HyperDB: a Novel Key Value Store for Reducing Background Traffic in Heterogeneous SSD Storage

Ruisong Zhou, Yuzhan Zhang, Chunhua Li, Ke Zhou, Peng Wang, Gong Zhang, Ji Zhang, Guangyu Zhang
Huazhong University of Science and Technology & Huawei Technologies Co., Ltd.

International Conference in Parallel Processing (ICPP'24)
August 12–15, 2024, Gotland, Sweden

2025. 01. 15

Presented by Charles D. Jaranilla

cdjaranilla@dankook.ac.kr

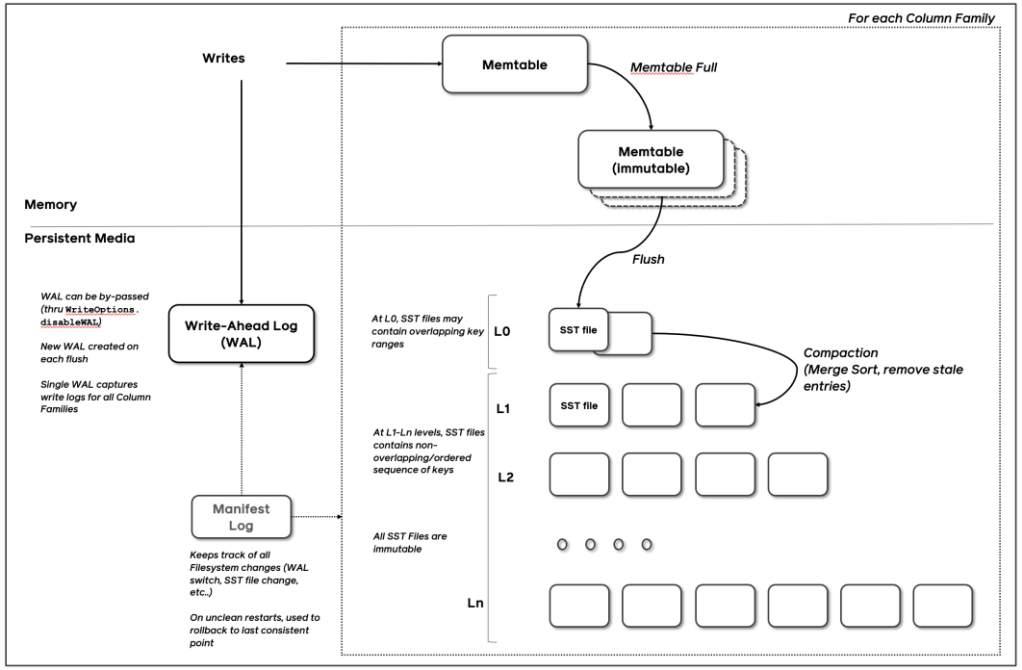
DANKOOK UNIVERSITY

Dankook University
System Software Laboratory

Log-Structured Merge-tree

The LSM-tree is a multi-level data structure primarily designed for block-based storage devices to handle write-intensive workloads.

It is primarily adopted by Key-Value Stores as their underlying storage engine.



Images taken from

https://github.com/facebook/rocksdb/wiki/RocksDB-Overview

https://dbdb.io/db/rocksdb

Multi-tier storage

NVMe

Performance tier

SATA

Capacity tier

Due to the significantly higher cost of NVMe storage in comparison to SATA SSDs,

the development of cost-effective multi-tier storage systems has emerged as a preferable choice.

Images taken from

https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/ / https://smicro.eu/intel-dc-s4610-960gb-2-5-ssd-tlc-sata-iii-oem-ssdsc2kg960g801-1?srsltid=AfmBOopW1uGnwutGCfXkcgPGPgo6Mhm-cFjnxN8pE2NCyqM7SQ41rcdd

https://www.veritis.com/wp-content/uploads/2024/08/AWS-Storage-Gateway-A-Bridge-to-Hybrid-Cloud-Storage.webp/ / https://cloudkul.com/blog/google-cloud-storage/ / https://www.linkedin.com/posts/huaweiitproducts_oceanstor-hybridflash-unleashdatapower-activity-7140905745457283073-m82b

Deployment of LSM-trees in Heterogeneous Storage

RocksDB represents the embedding architecture, which combines multiple storage devices through the **`db_path`**

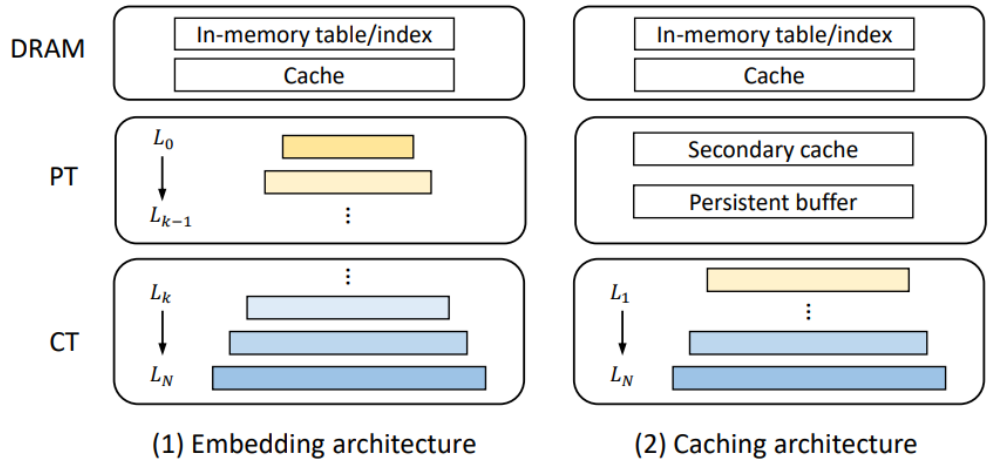


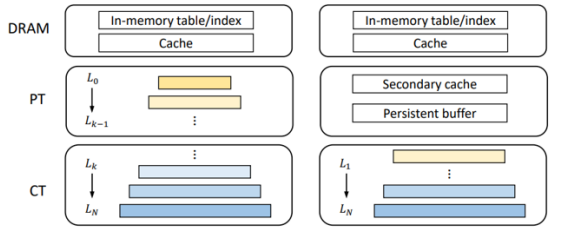
Figure 1: Two architectures for deploying LSM-trees in multi-tier storage, including embedding the top levels in the performance tier (a) and adding an extra cache tier (b).

PrismDB represents the caching architecture deploying a slab-like layout on NVMe storage and migrating objects to the capacity tier through compaction.

The **embedding method** puts the recently written data into a performance device in the upper layer, such as the top levels of an LSM-tree.

The **caching method** copies the recently accessed data from a large-capacity device and temporarily stores newly written data in the upper layer.

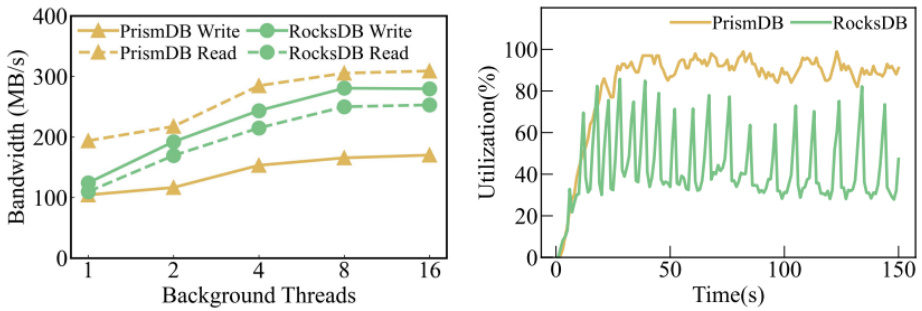
Challenges and Motivations



Goal: maximize end-to-end performance by fully utilizing the characteristics of each tier.

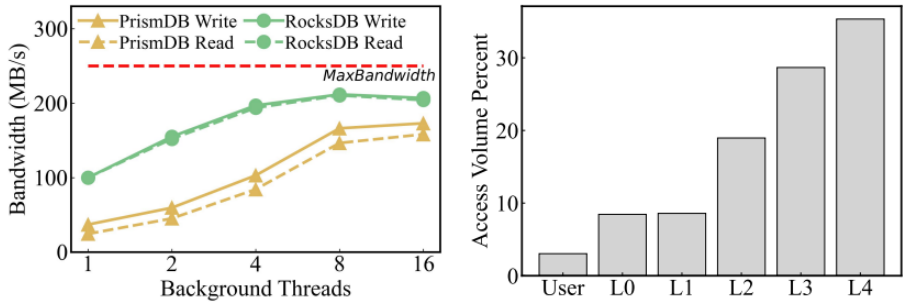
Motivation:

- Resource Utilization
  - Excessive bandwidth utilization of NVMe storage during writes in PrismDB.
  - RocksDB's utilization rate because of the size of LSM-tree levels.
- Compaction Overhead
  - RocksDB quickly reaches a bottleneck in bandwidth consumption at the capacity tier.



(a) bandwidth utilization (b) capacity utilization

Figure 2: The utilization of bandwidth (a) and capacity (b) in a multi-tier storage system for RocksDB and PrismDB where NVMe SSD serves as the performance tier and SATA SSD serves as the capacity tier.



(a) Bandwidth used by compaction (b) Compaction data breakdown

Figure 3: The compaction overhead of the LSM-tree in the capacity tier. The left figure displays the average bandwidth consumed by compaction threads while the right figure provides a detailed breakdown analysis of compaction I/O volume.

Design: Overview

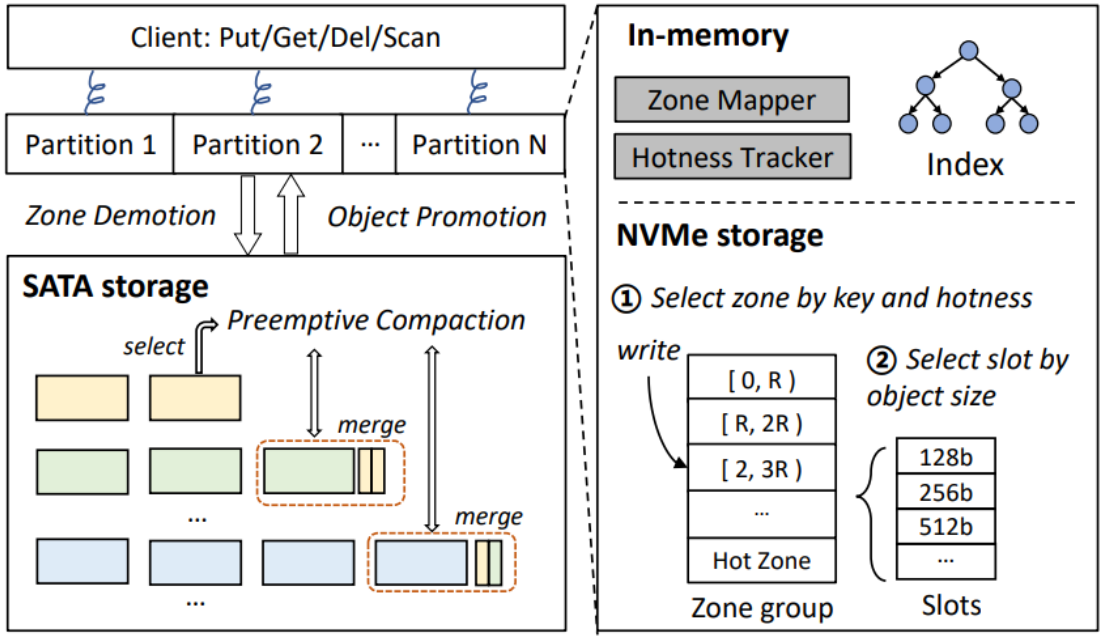


Figure 4: Architectural Overview of HyperDB

Key techniques

- Different data structures
  - NVMe storage: zone-based structure
  - Redesigned file format: *semi-SSTable*
- Lightweight object hotness tracker
- Preemptive compaction

Design: Data Structure

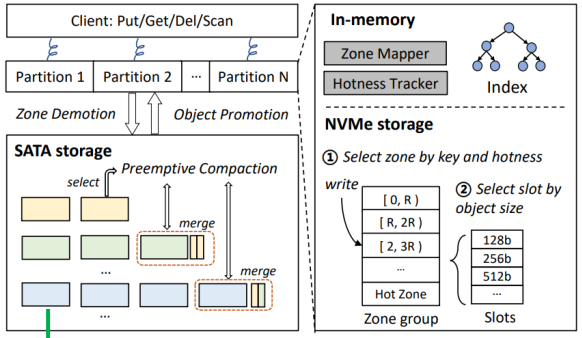


Figure 4: Architectural Overview of HyperDB

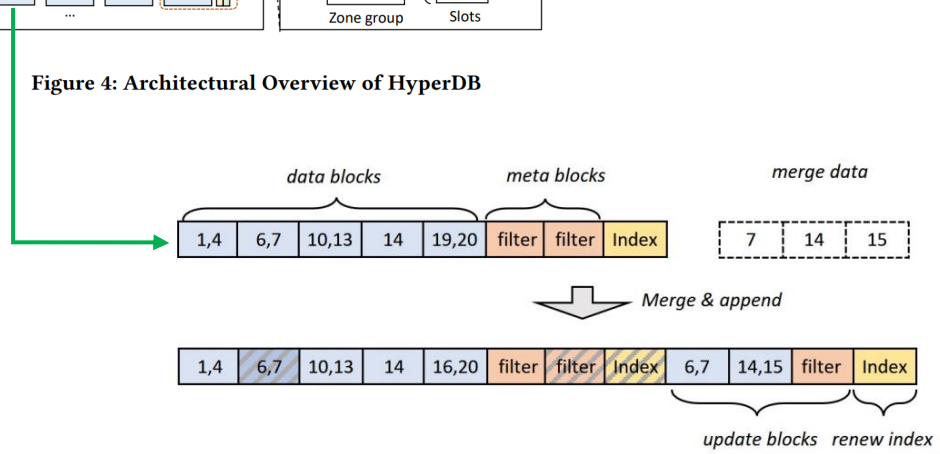


Figure 5: An example of a semi-SSTable and block-level merge operation is presented. Three new objects are merged into the original table, the blocks that need to be rewritten are considered dirty blocks. Objects from the dirty blocks and objects that need to be merged form a new block and the index is rebuilt.

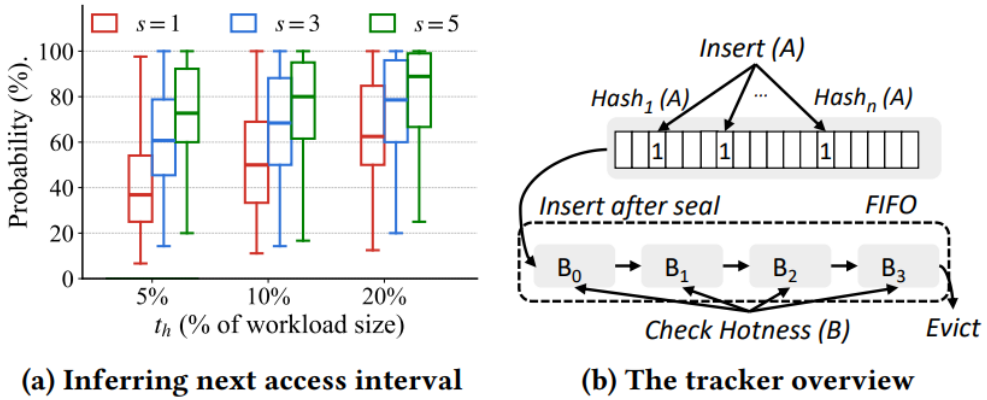
Zone-based structure

- Zone mapper maps the pages from slot files for each zone
- During a write of a new object, a new page is applied in the corresponding slot added to the zone mapper

Optimizing LSM-tree in SATA storage

- semi-SSTable
  - data blocks [sorted KV entries]
  - meta blocks [bloom filters]
  - index block [offsets, key ranges, validity :: prefix compression]

Design: Hotness of Objects



(a) Inferring next access interval (b) The tracker overview

Figure 6: The strong correlation between historical access intervals and the next visit time (a). And the tracker design (b) which employs a cascading discriminator to track access.

Tracking the hotness of objects

- Estimate the hotness with access interval: the threshold is the number of objects that NVMe storage can store. Objects with continuous access intervals less than the threshold are retained in NVMe storage.
- Track the hotness with cascading discriminator.

1. Open a bloom filter during the tracker initialization
2. Each read or update operation, tracker to insert the object into the opening bloom filter
3. When the bloom filter reaches its capacity, the tracker sets it as sealed and adds it to the cascading discriminator.
4. After the insertion, verify the existence of the object in all sealed bloom filters and identifies it as hot data only when it is present in a continuous series of bloom filters.

Design: Preemptive Block Compaction

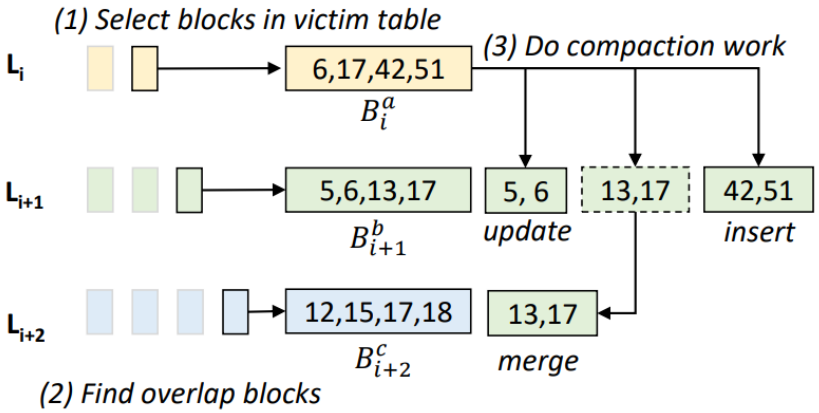


Figure 7: An example of preemptive block compaction. For a victim table in L_i, we update object (6), insert objects (42,51) into L_{i+1}, and preemptively merge object (17) with object (13) from L_{i+1} into L_{i+2}.

Compaction Process

1. Select a victim semi-SSTable in L_i (dirtiest / highest overlap)
2. Find overlap blocks recursively
3. Commence compaction work
   - Worker threads read the keys of the selected tables and find overlapping blocks in a top-down manner (index block)
   - Read all blocks in L_i and corresponding overlapping blocks in other levels
   - Construct new blocks
     - [Insert] Objects with continuous keys that do not overlap with existing blocks in L_{i+1}
     - [Update] Consecutive keys are updated in L_{i+1}, the object will be updated for the corresponding blocks and form new blocks
     - [Merge] Repeated keys in the next level are merged into a deeper level

Design: Migration Cross Tiers

HyperDB monitors the usage capacity of NVMe storage and performs migration tasks.

- Zone demotions: HyperDB triggers background demotion migration jobs to demote cold objects to SATA storage.
- Object promotions: When a read operation targets hot objects in the SATA storage, it promotes these objects and inserts them into the hot zones.

Evaluation: Experimental System

CPU: 16-core Intel Xeon Silver 4314

Memory: 64 GB

NVMe storage: 960 GB Samsung PM9A3

SATA storage: 960 GB Intel D3-S4610

YCSB experiments

Key and value size: 8 bytes and 128 bytes

Load: 100 GB randomly generated KV pairs

Each workload: 100M KV pairs

Evaluation: YCSB

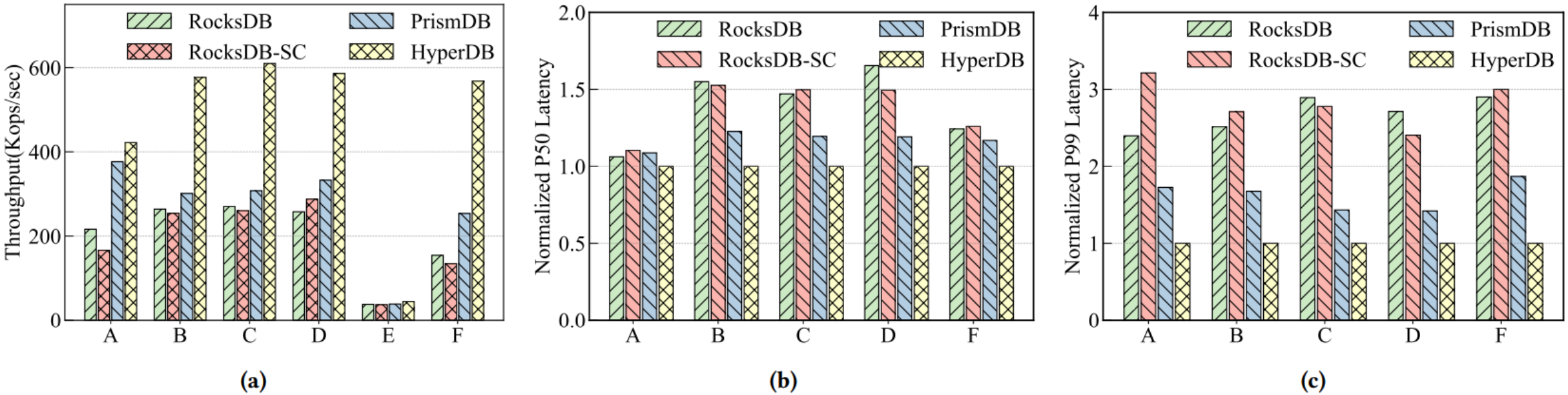
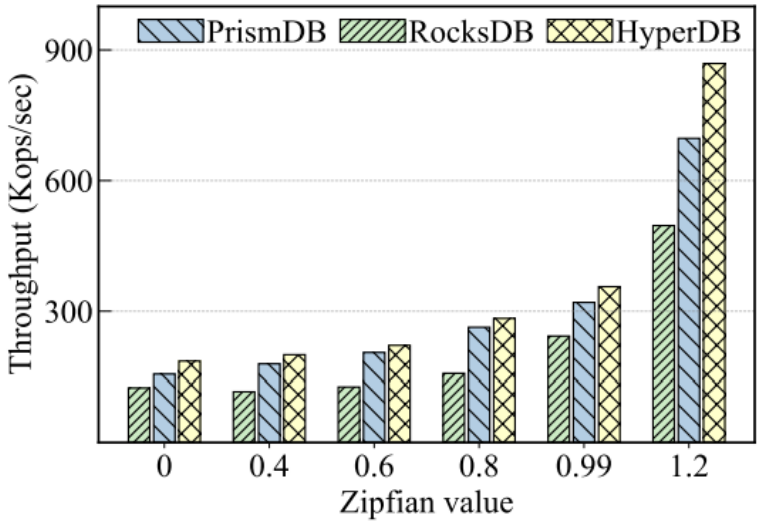


Figure 8: YCSB benchmark results for throughput (a), normalized median latency (b), and normalized P99 latency (c).

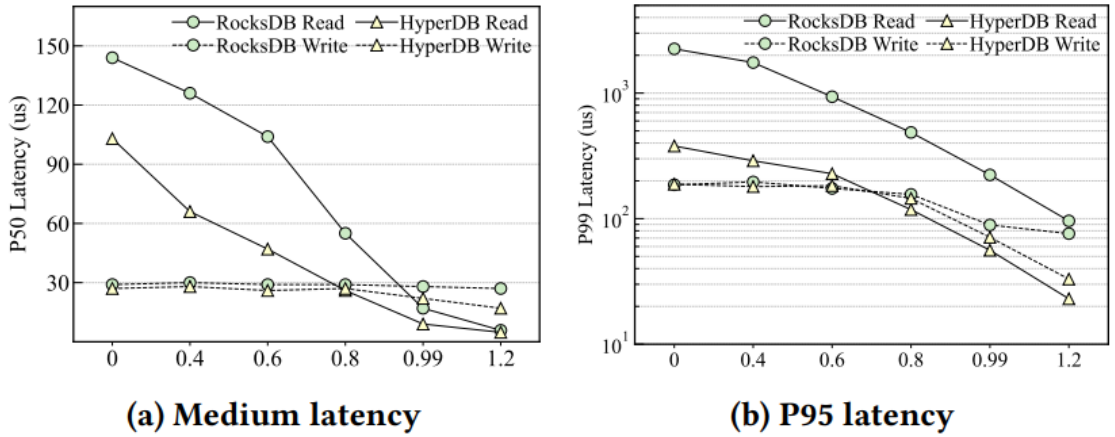
- HyperDB shows the best performance in throughput among all methods
- Read-intensive (BCD): 2.18-2.27x - frequently accessed objects are written to or promoted to the hot zones with majority of pages being cached.
- Write-intensive (AF): 2.81x improvement in average – HyperDB requires fewer bandwidth resources during migration compared to PrismDB, thus retaining more bandwidth for foreground tasks to prevent slowdowns

Evaluation: Impact of data skewness

Throughput: YCSB-A



(a) Impact of the workload skewness

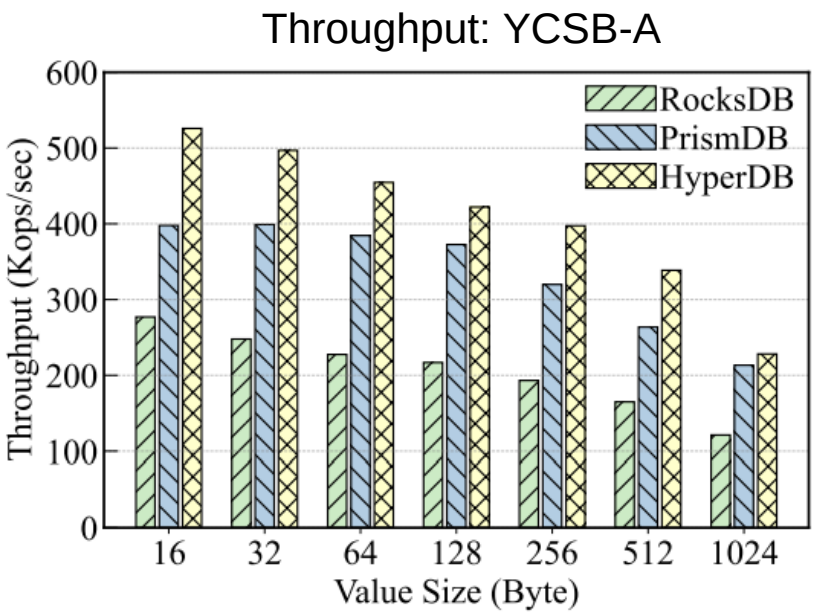


(a) Medium latency (b) P95 latency

Figure 10: Read/write latency breakdown with different workload skewness.

- HyperDB achieves 1.48-1.80× improvements compared to RocksDB and 1.08-1.25× improvements compared to PrismDB.
- For highly-skewed workloads, HyperDB delivers better performance because of the hotness tracking. For uniform workloads, the throughput shows a modest enhancement compared to RocksDB.
- HyperDB exhibits significantly lower read latency, whether considering median or tail latency

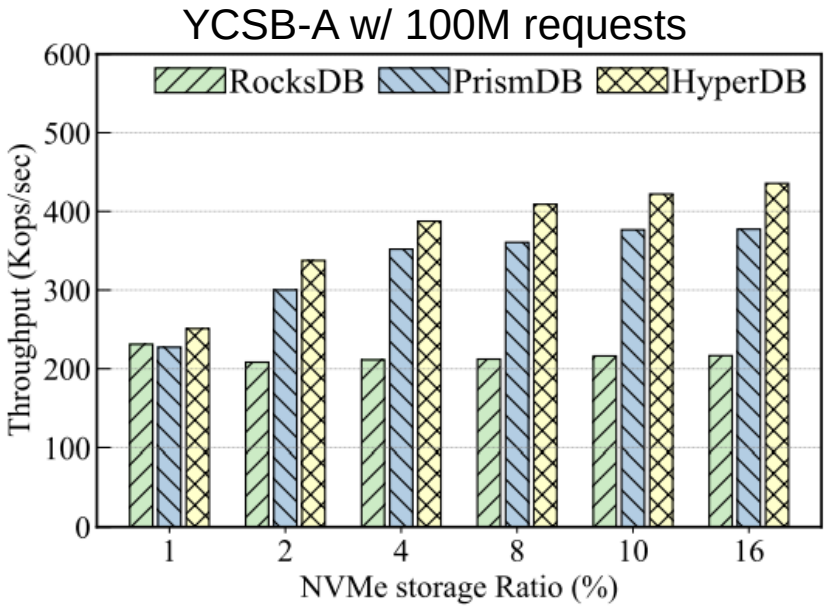
Evaluation: Impact of value size



(b) Impact of the workload value size

- RocksDB and HyperDB exhibit throughput improvements of 27.1% and 24.5%, respectively, when the value size is 16 bytes compared to 128 bytes. PrismDB only shows a 6.7% enhancement.
- At 16-byte value size, HyperDB exhibits a 1.88-2.05x throughput increase compared to RocksDB.

Evaluation: Impact of NVMe storage ratio



(c) Impact of the NVMe storage ratio

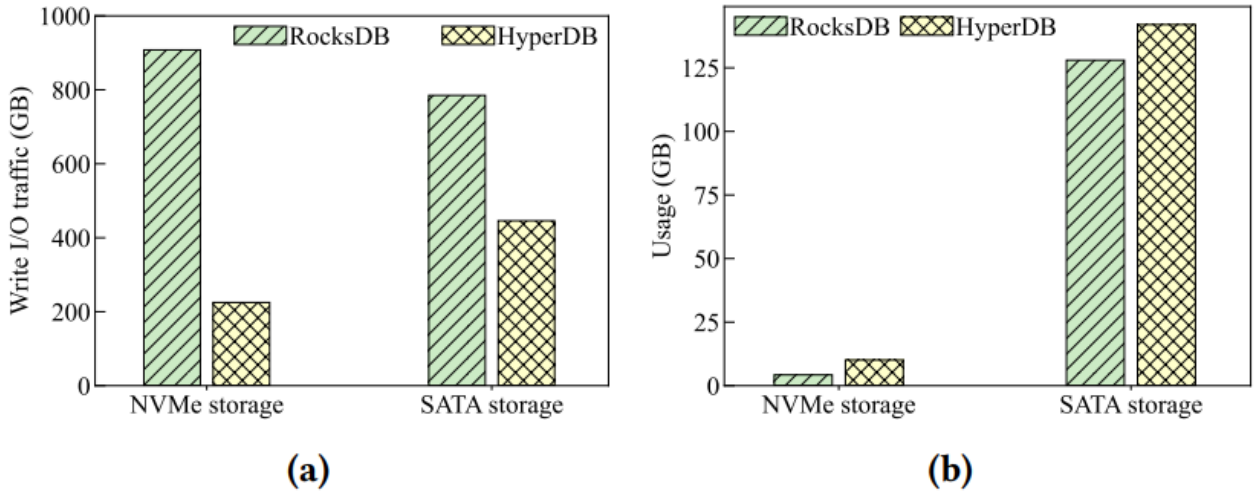


Figure 11: Total write I/O traffic (a) and space usage (b) under a uniform distribution workload.

- RocksDB does not exhibit significant performance improvements with the increase in NVMe storage capacity.
- PrismDB and HyperDB have shown respective enhancements of 1.66x and 1.73x at a 16% ratio compared to a 1% capacity.

- HyperDB exhibits the lowest write volume compared to RocksDB and better utilization of NVMe storage space.
- Although it results in a 10.9% increase in space consumption on SATA storage due to the presence of stale data in semi-SSTables.

Conclusion

HyperDB, a novel key-value store for heterogeneous SSD storage.

It introduces three technologies to reduce background traffic:

1. A zone-based layout in NVMe storage;
2. a lightweight hotness tracker; and
3. a semi-sorted table structure on the SATA storage managed with a preemptive compaction method.

Experimental results show that HyperDB achieves 2.25x faster on average throughput and a 60.3% reduction in background task traffic, compared to the standard use of RocksDB in data centers today.

THANK YOU!