SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores

Alexander Conway, *Rutgers University and Vmware Research;* Abhishek Gupta, *DropBox;* Vijay Chidambaram, *University of Texas at Austin and VMware Research;* Martin Farach-Colton, *Rutgers University;* Richard Spillane, *VMware;* Amy Tai and Rob Johnson, *VMware Research*

https://www.usenix.org/conference/atc20/presentation/conway

978-1-939133-14-4

SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores

Alex Conway[*] Abhishek Gupta[†] Vijay Chidambaran[‡] Martin Farach-Colton[§]
Rick Spillane[¶] Amy Tai[||] Rob Johnson[||]

June 5, 2020

[1]

Abstract

Modern NVMe solid state drives offer significantly higher bandwidth and lower latency than prior storage devices. Current key-value stores struggle to fully utilize the bandwidth of such devices. This paper presents SplinterDB, a new key-value store explicitly designed for NVMe solid-state-drives.

SplinterDB is designed around a novel data structure (the STB$^{\varepsilon}$-tree) that exposes I/O and CPU concurrency and reduces write amplification without sacrificing query performance. STB$^{\varepsilon}$-tree combines ideas from log-structured merge trees and B$^{\varepsilon}$-trees to reduce write amplification and CPU costs of compaction. The SplinterDB memtable and cache are designed to be highly concurrent and to reduce cache misses.

We evaluate SplinterDB on a number of micro- and macro-benchmarks, and show that SplinterDB outperforms RocksDB, a state-of-the-art key-value store, by a factor of 6–10× on insertions and 2–2.6× on point queries, while matching RocksDB on small range queries. Furthermore, SplinterDB reduces write amplification by 2× compared to RocksDB.

1 Introduction

Key-value stores form an integral part of system infrastructure. Google's LevelDB [22] and Facebook's RocksDB [8] are widely used, both within their companies and outside. Their importance has spurred research into several aspects of key-value store design, such as increasing write throughput, reducing write amplification, and increasing concurrency [1–3, 6, 8–17, 19, 21, 22, 26, 30–32, 34, 35, 37, 39, 42, 43, 45–47, 50–52].

Existing key-value stores face new challenges with the increasing use of high-performance NVMe solid state drives (SSDs). NVMe SSDs offer high throughput (500K-600K IOPS) and low latency (10-20 microseconds).

LevelDB and RocksDB struggle to utilize all the available bandwidth in modern SSDs. For example, we find that for the challenging but common case of small key-value pairs, RocksDB is able to use only 30% of the bandwidth supplied by an Optane-based Intel 905p NVMe SSD (even when using 20 or more cores).

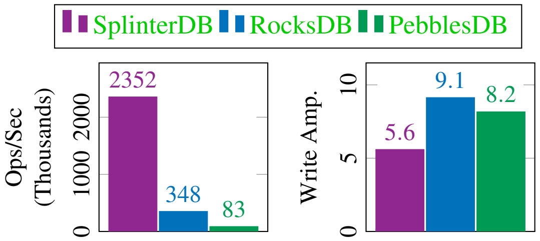


Figure 1: YCSB load throughput and write amplification benchmark results with 24-byte keys and 100-byte values.

We find that the bottleneck has shifted from the storage device to the CPU: reading data multiple times during compaction, cache misses, and thread contention cause RocksDB to be CPU-bound when running atop NVMe SSDs. Thus, there is a need to redesign key-value stores to avoid these CPU inefficiencies.

We present SplinterDB, a key-value store designed for high performance on NVMe SSDs. On workloads with small key-value pairs, SplinterDB is able to fully utilize the device bandwidth and achieves almost 2× lower write amplification than RocksDB (see Figure 1). We show that compared to state-of-the-art key-value stores such as RocksDB and PebblesDB, SplinterDB is able to ingest new data 6–28× faster (see Figure 1) while using the same or less memory. For queries, SplinterDB is 1.5-3× faster than RocksDB and PebblesDB.

Three novel ideas contribute to the high performance of SplinterDB: the ***STB$^{\varepsilon}$-tree***, a new compaction policy that exposes more concurrency, and a concurrent memtable and user-level cache that removes scalability bottlenecks. All three components are designed to enable the CPU to drive high IOPS without wasting cycles.

At the heart of SplinterDB is the STB$^{\varepsilon}$-tree, a novel data structure that combines ideas from log-structured merge trees and B$^{\varepsilon}$-trees. The STB$^{\varepsilon}$-tree adapts the idea of size-tiering (also known as fragmentation) from key-value stores such as Cassandra and PebblesDB and applies them to B$^{\varepsilon}$-trees to reduce write amplification by reducing the number of times a data item is re-written during compaction. The STB$^{\varepsilon}$-tree also introduces a new ***flush-then-compact*** policy that increases

[1] [*]Rutgers University and VMware Research Group; [*]Dropbox, Inc.; [‡]The University of Texas at Austin and VMware Research Group; [§]Rutgers University; [¶]VMware, Inc.; [||]VMware Research Group; {aconway, vchidambaram, robj, rspillane, taiam}@vmware.com; abhi.gupta0290@gmail.com; martin@farach-colton.com

compaction concurrency across the entire tree and exploits locality in the insertion workload to accelerate insertions. By enabling fine-grained, localized compactions, STB$^\varepsilon$-trees push ideas from PebblesDB to their logical conclusion.

Concurrency at the data structural level could be wasted if the data structure is accessed through a cache with poor concurrency. We designed a new user-level concurrent cache for SplinterDB that uses fine-grained, distributed reader-writer locks to avoid contention and ping-ponging of cache lines, as well as a direct map to enable lock-free cache operations. All the data read and written by SplinterDB flows through this concurrent cache.

SplinterDB is not without limitations. Like all key-value stores based on size-tiering, SplinterDB sacrifices the performance of small range queries, although less than one might expect. For large range queries, SplinterDB can use the full device bandwidth. Similarly, size-tiering is known to temporarily increase space usage until multiple versions of a single data item are compacted together. Finally, SplinterDB was designed for the most stringent requirements: small key-value pairs and restricted memory. In cases where key-value pairs are large or memory is plentiful, other choices may prove as good as SplinterDB, and we make some of those comparisons below.

In summary, the contributions of SplinterDB are as follows:

- We introduce the STB$^\varepsilon$-tree, which reduces write amplification and enables fine-grained concurrency in compaction operations (sections 2 to 3).
- We design and build a highly-concurrent memtable that is able to drive enough operations to the underlying STB$^\varepsilon$-tree (section 5).
- We combine the STB$^\varepsilon$-tree, memtable, and user-level cache in SplinterDB, a key-value store that can fully utilize NVMe SSD bandwidth. (section 6).

2 High-Level Design of STB$^\varepsilon$-trees

The basic STB$^\varepsilon$-tree design has three high-level goals:

- Handle inserts using bulk I/O, so that inserts are bandwidth-bound.
- Minimize the number of times each key-value pair gets read or written, so as to reduce write amplifiction, I/O amplification (i.e. read and write amplification), and CPU costs of inserts.
- Maintain sufficient indexing information so that, under normal conditions, each query requires at most one I/O.

In section 3 and section 4, we explain how we refine tree operations to support high concurrency.

The STB$^\varepsilon$-tree shares many ideas with LSM trees, B$^\varepsilon$-trees, and external-memory hash tables from the theory literature. See section 7 for details.

2.1 Overall Structure

The STB$^\varepsilon$-tree is a tree-of-trees, as shown in fig. 2. The backbone of the STB$^\varepsilon$-tree is the ***trunk tree*** (or just trunk). Each node of the trunk has pointers to a collection of B-trees, called

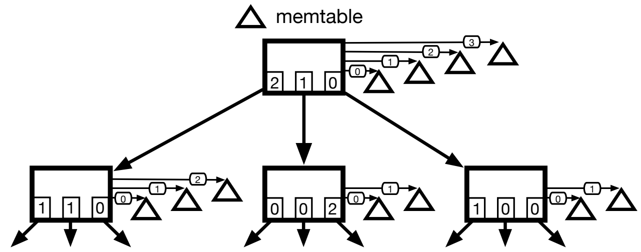


Figure 2: The structure of a STB$^\varepsilon$-tree. The rectangles represent trunk nodes and the triangles represent branch trees and the memtable. The inset boxes indicates the first active branch for each pivot, referencing the pointer labels to the branches.

branch trees (or just branches). The branches store all of the actual key-value pairs in the dataset. Each branch also has an associated quotient filter, which serves the same purpose as Bloom filters in LSM trees. Trunk nodes have a fanout of up to F (typically 8 to 16), which is also an upper bound on the number of branch trees. Branch trees have a fanout determined by the number of pivot keys that can be packed into a 4KB node.

The overall STB$^\varepsilon$-tree also has a memtable, which is used to buffer insertions, as explained below. The memtable is also a B-tree, just like the branches.

Within a trunk node, the branches are numbered from oldest to youngest, i.e. all the key-value pairs in branch i were inserted before any of the key-value pairs in branch $i+1$. For example, in fig. 2, the root of the trunk tree has four branches, numbered 0 through 3, with branch 0 being the oldest.

Furthermore, stored with each child pointer c in a trunk node is an integer a_c indicating the oldest branch that is ***active*** for that child. Inactive branches are ignored during queries, as explained below. So, for example, in fig. 2, only branches 2 and 3 are active for the root trunk node's leftmost child, branches 1, 2, and 3 are active for its middle child, and all branches (0 through 3) are active for its rightmost child.

2.2 Queries

Queries begin by searching in the memtable. If the queried item is not found in them memtable, then the query proceeds down the trunk tree. Recall that all the data is stored in the branches, and trunk nodes only contain metadata and pointers to branches and filters.

When a query for a key k arrives at a trunk node t, it first searches the pivots of t to determine the correct child c for k. It then iterates over the active branches for c, from youngest to oldest. For each branch b, it first queries b's associated quotient filter. If the quotient filter indicates that k is definitely not in b, then the query moves on to the next branch. Otherwise, is queries for k in b. If it finds a hit, it returns the result to the caller. Otherwise, it moves on to the next branch. If none of the branches contain k, then the query recurses to c.

Analysis. We now explain why queries take at most one I/O in common configurations: those which use at least 32-byte key-value pairs and have RAM which is at least 10% of the dataset size.

The memory used by filters and branch tree indices is bounded as follows. Quotient filters use about 1–2 bytes per key, so the overhead of quotient filters is greatest when key-value pairs are small. With 32-byte key-value pairs, quotient filters will be about 6% of the total database size. Branch trees will have a very high fanout, e.g. ≈ 128 for 4KB nodes, so the interior of the branch trees will be less than 1% of the total database size. Trunk nodes contain only metadata about the branches and filter, and so use negligiable RAM. Therefore all the indexing information will fit comfortably in a RAM that is $\approx 7\%$ of the total database size. For larger keys, e.g. 256 bytes, the branch trees will have a fanout of only about 16, and hence the interior nodes could be up to 6% of the total database size. However, the quotient filters will be less than 1% of the data size, so the indexing data will still be less than 10% of the database size.

Thus the only I/Os during a query will be to load leaves of branch trees. However, the false positive rate of quotient filters with two bytes per key is $< 1\%$, which is low enough to ensure that most queries do not encounter any false positives from the quotient filters that they query. Hence most queries will query exactly one branch tree, which will contain the desired key-value pair. Queries for keys that are not present in the database will usually require no I/Os at all.

2.3 Insertions

When an item is inserted, it is first buffered in the memtable. When the memtable is full, it is added to the root as a new branch, say branch i. We also construct a quotient filter for the memtable at this time. We call the process of adding the memtable to the root of the trunk an ***incorporation***.

The size of the memtable affects performance in the following ways. If the memtable gets too large, then some of its nodes will get evicted from RAM and, once enough nodes spill, a workload of random inserts would require essentially one random I/O per insert, contradicting our goal of handling inserts using bulk I/O. For most systems, this means the memtable should be kept substantially smaller than RAM, e.g. at most a few gigabytes for typical hardware configurations.

On the other hand, the memtable will eventually become a branch, and we want branches to be large enough that scanning a branch can use bulk I/O (i.e. if a branch consisted of only a handful of nodes, then reading the entire branch would be bottlenecked on I/O overheads, rather than bandwidth). Branch scanning performance is critical for compactions and range queries (see Sections 2.4 and 2.6). For most storage devices, it is sufficient to ensure the branches (and hence the memtable) are at least a few megabytes in size.

Thus, for most systems, the memtable can be anywhere from a few megabytes to a few gigabytes in size. Since we are specifically interested in building a system that is robust to low-memory situations, and since making the memtable larger has diminishing returns in terms of scanning throughput, we select a maximum memtable size m that is just comfortably large enough to ensure efficient scanning performance. In our prototype, $m = 24\text{MB}$.

2.4 Flushing and Compaction

We cannot keep adding new branches to the root trunk indefinitely. Eventually, the root will fill up and have no more room for branch pointers. This solved by compacting data and flushing it to its children.

This section describes a basic version of flushing and compaction which captures the basic underlying mechanics of a STB$^\varepsilon$-tree. In section 3, we describe SplinterDB's more-involved flush-then-compact policy which leverages its B$^\varepsilon$-tree structure to expose more compaction concurrency and optimize non-random insertion workloads.

In this simplified version, when a trunk node is full, data is removed from it by repeatedly ***compacting*** some of its branches into a single branch and ***flushing*** the resulting branch to a child until the parent is no longer full (see fig. 3). As in LSM trees, compaction is necessary to keep queries fast. Without compaction, the number of branches that must be queried would grow without bound.

A node p is considered to be full when its branches contain $F \times m$ bytes of active key-value pairs, where F is the fanout and m is the memtable capacity in bytes. When p becomes full, the child c with the most active key-value pairs is chosen and p is flushed into c. We construct a new branch b by compacting all the branches in p that are active for c. Note that the branches in p may contain keys for any of p's children, not just c, so when we compact the branches for a flush to c, we scan over only the portions of each branch that contain keys destined for c. We then add b to c as c's youngest branch. We also build a quotient filter for b and store a pointer to the filter in c. Finally, we mark all the branches in p as inactive for c, so they will not be flushed to c again. Any branches of p that were active only for c are no longer active for any of p's children and can be garbage collected.

Since branches are large, compacting the branches for c can proceed at disk bandwidth. Furthermore, we always flush to the child with the most pending items in the parent. This ensures that the resulting branch b will have at least m items in it, and hence will be efficient to scan when we, at some point in the future, flush it from the child to one of its children.

Analysis. Each time a key-value pair participates in a compaction, it moves one level down the trunk tree. Thus the worst-case write amplification of the STB$^\varepsilon$-tree is $O(\log_F N)$, which is the same as in a size-tiered LSM tree. Level-tiered LSM trees and (normal) B$^\varepsilon$-trees have a substantially larger write amplification of $O(F \log_F N)$. In Section 3, we describe our flush-then-compact strategy, which enables some key-value pairs to skip compaction at some levels, particularly when the workload exhibits locality.

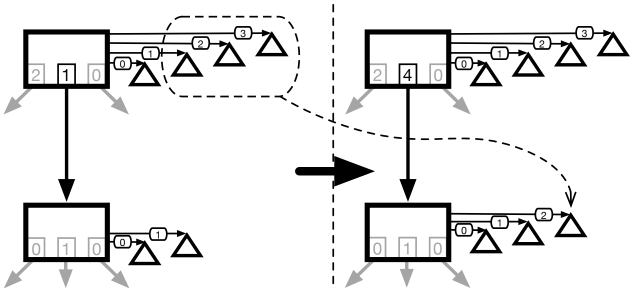


Figure 3: In a STB^{ε}-tree, a flush to a pivot P consists of compacting its active key-value pairs (from its branches) into a new branch in the child. The dashed arrow indicates compaction.

2.5 Splitting

When a trunk leaf is full, it is split, and similarly when a trunk internal node has more than F pivots, it is split. As in standard B-trees and B^{ε}-trees, the I/O costs of splitting and merging do not asymptotically change the costs of inserts. See Section 4 for how we make splitting and merging compatible with hand-over-hand locking in STB^{ε}-trees.

2.6 Iterators and Scans

To construct an iterator starting from key k, we walk the trunk search path for k, constructing B-tree iterators (also starting at k) for each active branch along the path. We then construct a merge iterator on top of the B-tree iterators, which simply returns the smallest key from among all of the underlying B-tree iterators. The merge iterator can be efficiently implemented using a heap.

While constructing the B-tree iterators, we also compute an upper bound u for the leaf of k's search path. As soon as the merge iterator returns a key that is greater or equal to u, we tear down the merge iterator and all the B-tree iterators and rebuild them, starting from u.

2.7 Deletions and Updates

Deletions are implemented through tombstone ***messages***, i.e. a key-value pair with special value indicating that the key has been deleted. More generally, the STB^{ε}-tree supports update messages that encode a function to be applied to the value associated with a key.

3 Flush-then-Compact

This section describes the flush-then-compact algorithm, which improves the I/O and CPU concurrency of compactions during flushing and improves the performance of update workloads with locality, such as sequential workloads. The idea behind flush-than-compact is to decouple the flushing step from the compaction step, as shown in fig. 4.

In a flush, the references to the child's active branches are copied from the parent to the child, along with references to their quotient filters, as shown in fig. 4. The child's active branch counter in the parent is updated to reflect the flush, just as before. At this point the parent and child are in a consistent state and any locks can be released. Since a flush is just a pointer swing, write locks are held very briefly.

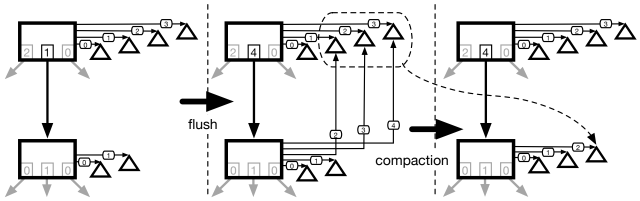


Figure 4: With the flush-then-compact policy, flushes are broken into two steps. First references to the active branches are flushed to the child and removed from the parent (by setting the pivot's active branch number). Then those branches are compacted by an asychronous process. The compaction stage is performed without holding a lock on the node, and during this time it can still flush, be flushed into, split and by queried.

Note that branches can now be referenced by multiple trunk nodes, so branches are reference counted.

From here, if the child is full, we will perform a flush from the child to its children, before initiating any compactions (hence the name "flush-then-compact"). This flush will copy the newly arrived branches from the child to one or more of its children, exactly the same way that the branches were flushed to the child. This process can repeat recursively.

Once all the flushes have completed, we schedule background jobs to compact all the new branches at each node that received a flush. The background jobs will construct the new branches and need to acquire write locks only to replace the old branch pointers with a pointer to the newly created branch.

Flush-then-compact accelerates non-random workloads. Since we perform flushes before compactions, some of the branches involved in a compaction at node p may already be inactive for some of p's children when we perform the compaction. This means we can skip over those keys when we compact those branches. And, since branches are stored as B-trees, we can skip over those key ranges efficiently. Thus we effectively avoid compacting those keys at p's level in the STB^{ε}-tree.

To see why this accelerates non-random insertion workloads, consider an extreme case: a workload of insertions all for a single trunk leaf, ℓ. For simplicity, assume also for the moment that all the nodes on the path from the root to ℓ have zero branches. The memtable will repeatedly fill with key-value pairs for ℓ and get incorporated into the root. Once the root fills, it will flush to its child c along the path to ℓ. This will cause c to immediately become full and flush to its child. This process will repeat until all the branches arrive at ℓ.

At that point, the system will schedule background compactions for each trunk node along the path from the root to ℓ. However, at each node other than ℓ, there will be no live data in any of the branches. The compactions will thus skip all the data in the branches, resulting in empty branches at each interior node. Consequently the interior nodes will again be left with zero branches. Thus only ℓ will have a non-degenerate compaction. As a result, the new key-value pairs will participate in only one compaction, and hence the STB$^\varepsilon$-tree will have a write amplification of 1 for this workload.

Now consider the case when the nodes along the path to ℓ do not have zero branches. The first few times the root fills, it may choose to flush to some child c' that is not along the path to ℓ. This would happen if the root happened to contain more items for c' than c. However, as long as the workload consists only of items for ℓ, eventually the root will contain more items for c than for any of its other children. From that point forward, it will always flush to c. Then the same process will repeat at c. Eventually, each time the root fills, the system will flush all the new branches to ℓ, as described above.

Thus this flushing protocol automatically adapts to the insertion workload without knowing a priori what that workload is. Furthermore, if the workload changes then, after some time, the flushing decisions will adapt to the new workload automatically.

Furthermore, this flushing algorithm exploits less-than-perfect locality automatically. For example, suppose the insertion workload consists almost entirely of key-value pairs for ℓ, but a few random items for other leaves. Almost every time the root fills, it will flush to c, and the process described above will occur. However, each flush will leave a few new items in the root. This cruft will accumulate, eventually causing the root to flush to a child other than c, cleaning out some cruft. After that, the root will resume flushing to c until enough cruft accumulates again. Thus most data will get flushed directly to ℓ, and hence have a write amplification of 1, but a small amount of data will get compacted at every level.

As these examples show, the flush-then-compact algorithm is much more robust than the special-case optimizations frequently implemented for sequential insertions. Special-case optimizations can be foiled by a few random insertions sprinkled into a sequential workload. Flush-then-compact, on the other hand, exploits *locality* rather than *sequentiality*. In section 6.3 we show empirically that flush-then-compact enables SplinterDB to outperform other systems on near-sequential workloads.

Flush-then-compact exposes concurrency. Flush-then-compact improves concurrency by setting up several compactions and then launching them simultaneously, which improves both CPU and I/O parallelism. The hierarchical nature of the B$^\varepsilon$-tree structure makes it trivially safe to perform compactions concurrently at different trunk nodes. In a standard compact-then-flush approach, each time the root is flushed, it initiates a single compaction. The system would not start another compaction until the root is filled (and flushed) again.

4 Preemptive Splitting for STB$^\varepsilon$-trees

Splits and merges pose problems for hand-over-hand locking in B-trees (and B$^\varepsilon$-trees). Hand-over-hand locking proceeds from root to leaf, but splits and merges proceed from the leaves up.

An approach to solving this issue in B-trees is to use preemptive splitting and merging [40]. During a B-tree insert, if a child already has the maximum number of children, then it is split while the insertion thread still holds a lock on its parent. Then the insertion can release the parent's lock and proceed down the tree, assured that the child will not need to split again as part of this insertion. Analogously, deletions merge a child with one of its neighbors if the child has the minimum number of children. This works because insertion and deletions can increase or decrease the number of children of a node by at most 1.

This approach does not work in B$^\varepsilon$-trees, because a flush to a leaf could cause that leaf to split multiple times. In STB$^\varepsilon$-trees with flush-then-compact, we can move all pending messages along a root-to-leaf path to the leaf before performing any compactions, splits, or merges. The total number of messages moved to the leaf is bounded by $O(Fm\log_F N)$, i.e. the capacity of a trunk node times the height of the tree. The leaf can therefore split into as many as $O(\log_F N)$ new leaves of size B. Similarly, a collection of flushes full of delete messages to several leaves of a single parent can reduce the parent's number of children by $O(\log_F N)$. In practice, $\log_F N$ is less than 10 for typical fanouts $F \approx 8$ and dataset size $N \leq 2^{80}$ key-value pairs.

We extend preemptive splitting and merging to STB$^\varepsilon$-trees as follows. We reserve space in each node to accommodate up to $F+H$ children, where H is an upper bound on the tree height, e.g. $H = 10$. We then apply preemptive splitting, except we preemptively split a node during a flush if its fanout is above F. For merges, we take a similar approach. If, during a flush, we encounter a node with less than $F/2$ children, then we merge or rebalance it with one of its siblings.

Thus all operations on the STB$^\varepsilon$-tree—flushes, compactions, splits, and merges—proceed from root to leaf and can therefore use hand-over-hand locking.

The mechanisms for flush-then-compact make it easy to handle branches during splits. Recall that each branch can be marked dead or alive for each child, and branches are refcounted and hence can be shared by multiple trunk nodes. Thus we can split a trunk node by simply giving its new sibling references to all the same branches as the node had before the split. In the new node, we copy the liveness information for each branch along with the children that are moved to the new sibling.

5 From STB$^\varepsilon$-trees to SplinterDB

In this section, we discuss the details of SplinterDB's implementation, which addresses the concurrency and memory bottlenecks associated with driving NVMe devices to full bandwidth.

5.1 Branch Trees and Memtables

SplinterDB uses the same B-tree implementation for both its branches and its memtables, although there are some differences to optimize for their use cases.

Branch trees, extents, and pre-fetching. When a branch is created from a compaction, its key-value pairs are packed into the leaves of the B-tree, and the leading edge of internal nodes are created to index them. The nodes in each level are allocated in extents of 32 pages, and the header of each node stores the address of the following node, but also of the next extent. In this way, the nodes of each level form a singly linked list.

Iteration through a branch is performed by walking the linked list formed by its leaves. Whenever the iterator reaches the beginning of a new extent, it issues an asynchronus prefetch request for the next extent. The extent length is configurable to tune to the latency of the storage device.

Memtables. The basic design of the memtables mirrors that of the branch B-trees, but includes some optimizations to increase their insertion performance and concurrency.

As in the case of the static branch trees, the nodes on each level of the memtable form a singly-linked list, and nodes are allocated in extents. However, because nodes are created on demand as nodes split, we do not try to guarantee that successive nodes reside in the same extent. Furthermore, since memtables are almost always in RAM, we do not perform prefetching during memtable traversals.

The memtable uses hand-over-hand locking together with preemptive splitting, as described by Rodeh [40]. To increase concurrency, write locks are only obtained on internal nodes when a split is required.

To ensure locks are held briefly, especially on nodes near the top of the tree, the tree uses a new technique called ***shadow splitting***. To split a node c, a write lock is obtained on c and the parent p. We allocate a physical block number (PBN) n for the new sibling, c' and add it as part of a new pivot in p. However, in the cache, we initially point n to c. At this point, we can release all locks on p. Now, we fill in the contents of c', update the PBN n to point to c' in the cache, and then release all locks on c'. Finally, we upgrade to a write lock on c, truncate its child list (via a metadata operation) and then release all locks on c.

5.2 User-level Cache and Distributed Locks

SplinterDB has a single user-level cache which keeps recently accessed pages in memory. Almost all the memory that SplinterDB uses comes from this cache, so pages from all parts of the data structure—trunk node pages, branch pages, filter pages and memtable pages—are all stored there. Only cache and file-system metadata, as well as small allocations used to enqueue compaction tasks are allocated from system memory.

This design allows nearly all the free memory to be used for whichever operations are being performed, so that parts of the data structure which are not in use can be paged out.

The cache at a high level is a clock cache, but with several features designed to improve concurrency.

Each thread has a thread-local hand of the clock, which covers 64 pages. The thread draws free pages from the hand, and if it has exhausted them, it acquires a new hand from a global variable using a compare-and-swap. It then writes out dirty pages from the hand which is a quarter turn ahead, and evicts any evictable pages in its new hand. Thus threads clean and evict pages from distinct cache lines within the cache metadata, avoiding contention and cache-line ping-ponging.

SplinterDB uses distributed reader-writer locks [24] to avoid cache-line thrashing between readers. Briefly, a distributed reader-writer lock consists of a per-thread reader counter and a shared write bit. Each reader counter is on a separate cache line to avoid cache-line ping-ponging when readers acquire the lock. Writers set the write bit (using compare and swap) and then wait for all the read counters to become zero. Readers acquire the lock by incrementing their read counter and then checking that the writer bit is 0. If it is not, they decrement their reader counter and restart.

Distributed reader-writer locks allow readers to scale essentially perfectly linearly, at the cost that acquiring a write lock is expensive. However, the design of SplinterDB makes writing rare enough that this is a good trade-off.

We make distributed reader-writer locks space efficent by storing each thread's reader counters in an array indexed by cache-entry index. Each reader counter is one byte, so the total space used by locks is $t \times c$ bytes, where t is the number of threads and c is the number of cache entries.

SplinterDB supports three levels of lock: read locks, "claims", and write locks. A claim is a read lock that can be upgraded to a write lock. Only one thread can hold a claim at a time. After obtaining a read lock, a thread may try to obtain a claim by trying to set a shared claim bit with a test-and-set. If this fails, they must drop the read lock and start over. Otherwise, they can upgrade their claim to a write lock by setting a shared write bit and waiting for all the read counters to go to zero.

5.3 Quotient filters

Bloom filters [7] are the standard filter for most LSMs [8, 22, 39]. However, the cost of Bloom filter insertions can dominate the cost of sorting the data in a compaction. Therefore modern key-value stores often use more efficient filters; for example, RocksDB uses blocked Bloom filters [38];

Similarly, SplinterDB uses quotient filters [4, 5, 36] instead of Bloom filters. A full presentation of quotient filters is out of scope for this paper, but we review their salient features for SplinterDB. See Pandey, et al. for a full presentation on quotient filters [36]. The key feature of quotient filters is that, like blocked Bloom filters, each insert or query accesses $O(1)$ cache lines (and hence $O(1)$ page accesses). Quotient filters are roughly as space efficient as Bloom filters—for the range of parameters used in SplinterDB, quotient filters use between $0.8\times$ and $1.2\times$ the space of a blocked Bloom filter. We view the space as essentially a wash. Quotient filter inserts and lookups also require only one hash function computation. In past work, quotient filter insertions and queries were shown to be 2-4× faster than in a Bloom filter.

A quotient filter for set S stores, without error, $h(S) = \{h(x) \mid x \in S\}$, where h is a hash function. Since the quotient filter stores $h(S)$ exactly, all false positives are the result of collisions under h. Thus each insertion or lookup requires only one hash function computation. Furthermore, a quotient filter stores the elements of $h(S)$ in sorted order in a hash table using a variant of linear probing. Thus most inserts and lookups in a quotient filter access only 1 or 2 adjacent cache lines. As a result, insertions and lookups in quotient filters are typically 2-4× faster than in a Bloom filter. Finally, quotient filters are space efficient, using slightly less space than Bloom filters whenever the false positive rate ε is less than $1/64$, which is typical. For example, a quotient filter with $\varepsilon = 0.1\%$ uses about 10% less space than a Bloom filter [36].

SplinterDB further reduces the CPU costs of filter building during compaction by using a bulk build algorithm. During the merging phase of compaction or when inserting into a memtable, SplinterDB builds an unsorted array of all the hashes of all the tuples compacted or inserted. The array is then sorted (by hash value) and the quotient filter is built. Since the quotient filter also stores the hashes in sorted order, this means that the process of inserting all the hashes is a linear scan of the sorted array and of the quotient filter. Hence it has good locality and can benefit from cache prefetching.

5.4 Logging and Recovery

SplinterDB uses per-thread write-ahead logical logging for recovery. By using per-thread logs, we avoid contention on the head of a single, shared log.

The challenge is to resolve the order of operations across logs after a crash. For this, we use a technique similar to "cross-referenced logs" [25]. Our scheme works as follows. Each leaf of the memtable has a generation number. Whenever a thread inserts a new message into the memtable, it records and increments the generation number of the memtable leaf for the inserted key. It then appends the inserted message to its per-thread log, tagged with the leaf's generation number. During recovery, the generation numbers in the logs give a total order on the operations performed on each leaf (and hence on all the keys for that leaf), so that the recovery procedure can replay the operations on each key in the proper order. When a leaf of the memtable splits, the new leaf gets the same generation number as the old leaf.

6 Evaluation

We evaluate the performance of SplinterDB on several microbenchmarks and on the standard YCSB application benchmark [20]. We compare this performance against that of two state-of-the-art key-value stores, RocksDB and PebblesDB. The following questions drive our evaluation:

- How much does SplinterDB improve insertion performance? To what extent is improvement achieved through reduced write amplification and other factors?
- Despite being size-tired, how much does SplinterDB mitigate [range] query performance? Can SplinterDB utilize device bandwidth for large range queries?
- How much faster are sequential (or otherwise local) insertions in SplinterDB? Do they have lower write amplification?
- Do point lookups scale with the number of threads?

6.1 Setup and Workloads

All results are collected on a Dell PowerEdge R720 with a 32-core 2.00 GHz Intel Xeon CPU, 256 GiB RAM and a 960GiB Intel Optane 905p PCI Express 3.0 NVMe device. The block size used was 4096 bytes.

In general, we use workloads derived from YCSB traces with 24B keys. We generally use 100B values, but also include a set of YCSB benchmarks for 1KiB values. We instrumented dry runs of YCSB in order to collect workload traces for the load and A–F YCSB workloads and replay them on each of the databases evaluated. In order to eliminate the overhead of reading from a trace file during the experiment, the trace replayer `mmaps` the trace file before starting the experiment. We use the same traces for each system.

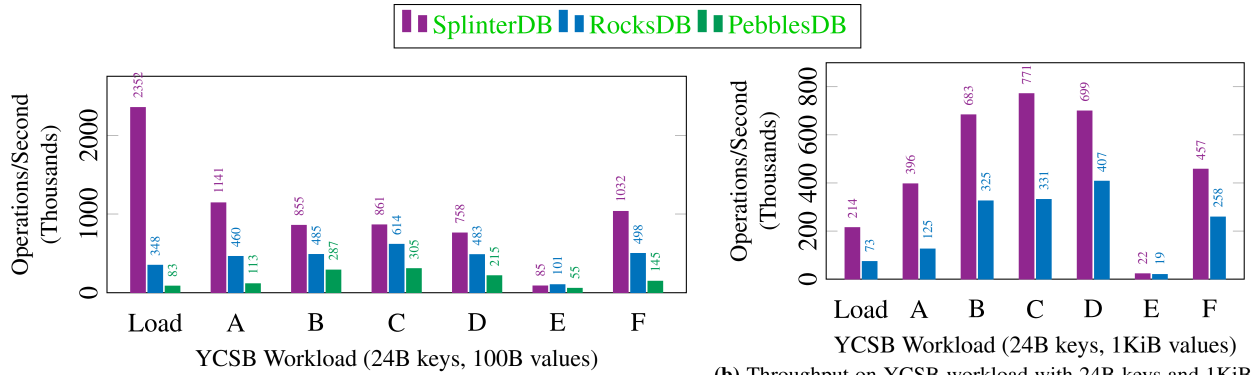
In general, we limit the available memory to 10% of the dataset size or less. In order to perform the benchmarks on reasonably sized datasets, we restrict the available system memory with a type 1 Linux `cgroup`, sized to the target memory size plus the size of the trace, which we pin so that it cannot be swapped out. Unless otherwise noted, the target memory size is 4GiB. PebblesDB has an apparent memory leak, which causes it to consume the available memory, so we allow it to use the full system memory. On the YCSB load benchmarks, this causes it to swap for a small portion at the end, but this was less than 10% of the run time.

Unless otherwise noted, SplinterDB uses a max fanout of 8, a memtable size of 24MiB and a total cache size of 3.25GiB. The difference between this cache size and the target memory size of 4GiB is to accommodate other in-memory data structures maintained by SplinterDB.

Each system is run with the thread count which yields the highest throughput. RocksDB is configured to use background threads equal to the number of cores minus the number of foreground threads, with a minimum of 4. PebblesDB uses its default number of background compaction threads. SplinterDB is configured without background compaction threads.

6.2 YCSB

We measure application performance using the Yahoo Cloud Services Benchmark (YCSB). The core YCSB workloads consist of load phases and run phases. The load phases create a dataset by inserting uniformly random key-value pairs. The run phases emulate various workload mixes. Workload A is 50% updates, 50% reads, workload B is 95% reads, 5% updates), workload C is 100% reads, workload D is read latest (95% reads, 5% insertions), workload E is short range scans (95% scans, 5% insertions) and workload F is read-modify-writes (50% reads, 50% RMWs).



(a) Throughput on YCSB workloads with 24B keys and 100B values. Load is 673M operations, E is 20M operations and others are 160M operations. Higher is better.

(b) Throughput on YCSB workload with 24B keys and 1KiB values. Load is 84M operations, E is 1.3M operations and others are 10M operations. Higher is better.

Figure 5: YCSB throughput and I/O benchmark results.

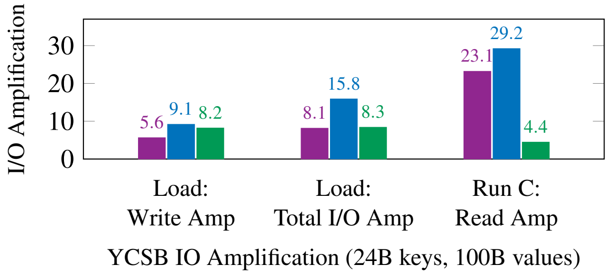


Figure 6: IO amplification on YCSB load and Run C workloads, as measured with iostat. Lower is better.

The results with 100B values and 1KiB values are shown in Figure 5 (both workloads use 24B keys). Figure 6 shows the write and I/O amplification in the 100B-value benchmark.

On the load phase, SplinterDB is faster than RocksDB by almost an order of magnitude. Because of size-tiering and its compaction/flushing policy SplinterDB has about 1/2 the write amplification of the other systems. Note PebblesDB performs almost no reads because it was given unlimited memory. Surprisingly PebblesDB does not show substantially lower write amplification than RocksDB.

On the run phases, which the exception of E, SplinterDB is 40–150% faster than RocksDB, the next fastest system. On E, SplinterDB is roughly 15% slower than RocksDB in the 100B-value case, and about 15% faster than RocksDB in the 1KiB-value benchmark.

Latency. SplinterDB maintains high throughput without sacrificing latency. Table 1 reports insertion latency for SplinterDB and RocksDB. Unsurprisingly, the latency of RocksDB is at least 3x that of SplinterDB on all metrics. This is because mechanisms such as flush-and-compact (Section 3) improve concurrency and eliminate stalls on the write path.

Table 2 reports read latency for SplinterDB and RocksDB. SplinterDB read latency is comparable to RocksDB, because the quotient filters (section 5.3) in SplinterDB behave similarly to Bloom filters in RocksDB.

KVell. KVell [33] is a key-value store also designed to utilize full NVMe bandwidth. It has an in-memory B-tree index that maps all keys to disk page offsets. It does well on large (4KiB) key-value pairs, but on small key-value pairs, the overhead of the in-memory index becomes a significant fraction of the dataset size. In particular, it was impossible to run KVell in a memory `cgroup` of 4GiB. Figure 7 shows KVell's performance on the YCSB workload with 100B values, for different memory sizes. At 22GiB, which is around the size of the in-memory index, KVell's performance starts to drop. At 20GiB, KVell becomes unusable. Therefore in realistic memory settings, KVell is not a viable option for the small key-value sizes that SplinterDB targets.

system	mean	median	P95	P99
SplinterDB	7.0	3.1	12.4	27.7
RocksDB	40.2	29.7	50.5	86.7

Table 1: Insertion latency (μs) for the workload in fig. 5a.

system	mean	median	P95	P99
SplinterDB	46.4	13.3	126.3	216.1
RocksDB	51.1	28.8	108	221.1

Table 2: Read latency (μs) for the workload in fig. 5a.

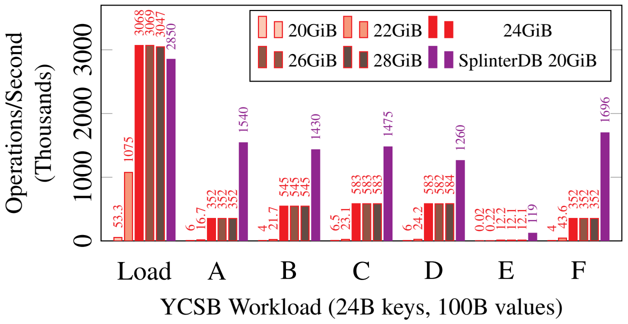


Figure 7: Throughput of Kvell on YCSB workloads with varying amounts of available RAM, 100B values. Throughput of SplinterDB with 20GiB RAM shown for comparison. Load consists of 673M operations, E consists of 20M operations and all other workloads consist of 160M operations. Higher is better.

SplinterDB is designed to work well even under low-memory scenarios (less than 10% of total data size). However, we also run the YCSB experiment with higher memory, 20 GiB, to compare with KVell in a regime where KVell per-

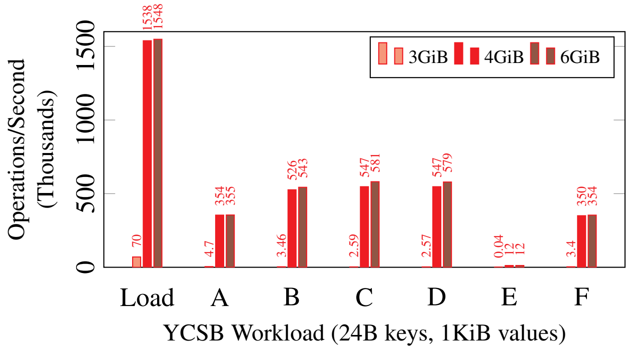


Figure 8: Throughput of Kvell on YCSB workloads with varying amounts of available RAM, 1 KiB values. Load consists of 84M operations, E consists of 1M operations and all other workloads consist of 10M operations. Higher is better.

forms well. As shown in fig. 7, we find that SplinterDB almost matches KVell on insertions, but outperforms KVell by roughly a factor of 2.5 on queries.

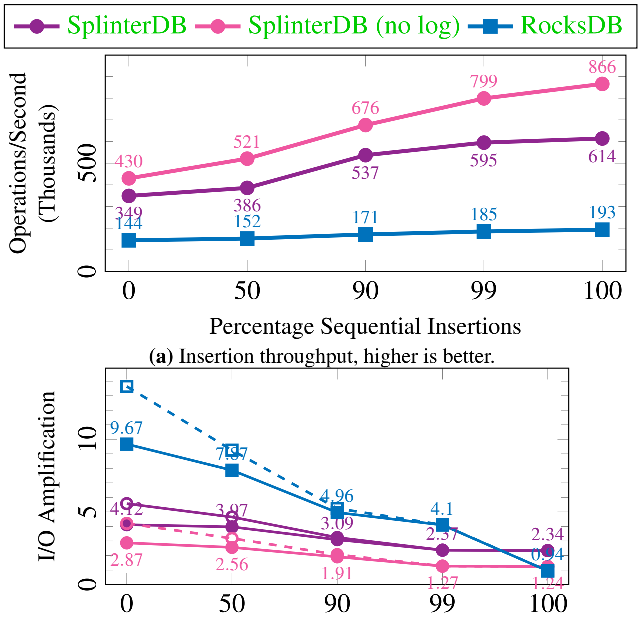
For larger values, the memory cliff for KVell is much lower. We run the same YCSB workload on both systems, but with 1KiB values. In this case, KVell's memory cliff is between 3GiB and 4GiB, as shown in fig. 8. For these larger values, KVell outperforms SplinterDB insertions (see Figure 5b) due to a low write amplification, but still can only achieve 55-77% query throughput of SplinterDB on the other YCSB workloads. KVell's range-query performance (workload E) is particularly lower than SplinterDB's because KVell does not keep key-value pairs sorted. Thus each 1KiB key-value pair in the range requires a separate, random 4KiB I/O, resulting in a read amplification of about 4×. Splinter, on the other hand, sorts and packs key-value pairs into 4KB blocks, for a read amplification close to 1 during range queries.

As soon as the memory cliff hits, KVell exhibits the same performance drop as in the previous experiment. However, when values are so large, this may not be so important, since indexing information can easily fit in RAM.

6.3 Sequential Insertion Performance

Because of the flush-then-compact policy, we expect SplinterDB's performance will improve substantially on insertion workloads with a high degree of locality (see section 3). We demonstrate this by performing 20GiB of single-threaded insertions from a trace composed of interleaved sequential and random keys in different proportions. For comparison, we perform the same workload on RocksDB.

As shown in fig. 9a, SplinterDB's performance improves smoothly from 349K insertions per second for a purely random workload to 614K insertions per second for a purely sequential workload, which is 76% faster. This improvement is partially obscured by the log, which adds a constant additive IO overhead. If we disable the log, SplinterDB improves from 430K insertions per second on a purely random workload to 866K operations per second on a purely sequential workload, 100% faster. Note that we would expect the intermediate throughputs in the best case to be the [weighted] harmonic mean of the pure cases, because they are rates. At 50% random, 50% sequential for SplinterDB with no log this is 575K insertions/second, so its actual performance of 521K insertions/second captures a substantial amount of the potential improvement.



(a) Insertion throughput, higher is better.

(b) Write amplification (solid) and total IO amplification (dashed) as measured with `iostat`. Lower is better.

Figure 9: Single-threaded insertion throughput by varying mixed sequential/random locality percentage. X-axis not to scale.

RocksDB also improves as the workload becomes more sequential, but this effect is much smaller, a 35% speedup. Furthermore, RocksDB shows less than 20% speedup until the workloads becomes 99% sequential.

Figure 9b shows that as predicted, SplinterDB incurs less IO amplification on more sequential workloads. With the log disabled, its write amp approaches 1 as the workload approaches purely sequential. In contrast, while RocksDB also has less IO amplification on more sequential workloads, it still incurs write amplification of 4.1 even when 99% of the keys are sequential. It is only when the workload becomes 100% sequential that the write amplification becomes close to 1 (because of caching it even falls below 1).

6.4 Concurrency Scaling

SplinterDB is designed to scale with the number of available cores up to the performance limits of the storage device. This is especially true for reads, where the use of distributed reader-writer locks and a highly concurrent cache design, together with a careful avoidance of dirtying cache lines, can avoid almost all contention between threads.

Read Concurrency. We test the read concurrency scaling of SplinterDB by running YCSB workload C with 160M key-value pairs, where, as in fig. 5a each instance of the test divides

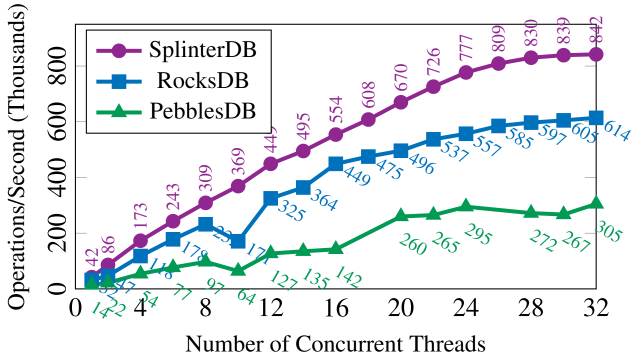


Figure 10: Read concurrency: read throughput (YCSB workload C) by number of threads. Each instance performs 160M reads divided evenly between threads. Higher is better.

the keys evenly into *N* batches, which are then performed in parallel by *N* threads. The results are in fig. 10.

The results show nearly linear scaling—throughput with 24 threads is 18.5× the single-threaded throughput. Between roughly 24 and 32 threads, the scaling flattens out, but at that point the measured throughput is 2.07–2.24 GiB/sec, which is 88–95% of the device's advertised random read capability.

While RocksDB also scales well, its throughput with 24 threads is 17.4× its single-threaded throughput, and with 32 threads it uses 91% of the device's advertized random read capability. Therefore, even though SplinterDB can perform more operations per second, RocksDB is still making nearly full use of the device for reads. We conclude here that SplinterDB is making better use of the available memory for caching, since it has noticeably lower read amplification. Finally, PebblesDB is unable to scale with more threads, flattening out at around 300K reads/sec.

Insertion Concurrency We test the insertion concurrency scaling of SplinterDB by running the YCSB load workload with 673M key-value pairs divided into *N* batches, each of which is inserted in parallel by a different thread. fig. 11 reports throughput for various *N*.

The results show that SplinterDB scales almost linearly up to 10 threads. With 10+ threads, it performs 2.0-2.4M insertions per second with IO amplification around 7.5, which implies that it uses 1.9-2.2GiB/sec of bandwidth, which is at or near the device's sequential bandwidth of 2.2GiB/sec.

RocksDB's insertion performance also scales as the number of threads increase up to 14 threads, by a factor of 2.7. At its peak, it uses 754GiB/sec of bandwidth. PebblesDB scales slightly as well. For both RocksDB and PebblesDB, as many background threads as available are used for flushing and compaction during this benchmark.

6.5 Scan Performance

An inherent disadvantage of size-tiering is that short scans must search every branch along the root-to-leaf path to the starting key. Each of these searches is likely to incur an IO to the device. As a result, as seen in fig. 5a, SplinterDB with 124B key-value pairs has scan throughput on small ranges

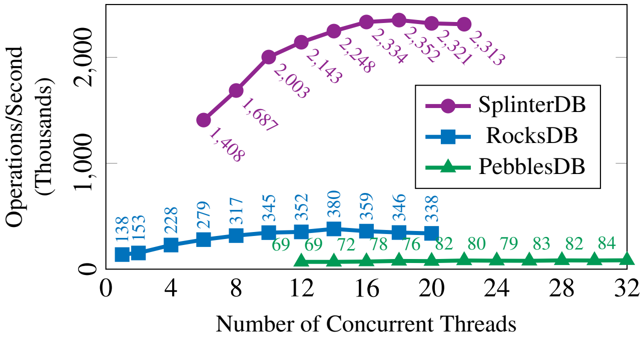


Figure 11: Insertion concurrency: insert throughput (YCSB Load) by number of threads. Each instance performs 673M writes divided evenly between threads. Higher is better.

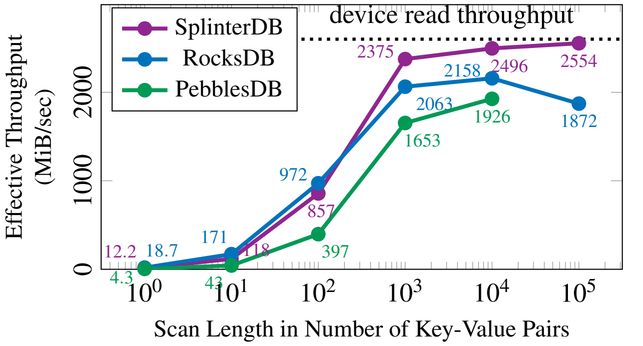


Figure 12: Scan throughput in MiB/sec as a function of scan length. For small scans, the start up cost dominates, but as the scans get longer, the throughput approaches the device's advertised bandwidth (2.6GiB/sec). The x-axis is on a log scale. Higher is better.

that is about 85% that of RocksDB. During that workload, SplinterDB performed 2.26 GiB/sec of IO, which is within 96% of the devices advertised random read capability (short scans of small key-value pairs are essentially random reads).

However, once the initial search for the successor to the starting key has completed, the root-to-leaf path within each relevant branch will be in memory. Together with prefetching, this allows subsequent keys to be fetched at near disk bandwidth. Therefore, we expect that scans have a relatively high startup cost for the search to the starting key, followed by a very low iteration cost of obtaining subsequent keys.

Thus, when the amount of data requested grows to multiple pages, the disadvantage begins to dissipate. One way this happens is with larger key-value pairs: with 1kib values, SplinterDB is about 16% faster than RocksDB.

Another way this can happen is with scans of more key-value pairs. We modify YCSB workload E to have only fixed-length scans of *N* key-value pairs, where *N* is 1, 10, 100, 1K, 10K or 100K. We perform runs of 10M scans of length 1, 10 and 100, 1M scans of length 1000, 100K scans of length 10000 and 10K scans of length 100000. Each run is performed on a dataset of 80GiB (with 24B keys and 100B values) and 4GiB memory.

The result is shown in fig. 12. Short scans on SplinterDB have low effective bandwidth, and in fact the bandwidth scales close to linearly with the scan length for scans of up to 100 key-value pairs. This suggests that for scans of this length, the startup cost dominates the iteration cost, which is as expected. As the scan length increases, the effective bandwidth of the scans approaches the device's advertised sequential read bandwidth, delivering 91% at scans of 1,000 key-value pairs. At scans as small as 100 key-value pairs, SplinterDB returns data at nearly half the bandwidth of the device.

7 Related Work

The STB$^{\varepsilon}$-tree is based on a B$^{\varepsilon}$-tree, a data structure that has been used in several file systems and databases [18, 27–29, 44, 48, 49]. The closest work to ours is Tucana [37], a B$^{\varepsilon}$-tree optimized for SSDs. They also focus on CPU cost, concurrency, and write amplification. Our work pushes this to the even more demanding case of NVMe devices. SplinterDB improves on techniques that have been applied to log-structured merge (LSM) trees and key-value stores to reduce write amplification and increase concurrency.

Size-Tiering. Cassandra [19], Scylla [42] PebblesDB [39], and RocksDB [8] (in "universal compaction" mode) use size-tiering to reduce write amplification. Size tiering delays compaction of sorted runs in order to reduce write amplification. This can harm query performance because queries must search more runs to find the queried item. Fluid LSMs [16], Dostoevsky [16], LSM bushes [17], and Wacky [17] use hybrids between size-tiering and level-tiering to tune the trade-off between write amplification and query performance. See [39] for a survey of LSM-compaction schemes.

Size-tiering also decreases write amplification in SplinterDB. Because of the design of the STB$^{\varepsilon}$-tree, SplinterDB further leverages size-tiering for flush-and-compact, which greatly increases the concurrency of background operations.

Write amplification vs. range queries. Several systems sacrifice range-query performance in order to reduce write amplification in other ways. Wisckey [34] reduces write amplification by declustering their key-value store: they log values and only store keys in the LSM-Tree. Since values are stored on disk in arrival order, a range query must gather values from the log. On NVMe, this is not a problem once the values are 4KB or larger. However, for smaller values, this can induce huge read amplification, limiting range query performance to a tiny fraction of device bandwidth. HashKV [10] builds on Wisckey by introducing hash-based data-grouping to further reduce write amplification, but inherits Wisckey's range query performance limitations.

Other approaches improve write amplification by sacrificing range queries altogether. Conway et al. [14] describe a write-optimized hash table, called the BOA, that also uses size-tiering with an LSM. They also introduce the concept of a routing filter, which extends the functionality of Bloom filters, in order to speed up queries. The principle advantage of routing filters is that performance does not degrade as much when they don't fit in RAM. The BOA meets a provable lower bound on the I/O costs of insertions and queries [26]. The downside is that the BOA does not support range queries, which are crucial to many key-value-store applications. LSM-tries [46] organize the LSM tree using tries, resulting in reduced write amplification. However, LSM-tries do not support range queries.

Other approaches. Researchers have also attempted to reduce write amplification by exploiting special hardware features such as flash translation layers [35] and vector interfaces [45]. VT-Tree [43] uses indirection to avoid copying data that is already sorted, similar to "trivial moves" in RocksDB and PebblesDB. TRIAD [1] reduces write amplification by holding hot keys in memory, delaying compaction until different runs have significant key overlap, and by reducing redundancy between log and LSM tree writes. All these techniques are orthogonal to our work and can be used in conjunction with our techniques.

Concurrency is also an important aspect of key-value store performance. One of the first works in increasing concurrency in LSM-based stores was cLSM [21] which introduces a new compaction algorithm. Zuo et al. [52] show how to tune a cuckoo hash for NVM. Such a scheme suffers from high write amplification, since each insertion must re-write all keys in a data block. Zuo et al. do not report write amplification numbers but instead focus on concurrency.

Kourtis, et al. describe several systems-level optimizations for improving key-value-store throughput on NVMe, such as efficient use of user-level asynchronous I/O and low-latency scheduling [31]. Their techniques are largely orthogonal to the work in this paper.

8 Analysis

We begin with a disk-space analysis, showing that, in STB$^{\varepsilon}$-tree, size-tiered compaction and flush-then-compact do not blow up the on-disk space usage by more than a constant factor. We then use this to analyze memory usage from indexes and filters, and finally summarize STB$^{\varepsilon}$-tree's asymptotic performance.

Disk-space. Like level-tiered and size-tiered LSM trees and B$^{\varepsilon}$-trees, the STB$^{\varepsilon}$-tree can have a space overhead when there are updates to existing keys. This is because all of these data structures buffer updates and apply them lazily. We begin by showing that the space used by the STB$^{\varepsilon}$-tree is $O(N)$, where N is the number of distinct keys in the database. This compares quite favorably to the space of a size-tiered LSM, which can be as bad as $\Theta(FN)$.

Theorem 1. *Let N be the number of distinct keys in a STB$^{\varepsilon}$-tree. Then the STB$^{\varepsilon}$-tree uses $O(N)$ space on disk.*

Proof. We give only a sketch. The four key observations in the proof are that (1) every leaf must be at least half full of distinct keys due to the splitting and compaction policy, (2) each branch has size at most mF due to the flushing policy, (3) each non-leaf trunk node references at most $3F$ branches due to the flushing policy, and (4) the number of non-leaf trunk nodes is at most $O(1/F)$ times the number of leaves. Together, these prove that the total amount of data referenced in the interior of the tree is at most a constant factor times the number of distinct keys in the leaves.

□

For a workload of random updates to existing keys, we estimate that the space blowup would be roughly a factor of 3. If the workload also contains insertions of new keys, then the blowup would be even lower.

Asymptotic analysis. The height of the trunk is $O(\log_F N/Fm)$, and each item gets compacted at most once per level, so the I/O complexity of random insertions are $O(\frac{\log_F N/Fm}{B})$, which is the same as in a size-tiered LSM tree.

Assuming that all index nodes and filters fit in RAM, the I/O complexity of random point queries is $O(1)$ I/Os, since the filters will eliminate all but the correct branch from being searched.

Long sequential insertion workloads will cost $O(1/B)$ I/Os per item. The I/O efficiency comes from the fact that, once the first batch of items gets flushed to a leaf, the root-to-leaf path for future insertions will be in cache, so no more I/O will be needed, except to write out the new data. This also workload has $O(1)$ pass complexity because our flush-then-compact policy will skip compactions at intermediate layers. A straightforward implementation of a size-tiered LSM, on the other hand, will have the same I/O and pass complexity for both random and sequential insertion workloads.

Range queries returning k items cost $O(F \log_F N/Fm)$ I/Os to get started (since the range query must perform a query in every branch along the root-to-leaf path of the query key). Thereafter, they cost $O(k/B)$ I/Os to return all the items. This is comparable to the I/O cost of range queries in a size-tiered LSM tree.

9 Conclusion

Our work shows that, by combining ideas from LSM trees and B$^{\varepsilon}$-trees, we can build a key-value store that outperforms current key-value stores by up to an order of magnitude on insertions, matches or outperforms on lookups, and is competitive on range queries.

SplinterDB targets the common case of small key-value pairs and non-uniformly random workloads. Many real-world key-value workloads come from different clients, some of which might be performing very localized operations, while others are performing relatively random operations. SplinterDB exploits whatever locality is available.

SplinterDB makes contributions to both the data-structural and systems design of high-performance key-value stores. We show how to get the low write amplification of size-tiered data structure while maintaining the high query throughput and workload-adaptivity of a B$^{\varepsilon}$-tree. We also describe several systems issues, such as cache, lock, and memtable design, that one must address to extract the full performance of high-performance NVMe devices.

10 Acknowledgements

We would like to thank Ittai Abraham for his insight and contribution to this project.

We would also like to thank the anonymous reviewers and our shepherd, Ashvin Goel, for their insightful comments.

References

[1] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. TRIAD: creating synergies between memory, disk and log in log structured key-value stores. In Dilma Da Silva and Bryan Ford, editors, *USENIX ATC*, pages 363–375. USENIX Association, 2017.

[2] Michael A. Bender, Jonathan W. Berry, Rob Johnson, Thomas M. Kroeger, Samuel McCauley, Cynthia A. Phillips, Bertrand Simon, Shikha Singh, and David Zage. Anti-persistence on persistent storage: History-independent sparse tables and dictionaries. In Tova Milo and Wang-Chiew Tan, editors, *SIGMOD*, pages 289–302. ACM, 2016.

[3] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. Cache-oblivious streaming b-trees. In Phillip B. Gibbons and Christian Scheideler, editors, *SPAA*, pages 81–92. ACM, 2007.

[4] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proc. VLDB Endow.*, 5(11):1627–1637, 2012.

[5] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. In Irfan Ahmad, editor, *HotStorage*. USENIX Association, 2011.

[6] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. Write-optimized skip lists. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *SIGMOD*, pages 69–78. ACM, 2017.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[8] Dhruba Borthakur. Rocksdb github wiki – performance benchmarks, 2013.

[9] Gerth Stølting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 546–554. ACM/SIAM, 2003.

[10] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. Hashkv: Enabling efficient updates in KV storage via hashing. In Gunawi and Reed [23], pages 1007–1019.

[11] Alex Conway, Ainesh Bakshi, Yizheng Jiao, Yang Zhan, Michael A. Bender, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. How to fragment your file system. *login Usenix Mag.*, 42(2), 2017.

[12] Alex Conway, Eric Knorr, Yizheng Jiao, Michael A. Bender, William Jannen, Rob Johnson, Donald E. Porter, and Martin Farach-Colton. Filesystem aging: It's more usage than fullness. In Daniel Peek and Gala Yadgar, editors, *HotStorage*. USENIX Association, 2019.

[13] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In Geoff Kuenning and Carl A. Waldspurger, editors, *USENIX FAST*, pages 45–58. USENIX Association, 2017.

[14] Alexander Conway, Martin Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 39:1–39:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[15] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In Salihoglu et al. [41], pages 79–94.

[16] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *SIGMOD*, pages 505–520. ACM, 2018.

[17] Niv Dayan and Stratos Idreos. The log-structured merge-bush & the wacky continuum. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *SIGMOD*, pages 449–466. ACM, 2019.

[18] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. The tokufs streaming file system. In Raju Rangaswami, editor, *HotStorage*. USENIX Association, 2012.

[19] Apache Software Foundation. Apache Cassandra, 2019.

[20] Steffen Friedrich and Norbert Ritter. YCSB. In *Encyclopedia of Big Data Technologies*. Springer, 2019.

[21] Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. Scaling Concurrent Log-structured Data Stores. In *Proceedings of the Tenth European Conference on Computer Systems (Eurosys 15)*, page 32. ACM, 2015.

[22] Inc. Google. Leveldb, 2019.

[23] Haryadi S. Gunawi and Benjamin Reed, editors. *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 2018.

[24] W. C. Hsieh and W. E. Weihl. Scalable reader-writer locks for parallel systems. In *IPPS*, 1992.

[25] Yihe Huang, Matej Pavlovic, Virendra J. Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In Gunawi and Reed [23], pages 967–979.

[26] John Iacono and Mihai Patrascu. Using hashing to solve the dictionary problem (in external memory). *CoRR*, abs/1104.2799, 2011.

[27] William Jannen, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Lazy analytics: Let other queries do the work for you. In Nitin Agrawal and Sam H. Noh, editors, *HotStorage*. USENIX Association, 2016.

[28] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: A right-optimized write-optimized file system. In Jiri Schindler and Erez Zadok, editors, *USENIX FAST*, pages 301–315. USENIX Association, 2015.

[29] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *TOS*, 11(4):18:1–18:29, 2015.

[30] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, 2019. USENIX Association.

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

[32] Bredley Kuszmaul. Tokutek White Paper: A Comparison Of Log-Structured Merge (LSM) And Fractal Tree Indexing, 2014.

[33] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In Tim Brecht and Carey Williamson, editors, *SOSP*, pages 447–461. ACM, 2019.

[34] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, 2016.

[35] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. Nvmkv: a scalable, lightweight, ftl-aware key-value store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 207–219, 2015.

[36] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. A general-purpose counting filter: Making every bit count. In Salihoglu et al. [41], pages 775–787.

[37] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, Denver, CO, 2016. USENIX Association.

[38] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[39] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 497–514. ACM, 2017.

[40] Ohad Rodeh. B-trees, shadowing, and clones. *Transactions on Storage*, 2008.

[41] Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors. *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 2017.

[42] Inc. Scylla. ScyllaDB: The real-time big data database, 2019.

[43] Pradeep J. Shetty, Richard P. Spillane, Ravikant R. Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 17–30, 2013.

[44] Tokutek, Inc. TokuDB, 2014. http://www.tokutek.com.

[45] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC 12)*, page 8. ACM, 2012.

[46] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In Shan Lu and Erik Riedel, editors, *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*, pages 71–82. USENIX Association, 2015.

[47] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. Geardb: A gc-free key-value store on HM-SMR drives with gear compaction. In Arif Merchant and Hakim Weatherspoon, editors, *USENIX FAST*, pages 159–171. USENIX Association, 2019.

[48] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Optimizing every operation in a write-optimized file system. In Angela Demke Brown and Florentina I. Popovici, editors, *USENIX FAST*, pages 1–14. USENIX Association, 2016.

[49] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Writes wrought right, and other adventures in file system optimization. *TOS*, 13(1):3:1–3:26, 2017.

[50] Yang Zhan, Alexander Conway, Yizheng Jiao, Eric Knorr, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. The full path to full-path indexing. In Nitin Agrawal and Raju Rangaswami, editors, *USENIX FAST*, pages 123–138. USENIX Association, 2018.

[51] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. How to copy files. In Sam H. Noh and Brent Welch, editors, *USENIX FAST*, pages 75–89. USENIX Association, 2020.

[52] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *OSDI*, pages 461–476. USENIX Association, 2018.