KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression

Xubin Chen[†], Ning Zheng[‡], Shukun Xu[‡], Yifan Qiao[†], Yang Liu[‡], Jiangpeng Li[‡], Tong Zhang[†‡]

[†] Rensselaer Polytechnic Institute, NY, USA

[‡] ScaleFlux Inc., CA, USA

ABSTRACT

This paper studies the design of a key-value (KV) store that can take full advantage of modern storage hardware with built-in transparent compression capability. Many modern storage appliances/drives implement hardware-based data compression, transparent to OS and applications. Moreover, the growing deployment of hardware-based compression in Cloud infrastructure leads to the imminent arrival of Cloud-based storage hardware with built-in transparent compression. By decoupling the logical storage space utilization efficiency from the true physical storage usage, transparent compression allows data management software to purposely *waste* logical storage space in return for simpler data structures and algorithms, leading to lower implementation complexity and higher performance. This work proposes a table-less hash-based KV store, where the basic idea is to hash the key space directly onto the logical storage space without using a hash table at all. With a substantially simplified data structure, this approach is subject to significant logical storage space under-utilization, which can be seamlessly mitigated by storage hardware with transparent compression. This paper presents the basic KV store architecture, and develops mathematical formulations to assist its configuration and analysis. We implemented such a KV store *KallaxDB* and carried out experiments on a commercial SSD with built-in transparent compression. The results show that, while consuming very little memory resource, it compares favorably with the other modern KV stores in terms of throughput, latency, and CPU usage.

1 INTRODUCTION

This paper presents a key-value (KV) store design solution optimized for a growing family of storage hardware with built-in transparent data compression. Commercial market has witnessed the rise of storage appliances/devices that implement hardware-based data compression, transparent to OS and applications. Modern all-flash arrays (e.g., Dell EMC PowerMAX [12], HPE Nimble Storage [14], and Pure Storage FlashBlade [25]) come with built-in hardware-based transparent compression. Commercial SSDs with built-in transparent compression are emerging (e.g., computational storage drive from ScaleFlux [30] and Nytro SSD from Seagate [13]). Cloud vendors have started to integrate hardware-based compression capability into their storage infrastructure (e.g., Microsoft Corsia [7] and AWS Graviton2 [2]), leading to imminent arrival of cloud-based storage hardware with built-in transparent compression.

Beyond transparently reducing storage cost without affecting application performance, storage hardware with built-in transparent compression brings new opportunities to innovate data management software. This work shows that such storage hardware can relieve hash-based KV store from maintaining the costly in-memory hash table, leading to a new family of memory-efficient *table-less* hash-based KV store. The core is to leverage the fact that such storage hardware **decouples** the logical storage space utilization efficiency from the physical storage space utilization efficiency. When running on conventional storage hardware, KV store is *solely responsible* for the physical storage space utilization efficiency (i.e., physical storage cost). As a result, it faces a stringent trade-off between storage cost and implementation complexity: To reduce the storage cost, KV store must strive to make full use of the logical storage space and fill each 4KB LBA (logical block address) sector with KV pairs, which inevitably demands sophisticated and costly data structures and algorithms. For example, hash-based KV store uses a memory-hungry hash table to ensure the compact placement of KV pairs over the logical storage space in order to reduce the storage cost. The high memory cost of hash-based KV store is one of the main reasons why its real-world deployment pales in comparison with KV stores built upon memory-efficient data structures, e.g., log-structured merge (LSM) tree [23].

To fundamentally overcome the memory cost barrier of hash-based KV store, the only option is to hash the key space *directly* onto the logical storage space, without using a hash table at all. Nevertheless, KV pairs can no longer be tightly placed over the logical storage space, leading to substantial space *under-utilization* (e.g., all the 4KB LBA blocks have a large amount of empty space left unoccupied). Therefore, when running on conventional storage hardware, such table-less KV store will suffer from prohibitively high storage cost and hence is not practically viable. In contrast, once we pad the unoccupied space with highly compressible content (e.g., all-zero), storage hardware with built-in transparent compression can largely retain the physical storage space utilization efficiency. Therefore, when running on storage hardware with built-in transparent compression, table-less hash-based KV store can eliminate the memory cost obstacle without sacrificing storage cost. This could, for the first time, make hash-based approach a viable alternative to its tree-based counterparts on implementing large-capacity KV store.

DAMON'21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8556-5/21/06…$15.00
https://doi.org/10.1145/3465998.3466004

This paper presents a table-less hash-based KV store architecture, and develops a set of mathematical formulations to assist its analysis and configuration. Accordingly, we implement the proposed design techniques in KallaxDB, and compared it with three state-of-the-art tree-based KV stores including RocksDB [28], WiredTiger [32], and KVell [18]. Using the YCSB benchmarks, we carried out experiments on a commercial PCIe Gen3x4 SSD with built-in transparent compression [30]. The results show that, by consuming little memory resource, KallaxDB compares favorably with the other KV stores in terms of throughput, latency, and CPU usage. For example, under 400-byte KV size and 400GB dataset and YCSB 50:50 mixed read/write workload, compared with RocksDB, it achieves 1.6× higher ops/s, 1.5× shorter read latency, and 2.3× lower CPU utilization in terms of cycles per operation. It is our hope that this work will contribute to attracting more research interest on leveraging modern storage hardware with built-in transparent compression to innovate future data management systems.

2 BACKGROUND

To make user applications seamlessly benefit from data compression, the storage community has integrated the transparent compression capability into the lower-level IO stack (e.g., filesystem, block layer, or storage hardware). For example, ZFS [4] and Btrfs [29] support filesystem-level transparent compression, and Red Hat Enterprise Linux contains a VDO (Virtual Data Optimizer) module that realizes block-level transparent compression. However, software solutions consume the host CPU resource and hence suffer from the performance vs. storage cost trade-off. If we push data compression into the storage hardware layer, systems will be free from the performance vs. storage cost trade-off. Modern all-flash array appliances and some latest SSDs [13, 30] can support transparent compression. Fig. 1 illustrates an SSD with built-in transparent compression: Inside the SSD controller, (de-)compression are carried out on the IO path by the hardware, and the FTL (flash translation layer) manages the mapping/indexing of all the variable-length compressed data blocks. In order to maintain good random IO performance, the compression should be done on the per-4KB basis.

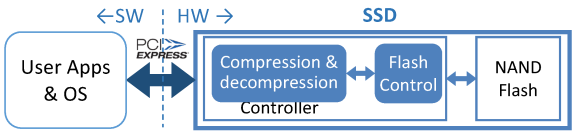

Figure 1: Illustration of an SSD with built-in transparent compression.

As illustrated in Fig. 2, storage hardware with built-in transparent compression has the following two properties: (a) The storage hardware can expose an LBA space that is much larger than its internal physical storage capacity. (b) Since special data patterns (e.g., all-zero) can be highly compressed, we can leave one 4KB LBA partially filled with valid data without wasting the physical storage space. These two properties decouple the logical storage space utilization efficiency from the physical storage space utilization efficiency. This allows data management software to purposely under-utilize the logical storage space to gain performance benefits, without sacrificing the true physical storage cost.

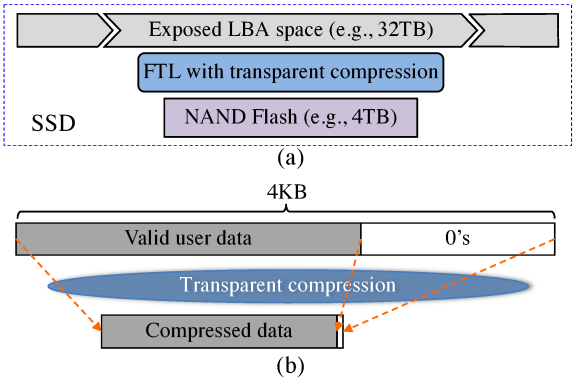

Figure 2: Illustration of the decoupled logical and physical storage space utilization efficiency enabled by storage hardware with built-in transparent compression.

3 PROPOSED DESIGN SOLUTION

3.1 Basic Idea

Leveraging the decoupled logical vs. physical storage utilization efficiency on storage hardware with built-in transparent compression, we propose a *table-less hash-based* KV store design approach as illustrated in Fig. 3. Its basic idea is to directly hash the key space onto the logical storage space, without going through an intermediate hash table. In conventional practice, as illustrated in Fig. 3(a), each key is first hashed to an entry in a hash table that further points to the KV pair's location in storage space. Through an intermediate hash table, the *indirect* mapping can ensure tight placement of KV pairs over the logical storage space. In order to serve each KV GET request with only one IO, the hash table must entirely reside in the memory, leading to a high and even prohibitive memory cost.

Storage hardware with built-in transparent compression makes it possible to eliminate the in-memory hash table altogether, as illustrated in Fig. 3(b). Let $\mathbb{K}$ denote the key space, and $\mathbb{L}$ denote the KV store logical storage space that is accessed in the unit of *pages*. Each page spans over one or multiple consecutive LBAs. We use a hash function $f_{K\to L}$ to hash each key $K_i \in \mathbb{K}$ onto one page $L_j \in \mathbb{L}$. Without using a hash table, it eliminates the memory cost obstacle of hash-based KV store. Moreover, by relieving CPU from managing/searching the hash table, it consumes less CPU resource. Meanwhile, as illustrated in Fig. 3(b), the proposed approach is subject to significant under-utilization of the logical storage space (i.e., almost all the pages have a large amount of empty space left unoccupied). Once we fill the unoccupied space with zeros, storage hardware with built-in compression can maintain a high utilization efficiency of the physical storage space.

3.2 Implementation of KallaxDB

We implemented a table-less hash-based KV store called *KallaxDB* with the architecture as shown in Fig. 4. Except those being logged in the WAL (write-ahead log), all the KV pairs are kept in two stores:

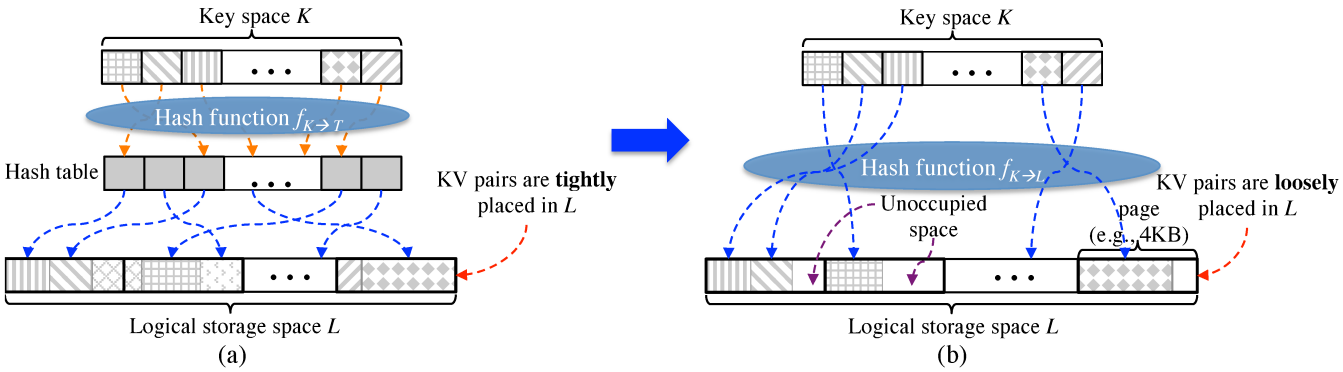

Figure 3: Illustration of the proposed table-less hash-based KV store design approach.

the main *hash-based store* that holds the vast majority of KV pairs, and the *overflow store* that holds the KV pairs that cannot fit into their destined pages in the main hash-based store. Let l_{pg} denote the hash-based store page size. If more than size-l_{pg} amount of KV pairs are hashed to the same page, we have to spill-over one or more KV pairs into the overflow store. Let $\mathbb{L}$ denote the overall

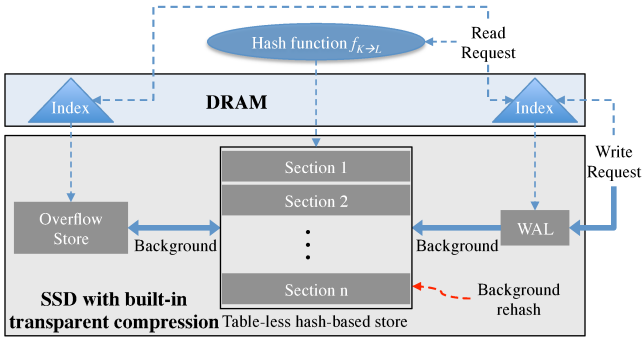

Figure 4: Illustration of KallaxDB architecture.

logical storage space of the main hash-based store. To improve the parallelism, we partition $\mathbb{L}$ into n equal-size sections, and each section $\mathbb{L}_j$ associates with a background write thread. To write a KV pair with key $K_i \in \mathbb{K}$ into the hash-based store, suppose we have $f_{K\to L}(K_i) = L_i \in \mathbb{L}_j$, the background write thread associated with the section $\mathbb{L}_j$ carries out a read-modify-write operation (i.e., fetch the page at L_i, modify its content according to the write request, and write the modified page back to L_i). To read a KV pair from the main hash-based store, we simply calculate the hash to identify and fetch the page from which we extract the requested KV pair.

KallaxDB implements both the WAL and overflow store in a log structure: Each incoming KV pair is appended to an on-SSD segment, and its index (i.e., key and address) is recorded in an in-memory B+ tree. Once the size of one on-SSD WAL segment reaches a threshold, this segment will be sealed, and a new on-SSD WAL segment will be created to receive incoming KV pairs. For KV pairs that cannot fit into the destined page, KallaxDB keeps them in the overflow store. The background write threads handle KV migration from WAL to the main hash-based store and reclaim the storage space occupied by sealed WAL segments. For segments of the overflow store, they will be re-scanned in the rehash procedure (described later), during which the storage space is reclaimed.

In order to keep page overflow probability sufficiently low and hence make the overflow store small, we must maintain appropriate $|\mathbb{K}| : |\mathbb{L}|$ ratio, where $|\mathbb{K}|$ denotes the number of keys in the key space $\mathbb{K}$ and $|\mathbb{L}|$ denotes the number of pages in the hash-based store logical storage space $\mathbb{L}$. Therefore, in adaptation to the runtime varying $|\mathbb{K}|$, we should adjust $|\mathbb{L}|$, for which we must accordingly change the hash function $f_{K\to L}$ and hence *rehash* the entire table-less hash-based store. For each rehash process, let $\mathbb{L}_{old}$ and $\mathbb{L}_{new}$ denote the logical storage space of the store before and after rehash. KallaxDB uses one or multiple background rehash threads, where each thread rehashes KV pairs in a batch-mode: Let n_b denote the batch size (e.g., 16). During each batch processing, one rehash thread reads n_b pages from $\mathbb{L}_{old}$ and extracts all the KV pairs, and rehashes every KV pair using the new hash function and accordingly writes the KV pair to its destined page in $\mathbb{L}_{new}$. If multiple KV pairs in the same batch are hashed to the same page in $\mathbb{L}_{new}$, we can coalesce their rehash to a single page write IO. To maximize the write coalescing, KallaxDB uses the following strategy to generate each hash function: Let $\mathbb{L}_M$ denote a logical storage space that is substantially larger than the maximum possible logical storage space of the hash-based store, and let $f_{K\to L_M}$ denote a fixed *master* hash function based on which we drive all the hash functions throughout the KV store lifetime. Given the target value of $|\mathbb{L}|$, define $R_s = \lceil |\mathbb{L}_M|/|\mathbb{L}| \rceil$ and we construct the corresponding hash function $f_{K\to L}$ as $\lfloor f_{K\to L_M}/Rs \rfloor$. When rehashing the logical storage space from $\mathbb{L}_{old}$ to $\mathbb{L}_{new}$, each batch processing rehashes all the KV pairs in a number of consecutive pages over $\mathbb{L}_{old}$. Because the old and new hash functions only differ at the divisor R_s, the KV pairs in consecutive pages over $\mathbb{L}_{old}$ will be rehashed to consecutive pages over $\mathbb{L}_{new}$, which can maximize the write coalescing.

Based on the above discussions, we can summarize the major operations of KallaxDB as follows:

Read: To serve a read request with the key K_i, KallaxDB first checks whether K_i locates in WAL by searching the corresponding index. If not, KallaxDB tries to fetch the KV pair from the main hash-based store via direct hashing. If still not found, KallaxDB turns to the overflow store. Due to the small sizes, WAL and overflow store

can easily keep their indexes entirely in the host memory. Hence, KallaxDB serves a read request via a single storage IO. To minimize the read latency and be compatible with typical KV store API, KallaxDB serves read requests through synchronous IOs.

Write: To serve a write request (*insert* or *update*), just like most other KV stores, KallaxDB logs the write request in WAL, updates the WAL index, and then immediately responds *write completion* to the client. KV pairs are moved from WAL into the hash-based store (or overflow store) during the background data migration, as illustrated in Fig. 4.

Background data migration: In the background, KallaxDB migrates KV pairs from WAL into the hash-based store (via read-modify-write) or overflow store. Since each section associates one background write thread, data migration can occur in parallel over all the n sections. All the background data migrations are realized through asynchronous IOs in order to better utilize the bandwidth of the storage hardware. The background write threads also periodically check whether there are section files needed to be rehashed and accordingly push them to the rehash threads.

Background rehash: KallaxDB expands/shrinks the logical storage space of the main hash-based store by applying a new hash function to rehash the key space. To avoid the contention between background write threads and background rehash threads, when one file in one section is being rehashed, no data will be migrated from WAL into this file. Only after a file has been completely rehashed, the new hash function will be used to serve client read requests and background data migration. Besides, the corresponding overflow store segment relative to this file will also be re-scanned and KVs in it will be filled into the new rehashed logical space accordingly. KallaxDB controls the speed of background rehash by configuring the batch size and the number of rehash threads.

3.3 Mathematical Formulation

This subsection presents a set of mathematical formulations to estimate two important KallaxDB operational metrics: (1) overflow statistics (including the page overflow probability and overflow store dataset size), and (2) page fill-factor (i.e., how full each page is filled with valid KV pairs). Accurate predication on these two metrics are critical for KallaxDB to appropriately configure the logical storage space of the main hash-based store. To facilitate the mathematical formulation, we model the KV pair size as *independent and identically distributed* random variables throughout the paper.

3.3.1 Overflow Statistics. Let $h_{kv}(l)$ denote the probability density function of KV pair size, and $f_{kv}(m,l)$ denote the probability density function of the random variable $l = \sum_{i=1}^{m} l_i$, where l_i is a random variable following the distribution of $h_{kv}(l)$. The open literature (e.g., see [17]) has well studied the formulation of $f_{kv}(m,l)$ for popular distributions (e.g., Gaussian and uniform distribution). Under the condition that m KV pairs are hashed to the same page, let $P_{ovf}(m)$ and $L_{ovf}(m)$ denote the corresponding page overflow probability and the spilled-over data size, we have

$$\begin{cases} P_{ovf}(m) &= \int_{l_{pg}}^{\infty} f_{kv}(m,l)dl, \\ L_{ovf}(m) &= \int_{l_{pg}}^{\infty} \big(f_{kv}(m,l)\cdot(l-l_{pg})\big)dl, \end{cases} \tag{1}$$

where l_{pg} denotes the page size that is 4KB or a multiple of 4KB. Let $P_{pg}(m)$ denote the probability that m KV pairs are hashed to the same page, we can calculate the average page overflow probability P_{pg_ovf} and spilled-over data size L_{pg_ovf} as

$$\begin{cases} P_{pg\_ovf} &= \sum_{m=1}^{\infty} \big(P_{pg}(m)\cdot P_{ovf}(m)\big), \\ L_{pg\_ovf} &= \sum_{m=1}^{\infty} \big(P_{pg}(m)\cdot L_{ovf}(m)\big). \end{cases} \tag{2}$$

Recall that $\mathbb{K}$ and $\mathbb{L}$ denote the key space and logical storage space of the hash-based store, and $|\mathbb{K}|$ and $|\mathbb{L}|$ represent the number of keys and pages in the hash-based store. Hence, we can express the total amount of data as $|\mathbb{K}|\cdot E\big(h_{kv}(l)\big)$, where $E\big(h_{kv}(l)\big)$ represents the expectation (or mean) of the KV pair size. Meanwhile, we can express the total amount of spilled-over data as $|\mathbb{L}|\cdot L_{pg\_ovf}$. Accordingly, we can calculate the data overflow ratio as

$$R_{ovf} = \frac{|\mathbb{L}|\cdot L_{pg\_ovf}}{|\mathbb{K}|\cdot E\big(h_{kv}(l)\big)}, \tag{3}$$

which is the ratio between the amount of data spilled-over into the overflow store and the total amount of data in KV store. Since the hash value uniformly distributes across all the $|\mathbb{L}|$ pages (i.e., one KV is hashed to a given page with the probability of $\frac{1}{|\mathbb{L}|}$), we can calculate the probability $P_{pg}(m)$ as

$$P_{pg}(m) = \binom{|\mathbb{K}|}{m}\left(\frac{1}{|\mathbb{L}|}\right)^{m}\left(1-\frac{1}{|\mathbb{L}|}\right)^{|\mathbb{K}|-m}. \tag{4}$$

Since the value of $|\mathbb{L}|$ is typically very large in practice and $(1-\frac{1}{n})^m \approx e^{-\frac{m}{n}}$ for large n, we approximate the Eq. 4 as

$$P_{pg}(m) \approx \binom{|\mathbb{K}|}{m}\left(\frac{1}{|\mathbb{L}|}\right)^{m} e^{-\frac{|\mathbb{K}|-m}{|\mathbb{L}|}}. \tag{5}$$

For the purpose of illustration, let us consider the following example: Assume that $|\mathbb{K}|$ and $|\mathbb{L}|$ are 1×10^{10} and 2×10^{9}, respectively, and the KV pair size follows a Gaussian distribution $\mathcal{N}(\mu,\sigma^2)$, where μ is 400 bytes, and σ is 25 bytes. Hence, the probability density function $f_{kv}(m,l)$ is $\mathcal{N}(m\cdot\mu, m\cdot\sigma^2)$. Meanwhile, we set the page size l_{pg} as 4KB. Applying the above formulations, we can obtain the $P_{pg}(m)$ and $P_{ovf}(m)$ when m varies between 0 and 15, as shown in Fig. 5. Accordingly, we can calculate that the page overflow probability P_{pg_ovf} is 1.9% and data overflow ratio R_{ovf} is 0.4%.

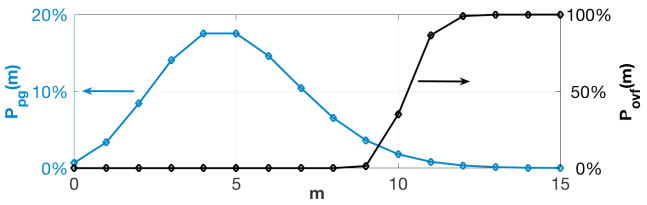

Figure 5: The probability $P_{pg}(m)$ that m KV pairs are hashed to one page, and the corresponding page overflow probability $P_{ovf}(m)$, when m varies between 0 and 15.

3.3.2 Page Fill-factor. Let $\alpha_{pg\_fill} \in [0, 100\%]$ denote the page fill-factor, i.e., the percentage by which one page is filled with valid KV pairs (and the rest is filled with all zeros). Based on the discussion and formulation presented above in Section 3.3.1, we can express the distribution of the page fill-factor as

$$\sum_{m=0}^{\infty} \left(P_{pg}(m) \cdot \frac{f_{kv}(m,l)}{l_{pg}} \right). \tag{6}$$

Accordingly, we can calculate the average value of the page fill-factor as

$$\alpha_{pg\_fill} = \sum_{m=0}^{\infty} \left(P_{pg}(m) \cdot m \cdot \frac{E(h_{kv}(l))}{l_{pg}} \right). \tag{7}$$

Regardless of the specific compression algorithm (e.g., lz4 or zlib) being used by the storage hardware, the physical storage space utilization degrades as the page fill-factor α_{pg_fill} reduces. This is because, when pages are less filled with KV pairs, the storage hardware will have less amount of valid data to compress in each 4KB sector. This will lead to a worse KV store data compression ratio. Hence, from the physical storage cost perspective, we should increase α_{pg_fill}, for which we must increase the probability $P_{pg}(m)$ according to Eq. 7. Meanwhile, according to Eq. 2, the page overflow probability and data overflow ratio will increase when $P_{pg}(m)$ increases. This reveals a trade-off between the physical storage cost and data overflow ratio. According to Eq. 4, $P_{pg}(m)$ will increase when either $|\mathbb{K}|$ increases and/or $|\mathbb{L}|$ decreases. Therefore, we can dynamically adjust the trade-off by configuring the value of $|\mathbb{L}|$ in response to the runtime value of $|\mathbb{K}|$.

Let us consider the following example: Assume the same KV pair size distribution as the example in Section 3.3.1, and set the page size l_{pg} as 4KB. Fig. 6 shows the calculated data overflow ratio R_{ovf} and page fill-factor α_{pg_fill}. Fig. 6(a) shows the impact of logical storage space size $|\mathbb{L}|$ when the key space size $|\mathbb{K}|$ fixes as 1×10^{10}, and Fig. 6(b) shows the impact of the key space size $|\mathbb{K}|$ when the logical storage space size $|\mathbb{L}|$ fixes as 2×10^{9}. This example shows the importance of appropriately adjusting $|\mathbb{L}|$ in adaptation to $|\mathbb{K}|$, which will be realized through the rehash process.

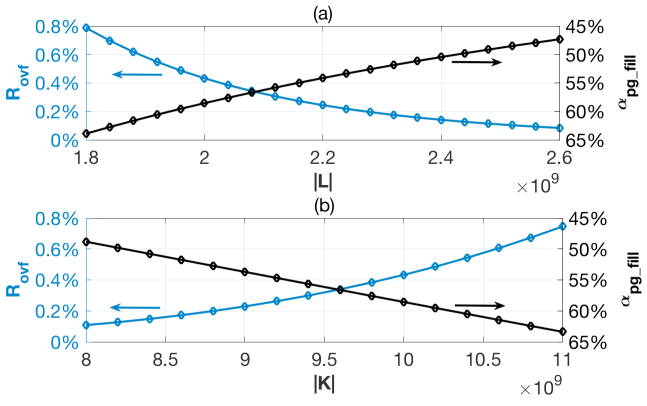

Figure 6: The data overflow ratio R_{ovf} vs. page fill-factor α_{pg_fill} when (a) $|\mathbb{L}|$ varies while $|\mathbb{K}|$ fixes as 1×10^{10}, and (b) $|\mathbb{K}|$ varies while $|\mathbb{L}|$ fixes as 2×10^{9}.

4 EVALUATION

4.1 Experimental Setup

We ran all the experiments on a server with 24-core 2.6GHz Intel CPU, 64GB DDR4 DRAM, and a 3.2TB SSD with built-in transparent compression. Manufactured by ScaleFlux [30], this SSD carries out hardware-based zlib compression on each 4KB block along the IO path. The per-4KB compression/decompression latency of the hardware zlib engine is around 5μs, which is more than 10× shorter than the NAND flash memory read latency (~80μs) and write latency (~1ms). Operating with PCIe Gen3×4 interface, this SSD can achieve up to 3.2GB/s sequential throughput and 650K (520K) random 4KB read (write) IOPS (IO per second) over 100% LBA span. In comparison, leading-edge commodity NVMe SSDs (e.g., Intel P4610) achieve similar sequential throughput and random 4KB read IOPS, but have much worse random 4KB write IOPS (e.g., below 300K). This is because built-in transparent compression can significantly reduce the GC (garbage collection) overhead inside SSDs. We use the following five YCSB benchmarks [8]: YCSB A (50% reads, 50% updates), YCSB B (95% reads, 5% updates), YCSB C (100% reads), YCSB D (95% reads, 5% inserts), YCSB F (50% reads, 50% read-modify-writes). The hash-based nature of KallaxDB makes it difficult to directly support the scan-centric benchmark YCSB E (i.e., 5% inserts and 95% scan). As discussed later in Section 4.6, we could complement KallaxDB with an auxiliary key-tracing data structure in order to support scan operations. Recent studies [5] suggest that KV pair size in real-world workloads is typically no more than a few hundred bytes. Hence, we focus on the value size ranging between 100B and 800B.

Table 1: YCSB core workloads.

Workload	Description
YCSB A	50% reads, 50% updates
YCSB B	95% reads, 5% updates
YCSB C	100% reads
YCSB D	95% reads, 5% inserts
YCSB F	50% reads, 50% read-modify-writes

4.2 Baseline Comparison

We first carried out a baseline comparison with three KV stores: (1) RocksDB 6.10 [28], which employs the LSM tree structure and is widely deployed in production environment. We set its maximum number of compaction and flush threads as 12 and 4, set the Bloomfilter as 10 bits per KV pair, and left all the other parameters as their default settings and did not turn on its block compression. (2) WiredTiger 3.2 [32], which supports both B-tree and LSM tree structure and is the default storage engine of MongoDB. We configured it to use B-tree structure in our experiments, set its leaf node size as 4KB to maximize its performance under YCSB workloads, and left all the other parameters as their default settings and did not turn on its page compression. (3) KVell [18], which employs B-tree structure and uses asynchronous IOs to serve both read and write requests. Different from WiredTiger, KVell keeps the pointer to each KV pair in order to avoid sorting KV pairs on storage. KVell does not support compression on its own, and we left all the parameters as their default settings.

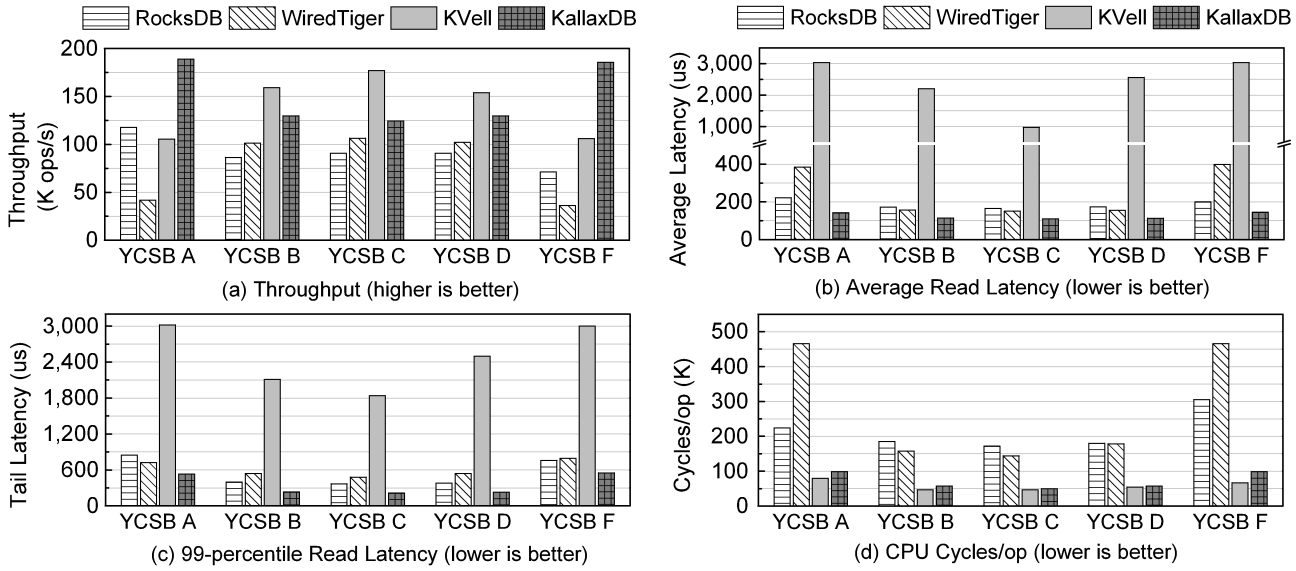

Figure 7: Baseline comparison with state-of-the-art key value stores under 400GB dataset with 400B value size.

We set the size of key and value as 16 bytes and 400 bytes, respectively. The dataset contains 1 billion KV pairs (i.e., $|\mathbb{K}| = 1 \times 10^9$), hence the total raw data volume is about 400GB. Regarding the KallaxDB configuration, the main hash-based store contains 16 sections (i.e., 16 background write threads), and we set the page size as 4KB and the total number of pages $|\mathbb{L}| = 2 \times 10^8$ (i.e., the total logical storage space is about 800GB). According to the formulation presented in Section 3.3, the page overflow probability P_{pg_ovf} is 1.37% and data overflow ratio R_{ovf} is 0.38% (i.e., about 1.5GB of KV pairs are stored in the overflow store). When running YCSB workloads, we set the number of clients as 16 for RocksDB, WiredTiger, and KallaxDB. Since KVell uses asynchronous IOs for both read and write, our experiments show that using a smaller number of client threads (e.g., 4) can achieve much better performance, and accordigly we set the number of client threads as 4 for KVell. Fig. 7 shows the measured results of average operational throughput (i.e., ops/s), average read latency, 99-percentile read latency, and CPU efficiency. This work quantifies the CPU efficiency in terms of *Cycles/op* [24], which can be estimated as follows: Let U_{CPU} denote the measured CPU utilization, f_{CPU} and n_{core} denote the CPU clock frequency and number of CPU cores, and C_{ops} denote the average ops/s, we express *Cycles/op* as $(U_{CPU} \cdot f_{CPU} \cdot n_{core})/C_{ops}$. The results in Fig. 7 reveal the following observations.

Operational throughput ops/s: For write-intensive workloads YCSB A and F (with only 50% reads), KallaxDB achieves the best throughput. In particular, under workload YCSB A, KallaxDB outperforms RocksDB, WiredTiger, and KVell by 1.6×, 4.5×, and 1.8×, respectively. Under workload YCSB F, KallaxDB outperforms RocksDB, WiredTiger, and KVell by 2.6×, 5.8× and 1.8×, respectively. For read-intensive workloads YCSB B, C, and D (with 95% or 100% reads), KVell achieves the best throughput. This is because KVell serves read requests in the asynchronous manner, which can invoke a large IO queue depth (e.g., 64) and hence higher read throughput. In contrast, the other three KV stores serve read requests in the synchronous manner. As a result, with 16 client threads, they at most have an IO queue depth of 16, leading to a lower read throughput than KVell. Nevertheless, as shown in Fig. 7, asynchronous reads are subject to very long read latency.

Read latency: By serving read requests in the asynchronous manner, KVell essentially trades longer read latency for higher read throughput. As shown in Fig. 7, the read latency (both average and 99-percentile) of KVell is significantly longer (e.g., even over 10×) than the other three. KallaxDB achieves the shortest read latency (both average and 99-percentile) consistently across all the five YCSB workloads. Compared with RocksDB and WiredTiger, KallaxDB achieves 1.4~2.7× shorter average read latency across all the five YCSB workloads, because KallaxDB has the smallest read amplification.

CPU efficiency: As discussed above, because of its asynchronous nature on serving read requests, KVell achieves the best performance under a small number of client threads (e.g., 4 client threads in this study). Therefore, not surprisingly, KVell has the best CPU efficiency as shown in Fig. 7, which nevertheless comes at the cost of very long read latency. Regarding the comparison among the other three, KallaxDB achieves better CPU efficiency than RocksDB and WiredTiger consistently across all the five YCSB workloads. On average, the CPU efficiency of KallaxDB is 2.3~ 3.2× better than that of RocksDB, and 2.8~ 4.7× better than that of WiredTiger.

Table 2 lists the measured storage space (both logical and physical) and index memory usage. All experiments run on the same 3.2TB SSD with 400GB user dataset. The storage space usage is measured using the compression statistics utility commands provided by ScaleFlux [30] CSD 2000 Drive. The logical storage space is the total amount of space that each KV store occupied on the file system, and the physical storage space is the amount of space that each KV

store occupied inside the SSD with built-in transparent compression. Although KallaxDB occupies almost 2× bigger logical storage space than the others, its physical storage space is very similar to the others. Compared with RocksDB and KVell, WiredTiger has a larger logical space amplification. Among all the four KV stores, the physical storage space of WiredTiger is noticeably larger than the other three. As shown in Table 2, KVell consumes much larger index memory than the others, because it keeps the pointer to each individual KV pair. With 4KB leaf node size, WiredTiger consumes about 11.4GB for its B-tree index. RocksDB consumes about 4.4GB for its index (mainly including index blocks and Bloomfilters). By eliminating the hash table, KallaxDB consumes the least amount of index memory (i.e., 1.2GB), which is mainly for storing the index of its WAL and overflow store.

Table 2: Storage space and index memory usage comparison.

	Storage space usage		Index
	Logical	Physical	Memory usage
RocksDB	419GB	209GB	4.4GB
WiredTiger	573GB	276GB	11.4GB
KVell	512GB	197GB	34.0GB
KallaxDB	805GB	218GB	1.2GB

4.3 Impact of Dataset and Key-Value Size

We further studied the impact of total dataset size and KV pair size. Under the value size of 400 bytes, in addition to the 400GB dataset with 1 billion KV pairs, we considered two other dataset sizes: (1) 100GB with 250 million KV pairs, and (2) 1TB with 2.5 billion KV pairs. We were not able to run KVell over the 1TB dataset because its index cannot entirely fit into server's 64GB memory. Hence, we focused on the comparison among RocksDB, WiredTiger, and KallaxDB in this context. We kept the same configurations of RocksDB and WiredTiger as above. Regarding KallaxDB, we set the page size as 4KB, and the logical storage space as 200GB (i.e., $|\mathbb{L}| = 5 \times 10^7$) and 2TB (i.e., $|\mathbb{L}| = 5 \times 10^8$) when dataset size is 100GB and 1TB, respectively. Accordingly, they have the same page overflow probability (i.e., 1.37%) and data overflow rate (i.e., 0.38%) as the 400GB case. Fig. 8 shows the YCSB A and B throughput ops/s and 99-percentile read latency under the three different datasets. The results show that the performance of KallaxDB is almost independent from the dataset size. In comparison, dataset size tends to have a noticeable impact on the performance of RocksDB and WiredTiger. The indexing complexity of RocksDB and WiredTiger is proportional to the dataset size, while the indexing of KallaxDB is independent from the dataset size.

Under the same dataset size of 400GB, we considered two other value sizes: (1) 100 bytes with total 4 billion KV pairs, and (2) 800 bytes with total 500 million KV pairs. We kept the same configurations of RocksDB and WiredTiger as above. We configure KallaxDB as follows: In the case of 100-byte value size, the logical storage space is set to 640GB (i.e., $|\mathbb{L}| = 1.6 \times 10^8$), leading to the page overflow probability of 4.0% and data overflow rate of 0.31%. In the case of 800-byte value size, the logical storage space is set to 1TB (i.e., $|\mathbb{L}| = 2.5 \times 10^8$), leading to the page overflow probability of 1.7% and data overflow rate of 1.0%. Fig. 9 shows the YCSB A and B

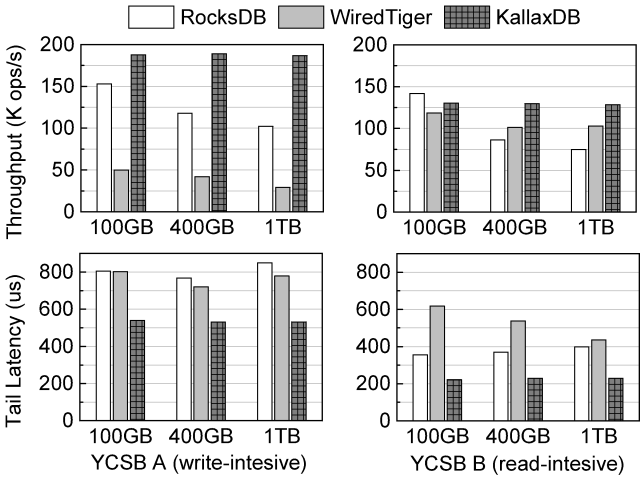

Figure 8: Impact of dataset size on throughput ops/s and 99-percentile read latency under YCSB A and B.

throughput ops/s and 99-percentile read latency. The performance of KallaxDB is independent from the value size. In contrast, value size could have relatively more noticeable impact on the performance of RocksDB. For example, under workload YCSB A, the ops/s and 99-percentile read latency of RocksDB degrade by 16% and 1.5× as the value size increases from 400B to 800B.

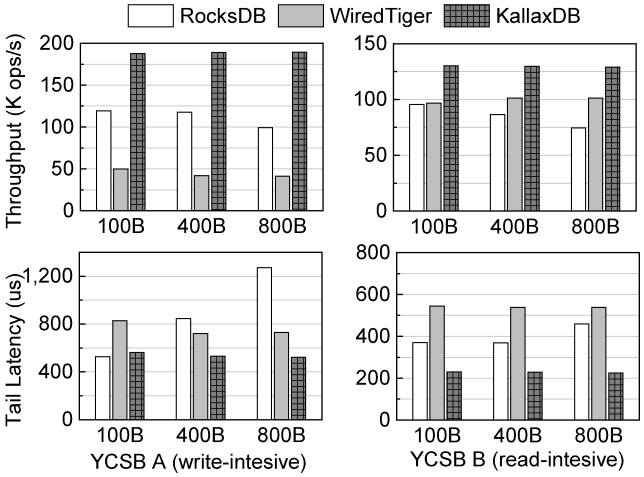

Figure 9: Impact of value size on throughput ops/s and 99-percentile read latency under YCSB A and B.

Fig. 10 shows the total data write size and write amplification for YCSB A by running 200 million requests of YCSB A on three 400GB datasets with different value size. Due to the nature of LSM tree, the write amplification of RocksDB is relatively independent from the value size. As shown in Fig. 10, its write amplification remains between 10 and 20. Accordingly, the total data write size of RocksDB is proportional to the value size. In contrast, the total data write size of WiredTiger and KallaxDB is relatively independent from the value size (especially KallaxDB).

As a result, their write amplification is inversely proportional to the value size, as shown Fig. 10.In the case of 100B value size, KallaxDB has over 3× larger write amplification than RocksDB. Meanwhile, Fig. 9 shows that KallaxDB noticeably outperforms RocksDB under 100B value size. This suggests that write amplification on its own does not necessarily play an important role on KV store performance, especially when the system is far from being IO-bound. For example, even with the write amplification of 35 under 100B value size, KallaxDB has an average write IO throughput of about 378MB/s, which is well below the maximum IO bandwidth of the SSD.

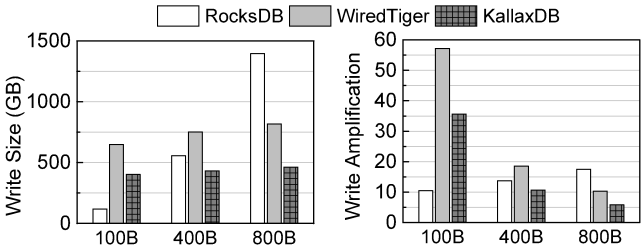

Figure 10: Total write size and write amplification when running 200 million requests of YCSB A, where the dataset size is 400GB and value size is 100B, 400B and 800B, respectively.

4.4 Impact of Page Overflow Rate

We further studied the impact of the page overflow rate on the performance of KallaxDB. Different page overflow rates correspond to different size of overflow store index, leading to different memory usage and overall KV store performance. Under the same logical storage space size of 800GB (i.e., $|\mathbb{L}| = 2 \times 10^8$) and 400-byte value size, we considered four different total amount of raw dataset, including 400GB, 480GB, 560GB, and 640GB. Table 3 compares these four cases, where P_{pg_ovf} denotes the page overflow rate and α_{pg_fill} denotes the page fill-factor. As P_{pg_ovf} increases, the overflow store must host more data and hence will consume more memory for its index. Meanwhile, when the page overflow rate increases, the page fill-factor increases and hence the overall data compression ratio (i.e., the ratio of dataset size over physical space size) will improve. Results also show that performance impact is negligible for low page overflow rate. Even with the overflow rate as high as 18%, the throughput of KallaxDB degrades by only 10.1%, 3.6%, 2.3%, 4.5%, and 8.4% under the five YCSB workloads.

Table 3: Four cases with different page overflow rate.

Dataset size	400GB	480GB	560GB	640GB
P_{pg_ovf}	1.37%	4.2%	10%	18%
Index memory	0.7G	2.2GB	5.2GB	10GB
Physical space	218GB	248GB	282GB	319GB
α_{pg_fill}	0.5	0.6	0.7	0.8
Comp. ratio	1.83	1.94	1.99	2.01

Fig. 11 shows the throughput ops/s of the four cases under the five YCSB workloads. The results show that the performance of KallaxDB is weakly dependent on the page overflow rate. Even with the overflow rate as high as 18%, the performance of KallaxDB degrades by only 10.1%, 3.6%, 2.3%, 4.5%, and 8.4% under the five YCSB workloads.

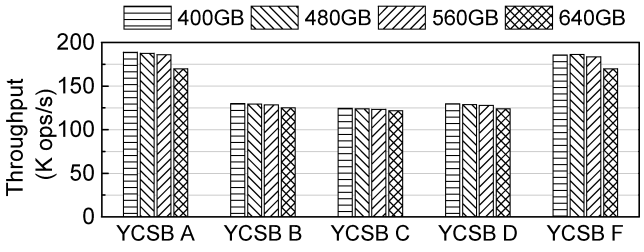

Figure 11: Operational throughput ops/s under four cases with different page overflow rates.

4.5 Impact of Store Rehash

We studied the impact of store rehash on the performance of KallaxDB, although rehash occurs rarely (especially when the dataset size has become relatively stable). The longer time the rehash process takes, the less impact it will have on the KV store performance. We can control the rehash latency by configuring the number of background rehash threads (denoted as n_t) and the rehash batch size (denoted as n_b). During rehash, each thread fetches and processes n_b pages at a time. We can configure the performance impact vs. rehash latency trade-off by adjusting n_t and/or n_b. We carried out experiments on a dataset with total 600 million KV pairs and value size of 400B. Before rehash, the logical storage space is 400GB, corresponding to a page overflow rate of 4%. After rehash, the logical storage space is 800GB, corresponding to a page overflow rate of 0.03%. We considered three settings on $\{n_t, n_b\}$: {8, 4}, {16, 4}, and {16, 16}. Table 4 summarizes the average ops/s and read latency results when running YCSB A and B with and without rehash.

Table 4: Performance impact of store rehash.

	YCSB A		YCSB B	
	Ops/s	Latency	Ops/s	Latency
No rehash	186K	144μs	130K	114μs
{8, 4}	155K	183μs	116K	137μs
{16, 4}	127K	215μs	97K	154μs
{16, 16}	107K	250μs	84K	187μs

Fig. 12 further shows the operational throughput ops/s of before/during/after rehash under YCSB A and B. The results clearly show the performance impact vs. rehash latency trade-off. Because KallaxDB issues heavier IO traffic under write-intensive workloads (e.g., YCSB A) than read-intensive workloads (e.g., YCSB B), rehash-induced performance impact is more significant under write-intensive workloads.

4.6 Enhancement to Support Scan

The hash-based nature of KallaxDB makes it difficult to directly support scan. Hence, above we only considered the five YCSB benchmarks that do not contain scans. For systems that demand the

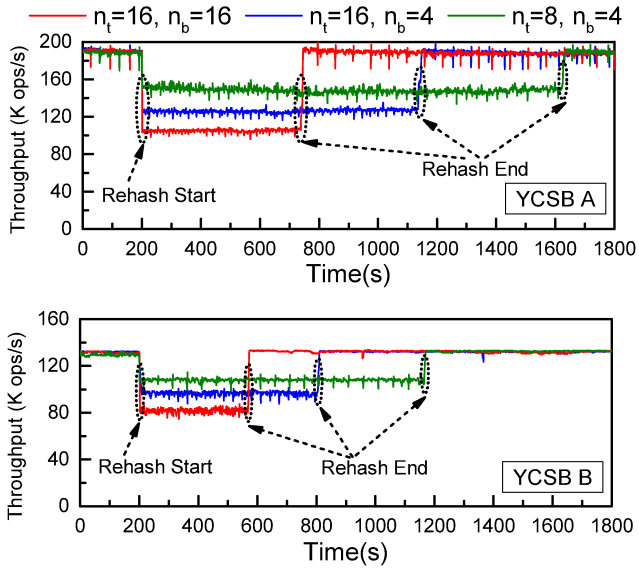

Figure 12: Throughput ops/s of before/during/after rehash under workloads YCSB A and B.

support of scan, we can complement KallaxDB with an auxiliary key-tracing store: We use a separate tree-based data structure to store all the keys that are present in KallaxDB. Upon receiving a scan request, we first search the auxiliary key-tracing store to obtain all the matching keys that fall into the scan range. Then we GET the corresponding KV pairs from KallaxDB to serve the scan request. We carried out experiments under the settings of 16-byte key size, 400-byte value size, and 1 billion KV pairs. Let *KallaxDB-AUX* denote the implementation of enhancing KallaxDB with the auxiliary key-tracing store. Table 5 compares the storage space and index memory usage of KallaxDB and KallaxDB-AUX. The results show that KallaxDB-AUX does not incur significant storage space overhead, because of the small key size relative to the value size.

Table 5: Storage and index memory usage comparison.

	Storage space		Index
	Logical	Physical	Memory
KallaxDB	805GB	218GB	1.2GB
KallaxDB-AUX	812GB	222GB	1.4GB

We ran the scan-intensive YCSB E workload (i.e., 95% scans and 5% inserts) on RocksDB, WiredTiger, KVell, and KallaxDB-AUX. In YCSB E, on average one scan request spans over 50 KVs. All the experiments used 16 client threads. Table 6 compares the average throughput (ops/s), average read latency (τ_{avg}), 99-percentile read latency (τ_{tail}), and CPU Cycles/op (C_{op}). By keeping KVs sorted on storage, WiredTiger and RocksDB achieve much better scan performance than KVell and KallaxDB-AUX. Since RocksDB must scan all the levels, it suffers from a higher read amplification and hence worse scan performance than WiredTiger. Nevertheless, RocksDB still achieves 11× and 5× higher performance than KVell and KallaxDB-AUX, respectively.

Table 6: Results of the scan-intensive YCSB E workload.

DB type	ops/s	τ_{avg}	τ_{tail}	$C_{op}(K)$
RocksDB	18.8K	0.84ms	1.8ms	942
WiredTiger	25.0K	0.77ms	2.0ms	631
KVell	1.7K	4.03ms	32.7ms	10,814
KallaxDB-AUX	3.6K	3.7ms	6.8ms	3293

Table. 7 further compares the throughput and CPU efficiency between KallaxDB and KallaxDB-AUX under the other five YCSB workloads and one update-only workload. Since the auxiliary key-tracing store is active only during PUT/DELETE requests, we incorporate an update-only workload in order to reveal the largest difference between KallaxDB and KallaxDB-AUX. The results show that, except the update-only workload, both have almost the same throughput performance under the other YCBS workloads. Under the update-only workload, KallaxDB-AUX suffers from about 8% throughput performance degradation. The difference between KallaxDB and KallaxDB-AUX mainly reflects from the CPU efficiency. For the write-intensive workloads, the CPU efficiency of KallaxDB-AUX degrades by 17% (YCSB A), 15% (YCSB F), and 17% (update-only), respectively.

Table 7: Throughput and CPU utilization efficiency of KallaxDB and KallaxDB-AUX.

	Throughput (ops/s)		e_{cpu} (Cycles/op)	
	KallaxDB	AUX	KallaxDB	AUX
YCSB A	184K	183K	118K	117K
YCSB B	131K	131K	77K	58K
YCSB C	129K	127K	79K	52K
YCSB D	129K	130K	77K	57K
YCSB F	184K	182K	115K	115K
Update only	201K	186K	185K	217K

5 RELATED WORK

Because LSM tree is relatively memory-efficient and matches well with the flash memory operational characteristics, KV store built upon LSM tree has received most attentions in research community and been widely deployed (e.g., RocksDB [28] and Cassandra [1]). Most prior work on LSM tree KV store aimed at reducing the write amplification by modifying the data architecture. PebblesDB [26] reduces the write amplification by developing a fragmented LSM tree structure. Dostoevsky [9] reduces the write amplification by developing a lazy leveling scheme, and also presents a generalization of the entire LSM tree design space. TRIAD [3] reduces the write amplification by dynamically separating hot and cold keys and deferring the compaction. Skip-Tree [34] reduces the write amplification by allowing certain KV pairs to skip the level-by-level compaction. VT-tree [31] reduces the write amplification by using stitching operation to avoid unnecessary data copies for sequential data. LSM-trie [33] reduces the write amplification by integrating exponential data growth pattern with linear data growth pattern. SlimDB [27] reduces the write amplification by customizing the data structure for semi-sorted data. X-Engine [15] reduces the write amplification by identifying and recycling the data blocks whose

key ranges do not overlap with any other data blocks during compaction. Write amplification can also be reduced by separating the storage of key and value, which has been well demonstrated in prior work (e.g., Wisckey [20] and HashKV [6]).Luo and Carey [21] present a comprehensive survey on LSM tree KV stores.

Prior work also has well studied KV store built upon other data structures. SILT [19] organizes all the KV pairs across several different data structures with different performance vs. cost trade-offs. Tucana [24] presents a KV store built upon a modified Bϵ-tree, incorporating several techniques to improve caching and IO efficiency. KVell [18] presents a KV store built upon B-tree, which applies KV-size-based data partition to reduce the impact of background garbage collection. Prior work [10, 11, 16] also studied the design of hash-based KV store, which stores KV pairs in log-structure on SSD and realizes addressing through in-memory hash table. uDepot [16] holds the hash table entirely in host memory and aims to improve IO efficiency through a user-space IO stack. FlashStore [10] and SkimpyStash [11] present techniques to reduce the memory cost of hash table at the penalty of higher read amplification. NVMKV [22] implements a hash-based KV store by deeply coupling the SSD FTL customization and KV store design. In comparison, by taking advantage of storage hardware with built-in transparent compression, our proposed design approach fundamentally removes the memory cost barrier of hash-based KV store.

6 CONCLUSIONS

This paper presents a table-less hash-based KV store customized for modern storage hardware with built-in transparent compression capability. By decoupling logical vs. physical storage space utilization efficiency, such new storage hardware enables KV store purposely under-utilize the logical storage space in return for simpler data structures and algorithms, which can lead to higher performance and/or lower CPU/memory usage. Following this theme, the proposed table-less hash-based KV store completely obviates the in-memory hash table by directly hashing the key space onto the logical storage space, and meanwhile relies on the transparent compression in storage hardware to retain the physical storage cost efficiency. This paper presents a set of mathematical formulations that can assist its configuration and analysis. Experimental results show that table-less hash-based KV store compares favorably with the other modern KV stores in terms of throughput, latency, and CPU and memory usage.

REFERENCES

[1] Apache Cassandra. [n.d.]. . http://cassandra.apache.org/.

[2] AWS Graviton Processor. [n.d.]. . https://aws.amazon.com/ec2/graviton/.

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

[4] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The zettabyte file system. In *Proceedings of the Usenix Conference on File and Storage Technologies (FAST)*, Vol. 215.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[6] Helen HW Chan, Chieh-Jan Mike Liang, Yongkun Li, Wenjia He, Patrick PC Lee, Lianjie Zhu, Yaozu Dong, Yinlong Xu, Yu Xu, Jin Jiang, et al. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 1007–1019.

[7] Derek Chiou, Eric Chung, and Susan Carrie. 2019. (Cloud) Acceleration at Microsoft. *Tutorial at Hot Chips* (2019).

[8] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*. ACM, 143–154.

[9] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 505–520.

[10] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1414–1425.

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

[12] Dell EMC PowerMax. [n.d.]. . https://delltechnologies.com/.

[13] E. F. Haratsch. 2019. SSD with Compression: Implementation, Interface and Use Case. In *Flash Memory Summit*.

[14] HPE Nimble Storage. [n.d.]. . https://www.hpe.com/.

[15] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An optimized storage engine for large-scale E-commerce transaction processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 651–665.

[16] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *USENIX Conference on File and Storage Technologies (FAST)*. 1–15.

[17] Harold J Larson. 1995. *Introduction to Probability*. Addison-Wesley, Reading, MA.

[18] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*. 447–461.

[19] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 1–13.

[20] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. WiscKey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 5.

[21] C. Luo and M.J. Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29 (2020), 393–418.

[22] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *USENIX Annual Technical Conference (ATC)*. 207–219.

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

[24] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 537–550.

[25] Pure Storage FlashBlade. [n.d.]. . https://purestorage.com/.

[26] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 497–514.

[27] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment* 10, 13 (2017), 2037–2048.

[28] RocksDB. [n.d.]. . https://github.com/facebook/rocksdb.

[29] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32.

[30] ScaleFlux Computational Storage. [n.d.]. . http://scaleflux.com.

[31] Pradeep J Shetty, Richard P Spillane, Ravikant R Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. 17–30.

[32] WiredTiger. [n.d.]. . https://github.com/wiredtiger/.

[33] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 71–82.

[34] Yinliang Yue, Bingsheng He, Yuzhe Li, and Weiping Wang. 2016. Building an efficient put-intensive key-value store with skip-tree. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (2016), 961–973.