KVell: the Design and Implementation of a Fast Persistent Key-Value Store

Baptiste Lepers
University of Sydney

Oana Balmau*
University of Sydney

Karan Gupta
Nutanix

Willy Zwaenepoel
University of Sydney

Abstract

Modern block-addressable NVMe SSDs provide much higher bandwidth and similar performance for random and sequential access. Persistent key-value stores (KVs) designed for earlier storage devices, using either Log-Structured Merge (LSM) or B trees, do not take full advantage of these new devices. Logic to avoid random accesses, expensive operations for keeping data sorted on disk, and synchronization bottlenecks make these KVs CPU-bound on NVMe SSDs.

We present a new persistent KV design. Unlike earlier designs, no attempt is made at sequential access, and data is not sorted when stored on disk. A shared-nothing philosophy is adopted to avoid synchronization overhead. Together with batching of device accesses, these design decisions make for read and write performance close to device bandwidth. Finally, maintaining an inexpensive partial sort in memory produces adequate scan performance.

We implement this design in KVell, the first persistent KV able to utilize modern NVMe SSDs at maximum bandwidth. We compare KVell against available state-of-the-art LSM and B tree KVs, both with synthetic benchmarks and production workloads. KVell achieves throughput at least 2x that of its closest competitor on read-dominated workloads, and 5x on write-dominated workloads. For workloads that contain mostly scans, KVell performs comparably or better than its competitors. KVell provides maximum latencies an order of magnitude lower than the best of its competitors, even on scan-based workloads.

*Oana Balmau is supported by an SOSP 2019 student scholarship from the ACM Special Interest Group in Operating Systems (SIGOPS).

SOSP '19, October 27–30, 2019, Huntsville, Ontario, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359628

Keywords Key-Value Store, Persistence, Performance, SSD, NVMe, B+ Tree, Log-Structured Merge Tree (LSM)

ACM Reference Format:
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 SOSP '19: ACM Symposium on Operating Systems Principles (SOSP '19).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341301.3359628

1 Introduction

Key-value stores (KVs) have become the standard platform for providing storage in a wide range of cloud applications, including, among others, caching [44], metadata management [5], messaging [1] and online shopping [15]. In this paper, we are targeting KVs on block-addressable storage devices that provide persistence guarantees (i.e., data and updates must not be lost in case of failures) and where the working set does not fit entirely in main memory.

Historically, the speed gap between storage devices and CPUs has been so large that investing CPU cycles in optimizing storage access was a beneficial tradeoff. As a result, some KVs include elaborate indices such as B+ trees. Log-Structured Merge (LSM) KVs force sequential writes because they are more efficient than random writes. All KVs include some form of cache. To support efficient scans that retrieve all items in a given key range, KVs maintain items in sorted order in memory and on disk. Although all these optimizations require some investment of CPU cycles, the storage device remained the bottleneck, and the optimizations proved beneficial because they relieved that bottleneck.

When we measure the performance of state-of-the-art KVs running on block-addressable NVMe SSD devices, we find that the CPU is the bottleneck and not the storage device, confirming earlier qualitative observations [40, 41]. NVMe SSDs exhibit very high bandwidth and comparable performance for random and sequential access. As a result, many of the optimizations developed for conventional storage devices are no longer useful and in fact counterproductive, because their cost in terms of CPU cycles worsens the CPU bottleneck. An obvious example is the attempt by LSM KVs to force sequential writes. There is no longer any gain in terms of device access, and the maintenance operations required by LSM KVs (mainly compaction) burden the CPU and

therefore negatively impact performance. A more surprising example is that synchronization on shared (in-memory) data structures for caching and maintaining order contribute to the CPU being the bottleneck.

We conclude that the change in storage device characteristics necessitates a paradigm shift in the design of KVs, focusing on streamlined use of the CPU. In this paper, we introduce the following four design principles for KVs that we have found useful in doing so.

1. Shared-nothing: all data structures are partitioned among cores so that each core can operate almost entirely without any synchronization.
2. Items are not sorted on disk. An in-memory sorted index is maintained per partition for efficient scans.
3. No attempt is made to force sequential access, but I/O operations are batched to reduce the number of costly system calls. Batching furthermore provides control over the length of the device queues, making it possible to simultaneously achieve low latency and high throughput.
4. No commit log: updates are acknowledged only after they have been persisted at their final location on disk.

Besides providing better average throughput, by doing away with a number of complex maintenance procedures, these principles also lead to more predictable performance, both in terms of throughput and latency. Some of these design decisions are not without their tradeoffs. For instance, the shared-nothing architecture runs the risk of load imbalance. The absence of a sorted order impacts scans. We show in the evaluation that these drawbacks are outweighed by the benefit of a streamlined CPU operation on workloads that mainly consist of medium to large key-value pairs (400B+).

We implement these techniques in KVell and we show that KVell compares favorably against four state-of-the-art KVs: RocksDB [15] and PebblesDB [43], both LSM KVs, and TokuMX [42] and WiredTiger [48], both based on B tree derivatives. We use the industry-standard YCSB benchmarks, with both uniform and skewed key access distributions. Furthermore, we show that throughput remains steady throughout the experiments, whereas other systems experience periods of poor performance. We confirm these results with measurements on alternative workloads and platforms.

Contributions. The contributions of this paper are:

1. An analysis of conventional LSM and B tree KVs, demonstrating that they become bottlenecked by the CPU on NVMe SSDs.
2. A new paradigm for persistent KVs to leverage the properties of NVMe SSDs, focusing on streamlined use of the CPU.
3. The KVell KV that incorporates this new paradigm.
4. An in-depth evaluation of KVell compared to state-of-the-art LSM and B tree KVs on synthetic benchmarks and production workloads.

Roadmap. The rest of the paper is organized as follows. Section 2 shows the evolution of SSD performance. Section 3 discusses limitations of current KV designs. Section 4 presents the design principles underlying KVell. Section 5 describes in more detail the technical solutions adopted in KVell. Section 6 presents an in-depth evaluation of KVell. Section 7 discusses related work and Section 8 concludes the paper.

2 Evolution of SSD Performance

Hardware. We consider the following three devices that were introduced over the last 5 years:

- **Config-SSD.** A 32-core 2.4GHz Intel Xeon, 128GB of RAM, and a 480GB Intel DC S3500 SSD (2013).
- **Config-Amazon-8NVMe.** An AWS i3.metal instance, with 36 CPUs (72 cores) running at 2.3GHz, 488GB of RAM, and 8 NVMe SSD drives of 1.9TB each (brand unknown, 2016 technology).
- **Config-Optane.** A 4-core 4.2GHz Intel i7, 60GB of RAM, and a 480GB Intel Optane 905P (2018).

IOPS. Table 1 shows the maximum number of read and write IOPS and the random and sequential bandwidth of these three devices. First, the number of IOPS and bandwidth have increased dramatically. Second, on older devices, random writes are significantly slower than sequential writes, but this is no longer the case. Similarly, mixing random reads and writes no longer results in poor performance as with older devices. For instance, the Config-Optane, using an Intel Optane 905P drive released in Q3 2018, can sustain 500K+ IOPS regardless of the mix of reads and writes, and random accesses are only marginally slower than sequential accesses.

Latency and bandwidth. Table 2 presents latency and bandwidth measurements for all three devices as a function of the device queue depth, with one core performing random writes. Previous generation devices can only sustain sub-millisecond response time with a small number of simultaneous I/O requests (32 in the case of the Config-SSD). The Config-Amazon-8NVMe and Config-Optane support higher parallelism, with both drives able to respond to 256 simultaneous requests with sub-millisecond latency. Both drives only reach a fraction of their bandwidth when there are too few requests in the device queue. Thus, even on modern devices, a fine balance has to be maintained between sending too few simultaneous requests (resulting in sub-optimal bandwidth) and sending too many (resulting in high latency).

Throughput degradation. Figure 1 shows IOPS over time on all three devices. This figure also shows the progress of device technology. For instance, the Config-SSD can sustain 50K write IOPS for 40 minutes, but then its performance slowly degrades to 11K IOPS. Newer SSDs do not suffer from this problem and offer high and constant IOPS over time.

Disk	Read IOPS (1,000s)	Writes IOPS (1,000s)	Mix 50%r-50%w IOPS (1,000s)	Seq. Read (GB/s)	Rand. Read (GB/s)	Seq. Write (GB/s)	Rand. Write (GB/s)	Mix rand. rw (GB/s)
Config-SSD (2013)	75	11	63	0.5	0.3	0.4	0.04	0.2
Config-Amazon-8NVMe (2016)	412	180	175	1.9	1.6	0.8	0.7	0.7
Config-Optane (2018)	575	550	560	2.6	2.3	2.0	2.0	2.0

Table 1. IOPS and bandwidth depending on the workload for 3 SSDs. IOPS and bandwidth have dramatically increased in new drives and there is little performance difference between random and sequential accesses.

Queue depth	Config-SSD lat. (us)	Config-SSD bw. (MB/s)	Config-Amazon-8NVMe lat. (us)	Config-Amazon-8NVMe bw. (MB/s)	Config-Optane lat. (us)	Config-Optane bw. (MB/s)
1	65	59	33	102	11	370
16	380	180	80	453	34	1099
32	760	195	192	618	68	1230
64	1500	204	385	671	137	1507
256	6000	210	770	672	550	1585
512	12000	211	1500	642	1100	1622

Table 2. Average latency and bandwidth depending on the queue size, doing random writes on a single core. A proper choice of queue length is required to simultaneously achieve low latency and high bandwidth utilization.

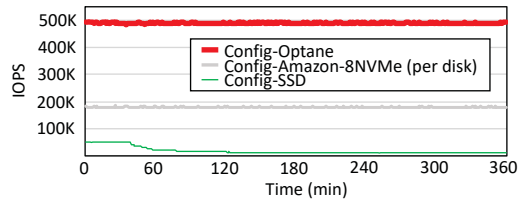

Figure 1. Number of IOPS over time for 3 SSDs. Old generation SSDs only support short bursts of I/Os at max IOPS.

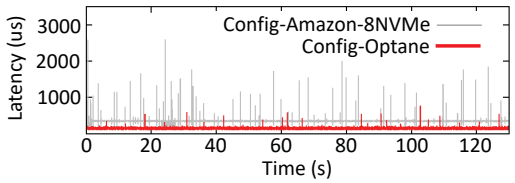

Figure 2. Latency of 4K write requests (queue depth 64) over time on AWS and Intel Optane.

Latency spikes. Figure 2 shows the latency of 4K writes with a queue depth of 64 on Config-Amazon-8NVMe and Config-Optane. Older generation SSDs suffer from latency spikes in write-heavy workloads due to internal maintenance operations. On Config-SSD we observe latency spikes of up to 100ms after 5 hours, compared to the normal write latency of 1.5ms. These results are not shown in Figure 2 because the magnitude of the spikes for this device would obscure the results for the other devices. The Config-Amazon-8NVMe drives also suffer for periodic latency spikes. The maximum observed latency is 15ms (vs. a 99^{th} percentile of 3ms). On the Config-Optane drive, latency spikes happen irregularly, and their magnitude is usually less than 1ms, with an observed maximum of 3.6ms (vs. a 99^{th} percentile of 700us).

3 Issues with current KVs on NVMe SSDs

Currently there are two major paradigms for persistent KVs: (1) LSM KVs, which are widely accepted as the best choice for write-dominated workloads, and (2) B tree KVs, which are considered more suitable for read-intensive workloads. LSM KVs are used in popular systems, such as RocksDB [15] and Cassandra [16]. B trees and their variants are used in MongoDB [42, 48]. We next demonstrate that both designs become CPU-bottlenecked on NVMe SSDs and suffer from severe performance fluctuations. We go beyond existing work [6, 40, 41] in showing that these observations also hold for B tree KVs and in providing a detailed accounting of the CPU overheads.

3.1 CPU is the bottleneck

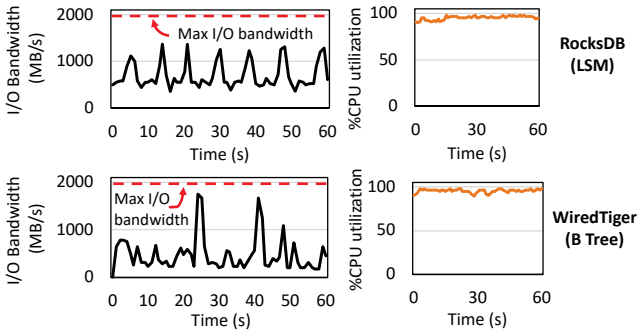

Figure 3. Config-Optane. I/O bandwidth (left) and CPU consumption (right) timelines for RocksDB (LSM KV) and for WiredTiger (B tree KV). Neither system can make use of the full device I/O bandwidth. Workload: YCSB A 50% write–50% read, uniform key distribution, 1KB KV-item size.

Figure 3 shows the disk utilization (left) and the CPU utilization (right) on Config-Optane for two state-of-the-art KVs: RocksDB, an LSM KV, and Wired Tiger, a B tree KV (results are similar for other KVs). We use the YCSB core workload A (write-intensive) with uniform key distribution for this example. Both systems saturate the CPU and do not take full advantage of the available bandwidth. We now explain these behaviors.

CPU is the bottleneck in LSM KVs. LSM KVs are optimized for write-intensive workloads by absorbing updates in an in-memory buffer [36, 39]. When the buffer is full, it is flushed to disk. The flushed buffer is then merged by background threads in a tree-like structure maintained in persistent storage. The disk structure contains multiple levels with increasing sizes. Each level contains multiple immutable sorted files with disjoint key ranges (except for the first level which is reserved for writing the memory buffer). To preserve this structure on disk, LSM KVs implement CPU and I/O intensive maintenance operations called compactions, which merge data from the lower to the higher levels of the LSM tree, maintaining item ordering and discarding duplicates.

Compactions are known to compete for disk bandwidth on older devices [5, 6], but merging, indexing, and kernel code also compete for CPU time, and on newer devices the CPU has become the primary bottleneck. Profiling shows that RocksDB spends up to 60% of CPU time on compactions (28% on merging data, 15% on building indexes, and the remainder on reading and writing data to disk). The need for compaction stems from the LSM design requirements of sequential disk access and keeping data sorted on disk. This design was beneficial for older drives where it was worth spending CPU cycles to keep data sorted and ensure long sequential disk accesses.

CPU is the bottleneck in B tree KVs. There are two variants of B trees that are designed for persistent storage: B+ trees and Bϵ trees. B+ trees contain KV items in the leaves. The internal nodes only contain keys and are used for routing. Generally, the internal nodes reside in memory and the leaves reside in persistent storage. Each leaf holds a sorted range of KV items and the leaves are chained in a linked-list, making it convenient to scan. State-of-the-art B+ trees (e.g., WiredTiger) rely on caching to achieve good performance [49]. Updates are first written in per-thread commit logs, and then in the cache. Eventually, the tree is updated with the new information when data is *evicted* from the cache. Updates use sequence numbers, which are necessary for scans. Reads go through the cache, accessing the tree only if the item is not cached.

There are two types of operations that persist data to the tree: (1) *checkpointing* and (2) *eviction*. Checkpointing happens periodically or when a certain size threshold is reached in the logs. Checkpointing is necessary to keep the commit logs bounded in size. Eviction writes dirty data from the cache to the tree. Eviction is triggered when the amount of dirty data in the cache exceeds a certain threshold.

The Bϵ tree is a variant of B+ tree, augmented with buffers that temporarily store keys and values at each node. Eventually, the KV items trickle down the tree structure and are written to persistent storage, as the buffers get full.

B tree designs are prone to synchronization overheads. Profiling in WiredTiger reveals that worker threads spend 47% of the total time busy waiting for slots in logs (in the `__log_wait_for_earlier_slot` function that uses the `sched_yield` system call to busy wait). The problem stems from not being able to advance the sequence numbers for updates fast enough. In the main code path of updates, excluding the time spent in kernel, WiredTiger only spends 18% of its time performing client request logic and the rest of its time is spent waiting. WiredTiger also needs to perform background operations: eviction of dirty data from the page cache accounts for 12% of the total time and management of the commit logs for 5% of the total time. Only 25% of the time spent in the kernel is dedicated to read and write calls, while the rest of the time is spent in `futex` and `yield` functions.

Bϵ trees also suffer from synchronization overheads. Since Bϵ trees keep data ordered on disk, worker threads end up modifying a shared data structure, resulting in contention. Profiling of TokuMX shows that threads spend up to 30% of their time in locks or atomic operations used to protect shared pages. Buffering also proves to be a major source of overhead in Bϵ trees. In the YCSB A workload TokuMX spends more than 20% of its time moving data from buffers to their correct location in the leaves. These synchronization overheads dwarf other overheads.

In modern NVMe SSDs configurations it is beneficial to minimize sharing as much as possible in order to avoid CPU overhead related to synchronization. Logs, previously used to provide rapid persistence guarantees end up being major bottlenecks. Buffering also introduces overhead – an observation already made for in-memory KVs [26], which, interestingly, now also applies to persistent storage.

3.2 Performance fluctuations in LSM and B tree KVs

In addition to being CPU-bound, both LSM and B tree KVs suffer from significant performance fluctuations. Figure 4 shows the throughput fluctuation over time for RocksDB [15] and for WiredTiger [48] running YCSB core workload A. The throughput is measured every second. In both LSM and B tree KVs the fundamental issue is similar: client updates are stalled because of maintenance operations.

In LSM KVs, throughput drops because sometimes updates need to wait for compactions to finish. When the first level of the LSM tree is full, updates need to wait until space is made available through compaction. However, we have seen that compactions hit the CPU bottleneck when LSM KVs are running on modern drives. The performance difference in throughput over time goes up to one order of magnitude: RocksDB sustains on average 63K requests/s but drops to 1.5K when writes are stalled. Profiling reveals that writer threads spend approximately 22% of their time stalled waiting for the memory component to be flushed. Flushes of the memory component are delayed because compactions cannot keep up with the speed of updates.

Solutions have been proposed to reduce the effect of compaction on performance, by delaying compactions [43] or

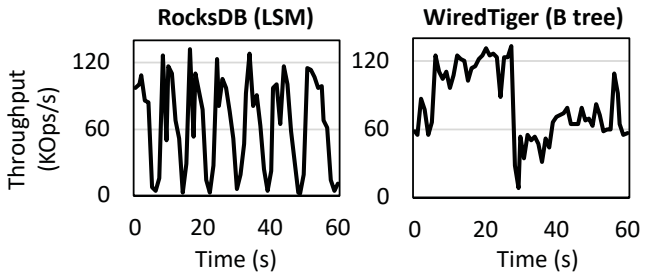

Figure 4. Config-Optane. Throughput fluctuation in RocksDB and WiredTiger. Workload: YCSB A 50% write–50% read, uniform key distribution, 1KB KV-item size.

running them only when the system is idle [6], but these solutions are not well suited on high-end SSDs. For instance, the Config-Optane machine flushes memory components at 2GB/s. Delaying compactions by more than a few seconds results in a huge backlog of work and wasted space.

In B trees performance is also affected by stalls in the user workload. User writes are stalled because evictions are not able to make progress fast enough. Stalls are causing throughput to drop by an order of magnitude, from 120 Kops/s to 8.5 Kops/s. We conclude that maintenance operations such as compactions and evictions heavily interfere with the user workload in both cases, causing stalls that can last up to *several seconds*. Therefore, new KV designs should avoid maintenance operations.

4 KVell design principles

In order to efficiently make use of modern persistent storage KVs now need to emphasize low CPU overhead. We show that the following principles are key for achieving peak performance on modern SSDs.

1. **Share nothing.** In KVell this translates to favoring parallelism and minimizing shared state between KV worker threads to reduce CPU overhead.
2. **Do not sort on disk, but keep indexes in memory.** KVell persists items un-ordered on disk at their *final location*, avoiding expensive re-ordering operations.
3. **Aim for fewer syscalls, not for sequential I/O.** KVell does not aim for sequential disk access, leveraging the fact that random accesses are almost as efficient as sequential accesses on modern SSDs. Instead, it strives to minimize CPU overhead caused by system calls through batching I/O.
4. **No commit log.** KVell does not buffer updates and thus does not need to rely on a commit log, avoiding unnecessary I/O.

4.1 Share nothing

For the common case of individual reads and writes, the worker threads that handle a request do so without any synchronization with other threads. Each thread handles requests for a given subset of the keys and maintains a thread-private set of data structures for managing this set of keys. The key data structures are: (i) a lightweight, in-memory B tree index that keeps track of keys' location in persistent storage, (ii) I/O queues, responsible for efficiently storing and retrieving information from persistent storage, (iii) free lists, partially in-memory lists of disk blocks that contain free spots to store items and (iv) a page cache – KVell uses its own internal page cache and does not rely on OS-level constructs. Scans are the only operation where minimal synchronization is required on the in-memory B tree indexes.

This shared-nothing approach is a key difference with respect to conventional KV design where all or most of the major data structures are shared by all worker threads. The conventional approach entails synchronization for each request, which KVell avoids entirely. Partitioning requests may lead to load imbalance, but we find that with suitable partitioning the effects are minor.

4.2 Do not sort on disk, but keep indexes in memory

KVell does not sort data in the working sets of the worker threads. Because KVell does not order keys, it can persist the items on disk at their *final location*. This complete lack of ordering on disk reduces the overhead of inserting items (i.e., finding the right location to insert) and eliminates CPU overhead associated with maintenance operations on disk (or, alternatively, sorting before writing to disk). Storing keys out of order on disk especially benefits write operations, and helps to achieve low tail latencies.

During scans, successive keys are no longer in the same disk block, which may appear to be a drawback. However, maybe surprisingly, scan performance is not significantly affected for workloads with medium-size and large KV-items (e.g., scans in the YCSB benchmark or in our production workloads, as we show in Section 6).

4.3 Aim for fewer syscalls, not for sequential I/O

In KVell, all operations (including scans) perform random accesses to disk. Since random accesses are as efficient as sequential access, KVell does not waste CPU cycles trying to enforce sequential I/O.

Similar to LSM KVs, KVell batches requests to disk. The goal is, however, different. LSM KVs primarily use batching I/O and sorting KV items in order to leverage sequential disk access. KVell batches I/O requests with the primary goal of reducing the number of system calls, hence reducing CPU-overhead.

Batching comes with a trade-off. As seen in Section 2, disks need to be kept busy at all times to achieve their peak IOPS, but only respond with sub-millisecond latency if their hardware queues contain less than a given number of requests (e.g., 256 on Config-Optane). An efficient system should push enough requests to the disks to keep them busy, but should

not overwhelm them with large queues of requests which lead to high latency.

In configurations with multiple disks, each worker only stores files on one disk. This design decision is key to limiting the number of pending requests per disk. Indeed, since workers do not communicate with each other, they do not know how many requests other workers have sent to a given disk. If workers store data on a single disk, then the number of requests to a disk is bounded by (batch size × number of workers per disk). If workers were to access all disks, then a disk might have up to (batch size × total number of workers) pending requests.

Because requests are assigned to workers based on their key, and because workers only access one disk, it is possible to design a workload that will mostly access data located on one disk and leave other disks idle. In workloads that exhibit skewed behavior, however, the data skew is absorbed by the internal page cache. Consequently, most of the load imbalance does not result in disk I/O.

4.4 No commit log

KVell acknowledges updates only after they have been persisted to disk at their final location, without relying on a commit log. Once an update is submitted to a worker thread, it is persisted to disk in the next I/O batch. Removing the commit log allows KVell to use disk bandwidth only for useful client request processing.

5 KVell implementation

Although KVell's design principles may seem straightforward, they are challenging to properly implement in practice. The source code of KVell is available at https://github.com/BLepers/KVell.

5.1 Client operations interface

KVell implements the same core interface as LSM KVs: writes `Update(k,v)`, reads `Get(k)`, and range scans `Scan(k1,k2)`. `Update(k,v)` associates value v to key k. `Update(k,v)` only returns once the value has been persisted to disk. `Get(k)` returns the most recent value of k. `Scan(k1,k2)` returns the range of KV items between $k1$ and $k2$.

5.2 On-disk data structures

To avoid fragmentation, items that fit in the same size range are stored in the same files. We call each of these files a *slab*. KVell accesses slabs at block granularity, which is the page size (4KB) on our machines.

If items are smaller than the page size (i.e., several items fit in one page), KVell prefixes items in the slab with a timestamp, the key size, and the value size. Items larger than 4K have a timestamp header at the beginning of each block on disk. For items that are less than the page size updates are done in place. For items that are larger, an update consists of appending the item to the slab and then writing a tombstone where the item used to be. When an item changes size, KVell first writes the updated item in its new slab and then deletes it from the old one.

5.3 In-memory data structures

Index. KVell relies on fast and lightweight in-memory indexes that have predictable insert and lookup times to find the location of items on disk. KVell uses one in-memory B tree per worker to store the location of items on disk. Items are indexed by (the prefix of) their key. We use prefixes and not hashes to preserve the order of keys for range scans. B trees offer poor performance to store medium/large items on disk but are fast when the data (mostly) fits in memory and keys are small. KVell leverages this property and only makes use of B trees to store lookup information (prefix and location information account for 13B).

KVell's tree implementation currently uses 19B per item on average (storing the prefix, location information, and the B tree structure), which amounts to 1.7GB of RAM to store 100M items. On YCSB workloads (1KB items), the index amounts to 1.7% of the database size. We find this value to be reasonable in practice. KVell currently does not explicitly support having part of the B tree flushed to disk, but B tree data is allocated from an `mmap-ed` file and can be paged out by the kernel.

Page cache. KVell maintains its own internal page caches to avoid fetching pages that are frequently accessed from persistent storage. The size of the page caches is a system parameter. The page cache remembers which pages are cached in an index and evicts pages from the cache in LRU order.

Ensuring that the lookups and insertions in the index have minimal CPU overhead is crucial for good performance. Our first implementation of the page cache used the fast `uthash` hashtable as an index. However, when the page cache is large, insertions in the hash can take up to 100ms (time to grow the hashtable), driving up the tail latency. Switching to a B tree eliminated these latency spikes.

Free list. When an item is deleted from a slab, its location in the slab is inserted in a per-slab in-memory stack, which we call the slab's *free list*. A tombstone is then written at the item location on disk. To bound memory usage, we only keep the last N freed positions in memory (N is currently set to 64). The goal is to limit memory usage while retaining the ability to reuse multiple free spots per batch of I/O without additional disk access.

When the $(N+1)^{th}$ item is freed, KVell makes its disk tombstone point to the first freed location. KVell then removes the first freed location from the in-memory stack and inserts the$(N+1)^{th}$ freed location. When the $(N+2)^{th}$ item is freed, its tombstone points to the second freed position, etc. In short, KVell maintains N separate stacks whose heads reside in memory, the rest being on disk. This allows KVell

to reuse up to N free spots per batch of I/O. With only one stack, KVell would have to read N tombstones in sequence from disk to find the next N freed spots.

5.4 Efficiently performing I/O

KVell relies on the asynchronous I/O API of Linux (AIO) to send requests to disk, in batches of up to 64 requests. By batching requests, KVell amortizes the overhead of system calls over multiple client requests. We choose to use Linux asynchronous I/O because it offers us a method to perform multiple I/Os with a single system call. We estimate that if such a call were available in the synchronous I/O API, performance would be roughly the same.

We reject two popular alternatives for performing I/O: (1) using `mmap` relying on the OS page cache (e.g., RocksDB), and (2) using read and write `direct I/O` system calls (e.g., TokuMX). Both techniques are less efficient than using the AIO interface. Table 3 summarizes our findings, presenting the maximum IOPS achievable on Config-Optane, writing 4K blocks randomly (which entails a read-modify-write on the device). The accessed data-set is 3 times larger than the available RAM.

Technique	IOPS
OS Page Cache + MMap (1 thread)	10K
OS Page Cache + MMap (8 threads)	60K
Read/write direct I/O (1 thread)	88K
Async I/O (1 thread, queue depth 1)	91K
Async I/O (1 thread, queue depth 64)	376K

Table 3. Config-Optane. Maximum IOPS depending on disk access technique.

The first approach is to rely on the OS-level page cache. In the single-threaded case, this approach has sub-optimal performance because it can only issue one disk read at a time when a page fault occurs (read ahead value is set to 0 because data is accessed randomly). In addition, dirty pages are only periodically flushed to disk. This results in sub-optimal queue depth most of the time, followed by bursts of I/Os. When the data-set does not completely fit in RAM, the kernel also has to map and un-map paged-out pages from the virtual address space of processes, which incurs significant CPU overhead. With multiple threads, the page cache suffers from locking overhead when flushing the LRU (on average one lock every 32KB flushed to disk) and from the speed at which the system can invalidate the TLB entries of remote cores. Indeed, when a page is un-mapped from the virtual address space, the virtual-to-physical mapping needs to be invalidated on all the cores that have accessed the page, incurring significant overhead due to IPI communication [33].

The second approach is to rely on direct I/O. However, direct I/O read/write system calls do not fill the disk queues when requests are done synchronously (1 pending request per thread). Since there is no need to handle the complex logic of mapping and unmapping pages from the virtual address space, this technique outperforms the `mmap` approach.

In contrast, batching I/O only requires a single system call per batch, and allows KVell to control the device queue length for low delay and high bandwidth.

Even though in theory I/O batching techniques could be applied to LSM and B tree KVs, the implementation would require significant effort. In B trees different operations may have conflicting effects on I/O (for instance, a split of a leaf caused by an insert, followed by the merge of two leaves). Moreover, data may be moved on disk due to re-ordering, also making asynchronous batched requests difficult to implement. Batching of write requests is already implemented in LSM KVs, via the memory component. However, batching slightly increases complexity on the read path because workers would have to ensure that all the files that need to be read are not removed by compaction threads.

Algorithm 1 Main KV path for single-page KV pairs.

```
1  Client thread:
2    worker_id1 = prefix(k1) % nb_workers
3    queues[worker_id1].push({k1, GET, callback1})
4    worker_id2 = prefix(k2) % nb_workers
5    queues[worker_id2].push({(k2,v2), UPDATE, callback2})
6
7  Worker thread:
8    push I/Os to disk (io_submit async I/O)
9
10   int processed_requests = 0;
11   while(request r = queues[my_id].pop()
12               && processed_requests++ < batch_size)
13     location = [file, index] = lookup(prefix(r.key))
14     page = get_page_from_cache(location)
15
16     switch r.action:
17      case GET:
18       if(!location || page.contains_data)
19         callback(..., page) // synchronous call
20       else
21         read_async(page, callback) // enqueue I/O
22       break;
23
24      case UPDATE:
25       file = get_file_based_on_size((k,v))
26       if(!location)
27         ... // asynchronously add item in file
28       else if(location.file != file) // size changed
29         ... // delete from old slab, add in new slab
30       else if(!page.contains_data) // page is not cached
31         // read data asynchronously first...
32         get({ (k,v), UPDATE, callback });
33       else // ...then update & flush page asynchronously
34         update cached page
35         write_async(location, callback)
36
37   events = get processes I/O (io_getevents async I/O)
38   foreach(e in events)
39     callback(..., e.page)
```

5.5 Client operation implementation

Algorithm 1 summarizes KVell architecture. For simplicity, the algorithm only shows single-page KV items. When a

request enters the system, it is assigned to a worker based on its key (Lines 3 and 5, Algorithm 1). Worker threads perform disk I/O and handle the logic of client requests.

Get(k). Reading an item (Lines 17–22, Algorithm 1) consists of getting its location on disk from the index and reading the corresponding page. If the page is already cached, there is no need to access persistent storage and the value is synchronously returned to the client. If not, the worker that handles the request pushes it in its I/O engine queue.

Update(k,v). Updating an item (Lines 24–35, Algorithm 1) consists of first reading the page where it is stored, modifying the value, and then writing the page to disk. Deleting an item consists of writing a tombstone value and adding the item location in the slab's free list. Freed locations are reused when new items are added, and items are appended if no free spot exists. KVell only acknowledges that an update has completed once the updated item has been fully persisted to disk (i.e., once the `io_getevents` syscall informs us that the disk write corresponding to the update has completed, Line 37 of Algorithm 1). The dirty data is immediately flushed to disk, as the page cache of KVell is not used to buffer updates. In this way KVell offers stronger persistence guarantees than state-of-the-art KVs. For instance, RocksDB only guarantees persistence at the granularity at which the commit log is synced to disk. In typical configurations syncs only happen in batches of a few updates.

Scan(k1,k2). Scanning consists of (1) getting the keys' locations from the indexes and (2) reading the corresponding pages. To compute the list of keys, KVell scans all the indexes: a thread briefly locks, scans and unlocks the indexes of all the workers in turn, and finally merges the results to get a list of keys to be read from the KV. Reads are then issued using a Get() command that bypasses the index lookup (because KVell has already accessed the indexes). Scans are the only operations that require sharing between threads. KVell returns the most recent value associated with each key touched by the scan. In contrast, both RocksDB and WiredTiger perform the scan on a KV snapshot.

5.6 Failure model and recovery

The current implementation of KVell is optimized for failure-free operation. In case of a crash, all slabs are scanned and the in-memory indexes are reconstructed. Even though the scan maximizes the sequential disk bandwidth, recovery can still take minutes on very large datasets.

If an item is present twice on disk (e.g., if a crash happened in the middle of migrating an item from a slab to another), only the most recent item is kept in the in-memory index and the other item is inserted in the free list. For items larger than the block size, KVell uses the timestamp headers to discard items that have been only partially written.

KVell is designed for drives that can write 4KB pages atomically, even in case of a power failure. This constraint can be lifted by avoiding in-place modification of pages, writing the new value in a new page and then adding the old location in the slab's free list once the first write has been fully acknowledged.

6 Evaluation

6.1 Goals

We evaluate KVell with a variety of production and synthetic workloads, comparing it against state-of-the-art KVs. The evaluation sets out to answer the following questions:

1. How does KVell compare to existing KVs on modern SSDs in terms of throughput, performance fluctuation and tail latency for reads, writes, and scans?
2. How does KVell perform on large databases and production workloads?
3. What are the trade-offs and limitations of using KVell in workloads that are beyond its design scope (small items, environments with extreme constraints on memory, and older drives)?

6.2 Experimental setting

Hardware. We use the same three hardware configurations described in Section 2. Our focus is Config-Optane since it is the configuration with the most recent drive out of the three. We also evaluate KVell in Config-Amazon-8NVMe, aiming to show the system behavior in a large configuration. Finally, we evaluate KVell in Config-SSD, showing the trade-offs of using KVell on older hardware.

Workloads. We use YCSB [10], and two production workloads from Nutanix. Our focus is on the YCSB benchmark, since it contains various types of workloads, providing a more complete view of KVell's behavior. Table 4 shows a summary of the YCSB core workloads. We evaluate both the uniform and the Zipfian key distribution for all YCSB workloads. The KV item size is 1024B, and the total data set size is approximately 100GB (100M keys) on the small test and 5TB (5B keys) on the large test. The production workloads are two write-intensive workloads, with a profile of 57:41:2 write:read:scan ratio. The KV item sizes range between 250B and 1KB, with a median of 400B. The total dataset size for the production workload is 256GB. The difference between the two workloads is the data skew: Production Workload 1 is closer to a uniform key distribution, while Production Workload 2 is more skewed.

Existing KVs. We compare KVell to four state-of-the-art systems: (1) RocksDB 6.2 [15], an LSM KV store developed by Facebook and heavily used in industry, (2) PebblesDB [43], a recent academic LSM KV store which makes significant improvements on the LSM housekeeping overhead (3) TokuMX [42], a B epsilon tree [7] supported as a

Workload	Description
YCSB A	write-intensive: 50% updates, 50% reads
YCSB B	read-intensive: 5% updates, 95% reads
YCSB C	read-only: 100% reads
YCSB D	read-latest: 5% updates, 95% reads
YCSB E	scan-intensive: 5% updates, 95% scans; average scan length 50 items
YCSB F	50% read-modify-write, 50% reads

Table 4. YCSB core workloads description.

storage engine by MongoDB [35], and (4) WiredTiger 3.2 [48] configured to use a B+ tree. We query WiredTiger using its LevelDB interface, which we ported from WiredTiger 3.1. We also tried CouchDB [2], an optimized B+ tree implementation, but do not report the results in the evaluation as it was consistently slower than TokuMX.

System configuration. All systems are allocated the same amount of memory, set with `cgroups`. The memory size is a third of the dataset size, to ensure that requests are served both from memory and from persistent storage. When the database is bigger than 3× the size of available RAM, we don't use `cgroups` (i.e., applications can use the whole available RAM). For the B trees, we configure the block size to be 4KB. Both LSM KVs are configured to have a maximum of 5 levels and two 128MB memory components (one active and one immutable). In addition, we set the write-ahead-log to buffer 1MB, so that it is only infrequently flushed and does not disadvantage the LSM KVs.

6.3 Results in the main experimental setting

6.3.1 Throughput

Figure 5 shows the average throughput of KVell and the competitor systems for YCSB in the uniform and Zipfian key distributions on Config-Optane.

Write-intensive workload (YCSB A and F). KVell outperforms TokuMX by 15×, WiredTiger and RocksDB by 8× and PebblesDB by 5.8× on the YCSB A workload (50% reads, 50%writes). In this workload, un-cached reads result in 1 I/O, cached reads in 0 I/Os, un-cached writes in 2 I/Os (1 read + 1 write) and cached writes in 1 I/O (1 write). Since the page cache contains 1/3 of the database, on average 1 request results in 1.17 I/Os on disk, which means a maximum theoretical throughput of 500K IOPS/1.17 = 428K requests/s. KVell maintains an average throughput of 420K requests/s. KVell thus utilizes the disk at 98% of its peak bandwidth without becoming CPU-bound, as shown in Figure 6. In this workload, KVell spends 20% of its time in B tree lookups done by the page caches and the in-memory indexes, 20% of its time in the I/O functions, and 60% of its time waiting.

As seen in Section 3, LSM KVs are limited by the cost of compactions. WiredTiger is limited by contention on the logs (accounting for 50% of the total cycles). TokuMX is limited by contention on shared data structures and unnecessary buffering (both accounting for 50% of the total cycles).

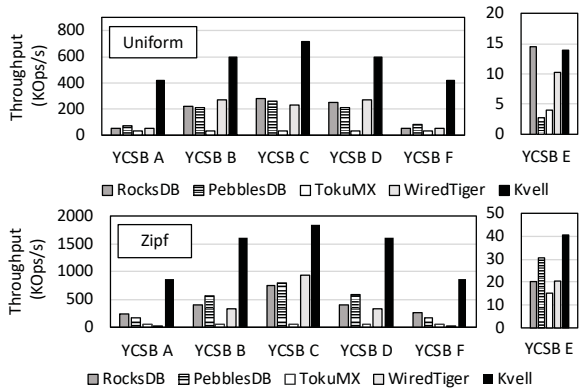

Figure 5. Config-Optane. Average throughput for YCSB workloads with uniform and Zipfian key distribution. KVell outperforms the next best competitor by up to 5.8x in write intensive and 2.2x in read-intensive workloads, while providing good scan performance (comparable to the best performing system on uniform and 32% better on Zipfian).

Read-intensive workloads (YCSB B, C, D). KVell outperforms existing KVs by 2.2× on YCSB B and 2.7× on YCSB C, utilizing disk at close to full IOPS. On YCSB C, KVell spends 40% of its time in lookups, 20% of its time in I/O functions and 40% of its time waiting. In these workloads, existing KVs perform sub-optimally because of their shared caches or because they do not batch read requests to disk (1 syscall per read). For instance, RocksDB spends up to 41% of its time in pread() syscalls and is CPU-bound.

Scan workloads (YCSB E). Maybe surprisingly, KVell performs well in the scan-intensive workloads, both in the uniform and the Zipfian key distributions. Note that for fairness we modified the YCSB workload so that it inserts keys in random order in KVell. By default, YCSB inserts keys in order.

On the Zipfian workload, KVell outperforms all systems by at least 25%. Because of the workload skew, the hot data is captured in KVell's cache. The shared-nothing design, together with the low-overhead cache implementation gives KVell an advantage over its competitors.

On the uniform workload, KVell outperforms PebblesDB by 5×, WiredTiger by 33%, and offers comparable performance to RocksDB (13.9K scans/s vs. 14.4K scans/s). Since KVell does not keep data sorted on disk, it accesses on average one page per scanned item. In contrast, RocksDB, which keeps the data sorted, accesses roughly speaking three items on each page (the page size is 4K and the item size is 1024K). The resulting maximum throughput for RocksDB is therefore approximately three times higher than that of KVell. Figure 7, which shows the throughput timeline for each of the systems,

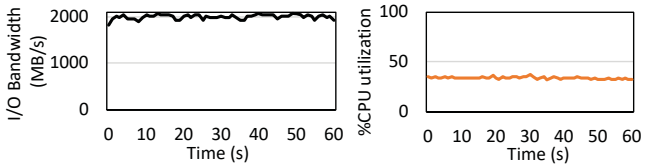

Figure 6. Config-Optane. KVell I/O bandwidth (left) and CPU utilization (right) on YCSB A, uniform. KVell uses the full disk I/O bandwidth without becoming CPU-bound.

confirms this reasoning. For YCSB E, RocksDB peak throughput reaches 55Kops/s versus a maximum of 18Kops/s for KVell. RocksDB maintenance operations interfere, however, with the client load, causing large fluctuations, while KVell's throughput remains steady around 15Kops/s. On average, therefore, RocksDB and KVell perform similarly.

6.3.2 Throughput over time

Figure 7 presents throughput over time for KVell, RocksDB, PebblesDB and WiredTiger. Throughput is measured every second. In addition to providing high average throughput, KVell does not suffer from performance fluctuations introduced by maintenance operations. In YCSB A, RocksDB's throughput drops to a minimum of 1.4K requests/s, PebblesDB's throughput to 10K requests/s, and WiredTiger's throughput to 8.5K requests/s. In the scan-intensive workload which only contains 5% updates, RocksDB's throughput drops to 1.8K scans/s and PebblesDB's drops to 1.1K scans/s. These drops occur frequently.

In comparison, KVell's performance stays constant after a short ramp-up phase (time for the page cache to fill up). KVell maintains a minimum of 400K requests/s on YCSB A and 15K scans/s after the ramp-up phase.

6.3.3 Tail latency

Table 5 presents the 99^{th} percentile and maximum latencies for KVell, RocksDB, PebblesDB, and WiredTiger. LSM KVs have a tail latency of over 9 seconds and WiredTiger has a maximum latency of 3 seconds. This is due to maintenance operations which directly impact the tail latency of LSM and B tree KVs. Such numbers are not atypical in LSM KVs and have been reported in previous work [6]. In comparison, KVell provides low maximum latency (3.9ms), while providing strong persistence guarantees.

Latency	KVell	RocksDB	PebblesDB	WiredTiger
99p	2.4ms	5.4ms	2.8ms	4.7ms
Max	3.9ms	9.6s	9.4s	3s

Table 5. 99^{th} percentile and maximum request latency on the YCSB A workload (write-intensive).

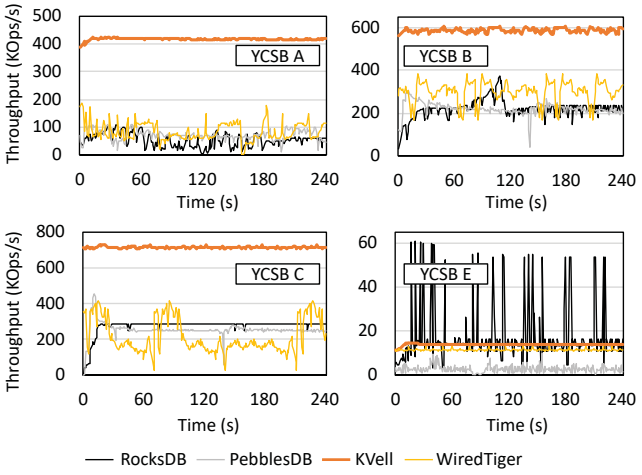

Figure 7. Config-Optane. Throughput timelines for KVell, RocksDB, PebblesDB, and WiredTiger for YCSB A (write-intensive), YCSB B (read-intensive), YCSB C (read-only) and YCSB E (scan-intensive) with uniform key distribution. Throughput is measured per second. KVell provides high and steady throughput, compared to LSM and B tree KVs which have significant fluctuations due to maintenance operations.

6.4 Alternative configurations and workloads

6.4.1 YCSB on Config-Amazon-8NVMe

Figure 8 presents the average throughput of YCSB for the uniform key-distribution. All systems are configured to use a 30GB cache (1/3rd of the database size). KVell outperforms RocksDB by 6.7× and PebblesDB by 8×, TokuMX by 13× and WiredTiger by 9.3× on average on YCSB A. For YCSB E, KVell slightly outperforms RocksDB and is faster than other systems. Though the profile of the results is similar to Config-Optane above, we can see how the performance gap increases between KVell and the competition, because KVell is capable of better utilizing the disks compared to the other systems which become CPU-bound.

AWS disks have different maximum IOPS depending on the read/write ratio. KVell maximizes disk IOPS on workloads that mix read and write operations. On these workloads, KVell spends 50% of its time waiting, 20% in lookups, 10% managing callbacks (`malloc` and `free`) and 20% of its time in I/O functions. KVell becomes CPU-bound in the uniform key distribution at 3.8m req/s, using 75% the 3.3m read-only IOPS that the disk can sustain. However, it is still 5.6× faster than existing KVs. Being CPU-bound on such a workload is expected on the Config-Amazon-8NVMe machine. This machine is unable to reach peak read-only IOPS when cores spend more than a marginal fraction of their time outside of I/O functions. In a microbenchmark, we measured that using more than 3us worth of CPU cycles per I/O request limits achievable IOPS to 75% of the maximum IOPS. Despite these limitations, KVell outperforms existing KVs even on scan

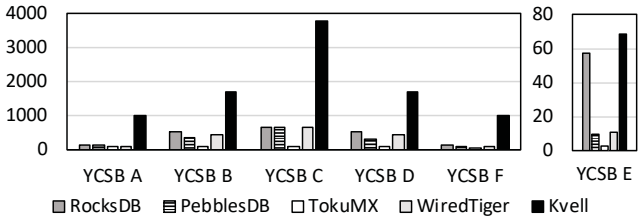

Figure 8. Config-Amazon-8NVMe. YCSB Average Throughput, uniform key distribution. KVell outperforms competitor systems across the board.

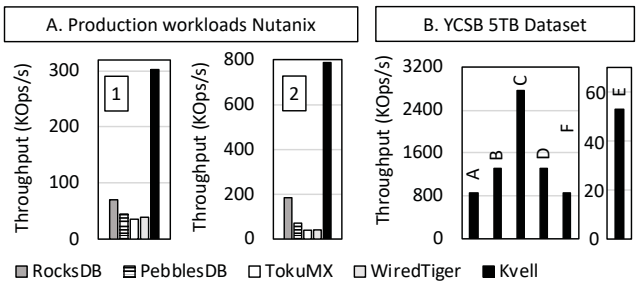

Figure 9. A. Config-Optane. Production workloads average throughput. KVell outperforms all competitor systems by approximately 4x in both workloads. **B. Config-Amazon-8NVMe**. KVell throughput on a YCSB 5TB dataset with uniform key distribution. KVell scales with the size of the dataset in all workloads.

workloads because of its lower overhead (i.e., no contention on page caches and batching requests to disk).

As in the Config-Optane case, LSM and B tree KVs suffer from fluctuating throughput and high tail latency (10+ seconds). In YCSB E, RocksDB's per second throughput frequently goes bellow 6K scans/s (vs. 57K scans/s average). RocksDB, PebblesDB, and WiredTiger can only sustain a fraction of their average speed while maintenance operations are running (e.g., 5% of average on YCSB A for RocksDB).

The maximum latency we observe in KVell is 20ms (99p 12ms) for YCSB A, and 12ms (99p 2.9ms) for YCSB C.

6.4.2 Nutanix production workloads

Figure 9 A presents the performance of RocksDB, PebblesDB, TokuMX, WiredTiger and KVell on two write-intensive production workloads at Nutanix. In the first production workload, 21% of the read requests are served from cache and in the second workload, 99% of the read requests are served from cache. KVell outperforms RocksDB (the next best-performing competitor) by approximately 4x in both production workloads in terms of average throughput. KVell also manages to maximize disk bandwidth in the first production workload and uses 78% of disk bandwidth in the second workload (in the second workload the disk is only used for updates, all other requests being served from cache).

Similarly to previous experiments, RocksDB presents large fluctuations in throughput over time, reaching as low as 3K requests/s. KVell on the other hand presents minimal fluctuations in throughput. In terms of latency, KVell outperforms RocksDB by up to 3 orders of magnitude for the maximum latency (8s for RocksDB compared to 2.5 ms for KVell) and by one order of magnitude at the 99^{th} percentile (12ms for RocksDB compared to 1.7 ms for KVell).

6.4.3 YCSB 5TB database on Config-Amazon-8NVMe

Figure 9 B shows the performance of KVell in the YCSB benchmark on a database containing 5B keys (5TB). Just as in the previous experiments, KVell is configured to use a 30GB page cache. We performed this experiment in order to test the scalability of KVell with the size of the dataset.

In this experiment, KVell only caches 0.6% of the items. Since keys are accessed uniformly, most reads and writes are served from disk. In YCSB A this results in 1.5 I/Os on average per request, i.e., a maximum of 1.4m IOPS/1.5 = 935K requests/s. KVell achieves 866K requests/s, 92% of peak bandwidth. On YCSB C and E, KVell performs up to 2.7m requests/s and 52K scans/s; numbers are slightly lower than on the 100M item dataset because less data is cached and lookups in the in-memory indices take on average 25% longer to complete.

6.5 Trade-offs and Limitations

6.5.1 Average latency

KVell submits I/O requests in batches. A trade-off is made between saturating IOPS (big batches) and minimizing average latency (small batches). For YCSB A on Config-Optane, KVell maximizes disk bandwidth with batches of 64 requests per worker and achieves an average latency of 158us. With a batch size of 32 requests, the average latency drops to 76us and disk is used at 88% of its maximum bandwidth.

6.5.2 Impact of item size

Item size does not impact the performance of `Get()` and `Update()` requests for items that are not cached. However, item size impacts the speed of scans for KVs that keep items ordered. Figure 10 shows the average throughput of RocksDB, RocksDB while performing compactions, and KVell, varying item size on YCSB E (scan dominated). Since KVell does not order items on disk, it reads on average one 4KB page per scanned item regardless of item size and has constant performance. For small items, RocksDB outperforms KVell because it reads fewer pages than KVell (64× less for 64B items). As the item size grows, the advantage of keeping items sorted decreases.

On all item sizes, RocksDB can only maintain a fraction of its throughput while compactions are running. While KVell

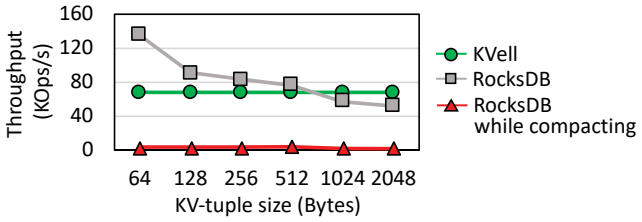

Figure 10. Config-Amazon-8NVMe. YCSB E (scan dominated) throughput for RocksDB and KVell.

is not designed to handle small items, it provides predictable performance in all configurations.

6.5.3 Memory size impact

Table 6 presents the number of lookups that the indexes can perform depending on the ratio of index size to available RAM on Config-Amazon-8NVMe. When the indexes fit in RAM, workers can do a total of 15 million lookups/s on a uniform workload (24M when the indexes are accessed in a Zipfian manner due to better caching of data). When the indexes are 5 times larger than allocated RAM, this number drops to 109K lookups/s on a uniform workload. The indexes can thus become a bottleneck when they do no fit in RAM.

KVell only supports having indexes partially flushed to disk to avoid crashing when indexes exceed available RAM but is not optimized to work in that situation. In practice, the memory overhead of the indexes is low enough so that they fit in RAM for most workloads (e.g., 1.7GB indexes for a 100GB dataset in the case of YCSB).

Index size / RAM	Zipf - Ops/s	Uniform - Ops/s
0.8	24M	15M
1.03	2.4M	1.4M
1.2	614K	540K
2.6	348K	156K
5.	280K	109K

Table 6. Config-Amazon-8NVMe. Number of operations (lookup or inserts) that the in-memory index can sustain in constrained memory environments with 100M keys.

6.5.4 Older drives (Config-SSD)

On Config-SSD, KVell is on par with LSM KVs for reads and writes, but offers relatively lower average performance on scans (3K scans/s for KVell vs. 15K scans/s for RocksDB and 5K scans/s for PebblesDB). On older drives, spending CPU cycles to optimize disk accesses is on average beneficial and no system is CPU-bound. Compactions still compete for disk resources, create latency spikes (18s+) and fluctuations in throughput (as extreme as 11 scans/s vs. 15K average on RocksDB). KVell's latency is only bounded by peak disk latency (100ms on Config-SSD). Using KVell vs. LSM KVs is thus a trade-off on older drives: if stability and predictability of latency are important, KVell is a better choice than LSM KVs, but for scan dominated workloads, LSM KVs provide higher average performance on older drives.

6.6 Recovery time

KVell's recovery time depends on the database size, while the recovery time of other systems mainly depends on the maximum size of the commit logs. For all systems, we use the default configuration for commits logs and we measure recovery time on the YCSB database (100M keys, 100GB). We simulate a crash by killing the database in the middle of a YCSB A workload and measure the databases' recovery times on Config-Amazon-8NVMe.

KVell takes 6.6s to scan the database and to recover from the crash, maximizing disk bandwidth. RocksDB and WiredTiger take 18s and 24s on average to recover, respectively. Both systems mainly spend time replaying queries from their commit logs and rebuilding indexes. Even though KVell is optimized for failure-free operation and has to scan the whole database to recover from a crash, its recovery time compares favorably to that of existing systems.

7 Related Work

7.1 KVs for SSDs

LSM [36, 39] is one of the most prevalent designs for write-optimized KVs, e.g., employed in LevelDB [11], RocksDB [15], HyperLevelDB [18], HyperDex [13], and Cassandra [16]. Much work had been done to adapt LSM KVs—originally designed for hard drives—to SSDs. WiscKey [30] and HashKV [8] are SSD-optimized by keeping keys sorted in the LSM tree, while values are stored separately in a log. Similarly to KVell, they explore the idea of breaking sequentiality, however, both systems still perform expensive background compactions which compete with the client operations.

PebblesDB [43] uses fragmented LSM trees to reduce the impact of compactions by postponing them to the last level of the LSM tree. SILK proposes an I/O scheduler for LSM KVs to decrease the impact of compactions on client request latency [6]. TRIAD [5] uses a combination of different techniques to reduce write amplification. Under high load, all three systems eventually need to run compactions, causing latency spikes and throughput drops for client operations.

SlimDB [45] leverages SSDs, improving the indexing and filtering scheme in LSM KVs to obtain good read performance. NoveLSM [21], PapyrusKV [22] and NVMRocks [14] adapt the LSM design to persistent memory, smoothing the transition between RAM and SSD. These LSM enhancements still preserve key sequentiality, unlike KVell.

BetrFS [19] and TokuMX [42] make use of Bϵ trees [7] to strike a balance between write-optimized and read-optimized data structures, still leveraging the sequential order of keys.

As discussed in Section 3, while trees are efficient for small keys, they perform poorly on medium and large keys, which is the target workload of KVell.

More generally, the observation that CPU overhead is limiting performance on fast disks has been made in previous work, and new KV designs have been proposed. SILT [28] is a KV designed for flash drives. SILT explors the idea of having small in-memory indexes to perform efficient lookups on disk but relies on expensive background operations to convert and merge data into HashStores and SortedStores on disk. In contrast, KVell requires no background operation to persist data on disk. Udepot [24] uses an in-memory hashtable to perform lookups of data stored on NVMe drives. Udepot uses locks to prevent races on disk pages and memory structures, a garbage collector to avoid fragmentation on disk, and does not support scans. Papagiannis et al. [40, 41] proposed alternative KV designs that favor random I/O to reduce CPU overhead on SSD and NVMe drives. In Tucana [40], Papagiannis et al. modified a Bϵ tree, removing buffering at the index level. Tucana still relies on buffering at the leaf level to achieve good performance and on the page cache to cache data. In KVell we show that buffering is no longer useful on fast drives, that the page cache severely limits performance, and that contention on a single shared data structure may lead to unnecessary overheads. In Kreon [41], Papagiannis et al. modify LSM KVs to reduce the overhead of compactions. They only order keys on disk, and merge them in the different levels of the LSM tree using small random I/Os. Kreon relies on buffering to achieve good performance, and only flushes its L_0 to disk every minute to achieve peak throughput. Similarly to traditional LSM designs, the accumulated maintenance work results in spikes of CPU usage which likely results in performance fluctuation. In contrast, KVell does not use any maintenance operation and offers stable throughput.

LOCS [47], BlueCache [51], and NVMKV [32] are efficient KVs on SSDs that expose FTL operations to the operating system. Such SSDs are rare and optimizations are tied to a specific hardware design. In this work we show that low-level hardware specific optimizations are not necessary and that it is possible to perform I/O operations with low latency and maximum disk bandwidth with generic system calls.

Mickens et al. [34] and Klimovic et al. [23] employ storage disaggregation to be able to utilize the full disk bandwidth in datacenters. KVell aims at utilizing full disk bandwidth on a single machine.

7.2 KVs for byte-addressable persistent memory

A large body of work proposes data structures optimized for byte-addressable persistent memory (PM) [4, 17, 50, 53, 54]. HiKV [50] is a hybrid KV store. HiKV focuses on improving the latency of requests performed to a hashtable stored in PM. The hashtable is used for fast lookups and a global B+ tree stored in RAM is used to speed up range queries. KVell faces different latency challenges on block devices, stores data unordered on disk to avoid expensive migrations and uses per-thread structures to avoid contention points in the system. Bullet [17] uses cross-referencing logs to create a seamless transition between DRAM and PM accesses in KV stores. Zuo et al. [54] propose a write-optimized hashing index scheme optimized for PM. This technique optimizes for point queries but does not support efficient range queries. Various tree algorithms have also been adapted to persist data directly in PM without the need for DRAM structures [3, 9, 25, 38, 46, 52]. In contrast, KVell focuses on fast block-addressable SSDs, which we believe will remain the more cost-effective choice for large data stores. SLM-DB [20] combines the LSM design and a B tree to leverage PM. Like KVell, it maintains a B tree index fast lookups (SLM-DB's index lives in PM, while KVell maintains it in DRAM). SLM-DB relies on compaction operations to sort persisted data.

7.3 KVs for in-memory data stores

Similarly to KVell, MICA [29] and Minos [12] employ a partitioned design, assigning non-overlapping shards of KV items to each system thread. However, instead of partitioning on the key range, Minos partitions on the KV item size. Masstree [31] uses a combination of concurrent B+ trees and tries, emphasizing efficient use of caches. Unlike KVell, Masstree assumes the entire working set fits in memory. RAMCloud [37] is a DRAM-based KV store that emphasizes fast parallel recovery. Li et al. [27] develop a full-stack in-memory KV store that achieves high throughput, taking into account hardware properties to create an efficient design – hardware properties are also taken into account by KVell.

8 Conclusion

Existing KV store designs are efficient on older generation SSDs, but perform sub-optimally on modern SSDs. We have shown that a streamlined approach relying on a share-nothing architecture, unsorted data on disk, and batches of random I/Os outperforms existing KVs on fast drives, even for scan workloads. We have prototyped these ideas in KVell. KVell provides high and predictable performance and strong latency guarantees.

Acknowledgements. We would like to thank our shepherd, Michael Kaminsky, and the anonymous reviewers for all their helpful comments and suggestions. This work was supported in part by the Swiss National Science Foundation Grants No. 513954 and 514009 and by a gift from Nutanix, Inc.

References

[1] Amitanand S Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. 2012. Storage Infrastructure Behind Facebook Messages: Using HBase at Scale. *IEEE Data Eng. Bull.* 35, 2 (2012).

[2] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: The Definitive Guide: Time to Relax.* " O'Reilly Media, Inc.".

[3] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proceedings of the VLDB Endowment* 11, 5 (2018).

[4] Joy Arulraj and Andrew Pavlo. 2019. Non-Volatile Memory Database Management Systems. *Synthesis Lectures on Data Management* 11, 1 (2019).

[5] 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 ATC*.

[6] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of USENIX ATC*.

[7] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to Bϵ-trees and Write-Optimization. *;login:* 40, 5 (2015).

[8] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *Proceedings of USENIX ATC*.

[9] Shimin Chen and Qin Jin. 2015. Persistent B+-trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (2015).

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of SoCC*.

[11] Jeffrey Dean and Sanjay Ghemawat. 2018. LevelDB. https://github.com/google/leveldb.

[12] Diego Didona and Willy Zwaenepoel. 2018. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *Proceedings of NSDI*.

[13] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A Distributed, Searchable Key-Value Store. In *Proceedings of SIGCOMM*.

[14] Facebook. 2017. NVMRocks: RocksDB on Non-Volatile Memory Systems. http://istc-bigdata.org/index.php/nvmrocks-rocksdb-on-non-volatile-memory-systems.

[15] Facebook. 2018. RocksDB: a Persistent Key-Value Store for Fast Storage Environments. https://rocksdb.org.

[16] Apache Software Foundation. 2018. Cassandra NoSQL Key-Value Store. http://cassandra.apache.org/.

[17] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *Proceedings of USENIX ATC*.

[18] Hyperdex. 2018. HyperLevelDB. https://github.com/rescrv/HyperLevelDB.

[19] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: Write-Optimization in a Kernel File System. *ACM Transactions on Storage (TOS)* 11, 4 (2015).

[20] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *Proceedings of FAST*.

[21] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of USENIX ATC*.

[22] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. 2017. PapyrusKV: a High-Performance Parallel Key-Value Store for Distributed NVM Architectures. In *Proceedings of SC*.

[23] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of EuroSys*.

[24] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the Performance of Fast NVM Storage with uDepot. In *Proceedings of FAST*.

[25] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of FAST*.

[26] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-memory Data Management Beyond Main Memory. In *Proceedings of ICDE*.

[27] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. 2016. Achieving One Billion Key-Value Requests Per Second on a Single Server. *IEEE Micro* 36, 3 (2016).

[28] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of OSDI*.

[29] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of NSDI*.

[30] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of FAST*.

[31] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of EuroSys*.

[32] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of USENIX ATC*.

[33] Mel Gorman 2015. mm: Send one IPI per CPU to TLB Flush All Entries After Unmapping Pages. https://lore.kernel.org/patchwork/patch/575960/.

[34] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. 2014. Blizzard: Fast, Cloud-Scale Block Storage for Cloud-Oblivious Applications. In *Proceedings of NSDI*.

[35] MongoDB. 2018. MongoDb. https://www.mongodb.com/.

[36] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (1996).

[37] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. 2011. Fast Crash Recovery in RAMCloud. In *Proceedings of SOSP*.

[38] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of SIGMOD*.

[39] John Ousterhout and Fred Douglis. 1989. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. *ACM SIGOPS Operating Systems Review* 23, 1 (1989).

[40] 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 ATC*.

[41] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of SoCC*.

[42] Percona. 2018. TokuMX. https://www.percona.com/software/mongo-database/percona-tokumx.

[43] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees. In *Proceedings of SOSP*.

[44] RedisLabs. 2019. Redis: In-Memory Data Structure Store, Used as a Database, Cache and Message Broker. https://redis.io/.

[45] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-efficient Key-value Storage Engine for Semi-Sorted Data. *Proceedings of VLDB Endowment* 10, 13 (2017).

[46] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of FAST*.

[47] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design and Implementation of LSM-Tree Based Key-Value Store on Open-Channel SSD. In *Proceedings of EuroSys*.

[48] WiredTiger. 2018. WiredTiger. http://www.wiredtiger.com/.

[49] WiredTiger. 2019. WiredTiger Caching and Eviction. http://source.wiredtiger.com/3.2.0/tune_cache.html.

[50] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: a Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of USENIX ATC*.

[51] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, et al. 2016. Bluecache: A Scalable Distributed Flash-Based Key-Value Store. *Proceedings of the VLDB Endowment* 10, 4 (2016).

[52] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of FAST*.

[53] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. 2015. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *ACM SIGARCH Computer Architecture News*, Vol. 43.

[54] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of OSDI*.