# EvenDB: Optimizing Key-Value Storage for Spatial Locality


## Abstract

Applications of key-value (KV-)storage often exhibit high *spatial locality*, such as when data items with identical composite key prefixes are created or scanned together. This prevalent access pattern is underused by the ubiquitous LSM tree design underlying KV-stores today.

We present EvenDB, a general-purpose persistent KV-store optimized for spatially-local workloads. EvenDB forgoes the temporal data organization of LSM trees and instead partitions data by key to exploit locality. It does so via a novel design that ensures consistency under multi-threaded access and offers much faster in-memory access than existing KV-stores, while also reducing write amplification.

Experiments with real-world data from a large analytics platform show that EvenDB consistently outperforms the state-of-the-art. For example, on a 256GB analytics dataset, EvenDB ingests data 3.7x faster than RocksDB, scans recently ingested data up to 27% faster, and reduces write amplification by nearly 4x. EvenDB further outperforms existing solutions whenever the system has sufficient DRAM to hold most of the active working set. In traditional YCSB workloads with working sets that are larger than the available DRAM size, EvenDB is on par with RocksDBS and significantly outperforms other open-source solutions we tested.



**ACM Reference format:**
. 2019. EvenDB: Optimizing Key-Value Storage for Spatial Locality. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 15 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 [4].

KV-stores provide a simple programming model. Data is an ordered collection of key-values 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 first attribute – called the *primary key* – has a skewed distribution, and so access via the composite key exhibits *spatial locality*, as popular primary keys result in popular *key ranges*.

One example of this arises in real-time mobile analytics platforms such as Google Analytics or Flurry Analytics.

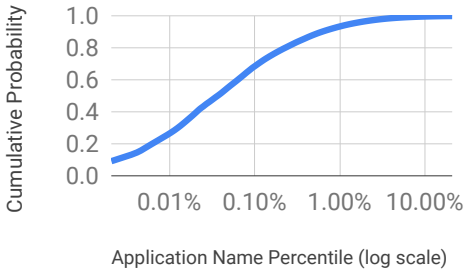



**Figure 1.** CDF of app names in a real-world mobile analytics event stream consisting of 200 million events.

These platforms ingest massive streams of event reports (e.g., app start/stop, exercising a particular code path within an app) from over a billion mobile users. The data store is indexed by a composite key consisting of app name (or id) and additional attributes (user id, device, time, location, event type, etc.) We examine a real trace of 200 million events captured from a production mobile analytics engine over the course of 4 minutes. Figure 1 shows the distribution of app names in this stream. Although the stream includes reports from 40K apps, we find that 10% of the apps cover over 99.5% of the events; 1% of the applications cover 94% of the events; and less than 0.1% cover 70% of the events.

Composite keys arise in many additional domains, including messaging and social networks. For example a query in Facebook Messenger may retrieve the last 100 messages for a given user [22], where the primary key is user id. In Facebook's social network, a graph edge is indexed by a key consisting of two object ids and an association type [17]. Note further that spatial locality arise also with simple (non-composite) keys, for example, when reverse URL domains are used as keys for web indexing [27].

The prevalence of skewed access in real workloads is widely-recognized, and indeed standard benchmarks (e.g., YCSB [29]) feature skewed key-access distributions like Zipf. But these benchmarks fail to capture the spatial aspect of locality. This, in turn, leads to storage systems being optimized for a skewed distribution on individual keys with no spatial locality, e.g., by partitioning data by recent access time as opposed to by key. In this work, we make spatial locality a first class consideration in KV-store design.

The de facto standard approach to building KV-stores today is *LSM* (log-structured merge) trees [40]. The LSM design initially groups writes into files *temporally*, and not 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 (lower level) files are always merged with old (higher level) ones, "cold" key ranges continue to be repeatedly re-located by compactions.

Additionally, we note that LSM's temporal organization optimizes disk I/O but induces a penalty on in-memory operation. All keys – including popular ones – are flushed to disk periodically, even though persistence is assured via a separate *write-ahead-log (WAL)*. Beyond increasing write amplification, this makes the flushed keys unavailable for fast read from memory, which is wasteful if the system incorporates sufficient DRAM to hold most of the active working set. The drop in DRAM prices (more than 6.6x since 2010 [12]) makes the latter scenario increasingly common.

### 1.2 Designing for spatial locality

The main goal of this work is to draw attention to the importance of spatial locality in today's KV-store workloads and to propose a design alternative that is better suited for such locality. Obviously, we do not claim that spatial data partitioning is new; indeed, classical B-trees [20, 28] pre-date LSM trees, and many B-tree variants [21, 24, 37] have emerged over the years. However, it is important to note that these trees are conceptual constructs rather than storage systems; employing these concepts within a practical data store over a memory hierarchy raises multiple challenges, which perhaps explains their limited adoption in industrial KV-stores.

A key challenge is what to persist when and in what format. In classical B-trees, all updates induce random I/O to leaves, resulting in poor write performance. A $B^\epsilon$-tree [24] reduces this overhead using write buffers in internal tree nodes. However, this slows down lookups, which now have to search in multiple unordered buffers. If internal nodes reside on disk, the lookup time is unacceptably slow, whereas if they reside in RAM then they have to be complemented by a separate persistence mechanism.

A second challenge is ensuring consistency – in particular, atomic scans – in the face of concurrent access. This can be tricky when a scan spans both memory-resident and disk-resident data. In fact, we are not aware of any published solution addressing atomic scans and persistence with concurrent access (multi-threading) in B- or $B^\epsilon$-trees.

We present EvenDB, a high-performance persistent KV-store geared towards spatial locality. We forgo LSM's temporal organization and instead partition data by key, as in B- and $B^\epsilon$-trees. Data is organized as a list (rather than a tree) of large *chunks* holding contiguous key ranges. An auxiliary volatile index provides direct access (one I/O operation) to the on-disk chunk holding a given key. EvenDB supports atomic scans using low-overhead multi-versioning, where versions are increased only by scans and not by updates.

Like LSMs, EvenDB optimizes I/O by absorbing updates in memory before writing to disk. We achieve this without a temporal data organization thanks to (1) caching popular chunks in RAM and (2) using per-chunk WALs to transform random I/O to sequential I/O at the chunk level.

Spatially-organized chunks have a number of advantages: (1) there is little fragmentation of key ranges and hence better performance for workloads with spatial locality; (2) in-memory chunk compaction saves disk flushes and reduces write volume; (3) keeping hot ranges in memory leads to better performance, especially when most of the working set fits in RAM; and (4) in-chunk WALs allow quick recovery from crashes with no need to replay any WALs.

The downside of spatial partitioning is that if the data lacks spatial locality and the active working set is big, caching an entire chunk for a single popular key is wasteful. We mitigate this by adding a dedicated *row cache* for hot keys. But since the row cache only serves the read-path, this design is less optimal for mixed read/write workloads lacking spatial locality, where the LSM approach may be more suitable.

### 1.3 Contributions and roadmap

EvenDB is a novel KV-store, which organizes data differently from classical LSM, B-, and $B^\epsilon$-trees; §2 presents our key design goals and choices. The EvenDB algorithm, detailed in §3, supports concurrency among any number of threads performing put, get, and scan operations, as well as background maintenance (compaction). It ensures consistency and correct recovery from failures. We provide details on its C++ implementation in §4.

In §5, we extensively evaluate EvenDB. We experiment with three types of workloads: (1) a production trace collected from a large-scale mobile analytics platform; (2) workloads with synthetically-generated composite keys exercising the standard scenarios of the popular YCSB benchmark suite [29]; and (3) YCSB's traditional benchmarks, which employ simple (non-composite) keys. In all cases, we compare EvenDB to the recent (Oct 2018) release of RocksDB [10], a mature industry-leading LSM KV-store. We further experimented with two additional open-source KV-stores, (1) the LSM-based PebblesDB [43], and (2) PreconaFT/TokuDB [14] – the only publicly-available KV store that uses a $B^\epsilon$-tree design. However, both of these performed significantly worse than both RocksDB and EvenDB across the entire YCSB benchmark suite, which is in line with previous studies of PerconaFT [41], so we excluded them from further tests.

Our experiments show the following: (1) EvenDB significantly outperforms RocksDB when spatial locality is high. For instance, on a 256GB production dataset, EvenDB ingests data 3.7x faster than RocksDB, scans recent data up to 27%

faster, and reduces write amplification by nearly 4x. And in large synthetic composite-key workloads, EvenDB improves over RocksDB's throughput by 24% – 75%. (2) EvenDB significantly outperforms RocksDB whenever most of the working set fits in RAM. For example, with synthetic composite keys and a memory-resident working set, EvenDB accelerates scans by up to 3.5x, puts by up to 2.3x, and gets by up to 2x. (3) In traditional YCSB workloads, EvenDB is comparable to RocksDB. As can be expected, RocksDB performs better than EvenDB in mixed read/write workloads with large active working sets and no spatial locality.

Finally, §6 surveys related work and §7 concludes.

## 2 Design Principles

### 2.1 Guarantees and optimization goals

EvenDB is a persistent ordered key-value store. Similarly to popular industrial KV-stores [8–10], it supports concurrent access by multiple threads and ensures strong consistency. Specifically its *put*, *get*, and *range scan* (or 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.

Additionally, EvenDB ensures *consistent recovery*: following a crash, the system recovers to a well defined execution point some time before the crash. When persistence is *asynchronous*, puts are buffered and persisted to disk 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 all ensuing (and thus possibly dependent) puts are lost as well.

Our key optimization goals are the following:

1. *Focus on spatial locality.* Many NoSQL applications embed multi-dimensional data in a single-dimension composite key. This design provides high spatial locality on the primary dimension (key prefix). We strive to express this locality in physical data organization.
2. *Low write amplification.* We seek to minimize disk writes in order to boost performance and reduce disk wear, especially for SSD devices.
3. *High performance with memory-resident working sets.* To sustain high speed, key-value stores nowadays leverage increasing RAM sizes where they can hold most of the active working set. We strive for maximum performance in this "hyper-local" case. Note that we do *not* expect the entire database to fit in memory, only the active working set.
4. *Fast recovery.* Because crashes are inevitable, the downtime they entail should be kept short.

### 2.2 Design choices

***Data organization and caching.*** EvenDB combines the spatial locality of B-trees with the optimized I/O and quick access of LSM stores. Its data layout is illustrated in Figure 2a. For spatial locality, we organize data, both on-disk and in-memory, in large chunks pertaining to key ranges. Each chunk has a file representation called *funk* (file chunk), and may be cached in a memory data structure called *munk* (memory chunk).

Chunks are organized in a linked list. To speed up key access, we index the chunks using a volatile index (a sorted array in our implementation). We also partially sort the keys in each chunk for fast in-chunk search. To expedite access to keys whose chunks are only on-disk (i.e., have no munks), individual popular keys are cached in a *row cache*, and *Bloom filters* are used to limit excessive access to disk.

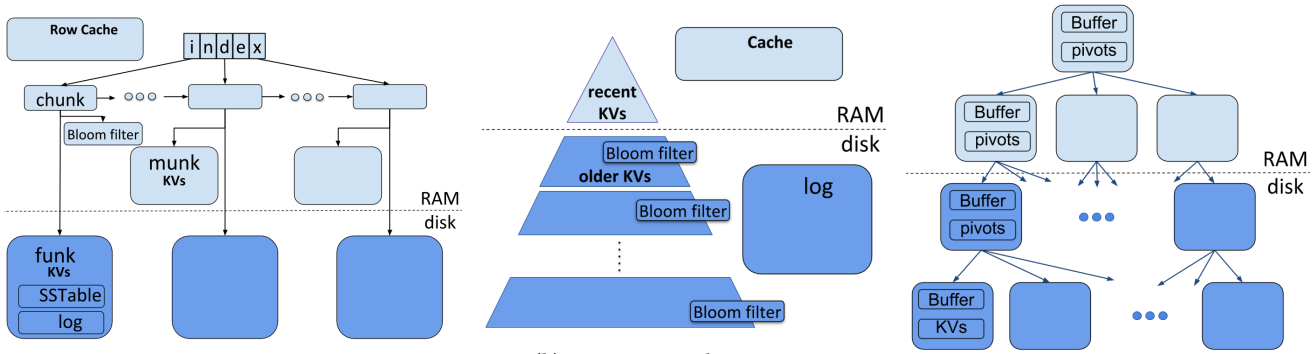
For comparison, an LSM tree (Figure 2b), organizes data temporally, keeping only the most recent updates in memory. The in-memory (top level) store optimizes the write-path. To optimize the read path, it uses a large cache. For persistence, data is logged to a WAL (write-ahead log).

Another related data organization is that of B-trees and $B^\epsilon$-trees (Figure 2c), which also partition data by key range. $B^\epsilon$-trees use in-node update buffers, which resemble EvenDB's in-funk WALs. However, their buffers are not used for persistence — their intent is to amortize the $O(\log n)$ tree traversal cost across multiple updates.

EvenDB logs writes within funks and avoids duplicating the updates in a separate WAL. This reduces write amplification and expedites recovery. The log is merged into the chunk's sorted key-value table (*SSTable* – Sorted String Table [26]) via an infrequent background compaction process. While a chunk is cached (has a munk), there is no urgency to improve its on-disk organization since queries do not access it. Therefore, EvenDB infrequently performs reorganization (compaction) on such funks, and compacts them only in-memory. Conversely, when a funk holds cold data, its organization hardly deteriorates, so no compaction is needed. Note that this is unlike LSM trees, where all disk components are compacted, regardless of which keys reside in memory.

***Concurrency and multi-versioning.*** EvenDB allows high parallelism among threads invoking its API calls. Read 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. Thus, version management incurs a low overhead (as 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.

To support atomic scans, production LSM stores employ standard MVCC on top of the LSM tree. Each write operation is assigned an auto-incremented sequence number. Since




**(a)** EvenDB: data partitioned by key

**(b)** LSM: temporal partitioning

**(c)** B$^\epsilon$ tree: data partitioned by key

**Figure 2.** EvenDB's data layout compared to LSMs and B$^\epsilon$ trees.

each write adds an additional version to the key it writes to, the memory component often overflows even if the working set is small enough to fit in memory. This causes frequent data compactions and increases write amplification.

## 3 EvenDB's Design

In §3.1 we discuss EvenDB's data organization. We move to atomic scans in §3.2, and summarize EvenDB's normal (maintenance-free) operation flow in §3.3. The data structure's maintenance is discussed in §3.4. Finally, §3.5 discusses data flushes and failure recovery.

### 3.1 Data organization

***Chunks, funks, and munks.*** EvenDB's data layout is depicted in Figure 2a. Data resides in chunks, each holding a contiguous key range. This improves the efficiency of both disk and memory accesses, mostly for range scans. Each chunk's data (consisting of keys in the corresponding range and values associated with them) is kept on disk (funks, for persistence), and possibly in memory (munks, for fast access). Chunk metadata objects are significantly smaller than munks and funks (typically, less than 1 KB vs. tens of MBs). EvenDB thus always holds them in memory. Further, it keeps a volatile *chunk index*, mapping keys to chunks. Index updates are lazy, offering only best-effort expedited search.

A chunk consists of two files: a compacted and sorted key-value map *SSTable* and a write *log*. When a funk is created, the SSTable holds all the chunk's keys with corresponding values, and the log is empty. New key-value pairs are subsequently appended to the unsorted log. If a key is over-written, its old value remains in the SSTable, and the new one value is included in the log. Namely, the log is more up-to-date.

This structure allows us to benefit from sorted searches on the SSTable, and at the same time allows for updating chunks without re-writing 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.

A subset of the chunks is also cached (as munks) in memory to allow fast access. Munks are volatile and can be removed and recreated from funks at any time based on an arbitrary replacement policy.

A munk holds key-value 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 key-value 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 at the suffix of the array. This approach was previously used in in-memory data structures, e.g., Kiwi [19].

As key-value pairs are added, overwritten, and removed, munks and funks need to undergo reorganization. This includes (1) *compaction* to garbage-collect removed and over-written data, (2) *sorting* keys to make searches more efficient, and (3) *splitting* overflowing chunks. All reorganizations are performed by EvenDB's *rebalance* operation. If the chunk has a munk, the rebalance creates a new compacted and sorted munk instead of the existing one. Note that munk rebalance happens in-memory, independently of munk replacement and disk flushes. Funks are also compacted and replaced by new funks, albeit less frequently. Splits create new chunks as well as new funks (and possibly munks).

***Expediting reads.*** As long as a chunk is memory-resident, the munk data structure serves both the read-path and the write-path for keys in this chunk's range. The chunk metadata is quickly located using the index, and its munk's sorted

prefix allows for fast binary search. Thus, EvenDB is particularly fast when almost the entire working set is memory-resident. We take two measures to mitigate the performance penalty of accessing keys in non-cached chunks.

First, we keep a *row cache* holding popular key-value pairs 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. Whereas, popular key ranges can be scanned quickly from munks, isolated hot keys can be quickly found using the row cache.

For working sets that are 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 log searches, a chunk with no munk holds a *Bloom filter* for the corresponding funk's log, which eliminates most of the redundant log searches. To reduce the log search time in case it does happen, we further partition the Bloom filter into a handful of filters, each summarizing the content of part of the log; this allows us to know not only whether or not to search the log, but also which part of it to search.

***Thread synchronization variables.*** EvenDB supports concurrent execution of all operations. The replacement of a chunk (due to a split) or rebalance of a funk or munk must be executed atomically, and moreover, must be synchronized with concurrent puts. This is controlled by the chunk's *rebalanceLock*, which is held for short time periods during chunk, funk, and munk replacements. It is a shared/exclusive lock, acquired in shared mode by puts and in exclusive mode 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. Rebalance makes the new chunk accessible only after it is populated. Because a chunk is immutable during rebalance (while rebalance lock is in exclusive mode), reading either the old or the new version is acceptable.

To minimize I/O, we allow at most one thread to rebalance a funk at a given time; this is controlled by the funkChangeLock. This lock is held throughout the creation of the new funk. It is acquired using a try_lock call, and threads that fail to acquire it do not retry, but instead wait for the winning thread to complete the funk's creation.

In addition to the chunk list and chunk index, EvenDB keeps a *global version (GV)* for supporting atomic scans (described in the next section) and tracks active threads' activities in the *Pending Operations (PO)* array, which has one entry per active thread. The PO is used to synchronize puts with scans, as well as for garbage collection purposes (old versions not needed by any active scan can be reclaimed). We note that using a single pending array to synchronize all operations can cause contention, which we eliminate in our implementation by tracking the pending puts in per-chunk arrays with per-thread entries as done in [19].

### 3.2 Multi-versioning for atomic scans

We support atomic scans via multi-versioning using the system-wide global version, GV. A scan operation creates a *snapshot* associated with GV's current value by incrementing GV, which signals to ensuing put operations that they must not overwrite values with the highest version smaller than the new GV value. This resembles a *copy-on-write* approach, which virtually creates a snapshot by indicating that data pertaining to the snapshot should not be modified in place.

To allow garbage collection of old versions, EvenDB tracks snapshot times of active scans in the pending operations array, PO. The compaction process that runs as part of rebalance removes old versions that are no longer required for any scan listed in PO. Specifically, for each key, it removes all but the last 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. Consider a put, which 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. The scan then proceeds to read the appropriate chunk, missing 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 as explained below. This mechanism is a simplification of the non-blocking put-scan synchronization employed (and proven correct) in [19].

### 3.3 Normal operation flow

***Get and put.*** Algorithm 1 presents pseudocode for the normal operation flow, without rebalance. 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 searches the index first. However, index updates are lazy – they occur after the new chunk is already in the linked list – and therefore the index may return a stale chunk that has already been replaced by rebalance or miss newly added chunks. 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. A similar approach was used in earlier works [19, 45].

Gets proceed without synchronization. If the chunk has a munk, get locates the key by first using a binary search on its sorted prefix and then traversing linked list edges. Otherwise, the get looks for the key in the row cache. If not found, it queries the Bloom filter to determine if the key might

**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          ▷ fetch and increment global version
15:     PO[i] ← ⟨put, key, gv⟩                ▷ and write 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
20:         update ⟨key, val⟩ in row cache (if key is present)
21:     unlock(C.rebalanceLock)
22:     PO[i] ← ⊥

23: procedure SCAN(key1, key2)
24:     PO[i] ← ⟨scan, key1, key2, ⊥⟩  ▷ publish scan's intent
25:     gv ← F&I(GV)                    ▷ read 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
```

be present in the target chunk's log, and if so, searches for it there. If the key is in non of the above, the SSTable is searched.

Upon locating the chunk, a put grabs its rebalanceLock in shared mode to ensure that the chunk is not being rebalanced during the put operation. 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 key-value 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. In case the put's version is the same as the current latest one, it over-writes the current one in the munk but appends a new one in the funk's log.

A per-chunk monotonically increasing counter is used to determine the order of concurrent put operations updating the same key (similarly to [19]). We enforce updates to occur in order of version-counter pairs by attaching it to the value in the munk, PO, the funk, and the row cache. Following a split, the new chunks inherit the counter from the one they replace. Finally, a put unregisters itself from PO, indicating completion, and releases the chunk's lock .

***Scan.*** A scan first publishes its intent to obtain a version in PO, to signal to concurrent rebalances to not 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, which 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

Rebalance is used to improve data organization in a funk or munk by removing old versions that are no longer needed for scans, removing deleted items, and sorting all the keys in the chunk. It is typically triggered when the munk (funk) exceeds a capacity threshold. In case a chunk has a munk, rebalance reorganizes only the munk, since all searches are served by it. The respective funk is reorganized much less frequently, merely in order to bound the log's growth. Both types of rebalance operations create new munks (funks) rather than reorganize in-place, in order to reduce impact on the concurrent accesses. At the end of data reshuffle, EvenDB flips the chunk's reference to the new munk or funk.

EvenDB executes munk rebalance in the context of the application thread that triggered it. The algorithm blocks the puts to the munk throughout the rebalance, so that the concurrent gets and scans can exploit the munk's immutability and proceed without synchronization. The chunk's rebalanceLock, which is acquired in shared mode by puts and in exclusive mode by rebalance, serves this purpose (see §3.1).

The algorithm iterates through the PO to collect the minimum version number among the active scans that cannot be reclaimed yet. Since each rebalance operates on a single chunk with a known key range, scans targeting non-overlapping ranges are ignored. If a scan published its intent

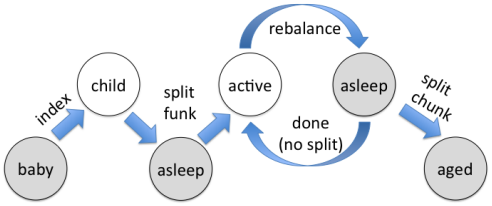



**Figure 3.** Chunk life cycle; immutable states are grey and mutable ones are white. Chunk splits create new chunks in immutable *baby* state, which changes to the mutable *child* state once they are indexed. When the appropriate funks are created, the chunks become *active*. All rebalance operations go through an *asleep* state when the chunk is immutable.

in the PO but published no version yet, the rebalance waits until the version is published. When the new munk is ready, the munk reference is flipped and rebalanceLock is released.

Funk rebalance creates a new (SSTable, log) pair. EvenDB executes this procedure in a dedicated daemon thread. If the chunk has a munk, the most efficient way of performing funk rebalance is through munk rebalance that is followed by the log's truncation and the reordered munk's flush to the new SSTable. Otherwise, the new SSTable is created by merge of the old SSTable with its log. This procedure involves I/O and may take a long time, so we do not block puts for the most of its duration. The puts that happened after the merge started get logged but do not contribute to the merge result. At the end, EvenDB performs a catch-up: (1) new puts are blocked using the rebalanceLock, (2) the unaccounted puts are copied to the new log, (3) the funk reference is flipped, and (4) rebalanceLock is released. Simultaneous rebalance of the same funk by two threads is prevented in order to avoid redundant work, through a separate exclusive lock (see §3.1).

***Splits and chunk life cycle.*** If a munk rebalance exceeds the data threshold in a new munk, it resorts to chunk split. The algorithm works as follows: (1) split the munk into two ordered halves (as an optimization, reuse the first half already created by the normal rebalance), (2) create two new chunks, each referencing a different new munk but temporarily sharing the same funk, (3) insert the new chunks into the list after the old chunk, then insert them into the index, (4) remove the old chunk from the index and the list, (5) split (in the background) the shared funk in two, using the new munks, and update the funk references. Note that the old chunk remains immutable throughout the process.

In order to guarantee correctness of this multi-step process, we model a chunk's life cycle as a finite state machine, depicted by Figure 3. A new chunk is created in *baby* state, in which it is immutable. Once added to the index, a chunk transitions to *child* state, in which it is mutable but cannot be rebalanced yet. Finally, when a chunk gets its own funk it becomes *active* – i.e., eligible for all operations. A chunk that has been split into two new chunks becomes *aged* – a state in which it remains immutable but can serve the gets and the scans that are concurrently accessing it via the index. Once it gets removed from the chunk list and has no outstanding reads, it can be disposed. We omit the formal correctness proof for lack of space; see [19] for similar proofs.

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

### 3.5 Disk flushes and recovery

All puts write to the funk log, regardless of whether the chunk has a munk. Like most popular KV-stores, EvenDB supports two modes of operation – *synchronous* and *asynchronous*. In the former, updates are persisted to disk before returning to the user, ensuring the written data will survive failures once the operation completes. The drawback of this approach is that it is roughly an order-of-magnitude slower than the asynchronous mode in existing KV-stores like RocksDB [10] as well as in EvenDB. In asynchronous mode, updates to funk logs are passed to the OS but might not be physically persisted until an explicit fsync call. Asynchronous I/O 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; 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 suffix 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 snapshot* of the data store is important, since later updates may depend on earlier ones.

***Checkpointing for consistent recovery.*** To support recovery to a consistent snapshot in asynchronous mode, we use a background process to periodically create and persist *checkpoints* of the data store. The checkpointing process creates a snapshot similarly to the atomic scan mechanism. It begins by fetching-and-incrementing 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 flushes all the pending writes to disk (using fsync). Once the flush completes, it writes gv to a dedicated *checkpoint file* on disk. This enforces the correctness invariant: at all times, all updates pertaining to versions smaller than or equal to the version recorded in the checkpoint file have

| epoch | last checkpointed version |
|---|---|
| 0 | 1375 |
| 1 | 956 |

**Table 1.** Example recovery table during epoch 2.

been persisted. Note that some puts with higher versions than gv might be reflected on disk while others are not.

EvenDB recovery is lazy. All data is initially on disk, fetched into munks upon need during the course of the normal operation mode. 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 number 0 in the new epoch.

# 4 Implementation

We implement EvenDB in C++. We borrow the SSTable implementation from the RocksDB open source [10]. 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 simple LFU eviction policy. We use exponential decay to maintain the recent access counts, similar to [30]: 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 must never serve stale values. Therefore, a put updates the cache if a previous version of that key is already in the cache. If the key is not present in the cache, the put does not update 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 baseline for our evaluation is RocksDB – a mature and widely-used industrial KV-store. We use the most recent RocksDB release 5.17.2, available Oct 24, 2018. It is worth noting that RocksDB's performance has significantly improved over the last two years, primarily through optimized LSM compaction algorithms [25]. We also compare against PebblesDB, a research LSM tree prototype that reduces write amplification and demonstrated performance advantages over RocksDB in some scenarios [43]. We further attempted to experiment with TokuDB [14] – the only publicly available KV-store whose design is inspired by $B^\epsilon$-trees. However, TokuDB crashed in all executions with more than one thread, and in all single-threaded executions its performance was inferior to EvenDB's, hence these results are not presented.

The experiment setup is described in §5.1. Performance results for EvenDB and RocksDB in different workloads are presented in §5.2. We then study EvenDB's sensitivity to different configuration settings and its scalability in §5.3. Finally, §5.4 compares EvenDB with PebblesDB.

## 5.1 Setup

***Testbed.*** We employ a C++ implementation [11] of YCSB [29], the de facto standard benchmarking platform for KV-stores. YCSB provides a set of APIs and a workload suite.

A YCSB workload is defined by (1) the ratios of get, put, and scan accesses, and (2) a synthetic distribution of key access frequencies. YCSB benchmarks are inspired by real-life applications. A typical YCSB experiment stress-tests the backend KV-store through a pool of concurrent worker threads that drive identical workloads.

Our hardware is a 12-core Intel Xeon 5 machine with 4TB SSD disk. Unless otherwise stated, the YCSB driver exercises 12 workers. In order to guarantee fair memory allocation across KV-stores, we run each experiment within a Linux container with 16GB RAM.

***Metrics.*** Our primary performance metrics are *throughput* and *latency percentiles*, as produced by YCSB. In addition, we measure *write amplification*, namely, bytes written to storage over bytes passed from the application.

***Workloads.*** We vary the dataset size from 4GB to 64GB in order to exercise multiple locality scenarios with respect to the available 16GB of RAM. Similarly to the published RocksDB

benchmarks [7], the keys are 32-bit integers encoded in decimal form (10 bytes), which YCSB pads with a fixed 4-byte prefix (so effectively, the keys are 14 byte long). The values are 800-byte long. The data is stored uncompressed.

We study the following key-access distributions:

**Zipf-simple** – the standard YCSB Zipfian distribution over simple (non-composite) keys. Key access frequencies are sampled from the heavy-tailed Zipf distribution following the description in [32], with $\theta = 0.8$. The ranking is over a random permutation of the entire key range, so popular keys are uniformly dispersed. This workload exhibits medium temporal locality (e.g., the most popular key's frequency is approximately 0.7%) and no spatial locality.

**Zipf-composite** – a Zipfian distribution over composite keys. The key's 14 most significant bits comprise the primary attribute. The primary attribute is drawn from a Zipf ($\theta = 0.8$) distribution over its range. The remainder of the key is drawn uniformally at random. We also experimented with a Zipfian distribution of the key's suffix; the performance trends were similar since performance is most affected by the primary dimension's distribution.
Zipf-composite exhibits high spatial locality. It represents workloads with composite keys, such as message threads [22], social network associations [17], and analytics databases [2], and other scenarios with spatial locality of keys, e.g., reverse URL domains.

**Latest-simple** – reading recently added simple keys. Keys are inserted in sequential order; the read keys' distribution is skewed towards recently added ones. Specifically, the sampled key's position wrt the most recent key is distributed Zipf. This is a standard YCSB workload with medium spatial and temporal locality, which represents, e.g., status updates and reads.

**Uniform** – ingestion of keys in random order (the keys are sampled uniformly at random). RocksDB reports a similar benchmark [6]; we present it for completeness.

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). In order to stress the system even more on the write side, we introduce a new workload, P, comprised of 100% puts. It captures a non-sequential data load scenario (e.g., an ETL from an external data pipeline [2]).

***Methodology.*** Each experiment consists of three stages. The first stage builds the dataset by filling an initially empty store with a sequence of KV-pairs, ordered by key. The second phase is warm-up, running read-only operations to warm the caches. The third phase exercises the specific scenario; all worker threads follow the same access pattern. We measure performance only during the steady state (skipping the load and warm-up phases). Most experiments perform 80 million data accesses. Experiments with scans perform 4 to 16 million accesses, depending on the scan size.

To make sure that the results are reproduced, we run 5 experiments for each data point and present the median measurement. 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%. To avoid cluttering the graphs, we do not present the STD.

***Configuration.*** We focus on the asynchronous logging mode. Synchronous logging is approximately 10x slower, trivializing the results of scenarios that includes puts.

All experiments in §5.2 use the same default configuration, to avoid over-tuning; §5.3 then explores different configurations and provides insights on parameter choices.

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, and the rebalance size trigger to 7MB. The funk log size limit is 2MB for munk-less chunks, and 20MB for chunks with munks.

We evaluate RocksDB with its default configuration, which is also used by its public performance benchmarks [7]. In §5.3, we follow the RocksDB performance guide [1] to tune its memory resources. The results do not affect our conclusions significantly. We run PebblesDB with the default configuration after fixing a data race reported in the project's repository [16].

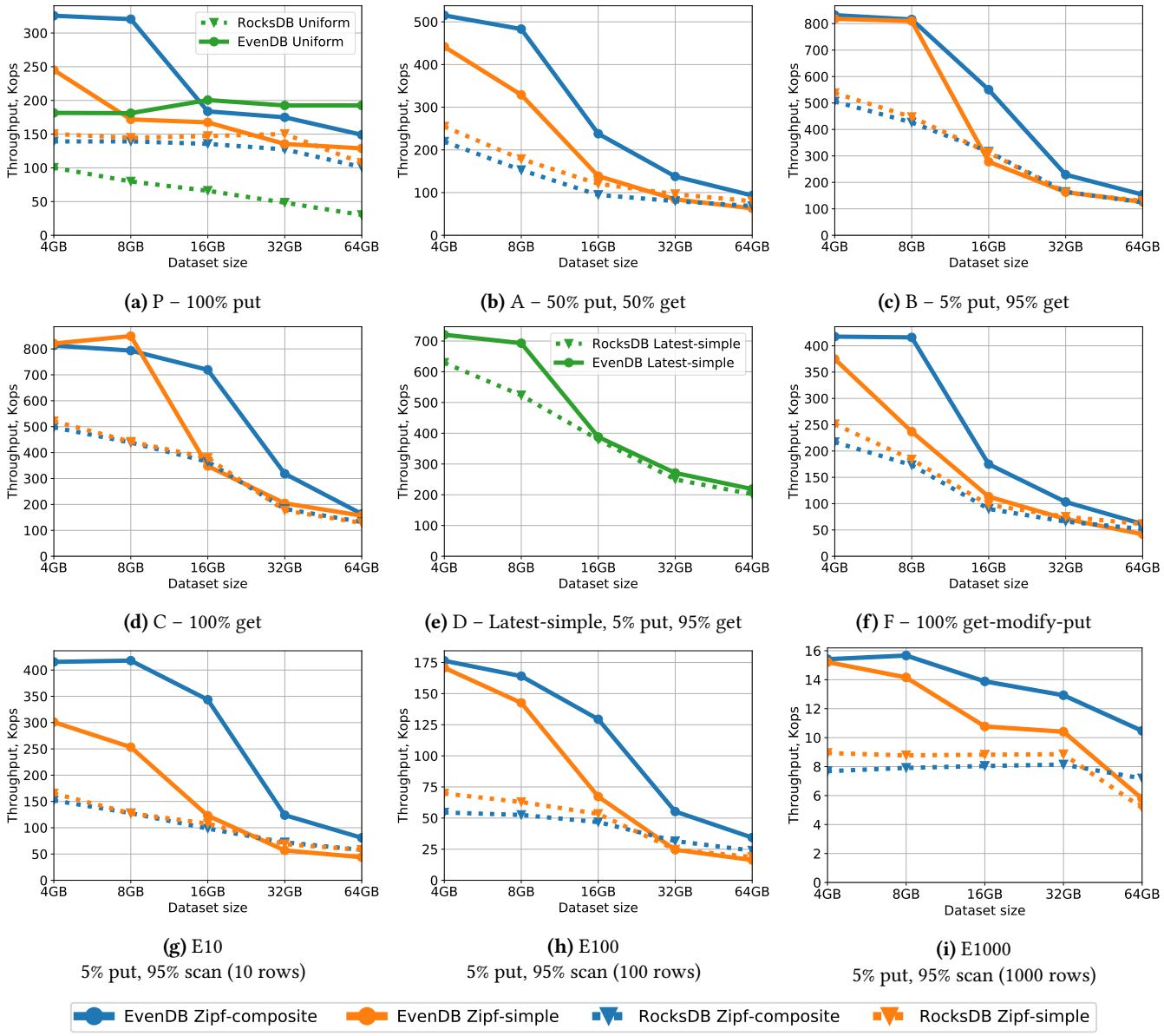
### 5.2 EvenDB versus RocksDB

Figure 4 presents throughput measurements of EvenDB and RocksDB in all YCSB workloads. Except workload D, which exercises the Latest-simple access pattern (depicted in green), 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 todo: change). 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)*** is tested in workload P (100% put, Figure 4a). EvenDB's throughput is 1.8x to 6.4x that of RocksDB's with uniform keys, 1.3x to 2.3x with Zipf-composite keys, and 0.9x to 1.6x 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




**(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 4.** EvenDB vs RocksDB throughput (Kops), under YCSB workloads with various key distributions.

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. Note that EvenDB's recovery time is not impacted by long funk logs since they are not replayed upon recovery.

Under the Zipf-simple workload, the gains are moderate due to the low spatial locality. They are most pronounced for small datasets that fit into RAM.

The Uniform workload exhibits no locality of any kind. EvenDB benefits from this because the keys get dispersed evenly across all 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 5. 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).

***Mixed put-get*** is represented in workloads A (50% put, 50% get, Figure 4b) and F (100% get-modify-put, Figure 4f). Note that the latter exercises the usual get and put API (i.e., does not provide atomicity). In this scenario, EvenDB works

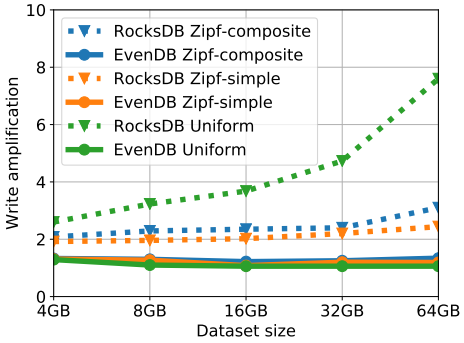
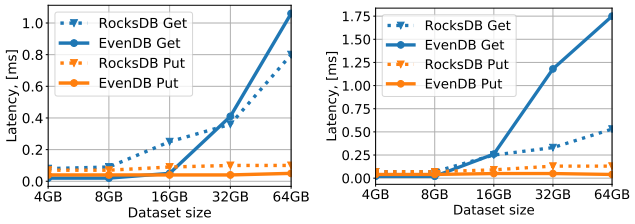
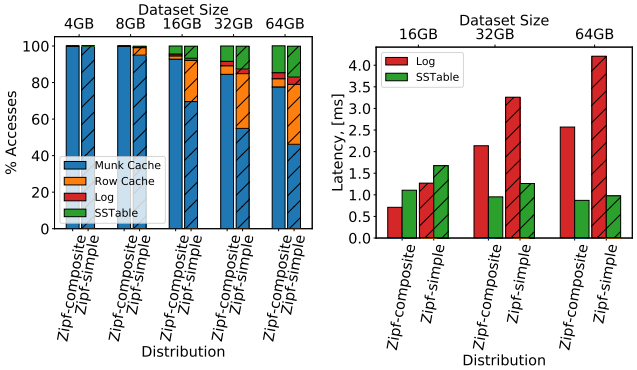



**Figure 5.** EvenDB vs RocksDB write amplification under the put-only workload P.




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

**Figure 6.** EvenDB vs RocksDB 95% latency (ms), under a mixed get-put workload A.




**(a)** Fraction of get accesses **(b)** On-disk get access latency

**Figure 7.** EvenDB get latency breakdown by serving component, under a mixed get-put workload A.

well with composite keys, e.g., in workload A it achieves 1.4x to 3.5x the throughput of RocksDB due to better exploitation of spatial locality. With simple keys, on the other hand, the get-put mix is particularly challenging for EvenDB, which 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. Note that EvenDB is still reasonably close to RocksDB in the worst case (0.75x and 0.7x throughput for the 64GB dataset in A and F, respectively).

Figure 6, 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 6a). With simple keys (Figure 6b), 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 7. Figure 7a classifies gets by the storage component that fulfills the request, and Figure 7b 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 4.0% 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 the funks are cached less effectively by the OS, and the disk becomes a bottleneck due to the higher I/O rate.

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.

***Get-dominated*** workloads (B–D, Figures 4c–4e) are favorable to EvenDB, which has a marked advantage in all key distributions with small datasets (up to the available RAM size) and also with Zipf-composite keys in large datasets. For example, in workload C (100% get, Figure 4d), EvenDB performs 1.2x to 2x better than RocksDB with composite keys, and up to 1.9x 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. As we discuss in §5.3 below, RocksDB's performance in this scenario can be improved by using a larger application-level block cache, but this hurts performance in other benchmarks.

***Scan-dominated*** benchmarks (E10–1000, 5% put, 95% scan, Figures 4g- 4i) iterate through a number of items sampled uniformly in the range [1,S], where S is 10, 100, or 1000. This workload (except with short scans on large datasets) is favorable to EvenDB, since it exhibits the spatial locality the system has been designed for. Under Zipf-composite, EvenDB achieves 1.4x to 3.2x throughput vs RocksDB. With simple keys, EvenDB improves over RocksDB when scans are long or the dataset is small. In §5.3 we show that EvenDB's scan performance on big datasets can be improved by adapting the funk log size limit to this workload.

### 5.3 Insights

***Vertical scalability.*** Figure 8 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 (7.7x for composite keys, 7.8x

for simple ones). The other workloads scale slower, due to read-write and write-write contention as well as background munk and funk rebalances.

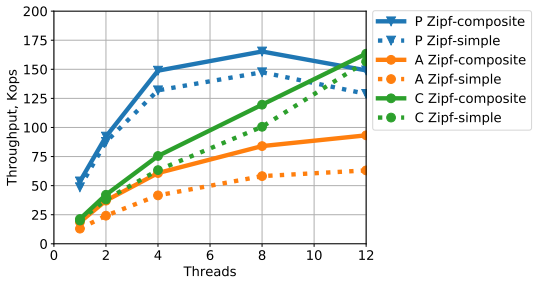
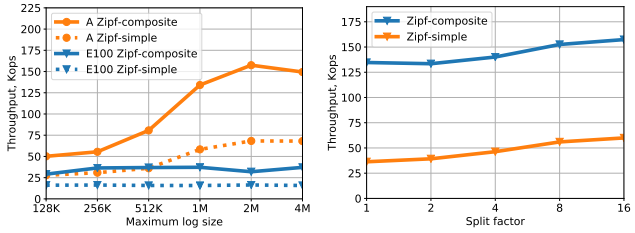



**Figure 8.** EvenDB scalability with the number of threads for the 64GB dataset and different workloads.




**(a)** Maximum log size, workloads A and E100

**(b)** Bloom filter split factor, workload A

**Figure 9.** 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 9a 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 9b 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; Idit: the following was not shown: beyond this point the benefit levels off. The impact of Bloom filter partitioning on EvenDB's memory footprint is negligible.

| P | A | B | C | D | E10–1000 | F |
|---|---|---|---|---|---|---|
| 1.5–2.9x | TBD | TBD | 2.1–3.2x | TBD | 2.2–4.5x | TBD |

**Table 2.** EvenDB throughput improvement over PebblesDB, on a 32GB dataset with Zipf-simple keys, with 1 to 8 threads.

***RocksDB memory tuning.*** In RocksDB's out-of-the-box default configuration, the block cache is 8MB. We next experiment 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 1.3x and 1.6x 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.

Eran please add a graph

**Figure 10.** RocksDB's speedup with larger block caches than its default configuration. Results below 1 indicate slowdown.

However, for bigger datasets (32GB and 64GB), using a bigger block cache degrades RocksDB's performance. Figure 10 shows RocksDB's speedup with different block cache sizes for the 64GB data set in the various workloads. We see that the default configuration gives the best results for most of the workload suite (i.e., the speedups are mostly below 1). We therefore used this configuration in §5.2 above.

### 5.4 EvenDB versus PebblesDB

We compare EvenDB to PebblesDB, which was shown to significantly improve over RocksDB [43], mostly in single-thread experiments, before RocksDB's recent version was released. We compare EvenDB to PebbelesDB 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, and 8 threads. The results are summarized in Table 2. In all experiments, EvenDB is consistently better than PebblesDB, with its performance improvements ranging from 1.5x with a single thread on P, to 4.5x with eight threads on E10. In all benchmarks, EvenDB's advantage grows with the level of parallelism. We observed a similar trend with smaller datasets. Idit: can we say more?

We note that in our experiments, RocksDB also consistently outperforms PebblesDB. The gap with the results reported in [43] can be attributed, e.g., to RocksDB's evolution, resource constraints (running within a container), different hardware, and increased benchmark parallelism.

## 6 Related Work

The vast majority of industrial mainstream NoSQL KV-stores are implemented as LSM trees [8, 10, 15, 26, 36], building on the foundations set by O'Neil et al. [39, 40].

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 [25, 47] and research prototypes [18, 34, 43, 44]. Other suggestions include storage optimizations [34, 38, 42–44], boost of in-memory parallelism [15, 31], and leveraging workload redundancies to defer disk flushes [18, 23].

A number of systems focus on reducing write amplification. PebblesDB [43] introduces fragmented LSM trees in which level files are sliced into *guards* of increasing granularity and organized in a skiplist-like layout. In contrast, EvenDB eliminates the concept of levels altogether, and employs a flat layout. WiscKey [38] separates key and value storage in SSTables, also in order to reduce amplification. This optimization is orthogonal to EvenDB's concepts, and could benefit our work as well.

VT-Trees [44] are LSM-trees that apply stitching to avoid rewriting already sorted data. This improves performance and significantly reduces write amplification in some scenarios (e.g., time-series ingestion).

SLM-DB [34] is an LSM design that relies on persistent memory for its mutable component and thus avoids WAL maintenance. It also re-designs the traditional LSM read path, utilizing a $B^+$-tree index in persistent memory on top of a flat list of SSTables with LSM-like temporal data partitioning. 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 [23] 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.

EvenDB's range-based data partitioning resembles classic B-trees [35], which suffer from a write bottleneck in random updates to leaf blocks (similar to chunks). EvenDB overcomes this limitation through (1) transforming random I/O to sequential I/O at the chunk level, (2) managing a write-through chunk cache in memory (munks), and (3) reducing I/O through in-memory (munk) compaction.

A $B^\epsilon$-tree [24]is a B-tree variant that uses overflow write buffers in internal nodes. This design speeds up writes and reduces write amplification, however lookups are slowed down by having to search in unordered buffers. $B^\epsilon$-trees have been used in KV-stores (TokuDB [14]) and filesystems (BetrFS [33]). §3 compares $B^\epsilon$-tree concepts to EvenDB.

Tucana [41] is an in-memory $B^\epsilon$-tree index over a persistent log of KV-pairs. It applies multiple system optimizations to speed up I/O (all orthogonal to our work): block-device storage access, memory-mapped files, and copy-on-write on internal nodes. However, Tucana provids neither strong scan semantics nor consistent recovery, and does not support concurrent put operations.

A different line of work focuses on in-memory KV-stores [3, 5, 13, 46] providing fast volatile data storage, 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. We are unaware of consistency guarantees or performance optimizations with regards to disk-resident data in such systems.

## 7 Conclusions

We presented EvenDB – a novel persistent KV-store optimized for workloads with high spatial locality, as prevalent in modern data-driven 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. EvenDB is presented as a conceptual prototype, which can be improved in multiple ways through ideas borrowed from other designs.

## References


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

[2] Flurry analytics. https://developer.yahoo.com/flurry/docs/analytics/.

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

[4] 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.

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

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

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

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

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

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

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

[12] Average selling price of DRAM 1Gb equivalent units from 2009 to 2017 (in U.S. dollars). https://www.statista.com/statistics/298821/dram-average-unit-price/, 2018.

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

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

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

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

[17] 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.

[18] 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.

[19] 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.

[20] Bayer, R., and McCreight, E. M. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)* (1970), pp. 107–141.

[21] Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. An introduction to b$\epsilon$-trees and write-optimization. *;login: 40*, 5 (2015).

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] Comer, D. Ubiquitous b-tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121–137.

[29] 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.

[30] 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.

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

[32] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (1994), SIGMOD '94, pp. 243–252.

[33] Jannen, W., Yuan, J., Zhan, Y., Akshintala, A., Esmet, J., Jiao, Y., Mittal, A., Pandey, P., Reddy, P., Walsh, L., Bender, M., Farach-Colton, M., Johnson, R., Kuszmaul, B. C., and Porter, D. E. BetrFS: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (2015), pp. 301–315.

[34] 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.

[35] 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.

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

[37] Lehman, P. L., and Yao, s. B. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst. 6*, 4 (Dec. 1981), 650–670.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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