Scaling Concurrent Log-Structured Data Stores

Guy Golan-Gueta
Yahoo Labs
Haifa, Israel
ggolan@yahoo-inc.com

Edward Bortnikov
Yahoo Labs
Haifa, Israel
ebortnik@yahoo-inc.com

Eshcar Hillel
Yahoo Labs
Haifa, Israel
eshcar@yahoo-inc.com

Idit Keidar
Technion, Yahoo Labs
Haifa, Israel
idish@ee.technion.ac.il

Abstract

Log-structured data stores (LSM-DSs) are widely accepted as the state-of-the-art implementation of 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 at masking the I/O bottleneck, scaling them up on multicore CPUs remains a challenge. 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, an algorithm for scalable concurrency in LSM-DS, which exploits multiprocessor-friendly data structures and non-blocking synchronization. cLSM supports a rich API, including consistent snapshot scans and general non-blocking read-modify-write operations.

We implement cLSM based on the popular LevelDB key-value store, and evaluate it using intensive synthetic workloads as well as ones from production web-serving applications. Our algorithm outperforms state of the art LSM-DS implementations, improving throughput by 1.5x to 2.5x. Moreover, cLSM demonstrates superior scalability with the number of cores (successfully exploiting twice as many cores as the competition).

EuroSys '15, April 21–24, 2015, Bordeaux, France.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

Over the last decade, *key-value stores* have become prevalent for real-time serving of Internet-scale data [16]. Gigantic stores managing billions of items serve Web search indexing [33], messaging [12], personalized media, and advertising [18]. A key-value store is essentially a persistent map with atomic get and put operations used to access data items identified by unique keys. Modern stores also support consistent snapshot scans and range queries for online analytics.

In write-intensive environments, key-value stores are commonly implemented as *Log-Structured Merge Data Stores (LSM-DSs)* [2, 4, 8, 16, 18, 25, 36] (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. A major bottleneck of such data stores is their limited in-memory concurrency, which, as we show in Section 5, 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 multicore hardware. Nowadays, however, servers with more cores have become cheaper, and 16-core machines commonplace in production settings.

Our goal in this work is to improve the scalability of state-of the art key-value stores on multicore servers. We focus on a data store that runs on a single multicore machine, which is often the basic building block for a distributed database that runs on multiple machines (e.g., [16, 18]). Although it is possible to scale up by further partitioning the data and running multiple LSM-DS's on the same machine, there are significant advantages to consolidation [11]; see more detailed discussion in Section 2. We therefore strive to scale up a single LSM-DS by maximizing its parallelism.

We present (in Section 3) cLSM, a scalable LSM-DS algorithm optimized for multi-core machines. We implement cLSM in the framework of the popular LevelDB [4] library (Section 4), and evaluate it extensively (Section 5), showing better scalability and 1.5x to 2.5x performance improvements over the state-of-the art.

Contributions

This paper makes the following contributions:

Non-blocking synchronization. cLSM overcomes the scalability bottlenecks incurred in previous works [4, 21] by eliminating blocking during normal operation. It never explicitly blocks get operations, and only blocks puts for short periods of time before and after batch I/Os.

Rich API. Beyond atomic put and get operations, cLSM also supports consistent snapshot scans, which can be used to provide range queries. These are important for applications such as online analytics [16], and multi-object transactions [41]. In addition, cLSM supports fully-general non-blocking atomic *read-modify-write* (RMW) 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 [18, 19].

Generic algorithm. Our algorithm for supporting puts, gets, snapshot scans, and range queries is decoupled from any specific implementation of the LSM-DS's main building blocks, namely 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. This allows one to readily benefit from numerous optimizations of other components (e.g., disk management [8]) which are orthogonal to our contribution.

Implementation. We implement a working prototype of cLSM based on LevelDB [4], a state-of-the-art key-value store. Our implementation supports the full functionality of LevelDB and inherits its core modules (including disk and cache management), and therefore benefits from the same optimizations.

Evaluation. We compare cLSM's performance to LevelDB and three additional open-source key-value stores, HyperLevelDB [21], bLSM [36], and RocksDB [8], on production-grade multi-core hardware. We evaluate the systems under large-scale intensive synthetic workloads as well as production workloads from a web-scale system serving personalized content and ad recommendation products.

In our experiments, cLSM achieves performance improvements ranging between 1.5x and 2.5x over the best competitor, on a variety of workloads. cLSM's RMW operations are also twice as fast as a popular implementation based on lock striping [22]. Furthermore, cLSM exhibits superior scalability, successfully utilizing at least twice as many threads, and also benefits more from a larger RAM allocation to the in-memory component.

2. Architecture Principles

We overview our design choices, as motivated by today's leading key-value store implementations. We discuss their API, approaches to scaling them, and the LSM approach to data management.

2.1 Data Model and API

In key-value stores [2, 16, 18], the data is comprised of items (*rows*) identified by unique keys. A row value is a (sparse) bag of attributes called *columns*. The internal structure of data items is largely opaque for the rest of our discussion.

The basic API of a key-value store includes *put* and *get* operations to store and retrieve values by their keys. Updating an item is cast into putting an existing key with a new value, and deleting one is performed by putting a *deletion marker*, $\perp$, as the key's value.

To cater to the demands of online analytics applications (e.g., [33]), key-value stores typically support *snapshot scans*, which provide consistent read-only views of the data. A scan allows the user to acquire a snapshot of the data (*getSnap*), from which the user can iterate over items in lexicographical order of their keys by applying *next* operations.

Geo-replication scenarios drive the need to reconcile conflicting replicas. This is often done through vector clocks [19], which require the key-value store to support conditional updates, namely, atomic read-modify-write operations.

2.2 Scalability in Distributed Key-Value Stores

Distributed key-value stores achieve scalability by sharding data into units called *partitions* (also referred to as *tablets* [16, 18] or *regions* [2]). Partitioning provides *horizontal scalability* – stretching the service across multiple servers. Nevertheless, there are penalties associated with having many partitions, as argued in [11]: First, the data store's consistent snapshot scans do not span multiple partitions. Analytics applications that require large consistent scans are forced to use costly transactions across shards. Second, this requires a system-level mechanism for managing partitions [2, 16, 18], whose meta-data size depends on the number of partitions, and can become a scalability bottleneck.

The complementary approach of increasing the serving capacity of each individual partition is called *vertical scalability*. First, this necessitates optimizing the speed of I/O-bound operations. The leading approach to do so, especially in write-intensive settings, is LSM (discussed in Section 2.3), which effectively eliminates the disk bottleneck. Once this is achieved, the rate of in-memory operations becomes paramount (as we show in Section 5). Increasing this rate is the challenge we focus on in this paper.

We argue here that we can improve performance while reducing the number of partitions, which in turn allows for

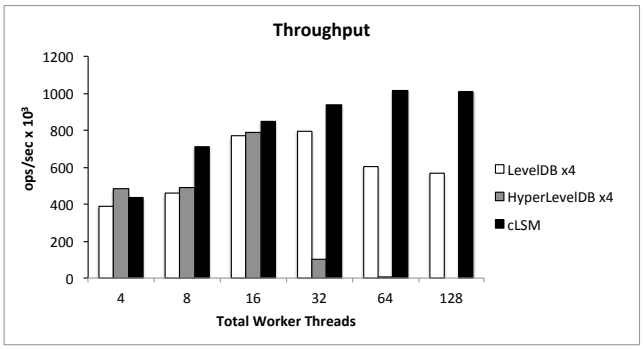


Figure 1: **Comparing two approaches to scalability with production workload. The resource-isolated configuration exercises LevelDB and HyperLevelDB with 4 separate partitions, whereas the resource-shared configuration evaluates cLSM with one big partition.**

larger snapshot scans and reduces meta-data size. To illustrate this point, Figure 1 shows sample results from our experiments (the experiment setup is detailed in Section 5). In this example, we evaluate cLSM with one big partition versus LevelDB and HyperLevelDB with four small partitions, where each small partition's workload is based on a distinct production log, and the big partition is the union thereof. Each of the small partitions is served by a dedicated one quarter of the thread pool (resource separation), whereas the big partition is served by all worker threads (resource sharing). We see that cLSM's improved concurrency control scales better than partitioning, achieving a peak throughput of above 1 million operations/sec – approximately 25% above the competition.

2.3 Log-Structured Merge

Disk access is a principal bottleneck in storage systems, and remains a bottleneck even with today's SSDs [10, 38, 40]. Since reads are often effectively masked by caching, significant emphasis is placed on improving write throughput and latency [38]. It is therefore not surprising that log-structured merge solutions [31], which batch writes in memory and merge them with on-disk storage in the background, have become the de facto choice for today's leading key-value stores [4, 8, 12, 16, 21, 36].

An LSM data store organizes data in a series of components of increasing sizes, as illustrated in Figure 2a. 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. For simplicity, in the context of this work, they are perceived as a single component, C_d. An additional important building block is the *merge* procedure, (sometimes called *compaction*), which incorporates the contents of the memory component into the disk, and the contents of each component into the next one.

A put operation inserts an item into the main memory component C_m, and logs it in a sequential file for recovery purposes. Logging can be configured to be synchronous (blocking) or asynchronous (non-blocking). The common default is asynchronous logging, which avoids waiting for disk access, at the risk of losing some recent writes in case of a crash.

When C_m reaches its size 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, are scanned and merged. The new merged component is then migrated to disk in bulk fashion, replacing the old component. When considering multiple disk components, C_m is merged with component C_1. Similarly, once a disk component C_i becomes full its data is migrated to the next component C_{i+1}. Component merges are executed in the background as an automatic maintenance service.

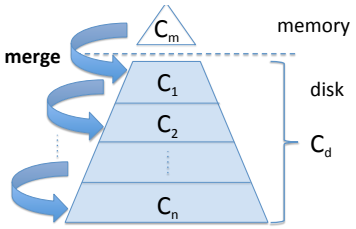
The get operation may require going through multiple components until the key is found. But when get operations are applied mostly to recently inserted keys, the search is completed in C_m. Moreover, the disk component utilizes a large RAM cache. Thus, in workloads that exhibit locality, most requests that do access C_d are satisfied from RAM as well.

During a merge, the memory component becomes immutable, at which point it is denoted as C'_m. To allow put operations to be executed while rolling the merge, a new memory component C_m then becomes available for updates (see Figure 2b). 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 complete, the previous memory component is discarded. Allowing multiple puts and gets to be executed in parallel is discussed in the sequel.

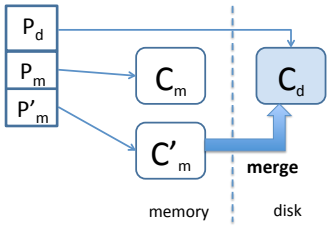
3. cLSM Algorithm

We now present cLSM, our algorithm for concurrency support in an LSM-DS. Section 3.1 presents our basic approach for providing scalable concurrent `get` and `put` operations; this solution is generic, and can be integrated with many LSM-DS implementations. In Section 3.2, we extend the functionality with snapshot scans, which are implemented in state-of-the-art key-value stores (e.g., [4, 21]). This extension assumes that the in-memory data structure supports ordered iterated access with weak consistency (explained below), as various in-memory data structures do (e.g., [1, 7, 15]). Finally, in Section 3.3, we 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 collection of sorted linked lists).

cLSM optimizes in-memory access in the LSM-DS, while ensuring correctness of the entire data store. Specifi-



(a) LSM-DS consists of a small memory component, and a large disk component comprised of a series of components of increasing sizes.



(b) Global pointers P_m to current (mutable) memory component C_m, P'_m to previous (immutable) memory component C'_m, and P_d to disk component C_d. Merge incorporates C'_m into C_d, while new items are added to C_m.

Figure 2: **LSM-DS architecture.**

cally, if the in-memory component's operations ensure serializability [32], then the same is guaranteed by the resulting LSM-DS.

3.1 Put and Get Operations

We assume a 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., see [1, 20, 23]), provide this functionality in a non-blocking and atomic manner. 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`:

`insert(k,v)` – inserts the key-value pair (k, v) into the map. If k exists, the value associated with it is overwritten.

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

The disk component and merge function are implemented in an arbitrary way.

We implement our concurrency support 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, synchronizing between the update of global pointers and normal operation is a subtle issue.

We observe that for `get` operations, no blocking synchronization is needed. This is because the access to each of the pointers is atomic (as it is a single-word variable). The order in which components are traversed in search of a key follows the direction in which the data flows (from P_m to P'_m and from there to P_d) and is the opposite of the order in which the pointers are updated in `beforeMerge` and `afterMerge`. Therefore, if the pointers change after `get` has searched the component pointed by P_m or P'_m, then it will search the same data twice, which may be inefficient, but does not violate safety.

We use reference counters to avoid freeing a memory component while it is being read. In addition, we apply an RCU-like mechanism to protect the pointers to memory components from being switched while an operation is in the middle of the (short) critical section in which the pointer is read and its reference counter is increased. As we only use reference counters per component (and not per row), their overhead 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 [20]), *Lock*, in order to synchronize between `put` operations and the global pointers' update in `beforeMerge` and `afterMerge`. (Such a lock does not block shared lockers as long as no exclusive locks are requested.) 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. Such a lock implementation is given, e.g., in [1].

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

3.2 Snapshot Scans

We implement serializable snapshot scans using the common approach of multi-versioning: each key-value pair is stored in the map together with a unique, monotonically increasing, timestamp. That is, the elements stored in the

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 ← find k in P_m, P'_m, or P_d, in this order
7:      return v
8:  procedure BEFOREMERGE
9:      Lock.lockExclusiveMode()
10:     P'_m ← P_m
11:     P_m ← new in-memory component
12:     Lock.unlock()
13: procedure AFTERMERGE(DiskComp N_d)
14:     Lock.lockExclusiveMode()
15:     P_d ← N_d
16:     P'_m ← ⊥
17:     Lock.unlock()
```

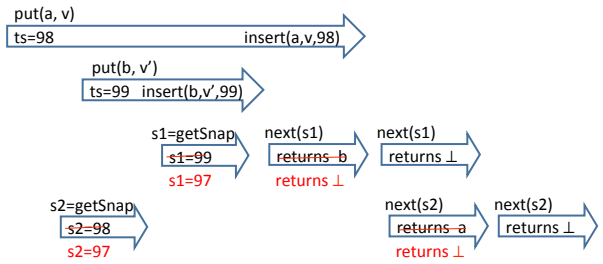


Figure 3: Snapshots s_1 and s_2 cannot use the current values of *timeCounter*, 99 and 98 respectively, since a `next` operation pertaining to snapshot s_1 may miss the concurrently written key a with timestamp 98, while a `next` operation pertaining to snapshot s_2 filters out the key b with timestamp 99. The snapshot time should instead be 97, which excludes the concurrently inserted keys.

underlying map are now key-timestamp-value triples. The timestamps are internal, and are not exposed to the LSM-DS's application.

Here, we assume the underlying map is sorted in lexicographical order of the key-timestamp pair. Thus, `find` operations can return the value associated with the highest timestamp for a given key. We further assume that the underlying map provides iterators with the so-called *weak consistency* property, which guarantees that if an element is included in the data structure for the entire duration of a complete snapshot scan, this element is returned by the scan. Several map data structures and data stores support such sorted access and iterators with weak consistency (see [7, 15]).

To support multi-versioning, a `put` operation acquires 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 (e.g., see [20]). A `get` operation now returns the highest timestamped value for the given key.

Our support for snapshots and full scans thereof is explained in Section 3.2.1. We discuss other snapshot-based operations (like range queries) in Section 3.2.2.

3.2.1 Snapshot Management Mechanism

A snapshot is associated with a timestamp, and contains, for each key, the latest value updated up to this timestamp. Thus, although a snapshot scan spans multiple operations, it reflects the state of the data at a unique point in time.

The `getSnap` operation returns a snapshot handle s, over which subsequent operations may iterate. In cLSM, a snapshot handle is simply a timestamp ts. A scan iterates over all live components (one or two memory components and the disk component) and filters out items that do not belong to the snapshot: for each key k, the `next` operation filters out items that have higher timestamps than the snapshot time, or are older than the latest timestamp (of key k) that does not exceed the snapshot time. When there are no more items in the snapshot, `next` returns $\perp$.

Our snapshot management algorithm appears in Algorithm 2. Determining the timestamp of a snapshot is a bit 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 scans, as illustrated in Figure 3. In this example, `next` operations executed in snapshot s_2, which reads 98 from *timeCounter*, filter out a key written with timestamp 99, while `next` operations executed in snapshot s_1, which reads timestamp 99, read this key, but miss a key written with timestamp 98. The latter is missed because the `put` operation writing it updates *timeCounter* before the `getSnap` operation, and inserts the key into the underlying map after the `next` operation is completed. This violates serializability as there is no way to serialize the two scans.

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. The `getSnap` operation chooses a timestamp that is earlier than all active ones. In the above example, since both 98 and 99 are active at the time s_1 and s_2 are invoked, they choose 97 as their snapshot time.

Note that a race can be introduced between obtaining a timestamp and inserting it into *Active* as depicted in Figure 4. In this example, a `put` operation reads timestamp 98 from *timeCounter*, and before it updates the *Active* set to include it, a `getSnap` operation reads timestamp 98 from *timeCounter* and finds the *Active* set empty. The snapshot timestamp is therefore set to 98. The value later written by the `put` operation is not filtered out by the scan, which may lead to inconsistencies, as in the previous example. To overcome this race, the `put` operation verifies that its chosen timestamp exceeds the latest snapshot's timestamp (tracked

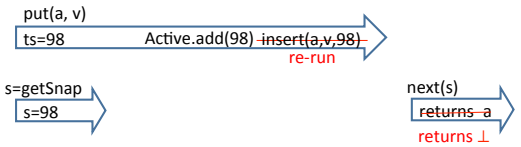


Figure 4: The put operation cannot use the value 98 since a snapshot operation already assumes there are no active put operations before timestamp 99. Using the timestamp 98 may lead to the problem depicted in Figure 3. The put operation should instead acquire a new timestamp.

in the *snapTime* variable), and re-starts if it does not, while `getSnap` waits until all active `put` operations have timestamps greater than *snapTime*.

We note that our scan is serializable but not linearizable [24], in the sense that it can read a consistent state "in the past". That is, it may miss some recent updates, (including ones written by the thread executing the scan). To preserve linearizability, the `getSnap` operation could be modified to wait until it is able to acquire a *snapTime* value greater than the *timeCounter* value at the time the operation started. This can be done by omitting lines 10-11 in Algorithm 2.

Since puts are implemented as insertions with a new timestamp, the key-value store potentially holds many versions for a given key. Following standard practice in LSM-DS, 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.

To consolidate with the merge operation, `getSnap` installs the snapshot handle in a list that captures all active snapshots. Ensuing merge operations query the list to identify the maximal timestamp before which versions can be removed. To avoid a race between installing a snapshot handle and it being observed by a merge, the data structure is accessed while holding the lock. The `getSnap` operation acquires the lock in shared mode while updating the list, and `beforeMerge` queries the list while holding the lock in exclusive mode. The timestamp returned by `beforeMerge` is then used by the merge operation to determine which elements can be discarded. As in levelDB, we assume handles of unused snapshots are removed from the list either by the application (through an API call), or based on TTL; failing to do so may reduce the amount of available memory for useful data.

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[1])

[1] *Compare and Swap* operation [23].

Algorithm 2 cLSM snapshot algorithm.

```
1:  procedure PUT(Key k, Value v)
2:      Lock.lockSharedMode()
3:      ts ← getTS()
4:      P_m.insert(k, ts, v)
5:      Active.remove(ts)
6:      Lock.unlock()
7:  procedure GETSNAP
8:      Lock.lockSharedMode()
9:      ts ← timeCounter.get()
10:     ts_a ← Active.findMin()
11:     if ts_a ≠ ⊥ then ts ← ts_a − 1
12:     atomically assign max(ts, snapTime) to snapTime
13:     while Active.findMin() < snapTime do nop
14:     ts_b ← snapTime
15:     install ts_b in the active snapshot list
16:     Lock.unlock()
17:     return ts_b
18: procedure GETTS
19:     while true do
20:         ts ← timeCounter.incAndGet()
21:         Active.add(ts)
22:         if ts ≤ snapTime then Active.remove(ts)
23:         else break
24:     return ts
25: procedure BEFOREMERGE
26:     Lock.lockExclusiveMode()
27:     P'_m ← P_m
28:     P_m ← new in-memory component
29:     ts ← find minimal active snapshot timestamp
30:     Lock.unlock()
31:     return ts
```

in line 12. The rollback loop in `getTS` may cause the starvation of a `put` operation. We note, however, that each repeated attempt to acquire a timestamp implies the progress of some other `put` and `getSnap` operations, as expected in non-blocking implementations.

3.2.2 Partial Scans and Snapshot Reads

A full snapshot scan traverses all keys starting with the lowest and ending with the highest one. More common are partial scans, (e.g., range queries), in which the application only traverses a small consecutive range of the keys, or even simple reads of a single key from the snapshot. Given our snapshot management mechanism, it is straightforward to support these by using a find function to locate the first entry to be retrieved (like finding a key in a `get` operation).

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. Such operations are useful for many appli-

Algorithm 3 RMW algorithm for linked list memory component.

```
1:  procedure RMW(Key k, Function f)
2:      Lock.lockSharedMode()
3:      repeat
4:          find (k, ts, v) with highest ts in P_m, P'_m, or P_d
5:          prev ← P_m node with max(k', ts') ≤ (k, ∞)
6:          if prev.key = k and prev.time > ts then continue
                                                        ▷ conflict
7:          succ ← prev.next
8:          if succ.key = k then continue               ▷ conflict
9:          ts_n ← getTS()
10:         create newNode with (k, ts_n, f(v))
11:         newNode.next ← succ
12:         ok ← CAS(prev.next, succ, newNode)
13:         if ¬ok then Active.remove(ts_n)             ▷ conflict
14:     until ok
15:     Active.remove(ts_n)
16:     Lock.unlock()
```

cations, ranging from simple vector clock update and validation to implementing full-scale transactions.

Our solution is efficient and avoids blocking. It 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. In a non-blocking implementation of such a data structure, `put` updates the *next* pointer of the predecessor of the inserted node using a CAS operation [23].

The pseudo-code for read-modify-write on an in-memory linked-list appears in Algorithm 3. The idea is to use optimistic concurrency control – having read v as the latest value of key k, our attempt to insert $f(v)$ fails (and restarts) in case a new value has been inserted for k after v. This situation is called a *conflict*, and it means that some concurrent operation has interfered between our read step in line 4 and our update step in line 12.

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. In line 5, we locate, and store in *prev*, the insertion point for the new node. If *prev* is a node holding key k and a timestamp higher than ts, then it means that another thread has inserted a new node for k between lines 4 and 5 — this conflict is detected in line 6. In line 8, we detect a conflict that occurs when another thread inserts a new node for k between lines 5 and 7 — this conflict is observed when *succ* is a node holding key k. If the conflict occurs after line 7, it is detected by failure of the CAS in line 12.

When the data store consists of multiple linked lists, as *libcds*'s lock-free skip-list does [1], items are inserted to the lists one at a time, from the bottom up [23]. Only the bottom list is required for correctness, while the others ensure the logarithmic search complexity. Our implementation thus 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 12, but with no need for a new timestamp 9 or conflict detection as in lines 6 and 8.

We note that the lock-free skip-list [1] (which is based on the skip-list algorithm in [23]) satisfies the requirements specified in Section 3.2 — weak consistency is guaranteed as long as items are not removed from the skip-list, as is the case in cLSM.

4. Implementation

We implement cLSM in C++ based on the popular open source LevelDB LSM-DS library [4]. LevelDB is used by numerous applications including Google Chrome and Facebook's embeddable key-value store [8].

LevelDB implements a rich API that includes read (get), write (put), and various snapshot operations. Its memory component is implemented as a skip list with custom concurrency control. Every write is logged to a sequential file following the LSM-DS update. Typically, the data store is configured to perform logging asynchronously, which allows writes to occur at memory speed; hence, a write only queues the request for logging and a handful of writes may be lost due to a crash. LevelDB features a number of optimizations, including multilevel merge, custom memory allocation, caching via memory-mapped I/O, Bloom filters [14] to speed up reads, etc.

The original LevelDB acquires a global exclusive lock to protect critical sections at the beginning and the end of each read and write. The bulk of the code is guarded by a mechanism that allows a single writer thread and multiple reader threads to execute at any given time. Snapshots are implemented using timestamps – the timestamp management is simpler than ours (i.e., no need for *Active* set) since concurrent write operations are not permitted. LevelDB supports an atomic batch of write operations that is implemented using coarse-grained synchronization of simple write operations.

cLSM supports the full functionality of LevelDB's API. Its implementation inherits the core of LevelDB's modules (disk component, cache, merge function, etc), and benefits from the same optimizations. It implements the algorithm described in Section 3, which eliminates the blocking parts of the LevelDB code. Our support for atomic batches of write operations continues to block (similarly to the original LevelDB) – its synchronization is implemented by holding the shared-exclusive lock in exclusive mode.

We harness the *libcds* concurrent data structures' library [1] to implement the in-memory store and the logging queue (via the non-blocking skip list and queue implementations, respectively). We also implement multiple custom tools based on atomic hardware instructions: a shared-exclusive lock, and a non-blocking memory allocator [29]. All accesses we add to shared memory are protected by

memory fences, whereas the libraries we use include fences where deemed necessary by their developers.

Relaxing LevelDB's single-writer constraint implies that writes might get logged out of order. Since all the log records bear cLSM-generated timestamps, the correct order is easily restored upon recovery.

5. Evaluation

We evaluate our cLSM implementation versus a number of open source competitors. In Section 5.1, our experiments are based on synthetic CPU-bound workloads. In Section 5.2 we use real web-scale application workloads. Finally, in Section 5.3, we use a synthetic disk-bound benchmark from RocksDB's benchmarks suite [10].

Our platform is a Xeon E5620 machine with 2 quad-core CPUs, each core with two hardware threads (16 hardware threads overall). The server has 48GB of RAM and 720GB SSD storage[2].

We vary the concurrency degree in our experiments from one to sixteen worker threads performing operations; these are run in addition to the maintenance compaction thread (or threads in Section 5.3).

We compare cLSM with four open-source LSM data stores: LevelDB [4], HyperLevelDB [5, 21], RocksDB [8], and bLSM [36]. HyperLevelDB and RocksDB are extensions of LevelDB that employ specialized synchronization to improve parallelism (see [3]), and bLSM is a single-writer prototype that capitalizes on careful scheduling of merges. Unless stated otherwise, each LSM store is configured to employ an in-memory component of 128MB (this is the standard value in key-value stores like HBase); we use the default values of all other configurable parameters.

Recall that in LSM-DS, component merges occur as a background process, which is often called *compaction*. All systems except RocksDB use a single background thread for compaction. RocksDB has a configurable parameter determining the maximum number of compaction threads, which we set to one[3], except in Section 5.3. We note that in experiments that involve writes (i.e., put operations), the compaction thread is working a significant portion of the time — in the CPU-bound experiments reported in Sections 5.1 and 5.2, we found that it runs roughly between a quarter and three-quarters of the time, in all systems. In Section 5.3 we consider disk-bound workloads, where compaction runs virtually all the time, and creates a bottleneck.

5.1 Synthetic Workloads

We start with a set of benchmarks 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 publicly available benchmark [6]). The key-value pairs have 8-byte keys, and the value size is 256 bytes.

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

Figure 5a depicts the results in terms of throughput. LevelDB, HyperLevelDB, and cLSM start from approximately the same point, but they behave differently as we increase the number of threads. LevelDB, bLSM and RocksDB are bounded by their single-writer architectures, and do not scale at all. Moreover, having multiple threads contending for a single synchronization point (e.g., a writers queue) causes the throughput to decrease. 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. The degragation in write performance can be explained by cross-chip latency and cache invalidations, since only the 16 threads experiment spans more than one chip. cLSM's peak rate exceeds 430K writes/sec, in contrast with 240K for HyperLevelDB, 160K for LevelDB and 65K for RocksDB.

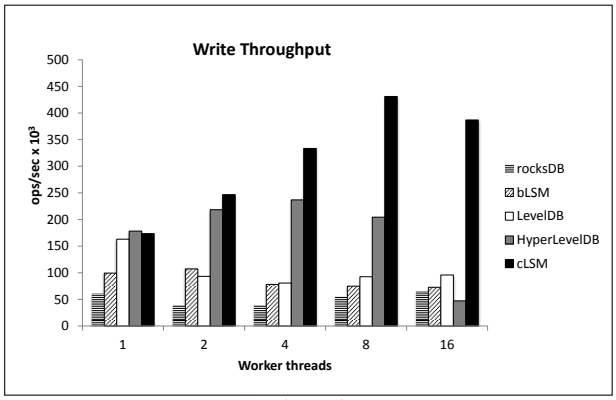
Figure 5b refines the results by presenting the throughput-latency perspective, where the latency is computed for the 90-th percentile; other percentiles exhibit similar trends. For better readability we delineate improvement trends and omit points exhibiting decreasing throughput. This figure marks the point in which each implementation saturates, namely, either achieves a slight throughput gain while increasing the latency by a factor of 2x-3x or achieves no gain at all. It is clear that cLSM scales better than all competitors.

Read performance. We turn to evaluate performance in a read-only 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 "popular" blocks that comprise 10% of the database. The rest are drawn u.a.r. from the whole range. This workload is both dispersed and amenable to caching. Its locality is similar to that of production workloads analyzed in Section 5.2. All the following experiments exercise this distribution.

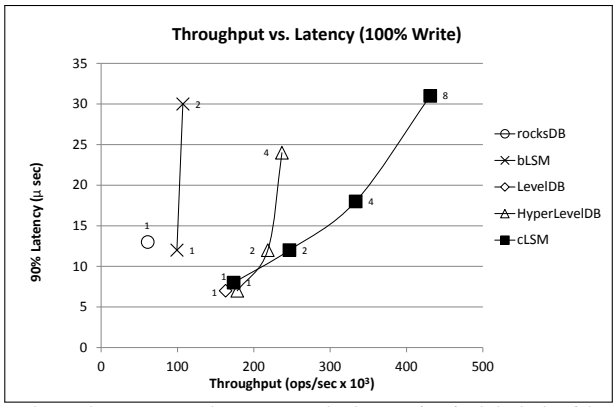
Figure 6a demonstrates throughput scalability. LevelDB and HyperLevelDB exhibit similar performance. Neither scales beyond 8 threads, reflecting the limitations of LevelDB's concurrency control, namely, read operations blocking even when data is available in memory. On the other hand, cLSM and RocksDB scale all the way to 128 threads, far beyond the hardware parallelism (more threads than cores are utilized, since some threads block when reading data from disk). In all cases, RocksDB is not only slower than cLSM, but even slower than LevelDB. In this exper-

[2] Composed of four 240GB SSD SATA/300 OCZ Deneva MLC, configured as RAID-5.

[3] This is the default value in RocksDB.



(a) Throughput



(b) Throughput versus latency; each data point is labeled with the number of threads

Figure 5: **Write performance – a 100% write scenario, with the keys uniformly distributed across the domain. cLSM scales to 8 threads and achieves 80% throughput advantage over the closest competitor, which only scales to 4.**

iment, the peak throughput of cLSM is almost 1.8 million reads/sec – 2.3x as much as the peak competitor rate.

Again, Figure 6b shows the throughput-latency (90-th percentile) perspective. This figure emphasizes the scalability advantage of cLSM: it shows that while RocksDB scales all the way, this comes at a very high latency cost, an order of magnitude higher than other LevelDB-based solutions with the same throughput (800K reads/sec).

Mixed workloads. Figure 7a depicts the throughput achieved by the different systems under a 1:1 read-write mix. The original LevelDB fails to scale, even though the writes are now only 50% of the workload. HyperLevelDB slightly improves upon that result, whereas cLSM fully exploits the software parallelism, scaling beyond 730K operations/sec with 16 workers.

We note that while under cLSM and HyperLevelDB the reads and the writes scale independently (and the throughput numbers are roughly the avarage of the 100% writes and 100% reads scenarios), in LevelDB and RocksDB the writes impede the reads' progress, and therefore the absolute numbers are lower than the average of the 100% writes and 100% reads scenarios.

Figure 7b repeats the same experiment with reads replaced by range scans. (bLSM is not part of this evaluation because it does not directly support consistent scans). The size of each range is picked uniformly between 10 and 20 keys. The number of scan operations is therefore smaller than the number of writes by an order of magnitude, to maintain the balance between the number of keys written and scanned. The cumulative throughput is measured as the overall number of accessed keys. Similarly to the previous cases, the competitors are slower than cLSM by more than 60%. Note that scans are faster than read operations since in each scan operation, the scanned items are located close to the first item, which results in write operations running substantially more than 50% of the time, and the cross-chip effect causes a small degragation in cLSM's throughput with 16 worker threads.

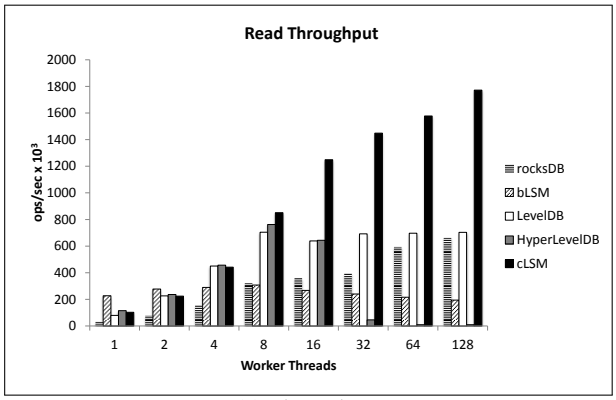
We next evaluate how the system may benefit from additional RAM. Figure 8 compares LevelDB's and cLSM's benefit from larger memory components, under the read-write workload, with 8 working threads. LevelDB performs nearly the same for all sizes beyond 16MB, whereas cLSM keeps improving with the memory buffer growing to 512MB. In general, LSM data stores may gain from increasing the in-memory component size thanks to better batching of disk accesses [11]. However, this also entails slower in-memory operations. We see that cLSM successfully masks this added latency via its high degree of parallelism, which the less scalable alternatives fail to do.

Read-Modify-Write. We now explore the performance of atomic RMW operations (put-if-absent flavor [37]). To establish a comparison baseline, we augment LevelDB with a textbook RMW implementation based on lock striping [22]. The algorithm protects each RMW and write operation with an exclusive granular lock to the accessed key. The basic read and write implementations remain the same.

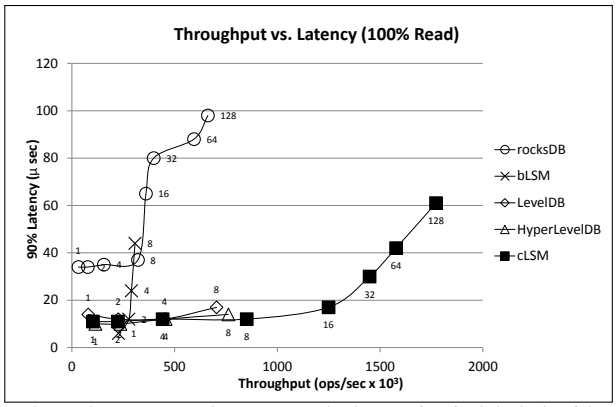
We compare the lock-striped LevelDB with cLSM. The first workload under study is comprised solely of RMW operations. As shown in Figure 9, cLSM scales to almost 400K operations/sec – a 2.5x throughput gain compared to the standard implementation. This volume is almost identical to the peak write load.

5.2 Production Workloads

We study a set of 20 workloads logged in a production key-value store that serves some of the major personalized content and advertising systems on the web. Each log captures the history of operations applied to an individual partition server. The average log consists of approximately 5 million operations. Operations have variable key and value sizes, averaging 40-bytes per key, and 1KB values. The captured

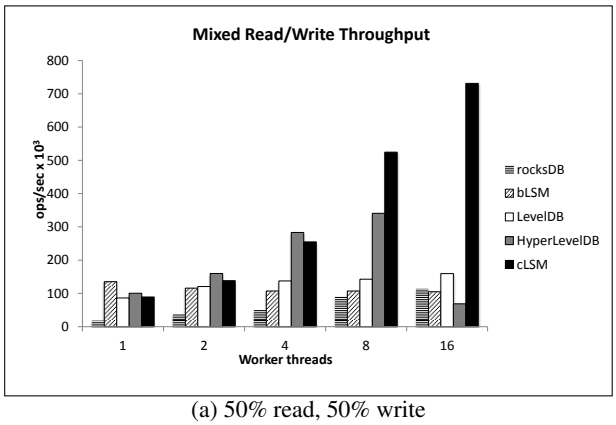


(a) Throughput

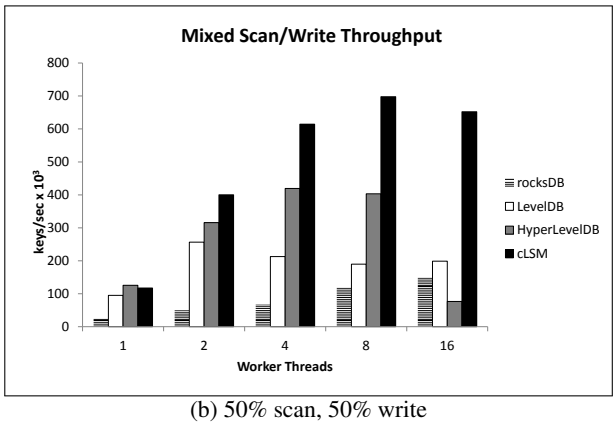


(b) Throughput versus latency; each data point is labeled with the number of threads

Figure 6: **Read performance – a 100% read scenario with locality (90% of keys picked from 10% popular blocks).**



(a) 50% read, 50% write



(b) 50% scan, 50% write

Figure 7: **Throughput in mixed workloads.**

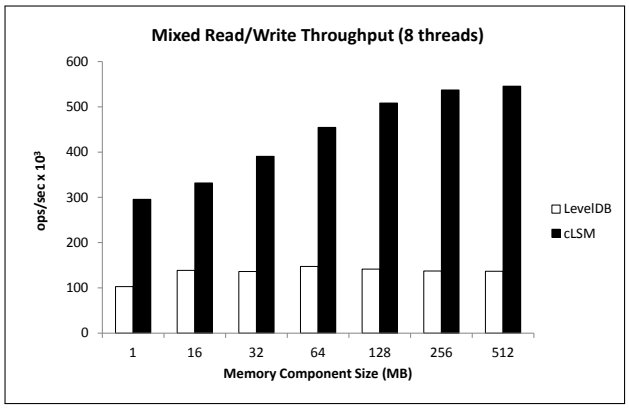


Figure 8: **Mixed reads and writes benefit from memory component size with 8 threads. cLSM successfully exploits RAM buffers of up to 512MB, whereas LevelDB can only exploit 16MB.**

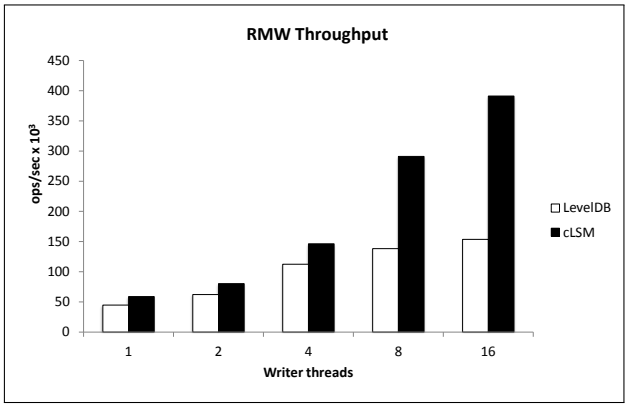
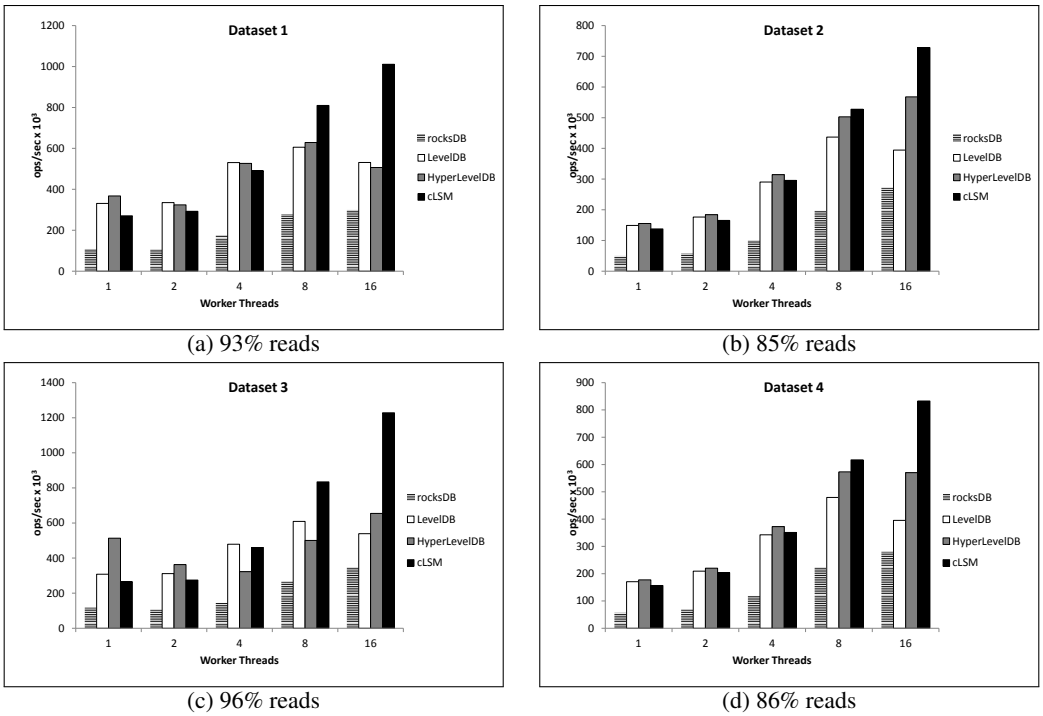


Figure 9: **Read-modify-write (RMW) throughput – a 100% put-if-absent scenario with locality. cLSM improves upon lock-striping by 150%.**



(a) 93% reads (b) 85% reads

(c) 96% reads (d) 86% reads

Figure 10: **Throughput in workloads collected from a production web-scale system.**

workloads are read-dominated (85% to 95% reads). The key distributions are heavy-tail, all with similar locality properties. In most settings, 10% of the keys stand for more than 75% of the requests, while the 1-2% most popular keys account for more than 50%. Approximately 10% of the keys are only encountered once.

Figure 10 depicts the evaluation results for 4 representative workloads. Although cLSM is slower than the alternatives with a small number of threads, its scalability is much better. These results are similar to the results shown in 7a. However, our advantage over the competitors is reduced, because, with larger keys and values, the synchronization overhead is less pronounced.

5.3 Workloads with Heavy Disk-Compaction

The above experiments demonstrate situations in which the in-memory access is the main performance bottleneck. Recently, the RocksDB project has shown that in some scenarios, the main performance bottleneck is disk-compaction [10]. In these scenarios, a huge number of items is inserted (at once) into the LSM store, leading to many heavy disk-compactions. As a result of the high disk activity, the C_m component frequently becomes full before the C'_m component has been merged into the disk. This causes client operations to wait until the merge process completes.

We use a benchmark from [10] to demonstrate this situation. In this benchmark, the initial database is created by sequentially inserting 1 billion items. During the benchmark, 1 billion update operations are invoked by the worker threads. As in [10], each key is of size 10 bytes; however, each value is of size 400 bytes (instead of 800) to ensure that our 720GB disk is sufficient.

We compare cLSM with RocksDB following the configuration in [10]. RocksDB is configured to use multi-threaded compactions so that multiple threads can simultaneously compact non-overlapping key ranges in multiple levels. For each parameter that appears both in cLSM and RocksDB, we configure the systems to use the same values. Specifically, these parameters are: size of in-memory component (128MB), total number of levels (6 levels), target file size at level-1 (64MB), and number of bytes in a block (64KB).

Figure 11 depicts the results of this benchmark. The results show that both cLSM and RocksDB scale all the way to 16 worker threads (despite the fact that disk-compaction is running most of the time). At 16 threads, cLSM becomes equivalent to RocksDB. Notice that RocksDB uses an optimized compaction algorithm that utilizes several background threads, whereas cLSM uses a simpler compaction algorithm executed by a single background thread. It should be noted that RocksDB's compaction optimizations are orthogonal to our improved parallelism among worker threads.

6. Related Work

The basis for LSM data structures is the *logarithmic method* [13]. It was initially proposed as a way to efficiently transform static search structures into dynamic ones.

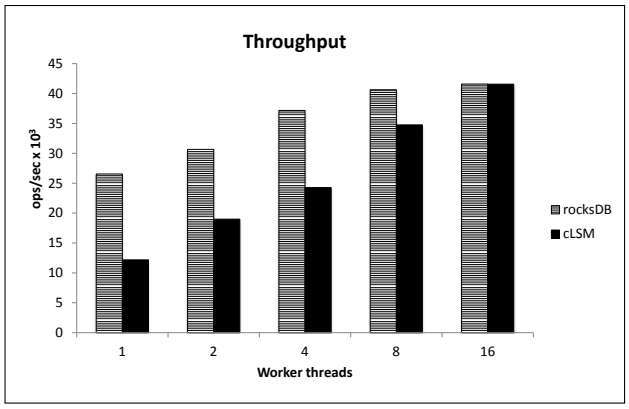


Figure 11: **Workload with heavy disk-compaction.**

This method inspired the original work on *LSM-trees* [31] and its variant for multi-versioned data stores [30]. LSM-trees provide low-cost indexing for key-value stores with high rates of put operations, by deferring in-place random writes and batching them into sequential writes. The LSM-tree indexing approach employs B^+-tree-like structures as its disk components, and for the main memory component, an efficient key-lookup structure similar to a (2-3)-tree or—more common in recent implementations—a skip-list [34].

Nowadays, key-value stores are commonly implemented as LSM data stores [2, 8, 16, 18, 25]. Google's LevelDB [4] is the state-of-the-art implementation of a single machine LSM that serves as the backbone in many of such key-value stores. It applies coarse-grained synchronization that forces all puts to be executed sequentially, and a single threaded merge process. These two design choices significantly reduce the system throughput in multicore environment. This effect is mitigated by HyperLevelDB [5], the data storage engine that powers HyperDex [21]. It improves on LevelDB in two key ways: (1) by using fine-grained locking to increase concurrency, and (2) by using a different merging strategy. Our evaluations show that cLSM outperforms both of them.

Facebook's key-value store, RocksDB [8] also builds on LevelDB. Much effort is done in order to reduce critical sections in the memory component [3, 9]. Specifically, readers avoid locks by caching metadata in their thread local storage. Only when a newer version becomes available readers use locks to get hold of a reference to it. In addition, the merge process of disk components is executed by multiple threads concurrently, and some thread is always reserved for flushing the memory component to the disk.

In the same vein, bLSM [36] introduces a new merge scheduler, which bounds the time a merge can block write operations. As bLSM optimizations focus on the merging process and disk access, it is orthogonal to our work on memory optimizations.

Several approaches for optimizing the performance of the general logarithmic method have been proposed in recent years. One such approach suggests adopting a new tree-indexing data structure, *FD-tree* [28], to better facilitate the properties of contemporary flash disks and solid state drives (SSDs). Like components in LSM-trees, FD-trees maintain multiple *levels* with cross-level pointers. This approach applies the *fractional cascading* [17] technique to speed up search in the logarithmic structure. A follow-up work [39] further refines FD-trees to support concurrency, allowing concurrent reads and writes during ongoing index reorganizations.

With a similar goal of exploiting flash storage as well as the caches of modern multi-core processors, *Bw-tree* [27] is a new form of a B-tree, used as an index for a persistent key-value store. The implementation is non-blocking, 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 [26], uses CAS instructions, and therefore blocks only rarely, when fetching a page from disk. At its storage layer, Bw-tree uses log structuring [35].

None of these new approaches support consistent scans or an atomic RMW operation (as cLSM does). In addition, each of these algorithms builds upon a specific data structure as its main memory component, whereas our work can employ any implementation of a concurrent sorted map to support the basic API.

7. Discussion

Leading key-value stores today rely on LSM-DS methodology for serving requests mostly from RAM. With this approach, the implementation of in-memory building blocks is critical for performance, as we have demonstrated in Section 5. The primary challenge such systems face is scaling up with the available hardware resources most notably, the number of CPU cores. In this context, the concurrency control that protects shared data structures can be a major performance roadblock. Our work overcomes this roadblock and presents cLSM, an efficient concurrent LSM-DS implementation. Scalability is achieved by eliminating blocking in scenarios that do not involve physical access to disk.

In addition to atomic reads and writes, cLSM supports consistent snapshot scans, range queries, and atomic read-modify-write operations. Our algorithm is generic, and can be applied to a range of implementations. Such decoupling allows our solution to be combined with other optimization applied to the disk components and merge utility.

Our evaluation versus state-of-the-art LSM implementations shows performance improvements and superior scalability, even when the competitors utilize smaller partitions. The latter, along with other disadvantages of partitioning discussed in Section 2, suggests that our approach can potentially serve as an alternative for vertical scalability.

References

[1] libcds: Library of lock-free and fine-grained algorithms. http://libcds.sourceforge.net/, Nov. 2013.

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

[3] Avoid expensive locks in RocksDB. http://rocksdb.org/blog/677/avoid-expensive-locks-in-get/, June 2014.

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

[5] Hyperleveldb. https://github.com/rescrv/HyperLevelDB, Apr. 2014.

[6] Hyperleveldb performance benchmarks. http://hyperdex.org/performance/leveldb, Apr. 2014.

[7] Java's ConcurrentSkipListMap. http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html, Apr. 2014.

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

[9] Reducing lock contention in RocksDB. http://rocksdb.org/blog/521/lock/, May 2014.

[10] RocksDB performance benchmarks. https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks, May 2014.

[11] Why cannot I have too many regions? https://hbase.apache.org/book/regions.arch.html, Apr. 2014.

[12] AIYER, A., BAUTIN, M., CHEN, G., DAMANIA, P., KHEMANI, P., MUTHUKKARUPPAN, P., RANGANATHAN, K., SPIEGELBERG, N., TANG, L., AND VAIDYA, M. Storage Infrastructure Behind Facebook Messages: Using HBase at Scale. *IEEE Data Eng. Bull 35* (2012), 4–13.

[13] BENTLEY, J. L., AND SAXE, J. B. Decomposable searching problems i. static-to-dynamic transformation. *Journal of Algorithms 1*, 4 (1980), 301–358.

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

[15] BRONSON, N. G., CASPER, J., CHAFI, H., AND OLUKOTUN, K. A practical concurrent binary search tree. *SIGPLAN Not. 45*, 5 (2010), 257–268.

[16] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).

[17] CHAZELLE, B., AND GUIBAS, L. J. Fractional cascading: I. a data structuring technique. *Algorithmica 1*, 2 (1986), 133–162.

[18] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow. 1*, 2 (2008), 1277–1288.

[19] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007), pp. 205–220.

[20] DUFFY, J. *Concurrent Programming on Windows*. Addison-Wesley, 2008.

[21] ESCRIVA, R., WONG, B., AND SIRER, E. G. Hyperdex: a distributed, searchable key-value store. In *SIGCOMM 2012* (2012), pp. 25–36.

[22] GRAY, J., AND REUTERS, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[23] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[24] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *TOPLAS 12* (1990).

[25] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[26] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst. 6*, 4 (1981), 650–670.

[27] LEVANDOSKI, J. J., LOMET, D. B., AND SENGUPTA, S. The Bw-tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering (ICDE)* (2013), pp. 302–313.

[28] LI, Y., HE, B., YANG, R. J., LUO, Q., AND YI, K. Tree indexing on solid state drives. *Proc. VLDB Endow. 3*, 1-2 (2010), 1195–1206.

[29] MICHAEL, M. M. Scalable lock-free dynamic memory allocation. *SIGPLAN Not. 39*, 6 (2004), 35–46.

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

[31] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Inf. 33*, 4 (June 1996), 351–385.

[32] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (Oct. 1979), 631–653.

[33] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *OSDI* (2010), pp. 4–6.

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

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

[36] SEARS, R., AND RAMAKRISHNAN, R. bLSM: A general purpose log structured merge tree. In *SIGMOD 2012* (2012), pp. 217–228.

[37] SHACHAM, O., YAHAV, E., GOLAN-GUETA, G., AIKEN, A., BRONSON, N., SAGIV, M., AND VECHEV, M. Verifying atomicity via data independence. In *ISSTA* (2014).

[38] TANENBAUM, A. S., AND BOS, H. *Modern Operating Systems*, 4th ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2014.

[39] THONANGI, R., BABU, S., AND YUNG, J. A practical concurrent index for solid-state drives. In *CIKM 2012* (2012), pp. 1332–1341.

[40] WU, G., HE, X., AND ECKART, B. An adaptive write buffer management scheme for flash-based ssds. *Trans. Storage 8*, 1 (Feb. 2012).

[41] YABANDEH, M., GOMEZ-FERRO, D., KELLY, I., JUNQUEIRA, F., AND REED, B. Lock-free transactional support for distributed data stores. In *ICDE '14*.