View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

FloDB: Unlocking Memory in Persistent Key-Value Stores

Oana Balmau[1], Rachid Guerraoui[1], Vasileios Trigonakis[2] *, and Igor Zablotchi[1] †
[1]*EPFL*, {first.last}@epfl.ch
[2]*Oracle Labs*, {first.last}@oracle.com

Abstract

Log-structured merge (LSM) data stores enable to store and process large volumes of data while maintaining good performance. They mitigate the I/O bottleneck by absorbing updates in a memory layer and transferring them to the disk layer in sequential batches. Yet, the LSM architecture fundamentally requires elements to be in sorted order. As the amount of data in memory grows, maintaining this sorted order becomes increasingly costly. Contrary to intuition, existing LSM systems could actually lose throughput with larger memory components.

In this paper, we introduce FloDB, an LSM memory component architecture which allows throughput to scale on modern multicore machines with ample memory sizes. The main idea underlying FloDB is essentially to bootstrap the traditional LSM architecture by adding a small in-memory buffer layer on top of the memory component. This buffer offers low-latency operations, masking the write latency of the sorted memory component. Integrating this buffer in the classic LSM memory component to obtain FloDB is not trivial and requires revisiting the algorithms of the user-facing LSM operations (search, update, scan). FloDB's two layers can be implemented with state-of-the-art, highly-concurrent data structures. This way, as we show in the paper, FloDB eliminates significant synchronization bottlenecks in classic LSM designs, while offering a rich LSM API.

We implement FloDB as an extension of LevelDB, Google's popular LSM key-value store. We compare FloDB's performance to that of state-of-the-art LSMs. In short, FloDB's performance is up to one order of magnitude higher than that of the next best-performing competitor in a wide range of multi-threaded workloads.

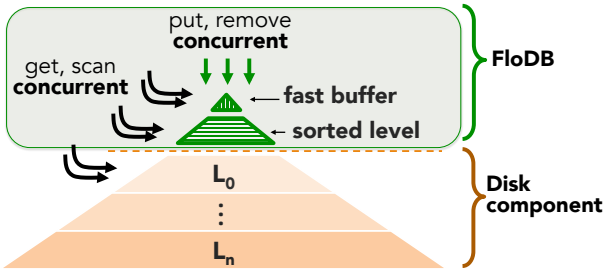

Figure 1: LSM data store using FloDB.

1. Introduction

Key-value stores are a crucial component of many systems that require low-latency access to large volumes of data [1, 6, 12, 15, 16]. These stores are characterized by a flat data organization and simplified interface, which allow for efficient implementations. However, the amount of data targeted by key-value stores is usually larger than main memory; thus, persistent storage is generally required. Since accessing persistent storage is slow compared to CPU speed [41, 42], updating data directly on disk yields a severe bottleneck.

To address this challenge, many key-value stores adopt the log-structured merge (LSM) architecture [35, 36]. Examples include LevelDB [5], RocksDB [12], cLSM [26], bLSM [39], HyperLevelDB [6] and HBase [11]. LSM data stores are suitable for applications that require low latency accesses, such as message queues that undergo a high number of updates, and for maintaining session states in user-facing applications [12]. Basically, the LSM architecture masks the disk access bottleneck, on the one hand, by caching reads and, on the other hand, by absorbing writes in memory and writing to disk in batches at a later time. Although LSM key-value stores go a long way addressing the challenge posed by the I/O bottleneck, their performance does not however scale with the size of the in-memory component, nor does it scale up with the number of threads. In other words, and maybe surprisingly, increasing the in-memory parts of existing LSMs only benefits performance up to a relatively small size. Similarly, adding threads does not improve the throughput of many existing LSMs, due to their use of global blocking synchronization.

As we discuss in this paper, the two aforementioned scalability limitations are inherent to the design of traditional

* The project was completed while the author was at EPFL.

† Authors appear in alphabetical order.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '17, April 23-26, 2017, Belgrade, Serbia

© 2017 ACM. ISBN 978-1-4503-4938-3/17/04…$15.00
DOI: http://dx.doi.org/10.1145/3064176.3064193

LSM architectures. We circumvent these limitations by introducing FloDB, a novel LSM memory component, designed to scale up with the number of threads as well as with its size in memory. Traditionally, LSMs have been employing a two-tier storage hierarchy, with one level in memory and one level on disk. We propose an additional in-memory level. In other words, FloDB is a *two-level memory component* on top of the disk component (Figure 1), each in-memory level being a concurrent data structure [2, 3, 23, 29]. The top in-memory level is a small and fast data structure, while the bottom in-memory level is a larger, sorted data structure. New entries are inserted in the top level and then drained to the bottom level in the background, before being stored onto disk.

This scheme has several advantages. First, it allows scans and writes to proceed in parallel, on the bottom and on the top levels respectively. Second, the use of a small, fast top-level enables low-latency updates regardless of the size of the memory component, whereas existing LSMs see their performance drop as the memory component size increases. Larger memory components can support longer bursts of writes at peak throughput. Third, maintaining the bottom memory layer sorted allows flushing to disk to proceed without an additional—and expensive—sorting step.

We implement FloDB using a small high-performance concurrent hash table [21] for the top in-memory level and a larger concurrent skiplist [29] for the bottom in-memory level. At a first glance, it might seem that implementing FloDB requires nothing more than adding an extra hash table-based buffer level on top of an existing LSM architecture. However, this seemingly small step entails subtle technical challenges. The first challenge is to ensure efficient data flow between the two in-memory levels, so as to take full advantage of the additional buffer while not depleting system resources. To this end, we introduce the *multi-insert* operation, a novel operation for concurrent skiplists. The main idea is to insert n sorted elements in the skiplist in *one* operation, using the spot of the previously inserted element as a starting point for inserting the next elements, thus reusing already traversed hops. The skiplist multi-insert is of independent interest and can benefit previous single-writer LSM implementations as well, by increasing the throughput of updates to the memory component. The second challenge is ensuring the consistency of user-facing LSM operations while enabling a high level of concurrency between these operations. In particular, FloDB is the first LSM system to simultaneously support consistent scans and in-place updates.

Our experiments show that FloDB outperforms current key-value store solutions, especially in write-intensive scenarios. For instance, in a write-only workload, FloDB is able to saturate the disk component throughput with just one worker thread and continues to outperform the next best performing system up to 16 worker threads, by a factor of 2x on average. Moreover, for skewed read-write workloads, FloDB obtains up to one order of magnitude better throughput than its highest performing competitor.

To summarize, our contributions are threefold:

- FloDB, a two-level architecture for the log-structured merge memory component. FloDB scales with the amount of main memory, and has a rich API, where reads, writes and scans can all proceed concurrently.
- A publicly available implementation in C++ of the FloDB architecture as an extension of LevelDB, as well as an evaluation thereof with up to 192GB memory component size on a Xeon multicore machine.
- An algorithm for a novel multi-insert operation for concurrent skiplists. Besides FloDB, multi-insert can benefit any other LSM architecture employing a skiplist.

Despite its advantages, we do not claim FloDB is a silver bullet; it does have limitations. First, as in all LSMs, the steady-state throughput is bounded by the slowest component in the hierarchy: writing the memory component to disk. The improvements FloDB makes in the memory component are orthogonal to potential disk-level improvements, which go beyond the scope of this paper. Disk-level improvements could be used in conjunction with FloDB to further boost the overall performance of LSMs. Second, it is possible to devise workloads that prove problematic for the scans of FloDB. For example, in write-intensive workloads with heavy contention the performance of long-running scans decreases. Third, for datasets much larger than main memory, FloDB improves read performance less than write performance. This is due to the fact that in LSMs read performance largely depends on the effectiveness of caching, which is also outside the scope of our paper.

Roadmap. The rest of this paper is organized as follows. In Section 2, we give an overview of state-of-the-art LSMs and their limitations. In Section 3, we present the FloDB architecture and the flow of its operations. In Section 4, we describe in more details the technical solutions we adopted in FloDB. In Section 5, we report on the evaluation of FloDB's performance. In Section 6, we discuss related work and we conclude in Section 7.

2. Shortcomings of Current LSMs

In this section, we give an overview of current LSM-based key-value stores and highlight two of their significant limitations that we address with this work.

2.1 LSM Key-Value Store Overview

Key-value stores [1, 4, 18, 19, 22, 24] represent each data item by a key-value tuple (k, v), uniquely identified by the key k. The basic operations in key-value stores are *put*, *get* and *remove*. *Put* takes a key and a value and inserts the pair in the store. If a data entry with the same key already exists, the newly inserted entry will typically replace the existing one. *Get* takes a key and returns the value associated to it in the store or an indication that no pair with that key exists.

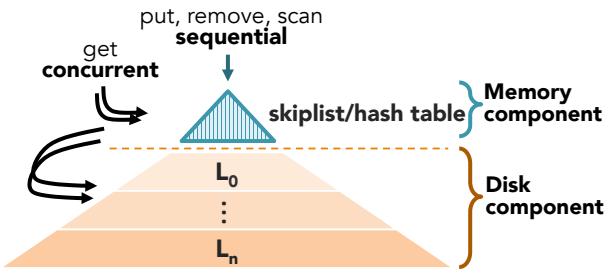

Figure 2: High-level view of a typical LSM.

Remove takes a key and (logically) deletes the pair with that key from the store, if such a pair exists. Additionally, many key-value stores support *scans* [5, 12, 24]. A scan takes two keys—a lower bound and an upper bound for the scan—and returns all key-value pairs whose keys are between the two input keys according to some comparator function. It is common for scans to implement *point-in-time* semantics (we call such scans *serializable* [37]): the returned view is a consistent state of the data store at some point in time, possibly from the past (before the scan was invoked).

The *log-structured merge* (LSM) architecture is aimed at reducing the impact of high-latency storage on the performance of data stores. In general, LSMs adopt a two-tier architecture with one level in memory to absorb writes and one level on disk to hold the bulk of the data. A high-level view of a typical LSM data store is depicted in Figure 2. A typical LSM has two components: one residing in main memory (the *memory component*) and one residing in persistent storage (the *disk component*). The memory component acts as a buffer/cache: updates (*put* and *delete* operations) are completed in the memory component and return immediately. Reads (*get* operations) first check the memory component; if a key-value pair with the given key is found, then the read can return immediately, since the memory component contains the most recent items. Otherwise, the read checks the disk component as well. The disk component is structured into several *levels*, where each level holds one or more sorted files. If there is a strict requirement not to lose any data in case of failure, then updates are appended to an on-disk commit-log before being applied to the in-memory component. This way, the recovery process can re-construct any lost operations from the log.

A key procedure in LSMs is *compaction* [36], a background operation performed by one or more dedicated threads. When the memory component reaches a certain size, it is merged into the disk component and a new, empty memory component is created. In order to allow new operations to complete during the compaction process, the old memory component is made immutable but remains accessible to readers, while write operations complete in the new memory component. Broadly, compaction in LSMs consists of (1) writing the immutable memory component to disk and (2) reorganizing the on-disk hierarchy, i.e., merging and moving files to different levels in the on-disk structure, to maintain the sorted-file structure, with non-overlapping key ranges between files on the same level.

2.2 Limitation—Scalability with Number of Threads

The presence of multiple processing cores can boost the performance of LSM data stores. Although existing LSM systems allow for some level of concurrency [6, 12, 26], they leave significant opportunities for parallelism untapped.

For instance, LevelDB—a basis for many LSM key-value stores—supports multiple writer threads, but serializes writes by having threads deposit their intended writes in a concurrent queue; the writes in this queue are applied to the key-value store one by one by a single thread. Moreover, LevelDB also requires readers to take a global lock during each operation so as to access or update metadata. HyperLevelDB [6] builds upon LevelDB, improving concurrency by allowing concurrent updates; yet, expensive synchronization is still needed in order to maintain the order of the updates, through version numbers. RocksDB [12], a key-value store also stemming from LevelDB, increases concurrency by introducing multithreaded merging of the disk components. While multithreaded compaction does indeed improve overall performance, RocksDB still keeps points of global synchronization to access in-memory structures and to update version numbers, similarly to HyperLevelDB. cLSM [26] goes even further by removing any blocking synchronization from the read-only path, but system scalability is still impaired by the use of global shared-exclusive locks to coordinate between updates and background disk writes. As we show in Section 5, these scalability bottlenecks indeed manifest in practice.

2.3 Limitation—Scalability with Memory Size

Existing LSM memory components can either be sorted (e.g., skiplist) or unsorted (e.g., hash table) [7]. Both options have their advantages and disadvantages, but surprisingly, neither is able to scale to large memory components.

On the one hand, when a skiplist is used, in-order scans are natural. Also, the compaction stage is little more than a direct copy of the component to disk; thus it has low overhead. However, writes require logarithmic time in the size of the data structure to maintain the sorted order (reads satisfied directly from the memory component also require logarithmic time in the size of the data structure. With large datasets, however, most reads are satisfied from the disk component, so the memory component data structure choice influences read latency less than write latency). Hence, as can be seen in Figure 3, allocating more memory actually increases write latency. The figure shows the median read and write latencies as functions of the memory component size in RocksDB, one of the most popular state-of-the-art LSMs. At the 99th percentile, we observed similar read and write latency trends. Latencies are measured using RocksDB's `readwhilewriting` benchmark (from `db_bench`

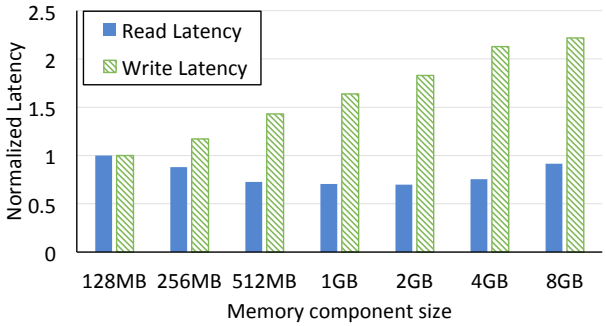

Figure 3: RocksDB skiplist-based memory component. Median read and write latencies, with increasing memory.

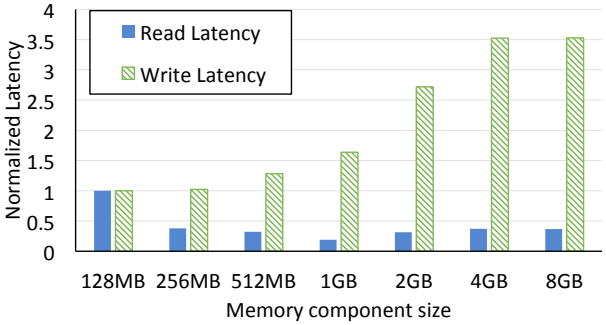

Figure 4: RocksDB hash table-based memory component. Median read and write latencies, with increasing memory.

[14]), with eight reader threads and one writer thread acting on a 1 million-entry database. The key size is 8 bytes and the value size is 256 bytes. Latencies are normalized to the 128 MB memory component.

On the other hand, with a hash table, writes complete in constant time, but in-order scans are not practical and the compaction stage is more complex. Compaction requires full sorting of the memory component before it can be written to disk, to preserve the LSM structure. Indeed, our measurements show that the mean compaction time for hash table-based memory components is at least an order of magnitude higher than for skiplist-based memory components of the same size: as the hash table becomes larger, it takes increasingly longer to sort and persist to disk. In the time the immutable memory component is sorted and persisted, it is possible for the active (mutable) memory component to also become full. In this case, the writers are delayed, since there is no room to complete their operations in memory. As a consequence, end-to-end write latency increases with memory size, as can be seen in Figure 4, which shows the same experiment as Figure 3 with a hash table instead of a skiplist.

Therefore, for both the skiplist and the hash table, the end-to-end system throughput plateaus or even decreases as more memory is added. This limitation is inherent in the single-level memory component of traditional LSMs and limits LSM users from leveraging the abundance of memory in modern multi-cores. In what follows, we show that it is possible to combine the benefits of hash tables and skiplists, to boost write throughput in LSMs while still allowing in-order scans.

3. FloDB Design

Recall from the previous section that existing LSMs have inherent scalability problems, both in terms of memory size and in terms of number of threads. The latter is caused by scalability bottlenecks, whereas the former stems from a *size–latency* trade-off.

This trade-off manifests for both sorted and unsorted memory components. A sorted component allows scans and is straightforward to persist to disk, but has significantly higher access times as it gets larger. An unsorted component can be fast regardless of size, but is not practical for scans and needs to be sorted in linearithmic [40] time before being flushed to disk, potentially delaying writers.

FloDB's memory component architecture is designed to circumvent these problems. The overarching goals of FloDB are (1) to scale with the amount of memory given, (2) to use minimal synchronization (for scaling up) and (3) to leverage the memory component in order to improve write performance without sacrificing read performance.

Below, we overview FloDB's memory component architecture, as well as the main operations that leverage this new structure: *Get*, *Put*, *Delete*, and *Scan*.

3.1 In-memory Levels

In short, the basic idea of FloDB is to use a two-level memory component, where one level is a small, fast data structure and the second level is a larger, sorted data structure. This design allows FloDB to break the size–latency trade-off, as well as to minimize synchronization.

The first level, called the *Membuffer*, is small and fast but not necessarily sorted. The second level, called the *Memtable*, is larger, in order to capture a larger working set and thus to better mask the high I/O latency. Moreover, the Memtable keeps elements sorted, so it is directly flushable to disk. Both levels are concurrent data structures, enabling operations to proceed in parallel. Similarly to other LSMs, data flows from the smallest component (the Membuffer) down towards the largest component (the disk), as the various levels get full.

The disk-level component is not our focus and is outside the scope of this paper. Since the disk component and in-memory handling of data are orthogonal to each other in LSM key-value stores, the methods we show for in-memory optimization can be used together with any mechanism for persisting to disk. For instance, FloDB's memory component could be combined with a multithreaded compaction scheme similar to RocksDB [12], or a technique that decreases write amplification and thus obtains a better on-disk structure, as in LSM-trie [43].

3.2 Operations

We give the high-level design of FloDB's main operations. In Section 4.4 we describe a concrete implementation of this design. FloDB's two-tier memory component allows very simple basic operations (i.e., *Get*, *Put*, *Delete*) but incurs additional complexity in the case of scans.

Get. *Get* operations in FloDB do not require synchronization other than the synchronization in LSM data structures. The Membuffer is the first to be searched, then the Memtable and finally the disk. If the desired element is found at one of the levels, the read can return immediately. Obviously, it is not necessary to search in the lower levels once the element has been found, because the higher levels in the hierarchy always contain the freshest value.

Update. The *Put* and *Delete* operations are essentially identical. A delete is done by inserting a special *tombstone* value. From this point on, we will refer to both *Put* and *Delete* operations as *Update*. First, the update is attempted in the Membuffer. If there is no room in the Membuffer, then the update is made directly in the Memtable. If the key already exists in either the Membuffer or the Memtable, the corresponding value is updated in-place.

The alternative approach to in-place updates is *multi-versioning*: keeping several versions of the same key and discarding the old versions only during the compaction phase. Multi-versioning is used by all existing LSMs[1]. However, the multi-versioning approach cannot leverage the locality of skewed workloads. In fact, continually updating a single key is enough to fill up the memory component and trigger frequent flushes to disk. In contrast, with in-place updates, repeated writes to the same key do not occupy additional memory, so in-memory storage is in the order of the size of the data. Updating in-place, combined with a large memory component, allows us to capture large, write-intensive, skewed workloads efficiently, as we show in Section 5.

Scan. The main idea behind our *Scan* algorithm is to scan the Memtable and the disk, while allowing concurrent updates to complete in the Membuffer. One challenge with this approach is that a scan in the Memtable might return a stale value of a key which was still in the Membuffer when the scan started. We solve this challenge by draining the MemBuffer in the Memtable before a scan.

Another challenge is that if a scan takes a long time to complete, if scans are called often, or if there are many threads performing updates during scans, then the Membuffer that is absorbing the updates might get full. In this case, we allow the writers and scanners to proceed in parallel in the Memtable. However, if writers are allowed to naively update the Memtable during a scan A, an entry that is in A's range might be modified while A is in progress, leading to inconsistencies. We solve this problem by introducing *per-entry sequence numbers* at the level of the Memtable. In this way, A can verify if a new value in its range has been written in the Memtable since A started; if this is the case, A is invalidated and restarts. A fallback mechanism is used to ensure liveness (i.e., that a scan is not caused to restart indefinitely by writers). It is important to note that our way of using sequence numbers is different from multi-versioning mentioned above; in our algorithm, when an update of an existing key k occurs in the Memtable, k's value and sequence numbers are updated in-place.

Summary. There are several benefits of FloDB's two-level design. First, a large fraction of writes complete in the Membuffer, therefore FloDB gets the benefit of fast memory components (typically reserved for overall unsorted memory components, such as hash tables). Second, the sorted bottom component allows scans and can be directly flushed to disk. Third, the separation of levels enables writes and reads to proceed in parallel with scans. Another benefit of the two-layer hierarchy is that the overall size of the memory component can be increased to obtain better performance (in contrast to existing systems). Finally, our design allows in-place updates, while supporting consistent scans.

4. FloDB Implementation

We implement FloDB on top of Google's LevelDB key-value store [5]. The memory component of LevelDB is completely replaced with the FloDB architecture. We keep the persisting and compaction mechanisms of LevelDB[2].

In what follows, we discuss key implementation details of FloDB: (1) our data structure choices for the Memtable and Membuffer, (2) the mechanisms used to move data down from level to level, (3) our novel multi-insert operation used to streamline the flow of data between the Memtable and Membuffer, as well as (4) the implementation of the user-facing operations.

4.1 Memory Component Implementation

An important question that needs to be addressed when implementing the FloDB architecture is what would make good choices of data structures for the in-memory levels. To make writes as fast as possible, a suitable choice for the first level is a hash table [20]. As Figure 5 shows, a modern hash table can provide a throughput of over 100 Mops/s, even with billion-entry workloads. However, even though hash tables are fast, they do not sort their entries, meaning that it is not straightforward to iterate in-order over the data. For this reason, a data structure that keeps data sorted and is easily

[1] RocksDB does offer an in-place update option. However, activating this option removes point-in-time consistency guarantees for snapshots. We believe this is not a practical trade-off.

[2] The original approach in LevelDB is to keep thread-local versions and one shared version of the file-descriptor cache (*fd-cache*) in memory, acquiring a global lock to access the shared version of the fd-cache when necessary. In our preliminary tests, we found this global lock to be a major scalability bottleneck. To remove this bottleneck, as part of our in-memory optimizations, we replace the LevelDB fd-cache implementation with a more scalable, concurrent hash table [8].

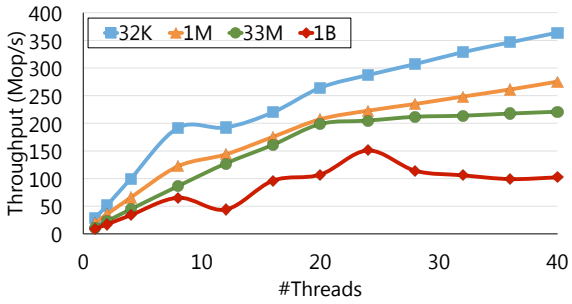

Figure 5: Performance of a concurrent hash table on a mixed read-write workload with different numbers of threads and dataset sizes (32K, 1M, 33M, 1B entries)

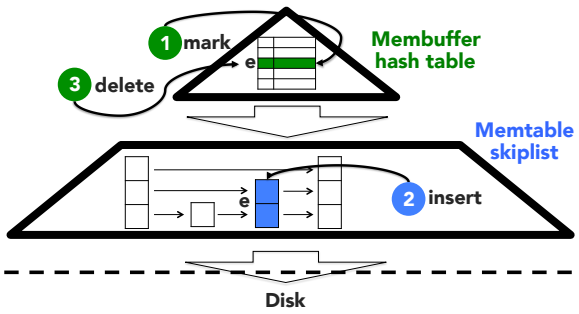

Figure 6: Draining steps.

iterable, such as the skiplist [38] already used as the memory component in traditional LSM implementations, is a good choice for the second level.

Hence, FloDB's in-memory hierarchy consists of a small high-performance concurrent hash table [8, 21] and a larger concurrent skiplist [9, 29]. The hash table stores key-value tuples. The skiplist stores key-value tuples, along with sequence numbers, which are used for the *Scan* operation.

The size ratio between the Membuffer and the Memtable presents a trade-off. On the one hand, a relatively small Membuffer will fill up faster, forcing more updates to complete in the Memtable. This is problematic because the Memtable is slower, but also because Memtable-bound updates might force more scans to restart. On the other hand, a large Membuffer will take longer to drain into the Memtable, which can delay scans.

4.2 Interaction Between Levels

Background threads. Since we do not expect data to be static (i.e., data is constantly being inserted or updated), FloDB has two mechanisms to move data across the levels of the hierarchy: *persisting* and *draining*. *Persisting* moves items from the Memtable to disk, through a dedicated background thread – an established technique in LSM implementations. Persisting is triggered when the Memtable becomes full. *Draining* moves items from the Membuffer to the Memtable, and is done by one or more dedicated background threads. Draining is a continuously ongoing process, since it is desirable to keep the Membuffer occupancy as low as possible. Indeed, writes only benefit from the two level hierarchy if they complete in the Membuffer.

Persisting. A standard technique in LSMs for persisting the memory component to disk is to make the component immutable, install a fresh, mutable component, and write the old component to disk in the background. In this way, while the old component is being persisted, writers can still proceed on the new component, and the data in the immutable component is still visible to readers. However, the switch between memory components is typically done using *locking*. FloDB has a more efficient *RCU approach* [27, 33, 34] to switching the memory components, which never blocks any updates or reads. When persisting, RCU is used first to make sure that all pending updates to the immutable Memtable have completed before allowing the background thread to copy the Memtable to disk. Second, after the Memtable is copied to disk, we use RCU to ensure that no reader threads are reading the immutable Memtable before the background thread can proceed to free it.

Draining. Draining (Figure 6) is done concurrently with updates, reads, or other drains, by one or more delegated background threads, and proceeds as follows. To move a key-value entry e from the Membuffer to the Memtable, e is first retrieved and marked in the Membuffer. This is done to ensure that no other background thread is going to attempt to also move e to the Memtable. Then, e is assigned a sequence number (obtained via an atomic increment operation) and is inserted in the Memtable by the background thread. Finally, e is removed from the Membuffer. A special case of draining occurs at the beginning of a *Scan*. In order for a *Scan* to be able to proceed only on the Memtable and disk levels, but still include any recent updates that are still in the Membuffer, a full drain of the Membuffer to the Memtable is performed at the beginning of a scan. This type of drain is done by making the current Membuffer immutable, installing a new Membuffer (using RCU), and then moving all entries from the old Membuffer to the Memtable, similarly to how the Memtable is persisted to disk.

4.3 Skiplist Multi-inserts

Figure 5 and Figure 7 show that a concurrent skiplist is roughly one to two orders of magnitude slower than a concurrent hash table of the same size. Thus, to enable a large fraction of updates to proceed directly in the hash table, we need to move items between levels as fast as possible.

Intuition. We introduce a new *multi-insert* operation for concurrent skiplists, to increase the throughput of draining threads. The intuition behind the multi-insert operation is simple. To insert n elements in the skiplist, rather than calling the *insert* operation n times, we insert these elements in

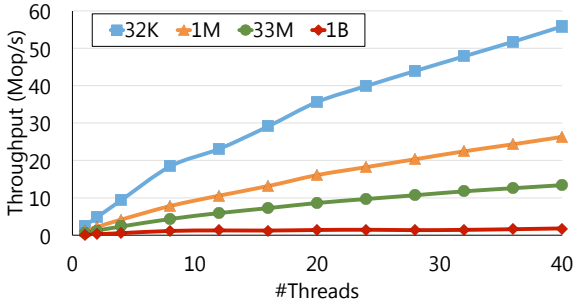

Figure 7: Performance of a concurrent skiplist on a mixed read-write workload with different numbers of threads and dataset sizes (32K, 1M, 33M, 1B entries).

only *one multi-insert* operation. The n elements are inserted in *ascending order*, using the progress already made (i.e., hops traveled) to insert the previous elements as a starting point for inserting the next elements.

Besides increasing draining throughput in FloDB, multi-inserts could also benefit other applications making use of concurrent skiplists. For instance, LSMs such as LevelDB which make use of flat combining [28] for updates (Section 2.2) could speed up their updates in the following way. The combiner thread could apply all updates in a single multi-insert operation, instead of applying them separately one after the other. Even more, several combiner threads could concurrently apply updates through multi-inserts.

Pseudocode. The pseudocode of the multi-insert operation is shown in Algorithm 1. The operation input is an array of $(key, value)$ tuples. First, the input tuples are *sorted* in ascending order. Then, for each tuple, its place in the skiplist is located, using *FindFromPreds*. *FindFromPreds* searches for the current element's spot in the skiplist starting from the predecessors of the previously inserted element (lines 5–8). If the key of the stored predecessor on the current level is larger than the key of the current element to insert, a jump can be made directly to the stored predecessor from the previous element. This is the core of the multi-insert operation, where the path-reuse idea is applied. After the spot in the skiplist is found for a tuple, the insertion of that tuple proceeds similarly to a normal insert operation [29].

Concurrency. Multi-inserts are concurrent with each other, as well as with simple inserts and reads. However, the correctness of the multi-insert relies on the fact that elements are not concurrently removed from the skiplist. This is not a problem in FloDB, by design; entries are removed from the skiplist only when they are persisted to disk. Moreover, while each insert in the multi-insert is atomic, the multi-insert itself is not linearizable, in the sense that the entire array of elements is not seen as inserted at a single point in time (i.e., intermediary state is visible).

Neighborhoods. Key proximity is a major factor in the performance of multi-insert. Intuitively, path reuse is maximized if the keys that are being multi-inserted in the skiplist will end up close together in the skiplist. Figure 8 depicts the results of an experiment comparing the throughput of simple inserts against 5-key multi-inserts, as a function of the key proximity, in an update-only test. In this experiment, a *neighborhood size* of n for the key range means that all keys in a multi-insert are at maximum $2n$ distance from each other. It can clearly be seen that the multi-insert becomes more efficient as the neighborhood size becomes smaller.

In FloDB, we create *partitions* inside the hash table, to take advantage of the performance benefits of having keys closer together in multi-inserts (the neighborhood effect).

```
1  FindFromPreds(key, preds, succs):
2  // returns true iff key was found
3  // side-effect: updates preds and succs
4   pred = root
5   for level from levelmax downto 0:
6    if (preds[level].key > pred.key):
7     pred = preds[level]
8   curr = pred.next[level]
9   while(true):
10   succ = curr.next[level]
11   if (curr.key < key):
12    pred = curr
13    curr = succ
14   else:
15    break
16  preds[level] = pred
17  succs[level] = curr
18  return (curr.key == key)
19
20 MultiInsert(keys, values):
21  sortByKey(keys, values)
22  for i from 0 to levelmax:
23   preds[i] = root
24  for each key-value pair (k,v):
25   node = new node(k,v)
26   while(true):
27    if (FindFromPreds(k, preds, succs)):
28     SWAP(succs[0].val, v)
29     break
30    else:
31     for lvl from 0 to node.toplvl:
32      node.next[lvl] = succs[lvl]
33     if (CAS(preds[0].next[0],
34             succs[0], node)):
35      for lvl from 1 to node.toplvl:
36       while(true):
37        if (CAS(preds[lvl].next[lvl],
38                succs[lvl], node)):
39         break
40        else:
41         FindFromPreds(k, preds, succs)
42      break
```

Algorithm 1: Multi-inserts

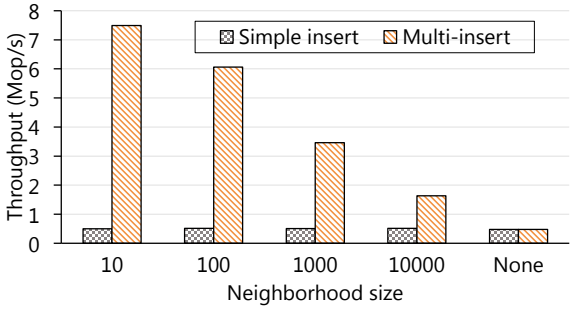

Figure 8: Performance comparison of simple insertions against multi-inserts with 5 keys/multi-insert. The initial skiplist size is 100M elements.

When a key-value tuple is inserted in FloDB, the ℓ most significant bits of the key are used to determine the partition of the hash table where the tuple should be inserted. Then, the rest of the key's bits are hashed to determine the position in the partition (i.e., the bucket). Because ℓ is a parameter, it is possible to control the size of the neighborhood easily.

While our partitioning scheme leverages the performance benefits of multi-insert, it also makes the hash table vulnerable to data skew. If there exist popular keys that have a common prefix (which is the case if the data skew concerns a certain key range), the buckets corresponding to the popular keys will become full faster than buckets corresponding to less popular keys. This in turn will lead to a smaller proportion of updates being able to complete in the Membuffer. This effect of skewed workloads is discussed in Section 5.

4.4 Implementation of FloDB Operations

Algorithm 2 presents the pseudocode for the *Get*, *Put* and *Delete* operations (to improve readability, we omit code for entering and exiting RCU critical sections).

Get. The *Get* operation simply searches for a key on every level, in the following order: the Membuffer (MBF), the immutable Membuffer (IMM_MBF), if any, the Memtable (MTB), the immutable Memtable (IMM_MTB), if any, and finally on disk. *Get* returns the first occurrence of a key that it encounters, which is guaranteed to be the freshest one, since the levels are checked in the same order as the flow of data.

Update. As mentioned before, the *Delete* operation is a *Put* with a special tombstone value, so we only need to describe the latter operation. Essentially, the *Put* operation proceeds by trying to insert a key-value pair e in the Membuffer (line 10); if the target hash table bucket for e is not full, the add to the Membuffer succeeds, and the operation returns (line 11); otherwise e is inserted in the Memtable instead (line 20). In addition, the full *Put* pseudocode in Algorithm 2 also includes mechanisms for synchronizing with the persisting thread and with concurrent scanners. First, if inserting e into the Membuffer was unsuccessful, and the *pauseWriters* flag is set, the writer helps with the draining of

```
1  Get(key):
2   for c in MBF IMM_MBF MTB IMM_MTB DISK:
3    if (c != NULL):
4     value = c.search(key)
5     if (value != not_found):
6      return value
7   return not_found
8
9  Put(key, value):
10  if (MBF.add(key, value) == success):
11   return
12  while pauseWriters:
13   if MBFNeedsDraining():
14    IMM_MBF.helpDrain()
15   else:
16    wait()
17  while MTB.size > MTB_TARGET_SIZE:
18   wait()
19  seq = globalSeqNumber.fetchAndIncrement()
20  MTB.add(key, value, seq)
21
22 Delete(key):
23  Put(key, TOMBSTONE)
```

Algorithm 2: Basic operations

the immutable Membuffer if necessary or waits for the flag to be unset otherwise (lines 12–16). As we explain below, the *pauseWriters* flag is used by the *Scan* to signal to writers that the Membuffer is being completely drained into the memtable, in preparation for a scan, and that writers should either wait or help with the draining. Second, a writer waits until there is room in the Memtable before starting its *Put* (lines 17–18). This is typically a very short wait: only the time for the persisting thread to prepare a new Memtable after the current one has become full.

Scan. The *Scan* operation takes two parameters as input: *lowKey* and *highKey*. It returns an array of all the keys and corresponding values in the data store that are between the low and high input keys.

For clarity, we separate the presentation of the scan algorithm in two parts. First we present the algorithm for a scan that can proceed in parallel with reads and writes but not with another scan. Then, we introduce the necessary additions to also allow multithreaded scans.

Single-threaded scans, multithreaded reads and writes. The pseudocode for a single-threaded *Scan* is shown in Algorithm 3 (again, we omit RCU critical section boundaries for clarity). The first step is to freeze updates to the current Membuffer and to the Memtable and to drain the current Membuffer into the Memtable. For this, we pause the background draining (line 4), signal writers to stop making updates directly in the Memtable (line 5), and wait for all ongoing Memtable writes to complete (line 9). Note that scans never completely block writers; even though writers cannot

update the Memtable between lines 5 and 13 of the scan, they can try to update the Membuffer or help with the drain. Helping ensures that the drain completes even if the scanner thread is slow, and thus reduces the time in which the writers are not allowed to update the Memtable.

Then, the current Membuffer is made immutable and a new Membuffer that is meant to absorb future updates is created (lines 6–8). The thread initiating the scan then starts draining the Membuffer to the Memtable (line 10). After the Membuffer has been drained, the scan gets a sequence number (through an atomic increment operation) (line 12). It is now safe to allow the background draining from the new Membuffer and the writers to make updates on the Memtable (lines 13–14), if necessary, because the sequence number of the scan will be used to ensure consistency. The actual scan operation (lines 15–28) starts now, first on the Memtable and the immutable Memtable (if it exists) and finally on disk. When a key that is in the scan range is encountered, its sequence number is checked. If it is lower than the scan sequence number, the key-value tuple is saved. If a key that was already saved is encountered, the value corresponding to the largest sequence number lower than the scan sequence number is kept. Else, if the key sequence number is higher than the scan sequence number, the scan restarts, because a sequence number higher than the scan sequence number may mean that the value corresponding to that key was overwritten by a concurrent operation. Finally, the array of saved keys and values is sorted and returned.

Restarts are expensive since they entail a full re-drain of the Membuffer. To avoid scans restarting arbitrarily many times in write-intensive workloads, we add a fallback mechanism, called *fallbackScan*, which is triggered when a scan was forced to restart too many times. *fallbackScan* works by blocking writers from the Memtable until it completes scanning. In our experiments (Section 5), the fallback scan is triggered rarely and does not add significant overhead.

```
1  Scan(lowKey, highKey):
2   restartCount = 0
3  restart:
4   pauseDrainingThreads = true
5   pauseWriters = true
6   IMM_MBF = MBF
7   MBF = new MemBuffer()
8   MemBufferRCUWait()
9   MemTableRCUWait()
10  IMM_MBF.drain()
11  IMM_MBF.destroy()
12  seq = globalSeqNumber.fetchAndIncrement()
13  pauseWriters = false
14  pauseDrainingThreads = false
15  results = ∅
16  for dataStructure in MTB IMM_MTB DISK:
17   iter = dataStructure.newIterator()
18   iter.seek(lowKey)
19   while (iter.isValid() and
20          iter.key < highKey):
21    if iter.seq > seq:
22     restartCount += 1
23     if restartCount < RESTART_THRESHOLD:
24      goto restart
25     else:
26      return fallbackScan(lowKey, highKey)
27    results.add(iter.key, iter.value)
28    iter.next()
29  results.sort()
30  return results
```

Algorithm 3: Scan algorithm

Multithreaded scans. If multiple threads are scanning concurrently, additional synchronization is required to avoid the situation where several threads each create a copy of the Membuffer and try to drain it. To this end, we distinguish between two types of scans: *master scan* and *piggybacking scan*. A *master scan* is a scan that starts when no other scan is concurrently running. A *piggybacking scan* is a scan that starts while some other scan is concurrently running. At any given time, only one master scan may be running. We ensure that the master scan executes lines 4–14 of Algorithm 3 and that all scans execute lines 2 and 15–30. Piggybacking scans will wait until the master scanner publishes the sequence number obtained at line 12 and then proceed to the actual scanning in lines 15–28. Note that it is possible for a piggybacking scan to start when no master scan is running if they are concurrent with another piggybacking scan that started while a master scan was running. This process can repeat itself, creating long chains of piggybacking scans that reuse the same sequence number of the master scan at the start of the chain. We limit the length of these chains through a system parameter, to avoid having a large percentage of scans that restart due to the use of a stale sequence number.

When many scans run concurrently, the scheme described above yields good performance, due to the piggybacking mechanism that spreads the overhead of draining over many scan calls and to the fact that piggybacking scans do not re-drain the Membuffer when they restart. An optimization for the low concurrency case is to also allow master scans (in addition to piggybacking scans) to reuse the sequence number of a previous master scan. This avoids fully draining the Membuffer too often.

Correctness. In terms of safety, master scans that establish a new scan sequence number are linearizable [30] with respect to updates. The linearization point is on line 7 in Algorithm 3, on the instruction that installs a new mutable Membuffer. Draining the immutable Membuffer ensures that all updates up to the linearization point are included in the scan, and the sequence number obtained in line 12 ensures that none of the updates after the linearization point are included in the scan. Piggybacking scans (and master scans that reuse an existing sequence number), however, are not

linearizable with respect to updates (but they are serializable), since they can miss updates that have occurred after their sequence number was established. If a more strict scan consistency is required at the application-level, a scan can be instructed to wait until it can establish a new sequence number, or scan piggybacking can be disabled altogether. Thus, all scans in FloDB can be linearizable with respect to updates, at the cost of performance (every scan would have to drain the Membuffer before proceeding, which is an expensive operation). In terms of liveness, all scans eventually complete, due to the *fallbackScan* mechanism which cannot be invalidated by writers and thus is guaranteed to return.

4.5 Implementation Trade-offs

As Section 5 shows, FloDB obtains better performance than its competitors across a wide range of workloads. Nonetheless, FloDB's performance does come at a cost: we trade resource usage for performance. Connecting the two in-memory levels requires at least one background thread (i.e., the draining thread) to run almost continuously in write-intensive workloads. Moreover, FloDB's scan operation presents the following limitation. Although in theory a scan on the range $(-\infty, +\infty)$ (that would return a snapshot of the entire database) could be invoked, large scans may restart many times, triggering a costly block of all writers in order to complete successfully. This scan algorithm is only practical for small and medium scans.

5. Evaluation

In this section, we evaluate FloDB, confirming the following design goals:

1. FloDB scales with the number of threads and performs well compared to the state of the art in multi-threaded read-only, write-only and mixed read-write workloads.
2. FloDB scales with memory—increasing the size of the memory component leads to better performance.
3. FloDB's in-place updates increase performance in skewed workloads.
4. The FloDB memory component, taken separately, performs better with both levels than with a single level.
5. FloDB is able to saturate the disk component throughput and is capable of better performance, if connected to a faster disk component.

5.1 Experimental Setup

We compare FloDB with three state-of-the-art LSM data stores: LevelDB [5], RocksDB [12], and HyperLevelDB [6]. Our code is publicly available at:

`https://lpd.epfl.ch/site/flodb`.

Our evaluation does not include the recent cLSM system [26], as the code for cLSM is proprietary. A preliminary public implementation based on RocksDB exists [10]. This incomplete implementation performs worse than RocksDB in all of our experiments, therefore we do not include it in our evaluation[3]. Separately from the implementation of cLSM on top of RocksDB, some of the ideas of cLSM have been integrated into RocksDB and can be enabled through parameters [13]. We include this version of RocksDB in our evaluation and label it "RocksDB/cLSM".

Unless stated otherwise, all systems are set up with a 128MB memory component and with all other parameters at their default values. For FloDB, as we discuss in Section 4, the choice of size ratio between the Memtable and the Membuffer poses a trade-off. In our experiments, we split the total size of the memory component between the Membuffer and the Memtable in $1/4$ and $3/4$ respectively (e.g., for a 128MB memory component, the Membuffer size is 32MB and the Memtable size is 96MB). This split was determined empirically to yield good results.

In all our experiments, the data set is roughly 300GB, where each key-value entry has a key size of 8B and a value size of 256B. The evaluation is performed on a 20-core machine, with two 10-core Intel Xeon E5-2680 v2 processors operating at 2.8 GHz (with up to 40 virtual cores, if hyper-threading is used). The machine has 32 KB of L1 data cache and 256 KB of L2 cache per core, 25 MB of L3 cache per processor, 256 GB of RAM, a 960 GB SSD Samsung 843T, and is running Ubuntu 16.04. For our experiments, hyper-threading was enabled and each thread was mapped to a different core whenever possible.

5.2 General Performance

We compare FloDB, LevelDB, RocksDB and HyperLevelDB using read-only (Figure 10), write-only (50% inserts, 50% deletes; Figure 9), and mixed workloads (Figure 11, Figure 12 and Figure 13). We run three types of mixed workloads. The first is balanced between updates and reads (50% reads, 25% inserts, 25% deletes). The second has only one writer thread, while the other threads in the system are reading. The third mixed workload consists of 95% updates and 5% scans, where each scan has a range of 100 keys. With these settings, 100 operations involve around 95 updates, thus 95 keys updated, and around 5 scans at 100 keys read per scan, thus 500 keys read. Therefore, in the 5% scan workload, around 84% of key accesses are reads (500 key reads out of 595 total accesses), while 16% are updates (95 key reads out of 595 total accesses). For scans, unless otherwise stated, we measure throughput as the number of keys accessed per second, as in Golan-Gueta et al. [26].

Each experiment consists of a number of threads concurrently performing operations on the data store – searching, inserting or deleting keys – continually. Each operation is chosen at random, according to the given workload proba-

[3] cLSM improves upon classic LSM implementations by replacing the memory component with a highly concurrent data structure. Intuitively, we believe FloDB would perform better than a full cLSM implementation, since in addition to highly-concurrent data structures, FloDB also has a two-tier structure that enables scalability with memory component size.

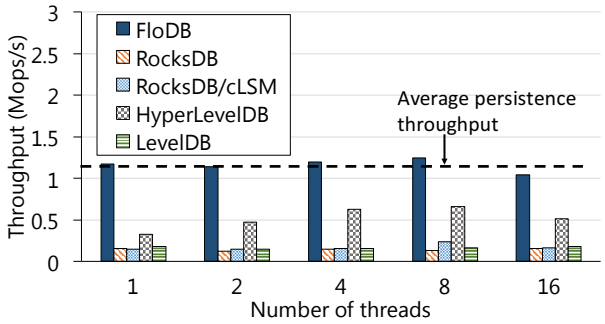

Figure 9: Write-only workload.

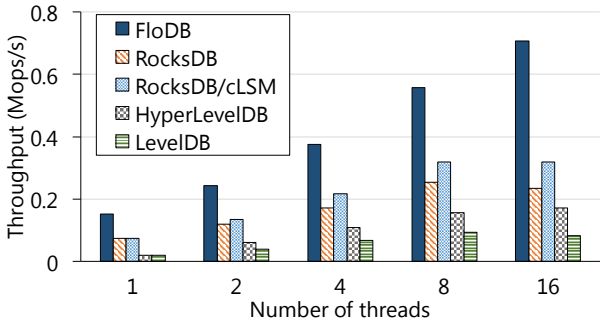

Figure 11: Mixed read-write workload.

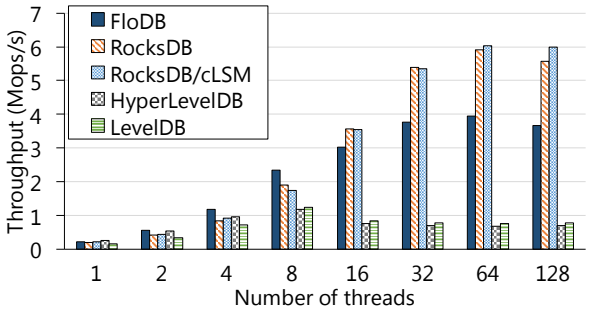

Figure 10: Read-only workload, sequential initialization.

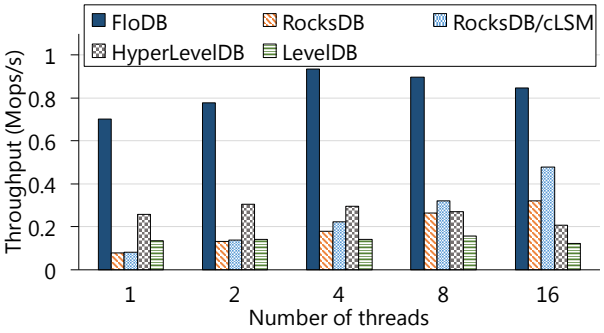

Figure 12: Mixed workload: one writer, many readers.

bility distribution, and performed on a key drawn uniformly at random, unless otherwise stated.

Before running the experiment, the database is initialized as follows. For the mixed workloads, key-value tuples covering half of the dataset are inserted in random order in the database. For the read-only workload, the same data is inserted in sorted order. The in-order initialization of the database creates an optimal structure on-disk for all four systems, which allows us to analyze performance whilst minimizing the effect of the compaction algorithm used. When initializing FloDB sequentially, the hash table is disabled to preserve the ordering of the inserted keys. After inserting the initial data, we wait until draining to disk and compactions have completed before starting the experiment. The write-only workload is run on a fresh data store, to minimize the effects of background compaction on performance.

Write-only. Figure 9 shows the throughput of the four systems as a function of the number of threads. FloDB outperforms the next best performing system, HyperLevelDB, by a factor ranging from 1.9x to 3.5x. Note the dashed line which indicates the average throughput of FloDB's persisting thread (around 1.2 millions of key-value pairs per second). FloDB manages to saturate this persistence throughput already with one thread, due to the fact that the hash table is kept empty by the draining thread, making the writer thread's inserts complete directly in the hash table. FloDB's memory component is capable of even higher throughput (as we show in Section 5.5), but, in this case, the persistence throughput is a bottleneck. RocksDB and LevelDB use a single-writer design: writes by concurrent threads are appended to a queue and applied sequentially by a single write leader. Due to this concurrency bottleneck, LevelDB and RocksDB do not scale with the number of threads. On the other hand, HyperLevelDB's higher write concurrency allows it to scale, improving upon LevelDB by up to 4x.

Read-only. Figure 10 shows the results of the read-only experiment, with sequential initialization. Due to the large dataset size and the size of the memory components, virtually all reads go to disk. LevelDB and HyperLevelDB are limited by LevelDB's concurrency control: each read operation requires two critical sections on the global mutex lock. FloDB and RocksDB both scale up to 64 threads, due to their improved parallelism of reads. It can be seen how FloDB's simplified Get operation, as well as its use of high-performance concurrent data structures lead to a significant improvement over LevelDB (by up to two orders of magnitude), even if the compaction algorithm is largely unchanged. RocksDB, as well as its cLSM variant, also have a highly concurrent Get operation and, in addition, an optimized disk component, which allow them to outperform FloDB as the thread count is increased beyond 16.

Mixed. Figure 11, Figure 12 and Figure 13 show the throughput of the five systems in the three mixed workloads. FloDB outperforms the other key-value stores in all three scenarios.

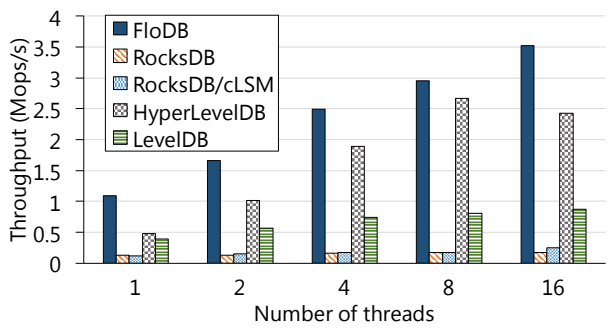

Figure 13: Mixed scan-write workload.

It can be seen that the read and write throughputs are not independent: as writes proceed, new files are created on disk (as the immutable skiplists are persisted). These files have not been fully compacted yet, thus slowing down reads that touch these files. Note that scans are faster than reads for the same number of key-value pairs accessed. This is because the scanned entries are close to each other on disk. Once the first item is located, the scan only needs to iterate within the same file or in adjacent files, as opposed to reads, which need to locate the correct file for every item.

It is interesting to observe that HyperLevelDB performs well in the scan experiment (within 43%–90% of FloDB). This is due to HyperLevelDB's efficient compaction, which in our experiments produced more than 200 times less files than RocksDB for the same number of key-value entries.

We also examine the effect of the scan ratio on the performance of FloDB. Figure 14 shows the results of five mixed scan-write experiments with 16 threads, where the ratio of scans in the workload is varied from 2% to 50%. The left part of the figure shows throughput in operations per second (broken down into scans and writes), while the right part of the figure shows throughput in keys accessed per second. As expected, with increasing scan ratios, the number of operations per second decreases. This is due to the fact that each individual scan takes longer than a write to complete. On the other hand, increasing the scan ratio actually increases the number of keys accessed per second. With fewer writes to interfere, more scans complete without the need to restart. Also, a scan contributes 100 times more to the key-throughput than a write, therefore a higher scan ratio naturally yields higher key-throughput.

Furthermore, we evaluate the scan restart rate, by measuring how often the heavyweight fallback mechanism is invoked (described in Section 4.4). We vary the scan range (from 10 keys to 10000 keys), the size of the memory component (from 128MB to 4GB) and the number of threads (from 1 to 16). In all of our experiments, the ratio of fallback scans to total completed scans was less than 1%. Thus, the scan fallback mechanism does not significantly penalize low- and medium- range scans.

5.3 Memory Component Size

In Section 2.3 we discuss the inherent inability of the classic LSM architecture, with a single-level memory component, to scale with the size of the memory component. In this experiment, we show how FloDB scales with the memory-size due to its two-tier design. We evaluate the scalability of FloDB, RocksDB, HyperLevelDB and LevelDB when the size of the memory component is increased. The benchmark uses a write-only workload with 16 threads, and the size of the memory component is varied from 128MB to 192GB.

An experiment showing steady state write performance would produce similar results to Figure 9, because of the persistence bottleneck. Since optimizing the disk component is outside the scope of this paper, to showcase scalability with the memory component, we run this experiment as a 10-second write burst (empirically chosen such that the system is not limited to its steady-state write throughput).

Figure 15 shows that FloDB outperforms the next best performing system by at least 2.3x for every memory component size, and up to an order of magnitude for the memory component sizes above 4GB. In the case of RocksDB, LevelDB and HyperLevelDB, we can see how the throughput is degrading as the memory is increased, because it becomes slower to make updates into a larger-sized skiplist. On the other hand, in FloDB, the penalty of inserting into a large-sized skiplist is avoided, by absorbing the updates in the smaller and fast hash table.

5.4 In-place Updates

We evaluate the benefits of in-place updates on a skewed workload: 2% of the dataset is accessed by 98% of operations (a common evaluation approach for skewed workloads [26]). The benchmark uses a mixed workload (50% reads, 50% updates) with 16 threads and the size of the memory component is again varied from 128MB to 192GB. As Figure 16 shows, FloDB's throughput is on average 8x higher than that of the next best-performing system, and up to 17x higher for the 128GB size. In-place updates on a large memory component enable FloDB to capture the skewed workload in the large memory component experiments. Indeed, 2% of the 300GB dataset amounts to roughly 6GB of entries that are accessed most of the time. Thus, with in-place updates, the memory components above 8GB can capture most of the workload in memory, leading to high throughput (up to 8 Mops/s). In contrast, the other systems do not provide in-place updates and cannot capture the skewed workload at any memory size. Unsurprisingly, FloDB is outperformed by the other systems at lower memory sizes. This is due to the hash table partitioning mechanism, which is sensitive to data skew, especially at small sizes (Section 4.3).

5.5 Membuffer and Multi-insert Draining

In this write-only experiment, we explore how the two-level memory component, along with the multi-insert draining en-

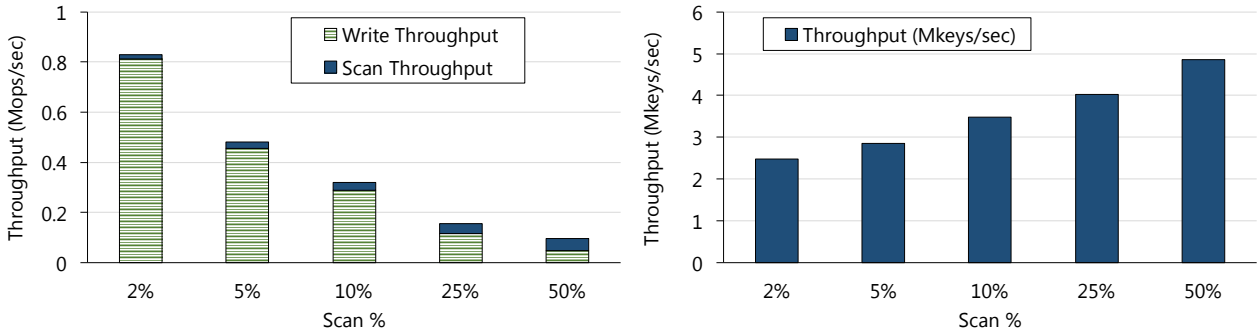

Figure 14: Impact of scan ratio on operation- and key- throughput.

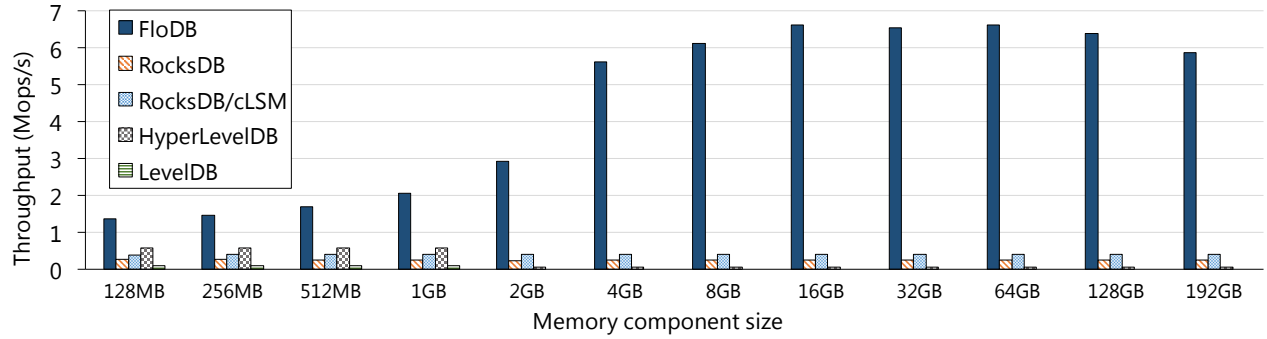

Figure 15: Write-only workload, increasing memory component size.

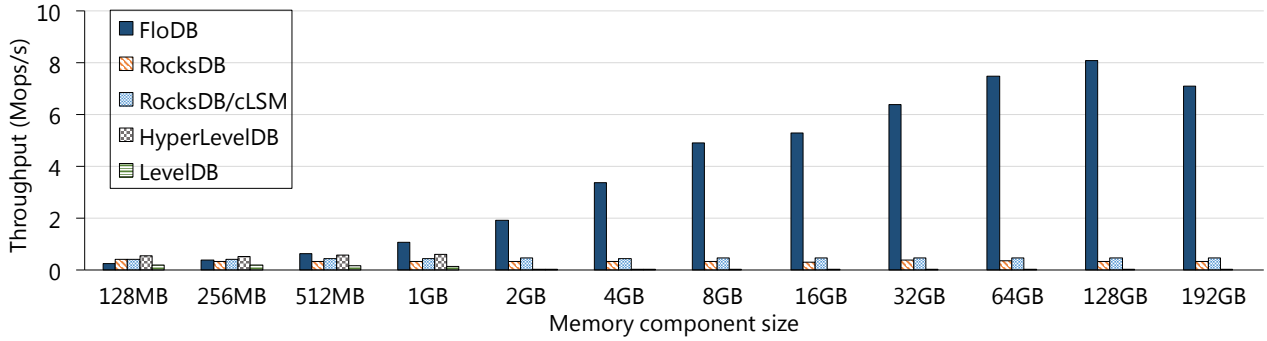

Figure 16: Mixed read-write workload, 98%-2% skew, increasing memory component size.

hance the write performance in FloDB. We compare different variants of FloDB: (1) FloDB with the Membuffer disabled and the Memtable accounting for the whole memory component size, which is the classic LSM design (*No HT*), (2) FloDB with both in-memory levels and simple insert draining (*HT, simple insert SL*), and (3) FloDB with both in-memory levels and multi-insert draining (*HT, multi-insert SL*). To exhibit the potential performance of FloDB in write-dominated scenarios, in this experiment we disable the disk persisting and compaction. Instead of being persisted to disk, the immutable Memtables are dropped so that only the throughput of the in-memory component is captured.

Figure 17 conveys the throughput as a function of the memory component size, with 8 threads for the three configurations. We also present a single-thread configuration—the first column cluster—to emphasize the benefits of multi-insert in single-writer scenarios. The proportion of entries inserted directly in the hash table are highlighted in the black boxes. We make several observations. First, the absence of the Membuffer is detrimental in FloDB: as expected, the *No HT* FloDB variant decreases in throughput as more memory is added, whereas both two-tier variants scale with memory size. Second, the proportion of updates that complete directly in the Membuffer increases with the mem-

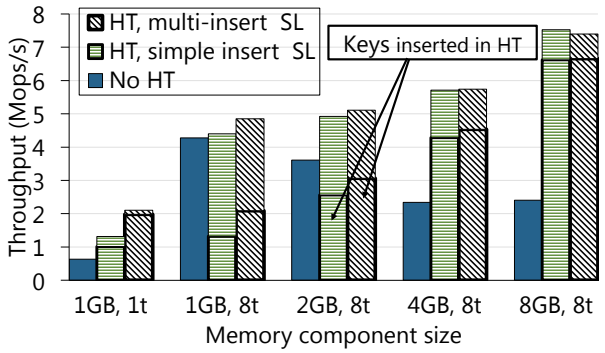

Figure 17: Effects of Membuffer and multi-insert draining. Boxed areas show proportion of direct Membuffer updates.

ory size, explaining the increase in overall throughput. Third, in the single-writer scenario, multi-inserts boost throughput by 3.1x compared to the single layer case and by 2x compared to the simple-insert FloDB variant. Finally, we see that the disk component is indeed the bottleneck: FloDB can reach a throughput of over 7 Mops/s without disk persistence, compared to 1.2 Mops/s with disk persisting (Figure 9). Therefore, future advancements in disk components will directly improve end-to-end performance.

6. Related Work

Many key-value stores have adopted the LSM model. In Section 2.2 we briefly give an overview of concurrent LSMs and their scalability bottlenecks. In this section, we discuss each of them more generally, as well as overview other related key-value stores.

LevelDB [5] is one of the earliest persistent LSM-based key-value stores and numerous modern LSMs are built on top of it [6, 12, 26]. In order to address concurrency in the memory component, LevelDB uses a global mutex lock to synchronize threads. Reads need to acquire the lock briefly at the beginning and end of the *get* operation. Writes are serialized through a *write leader* which combines [28] the updates of threads concurrent to it and applies them sequentially. The compaction process of LevelDB is single-threaded. HyperLevelDB [6] addresses the write and compaction issues in LevelDB. HyperLevelDB replaces LevelDB's sequential memory component with a concurrent one, which allows writers to apply their updates in parallel on the memory component. However, writers still need to acquire a global mutex lock at the start and end of each operation, which is a scalability bottleneck. RocksDB [12] improves upon LevelDB, by (a) carefully reducing the size and number of critical sections on the global lock and (b) caching metadata locally in order to reduce the number of synchronized accesses to global metadata. Furthermore, RocksDB introduces multithreaded disk-to-disk compaction which runs in parallel with memory-to-disk persistence.

cLSM [26] is also based on LevelDB with the design goal of increasing scalability. cLSM replaces the global mutex lock with a global reader-writer lock and uses a concurrent memory component. Thus, operations can proceed in parallel, but need to block at the start and end of each concurrent compaction. cLSM also uses RCU during the critical section when a memory component becomes immutable and a new one is installed. However, the use of RCU in cLSM blocks writers, whereas our RCU-based memory component switch never blocks readers or writers, only background threads.

Other approaches seek to improve single-threaded LSM performance, e.g., bLSM [39] proposes carefully scheduling the compaction process to reduce its negative impact on write performance. LSM-trie [43] organizes data on disk in a way that reduces write amplification and minimizes index size, thus making room for stronger Bloom filters, which in term lead to faster searches on disk. Since both bLSM and LSM-trie target the disk component, they are orthogonal to our improvements of the memory component.

Significant work has also been done in the direction of *concurrent in-memory* key-value stores, such as Masstree [32], MICA [31], KiWi [17], and MemC3 [25]. Their main goal is to have scalable performance as the number of threads is increased. Nonetheless, these systems make the assumption that all of the data they need to store fits in main memory.

7. Conclusion

LSM is the architecture of choice for many state-of-the-art key-value stores. LSMs allow writes to complete from memory, thus masking the significant latency of I/O. However, existing LSMs are not designed to benefit from the ample memory sizes of modern multicore machines. Their throughput scales neither to large memory components, nor with the number of threads, due to synchronization bottlenecks.

We address these limitations of current LSMs through FloDB, a novel two-tier memory component architecture that allows LSMs to scale with memory, as well as with the number of threads. The main idea behind FloDB is to add a buffer level on top of the classic LSM architecture, in order to hide the increased latency that comes with a large sorted memory component. We implement FloDB using highly concurrent data structures, and obtain a persistent key-value store, where reads, updates and scans can proceed in parallel. FloDB outperforms state-of-the-art LSM key-value stores, especially in write-intensive scenarios. The concepts that form the basis of FloDB are implementation independent. This opens the possibility for our proposed improvements to be combined with optimizations to other levels of the LSM architecture.

Acknowledgements

We wish to thank our shepherd, Jens Teubner, and the anonymous reviewers for their helpful comments on improving the paper. This work has been supported in part by the European Research Council (ERC) Grant 339539 (AOC) and by the Swiss National Science Foundation (SNSF) Grant 166306.

References

[1] Apache Cassandra. http://cassandra.apache.org.

[2] libcds: Library of lock-free and fine-grained algorithms. http://libcds.sourceforge.net.

[3] Intel Thread Building Blocks: a C++ template library for task parallelism. http://www.threadingbuildingblocks.org.

[4] Project voldemort: A distributed key-value storage system. http://project-voldemort.com.

[5] LevelDB, a fast and lightweight key/value database library by Google, 2005. https://github.com/google/leveldb, commit 2d0320a458d0e6a20fff46d5f80b18bfdcce7018.

[6] HyperLevelDB, a fork of LevelDB intended to meet the needs of HyperDex while remaining compatible with LevelDB, 2014. https://github.com/rescrv/HyperLevelDB, commit 40ce80173a8d72443c5f92e3c072a54ed910bab9.

[7] RocksDB hash-based memtable implementations, 2014. https://github.com/facebook/rocksdb/wiki/Hash-based-memtable-implementations.

[8] CLHT, a very fast and scalable (lock-based and lock-free) concurrent hash table with cache-line sized buckets, 2015. https://github.com/LPD-EPFL/CLHT.

[9] ASCYLIB, a concurrent-search data-structure library with over 40 implementantions of linked lists, hash tables, skip lists, binary search trees, queues, and stacks, 2016. https://github.com/LPD-EPFL/ASCYLIB.

[10] Public implementation of the cLSM algorithm, on top of RocksDB, 2016. https://github.com/guyg8/rocksdb/commits/write_throughput.

[11] Apache HBase, a distributed, scalable, big data store, 2016. http://hbase.apache.org/.

[12] RocksDB, a persistent key-value store for fast storage environments, 2016. http://rocksdb.org/, commit efd013d6d8ef3607e9c004dee047726538f0163d.

[13] RocksDB commit describing how to enable cLSM features, 2016. https://github.com/facebook/rocksdb/commit/7d87f02799bd0a8fd36df24fab5baa4968615c86.

[14] RocksDB performance benchmarks, 2016. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks.

[15] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage infrastructure behind facebook messages: Using HBase at scale. *IEEE Data Engineering Bulletin*, 35(2), 2012.

[16] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. *ACM SIGMETRICS Performance Evaluation Review*, 40(1), 2012.

[17] D. Basin, E. Bortnikov, A. Braginsky, G. Golan Gueta, E. Hillel, I. Keidar, and M. Sulamy. Brief announcement: A key-value map for massive real-time analytics. PODC 2016.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2), 2008.

[19] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2), 2008.

[20] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[21] T. David, R. Guerraoui, and V. Trigonakis. Asynchronized concurrency: The secret to scaling concurrent search data structures. ASPLOS 2015.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6), 2007.

[23] J. Duffy. *Concurrent Programming on Windows*. Microsoft Windows Development Series. Pearson Education, 2008.

[24] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. *SIGCOMM Computer Communication Review*, 42(4), 2012.

[25] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. NSDI 2013.

[26] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar. Scaling concurrent log-structured data stores. Eurosys 2015.

[27] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux. *IBM Systems Journal*, 47(2), 2008.

[28] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. SPAA 2010.

[29] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., 1st edition, 2012.

[30] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[31] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. *management*, 15(32), 2014.

[32] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. Eurosys 2012.

[33] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. PDCS 1998.

[34] P. E. McKenney, D. Sarma, A. Arcangeli, A. Kleen, O. Krieger, and R. Russell. Read copy update. Ottawa Linux Symposium 2001.

[35] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 1996.

[36] J. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1), 1989.

[37] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4), 1979.

[38] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), 1990.

[39] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. SIGMOD/PODS 2012. ACM, 2012.

[40] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 4th edition, 2014.

[41] A. Tanenbaum and H. Bos. *Modern Operating Systems*. Prentice Hall, 2014.

[42] G. Wu, X. He, and B. Eckart. An adaptive write buffer management scheme for flash-based SSDs. *ACM Transactions on Storage*, 8(1), 2012.

[43] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. USENIX ATC 2015.