the given filename can be used to retrieve the SSTable. The name of each SSTable along with its size can, therefore, also be stored in the index structure. However, SSTables of different levels are treated the same and are indistinguishable for the file system. This can cause problems with hot and cold separation. That is because L0 is hotter than LN and it is, therefore, not advisable to put them in the same zones. Similarly, as both SSTables are the same type of file and they have the exact same internal logic, which misses optimisation opportunities.

Using ZenFS as a file system alleviates the problem of hot and cold separation by ensuring that SSTables are most of the time persisted into their own zones (24). Nevertheless, there is still no clear distinction between SSTables and other file types or between SSTables of different LSM-tree levels.

### 4.5.2 TropoDB Approach for Storing SSTables

TropoDB differentiates between the logic for L0 and LN. That is because L0 is used differently than higher LSM-tree levels and TropoDB makes effective use of these differences. SSTables in LSM-trees are generally not rewritten to L0 (exceptions such as TRIAD (14) do exist) and L0 contains overlapping tables. All SSTables are eventually moved to higher levels than L0. Further on, SSTables in L0 are typically smaller than SSTables at higher levels. L0 can, therefore, be seen and implemented as a waiting queue. Whereas, LN SSTables need overwrites to support merges (merges can lead to DR2 issues), preventing such usage. We will describe both implementations (L0 and LN) independently in section Chapter 4.5.3 and Chapter 4.5.4 respectively. The result of this design decision is that SSTables of both levels are also stored a bit differently, as they are used by different data structures. The data is first RLE encoded. Apart from the RLE-encoding, no additional data is added to the SSTables themselves. For the next piece of information, we recommend first taking a look at the L0 and LN implementation itself, as the information is intertwined.

L0 supports storing data only at the level of pages. On L0 each SSTable is thus aligned on the level of pages. This requires setting a padding of at most $pagesize - 1$ bytes for each

SSTable. On a successful write, L0 returns the location of the first page of the SSTable and the number of pages used to store the table. This metadata is what needs to be stored to retrieve SSTables on later reads. This is used instead of a filename. No additional metadata needs to be stored in the index structure that is different from the file system implementation.

LN only allows claiming entire zones. An SSTable, therefore, claims a number of zones and not a number of pages. However, similar to L0, the actual data itself is aligned to pages. So this still only requires setting padding of at most $pagesize - 1$ bytes for each SSTable. The only difference is that there can be multiple pages that are not written to at all, up to $zonecap - 1$ pages or $(zonecap - 1) * pagesize$ bytes in total. This also requires a bit more metadata to maintain than L0. On a successful write to LN, LN returns a number of *zone ranges* and the total number of pages written. Zone ranges are contiguous sets of zones and are defined as the starting number of a zone and the number of adjacent zones claimed (1 for only this zone, 2 to also claim the next zone, ...). Up to 6 zone regions are allowed to be claimed by TropoDB SSTables by default. However, this is not necessarily the best number to pick and made configurable. The larger the number of regions, the larger SSTables can be, but the more metadata needs to be stored for each SSTable as well. In general, for each SSTable in LN, “n" zone ranges and a number representing the number of used pages needs to be maintained. In short, LN SSTables in this design are less space efficient and need more metadata.

### 4.5.3 TropoDB Storage Design for L0

L0 is different from the other levels of the LSM-tree. Memtables are flushed to L0 as they are and are appended to the rest of the SSTables on L0. SSTables on L0, therefore, can not be bigger than the size of memtables. The size of memtables is by default relatively small (a few gigabytes at most), which results in small SSTables on L0 as well. Further on, as the SSTables are simply appended, there can be SSTables with overlapping key ranges on L0. SSTables on L0 are never merged and not rewritten back to L0 by any standard operation. Instead at a certain point in time, L0 SSTables are moved/merged to the next level. In other words, data from L0 is relatively small, only written once and never changed in any way. Therefore, there is no need to implement overwrites for L0 (DR2) and it is accurate to say that L0 is essentially a *waiting queue*. Lastly, L0 SSTables have a unique temperature, that is different from the other structures. L0 is hotter than L1 or LN, but colder than WALs.

**L0 hot and cold separation**

Similar to how WALs get their own set of zones, so does L0. L0 gets a predefined contiguous number of zones as L0 has its own unique temperature. This not only helps with hot and cold separation (aids DR2), but also makes it easier to create a structure that can manage the data in this region of zones to deal with the temperature as we will see later.

**Fitting L0 data in zones**

It was decided that since SSTables on L0 can be small, it is wasteful to reserve entire zones just for these SSTables. This will lead to large space-inefficiency. Instead, it is better to allow zones to contain multiple SSTables. Such a design would normally lead to problems when data needs to be overwritten because zones will contain both alive and dead SSTables, necessitating garbage collection operations (DR2 issue), which would in turn create write amplification. A zone can contain both alive and dead SSTables because of the MVCC pattern used in the index, see Chapter 4.4. This is not a problem for L0 as data is only written once and never altered.

On each flush, the SSTable in question is appended to the rest of the L0 SSTables (conforming DR1). SSTables have a maximum size. If the data of a flush exceeds this size, the flushed data is split into multiple SSTables. This is done to ensure that the size of L0 SSTables does not become too big, which can result in expensive merges. How appends work in this design can be seen in Figure 4.11 and will be described in further detail. In this example, a memtable is too big to fit in one SSTable. Therefore, it is split into two smaller SSTables. The version number is incremented and both tables get a unique number. These tables are then appended in order to the head of the log of L0. The resulting data layout will look similar to Figure 4.12 (ignore the dead and live attributes for now). This metadata will then be stored in the index structure. In this example, all zones of the log of L0, are filled up. When the log of SSTables starts to fill up, a few of the SSTables need to be moved to the next level and deleted in this level (ideally this should be done before, which we will see later on for compactions in Chapter 4.6.2). However, this deletion of SSTables poses one challenge. It is only possible to delete entire zones and we have just stated that zones can contain multiple SSTables in L0. The method that TropoDB uses to tackle this challenge is to maintain metadata of dead SSTables as well. These dead SSTables are also persisted along with the rest of the metadata in the index structure. They are virtually identical to ordinary SSTables, except that they are treated differently. Only when a zone only contains dead SSTables, is the zone considered for deletion. Then, when the zone is deleted, the dead SSTables it contains can and must be removed as well. Keeping the metadata would be disastrous, as that might lead to double deletions (double free) later on, which can delete newly stored data in these zones (remember after a delete, they are considered free again). In case a dead SSTable contains multiple zones, we can not remove the entire dead SSTable just yet. In this case the dead SSTable in question is split into a truly deleted part and a part that still needs to be deleted. The part that is truly deleted is removed and the rest is maintained in a *mock* dead SSTable. This is also visible in Figure 4.12. In this example, all L0 tables except for SSTable 4 need to be removed. The rest of the SSTables are all deleted by an external process. However, only SSTable 1 up to 3 can be removed as the tail can not move past live files. SSTable 3 can also not be fully deleted as it partially lives in zone 2 which contains the live SSTable 4. In this example, the garbage collector will reset zone 1 and add a mock SSTable for the part of SSTable 3 that is present in zone 2. The resulting state will end up in a situation similar to Figure 4.13.

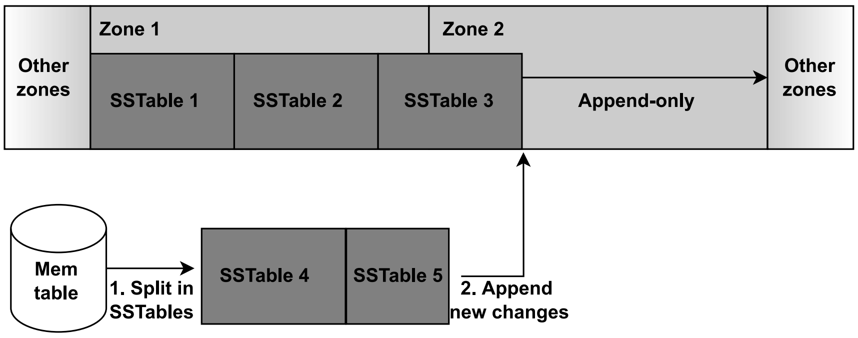


**Figure 4.11:** L0 design: Flushing SSTables to zones

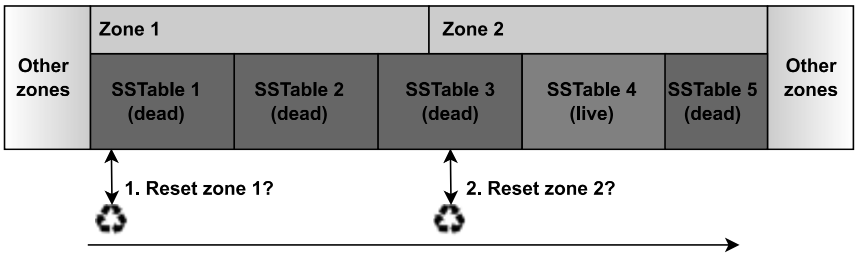


**Figure 4.12:** L0 design: Deleting SSTables from storage

There is still one big challenge. This challenge is that as described in Chapter 4.4, older versions of the index structure may still consider some SSTables to be alive, while they are considered dead in a more recent version. The dead SSTables contain in the worst case all pages between the head and tail, which dependent on the number of reserved L0 zones, can be multiple gigabytes. It is not investigated, how common such an occurrence is. Deleting dead SSTables that are still alive in a few versions, can lead to fatal errors when resetting the zones of these SSTables. Therefore, dead SSTables are only considered for deletion when they are not used in any live version at all. In other words, only the disjoint set of dead SSTables and all live SSTables for all live versions is used for deletions. All tables that could not be deleted are then added to the new index structure again as dead tables, as they still need to be deleted later on. This design mitigates most concurrency issues related to deletes and prevents leaking storage with *persistent storage leaks*. Persistent storage leaks are leaks where deletes did not complete, but the system no longer knows the deletion needs to take place.

**L0 as a circular log**

There are multiple strategies that can be taken to determine where to append new SSTables and what SSTables to consider for compaction to the next level. In the end, it was decided to treat L0 as a circular log. This design was taken for a couple of reasons. Firstly, ZNS is limited in the number of open zones (see Chapter 2.1.5, addresses DR3 issues). This means that only a few zones can be neither empty nor full. Therefore, L0 can not write to many different zones at the same time or is required to finish zones regularly as a counter-measure, which is space-inefficient (finishing a zone wastes a couple of pages left in a zone). Secondly, data is always appended to L0, which makes using one write head convenient and natural. Lastly, data is only removed during compaction from L0 to L1, so by one task. This makes it convenient as well to use just one tail.

L0 is thus implemented as a circular log that exists out of the N zones assigned to L0. When an SSTable is flushed, it is appended directly to the head of the log (DR1), provided that there is space left. SSTables are in this implementation not limited to the size of a zone, they can be less or more than a zone and cross borders of multiple zones. As the log is circular they can even start at the last zone and loop back to the first zone. In Figure 4.13 L0 can be seen as a circular log. The tail is still present at the beginning of zone 2, but zone 1 is already free. Therefore, the head of the circular log is free to write any data to zone 1. Compaction in the circular log design always needs to happen at the tail of the log as it is a circular log. Technically it would be allowed to use a different SSTable for compaction, but when a different table is picked, no space can be reclaimed as the tail can not move. Essentially, L0 would then be as filled as it is before, making no progress and stalling the client longer than needed (causing DR4 issues). Therefore, compaction is forced to start at the tail of L0 in TropoDB. This is an issue that we will explain in more detail later on and turns out to be a bigger problem than was initially thought.

However, this does make compaction a bit more complicated. That is because as a performance optimisation, it is common to pick all overlapping L0 SSTables on a compaction.

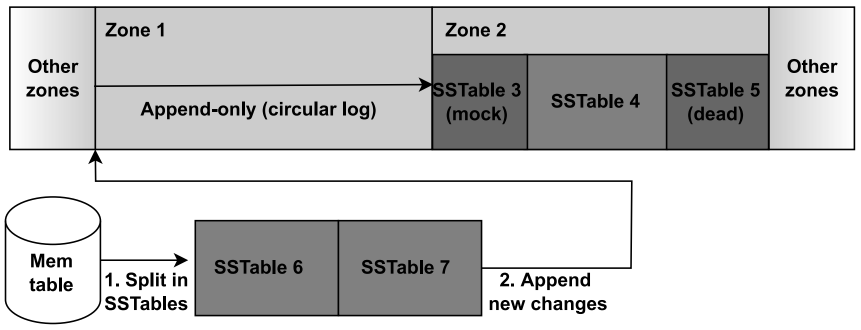


**Figure 4.13:** L0 design: L0 as a circular log

When this is not enabled and there are many overlapping tables, we might need to do a lot of merges when only one merge would have sufficed; creating a lot of write amplification in the end (indirectly leading to problems with DR2). Further on, it might require a lot more compactions, hindering the latency stability (do note that using one big compaction has its own latency problems). Therefore, on a compaction all SSTables that overlap with the SSTable on the tail are picked, even in TropoDB. In TropoDB these SSTables are then marked as *deleted SSTables* just like before, but they can not be deleted yet. Remember that SSTables that are not on the tail of the circular log are not even considered for deletion. We essentially end up with a circular log with *holes*. Holes that will be reclaimed only when the tail of the log passes over them. Deletes are thus delayed. An advantage is that only full zones are considered for resets and resets are, therefore, used to their fullest potential.

The process of reclaiming needs some further explanation. This is done with a separate background operation, that we refer to as *reclaiming invalid L0 zones*. It verifies what zones can be deleted, following the definition of dead SSTables and then eagerly deletes all of these zones at once. This operation is in TropoDB called right after a compaction is finished (compaction itself we will describe in more detail in Chapter 4.6.2), on startup of the database and once L0 is filled and can no longer be used. Deletion is thus always done eagerly; it is done after each moment that more data might be ready for deletion and it always deletes all data possible. Generally, this should always lead to minimal latency fluctuations because of deletions (addressing DR4). There are no big deletions once in a while or long waits because there is no more space left in L0 because of stale deletes. Instead, there will be constant small latency fluctuations. Alternative strategies have not been tested and are something future work could look into. For example, it might be problematic to schedule deletions when it is important to do a flush at the same time. For example, when a client needs to wait for a new memtable. In such a case it might be better to postpone deletes just a little while as generally a client should wait for their operation to finish as short as possible (alternatives to address DR4).

**L0 SSTable metadata**

In order to make the changes persistent, we need to store metadata about where the data of L0 is stored. Since the key-value store keeps track of both live and dead SSTables, we need a list of both. For each SSTable, dead or alive, we keep track of the starting page, the number of pages it contains and the number of bytes that are used by the SSTable. The number of bytes that the SSTable uses is necessary because SSTables by themselves do not have to be aligned to the page size and SSTables on storage, therefore, contain some additional padding. The circular log itself can take care of the wrapping of the address space and zone boundaries.

**Addressing the design requirements**

The circular log design addresses the design requirements (and RQ2) in the following way:

- DR1: The circular log makes exlcusive use of append operations
- DR2: Data in L0 is written once and never written again to L0, regardless of the storage design used. L0 functions as a waiting hub and all SSTables have an equal temperature. The circular log design ensures that this still holds true. Zones are only ever reset if they only contain dead data. Deletes of SSTables thus never rewrite data to L0 and cause additional write amplification or extra resets. All write amplification caused in L0 will thus be because of flushes and compactions, not because of overwrites or storage itself. The circular log design also ensures that zones that are reset are completely filled and only contain dead SSTables. No data needs to be moved and the resets are used to maximum efficiency (no partial resets). This would not have been possible if each SSTable got its own zone. Dead SSTables in L0 can not be reused, which means that the invalidated SSTables need to be erased at some point in time. Fundamentally it does not matter if the reset is postponed or not. The zone does not contain usable data and can not be reused until it is reset. The circular log design, therefore, does not issue more resets than necessary.
- DR3: The circular log does not try to increase concurrency capabilities explicitely. It does try to keep its number of active zones to a minimum, to leave more concurrency for other structures (even other circular L0 logs as we will see later in Chapter 4.7).

- DR4: Similar to most other data structures implemented in TropoDB, an eager reset policy is used. Whenever a compaction in L0 completes, all valid resets are issued. This should ensure that clients should generally not have to wait for long resets. However, the circular log can contain various holes, which can not immediately be deleted because of the circular log design. This can cause a large number of resets to still occur at later times. Further on, it can cause the circular log to be "full", even if it only contains a handful of live SSTables. This pseudo-full state, in turn, will require the client to wait for compactions. The circular log design might thus not be the best design to guarantee latency stability.

**Disadvantages of the proposed design**

The L0 design comes with a couple of disadvantages. Some disadvantages also add more challenges for the research question itself rather than solving them, such as latency stability (causes DR4 problems). Many of these disadvantages were only found during the implementation and long benchmarks running the implementation. We, therefore, recommend also trying out other designs for L0 in future implementations or designs for LSM-trees. The biggest disadvantage is the explicit head and tail of the circular log. Compactions from L0 might always pick one table from the tail, but they can also pick additional tables from anywhere between the head and tail of the log. These tables are not present on the tail and can not be deleted. Instead, they will remain stored until all SSTables in their zone can be deleted and the tail is present in the zone. This means that there are no partial resets, so resets are only used to their fullest potential (generally good for RF2), but it also creates *holes*. These holes are problematic as they can lead to highly fragmented situations; sometimes with more dead SSTables than live SSTables. The result is that the log might fill up, but not because it contains a lot of usable data and the client will have to wait even if there is ample space. Further on, it can take multiple compactions before the zone on the tail can be reset. If there is no space, the client has to wait for all of them to finish. This creates large latency instability and issues for the tail latency.

The next problem is adjacent to the earlier stated disadvantage. If immutable memtables are too big to fit in one L0 SSTable, they are split into multiple smaller SSTables. All of these SSTables are guaranteed to be disjoint, but they are added to the same circular log in order. Compactions from L0 to L1 (which we will discuss later) try to pick overlapping tables for compaction. This amplifies the problem of holes as it is unlikely to pick the SSTable next to the SSTable on the tail as the ranges do not overlap. It is in fact more likely to use SSTables that came from other immutable memtables. This could have been solved by using a few circular logs instead of one and adding the data to the circular log with the most overlap with the SSTable on the head (the adjacent SSTable).

Another problem is the explicit number of zones that are allocated for L0. This is not trivial to configure, but has profound effects on the latency of the key-value store. If L0 is full, no flushes can continue and if the memtable is full, clients will have to wait (DR4). If L0 is made large, there will be less space for LN, but the stall can also be postponed and eventually become a bigger problem (again DR4). In Chapter 6.6.1 we have investigated the effect of increasing and decreasing L0 and shown that it has a big impact on latency, write amplification and the number of resets. Using a large L0 can decrement performance, but a small L0 as well. Therefore, the size of L0 log must be properly chosen by clients of the database, but it is not yet known how an appropriate size for L0 can be picked. LevelDB and RocksDB make use of a variable that contains the maximum numbers of SSTables in L0. The size of L0 approximates this variable, but the problem is that in the case of L0 it also contains dead SSTables. This is not easy to configure for. It is not possible to set L0 equal to the size of this constant (max SStable size * max number of SSTables in L0), it must be set higher to include deleted SSTables.

The last problem is recovery and crash tolerance. This is not a focus of TropoDB, but the current design will need significant alterations to support these functionalities. The circular log may not have physical holes, only logical holes. Any hardware error in a zone between the head and tail zone will lead to the circular log being unable to recover. Solving this would require extra metadata to verify L0 correctness and patch the log if necessary (filling mock data in broken zones). It also does not log what changes will be applied to the circular log before it applies the changes, it only returns once it is finished with what it was ordered to do. For example, if during an append to L0 the process crashes, no component has a reference to the bytes written by the append. This data will be neither dead nor alive and is not caught by log or the key-value store. No one knows that a write to the log occurred and that it did not finish. The result is that once the tail of the log reaches this garbage data, it can never reset and the log is broken. Similarly if during a cleanup operation, the process crashes, no component knows that some data might already be deleted. The key-value store will reattempt deleting its SSTables, which might have already been deleted. The tail has already passed them and again the key-value store is broken. These issues can be solved by first logging what change is going to be applied, then applying it, and only after applying it, removing the log or marking that it is finished. For example, with the help of a small WAL. Then on recovery, this log can be scanned and checked for partial operations and if necessary undo or reapply them. This will add more write amplification and needs another active zone, but is necessary to guarantee better crash tolerance.

### 4.5.4 TropoDB Storage Design for LN

LN tables are the colder on-storage structures of the LSM-tree. They exist out of multiple levels and typically contain larger SSTables than L0. Unlike L0 they also contain data that can be overwritten. That is because each compaction can lead to a merge in the next level. Therefore, updates to LN need support for these merges. A circular log design like L0 would not work as well in this case because data can be modified anywhere within the log. Further on, storing multiple SSTables in a zone can be problematic. As updates can be anywhere, this can lead to situations where multiple zones contain dead SSTables, but no zone can be deleted. If no sane compaction policy is picked in this case, eventually there might be no space left because no zone can be cleared, requiring garbage collection operations to defragment the data (which cause DR2 issues).

**LN hot and cold separation**

LN gets a contiguous region of zones similar to the other data structures. This allows separation of hot and cold data. However, L1, L2, ... and LN all store their data in the same region of zones. This is done because as we will discuss later, SSTables in LN only contain entire zones. Data will thus by design never be stored in the same zone, from L1 onward. At this point, hot and cold separation techniques such as assigning unique zones make little sense. Similarly, this can lead to space issues. Each additional level is bigger, but stores the same data. L1 is already a lot bigger than L0 and the WALs. When all data moves to the next level, there always needs to be enough space to store all of this data. If L1, L2, ..., LN all have their size to a predefined constant that is their maximum we have an issue. All levels all reserve the maximum size that they can, because in TropoDBs design each component is assigned a number of zones from the start. This number of zones can not be changed after database creation. The results is that LN can not be bigger than $SSD_{cap} - metadata_{reserved} - WAL_{reserved} - L0_{reserved} - \sum_{i=1}^{N-1} Li_{reserved}$. This in turn can result in LN not having a lot of space left. In this case, even if a multiple terabyte SSD is used, less than a few hundred gigabytes might be available to use for LN at maximum. Instead, since it is already determined that hot and cold separation from L1 onward is less of an issue, it is saner to store them all in the same region. The levels can then still have unique logic, for example, by specifying how big their SSTables are allowed to become, but LN will not be limited in size by the lower levels. Space that was claimed by those levels can then be used by LN instead, leading to better space utilisation.

**Fitting LN data in zones**
TropoDB claims entire zones for SSTables from levels L1 up to LN. This is easier to justify as SSTables are generally bigger at higher levels. It is not uncommon to see SSTables spanning multiple zones, making the space-inefficiency issues that would otherwise occur on L0 less. This design also simplifies deletion. When an SSTable needs to be deleted, it is not dependent on other SSTables in a zone. Instead, all zones of an SSTable can be safely reset (no DR2 issues ass well). After an SSTable needs to be deleted, a deleted SSTable is still generated, similar to in L0. Then when this SSTable needs to be deleted it is first verified if it is still live in any version. If it is not, it is deleted and its metadata as well.
**LN as a fragmented log**
LN is also not implemented as a circular log to prevent issues that can occur with the tail of a circular log. LN is implemented as a *fragmented log* of zones instead. In this implementation, each SSTable has a round number of zones and zones are not shared between SSTables. What zones an SSTable receives are determined by a *zone allocator*. A zone allocator assigns zones to callees and is similar to how a memory allocator assigns memory to its callees. This zone allocator can be designed in any way that fits the need of the key-value store. This allows creating an allocation pattern that fits the usage of the key-value store. Theoretically it could, therefore, also be implemented similarly to a circular log. However, a few implementation requirements are set for the allocators, to guarantee that they are in fact usable. Internally it should maintain a structure that is able to keep track of all live and free *zone regions*, which is the structure that the zone allocator uses. A zone region is a contiguous set of zones and consists of a number identifying the first zone in the region and the number of additional adjacent zones present in the region (a number of contiguous zones). Client operations on the zone allocator should also make use of zone regions. Reads, appends and resets all need to use zone regions. On a successful put, it returns a set of zone regions acquired during the append and a number identifying the number of pages appended. If not enough space was present, no zone regions are claimed and an error is returned. The zone regions returned by a put can later be used to retrieve data. A get operation needs a set of zone regions and a number of pages to read to retrieve the data. It reads the zone regions specified in order and reads up to the maximum number of pages specified. Using more pages than fits in the zone regions is undefined behaviour. In Figure 4.14 a possible state for the fragmented log can be witnessed. In this situation, two SSTables are stored in a small LN log. SSTable 1 is stored in multiple zone regions and needs a little bit more than 2 zones to store all of its data. SSTable 2 needs less than one zone and only requires one zone region as well. An issue that is visible in this example

and that is a problem with the fragmented log design in general is that if an SSTable takes less space than a zone region, some space is wasted. This causes gaps to occur in zones and causes minor space-inefficiency issues. This can be solved with approaches like *tail packing* as done in file systems such as ReiserFS (47), but this is not attempted as forcing unrelated data together in similar zones can cause garbage collection issues (DR2 issues). The design sacrifices space efficiency to gain lower garbage collection costs.

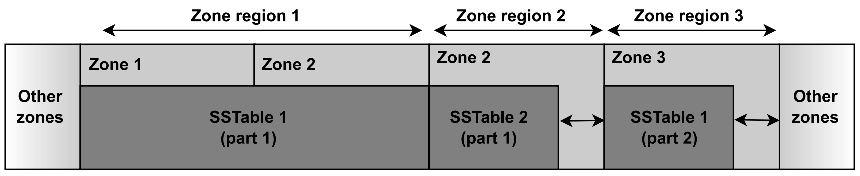


**Figure 4.14:** LN design: SSTables stored in zone regions

A zone allocator is not allowed to do any garbage collection of its own volition. This must be explicitly issued with an erasure operation. This erasure operation again needs a set of zone regions. All zones in the requested zone regions will be erased and their zones will be added to the free list. If we are to delete SSTable 1 of Figure 4.14, it would look like Figure 4.15. In the example it can be seen as the zone regions of SSTable 1 only contain data of SSTable 1, they can be safely reset. The garbage collector thus resets all zone regions with their assigned zones of the SSTable in order. It ignores the rest of the fragmented log. Zone allocators also come with serialisation and deserialisation functionalities. This allows making the changes persistent, also on shutdown. This data should allow returning the zone allocator to the state of the zone allocator as it was before a shutdown. For example, it should still contain the correct zone regions that are free and used. Leaving serialisation and deserialisation up to higher functions allows determining for the level of reliability needed. For the best reliability, it is advisable to serialise and store the data on each change.

### LN zone allocation schemes

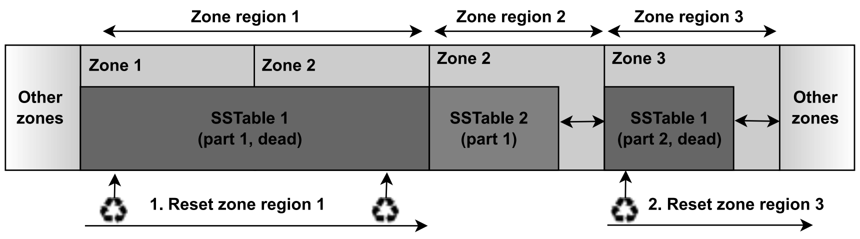


**Figure 4.15:** LN design: Deleting SSTables from storage

One zone allocation scheme has been implemented and tested for TropoDB. A second allocation scheme that addresses most shortcomings of the implemented allocation scheme has been designed, but is not implemented. Its design will be described, but its implementation is left to future work. The implemented allocation scheme is similar to a traditional heap implementation of memory. The design is visible in Figure 4.16 and we will describe the states in this figure step by step. This implementation internally starts out as a linked list with only one entry **(1)**. This entry consists of one large zone region, containing all zones present in LN and is marked as *free*. When a number of zones is requested, a new entry is added to the linked list. This entry takes a few of the zones in the zone region of *free* entry and creates a new entry specifying a new zone region, containing only the zones it needs. This entry is marked as *claimed* **(2)**. The number of zones present in the zone region of the old free entry decrements. The next time space is needed, this process is repeated. However, as the linked list now also contains claimed entries, it can not split all entries. Instead, the linked list is traversed to find free entries. When more zones are needed than are present in an entry, the entire entry is transitioned to the marked state and the searching continues for extra free entries. In this case, multiple entries are returned and each entry corresponds to a zone region. Eventually, the fragmented log will contain various zone regions that are either claimed or free **(3)**. On a free, multiple things are done. First, the entry in question is transitioned to the free state again. If the neighbour of this region is also free, they are merged together to form a bigger zone region. This is also visible in step 4 of Figure 4.16. In this example, the first zone regions are erased and merged back together as one bigger free zone region. Eventually, when all entries are freed, we should be back to the original situation and there will be one big entry again. This implementation only serialises each entry in the linked list and as this data can be very compact, this can require minimal metadata. Even, when there are many zones. Nevertheless, a big disadvantage of this zone allocation scheme that was discovered during testing is that it does not properly spread its heat. As can be seen in the evaluation at Chapter 6.5.5, data tends to stick to the beginning or end of LN as that is where all splits begin or continue. This is thus not the best implementation to reduce wear levelling. For the ZNS device used, the metadata effect of maintaining metadata for each zone is also negligible as the total zone count is less than 10000 and using merging and splitting to reduce cost, will have little effect on performance, both in space as in speed (traversing the small free list).

The other allocation scheme that has only been designed is more simplistic, but is better at spreading the heat, at the cost of needing more metadata. In fact, the amount of needed

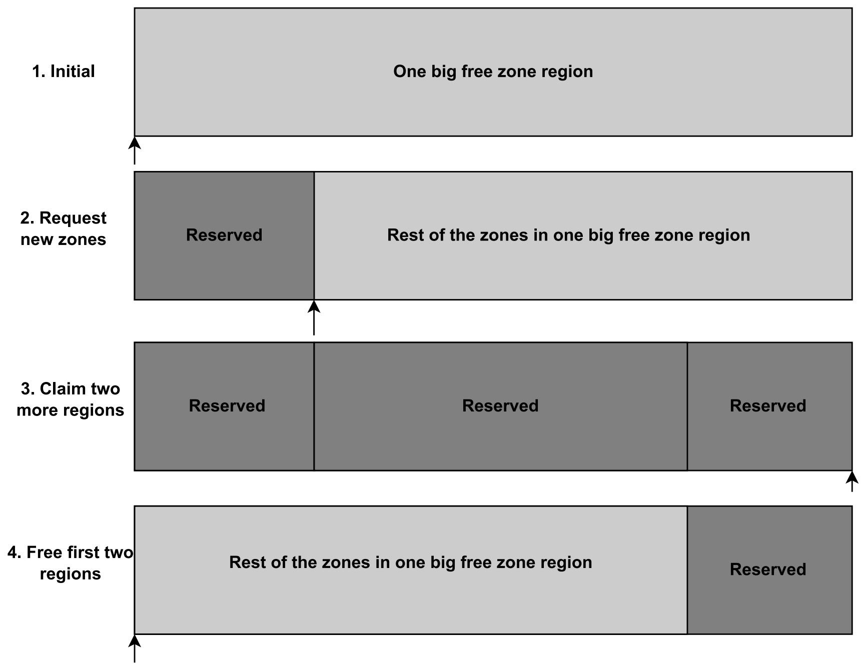

1. Initial
One big free zone region
2. Request new zones
Reserved
Rest of the zones in one big free zone region
3. Claim two more regions
Reserved
Reserved
Reserved
4. Free first two regions
Rest of the zones in one big free zone region
Reserved

**Figure 4.16:** Zone allocator design: Merging allocator

metadata is fixed and scales linearly with the number of regions. It contains one entry for each zone, specifying whether it is occupied or not and one general pointer, that points to the last claimed zone. This design is visible in Figure 4.17. At the beginning, all zones start in the *free* state and the pointer points to the first zone **(1)**. When zones need to be claimed, it first checks if enough zones are still available. If there are not, an error is returned. Otherwise, a loop is started. In this loop, the pointer increments and claims all zones it needs one by one **(2)**. If the zone passes the last zone, it loops back to the first zone. If any of these zones are adjacent, they are returned to the client as just one zone region. It limits itself to one entire loop. If no space was found, it is considered full **(3)**. On a deletion, it only resets the zone regions of the SSTables to the free state(3). No data is merged or split as was done for the merging allocation scheme. This solution leads to a better spread of heat. It has no bias towards any part of its zones as it always continues where it left off and walks among all of its zones. A major disadvantage of this design is that it can lead to a reasonable amount of metadata if there are a lot of zones. It is not known if this is common for ZNS.

**LN SSTable and allocator metadata**

TropoDB adds some restrictions to the usage of the fragmented log and its allocator. This is required because of the format in which TropoDB stores its SSTables. For example, it

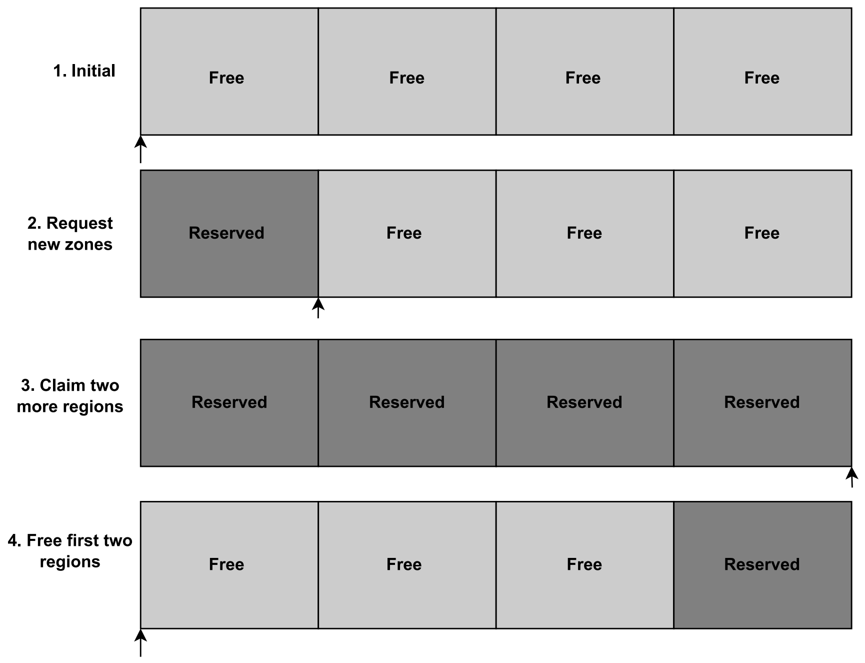


**Figure 4.17:** Zone allocator design: Simple allocator

only allows using a maximum of "n" (configurable) zone regions for each SSTable. Limiting the maximum size of SSTables to a maximum of n zones. These "n" regions are stored in the metadata of LN SSTables. These tables then store up to n records that contain where each region begins and how many zones this region contains. This is used instead of a starting location as is done for the circular log of L0. Further on, to guarantee persistence TropoDB needs to store the metadata of the zone allocator after each compaction or deletion of LN data somewhere. On a restart, this metadata needs to be used to get the zone allocator in a consistent state. This data is stored along with the rest of the metadata of the index structure, but separate from the SSTables themselves. Theoretically, the metadata could also be stored in other metadata regions, this would have the advantage of separating concerns. For example, the metadata log of the index structure does not need to know about the freelist of LN, but in the current design it does. Nevertheless, the current design is beneficial as the data in the metadata log already needs to be updated after compaction and deletion. This allows issuing fewer appends in total and storing all the metadata updates to one location only.

### Addressing the design requirements

The fragmented log design addresses the design requirements (and RQ2) in the following way:

- DR1: The fragmented log makes exclusive use of appends
- DR2: The LN design never stores multiple SSTables in zones. This has the benefit that a zone never contains both dead and live data. In turn, this ensures that when an SSTable needs to be deleted, it can always be reset on storage at no additional cost. No data needs to be moved out of the zones. The LN storage thus adds no additional resets, overwrites or additional write amplification on top of what is issued by the key-value store compactions itself. The only added amplification is caused by the extra metadata needed in the index structure, which is relatively small with the proposed zone allocators, such as the merging zone allocator. However, garbage collection of invalid SSTables will not cause extra overwrites. Resets do have one problem and that is that they can happen in zones that are not completely filled. The relative cost of resets can thus be higher with the proposed design.
- DR3: The fragmented log does not necessarily use of extra concurrency by itself, but it does allow for efficient parallelism as we will see and explain in more detail in Chapter 4.6. Multiple concurrent threads can make use of the fragmented log as each SSTable gets its own unique zones and each write or read to an SSTable can, therefore, complete concurrently. It only requires a bit of synchronisation for the zone allocator, as there need to be consensus on what zones to assign to each SSTable.
- DR4: By using an eager reset policy that resets old SSTables whenever it has the chance, we prevent cases where a large number of zones need to be reset. Resetting a large number of zones could lead to latency spikes. Spreading the load by ensuring that the number of zones that need resetting remains low, can spread the latency of all of these resets. Further on, LN is given a small number of active zones to use and is not designed to issue a large number of I/O in parallel. This is done on purpose. This leaves more bandwidth and storage for more important operations such as WAL updates or flushes.

**Disadvantages of the proposed design**

The proposed design for LN comes with a number of disadvantages. The first disadvantage is space amplification. An SSTable is unlikely to be exactly equal to a round number of zones. The LN design only allows storing one (partial) SSTable in each zone and finishes the last zone on completion. The result is that the last zone of an SSTable in LN will contain unused pages. Each SSTable has this issue, which can result in a large number of pages on the SSD that will never be used and limits the size the LSM-tree can have on storage. This could have been solved, by allowing multiple SSTables to be stored in this last zone with solutions like *tail packing* from file systems, but this would store unrelated data together. Storing multiple SSTables together is problematic as it can lead to two cases. Either a zone can only be deleted if all its SSTables are dead or on a delete live SSTables are moved to other zones. The first approach is not a good approach as it is possible for SSTables of the highest level to be stored with the lowest LN level. This can lead to situations in which no zones can be reset and there is a need for space. Essentially, the system will be locked because it can not safely delete any data. The second approach will lead to write amplification and a more involved garbage collector (DR2). The goal of TropoDB is to reduce such effects and this approach is, therefore, not taken. The LN design thus trades write amplification for space amplification (DR2 solution).

The second problem lies in the default zone allocator, the merging allocator. The merging allocator is not able to properly spread its heat. It does not explicitly rotate its zones. If a zone is reset, the zone is free to use again and the merging allocator is likely to pick this zone again. The result is that zones that are invalidated, will be used again, which can lead to peaks of resets. An effect we have also discussed for WALs in Chapter 4.3.2. The solution is to use a different zone allocator. That is able to spread its load better.

The third problem is a concurrency problem (an issue with DR3). There is only one zone allocator for LN and this is shared by all compaction threads. The zone allocator will need consensus on what to do, which will decrement performance. This means that no matter how many compaction threads are used, there will always need to be some form of synchronisation to the zone allocator beforehand. The zone allocator of TropoDB uses locking to achieve this, which can decrement performance. This can be solved by making the zone allocator more concurrency friendly, but this has not been designed/implemented.

The last problem lies in crash tolerance and reliability. Similar to L0, all I/O operations done in L0 are conducted without first logging that they have been issued, have started or have finished. The LN design does not make use of journalling of any kind. This leads to partial/non-atomic changes if the process crashes during the change. This can happen for appends and deletes. For appends, the zone allocator will have no knowledge that one of its zones contains garbage data and will still give away its zone. That is because the zone allocator makes use of its metadata to check if a zone is full or not, and since the append that was issued during the crash has not been completed, the metadata was not updated. Therefore, the metadata will still state that the zones used by the last append are free. The result is that the appends to this zone in LN will fail and this compaction will fail.

This zone is lost for good. For deletes, the problem is less. A delete is only completed if the zone is reset and the zone is readded to the free list. If the delete did not complete, either because the physical delete was not finished or the metadata was not updated, this will not matter. The reset operation is namely idempotent and can be done again to the specified zones. Similar to L0, the specified issues can be solved by making a log containing the changes before they are applied. Then on database recovery, the log can be scanned and undone/reapplied.

### 4.5.5 Summary

SSTables are always RLE-encoded before they are persisted to storage. SSTables of different LSM-tree levels are persisted differently to storage in TropoDB. All SSTables in the first level, L0, are stored in a *circular log*. The circular log supports SSTables at the granularity of pages. Multiple SSTables can be persisted to the same zone, SSTables can be stored in multiple zones and SSTables can wrap from the last zone of the log back to the first zone of the log. This supports flushing small SSTables with little space-inefficiency. Since L0 is a circular log, SSTables can only be added to the head of the log and SSTables can only be physically removed from the tail. L0 is thus implemented as a waiting queue. Further on, SSTables are only truly removed when an entire zone of SSTables becomes invalid. To support deletes of SSTables that do not live on the tail, L0 supports pseudo-deletes. SSTables that are marked as deleted, but that are only physically deleted once the tail passes them.

All SSTables from the second LSM-tree level onward are stored in a structure we refer to as a *fragmented log*. This fragmented log is shared between level L1, all the way up to LN. The fragmented log only supports reserving entire zones. This means that each SSTable in the fragmented log, has exclusive access to a round number of zones. These zones are allocated with the help of a *zone allocator*. A zone allocator decides what zones a SSTable will receive and can decide for itself what access pattern it wants to use. The zone allocator we used in TropoDB is the *merging allocator*. An allocator that tries to keep a minimal amount of metadata, but does not focus on spreading heat across its zones, which can lead to improper heat distribution. Resetting SSTables in LN is trivial as an SSTable has an exlusive set of zones. On a delete of an LN SSTable, the zones are reset and the zones are given back to the zone allocator. To guarantee persistence, the metadata of the allocator needs to be stored. This metadata is stored to along with the metadata of the index structure (see Chapter 4.4) and is stored after each change in LN.

## 4.6 Background Operations

An LSM-tree makes use of various operations to move data from one layer to the next. How these operations are performed drastically alters how the LSM-tree functions. For example, they have an effect on the size of each component, the latency, and throughput of the database and how and when garbage collection is performed. Further on, most of these operations are disjoint and can run in parallel. They can interfere, which requires proper coordination for optimal results, depending on what is required. In this section, we will take a look at the individual operations and how they coordinate. They will be compared to existing solutions in RocksDB and LevelDB.

### 4.6.1 Flush Operations

Flushing operations occur when an immutable memtable is ready to be written to L0. This is done by a separate thread. LevelDB, RocksDB and TropoDB all schedule a flush operation, which can be picked up by a background thread. This flush operation reads the memtable pair by pair and writes the data to storage. LevelDB writes all of this data to one new file. TropoDB has a maximum size for SSTables on L0, different from LN. This is implemented because LN in TropoDB only supports bigger I/O at the scale of zones. This is not necessary for L0 and allows more fine-grained control. Therefore, in TropoDB multiple SSTables can be generated with one flush, by splitting the memtable, each with a different range of key-value pairs. Once the writes are complete, the changes are added to the index structure and then the new index structure is written to storage as well. This is the same for both LevelDB and TropoDB. After this operation, the immutable memtable is removed and the flush is completed. However, TropoDB adds one additional operation. As the memtable is inherently tied to one WAL, the WAL needs to be erased as well. Therefore, the flush thread also issues a reset of the entire WAL in question after the flush has been completed.

We mentioned that a flush can lead to multiple SSTables. This is because a flush can contain quite a few key-value pairs. Especially, when a large memtable size is picked. Therefore, a maximum L0 SSTable size is defined. If the size of a flush exceeds this size, it is split. Each split gets a guaranteed to be a disjoint set of key-value pairs. This prevents creating excessively large SSTables that need to be merged into L1 later on. The memtable split does not divide the memtable in equal parts. Instead, it cuts the SSTable right at the moment that it becomes too big. Therefore, it is possible for the last SSTable to only be a few key-value pairs in size, even if the memtable itself is large. As the data will not remain in L0 and the circular log has support for SSTables the size of pages, the effect is considered to be negligible.

### 4.6.2 Compaction Operations

Compaction can happen from any level between L0 and LN, but the data is always sent to L1 and up. The data structure for LN is designed to match garbage collection and garbage collection in TropoDB is designed to match LN. Garbage collection requires some data to determine what task it should do first. It at least requires a policy to determine if a compaction is necessary, what compaction should be picked first and what SSTables should be picked from each level for this compaction. It also needs to determine how big the resulting SSTables will be. Commonly, the level that is compacted *from* determines what SSTables it wants to compact for the next level and the level that is compacted *to* picks all overlapping SSTables. What SSTables are picked by the level that is compacted from can have a large influence. For example, on write amplification of the LSM-tree. Doing many small compactions instead of one large compaction can lead to extra write amplification, but large compactions can lead to expensive operations stalling clients (DR4). There exist various studies that examine compaction strategies in more detail (14, 15, 97, 102), but that is out of scope for this research. Instead, a strategy is picked that is similar to what LevelDB uses. LevelDB uses round-robin scheduling for picking SSTables, TropoDB does the same. This is possible by maintaining a *compaction pointer* pointing to the last picked SSTable. The next compaction then continues from this compaction pointer onward. This pointer is maintained in the index structure as well and requires one for each level. TropoDB does make one exception for L0. On L0, compaction always starts from the tail. This is necessary for the circular log of L0 to make progress. The SSTables that overlap in the next level are then picked as well. The picked SSTables are then used for the compaction.

**Limiting SSTable size**

For ordinary merges, it is possible for many SSTables to be involved. It is not advisable, to store the result of this merge in just one big SSTable. This big SSTable will be expensive to read by later reads and will both be an expensive target for later merges and will be merged into more frequently (it is likely to have overlap). It is considered better to restrict the size of an LN SSTable and to write multiple LN SSTables to storage after a compaction if necessary. This approach is taken by both LevelDB, RocksDB and TropoDB. A problem that this splitting can cause, is suboptimal space utilisation. Earlier, we stated that SSTables for LN reserve a round number of zones. If the maximum number of zones for SSTables is configured incorrectly this can lead to holes. Further on, it is possible that the last few LBAs of a compaction will be stored in a separate SSTable. This SSTable will claim quite a bit of space and will be hard to grow in size as it contains a small range of keys. It is important that this is prevented as much as possible. Possibly by increasing the maximum size dynamically for a few edge cases. This has not been investigated further as space utilisation is not the prime objective of this research and is left to future research. In the tests run for this research the maximum size was set to a little bit less than 2 zones. Not using a tiny bit of space at the end of zone 2 is done to avoid creating a very small SSTable at the end. The next SSTable that is written during the compaction needs to be at least the size of this tiny gap (else it would not have been added to a new SSTable in the first place).

**The compaction procedure**

Compactions consist of a few operations. The SSTables are read into memory, merged into new SSTables in memory, the new SSTables are then persisted to storage and lastly, the changes are added to a new version in the index structure. Nevertheless, not all of these operations need to happen in order or synchronous. TropoDB continuous on the compaction operation of LevelDB. LevelDB uses *lazy reading* (we call it lazy reading at least) to read its SSTables and uses a variant of merge-sort to sequentially create new tables. The lazy reading is done by creating an iterator existing out of two smaller iterators. One iterator is for L0 and one iterator is for LN. Data can be merged from these iterators, by reading the smallest key-value pair from both iterators iteratively and incrementing the iterator whose pair was picked. One iterator is for tables from L0 (not present with all merges) and one for LN. As all tables from L0 can overlap, they are immediately read and stored in memory from the start. The first iterator can be used to iterate over these SSTables in order (pick the smallest key from all L0 SSTables). The LN iterator makes use of the fact that no SSTable can overlap in LN. This requires only one SSTable from this level to be present in memory at any point in time. Therefore, only one LN SSTable is read at the start of the merge. Once the LN iterator finishes, a new SSTable is then loaded from LN. This prevents needing to read the entirety of LN into memory and allows for very large compactions. Once the result of the merge reaches a certain size, it is sent to storage and a new SSTable is allocated. Only once the write is done (synchronous or asynchronous), will the merge continue. Only once all data is merged, the compaction is considered done. At this point, the metadata of the new SSTables is added to the index structure and the old SSTables are marked as ready to delete.

TropoDB modifies a few parts of the compaction procedure to make it more optimal. These optimisations rely on the fact that not all operations of the merge need to be done sequentially and that there is full control over the storage with TropoDBs approach. Firstly, TropoDB *prefetches* a few LN SSTables from a separate thread (DR3 optimisation). As stated before, only one SSTable from LN is read and present in memory by default. This can cause multiple waits during the merge, where the compaction thread has to wait for a new SSTable to be loaded into memory, instead of doing merge operations. To alleviate this problem, it is possible to prefetch SSTables during the merge in memory. In this case fewer waits need to happen as the table might already be present in memory. Therefore, TropoDB preloads a few SSTables lazily with a separate *reader thread* to reduce the number of merge stalls. Prefetching can happen from any level, just like compactions. So, L0 and LN SSTables can both be prefetched. These reads happen while the compaction thread is merging. This also makes sure that the storage is still used, even during a merge, more efficiently using the available hardware. The reader thread internally limits the amount of SSTables that are prefetched to make sure that not too much memory is used and once a prefetch is no longer needed, it is unloaded.

The other optimisation TropoDB makes use of is deferring the writes of SSTables to a separate *writer thread* (DR3 optimisation). In LevelDB, the compaction thread has to wait before the new SSTable is written to the underlying file system. During this wait, more data could have been merged in memory as writing SSTables and merging are independent jobs. This can be especially problematic when SSTables are large and no buffering is used by the underlying storage system, which is the case for direct writes with file systems or always the case for fragmented logs in TropoDB. TropoDB solves this issue by sending the SSTable to the aforementioned writer thread instead. The task then still needs to be done, but it will not be done by the compaction thread itself. This *writer thread* can hold a few SSTables in a memory buffer and writes the SSTables this buffer contains to storage in order. This allows doing both writes and merges in parallel (task parallelism). However, once the buffer of the writer thread is full, the compaction thread still has to wait for space to become available. Similarly, once all merges are complete, the compaction thread has to wait for the writing thread to finish. This ensures that the compaction itself is still synchronous.

**Lazy compaction by only updating metadata**

It is possible to have compaction operations in which only 1 SSTable is involved. This can happen when 1 SSTable moves to the next level, but has no overlap. In this case no merge is necessary and the SSTable can simply be written to the next layer as a copy. The compaction operation is thus copying the exact same data to the next level and altering the metadata to represent the next level. This only requires reading one SSTable, writing one SSTable and altering the index structure. However, SSTables on LN live in the same region. It is thus not necessary to physically rewrite the data. Therefore, as an optimisation, when an SSTable is copied in any layer above L0, the table itself is not even physically copied (Not DR2, but similar). In this case no SSTable reads or writes are issued. Instead, only the metadata is updated. Such an approach can reduce write amplification effects of compactions to a limited degree.

### 4.6.3 Background Operation Coordination

A common optimisation to reduce the effects of compaction is to do multiple compactions in parallel. For example, compactions from L1 to L2 do not depend on data from L3. Therefore, it is possible to do compactions on both L1 to L2 and L3 to L4 at the same time. This can prevent long stalls (DR4), where many levels need compactions and need to wait on one another. If this queue becomes large enough, it is possible that eventually there is no space left to flush. At this point, there is essentially a traffic jam of compactions and client operations will be stalled. Doing multiple compactions in parallel can break this jam (DR3). However, doing multiple compactions at the same time requires more CPU cores to perform and will ask more from the storage. Every additional concurrent compaction will increase the time each of the other compactions takes as they contest over the same resources. It is a trade-off. Another optimisation that is orthogonal is making use of preemptions. For example, if during a compaction another important background operation is required that is more urgent, the current task is put on hold, and the thread moves to the other operation. This can be seen as an *in-software context switch*. It can prevent having to wait on a flush that can not continue because of another operation, but it does cause stalling issues further down the LSM-tree.

LevelDB uses just one garbage collection thread in total (135). This thread is shared between both flushes and compactions. To alleviate this problem, LevelDB does allow for the preemption of tasks. Nevertheless, with such a design only one task can be busy at most, even with preemption. RocksDB takes a more advanced approach and allows running multiple background threads in parallel (15, 70). These background threads can each do either a compaction or a flush and a maximum number of both flush and compaction threads can be specified. This allows configuring how many threads make sense to use for flushes and/or compactions. Similar to LevelDB, some degree of preemption is available. This design can solve many congestion problems and is also beneficial for SSDs as SSDs have the capability to do many operations in parallel. Only using one background operation would not allow this.

TropoDB also allows doing multiple operations in parallel (DR3). The thread model TropoDB uses is visible in Figure 4.18. TropoDB uses multiple operations in parallel to match the concurrency of the SSD and lower the effects of stalling, as latency stability is one of its goals. TropoDB also has some congestion issues in L0. This is because the maximum size of L0 is limited to its pre-allocated zones. Once it is filled, there is no more space left for L0. This makes it important that data will move to at least L1 at a regular interval, to prevent stalling. However, TropoDB originally inherited its compaction process from LevelDB, which is not trivial to copy to RocksDBs design. Therefore, a design that is in between LevelDB and RocksDB is used instead. TropoDB uses a set number of background threads, each with their own predefined role. Essentially, preemption and background threads are moved into one concept. TropoDB always uses exactly 1 flush thread and 2 compaction threads. The flush thread is only responsible for appending SSTables to L0. In the case no more space is available, it simply waits till space becomes available. One of the compaction threads is known as *L0Compaction* and is responsible for moving data from L0 to L1. It also in charge of cleaning the old data in L0. This thread maintains the bridge between the circular log region of L0 and the fragmented log region of LN. This compaction in particular gets its own thread because of the aforementioned problem with the size limit of L0. The other compaction thread is responsible for compactions from L1 and up and cleaning up the fragmented logs from L1 and up.

**Stalling client issued puts** A technique that is used by both LevelDB and RocksDB

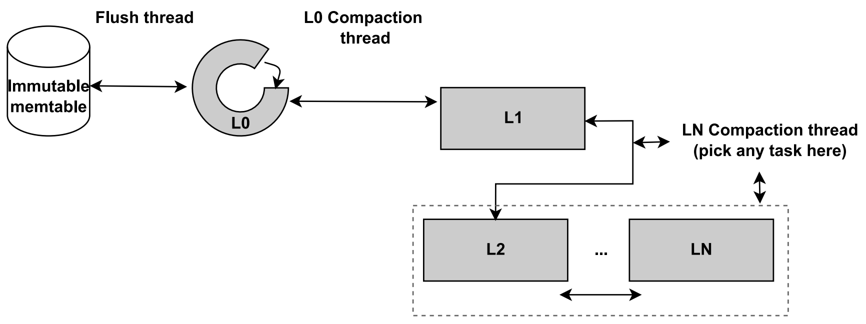


**Figure 4.18:** TropoDB design: Background threads

that is important to mention is stalling writes when the key-value store is *busy*. This is important as it has a big influence on latency experiments we discuss later on in Chapter 6 This delay is caused by giving *put* operations a small extra delay when the database is busy. For example, by making the thread *sleep*. If such a stall is not done and put operations keep on writing, eventually there might be no space left in the memory table as a flush still has to complete. A few client-issued puts would then have to wait for long periods of time, but most clients would have a lower latency. This creates a skewed latency graph and is detrimental to tail latency. Stalling writes can make latency stable for more clients. Instead of making a few clients wait long, many clients will have to wait a bit longer, provided the flush can finish on time. It can thus be used as a *latency stabiliser* (DR4 related, but not for storage). TropoDB also uses a delay to control the latency and uses a similar approach to LevelDB. When L0 reaches a certain size, every put operation has to wait for a short while. This size is set to be significantly larger than the trigger that causes compaction from L0, indicating a large queue of compactions is awaiting in L0 and a traffic jam is likely to occur soon.

**Addressing the design requirements**

The background operation design addresses the design requirements (and RQ2) in the following way:

- DR1: Background operations do not do I/O themselves.
- DR2: A few design decisions are made to reduce the overwrite issues of background threads. LSM-trees basically overwrite data by rewriting the data to the next level, which causes write amplification effects. Lazy compaction in LN ensures that if tables are merely copied in LN, they are not rewritten, reducing write amplification effects. Further on, tweaking the size of both L0 and LN SSTables can allow making trade-offs in sizes. If the size of SSTables in L0 is big, it is likely to overlap with more SSTables, which can lead to bigger merges in L1. This can lead to more expensive compactions in latency, overwrites and resets, but using a smaller size will require more merges in the end. Each flush can lead to multiple smaller tables that each require a merge later on. The different compaction threads also indirectly influence write amplification. If the L0 to LN is more aggressive than the LN compaction threads, it can lead to L1 flooding and the LN compaction thread being able to keep up. This in turn will lead to large write amplification in L1 because of frequent merges. Resets are assigned uniquely to threads in TropoDB. The flush threads erase WALs, the L0 to L1 compaction thread erases L0 data and the LN compaction thread erases LN data. This design allows for a low amount of locking and proper separation of responsibilities, but it does inherently lead to more resets. However, as all background threads make use of eager resets, it can lead to more resets than

alternative designs. All threads make sure that all of the invalid data is erased. Other designs could have postponed said erasures.

- DR3: Multiple background operations try to make effective use of the available parallelism. Flushes, L0 compactions and LN compactions can all be done in parallel. Each of these operations uses a different selection of active zones on the SSD and can completely proceed in paralllel. Additionally, compactions make use of multiple concurrent reader threads to prefetch SSTables and allow for deferring writes. These optimisations allow for task parallelism in the key-value store and make effective use of both the storage capabilities and the host capabilities.

- DR4: Many design decisions have been made to reduce latency issues. The default LevelDB design only has one background thread. This can lead to long waits for background operations to finish. Especially when it is possible to do multiple operations in parallel. By using multiple background threads, clients have to wait shorter for flushes or L0 to L1 compactions to complete, which can reduce tail latency. Properly adding a bit of delay once the L0 log starts to fill up can also aid in tail latency, because it prevents flooding the L0 log and ending up in a situation where clients have to wait long for background threads to complete. Delaying prematurely can allow more data to move out of L0.

**Disadvantages of the proposed design**

The proposed design is not an ideal design and leaves room for optimisation. Initially, most logic was built on top of LevelDBs background thread model, which did not scale as well as was required for TropoDB in the tests conducted. Above all, it could not achieve proper tail latency (DR4). Therefore, the design had to be altered, but moving to RocksDBs background thread model was incompatible with the components that were already built. As a result, a design was picked that was halfway between LevelDB and RocksDB. This design is made to do more in parallel (DR3), but still less so than RocksDB.

The biggest issue lies in the communication between the L0 compaction thread and the LN compaction thread (DR3). This happens because both can have compactions in L1. The L0 compaction thread always compacts its data to L1 and the LN compaction thread can compact from L1. If both a L0 compaction and an LN compaction are issued at the same time and their tables overlap, synchronisation is needed and compactions can be stalled. The result is that it is not always possible to do concurrent compactions and that in the worst case the design approximates a design with one compaction thread. This can be solved by prematurely ensuring that the compactions will not use SSTables that overlap or adding logic that can support overlapping compactions, at the cost of some extra write amplification (dead tables are written again).

The next issue is that there is only support for one L0 to L1 compaction thread and one LN compaction thread. This limits scalability (DR3 and DR4) and is not able to use the full potential that some SSDs can give. RocksDB can use an arbitrary number of compaction threads. It would have been better if TropoDB used a similar design, but it does not. The reason is that it is not trivial to move TropoDBs design to a model with an arbitrary number of threads. The compaction procedures contain various contention points that are not thread-safe. For example, the erasures in L0 and LN. These operations are atomic and replace all current deletes for their level. In order to support an arbitrary number of threads, these components need to be significantly altered to support this functionality. We recommend future designs do this from the start. The current design in which each compaction thread is responsible for different tasks (task parallelism) is a mitigation strategy in which concurrency issues do not occur because there is only one thread addressing them.

### 4.6.4 Summary

TropoDB makes use of a unique background operation model. TropoDB makes use of a fixed number of concurrent background threads (DR3), each with a different purpose. There is a *flush thread*, a *L0 compaction thread* and a *LN compaction thread*. The flush thread is responsible for flushing immutable memtables to L0 and resetting old WALs. WALs are reset eagerly, which means that after a flush all WALs are removed that are no longer used. The L0 compaction thread is responsible for compactions from L0 to L1. It reads SSTables from the circular log of L0 and merges them with SSTables in the fragmented log of L1. It is also in charge of deletions in L0. Deletions in L0 are done eagerly, which means that after each compaction all SSTables that can safely be deleted are deleted. Further on, SSTables in L0 are removed when there no longer is any space left available and SSTables can be safely removed. The LN compaction thread is in charge of compaction from each level after L1 onward. It decides what LSM-tree level needs compaction the most and then executes this compaction without interruption. It is also in charge of all SSTable deletes from level L1 up to LN. These resets are also done eagerly and are done immediately after each compaction. Deletes in LN are also triggered when space runs out. To guarantee that the L0 compaction thread and the LN compaction thread do not interfere, they have the ability to coordinate (DR3 and DR4 issues).

## 4.7 Increasing Concurrency

The default LSM-tree design is serialised and has problems with a few bottlenecks. These bottlenecks cause other operations to stall and hurt latency stability (DR4). For example, the need to wait for a flush to complete when an immutable memtable already exists and a new memtable is filled. With modern hardware, it is not necessary to limit the database to use a serialised approach (DR3). Modern hardware such as SSDs support doing multiple operations in parallel, so serialising is not ideal. Therefore, it is not necessary to limit database design to only one mutable/immutable memtable pair and one “concurrent" flush. Instead of waiting it should be possible to do multiple flushes concurrently, which is supported by the parallelism capabilities of SSDs. Such designs are known to have merit and have, for example, already been tried out for *Open-channel SSDs* (OCSSDs, a precursor to ZNS). For OCSSDs it is possible to control individual channels of the SSD and do as many operations in parallel as there are channels. Some key-value stores make use of this characteristic to increase parallelism. For example, LOCS and FlashKV. LOCS uses a memtable for each channel of OCSSDs (124). In this case flushes can use the full bandwidth available of the SSD and do not idle because of database design. ZNS SSDs do not come with full control over channels, but they do allow using multiple zones concurrently. This concurrency is limited by the number of active zones (Chapter 2.1.5). Each of the active zones can be used for different I/O operations. This allows various operations to happen at the same time and can be used similarly to LOCS. Therefore, it was decided to investigate if and how the extra active zones can be used to get the same effect. TropoDB by default only requires 5 active zones: 1 for metadata, 1 for WALs, 1 for L0 and 2 for LN. If more active zones are available than 5 on the SSD used, such an approach makes sense. Otherwise, it does not. Similarly, it might be that increasing the number of parallel structures has a detrimental effect on performance. We already know from decades of research that increasing the number of threads for a problem can also decrease performance or hamper latency stability. For example, io_uring needs a lot of multithreading in some cases and if these are not available, it will lead to performance degradation (40). Therefore, increasing concurrency should always be optional and configurable.

TropoDB needs to reserve zones beforehand and can not make it fully optional. Instead, this is only optional for database creation. On database creation, the level of concurrency should be specified, which is then used by the inner structures to determine the layout. The approach TropoDB takes to match the number of active zones and increase parallelism is similar to LOCS. TropoDB allows increasing the number of memtables dynamically.

Each memtable will then be linked to a unique WAL, a separate flush thread and its own circular L0 log. The level of concurrency can be safely set to anywhere between 1 and $floor(\frac{maximum\_number\_of\_activezones-3}{2})$. The number "3" in this formula is based on the fact that in TropoDBs design there will always be two compaction threads in LN that can lead to one active zone each and that the metadata log can also have an active zone. The division by "2" is based on the fact that for each additional level of concurrency (from 1) 2 additional zones can be opened: one for WALs and one for L0. With this design, most active zones can theoretically be used by TropoDB. To support this feature, the database components need to be altered, we will explain the changes applied to each component in this section. The resulting design can be seen in Figure 4.19. Notice, that the number of circular L0 logs, WALs and memtables has increased.

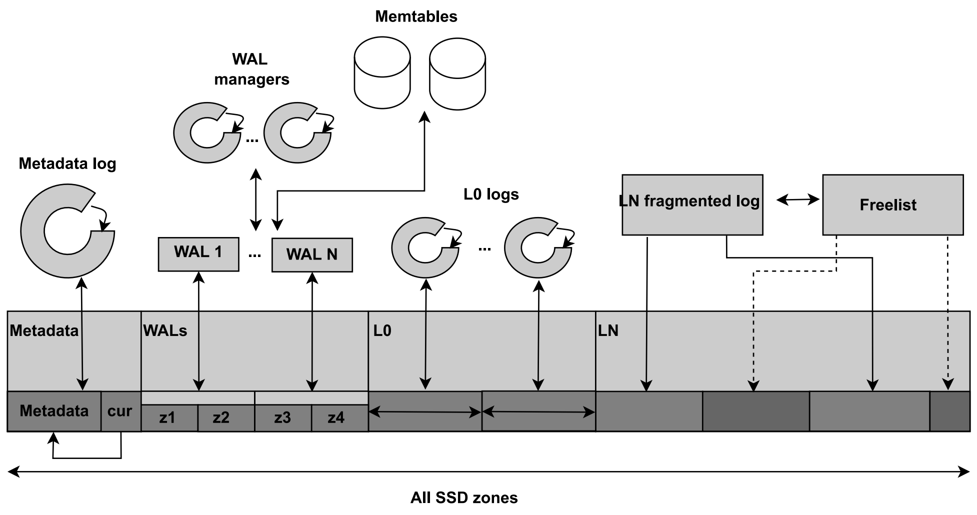


**Figure 4.19:** TropoDB design: Dividing an SSD among LSM-tree components with added concurrency capabilities

### 4.7.1 Concurrent Memtables and WALs

TropoDB has limited support for *concurrent WALs*, which are multiple WALs that are live concurrently (DR3). Each WAL is linked to its own unique WAL manager, memtable and immutable memtable (if any) in a 1:1:1:1 mapping. The linking is exclusive, which means that each component can only be linked to one of each kind. So for each concurrent memtable, there is one unique WAL manager and one unique live WAL. In addition, each WAL manager has its own distinct selection of zones. The individual WAL managers have no knowledge of each other and do not need knowledge about each other. They all have their own circulation scheme to spread the temperature of their own zones. On database creation, the zones that would ordinarily be given to one WAL manager, are divided evenly among the WAL managers in contiguous regions of zones. We thus now also have a log of WAL managers.

On database startup, each memtable is linked to a WAL manager. This WAL manager thenn gives the memtable a WAL from its WAL log to use. When the memtable needs to be refreshed (flush operations), the memtable requests a new WAL from its WAL manager. All changes applied to this memtable are only applied to this WAL. The WAL is not aware that there are other memtables or WALs, therefore, nothing is inherently different in the WAL design. The same holds true for the design of individual WAL manager and memtables. In fact, the only changes needed lie in the implementation of the *put* and *get* operations themselves and some logic on top of the memtables.

Put operations need to decide what memtable/wal combination to use and get operations will need to fetch the most recent change among all memtables. We will explain how both are done in TropoDB and we will explain some alternatives as well. To guarantee that the most recent change is picked, a simple solution is to look for the change in all live (immutable) memtables and pick the most recent change, which is possible as each change comes with a sequence number, that is still synchronised globally. This approach adds a little more read amplification, but only for the in-memory parts of the database. This is the approach TropoDB uses. Put operations are less trivial to design and can be done with various strategies. A challenge is that there can be fewer or more memtables/WALs than there are concurrent put threads. It is thus not as simple as linking a memtable/WAL to a separate put thread. Such a mapping can be done, but is not investigated in this research. During the put request itself, it is decided what memtable/WAL will be the target. Examples of such approaches are picking the memtable that contains a similar range of key-value pairs as the data present in the put operation, using round robin scheduling or using a hashing algorithm. In TropoDB a simple design is picked as the focus of the research lies on the storage aspects, the updates are applied in a round-robin fashion. Each memtable/WAL gets a write in order. This design is picked because it evenly distributes writes. Further on, in such a design a WAL is unlikely to get two writes in a row, decreasing the load one WAL has to endure (queue depth of appends). However, the approach taken by TropoDB does have a number of issues. For example, because round-robin is used, all memtables are likely to flush at similar times. Additionally, it is not a true concurrent memtable design (DR3). That is because all updates still arrive at the same put operation that contains some locking to determine what memtable to pick. Further research can look for a more optimal solution or an alternative strategy. There exist various concurrent

WAL implementations that are known to scale (32, 35), such approaches might achieve more success.

The main advantage of the concurrent WAL/memtable design is that multiple updates can be done concurrently to different zones as each WAL has its own zone head. This approach can, therefore, improve performance and better match the maximum performance that the SSD can give. However, the exact effect has only been measured for the entire concurrent implementation, which is shown in Chapter 6.7.

### 4.7.2 Concurrent L0 Logs and Concurrent Flush Operations

A problem with the circular log design of L0 is that circular logs are limited to one write head only. This necessitates serialising multiple flushes and limits concurrency capabilities. In the previous section, we mentioned that in the concurrent design there can be multiple memtables and immutable memtables. Each of these immutable memtables can be flushed. These memtables are disjoint, but are still serialised to storage when one circular log is used. This misses an opportunity to do multiple storage operations in parallel. When multiple L0 logs are used, this effect is removed. As each concurrent flush can happen to a different log. In TropoDB one circular log is used for each additional concurrent immutable memtable. This ensures that each immutable memtable can always be written to one circular log and does not have to wait for other flushes from other memtables to finish. Using concurrent flushes in turn can help with reducing performance penalties when clients are waiting for flushes to complete. Concurrent L0 logs are supported by TropoDB and are possible because each circular log used by L0 gets its own distinct set of zones. This makes it possible to add multiple L0 circular logs, each with its own set of zones, at the cost of one extra active zone. Each flush, can then write to any of the L0 logs. Further on, as reads on L0 are issued to a physical block address and not to a distinct log, no change is needed for reads to support this feature as well. Resets do require some workarounds. Resets are hindered by the tail of each log and L0 logs are only allowed to do resets on their own selection of zones. Therefore, when concurrent logs are used some metadata is added to L0 SSTables, which indicates the *log number*. On a reset, each L0 log should only get reset requests that correspond with its own log number.

As stated before, there is an equal amount of live memtables/immutable memtables and circular L0 logs. Each memtable is, therefore, also linked to a unique circular L0 log and each L0 log gets its own flush thread. The updated model of background threads can be seen in Figure 4.20. Once one of the memtables is full and converted to its immutable memtable, it notifies its personal flush thread. This flush thread functions exactly the same as ordinary flush threads, but only appends to its assigned L0 log. Compaction threads do not require knowledge of the L0 logs for reads. They read from any L0 log and merge the data into SSTables for L1. However, they do require knowledge about the logs for deletion. On a deletion, they only send the deletes to the logs corresponding with the metadata of the SSTable to delete. This can be done because we store the log numbers along with the metadata of SSTables in L0.

This design works, but has one issue that makes such a design inoperable: ordering. Concurrent flushes can happen in this design and are in fact its purpose, but concurrent flushes also mean that the ordering of appends to L0 is no longer correct. On a read, only the most recent change needs to be read. In the case of memtables, this can be solved trivially. Always read all live memtables and pick the most recent change at the cost of read amplification. However, at L0 this is unfeasible as that requires reading many SSTables from storage. To circumvent this issue, an additional versioning number is added on top of the versioning numbers used for individual SSTables (see Chapter 4.5). This versioning number is also stored along with each L0 SSTable. Each update to L0 gets its own version numbering, which is incremented on each L0 update. All SSTables added by one flush, get the same L0 version number. This allows reading L0 in order of freshness and solves the earlier stated challenge.

### Addressing the design requirements

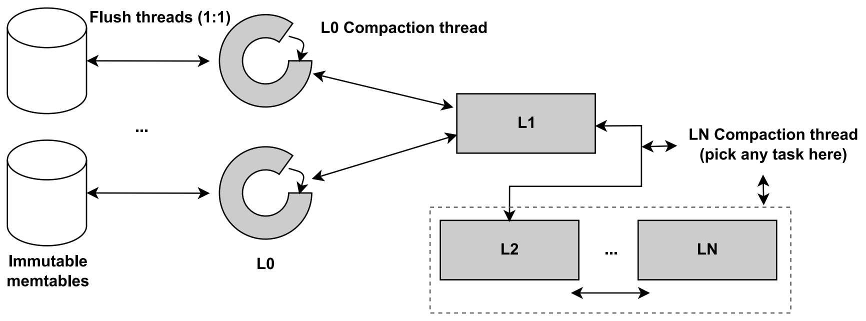


**Figure 4.20:** TropoDB design: Background threads with added concurrency capabilities

The concurrency design addresses the design requirements (and RQ2) in the following way:

- DR1: All datastructures still make use of appends.
- DR2: We will discuss overwrites only in terms of write amplification for now. As the number of overwrites is not inherently different for the underlying structure. Using multiple concurrent structures can lead to either more write amplification and resets

or less write amplification and resets. It depends on the individual components and the split. The WALs and L0 are split, but are still internally the same structures. Each write still only happens to one of the respective components, so there is no data duplication. The amount of data written to storage should still be inherently the same. For example, a write happens to one of the WALs and there is still the same amount of WAL regions in total. For L0 the situation is a bit different. The size of L0 circular logs can have an effect on write amplification. This is a side-effect of the concurrent design, but not a goal. The proposed design does not explicitly focus on reducing/increasing write amplification, but its effects are properly investigated to allow picking a level of concurrency in which the effects on the number of resets are positive.

- DR3: The concurrent design focuses on this functional requirement. It adds extra concurrent WALs and circular L0 logs. Both can be addresses concurrently, which can allow to use parallelism of the ZNS SSD more effectively.
- DR4: Using multiple concurrent structures can more properly utilise the storage device. This can lead to more concurrent I/O operations and prevent postponing expensive operations. If instead multiple smaller I/O operations are issued concurrently, the total time of the operations can be reduced. This can in turn result in clients needing to wait shorter periods of time if flushes or compactions are needed before puts or gets can proceed. However, using more I/O for the background can result in less I/O being available for the client. So it can also have the opposite effect by increasing the average latency of client operations or causing latency peaks during heavy background operations. Using multiple concurrent logs can also result in concurrent logs being filled sooner and require client operations to be stalled earlier. Using multiple objects concurrently can have a large number of effects. The effects are explicitly tested. This can aid in coming up with a design that fits the concurrency available, but does allow for stable latency.

**Disadvantages of the proposed design**

The proposed design has various disadvantages. It was mainly designed and implemented to test the effect of adding extra concurrency on top. It is thus more meant for research and less for actual use. If benefits were measured with this design, which there were not (see Chapter 6.7), it would have led to a more refined design than this proof of concept. The result is that design is inherently less refined than for the individual components and misses various optimisations. A few of these need to be explained in more detail.

Concurrent WALs are not really concurrent (FR issues). Writes are striped, independent of the thread used. Each write is sent to a different memtable and WAL. However, the design still needs some synchronisation before each put to determine what memtable and WAL to pick. Additionally, there are striped to a *group log* as group logging is used(see Chapter 4.3.3). The result is that there is a bit of overhead on top of each call. A truly concurrent design should not require this, by for example assigning WALs to threads as done in SplinterDB (35). Using concurrent WALs in the used design has minimal performance benefit for memtables and only makes sense if the WALs can not keep up with the updates it receives. If one WAL is busy, the update can then be seny to another WAL. This happens with synchronous I/O or if the asynchronous WAL queues fill up. A side-effect is that the workload send to the SSD increases significantly depending on the number of WALs; each WAL added, leaves less bandwidth for other structures. Striping is also problematic for other reasons. All WALs fill up at equal speed, which means that it is likely for all flushes to occur at the same time. This can lead to a burst of flushes and it would have been better to spread this workload by doing the flushes at different intervals. Further on, striping is irrespective of the data. This means that data can be in any of the memtables and no sorting is done. The result is that all WALs can contain very similar data and eventually all flushes will lead to overlapping SSTables. This could have been reduced by sending different data to different memtables/WALs. This would have led to less overlap and fewer compaction problems later on.

Multiple concurrent L0 logs lead to various issues. The original design was not designed to support concurrent L0 tables. There is still only one L0 to L1 compaction thread. The circular logs can only clean up their data if SSTables are removed on their tail. This was done by forcing compaction to at least pick one SSTable on the tail, but this is not a good idea to do with concurrent logs. The tables might not overlap, which can lead to non-related SSTables being used for more expensive merges in L1 (more overlap in L1). TropoDB tries to solve this by rotating what circular log is guaranteed to have its SSTable used for the compaction, but this still means that not all logs will make progress. Similarly, it is possible for the merge to only pick SSTables from one of the logs. The result is that the number of live and dead SSTables can remain the same for a few logs and that if a flush waits for this log to become clean, it has to wait. If in turn a memtable is filled, the client will have to wait for the flush to finish and a compaction to clean the circular log to finish. This can lead to unpredictable latency. This problem can be reduced by picking a different compaction strategy or using multiple L0 to LN compaction threads.

Since there are multiple flush threads and only one L0 to L1 compaction thread, we also have a congestion issue. There are more threads that are able to quickly fill up L0 with flushes, but fewer threads to move data out of L0. This can lead to better performance in the short run, but once L0 becomes full, it will remain congested. All flush threads will regularly have to wait for data/garbage to move out of L0 and the effects of concurrency will be minimised. We will end up in a state where the same amount of work is done as in the non-concurrent implementation, but more threads are idling and consuming CPU resources (DR4). This should (perhaps contradictory) be solvable by using more threads from L0 to L1, to alleviate the congestion point. However, then the problem will move to the compactions in LN. Essentially, the problem will move further down the tree.

The last challenge lies in the get implementation. Using multiple concurrent structures will result in more structures that need to be read. It requires a read from multiple memtables and multiple L0 logs (L0 logs can contain different data of different ages), which will lead to read amplification. This is only justifiable to use in cases where increasing write throughput and reducing write latency is more prudent than reducing read amplification and when it is known that the concurrent design can in fact increase write performance.

### 4.7.3 Summary

TropoDB supports multiple concurrency levels. By default TropoDB uses 5 active zones at most, which does not use all resources available on some ZNS SSDs (DR3). Extra concurrency is added by increasing the number of concurrent memtables, concurrent WALs, circular L0 logs and flushs threads. Each extra level of concurrency adds one of each, which are al linked together in a chain. So, each memtable uses a different WAL, flush thread and L0 log. All put operations are striped across the concurrent memtables/WALs. Once a memtable or WAL is filled it is flushed with its personal flush thread to its L0 log. The L0 compaction thread can read data from any L0 log and is not aware that there is more than one. To guarantee correctness and ordering, more metadata is added to each SSTable. On a read, all live memtables are read and the most recent change is returned to the client. If the data was not available in any memtable, the most recent L0 SSTable is read, which is determined with the help of a sequence number in the SSTable. If no L0 SSTable contains the data, the read proceeds to LN as usual.

## 4.8 Summary

TropoDB uses a novel design that is optimised for ZNS Storage. Its design answers RQ2 and tries to use unique optimisation opportunities given by ZNS. Such ZNS opportunities include append operations, limiting number of overwrites and resets, implicit concurrency capabilities and control over the garbage collection process, which can aid in latency issues. TropoDB splits the ZNS storage in distinct zone regions for each component. Each individual component gets a data structure, number of zones and a garbage collection scheme, optimised for its use-case. Further on, all data structures only make use of appends, no writes are used. Metadata of the index structure is stored in a circular log in the first few zones of the SSD. WAL data is stored in multiple once logs, uses a novel circulation scheme to spread heat and makes effective use of appends to increase concurrency capabilities. SSTables in L0 are stored to a circular log and SSTables in higher levels are stored to fragmented logs. All background operations make use of eager garbage collection procedures. TropoDB uses three background threads by default. One background thread is in charge of flushes, one background thread is in charge of compactions from L0 to LN and one background thread is in charge of compactions in higher levels. To support higher concurrency, TropoDB supports adding multiple concurrent memtables, WALs, flush thread and L0 logs.

# 5

# Implementation of TropoDB

In this section we answer RQ3: *"How to implement and evaluate the design of an LSM-based key-value store on ZNS devices?"*. TropoDB is an implementation of the key-value store design given in Chapter 4 for the storage engine SPDK on GNU/Linux and closely matches the proposed design. Nevertheless, implementations are always different than designs. In the case of TropoDB, it contains some deviations and optimisations that are made possible because of SPDK. Additionally, we did not discuss the components of the database that are completely in-memory. For example, memtables or the SSTable cache. In this section we will describe these decisions/implementations as well, why they were chosen and if they have any disadvantages that are worth mentioning.

We will explain our implementation in a number of steps. We begin by setting requirements for the implementation. We will then explain our lower storage API, SZD. This will be followed by looking at how we intent to evaluate the implementation. To continue, we will look at our implementations for data structures for both storage and memory. We will finish by looking at the implementations for background operations and the key-value store deletion operation.

## 5.1 Requirements

There exist various possible implementations of the proposed design. It is important that we set up a number of requirements to ensure that we come with an implementation that does the design justice and addresses the main research question. We define the following **Implementation Requirements (IR)**:

- **IR1**: *The implementation should be build bottom-up in RocksDB*:
  In Chapter 3 we have stated we will built TropoDB in RocksDB with bottom-up

approach. This should be adhered.

- **IR2**: *Storage should be managed in user-space*:
  In Chapter 3 we have stated we will built TropoDB in user-space with SPDK. This should be adhered as well.

- **IR3**: *TropoDB should rely on one ZNS SSD only*:
  TropoDB is an implementation for ZNS SSDs only. It is not allowed to make use of other devices to store extra data. For example, storing metadata on a different disk that does allow for random I/O is not allowed, even for debug storage. Remember, we are building for ZNS here and should not force an implementation that requires extra non-related storage devices. Additionally, TropoDB should not require the use of multiple SSDs as well. In order to evaluate the system, concurrency should be testable. When multiple devices are used, such an evaluation is hindered.

- **IR4**: *The implementation should support db_bench with minimal alterations*:
  The implementation will be evaluated using db_bench. This requires the database to support db_bench. It is not allowed to modify db_bench beyond adding support for interfacing with TropoDB. For example, some minor fluctuations are needed to allow db_bench to interface with TropoDB, but no workloads should be altered. The system should be evaluated *fairly.*

## 5.2 Simple ZNS Device (SZD)

The implementation of TropoDB is split into two parts: the database itself and a library to interface with SPDK. This allows decoupling the database logic from the storage logic. The interface with SPDK is named *Simple ZNS Device* (SZD) and handles all generic I/O logic. We consider it necessary to shortly explain SZD as SPDK differs from other storage APIs and SZD is not raw SPDK itself. SZD itself is split into two parts of code. A low-level C API and a high-level C++ API. The resulting layered design looks like Figure 5.1.

The SZD C API directly calls SPDK functions. It is in charge of discovering, opening, closing and reopening ZNS devices. This is entirely name-based on PCIe addresses, making it trivial to open devices based on their PCIe addresses. It comes with functions to allocate DMA-backed memory through huge pages and release it. Additionally, it comes with synchronous and asynchronous I/O functions. It supports commands for appends, reads, retrieving zone heads, resets and zone finishing. It does not expose writes to users by design, it is append-only. All I/O is done with SPDKs *QPair* implementation, which is

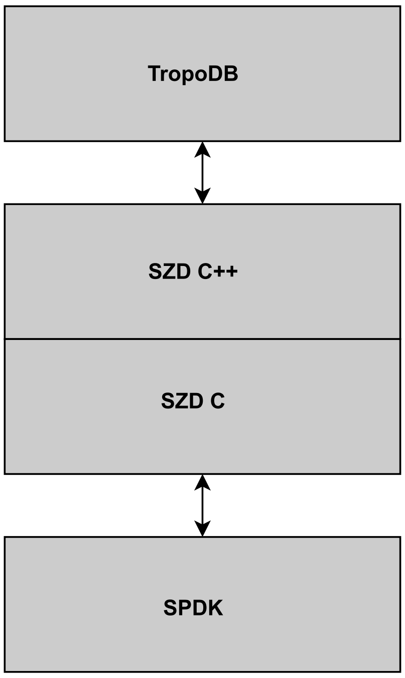


**Figure 5.1:** The software layers in TropoDB

an abstraction around software/hardware queue pairs. There can be more software QPairs than there are hardware queue pairs for example. Qpairs are not thread-safe and require external synchronisation. In addition, all buffers require the memory to be present in *huge pages* to properly make use *Direct Memory Access* (DMA), which in turn allows the storage to read and write to memory. This is also left up to the host to guarantee. As we will discuss later on, the C++ API hides some of this logic away. Synchronous I/O makes use of busy polling and only returns once the polling it conducts, has either completed or errored. Asynchronous I/O returns an object which can be polled at any convenience. The operation supports cleaning data on completion, but it needs to be polled at least once in this state. SZD does not contain any interrupt logic. SZD does come with various forms of error handling and automation. Error handling prevents I/O from being sent that would be erroneous and prevents out-of-bounds I/O for, for example, appends to incorrect read addresses. Automatisation is done to prevent having to deal with all ZNS intrinsics. Appends larger than ZASL are split into smaller parts and reads larger than MDTS are split into smaller reads. Further on, when I/O crosses the border of a zone, it automatically moves on to the next zone. A final feature that is unique to SZD is simple containerisation. SZD allows specifying the minimum and maximum zone that an application can use. When an application is initialised this way, no I/O may cross their respective border. This should make it possible to store data from multiple applications or databases on the same device.

The C++ API adds many conveniences on top. For example, it allows creating a DMA-backed buffer. This buffer can then be reused when necessary. This prevents repetitive patterns of allocation, delete, allocation and delete. The C++ API also does not directly expose QPairs, but comes with a simple abstraction on top, which we refer to as *channels*, a reference to OCSSDs (25) but unrelated in practice. Channels come with convenience functions for I/O. A major feature of channels is that they require setting a specific range to which they may conduct I/O. This prevents out-of-bounds issues that could occur. For example, WAL channels should not have access to L0 zones. It is possible to send I/O requests without DMA-backed memory in which case the channels themselves create some DMA-backed memory, copies the data to this memory and then does the deletion of this *new* memory automatically in the background after completion. Because DMA in SPDK is based on hugepages, the number of DMA memory that can be allocated is limited and I/O is limited to ZASL and MDTS, it is problematic to deal with large I/O. Therefore, large I/O is split in smaller parts of ZASL and MDTS in this case. Only a buffer of up to ZASL is allocated for appends and MDTS for reads. The I/O is then sent in parts and the buffer is reused. Additionally, it allows writing and reading to any address in the earlier mentioned DMA-backed buffer, allowing for simple address virtualisation. Asynchronous I/O comes with polling abstractions on top and automatic memory management. Lastly, it comes with some diagnostic tools on top of the channels. Every reset, write or read can be tracked and requested at any moment. These variables are implemented as atomics to guarantee that there will be no concurrency errors. Channels themselves are, however, still not thread-safe.

SZD also builds some basic data structures on top of the channels. For example, the *once log* used in WALs, the *fragmented log* and its allocators used in LN and the *circular log* used for metadata and L0. TropoDB only adds a little bit of data on top of each of these structures to fit its needs. Locking and thread synchronisation is still kept to a minimum. This is mainly done by allocating multiple channels for a structure. For example, using a channel solely for reads and a channel solely for appends. Solving the problem of concurrent access to QPairs. How the implementation of these structures functions will be described component by component in their respective sections.

A major problem and design flaw with the current implementation of the SZD C++ API is that the logic of zone sizes and zone capacities is hidden away. The entire ZNS device is exposed to the user as a set of zones with a zone size equal to the zone capacity. Internally SZD translates all I/O back to the correct locations. To do this SZD only support one homogeneous zone capacity that must be the same for all zones. So zone capacity may be different than the zone size, but must be the same everywhere. This was implemented by mistake as SZD was originally tested with the incorrect assumption that zone capacity would be constant. This is an implementation flaw of SZD, not a design flaw.

## 5.3 Benchmark Tooling

TropoDB can reuse most benchmark tooling from db_bench (28). This is enough for retrieving information such as put and get operation tail latency, but it provides no metrics that concern I/O itself. Therefore, custom designs are made for TropoDB and SZD. Every append operation logs the number of appends needed for the operation (updates that are bigger than ZASL require more than one), the bytes written by the append and the zones affected by the append. The same is done for all reads and resets. Retrieving this information is made thread-safe with atomics and allows getting insight into how much I/O has happened and where it has exactly happened. This adds a bit of extra overhead to each call. The information is lost on database shutdown as it is not stored. Latencies for WAL operations are measured around their respective calls in nanoseconds. For recovery of WALs this means measuring around the recovery procedure of one WAL and for WAL appends this means measuring around the call to the once log of WALs (only concerning the I/O functionality, not the database functionality).

## 5.4 Implementations of On Data Structures for Storage

Various storage structures have been designed for TropoDB. In this section we will describe how these structures are implemented. All of these structures make use of general structures that are implemented within SZD. Therefore, we will describe the implementation of the storage structure, by describing the used SZD components and what TropoDB components make use of these components. In particular, we will look at SZD once logs and how WALs make use of them, SZD circular logs and how both metadata and L0 SSTables make use of them, and we will look at SZD fragmented logs and how LN SSTables make use of them.

### 5.4.1 Once Log and WALs

The WAL implementation is built on top of the concept of once logs as described in Chapter 4.3.2. The WAL itself only adds a bit of logic on top of the once log implemented in SZD. That is to say, while once logs are not restricted to the use of WALs, we will only describe how the once log functions in regard to the WAL.

Internally the once logs only maintain 1 SZD channel. This channel is in charge of all reads and resets. This is safe because these operations never happen concurrently. Once logs borrow an append channel from the WAL manager, this is done because only one WAL can be active for appends at the same time. It would be a waste of a QPair to allocate a QPair for appends for each possible WAL. Especially since the number of QPairs is limited. The once log internally has support for a queue up to the queue depth of the device. This queue is bound to the append channel. For each append, it determines if doing an append is safe (less than ZASL, not crossing borders). If it is, it verifies the number of outstanding requests. If this is more than or equal to what is allowed, it needs to wait. It repeatedly polls till one of its requests completes. The data from the finished requests is freed and the new request is scheduled. In the case an append is not saved, the suboptimal approach is taken of clearing the entire append queue. In this case, the implementation keeps polling till all requests are finished and then sends one request (the incompatible append) and sets the queue depth temporarily to 1. To ensure that I/O will always happen to the wp, the once log itself maintains a wp. After each (async) append it increments its wp, regardless of if it already happened. The associated zone of the virtual wp can then be sent to SPDK for the append. Internally it also maintains how much space is still left to append, which can be used by for example the WAL on top of the once log to determine if it is full and it is time to request a new WAL.

In case the WAL needs to be refreshed or the database closes, it is very important that all I/O requests finish. Therefore, we need an explicit sync call that polls till all requests are done. This delays database shutdown a little, but guarantees that all I/O to the WAL has been completed and is not just queued. Additionally, when a WAL is done, we finish the last zone of the WAL as we do not want to leak an active zone.

All WAL appends are added to a buffer that is a bit bigger than the original append. That is because an extra header needs to be prepended to the WAL. This header contains the following data in order: a sequence number (8 unsigned bytes) and the total size of the append in bytes (4 unsigned bytes). The total size is used to remove the padding that is added to fit WAL appends in the size constraints of pages on appends/reads. The key-value store does not need to know about the padding that flash internally requires. The sequence number is used for the recovery of unordered appends.

Recovering a once log itself is done by looping through all zones allocated to the WAL and verifying their zone heads. This is done from end to beginning to ensure that we always pick the last possible head (in case an external force resets a zone in the middle). The last zone that contains any data or the first zone, is set to be the head zone. The WAL recovery waits for the once log recovery to complete and then reads the data in steps of MDTS. It then does exactly the procedure as described in Chapter 4.3.2. It reads all data in order and splits the data in sequence numbers with a pointer and data to which the pointer points. The sequence numbers are then sorted. Then the sorted sequence numbers are looped through and their associated data is applied to the memtable.

### 5.4.2 Circular Logs

Two components in TropoDB make use of circular logs: metadata and L0. Both of these circular logs can share a common lower-level structure and add some minor logic on top of it. Therefore, a generic circular log design is used from SZD. This circular log structure comes with a few implementation details that we will shortly describe and then we will look at some logic that we added on top of the TropoDB structures.

**Generic circular log**:
The circular log design makes use of multiple SZD channels. One channel is used for appends, one for resets and multiple for reads. There is a separate channel for appends and resets because those operations can be issued by different components/threads. For example, for L0 the flush thread appends and the L0 compaction thread resets. It does not make sense to support more than one append and reset channel as the circular log design is bound by one tail and one head inherently. Multiple reader threads are supported because reads are unrestricted by the circular log design; any read between tail and head is valid, allowing multiple threads to complete concurrently. Each reader has a unique *number* that can only be used by one thread concurrently. This still guarantees that the channels are only used by one thread. It is up to the callee to ensure this rule is respected. Not putting explicit concurrency-safe mechanics in reads prevents unnecessary complications with locking when it is not required to do so.

Appends can be done either with ordinary heap/stack-allocated arrays or DMA-backed memory buffers (SZD buffers). The client never specifies an address, this is left entirely up to the log. It only appends. The client only specifies the data to write and the number of bytes to write. There is only support for synchronous I/O, which results that each append will only complete once the data is written in its entirety to storage. On a successful return, the LBA address of the first page is returned along with the number of pages that are appended. If the append crosses the last zone and needs to wrap around to the beginning of the log, the append is split in two, but the client will still only get the first

LBA and the number of pages appended. Additionally, if the write is not properly aligned (not a round number of pages), a bigger buffer is made that is aligned and the data is copied to this buffer. The client does not need to know about the internals of the log. After each append, the number of available pages is decremented and if upon an append the space available is less than the append size, it is immediately aborted beforehand. On a restart, the available space can easily be inferred from the zone heads in the log, just like the head and tail can. As a safeguard, the tail and head may never touch or else we can no longer distinguish the head from the tail (one zone is always lost because of this workaround, so a log requires at least three zones to function).

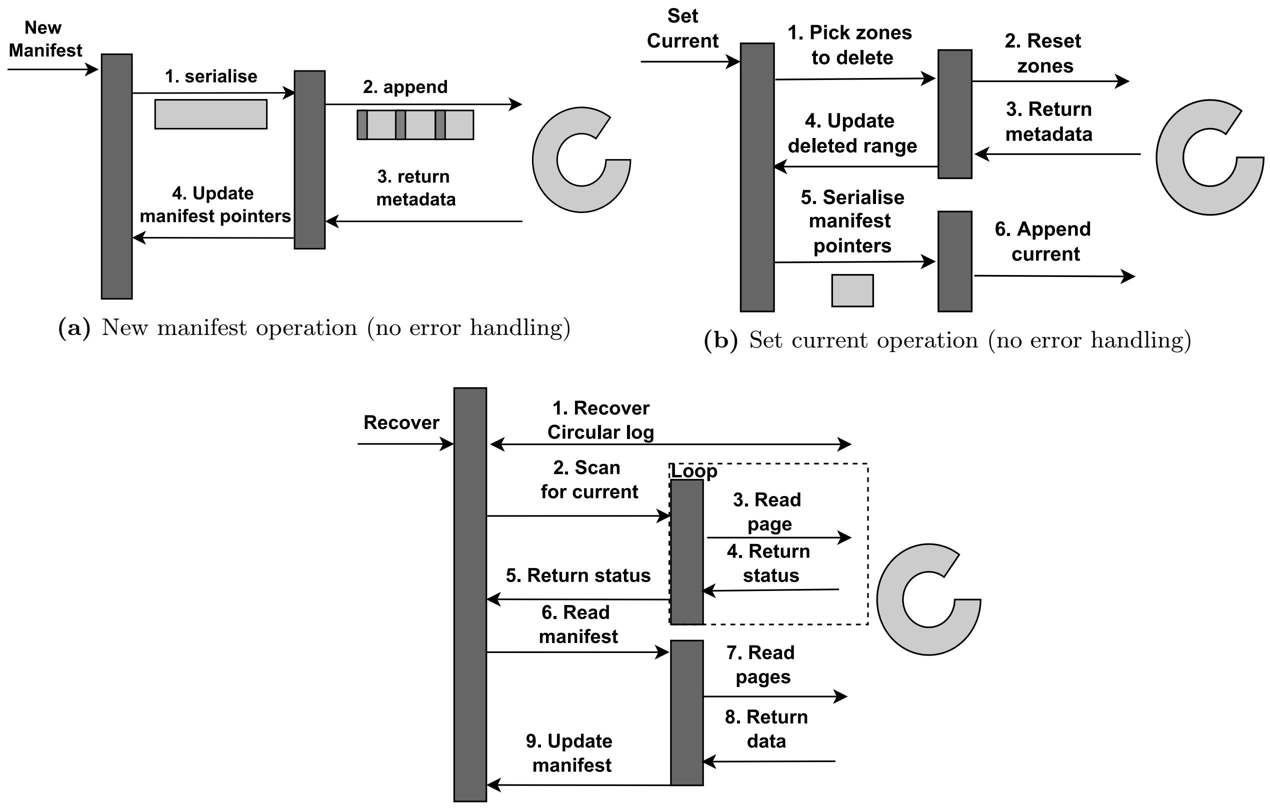
On resets, the client must specify the location and the number of pages. The location must always be equal to the LBA of the tail. This is a safeguard to prevent misuse of the circular log and is used for proper error handling. That is because if no address was given, the client would be unaware that incorrect data was deleted. The tail is incremented by the number of pages requested and deletes all zones it passes, again with wraparound. We refer to this operation as *consume tail.* One important detail and inconvenience is that if the number of pages to reset is not a round number of zone capacities, the delete is not persistent. That is because, on recovery, the tail is reset to the zone head. Resets of this kind only prevent reads occurring on these pages, but should not be used for persistent deletes, unless somewhere else the "unpersistent delete" is maintained for recovery. Alternatively, this could have been stored in some metadata that can be given to the circular log on recovery, but this is not tried.

Reads proceed in a few steps. They first check if the read is in between head and tail and fail (with wraparound) if it is not, preventing reads of unwritten data. If the check passed, the read is executed. The log uses the specified reader (from the reader number) to reads all pages either into either a specified heap/stack array or an SZD DMA-backed buffer (if future appends are expected this is more optimal). In case the read transcends the last zone, the read will wrap around to the beginning. Similarly, if the number of bytes requested is not a round number of pages, the last page will only be copied partially. The partial copy requires some further explanation. Ordinarily, the read can simply read to the client-provided buffer, but in case it is not aligned, a bigger buffer is needed or else we will have an overflow in the read operation (SPDK will read more bytes than fit in the client-provided buffer). TropoDB solves this by providing a small *spill buffer* for each reader channel of the circular log. This buffer is exactly the size of one page. In case the read is unaligned, the last page to read will be written to the spill buffer. Then only the bytes requested by the read operation will be copied from the spill buffer back to the main buffer. Using unaligned reads will thus always require exactly 1 read to the spill buffer and 1 additional copy to the main buffer.

On a recovery, we retrieve the zone heads of all zones present in the circular log. From this, we can infer the head and tail, provided there is at least one zone free between the two or all zones are empty. If all zones are empty, the head and the tail are equal and set to the beginning of the log. Otherwise, the head is equal to the last zone that is not completely full or the zone next to the last full zone. The tail is in this case set to the first zone that contains any data. *First* and *last* is in this explanation a bit misleading. As first can be later than last. It is based on a series of sequential zones that contain data, potentially with a wraparound. Before the tail, there should be an empty zone and after the head, there should be an empty zone.

**Circular log of metadata**:
The circular log of metadata hides away the circular log logic. It comes with a *new manifest* operation, *set current* and *recover* operation. The operations can also be seen in Figure 5.2, albeit in a simplified form. Note that the figure does not show error handling operations such and should only be used to get a general idea of the layers of abstraction. The new manifest operation does a number of things. First, it breaks the serialised index structure metadata into blocks in the size of pages. For each page it adds a small header, containing how much of the page is used, what fragment of the serialised data it is (header, center, trailer) and a CRC32 code. This is done entirely in memory. Then it is verified if there is enough space left in the log and if there is, the data is written. Internally in memory, it is maintained, where the last manifest begins (head before the append occurs) and how much data has up until that point in time been written to the log. Additionally, the metadata log keeps track of what data is old and needs to be deleted at some point, which is defined as a *deleted range* containing the circular log tail and the number of pages to delete. The *set current* operation deletes old data and tries to write new *current* data. First, it is checked what data can be safely deleted. This is done by rounding the deleted range down (floor) to a round number of zones, remember that only deleting entire zones in the circular log is persistent. This rounded number of zones is then reset. The deleted range is then set to a new pair. The start of this range is set to an LBA specifying the new tail of the log. The number of pages in the deleted range is set to the remainder of pages after deletion (the pages that could not be deleted safely) and the *last* number of manifest pages. The last number of manifest pages is added because we will change the last manifest to the new manifest. In the next iteration, we should be able to delete the last manifest. It is critical that we do not immediately consider the last manifest for deletion as this will lead to issues

when the process quits during the creation of a new current. This will lead to the situation where the old current is gone, new current has not been written yet and, therefore, all data is lost. Then after altering the deleted range, we write the actual current. The current data is always exactly one page containing the magic bytes *"CURRENT:"*, followed by some metadata. This metadata contains 8 bytes for the beginning of the new manifest data, 8 bytes for the number of pages of the new manifest data, 8 bytes for the beginning of the new deleted range and 8 bytes for the number of pages in the deleted range.



**(a)** New manifest operation (no error handling)

**(b)** Set current operation (no error handling)

**(c)** Recover operation (no error handling)

**Figure 5.2:** TropoDB implementation: Metadata log abstraction

On a recovery, we first recover the circular log itself. If this succeeds, we try to recover the current manifest by starting from head to tail. It is assumed that the current closest to the head contains the metadata of the most recent manifest. This is done, by moving page by page from head to tail. The current data is at most one page, so this is safe. For each of those pages, it verifies if it contains the magic bytes of the current and if the manifest pointers are in bounds of the log and valid. If they are, we can use this current. If not, we continue till the beginning. If no valid page is found, the database can not be recovered and we need to create a new one. However, if the data was valid, we use the current data to read the manifest. The manifest is read in one go from beginning to end. The CRC32 of each page in the manifest data is verified and the database is only recovered with success if they are all valid. At this point, the data is deserialised and set to the current index structure and the database is opened.

**Circular log of L0**:

The circular log of L0 adds minimal logic on top of the generic circular log. Most logic is related to the metadata, concurrent reads and resets. On an append, it first creates some metadata. It sets the LBA of the SSTable to the head of the circular log and requests the append to the circular log. The circular log then sets the number of pages written itself. This is the only change required. L0 adds support for concurrent reads with a *read queue*. This read queue requires one lock. This queue is as long as there are readers in the fragmented log. After locking the queue it picks the first reader that is not claimed and sets it to the claimed state. Then it unlocks, reads with the claimed reader, locks again, sets its reader back to unclaimed and sets a signal that a reader has become available (to avoid spinlocks). If no reader is available, it waits till one is available. This is done with a signal and not with a spinlock. The read data is verified by checking the header, which contains the number of bytes it contains and the number of entries. If this is valid, an iterator is created, which either returns a value from the iterator or itself (a merge needs the iterator for example).

Resets are more involved because there are no partial resets of zones. TropoDB only allows full resets, not partial resets. This is by design as deletes that are not persistent, lead to leaks and all sorts of problems on restart. On a reset, all deleted L0 SSTables are sorted on recency. Then only if the LBA of the first SSTable aligns with the tail address of the circular log, do we consider deletes. First, we create a *shadow tail*, which is used to get an idea of where the tail will end up. Then we walk through the rest of the deleted SSTables. In each step, we increment the shadow tail with the number of pages present in the deleted SSTable. We stop for the first entry where the shadow tail does not align with the LBA of the deleted SSTable (we can no longer safely delete). Then we divide the number of pages to delete by the zone capacity (floor) and only delete these zones. Then we create a *mock sstable* as described in the design. This SSTable has the number of the last deleted SSTable (guaranteeing recency order is respected), an LBA equal to the tail and a page count equal to the difference between the shadow tail and the tail. All non-deleted SSTables are readded to the deleted SSTable list. Now provided that we save the new metadata properly to the metadata log, we have persistent deletes.

### 5.4.3 Fragmented Logs of LN

Fragmented logs are implemented in SZD and are used in TropoDB by SSTables from L1 up to LN. Fragmented logs internally support multiple writers and readers. This is done by making use of multiple SZD append channels and SZD reader channels. Each channel can only be used by one thread concurrently, but this is left up to the client. Each append or read requires specifying a *reader number* or *writer number*. All operations are done synchronously to the fragmented log; they are only completed once the data is persisted to storage.

Appends can similarly to the earlier data structures (once logs and circular logs) be done with either stack/heap allocated memory or SZD-buffers. Then I/O proceeds in a number of operations. First, we check if the fragmented log is created with more than 1 writer. If it is, we need to lock, because we need to reserve zones from the zone allocator and zone allocators do not have support for concurrency. Zone allocators in turn do not support concurrency because this would cause issues with space reservation and internal schemes that might be used, say reserving zones sequentially. After locking, it is first verified if there even is space. This prevents the zone allocator from partially allocating and leaving a persistent leak (data is reserved, but there is an error halfway). Then zone regions are reserved and the lock is relieved. This is followed by appending to each zone in the reserved zone regions sequentially. Once a zone region is full, it continues onto the next zone region, splitting the data. After the appends are completed, it is checked if the last written zone is not full. If it is not, the zone is finished, preventing leaks of active zones. The zone regions along with the number of pages written are returned to the client.

Writers can also issue resets. Unlike circular logs, we do not create additional channels for resets. It is left up to the client to synchronise between writes and resets. Such an approach is possible because appends and resets are disjoint for fragmented logs. Appends do not rely on tails like for circular logs and resets do not care if data is written to zone regions or not. The fragmented log itself is unknowing of its own layout, apart from the free list, requiring minimal coordination. This is used to great effect by LN. The L0 compaction thread always uses the first writer channel and the LN compaction thread the second. Only the LN compaction thread resets and it uses the same channel for both its writes and resets.

When we implemented the fragmented logs, we also tried out concurrent appends. In this design, each writer has multiple channels. It only has an effect if an append needs more than one zone, which can happen for large SSTables for example. It uses a form of striping in this case and requires no changes to reads or resets at all. This is possible because we already know what data needs to be written to each of the zones as the sizes are known beforehand. Data for zone 1 in a zone region is still always written to zone 1 of the zone region and data for zone 2 in a zone region is still always written to zone 2. The operation then works in a number of steps. The number of needed zones is divided by the number of writers. Each writer then gets this amount of zones to write. Then I/O proceeds in a number of rounds. In a round, each writer writes up to ZASL asynchronously. Then after all writers have issued their I/O, the writers are polled. Once this is done, the next round starts. If a writer no longer has I/O to write, it simply stops. This allows doing multiple appends in parallel for large appends in LN. Nevertheless, it uses a large number of channels and QPairs of which there are limited amounts (SPDK sets a limit on the number of QPairs). The tests for performance showed a minimal gain and concurrency efforts were shifted to L0 and WAL later on. Future work could still look into this approach for fragmented logs and look into more optimal designs. It is suspected that more could have been gained by not waiting for all I/O operations to succeed after each append round.

Reads are more simplistic. Reads do not check if the zones it reads are full or empty, they just read the zone regions requested up to the specified number of bytes and do some minor bound checking. Similar to circular logs a *spill buffer* is used for unaligned reads. It issues one read to its read channel for each zone region specified. They rely on SZD channels supporting reads of arbitrary sizes.

Fragmented logs are the only structure in SZD that require metadata to function properly. This serialisation is left entirely up to the zone allocator used. The zone allocator itself should return data that is enough to guarantee persistence. The only zone allocator implemented, the merging allocator, needs a bit of metadata. For each entry in the free list it encodes the zone number (fixed 8 bytes), the number of zones (fixed 8 bytes) and 1 byte to indicate if it is in use or not (free or claimed). Additionally, it stores the last allocated zone, this allows continuing zone allocation from the exact position before database shutdown after recovery. Lastly, it maintains the number of bytes it has serialised to prevent reading null bytes on recovery.

LN SSTables add a little functionality on top of fragmented logs. Notably, SSTables only support a limited number of zone regions and save zone regions in a different format. In this format, zone regions are represented by the total number of used zone regions and for each individual zone region a number representing the starting LBA of the region and the number of pages used for the region (not in zones like fragmented logs expect). This is done because it allows using data similarly to L0. Reads and resets then translate it back to the original fragmented log format. Reads also add logic to support safe concurrent reads. Internally it contains a *read queue*. This is the same read queue as used for circular L0 logs. In case an iterator is requested for an SSTable, the entire table is still read in its entirety and the resulting buffer in memory is sent to the iterator.

## 5.5 Implementations of Data Structures for Memory

This thesis does not focus on in-memory structures. Nevertheless, an LSM-tree does require a few. In this section we will describe the in-memory structures that were used. In particular, we will look at the memtable implementation and a cache for SSTables.

### 5.5.1 Memtable

The memtable is a part of the key-value store that is not integral to this research. That is because it generally does not depend on the storage used. Therefore, we have decided to reuse an existing implementation with some alterations. TropoDB uses the default memtable implementation supported in RocksDB, an implementation that makes use of *skiplists* (Jellyfish by Yeon et al. has a good explanation of how skiplists are used in RocksDB (134)). Each change to a key-value pair is added to this skiplist along with a version number. This version number increments after each change and can guarantee that only the most recent change is used (in for example a merge later on).

### 5.5.2 SSTable Cache

As a performance optimisation, LSM-trees can implement a SSTable cache. An SSTable cache supports caching a few SSTables in memory, such as in 4GB of DRAM. This prevents needing to read SSTables from storage for each individual read. Instead on a read, it is first checked if the SSTable is already present in cache. If it is, it is possible to search for the requested key within the cache, reducing the need to do expensive I/O. If the SSTable was not present, the SSTable is first read from storage. Then the SSTable is added to the cache and only then is the requested key searched in the SSTable. This can amortise a large number of gets, provided at least a few reads are issued to the same SSTables, but can lead to a few more in-memory operations for each read. LevelDB and RocksDB take such an approach, but do not allow disabling the cache explicitly. This is not a ZNS-specific optimisation and, therefore, not a goal of this research. Nevertheless, a minimal implementation is used in TropoDB, using most of the implementation of LevelDB.

## 5.6 Background Operations

Most of the background operations are already described in the design and most of the other logic is not relevant for the storage research. We will, therefore, only shortly discuss the parts that are. Those are the policies to pick for flushes and compactions and the locking mechanics as these affect the underlying storage components.

Flushes are scheduled only when a memtable is big enough. However, it might be that there is not enough space left in L0. In this case, a compaction in L0 is scheduled (if there is not already) and the flush waits for the compaction to finish by waiting for a signal from the L0 compaction thread. This is also true for the concurrent design of WALs and L0. Once the circular log in L0 reaches a certain number of SSTables, client-issued puts are also slowed down with explicit delay operations to prevent flooding L0. Compactions in L0 are scheduled if the number of SSTables in L0 reaches a certain amount or the size left in any L0 log is less than a configurable threshold. The second constraint is unique to TropoDB as L0 can not physically become bigger than its assigned zone region and must be set to a safe value below 100% of the L0 log or else there might be out-of-space issues during a flush. Flush operations can be scheduled both by clients or flushes themselves (recursive scheduling). Compaction from L0 always picks the L0 SSTable present on the tail and picks a number of overlapping tables in L0. The maximum number of overlapping tables is limited to the maximum size of compactions, this prevents loading more SSTables in memory than fit. In case multiple L0 logs are used, the tail to pick is switched in round-robin fashion, but overlapping tables can be picked from both logs. All overlapping SSTables in L1 are picked and the size of L1 is not taken into account for the merge.

Compactions from L1 and up, make use of a score function. This score function divides for each level the number of bytes present in the level by the maximum allowed number of bytes in the level and multiplies it with a level modifier. The level with the highest score is considered for compaction, if the score is high enough. Each higher level has a bigger modifier, which allows prioritising compactions at lower levels. The SSTables considered for compaction in the level are done in round-robin fashion, which is made persistent by storing a *compaction pointer* along with the rest of the metadata similar to LevelDB (51). Compaction in these levels can also occur if the space left in the fragmented logs is too little. If this is the case and no level is considered for compaction, we have a lot of SSTables to delete, not enough space (or a persistent leak...). In this case, the compaction thread only tries to delete old data. Compactions on this level are scheduled by the L0 compaction thread or on startup of the database.

There still is one problem with compaction. Compaction from L0 to L1 and compaction from L1 to L2 might refer to the same SSTables. This is not safe as L0 to L1 invalidates data in L1 and L1 to L2 wants to move this invalidated data to the next level. Therefore, some locking is needed. Compactions from L0 to L1 create a global list of SSTables in L1 that are considered for compactions. Compactions from L1 to L2 do the same, but to a different global list. Before compacting, both of these threads first lock on a *L1 compaction mutex*. Then they verify if there are overlaps in their L1 list with the other list. If there is, they can not continue. In that case, they wait till the other operation is finished, then reconsider their compaction and repeat the procedure. However, this is naive and leads to *starvation* issues. It can lead to situations where either L0 to L1 gets a lot more compactions done than L1 to L2 or the other way around. Therefore, we need to add some *fairness*. This is done with a very simple algorithm. Whenever one of the compaction threads has to wait for their compaction to finish, they set a personal counter, which is a token that they should be preferred later on. Then when they are to create their L1 compaction list, they also verify the list if the counterpart *needs* a compaction and what SSTables they will use if they need a compaction (precaution). If this overlaps and their counterpart holds a token, they wait. Their counterpart has already waited once and will again if we continue. If not the operation in question can proceed and reset its token to 0, this operation does not need to be preferred next round. It already had its turn.

We also need to set some locks for client operations and background operations. Most locking can be prevented by separating the responsibility of tasks for each background thread. However, it can not be avoided with the current design. On a get, we need to do a global lock to ensure that we get a snapshot of the current database state and that it remains alive during the entire get. It is not acceptable that in a snapshot the tables in L1 are from a different version than L0. Therefore, we lock on a get and pin the current version of the WAls, memtables and the entire index structure. All are referenced, which in combination with using ref counting for memory management, guarantees that a snapshot is read. For a put we need to globally lock as well, to reference the current WAL, find out if there is enough space left in the memtable and the WAL and schedule background I/O if necessary. This is all similar to LevelDB. However, background operations require a very different approach.

There can be concurrent flushes, L0 compactions and LN compactions, but only one version head. Therefore, they work on different parts of the index structure, to guarantee consistent changes. At the end of their operation, they always lock globally, load the latest index structure, apply their changes to this index structure, write the new version to storage and only then unlock and complete. Flushes also lock globally temporarily to reset old WALs and dereference the current immutable memtable (so that it can be deleted when 0 refs are reached). L0 compaction threads do some more locking at the end for reset operations. The L0 compaction thread is the only thread that is allowed to add deleted SSTables for L0 and to remove deleted SSTables for L0. This change must always be applied to the most recent version, requiring locking. After finishing a compaction and adding the deleted SSTables during the compaction, it might be safe to also physically remove some SSTables. Therefore, after updating the current, it unlocks, removes all dereferenced deleted SSTables that it can, locks again globally and sets the deleted SSTables to the newly deleted SSTables for L0. This is safe as long as the process does not crash during the physical deletion. LN compaction is the only component in charge of physical deletion on L1 to LN, but L0 compaction can still add deletes to LN. This is still safe as long as the LN compaction thread locks around adding or removing dead SSTables from LN or else there will be persistent leaks (if LN deletes dead SSTables by accident by shadowing), which is also done in TropoDB.

## 5.7 Key-value Store Delete Operation

Some key-value stores support and explicit delete operation apart from put and get operations. Both LevelDB and RocksDB support this. Since TropoDB builds on top of the interface of RocksDB, it was decided to support deletes as well. This operation can be done in the form of *delete(keyname).* There exist multiple approaches to implement this method. One valid strategy is to store the key along with an empty value and treat it like an ordinary key-value pair. When on a get, the value retrieved is empty, it is assumed that the key-value pair has been deleted. An issue with this approach is that for some use cases an empty value can also be a valid value. For example, think about a simple use case where all middle names of clients are stored. Some clients might not have a middle name, and for clients of the key-value store it might make sense (not every client is the same) to store the middle name as an empty value in such cases. When empty values are treated as deletes, this will lead to unexpected behaviour.

Another approach is to make use of *tags.* This is done in LevelDB, RocksDB and TropoDB. We will explain the approach taken in TropoDB. In TropoDB each key-value pair update can be stored with the*insert* tag or the *delete* tag. Inserts are persisted to storage as the triple *(insert tag, key, value)* and deletes are stored as the tuple *(delete tag,* *key)*. On a get, the tag is first verified. If the tag is an insert, the data is assumed to be valid and the value is retrieved. If the tag is a delete, the data is assumed to be deleted and the client is notified that the key is not available.

An optimisation that is done in both LevelDB, RocksDB and TropoDB is to prevent writing deletes to higher LSM-tree levels needlessly. When a memtable is the only datastructure containing the deleted key, there is no need to write it to storage. We already know that all reads that request this key, will return that the key is not present. Similarly, if L0 is the highest LSM-tree level that contains the deleted key, there is no need to move the deleted key to L1. This is true for all levels. However, if a later level does contain the key, it is absolutely necessary to push the deleted key upwards and the delete can not be removed under any circumstances (yet). Reads always return on the first valid occurence of the key. Therefore, if a key is stored as a deleted pair in the memtable, it does not matter if the data is stored as a live key in, for example, L0. As long as there is always a dead entry of the key in between the top level and the live entry of the key. The dead key must slowly move down till it reaches the level that contains the key as a live entry. At this point the two will be merged and the key will be dropped. It is no longer necessary and can safely be deleted.

This optimisation can be implemented as follows. On each flush or compaction, all delete keys are fist checked before they are added to their respective SSTable. If there is no SSTable in any higher level that contains this key, the key is simply dropped and not added to the SSTable. If it is, it is added to the SSTable as a deleted entry. Additionally, if the next key returned during the merge returns the same key and this duplicate key is older, this key is dropped. The duplicate key refers to the key before it was deleted. This can occur since during a merge, multiple SSTables can contain the same key. Normally, we would simply pick the most recent key-value pair, but we need to treat deletes differently. Note that if the workload is delete-heavy and a SSTable only contains deletes, this can lead to flushes/compactions that write no SSTables at all, as all key-value pairs can be dropped.

## 5.8 Summary

In this chapter we have examined the implementation and evaluation tooling of TropoDB, answering RQ3. We have looked at the software stack used in TropoDB. This software stack exists out of SPDK, a C-interface on top of SPDK known as SZD and TropoDB itself built within the RocksDB ecosystem. SZD adds abstractions for ZNS storage and only support appends, not writes. It also comes with its own benchmarking tooling, which can aid in understanding the system. For example, by measuring the distribution of zone resets.

Then, we looked at implementations for the storage components designed in Chapter 4. We explained how WALs are implemented on top of SZD's once logs. In particular, we explained how once logs make effective use of queue pairs, perform asynchronous I/O, buffer appends, how padding is used, what metadata is stored in once logs and how to minimise I/O on recovery. We also looked at SZD circular log implementation and how both the circular log of L0 and the metadata log of the index structure are built on top of this abstraction. For the generic log, we describe how queue pairs are used for the circular log, how buffering is used and how resets are performed. For the metadata log we describe the abstraction we built on top of the log and how the design of the metadata log described earlier is realised. For the circular log in L0, we described concurrent reads and a more advanced reset procedure that can also do partial deletes (instead of deleting entire zones). We also take a look at the fragmented log implementation and how queue pairs and buffering are used in them. Additionally, we take a look at how to implement fragmented logs to allow for concurrent writers, readers and resetters.

We continued by looking at the implementations of in-memory components in TropoDB. The memtable of TropoDB is implemented with RocksDB's skiplist. We have also described how an SSTable cache is used to cache SSTables in memory to reduce the cost of client-issued reads.

We did not limit ourselves to just structures. We also took a look at the implementations for background operations. We described the flush and compaction process. We also showcased a contention issue in the communication between the L0 compaction thread and the LN compaction thread. To solve this issue, we came with a communication procedure, that reduces both starvation and fairness issues when both background operations contest for the same resources. We also described the locking implementation for both background operations and client operations.

Lastly, we took a look at the delete operation in TropoDB, which is implemented by tagging an update as a delete. Deletes are only flushed or compacted to the next level if necessary, else they are dropped.

# 6

# Experimental Evaluation

In this section, we answer RQ4: *"What is the impact of the optimisation process of TropoDB on the key-value store and ZNS device?"*. To answer this question, we will evaluate TropoDB as a whole against the state-of-the-practice and state-of the-art and we will evaluate various components of TropoDB itself as well. Additionaly, we will evaluate ZNS itself to get an idea of the achievable performance. However, before we discuss any evaluations, we will start with our evaluation plan, which specifies what we intent to evaluate, why and how. We will then continue with the evaluations themselves, one by one.

## 6.1 Evaluation plan

We intend to test various parts of TropoDB. We first need to get a good understanding of ZNS performance itself. We need to understand the performance characteristics of all I/O operations. Once we understand ZNS performance, we know what is maximally achievable. Then we should continue, by evaluating how the default implementation of TropoDB fares against the state-of-the-practice and the state-of-the-art. In particular, we want to measure the garbage collection effects as described in Chapter 3.4.3. We will then finish by looking at individual components of TropoDB. We will again test following what is specified in Chapter 3.4.3, but we will test TropoDB configurations against TropoDB configurations. To limit interaction effects, we will for each of these tests only alter *one* parameter at a time and run the exact same workload configuration (same random seed).

For each chosen evaluation, we will specify its purpose, how it is conducted and its configuration parameters. The following **evaluation plan** was generated from these **evaluation points (EV)**:

- **EV1**: *ZNS SPDK performance evaluation*:
  The purpose of this evaluation is to test raw performance of ZNS when SPDK is used. The intent of this research is to evaluate what the impact of the optimisation process of TropoDB is on both the key-value store and the ZNS device. To answer this, we need to understand what ZNS can deliver in the best case scenario, to allow for a proper evaluation. Therefore, all raw I/O operations will be evaluated within SPDK. This will be evaluated by measuring performance of (concurrent) write operations, (concurrent) append operations, (concurrent) random read operations, (concurrent) sequential read operations and reset operations. This evaluation will be conducted in Chapter 6.3

- **EV2**: *TropoDB WAL evaluation*:
  The WAL of TropoDB is designed to allow for high concurrency by using a novel approach with asynchronous append operations, *unordered appends*(Chapter 4.3.2). We need to understand the advantages that ZNS can give for this research, therefore, each novelty that makes use of ZNS, needs to be evaluation. To support the claim that unordered appends for WALs have an effect, we need to evaluate WALs. We are interested in the performance of unordered appends in a closed environment. This benchmark is independent of TropoDB as a whole. Therefore, the experiment will be run with background operations disabled (no interference) and is conducted in Chapter 6.4. Additionally, we discuss that unordered appends cause a recovery overhead in Chapter 4.3.2. This is evaluated as well in Chapter 6.4, as it determines whether or not the approach justifies the overhead.

- **EV3**: *TropoDB Garbage Collection Evaluation*:
  In order to evaluate the optimisation process in its entirety, we need evaluate TropoDB in its entirety against valid alternatives. While, many optimisations are limited to ZNS and not possible to measure against alternatives, we can measure garbage collection. To achieve this, we have defined how to measure garbage collection in Chapter 3.4.3. We will trigger a garbage collection-heavy scenario and measure write amplification, resets and tail latency. Alternatives are defined as in Chapter 3. TropoDB will be evaluated against the state-of-the-art, RocksDB + ZenFS, and the state-of-the-art, RocksDB + F2FS. This will allows us to determine if TropoDB has succeeded in reducing garbage collection effects (at least according to the alternatives). This will be conducted in Chapter 6.5

- **EV4**: *TropoDB L0 circular log evaluation*:
  All data structures within TropoDB have not been evaluated yet before this research, as they are novel. Therefore, we do not know the effects the components have on the key-value store and ZNS storage in practice. We could have picked any structure, but we have decided to evaluate the log size of L0, as L0 are the hottest next to WALs (which are already evaluated in EV2), and the size if L0 is its main configuration parameter. The evaluation is conducted by using the same setting as used for the garbage collection in EV3. Only the difference is that the circular log size is altered. It is conducted in Chapter 6.6

- **EV5**: *TropoDB effect of using multiple L0 circular logs and WALs concurrently*:
  In Chapter 4.7 we have described a novel alteration to TropoDB to increase concurrency. We need to understand if this is able to achieve gains in performance, as it has indications on the scalability of the design. If it does not, it indicates that TropoDB will not be able to use a ZNS SSD with high parallelism capabilities efficiently, and TropoDB will not reach the performance that some ZNS SSD can give. To conduct this evaluation, the same configuration as EV3 will be used, but the level of concurrency will be increased. Each level of concurrency that is added, will add an extra memtable, WAL, L0 log and flush thread. The evaluation will be conducted in Chapter 6.7.

## 6.2 Experiment Preparation

In this section, we will describe what hardware-, operating system- and software configurations are used for the experiments. We will also come with some small evaluations that validate the approach taken. For full details, including the specific software versions used we refer to the appendix in Chapter 8.

### 6.2.1 Hardware Configuration

All benchmarks for TropoDB that are reported in this report, made use of the same physical machine. This machine comes with:

- a 20-core 2.40GHz Intel(R) Xeon(R) Silver 4210R CPU with two sockets connected in NUMA mode. Each socket comes with a total of 10 physical cores and 2 threads for each core.

- 256GB of DDR4 DRAM

- 7TB ZNS SSD, Western Digital Ultrastar DC ZN540
- 280 GB Optane SSD, model INTEL SSDPE21D280GA
- 447 GB SATA SSD, model INTEL SSDSC2kB48

Nevertheless, all benchmarks are run on a virtual machine using only a part of the available device. The reason for not using the device as a whole is to limit usage to only one NUMA node and use a configuration that is more optimal/explainable for the benchmarks. The hardware configuration used by the VM is:

- a 10-core 2.40GHz Intel(R) Xeon(R) Silver 4210R CPU with one socket of 10 physical cores and 2 threads for each core. Using just one NUMA node.
- 128GB of DRAM, the amount accessible by one NUMA node.
- 7TB ZNS SSD, model WZS4C8T4TDSP303
- Various 8GB emulated generic ZNS SSDs (for testing)
- Various 1GB emulated generic ZNS SSDs (for testing)
- 160.8 GB Optane SSD, model INTEL SSDPE21D280GA (a partition of the device)
- 447 GB SATA SSD, model INTEL SSDSC2kB48

Some tests are done with a different machine that is very similar to this machine. This additional machine is used for investigations to get an idea of the general performance characteristics. It is for example used to determine the effect of NUMA on the database benchmarks. However, this node has never been used for the benchmarks reported later on in the report as those tests should all preferably be run on the same machine to discard hardware differences as an interaction effect. This additional machine has a few notable differences:

- a 20-core 2.40GHz Intel(R) Xeon(R) Silver 4210R CPU with two sockets connected in NUMA mode (at distance of 21). Each socket comes with a total of 10 physical cores and 1 threads for each core. So the number of threads per core is less in this node.
- there was no Optane disk with a 160.8 GB partition present on this node. This node could, therefore, not be used for F2FS benchmarks.

### 6.2.2 Virtual machine/Operating System Configuration

The server running on the physical machine was running an older Operating System configuration than was required for the experiments. This was Ubuntu 20.04.2 LTS with kernel version 5.12. The server could not be updated, so a VM was used instead. The VM used QEMU 6.1.0 with KVM enabled, running Ubuntu 20.04.4 LTS with kernel version 5.17.1. The VM was running its image from the SATA SSD. The 280GB Optane SSD was made accessible with paravirtualisation and the ZNS SSD was made accessible with PCIe passthrough within QEMU. The VM uses just 160.8GB of the Optane storage. The reason for this is that the VM is only allowed to use one partition that was configured on the host Linux machine, this also lead to the decision to use paravirtualisation and not passthrough. The VM was forced to just use one NUMA node with the help of numactl.

### 6.2.3 Verifying Configuration Sanity

Various configurations on the machine could have been used instead of the one we have specified. It is important that we run in a setting that is valid for our experiment, which means that the hardware configuration must be valid as well. A valid configuration is in our case one where all of the benchmarks must be run on hardware representative of real key-value store workloads and where we are sure I/O can be used to its fullest. Hardware must not be used in an inefficient setting, for example, by using different cores for interrupts and the key-value store. We need to be able to use I/O to its fullest as we are testing I/O for our experiments and must, therefore, not be bottlenecked by other parts of the configuration.

In order to verify such issues, we have run a set of small evaluations that led to this configuration. In particular, we have focused our efforts on a few different NUMA configurations. NUMA is a concern because it can be a problem if the devices, the interrupts of the devices and the key-value store are running on different NUMA nodes (7).

This experiment is conducted by running the VM with a few different NUMA configurations. This test uses a different machine than used for TropoDB benchmarks, but it is similar enough that the differences should be similar. This was the node with 1 thread per CPU core. The configurations tested are:

- Only one NUMA node is used and pinned by the VM. This includes the cores used by the NUMA node and its memory. This NUMA node is different from the NUMA node used by the storage devices.

- Only one NUMA node is pinned used by the VM. This includes the cores and its memory. This NUMA node is the same as the one used by the storage devices.
- All NUMA nodes are pinned and given to the VM. The VM itself does not do any pinning.
- No NUMA nodes are pinned implicitly by the host machine. Instead the VM pins NUMA nodes itself.
- No NUMA nodes are pinned by the host machine or the VM.

The experiment is run multiple times for each individual configuration. The benchmark tested the key-value store RocksDB with the help of its benchmarking tool *db_bench*. At the moment *ZenFS* is the state of the art for ZNS devices, so ZenFS is used as the representative file system. To ensure the test was configured properly, the benchmark configuration from the ZenFS repository itself is taken and altered to match the zone sizes of the device used. This configuration is (see Chapter 8 for more info):

- Use the filesystem ZenFS on the ZNS SSD
- A write buffersize of 2GB
- A target filesize equal to about 2 zones
- Use Direct I/O for flushes and compaction
- Use Direct I/O for reads

Various workloads are tested with this configuration. In all cases, 100 GBs of key-value pairs are written. This writes enough I/O to get a rough idea of the performance of the key-value store, but for testing key-value stores themselves more data will need to be written. It is thus picked to get an idea of the effect NUMA has on key-value stores. Half of the tests write sequential I/O with the db_bench workload *fillseq* and the others are done in random order with db_bench workload *fillrandom*. Testing different orders is important, as random order triggers more background operations and can therefore create more expensive I/O operations. If it is a problem that the NUMA node used by the VM and the SSD are different, we should expect it here as more I/O is required. For both the sequential and random benchmarks different value sizes are tested as well. The value sizes tested are 100 bytes, 200 bytes, 400 bytes, 1.000 bytes, 2.000 bytes and 8.000 bytes. This helps to differentiate between many small I/O operations and a few large I/O operations.

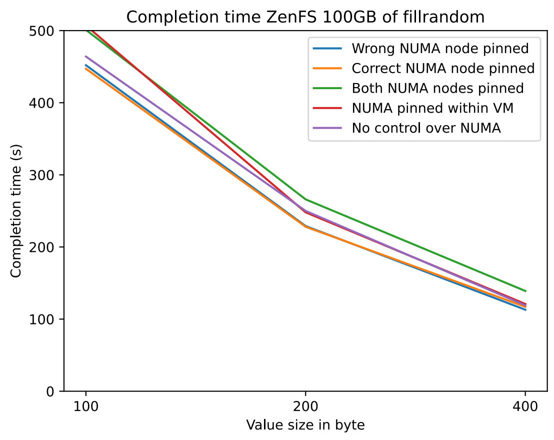
The value sizes picked are similar to value sizes we will test later on (see for example the WAL evaluations in Chapter 6.4.1 and Chapter 6.4.2). At the end of each run, the time till completion of the benchmark was recorded. Similarly, it was tested if QEMU only used the memory of the requested nodes with numastat. In the case of pinning one NUMA node, it could be observed that QEMU resorted to only making use of the pinned memory and CPU cores.

The results for all tests running sequential I/O showed minimal differences. During the fillrandom test, some differences are noted, but it can not be concluded that there is a significant difference. Many of the tests themselves already have some difference in completion time, regardless of the configuration used. For example, writing 100GB of 100-byte key-value pairs can take anywhere from 7 to 8 minutes. This is the same for all configurations. The few runs done to test if there is a difference, therefore, were not enough to find out if NUMA in fact has an effect. During the fillrandom benchmark, generally, a few configurations were slower than the rest. Not enough runs were run to test if this is always the case, but it is assumed that there is under ordinary circumstances a small difference. This can be seen in Figure 6.1. It shows the completion time of 1 run for each configuration with fillrandom. Note that both graphs have a different y-scale. For all configurations the completion time decreases, when the value size increases. Further on, for all value sizes, explicitly pinning all NUMA nodes has a negative effect on completion time. Similarly, not pinning a NUMA node or pinning a node within the VM is a bit worse than explicitly pinning a node. In the case of pinning one node, it did not matter what node was picked.

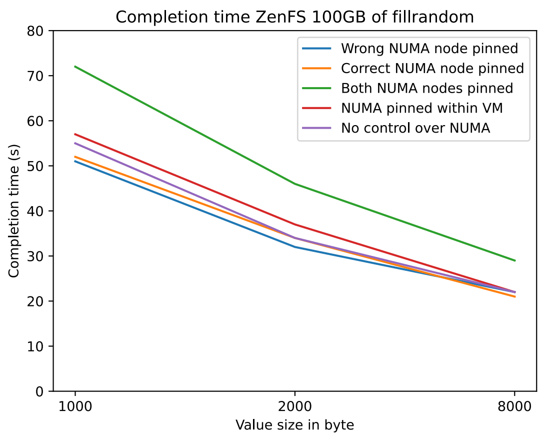
From these results, we can infer that the performance differences, in this case, are minimal, but that there is a small non-negligible difference. Since using just one NUMA node does not harm the performance and is easier to reason about for the other experiments, it was decided to use just one NUMA node for all of the experiments conducted, which lead to our final configuration. We did not run benchmarks for the rest of the components, as we consider them less of a problem. During no experiment did the memory reach above 128GB and no key-value store configuration that we used, used more than 20 threads implicitly.

## 6.3 ZNS SPDK Performance Evaluation

The basic operations of the ZNS device used in this research are benchmarked as the characteristics of these operations were not publicly available before this research was



**(a)**



**(b)**

**Figure 6.1:** NUMA configuration: Completion time (s) of fillrandom for RocksDB + ZenFS

conducted. Knowing the performance characteristics of each I/O operation allows for making better decisions on what operations are and are not expensive. It also allows making more sound decisions on what operations to use or avoid for the design of each individual database component. In this section, the results of those tests are given. All of these tests test I/O operations when SPDK is used and make use of synthetic workloads without background interference. The performance that can be achieved when SPDK and ZNS are used with applications or with multiple different operations in parallel will differ. These experiments are, therefore, only to be used to get an understanding of the ZNS device and to make a cost estimate. In all cases, the same hardware and software configuration as stated in Chapter 6.2 is used.

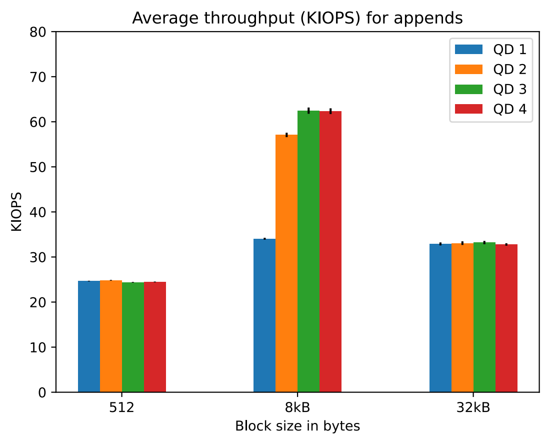
### 6.3.1 Performance of Appends and Writes

Key-value stores like TropoDB issue many writes. It is known that writes are more expensive than reads on flash-based SSDs(see Chapter 2), but to design the database around these properties, it is important to know how expensive writes exactly are. This makes it possible to understand how close TropoDB is to the maximum achievable capabilities of the device. In particular, the throughput and latency should be measured. However, measuring raw I/O calls will not be valid, as such calls will not be used. Instead, I/O will always be conducted through SPDK, so SPDK calls should be measured instead. SPDK comes with a plugin for fio, which allows measuring I/O performance for both latency and throughput. This plugin also supports ZNS. However, ZNS complicates the issue. That is because ZNS also comes with an append command, which is an alternative to writes. Unlike writes, appends support a higher queue depth than 1 and can, therefore, do concurrent writes to the same zone. The WAL implementation of TropoDB relies on using a higher queue depth to increase performance. So, we must also test the difference between writes and appends at various different queue depths. An alternative method to improve write performance would be to stripe writes/appends across multiple zones, therefore, the performance of concurrent writes to different zones is tested as well. Lastly, as the size of writes has a big impact on performance, various block sizes are to be tested as well.

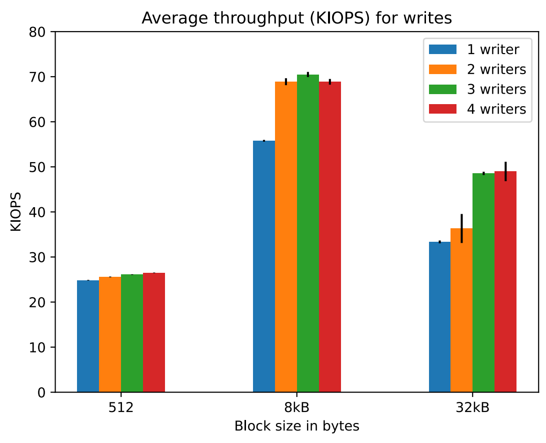
The resulting fio test consists of various smaller tests, each with a different write configuration and block size. At the beginning of each test, the entire ZNS device is always reset to a clean state to reduce the interference effects of resets during the tests. Following the reset, each test writes a sequential workload for 65 seconds up to a maximum of 32 zones, for which the results for the first 5 seconds are discarded (startup time). Since the zone capacity is about 1 GB, each test can write up to 32 GB at most. This is enough I/O to get a reasonable idea of the performance of ZNS with SPDK. Each test is tested in order with block sizes of: 512 bytes (minimum page size), 1KB, 2KB, 4KB, 8KB, 16KB and 32KB. The following tests are issued:

- Using appends with various queue depths. This is done with a queue depth of 1, 2, 3 and 4.
- using writes instead of appends. This test is multi-threaded with each thread writing to a separate subset of zones. This is done with 1,2,3 and 4 threads.

The results for throughput and latency are visible in Figure 6.2 and Figure 6.3 respectively. In both graphs, the x-axis represents the block sizes used for appends/writes in fio. For Figure 6.2 the y-axis represents the number of I/O operations conducted each second on average. For Figure 6.3 the y-axis represents the average latency of each individual operation in microseconds. The block sizes shown are 512 bytes, 8KB and 32KB, but earlier we mentioned that we also tested for different block sizes. These results are excluded because they largely correspond with the other results (1KB, 2KB and 4KB correspond more with 512 and 16KB more with 8KB) and can be viewed as raw data in the source code (see Chapter 8). In all cases, using writes instead of appends achieved better results. The throughput of writes is higher and latency is lower. Spreading writes across multiple zones also scales better, because unlike appends increasing the number of writes with a block size of 32KB and beyond still has a positive effect on throughput. For appends, the

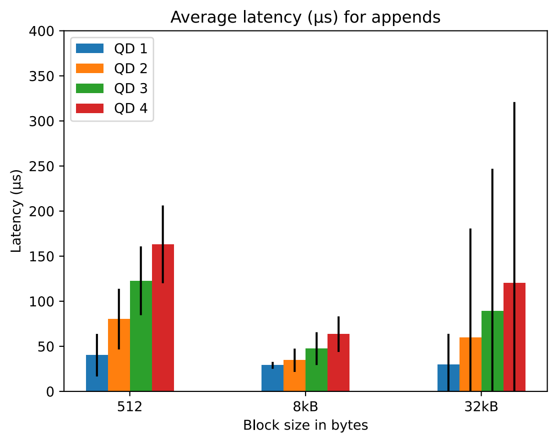


**(a)** Append operation

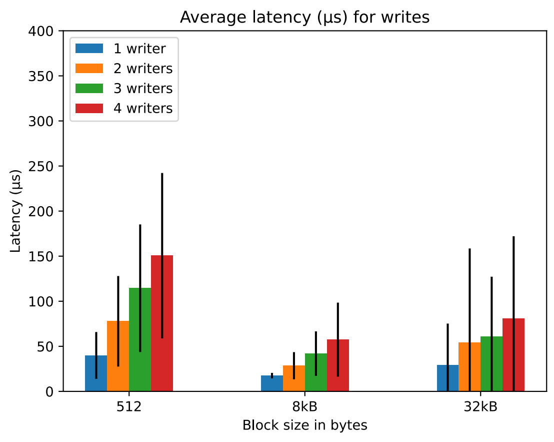


**(b)** Write operation, each writer writes to different zones

**Figure 6.2:** SPDK with fio: Write/append operation throughput (KIOPS)



**(a)** Append operation



**(b)** Write operation, each writer writes to different zone

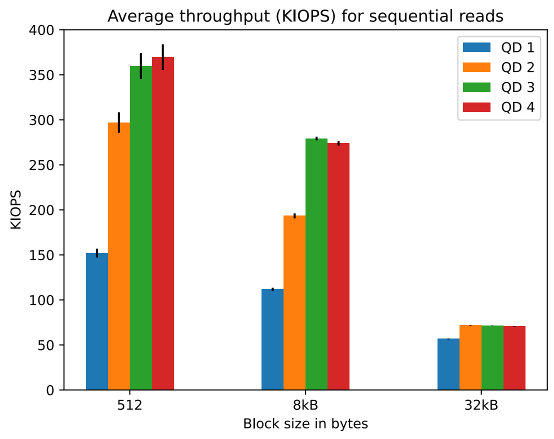
**Figure 6.3:** SPDK with fio: Write/append operation latency ($\mu$s)

latency only becomes higher at this point and using a higher queue depth is no longer beneficial for throughput. For the specific ZNS device used in the experiments, it would thus have been better to use writes instead of appends. It might, therefore, have been more beneficial to increase WAL parallelism with writes to multiple zones instead for this device. However, this would increase the number of active zones, which would leave less of the device to use for the rest of the database. We know that for some ZNS devices the performance of appends is shown to better than the performance of writes (23). Therefore, what optimisation works best, is dependent on the specific ZNS device used. How to decide when to use appends/writes or a ratio of appends and write is a topic left for future research. Nevertheless, the approach taken for TropoDB works, but will not reach significant performance benefits. Using appends instead of writes to reach better performance can not be justified for this device and no such claim about appends is, therefore, made for TropoDB as well. However, it is noticeable that increasing queue depth has an effect on latency and throughput for appends in some cases. Depending on the size, increasing queue depth can increase throughput, but it generally does not reduce throughput by much in other cases. Especially around 8KB an interesting pattern occurs. This pattern is also visible for writes. The exact reason for this peak is unknown. Nevertheless, using a larger queue depth with small I/O such as 512 bytes and large I/O such as 32KB has little effect. For large I/O the throughput of appends even approximates the throughput of writes. Therefore, we expect the WAL implementation to also scale as well when queue depth is increased, but only around a few block sizes such as at 8KB.

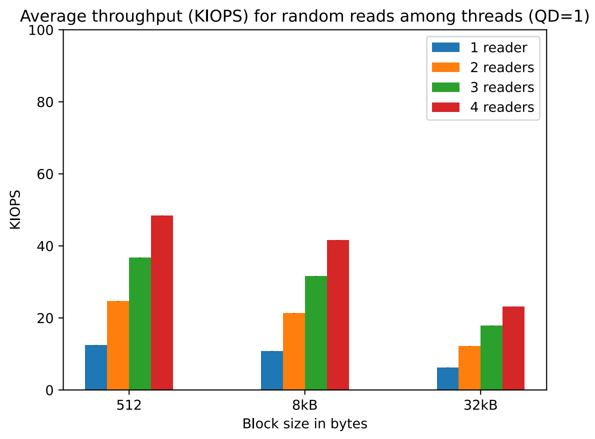
### 6.3.2 Read Performance

Read performance for flash-based SSDs is known to be better in both latency and throughput than writes and resets of flash-based SSDs, but still worse than reading from memory. Many operations in key-value stores make use of reads. For example, *get* operations, but also compactions. Compactions can take a long time, which makes it important to get an idea of how long it takes to read data. Therefore, read performance is tested as well. Similar to the append and write tests conducted, fio is used. Before reads can be used, the device has to be filled at least partially. Therefore, before the tests are run, a large sequential test for fio is run to fill up the entire device. Then we identify two tests to run on the filled device: sequential reads with a larger queue depth and concurrent random reads happening to different zones. This allows showcasing the effect of both improving the performance of one read by using multiple asynchronous reads and showcasing the effect of multiple reads happening concurrently. Especially concurrent reads are interesting as these happen frequently in a key-value store. For example, get operations reading from SSTables and compactions reading from SSTables, both stored in different zones. It is important to measure if there is a lot of interference. By using random reads, we are also able to see the difference between sequential and random reads, which is an additional benefit.

All read experiments are run similar to the append tests, but with a bigger maximum size as reads are faster. A maximum of 128 zones is read by each job. A startup time of 5 seconds is used, followed by 60 seconds of reading. Each test is tested in order with block sizes of: 512 bytes (minimum block size), 1KB, 2KB, 4KB, 8KB, 16KB and 32KB. Further on, sequential reads with queue depths are tested with queue depths of 1,2,3 and 4 and random reads that happen concurrently to different zones are tested with 1,2,3 and 4 readers.

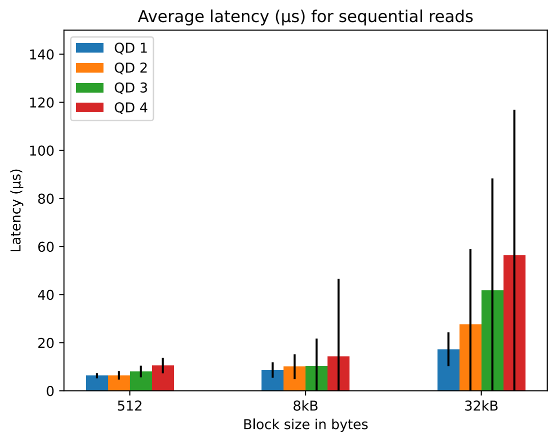


**(a)** Sequential I/O: One reader with different queue depths

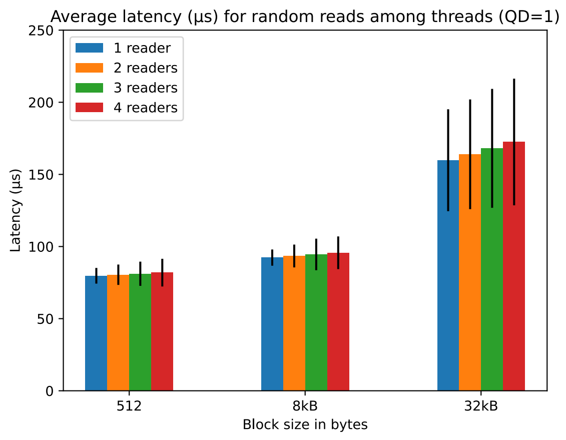


**(b)** Random I/O: Multiple readers each with a queue depth of one

**Figure 6.4:** SPDK with fio: Read operation throughput (KIOPS) for sequential and random I/O



**(a)** Sequential I/O: One reader with different queue depths



**(b)** Random I/O: Multiple readers each with a queue depth of one

**Figure 6.5:** SPDK with fio: Read operation latency ($\mu s$) for sequential and random I/O

Similar to appends, we only showcase results for 512 bytes, 8KB and 32KB. The other results are again very similar. The throughput in KIOPS of sequential and random I/O can be seen in Figure 6.4 and the latency of sequential and random I/O can be seen in Figure 6.5. In both graphs, the x-axis represents the block size of individual read operations. For Figure 6.4 the y-axis represents the number of reads conducted per second on average for Figure 6.5 the y-axis represents the average latency of each read operation. As can be expected of flash storage, reads perform significantly better than writes, both for throughput and latency. Another notable result is that random reads have significantly lower throughput and higher latency than sequential reads. This means that it is preferable to use sequential reads over random reads. Therefore, it is advantageous to store and load related data together, which is already done for TropoDB. It can also be seen that increasing queue depth, increases throughput significantly at the cost of latency. With a higher queue depth and larger block size, latency can become significantly bigger. It is thus a trade-off to pick between throughput and latency for reads. TropoDB only uses a queue depth of 1 for reads, but this graph showcases that more read performance could have been gained. Future work can look at the effect that increasing queue depth for reads can give for an LSM-tree design. What is also noticeable, is that the effect of increasing queue depth is most pronounced for small I/O and that small I/O, in general, is able to reach higher IOPS.

### 6.3.3 Zone Reset Performance

Garbage collection is generally in control of all erasures/zone resets. It is stated that these operations can take a long time. Properly designing a GC requires knowing exactly how expensive such an operation is. Therefore, the performance effects of resets need to be analysed. SPDK does not come with tests to measure zone reset performance by default and neither does fio. Since no public benchmarking tool for measuring resets with SPDK exists yet (as of August 2022), this requires an alternative approach. A custom test was made to test the effect of resets. This is a very simple test that test resets without the interference of other operations. The cost resets have on other concurrent I/O operations or that other concurrent I/O operations have on zone resets is not measured. The effect in a real workload might, therefore, be a bit different. This test is mainly used to show what the performance of resets is an ideal situation to allow making an estimate of the approximate cost of resets compared to other operations.

In the reset test, the entire ZNS device is first filled with fio with the workload “write". This is similar to how the read test requires a filled device. The data stored to storage with fio should be relatively random, which prevents issues that can occur when only zeros are written (optimisations in storage can happen in some devices). Future work can also look in the cost of resetting zones with zeros, partially-filled zones and empty zones. However, for now we limit ourselves to randomly-filled zones. After the device is filled, the device is reset zone by zone. This is done with our custom library, SZD, on top of SPDK, which also specifies a zone reset command. The time this reset call takes is measured in this experiment. This call contains some out-of-bounds checks, submission of the reset command and polling till the reset is finished. It is assumed that this accurately resembles a real reset.

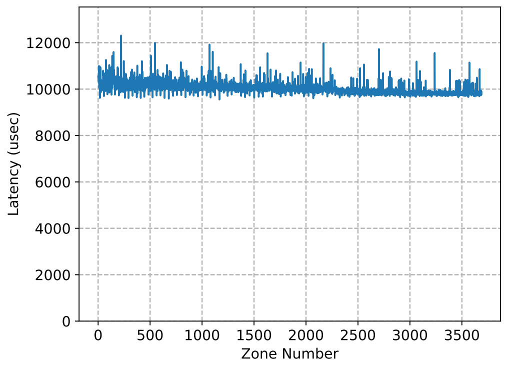


**Figure 6.6:** SPDK with custom test: Latency ($\mu$s) of zone resets when resetting zone by zone

The resulting time a reset took is 10037.95 microseconds on average or about 99.62 resets per second. The standard deviation of the zone reset time is 249 microseconds. Therefore, without additional interference of other I/O operations, reset operations are stable in their latency for this device. To understand if the cost of resets is also uniform across all zones, the reset time is plotted for each individual zone. This is visible in Figure 6.6 and showcases that reset time is also relatively stable across each zone. The x-axis represents each zone number of the device (in order) and the y-axis represents the latency of each reset operation conducted on this zone. As the result is just for one run, some spikes are to be expected. However, what is interesting is that towards the end, the cost stabilises more. It is not known if this is because of the zone number or because of another reason. This should be investigated further in future research.

### 6.3.4 Summary

In short, we have investigated the raw performance of SPDK operations for ZNS storage. We have investigated the latency and throughput of reads, writes, appends and resets.

Further on, reads have also been investigated both for sequential and random I/O. When we compare the latencies of write-, append-, read-, and append operations, resets take significantly longer. All I/O operations tested take a few hundred microseconds at most, but resets take multiple milliseconds in all cases. However, it has to be mentioned that because resets operate on entire zones and zones for this device are about 1GB in their capacity, the cost of resets is not trivial to properly compare to the cost of the other I/O operations which operate on smaller units, such a few KB or MB. In general, we see that the relative cost of I/O operations is in order of ascending cost in both latency and throughput: sequential reads, random reads, writes, appends and resets. Latency and throughput are also determined largely by the block size used by the application. The exact effect the block size has depends on the operation used.

## 6.4 TropoDB WAL Evaluation

In this section, all experiments related to the WAL are described and the results of those tests are given. The experiments focus on the performance impact that unordered appends have compared to ordered appends. This will be both for their throughput, latency and time to recover. In addition, various other attributes of the WAL would have been valuable to evaluate. For example, the impact of the WAL regulation scheme. Remember that the WAL manager should rotate the zones used for WALs. The exact effect of this circulation scheme has not been measured, the results for fluctuating the number of regions assigned for WALs can still be measured in future work. In Chapter 6.5.5 we do see the total distribution of zone resets for the entire key-value store and it is noticeable that the WAL resets are relatively spread out. It is assumed that when the total WAL region size decreases, more resets will be issued in a smaller region, but the load will still be spread evenly. Further on, investigating the effect of buffering on WALs would have been valuable. This can showcase if the effect of buffering negates the effect that appends can have and how much performance is gained by the use of buffering.

### 6.4.1 Performance Impact of Asynchronous Appends for the WAL

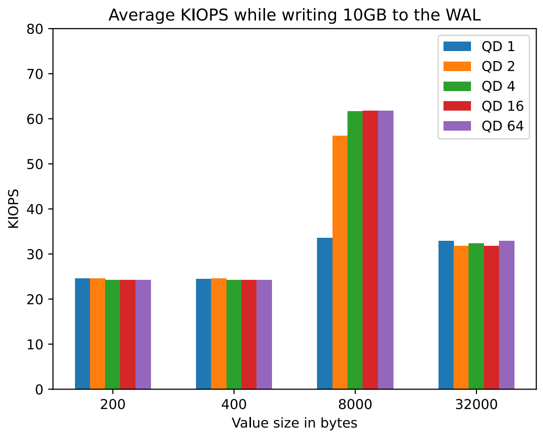
WALs are the hottest storage structure used by the LSM-tree. Every put request needs to be written to the WAL and the client has to wait for this request to proceed, even when this is done asynchronously. Therefore, it is beneficial to make sure that this can be done at high throughput and low latency. As discussed in Chapter 4.3.2, TropoDB allows asynchronous appends and makes use of the ability to send multiple appends to the same zone. This can theoretically increase the number of outstanding put requests, increasing throughput. Nevertheless, to test if this approach also has a real performance benefit, it needs to be evaluated. In Chapter 6.3.1 we have already seen that in many cases the performance effects that multiple outstanding append requests have is negligible. In fact, it is equally or more beneficial to stripe writes to multiple zones in parallel.

To test if it has the same effect on WALs, different levels of asynchronicity must be tested for the WAL as well. Therefore, WALs are tested with asynchronous appends with a queue depth of 1 (serialising) and WALs are tested with asynchronous appends with queue depths of 2,4,16 and 64. Asynchronous appends with a queue depth of more than 1 need extra metadata and recovery logic, this is not needed with a queue depth of 1. It would be unfair to still use this extra logic in this case. So, when a queue depth of 1 is used, this logic is disabled. Further on, in Chapter 6.3.1 it was evident that using different block sizes has a great effect on the performance of I/O. The same effect must be tested with WAL appends as well, especially around 8KB. To achieve this, the value size of puts is altered between tests. Key sizes are fixed to 16 and value sizes are tested with 200, 400, 2000, 8000, 16000 and 32000 bytes. The value sizes are not arbitrarily chosen, they are deliberately always set to be less than their target, with the target matching the block sizes from the append tests. That is because the exact sizes sent to storage differ a bit from $keysize + valuesize$, as some additional metadata is needed and the data needs to be aligned to the page level. For example, as the page size of the used ZNS SSD is 512 bytes, both 200 and 400 bytes will lead to similar outcomes (just with a different amount of padding). Testing both 200 and 400 will give more insight into the effect of padding. Further on, it is possible for multiple WAL writes to be merged together as one change (group logging). The value sizes are picked to approximate the values picked for the raw SPDK append tests in Figure 6.2 and Figure 6.3. To give an idea of what they approximate, 200 and 400 bytes approximate 512 bytes, 2.000 approximates 2KB, 8000 approximates 8KB, 16.000 approximates 16KB and 32.000 approximates 32KB. This should aid in determining whether the WAL can reach the level of performance that appends in SPDK can give. In later tests for TropoDB/RocksDB, we will also use similar value sizes. We want to test the database itself with value sizes that we know have been tested for WALs as well. These are tests for garbage collection of TropoDB in Chapter 6.5 and tests for individual components in Chapter 6.6.1 and Chapter 6.7.

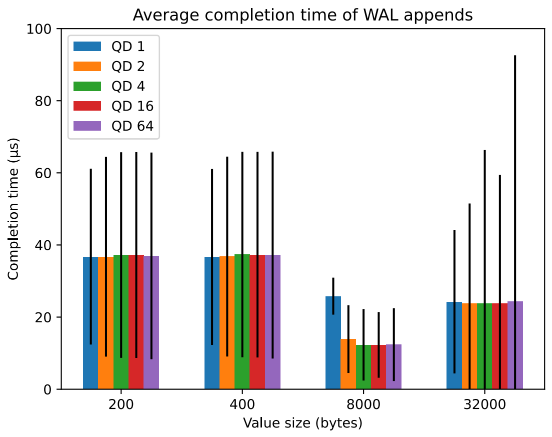
The zone capacity of the ZNS SSD that is used, is 1129316352 ($\sim$1.1GB) bytes. The WAL size in the test is configured to be 3 zones ($\sim$3.3GB) and the WAL region to 40 WALs ($\sim$132GB). To test the latency and throughput of WALs, it was decided to test a standard workload of about 50GB with this configuration. This load is generated with db_bench and the pattern *fillrandom*. The WAL has no knowledge of the pattern itself and its performance does not fluctuate depending on the load, therefore, any workload could have been chosen instead of fillrandom as well. This db_bench configuration should fill up a few WALs and erase a few WALs for each run. After each run, the database is reset in its entirety and the ZNS device is cleaned (full reset). This allows getting a clean slate for each test. Further on, background operations such as flushes and compactions can lead to significant latency fluctuations and are unpredictable. Additionally, the background operations can stall the client, allowing no more WAL operations to be issued. Such stalls can lead to asynchronous I/O having less of an effect. The purpose of this experiment is to test the WAL performance only. Therefore, such background operations are disabled entirely. In fact, only WAL resets can still be used in the background in this test. Puts are thus limited to memtable and WAL writes only. The benchmark issued is thus **not** representative of how TropoDB itself functions under normal conditions. Tests on how the database as a whole (WAL included) performs will be conducted later in Chapter 6.5. This test is merely to test what performance can be gained for WALs in the best case without the additional interaction effects.

To measure the WAL performance, we want to measure both the throughput latency of WAL appends. The average latency and its standard deviation are measured as the *completion time* of appends issued to the WAL. The completion time is defined as the interval between when appends are serialised and issued to the once log till the append itself has been processed by the WAL. An append is processed either when it is completed synchronously or when it is added to the asynchronous queue (when queue depth is more than 1). This metric is measured in this manner, to ensure that the measured latency only conveys the latency of WAL appends. The number of put operations per second is measured differently. This metric should give an overview of the number of client-issued puts that can be completed each second. This is measured by dividing the total amount of time all put requests took collectively (both WAL and memtable) by the number of put requests. So this metric does not exclude operations outside of the WAL.

Similar to the evaluation for appends, writes and reads we limit the results presented in the paper to 512 bytes, 8KB and 32KB for the figures. The results for the number of put operations completed per second can be seen in Figure 6.7 and the results for the completion times of append operations to the WAL can be seen in Figure 6.7 as well. For both graphs, the x-axis presents the value sizes of the key-value pairs (so the metadata and key size are excluded). For the throughput of WAL appends the y-axis represents



(a) Put operation throughput (KIOPS)



(b) WAL append operation completion time ($\mu$s))

**Figure 6.7:** WAL append performance: put operation throughput and completion time of WAL appends

the number of puts processed per second in KIOPS. For the latency graph, the y-axis represents the average completion time of puts. The results are very similar to the results for raw appends shown in Figure 6.2 and Figure 6.3. Increasing the queue depth to larger values has relatively little effect on the WAL as well. For small value sizes, the KIOPS decreases a little and once value sizes of 32 KB are reached, the KIOPS of all the different queue depths is similar. Increasing the queue depth does have a big effect for a value size of 8KB, similar to appends. In this case, setting the queue depth higher than two can achieve almost twice the amount of KIOPS. It shows that increasing queue depth can have an effect, but is highly situational. It also shows that WALs are able to match the performance of appends in SPDK closely for throughput. Latency results are also very similar, which makes sense as latency and throughput have an inverse relation. Completions take a little longer on average for small and large values when a higher queue depth is used. Similar to how KIOPS were higher for 8KB, so is latency significantly lower for 8KB. However, increasing the queue depth creates a larger standard deviation in all cases. This is especially noticeable for 32KB writes and a queue depth of 64. This implies that some clients have to wait significantly longer for their put operation to complete, probably because a large queue of I/O needs to be processed by the SSD. Thus, increasing queue depth, can also lead to more latency instability. Even in cases where increasing the queue depth has little effect on the throughput.

In short, using unordered appends for the WAL has an effect on throughput and latency, but it is highly dependent on the value size used. In most cases, the effect on KIOPS with unordered appends is negligible and in fact worse when a higher queue depth is used. Only for a few value sizes will the amount of KIOPS increase. However, increasing the queue depth does have a big effect on the latency stability. Decreasing the queue depth for WALs achieves better latency stability. Therefore, it is recommendable to investigate the characteristics of SSDs (73) and the workload used for the key-value store before deciding to increase the queue depth. For example, TropoDB has made WAL queue depth *configurable* and set it to 1 by default.

### 6.4.2 WAL Recovery Time

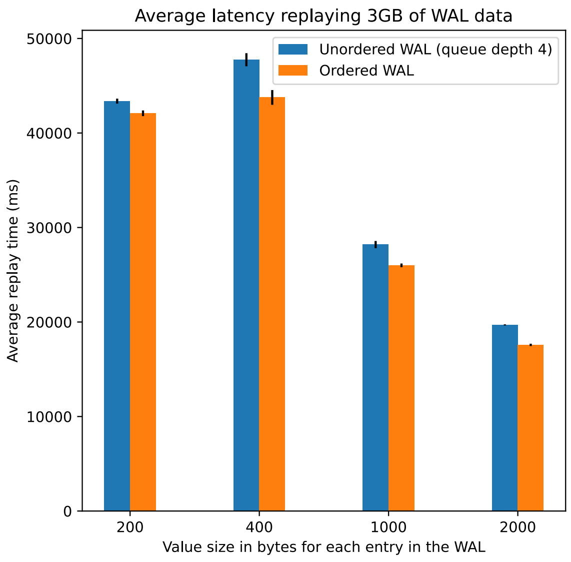
As discussed in Chapter 4.3.2, TropoDB supports both ordered appends and unordered appends to the WAL. Unordered appends can gain some performance speeds for puts, but they can also take longer to replay. It is important to measure the exact overhead that occurs when restoring the database, as this can greatly impact the startup time of the key-value store. Therefore, the time difference it took to replay WALs is measured. To properly measure the difference, the replay time should match the replay time of a real workload, but should still properly reflect the replays that take longer as well. TropoDB by default only has one WAL active at the same time, which requires one full replay at most. To ensure that no WAL is removed in the background, WAL erasure is temporarily disabled for the WAL recovery test. Therefore, it is best to fill up exactly one WAL with ordinary data. Apart from the size of the key-value pairs, it does not matter what the individual key-value pairs are for the WAL. That is because WALs have no knowledge about this data. Nevertheless, it might create some fluctuations when applied to the memtable. Therefore, a few different workloads should be used and both ordered- and unordered appends will use the same workloads.

To test the effect, db_bench is used. Db_bench allows generating a proper workload that matches more realistic access patterns. This is done by first generating a workload with random key-value pairs, using *fillrandom*, which is a bit larger than one WAL. This is then issued to the key-value store. Once, this is completed the database is reopened and the recovery time of WALs is measured. This is done by only measuring the time gap that is present between when a WAL is opened and when it is recovered. The effect should be measured between ordered and unordered appends. Therefore, each test is run with the exact same seed and configuration for both. Further on, they are both tested with 5 different seeds. This allows measuring the possible fluctuations in recovery. The number of entries in a WAL can have a big effect. A larger number of entries can result in more entries that need to be sorted. Therefore, a few different value sizes for key-value pairs are tried, to fluctuate the number of entries present in the WAL. In the end, the following fixed configuration was used:

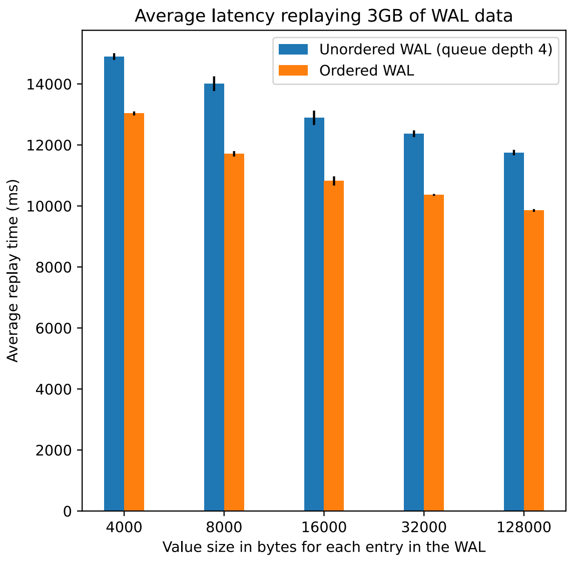
- WAL size of 3 zones, with each zone thus having a capacity of 1129316352 bytes (∼1.1GB), for a total size of 3387949056 (∼3.3GB) for each WAL.
- WAL region of 40 WALs (never requiring resets of old WALs)
- DB_bench workload *fillrandom* with a fixed workload size, matching the WAL size, of 3387949056 (∼3.3GB).
- Each tested is repeated 5 times, each time with a different fixed seed
- A key size of 16 bytes
- WAL erasure is disabled

In addition, each configuration is repeated with different value sizes. In particular, value sizes of 200, 400, 2000, 4000, 8000, 16000, 32000 and 124000 are tested. These are similar values as tested for appends, with a few additional larger values. Then the two different WAL designs: TropoDB with ordered appends and TropoDB with unordered appends, are tested with the exact same tests. TropoDB with unordered appends uses a queue depth of 4 as this queue depth has had reasonable results in Chapter 6.4.1.

The final results can be seen in Figure 6.8. The x-axis represents the value sizes used in the stored key-value pairs and the y-axis represents the average completion time of a replay of a WAL. Notice the difference in the y-scale between Figure 6.8(a) and Figure 6.8(b). The completion time decrements as the value size increases, except for the gap between a value size of 200 and 400. As can be seen in the figures, replaying unordered WALs always takes longer than ordered WALs. The exact difference between ordered appends and unordered appends is different for each value size, but fluctuates between ∼5% and ∼15%. Further on, the difference never reaches more than a 20% overhead and replaying unordered writes does not take more than 2 seconds longer than ordered appends in any test conducted. This difference in recovery latency is expected as ordered appends do not require sorting operations on replay. However, unordered appends do require sorting. Considering that the replay is only done on startup of the database and that the database itself can properly run undisturbed for many hours, in the general case such an increase in startup time is negligible. In the case databases do have a short lifetime and are restarted



(a)



(b)

**Figure 6.8:** WAL replay time: Replay time ($\mu$s) for both ordered and unordered WALs

frequently, it might be better to leave this option turned off. The difference between replay times of ordered and unordered appends, is also bigger for larger entry sizes, at least in the results shown here. Larger value sizes can thus become more expensive for replays. This should be taken into account when deciding whether to use unordered appends or not. The results also showcase the effect that the size of entries has on replay latency. When the size of an entry is smaller, the WAL can contain more of such entries. This will result in more operations needed on a replay regardless of whether ordered or unordered appends are used. More operations are needed because more entries need to be deserialised, verified and added to the memtable. In the case of unordered appends, there also is a need to sort more key-value pairs, which adds an extra overhead. Interestingly, a value size of 400 is more expensive than 200. This is because an equal amount of key-value pairs with a value size of 400 and 200 fit in a WAL, as the minimum page size is 512 bytes. Yet, more data will need to be read from each page. Replays will thus be most expensive for value sizes just a bit below the minimum page size. Therefore, it can not be concluded that a smaller value size will always result in more expensive replays for a WAL, but in the general case, it does.

To conclude, using unordered appends does come with a trade-off. When unordered appends are used, the key-value store will take longer to restart. Whether, the performance benefits of unordered appends are worth it, depends on the use case. In Chapter 6.4.1, we have already seen that unordered appends for the SSD used, provide little performance benefit. Therefore, TropoDB has made this option *configurable* and turned off by default.

## 6.5 TropoDB Garbage Collection Evaluation

In the previous sections we have looked at individual components of TropoDB, but not at TropoDB as a whole. In this section, we will take a look at TropoDB as a whole. We will investigate how TropoDB performs compared to the alternatives. The alternatives are in this case: the state of the practice, RocksDB + F2FS, and the state of the art, RockDB + ZenFS. In particular, we will look at the effects of garbage collection. That is to say the tail latency, the average bytes written for each key-value pair and average number of zones reset for each key-value pair. For all three effects, the exact same benchmark will be run, seed included. First, we will describe how the experiment is set up and why it is set up in this manner. The setup is a bit more complicated as each target is defined differently. Then we will look at the individual results of these benchmarks.

### 6.5.1 Benchmark Setup

The intent of this test is to measure the effects of garbage collection. Therefore, garbage collection must be forced on all candidates and the workload must be garbage collection heavy. This requires triggering garbage collection as frequently as possible, to see and measure its effects. Some systems might do garbage collection lazily and postpone it until they no longer can. Garbage collection will also not happen with small workloads in general, as the number of flushes and compactions will be limited. Further on, when no data overlaps, garbage collection will also be limited as the number of merges needed for compactions is minimal. In order to force garbage collection, filling the device is beneficial. The SSD then reaches a state that is comparable to a *steady state*(for more on the steady state see a study of steady state by Didona et al. (39)). When more data is written than fits on the device, at least a few resets must be forced. When there is little data, this is not needed. The steady state is in this case a state that requires garbage collection. Further on, it is a good idea to use a workload that overwrites data. Relying on overwrites ensures that data will overlap and compactions will be both big and necessitate deleting old SSTables. Garbage collection will be most pronounced in these cases. It is also important to measure the effects garbage collection has on client operations, such as gets and puts.

ZenFS comes with a few benchmark configurations of its own. One of those fits the exact workload we want to test. This benchmark configuration makes use of db_bench and uses three workloads in order. First, it fills the SSD with random data, then it overwrites the data and then it issues many writes and reads concurrently. These are the db_bench workloads *fillrandom*, *overwrite* and *readwhilewriting*. The first test brings the device to the steady state, the second can be used to measure garbage collection effects and the last test showcases the effect of puts and garbage collection on the performance of gets. This exact test is, therefore, also used in the experiment to measure garbage collection effects.

The ZNS SSD used has a little more than 3TB of storage that is accessible to the user of the SSD. Therefore the following test configurations will be sufficient (note that 1TB of key-value pairs will write a lot more than 1TB of data, otherwise WA would have been 1 and there would not have been an issue):

- fillrandom, 1TB of data by writing 1.000.000.000 entries with key size 16 bytes and value size 1000 bytes.
- overwrite, 1TB of data by writing 1.000.000.000 entries with key size 16 bytes and value size 1000 bytes.
- readwhilewriting, 1 hour of additional writes during reads. Again the key size is 16 bytes and value size is 1000 bytes. Additionally 32 client threads are used (this is more than the number of physical cores available). As it stops after 1 hour, the amount of data written differs between and this test is not suitable to measure either bytes written or zones reset, even when the metrics are normalised over the number of key-value pairs written.

Additionally for all tests, a few additional configurations are used. These are again based on ZenFS, but considered sensible for the characteristics of the device and the file systems:

- compression_type set to none. We do not want to measure compression effects.
- use_direct_io_for_flush_and_compaction. We want to measure storage effects, not buffering effects. The alternative is also not supported in TropoDB. We only have direct I/O for flushes and compactions.
- use_direct_reads. Again raw reads should be used to measure storage effects.
- max_bytes_for_level_multiplier is set to 4. Each level is 4 times as big as the previous level.

- target_file_size_base is set to the capacity of about 2 zones. This is the target size of SSTables among others. This is similar to the size of SSTables as used for LN in TropoDB.
- num_levels is set to 7 for F2FS and ZenFS and 6 for TropoDB. TropoDB uses 6 as it needs to reserve space beforehand, which causes issues if a level is too big and the 7th level was never reached in the first place.
- max_background_jobs is set to 8 for F2FS and ZenFS. This allows at least a thread for a compaction in each level and a flush. TropoDB uses its custom design with a flush thread and 2 compaction threads.
- write_buffer_size is set to 2GB. This corresponds with the max memtable size and fits properly for the size of WALs in TropoDB as well.
- sync is disabled. We will explain why soon.

Note that sync is disabled. Disabling sync, allows the tests to complete faster and is the default option for RocksDB. It will also lead to a lower average latency (not necessarily tail latency). This comes at the cost of some reliability. Ideally, this should be tested as well, this allows testing latency as well in the case the key-value store itself should provide all reliability. If during later research, sync is disabled, buffering of WAL writes should be disabled as well. The buffer size of the key-value store on top of the file system can in this case be set to 0. In both ZenFS and F2FS as well (Posix file system as defined in RocksDB).

Additionally, we have some configurations for each of the three targets. For F2FS tests, the device needs to start with a clean slate. Therefore, before the fillrandom test, the ZNS drive and Optane drive (see why Optane is used as well in Chapter 6.2) are reset/formatted, the file system is made and the file system is mounted. This allows F2FS to be compared against the alternatives more fairly.

The same is done for ZenFS. Before the fillrandom test is started, the ZNS drive is reset and the file system is made on top of this drive. The auxiliary path, which ZenFS uses for logging, is set to the tmp directory in Linux of the VM. This directory will also be emptied before the fillrandom test starts. It is said to contain a minimal amount of information, such as LOCK files and debugging information. The exact amount of data written to this directory has not been measured and is left to future research.

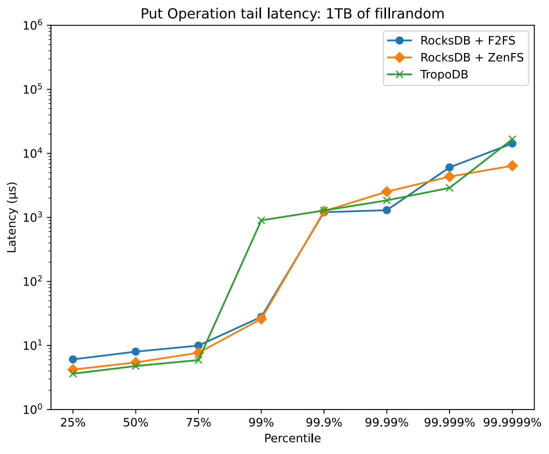
TropoDB has some additional configurations. Most options are based on earlier TropoDB investigations:

- Number of Manifest zones is set to 4 zones.
- WAL size is set to 4 zones.
- WAL count is set to 40 WALs.
- WAL queue depth is set to 4.
- Maximum size of L0 SSTable: set to exactly 512MB.
- Number of SSTables on L0 before client puts are slowed down: 80.
- Number of reserved zones for L0 is set to 100.
- Amount of deferred writes during compaction is set to 6 and amount of prefetched reads as well.
- Concurrency level of L0 and WALs (concurrent L0) is set to 1.
- LN zone allocator is set to the merging allocator.
- Compaction triggers (before a compaction is considered): 8 L0 SSTables (4GB), 16GB, 64GB, 256GB, 1TB, 4TB (last level can not compact anyway).

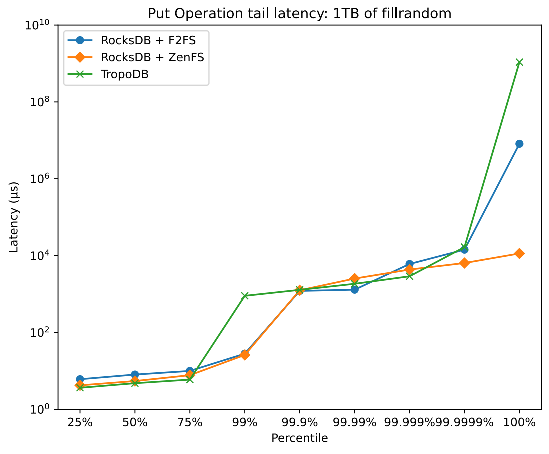
### 6.5.2 TropoDB Tail Latency

Tail latency can be measured with the help of db_bench. Db_bench measures the tail latency up to a certain degree and allows logging it at the end of a run. This can be done for F2FS, ZenFS and TropoDB all the same. By default, db_bench does not log 25%, 99.999%, 99.9999% and 100% for tail latency, so this functionality is added. This gives a more accurate view as it does not hide away the cheapest/most expensive operations. After each run, the tail latencies are logged.

The results of the tests can be seen in Figure 6.9 for fillrandom and Figure 6.10 for filloverwrite. The tail latency graphs are cumulative: the x-axis represents the percentile of operations that are completed with a completion time less than the latency specified in the y-axis. Note that the y-axis represents the latency of put operations on a logarithmic scale. For each workload, we showcase both tail latency up to 99.9999% and up to 100% as the max is also essential to discuss, but makes the tail latency graph harder to read. Additionally, for RocksDB + F2FS and RocksDB + ZenFS there are also results for read-whilewriting, which can be seen in Figure 6.11. Unfortunately, TropoDB crashes on this test and the issue has not been found (as of August 2022). It likely is a concurrency bug in

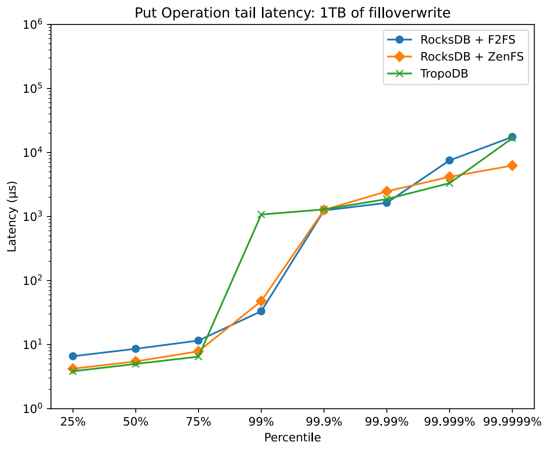


**(a)**

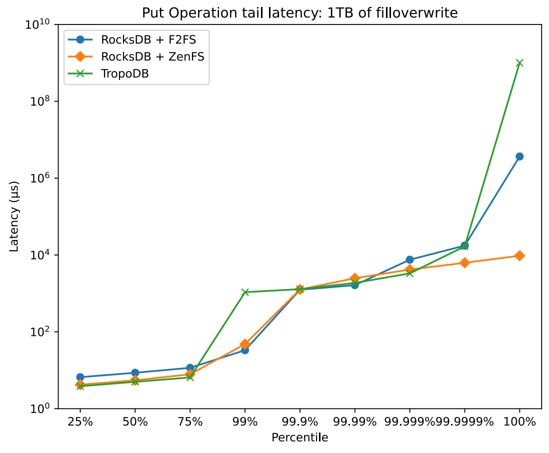


**(b)**

**Figure 6.9:** Put operation tail latency: Workload fillrandom



**(a)**



**(b)**

**Figure 6.10:** Put operation tail latency: Workload filloverwrite

the read path of get requests, which is more likely to occur as 32 reader threads are used. There are quite a few interesting details that can be inferred from the latency graphs.

For fillrandom RocksDB + ZenFS is able to remain relatively stable for all percentiles. For some percentiles such as 99.99%, it is a bit slower than the alternatives, but it does not suffer from latency spikes at any point, even at 100% there is no peak. In general, ZenFS is able to guarantee better SLAs for the workload provided. F2FS seems more stable when

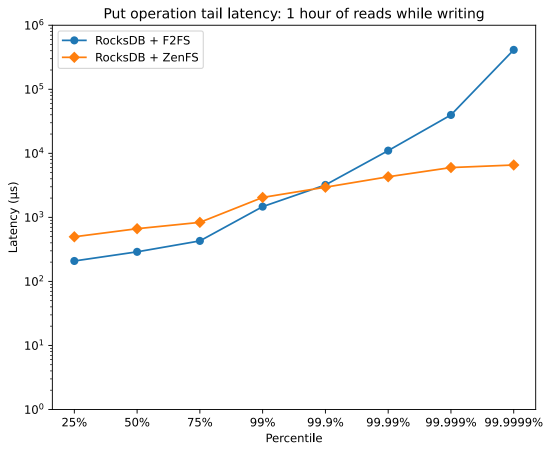


**Figure 6.11:** Put operation tail latency: Workload readwhilewriting.

only looking at 25% up to 99.99%, but it suffers from issues shortly after. A few clients have to wait significantly longer for their put requests to finish. TropoDB is the worst in this regard. It has the lowest latency for 25%, 50% and 75%, but starts to spike at 99%. Then it smoothens out a bit, but has an unacceptable spike at 100%. Even if it is the only client-issued put that resulted in this latency peak and may, therefore, be an outlier, it can not be ignored.

Some client-issued puts in TropoDB will take minutes. Looking at the debug log of TropoDB, it is noticeable that these situations happen once in a while. They occur when L0 is filled up, but not only because of the number of live tables, but also of dead tables. Remember that the circular log of L0 can only remove entries from its tail, but it can already have gaps later on. During fillrandom it is common that at a certain point in time more than half of the log will contain dead SSTable and is considered full. At this point, L0 has to be compacted as there is no space left and no flushes can continue. As no flushes can continue, clients have to be stalled. There is no guarantee that a compaction compacts enough tables for a tail zone to be erased, which means that it can take multiple compactions for the client to continue.

Additionally, there are situations in which an L0 compaction has to wait for an L1 compaction to finish. This happens when they operate on the same SSTables, this is a lock contention issue. As stated in Figure 4.6.3, put operations can be delayed when L0 reaches a certain size. This happens very frequently because of similar issues and can explain the spike at 99%. Later on, we will see that this is not always the case for TropoDB. Lastly, TropoDB's background operations were initially based on LevelDB, which is known to scale less than RocksDB because of its background thread model. The background process used might, therefore, also be part of the blame. It would be a good research opportunity to try out the same approach with RocksDBs background processes.

In the end, this design as it is now, regardless of the reason, does not scale as well as should be possible with the hardware. The result is that TropoDB can not state that it is able to get more stable latencies because of garbage collection effects than RocksDB as TropoDB regularly is involved in long forced background operations that force stalls. Background operations in TropoDB can take from a few seconds to more than 10 minutes in the worst case.

The latencies of the filloverwrite test, differ less from the latencies of fillrandom than was initially suspected. The biggest difference is notable for RocksDB + ZenFS. ZenFS is unlike in fillrandom involved in at least one very expensive put operation, which was not the case for fillrandom. However, it is still smooth up until 99.9999%, which makes it questionable whether it was an edge case or not. The results for TropoDB do not differ significantly. TropoDB is still not able to guarantee stable performance and still has the issue from 99.9999% up to 100% and so does F2FS.

The results from the readwhilewriting test are insightful for the performance of ZenFS and F2FS. F2FS has severe problems with tail latency during this test. F2FS is not able to guarantee the same latency stability that it was able to maintain during the previous two tests. F2FS has to stall many read operations because of other reads, writes and background operations, which lead to latency instability for reads. ZenFS at the same time is able to remain stable. This makes it likely that indeed the file system is to blame for the stalling of operations. This requires a more in-depth investigation, to find the exact cause. For now, we know that the file system F2FS can at this point no longer guarantee latency stability, but the file system ZenFS can guarantee latency stability even beyond 99.99% tail latency.

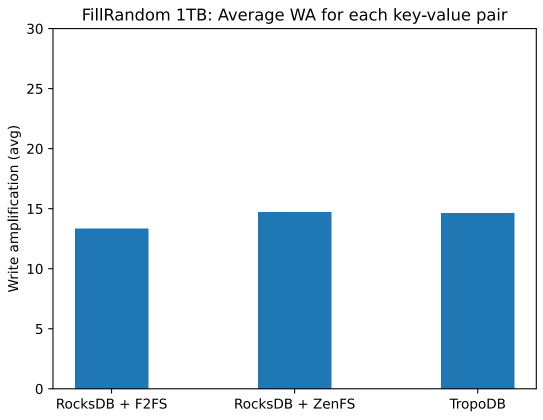
In the end, ZenFS is able to maintain relatively stable throughout all tests and is able to guarantee the best SLAs under the workloads used here in combination with the hardware/software configuration used here. This shows that optimising file systems for ZNS has merit and that optimising file systems or storage for key-value stores also has merit. It can aid in gaining a more stable key-value store. On the other hand, TropoDB has not been able to confirm this. Future research can look into other approaches than taken by TropoDB (Garbage collection threads, L0 and LN designs) and find a better storage solution that is able to lead to similar/better results as RocksDB + ZenFS.

### 6.5.3 TropoDB Write Amplification

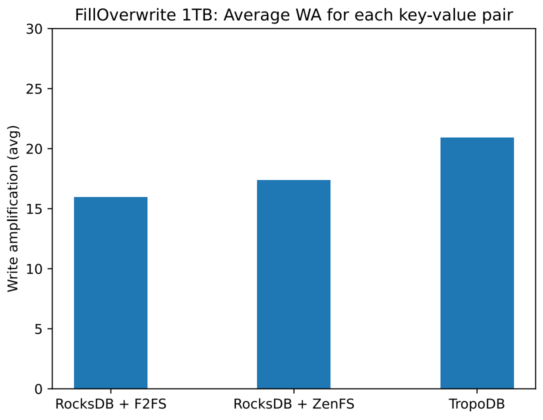
For both ZenFS and F2FS it is not possible to directly measure the number of bytes written by RocksDB. In general, file systems do not log the amount of data written by one application, they only maintain how much logical data a file currently contains. RocksDB itself can also not measure how much data it has written explicitly, as it has no control over the data movement of the underlying file systems. For this experiment, we only require to know how much data is written in total during the benchmark. If no other process is busy writing data to the file system, it is valid to measure the total amount of data written by the file system to the SSD. This is possible as the SSD used has S.M.A.R.T. log support (1). S.M.A.R.T. log support allows requesting data from the SSD, such as how many bytes are written to this SSD in total over its lifetime. This is what we use for benchmarks for ZenFS and F2FS. Before a benchmark is run and after a benchmark is run, the S.M.A.R.T. log data is requested. The amount of data written by the benchmark is then estimated to be the difference in values between the two measured values. However, the S.M.A.R.T. log as defined in the NVMe specification does not use conventional units to measure bytes written. It is defined in units of 1000 blocks with a block size of 512 each and is rounded up. Therefore, there is a small measurement error for this design, equalling $512 * 999$ bytes at most, but considering that more than 1 TB is written for both the fillrandom and the filloverwrite test, a few hundred kilobytes as an error margin is negligible.

TropoDB uses another approach as TropoDB uses SPDK and, therefore, the device is not bound to the kernel during the benchmarks. It should still be possible to access the S.M.A.R.T. log from SPDK itself, by adding some additional measurement tools in SPDK, but this has not been attempted. Instead, TropoDB comes with its own measuring tool. Each write to storage is measured, along with its exact size in bytes. This does not have the measurement errors of the other two approaches. The total number of bytes written is reported after each benchmark is run. This is done for both the entire database and for each individual component, allowing an accurate depiction of what structures are I/O heavy.

In order to estimate the cost in bytes written of put operations for the key-value stores, the cost is normalised over all put operations. It is normalised by dividing the total number of bytes written during the benchmark by the size of all inserted key-value pairs ( $(1000 + 16) * 1.000.000.000$). The resulting number should be equal to the average write amplification of each key-value pair insertion. It divides the total I/O workload by the amount of data that is logically inserted. The results of the tests can be seen in



**(a)** Workload fillrandom



**(b)** Workload filloverwrite

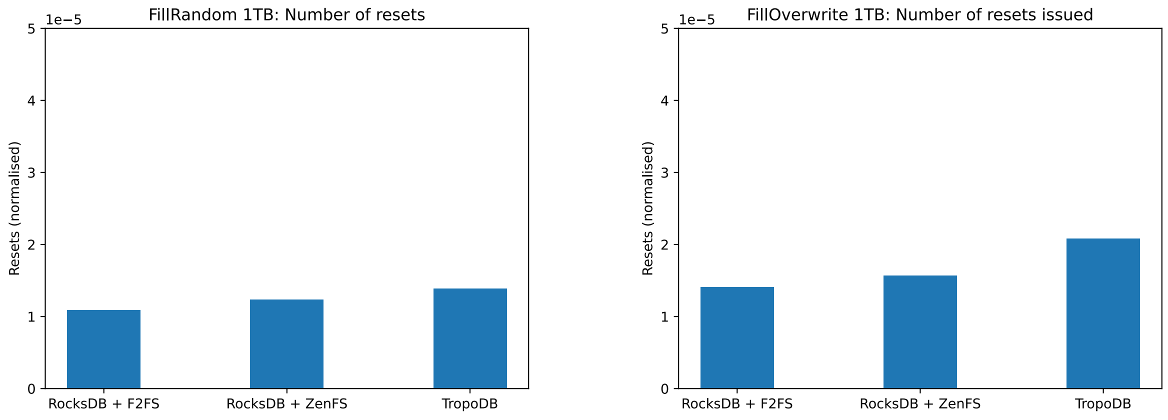
**Figure 6.12:** Write amplification: Measured as number of bytes written divided by the size of all stored key-value pairs

Figure 6.12 for fillrandom and filloverwrite. The x-axis represents the different database configuration and the y-axis represents the write amplification as it has just been described. For fillrandom all three candidates: RocksDB + F2FS, RocksDB + ZenFS and TropoDB do not differ that much. They are all close to a total amplification of 14 and 15. What is noticeable and a bit unexpected considering F2FS had less stable latency than ZenFS during the previous test, is that F2FS has the lowest amplification. It is not sure if this is because of buffering in F2FS or because some of its data is written to Optane. This requires further investigations. For the test run, both ZenFS and TropoDB have not been able to reduce the I/O traffic. TropoDB can not claim that it has reduced the garbage collection effects in the form of write amplification.

In the overwrite benchmark, for all three candidates the write amplification increases. The difference between ZenFS and F2FS remains about the same, but the write amplification of TropoDB increases significantly. TropoDB only rewrites data when it is issued by the key-value store, which means that there is a device level write amplification of about 1. Nevertheless, the database design of TropoDB is issuing frequent appends because of flushes and compactions, causing high write amplification from the application. TropoDB uses a different compaction strategy and implementation than RocksDB, based on LevelDB, which is likely the cause of the issue. In order, to verify that this is the case, it is worthwhile to investigate either the same approach in RocksDB itself or by measuring the total number of compactions, flushes and their individual sizes. The second approach allows making an estimate of the application-level write amplification. Nevertheless, regardless of the reason why TropoDB has a higher write amplification than RocksDB, it has not been able to reach its goal to reduce write amplification of the garbage collector. Proper communication between storage and the key-value store is not the only contributor to write amplification and this result is able to highlight this issue.

### 6.5.4 TropoDB Zone Reset Count

Measuring zone resets is more complicated than the other metrics. By default, there is no default tool to measure how many zones are reset. This metric is also not included in the S.M.A.R.T. log used to measure the number of written bytes. However, as both ZenFS and F2FS are used within kernel space (up to some regard), it is possible to trace the system calls for resets issued by the file systems. This is done with a custom BPFTrace command that tracks each reset and its location (the location of the zone). This script is started just before each benchmark and finished just after each benchmark. This allows tracing all reset calls that occurred during the tests. This approach is not possible with SPDK as SPDK lives in user space. SPDK has added limited support for BPF, but this has not been used to measure resets for TropoDB. Instead, a similar approach is taken as we did to measure the number of bytes written for TropoDB. All resets are logged manually by the database. The results are returned to the callee of the benchmark after each test has been completed.



**(a)** Workload fillrandom

**(b)** Workload filloverwrite

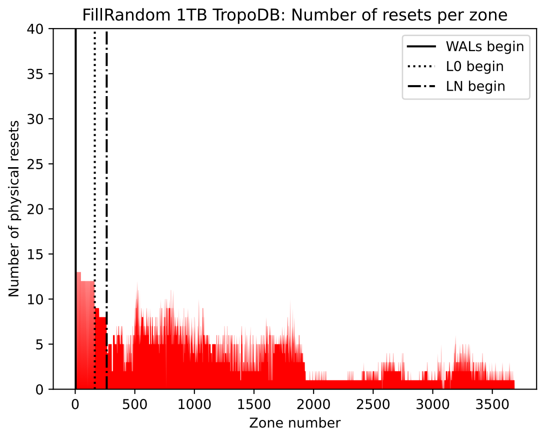
**Figure 6.13:** Number of physical resets: Normalised over the number of stored key-value pairs

The results of the tests can be seen in Figure 6.13 for fillrandom and filloverwrite. The x-axis represents the databases (configurations) and the y-axis the raw number of resets issued to the ZNS device during the entire workload. The results are similar to the write amplification results we saw earlier. Both RocksDB + ZenFS and TropoDB need more resets than F2FS for both workloads. TropoDB needs slightly fewer resets than RocksDB + ZenFS for the fillrandom workload, but significantly more than RocksDB + ZenFS for the filloverwrite workload. In short, F2FS needs the lowest number of resets to deal with the workloads presented in all cases. To find the exact reason why F2FS needs the fewest resets, we need more in-depth investigations. In general, the reason why more resets are needed for TropoDB can be because of a number of reasons. For example, F2FS uses an active GC in the background, while both TropoDB and ZenFS do not postpone garbage collection, which can lead to more aggressive resets. It might, therefore, be that F2FS contains more zones that can be reset (and are thus invalid), but are not reset yet. Similarly, TropoDB has a different flush and compaction strategy. This can result in more or fewer compactions, bigger merges and in general very different behaviour. If more data becomes invalidated in this manner, it also inherently needs more resets. TropoDB never moves or invalidates data by an operation that is not part of the LSM-tree. Zones can only be invalidated by discarding WALs and compactions. So it is not likely that it is because of hot and cold separation or a storage design constraint. Lastly, in the conducted tests, F2FS stores its metadata in second device. This device is not accounted for in the tracing script. Therefore, not all deletes or overwrites issued by F2FS are also measured. The real number of resets needed by F2FS is likely to be higher than what is shown in the graph.

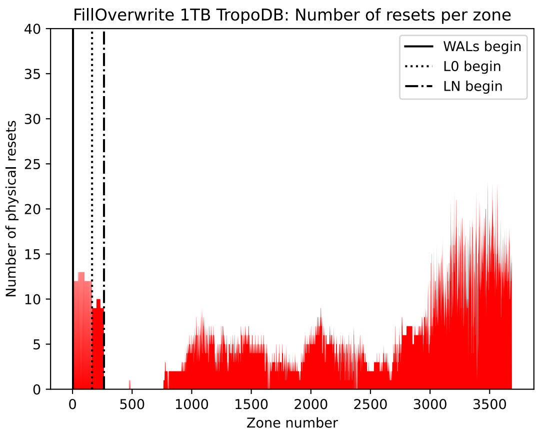
### 6.5.5 TropoDB Heat Distribution

As stated before, TropoDB logs both reset and append operation issued to zones. However, TropoDB also logs this data individually for each zone. This makes it possible to see the temperature of each individual zone and makes it possible to see if the heat of writes/resets is spread evenly. Similarly, the BPFTrace script used to log the resets for ZenFS and F2FS also logs the zones that are reset. This makes it possible to see where all resets happen for all database(s) (configurations). If resets only happen in a small number of zones, this can lead to a few zones burning faster than others and lead to a shorter SSD lifespan. It can also help to explain how each of the individual solutions does their garbage collection and in the case of TropoDB we can even see how each individual component spreads its heat.

Therefore, we have also made plots of the heat distribution of resets during the fillrandom and filloverwrite benchmarks. The results for TropoDB are visible for fillrandom and

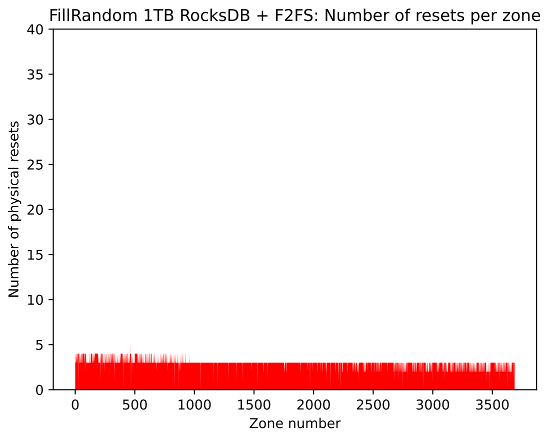


(a) Workload fillrandom

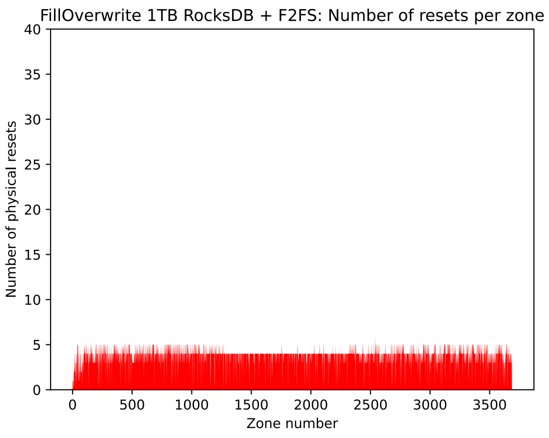


(b) Workload filloverwrite

**Figure 6.14:** Distribution of zone resets for TropoDB



(a) Workload fillrandom

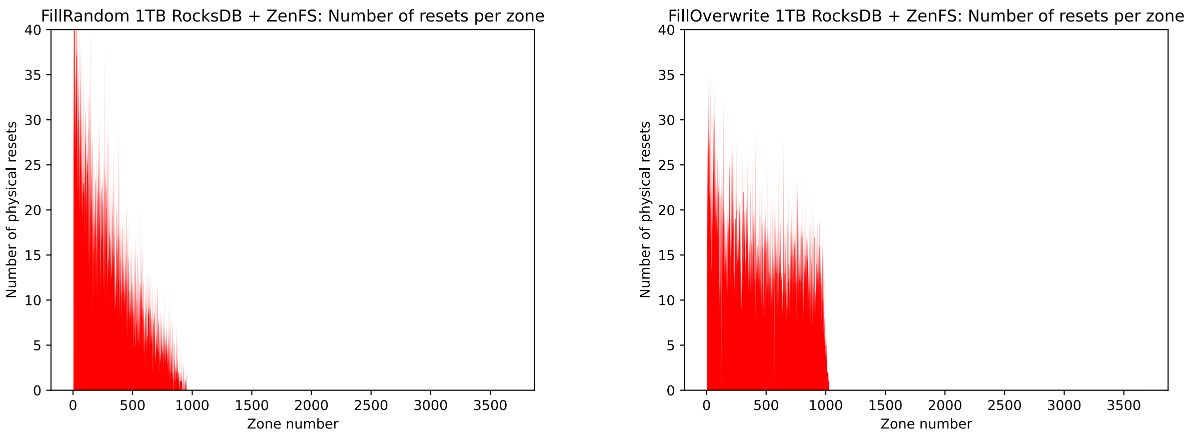


(b) Workload filloverwrite

**Figure 6.15:** Distribution of zone resets for RocksDB + F2FS

filloverwrite in Figure 6.14. The results for F2FS and ZenFS can be seen in Figure 6.15 and Figure 6.16 respectively. The x-axis represents all zones of the SSD as a flat sequential array and the y-axis represents the number of physical resets issued to each zone during the benchmarks. The results are insightful as they showcase that all three solutions use a different strategy for what zones to pick for new data and what zones to reset.

F2FS is able to spread its resets evenly across all zones and properly spreads the temperature of resets in this manner. This is true for both workloads. ZenFS does not evenly



**(a)** Workload fillrandom

**(b)** Workload filloverwrite

**Figure 6.16:** Distribution of zone resets for RocksDB + ZenFS

spread its workload and has issues that can be associated with zone burning. All zone resets happen in the first zones. This happends more in fillrandom than in filloverwrite. ZenFS is unable to properly spread the heat of its resets. This can be explained by the strategy used by ZenFS to reset zones and reserve zones for new files. It does not make use of active garbage collection. Active garbage collection is a separate process in the background that determines whether data should be moved to different zones or zones need to be reset. The opposite of active garbage collection, is only doing issuing garbage collection during other foreground/background operations. ZenFS does not make use of such a process, instead it invalidates zones immediately that are invalidated and then reuses these zones when necessary. An LSM-tree will lead to frequent invalidations because of compactions. Old SSTable files will be invalidated and a new one will be generated. If the zone allocator used does not explicitly rotate the zones, results such as in Figure 6.16 can occur.

TropoDB is able to spread its heat better than ZenFS, but less than F2FS. In the graph, we have explicitly marked where each component begins. For example, WALs begin at zone 4 and L0 begins at zone 164. Each component clearly has a different amount of resets needed for each zone. WALs need significantly more resets for each zone, even with the strategy of rotating the zones. It is presumed that if no such circulation was used, the peaks would have been higher. L0 also needs more resets than LN. However, both WALs and L0 spread their resets properly, which makes sense as they rotate the zones used (WAL manager's circulation scheme and circular buffer in L0). The load for WALs and L0 is similar for both fillrandom and filloverwrite. LN has more problems with spreading its assigned load. While LN does use all zones it can, some zones are reset more frequently than others. This moves from the first couple of LN zones in fillrandom to the last couple of LN zones in filloverwrite. It is suspected that LNs improper heat distribution is because of the zone allocation scheme used, the merging allocation scheme. The merging allocation scheme does not properly rotate and is likely to reuse a small set of zones more frequently. Another detail that is valuable to investigate is the lack of resets between approximately zone 150 till zone 750 for filloverwrite. Zones are only ever reset if they are invalidated, this makes it likely that the zones present in this range were not invalidated and thus not involved in garbage collection. This is only possible if these *cold* SSTables are not stored together with *warmer* SSTables, necessitating data movement. Such patterns are not observed in ZenFS and F2FS. This might mean that TropoDB has succeeded in separating cold LN tables from warmer tables, but this will need a more in-depth investigation to be sure. Without a more thorough analysis, we can not even exclude a large persistent leak, as unlikely as it may seem.

### 6.5.6 Additional Experiments

We have done a few benchmarks to measure the effects on garbage collection, but there are still a lot of experiments that can be done. We will shortly name a few experiments that we consider to be important to get a bigger picture of the effect of garbage collection.

Firstly, it would have been a valuable experiment to see the results with a few different options. For all three tests conducted buffering for WALs is enabled and sync is disabled. These options do not guarantee high reliability in the key-value store itself. The inverse of these options should be tested to see the effects of garbage collection for use cases that require high reliability from the key-value store itself. Further on, we have tested with a fixed key and value size. We have seen multiple times that I/O size matters for performance (Chapter 6.3 and Chapter 6.3.1 for example), this should be tested for a full workload as well.

Secondly, it is a good idea to measure and compare the number of flushes, compactions and the involved SSTables that occurred during the workloads. TropoDB and RocksDB use different strategies to determine when to flush and compact and what to flush or compact. This makes it non-trivial to determine if latency, write amplification or reset counts per key-value pair are because of these strategies or because of garbage collection. If we have a good overview of these strategies and how they behave for the workloads, it should be possible to see a relation between the two strategies and the measured effects.

Thirdly, the designs should also be tested on different ZNS SSDs. The ZNS SSD used has a relatively large zone capacity of more than 1 GB. Some devices have only zone capacities of an MB. The resulting storage will behave very differently in such cases and the exact effect should be investigated.

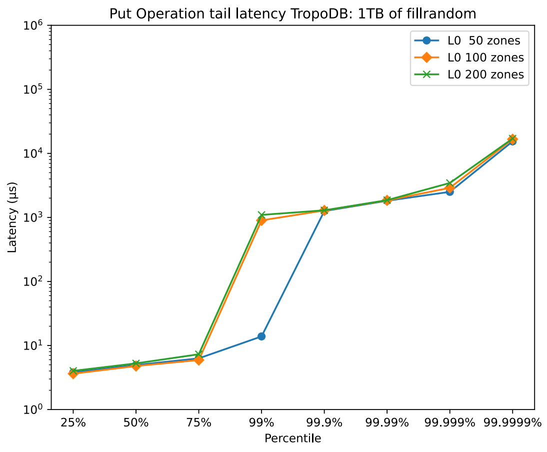
## 6.6 TropoDB L0 Circular Log Evaluation

The circular log used in L0 comes with a few different options. As the circular log design for L0 is new, it is not yet known what the relation between these configurations and the garbage collection process is. In this section, this will be investigated. The tests run will be similar to the garbage collection tests in the previous section, but the tests will compare TropoDB configurations instead of TropoDB and RocksDB configurations. In particular, the effect of the reserved size for L0 will be observed. Other tests that could have been conducted and are insightful are the effect of the size SSTables in L0, the slowdown threshold of L0 (once more tables are stored than this, puts are stalled) and using fragmented logs in L0 instead of circular logs.

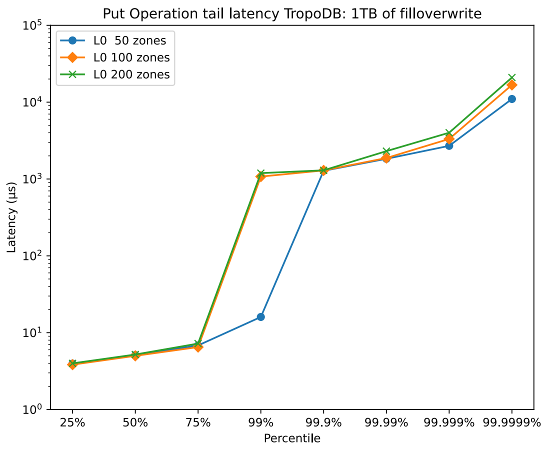
### 6.6.1 Effect of Increasing the Size of the L0 Region

The circular log of L0 reserves a fixed number of zones. This can be configured in TropoDB. The size determines the maximum amount of data that can be stored in the circular log of L0. The policy to compact from L0 to L1 is always set to be lower than this threshold to prevent having to wait for L0 compactions after L0 is filled. It is always set lower because setting the policy larger than the size of L0 will repeatedly get L0 back to the full state repeatedly after each flush, each time forcing a wait. However, this policy does not prevent L0 from filling up. In this case, L0 buffers extra SSTables for later. This can have various effects. Firstly, as the size of L0 increases, more SSTables are available to pick for merges. When more SSTables are considered for a merge (and picked for a merge), the amount of memory used for a compaction increases. When more SSTables are used for a compaction, this particular compaction will also take longer to complete. TropoDB does not allow for preemption, which results in other compactions waiting for this compaction to finish. Whether or not this is common and leads to issues, requires investigations. Secondly, once the log is full, more needs to be done to clean it. The exact effect such a situation has is not known yet.

The exact effects need to be investigated. Therefore, TropoDB is tested with a few different L0 sizes. In total three L0 sizes sizes are picked: 50 L0 zones, 100 L0 zones and

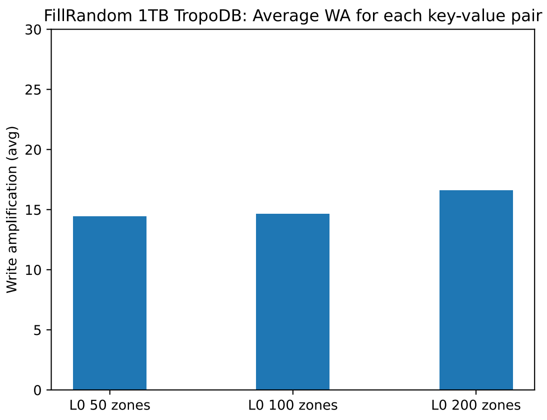


(a) Workload fillrandom

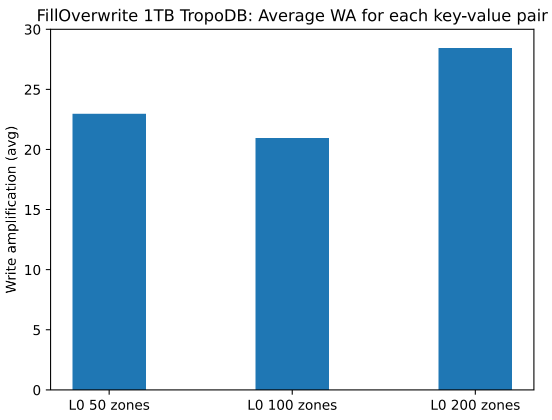


(b) Workload filloverwrite

**Figure 6.17:** TropoDB with varying L0 size: Put operation tail latency



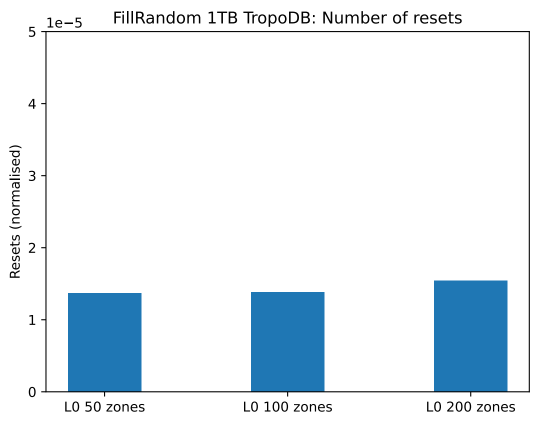
(a) Workload fillrandom



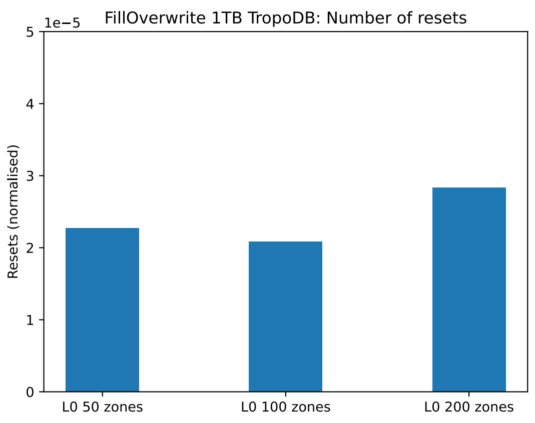
(b) Workload filloverwrite

**Figure 6.18:** TropoDB with varying L0 size: Write amplification, measured as number of bytes written divided by the size of all stored key-value pairs

200 L0 zones. Apart from this, the exact same data base configuration is used for each candidate. They will also use the exact same workload. This is the same workload as used for the tests to retrieve tail latency: fillrandom followed by filloverwrite and then finished with readwhilewriting. Readwhilewriting is not shown, as it crashed (the same crash as described in Chapter 6.5). The results for filloverwrite and fillrandom are visible in Figure 6.17 for tail latency, Figure 6.18 for write amplification and Figure 6.19 for the



**(a)** Workload fillrandom



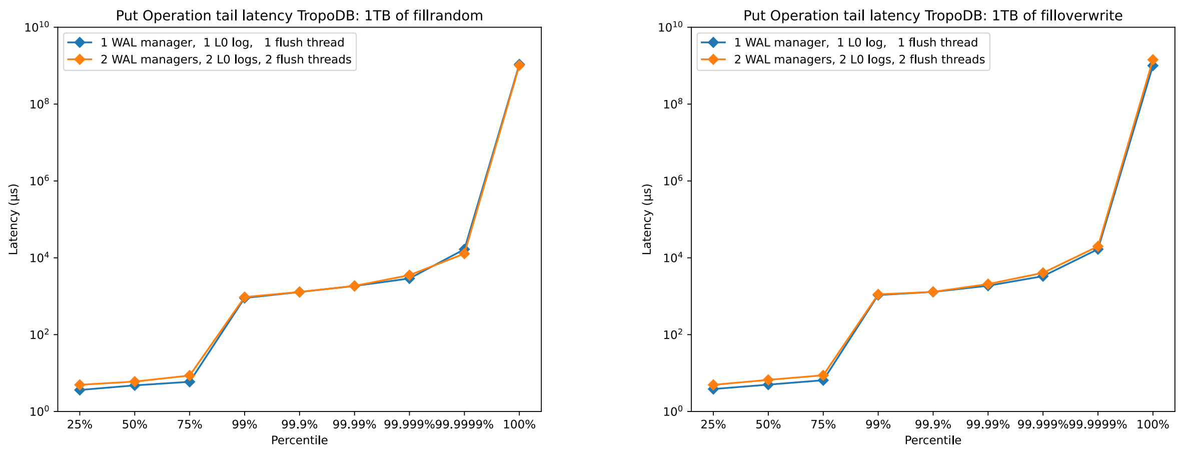
**(b)** Workload filloverwrite

**Figure 6.19:** TropoDB with varying L0 size: Number of resets issued

number of resets normalised over the number of key-value pairs. What is noticeable in these results is that lowering the number of zones to 50 from the default 100, has a significant impact on the tail latency at 99% (the same was witnessed for fillrandom and the test has been run multiple times to confirm). It is suspected that when the number of zones for L0 is increased, more SSTables can be stalled in L0 before they are compacted to L1, but this causes a larger compaction in the end. This is amplified by the issue of deletes in L0 that we mentioned earlier. The design has (un)intentionally created a delayed compaction, which is generally not a good idea to do in LSM-trees (15) as it does indeed cause latency spikes. The difference in tail latency between the fillrandom and filloverwrite workload is small. The effects on write amplification and resets are less clear and we need more data to be sure what they exactly convey. Using 50 zones leads to higher write amplification and more resets for filloverwrite than using 100 zones in this case, but less than 200 zones. At the same time, using 50 zones leads to lower write amplification and requires fewer resets than using 100 zones for the fillrandom workload. A possible explanation for the difference is that when the size of L0 decreases, more compactions are required to occur from L0 to L1 (less space for stalling). When more compactions are reqquired, more data is merged repeatedly as well. It is only possible to stall a number overlapping SSTables on L0, which depends on the size of L0. However, as stated before to make a conclusion, more data will need to be retrieved. The relation to tail latency is clear on the other hand.

## 6.7 TropoDB Effect of Using Multiple L0 Circular Logs and WALs Concurrently

In this section, we will evaluate the effects of the design proposed in Chapter 4.7. During implementing the design and testing its validity, a decrease in performance was measured for small testing benchmarks. This can be because of numerous reasons, but it is still valuable to get an idea of how this design exactly behaves for representative workloads and for larger I/O. This makes it possible to see if there is a bigger underlying issue and what to avoid, or if the decrease performance is due to incorrect testing tools. Therefore, some tests were conducted with different levels of concurrency. Again the same tests were conducted as in Chapter 6.5. The only difference is that the level of concurrency is altered for each test run in the configuration file. The following levels of concurrency were tested: 1,2 and 3. Each additional level of concurrency adds one WAL, one circular L0 log and one flush thread.

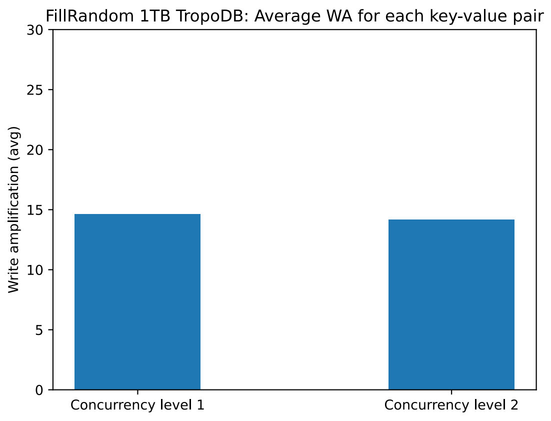


**(a)** Workload fillrandom

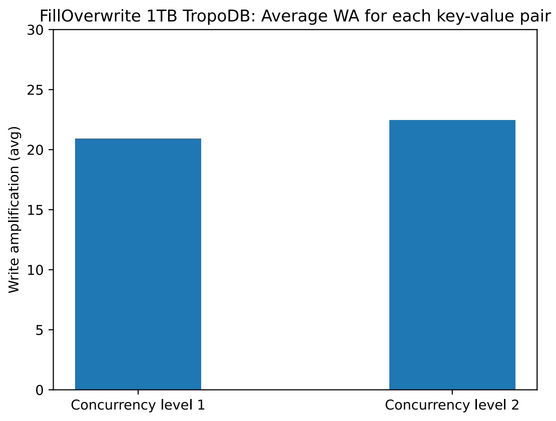
**(b)** Workload filloverwrite

**Figure 6.20:** TropoDB with higher concurrency: Put operation tail latency

We only have results for the levels of concurrency of 1 and 2, as the test case with a concurrency level of 3 has failed. This is likely because of an error in the implementation of the thread model of concurrent flushes. The results are visible in Figure 6.20 for tail latency, Figure 6.21 for write amplification and Figure 6.22 for total number of resets issued. The x-axis of the write amplification- and reset graphs shows the concurrency level. Each concurrency level adds a WAL manager, L0 log and flush thread. What is noticeable

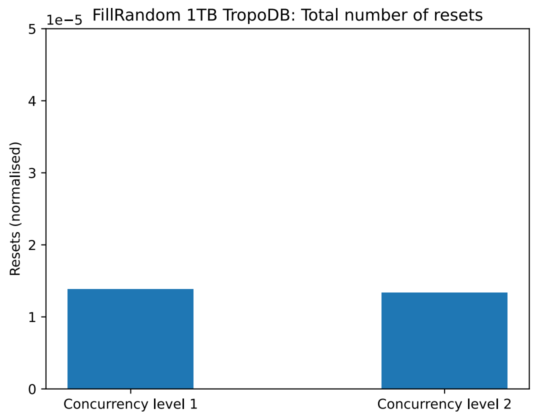


**(a)** Workload fillrandom

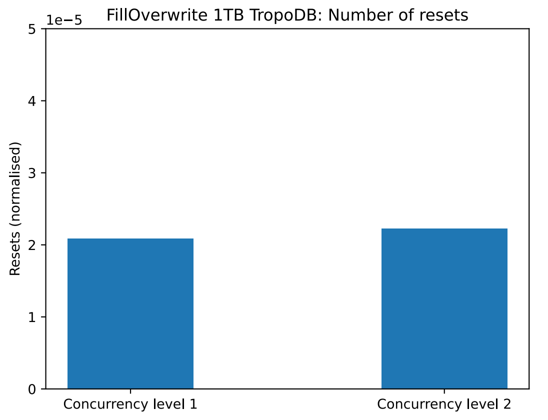


**(b)** Workload filloverwrite

**Figure 6.21:** TropoDB with higher concurrency: Write amplification, measured as number of bytes written divided by the size of all stored key-value pairs



**(a)** Workload fillrandom



**(b)** Workload filloverwrite

**Figure 6.22:** TropoDB with higher concurrency: Number of resets issued

in the resulting graphs, is that the implementation of higher concurrency has decreased performance in all aspects. Parallelism does not decrease latency or stabilise latency. Similarly, write amplification and the number of resets are higher when concurrency is increased. This is true for both workloads, except for a lower write amplification with a concurrency level of 2 during the fillrandom workload.

To get more insights into the behaviour of the concurrent implementation, we decided to inspect the results of one of the workloads in more detail. A closer inspection of the number of bytes written for each component during the filloverwrite workload showed that the number of bytes written to the WALs is identical, which is as expected. The number of writes written to L0 is a little bit bigger, but the number of bytes written to LN is close to 4% more. It also shows a large increase in compactions from L1 to LN onwards. There are more SSTables in level 2 and level 3. This can happen if fewer compactions have interfered with compactions from L0 to L1 and as a result, more compactions could have been completed. The increased number of compactions is also assumed to be the reason for the small increase in latency. The exact reason for the increase in compactions is unknown, but it is likely to be related to the new characteristics of L0. There are now two smaller L0 logs and we already know that the size of L0 has a big effect on write amplification and resets, which we saw in the L0 size experiment (see Chapter 6.6.1). However, as no gains are achieved in any way, it is likely that the proposed concurrency design is not ideal and needs significant alterations to be so. Yet, as the number of tests conducted to measure performance differences for higher concurrency levels is limited, we need more tests to be absolutely certain.

## 6.8 Summary

We have evaluated TropoDB in this chapter. In particular we have evaluated performance of ZNS with SPDK, evaluated garbage collection effects of TropoDB as a whole and looked at garbage collection effects of individual TropoDB components.

The evaluations for the performance of ZNS with SPDK, showcase various details about ZNS performance. For the device investigated, the latency and throughput of I/O operations depend on the operation issued and the two metrics are related. I/O operations with higher throughput are also able to achieve lower latencies. The I/O operations achieve in order of latency and throughput: resets, appends, writes, random reads, sequential reads. Notably, we have also shown that striping writes across zones achieves higher througput and lower latency than issuing multiple appends asynchronously to the same zone.

The garbage collection effect evaluations show that RocksDB + ZenFS is able to achieve the lowest tail latencies for all workloads tested and is able to achieve stable latency. This is true for write-heavy workloads, overwrite-heavy workloads and workloads performing writes and reads concurrently. RocksDB + F2FS and TropoDB are not able to achieve low tail latencies stable latencies like ZenFS, but the two solutions are shown to achieve comparable results. Write amplification and the number of resets issued during workloads are also shown to be correlated. When one of the two increases, so does the other metric.

RocksDB + F2FS is able to achieve the lowest write amplification and require the lowest number of resets. This is followed by RocksDB + ZenFS and RocksDB + ZenFS is followed by TropoDB. Lastly, we took a look at the distribution of resets across zones. RocksDB + F2FS is able to distribute its heat evenly across all zones. TropoDB has some peaks and is not able to achieve the same stability. RocksDB + ZenFS is able to achieve worst in this regard and only issues resets to a couple of zones, leading to zone burning.

The evaluations for the WAL component are also fundamental. The performance of WALs is shown to match the performance of ZNS with SPDK, both for ordered and unordered WALs. Unordered WALs with higher queue depths for appends also match the performance of appends with higher queue depths. The recovery cost in latency of unordered WALs is shown to be more than the recovery cost of ordered appends and lies between 5 and 15%.

Lastly, we evaluated the impact of the size of the circular log in L0 and the impact of increasing concurrency in TropoDB. Altering the size of the circular log in L0 has mixed results. Using a large size for the circular log can result in higher tail latencies. However, both increasing and decreasing the circular log can lead to higher write amplification and more resets. Increasing the concurrency in TropoDB also leads to mixed results. Increasing concurrency leads to higher tail latencies and average latencies. Similarly, it can both lead to higher write amplification and more resets, but the contrary is also possible. No clear relation between higher concurrency and write amplifications/resets has been discovered.

# 7 Conclusion

In this thesis we have presented a new key-value store, TropoDB, a key-value store built entirely on top of ZNS SSDs without file systems or any other layer in between the key-value store and storage. The goal of TropoDB is to implement a collaborative GC in which the GC patterns of storage are directly mapped to the GC patterns of the key-value store with minimal interference. The expected result is that by improving the collaboration of the GCs, their impact can be decreased, with results such as a more stable tail latency, lower write amplification and lower amount of resets needed. Our results have not shown that the approach taken by TropoDB can reduce these effects. However, this does not mean that collaborative GC can not have positive positive effects. It merely shows that the approach taken by TropoDB is not effective enough. The designs shown in Chapter 4 and the implementation described in Chapter 5, can be used to look for further optimisations. Additionally, as it is the first of its kind, many of the ideas proposed, both positive and negative, can be used to get a better understanding of what works and does not work. This can in turn be used to come with better designs built on top of ZNS or how existing key-value stores can be modified to get ZNS benefits. It is thus a proper proof of concept and the first step to better collaboration between flash storage and key-value stores. Lastly, we have seen impressive differences in latency stability between ZenFS and F2FS for larger workloads. ZenFS is more optimised for ZNS and has less of a semantic gap between the key-value store and ZNS, provided that optimisations for ZNS can have significant advantages if done correctly.

## 7.1 Research Questions

**RQ1: What are the key-value store integration choices available on top of ZNS SSDs?**

There exist multiple approaches to integrate a key-value store and ZNS SSDs, as described in Chapter 3. We identified the following approaches: creating a file system that supports ZNS in between the key-value store and ZNS, creating a key-value store from the ground up on ZNS and two approaches that integrate ZNS into a state-of-the-art key-value store known as RocksDB. The approaches that integrate into RocksDB are a top-down approach, where the benchmarking tool will be reused and RocksDB ecosystem will be available, and a bottom-up approach, where RocksDB will be modified to support ZNS. There is a trade-off to pick between the different integration choices. Removing layers of abstractions, adds more control over ZNS and tighter integration, but requires (re)implementing more logic and requires a hefty developer investment. In the end, the bottom-up approach was taken for TropoDB. The bottom-up approach gives the key-value store direct control over the storage, and allows reusing existing key-value store logic and tools, such as benchmarks and in-memory components.

**RQ2: What unique optimisation opportunities does a ZNS device offer for LSM-tree based key-value stores?**

ZNS devices offer various unique optimisation opportunities. In Chapter 4, we describe in detail how TropoDB makes use of all of them. ZNS comes with a novel append operation, that allows increasing concurrency. This operation can be used to great effect for structures that require frequent I/O, such as *write-ahead logs* (WALs) from LSM-trees.

Additionally, ZNS comes with explicit control over the physical locations of data storage, which can reduce the number of overwrites and resets. LSM-tree inherently already limit the number of overwrites and resets as all data is stored sequentially, but without direct control over storage, data can still be reordered on storage (and require overwrites and resets). With ZNS, this is no longer an issue as it can be guaranteed that no data is overwritten and no additional resets are necessary, beyond what is specified by the LSM-tree. This can be done by implementing all data structures as append-only structures and disallow any overwrite. As we have explicit control over data locations, this also allows for optimisations such as not rewriting SSTables when they do not change, but simply only updating the metadata.

To continue, ZNS comes with excellent concurrency capabilities. The ZNS interace allows multiple operations to occur concurrently at entirely different locations on storage.

This can be used to great effect for the task parallelism of LSM-trees. For example, by separately storing data of LSM-tree components such as WALs and SSTables and assigning these components to different threads, it allow them to function concurrently with minimal interference.

Lastly, ZNS gives explicit control over garbage collection. This allows control over not just the garbage collection of an LSM-tree, but of storage as well. This allows coordinating when to erase data on storage and when to issue background operations on the LSM-tree and can reduce latency effects.

**RQ3: How to implement and evaluate the design of an LSM-based key-value store on ZNS devices?**

The design of TropoDB as described in RQ2 and Chapter 4 has been implemented on ZNS. There exist multiple valid methods to implement TropoDB, but we have only looked at one. An implementation that removes the kernel from the I/O path as well, and works entirely in user-space. When the key-value store is implemented in user-space, it adds more control over the latency effects of storage and can augment the performance targets of TropoDB. It has been implemented, following both Chapter 5 and Chapter 3. It is created entirely within SPDK and RocksDB with a bottom-up approach. In order to implement the design, it is also required to be able to evaluate the design. As there is key-value store that is entirely built on ZNS yet, this required using an existing benchmarking tool in the form of db_bench and measuring the on-storage effects of the database. For example, garbage collection effects should be measurable by measuring write amplification, reset count and tail latency of the key-value store(s).

**RQ4: What is the impact of the optimisation process of TropoDB on the key-value store and ZNS device?**

The performance impact of TropoDB has been measured on the key-value store and ZNS device in Chapter 6. We have measured garbage collection effects of TropoDB and seen than TropoDB has not been able to reduce garbage collection effects such as write amplification, reset count and tail latency instability. TropoDB is able to come close to the state-of-the-art, RocksDB + ZenFS, and the state-of-the-practice, RocksDB + F2FS, but suffers from tail latency instability at 99.99% and up, which can take multiple milliseconds to complete for each I/O operation. Write amplification and number of resets is slightly worse, and generally differs between 10 and 20%. However, we have shown that various optimisations made in TropoDB have effect. We have seen that *unordered appends* of the TropoDB WAL component (Chapter 4.3.2) are able to come close to the raw performance of the ZNS append operation in both latency and throughput and do not add significant overhead to read performance.

**Summary**

In short, we have looked at various integration options for building a key-value store on top of ZNS. We have taken a bottom-approach within the RocksDB ecosystem in which we directly interface the key-value store with ZNS. The resulting design, known as TropoDB, makes use of various ZNS novelties effectively by giving each individual ZNS component distinct logic that matches its characteristics. Each component uses ZNS-specific logic where possible. The resulting design was implemented in user-space with SPDK and comes with evaluations within the db_bench benchmark and some custom tooling. The implementation has not been able to reduce garbage collection effects, but has shown that its WAL implementation can reach the performance of the ZNS append operation.

## 7.2 Contributions and New Findings

In this section we will highlight the contributions made in the thesis. The contribution list the same as listed in Chapter 1.6. We will also describe some new findings that were originally not the target of the thesis, but that we think are worth sharing.

1. Design considerations for building a key-value store on top of ZNS SSDs. Specifically how LSM-trees fit in zones and how garbage collection of LSM-trees can look for ZNS SSDs.

2. SimpleZNSDevice (SZD), a small API built on top of SPDK, meant to interface with ZNS. This API comes with a simple configuration and some data structures optimised for ZNS. It can be reused for any ZNS project.

3. Benchmarks for ZNS SSDs that showcase the raw device characteristics of reads, appends, writes and resets when SPDK is used.

4. TropoDB, a key-value store built directly on top of SZD in user-space with the help of SPDK. We explore design choices regarding the underlying ZNS storage with a focus on garbage collection.

5. An investigation of TropoDB, showcasing its tail latency, write amplification and zone reset characteristics compared to the state-of-the-art and the state-of-the-practice.

6. Publicly available source code, both for SZD, SPDK and the benchmarks. The source code can be found respectively for TropoDB, SZD and the benchmarks at `https:`

`//github.com/Krien/TropoDB`, `https://github.com/Krien/SimpleZNSDevice` and `https://github.com/Krien/ZNS_SPDK_Benchmarks`. The sources are described in more detail in Chapter 8.

The original research for TropoDB is not about raw ZNS performance or using multiple appends asynchronously for WALs. Nevertheless, we did investigate both of these concepts regardless. The results of those investigations are worthwhile sharing and do show potential. We have shown the raw append, write, read and reset performance that the used ZNS device can give with SPDK in the best-case scenario and that the WAL design can match this performance. We have seen that using concurrent appends to the WAL with multiple outstanding asynchronous appends for one zone matches the raw asynchronous append performance of ZNS with SPDK. The overhead for recovery, while not negligible, is less than the performance gains that can be achieved for appends in certain use cases. Therefore, it would be worthwhile to also investigate the WAL design for ZNS SSDs where append performance is able to scale better and achieves better than writes. If it is, it would be a good idea to make use of concurrent appends where possible, also for existing key-value stores. For example, making a separate design for WALs in RocksDB that can make use of multiple outstanding appends.

## 7.3 Limitations

There are a few limitations in the research that is conducted. It is only right that these are properly described, so that a correct conclusion can be formulated. Most of these limitations come from the novelty of TropoDB, the lack of proper tooling to verify correctness, measure performance and correctly measure garbage collection effects of TropoDB.
**Inaccuracies in measurements of garbage collection effects**:
Measuring the effects of garbage collection is not trivial. We were not able to measure the effects of garbage collection of TropoDB during the readwhilewriting measurements because of a bug, but there are also other issues. Garbage collection affects many components and is inherently tied into the entire storage stack. This makes it problematic to exclude interaction effects from the equation. Generally, in this research, an approach is taken that is only able to measure some effects by approximation. Inherently RocksDB and TropoDB are different key-value stores, even if they are similar in many regards. This can not be understated as the effects of the differences directly alter the characteristics of the key-value stores. Therefore, it is not possible to compare garbage collection of

the two designs without also inevitably adding general performance characteristics of the two databases into account. For example, compactions might happen at less fortunate moments, be more frequent, involve fewer SSTables or be prioritised differently. This is worsened by the different background thread models. RocksDB can use a large number of compaction threads, TropoDB can not. The real garbage collection effects might (and probably are) be a bit different than shown in the conclusion. The results are still valuable as they are a close approximation and showcase if TropoDB is able to outperform the state-of-the-art. It is just necessary to state that the lesser results might be from different aspects of the key-value store than just the storage aspects themselves.

Additionally, the effects measured are noisy. Even if a large number of I/O is used, latency will fluctuate and so will the measured tail latency. There is a great deal of randomness involved when using multiple background threads. Further on, there might be inaccuracies in the write amplification measured or the number of resets issued. TropoDB has its own set of measurement tools for these measurements, which might work differently than the ones used for RocksDB. As they are less battle-tested than for example the S.M.A.R.T. log, they may even be incorrect in the worst case, even if the functions are properly tested (mistakes do happen). Further on, while TropoDB and ZenFS can only operate while the database is running, F2FS is not hindered by this constraint. This might result in F2FS resetting zones or writing data after the database is closed. Some of these operations might, therefore, not be measured.

Lastly, comparing against F2FS is not entirely fair. F2FS in the evaluation makes use of a second Optane disk for file system metadata operations. Such operations are assumed to be negligible, but are discarded from the results. The F2FS results, therefore, do not include all written bytes or resets (internal Optane deletes).

**TropoDB is the first of its design**:
TropoDB is the first key-value store that attempts to build a key-value store directly on ZNS (and in user-space at the same time). This makes it challenging to properly compare it to alternatives. The limitation is similar to the limitation of measuring garbage collection effects: not totally accurate comparisons. For example, there is no alternative ZNS WAL implementation or L0 ZNS implementation to compare against. It is only possible to test either the global effects changing a component has compared to another full implementation (which is probably differently designed) or by implementing multiple components in TropoDB and measuring their differences. The second approach leads to challenges itself. As all research is dependent on by one team of researchers (in this thesis at least), which can lead to bias for a data structure. Future research, if conducted, should

also depend on different components being designed and properly tested against each other (and if possible by multiple researchers).

**Limitations of working in user-space**:
Working in user-space comes with benefits such as more control over storage and generally higher performance, but it also comes with major limitations. For example, many tools are missing. This includes many debugging and performance tools. If those tools are available, it requires fewer reimplementations of existing tools, but it also prevents mistakes. Mistakes in debugging tools or non-trivial debugging tools can lead to undetected errors and safety issues. Mistakes in performance tools can lead to incorrect results, which can lead to incorrect conclusions.

**Evaluating key-value stores and ZNS performance in a virtualised environment**:
All experiments were run within QEMU. It is assumed that the effects of such an approach are limited, but this has not been measured. We have only roughly measured the impact of NUMA (Chapter 6.2). Further on, the ZNS SSD is used within the VM with PCIe passthrough and the Optane disk with paravirtualisation. Some performance could have been lost in the configuration used, which is again not tested.

**Reliability and usability of TropoDB**:
TropoDB is not a complete database. It misses many functionalities that RocksDB has implemented and some database users may consider to be essential. For example, there are no scan operations, filters (bloom filter, cuckoo filters, ribbon filters to name a few) or column families implemented. In addition, crash tolerance is not guaranteed, tested or a goal of TropoDB. Already existing solutions, generally do provide better reliability. This raises issues for TropoDB. If those features would have been implemented, the key-value store would behave differently and it might perform worse in the aspects tested. Therefore, it is not completely fair to compare TropoDB to RocksDB. It must be stated that RocksDB is thus tested against a proof of concept and that it is an approximation of what a full key-value store on ZNS can achieve with TropoDBs design.

## 7.4 Future Work

Throughout this report, we have mentioned various designs, implementations and benchmarks that have not been implemented/conducted, but that would be valuable to investigate in future research. We will not restate all of these issues in this section, but we will give a short overview and come up with some broader ideas for future work, that we think are good next steps to conduct after TropoDB.

**Designing new components and evaluating individual components for TropoDB**:
Many components for TropoDB have not been properly benchmarked to investigate their effect. To mention a few: effect of SSTable size in L0 and LN, zone allocator designs, effects of the number of WAL regions assigned, effects of WAL sizes, performance of writes in WALs, asynchronous I/O for LN, performance of get operations, different thread models and their impact, different compaction strategies. We have also limited ourselves to a predefined set of benchmarks. Using different value sizes or workloads, for example delete heavy, or entirely different benchmarks such as YCSB, can also give insights. Additionally, we have only designed and implemented a few unique storage data structures for each component. We have seen that a circular log in L0 comes with numerous challenges, different designs for L0 such as a fragmented logs would also be good research research options. Lastly, we have tested all I/O on one type of ZNS SSD. It would also be valuable to verify the results on a different ZNS SSD with other characteristics. For example, better append scalability, smaller zones or a different number of active zones.
**Adding multi-tenancy support to TropoDB**:
TropoDB does not support multi-tenancy (68). It is not possible for multiple connections to open to the database or for multiple databases to exist concurrently on the same ZNS device. This is done by design as it gives the key-value store the maximum that is achievable on the storage. However, this is not entirely realistic in reality. Many applications run in parallel with others on the same machines in practice (9, 19). Storage, CPU and memory are shared in such cases. To make a ZNS approach more usable, it should support these scenarios as well. SZD already comes with simple *containerisation* by limiting the zones that can be used by an application, but this should be taken further. For example limiting the number of open/active zones an application can use, prioritising applications and adding permissions for applications. Additionally, running multiple applications in parallel has profound implications on the performance of the applications, garbage collection included. This can be investigated further in future research.
**Implementing TropoDB in non kernel-space**:
Running in user-space comes with many disadvantages, but is also not always possible. Many key-value stores run in the cloud along with many other applications on similar machines (multi-tenancy). In general, permissions are limited and applications can not always be used in user-space (74). TropoDB will thus not work in such situations. For such situations, it is important that there is also a solution that requires fewer permissions and does not need to run the storage in user-space, but can also run in kernel-space instead. Alternatively, it should also be possible to use a swappable storage backend that

uses user-space if possible and not if it is not. This achieves better performance if user-space is available and still guarantees availability in the general case. A similar approach to uDepot can be taken to achieve this (74).

**Creating an entirely different design for TropoDB**:
TropoDB is mainly built on top of LevelDB logic. This logic is not made for modern flash storage and is not able to scale as well as the storage can provide. Many of those challenges can be altered, but inherently it continues on a design that is not made for the storage used. This leads to locking issues and a workaround to make the model work properly with various threads. This might have also caused the concurrent design we added later on to TropoDB, to decrement in performance, rather than to improve in performance. It might, therefore, be a better approach to begin with a completely new design from the start that is made to handle the level of concurrency from the start. This prevents inefficient workarounds. To achieve such designs, approaches for faster storage with high parallelism such as persistent memory are a good start for investigations. Such designs can be reused if applicable.

Additionally, TropoDB takes a unique approach to pre-allocate the amount of space that each component might require. This comes with advantages, but also disadvantages. For example, not being able to alter most components after database creation. Future designs can look into alternative solutions that are able to modify the structure on the fly. Hot and cold separation could have also been done differently. For example, using an algorithm similar to CAZA (81), instead of separating hot and cold SSTable data by separating levels.

# References

[1] **NVMe command set specifications**. https://nvmexpress.org/developers/nvme-command-set-specifications/, last accessed on 2022-08-14. 5, 20, 31, 161

[2] **NVMe Zoned Namespaces (ZNS) SSDs**. https://zonedstorage.io/docs/introduction/zns, last accessed on 2022-08-14. 20, 31

[3] **Storage Performance Development Kit**. https://spdk.io/, last accessed on 2022-02-14. 8, 31

[4] **Non-volatile Memory Express (NVMe) Market by Product (SSDs, Servers, All-flash Arrays, Adapters), Deployment Location (On-premise, Remote, Hybrid), Communication Standard (Ethernet, Fibre Channel, InfiniBand), Vertical, and Region - Global Forecast to 2025**. https://www.marketsandmarkets.com/Market-Reports/non-volatile-memory-express-market-30458978.html, February, 2022. 1

[5] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. **Design Tradeoffs for {SSD} Performance**. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008. 3

[6] Deepak Ajwani, Itay Malinger, Ulrich Meyer, and Sivan Toledo. **Characterizing the performance of flash memory storage devices and its impact on algorithm design**. In *International Workshop on Experimental and Efficient Algorithms*, pages 208–219. Springer, 2008. 16

[7] Shoaib Akram, Manolis Marazakis, and Angelos Bilas. **NUMA implications for storage I/O throughput in modern servers**. In *3rd Workshop on Computer Architecture and Operating System co-design (CAOS'12)*, 2012. 137

[8] Shahriar Akter and Samuel Fosso Wamba. **Big data analytics in E-commerce: a systematic review and agenda for future research**. *Electronic Markets*, **26**(2):173–194, 2016. 1

[9] Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. **Multi-tenancy in cloud computing**. In *2014 IEEE 8th international symposium on service oriented system engineering*, pages 344–351. IEEE, 2014. 182

[10] Ahmed Izzat Alsalibi, Sparsh Mittal, Mohammed Azmi Al-Betar, and Putra Bin Sumari. **A survey of techniques for architecting SLC/MLC/TLC hybrid Flash memory–based SSDs**. *Concurrency and Computation: Practice and Experience*, **30**(13):e4420, 2018. 16

[11] David G Andersen and Steven Swanson. **Rethinking flash in the data center**. *IEEE micro*, **30**(04):52–54, 2010. 15, 16

[12] Manos Athanassoulis, Anastasia Ailamaki, Shimin Chen, Phillip Gibbons, and Radu Stoica. **Flash in a DBMS: Where and How?** *IEEE Data Engineering Bulletin*, **33**(ARTICLE), 2010. 16

[13] Paul Avery. **Data Grids: a new computational infrastructure for data-intensive science**. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, **360**(1795):1191–1209, 2002. 1

[14] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. **TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores**. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 363–375, 2017. 36, 77, 96

[15] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. **SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores**. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 753–766, 2019. 5, 35, 36, 96, 99, 170

[16] Felix Beierle, Vinh Thuy Tran, Mathias Allemand, Patrick Neff, Winfried Schlee, Thomas Probst, Johannes Zimmermann, and Rüdiger

Pryss. **What data are smartphone users willing to share with researchers?** *Journal of Ambient Intelligence and Humanized Computing*, **11**(6):2277–2289, 2020. 1

[17] Emery D. Berger, Stephen M. Blackburn, Matthias Hauswirth, and Michael W. Hicks. **A Checklist Manifesto for Empirical Evaluation: A Preemptive Strike Against a Replication Crisis in Computer Science (2019)**. `https://blog.sigplan.org/2019/08/28/a-checklist-manifesto-for-empirical-evaluation-a-preemptive-strike-against-a-replic` 2019. 10

[18] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. **{ZNSwap}:{un-Block} your Swap**. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1–18, 2022. 37

[19] Cor-Paul Bezemer and Andy Zaidman. **Multi-tenant SaaS applications: maintenance dream or nightmare?** In *Proceedings of the joint ercim workshop on software evolution (evol) and international workshop on principles of software evolution (iwpse)*, pages 88–92, 2010. 182

[20] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro Fernandes, Edit Gørøgh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer, Fotis Psomopoulos, Tony Ross-Hellauer, Rene Schneider, Jon Tennant, Ellen Verbakel, Helene Brinken, and Lambert Heller. **Open Science Training Handbook, Zenodo, DOI:**`https://doi.org/10.5281/zenodo.1212496`, April, 2018. 10

[21] Aadhya Bhatt, Ananta Bhatt, and Jinan Fiaidhi. **Next Generation Smart Fridge System using IoT**. 2020. 1

[22] Matias Bjørling. **From open-channel SSDs to zoned namespaces**. In *Proc. Linux Storage Filesyst. Conf.(Vault)*, **1**, 2019. 20

[23] Matias Bjørling. **Zone Append: A new way of writing to zoned storage**. *Santa Clara, CA, February. USENIX Association.*, 2020. 143

[24] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. **ZNS: Avoiding the Block Interface Tax for Flash-based SSDs**. In *2021 USENIX*

*Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021. 2, 5, 8, 19, 20, 30, 33, 34, 37, 43, 44, 55, 77

[25] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. **LightNVM: The Linux Open-Channel SSD Subsystem**. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, February 2017. USENIX Association. 34, 116

[26] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. **Specifying and checking file system crash-consistency models**. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83–98, 2016. 56

[27] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. **POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database**. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, 2020. 2, 12

[28] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. **Characterizing, Modeling, and Benchmarking RocksDBKey-Value Workloads at Facebook**. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020. 23, 24, 31, 44, 117

[29] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. **LDC: a lower-level driven compaction method to optimize SSD-oriented key-value stores**. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 722–733. IEEE, 2019. 36

[30] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. **Bigtable: A distributed storage system for structured data**. *ACM Transactions on Computer Systems (TOCS)*, **26**(2):1–26, 2008. 2

[31] Feng Chen, David A Koufaty, and Xiaodong Zhang. **Understanding intrinsic characteristics and system implications of flash memory based solid state drives**. *ACM SIGMETRICS Performance Evaluation Review*, **37**(1):181–192, 2009. 20

[32] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. **SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage**. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021. 59, 64, 107

[33] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. **A New LSM-style Garbage Collection Scheme for ZNS SSDs**. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020. 33

[34] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. **A survey of flash translation layer**. *Journal of Systems Architecture*, **55**(5-6):332–343, 2009. 3, 18

[35] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. **SplinterDB: Closing the bandwidth gap for nvme key-value stores**. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, pages 49–63, 2020. 59, 107, 110

[36] Michael Cornwell. **Anatomy of a solid-state drive**. *Communications of the ACM*, **55**(12):59–63, 2012. 17

[37] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. **Data center energy consumption modeling: A survey**. *IEEE Communications surveys & tutorials*, **18**(1):732–794, 2015. 16

[38] Biplob Debnath, Sudipta Sengupta, and Jin Li. **SkimpyStash: RAM space skimpy key-value store on flash-based storage**. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 25–36, 2011. 2, 23

[39] Diego Didona, Nikolas Ioannou, Radu Stoica, and Kornilios Kourtis. **Toward a better understanding and evaluation of tree structures on flash ssds**. *arXiv preprint arXiv:2006.04658*, 2020. 48, 154

[40] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. **Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring**. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, 2022. 32, 104

[41] Krijn Doekemeijer and Animesh Trivedi. **Key-Value Stores on Flash Storage Devices: A Survey**. *arXiv preprint arXiv:2205.07975*, 2022. 1, 2, 9, 10, 13, 23, 24, 29, 32

[42] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. **Optimizing Space Amplification in RocksDB.** In *CIDR*, **3**, page 3, 2017. 68

[43] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. **Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience**. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021. 2, 8, 23, 29

[44] Adanma Cecilia Eberendu et al. **Unstructured Data: an overview of the data of Big Data**. *International Journal of Computer Trends and Technology*, **38**(1):46–50, 2016. 1

[45] Robert Escriva, Bernard Wong, and Emin Gün Sirer. **HyperDex: A distributed, searchable key-value store**. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 25–36, 2012. 12

[46] Facebook. **RocksDB**. `https://github.com/facebook/rocksdb/`, last accessed on 2022-02-15. 29, 33, 55, 68

[47] Ricardo Galli. **Journal file systems in Linux**. *Update*, **2**(6):50–56, 2001. 88

[48] Mike Gancarz. *Linux and the Unix philosophy.* Digital Press, 2003. 31

[49] Ahmad Ghandour. **Big Data driven e-commerce architecture**. *International Journal of Economics, Commerce and Management*, **3**(5):940–947, 2015. 1

[50] Sanjay Ghemawat and Jeff Dean. **LevelDB**, 2011. 24, 29

[51] Google. **LevelDB**. `https://github.com/google/leveldb`, last accessed on 2022-02-15. 2, 24, 29, 31, 64, 68, 127

[52] YG Grange, VN Pandey, X Espinal, R Di Maria, and AP Millar. **Astronomical data organization, management and access in Scientific Data Lakes**. *arXiv preprint arXiv:2202.01828*, 2022. 1

[53] Casey S Greene, Jie Tan, Matthew Ung, Jason H Moore, and Chao Cheng. **Big data bioinformatics**. *Journal of cellular physiology*, **229**(12):1896–1900, 2014. 1

[54] Laura M Grupp, John D Davis, and Steven Swanson. **The bleak future of NAND flash memory.** In *FAST*, **7**, pages 10–2, 2012. 16

[55] Richard R Hamming. *Art of doing science and engineering: Learning to learn.* CRC Press, 1997. 10

[56] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. **ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction**. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 147–162, 2021. 38

[57] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. **The unwritten contract of solid state drives**. In *Proceedings of the twelfth European conference on computer systems*, pages 127–144, 2017. 48

[58] Gernot Heiser. **Systems Benchmarking Crimes**. `www.cse.unsw.edu.au/~Gernot/benchmarking-crimes.html`, 2020. 10

[59] Izumi V Hinkson, Tanja M Davidsen, Juli D Klemm, Ishwar Chandramouliswaran, Anthony R Kerlavage, and Warren A Kibbe. **A comprehensive infrastructure for big data in cancer research: accelerating cancer research and precision medicine**. *Frontiers in cell and developmental biology*, **5**:83, 2017. 1

[60] Tod Hoff. **Latency is Everywhere and it Costs You Sales - How to Crush it**. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`, July, 2009. 4

[61] Taeho Hwang, Jaemin Jung, and Youjip Won. **Heapo: Heap-based persistent object store**. *ACM Transactions on Storage (TOS)*, **11**(1):1–21, 2014. 2

[62] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking**

**Research in the Netherlands: A Manifesto**. *arXiv preprint arXiv:2206.03259*, 2022. 1, 12

[63] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019. 10

[64] ISO. **ISO/IEC 14882: 2017 information technology—programming languages—C++.**, 2017. 47

[65] Raj Jain. *The art of computer systems performance analysis.* john wiley & sons, 2008. 10

[66] Jhuyeong Jhin, Hyukjoong Kim, and Dongkun Shin. **Optimizing host-level flash translation layer with considering storage stack of host systems**. In *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication*, pages 1–4, 2018. 19

[67] Jeeyoon Jung and Dongkun Shin. **Lifetime-leveling LSM-tree compaction for ZNS SSD**. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 100–105, 2022. 33

[68] Jaap Kabbedijk, Cor-Paul Bezemer, Slinger Jansen, and Andy Zaidman. **Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective**. *Journal of Systems and Software*, **100**:139–148, 2015. 182

[69] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. **Durable write cache in flash memory SSD for relational and NoSQL databases**. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 529–540, 2014. 2

[70] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. **Redesigning {LSMs} for Nonvolatile Memory with {NoveLSM}**. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, 2018. 99

[71] Naim Kheir. *Systems modeling and computer simulation.* Routledge, 2018. 10

[72] Dean Klein. **The history of semiconductor memory: From magnetic tape to NAND flash memory**. *IEEE Solid-State Circuits Magazine*, **8**(2):16–22, 2016. 15

[73] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. **Reflex: Remote flash local flash**. *ACM SIGARCH Computer Architecture News*, **45**(1):345–359, 2017. 151

[74] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. **Reaping the performance of fast NVM storage with udepot**. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019. 32, 36, 182, 183

[75] Se Jin Kwon, Arun Ranjitkar, Young-Bae Ko, and Tae-Sun Chung. **FTL algorithms for NAND-type flash memories**. *Design Automation for Embedded Systems*, **15**(3):191–224, 2011. 18

[76] Avinash Lakshman and Prashant Malik. **Cassandra: a decentralized structured storage system**. *ACM SIGOPS Operating Systems Review*, **44**(2):35–40, 2010. 8

[77] Dave Landsman and D Walker. **AHCI and NVMe as interfaces for SATA Express™ Devices**, 2013. 18

[78] Damien Le Moal and Ting Yao. **zonefs: Mapping {POSIX} File System Interface to Raw Zoned Block Device Accesses**. 2020. 33, 44

[79] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. **{F2FS}: A New File System for Flash Storage**. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015. 2, 44

[80] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. **Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs**. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, 2019. 1

[81] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. **Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs**. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 93–99, 2022. 33, 183

[82] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. **A case for flash memory SSD in enterprise database applications**. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086, 2008. 15

[83] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. **Enabling low tail latency on multicore key-value stores**. *Proceedings of the VLDB Endowment*, **13**(7):1091–1104, 2020. 2, 4

[84] Yair Levy and Timothy J Ellis. **A systems approach to conduct an effective literature review in support of information systems research.** *Informing Science*, **9**:181–212, 2006. 10

[85] Bingzhe Li, Nae Young Song, Li Ou, and David HC Du. **Can we store the whole world's data in DNA storage?** In *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems*, pages 15–15, 2020. 1

[86] Tonglin Li, Xiaobing Zhou, Kevin Brandstatter, and Ioan Raicu. **Distributed key-value store on hpc and cloud systems**. In *2nd Greater Chicago Area System Research Workshop (GCASR)*, **238**, 2013. 12

[87] Greg Linden. **Make Data Useful**. `https://glinden.blogspot.com/2006/12/slides-from-my-talk-at-stanford.html`, December,2006. 4

[88] Chun-Yi Liu, Yu-Ming Chang, and Yuan-Hao Chang. **Read leveling for flash storage systems**. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–10, 2015. 3, 16

[89] Li Liu and Ke Zhou. **PTierDB: Building Better Read-Write Cost Balanced Key-Value Stores for Small Data on SSD**. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 796–801. IEEE, 2021. 28, 35

[90] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. **Wisckey: Separating keys from values in SSD-conscious storage**. *ACM Transactions on Storage (TOS)*, **13**(1):1–28, 2017. 24, 28, 29

[91] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. **NVMeSSD**

**Failures in the Field: the Fail-Stop and the Fail-Slow**. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1005–1020, 2022. 1

[92] Kainan Ma, Ming Liu, Tao Li, Yibo Yin, and Hongda Chen. **A low-cost improved method of raw bit error rate estimation for NAND flash memory of high storage density**. *Electronics*, **9**(11):1900, 2020. 16

[93] Peter Macko, Xiongzi Ge, John Haskins Jr, James Kelley, David Slik, Keith A Smith, and Maxim G Smith. **Smore: A cold data object store for smr drives (extended version)**. *arXiv preprint arXiv:1705.09701*, 2017. 2

[94] Umesh Maheshwari. **From blocks to rocks: A natural extension of zoned namespaces**. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 21–27, 2021. 21, 38, 63

[95] Yoshinori Matsunobu, Siying Dong, and Herman Lee. **MyRocks: LSM-tree database storage engine serving Facebook's Social Graph**. *Proceedings of the VLDB Endowment*, **13**(12):3217–3230, 2020. 2, 23

[96] Marissa Mayer. **Greg Linden: Marissa Mayer at Web 2.0**. `https://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html`, November, 2006. 4

[97] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. **SifrDB: A unified solution for write-optimized key-value stores in large datacenter**. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 477–489, 2018. 28, 35, 96

[98] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. **SFS: random write considered harmful in solid state drives.** In *FAST*, **12**, pages 1–16, 2012. 16

[99] Andy Oram and Greg Wilson. *Beautiful code Leading programmers explain how they think*. O'Reilly Media, Inc, 2007. ii

[100] John Ousterhout. **Always measure one level deeper, DOI:**`https://doi.org/10.1145/3213770`. *Communications of the ACM*, **61**(7):74–83, 2018. 10

[101] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. **The log-structured merge-tree (LSM-tree)**. *Acta Informatica*, **33**(4):351–385, 1996. 24

[102] Fengfeng Pan, Yinliang Yue, and Jin Xiong. **dCompaction: Delayed compaction for the LSM-tree**. *International Journal of Parallel Programming*, **45**(6):1310–1325, 2017. 96

[103] Jay Parikh. **Facebook processes more than 500 TB of data daily**. `https://www.cnet.com/tech/services-and-software/facebook-processes-more-than-500-tb-of-data-daily/`, August,2012. 1

[104] Aditya B Patel, Manashvi Birla, and Ushma Nair. **Addressing big data problem using Hadoop and Map Reduce**. In *2012 Nirma University International Conference on Engineering (NUiCONE)*, pages 1–5. IEEE, 2012. 1

[105] Andy Patrizio. **IDC: Expect 175 zettabytes of data worldwide by 2025**. *Network World*, 2018. 1

[106] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. **A design science research methodology for information systems research**. *Journal of management information systems*, **24**(3):45–77, 2007. 10

[107] Devashish R Purandare, Peter Wilcox, Heiner Litz, and Shel Finkelstein. **Append is Near: Log-based Data Management on ZNS SSDs**. 8, 28, 30, 33, 38, 63

[108] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. **PebblesDB: Building key-value stores using fragmented log-structured merge trees**. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 497–514, 2017. 35

[109] Tore Risch. **Introduction to NoSQL Databases**. *NoSQLDatabases.pdf*, 2015. 23

[110] Sanam Shahla Rizvi and Tae-Sun Chung. **Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems**. In *2010 2nd International Conference on Computer Engineering and Technology*, **7**, pages V7–297. IEEE, 2010. 16

[111] A Harry Robinson and Colin Cherry. **Results of a prototype television bandwidth compression scheme**. *Proceedings of the IEEE*, **55**(3):356–364, 1967. 76

[112] Marek Rusinkiewicz and Amit P Sheth. **Specification and Execution of Transactional Workflows.** *Modern database systems*, **1995**:592–620, 1995. 23

[113] Vatika Sharma and Meenu Dave. **Sql and nosql databases**. *International Journal of Advanced Research in Computer Science and Software Engineering*, **2**(8), 2012. 23

[114] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. **Didacache: an integration of device and application for flash-based key-value caching**. *ACM Transactions on Storage (TOS)*, **14**(3):1–32, 2018. 30

[115] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. **Exploring performance characteristics of ZNS SSDs: Observation and implication**. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–5. IEEE, 2020. 37

[116] Swaminathan Sivasubramanian. **Amazon dynamoDB: a seamlessly scalable non-relational database service**. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012. 2

[117] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. **Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete**. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 144–151, 2021. 8, 28, 33

[118] Iain Sutherland, Huw Read, and Konstantinos Xynos. **Forensic analysis of smart TV: A current issue and call to arms**. *Digital Investigation*, **11**(3):175–178, 2014. 1

[119] Swamit Tannu and Prashant J Nair. **The Dirty Secret of SSDs: Embodied Carbon**. *arXiv preprint arXiv:2207.10793*, 2022. 16

[120] Nick Tehrany and Animesh Trivedi. **Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices**. *arXiv preprint arXiv:2206.01547*, 2022. 31, 37, 44

[121] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is big data performance reproducible in modern cloud networks?** In

*17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 513–527, 2020. 10

[122] Pamela Vagata and Kevin Wilfong. **Scaling the Facebook data warehouse to 300 PB**. `https://engineering.fb.com/2014/04/10/core-data/scaling-the-facebook-data-warehouse-to-300-pb/`, April,2014. 1

[123] Tobias Vinçon, Sergej Hardock, Christian Riegger, Julian Oppermann, Andreas Koch, and Ilia Petrov. **Noftl-kv: Tackling write-amplification on kv-stores with native storage management**. In *Advances in database technology-EDBT 2018: 21st International Conference on Extending Database Technology, Vienna, Austria, March 26-29, 2018. proceedings*, pages 457–460. University of Konstanz, University Library, 2018. 34

[124] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. **An efficient design and implementation of LSM-tree based key-value store on open-channel SSD**. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014. 34, 104

[125] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E Bourne, et al. **The FAIR Guiding Principles for scientific data management and stewardship**. *Nature Scientific Data, 3.10.1038/sdata.2016.18*, 2016. 10

[126] Ross Williams et al. **A painless guide to CRC error detection algorithms**. *Internet publication, August*, 1993. 75

[127] Greg Wong. **SSD market overview**. In *Inside Solid State Drives (SSDs)*, pages 1–17. Springer, 2013. 18

[128] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. **Towards an unwritten contract of intel optane {SSD}**. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019. 1

[129] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. **Performance analysis of NVMe SSDs and their implication on real world**

**databases**. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–11, 2015. 1, 17

[130] Fei Yang, Kun Dou, Siyu Chen, Mengwei Hou, Jeong-Uk Kang, and Sangyeun Cho. **Optimizing NoSQL DB on flash: a case study of RocksDB**. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, pages 1062–1069. IEEE, 2015. 8

[131] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. **Garbage collection and wear leveling for flash memory: Past and future**. In *2014 International Conference on Smart Computing*, pages 66–73. IEEE, 2014. 4

[132] Yadong Yang, Xunong Dong, Bingbing Xie, Nan Ding, Juan Chen, Yongjun Li, Qian Zhang, Hongzhu Qu, and Xiangdong Fang. **Databases and web tools for cancer genomics study**. *Genomics, proteomics & bioinformatics*, **13**(1):46–50, 2015. 1

[133] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. **SPDK: A development kit to build high performance storage applications**. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017. 31, 32

[134] Jeseong Yeon, Leeju Kim, Youil Han, Hyeon Gyu Lee, Eunji Lee, and Bryan S Kim. **JellyFish: A Fast Skip List with MVCC**. In *Proceedings of the 21st International Middleware Conference*, pages 134–148, 2020. 25, 126

[135] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. **Flashkv: Accelerating kv performance with open-channel ssds**. *ACM Transactions on Embedded Computing Systems (TECS)*, **16**(5s):1–19, 2017. 4, 28, 30, 34, 99

8

# Appendix

## 8.1 Artifact Description: TropoDB

### 8.1.1 Abstract

This artifact description describes how to setup TropoDB and reproduce the results as seen in the thesis. We describe how to obtain the software, setup the same benchmarking environment and do the benchmarking. The software consists out of multiple parts: the database TropoDB, a modified variant of db_bench to support TropoDB, a library to interface with SPDK and ZNS known as SimpleZNSDevice (SZD) and a set of benchmarks and plotting tools to get the plots seen in the thesis and benchmark fio.

### 8.1.2 Artifact Check-list (Meta-information)

- Program: TropoDB (`https://github.com/Krien/TropoDB`), SimpleZNSDevice (`https://github.com/Krien/SimpleZNSDevice`), SPDK fio benchmarks, raw benchmark data and plotting tools (`https://github.com/Krien/ZNS_SPDK_Benchmarks`)

- Compilation: C++17, CMake 3.16, Python3 (Python3 is needed for the plotting tools only)

- Experiments: Experiments and configurations for TropoDB experiments are maintained in `https://github.com/Krien/TropoDB/tree/master/implementation/rocksdb/zns_tests`. SPDK fio experiments are maintained in `https://github.com/Krien/ZNS_SPDK_Benchmarks/tree/main/SPDK`. The reset test for SPDK is available in `https://github.com/Krien/SimpleZNSDevice/tree/main/tools/reset_perf`.

- Publicly available?: Source code publicly available.

- Code licenses (if publicly available)?: TropoDB is dual licensed because it copies RocksDBs license, which is a combination of the LevelDB License, Apache license and the GPLv2 license. SZD is MIT Licensed and the SPDK fio benchmarks are MIT licensed.

### 8.1.3 Description

**How to access**
It is not necessary to retrieve SZD or SPDK manually as the correct versions already come as git submodules with TropoDB. The software needed to run TropoDB with benchmarks included can thus be obtained from Github with:

```
$ git clone https://github.com/Krien/TropoDB
$ git submodule update --init
```

The implementation can then be found in "implementation/rocksdb". An initial investigation for LevelDB can be found in "implementation/db" and a CMake script to support ZenFS in modified RocksDB can be found in "implementation/zenfs_cmake". However, the submodu

The fio benchmarks and plot tools can be obtained from Github with:

```
$ git clone https://github.com/Krien/ZNS_SPDK_Benchmarks
```

**Software Dependencies**
The software will only run on GNU/Linux. Note that more software dependencies are required than listed below, but the full extent is not known and highly dependent on the kernel version and configuration used. Dependencies needed to run TropoDB (pinned versions other versions might work as well):

- SPDK v22.05
  (commit c58d5161e9cb9a83b71f308f4e39a0a30ca75014)
- DPDK 21.11.0
- All RocksDB v7.0 dependencies

Additional dependencies needed to run the benchmarks for TropoDB:

- RocksDB v7.0
  (commit 062396af15ea9b295ea72d98cac32206f3dcf299)

- ZenFS v2.1.0
  (commit a0ced69391d3ff3b6663d4749c1af190753798ef)
- F2FS-tools v1.15.0
  (commit 64f2596142800c215cb40a658ebd5793ed37c936)
- nvme 2.0
  (git 2.0-113-geca80c9)
- libnvme 1.0
  (git 1.0-157-g1d62f8a)
- NVMe-CLI
  (commit eca80c919ee988a67e956e524550887cee831cfb)
- BPFtrace
  (git v0.15.0-43-g7667)
- Zoned_BPFTrace
  (commit 2db55de45ebcb3b9c8b263fe431e869f96b5ff8c)

Dependencies needed to run the fio test:

- SPDK v22.05
  (commit c58d5161e9cb9a83b71f308f4e39a0a30ca75014)
- DPDK 21.11.0
- fio
  (git 3.30-73-gd622)

Dependencies needed to make the plots:

- Python 3 (see the requirements.txt in the projects to install the library dependencies)

### 8.1.4 Software and Hardware Configuration

All tests are run on top of QEMU 6.1.0 with KVM enabled.

**Hardware configuration host**

- a 20-core 2.40GHz Intel(R) Xeon(R) Silver 4210R CPU with two sockets connected in NUMA mode. Each socket comes with a total of 10 physical cores and 2 threads for each core.

- 256GB of DDR4 DRAM
- 7TB ZNS SSD, model WZS4C8T4TDSP303
- 280 GB Optane SSD, model INTEL SSDPE21D280GA
- 447 GB SATA SSD, model INTEL SSDSC2kB48

**Hardware configuration host for the NUMA test**

- a 20-core 2.40GHz Intel(R) Xeon(R) Silver 4210R CPU with two sockets connected in NUMA mode. Each socket comes with a total of 10 physical cores and 1 thread for each core.
- 256GB of DDR4 DRAM
- 7TB ZNS SSD, model WZS4C8T4TDSP303
- 447 GB SATA SSD, model INTEL SSDSC2kB48

**Hardware configuration VM**

- a 10-core 2.40GHz Intel(R) Xeon(R) Silver 4210R CPU with one socket of 10 physical cores and 2 threads. Using just one NUMA node.
- 128GB of DRAM, the amount accessible by one NUMA node.
- 7TB ZNS SSD, Western Digital Ultrastar DC ZN540
- Various 8GB emulated generic ZNS SSDs (for testing)
- Various 1GB emulated generic ZNS SSDs (for testing)
- 160.8 GB Optane SSD, model INTEL SSDPE21D280GA (a partition of the device)
- 447 GB SATA SSD, model INTEL SSDSC2kB48

**Virtual machine/Operating System configuration** The server running on the physical machine was running an older Operating System configuration than was required for the experiments. This was Ubuntu 20.04.2 LTS with kernel version 5.12. The server could not be updated, so a VM was used instead. The VM used QEMU 6.1.0 with KVM enabled, running Ubuntu 20.04.4 LTS with kernel version 5.17.1. The VM was running its image from the SATA SSD. The 280GB Optane SSD was made accessible with paravirtualisation and the ZNS SSD as made accessible with PCIe passthrough. The VM uses just 160.8GB of the the Optane SSD. The reason for this is that the VM is only allowed to use one partition that was configured on the host Linux machine, this also lead to the decision to use paravirtualisation and not passthrough. The VM was forced to just use one NUMA node with the help of numactl.

**Operating systems used**

The host runs Ubuntu 20.04.2 LTS with kernel version 5.12. The guest OS in the VM runs Ubuntu 20.04.4 LTS with kernel version 5.17.1. The software will not run on non GNU/Linux Operating systems.

### 8.1.5 Installation

Installation is divided into a few steps that we will each describe individually. Not every installation is necessary for every functionality. Further on, the installation instructions on the repository will always be more to date and in-depth. The installation instructions given here should serve as a guideline only.

**Install TropoDB**

TropoDB comes prebundled with SZD and SPDK, but it does not automatically install SPDK. To install SPDK and DPDK navigate to “implementation/rocksdb/third-party/SimpleZNSDevice/dependencies/SPDK”. Then follow the default install instructions for SPDK in the README file present in this directory. If you do not want to install globally, only built the project. At this point navigate back to the “implementation/rocksdb” directory present in the TropoDB repo and call:

```
$ cmake -DCMAKE_BUILD_TYPE=Release .
$ make -j $nprocs
```

At this point TropoDB should be in a functional state. When at any point configurations need to be changed, alter “implementation/rocksdb/db/zns_impl/config.h” and rebuild the project. This must also be done for some of the benchmarks.

**Install plain RocksDB with similar tests**

While it is possible to also test ZenFS and F2FS from TropoDB we do not advise this for benchmarks. Instead we advise using the exact same version of RocksDB that TropoDB originated from. This can be retrieved and installed with:

```
$ git clone https://github.com/facebook/rocksdb/
$ git checkout 062396af15ea9b295ea72d98cac32206f3dcf299
$ cd rocksdb
$ mkdir build
$ cd ./build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ make -j $nprocs
$ cd ..
$ cp -r <TROPODB_DIR>/implementation/rocksdb/zns_tests ./zns_tests
```

Notice how we copy a directory from TropoDB to RocksDB. This directory contains the benchmarking scripts we want to run.

**Install ZenFS for RocksDB**

After cloning the ZenFS version we have used, it is as simple as moving the cloned directory to "rocksdb/plugins/" and calling:

```
$ cd <ROCKSDB_DIR>
$ DEBUG_LEVEL=0 ROCKSDB_PLUGINS=zenfs make
$ cd plugin/zenfs/util
$ make -j $nprocs
```

Note that we no longer use CMake for ZenFS as this is not supported.

**Install F2FS-tools**

After cloning the F2FS-tools version we used. Do:

```
$ cd f2fs-tools
$ ./configure
$ make # DO NOT INSTALL!
```

Remember where f2fs-tools is installed and for every test that needs F2FS, use the binary present in "f2fs-tools/mkfs/mkfs.f2fs".

**Install zoned_bpftrace**

After ensuring that BPFtrace is installed and the kernel is properly configured, it is only required to clone the repository:

```
$ git clone https://github.com/indraneel-m/zoned_bpftrace
```

**Install fio for SPDK**

Installing fio for SPDK as done in the experiments is:

```
$ cd <FIO_DIR>
$ ./configure
$ make -j $nprocs
$ make install
$ cd TropoDB/implementation/rocksdb/third-party/SimpleZNSDevice/dependencies/SPDK
$ ./configure --with-fio=<absolute path to earlier installed fio repo>
$ make -j
```

Now whenever calling fio for SPDK, this needs to be done with:

```
LD_LIBRARY_PATH=<SPDK_DIR>/build/lib LD_PRELOAD=<SPDK_DIR>/build/fio/spdk_nvme fio
```

**Installing the SPDK reset test**
The custom reset test is present in the SZD repository, but needs an extra build command. This can be done with:

```
$ cd <SZD_DIR>
$ cmake -DSZD_TOOLS="reset_perf" .
$ make -j reset_perf
```

**Installing the plotting tools** After having cloned `https://github.com/Krien/ZNS_SPDK_Benchmarks`, move to the directory of ZNS_SPDK_Benchmarks. Then for either fio or TropoDB plots move to their directory. It should be enough to run (provided venv is used for Python3 environments):

```
$ python3 -m venv venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

### 8.1.6 Experiment Workflow

Many of the experiments conducted are different and require different tools to run. The results of the tests are also not automatically stored along with the rest of the results. These were moved to `https://github.com/Krien/ZNS_SPDK_Benchmarks` manually. How to reproduce the results of the individual benchmarks will be told here.

#### 8.1.6.1 TropoDB Experiments

All experiments for TropoDB are run by the "benchmark.sh" script in:
"TropoDB/implementation/rocksdb/zns_tests". The test given here contains derivatives of ZenFS tests and comes with various options. It also allows automatically setting up file systems and the ZNS environment for TropoDB. Further on, it allows deleting file systems and TropoDBs environment. For the individual test we will explain how to run them. Beware that all tests wipe the drive used. Further on, always use absolute paths for environment variables, this prevents pathing errors in some of the bash scripts.

**Running db_bench with TropoDB**

Running db_bench for TropoDB is not much different than running it for RocksDB. The only requirement is that the device to use is no longer binded to the kernel and that a flag is set in db_bench. This can be done with:

```
$ export PCI_ALLOWED=${PCI_ADDR} # PCI_ADDR of device to use
$ <SPDK_DIR>/scripts/setup.sh
$ <TROPODB_DIR>/db_bench --use_zns=true --db=$PCI_ADDR
```

**WAL append tests**

WAL append tests are run by first properly setting the config file. The following configurations need to be altered after each test:

- the directive WALPerfTest is set.
- The directive WAL_BUFFERED is unset.
- For ordered appends WAL_UNORDERED is unset. It is set for unordered appends.
- wal_iodepth is set to the proper queue depth (1 for ordered appends).

The configurations used can also be found in ‘implementation/rocksdb/zns_tests/configs‘. In the configs: ‘WAL_ordered.h‘, ‘WAL_unordered_depth2.h‘, ‘WAL_unordered_depth4.h‘, ‘WAL_unordered_depth16.h‘, ‘WAL_unordered_depth64.h‘ to be precise. Then the test is run with:

```
# Run next command to bind dev/ to SPDK. If already done skip
$ ./zns_tests/benchmark.sh setup znslsm $device_name
$ ./zns_tests/benchmark.sh run wal znslsm $PCI_ADDR $PCI_ADDR
$ ./zns_tests/benchmark.sh clean znslsm $PCI_ADDR
```

Output is present in “./output/wall_bench_$valsize”

**WAL recovery tests**

WAL recovery tests are run by first properly setting the config file. The following configurations need to be altered after each test:

- The directive WAL_BUFFERED is unset.
- For ordered appends WAL_UNORDERED is unset. It is set for unordered appends.
- wal_iodepth is set to the proper queue depth: 1 for ordered appends, 4 for ordered.

This configuration is also present in ‘implementation/rocksdb/zns_tests/configs/WAL_recovery_test.h‘. Then the test is run with:

```
# Run next command to bind dev/ to SPDK. If already done skip
$ ./zns_tests/benchmark.sh setup znslsm $device_name
$ ./zns_tests/benchmark.sh run wal_recover znslsm $PCI_ADDR $PCI_ADDR
$ ./zns_tests/benchmark.sh clean znslsm $PCI_ADDR
```

Output is present in “./output/wall_recover_bench_$valsize”

**TropoDB latency tests**

For this test we need to test ZenFS and F2FS as well, which requires some extra steps. First we will describe how to run the TropoDB test. Set the config file to the default config (‘implementation/rocksdb/db/zns_impl/default_config.h‘. Then do:

```
# Run next command to bind dev/ to SPDK. If already done skip
$ ./zns_tests/benchmark.sh setup znslsm $device_name
$ ./zns_tests/benchmark.sh run long znslsm $PCI_ADDR $PCI_ADDR
$ ./zns_tests/benchmark.sh clean znslsm $PCI_ADDR
```

Output is present in “./output/long_fillrandom_TropoDB”, “./output/long_filloverwrite_TropoDB” and “./output/long_readwhilewriting_TropoDB”.

For ZenFS and F2FS move to the RocksDB directory and do:

```
# Setup BPFTrace Reset script
export BLOCKCNT=<PATH_TO_ZONED_BPFTRACE>
# F2FS
$ ./zns_tests/benchmark.sh setup f2fs $mntpath $znsdev $nvmedev
$ ./zns_tests/benchmark.sh run long f2fs $mntpath $znsdev
$ ./zns_tests/benchmark.sh clean f2fs $mntpath $znsdev
# ZenFS
$ ./zns_tests/benchmark.sh setup zenfs $znsdev
$ ./zns_tests/benchmark.sh run long zenfs $znsdev $znsdev
$ ./zns_tests/benchmark.sh clean zenfs $znsdev $znsdev
```

Output is present in “./output/long_fillrandom_F2FS”, “./output/long_filloverwrite_F2FS”, “./output/long_readwhilewriting_F2FS”, “./output/long_fillrandom_ZenFS”, “./output/long_filloverwri
and “./output/long_readwhilewriting_ZenFS”. Additionally for each file there will also be an extra file ending with “_BPF”. This file contains the zone resets.

**TropoDB L0 size test**

This test can be run the same as the TropoDB latency tests, but only for TropoDB and it needs to be run three times. Each time with a different configuration. It requires setting "L0_zones" to 50, 100 and 200. Once for each test. The configurationss are also present in 'implementation/rocksdb/zns_tests/configs/' with the config files: 'L0_50_zones.h', 'default.h' and 'L0_200_zones.h'.

**TropoDB higher concurrency test**
This test can be run the same as the TropoDB latency tests, but only for TropoDB and it needs to be run three times. Each time with a different configuration. It requires setting "lower_concurrency' to 1, 2 and 3. Once for each test. The configurations are also present in 'implementation/rocksdb/zns_tests/configs/' with the config files: 'default.h', 'parallelism2.h' and 'parallelism3.h'.

#### 8.1.6.2 SPDK Fio Tests

The fio tests require little to setup and should be automatic. Move to the SPDK directory of `https://github.com/Krien/ZNS_SPDK_Benchmarks`. Then before running any test call:

```
$ export PCI_ALLOWED=${PCI_ADDR} # PCI_ADDR of device to use
$ <SPDK_DIR>/scripts/setup.sh
```

Then alter in all ".fio" files, the line "filename=trtype=PCIe traddr=0000.00.04.0 ns=2" to match the device used for the test. Then all tests can be run with:

```
# Please provide a path to the SPDK_DIR
$ SPDK_DIR=<SPDK_DIR> ./test_append.sh
$ LD_LIBRARY_PATH=<SPDK_DIR>/build/lib LD_PRELOAD=<SPDK_DIR>/build/fio/spdk_nvme \
fio fill.fio
$ SPDK_DIR=<SPDK_DIR> ./read_test.sh
```

Data will be written to "./data/writes/out_$bs" with one file for each block size and "./data/reads/out_read_$blksize" with one file for each block size.

#### 8.1.6.3 SPDK Custom Reset Tests

Running the reset test requires first filling the device. This can be done either with reset_perf or with fio. The results we have shown are done with fio. The test can then be run with:

```
$ export PCI_ALLOWED=${PCI_ADDR} # PCI_ADDR of device to use
$ <SPDK_DIR>/scripts/setup.sh
$ LD_LIBRARY_PATH=<SPDK_DIR>/build/lib LD_PRELOAD=<SPDK_DIR>/build/fio/spdk_nvme \
fio fill.fio # See the fio tests
$ cd TropoDB/implementation/rocksdb/third-party/SimpleZNSDevice
$ ./bin/reset_perf $PCI_ADDR 0 > reset_out
```

Alternatively the device can be filled with reset_perf first with:

```
$ export PCI_ALLOWED=${PCI_ADDR} # PCI_ADDR of device to use
$ <SPDK_DIR>/scripts/setup.sh
$ cd TropoDB/implementation/rocksdb/third-party/SimpleZNSDevice
$ ./bin/reset_perf $PCI_ADDR 1 > reset_out
```

## 8.2 Self Reflection

I have learned a lot from this thesis. Many of the theories and tools used to retrieve the end results were not familiar to me beforehand. This required a large investment in time to become more acquainted. I had to become more proficient with C, C++ and CMake to both understand already existing code and write a new implementation. Additionally, I needed to better understand GNU/Linux as a whole: user-space, filesystems, BPF, perf and the virtual filesystem itself among others. In order, to compete with the already existing alternatives, it was required to better understand the alternatives. This required investigating ZenFS, F2FS, SPDK, LevelDB and RocksDB.

However, most time was spent writing code. Most of the C code was written in the first month, but modifying the key-value store and C++ code of SZD extended took several months. Notably, it took a month to get an initial working implementation for a small simple emulated ZNS device (unlimited active zones, no zone capacities). Then another month to make it function on real hardware stably up to 100 GB and one additional month to get all benchmarking tooling online and reach 1TB with the key-value store. In general, setting up benchmarking and debugging large workloads took more time than adding new changes themselves. From this point onwards, therefore, minimal changes were added as testing each change took considerable work.

In the next month, we got the key-value store up to 2TB and managed to stabilise the performance of TropoDB. The time to complete benchmarks went from 39 hours for the fillrandom workload to about 8 hours. This month, we also conducted all raw performance tests for SPDK and ZNS.

The last month was spent on writing the paper, trying out multiple TropoDB configurations and plotting the results. The database was no longer altered significantly. I mention all of the steps explicitly, as they showcase how the work conducted varied wildly as time passed. In all months some implementation code had to be added, which is expected when building a key-value store, but interestingly most of it was spent on hunting for concurrency bugs, configuring benchmarks and creating and tweaking diagnostic tools for ZNS SPDK. As a result, I have been able to get an idea of most layers involved when creating a key-value store. From measuring the performance of storage, to configuring the storage, to tweaking the LSM-tree itself, to creating benchmarking tooling and tests and reporting the results.

The result took a lot of hard work, but there is a lot learned. Especially things that I would do differently. For example, first benchmarking the raw performance instead of creating a key-value store first or investigating if the existing approach taken properly fits the use case (LevelDBs design). Another important lesson is scope. As a lot needs to be done to make a key-value store (especially without any pre-made layers in between) and get it stable for large I/O. The result will most likely not reach what is maximally achievable, unless a considerable investment in time was made, which was not enough for the time of the thesis. The result is that we have done a lot, but it is hard to make a solid claim. Something which is easier to do, when only a few parts can be investigated (which was not possible with the design approach taken).