PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees

Conference Paper · October 2017

DOI: 10.1145/3132747.3132765

4 authors, including:

Pandian Raju
University of Texas at Austin
3 PUBLICATIONS **26** CITATIONS
SEE PROFILE

Rohan Kadekodi
University of Texas at Austin
2 PUBLICATIONS **27** CITATIONS
SEE PROFILE

Vijay Chidambaram
University of Wisconsin–Madison
24 PUBLICATIONS **290** CITATIONS
SEE PROFILE

PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees

Pandian Raju
University of Texas at Austin

Rohan Kadekodi
University of Texas at Austin

Vijay Chidambaram
University of Texas at Austin and VMware Research

Ittai Abraham
VMware Research

ABSTRACT

Key-value stores such as LevelDB and RocksDB offer excellent write throughput, but suffer high write amplification. The write amplification problem is due to the Log-Structured Merge Trees data structure that underlies these key-value stores. To remedy this problem, this paper presents a novel data structure that is inspired by Skip Lists, termed Fragmented Log-Structured Merge Trees (FLSM). FLSM introduces the notion of guards to organize logs, and avoids rewriting data in the same level. We build PebblesDB, a high-performance key-value store, by modifying HyperLevelDB to use the FLSM data structure. We evaluate PebblesDB using micro-benchmarks and show that for write-intensive workloads, PebblesDB reduces write amplification by 2.4-3× compared to RocksDB, while increasing write throughput by 6.7×. We modify two widely-used NoSQL stores, MongoDB and HyperDex, to use PebblesDB as their underlying storage engine. Evaluating these applications using the YCSB benchmark shows that throughput is increased by 18-105% when using PebblesDB (compared to their default storage engines) while write IO is decreased by 35-55%.

CCS CONCEPTS

• **Information systems** → **Key-value stores**; *Record and block layout*;

KEYWORDS

key-value stores, log-structured merge trees, write-optimized data structures

SOSP'17, Oct 28-31, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10…$15.00
https://doi.org/10.1145/3132747.3132765

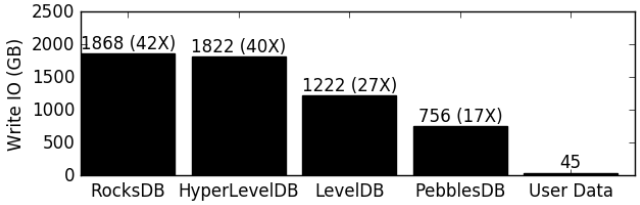


Figure 1: Write Amplification. The figure shows the total write IO (in GB) for different key-value stores when 500 million key-value pairs (totaling 45 GB) are inserted or updated. The write amplification is indicated in parenthesis.

ACM Reference Format:
Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of ACM Symposium on Operating Systems Principles (*SOSP'17*)*. ACM, New York, NY, USA, Article 4, 18 pages. https://doi.org/10.1145/3132747.3132765

1 INTRODUCTION

Key-value stores have become a fundamental part of the infrastructure for modern systems. Much like how file systems are an integral part of operating systems, distributed systems today depend on key-value stores for storage. For example, key-value stores are used to store state in graph databases [21, 31], task queues [5, 54], stream processing engines [7, 50], application data caching [35, 43], event tracking systems [46], NoSQL stores [18, 40], and distributed databases [30]. Improving the performance of key-value stores has the potential to impact a large number of widely-used data intensive services.

Great progress has been made in improving different aspects of key-value stores such as memory efficiency [9, 17, 34, 42, 58] and energy efficiency [6]. One fundamental problem that remains is the high write amplification of key-value stores for write-intensive workloads. Write amplification is the ratio of total write IO performed by the store to the total user data. High write amplification increases the load on storage devices such as SSDs, which have limited write cycles before the bit error rate becomes unacceptable [3, 26, 39]. With

the increasing size of user data sets (*e.g,.* Pinterest's stateful systems process tens of petabytes of data every day [46]), high write amplification results in frequent device wear out and high storage costs [41]. Write amplification also reduces write throughput: in the RocksDB [20] key-value store, it results in write throughput being reduced to 10% of read throughput [53]. Thus, reducing write amplification will both lower storage costs and increase write throughput.

Figure 1 shows the high write amplification (ratio of total IO to total user data written) that occurs in several widely-used key-value stores when 500 million key-value pairs are inserted or updated in random order. Techniques from prior research tackling write amplification have not been widely adopted since they either require specialized hardware [38, 55] or sacrifice other aspects such as search (range query) performance [57]. Conventional wisdom is that reducing write amplification requires sacrificing either write or read throughput [34]. In today's low-latency, write-intensive environments [27], users are not willing to sacrifice either.

Key-value stores such as LevelDB [25] and RocksDB are built on top of the log-structured merge trees [44] (LSM) data structure, and their high write amplification can be traced back to the data structure itself (§2). LSM stores maintain data in sorted order on storage, enabling efficient querying of data. However, when new data is inserted into an LSM-store, existing data is rewritten to maintain the sorted order, resulting in large amounts of write IO.

This paper presents a novel data structure, the *Fragmented Log-Structured Merge Trees* (FLSM), which combines ideas from the Skip List [47, 48] and Log-Structured Merge trees data structures along with a novel compaction algorithm. FLSM strikes at the root of write amplification by drastically reducing (and in many cases, eliminating) data rewrites, instead *fragmenting* data into smaller chunks that are organized using *guards* on storage (§3). Guards allow FLSM to find keys efficiently. Write operations on LSM stores are often stalled or blocked while data is compacted (rewritten for better read performance); by drastically reducing write IO, FLSM makes compaction significantly faster, thereby increasing write throughput.

Building a high-performance key-value store on top of the FLSM data structure is not without challenges; the design of FLSM trades read performance for write throughput. This paper presents PebblesDb, a modification of the HyperLevelDB [29] key-value store that achieves the trifecta of low write amplification, high write throughput, and high read throughput. PebblesDb employs a collection of techniques such as parallel seeks, aggressive seek-based compaction, and sstable-level bloom filters to reduce the overheads inherent to the FLSM data structure (§4). Although many of the techniques PebblesDb employs are well-known, together with the FLSM data structure, they allow PebblesDb to achieve excellent performance on both read-dominated and write-dominated workloads.

PebblesDb outperforms mature, carefully engineered key-value stores such as RocksDB and LevelDB on several workloads (§5) . On the `db_bench` micro-benchmarks, PebblesDb obtains 6.7× the write throughput of RocksDB and 27% higher read throughput, while doing 2.4-3× less write IO. When the NoSQL store MongoDB [40] is configured to use PebblesDb instead of RocksDB as its storage engine, MongoDB obtains the same overall performance on the YCSB benchmark [16] while doing 37% less IO (§5).

While the FLSM data structure is useful in many scenarios, it is not without its limitations. On a fully compacted key-value store, PebblesDb incurs a 30% overhead for small range queries. While the overhead drops to 11% for large range queries, the FLSM data structure is not the best fit for workloads which involve a lot of range queries after an initial burst of writes. Note that PebblesDb does not incur an overhead if the range queries are interspersed with writes.

In summary, the paper makes the following contributions:

- The design of the novel Fragmented Log-Structured Merge Trees data structure, which combines ideas from skip lists and log-structured merge trees (§3).
- The design and implementation of PebblesDb, a key-value store built using fragmented log-structured merge trees and several important optimizations (§4). We have made PebblesDb publicly available at `https://github.com/utsaslab/pebblesdb`.
- Experimental results demonstrating that PebblesDb dominates LSM-based stores such as RocksDB in many workloads, showing that it is possible to achieve low write amplification, high write throughput, and high read throughput simultaneously (§5).

2 BACKGROUND

This section provides some background on key-values stores and log-structured merge trees. It first describes common operations on key-values stores (§2.1) and discusses why log-structured merge trees are used to implement key-value stores in write-intensive environments (§2.2). It shows that the log-structured merge tree data structure fundamentally leads to large write amplification.

2.1 Key-Value Store Operations

Get. The `get(key)` operation returns the latest value associated with `key`.

Put. The `put(key, value)` operation stores the mapping from `key` to `value` in the store. If `key` was already present in the store, its associated value is updated.

Iterators. Some key-value stores such as LevelDB provide an iterator over the entire key-value store. `it.seek(key)` positions the iterator `it` at the smallest key ≥ `key`. The `it.next()` call moves `it` to the next key in sequence. The `it.value()` call returns the value associated with the key at the current iterator position. Most key-value stores allow the user to provide a function for ordering keys.

Range Query. The `range_query(key1, key2)` operation returns all key-value pairs falling within the given range. Range queries are often implemented by doing a `seek()` to `key1` and doing `next()` calls until the iterator passes `key2`.

2.2 Log-Structured Merge Trees

Embedded databases such as KyotoCabinet [32] and BerkeleyDB [45] are typically implemented using B+ Trees [14]. However, B+ Trees are a poor fit for write-intensive workloads since updating the tree requires multiple random writes (10-100× slower than sequential writes). Inserting 100 million key-value pairs into KyotoCabinet writes 829 GB to storage (61× write amplification). Due to the low write throughput and high write amplification of B+ Trees, developers turned to other data structures for write-intensive workloads.

The log-structured merge trees (LSM) data structure [44] takes advantage of high sequential bandwidth by *only* writing sequentially to storage. Writes are batched together in memory and written to storage as a sequential log (termed an *sstable*). Each sstable contains a sorted sequence of keys.

Sstables on storage are organized as hierarchy of *levels*. Each level contains multiple sstables, and has a maximum size for its sstables. In a 5-level LSM, Level 0 is the *lowest* level and Level 5 is the *highest* level. The amount of data (and the number of sstables) in each level increases as the levels get higher. The last level in an LSM may contain hundreds of gigabytes. Application data usually flows into the lower levels and is then compacted into the higher levels. The lower levels are usually cached in memory.

LSM maintains the following invariant at each level: all sstables contain disjoint sets of keys. For example, a level might contain three sstables: {1..6}[1], {8..12}, and {100..105}. Each key will be present in exactly one sstable on a given level. As a result, locating a key requires only two binary searches: one binary search on the starting keys of sstables (maintained separately) to locate the correct sstable and another binary search inside the sstable to find the key. If the search fails, the key is not present in that level.

LSM Operations. The `get()` operation returns the latest value of the key. Since the most recent data will be in lower levels, the key-value store searches for the key level by level, starting from Level 0; if it finds the key, it returns the value. Each key has a sequence number that indicates its version. Finding the key at each level requires reading and searching exactly one sstable.

The `seek()` and `next()` operations require positioning an iterator over the entire key-value store. This is implemented using multiple iterators (one per level); each iterator is first positioned inside the appropriate sstable in each level, and the iterator results are merged. The `seek()` operation requires finding the appropriate sstables on each level, and positioning the sstable iterators. The results of the sstable iterators are merged (by identifying the smallest key) to position the key-value store iterator. The `next()` operation simply advances the correct sstable iterator, merges the iterators again, and re-positions the key-value store iterator.

The `put()` operation writes the key-value pair, along with a monotonically increasing sequence number, to an in-memory skip list [48] called the *memtable*. When the memtable reaches a certain size, it is written to storage as a sstable at Level 0. When each level contains a threshold number of files, it is compacted into the next level. Assume Level 0 contains {2, 3} and {10, 12} sstables. If Level 1 contains {1,4} and {9, 13} sstables, then during compaction, Level 1 sstables are rewritten as {1, 2, 3, 4} and {9, 10, 12, 13}, merging the sstables from Level 0 and Level 1. Compacting sstables reduces the total number of sstables in the key-value store and pushes colder data into higher levels. The lower levels are usually cached in memory, thus leading to faster reads of recent data.

Updating or deleting keys in LSM-based stores does not update the key in place, since all write IO is sequential. Instead, the key is inserted once again into the database with a higher sequence number; a delete key is inserted again with a special flag (often called a *tombstone* flag). Due to the higher sequence number, the latest version of the flag will be returned by the store to the user.

Write Amplification: Root Cause. Figure 2 illustrates compaction in a LSM key-value store. The key-value store contains two sstables in Level 1 initially. Let us assume that Level 0 is configured to hold only one sstable at a time; when this limit is reached, compaction is triggered. At time t_1, one sstable is added, and a compaction is triggered is at t_2. Similarly, sstables are added at t_3 and t_5 and compactions are triggered at t_4 and t_6. When compacting a sstable, all sstables in the next level whose key ranges *intersect* with the sstable being compacted are rewritten. In this example, since the key ranges of all Level 0 sstables intersect with key ranges of all Level 1 sstables, the Level 1 sstables are rewritten *every* time a Level 0 sstable is compacted. In this worst-case example, Level 1 sstables are rewritten **three** times while compacting a *single* upper level. Thus, the high write amplification of

[1] Let {x..y} indicate a sstable with keys ranging from x to y

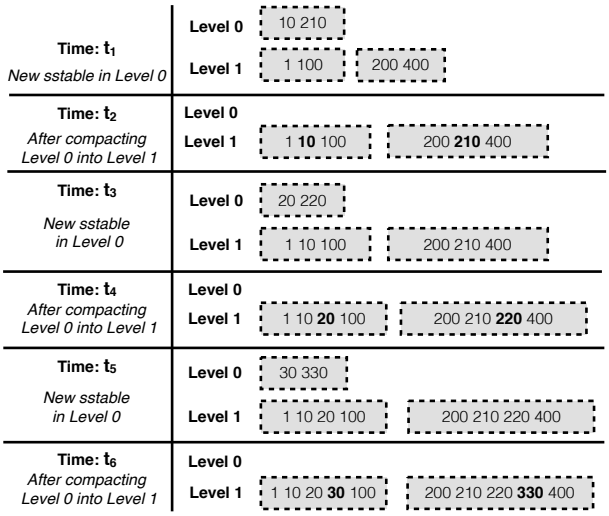


Figure 2: LSM Compaction. The figure shows sstables being inserted and compacted over time in a LSM.

LSM key-value stores can be traced to **multiple rewrites** of sstables during compaction.

The Challenge. A naive way to reduce write amplification in LSM is to simply not merge sstables during compaction but add new sstables to each level [19, 22]. However, read and range query performance will drop significantly due to two reasons. First, without merge, the key-value store will end up containing large number of sstables. Second, as multiple sstables can now contain the same key and can have overlapping key ranges in the same level, read operations will have to examine multiple sstables (since binary search to find the sstable is not possible), leading to large overhead.

3 FRAGMENTED LOG-STRUCTURED MERGE TREES

The challenge is to achieve three goals *simultaneously*: low write amplification, high write throughput, and good read performance. This section presents our novel data structure, Fragmented Log-structured Merge Trees (FLSM), and describes how it tackles this challenge.

FLSM can be seen as a blend of an LSM data structure with a Skip List along with a novel compaction algorithm that overall reduces write amplification and increases write throughput. The fundamental problem with log-structured merge trees is that sstables are typically re-written multiple times as new data is compacted into them. FLSM counters this by *fragmenting* sstables into smaller units. Instead of rewriting the sstable, FLSM's compaction simply appends a new sstable fragment to the next level. Doing so ensures that data is written *exactly once* in most levels; a different compaction algorithm is used for the the last few highest levels. FLSM achieves this using a novel storage layout and organizing data using *guards* (§3.1). This section describes how guards are selected (§3.2), how guards are inserted and deleted (§3.3), how FLSM operations are performed (§3.4), how FLSM can be tuned for different performance/write-IO trade-offs (§3.5), and its limitations (§3.6).

3.1 Guards

In the classical LSM, each level contains sstables with disjoint key ranges (*i.e.,* each key will be present in exactly one sstable). The chief insight in this work is that maintaining this invariant is the root cause of write amplification, as it forces data to be rewritten in the same level. The FLSM data structure discards this invariant: each level can contain multiple sstables with overlapping key ranges, so that a key may be present in multiple sstables. To quickly find keys in each level, FLSM organizes the sstables into guards (inspired from the Skip-List data structure [47, 48]).

Each level contains multiple guards. Guards divide the key space (for that level) into disjoint units. Each guard G_i has an associated key K_i, chosen from among keys inserted into the FLSM. Each level in the FLSM contains more guards than the level above it; the guards get progressively more fine-grained as the data gets pushed deeper and deeper into the FLSM. As in a skip-list, if a key is a guard at a given level i, it will be a guard for all levels $> i$.

Each guard has a set of associated sstables. Each sstable is sorted. If guard G_i is associated with key K_i and guard G_{i+1} with K_{i+1}, an sstable with keys in the range $[K_i, K_{i+1})$ will be attached to G_i. Sstables with keys smaller than the first guard are stored in a special *sentinel* guard in each level. The last guard G_n in the level stores all sstables with keys $\geq K_n$. Guards within a level never have overlapping key ranges. Thus, to find a key in a given level, only one guard will have to be examined.

In FLSM compaction, the sstables of a given guard are (merge) sorted and then *fragmented* (partitioned), so that each child guard receives a new sstable that fits into the key range of that child guard in the next level.

Example. Figure 3 shows the state of the FLSM data structure after a few `put()` operations. We make several observations based on the figure:

- A `put()` results in keys being added to the in-memory memtable (not shown). Eventually, the memtable becomes full, and is written as an sstable to Level 0. Level 0 does not have guards, and collects together recently written sstables.
- The number of guards increases as the level number increases. The number of guards in each level does not necessarily increase exponentially.

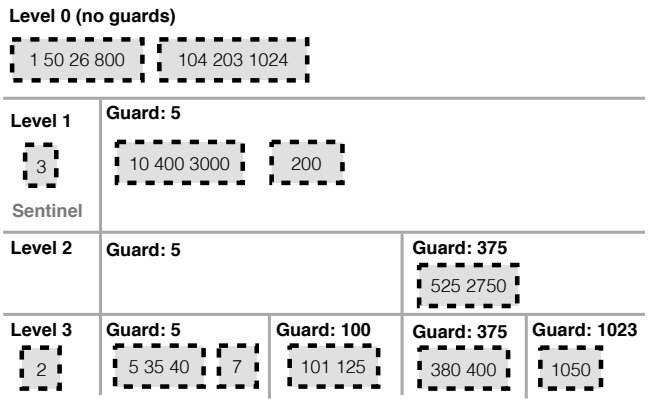


Figure 3: FLSM Layout on Storage. The figure illustrates FLSM's guards across different levels. Each box with dotted outline is an sstable, and the numbers represent keys.

- Each level has a sentinel guard that is responsible for sstables with keys < than the first guard. In Figure 3, sstables with keys < 5 are attached to the sentinel guard.
- Data inside an FLSM level is *partially sorted*: guards do not have overlapping key ranges, but the sstables attached to each guard can have overlapping key ranges.

3.2 Selecting Guards

FLSM performance is significantly impacted by how guards are selected. In the worst case, if one guard contains all sstables, reading and searching such a large guard (and all its constituent sstables) would cause an un-acceptable increase in latency for reads and range queries. For this reason, guards are not selected statically; guards are selected probabilistically from inserted keys, preventing skew.

Guard Probability. When a key is inserted into FLSM, *guard probability* determines if it becomes a guard. Guard probability `gp(key,i)` is the probability that `key` becomes a guard at level `i`. For example, if the guard probability is 1/10, one in every 10 inserted keys will be randomly selected to be a guard. The guard probability is designed to be lowest at Level 1 (which has the fewest guards), and it increases with the level number (as higher levels have more guards). Selecting guards in this manner distributes guards across the inserted keys in a smooth fashion that is likely to prevent skew.

Much like skip lists, if a key K is selected as a guard in level i, it becomes a guard for all higher levels $i+1, i+2$ etc. The guards in level $i+1$ are a strict superset of the guards in level i. Choosing guards in this manner allows the interval between each guard to be successively refined in each deeper level. For example, in Figure 3, key 5 is chosen as a guard for Level 1; therefore it is also a guard for levels 2 and 3.

FLSM selects guards out of inserted keys for simplicity; FLSM does not require that guards correspond to keys present in the key-value store.

Other schemes for selecting guards. The advantage of the current method for selecting guards is that it is simple, cheap to compute, and fairly distributes guards over inserted keys. However, it does not take into account the amount of IO that will result from partitioning sstables during compaction (this section will describe how compaction works shortly). FLSM could potentially select new guards for each level at compaction time such that sstable partitions are minimized; however, this could introduce skew. We leave exploring alternative selection schemes for future work.

3.3 Inserting and Deleting Guards

Guards are not inserted into FLSM synchronously when they are selected. Inserting a guard may require splitting an sstable or moving an sstable. If a guard is inserted on multiple levels, work is generated on all those levels. For this reason, guards are inserted asynchronously into FLSM.

When guards are selected, they are added to an in-memory set termed the *uncommitted* guards. Sstables are not partitioned on storage based on (as of yet) uncommitted guards; as a result, FLSM reads are performed as if these guards did not exist. At the next compaction cycle, sstables are partitioned and compacted based on both old guards and uncommitted guards; any sstable that needs to be split due to an uncommitted guard is compacted to the next level. At the end of compaction, the uncommitted guards are persisted on storage and added to the full set of guards. Future reads will be performed based on the full set of guards.

We note that in many of the workloads that were tested, guard deletion was not required. A guard could become empty if all its keys are deleted, but empty guards do not cause noticeable performance degradation as `get()` and range query operations skip over empty guards. Nevertheless, deleting guards is useful in two scenarios: when the guard is empty or when data in the level is spread unevenly among guards. In the second case, consolidating data among fewer guards can improve performance.

Guard deletion is also performed asynchronously similar to guard insertion. Deleted guards are added to an in-memory set. At the next compaction cycle, sstables are re-arranged to account for the deleted guards. Deleting a guard G at level i is done lazily at compaction time. During compaction, guard G is deleted and sstables belonging to guard G will be partitioned and appended to either the neighboring guards in the same level i or child guards in level $i+1$. Compaction from level i to $i+1$ proceeds as normal (since G is still a

guard in level $i+1$). At the end of compaction, FLSM persists metadata indicating G has been deleted at level i. If required, the guard is deleted in other levels in a similar manner. Note that if a guard is deleted at level i, it should be deleted at all levels $< i$; FLSM can choose whether to delete the guard at higher levels $> i$.

3.4 FLSM Operations

Get Operations. A `get()` operation first checks the in-memory memtable. If the key is not found, the search continues level by level, starting with level 0. During the search, if the key is found, it is returned immediately; this is safe since updated keys will be in lower levels that are searched first. To check if a key is present in a given level, binary search is used to find the single guard that could contain the key. Once the guard is located, its sstables are searched for the key. Thus, in the worst case, a `get()` requires reading one guard from each level, and all the sstables of each guard.

Range Queries. Range queries require collecting all the keys in the given range. FLSM first identifies the guards at each level that intersect with the given range. Inside each guard, there may be multiple sstables that intersect with the given range; a binary search is performed on each sstable to identify the smallest key overall in the range. Identifying the next smallest key in the range is similar to the merge procedure in merge sort; however, a full sort does not need to be performed. When the end of range query interval is reached, the operation is complete, and the result is returned to the user. Key-value stores such as LevelDB provide related operations such as `seek()` and `next()`; a `seek(key)` positions an iterator at the smallest key larger than or equal to `key`, while `next()` advances the iterator. In LSM stores, the database iterator is implemented via merging level iterators; in FLSM, the level iterators are themselves implemented by merging iterators on the sstables inside the guard of interest.

Put Operations. A `put()` operation adds data to an in-memory memtable. When the memtable gets full, it is written as a sorted sstable to Level 0. When each level reaches a certain size, it is compacted into the next level. In contrast to compaction in LSM stores, FLSM avoids sstable rewrites in most cases by partitioning sstables and attaching them to guards in the next level.

Key Updates and Deletions. Similar to LSM, updating or deleting a key involves inserting the key into the store with an updated sequence number or a deletion flag respectively. Reads and range queries will ignore keys with deletion flags. If the insertion of a key resulted in a guard being formed, the deletion of the key does not result in deletion of the related guard; deleting a guard will involve a significant amount of compaction work. Thus, empty guards are possible.

Compaction. When a guard accumulates a threshold number of sstables, it is compacted into the next level. The sstables in the guard are first (merge) sorted and then partitioned into new sstables based on the guards of the next level; the new sstables are then attached to the correct guards. For example, assume a guard at Level 1 contains keys $\{1, 20, 45, 101, 245\}$. If the next level has guards 1, 40, and 200, the sstable will be partitioned into three sstables containing $\{1, 20\}$, $\{45, 101\}$, and $\{245\}$ and attached to guards 1, 40, and 200 respectively.

Note that in most cases, FLSM compaction does not rewrite sstables. This is the main insight behind how FLSM reduces write amplification. New sstables are simply added to the correct guard in the next level. There are two exceptions to the no-rewrite rule. First, at the highest level (*e.g.,*. Level 5) of FLSM, the sstables have to be rewritten during compaction; there is no higher level for the sstables to be partitioned and attached to. Second, for the second-highest level (*e.g.,*. Level 4), FLSM will rewrite an sstable into the same level if the alternative is to merge into a large sstable in the highest level (since we cannot attach new sstables in the last level if the guard is full). The exact heuristic is rewrite in second-highest-level if merge causes 25× more IO.

FLSM compaction is trivially parallelizable because compacting a guard only involves its descendants in the next level; the way guards are chosen in FLSM guarantees that compacting one guard never interferes with compacting another guard in the same level. For example, in Figure 3 if guard 375 in Level 2 is split into guards 375 and 1023 in Level 3, only these three guards are affected. Compacting guard 5 (if it had data) will not affect the on-going compaction of guard 375 in any way. Thus, the compaction process can be carried out in parallel for different guard files at the same time. Parallel IO from compaction can be efficiently handled by devices such as flash SSDs that offer high random write throughput with multiple flash channels. Such parallel compaction can reduce the total time taken to compact significantly. A compacted key-value store has lower latency for reads; since parallel compaction gets the store to this state faster, it also increases read throughput.

3.5 Tuning FLSM

FLSM performance for reads and range queries depends upon a single parameter: the number of sstables inside each guard. If guards contain a large number of sstables, read and range query latencies become high. Therefore, FLSM provide users a knob to tune behavior, `max_sstables_per_guard`, which caps the maximum number of sstables present inside each guard in FLSM. When any guard accumulates

`max_sstables_per_guard` number of sstables, the guard is compacted into the next level.

Tuning `max_sstables_per_guard` allows the user to trade-off more write IO (due to more compaction) for lower read and range query latencies. Interestingly, if this parameter is set to one, FLSM behaves like LSM and obtains similar read and write performance. Thus, FLSM can be viewed as a *generalization* of the LSM data structure.

3.6 Limitations

The FLSM data structure significantly reduces write amplification and has faster compaction (as compaction in FLSM requires lower read and write IO). By virtue of faster compaction, write throughput increases as well. However, the FLSM data structure is not without limitations.

Since `get()` and range query operations need to examine all sstables within a guard, the latency of these operations is increased in comparison to LSM. Section 4 describes how this limitation can be overcome; using a combination of well-known techniques can reduce or eliminate the overheads introduced by the FLSM data structure, resulting in a key-value store that achieves the trifecta of low write amplification, high write throughput, and high read throughput.

3.7 Asymptotic Analysis

This section provides an analysis of FLSM operations using a theoretical model.

Model. We use the standard Disk Access Model (DAM) [2] and assume that each read/write operation can access a block of size B in one unit cost. To simplify the model, we will assume a total of n data items are stored.

FLSM Analysis. Consider a FLSM where the guard probability is $1/B$ (so the number of guards in level $i+1$ is in expectation B times more than the number of guards in level i). Since the expected fan-out of FLSM is B, with high probability, an FLSM with n data items will have $H = \log_B n$ levels. It is easy to see that each data item is written just once per level (it is appended once and never re-written to the same level), resulting in a write cost of $O(H) = O(\log_B n)$. Since in the DAM model, FLSM writes a block of B items at unit cost, the total amortized cost of any put operation is $O(H/B) = O((\log_B n)/B)$ over its entire compaction lifetime. However, FLSM compaction in the last level does re-write data. Since this last level re-write will occur with high probability $O(B)$ times then the final total amortized cost of any put operation is $O((B + \log_B n)/B)$.

The guards in FLSM induce a degree B Skip List. A detailed theoretical analysis of the B-Skip List data structure shows that with high probability each guard will have $O(B)$ children, each guard will have at most $O(B)$ sstables, and each sstable will have at most $O(B)$ data items [1, 12, 24]. Naively, searching for an item would require finding the right guard at each level (via binary search), and then searching inside all sstables inside the guard. Since the last level has the most guards (B^H), binary search cost would be dominated by the cost for the last level: $O(\log_2 B^H) = O(H \log_2 B) = O(\log_B n * \log_2 B) = O(\log_2 n)$. Since there are $O(H) = O(log_B n)$ levels to search, this yields a total cost of $O(\log_2 n \log_B n)$ in-memory operations for finding the right guards at each level.

However, in FLSM, the guards and bloom filters are all stored in memory. FLSM performs $O(\log_2 n \log_B n)$ in-memory operations during the binary search for the right guard in each level. Then, for each of the $H = \log_B n$ guards found, FLSM does a bloom filter query on each of the $O(B)$ sstables associated with the guard, with each query costing $O(\log(1/\epsilon))$ in memory operations. In the DAM model all this in-memory work has no cost.

Finally, on average, the bloom filter will indicate only $1 + o(1)$ sstables to be read (with high probability). Reading these sstables will cost $1 + o(1)$ in the DAM model. Therefore, the total read cost of a get operation (assuming sufficient memory to store guards and bloom filters) is just $O(1)$ in the DAM model.

FLSM cannot leverage bloom filters for range queries. The binary search per level is still done in memory. For each level, the binary search outputs one guard and FLSM needs to read all the $O(B)$ associated sstables. So the total cost for a range query returning k elements is $O(B \log_B n + k/B)$.

4 BUILDING PEBBLESDB OVER FLSM

This section presents the design and implementation of PebblesDb, a high-performance key-value store built using fragmented log-structured merge trees. This section describes how PebblesDb offsets FLSM overheads for reads (§4.1) and range queries (§4.2), different PebblesDb operations (§4.3), how PebblesDb recovers from crashes (§4.4), its implementation (§4.5), and its limitations (§4.6).

4.1 Improving Read Performance

Overhead Cause. A `get()` operation in FLSM causes all the sstables of one guard in each level to be examined. In contrast, in log-structured merge trees, exactly one sstable per level needs to be examined. Thus, read operations incur extra overhead in FLSM-based key-value stores.

Sstable Bloom Filters. A Bloom filter is a space-efficient probabilistic data structure used to test whether an element is present in a given set in constant time [13]. A bloom filter can produce false positives, but not false negatives. PebblesDb attaches a bloom filter to each sstable to efficiently detect if a given key could be present in the sstable. The sstable

bloom filters allow PebblesDb to avoid reading unnecessary sstables off storage and greatly reduces the read overhead due to the FLSM data structure.

RocksDB also employs sstable-level bloom filters. Many key-value stores (including RocksDB and LevelDB) employ bloom filters for each block of the sstable. If sstable-level bloom filters are used, block-level filters are not required.

4.2 Improving Range Query Performance

Overhead Cause. Similar to `get()` operations, range queries (implemented using `seek()` and `next()` calls) also require examining all the sstables of a guard for FLSM. Since LSM stores examine only one sstable per level, FLSM stores have significant overhead for range queries.

Seek-Based Compaction. Similar to LevelDB, PebblesDb implements compaction triggered by a threshold number of consecutive `seek()` operations (default: 10). Multiple sstables inside a guard are merged and written to the guards in the next level. The goal is to decrease the average number of sstables within a guard. PebblesDb also aggressively compacts levels: if the size of level i is within a certain threshold ratio (default: 25%) of level $i+1$, level i is compacted into level $i+1$. Such aggressive compaction reduces the number of active levels that need to be searched for a `seek()`. Although such compaction increases write IO, PebblesDb still does significantly lower amount of IO overall (§5).

Parallel Seeks. A unique optimization employed by PebblesDb is using multiple threads to search sstables in parallel for a `seek()`. Each thread reads one sstable off storage and performs a binary search on it. The results of the binary searches are then merged and the iterator is positioned correctly for the `seek()` operation. Due to this optimization, even if a guard contains multiple sstables, FLSM `seek()` latency incurs only a small overhead compared to LSM `seek()` latency.

Parallel seeks must not be carelessly used: if the sstables being examined are cached, the overhead of using multiple threads is *higher* than the benefit obtained from doing parallel seeks. Given that there is no way to know whether a given sstable has been cached or not (since the operating system may drop a cached sstable under memory pressure), PebblesDb employs a simple heuristic: parallel seeks are used only in the last level of the key-value store. The reason to choose this heuristic is that the last level contains the largest amount of data; furthermore, the data in the last level is not recent, and therefore not likely to be cached. This simple heuristic seems to work well in practice.

4.3 PebblesDb Operations

This section briefly describes how various operations are implemented in PebblesDb, and how they differ from doing the same operations on the FLSM data structure. The `put()` operation in PebblesDb is handled similar to puts in FLSM.

Get. PebblesDb handles `get()` operations by locating the appropriate guard in each level (via binary search) and searching the sstables within the guard. PebblesDb `get()` differs from FLSM `get()` in the use of sstable-level bloom filters to avoid reading unnecessary sstables off storage.

Range Query. PebblesDb handles range queries by locating the appropriate guard in each level and placing the iterator at the right position for each sstable in the guard by performing binary searches on the sstables. PebblesDb optimizes this by reading and searching sstables in parallel, and aggressively compacting the levels if a threshold number of consecutive `seek()` requests are received.

Deleting Keys. PebblesDb deletes a key by inserting the key into the store with a flag marking it as deleted. The sequence number of inserted key identifies it as the most recent version of the key, instructing PebblesDb to discard the previous versions of the key for read and range query operations. Note that bloom filters are created over sstables; since sstables are never updated in place, existing bloom filters do not need to be modified during key deletions. Keys marked for deletion are garbage collected during compaction.

4.4 Crash Recovery

By only appending data, and never over-writing any data in place, PebblesDb builds on the same foundation as LSM to provide strong crash-consistency guarantees. PebblesDb builds on the LevelDB codebase, and LevelDB already provides a well-tested crash-recovery mechanism for both data (the sstables) and the metadata (the `MANIFEST` file). PebblesDb simply adds more metadata (guard information) to be persisted in the `MANIFEST` file. PebblesDb sstables use the same format as LevelDB sstables. Crash-recovery tests (testing recovered data after crashing at randomly picked points) confirm that PebblesDb recovers inserted data and associated guard-related metadata correctly after crashes.

4.5 Implementation

PebblesDb is implemented as a variant of the LevelDB family of key-value stores. PebblesDb was built by modifying HyperLevelDB [29], a variant of LevelDB that was engineered to have improved parallelism and better write throughput during compaction. We briefly examined the RocksDB code base, but found that the HyperLevelDB code base was smaller, better documented (as it derives from LevelDB), and easier

to understand. Thus, HyperLevelDB was chosen as the base for PebblesDb.

We added/modified 9100 LOC in C++ to HyperLevelDB. Most of the changes involved introducing guards in HyperLevelDB and modifying compaction. Since guards are built *on top of* sstables, PebblesDb was able to take advantage of the mature, well-tested code that handled sstables. PebblesDb is API-compatible with HyperLevelDB since all changes are internal to the key-value store.

Selecting Guards. Similar to skip lists, PebblesDb picks guards randomly out of the inserted keys. When a key is inserted, a random number is selected to decide if the key is a guard. However, obtaining a random number for every key insertion is computationally expensive; instead, PebblesDb hashes every incoming key, and the last few bits of the hash determine if the key will be a guard (and at which level).

The computationally cheap `MurmurHash` [8] algorithm is used to hash each inserted key. A configurable parameter `top_level_bits` determines how many consecutive Least Significant Bits (LSBs) in the bit representation of the hashed key should be set for the key to be selected as a guard key in Level 1. Another parameter `bit_decrement` determines the number of bits by which the constraint (number of LSBs to be set) is relaxed going each level higher. For example, if `top_level_bits` is set to 17, and `bit_decrement` is set to 2, then a guard key in level 1 should have 17 consecutive LSBs set in its hash value, a guard key in level 2 should have 15 consecutive LSBs set in its hash value and so on. The `top_level_bits` and `bit_decrement` parameters need to be determined empirically; based on our experience, a value of two seems reasonable for `bit_decrement`, but the `top_level_bits` may need to be increased from our default of 27 if the users expect more than 100 million keys to be inserted into PebblesDb. Over-estimating the number of keys in the store is harmless (leads to many empty guards); under-estimating could lead to skewed guards.

Implementing Guards. Each guard stores metadata about the number of sstables it has, the largest and smallest key present across the sstables, and the list of sstables. Each sstable is represented by a unique 64-bit integer. Guards are persisted to storage along with metadata about the sstables in the key-value store. Guards are recovered after a crash from the `MANIFEST` log and the asynchronous write-ahead logs. Recovery of guard data is woven into the key-value store recovery of keys and sstable information. We have not implemented guard deletion in PebblesDb yet since extra guards did not cause significant performance degradation for reads in our experiments and the cost of persisting empty guards is relatively insignificant. We plan to implement guard deletion in the near future.

Multi-threaded Compaction. Similar to RocksDB, PebblesDb uses multiple threads for background compaction. Each thread picks one level and compacts it into the next level. Picking which level to compact is based on the amount of data in each level. When a level is compacted, only guards containing more than a threshold number of sstables are compacted. We have not implemented guard-based parallel compaction in PebblesDb yet; even without parallel compaction, compaction in PebblesDb is much faster than compaction in LSM-based stores such as RocksDB (§5.2).

4.6 Limitations

This section describes three situations where a traditional LSM-based store may be a better choice over PebblesDb.

First, if the workload data will fit entirely in memory, PebblesDb has higher read and range query latency than LSM-based stores. In such a scenario, read or range query requests will not involve storage IO and the computational overhead of locating the correct guard and processing sstables inside a guard will contribute to higher latency. Given the increasing amount of data being generated and processed every day [49], most datasets will *not* fit in memory. For the rare cases where the data size is small, setting `max_sstables_per_guard` to one configures PebblesDb to behave similar to HyperLevelDB, reducing the latency overhead for reads and range queries.

Second, for workloads where data with sequential keys is being inserted into the key-value store, PebblesDb has higher write IO than LSM-based key value stores. If data is inserted sequentially, sstables don't overlap with each other. LSM-based stores handle this case efficiently by simply *moving* an sstable from one level to the next by modifying only the metadata (and without performing write IO); in the case of PebblesDb, the sstable may be partitioned when moving to the next level, leading to write IO. We believe that real-world workloads that insert data sequentially are rare since most workloads are multi-threaded; in such rare cases, we advocate the use of LSM-based stores such as RocksDB.

Third, if the workload involves an initial burst of writes followed by a large number of small range queries, PebblesDb may not be the best fit. For such range queries over a compacted key-value store, PebblesDb experiences a significant overhead (30%) compared to LSM-based stores. However, the overhead drops as the range queries get bigger and entirely disappears if the range queries are interspersed with insertions or updates (as in YCSB Workload E).

5 EVALUATION

This section evaluates the performance of PebblesDb by answering the following questions:

- What is the write amplification of PebblesDb? (§5.2) What is the performance of various PebblesDb key-value store operations? (§5.2) What are the strengths and weaknesses of PebblesDb?
- How does PebblesDb perform on workloads resembling access patterns in various applications? (§5.3)
- How do NoSQL applications perform when they use PebblesDb as their storage engine? (§5.4)
- How much memory and CPU does PebblesDb consume? (§5.5)

5.1 Experimental Setup

Our experiments are run on a Dell Precision Tower 7810 with an Intel Xeon 2.8 GHz processor, 16 GB RAM, and running Ubuntu 16.04 LTS with the Linux 4.4 kernel. The ext4 file system is run on top of a software RAID0 array used over two high-performance Intel 750 SSDs (each 1.2 TB).

All workloads use datasets 3× larger than the main memory on test machine. All reported numbers are the mean of at least five runs. The standard deviation in all cases was less than 5% of the mean. PebblesDb performance is compared with widely-used key-value stores LevelDB, RocksDB and HyperLevelDB. To simplify results, compression is turned off in all stores. We have verified that compression does not change any of our performance results; it simply leads to a smaller dataset. HyperLevelDB does not employ bloom filters for sstables; to make a fair comparison (and to show our results do not derive just from sstable bloom filters), this optimization is added to HyperLevelDB: all numbers presented for HyperLevelDB are with bloom filters for sstables.

Key-Value Store Configurations. The key-value stores being evaluated have three parameters that affect performance: `memtable-size`, `level0-slowdown`, `level0-stop`. Note that Level 0 can have sstables with overlapping ranges; new sstables are simply appended to Level 0 (otherwise adding an sstable to Level 0 would trigger compaction, affecting write throughput). However, letting Level 0 grow without bounds will reduce read and range query throughput. The `memtable-size` parameter controls how big the memtable can grow before being written to storage. The other two parameters are used to slow down or stop writes to Level 0.

HyperLevelDB and RocksDB have different default values for these parameters. HyperLevelDB uses 4 MB memtables with `level0-slowdown` of 8 and `level0-stop` of 12. RocksDB uses 64 MB memtables with `level0-slowdown` of 20 and `level0-stop` of 24. When comparing PebblesDb with these systems, the default HyperLevelDB parameters are used. Certain experiments also report performance under RocksDB parameters.

	PebblesDB	HyperLevelDB
Average	17.23	13.33
Median	5.29	16.59
90th percentile	51.06	16.60
95th percentile	68.31	16.60

Table 1: SSTable Size. The table shows the distribution of sstable size (in MB) for PebblesDB and HyperLevelDB when 50 million key-value pairs totaling 33 GB were inserted.

5.2 Micro-benchmarks

This section evaluates PebblesDb performance using different single-threaded and multi-threaded micro-benchmarks and in various conditions. The single-threaded benchmarks help us understand the performance of different PebblesDb operations. The multi-threaded benchmark evaluates how PebblesDb performs in the more realistic setting of multiple readers and writers. PebblesDb is evaluated in different conditions such as when the dataset fits in memory, with small key-value pairs, with an aged file system and key-value store, and finally under extremely low memory conditions.

Write Amplification. We measure write amplification for workloads that insert or update keys in random order (key:16 bytes, value:128 bytes). Figure 4 (a) presents the results. PebblesDb write IO (in GB) is shown over the bars. PebblesDb consistently writes the least amount of IO, and the difference in write amplification between PebblesDb and other stores goes up as the number of keys increases. For 500M keys, PebblesDb lowers write amplification by 2.5× compared to RocksDB and HyperLevelDB and 1.6× compared to LevelDB.

Single-threaded Workloads. We use `db_bench` (a suite of micro-benchmarks that comes bundled with LevelDB) [33] to evaluate PebblesDb performance on various operations: 50M sequential writes, 50M random writes, 10M random reads and 10M random seeks. Reads and seeks were performed on the previously (randomly) inserted 50M keys. Each key was 16 bytes and the value was 1024 bytes. The results, presented in Figure 4 (b), show both the strengths and weaknesses of PebblesDb.

Random Writes and Reads. PebblesDb outperforms all other key-value stores in random writes due to the underlying FLSM data structure. PebblesDb throughput is 2.7× that of HyperLevelDB, the closest competitor. PebblesDb compaction finishes 2.5× faster than HyperLevelDB compaction. Random reads perform better in PebblesDb due to the larger sstables of PebblesDb (as shown in Table 1). The index blocks of all PebblesDb sstables are cached, whereas

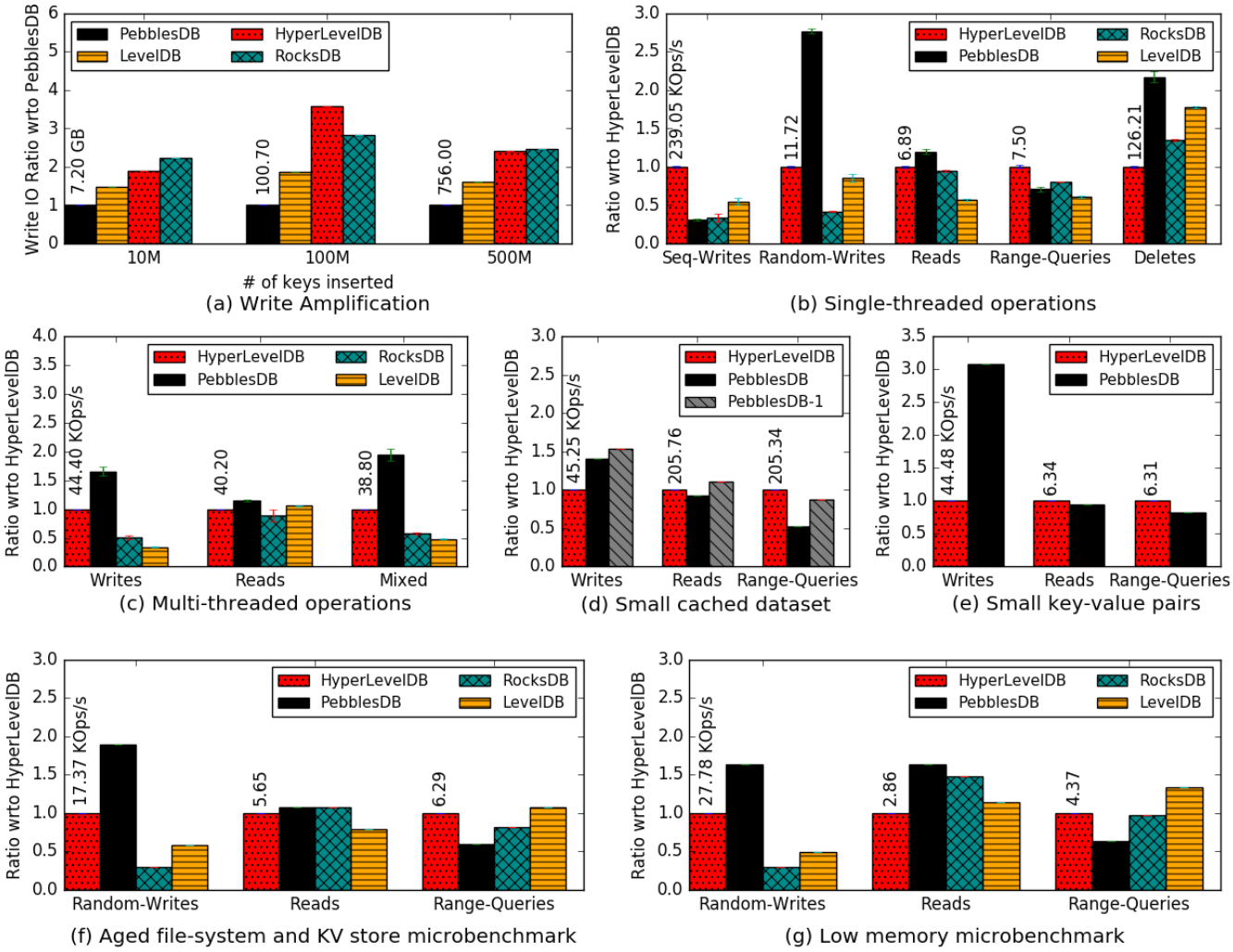


Figure 4: Micro-benchmarks. The figure compares the throughput of several key-value stores on various micro-benchmarks. Values are shown relative to HyperLevelDB, and the absolute value (in KOps/s or GB) of the baseline is shown above the bar. For (a), lower is better. In all other graphs, higher is better. PebblesDb excels in random writes, achieving 2.7× better throughput, while performing 2.5× lower IO.

there are cache misses for the index blocks of the many HyperLevelDB sstables. With larger caches for index blocks, PebblesDb read performance is similar to HyperLevelDB.

Sequential Writes. PebblesDb obtains 3× less throughput than HyperLevelDB on the sequential write workload; this is because sequential workloads result in disjoint sstables naturally (*e.g,*. first 100 keys go to the first sstable, second 100 keys go to the second sstable, *etc.*), LSM-based stores can just *move* the sstable from one level to another without doing any IO. On the other hand, PebblesDb always has to partition sstables based on guards (and therefore perform write IO) when moving sstables from one level to the next. As a result, PebblesDb performs poorly when keys are inserted sequentially. Many real-world workloads are multi-threaded, resulting in random writes; for example, in the YCSB workload suite which reflects real-world access patterns, none of the workloads insert keys sequentially [16].

Range Queries. A range query is comprised of an `seek()` operation followed by a number of `next()` operations. Range-query performance depends mainly on two factors: the number of levels in the key-value store on storage, and the number of `next()` operations. Figure 4 (b) shows key-value store performance for range queries comprising of only `seek()` operations, performed after allowing the key-value store time to perform compaction. As such, it represents a worst case for PebblesDB: the expensive `seek()` operation is not amortized by successive `next()` operations, and other key-value stores compact more aggressively than PebblesDB, since they do

	PebblesDB	HyperLevelDB	LevelDB	RocksDB
Insert 50M values	56.18	40.00	22.42	14.12
Update Round 1	47.85	24.55	12.29	7.60
Update Round 2	42.55	19.76	11.99	7.36

Table 2: Update Throughput. The table shows the throughput in KOps/s for inserting and updating 50M key-value pairs in different key-value stores.

not seek to minimize write IO. In this worst-case scenario, PebblesDB has a 30% overhead compared to HyperLevelDB, due to the fact that a `seek()` in PebblesDb requires reading multiple sstables from storage in each level. We note that in real-world workloads such as YCSB, there are many `next()` operations following a `seek()` operation.

Next, we measure range query performance in a slightly different setting. We insert 50M key-value pairs (key: 16 bytes, value: 1 KB), and immediately perform 10M range queries (each range query involves 50 `next()` operations). In this more realistic scenario, we find that PebblesDb overhead (as compared to HyperLevelDB) reduces to 15% from the previous 30%. If we increase range query size to 1000, the overhead reduces to 11%.

Unfortunately, even with many `next()` operations, PebblesDb range-query performance will be lower than that of LSM key-value stores. This is because PebblesDb pays both an IO cost (reads more sstables) and a CPU cost (searches through more sstables in memory, merges more iterators) for range queries. While the overhead will drop when the number of `next()` operations increase (as described above), it is difficult to eliminate both IO cost and CPU cost.

To summarize range-query performance, PebblesDb has significant overhead (30%) for range queries when the key-value store has been fully compacted. This overhead derives both from the fact that PebblesDb has to examine more sstables for a `seek()` operation, and that PebblesDb does not compact as aggressively as other key-value stores as it seeks to minimize write IO. The overhead is reduced for large range queries, and when range queries are interspersed with writes (such as in YCSB workload E).

Deletes and Updates. Deletes and Updates are handled similar to writes in LSM-based key-value stores. Updates do not check for the previous value of the key, so updates and new writes are handled identically. Deletes are simply writes with a zero-sized value and a special flag. We ran an experiment where we inserted 200M key-value pairs (key: 16 bytes, value: 128 bytes) into the database and deleted all inserted keys. We measure the deletion throughput. The results are presented in Figure 4 (b) and follow a pattern similar to writes: PebblesDb outperforms the other key-value stores due to its faster compaction.

We ran another experiment to measure update throughput. We inserted 50M keys (value: 1024 bytes) into the store, and then updated all keys twice. The results are presented in Table 2. We find that as the database becomes larger, insertion throughput drops since insertions are stalled by compactions and compactions involve more data in larger stores. While the other key-value stores drop to 50% of the initial write throughput, PebblesDB drops to only 75% of original throughput; we attribute this difference to the compaction used by the different key-value stores. The update throughput of PebblesDB is 2.15× that of HyperLevelDB, the closest competitor.

Space Amplification. The storage space used by PebblesDb is not significantly higher compared to LSM-based stores. LSM-based stores only reclaim space if the key has been updated or deleted. For a workload with only insertions of unique keys, the space used by RocksDB and PebblesDB will be identical. For workloads with updates and deletions, PebblesDB will have a slight overhead due to delay in merging. We inserted 50M unique key-value pairs. The storage-space consumption of RocksDB, LevelDB, and PebblesDb were within 2% of each other (52 GB). We performed another experiment where we inserted 5M unique keys, and updated each key 10 times (total 50M writes). Since the keys aren't compacted yet, PebblesDb consumes 7.9 GB while RocksDB consumes 7.1 GB. LevelDB consumed 7.8 GB of storage space.

Multi-threaded Reads and Writes. We use four threads to perform 10M read and 10M write operations (each) on the evaluated key-value stores. The reads are performed on the store after the write workload finishes. We use the default RocksDB configuration (64 MB memtable, large Level 0). Figure 4 (c) presents the results. PebblesDb performs the best on both workloads, obtaining 3.3× the write throughput of RocksDB (1.7× over baseline).

Concurrent Reads and Writes. In this experiment, two threads perform 10M reads each, while two other threads perform 10M writes each. Figure 4 (c) reports the combined throughput of reads and writes (*mixed*). PebblesDb outperforms the other stores. The lower write amplification leads to higher write throughput. Since compaction in PebblesDb is faster than the other stores, PebblesDb reaches a compacted state earlier with larger (and fewer) sstables, resulting in lower read latency and higher read throughput. Note that PebblesDb outperforms HyperLevelDB even when HyperLevelDB uses sstable-level bloom filters, thus demonstrating the gains are due to the underlying FLSM data structure.

Small Workloads on Cached Datasets. We run an experiment to determine the performance of PebblesDb on data

sets that are likely to be fully cached. We insert 1M random key-value pairs (key:16 bytes, value: 1KB) into HyperLevelDB and PebblesDb. The total dataset size is 1 GB, so it is comfortably cached by the test machine (RAM: 16 GB). We do 1M random reads and seeks. Figure 4 (d) presents the results. Even for small datasets, PebblesDb gets better write throughput than HyperLevelDB due to the FLSM data structure. Due to extra CPU overhead of guards, there is a small 7% overhead on reads and 47% overhead on seeks. When PebblesDb is configured to run with `max_sstables_per_guard` (§3.5) set to one so that it behaves more like an LSM store (*PebblesDB-1*), PebblesDb achieves 11% higher read throughput and the seek overhead drops to 13%.

Performance for Small Sized Key-Value Pairs. We insert 300M key-value pairs into the database (key: 16 bytes, value: 128 bytes). As shown in Figure 4 (e), PebblesDb obtains higher write throughput and equivalent read and seek throughputs (similar to results with large keys).

Impact of Empty Guards. We run an experiment to measure the performance impact of empty guards. We insert 20M key-value pairs (with keys from 0 to 20M, value size: 512B, dataset size: 10 GB), perform 10M read operations on the data, and delete all keys. We then repeat this, but with keys from 20M to 40M. We do twenty iterations of this experiment. Since we are always reading the currently inserted keys, empty guards due to old deleted keys will accumulate (there are 9000 empty guards at the beginning of the last iteration). Throughout the experiment, read throughput varied between 70 and 90 KOps/s. Read throughput did not reduce with more empty guards.

Impact of File-System and Key-Value Store Aging. Recent work has shown that file-system aging has a significant impact on performance [15]. To assess the impact of file-system and key-value store aging on PebblesDB, we run the following experiment. *File-system Aging*: We create a new file system on a 1.1 TB SSD, then use sequential key-value pair insertion to fill up the file system. We then delete all data in the file system, and fill the file system using the same process again until 130 GB of free space (11% of the file-system size) is left. *Key-Value Store Aging*: We then age the key-value store under evaluation by using four threads to each insert 50M key-value pairs, delete 20M key-value pairs, and update 20M key-value pairs in random order. Once both file-system and key-value store aging is done, we then run micro-benchmarks for writes, reads, and seeks (all in random order). The results are presented in Figure 4 (f). We find that the absolute performance numbers drop: 18% for reads and 16% for range queries (mainly because there is more data in the key-value store from the aging). As with a fresh file system, PebblesDB outperforms the other key-value stores on writes (although the throughput speedup reduces to 2× from 2.7×). Similarly, PebblesDB outperforms HyperLevelDB by 8% (down from 20% on a fresh file system) on reads, and incurs a 40% penalty on range queries (as compared to 30% on a fresh file system) compared to HyperLevelDB.

Workload	*Description*	*Represents*
Load A	100% writes	Insert data for workloads A–D and F
A	50% reads, 50% writes	Session recording recent actions
B	95% reads, 5% writes	Browsing and tagging photo album
C	100% reads	Caches
D	95% reads (latest values), 5% writes	News feed or status feed
Load E	100% writes	Insert data for Workload E
E	95% Range queries, 5% writes	Threaded conversation
F	50% reads, 50% Read-modify-writes	Database workload

Table 3: YCSB Workloads. The table describes the six workloads in the YCSB suite. Workloads A–D and F are preceded by Load A, while E is preceded by Load E.

Performance Under Low Memory. We evaluate the performance of PebblesDb when the total available memory is a small percentage of the dataset size. We insert 100M key-value pairs (key:16 bytes, value: 1K) for a total dataset size of 65 GB. We restrict the RAM on our machine using the `mem` kernel boot parameter to 4 GB. Thus, the total available DRAM is only 6% of the total dataset size (in our previous experiments, it was 30%). We evaluate the performance of PebblesDb under these conditions using micro-benchmarks. The results are presented in Figure 4 (g). All key-values stores evaluated use a 64 MB memtable and a large Level 0. We find that PebblesDb still outperforms the other key-value stores at random writes, although the margin (with respect to HyperLevelDB) reduces to 64%. PebblesDb outperforms HyperLevelDB on random reads by 63%. On the range query micro-benchmark, PebblesDb experiences a 40% penalty compared to HyperLevelDB. Thus, PebblesDb still achieves good performance in reads and writes when memory is scarce, although range queries experience more performance degradation.

Impact of Different Optimizations. We describe how the different optimizations described in the paper affect PebblesDb performance. If PebblesDb doesn't use any optimizations for range queries, range query throughput drops by 66% (48 GB dataset). The overhead drops to 48% if parallel

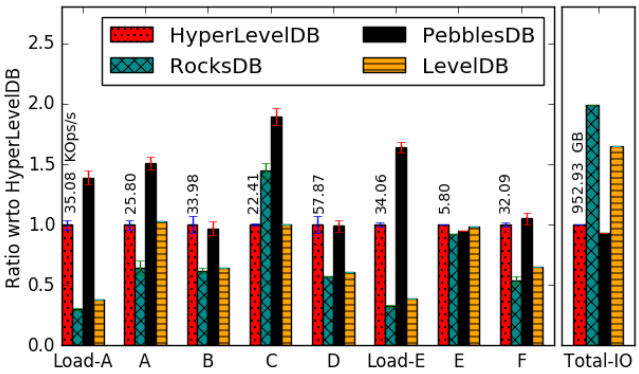


Figure 5: YCSB Performance. The figure shows the throughput (bigger is better except for Total-IO bars) of different key-value stores on the YCSB Benchmark suite run with four threads. PebblesDb gets higher throughput than RocksDB on almost all workloads, while performing 2× lower IO than RocksDB.

seeks are used, and only 7% if only seek-based compaction is used. Using sstable-level bloom filters improves read performance by 63% (53 GB dataset).

5.3 Yahoo Cloud Serving Benchmark

The industry standard in evaluating key-value stores is the Yahoo Cloud Serving Benchmark [16]. The suite has six workloads (described in Table 3), each representing a different real-world scenario. We modify db_bench [33] to run the YCSB benchmark with 4 threads (one per core) and using default RocksDB parameters (64MB memtable and large Level 0). We run RocksDB with 4 background compaction threads to further boost its performance. Load-A and Load-E do 50M operations each, all other workloads do 10M operations each. Figure 5 presents the results: PebblesDb outperforms both RocksDB and HyperLevelDB on write workloads, while obtaining nearly equal performance on all other workloads. Overall, PebblesDb writes 50% less IO than RocksDB.

On write-dominated workloads like Load A and Load E, PebblesDb achieves 1.5–2× better throughput due to the faster writes offered by the underlying FLSM data structure.

For the read-only Workload C, PebblesDb read performance is better than other key-value stores due to the larger sstables of PebblesDb. The key-value stores cache a limited number of sstable index blocks (default: 1000): since PebblesDb has fewer, larger files, most of its sstable-index-blocks are cached. The cache misses for the other key-value stores result in reduced read performance. When we increase the number of index blocks cached, PebblesDb read performance becomes similar to the other key-value stores. Note that the larger sstables of PebblesDb result from compaction: in workloads such as B and D, the constant stream of writes adds new sstables that are not compacted; as a result, PebblesDb throughput is similar to the other key-value stores.

For the range-query-dominated Workload E, PebblesDb surprisingly has performance close (6% overhead) to the other key-value stores. When we analyzed this, we found that the small amount of writes in the workload (Workload E has 5% writes) prevent any key-value store from full compacting; as a result, every key-value store has to examine multiple levels, which reduces the performance impact of the extra sstables examined by PebblesDb. When the YCSB workload is modified to contain only range queries, PebblesDb throughput is 18% lower than HyperLevelDB as expected. Each range query in this workload does N `next()` operations (N picked randomly from 1 to 100), and the `next()` operations also contribute in reducing range-query overhead.

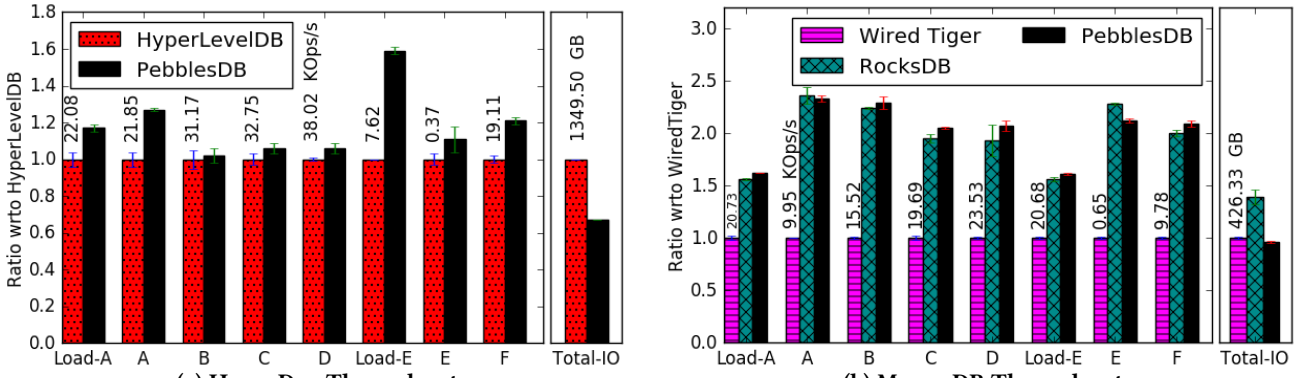
In Workload F, all writes are read-modify-writes: the workload does a `get()` before every `put()` operation. As a result, the full write throughput of PebblesDb is not utilized, resulting in performance similar to that of other key-value stores. We see similar read-modify-write behavior in applications such as HyperDex and MongoDB.

5.4 NoSQL Applications

We evaluate the performance of two real-world applications, the HyperDex and MongoDB NoSQL stores, when they use PebblesDb as the underlying storage engine. We use the Java clients provided by HyperDex and MongoDB for running the YCSB benchmark, with both the server and client running on the same machine (no network involved).

HyperDex. HyperDex is a high-performance NoSQL store that uses HyperLevelDB as it storage engine by default [18]. We evaluate the performance impact of using PebblesDb as the storage engine by running the YCSB benchmark with 4 threads. Load-A inserts 20M values, Load-E inserts 30M values, A–D and F perform 10M operations each, and E performs 250K operations (lower number of ops as HyperDex range-query latency is very high). We use the same setup used by HyperDex developers to benchmark their system using YCSB [28]. Both HyperLevelDB and PebblesDb are configured with the default HyperDex parameters (16 MB memtable size).

Figure 6 (a) presents the results. In every workload, using PebblesDb improves HyperDex throughput, with the highest gain of 59% coming when inserting 30M key-value pairs in the Load-E workload. HyperDex adds significant latency to operations done by YCSB. For example, the average latency to insert a key in HyperDex is 151 μs, of which PebblesDb accounts for only 22.3 μs (14.7%). Furthermore, HyperDex checks whether a key already exists before inserting, turning every `put()` operation in the Load workloads into a `get()` and a `put()`. This behavior of HyperDex reduces the performance gain from PebblesDb, because PebblesDb can handle much higher rate of insertions. Despite



(a) HyperDex Throughput **(b) MongoDB Throughput**

Figure 6: Application Throughput. The figure shows the YCSB throughput (bigger is better except last bar) of the HyperDex document store and MongoDB NoSQL store when using different key-value stores as the storage engine. The throughput is shown relative to the default storage option (HyperLevelDB for HyperDex, WiredTiger for MongoDB). The raw throughput in KOps/s or total IO in GB of the default option is shown above the bars.

this, PebblesDb increases HyperDex throughput while simultaneously reducing write IO.

When we increase the value size from the YCSB default of 1 KB to 16 KB, the speedup HyperDex achieves from using PebblesDb drastically increases: the geometric mean of the speedup is **105%** (not shown). As the value size increases, more IO is required for all operations, making the extra CPU overhead of PebblesDb negligent, and highlighting the benefits of the FLSM data structure.

MongoDB. We configure MongoDB [40], a widely-used NoSQL store, to use PebblesDb as the storage engine. MongoDB can natively run with either the Wired Tiger key-value store (default) or RocksDB. We evaluate all three options using the YCSB Benchmark suite. All three stores are configured to use 8 MB cache and a 16 MB memtable. Since Wired Tiger is not a LSM-based store (it uses checkpoints + journaling), it does not use memtables; instead, it collects entries in a log in memory. We configure the max size of this log to be 16 MB. Figure 6 (b) presents the results. We find that both RocksDB and PebblesDb significantly outperform Wired Tiger on all workloads, demonstrating why LSM-based stores are so popular. While RocksDB performs 40% more IO than Wired Tiger, PebblesDb writes 4% lesser IO than Wired Tiger.

We investigated why PebblesDb write throughput is not 2× higher than RocksDB as in the YCSB benchmark. As in HyperDex, MongoDB itself adds a lot of latency to each write (PebblesDb write constitutes only 28% of latency of MongoDB write) and provides requests to PebblesDb at a much lower rate than PebblesDb can handle. The slower request rate allows RocksDB's compaction to keep up with the inserted data; thus, PebblesDb's faster compaction is not utilized, and the two key-value stores have similar write throughput. Note that PebblesDb still writes 40% lesser IO then RocksDB, providing lower write amplification.

Workload	*HyperLevelDB*	*RocksDB*	*PebblesDb*
Writes (100M)	159	896	434
Reads (10M)	154	36	500
Seeks (10M)	111	34	430

Table 4: Memory Consumption. The table shows the memory consumed (in MB) by key-value stores for different workloads.

Summary. PebblesDb does not increase performance on HyperDex and MongoDB as significantly as in the YCSB macro-benchmark. This is both due to PebblesDb latency being a small part of overall application latency, and due to application behavior such as doing a read before every write. If the application is optimized for PebblesDb, we believe the performance gains would be more significant. Despite this, PebblesDb reduces write amplification, providing either equal (MongoDB) or better performance (HyperDex).

5.5 Memory and CPU Consumption

Memory Consumption. We measure memory used during the insertion of 100M keys (key size: 16 bytes, value size: 1024 bytes, total: 106 GB) followed by 10M reads and range queries. The results are shown in Table 4. PebblesDb consumes about 300 MB more than HyperLevelDB. PebblesDb uses 150 MB for storing sstable bloom filters, and 150 MB for temporary storage for constructing the bloom filters.

CPU Cost. We measured the median CPU usage during the insertion of 30M keys, followed by reads of 10M keys. The median CPU usage of PebblesDb is 170.95%, while the

median for the other key-value stores ranged from 98.3–110%. The increased CPU usage is due to the PebblesDb compaction thread doing more aggressive compaction.

Bloom Filter Construction Cost. Bloom filters are calculated over all the keys present in an sstable. The overhead of calculating the bloom filter is incurred only the first time the sstable is accessed. The time taken to calculate depends on the size of sstable. We observed the rate of calculation to be 1.2 s per GB of sstable. For 3200 sstables totaling 52 GB, bloom filter calculation took 62 seconds.

6 RELATED WORK

The work in this paper builds on extensive prior work in building and optimizing key-value stores. The key contribution relative to prior work is the FLSM data structure and demonstrating that a high performance key-value store that drastically reduces write amplification can be built on top of FLSM. This section briefly describe prior work and places the work in this paper in context.

Reducing Write Amplification. Various data structures have been proposed for implementing key-value stores. Fractal Index trees [11] (see TokuDB [36]) were suggested to reduce the high IO cost associated with traditional B-Trees. While FLSM and Fractal index trees share the same goal of reducing write IO costs, Fractal index trees do not achieve high write throughput by taking advantage of large sequential writes, and do not employ in-memory indexes such as bloom filters to improve performance like PebblesDb.

NVMKV [38] uses a hashing-based design to reduce write amplification and deliver close to raw-flash performance. NVMKV is tightly coupled to the SSD's Flash Translation Layer (FTL) and cannot function without using FTL features such as atomic multi-block write. Similarly, researchers have proposed building key-value stores based on vector interfaces (that are not currently available) [55]. In contrast, PebblesDb is device-agnostic and reduces write amplification on both commodity hard drives and SSDs. We should note that we have not tested PebblesDb on hard-drives yet; we believe the write behavior will be similar, although range query performance may be affected.

The HB+-trie data structure is used in ForestDB [4] to efficiently index long keys and reduce space overhead of internal nodes. FLSM and HB+trie target different goals resulting in different design decisions; FLSM is designed to reduce write amplification, not space amplification.

The LSM-trie [57] data structure uses *tries* to organize keys, thereby reducing write amplification; however, it does not support range queries. Similarly, RocksDB's universal compaction reduces write amplification by sacrificing read and range query performance [22]. PebblesDb employs additional techniques over FLSM to balance reducing write amplification with reasonable range query performance.

TRIAD [10] uses a combination of different techniques such as separating hot and cold keys, using commit logs as sstables, and delaying compaction to reduce write IO and improve performance. The TRIAD techniques are orthogonal to our work and can be incorporated into PebblesDb.

Improving Key-Value store Performance. Both academia and industry have worked on improving the performance of key-value stores based on log-structured merge trees. PebblesDb borrows optimizations such as sstable bloom filters and multi-threaded compaction from RocksDB. HyperLevelDB [29] introduces fine-grained locking and a new compaction algorithm that increases write throughput. bLSM [51] introduces a new merge scheduler to minimize write latency and maintain write throughput, and uses bloom filters to improve performance. VT-Tree [52] avoids unnecessary data copying for data that is already sorted using an extra level of indirection. WiscKey [37] improves performance by not storing the values in the LSM structure. LOCS [56] improves LSM compaction using the internal parallelism of open-channel SSDs. cLSM [23] introduces a new algorithm for increasing concurrency in LSM-based stores. We have a different focus from these work: rather than making LSM-based stores better, we introduce a better data structure, FLSM, and demonstrate that it can be used to build high performance key-value stores. Many of the techniques in prior work can be readily adapted for FLSM and PebblesDb.

7 CONCLUSION

This paper presents PebblesDb, a high-performance key-value store that achieves low write amplification, high write throughput, and high read throughput simultaneously. PebblesDb outperforms widely-used stores such as RocksDB on several workloads. PebblesDb is built on top of a novel data structure, Fragmented Log-Structured Merge Trees, that combines ideas from skip lists and log-structured merge trees. PebblesDb is publicly available at `https://github.com/utsaslab/pebblesdb`. Since it shares the same API as LevelDB, we hope this will aid in adoption by applications.

ACKNOWLEDGMENTS

We would like to thank our shepherd, Frans Kaashoek, the anonymous reviewers, and members of the LASR group and the Systems and Storage Lab for their feedback and guidance. This work was supported by generous donations from VMware and Facebook. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of other institutions.

REFERENCES

[1] Ittai Abraham, James Aspnes, and Jian Yuan. 2005. Skip B-trees. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*. 366–380.

[2] Alok Aggarwal, Jeffrey Vitter, et al. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[3] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the 2008 USENIX Annual Technical Conference*. 57–70.

[4] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Trans. Comput.* 65, 3 (2016), 902–915.

[5] Reed Allman. 2014. Rock Solid Queues @ Iron.io. `https://www.youtube.com/watch?v=HTjt6oj-RL4`. (2014).

[6] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array Of Wimpy Nodes. In *Poceedings of the ACM SIGOPS 22nd Symposium On Operating Systems Principles (SOSP 09)*. ACM, 1–14.

[7] Apache. 2017. Search Results Apache Flink: Scalable Stream and Batch Data Processing. `https://flink.apache.org`. (2017).

[8] Austin Appleby. 2016. SMHasher test suite for MurmurHash family of hash functions. `https://github.com/aappleby/smhasher`. (2016).

[9] Anirudh Badam, KyoungSoo Park, Vivek S. Pai, and Larry L Peterson. 2009. HashCache: Cache Storage for the Next Billion. In *Proceedings of the 6th USENIX Symposium on Network Systems Design and Implementation (NSDI 09)*. 123–136.

[10] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA, 363–375.

[11] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-Oblivious Streaming B-trees. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 81–92.

[12] Michael A. Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A. Phillips, and Helen Xu. 2017. Write-Optimized Skip Lists. In *Proceedings of the 36th ACM Symposium on Principles of Database Systems (PODS '17)*. ACM, New York, NY, USA, 69–78.

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

[14] Douglas Comer. 1979. Ubiquitous B-tree. *ACM Computing Surveys (CSUR)* 11, 2 (1979), 121–137.

[15] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Martin Farach-Colton. 2017. File Systems Fated for Senescence? Nonsense, Says Science!. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*. 45–58.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC 10)*. ACM, 143–154.

[17] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-Based Storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, 25–36.

[18] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: a distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference*. 25–36.

[19] Facebook. 2017. FIFO compaction style. `https://github.com/facebook/rocksdb/wiki/FIFO-compaction-style`. (2017).

[20] Facebook. 2017. RocksDB | A persistent key-value store. `http://rocksdb.org`. (2017).

[21] Facebook. 2017. RocksDB Users. `https://github.com/facebook/rocksdb/blob/master/USERS.md`. (2017).

[22] Facebook. 2017. Universal Compaction. `https://github.com/facebook/rocksdb/wiki/Universal-Compaction`. (2017).

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

[24] Daniel Golovin. 2010. The B-Skip-List: A Simpler Uniquely Represented Alternative to B-Trees. *CoRR* abs/1005.0662 (2010).

[25] Google. 2017. LevelDB. `https://github.com/google/leveldb`. (2017).

[26] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42)*. IEEE, 24–33.

[27] James Hamilton. 2009. The Cost of Latency. `http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/`. (2009).

[28] HyperDex. 2016. HyperDex Benchmark Setup. `http://hyperdex.org/performance/setup/`. (2016).

[29] HyperDex. 2017. HyperLevelDB Performance Benchmarks. `http://hyperdex.org/performance/leveldb/`. (2017).

[30] Cockroach Labs. 2017. CockroachDB. `https://github.com/cockroachdb/cockroach`. (2017).

[31] Dgraph labs. 2017. Dgraph: Graph database for production environment. `https://dgraph.io`. (2017).

[32] FAL Labs. 2011. Kyoto Cabinet: a straightforward implementation of DBM. `http://fallabs.com/kyotocabinet/`. (2011).

[33] LevelDB. 2016. LevelDB db_bench benchmark. `https://github.com/google/leveldb/blob/master/db/db_bench.cc`. (2016).

[34] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP 11)*. ACM, 1–13.

[35] LinkedIn. 2016. FollowFeed: LinkedIn's Feed Made Faster and Smarter. `http://bit.ly/2onMQwN`. (2016).

[36] Percona LLC. 2017. Percona TokuDB. `https://www.percona.com/software/mysql-database/percona-tokudb`. (2017).

[37] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*. 133–148.

[38] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: a Scalable, Lightweight, FTL-aware Key-Value Store. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 207–219.

[39] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R. Nevill. 2008. Bit Error Rate in NAND Flash Memories. In *Proceedings of the IEEE International Reliability Physics Symposium, (IRPS 08)*. IEEE, 9–19.

[40] MongoDB. 2017. MongoDB. `https://www.mongodb.com`. (2017).

[41] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. 2009. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference*

on Computer Systems (Eurosys 09). ACM, 145–158.

[42] Suman Nath and Aman Kansal. 2007. FlashDB: Dynamic Self-tuning Database for NAND flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*. ACM, 410–419.

[43] Netflix. 2016. Application Data Caching using SSDs. `http://techblog.netflix.com/2016/05/application-data-caching-using-ssds.html`. (May 2016).

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

[45] Oracle. 2017. Oracle Berkeley DB. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`. (2017).

[46] Pinterest. 2016. Open-sourcing Rocksplicator, a real-time RocksDB data replicator. `http://bit.ly/2pv5nZZ`. (2016).

[47] William Pugh. 1989. Skip lists: A probabilistic alternative to balanced trees. *Algorithms and Data Structures* (1989), 437–449.

[48] William Pugh. 1990. *A Skip List Cookbook*. Technical Report CS-TR-2286.1. University of Maryland.

[49] Parthasarathy Ranganathan. 2011. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *Computer* 44, 1 (2011), 39–48.

[50] Apache Samza. 2017. State Management. `http://samza.apache.org/learn/documentation/0.8/container/state-management.html`. (2017).

[51] Russell Sears and Raghu Ramakrishnan. 2012. bLSM: a General Purpose Log Structured Merge Tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 217–228.

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

[53] RocksDB Issue Tracker. 2014. Strategies to Reduce Write Amplification 19. `https://github.com/facebook/rocksdb/issues/19`. (2014).

[54] Uber. 2016. Cherami: Uber Engineering's Durable and Scalable Queue in Go. `https://eng.uber.com/cherami/`. (2016).

[55] Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. 2012. Using Vector Interfaces To Deliver Millions Of IOPS From A Networked Key-Value Storage Server. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC 12)*. ACM, 8.

[56] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An Efficient Design And Implementation Of LSM-Tree Based Key-Value Store On Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (Eurosys 14)*. ACM, 16.

[57] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-Tree-Based Ultra-Large Key-Value Store for Small Data Items. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 71–82.

[58] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. 2005. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*.