



@Large Research
Massivizing Computer Systems



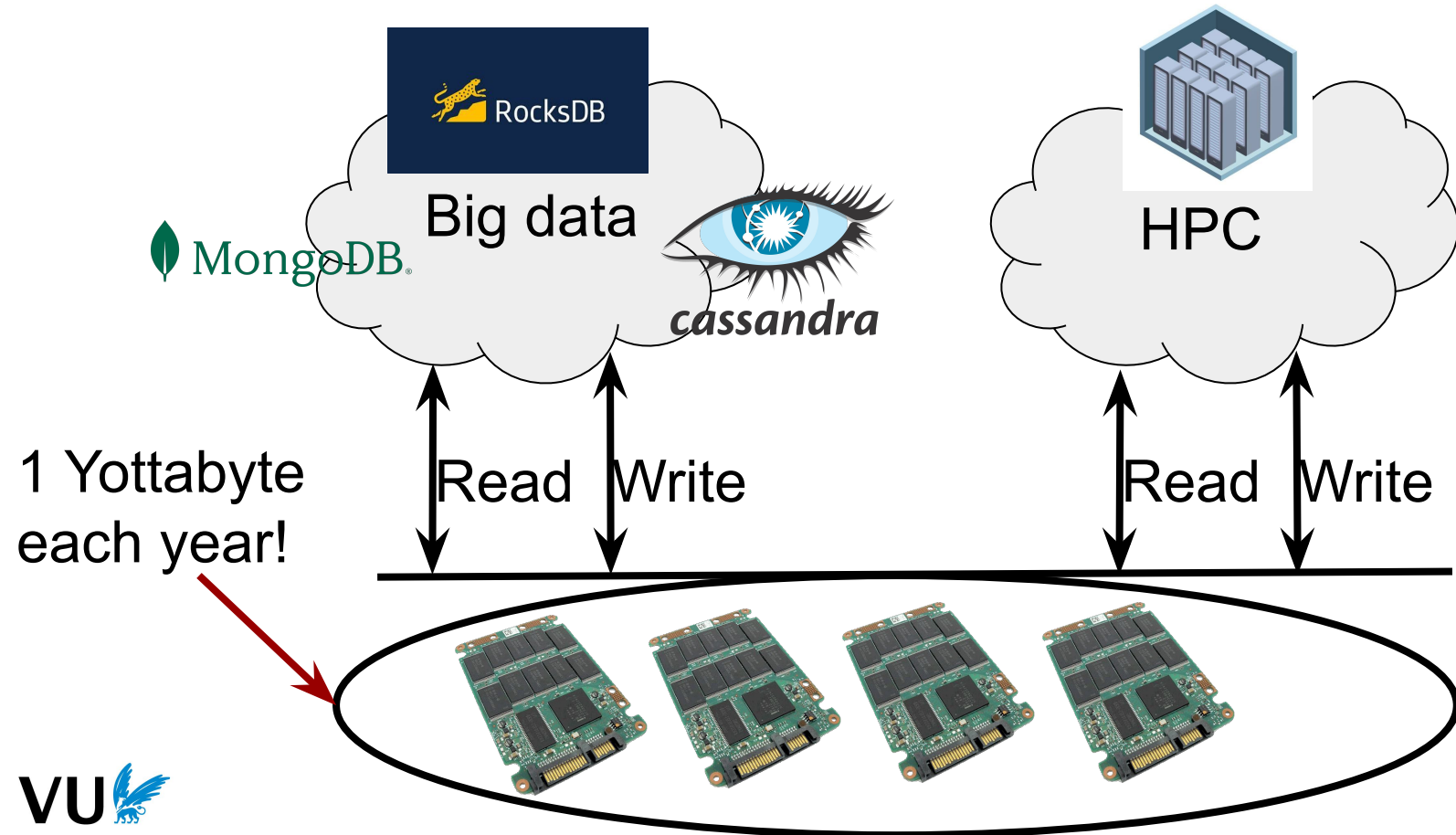
VRIJE
UNIVERSITEIT
AMSTERDAM

ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends

Krijn Doekemeijer, Zebin Ren, Nick Tehrany, and Animesh Trivedi

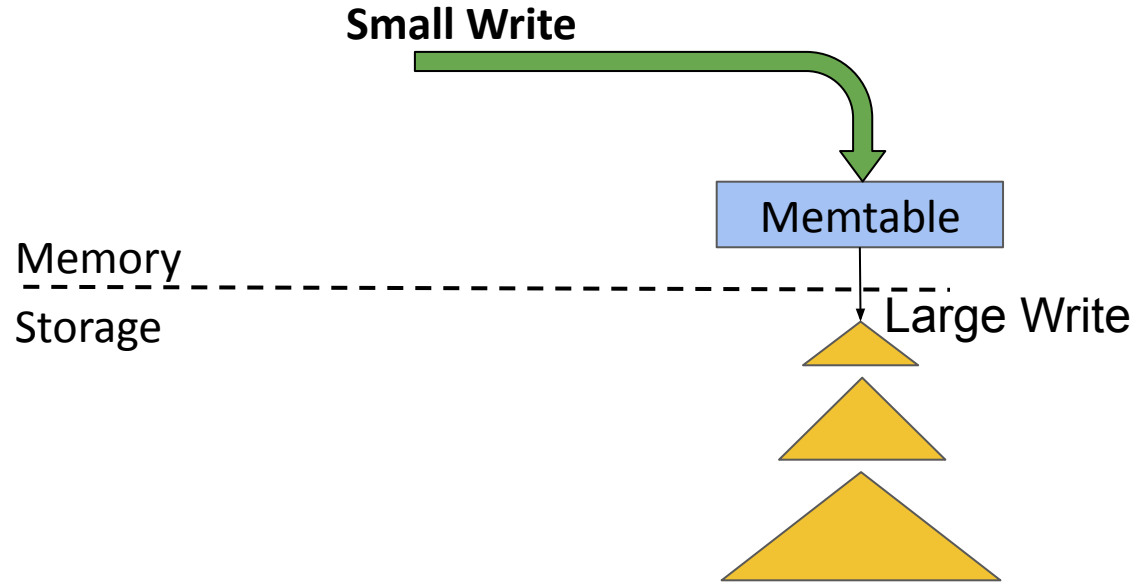
<https://krien.github.io/>

The amount of data is ever-increasing



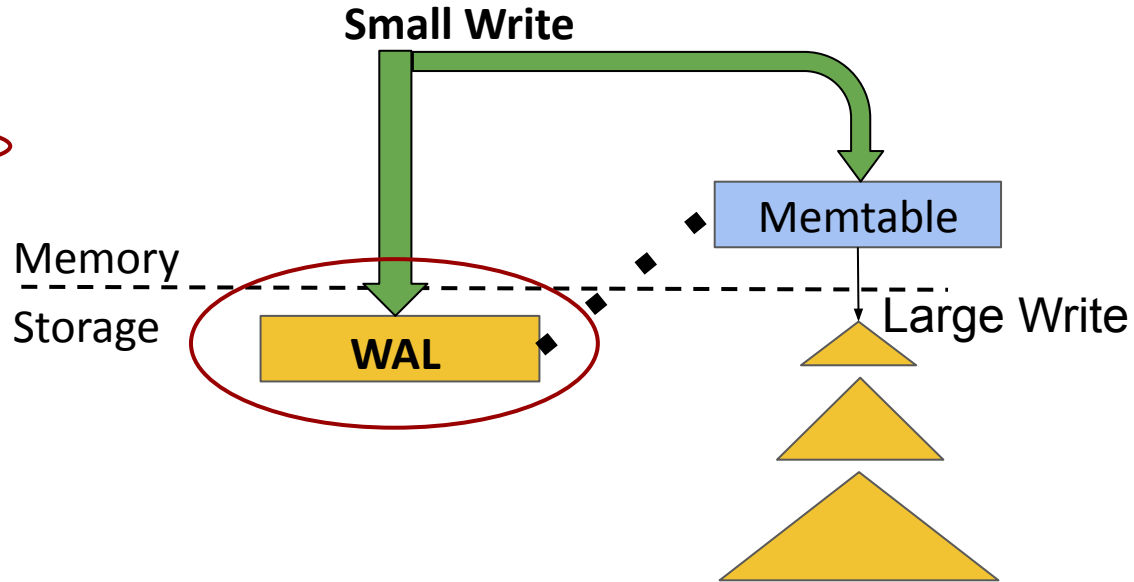
LSM-tree KV-stores

- First write to **ephemeral memory**

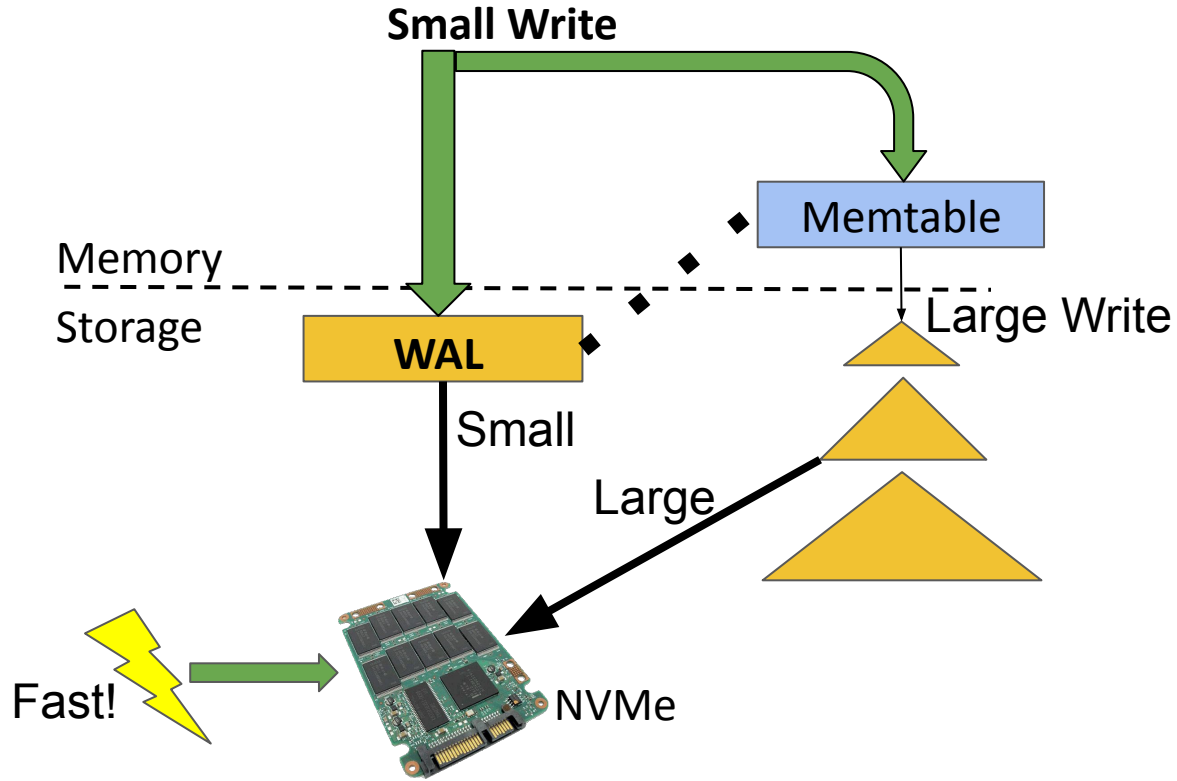


LSM-tree KV-stores

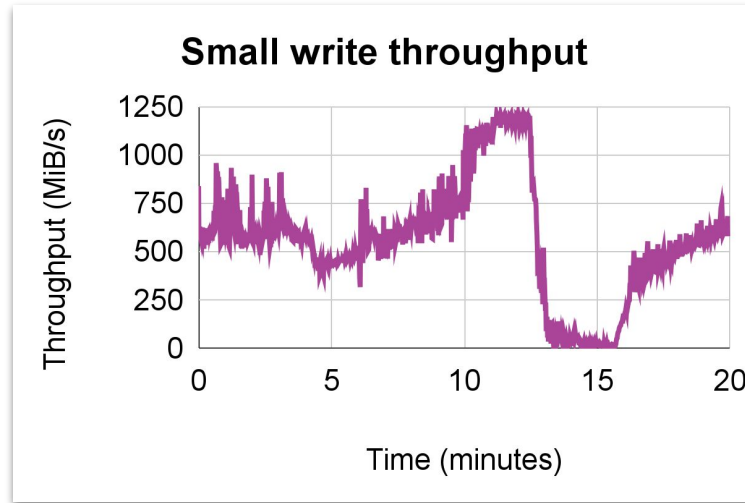
- First write to **ephemeral memory**
- Backup writes to the **WAL**



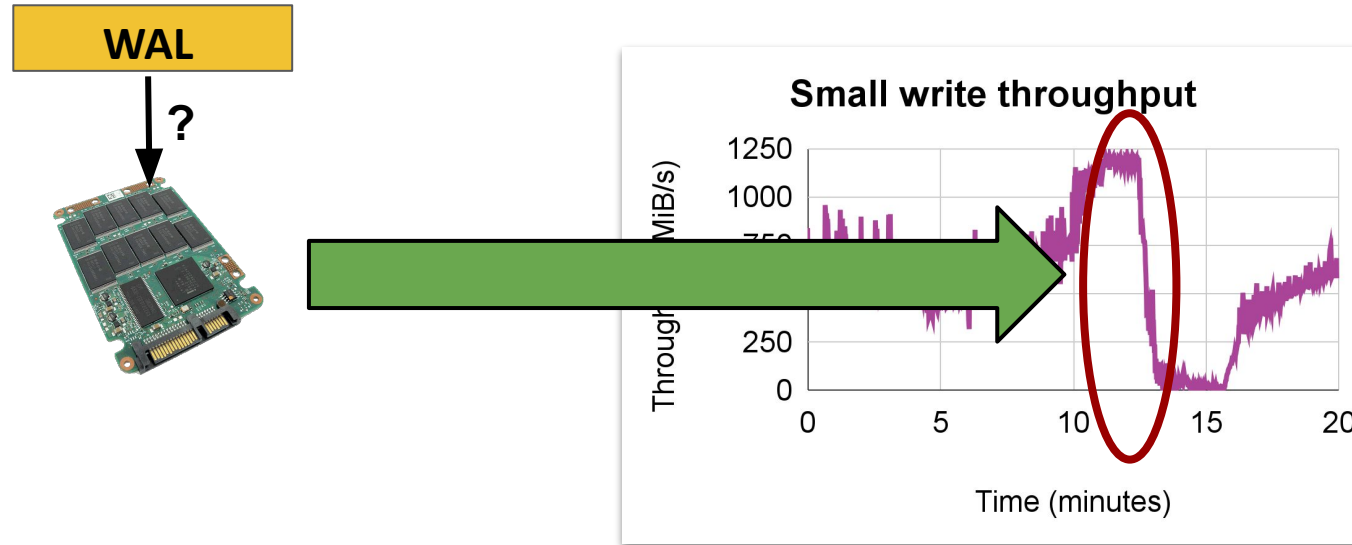
LSM-trees use fast NVMe flash



LSM-trees use fast, but unstable NVMe flash



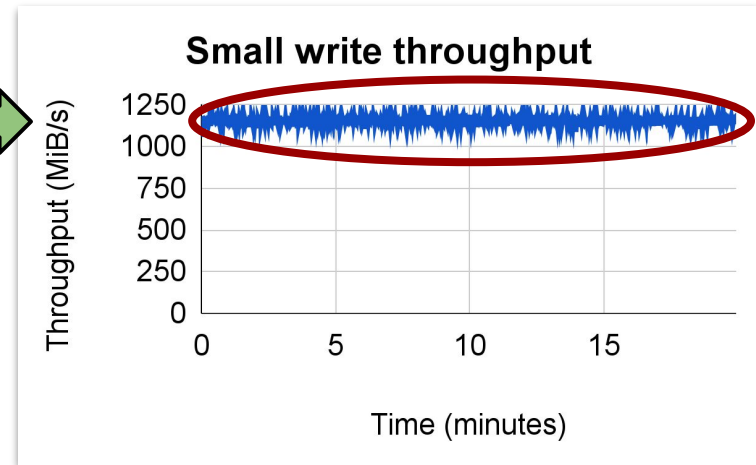
LSM-trees use fast, but unstable NVMe flash



What about WAL's requirements? Is there another interface?

Meet NVMe Zoned Namespace (ZNS)

- A new NVMe standard
- Firmware
- Stable performance
- Savior of LSM-trees!

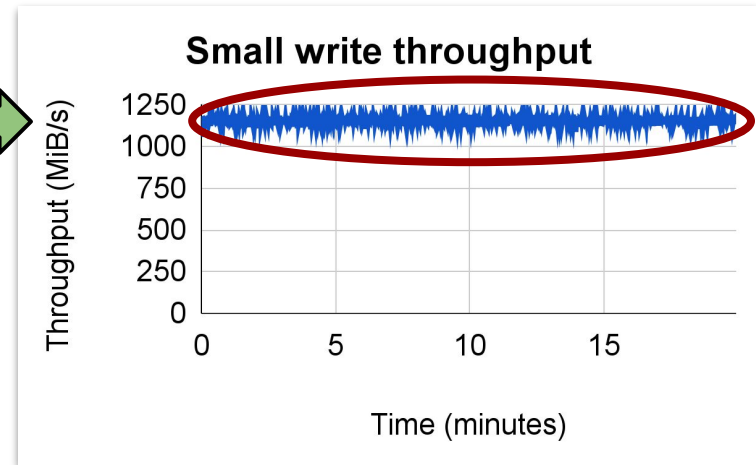


Meet NVMe Zoned Namespace (ZNS)

- A new NVMe standard
- Firmware
- Stable performance
- **Savior of LSM-trees?**

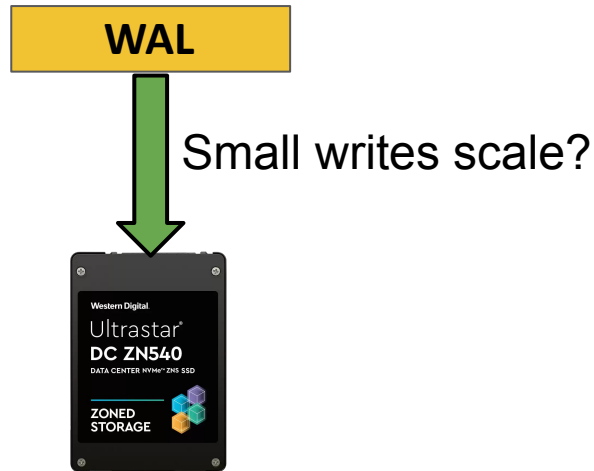
What is the catch?

- Different way to access storage...
- Small WAL writes do not scale



What we will discuss today

WALs for the **throughput stable** NVMe **ZNS** interface

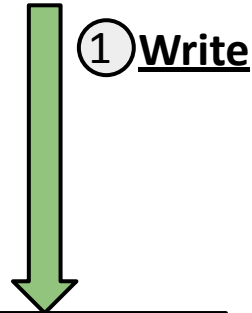


What operations does a WAL need?

An append-only log of all changes to KV-pairs with two key operations:

Write to WAL

Update(K1, V2)



Recover WAL

Update(K1, V1); ...

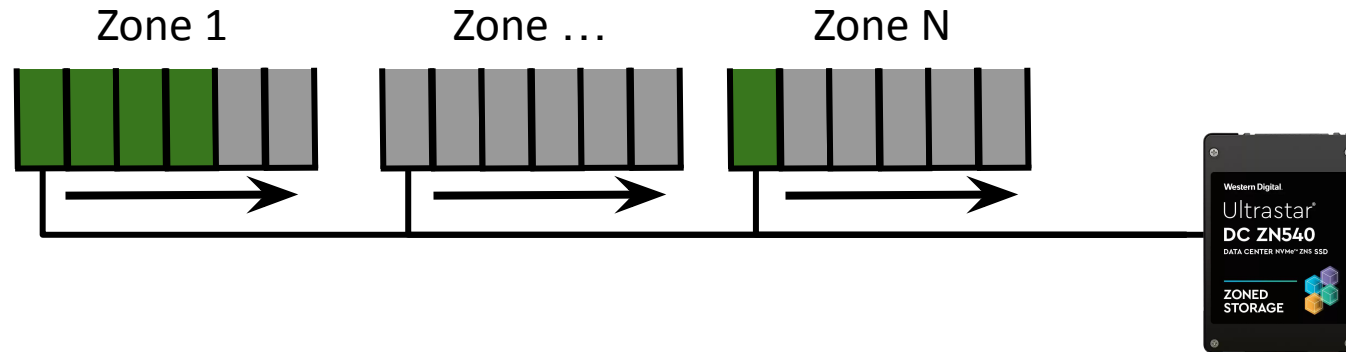
② Apply

The diagram shows a horizontal bar representing the WAL. The left part is green and contains the text 'U(K1, V1)' followed by an ellipsis '...'. The right part is green and contains the text 'U(K1, V2)'. A large green arrow labeled '① Read all' points upwards from the bar to the text 'Update(K1, V1); ...'. A second green arrow labeled '② Apply' points from the text to a blue box labeled 'Memtable' on the right. A black arrow at the bottom points to the right, indicating the direction of the log.

Memtable

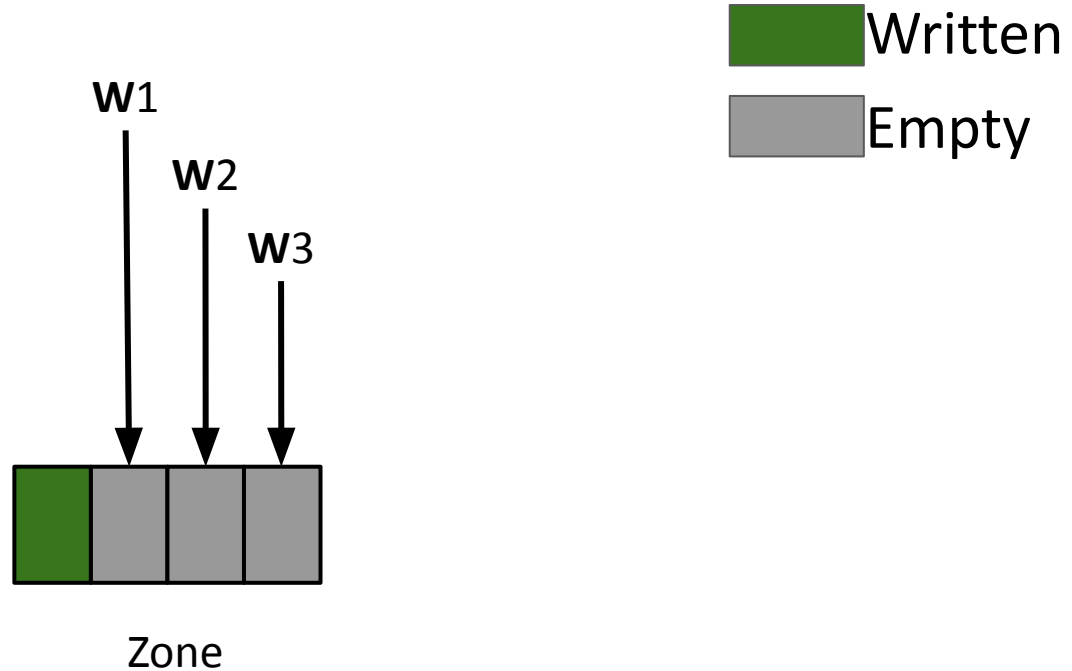
Background: Why ZNS writes do not scale

ZNS: storage as a series of **sequential write-only** zones



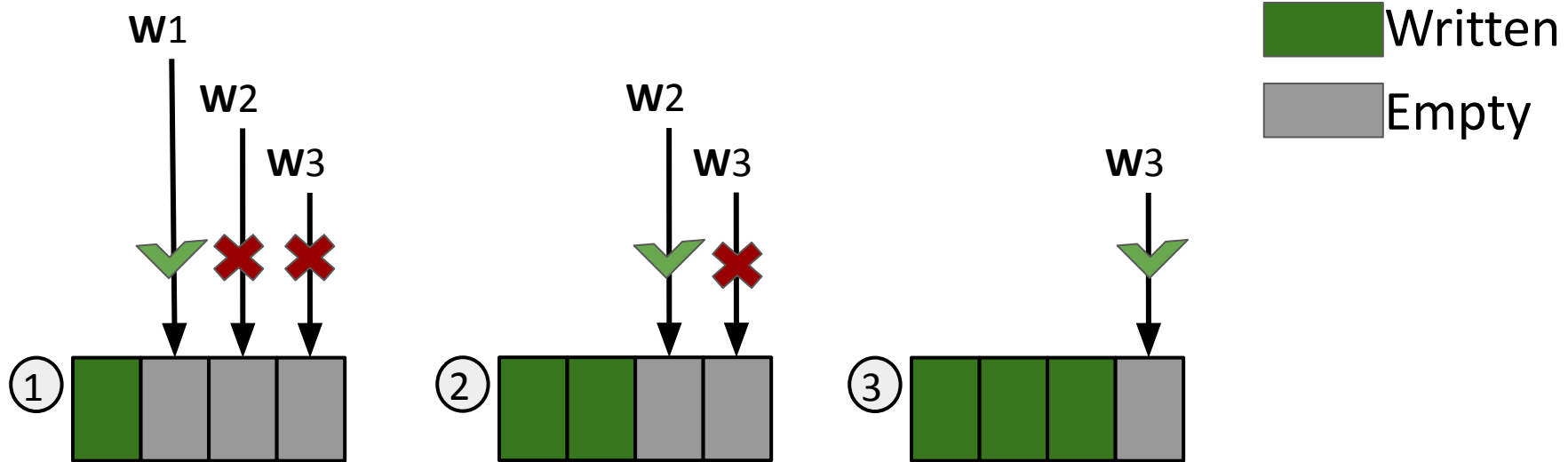
Background: Why ZNS writes do not scale

So how does ZNS deal with 3 consecutive **Writes**?

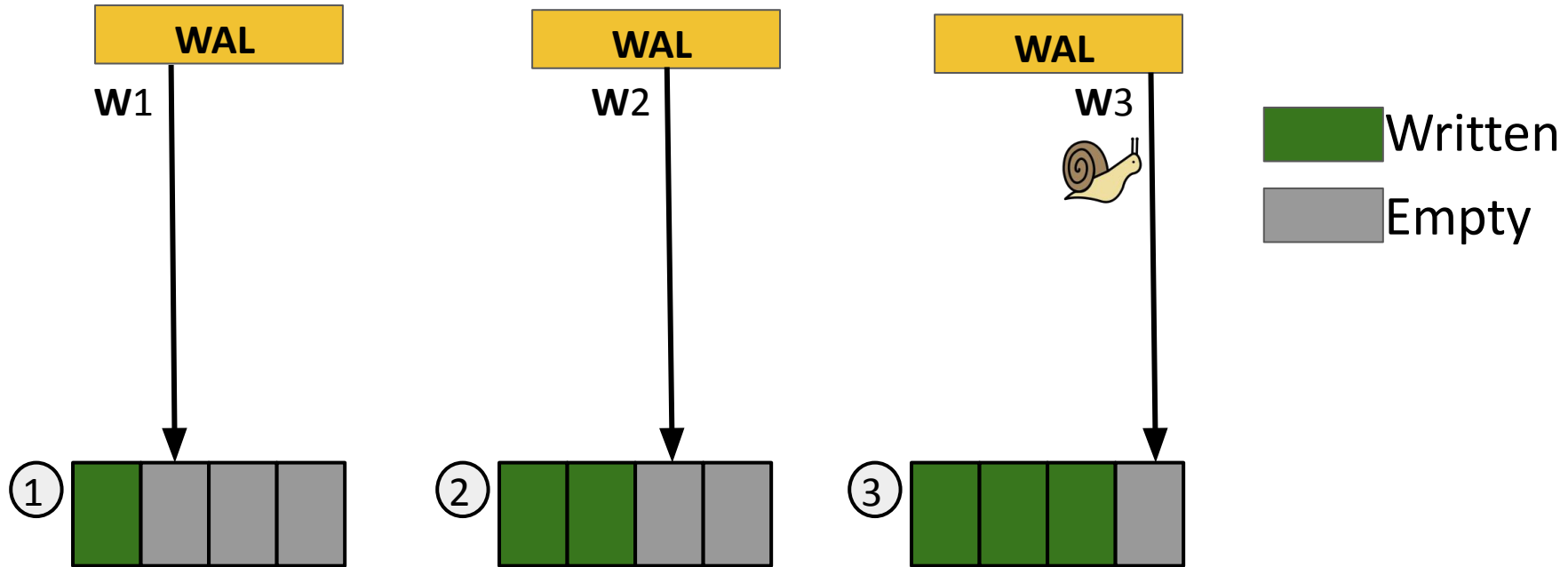


Background: Why ZNS writes do not scale

Subsequent **Writes** have to wait, **serializing I/O!**

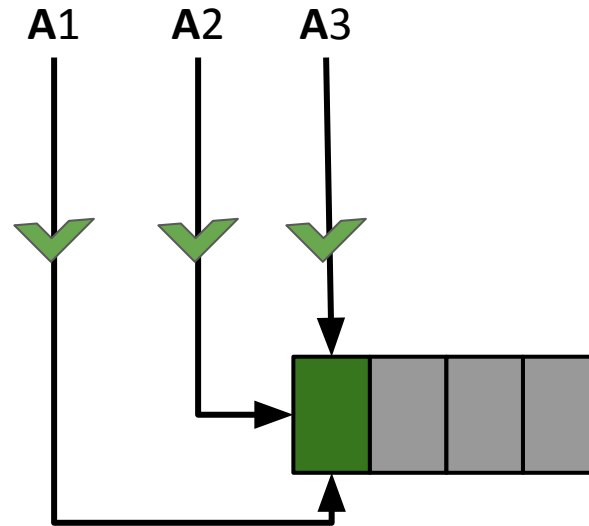


Thus WAL writes with ZNS writes do not scale!



Meet the ZNS Append operation

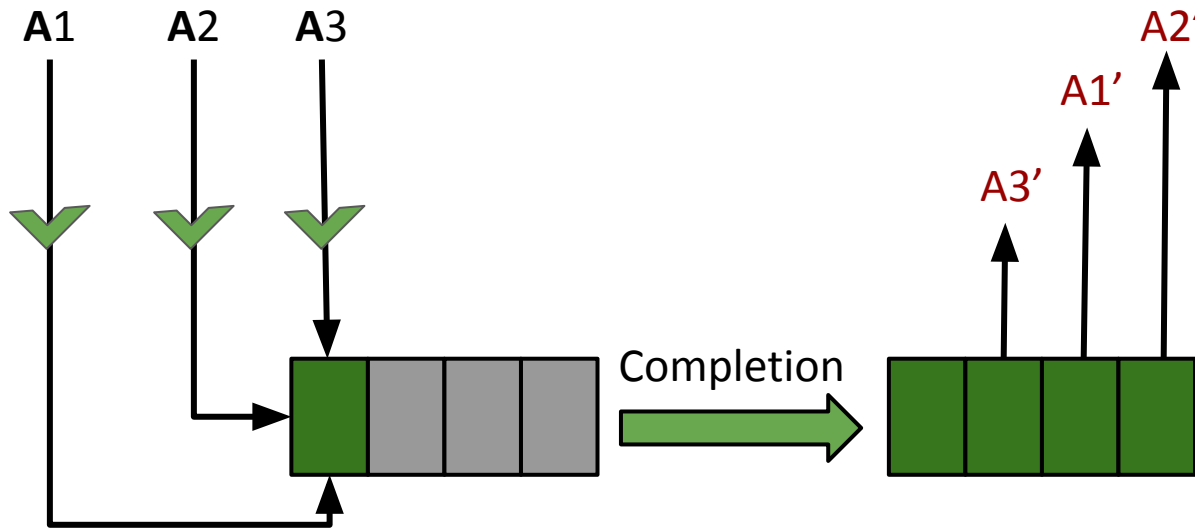
- ZNS has a scalable alternative for Writes, **Appends**
- How does ZNS deal with 3 consecutive **Appends**?



Zone appends are issued **concurrently** to a **zone**

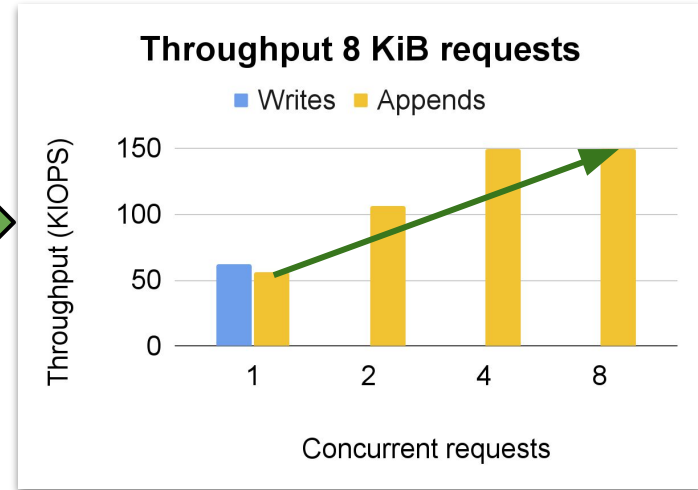
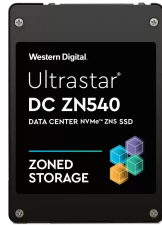
Meet the ZNS Append operation

- ZNS has a scalable alternative for Writes, **Appends**
- How does ZNS deal with 3 consecutive **Appends**?



Addresses returned on completion, but **can be anywhere and are ephemeral!**

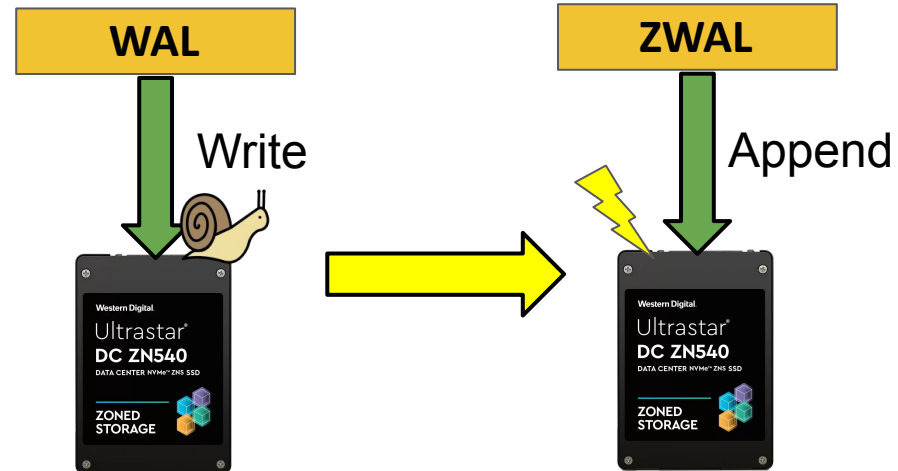
ZNS appends are fast!



Idea: **ZWALs**, use ZNS appends for the WAL to scale

Introducing ZWAL: WALs with appends

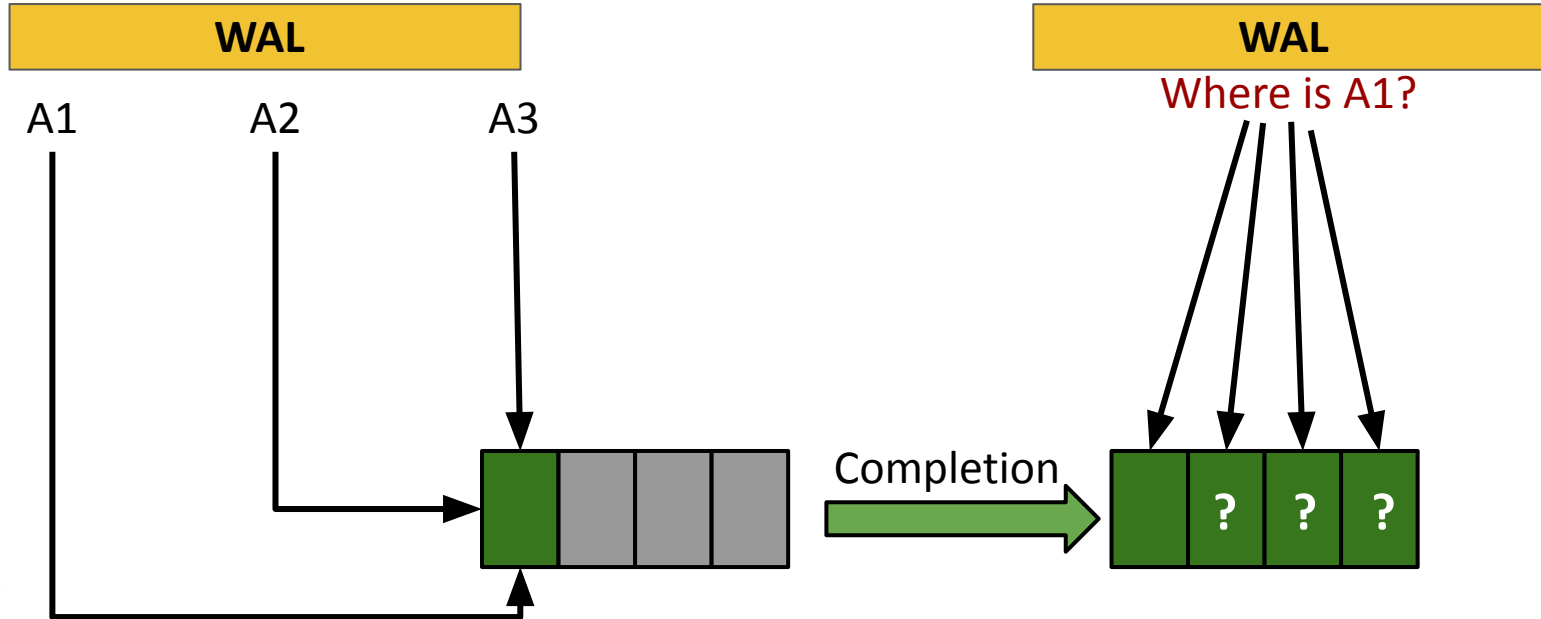
- What is the goal?
 - Get WAL writes to scale
- How?
 - Use ZNS appends
- What are the challenges?:
 1. Appends can be reordered
 2. Recovering data efficiently



Challenge-1: How to deal with reordering?

WAL can not use append as is:

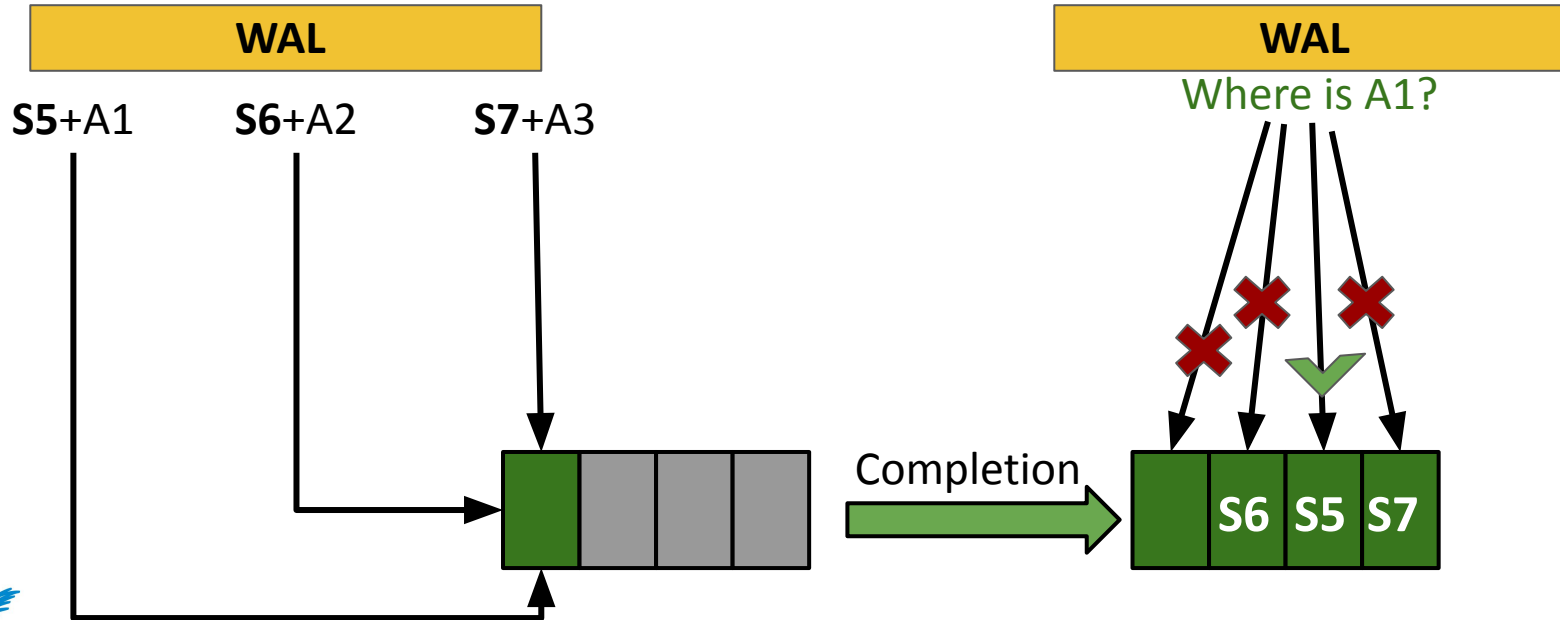
- WAL entries are reordered
- WAL entry addresses are ephemeral...



ZWAL's solution: add monotonic identifiers

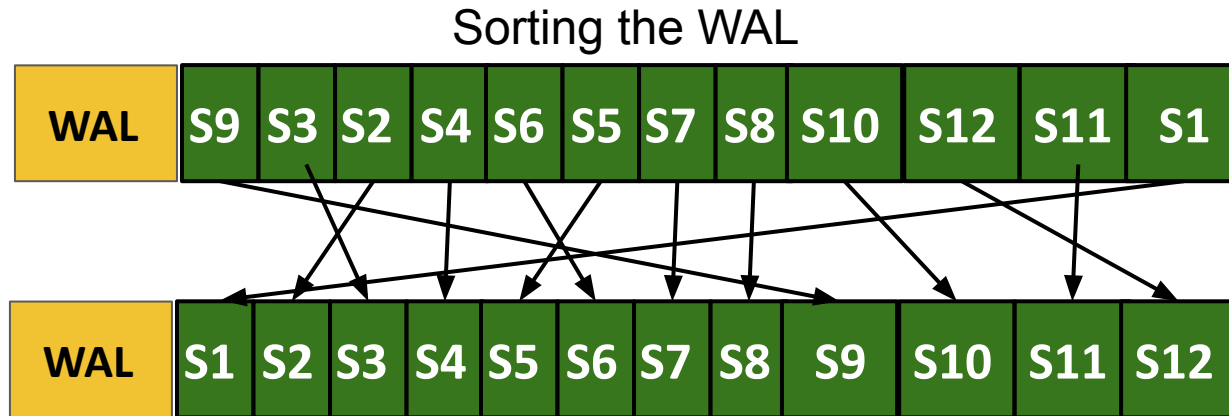
ZWAL solves the issue with:

- **Monotonic** identifier S for each WAL entry
- Infer ordering from identifiers



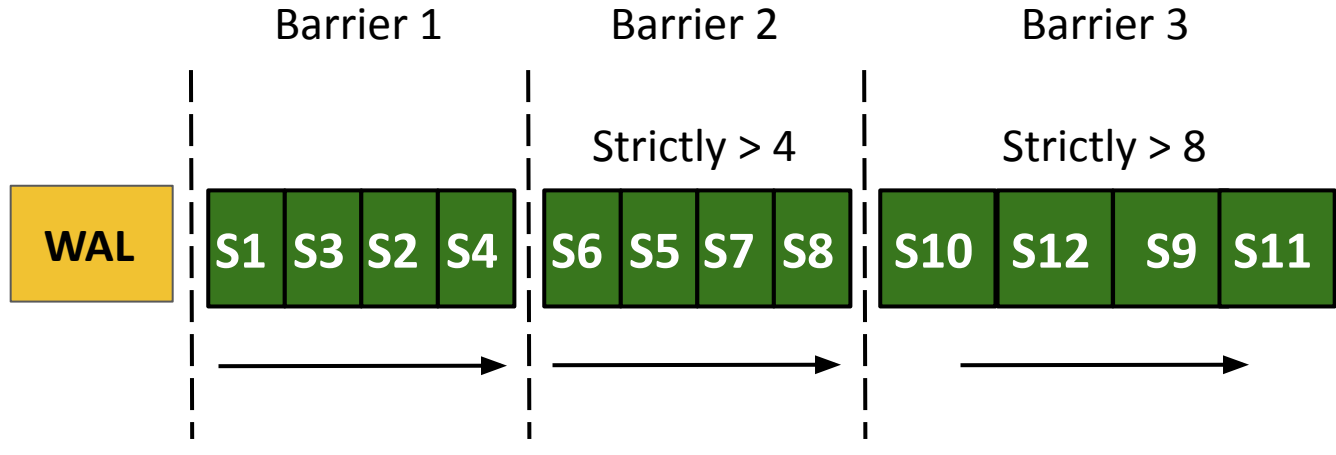
Challenge-2: expensive WAL recovery

- WAL recovered in order
- Location needed for each Read
- Scans the whole log for each Read



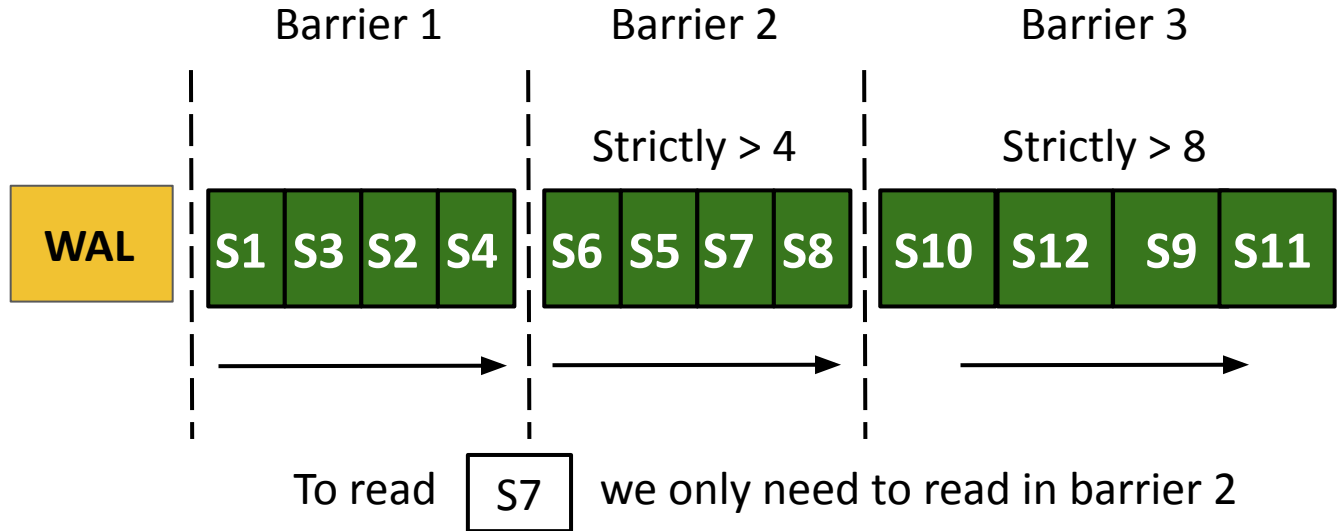
ZWAL's solution: Add barriers

- Bounds the number of Reads
- Sync **all Appends** after a barrier



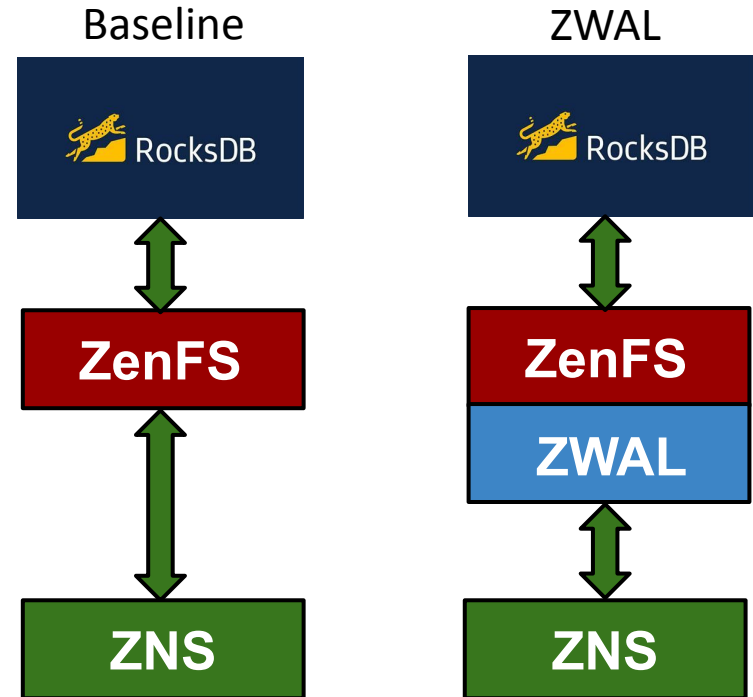
ZWAL's solution: Add barriers

- Bounds the number of Reads
- Sync **all Appends** after a barrier



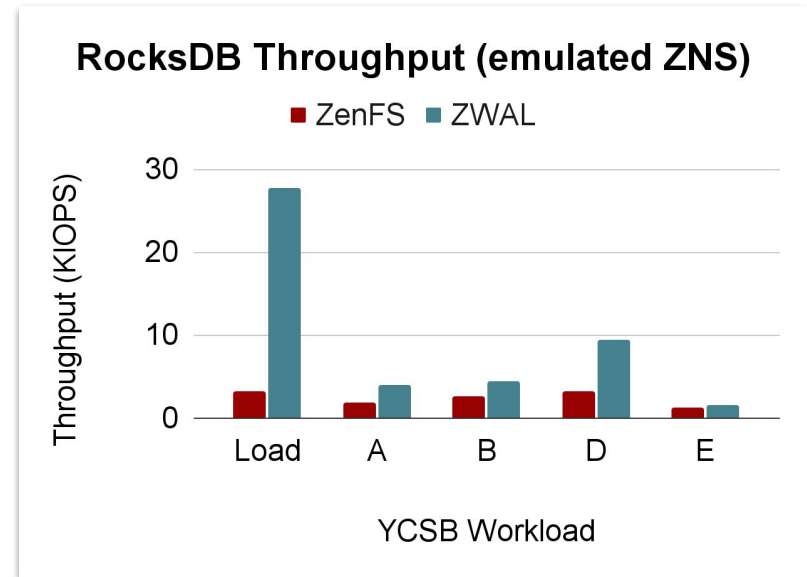
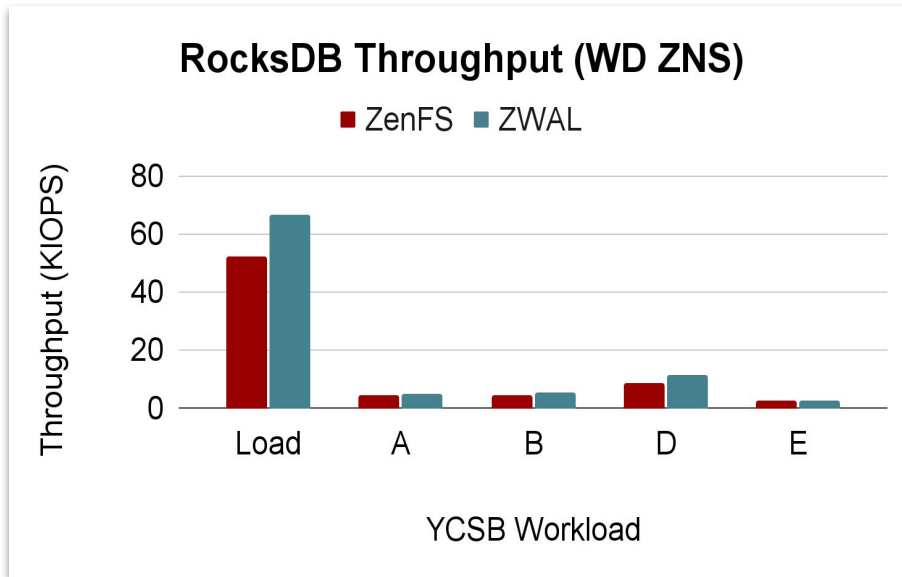
Evaluation setup

- State-of-the-art RocksDB + ZenFS
- Experiments:
 1. Evaluate write throughput with YCSB
 2. Evaluate WAL recovery
- Run on 2 ZNS SSDs
 - WD ZN540, 1.94TiB, 1.6GiB zones
 - ConfZNS emulator, 2 GiB zones



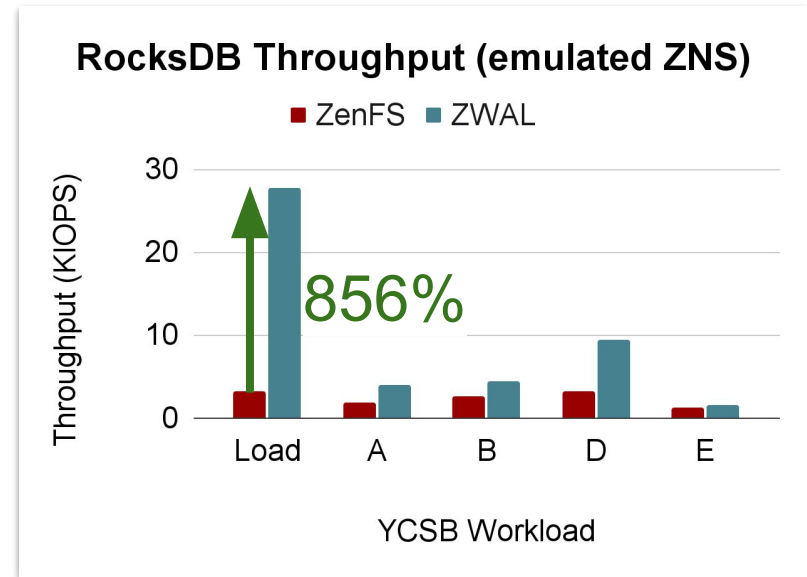
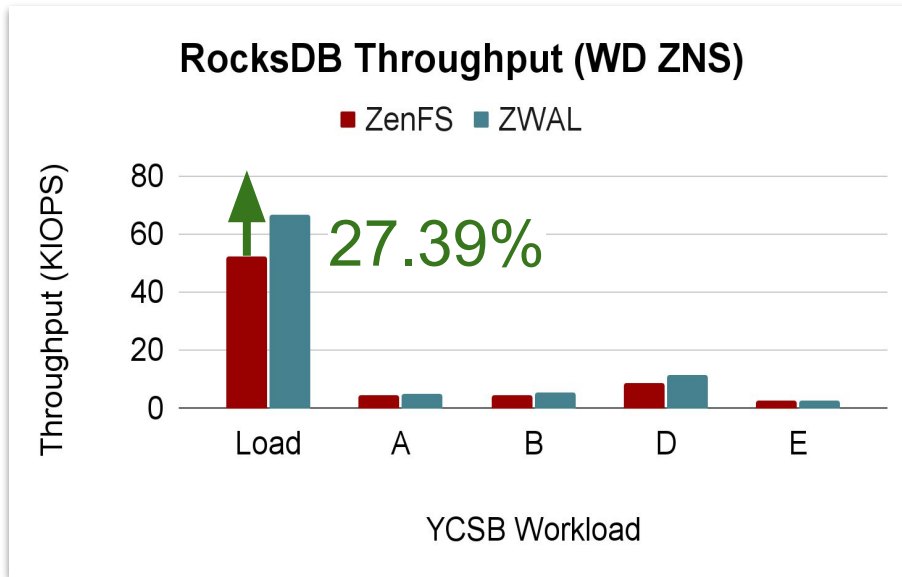
Experiment 1: ZWAL Write throughput

ZWAL Implemented in RocksDB + modified ZenFS



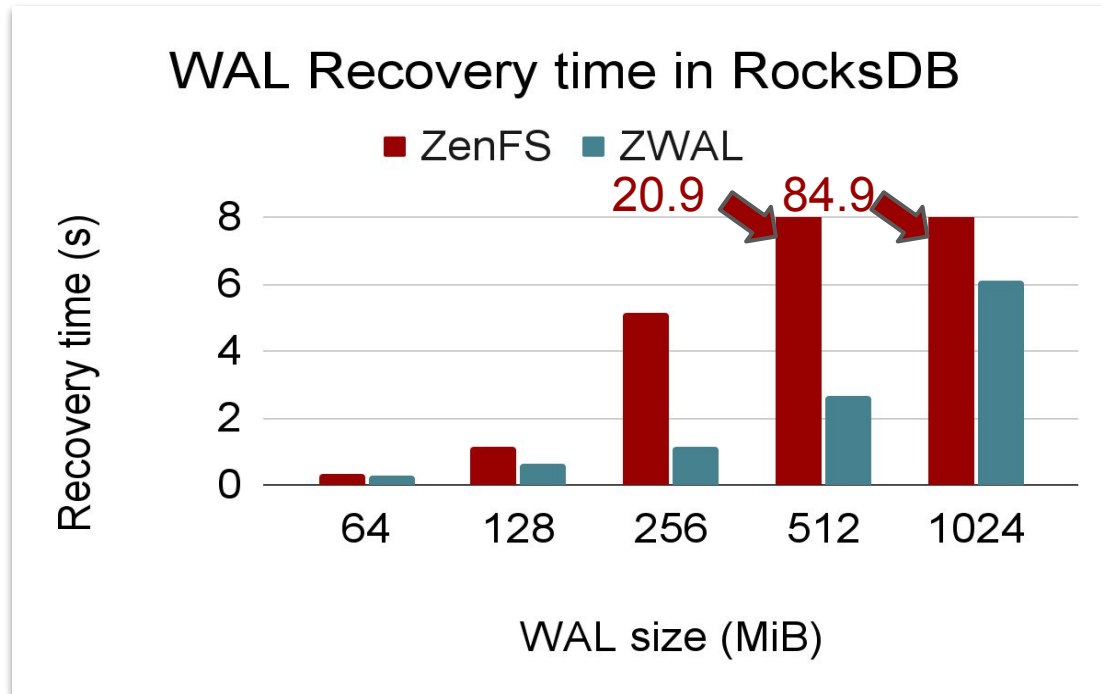
Experiment 1: ZWAL Write throughput

ZWAL Implemented in RocksDB + modified ZenFS



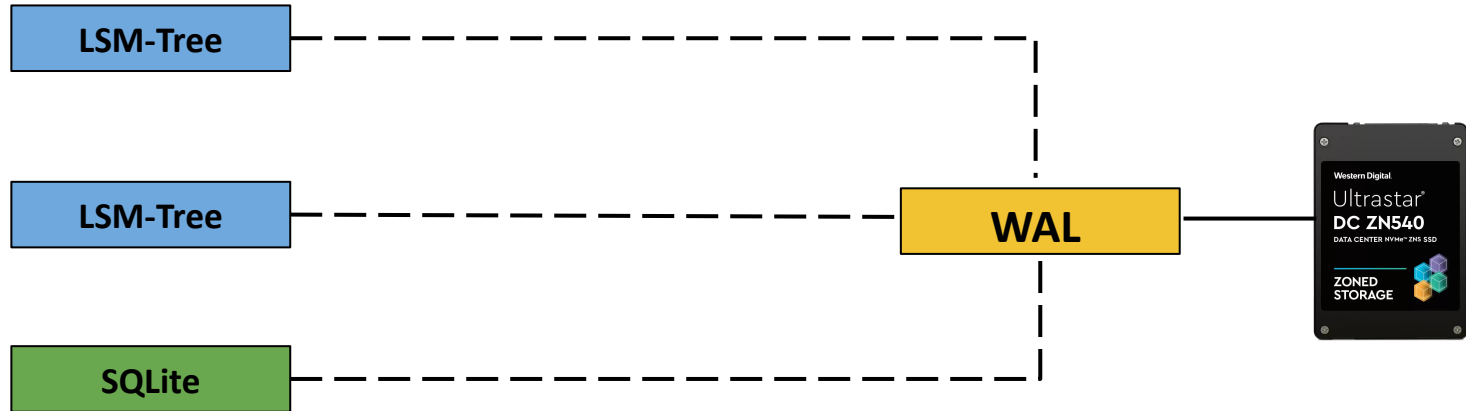
Experiment 2: Recovery overhead?

Contrary to expectations, **we reduced the required recovery time**



What next?

- Beyond ZenFS
- Use ZWALs in other databases (SQLite...)
- Use ZWALs in distributed settings
 - One SSD with WALs from multiple RocksDBs!
 - Disaggregated storage (NVMe-oF)



More details/results in the paper ...

ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends

Krijn Doekemeijer
Vrije Universiteit Amsterdam
The Netherlands

Nick Tehrani*
BlueOne Business Software LLC
Beverly Hills, CA, USA

Zebin Ren
Vrije Universiteit Amsterdam
The Netherlands

Animesh Trivedi
Vrije Universiteit Amsterdam
The Netherlands

Abstract

KV-stores are extensively used databases that require performance stability. Zoned Namespace (ZNS) is an emerging interface for flash storage devices that provides such stability. Due to their sequential write access patterns, LSM trees, ubiquitous data structures in KV stores, present a natural fit for the append-only ZNS interface. However, LSM-trees achieve limited write throughput on ZNS. This limitation is because the largest portion of LSM-tree writes are small writes for the write-ahead log (WAL) component of LSM-trees, and ZNS has limited performance for small write I/O. The ZNS-specific zone append operation presents a solution, enhancing the throughput of small sequential writes. Still, zone appends are challenging to utilize in WALs. The storage device is allowed to reorder the data of zone appends, which is not supported by WAL recovery. Therefore, we need to change the WAL design to support such reordering.

This paper introduces ZWALs, a new WAL design that uses zone appends to increase LSM-tree write throughput. They are resilient to reordering by adding identifiers to each append along with a novel recovery technique. We implement ZWALs in the state-of-the-art combination of RocksDB and ZenFS and report up to 8.56 times higher throughput on the YCSB benchmark. We open-source all our code at <https://github.com/stonet-research/zwal>.

CCS Concepts • Information systems → Storage management; Flash memory; • Software and its engineering → Secondary storage.

*Work done while the author was at the Vrije Universiteit Amsterdam.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions.acm.org.

CHEOPS 24, April 22, 2024, Athens, Greece
© 2024 ACM.
ACM ISBN 978-1-4503-XXXXX-0-18:00
<https://doi.org/XXXXXX.XXXXXX>

Keywords: Write-ahead log, Key-Value store, ZNS SSDs

ACM Reference Format:
Krijn Doekemeijer, Zebin Ren, Nick Tehrani, and Animesh Trivedi. 2024. ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends. In *Proceedings of The Fourth Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS 24)*, ACM, New York, NY, USA, 8 pages. <https://doi.org/XXXXXX.XXXXXX>

1 Introduction

Log-structured merge-tree (LSM-tree) based KV-stores are extensively used databases, with workloads ranging from graph processing to machine learning [5, 7, 12]. KV-stores store application data as KV-pairs with the PUT operation. The average KV-pair size issued by applications is small (e.g., 1 KiB) [5], resulting in many small writes to the LSM-tree. This paper focuses on optimizing LSM-tree write throughput for small writes on ZNS, an emerging storage interface.

We visualize the LSM-tree PUT operation in Fig. 1. Large sequential writes achieve higher throughput than small writes, therefore, LSM-trees buffer KV-pair updates in memory and periodically flush data to storage. The LSM-tree first store KV-pairs inside volatile memory to a size-bounded component known as the memtable. When this memtable is sufficiently large, the LSM-tree flushes the memtable to a tree-like structure on storage. To ensure no data is lost on shutdown, the LSM-tree writes PUT operations to an on-storage log known as the write-ahead log (WAL). The WAL maintains all KV-pair changes over time. When the KV-store restarts, the LSM-tree recovers its state using a process known as WAL recovery. WAL recovery reads all WAL data sequentially and (re)applies it to the memtable. WAL must be applied sequentially, as only the most recent change to a KV-pair is valid. The WAL is crucial for achieving high write throughput because each PUT writes to the WAL.

LSM-trees are typically deployed on fast and highly parallel NVMe flash SSDs. Flash storage performs better with sequential- than with random writes [17], precisely the access pattern of LSM-trees. However, with NVMe the SSD issues internal management operations that compete for storage resources with LSM-trees. This competition results in unstable throughput, which hinders achievable LSM-tree

CHEOPS 24, April 22, 2024, Athens, Greece

Doekemeijer, Ren, Tehrani, & Trivedi

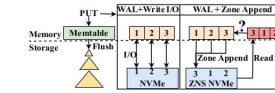


Figure 1. LSM-tree PUT operation with write I/O or zone append for the WAL.

throughput [3, 34]. Therefore, researchers and industry have proposed leveraging different interface(s) for flash SSDs [3, 4, 22]. One such interface is the recently standardized Zoned Namespaces (ZNS) interface [3]. ZNS presents the storage as sequential write-only regions known as zones and exposes the management operations to applications. ZNS delivers stable throughput by exposing these operations. Consequently, ZNS has led to several LSM-tree designs [21, 27, 31].

While ZNS achieves stable LSM-tree throughput, it leads to significant write throughput challenges for the WAL component of the LSM-tree. ZNS prohibits applications from issuing write I/Os concurrently to the same zone. ZNS prohibits this because (1) write I/Os need to be issued to sequential addresses of the zone (sequential write-only zones), and (2) SSDs are free to reorder I/O requests [30]. This restriction serializes writes to the WAL PUTs, limiting achievable throughput to the WAL as only 1 PUT can be processed concurrently [28].

To address write I/O's limited throughput, ZNS has introduced an alternative operation known as zone appends. Zone appends allow concurrent write operations to the same zone, saturating device parallelism and significantly increasing small write throughput [2]. High concurrency makes it a good alternative to use for WALs [2, 28, 32]. Nevertheless, we can not interchange zone appends for write I/O's without modifications. The main challenge is that zone appends are issued to a zone, not an address, and only return their address on completion. This address can be anywhere in a zone, and consequently, the SSD can reorder WAL data. Thus, the WAL needs to be resistant to data reordering. Therefore, current WAL designs on ZNS (such as RocksDB + ZenFS [36]) only use write I/O or only allow scaling zone appends by increasing threads [28]. We visualize the reordering challenge as “?” in Fig. 1.

This work proposes ZWALs, a zone append-friendly WAL for ZNS. ZWALs improve write throughput on ZNS and is resilient against data reordering. They achieve this feat by adding 64-bit atomically increasing sequence numbers to each PUT request. The sequence numbers specify the absolute ordering of data and are used to infer the order within the WAL. On recovery, the WAL reads all of its KV-pair changes and then sorts them back into their original order using the sequence number. After sorting, the LSM-tree applies the

changes in sequence. Considering that LSM-tree WALs are generally only recovered during database startup and WALs are small (e.g., 32 MiB), we consider tracing WAL read for better write throughput acceptable. To reduce the overhead of reordering and to prevent reading the entire WAL, we introduce the notion of WAL barriers. A ZWAL synchronizes all zone appends at a barrier. Barriers ensure that a read to the WAL only needs to read and sort between subsequent barriers, increasing WAL read performance.

We implement ZWALs in ZenFS, a state-of-the-art custom file system backend of RocksDB, and report that ZWAL leads to significantly higher write throughput than traditional WALs on commercially available ZNS SSDs, up to 33.02% higher throughput on the YCSB benchmark suite. Similarly, we repeat our experiments on the ComZNS [33] emulator and report that with high internal parallelism, ZWAL can deliver up to 8.56 times higher write throughput on YCSB.

In this paper, we make the following key contributions:

1. We characterize the performance of the zone append operation and explain how we can leverage them for WALs.
2. We design and implement ZWALs—a new WAL design for ZNS zone appends.
3. We evaluate ZWALs on both the micro- and macrolevel.
4. We open-source the code of our ZWAL implementation at <https://github.com/stonet-research/zwal>.

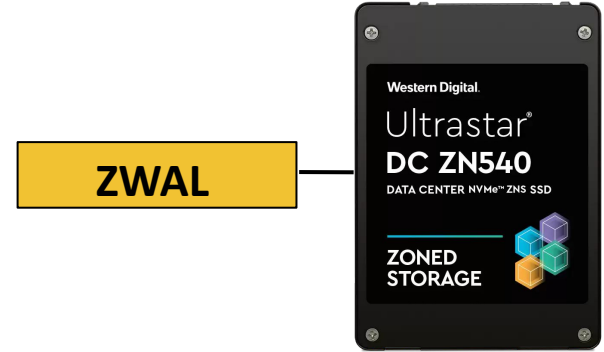
2 Motivation: Why use zone appends?

Below, we demonstrate a performance characterization of zone appends. The design of ZWAL relies on high write concurrency and throughput for small writes. In this section, we show how zone appends lead to higher write concurrency and throughput than write I/O to motivate their use-case in WALs. In our benchmarking, we use *fio* [19] (v3.32) as a workload generator. We use the *io_uring* storage interface with NVMe passthrough [20] since the Linux block layer does not support zone appends and follow recommended performance optimizations [10]. We modify *fio* to support zone appends for passthrough (~10 LOC). We show the rest of our benchmarking setup in Tab. 1.

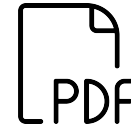
We evaluate the concurrency of zone appends by increasing the queue depth (QD)—the maximum number of concurrent zone appends—and measure throughput in I/O operations per second (IOPS). Since ZNS prohibits multiple write I/Os to the same zone, we only evaluate write I/O at QD 1. We issue all requests at a granularity of 8 KiB, which we evaluate as the optimal request size (i.e., lowest request latencies). Fig. 2a shows the throughput of zone appends in IOPS (y-axis, higher is better) with increasing QD (x-axis). Zone appends scale up to a QD of 4, beyond which the device's peak bandwidth is reached according to the device's specification sheet. We observe that write throughput is up to 2.41 times higher for zone appends (at high QD) than for

Take-away message

- LSM-trees use unstable NVMe flash storage
 - ZNS allows for stable performance!
- LSM-tree WALs do not scale with ZNS Writes
 - Use ZNS Appends instead!
- We introduce **ZWALs**, Append-friendly WALs for ZNS

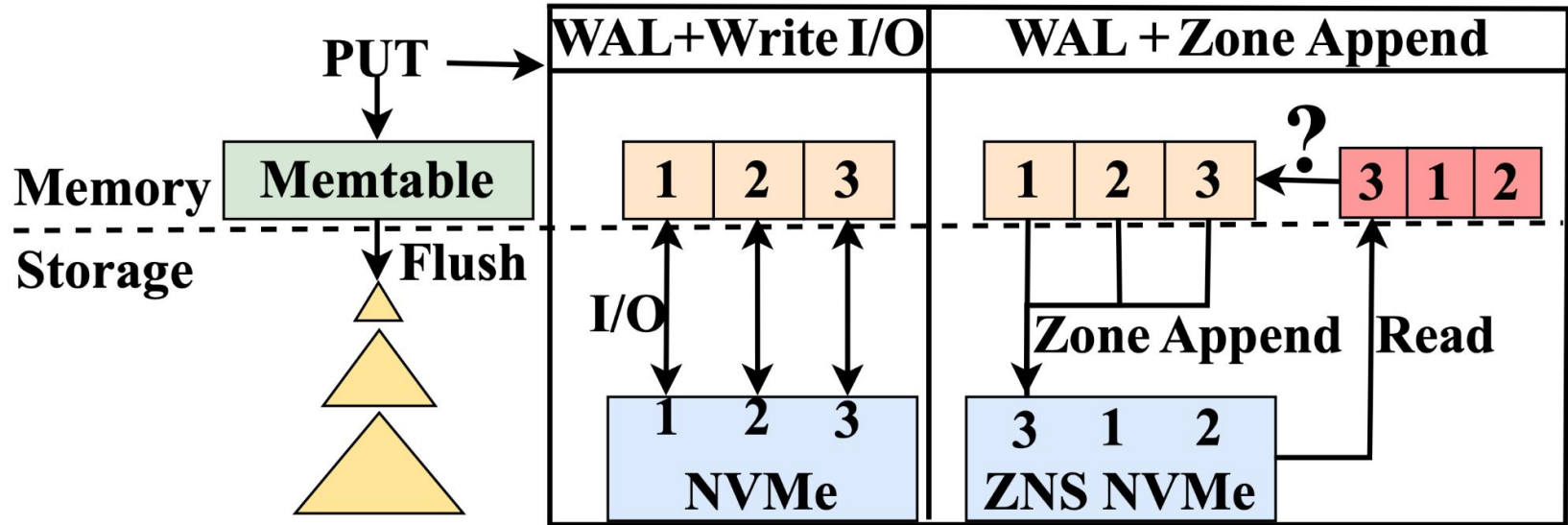


Paper: <https://atlarge-research.com/pdfs/2024-zns-wal.pdf>
Source code: <https://github.com/stonet-research/zwal>

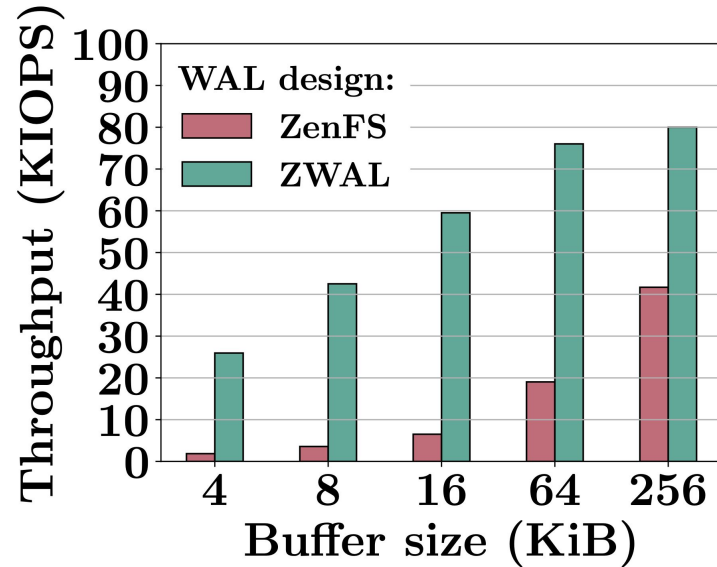
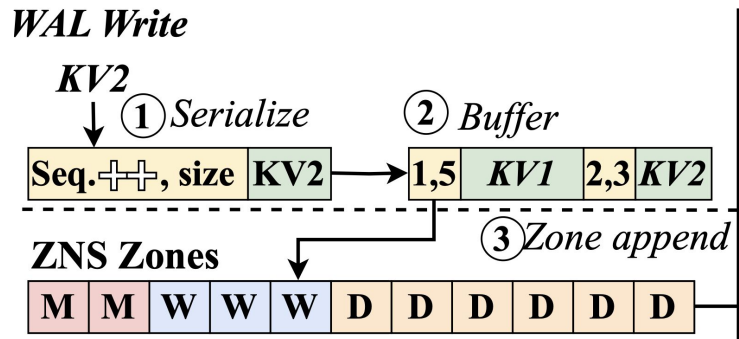


Backup slides

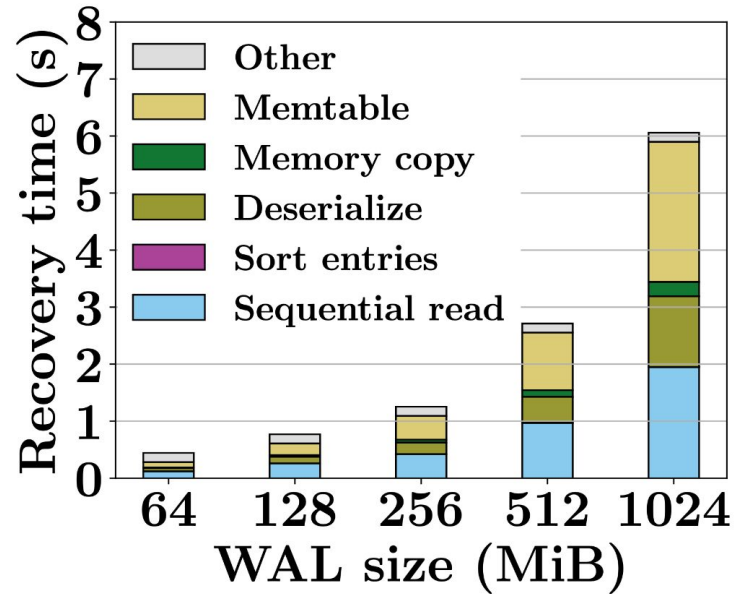
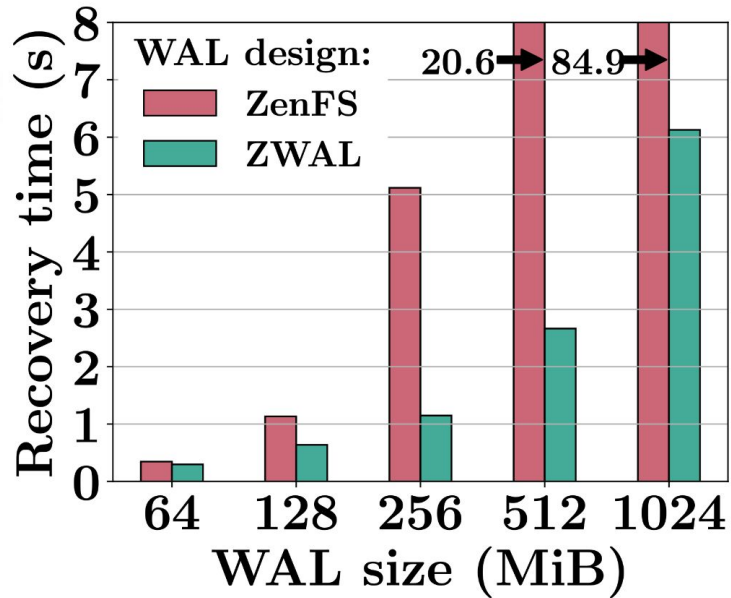
WAL versus ZWAL



ZWAL buffering



ZWAL Recovery breakdown



Background: NVMe interface

