Scaling Concurrent Log-Structured Data Stores

Abstract

Log-structured data stores (LSM-DSs) are widely accepted as the state-of-the-art implementation of NoSQL key-value stores. They replace random disk writes with sequential I/O, by accumulating large batches of updates in an in-memory data structure and merging it with the on-disk store in the background. While LSM-DS implementations proved to be highly successful with masking the I/O bottleneck, their main challenge is scaling up on multicore CPUs. This is nontrivial due to their often rich APIs, as well as the need to coordinate the RAM access with the background I/O.

We present cLSM, (pronounced Colosseum), an algorithm for scalable concurrency in LSM-DS, which exploits multiprocessor-friendly data structures and non-blocking synchronization in order to overcome the vertical scalability challenge. It supports a rich API, including snapshots, iterators, and general non-blocking read-modify-write operations. Our algorithm is non-blocking in all scenarios that do not involve access to disk. We implement cLSM based on the popular LevelDB key-value store, and evaluate it extensively using intensive synthetic workloads as well as ones from production Web-serving applications. Our algorithm conclusively outperforms the best-in-class LSM-DS implementations, improving throughput by 2x to 4x, and reducing latency by a similar factor. Moreover, cLSM demonstrates superior scalability both with the number of cores (successfully exploiting at least twice as many cores as the competition), and the RAM size.

1 Introduction

Over the last decade, *key-value stores* (also known as *NoSQL databases*) have become prevalent for real-time serving of Internet-scale data [3, 5]. For example, the major personalized media and ad services on the Web are backed by gigantic key-value databases that serve profile data of more than 1 billion users within milliseconds per request [**?**]. A key-value store is essentially a persistent map with atomic read and write access to data items identified by unique keys.

Modern key-value databases are commonly implemented as *Log-Structured Merge Data Stores (LSM-DSs)* [3, 5, **?**] (see Section 2); the main centerpiece behind such data stores is absorbing large batches of writes in a RAM data structure that is merged into a (substantially larger) persistent data store upon spillover. This approach masks persistent storage latencies from the end user, and increases throughput by performing I/O sequentially. The current bottleneck of such data stores is their limited in-memory concurrency, especially for writes, which, as we show in Section 4, restricts their vertical scalability on multicore servers. In the past, this was not a serious limitation, as large Web-scale servers did not harness high-end machines with many cores. Nowadays, however, servers with more cores have become cheaper, and 8-core machines with 16 hardware threads are commonplace.

Our goal in this work is to improve the scalability of state-of the art key-value stores on multi-core servers. We focus here on a single multi-core machine, and note that a single-machine data store is typically the basic building block for a distributed database that runs on multiple machines (e.g. [**?**]). We present (in Section 3) cLSM – pronounced Colosseum – a scalable concurrent LSM-DS algorithm optimized for multi-core machines. We implement cLSM in the framework of the popular LevelDB [**?**], and evaluate it extensively (in Section 4), showing 2x to 4x performance improvements.

More specifically, this paper makes the following contributions:

Non-blocking synchronization. cLSM never explicitly blocks read operations, and only blocks write operations for short periods of time before and after a batch I/O operation. Consequently, in the absence of physical disk access, all operations are non-blocking (lock-free).

Rich API. Beyond atomic put and get operations to access the key-value store, cLSM also supports consistent (serializable) snapshots and iterators, which can be used to provide range queries. The latter are provided nowadays by state-of-the-art key-value stores, and are important for many applications, e.g., serving online analytics [3] and snapshot-isolation transactions [8]). In addition, cLSM supports fully-general (customizable) non-blocking atomic read-modify-write operations. We are not aware of any existing lock-free support for such operations in today's key-value stores. Such operations are useful, e.g., for multisite update reconciliation [5, 6].

Generic algorithm. Our algorithm for supporting puts, gets, snapshots, and iterators is decoupled from any specific implementation of the LSM-DS's main building blocks: the in-memory component (a map data structure), the disk store, and the merge process that integrates the former into the latter. Only our support for atomic read-modify-write requires a specific implementation of the in-memory component as a skip-list data structure (or any such collection of sorted linked lists).

Implementation and evaluation. We implement a working prototype of cLSM based on LevelDB [**?**], a state-of-the-art key-value store. Our rich API supports all the functionality provided by LevelDB. We compare cLSM's performance to LevelDB and two additional leading open-source key-value stores, HyperLevelDB [**?**] and bLSM [**?**], on production-grade multi-core hardware. We test the systems with intensive synthetic workloads accessing large data sets as well as production workloads from major web products for content recommendation and advertising.

Results. In all of our experiments, cLSM's dominance is uncompromised. In write-only scenarios, for example, the throughput gap is 4x, and the 95th percentile latency is 60% faster. In read-dominated scenarios, the throughput and latency are 2x faster compared to the best competitor. cLSM's read-modify-write operations are at least two-fold better (in terms of both throughput and latency) than a popular implementation based on lock striping [9]. Across all workloads, cLSM exhibits superior scalability both in number of threads and in RAM size allocated to the in-memory component. Furthermore, cLSM's memory-based put and get operations are unaffected by background disk snapshot scans. Our results manifest consistently across magnetic and flash storage.

2 Background

The basic API of a key-value store includes *put* and *get* operations to store and retrieve values by their unique keys. The put operation can be used to implement an extended API, in which updating an item is cast into putting an existing key with a new value, and deleting an item is performed by putting a *deletion marker* as the key's value. Some key-value stores support in addition a *snapshot* operation which provides a consistent read-only view of the data. A snapshot allows to read the value of a specific item and to iterate over all items by the lexicographical order of their keys.

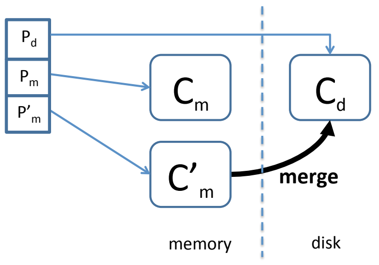


Figure 1: LSM-DS architecture: current (mutable) memory component C_m, previous (immutable) memory component $C_m^{'}$, and a disk component C_d. The merge incorporates the content of $C_m^{'}$ into C_d, while new items are added to C_m. All three components are accessible through global pointers P_m, $P_m^{'}$, P_d.

An *LSM data store* (LSM-DS) is a popular method for implementing key-value store. It organizes data in a series of components of increasing size. The first component, C_m, is an in-memory sorted map that contains most recent data. The rest of the components $C_1, \ldots, C_n$ reside on disk. In the context of this work, they are perceived as a single component, C_d. The architecture of LSM-DS is depicted in Figure 1. An additional very important building block is the *merge* procedure that incorporate the contents of the memory component into the disk. It is executed in the background as an automatic maintenance service.

In order to differentiate the interface of the internal map data structure from that of the entire LSM-DS, we refer to the corresponding functions of the in-memory data structure as `insert` and `find`. More precisely, the in-memory data structure supports at least the following two operations:

`insert(k,v)` – inserts the key-value pair (k, v) into the map. If k exists, the value associated with it is overwritten. A key is removed by inserting a $\perp$ value.

`find(k)` – returns a value v such that the map contains an item (k, v), or $\perp$ if no such value exists.

Initially, a put operation inserts an item into the main memory component C_m, and logs it in a sequential file for recovery purposes. When C_m exceeds its limit, which can be hard or soft, it is merged with component C_d, in a way reminiscent of merge sort. The items of both C_m and C_d, or sub-ranges thereof, are scanned and merged in memory buffers. The new merged component is then migrated to disk in a bulk fashion, replacing the old component.

The get operation may require going through multiple components until the key is found. In cases where get operations are applied mostly to recently inserted keys, the search is completed in C_m.

To allow put operations to be executed while rolling the merge the memory component becomes immutable, denoted as $C_m^{'}$, and a new memory component becomes available for updates (see Figure 1). The put and get operations access the components through three global pointers: pointers P_m and $P_m^{'}$ to the *current* (mutable) and *previous* (immutable) memory components, and pointer P_d to the disk component. When the merge is completed the previous memory component is discarded. Allowing multiple puts and gets to be executed in parallel is discussed in the following section.

3 cLSM Algorithm

We now present cLSM, our algorithm for concurrency support in an LSM-DS. Section 3.1 presents the basic support for concurrent get and put operations, which is generic, and can be integrated with many LSM-DS implementations. In Section 3.2, we extend the LSM-DS's functionality with snapshots and iterators, which are implemented by state-of-the-art NoSQL key-value stores [?]. This extension assumes that the in-memory data structure supports ordered iterated access with weak consistency, as numerous in-memory data structures do [?]. Finally, in Section 3.3, we go beyond the state of the art and provide general-purpose non-blocking atomic read-modify-write operations. These are supported in the context of a specific implementation of the in-memory store as a skip list data structure (or any such collection of sorted linked lists). In the next section, we describe our implementation of cLSM within LevelDB [?], whose in-memory component is indeed based on a skip list.

cLSM optimizes in-memory access in the LSM-DS, while ensuring correctness of the entire data store. Specifically, if the in-memory component's operations are atomic, then the resulting LSM-DS is also atomic. For example, the implementation of cLSM within LevelDB (described in the next section) supports atomic operations.

cLSM is almost entirely non-blocking (lock-free) in that it does not block threads during normal in-memory operation: If the underlying implementation of the in-memory component is non-blocking, then, other than inherent blocking of the LSM-DS (when get searches the disk component or put blocks due to hard size limits of the in-memory components), cLSM *never* blocks get operations and only blocks put operations during short intervals before and after a merge occurs, whilst the global pointers are being updated.

3.1 Concurrency in LSM-DS

In this section we present our support for `put` and `get` operations. We assume an existing thread-safe map data structure for the in-memory component, i.e., its operations can be executed by multiple threads concurrently. Numerous data structure implementations, (e.g., [?, ?, ?]), provide this functionality in a on-blocking and atomic manner.

The disk component and merge function are implemented in an arbitrary way. We implement our concurrency supports in two hooks, `beforeMerge` and `afterMerge`, which are executed immediately before and immediately after the merge process, respectively. The merge function returns a pointer to the new disk component, N_d, which is passed as a parameter to `afterMerge`. The global pointers P_M, P_M' to the memory components, and P_d to the disk component, are updated during `beforeMerge` and `afterMerge`.

Puts and gets access the in-memory component directly. Get operations that fail to find the requested key in the current in-memory component search the previous one (if it exists) and then the disk store. Recall that `insert` and `find` are thread-safe, so we do not need to synchronize `put` and `get` with respect to each other. However, a subtle issue introduced by the LSM design is synchronizing between the update of global pointers and normal operation.

We observe that for `get` operations, no synchronization is needed. This is because the access to each of the pointers is atomic (as it is a single-word variable), and if the pointers change after `get` has searched the component pointed by P_M (or P'), then it will search the same data structure twice, which may be inefficient, but does not violate safety. Following the pointer update, we use reference counters to avoid freeing memory components that are still being accessed by live read operations. Our experiments in the next section show that the overhead of such reference counters is negligible.

For `put` operations, a little more care is needed to avoid insertion to obsolete in-memory components. This is because such insertions may be lost in case the merge process has already traversed the section of the data structure where the data is inserted. To this end, we use a shared-exclusive lock (sometimes called readers-writer lock [?]), *Lock*, in order to synchronize between `put` operations and the global pointers' update in `beforeMerge` and `afterMerge`. The lock is acquired in shared mode during the `put` procedure, and in exclusive mode during `beforeMerge` and `afterMerge`. In order to avoid starvation of the merge process, the lock implementation should prefer exclusive locking over shared locking. Moreover, it should not block shared lockers as long as no exclusive locks are requested. Such a lock implementation is given, e.g., in [**?**].

The basic algorithm is implemented by the four procedures in Algorithm 1.

Algorithm 1 Basic cLSM algorithm.

```
1:  procedure PUT(Key $k$, Value $v$)
2:      $Lock$.lockSharedMode()
3:      $P_m$.insert($k, t$)
4:      $Lock$.unlock()
5:  procedure GET(key $k$)
6:      $v \leftarrow$ find $k$ in $P_m$, $P'_m$, or $P_d$
7:      return $v$
8:  procedure BEFOREMERGE
9:      $Lock$.lockExclusiveMode()
10:     $P'_m \leftarrow P_m$
11:     $P_m \leftarrow$ new in-memory component
12:     $Lock$.unlock()
13: procedure AFTERMERGE(DiskComp $N_d$)
14:     $Lock$.lockExclusiveMode()
15:     $P_d \leftarrow N_d$
16:     $P'_m \leftarrow \bot$
17:     $Lock$.unlock()
```

3.2 Snapshots and Iterators

We implement serializable snapshots and iterators using the common approach of multi-versioning: each key-value pair is stored in the map together with a unique, monotonically increasing, timestamp. Each snapshot and iterator is associated with a timestamp, and contains, for each key, the latest value inserted up to this timestamp. Here, we assume that `find` operations of the underlying map can return the value associated with the highest timestamp for a given key, and can subsequently iterate over all values in decreasing order of their timestamps. This can be achieved by sorting all elements in lexicographical order of the key-timestamp pair, where the timestamps' order is descending. We further assume that the map provides an iterator with the so-called *weak consistency* property, which guarantees that if an element is included in the data structure for the entire duration of an iterated scan, this value is returned by the scan. Many map data structures and data stores support such sorted access and iterators with weak consistency [**?**].

Snapshots The `getSnap` function returns a snapshot handle, from which subsequent `get` operations may read: a `get(s,k)` call returns the value associated with the key k in snapshot s. In our implementation, a snapshot handle is simply a timestamp ts, and the snapshot contains, for each key, the latest value inserted with timestamp no greater than ts. A put operation thus needs to acquire a timestamp before inserting a value into the in-memory component. This is done by atomically incrementing and reading a global counter, *timeCounter*; there are non-blocking implementations of such counters [**?**].

Determining the timestamp of a snapshot is a bit more subtle. In the absence of concurrent operations, one could simply read the current value of the global counter. However, in the presence of concurrency, this approach may lead to inconsistent snapshots, as illustrated in Figure 2. In this example, a `get` operation executed in a snapshot that reads timestamp 99 from *timeCounter* misses a value written with timestamp 98. The value is missed because the `put` operation writing it updates *timeCounter* before the `getSnap`, and inserts the value into the map after the `get` operation is complete. A later read of the same key in the same snapshot may find the value, violating consistency.

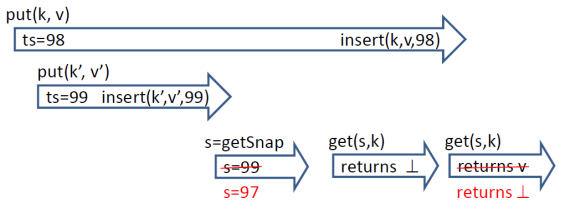


Figure 2: Potential inconsistency with naïve implementation of snapshots. Snapshot s cannot use the current value of *timeCounter*, 99, since a `get` operation pertaining to snapshot s may miss a concurrently written value with timestamp 98, and later find it. The snapshot time should instead be 97, which excludes the concurrently inserted value.

We remedy this problem by tracking timestamps that were obtained but possibly not yet written. These are kept in a set data structure, *active*, which can be implemented in a non-blocking manner (see, e.g., [**?**]). The `getSnap` operation chooses a timestamp that is earlier than all active ones. Furthermore, to overcome a race between obtaining a timestamp and inserting it into *active*, the `put` operation verifies that its chosen timestamp exceeds the latest snapshot's timestamp. The latter is tracked in the *snapTime* variable.

Our snapshot algorithm appears in Algorithm 2. The `put` and `getSnap` procedures select timestamps as explained above. The `get` operation now has an extra parameter – snapshot s; s is $\bot$ for the regular `put` operation. Note that since P_m always holds the latest inserted values, and P'_m holds values no older than those in P_d, `get` may return as soon as it finds a value older than the requested snapshot either in P_m or in P'_m. Because more than one `getSnap` operation can be executed concurrently, we have to update *snapTime* with care, to ensure that it does not move backward in time. We therefore atomically advance *snapTime* to ts (e.g., using a CAS) in line 15.

Algorithm 2 cLSM snapshot algorithm.

```
1: procedure PUT(Key $k$, Value $v$)
2:     $Lock$.lockSharedMode()
3:     $ts \leftarrow getTS()$
4:     $P_m$.insert($k, ts, v$)
5:     $active.remove(ts)$
6:     $Lock$.unlock()
7: procedure GET(Key $k$, Snapshot $s$)
8:     if $s = \perp$ then $s \leftarrow \infty$
9:     find $v$ s.t. $(k, ts, v)$ has highest $ts \leq s$ in $P_m$, $P'_m$, or $P_d$
10:    return $v$
11: procedure GETSNAP
12:    $ts \leftarrow timeCounter.get()$
13:    $ts_a \leftarrow active.findMin()$
14:    if $ts_a \neq \perp$ then $ts \leftarrow ts_a - 1$
15:    atomically assign $\max(ts, snapTime)$ to $snapTime$
16: procedure GETTS
17:    repeat
18:        $ts \leftarrow timeCounter.incAndGet()$
19:        $active.add(ts)$
20:        if $ts \leq snapTime$ then $active.remove(ts)$
21:    until $ts > snapTime$
22:    return $ts$
```

Since puts are implemented as insertions with a new timestamp, the key-value store potentially holds many versions for a given key. Old versions are not removed from the memory component, i.e., they exist at least until the component is discarded following its merge into disk. Obsolete versions are removed during a merge once they are no longer needed for any snapshot. In other words, for every key and every snapshot, the latest version of the key that does not exceed the snapshot's timestamp is kept in some component. We assume an API for removing snapshots allowing reclamation of associated values.

Iterators To implement consistent iterators, we use our snapshots and the weakly consistent iterator of the in-memory component. Since the disk component is immutable, it, too, can be scanned in a weakly consistent manner. When iteration begins, we take a snapshot. We subsequently iterate over all live components (one or two memory components and the disk component), and for each key, filter out items that have higher timestamps than the snapshot time or are older than the latest timestamp that does not exceed the snapshot time.

3.3 Atomic Read-Modify-Write

We now introduce a general read-modify-write operation, `RMW(k, f)`, which atomically applies an arbitrary function f to the current value v associated with key k and stores $f(v)$ in its place. Our implementation is efficient and avoids blocking. State-of-the-art data stores typically do not provide such general atomic read-modify-write operations. Nevertheless, such operations are extremely useful for many applications, ranging from simple vector clock update and validation to implementing full-scale transactions.

Our solution is given in the context of a specific implementation of the in-memory data store as a linked list or any collection thereof, e.g., a skip-list. Each entry in the linked list contains a key-timestamp-value tuple, and the linked list is sorted in lexicographical order.

The idea is to use optimistic concurrency control – having read v as the latest value of k, our attempt to insert $f(v)$ fails if a new value has been inserted for k after v. That is, some concurrent operation interfered between our read step and our write step. This situation is called a *conflict*. The challenge is to detect conflicts efficiently. Here, we take advantage of the fact that all updates occur in RAM, ensuring that all conflicts will be manifested in the in-memory component. We further exploit the linked list structure of this component. Namely, we observe that when attempting to insert a new node (with value $f(v)$) into a linked list that holds value v, a conflict will be manifested as a change to the successor of the node holding v. In case the list does not hold a value for k, and v is found in the disk component, a conflict is manifested as a change in either the successor or the predecessor of the node that we attempt to insert into the list. This is because if k exists in the map, an insertion of a new version for it changes its successor, whereas if does not exist, a new key can be added either before or after the intended location for k.

The algorithm for modifying a value that is found in the in-memory linked list appears in Algorithm 3. The case where (k, ts, v) is found in the disk component is similar, except that in line 5, *prev* is set to be the node holding the largest key smaller than k, and in line 6, any change in either the successor or the predecessor of the new node constitutes a conflict, and causes the search to start anew in the in-memory component.

When the data store consists of multiple linked lists, as [**?**] does, items are inserted to the linked list one at a time, from the bottom up. Only the bottom linked list is required for correctness, while the other lists ensure the logarithmic search complexity. Our implementation thus, like [**?**], first inserts the new item to the bottom list atomically using Algorithm 3. It then adds the item to each higher list using a CAS as in line 10, but with no

Algorithm 3 RMW algorithm on an in-memory linked list.

```
1: procedure RMW(Key $k$, Function $f$)
2:     Lock.lockSharedMode()
3:     repeat
4:         find $(k, ts, v)$ with highest timestamp in list
5:         set prev to this node, succ to its successor
6:         if succ holds key $k$ then continue          ▷ conflict
7:         $ts' \leftarrow getTS()$
8:         create newNode with $(k, ts', f(v))$
9:         newNode.next ← succ
10:        success ← CAS(prev.next, succ, newNode)
11:        if ¬success then active.remove(ts')          ▷ conflict
12:    until success
13:    active.remove(ts')
14:    Lock.unlock()
```

need for a new timestamp or conflict detection as in line 6.

4 Evaluation

We evaluate cLSM versus the best-in-class competitors, in terms of throughput and latency, on a variety of workloads. Our experiments include microbenchmarks as well as real web-scale application workloads. They exercise different combinations of reads, writes, read-modify-writes, and snapshot scans.

Our cLSM implementation is based on the popular open source LevelDB LSM-DS [**?**], with specific components modified to implement the cLSM algorithm. LevelDB features multiple optimizations, e.g., custom memory allocation, smart caching through memory-mapped I/O, Bloom filters [2] to speed up reads, etc. However, it is constrained to a single writer thread.

We compare cLSM with the original LevelDB as well as two other open source competitors. The first one is bLSM[1] – a single-writer prototype that capitalizes on careful scheduling of merges [17]. The other is HyperLevelDB – a LevelDB-based prototype that improves the store's performance through increased parallelism [7]. It supports multiple writers through fine grained locking.

Our hardware platform is an 8-core Xeon E5620 machine with hyperthreading (partial duplication of each core's resources). The server features 48GB of RAM, and SSD storage with RAID-5 protection[2].

Our benchmarks experiment with concurrency degrees running from 1 to 16 threads. In most cases, we configure the LSM stores to employ two in-memory components of 128MB and a disk cache of 20GB. To achieve peak write performance, the persistent log is generated in the background, with a slight lag after the original operations (standard configuration in many benchmarks).

[1]`https://github.com/sears/bLSM`

[2]The results with HDD storage are qualitatively very similar.

4.1 Real-Life Workloads

We start from evaluating the systems under a set of 20 workloads logged in a production environment at a a major Web company. Each log captures the history of operations performed by an individual backend server of a distributed key-value store. The average log size is 5GB, which translates to approximately 5M operations. The workloads are read-dominated (85% to 95% reads). The key distributions are well pronunced heavy-tail (Zipfian). Approximately 10% of the keys are only encountered once. In most settings, 20% of the keys stand for more than 80% of the requests, while 1-2% most popular stand for more than 50%.

Figure 3 depicts the overall througput evaluation, for three representative logs.

4.2 Microbenchmarks

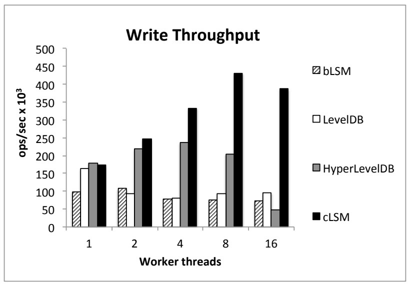
We perform a set of microbenchmarks that exercise the systems in a variety of controlled settings. Our experiment harnesses a 150GB dataset – 100x the size of the collection used to compare HyperLevelDB to LevelDB in the only publicly available microbenchmark[3]. In this context, the key-value pairs have 16-byte keys, and the value size is 1KB. Each worker thread drives operations spanning 1M keys.

Write performance. We start with exploring a 100%-write workload. The keys are drawn uniformly at random from the entire range. (Different distributions lead to identical results – recall that the write performance in LSM stores is locality-insensitive.)

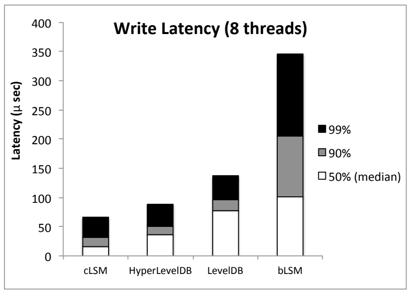
Figure 4a depicts the results in terms of throughput. In this context, LevelDB, HyperLevelDB and cLSM start from approximately the same point, however they behave differently as the workload scales. LevelDB is completely bounded by its single-writer architecture, and therefore does not scale at all. On the other hand, HyperLevelDB achieves a 33% throughput gain with 4 workers, and deteriorates beyond that point. cLSM's throughput scales 2.5x and becomes saturated at 8 threads. Its peak rate exceeds 430K writes/sec, in contrast with 240K for HyperLevelDB and 160K for LevelDB. Note that while cLSM 's peak performance is achieved with 8 threads. This is due to (1) nonlinear overhead of atomic hardware instructions, and (2) non-equivalence of hyperthreading to independent CPU cores.

Figure 4b compares the systems in terms of latency (the median, the 90% and the 99%), for 8 threads. cLSM's superiority is very clear – e.g., with 8 threads its

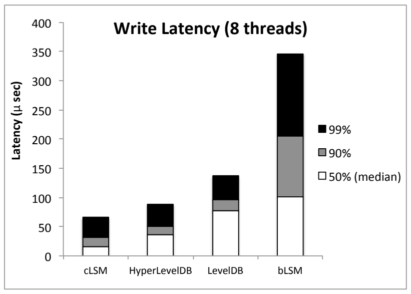
[3]`http://hyperdex.org/performance/leveldb`



(a) Throughput



(b) Latency



(c) Latency

Figure 3: **Real workload performance.**

median write latency is 20μs, whereas the 90's percentile is below XXX μs. The competitors lag far behind. The single-writer implementations are most vulnerable to latency variability – e.g., for LevelDB and bLSM the 90% latencies exceed XXX μs and XXX μs, respectively.

We finally measure the overhead of snapshot support in cLSM's writes (Section 3), by benchmarking with the auxiliary code stripped. The results show up to 15% of throughput boost versus the complete implementation.

Read performance. We turn to evaluating performance in a 100%-read scenario. In this context, uniformly distributed reads would not be indicative, since the system would spend most of the time in disk seeks, devoiding the concurrency control optimizations of any meaning. Hence, we employ a skewed distribution that generates a CPU-intensive workload. 90% of the keys are selected randomly from the "popular" blocks that comprise 10% of the database. The rest are drawn u.a.r. from the whole range. This workload is both dispersed enough and amenable to caching. In the sequel, all the experiments exercise this distribution.

Figure 5a demonstrates the throughput scalability. LevelDB and HyperLevelDB exhibit similar performance (which is explainable, since they only differ in write and merge optimizations). Neither scales beyond 8 threads, reflecting the limitations of LevelDB's concurrency control. On the other hand, in cLSM the reads bear no overhead at all, and hence, they scale all the way to 16 threads, speeding the throughput up 8x. The peak throughput is 1.25M reads/sec – almost twice as much as the peak competitor rate. At this concurrency level, cLSM exhibits a median latency of just 13 μs, and the 90% latency of XXX μs (Figure 5b). The 99% latencies are similar among all the data stores (two orders of magnitude bigger) since those reads are I/O-bound.

Read-Modify-Write performance. We now explore the performance of RMW operations (put-if-absent flavor). Neither of cLSM's competitors supports the RMW abstraction originally. To establish a comparison baseline, we augment LevelDB with a textbook RMW implementation based on lock striping [9]. The algorithm protects each RMW and write operation with an exclusive granular lock to the accessed key, thus treating it as a mini-transaction. The basic read and write implementations remain the same.

We compare the lock-striped LevelDB with cLSM. The first workload under study is comprised solely of RMW operations. In this context, cLSM scales to almost 400K operations/sec – a 2.5x throughput gain compared to the standard implementation (Figure 6a). This volume is almost identical to the peak write load. Similarly to the mixed read/write workload, this is explained by cLSM's high CPU utilization, which allows scaling to 16 threads. The median latency with that configuration is 17 μs, versus LevelDB's 36 μs (Figure 6b).

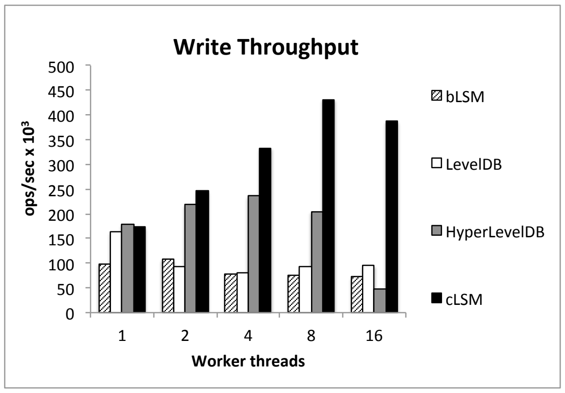
Mixed workloads. Figure 7 depicts the throughput achieved by the different systems under a 1:1 read-write mix. Due to its single-writer design, the original LevelDB fails to scale, despite that the writes are now only 50% of the workload. HyperLevelDB slightly improves upon that result. However, cLSM fully exploits the software parallelism, by scaling beyond 730K operations/sec with 16 workers. Note that under cLSM the reads and the writes scale independently, whereas in the other solutions the writes impede the reads' progress.

Figure 8 compares LevelDB's and cLSM's utilization of its memory component, under the same workload, with 8 working threads. LevelDB performs nearly the same for buffer sizes beyond 16MB, whereas cLSM exhibits diminishing returns only beyond 256MB. **TBD explain!**

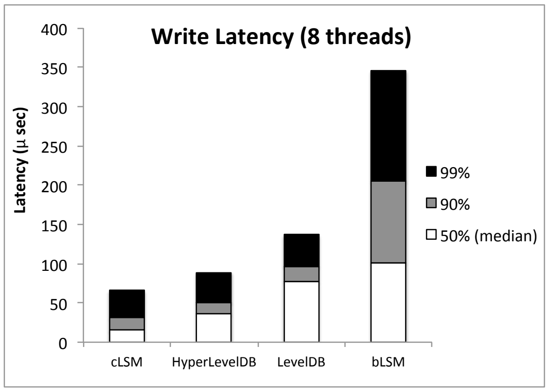
TBD reads, writes and scans!

5 Related Work

The *logarithmic method* was initially proposed in [1] as a way to efficiently transform static search structures into dynamic ones. A *binomial list* structure stored a sequence of sorted arrays, called *runs* each of size of power of two. Inserting an element triggers a cascaded series

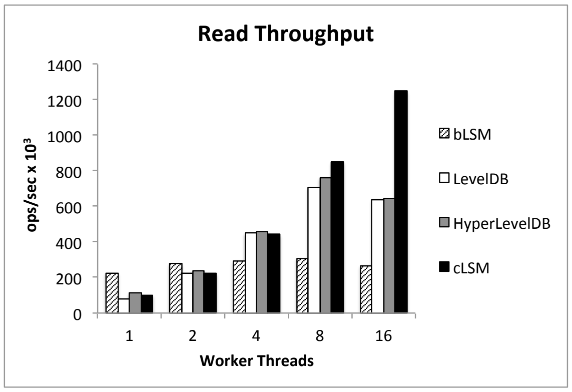


(a) Throughput

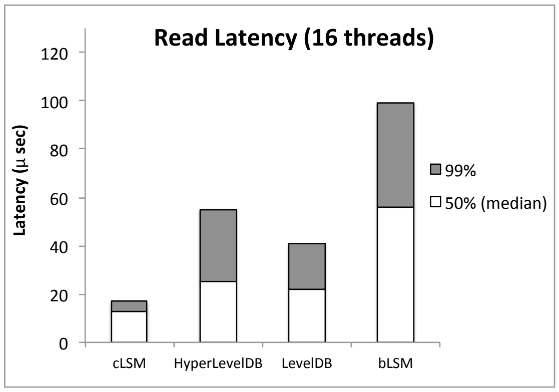


(b) Latency

Figure 4: **Write performance.**



(a) Throughput



(b) Latency

Figure 5: **Read performance.**

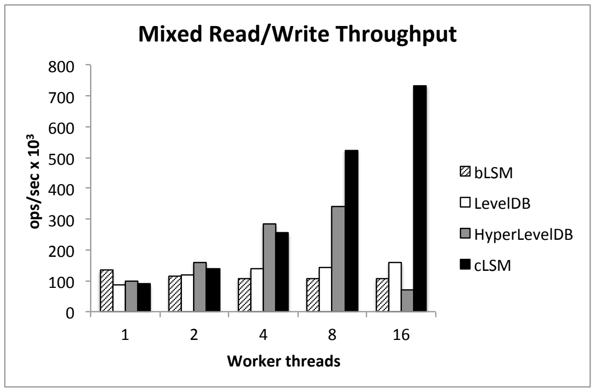


Figure 7: **Mixed reads and writes - throughput.**

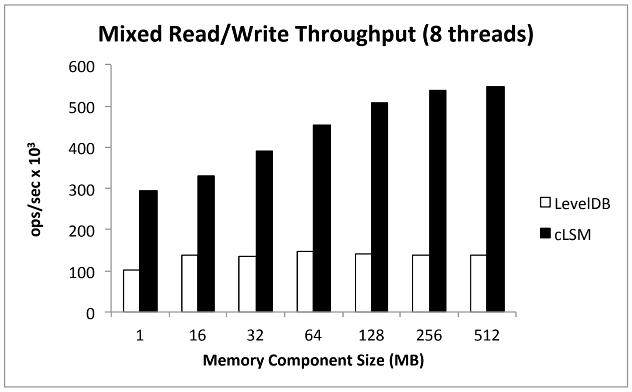
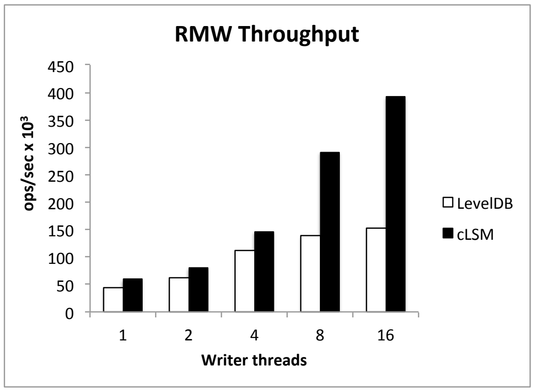


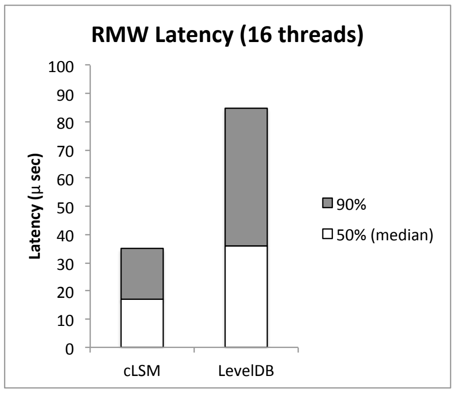
Figure 8: **Mixed reads and writes - memory component utilization.**

of merge-sorting of adjacent runs. Searching an element is done by applying a binary search on the runs starting with the smallest run until the element is found.

This method inspired the original work on *LSM-trees* [14] and its variant for multi-versioned data stores [13]. LSM-trees aim to provide low-cost indexing



(a) Throughput



(b) Latency

Figure 6: **Read-modify-write (RMW) performance.**

for key-value stores with high rate of put operations, by deferring in-place random writes and turning them into sequential writes. The LSM-tree indexing approach assumes B^+-tree like structures as its disk components, and as the main memory component an efficient key-lookup structure similar to a (2-3)-tree or—more common in recent implementations, e.g., in LevelDB—a skip-list [15].

Several different approaches for optimizing the performance of the logarithmic structure in key-value stores have been proposed in recent years.

One approach suggests adopting a new tree-indexing data structure, *FD-tree* [12], to better facilitate the properties of contemporary flash disks and solid state drives (SSDs). Like the runs in binomial lists, and components in LSM-trees, FD-trees maintain multiple *levels* with cross-level pointers. This approach applies the *fractional cascading* [4] technique to speed up search in the logarithmic structure.

A follow-up work [18] further refined FD-trees to support concurrency. The concurrency control scheme proposed in this work (*FD+FC*) allows concurrent read and writes during ongoing index reorganizations.

With a similar goal of exploiting flash storage as well as the caches of modern multi-cores processors, Levandoski et al. [11] present a new form of a B-tree, called *Bw-tree*, to be used as the index of a persistent key-value store. The implementation is lock-free allowing for better scalability (throughput), it also avoids cache line invalidation thus improving cache performance (latency). Instead of locks their implementation, which bares similarity to B-link design [10], uses CAS instructions, therefore it blocks only rarely, when fetching a page from disk. At its storage layer Bw-tree uses log structuring [16].

Neither one of the new approaches support atomic scans nor an atomic RMW operation. In addition these algorithm build upon a specific data structure as a main memory component, whereas in our work to support the basic API any implementation of a concurrent sorted map is applicable.

A different approach is presented by bLSM [17]. This work introduces a new merge scheduler, which bounds the time a merge can block write operations. In addition, bLSM leverages Bloom filters [2] to reduce the number of seeks (random reads). This results in accelerating searches of items that are not stored in the database – mostly important when invoking an *insert if not exists* operation. As bLSM optimizations focus on the merging process and disk access, it is orthogonal to our work on memory optimizations. We believe the two approaches can be combined.

6 Discussion

Acknowledgements

References

[1] J. L. Bentley and J. B. Saxe. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

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

[3] F. Chang et al. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[4] B. Chazelle and L. J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(2):133–162, 1986.

[5] 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. In *VLDB*, 2008.

[6] 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. In *SOSP*, pages 205–220, 2007.

[7] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: a distributed, searchable key-value store. In *SIGCOMM 2012*, pages 25–36, 2012.

[8] D. G. Ferro, F. P. Junqueira, B. Reed, and M. Yabandeh. Lock-free transactional support for distributed data stores. In *SOSP (poster session)*, 2012.

[9] J. Gray and A. Reuters. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

[11] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering (ICDE)*, pages 302–313, 2013.

[12] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree indexing on solid state drives. *Proc. VLDB Endow.*, 3(1-2):1195–1206, Sept. 2010.

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

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

[15] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[16] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[17] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *SIGMOD 2012*, pages 217–228, 2012.

[18] R. Thonangi, S. Babu, and J. Yung. A practical concurrent index for solid-state drives. In *CIKM 2012*, pages 1332–1341, 2012.