EvenDB: Optimizing Key-Value Storage for Spatial Locality

Eran Gilad
Yahoo Research, Israel

Edward Bortnikov
Yahoo Research, Israel

Anastasia Braginsky
Yahoo Research, Israel

Yonatan Gottesman
Yahoo Research, Israel

Eshcar Hillel
Yahoo Research, Israel

Idit Keidar
Technion and Yahoo Research, Israel

Nurit Moscovici
Outbrain, Israel

Rana Shahout
Technion, Israel

Abstract

Applications of key-value (KV-)storage often exhibit high *spatial locality*, such as when many data items have identical composite key prefixes. This prevalent access pattern is underused by the ubiquitous LSM design underlying high-throughput KV-stores today.

We present EvenDB, a general-purpose persistent KV-store optimized for spatially-local workloads. EvenDB combines spatial data partitioning with LSM-like batch I/O. It achieves high throughput, ensures consistency under multi-threaded access, and reduces write amplification.

In experiments with real-world data from a large analytics platform, EvenDB outperforms the state-of-the-art. E.g., on a 256GB production dataset, EvenDB ingests data 4.4× faster than RocksDB and reduces write amplification by nearly 4×. In traditional YCSB workloads lacking spatial locality, EvenDB is on par with RocksDB and significantly better than other open-source solutions we explored.

ACM Reference format:
Eran Gilad, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Eshcar Hillel, Idit Keidar, Nurit Moscovici, and Rana Shahout. 2020. EvenDB: Optimizing Key-Value Storage for Spatial Locality. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 16 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Motivation: spatial locality in KV-storage

Key-value stores (KV-stores) are widely used by a broad range of applications and are projected to continue to increase in popularity in years to come; market research identifies them as the "driving factors" of the NoSQL market, which is expected to garner $4.2B by 2020 [8].

KV-stores provide a simple programming model. Data is an ordered collection of key-value pairs, and the API supports random writes, random reads, and range queries.

A common design pattern is the use of *composite* keys that represent an agglomerate of attributes. Typically, the primary attribute (key prefix) has a skewed distribution, and so access via the composite key exhibits *spatial locality*, as popular key prefixes result in popular key ranges [28].

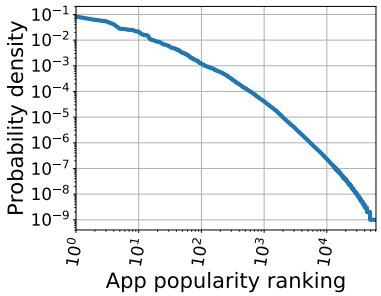


Figure 1. Distribution of mobile app events by app id (log-log scale) in a production analytics feed (2B events).

One example of this arises in mobile analytics platforms, e.g., AppsFlyer [2], Flurry [3], and Google Firebase [5]. Such platforms ingest massive streams of app event reports in real-time and provide a variety of insight queries into the data. For example, Flurry tracked events from 1M+ apps across 2.6B user devices in 2017 [4]. In order to offer per-app analytics efficiently, such services aggregate data in KV-stores indexed by a composite key prefixed by a unique app id, followed by a variety of other attributes (time, user id, device model, location, event type, etc.). We examine a trace of almost 2B events captured from a production mobile analytics engine over half an hour. Figure 1 shows the access frequency distribution over the 60K app ids occurring in this stream. It follows a marked heavy-tail pattern: 1% of the apps cover 94% of the events, and fewer than 0.1% cover 70% of them.

Composite keys arise in many additional domains, including messaging and social networks [28]. For example, a back-end Facebook Messenger query may retrieve the last 100 messages for a given user [24]; in Facebook's social network, a graph edge is indexed by a key consisting of two object ids and an association type [21]. Spatial locality also arises with simple (non-composite) keys, for example, when reverse URL domains are used as keys for web indexing [30].

The prevalence of skewed (e.g., Zipfian) access in real workloads is widely-recognized and reflected in standard

benchmarks (e.g., YCSB [31]). But these benchmarks fail to capture the spatial aspect of locality, which has gotten far less attention. In this work, we make spatial locality a first-class consideration in KV-store design.

1.2 Spatial locality: the challenge

The de facto standard design for high-throughput KV-stores today is *LSM* (log-structured merge) trees [42]. LSMs initially group writes into files *temporally* rather than by key-range. A background *compaction* process later merge-sorts any number of files, grouping data by keys.

This approach is not ideal for workloads with high spatial locality for two reasons. First, popular key ranges are fragmented across many files. Second, compaction is costly in terms of both performance (disk bandwidth) and *write amplification*, namely the number of physical writes associated with a single application write. The latter is particularly important in SSDs as it increases disk wear. The temporal grouping means that compaction is indiscriminate with respect to key popularity: Since new files are always merged with old ones, "cold" key ranges continue to be repeatedly re-located by compactions.

Another shortcoming of LSM is that its temporal organization, while optimizing disk I/O, penalizes in-memory operation. All updates – including ones of popular keys – are flushed to disk, even though persistence is assured via a separate *write-ahead-log (WAL)*.

Yet LSMs have supplanted the traditional spatial data partitioning of B-trees for a reason [40]. In B-trees, each update induces random I/O to a leaf, resulting in poor write performance. Moreover, the need to preserve a consistent image of a leaf while it is being over-written induces high write amplification. B^ϵ-trees [26] mitigate this cost using write buffers. However, this slows down lookups, which now have to search in unordered buffers, possibly on disk. LSMs, in contrast, achieve high write throughput by absorbing writes in memory and periodically flushing them as sequential files to disk; they expedite reads by caching data in DRAM.

The resounding performance advantage of the LSM approach over B- and B^ϵ-trees has been repeatedly demonstrated, e.g., in a recent study of the Percona MySQL server using three storage engines – RocksDB, TokuDB, and InnoDB – based on LSM, a B^ϵ-tree, and a B-tree, respectively [35]. Another advantage of LSMs is that they readily ensure consistency under multi-threaded access – in particular, atomic scans – via lock-free multi-versioning. In contrast, databases based on B- or B^ϵ-trees either use locks [7] or forgo scan consistency [43].

Our goal is to put forth a KV-store design alternative suited for the spatial locality arising in today's workloads, without forfeiting the benefits achieved by the LSM approach.

1.3 Our contribution: EvenDB

We present EvenDB, a high-throughput persistent KV-store geared towards spatial locality. EvenDB's architecture (§2) combines a spatial data organization with LSM-like batch I/O. The pillars of our design are large *chunks* holding contiguous key ranges. EvenDB's chunks are not merely a means to organize data on-disk (like nodes in a B-tree). They are also the basic units for read-write DRAM caching, I/O-batching, logging, and compaction. This approach is unique. Typical KV-stores rely on finer-grain OS- and application-level page caches (whereas chunks consist of many pages) and employ a global WAL.

Our novel chunk-based organization has several benefits. First, chunk caching is effective for spatially-local workloads. Second, chunk-level logging eliminates the need to log each update in a system-level WAL, thus reducing write amplification and expediting crash-recovery. Finally, using the same data structure for both the read-path (as a cache) and the write-path (as a log) allows us to perform *in-memory compaction*, reducing write amplification even further.

The downside of spatial partitioning is that if the data lacks spatial locality and the active working set is big, chunk-level I/O batching is ineffective. Moreover, caching an entire chunk for a single popular key is wasteful. We mitigate the latter by adding a *row cache* for read access to hot keys. Even so, our design is less optimal for mixed read/write workloads lacking spatial locality, for which the LSM approach may yield better performance.

Our algorithm (§3) is designed for high concurrency. It supports atomic scans using low-overhead multi-versioning, where versions are increased only by scans and not by updates. It ensures consistency and correct failure-recovery.

We implement EvenDB in C++ (§4) and extensively evaluate it (§5) via three types of workloads: (1) a production trace collected from a large-scale mobile analytics platform; (2) workloads with synthetically-generated composite keys exercising standard YCSB scenarios [31]; and (3) YCSB's traditional benchmarks, which employ simple (non-composite) keys. We compare EvenDB to the recent (Oct 2018) release of RocksDB [14], a mature industry-leading LSM KV-store. We experimented with two additional open-source KV-stores, PebblesDB [45] and TokuDB [17] (the only publicly-available B^ϵ-tree-based KV-store); both performed significantly worse than RocksDB and EvenDB, so we focus on RocksDB results. Our main findings are:

1. EvenDB is better than RocksDB under high spatial locality. For instance, on a 256GB production dataset, EvenDB ingests data 4.4× faster than RocksDB and reduces write amplification by almost 4×.
2. EvenDB significantly outperforms RocksDB whenever most of the working set fits in RAM, accelerating scans by up to 3.5×, puts by up to 2.3×, and gets by up to 2×.

3. EvenDB's performance is comparable to RocksDB's in traditional YCSB workloads without spatial locality.
4. RocksDB outperforms EvenDB (by 20–25%) in mixed read/write workloads with large active working sets and no spatial locality, although EvenDB's write amplification remains ∼ 2× smaller than RocksDB's.

Our results underscore the advantages of EvenDB's spatially-organized chunks: (1) eliminating fragmentation of key ranges yields better performance under spatial locality; (2) keeping hot ranges in memory leads to better performance when most of the working set fits in RAM; and (3) in-memory chunk compaction saves disk flushes and reduces write volume. In addition, in-chunk logging allows quick recovery from crashes with no need to replay a WAL.

§6 surveys related work and §7 concludes this paper.

2 Design Principles

2.1 Access semantics and optimization goals

EvenDB is a persistent ordered key-value store. Similarly to popular industrial ones [12–14], it supports concurrent access by multiple threads and ensures strong consistency. Specifically its *put*, *get*, and *scan* operations are *atomic*. For scans, this means that all key-value pairs returned by a single scan belong to a consistent snapshot reflecting the state of the data store at a unique point in time.

EvenDB persists data to disk to allow it to survive crashes. As in other systems [13, 14], it supports *asynchronous* persistence, where puts are buffered before being persisted in the background, trading durability for speed. In this case, some recent updates may be lost but recovery is to a *consistent* state in the sense that if some put is lost, then no ensuing (and thus possibly dependent) puts are reflected.

Our key optimization goals are the following:

1. Optimize for spatial locality, e.g., workloads that employ composite keys.
2. Minimize write amplification to reduce disk wear.
3. Strive for high performance in *sliding-local* scenarios, where most of the active working set fits in DRAM. Note that we do *not* expect the entire database to fit in memory, only the active data.
4. Ensure fast recovery to a consistent state.

2.2 Design choices

EvenDB combines the spatial data partitioning of B-trees with the optimized I/O and quick access of LSMs. Its data layout is illustrated in Figure 2. We now discuss its key components.

Chunks. To leverage spatial locality, we partition data – both on-disk and in-memory – by key. Our data structure is organized as a list of *chunks* holding consecutive key ranges. All chunks are represented in memory via lightweight volatile metadata objects, which are reconstructed from disk on recovery.

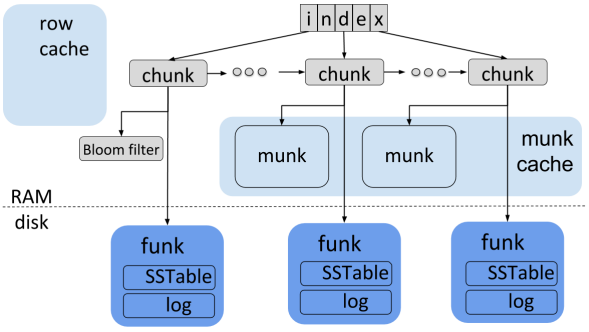


Figure 2. EvenDB's organization. Gray boxes depict metadata, light blue areas show RAM caches of KV-pairs, and blue areas represent on-disk KV storage.

Sequential I/O with in-chunk logging. For persistence, each chunk has a file representation called *funk* (file chunk), which holds all the KV-pairs in the chunk's range. Within funks, we adopt LSM's sequential I/O approach. To this end, the funk is divided into two parts: (1) a sorted *SSTable* (Sorted String Table [29]), and (2) an unsorted *log*. New updates are appended to the log; the log is merged into the SSTable via an infrequent background compaction process. Under spatial locality, popular chunks are targeted frequently, allowing effective batching of their log updates. Unlike LSMs, EvenDB logs writes exclusively within their funks and avoids duplicating the updates in a separate WAL. This reduces write amplification and expedites recovery.

Chunk-level caching with in-memory compaction. DRAM caches are instrumental for read performance. To favor workloads with spatial locality, we cache entire chunks: a popular chunk is cached in a memory data structure called *munk* (memory chunk). Like their on-disk counterparts, munks have a compacted sorted prefix while new updates are appended at the end and remain unsorted until the next compaction. Whereas LSM caches only serve the read-path, caching at the chunk granularity allows us to leverage munks also in the write-path, specifically, for in-memory compaction. We observe that compacting a funk's log is only required for performance (to expedite on-disk lookup) and is redundant whenever the chunk is cached. Thus, when a chunk has a munk, we compact it almost exclusively in memory and allow the disk log to grow. Note that if a chunk does not have a munk, it usually means that the chunk is "cold" and hence there is little or no new data to compact. So either way, disk compaction is rare, and write amplification is low.

Fast in-memory access. Chunks are organized in a sorted linked list. To speed up lookups, they are indexed using a volatile index (a sorted array in our implementation).

Row caches and Bloom filters. EvenDB adopts two standard mechanisms from LSMs. First, to expedite access to keys whose chunks are only on-disk (i.e., have no munks), a *row cache* of individual popular keys serves the read-path. The row cache is important for workloads that lack spatial

locality where caching an entire chunk for a single "hot" key is wasteful. Second, for munk-less chunks, we employ *Bloom filters* to limit excessive access to disk. The Bloom filter is maintained as long as the chunk has no munk, and is re-created whenever a munk is evicted.

Concurrency and multi-versioning. EvenDB allows high parallelism among threads invoking its API calls. Get operations are wait-free (never block) and puts use lightweight synchronization. To support atomic scans, we employ a light form of multi-versioning that uses copy-on-write to keep old versions only if they may be required by ongoing scans. In other words, if a put attempts to overwrite a key required by an active scan, then a new version is created alongside the existing one, whereas versions that are not needed by any scan are not retained. Version management incurs a low overhead because it occurs only on scans. In addition, tagging each value with a version allows EvenDB to easily recover to a consistent point in time, namely a version below which all puts have been persisted to disk.

3 EvenDB's Design

In §3.1 we present EvenDB's data organization. We discuss concurrency control and atomic scans in §3.2. §3.3 overviews EvenDB's normal operation flow, while the data structure's maintenance is discussed in §3.4. Finally, §3.5 discusses data flushes and failure recovery.

3.1 Data organization

EvenDB's data layout is depicted in Figure 2. Data is partitioned into chunks, each holding a contiguous key range. Each chunk's data (keys in its range and their corresponding values) is kept on-disk (funks, for persistence), and possibly in-memory (munks, for fast access). Munks can be replaced and loaded from funks at any time based on an arbitrary replacement policy. Chunk metadata objects are significantly smaller than munks and funks (typically, less than 1 KB vs. tens of MBs) and are always kept in memory.

A volatile *index* maps keys to chunks. Index updates are lazy, offering only best-effort expedited search.

A funk consists of two files: a compacted and sorted KV map *SSTable* and a write *log*. When a funk is created, the former holds all the chunk's KV pairs and the latter is empty. New KV pairs are subsequently appended to the log. If a key is over-written, its old value remains in the SSTable while the new one is added to the log (the log is more up-to-date).

This structure allows us to benefit from sorted searches on the SSTable and at the same time to update chunks without relocating existing data, thus minimizing write amplification. As a funk's log grows, however, searching becomes inefficient and the funk is no longer compact, i.e., it may contain redundant (over-written) values. Therefore, once the log exceeds a certain threshold, we reorganize the funk via a process we call *rebalance*, as explained below. The rebalance threshold controls the system's *write amplification*, namely, the additional space consumed on top of the raw data. Note that the additional space amplification induced by fragmentation is negligible, because chunks typically consist of ~1000 pages.

A munk holds KV pairs in an array-based linked list. When a munk is created, some prefix of this array is populated, sorted by key, so each cell's successor in the linked list is the ensuing cell in the array. New KV entries are appended after this prefix. As new entries are added, they create bypasses in the linked list, and consecutive keys in the list are no longer necessarily adjacent in the array. Nevertheless, as long as a sizable prefix of the array is sorted and insertion order is random, bypasses are short in expectation. Keys can thus be searched efficiently via binary search on the sorted prefix followed by a short traversal of a bypass path.

As KV pairs are added, overwritten, and removed, munks and funks need to undergo reorganization. This includes (1) *compaction* to garbage-collect removed and overwritten data, (2) *sorting* keys to make searches more efficient, and (3) *splitting* overflowing chunks. Reorganization is performed by three procedures: (1) Munk rebalance (creating a new compacted and sorted munk instead of an existing one) happens in-memory, independently of disk flushes. (2) Funk rebalance (on-disk) happens much less frequently. (3) Splits create new chunks as well as new funks and munks.

Whenever a chunk is cached (i.e., has a munk), access to this chunk is particularly fast: the chunk metadata is quickly located using the index, the munk's sorted prefix allows for fast binary search, and updates are added at the end of the munk's array and appended to the funk's log. We take two measures to mitigate the performance penalty of accessing keys in non-cached chunks.

First, we keep a row cache holding popular KV pairs only from munk-less chunks. Note that unlike munks, which cache key ranges, the row cache holds individual keys, and is thus more effective in dealing with point queries (gets as opposed to scans) with no spatial locality.

If the active working set is larger than the available DRAM, these two caches might not suffice, and so a certain portion of reads will be served from disk. Here, the slowest step is the sequential search of the log. To reduce such searches, a chunk with no munk holds a Bloom filter for the corresponding funk's log, which eliminates most of the redundant log searches. The Bloom filter is partitioned into a handful of filters, each summarizing the content of part of the log, limiting sequential searches to a small section of the log.

3.2 Concurrency and atomic scans

EvenDB allows arbitrary concurrency among operations. Gets are wait-free and proceed without synchronization. In order for scans to be atomic, they synchronize with puts, potentially waiting for some puts to complete (puts never wait for scans). Puts also synchronize with rebalance operations.

To support atomic scans we employ a system-wide *global version (GV)*. A scan creates a *snapshot* associated with GV's current value by fetching and incrementing GV. This signals to ensuing put operations that they must not overwrite the highest version smaller than the new GV value. This resembles a *copy-on-write* approach, which creates a virtual snapshot by indicating that data pertaining to the snapshot should not be overwritten in-place.

To allow garbage collection of old versions, EvenDB tracks the snapshot times of active scans. This is done in a dedicated *Pending Operations (PO)* array, which has one entry per active thread, and is also used to synchronize puts with scans as explained shortly. The compaction process removes old versions that are no longer required for any scan listed in PO. Specifically, for each key, it removes versions older than the highest version smaller than the minimal scan entry in PO and the value of GV when the rebalance begins.

A put obtains a version number from GV without incrementing it. Thus, multiple puts may write values with the same version, each over-writing the previous one.

If a put obtains its version before a scan increments GV, the new value must be included in the scan's snapshot. However, because the put's access to the GV and the insertion of the new value to the chunk do not occur atomically, a subtle race may arise. Assume a put obtains version 7 from GV and then stalls before inserting the value to the chunk while a scan obtains version 7 and increments GV to 8. If the scan proceeds to read the affected chunk, it misses the new value it should have included in its snapshot. To remedy this, puts announce (in PO) the key they intend to change when beginning their operations and scans wait for relevant pending puts to complete; see the next section for details.

Rebalance operations synchronize with concurrent puts using the chunk's *rebalanceLock*. This is a shared/exclusive lock, acquired in shared mode by puts and in exclusive mode (for short time periods) by rebalance. Gets and scans do not acquire the lock. Note that it is safe for them to read from a chunk while it is being replaced because (1) rebalance makes the new chunk accessible only after it is populated, and (2) a chunk is immutable during rebalance, so the newly created chunk holds the same content as the displaced one.

To minimize I/O, we allow at most one thread to rebalance a funk at a given time. This is controlled by the *funkChangeLock*, which is held by the thread rebuilding the chunk. It is acquired using a try_lock, where threads that fail to acquire it do not retry but instead wait for the winning thread to complete the funk's creation.

3.3 EvenDB operations

Algorithm 1 presents pseudocode for EvenDB's operations. Both get and put begin by locating the target chunk using the lookup function. In principle, this can be done by traversing the chunk list, but that would result in linear search time. To expedite the search, lookup first searches the index. But because index updates are lazy – they occur after the new chunk is already in the linked list – the index may return a stale chunk that had already been replaced by rebalance. To this end, the index search is repeated with a smaller key in case the index returns a stale chunk, and the index search is supplemented by a linked-list traversal.

Algorithm 1 EvenDB normal operation flow for thread i.

```
1:  procedure GET(key)
2:      C ← lookup(key)
3:      if C.munk then
4:          search key in C.munk linked list; return
5:      search key in row cache; return if found
6:      if key∈ C.bloomFilter then
7:          search key in funk.log; return if found
8:      search key in funk.SSTable; return if found
9:      return NULL

10: procedure PUT(key, val)
11:     C ← lookup(key)
12:     lockShared(C.rebalanceLock)
13:     PO[i] ← ⟨put, key, ⊥⟩      ▷ publish thread's presence
14:     gv ← GV                    ▷ read global version
15:     PO[i] ← ⟨put, key, gv⟩     ▷ write version in PO
        ▷ write to funk log, munk (if exists), and row cache
16:     append ⟨key, val, gv⟩ to funk.log
17:     if C.munk then
18:         add ⟨key, val, gv⟩ to C.munk's linked list
19:     else          ▷ no munk – key may be in row cache
20:         update ⟨key, val⟩ in row cache (if key is present)
21:     unlock(C.rebalanceLock)
22:     PO[i] ← ⊥                  ▷ put is no longer pending

23: procedure SCAN(key1, key2)
24:     PO[i] ← ⟨scan, key1, key2, ⊥⟩  ▷ publish scan's intent
25:     gv ← F&I(GV)  ▷ fetch and increment global version
26:     PO[i] ← ⟨scan, key1, key2, gv⟩  ▷ publish version in PO
27:     T ← PO entries updating keys in range [key1, key2]
28:     wait until ∀t ∈ T, t completes or has a version > gv
29:     C ← lookup(key1)
30:     repeat
31:         collect from C.munk or C.funk (log and SSTable)
            max version ≤gv for all keys in [key1, key2]
32:         C ← C.next
33:     until reached key2
```

Get. If the chunk has a munk, get searches the key in it by first running a binary search on its sorted prefix and then traversing linked list edges as needed. Otherwise, get looks for the key in the row cache. If not found, it queries the Bloom filter to determine if the key might be present in the target chunk's log, and if so, searches for it there. If the key is in none of the above, the SSTable is searched.

Put. Upon locating the chunk, put grabs its rebalanceLock in shared mode to ensure that it is not being rebalanced. It then registers itself in PO with the key it intends to put, reads GV, and sets the version field in its PO entry to the read version. The put then proceeds to write the new KV pair to the funk's log and to the munk, if exists. If the funk has no munk and the row cache contains an old value of the key, the row cache is then updated. The munk and funk are multi-versioned to support atomic scans, whereas the row cache is not used by scans and holds only the latest version of each key. Finally, a put unregisters itself from PO, indicating completion, and releases the chunk's rebalance lock.

We note that in case multiple puts concurrently update the same key with the same version, they may update the funk and munk (or the funk and row cache) in different orders, and so the latest update to one will not coincide with the latest update to the other. This can be addressed, for example, by locking keys. Instead, we opt to use a per-chunk monotonically increasing counter (not shown in the code) to determine the order of concurrent put operations updating the same key with the same version. We enforce updates to occur in order of version-counter pairs by writing them alongside the values in the munk, PO, funk, and row cache. Following a split, the new chunks inherit the counter from the one they replace.

Scan. A scan first publishes its intent to obtain a version in PO, to signal to concurrent rebalances not to remove the versions it needs. It fetches-and-increments GV to record its snapshot time gv, and then publishes its key-range and gv in PO. Next, the scan waits for the completion of all pending puts that might affect it – these are puts of keys in the scanned key range that either do not have a version yet or have versions lower than the scan time. This is done by busy waiting on the PO entry until it changes; monotonically increasing counters are used in order to avoid ABA races.

Then it collects the relevant values from all chunks in the scanned range. Specifically, if the chunk has a munk, the scan reads from it, for each key in its range, the latest version of the value that precedes its snapshot time. Otherwise, the scan collects all the relevant versions for keys in its range from both the SSTable and the log and merges the results. Finally, the scan unregisters from PO.

3.4 Rebalance

Munk (resp. funk) rebalance improves the data organization in a munk (funk) by removing old versions that are no longer needed for scans, removing deleted items, and sorting all the keys. It is typically triggered when the munk (funk) exceeds a capacity threshold. The threshold for funk rebalance is higher when it has a munk, causing most rebalances to occur in-memory. Rebalance creates a new munk (funk) rather than reorganize it in-place in order to reduce the impact on concurrent accesses. When it is ready, EvenDB flips the chunk's reference to the new munk (funk).

Munk rebalance acquires the chunk's rebalanceLock in exclusive mode to block puts to the munk throughout its operation, while concurrent gets and scans can exploit the munk's immutability and proceed without synchronization. Rebalance iterates through PO to collect the minimum version number among all active scans. Since each rebalance operates on a single chunk with a known key range, scans targeting non-overlapping ranges are ignored. If a scan has published its intent in PO but published no version yet, the rebalance waits until the version is published. When the new munk is ready, the munk reference in the chunk is flipped and rebalanceLock is released.

Funk rebalance creates a new (SSTable, log) pair. If the chunk has a munk, we simply perform munk rebalance on its munk and then flush the munk to the new SSTable file and the new log is empty. Otherwise, the new SSTable is created by merging the old SSTable with the old log. This procedure involves I/O and may take a long time, so we do not block puts for most of its duration. Rather, puts occurring during the merge are diverted to a separate log segment that is ignored by the merge. When the merge completes, rebalance proceeds as follows: (1) block new puts using the rebalanceLock; (2) set the new log to be the diverted puts segment; (3) flip the funk reference; and (4) release rebalanceLock. Simultaneous rebalance of the same funk by two threads is prevented (through a separate exclusive lock) in order to avoid redundant work.

If a munk rebalance exceeds some capacity threshold in a new munk, it triggers a *split*. Unlike single-chunk rebalances, splits entail changes in the chunks linked list as well as the index, and so are more subtle. A split proceeds in two phases. In the first, the chunk is immutable, namely, rebalanceLock is held in exclusive mode. In the second (longer) phase, the new chunks are mutable.

The first phase runs in-memory and so is fast (this prevents blocking puts for a long time). It proceeds as follows: (1) split the munk into two sorted and compacted halves; (2) create two new chunks (metadata), each referencing one of the new munks but temporarily sharing the same old funk, both immutable, with the first half munk pointing to the second; (3) insert the new chunks into the list instead of the old chunk; and finally (4) update the index. Note that once the new chunks are added to the list they can be discovered by concurrent operations. On the other hand, other concurrent operations might still access the old chunk via the index or stale pointers. This does not pose a problem because the chunks are immutable and contain the same KV pairs.

In the second phase, puts are enabled on the new chunks (rebalanceLock is released) but the new chunks cannot yet be rebalanced. Note that the old chunk remains immutable; it continues to serve ongoing reads as long as there are outstanding operations that hold references to it, after which

it may be garbage collected. This phase splits the shared funk. It uses the sorted prefixes of the new munks as SSTables and their suffixes as logs. Once done, the funk references in the new chunks are flipped and future rebalances are allowed.

Underflowing neighboring chunks (e.g., following massive deletions) can be merged via a similar protocol. Our current EvenDB prototype does not implement this feature.

3.5 Disk flushes and recovery

Recall that all puts write to the funk log, regardless of whether the chunk has a munk. Funk logs are not replayed on recovery, and so recovery time is not impacted by their length.

Like most popular KV-stores, EvenDB supports two modes of persistence – *synchronous* and *asynchronous*. In the former, updates are persisted to disk (using an fsync call) before returning to the user. The drawback of this approach is that it is roughly an order-of-magnitude slower than the asynchronous mode. Asynchronous I/O, where fsync is only called periodically, reduces write latency and increases throughput, but may lead to loss of data that was written shortly before a crash. The tradeoffs between the two approaches are well known, and the choice is typically left to the user.

Recovery semantics. In the synchronous mode, the funks always reflect all completed updates. In this case, recovery is straightforward: we simply construct the chunks linked list and chunk index from the funks on disk, and then the database is immediately ready to serve new requests, populating munks and Bloom filters on-demand.

In the asynchronous mode, some of the data written before a crash may be lost, but we ensure that the data store consistently reflects a *prefix* of the values written. For example, if put(k1, v1) completes before put(k2, v2) is invoked and then the system crashes, then following the recovery, if k2 appears in the data store, then k1 must appear in it as well. Such recovery to a consistent state is important, since later updates may depend on earlier ones.

Note that the temporal organization in LSMs inherently guarantees such consistency, whereas with spatial data organization, extra measures need to be taken to ensure it.

Checkpointing for consistent recovery. We use *checkpoints* to support recovery to a consistent state in asynchronous mode. A background process creates checkpoints using atomic scans: It first fetches-and-increments GV to obtain a snapshot version gv. Next, it synchronizes with pending puts via PO to ensure that all puts with smaller versions are complete. It then calls fsync to flush all pending writes to disk. Finally, it writes gv to a dedicated *checkpoint file* on disk. This enforces the following correctness invariant: at all times, all updates pertaining to versions smaller than or equal to the version recorded in the checkpoint file have been persisted. Note that some puts with higher versions than gv might be reflected on disk while others are not.

epoch	last checkpointed version
0	1375
1	956

Table 1. Example recovery table during epoch 2.

EvenDB's recovery is lazy. Data is fetched into munks as needed during normal operation. To ensure consistency, following a recovery, retrievals from funks should ignore newer versions that were not included in the latest completed checkpoint before the crash. This must be done by every operation that reads data from a funk, namely get or scan from a chunk without a munk, funk rebalance, or munk load.

To facilitate this check, we distinguish between pre-crash versions and new ones created after recovery using *epoch numbers*. Specifically, a version is split into an epoch number (in our implementation, the four most-significant bits of the version) and a per-epoch version number. Incrementing the GV in the normal mode effectively increases the latter. The recovery procedure increments the former and resets the latter, so versions in the new epoch begin from zero.

We maintain a *recoveryTable* mapping each recovered epoch to its last checkpointed version number. For example, Table 1 shows a possible state of the recovery table after two recoveries, i.e., during epoch 2.

The recovery procedure reads the checkpoint time from disk, loads the recoveryTable into memory, adds a new row to it with the last epoch and latest checkpoint time, and persists it again. It then increments the epoch number and resumes normal operation with version 0 in the new epoch.

4 Implementation

We implement EvenDB in C++. We borrow the SSTable implementation from the RocksDB open source [14]. Similarly to RocksDB, we use jemalloc for memory allocation.

The chunk index is implemented as a sorted array holding the minimal keys of all chunks. Whenever a new chunk is created (upon split), the index is rebuilt and the reference to the index is atomically flipped. We found this simple implementation to be fastest since splits are infrequent.

The munk cache applies an LFU eviction policy. We use exponential decay to maintain the recent access counts, similar to [33]: periodically, all counters are sliced by a factor of two. The row cache implements a coarse-grained LRU policy using a fixed-size queue of hash tables. New entries are inserted into the head table. Once it overflows, a new empty table is added to the head, and the tail is discarded. Consequently, lookups for recently cached keys are usually served by the head table, and unpopular keys are removed from the cache in a bulk once the tail table is dropped.

The row cache never holds stale values. Therefore, a put updates the cache whenever a previous version of the updated key is already in the cache. But if the key is not in the cache, put does not add it, to avoid overpopulating the cache

in write-dominated workloads. After a get, the up-to-date KV-pair is added to the head table unless it is already there. If the key's value already exists in another table, it is shared by the two tables, to avoid duplication.

5 Evaluation

The experiment setup is described in §5.1. We present performance results with production data in §5.2 and with standard synthetic workloads in §5.3.

The baseline for our evaluation is RocksDB – a mature and widely-used industrial KV-store. We use the recent RocksDB release 5.17.2, available Oct 24, 2018. It is worth noting that RocksDB's performance is significantly improved in this release [27]. In §5.4 we compare EvenDB against PebblesDB [45], a research LSM prototype, and TokuDB [17] – the only publicly available KV-store whose design is inspired by B^{ϵ}-trees. Both perform significantly worse than RocksDB and EvenDB, motivating our focus on RocksDB.

5.1 Setup

Methodology. Our hardware is a 12-core (24-thread) Intel Xeon 5 with 4TB SSD disk. We run each experiment within a Linux container with 16GB RAM. Data is stored uncompressed. We run 5 experiments for each data point and present the median measurement to eliminate outliers. Since experiments are long, the results vary little across runs. In all of our experiments, the STD was within 6.1% of the mean, and in most of them below 3%.

We employ a C++ implementation [15] of YCSB [31], the standard benchmarking platform for KV-stores. YCSB provides a set of APIs and a synthetic workload suite inspired by real-life applications. In order to exercise production workloads, we extend YCSB to replay log files.

In each experiment, a pool of concurrent worker threads running identical workloads stress-tests the KV-store. We exercise 12 YCSB workers and allow the data store to use 4 additional background threads for maintenance. We also experimented with different numbers of worker threads, finding similar scalability trends in RocksDB and EvenDB; these results are omitted for lack of space.

Configuration. To avoid over-tuning, all experiments use the data stores' default configurations. For RocksDB, we use the configuration exercised by its public performance benchmarks [11]. We also experimented with tuning RocksDB's memory resources based on its performance guide [1]; this had mixed results, improving performance by at most 25% over the default configuration in some workloads but deteriorating it by up to 25% in others. Overall, the default configuration performed best. We fixed the PebblesDB code to resolve a data race reported in the project's repository [19].

EvenDB's default configuration allocates 8GB to munks and 4GB to the row cache, so together they consume 12GB out of the 16GB container. The row cache consists of three hash tables. The Bloom filters for funks are partitioned 16-way. We set the EvenDB maximum chunk size limit to 10MB, the rebalance size trigger to 7MB, the funk log size limit to 2MB for munk-less chunks, and to 20MB for chunks with munks. The results of the experiments we ran in order to tune these parameters are omitted due to lack of space.

We focus on asynchronous persistence, i.e., flushes to disk happen in the background; (with synchronous I/O, performance is an order-of-magnitude slower, trivializing the results in all scenarios that include puts).

5.2 Production data

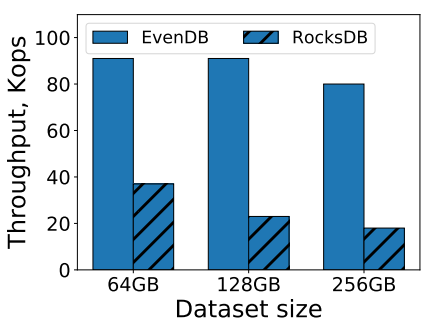
Our first set of experiments is driven by a log collected by a production mobile analytics engine. The log captures a stream of ~13M unique events per minute, with an average log record size of 800B, i.e., ~10GB/min. We load the logged events into a table indexed by app id and timestamp. As noted in §1, the app id distribution is heavy-tailed, i.e., the data exhibits spatial locality. We use this log to drive *put-only* (100% put) and *scan-dominated* (95% scan, 5% put) tests.

Put-only (data ingestion). Figure 3 presents our data ingestion results. Here, we load 64GB, 128GB and 256GB of data, in timestamp order; note that this order is different from the KV-store's primary key. Figure 3a depicts the throughput in each experiment. Clearly, EvenDB is much faster than RocksDB and its advantage becomes more pronounced as the dataset grows. For example, EvenDB ingests 256GB within 1.1 hours, whereas RocksDB requires 4.85 hours (4.4× slower). Figure 3b depicts the throughput dynamics for the 256GB dataset. RocksDB's throughput, while stable overall, suffers from stalls (lasting a minute or longer) caused by compactions. EvenDB delivers predictable performance, with a throughput constantly above 4× RocksDB's average rate.

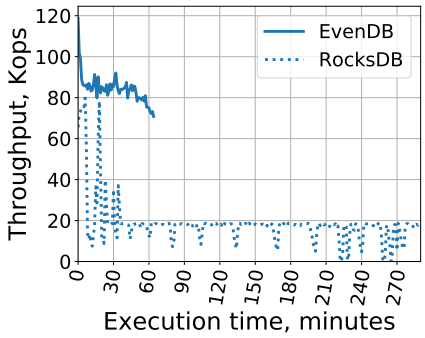
Figure 3c shows the write amplification in the same benchmark. While EvenDB's amplification is unaffected by scaling, RocksDB deteriorates as the dataset (and consequently, the number of LSM levels) grows. These results underscore the importance of EvenDB's in-memory compactions, which consume CPU but dramatically reduce the I/O load by minimizing on-disk compaction (funk rebalances).

This dichotomy between CPU and I/O usage can be observed in Table 2, which summarizes the overall resource consumption in the 256GB ingestion experiment. We see that EvenDB is CPU-intensive – it exploits 40% more CPU cycles than RocksDB, which means that its average CPU rate is 6.3× higher. On the other hand, RocksDB is I/O-intensive – it reads 43× (!) more data than EvenDB (recall that the workload is write-only; reading is for compaction purposes).

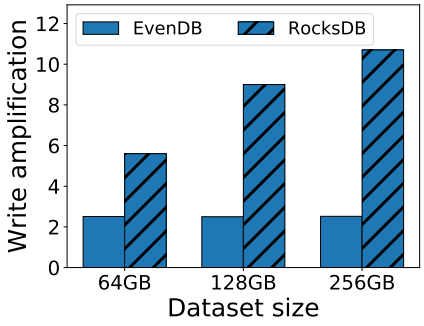
Figure 3d shows the disk space used during the ingestion. In EvenDB the space occupied by logs (dotted line) grows linearly with the data size, and is the main reason for space amplification (namely the gap between space consumption and input size). We see the log sizes level out at the end of



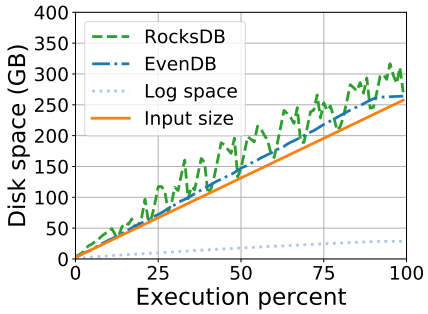
(a) Throughput, Kops



(b) Throughput dynamics, 256GB



(c) Write amplification



(d) Space consumption, 256GB

Figure 3. EvenDB vs RocksDB performance under ingestion (100% put) workload with production datasets.

	Duration	CPU time	Read I/O	Write I/O
EvenDB	1.1 hr	14.6 hr	47.6 GB	645.2 GB
RocksDB	4.85 hr	10.4 hr	2053.5 GB	2660.4 GB

Table 2. EvenDB vs RocksDB resource consumption during ingestion (100% put) of a 256GB production dataset.

the run. This occurs because as threads "run out" of data to ingest, they can complete a backlog of pending rebalances. RocksDB's space consumption increases sharply between compactions and then drops whenever compaction occurs.

Scan-dominated (analytics). We run 40M operations – 95% range queries, 5% puts – on the KV-store produced by the ingestion tests. Every query scans a sequence of recent events within one app (all the scanned rows share the app id key prefix). The app id is sampled from the data distribution (so popular apps are queried more frequently). Figure 4 depicts the performance dynamics for queries that scan 1-minute histories. We experimented also with shorter scans, with similar results, which are omitted for lack of space.

EvenDB completes all experiments 20% to 30% faster than RocksDB. Yet the two systems' executions are very different. EvenDB's throughput stabilizes almost instantly upon transition from ingestion to analytics because its working set munks are already in memory. In contrast, it takes RocksDB 10 to 35 minutes (!) to reach the steady-state performance. During this period, it goes through multiple compactions, which degrade its application-level throughput beyond 90%.

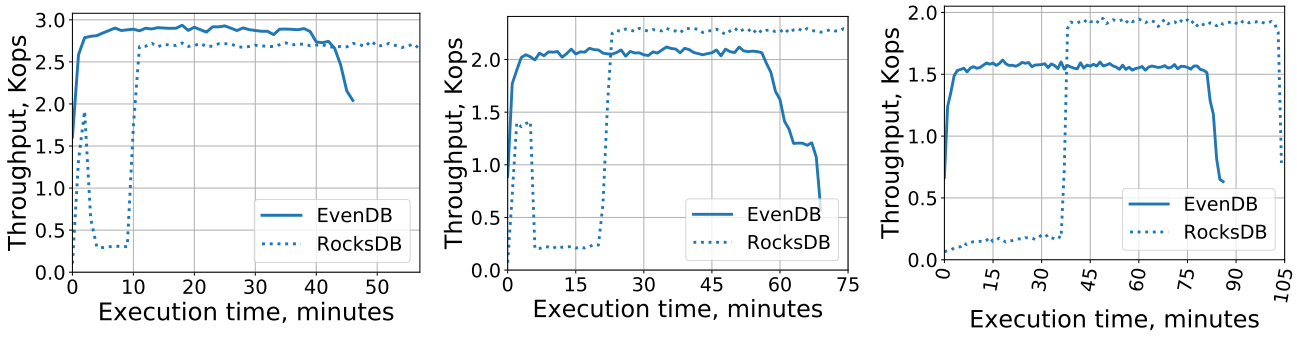
After all compactions are done, RocksDB's improved organization gives it an advantage over EvenDB (in large datasets) by up to 23%, because EvenDB searches in logs of unpopular funks. We note that this tradeoff between the maintenance cost (for compaction) and the scan performance after maintenance can be controlled via EvenDB's log size limit parameter. (E.g., we saw that scan throughput can grow by up to 20% by tuning the system to use 512KB logs instead of the default 2MB; this experiment is omitted for lack of space.)

5.3 Synthetic benchmarks

In this section, we closely follow the YCSB benchmarking methodology. Each experiment first loads a sequence of KV-pairs, ordered by key, to an initially empty store, then runs read operations to warm the caches, and then exercises – and measures – the specific experiment scenario. Experiments perform 80M data accesses (fewer if some of them are scans). Since the load is in key order, the data stores are sorted from the outset, and RocksDB's files cover disjoint key ranges. This reduces the overhead of compaction and mitigates the stabilization delay observed in §5.2. Thus, performance remains stable throughout the measurement period.

5.3.1 Workloads

We vary the dataset size from 4GB to 256GB in order to test multiple locality scenarios with respect to the available



(a) 64 GB dataset **(b)** 128 GB dataset **(c)** 256 GB dataset

Figure 4. EvenDB vs RocksDB throughput dynamics, scan-dominated (95% scans/5% put) workload with production data.

16GB of RAM. Similarly to the published RocksDB benchmarks [11], the keys are 32-bit integers in decimal encoding (10 bytes), which YCSB pads with a 4-byte prefix (so effectively, the keys are 14 byte long). Values are 800-byte long. We study the following four key-access distributions:

1. Zipf-simple – the standard YCSB Zipfian distribution over simple (non-composite) keys. Key access frequencies are sampled from a Zipf distribution. For most of the experiments, we use the YCSB standard = 0.99, (where the most frequent key occurs 4.87% of the time). We then study the impact of using a smaller θ, i.e., a less skewed distribution. The ranking is over a random permutation of the key range, so popular keys are uniformly dispersed.

2. Zipf-composite – a Zipfian distribution over composite keys. The key's 14 most significant bits comprise the primary attribute, which is drawn from a Zipf (with the same θ) distribution over its range. The remainder of the key is drawn uniformly at random. We also experimented with a Zipfian distribution of the key's suffix and the trends were similar.

3. Latest-simple – a standard YCSB workload reading simple keys from a distribution skewed towards recently added ones. Specifically, the sampled key's position w.r.t. the most recent key is distributed Zipf.

4. Uniform – ingestion of keys sampled uniformly at random. RocksDB reports a similar benchmark [10].

The workloads exercise different mixes of puts, gets, and scans. We use standard YCSB scenarios (A to F) that range from write-heavy (50% puts) to read-heavy (95% – 100% gets or scans). We also introduce a new workload, P, comprised of 100% puts (a data ingestion scenario as in §5.2).

5.3.2 Evaluation results

Figure 5 presents the throughput measurements in all YCSB workloads. Except for workload D, which exercises the Latest-simple pattern (depicted in red), all benchmarks are run with both Zipf-simple (orange) and Zipf-composite (blue). The P (put-only) workload additionally exercises the Uniform access pattern (green). EvenDB results are depicted with solid lines, and RocksDB with dotted lines.

We now discuss the results for the different scenarios.

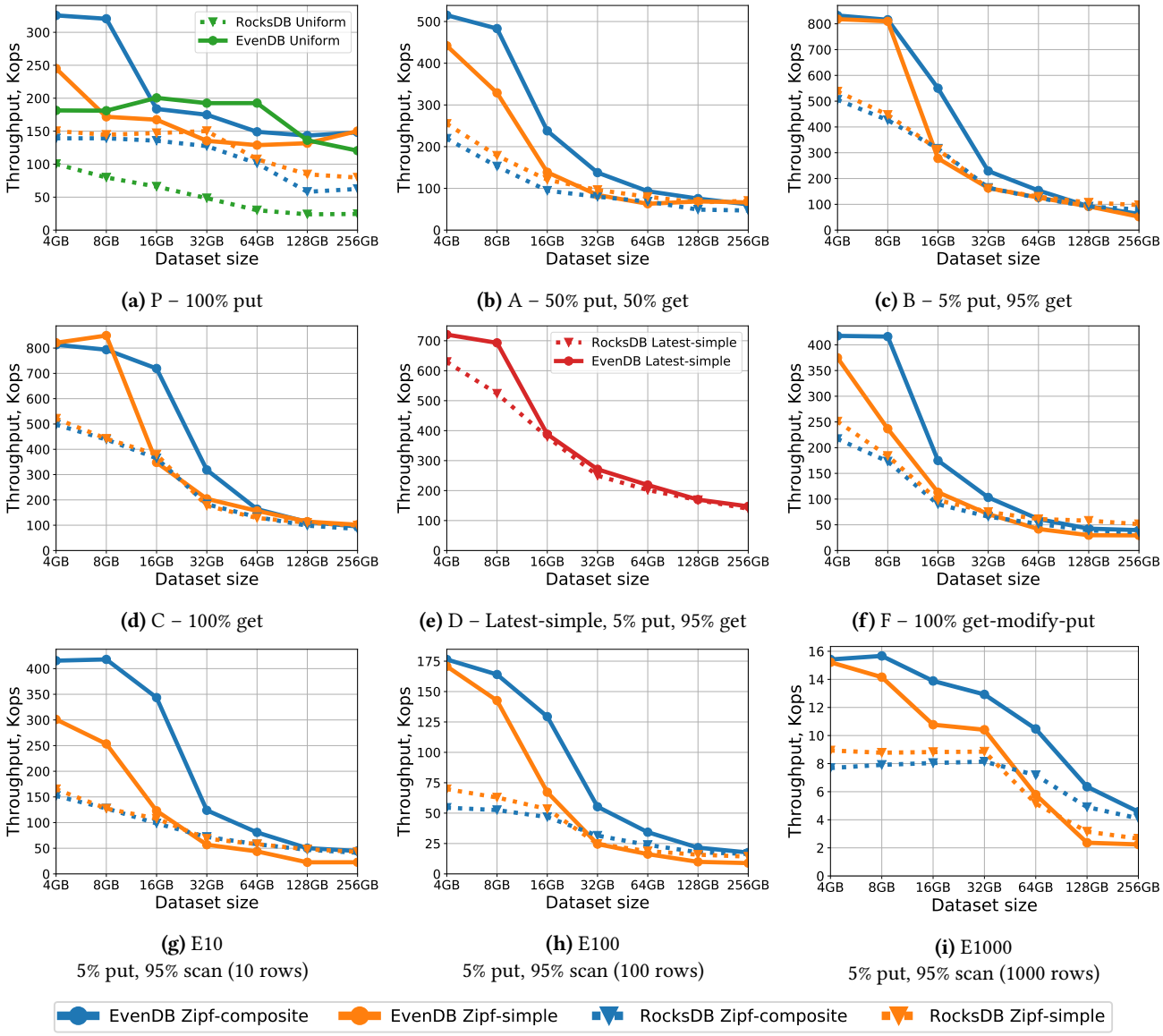
Put-only (data ingestion). In workload P (100% put, Figure 5a), EvenDB's throughput is 1.8× to 6.4× that of RocksDB's with uniform keys, 1.3× to 2.3× with Zipf-composite keys, and 0.9× to 1.6× with Zipf-simple keys. This scenario's bottleneck is the reorganization of persistent data (funk rebalances in EvenDB, compactions in RocksDB), which causes write amplification and hampers performance.

Under the Zipf-composite workload, EvenDB benefits from spatial locality whereas RocksDB's write performance is relatively insensitive to it, as is typical for LSM stores. For small datasets (4-8GB), EvenDB accommodates all puts in munks, and so funk rebalances are rare. In big datasets, funk rebalances do occur, but mostly in munk-less chunks, which are accessed infrequently. This is thanks to EvenDB's high log size limit for chunks with munks. Hence, in both cases, funk rebalances incur less I/O than RocksDB's compactions, which do not distinguish between hot and cold data.

The Uniform workload exhibits no locality of any kind. EvenDB benefits from this because keys are dispersed evenly across chunks, hence all funk logs grow slowly, and funk rebalances are infrequent. The throughput is therefore insensitive to the dataset size. In contrast, RocksDB performs compactions frequently albeit they are not effective (since there are few redundancies). Its throughput degrades with the data size since when compactions cover more keys they engage more files.

The write amplification in this experiment is summarized in Figure 6. We see that EvenDB reduces the disk write rate dramatically, with the largest gain observed for big datasets (e.g., for the 64GB dataset the amplification factors are 1.3 vs 3.1 under Zipf-composite, and 1.1 vs 7.6 under Uniform).

We further measured space amplification at the end of the run, and found that both EvenDB and RocksDB have roughly 17% space amplification. Note that in EvenDB, the space amplification is controlled by the log size threshold parameter.



(a) P – 100% put

(b) A – 50% put, 50% get

(c) B – 5% put, 95% get

(d) C – 100% get

(e) D – Latest-simple, 5% put, 95% get

(f) F – 100% get-modify-put

(g) E10
5% put, 95% scan (10 rows)

(h) E100
5% put, 95% scan (100 rows)

(i) E1000
5% put, 95% scan (1000 rows)

Figure 5. EvenDB vs RocksDB throughput under YCSB workloads with various key distributions.

Mixed put-get. Workloads A (50% put, 50% get, Figure 5b) and F (100% get-modify-put, Figure 5f) invoke puts and gets at the same rate. Note that the latter exercises the usual get and put API (i.e., does not provide atomicity). The get-put mix is particularly challenging for EvenDB, especially with simple keys, where it serves many gets from disk due to the low spatial locality. The bottleneck is the linear search in funk logs, which fill up due to the high put rate. RocksDB's caching is more effective in this scenario, so its disk-access rate in get operations is lower, resulting in faster gets. While EvenDB continues to outperform RocksDB in small data sets or when spatial locality is high, its performance deteriorates in large data sets with no such locality, where RocksDB is almost 2× faster.

Figure 7, which depicts tail (95%) put and get latencies in this scenario, corroborates our analysis. EvenDB has faster puts and faster or similar get tail latencies with composite keys (Figure 7a). With simple keys (Figure 7b), the tail put latencies are similar in the two data stores, but the tail get latency of EvenDB in large datasets surges.

To understand this spike, we break down the get latency in Figure 8. Figure 8a classifies gets by the storage component that fulfills the request, and Figure 8b presents the disk search latencies by component. We see that with large datasets, disk access dominates the latency. For example, in the 64GB dataset, 3.3% of gets are served from logs under Zipf-composite vs 4% under Zipf-simple, and the respective log search latencies are 2.6 ms vs 4.2 ms. This is presumably because in the latter, puts are more dispersed, hence

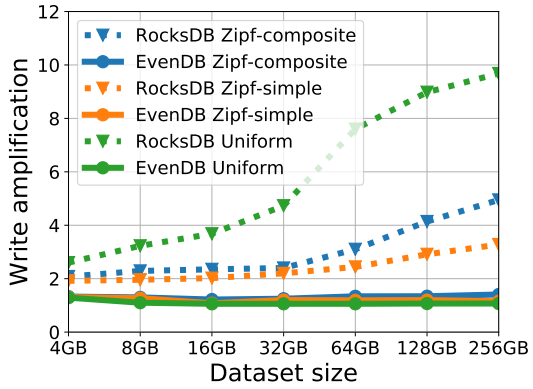
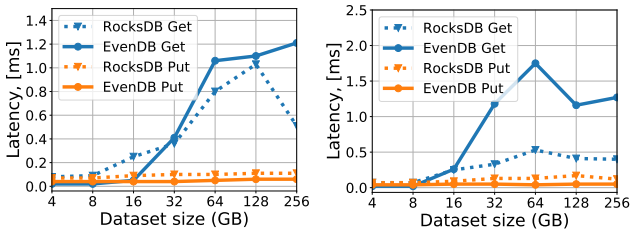


Figure 6. EvenDB vs RocksDB write amplification under the put-only workload P.

the funks are cached less effectively by the OS, and the disk becomes a bottleneck due to the higher I/O rate.



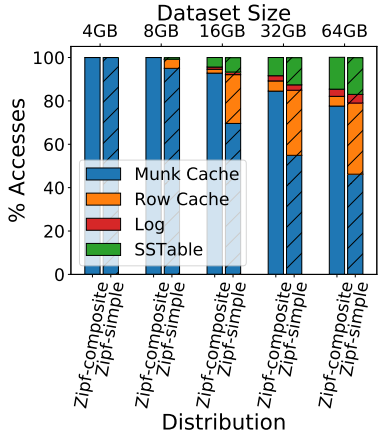
(a) Zipf-composite **(b)** Zipf-simple

Figure 7. EvenDB vs RocksDB 95% latency (ms), under a mixed get-put workload A.

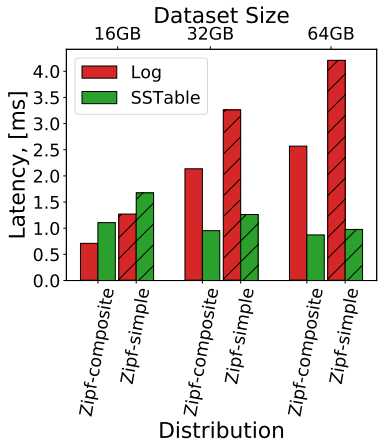
The figure also shows that the row cache becomes instrumental as spatial locality drops – it serves 32.8% of gets with Zipf-simple vs 4.5% with Zipf-composite.

Workloads B and D (Figures 5c and 5e) also mix gets and puts, but with a lower put rate. Here, the funk logs don't grow as quickly as under an even put-get mix, and EvenDB has a marked advantage in all key distributions with small datasets (up to the available RAM size) and also with Zipf-composite and Latest-simple keys in large datasets. But here, too, its advantage diminishes with the data set size, especially under the Zipf-simple key distribution.

Read-only. In workload C (100% get, Figure 5d), EvenDB performs 1.1× to 2× better than RocksDB with composite keys, and up to 1.9× with simple ones (for small datasets). In these scenarios, EvenDB manages to satisfy most gets from munks, resulting in good performance. RocksDB relies mostly on the OS cache to serve these requests and so it pays the overhead for invoking system calls. RocksDB's performance in this scenario can be improved by using a larger application-level block cache, but our experiments (not presented in this paper) have shown that this hurts performance for bigger datasets as well as in other benchmarks.



(a) Fraction of get accesses

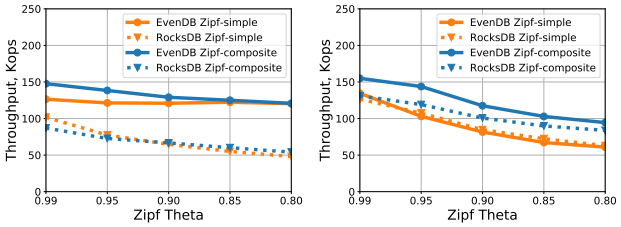


(b) On-disk get access latency

Figure 8. EvenDB get latency breakdown by serving component, under a mixed get-put workload A.

Scan-dominated. Benchmarks E10–1000 (5% put, 95% scan, Figures 5g- 5i) iterate through a number of items sampled uniformly in the range [1,S], where S is 10, 100, or 1000. Under Zipf-composite, this workload exhibits the spatial locality the system has been designed for, and indeed EvenDB outperforms RocksDB in this workload for all data set sizes and all lengths of scans. Its biggest advantage – 3.2× – is achieved with long scans over a small dataset. With simple keys on a large data set, in particular when scans are short, EvenDB begins to suffer from long search times in logs (as this data set also includes puts), and RocksDB has better performance. In §5.5 we show that EvenDB's scan performance on big datasets can be improved by adapting the funk log size limit to this workload.

Impact of skew. We now vary the distribution skew controlled by the parameter θ in the Zipf distribution. Figure 9 shows the impact of skew on performance. Table 3 shows the frequency of the most popular key under each distribution. Note that in the Zipf-composite distribution, only the most



(a) Put only **(b)** Get only

Figure 9. The impact of the Zipf distribution's θ parameter (skew) on throughput, 64GB dataset.

θ =	0.99	0.95	0.90	0.85	0.80
Zipf-simple	4.87	3.30	1.91	1.04	0.54
Zipf-composite	0.013	0.012	0.010	0.009	0.008

Table 3. Frequency (% of occurrences) of the most popular key under different θ values used for Zipf key generation.

significant bits are drawn from a Zipf distribution, hence the much lower frequency. As expected, the performance of both EvenDB and RocksDB deteriorates when the skew is smaller, because both exploit locality. While put performance is impacted similarly in the two data stores, EvenDB's reads are more sensitive than RockDB's reads to the lack of locality. This is because EvenDB's fall-back in case of munk/row cache misses is more costly.

5.4 Additional KV-Stores

We experimented with Percona TokuDB [20]; although deprecated, no newer version is available. The results were at least an order-of-magnitude slower than RocksDB across the board, and therefore we do not present them. Note that this is in line with previously published comparisons [32, 35, 43]. We did not compare EvenDB against InnoDB because the latter is not easily separable from the MySQL code. Yet previous evaluations have found InnoDB to be inferior to both RocksDB and TokuDB under write-abundant workloads [35].

We next compare EvenDB to PebblesDB, which was shown to significantly improve over RocksDB [45], mostly in single-thread experiments, before RocksDB's recent version was released. We compare EvenDB to PebblesDB in a challenging scenario for EvenDB, with a 32GB dataset and the Zipf-simple key distribution. We run each YCSB workload with 1, 2, 4, 8 and 12 threads. The results are summarized in Table 4. While PebblesDB is slightly faster on some single-threaded benchmarks, from 2 threads and onward EvenDB is consistently better in all experiments, with an average performance improvement of almost 1.8×. In all benchmarks, EvenDB's advantage grows with the level of parallelism. We observed a similar trend with smaller datasets.

We note that in our experiments, RocksDB also consistently outperforms PebblesDB. The discrepancy with the results reported in [45] can be attributed, e.g., to RocksDB's evolution, resource constraints (running within a container), a different hardware setting, and increased parallelism.

P	A	B, C	D	E10–1000	F
0.9–2.8×	0.9–1.6×	1.4–2.3×	1–2×	1–3.4×	0.8–1.2×

Table 4. EvenDB throughput improvement over PebblesDB, 32GB dataset, Zipf-simple keys, 1–12 worker threads. EvenDB's advantage grows with the number of threads.

5.5 Insights

Vertical scalability. Figure 10 illustrates EvenDB's throughput scaling for the 64GB dataset under Zipf-composite and Zipf-simple distributions. We exercise the A, C and P scenarios, with 1 to 12 worker threads. As expected, in C (100% gets) EvenDB scales nearly perfectly (7.7× for composite keys, 7.8× for simple ones). The other workloads scale slower, due to read-write and write-write contention as well as background munk and funk rebalances.

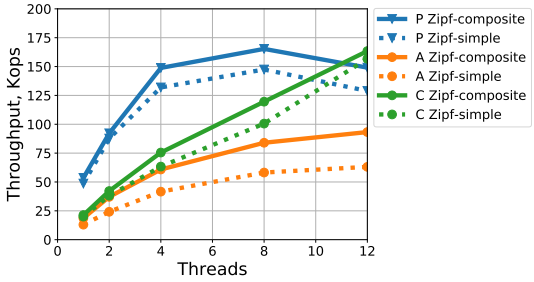
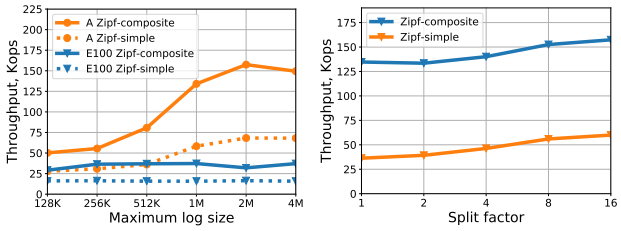


Figure 10. EvenDB scalability with the number of threads for the 64GB dataset and different workloads.



(a) Maximum log size, workloads A and E100 **(b)** Bloom filter split factor, workload A

Figure 11. EvenDB throughput sensitivity to configuration parameters, on the 64GB dataset under A (mixed put-get) and E100 (scan-dominant, 1 to 100 items).

EvenDB configuration parameters. We explore the system's sensitivity to funk-log configuration parameters, for the most challenging 64GB dataset, and explain the choice of the default values.

Figure 11a depicts the throughput's dependency on the log size limit of munk-less funks, under A and E100 with the Zipf-composite key distribution. The fraction of puts

in A is 50% (vs 5% in E), which makes it more sensitive to the log size. A low threshold (e.g., 128KB) causes frequent funk rebalances, which degrades performance more than 3-fold. On the other hand, too high a threshold (4MB) lets the logs grow bigger, and slows down gets. Our experiments use 2MB logs, which favors write-intensive workloads. E favors smaller logs, since the write rate is low, and more funk rebalances can be accommodated. Its throughput can grow by up to 20% by tuning the system to use 512KB logs.

Figure 11b depicts the throughput dependency on the Bloom filter split factor (i.e., the number of Bloom filters that summarize separate portions of the funk log) in workload A. Partitioning to 16 mini-filters gives the best result. The impact of Bloom filter partitioning on EvenDB's memory footprint is negligible.

RocksDB configuration tuning. In RocksDB's out-of-the-box default configuration, the block cache is 8MB. We further experimented with block cache sizes of 1GB, 2GB, 5GB, and 8GB. We note that RocksDB's performance manual recommends allocating 1/3 of the available RAM (~5GB) to the block cache [1]. The results are mixed. For a small dataset (4GB) with composite keys, the block cache effectively replaces the OS pagecache, and improves RocksDB's throughput by 1.3× and 1.6× for workloads C and E100, respectively, by forgoing the system call overhead. This only partly reduces the gap between RocksDB and EvenDB in this setting. However, for bigger datasets (32GB and 64GB), using a bigger block cache degrades RocksDB's performance. We found that the default configuration gives the best results for most of the workload suite. We therefore used this configuration in §5.3 above.

6 Related Work

The vast majority of industrial mainstream NoSQL KV-stores are implemented as LSM trees [12, 14, 18, 29, 38], building on the foundations set by O'Neil et al. [41, 42].

Due to LSM's design popularity, much effort has been invested into working around its bottlenecks. A variety of compaction strategies has been implemented in production systems [27, 49] and research prototypes [22, 36, 45, 46]. Improvements include storage optimizations [36, 39, 44–46], boosting in-memory parallelism [18, 34], and leveraging workload redundancies to defer disk flushes [22, 25] or avoid re-writing sorted data [46]. In contrast to these, EvenDB eliminates the concept of temporally-organized levels altogether and employs a flat layout with in-memory compaction.

SLM-DB [36] is an LSM design that relies on persistent memory and thus eliminates the need for a WAL. It utilizes a B^+-tree index in persistent memory on top of a flat list of temporally-partitioned SSTables. In contrast, EvenDB does not rely on special hardware. Moreover, to the best of our knowledge, SLM-DB does not support concurrent operations.

In-memory compaction has been recently implemented in HBase [25] by organizing HBase's in-memory write store as an LSM tree, eliminating redundancies in RAM to reduce disk flushes. However, being incremental to the LSM design, this approach fails to address spatial locality.

Range-based partitioning is also employed in B-trees [37] and their variants [26]. In §1.2 we discussed the key challenges faced by storage systems adopting these designs, which result in performance disadvantages compared to the LSM approach (as shown, e.g., in [35]), and the difficulty to support consistency – in particular, atomic scans – under multi-threaded access. In contrast to B-tree nodes, EvenDB's chunks are not merely a means to organize data on-disk; they are also the basic units for DRAM caching, I/O-batching, logging, and compaction. This allows us to apply chunk-level logging with sequential I/O and in-memory compaction.

Tucana [43] is an in-memory B^ϵ-tree index over a persistent log of KV-pairs. To speed up I/O, it applies system optimizations that are largely orthogonal to our work: block-device access, copy-on-write on internal nodes, etc. However, Tucana provides neither strong scan semantics nor consistent recovery, and does not support concurrent puts.

A different line of work develops fast in-memory (volatile) KV-stores [6, 9, 16, 48], e.g., for web and application caches. Over time, many evolved to support durability, yet still require the complete data set to reside in memory. Although EvenDB is optimized for "sliding-local" scenarios where most of the active working set is in memory at any given time, it does not expect *all* the data to fit in memory.

Finally, EvenDB's in-munk data management leverages techniques used in concurrent in-memory KV-maps, e.g., partially-sorted key lists [23, 50], lazy index updates [23, 47], and synchronizing scans with puts via a PO array [23]. These aspects are important, but since they do not involve I/O, they do not have a major impact on overall performance.

7 Conclusions

We presented EvenDB – a novel persistent KV-store optimized for workloads with high spatial locality, as prevalent in modern applications. EvenDB provides strong (atomic) consistency guarantees for random updates, random lookups, and range queries. EvenDB outperforms the state-of-the-art RocksDB LSM store in the majority of YCSB benchmarks, with both standard and spatially-local key distributions, in which it excels in particular. EvenDB further reduces write amplification to near-optimal under write-intensive workloads. Finally, it provides near-instant recovery from failures.

Beyond building a particular system prototype, this paper puts forth a novel KV-store design alternative that emphasizes spatial locality. We hope to see future realizations of this approach with various improvements, through novel ideas or ones borrowed from other existing solutions.

References

[1] https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning#block-cache-size.

[2] Appsflyer. https://appsflyer.com.

[3] Flurry analytics. https://flurry.com.

[4] Flurry state of the mobile 2018. https://www.verizonmedia.com/insights/flurry-analytics-releases-2017-state-of-mobile-report.

[5] Google firebase. https://firebase.google.com.

[6] Ignite database and caching platform. https://ignite.apache.org/.

[7] Innodblocking. https://dev.mysql.com/doc/refman/8.0/en/innodb-locking.html.

[8] NoSQL market is expected to reach $4.2 billion, globally, by 2020. https://www.alliedmarketresearch.com/press-release/NoSQL-market-is-expected-to-reach-4-2-billion-globally-by-2020-allied-market-research.html.

[9] Redis, an open source, in-memory data structure store. https://redis.io/.

[10] RocksDB performance benchmarks. https://github.com/facebook/rocksdb/wiki/performance-benchmarks.

[11] RocksDB tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guidel.

[12] Apache hbase, a distributed, scalable, big data store. http://hbase.apache.org/, Apr. 2014.

[13] A fast and lightweight key/value database library by google. http://code.google.com/p/leveldb, Jan. 2014.

[14] A persistent key-value store for fast storage environments. http://rocksdb.org/, June 2014.

[15] Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB. https://github.com/basicthinker/YCSB-C, 2014.

[16] Memcached, an open source, high-performance, distributed memory object caching system. https://memcached.org/, Dec. 2018.

[17] Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb, 2018.

[18] Scylla the real-time big data database. https://www.scylladb.com/, 2018.

[19] Versionset::removefilelevelbloomfilterinfo isn't thread-safe. https://github.com/utsaslab/pebblesdb/issues/19, January 2018.

[20] PerconaFT is a high-performance, transactional key-value store. https://github.com/percona/PerconaFT, 2019.

[21] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. Linkbench: A database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, ACM, pp. 1185–1196.

[22] Balmau, O., Didona, D., Guerraoui, R., Zwaenepoel, W., Yuan, H., Arora, A., Gupta, K., and Konka, P. Triad: Creating synergies between memory, disk and log in log structured key-value stores. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (2017), USENIX ATC '17, pp. 363–375.

[23] Basin, D., Bortnikov, E., Braginsky, A., Golan-Gueta, G., Hillel, E., Keidar, I., and Sulamy, M. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2017), PPoPP '17, ACM, pp. 357–369.

[24] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., and Aiyer, A. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011), SIGMOD '11, pp. 1071–1080.

[25] Bortnikov, E., Braginsky, A., Hillel, E., Keidar, I., and Sheffi, G. Accordion: Better memory organization for lsm key-value stores. *Proc. VLDB Endow. 11*, 12 (Aug. 2018), 1863–1875.

[26] Brodal, G. S., and Fagerberg, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2003), SODA '03, pp. 546–554.

[27] Callaghan, M. Name that compaction algorithm. https://smalldatum.blogspot.com/2018/08/name-that-compaction-algorithm.html, 2018.

[28] Cao, Z., Dong, S., Vemuri, S., and Du, D. H. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *USENIX Conference on File and Storage Technologies (FAST)* (2020).

[29] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[30] Cho, J., Garcia-Molina, H., and Page, L. Efficient crawling through url ordering. In *Proceedings of the Seventh International Conference on World Wide Web 7* (1998), WWW7, pp. 161–172.

[31] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), SoCC '10, pp. 143–154.

[32] Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., and Strum, M. Optimizing space amplification in rocksdb. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings* (2017), www.cidrdb.org.

[33] Einziger, G., Friedman, R., and Manes, B. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Storage 13*, 4 (Nov. 2017), 35:1–35:31.

[34] Golan-Gueta, G., Bortnikov, E., Hillel, E., and Keidar, I. Scaling concurrent log-structured data stores. In *EuroSys* (2015), pp. 32:1–32:14.

[35] Iyer, S. Comparing tokudb, rocksdb and innodb performance on intel(r) xeon(r) gold 6140 cpu. https://minervadb.com/index.php/2018/08/06/comparing-tokudb-rocksdb-and-innodb-performance-on-intelr-xeonr-gold-6140-cpu/, 2018.

[36] Kaiyrakhmet, O., Lee, S., Nam, B., Noh, S. H., and ri Choi, Y. SLM-DB: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019), pp. 191–205.

[37] Knuth, D. E. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[38] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[39] Lu, L., Pillai, T. S., Gopalakrishnan, H., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage 13*, 1 (Mar. 2017), 5:1–5:28.

[40] Matsunobu, Y. Myrocks: A space- and write-optimized mysql database. https://engineering.fb.com/core-data/myrocks-a-space-and-write-optimized-mysql-database/, 2016.

[41] Muth, P., O'Neil, P. E., Pick, A., and Weikum, G. Design, implementation, and performance of the lham log-structured history data access method. In *Proceedings of the 24rd International Conference on Very Large Data Bases* (1998), VLDB '98, pp. 452–463.

[42] O'Neil, P. E., Cheng, E., Gawlick, D., and O'Neil, E. J. The log-structured merge-tree (lsm-tree). *Acta Inf. 33*, 4 (1996), 351–385.

[43] Papagiannis, A., Saloustros, G., González-Férez, P., and Bilas, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference* (2016), USENIX ATC '16, pp. 537–550.

[44] Papagiannis, A., Saloustros, G., González-Férez, P., and Bilas, A. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, ACM, pp. 490–502.

[45] Raju, P., Kadekodi, R., Chidambaram, V., and Abraham, I. Pebblesdb: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17, pp. 497–514.

[46] Shetty, P. J., Spillane, R. P., Malpani, R. R., Andrews, B., Seyster, J., and Zadok, E. Building workload-independent storage with vt-trees. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (2013), pp. 17–30.

[47] Spiegelman, A., Golan-Gueta, G., and Keidar, I. Transactional data structure libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), PLDI '16, pp. 682–696.

[48] Srinivasan, V., Bulkowski, B., Chu, W.-L., Sayyaparaju, S., Gooding, A., Iyer, R., Shinde, A., and Lopatic, T. Aerospike: Architecture of a real-time operational dbms. *Proc. VLDB Endow. 9*, 13 (Sept. 2016), 1389–1400.

[49] Tribble, P. How to Ruin Your Performance by Choosing the Wrong Compaction Strategy. https://www.scylladb.com/2017/12/28/compaction-strategy-scylla/, 2017.

[50] Wu, X., Ni, F., and Jiang, S. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, ACM, pp. 18:1–18:16.