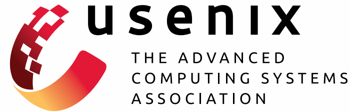


LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data

Xingbo Wu and Yuehai Xu, *Wayne State University;* Zili Shao, *The Hong Kong Polytechnic University;* Song Jiang, *Wayne State University*

https://www.usenix.org/conference/atc15/technical-session/presentation/wu

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the
2015 USENIX Annual Technical Conference
(USENIX ATC '15) is sponsored by USENIX.

LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data

Xingbo Wu[1], Yuehai Xu[1], Zili Shao[2], and Song Jiang[1]

[1] Wayne State University, {wuxb,yhxu,sjiang}@wayne.edu
[2] The Hong Kong Polytechnic University, cszlshao@comp.polyu.edu.hk

Abstract

Key-value (KV) stores have become a backbone of large-scale applications in today's data centers. The data set of the store on a single server can grow to billions of KV items or many terabytes, while individual data items are often small (with their values as small as a couple of bytes). It is a daunting task to efficiently organize such an ultra-large KV store to support fast access. Current KV storage systems have one or more of the following inadequacies: (1) very high data write amplifications, (2) large index set, and (3) dramatic degradation of read performance with overspill index out of memory.

To address the issue, we propose LSM-trie, a KV storage system that substantially reduces metadata for locating KV items, reduces write amplification by an order of magnitude, and needs only two disk accesses with each KV read even when only less than 10% of metadata (Bloom filters) can be held in memory. To this end, LSM-trie constructs a trie, or a prefix tree, that stores data in a hierarchical structure and keeps re-organizing them using a compaction method much more efficient than that adopted for LSM-tree. Our experiments show that LSM-trie can improve write and read throughput of LevelDB, a state-of-the-art KV system, by up to 20 times and up to 10 times, respectively.

1 Introduction

Key-value (KV) stores play a critical role in the assurance of service quality and user experience in many websites, including *Dynamo* [22] at Amazon, *Voldemort* [7] at LinkedIn, *Cassandra* [1] at Apache, *LevelDB* [4] at Google, and *RocksDB* [11] at Facebook. Many highly-demanding data-intensive internet applications, such as social networking, e-commerce, and online gaming, rely on quick access of data in the stores for quality service.

A KV store has its unique advantage on efficient implementation with a flat data organization and a much simplified interface using commands such as Put(key,value) for writing data, Get(key) for reading data, and Delete(key). However, there are several trends on workload characteristics that are seriously challenging today's state-of-the-art KV store implementations for high performance and high scalability.

First, very small KV items are widespread. As an example, Facebook had reported that 90% of its Memcached KV pools store KV items whose values are smaller than 500 bytes [13]. In one KV pool (USR) dedicated for storing user-account statuses all values are of 2 bytes. In its nonspecific, general-purpose pool (ETC) 2-, 3-, or 11-byte values add up to 40% of the total requests to the store. In a replicated pool for frequently accessed data, 99% of KV items are smaller than 68 bytes [26]. In the wildcard (the default pool) and a pool devoted for a specific application, 75% of items are smaller than 363 bytes. In Twitter's KV workloads, after compression each tweet has only 362 bytes, which contains only 46 bytes of text [3]. In one of Instagram's KV workloads the key is the media ID and the value is the user ID. Each KV item is just as large as a couple of bytes [10]. For a store of a given capacity, smaller KV items demand more metadata to locate them. The metadata may include index for locating a data block (e.g., a 4 KB disk block) and Bloom filters for determining data existence in the block.

Second, demand on a KV store's capacity at individual KV servers keeps increasing. The rising demand is not only due to data-intensive applications, but also because of the cost benefit of using fewer servers to host a distributed KV store. Today it is an economical choice to host a multi-terabytes KV store on one server using either hard disks or SSDs. However, this would significantly increase metadata size and make memory constrained, which is especially the case when significant applications, such as MapReduce jobs, are scheduled to the cluster hosting the store, competing the memory resource with the storage service [19, 33].

Third, many KV stores require high performance for both reads and writes. It has been reported that ratio of read and write counts in typical low-latency workloads at Yahoo had shifted from anywhere between 2 and 9 to around 1 in recent years [29]. Among the five core workloads in Yahoo's YCSB benchmark suite two of them have equal share of read and write requests [18]. There are KV stores, such as LevelDB, that are optimized for writes by organizing data in multiple levels. However, when not all metadata can be held in memory, multiple disk reads, each for medadata of a level, are needed to serve a read request, degrading read performance. In the meantime, for some KV stores, such as SILT [24], major efforts are made to optimize reads by minimizing metadata size, while write performance can be compromised without conducting multi-level incremental compactions.

In this paper, we propose LSM-trie, a KV storage system that can accommodate multi-billions of small items with a capacity of multi-terabytes at one server with limited memory demand. It supports a sustained throughput of over 500 K writes per second, and a sustained throughput of over 50 K reads per second even for workloads without any locality and thus with little help from caching[1]. To achieve this, LSM-trie uses three novel techniques. First, it integrates exponential growth pattern in the LSM tree (Log-Structured Merge-tree)—a commonly adopted KV-store organization—with a linear growth pattern. This enables a compaction design that can reduce write amplification by an order of magnitude and leads to much improved write throughput. A high write throughput is desired as data modifications and deletions are also processed as writes in the store implementation. Second, using a trie, or a prefix tree, to organize data in the store, LSM-trie almost eliminates index. This allows more and stronger Bloom filters to be held in memory, making service of read requests faster. Third, when Bloom filters become too large to be entirely held in the memory, LSM-trie ensures that on-disk Bloom filters are clustered so that in most cases only one 4 KB-block read is required to locate the data.

Experiments show that LSM-trie significantly improves write throughput over schemes in comparison, including LevelDB, RocksDB, and SILT, by up to 20 times regardless of system configurations such as memory size, store size, storage devices (SSD or HDD), and access pattern (uniform or Zipfian key distributions). LSM-trie can also substantially improve read throughput, especially when memory available for running the KV store is limited, by up to 10 times.

Note that LSM-trie uses hash functions to organize its data and accordingly does not support range search. This is a choice similarly made in the design of many important KV stores, including Amazon's Dynamo [22], LinkedIn's Voldermort [7], and SILT [24], as this command is not always required by their users. Furthermore, there are techniques available to support the command by maintaining an index above these hash-based stores with B-link tree [17] or dPi-tree [25], and experimental studies indicate that "there is no absolute winner" in terms of range-search performance between stores natively supporting it and those relying on external support [28].

[1] The throughput of read is significantly lower than that of write because one read needs access of at least one 4 KB block, while multiple small KV items in write requests can be compacted into one block.

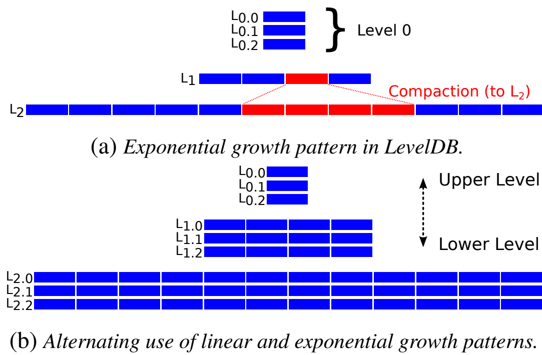
2 The design of LSM-trie

The design of LSM-trie was motivated by the excessively large write amplification of LSM-tree due to its data organization and compaction scheme [27]. In this section we will describe the issue in the context of LevelDB, a popular implementation of LSM-tree from Google. Then we will describe a trie-based LSM-tree implementation that can dramatically reduce write amplification in Section 2.3. However, this optimized LSM-tree still retains an index, which grows with the store size and eventually becomes a barrier to the system's scalability. In addition, it may require multiple reads of Bloom filters on the disk with a large store. In Section 2.4, we describe **LSM-trie**, where KV items are hashed into individual buckets, indices are accordingly removed, and Bloom filters are grouped together to support efficient access.

2.1 Write Amplification in LSM-tree

A KV store design based on LSM-tree has two goals: (1) new data must be quickly admitted into the store to support high-throughput write; and (2) KV items in the store are sorted to support fast data location. We use a representative design, LevelDB, as an example to explain the challenges on simultaneously achieving both of the goals.

To meet the first goal LevelDB writes to the disk in a large unit (a couple of megabytes) to generate an on-disk data structure called *SSTable*. Specifically, LevelDB first uses an in-memory buffer, called *MemTable*, to receive incoming KV items. When a MemTable is full, it is written to the disk to become an immutable SSTable. KV items in an SSTable are sorted according to their keys. An SSTable is stored as a file, and KV items are placed in 4 KB blocks of the file. To locate a KV item in the SSTable, LevelDB places an index in the file recording the key of the first KV item in each block. Conducting binary search on the index, LevelDB knows in which block a KV item can possibly be located. Because 4 KB block is a disk access unit, it is not necessary to maintain a larger index to determine byte offset of each item in a



(a) *Exponential growth pattern in LevelDB.*

(b) *Alternating use of linear and exponential growth patterns.*

Figure 1: *Using multi-level structure to grow an LSM-tree store. Each solid rectangle represents an SSTable.*

block. However, the index does not tell whether an item is actually in the block. If not, accessing the block is unnecessary and can substantially increase read latency. To this end, LevelDB maintains a Bloom filter for each block to indicate whether an item is in it [16]. To minimize its false positive rate, the filter must be sized proportionally to the number of items in a block, usually 10–16 bits per item.

To meet the second goal LevelDB builds a multi-level tree-like structure to progressively sort KV items. As shown in Figure 1a, new SSTables, which are just converted from MemTables, are placed in Level 0. To quickly admit incoming items, items in new SSTables are not immediately sorted with those in existing SSTables at Level 0. Instead, each of the SSTables becomes a sub-level ($L_{0.0}, L_{0.1}, L_{0.2}, \ldots$) of Level 0 (See Figure 1a). In the background, LevelDB merge-sorts a number of L_0 SSTables to produce a list of non-overlapping SSTables at Level 1 (L_1), an operation called **compaction**. To quickly have more data sorted into one list, starting from Level 1 there are no sub-levels and the ratio of two adjacent levels' sizes is large ($Size(L_{k+1})/Size(L_k)$, where $k = 0, 1, \ldots$). We name the ratio amplification factor, or AF in short, which is 10 in LevelDB by default. As every level (L_{k+1}) can be 10 times as large as its immediate upper level (L_k), the store keeps producing exponentially larger sorted list at each level and becomes very large with only a few levels.

However, this exponential growth pattern leads to an excessively large write amplification ratio, a ratio between actual write amount to the disk and the amount of data requested for writing by users. Because the range of keys covered by each level is roughly the same, to push one SSTable at a level down to its next lower level LevelDB needs to read this SSTable and ten SSTables in the lower level (in the worst case) whose entire key range matches the SSTable's key range. It then merge-sorts them and writes the 11 resulting SSTables to the lower level. That is, the write amplification ratio is 11, or $AF+1$. For a new KV item to reach Level k ($k = 0, 1, 2, \ldots$), the write amplification ratio can go up to $k \times (AF+1)$. When the k value reaches 5 or larger, the amplification ratio can become unacceptably large (55 or larger). Such an expensive compaction operation can consume most of the I/O bandwidth and leave little for servicing frontend user requests.

For a store of given capacity, efforts on reducing the write amplification by limiting number of levels would have counter effect. One example is the SILT KV store [24], which essentially has two levels (HashStore and SortedStore). When the store grows large, its SortedStore has to be much larger than HashStore (even when multiple HashStores are employed). This causes its very high write amplification (see Section 3 for measurements), which justifies the use of multiple levels for progressive compaction in the LSM-tree-based stores.

2.2 Challenge on Reducing Write Amplification in the LSM-tree Compaction

A compaction entails reading sorted lists (one SSTable from L_k and a number of SSTables matching its key range from L_{k+1}), merging-sorting them into one sorted list, and writing it back to L_{k+1}. While any data involved in the operation contribute to the write amplification, it is the larger data set from the lower level (L_{k+1}) that makes the amplification ratio excessively large. Because the purpose of the compaction is to push data to the lower level, the contribution to the amplification from accessing data at the upper level is necessary. If we manage to allow only data at the upper level to be involved in a compaction, the write amplification can be minimized.

To this end, we introduce the linear growth pattern. As shown in Figure 1b, in addition to Level 0 other levels also consist of a number of its sub-levels. Sub-levels belonging to the same level are of the same (maximum) size. When a new sub-level is produced at a level, the store linearly grows at this level. However, when a new level is produced, the store *exponentially* grows (by AF times). During growth of the store, new (sub)-levels are produced alternatively using the linear and exponential growth patterns. In other words, each LevelDB's level is replaced by multiple sub-levels. To minimize write amplification, we can merge-sort data in the sub-levels of a level (L_k) to produce a new sub-level of its next lower level (L_{k+1}). As similar amount of data in each sub-level, but no data in the next lower level, are involved in a compaction, write amplification can be minimized.

A key consideration in LevelDB's implementation is to bound each compaction's maximum cost in terms of number of SSTables involved, or $AF+1$, to keep service of user requests from being disruptively slowed down by the background operation. For the same purpose, in the

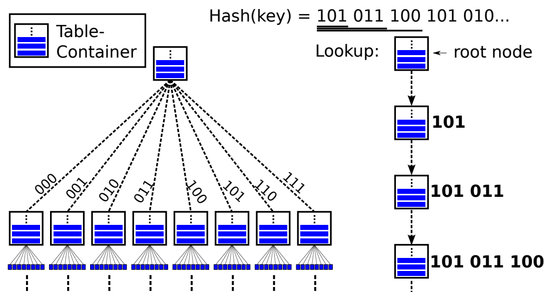


Figure 2: *A trie structure for organizing SSTables. Each node represents a table container, which contains a pile of SSTables.*

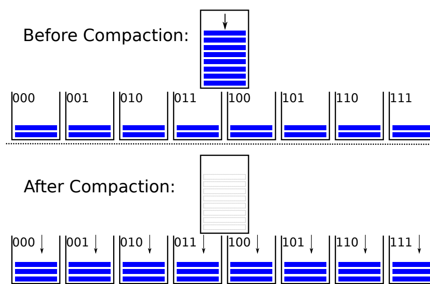


Figure 3: *A compaction operation in the trie.*

use of linear growth pattern in a compaction we select one SSTable at each sub-level of a level (L_k), and merge-sort these SSTables into a sequence of non-overlapping SSTables at Level L_{k+1}. The range of keys involved in a compaction represents the *compaction's key range*. Among all compactions moving data from L_k to L_{k+1}, we must make sure their key ranges are not overlapped to keep any two SSTables at Level L_{k+1} from having overlapped key ranges. However, this cannot be achieved with the LevelDB data organization because the sorted KV-items at each sub-level are placed into the SSTables according to the tables' fixed capacity (e.g., 32 MB). The key range size of an SSTable can be highly variable and the ranges' distribution can be different in different sub-levels. Therefore, ranges of the aforementioned compactions are unlikely to be un-overlapped.

2.3 *SSTable-trie*: A Design for Minimizing Write Amplification

To enable distinct key range in a compaction, we do not use a KV-item's ranking (or its position) in a sorted list to determine the SSTable it belongs to in a level. Instead, we first apply a cryptographic hash function, such as SHA-1, on the key, and then use the hashed key, or *hashkey* in short, to make the determination. This essentially converts the LevelDB's multi-level structure into a trie, as illustrated in Figure 2. Accordingly we name this optimized LevelDB **SSTable-trie**.

An SSTable-trie is a prefix tree whose nodes are table containers, each containing a number of SSTables. Each node has a fixed number of child nodes and the number is equivalent to the *AF* (amplification factor) in LevelDB. If the number is assumed to be 8, a node's children can be distinguished by a three-bit binary $(000, 001, \ldots, \text{or } 111)$. A node in the trie can also be identified by a binary, usually of more bits. Starting from the root node, we can segment the binary into consecutive three-bit groups with the first group indicating a root's child. As each bit group identifies a corresponding node's child, we can follow the bit groups to find a path to the node corresponding to the binary. All nodes of the same depth in a trie constitute a level in the trie structure, which is equivalent to a level in LevelDB. Each container has a pile of SSTables (see Figure 2). A trie level consists of a number of SSTable piles. All SStables at the same position of the piles at a trie level constitute a sub-level of the trie, which corresponds to a sub-level in LevelDB.

As each KV item is also identified by a binary (the hashkey), its location in a level is determined by matching the hashkey's prefix to the identity of a node in the level (see Figure 2). In contrast to the KV-item placement in a level of LevelDB, a KV-item's location in a trie level is independent of other keys in the same level. A compaction operation involves a pile of SSTables in only one container. After a compaction KV items in a pile are moved into the container's children according to their respective hashkeys, rather than their rankings in the sorted list as LevelDB does. By using hashkeys each compaction's key range is unique and SSTables produced by a compaction are non-overlapping. Such a compaction incurs minimal write amplification. Figure 3 illustrates a compaction operation in a trie. Note that use of SHA-1 as the hash function to generate hashkey guarantees a uniform distribution of KV items at each (sub)-level regardless of distribution of original keys.

2.4 *LSM-trie*: a Large Store for Small Items

Our goal is to enable very large KV stores in terms of both capacity and KV-item count in a server. A big challenge on designing such a store is the management of its metadata that often have to be out of core (the DRAM).

2.4.1 Out-of-Core Metadata

For a given KV item, there is at most one SSTable at each (sub)-level that may store the item in LevelDB because every (sub)-level is sorted and its SSTables' key ranges are not overlapped. The store maintains a very small in-memory search tree to identify the SSTable at each level. At the end of each SSTable file an index and Bloom filters are stored to facilitate search in the table. The index

is employed to identify a 4 KB block and a Bloom filter is maintained for each block to tell whether a KV item is possibly in the block. The indices and Bloom filters in a KV store can grow very large. Specifically, the size of the indices is proportional to the store's capacity (or number of 4 KB blocks), and the size of the Bloom filters is proportional to total item count. For a large store the metadata can hardly be accommodated in memory. For example, a 10 TB store holding 200 B-KV-items would require about 125 GB space for 10-bit-per-key Bloom-filters and 30 GB for indices. While it is well affordable now and even so in the near future to have an HDD array or even an SSD array as large as 10 TB in a server, it is not cost-effective to dedicate such a large DRAM only for the metadata. Therefore, we have to assume that significant portion of the metadata is only on the disk when the store grows large. Because locality is usually not assumed in KV-store workloads [14, 31], the fact can be that most reads require retrieval of metadata from the disk before data can be read. The critical issue is how to minimize number of metadata reads in serving a read request for a KV item. These metadata are possibly stored in multiple SSTables, each at a different level. As the metadata are associated with individual SSTables and are distributed over them, having multiple reads seems to be unavoidable in the current LSM-tree's structure.

SSTable-trie introduces the linear growth pattern, which leads to the design of **LSM-trie** that removes almost all indices and enables one metadata disk access per read request. Before describing the design, let us first address a concern with SSTable-trie. Using the linear growth pattern one can substantially increase number of levels. As a multi-level KV-item organization requires continuous search of levels, starting from Level 0, for a requested item until it is found, it relies on Bloom filters in each level to skip as many levels without the item as possible. However, as each Bloom filter has a false positive rate (about 0.82% for a setting of 10 bits per item), the probability of searching levels without the item increases with the increase of level count (e.g., from 5.7% for a 7-level structure to 46% for a 56-level one). Therefore, the Bloom filter must be beefed up by using more bits. For example, using a setting of 16 bits per item would ensure less than 5% false positive rate for an entire 120-level structure. Compared with the disk capacity, the additional on-disk space for the larger Bloom filters is minimal. As we will show, LSM-trie removes indices and uses only one disk access to read Bloom filters.

2.4.2 Removing Indices by Using HTables

LSM-trie represents an improvement over SSTable-trie by incorporating an efficient metadata management. A major change is to replace the SSTable in SSTable-trie

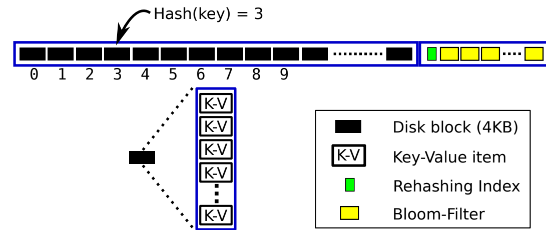
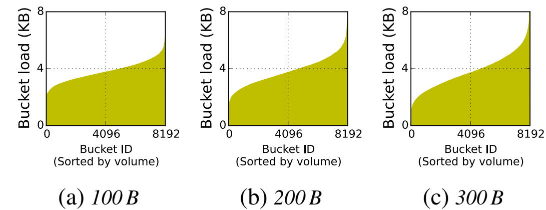


Figure 4: *The structure of an HTable.*



(a) *100 B* (b) *200 B* (c) *300 B*

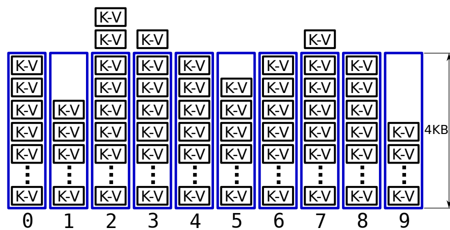
Figure 5: *Distribution of bucket load across buckets of an HTable with a uniform distribution of KV-item size and an average size of 100 B (a), 200 B (b), and 300 B (c). The keys follow the Zipfian distribution. For each plot, the buckets are sorted according to their loads in terms of aggregate size of KV items in a bucket.*

with **HTable**, a hash-based KV-item organization (see Figure 4). In an SSTable, items are sorted and index is needed for locating a block. In HTable, each block is considered as a bucket for receiving KV items whose keys are hashed into it. While each KV item has a SHA-1-generated 160 bit hashkey and its prefix has been used to identify an SSTable in SSTable-trie, or an HTable in LSM-trie, we use its suffix to determine a bucket within an HTable for the KV item. Specifically, if there are m buckets in an HTable, a KV item with Hashkey h would be placed in Bucket $(h \mod m)$.

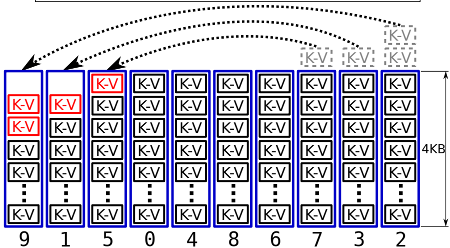
To eliminate the index in an HTable, LSM-trie must use buckets of fixed size. Further, as Bloom filter is applied on individual buckets, an entire bucket would be read should its filter indicate a possible existence of a lookup item in the bucket. Therefore, for access efficiency buckets should be of the same size as disk blocks (4 KB). However, a challenging issue is whether the buckets can be load balanced in terms of *aggregate size of KV items hashed into them*. It is known that using a cryptographic hash function allows each bucket to have statistically equal chance to receive a new item, and item count in each bucket follows a normal distribution. In addition to key's distribution, item size[2] and variation of item size also add to variation of the bucket load.

Figure 5 shows the distribution of bucket load across the buckets in an HTable after we store KV items, whose keys are of the Zipfian distribution, into a 32 MB HTable of 8192 4 KB-buckets. For each plot, the item size is of

[2]With larger KV items it is harder to balance the load across the buckets in an HTable.



(a) *KV items are assigned to the buckets by the hash function, causing unbalanced load distribution.*



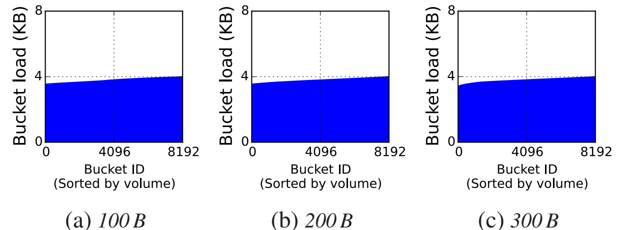
(b) *Buckets are sorted according to their loads and balanced by using a greedy algorithm.*

Figure 6: *Balancing the load across buckets in an HTable.*

the uniform distribution with different average sizes, and the size is in the range from 1 B to a size about doubling their respective averages. In each experiment we keep writing KV items to the store until it is 95% full. By using the highly-skewed Zipfian distribution, the results represent a conservative estimation of non-uniformity of bucket load distribution.[3] As shown, there are increasingly more over-loaded buckets and more under-loaded buckets with the increase of average item size.

Obviously LSM-trie must move excessive items out of over-loaded buckets to make sure every bucket has 4 KB or less data. Like SSTable, HTable is also immutable. During the construction of an HTable, we use a greedy algorithm to migrate some items that were originally hashed to an overloaded bucket to an under-loaded bucket for storage. As illustrated in Figure 6, the buckets are first sorted into a list according to their initial loads. We then conduct a paired migration operation within the list, in which a minimal number of KV items are moved out of the most overloaded bucket (the source) to the most under-loaded bucket (the destination) until the remaining items in the source can fit in the bucket. The source bucket is removed from the list and we keep the list sorted. We then repeat the migration operation on the shorter list. The operation continues until either a list's source bucket is not overloaded or the list's destination bucket is also overloaded. To minimize the chance

[3]Interestingly the results are little affected by the key distribution. Even the uniform key distribution produces similar results.



(a) *100 B* (b) *200 B* (c) *300 B*

Figure 7: *Bucket load distribution after load balancing for HTables with different average item sizes.*

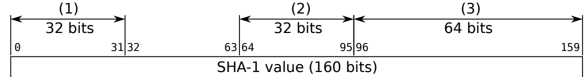


Figure 8: *Uses of a 160-bit SHA1 key: (1) the prefix is used for trie encoding; (2) The infix is used for sorting KV items in a bucket; and (3) the suffix is used for locating the KV items in an HTable.*

of having the second scenario, we set a limit on the aggregate size of KV items that can be stored in an HTable, which is 95% of the fixed HTable capacity (32 MB by default). This approach is effective. For example, with such a small reduction on usable capacity we have not observed a single item that is moved out of an over-loaded bucket but cannot be accommodated in an under-loaded bucket for HTables whose item sizes are 400 B on average and are uniformly distributed between 1 B and 800 B. Figure 7 shows the bucket load distribution after the load is balanced.

To handle the case of overflown items that cannot be accepted into any regular buckets, mostly due to excessively large KV items, during creation of a new HTable, LSM-trie sets up a special bucket to receive them. Items in the special bucket are fully indexed. The index is saved in the HTable file and is also cached in memory for efficiently locating the items. As the bucket is designed only for a few large KV items, its index should be of minimal size. Generally, workloads for accessing consistently large items (a few KBs or larger) should use SSTable-trie. In fact, such workloads do not pose a challenge on their metadata management in most KV stores.

There are several issues to address on the load balancing strategy. One is how to efficiently identify KV items overflown out of a bucket. To minimize the bookkeeping cost for the purpose, we use a hash function on the keys to rank KV items in a bucket and logically place them into the bucket according to their rankings. We then use the bucket capacity (4 KB) as the watermark. Any items that are across or above the watermark are considered as overflown items for migration. We only need to record the hash value for the item at the watermark, named **HashMark**, for future lookups to know whether an item has been migrated. For the hash function, we simply select a 32-bit infix in the 160-bit hashkey (e.g., from 64th bit to 95th bit), as illustrated in Figure 8. We

also record where the items are migrated (the destination bucket ID). A migrated item can be further migrated and searching for the item would need to walk over multiple buckets. To minimize the chance for an item to be repeatedly migrated, we tune the hash function by rotating the 32-bit infix by a particular number of bits, where the number is a function of bucket ID. In this way, different functions can be applied on different buckets, and an item is less likely to keep staying above buckets' watermarks for repeated migrations.

The metadata for each bucket about its overflown items comprise a source bucket ID (2 B), a migration destination ID (2 B), and a HashMark (4 B). They are stored in the bucket on the disk. A design issue is whether to cache the metadata in memory. If we cache every bucket's metadata, the cost would be comparable to the indices in SSTable, which records one key for each block (bucket). Actually it is not necessary to record all buckets' metadata if we do not require exactly one bucket read in an HTable lookup. As shown in Figure 5, distribution of overflown items over the buckets is highly skewed. So we only need to cache metadata for the most overloaded buckets (20% by default) and make lookup of these items be re-directed to their respective destination buckets without a disk read. In this way, with slightly increased disk reads LSM-trie can significantly reduce its cached metadata. For example, when KV items are of 100 B in average and their sizes are uniformly distributed between 1 B and 200 B, only 1.01 bucket reads per lookup are needed with only 14 KB (1792×8 B) of the metadata cached, about 1/10 of the size of an SSTable's indices.

Similar to LevelDB, LSM-trie maintains a Bloom filter for each bucket to quickly determine whether a KV item could be there. The migration of KV items out of a bucket does not require updating the bucket's Bloom filter, as these KV items still logically remain in the bucket and are only physically stored in other bucket(s). Their physical locations are later revealed through the bucket's migration-related metadata.

2.4.3 Clustering Bloom Filters for Efficient Access

LSM-trie does not assume that all Bloom filters can always be cached in memory. A Bloom filter at each (sub)-level needs to be inspected until a requested item is found. LSM-trie makes sure that all Bloom filters that are required to service a read request in a level but are not cached can be retrieved into memory with only one disk read. To this end LSM-trie gathers all Bloom filters associated with a column of buckets[4] at different sub-levels of an HTable container into a single disk block named

[4] As shown in Figure 9, the column of buckets refers to all buckets at the same position of respective HTables in a container.

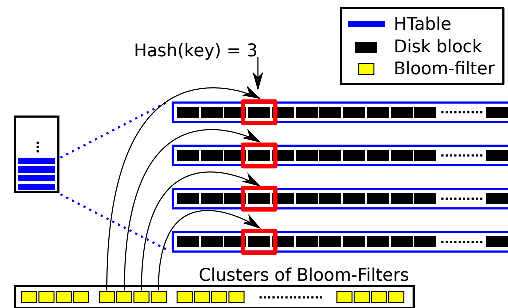


Figure 9: *Clustering Bloom filters*

bits/key	50 Levels	100 Levels	150 Levels
10	40.95%	81.90%	122.85%
12	15.70%	31.40%	47.10%
14	6.00%	12.00%	18.00%
16	2.30%	4.59%	6.89%
18	0.88%	1.76%	2.64%

Table 1: *Bloom filter false-positive rate.*

BloomCluster, as illustrated in Figure 9. Because the same hash function is applied across the sub-levels, a KV item can appear only in one particular column of buckets if it is in the container. In this way, only one disk read of Bloom filters is needed for a level.

While LSM-trie is designed to support up to a 10 TB store, its data is organized so that at most one read of metadata (Bloom filters) is required to access any item in the store. The prototyped LSM-trie system uses 32 MB HTables and an amplification factor (AF) of 8. The store has five levels. In the first four levels, LSM-trie uses both linear and exponential growth pattern. That is, each level consists of eight sub-levels.[5] All the Bloom filters for the first 32 sub-levels are of 4.5 GB, assuming a 64 B average item size and 16 bit Bloom filter per key. Adding metadata about item migration within individual HTables (up to 0.5 GB), LSM-trie needs up to only 5 GB memory to hold all necessary metadata. At the fifth level, which is the last level, LSM-trie uses only linear growth pattern. As one sub-level of this level has a capacity of 128 G, it needs 8 such sub-levels for the store to reach 1 TB, and 80 such sub-levels to reach 10 TB. All the sub-levels' Bloom filters are well clustered into a BloomCluster so that only one disk read of Bloom filter is required for a read request. Though the false positive rate increases with level count, it can be well capped by using additional bits per KV item, as shown in Table 1. When LSM-trie uses 16-bit-per-item Bloom filters, the false positive rate is only about 5% even for a 112-sub-level 10 TB KV store. In the worse case there are only 2.05 disk reads, one for a BloomCluster and 1.05 on average for data.

[5] Actual number of sub-levels in a level can change during compaction operations. It varies between 0 and 16 with an average of 8.

	SSD	HDD
Random Read 4 KB (IOPS)	52,400	70
Sequential Write (MB/s)	230	144
Sequential Read (MB/s)	298	138

Table 2: *Basic disk performance measurements.*

In the LSM-trie structure, multiple KV items of the same key, including special items for Delete operations, can simultaneously stay in different sub-levels of the last level without being merged as there are no merge-sort operations at this level. Among the items of the same key, only the item at the highest sub-level is alive and the others are considered as garbage. This may lead to under-utilized disk space, especially when the level contains substantial amount of garbage. To ameliorate the effect, we periodically sample a few random HTable containers and assess their average garbage ratio. When the ratio is larger than a threshold, we schedule garbage-collection operations in a container-by-container manner either periodically or when the system is not loaded.

3 Performance Evaluation

To evaluate LSM-trie's performance, we implement a prototype and extensively conduct experiments to reveal insights of its performance behaviors.

3.1 Experiment Setup

The experiments are run on a Dell CS23-SH server with two Intel Xeon L5410 4-core processors, 64 GB FB-DIMM memory, and 64-bit Linux 3.14. The SSD (Samsung 840 EVO, MZ-7TE1T0BW) has 1 TB capacity. Because of its limited storage capacity (1 TB), we install DRAM of moderate size on the computer (64 GB), a configuration equivalent to 256 GB memory with a 4 TB store. We also build a KV store on a hard disk, which is 3 TB Seagate Barracuda (ST3000DM001) with 64 MB cache and 7200 RPM. Table 2 lists the disks' performance measurements. As we can see, the hard disk's random read throughput is too small and it's not competitive considering SSD's rapidly dropping price. Therefore, we do not run read benchmarks on the hard disk. All experiments are run on the SSD(s) unless stated otherwise. In LSM-trie immediately after a table is written to the disk, we issue `fsync()` to persist its data.

In the evaluation, we compare LSM-trie with LevelDB [4], RocksDB (an optimized LevelDB from Facebook) [11], and SILT [24]. LSM-trie uses 32 MB HTables, LevelDB and RocksDB use 32 MB SSTables, and SILT uses 32 MB HashStore. We run SILT using its source code provided by its authors with its default setup [9]. We do not include experiments for SSTable-trie as its write performance is the same as LSM-trie, but

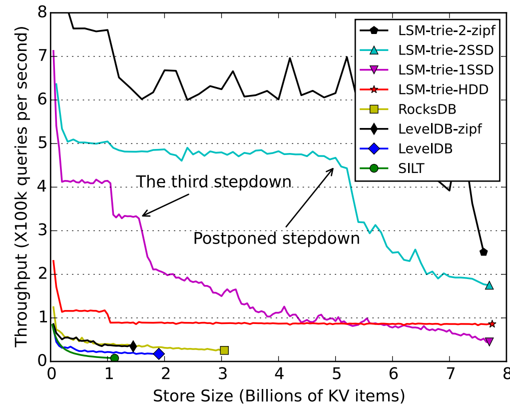


Figure 10: *Write throughput of different stores. For each store, the execution stops when either the store reaches 1TB or the run time reaches 24 hours, whichever occurs earlier.*

its read performance can be unacceptably worse than that of LevelDB when there are many levels and Bloom filters cannot be cached.

We use Yahoo's YCSB benchmark suite to generate read and write requests [18]. Average value size of the KV items is 100 B with a uniform distribution between 1 B to 200 B. The key size is 16 B. We use constant value size (100 B) for SILT as it does not support varied value size. By default, we use the uniform key distribution, as it represents the least locality and minimal overwrites in the workload, which helps increase a store's write pressure.[6]

3.2 Experiment Results

In this section we present and analyze experiment results for write and read requests.

3.2.1 Write Throughput

Figure 10 plots the write throughput, in terms of number of PUT queries served per second (QPS), for LSM-trie, LevelDB, RocksDB, and SILT with different store sizes, or numbers of KV items in the store. We have a number of interesting observations on the plots.

The LSM-trie store has throughput way higher than other stores. Even the throughput for LSM-trie on the hard disk (see the "LSM-trie-HDD" curve) more than doubles those of other stores on the SSD. It takes about 24 hours for LSM-trie to build a 1 TB store containing nearly 8 billions of small items on an HDD. As it is too slow for the other stores to reach the size of 1 TB within a reasonable time period, we stop their executions after they run for 24 hours. By estimation it would take RocksDB and LevelDB about 4–6 days and even longer time for SILT to build such a large store on the SSD.

[6] We do have a test for the Zipfian distribution in Section 3.2.

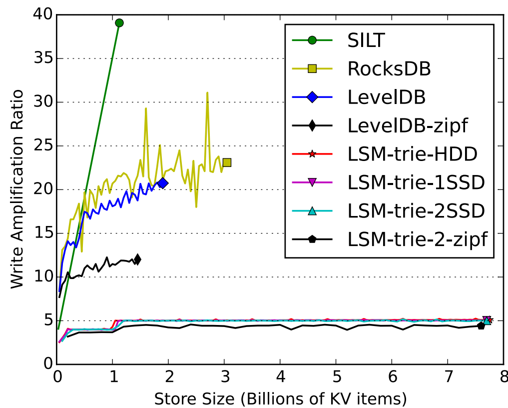


Figure 11: *Write amplification ratios of different stores. For each store, the execution stops when either the store reaches 1TB or the run time reaches 24 hours, whichever occurs earlier.*

Admittedly SILT is designed mainly to service read requests [24]. However, taking so long to build a large store is less desirable in the first place. To understand their big performance gaps, we draw the write amplification ratio (WAR) plots for the stores in Figure 11.

It's not a surprise to see SLIT's WAR increases almost linearly with the store size, as SILT does not adopt a multi-level organization. By maintaining a large SortedStore and merge-sorting much smaller HashStores into it, most of its compaction I/O is to access data in the SortedStore, and contributes to the WAR. While both LevelDB and RocksDB adopt LSM-tree's multi-level organization, its exponential growth pattern significantly compromises its WAR. The WAR curve of RocksDB is obtained by running its performance monitoring tool (`db_bench`). The curve exhibits large variations, mainly because of its choice of sampling points for performance measurements. While RocksDB generally has a higher WAR, its write throughput is higher than that of LevelDB because of its use of multiple threads to better utilize parallelism available in SSD and CPU. The WAR curves for LSM-trie ("LSM-trie-*" curves in Figure 11) have small jumps at about 0.12 and 1.0 billion items in the KV store, corresponding to the timings when the store grows into Levels 3 and 4, respectively (Figure 11). Once the store reaches its last level (Level 4), the WAR curves become flat at around 5 while the store increases up to 10 TB.

The write throughput curve for the hard disk ("LSM-trie-HDD") in Figure 10 has two step-downs, well matching the two jumps in its corresponding WAR curve. After the store reaches 1 billion items, its throughput does not reduce with the increase of the store. For LSM-trie on the SSD, we do see the first and second step-downs on the curve ("LSM-trie-1SSD" in Figure 10) corresponding to the two WAR jumps. However, we had been confused by the third step-down, as marked in Figure 10, when the store size reaches about 1.7 billion items or 210 GB. One might attribute this throughput loss to the garbage collection. However, we had made efforts to use large HTables (32 MB) and aligned them to the erase block boundaries. After investigation, it turns to be due to SSD's internal *static* wear-leveling.

As we know, frequency of data re-writing at different levels dramatically varies. The ratio of the frequencies between two adjacent levels (lower level vs. upper level) can be as high as 8. For data at Level 4 and at Level 0, the ratio of their re-write frequencies could be 4096 (8^4)! With such a large gap between the frequencies, dynamical wear-leveling is insufficient and SSD's FTL (Flash Translation Layer) has to proactively move data at the lower level(s) around to even out flash wear across the disk. The impact of the wear-levering becomes increasingly serious when more and more SSD's space is occupied. To confirm our speculation, we introduce a second SSD and move data at the two upper level (about only 2.5 GB) to it, and run LSM-trie on the two SSDs (see "LSM-trie-2SSD" in Figure 10). The third step-down is postponed to a significantly later time (from about 1.7 billion items to about 5.2 billion items). The new third step-down is caused by re-write frequency gaps among data at Levels 2, 3, and 4 in the first SSD. Using more SSDs and separating them onto different SSDs would eliminate the step-down. In practice, it is a viable solution to have a few small but wear-resistent SSDs (e.g., SLC SSD) to separate the first several levels of data.

We also issue write requests with the Zipfian key distribution to LSM-trie on two SSDs. It has a smaller WAR than those with the uniform key distribution (see "LSM-trie-2-zipf" in Figure 11), and higher throughput (see "LSM-trie-2-zipf" in Figure 10). Strong locality of the workload produces substantial overwrites, which are merged during the compactions. As a result, about one third of items are removed before they reach the last level, reducing write amplification and increasing throughput. The Zipfian distribution also allows LevelDB to significantly reduce its WAR (compare "LevelDB" and "LevelDB-zipf" in Figure 11) and to increase its write throughput (compare "LevelDB" and "LevelDB-zipf" in Figure 10).

In almost all scenarios, LSM-trie dramatically improves WAR, leading to significantly increased write throughput. The major reason of the improvements is the introduction of the linear growth pattern into the LSM tree and the adoption of the trie structure to enable it.

3.2.2 Performance of Read

Figures 12 and 13 plot the read throughput for various stores on one SSD with 64Gb and 4GB memory, respectively, except SILT. Keys of read requests are uniformly distributed. As explained, we cannot build a sufficiently

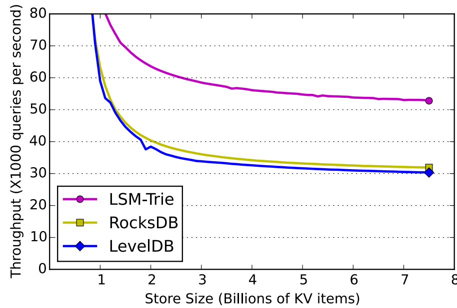


Figure 12: *Read throughput with 64 GB memory.*

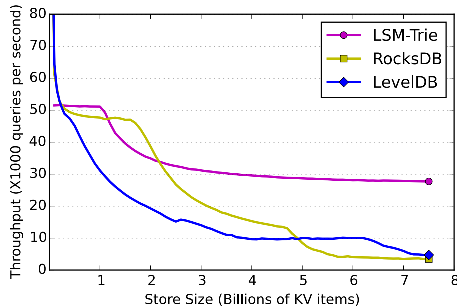


Figure 13: *Read throughput with 4 GB memory.*

large SILT store to measure its read performance. Instead, we will use the results reported in its paper for comparison [24]. To accelerate the building of the LevelDB and RocksDB stores, we use YCSB to generate a trace of write requests whose keys are sorted. The stores can then be quickly built without any compactions.

As shown in Figure 12, when the store size is relatively small (with fewer than about 1 billion KV items or 128 GB data), almost half of accessed data can be cached in memory and the throughput is very high (much higher than 80K QPS). This throughput is not explicitly shown in the figure, as it is less I/O related. LSM-trie has higher throughputs than LevelDB and RocksDB for both small and large store sizes. With a small store size, LSM-trie uses less memory to cache metadata and leaves more for caching data than other stores, producing higher hit ratios and read throughputs. When the store becomes larger, theh working set becomes larger due to uniform key distribution and the memory size becomes less relevant to the throughput. LSM-trie's higher throughputs with larger store are due to the alignment of its block to the SSD pages in its implementation. Without the alignment, one access of an SSTable-file's block may result in access of an additional page. For the following experiment we augment LevelDB and RocksDB by aligning their blocks to the SSD pages. LSM-trie's throughput with a large store (over 6 billions KV items) is around 96% of one SSD's raw read throughput in terms of number of 4 KB-blocks read per second. This is the same percentage reported in the SILT paper [24].

Considering the scenario where a server running a KV store may simultaneously run other application(s) demanding substantial memory resource, or where a KV store runs within a disk drive with small memory [8], we evaluate LSM-trie's performance with a constrained memory size. Figure 13 shows read throughput when the memory is only 4 GB [7]. Current LSM-trie's implementation always keeps metadata for the first four levels in the memory. More and more requests require one read of out-of-core metadata in addition to one read of data after the store grows beyond the first four levels. This is why the curve for LSM-trie starts to drop beyond 1.2-billion-item store size. The throughput curves of LevelDB and RocksDB also drop with the increase of store size. They drop much more than that of LSM-trie. RocksDB's throughput is higher than that of LevelDB initially, as it caches more metadata by giving metadata a caching priority higher than data.

Latency Percentile	5% read	50% read	95% read
95%	690 μs	790 μs	700 μs
99%	860 μs	940 μs	830 μs

Table 3: *Read Latency under mixed read/write workload.*

Our measurements show that all requests can be completed in 1 ms, and its 99% percentile latency is 0.92 ms. To know how read latency is affected by concurrent write requests, we list the 95% and 99% percentile latencies for different percentages of read requests among all the read/write requests in Table 3. The read latencies are not sensitive to write intensity. The KV store store many small items in write requests into one block while each read request has to retrieve an entire block. Thanks to the much reduced write compaction in LSM-trie, intensity of write requests has a small impact on read latency.

4 Related Work

Key-value stores have become an increasingly popular data management system with its sustained high performance with workloads challenging other systems, such as those generating a huge number of small data items. Most related works aim for efficient writes and reads.

4.1 Efforts on Supporting Efficient Writes

Most KV stores support fast writes/updates by using log-based write, such as FAWN [12], FlashStore [20], SkimpyStash [21], SILT [24], LevelDB [4], and bLSM [29]. Though log-appending is efficient for admitting new data, it is not sufficient for high write efficiency. There can be significant writes caused by internal data re-organization and their efficiency can be critical to the write throughput observed by users. A primary objective of the re-organization is to remove garbage from

[7] Note that write performance is not affected by the small memory.

the log. Some systems, such as FAWN, FlashStore, and SkimpyStash, focus mostly on this objective and incurs a relatively small number of additional writes. Though these systems are efficient for serving writes, they leave the data not well organized, and produce a large metadata set leading to slow reads with relatively small memory.

Another group of systems, such as LevelDB, SILT, and bLSM, aim to build a fully organized data structure—one (almost) sorted list of KV items. This is apparently ideal for reducing metadata size and facilitating fast reads. It is also essential for a scalable system. However, it can generate a very large write amplification. The issue quickly deteriorates with the growth of the store. To address the issue, RocksDB compacts more than two contiguous levels at once intending to sort and push data faster to the lower level [11]. However, the improvement is limited as the amplification is fundamentally due to the difference of the data set sizes at different levels. To mitigate the compaction cost, TokuDB uses a Fractal Tree, in which data is pushed to its next level by simply being appended into log files at corresponding tree nodes [23, 15]. Without well sorting its data, TokuDB has to maintain a much larger index, leading to larger memory demand and/or additional disk access for metadata. In contrast, with the support of the trie structure and use of linear growth pattern, LSM-trie minimizes write amplification.

4.2 Efforts on Supporting Efficient Reads

Read efficiency is mostly determined by two factors. One is metadata size and the other is the efficiency of retrieving metadata from the disk. Both determine how many disk reads are needed to locate a requested KV item.

As SILT has a fully sorted list of KV items and uses a highly compact index representation, it produces very small metadata [24]. In contrast, LevelDB's metadata can be much larger as they include both indices and Bloom filters. It may take multiple reads for LevelDB to load its out-of-memory metadata. FAWN [12] and FlashStore [20] have very large metadata as they directly store pointers to the on-disk items, especially when the items are small and the store is large. SkimpyStash stores hash table buckets on the disk, essentially leaving most metadata on the disk and may require many disk reads of metadata to locate the data [21]. In contrast, LSM-trie substantially reduces metadata by removing almost all indices. It requires at most one metadata read for each read request with its well clustered metadata.

4.3 Other Related Works

Sharding (or partitioning), as a technique to distribute heavy system load such as large working sets and intensive I/O requests across nodes in a cluster, has been widely used in database systems and KV stores [6, 5, 2]. It has been proposed as a potential method for reducing merge (or compaction) overhead by maintaining multiple smaller store instances (shards) at a node [24]. However, if the number of shards is moderate (fewer than one hundred) at a node, each shard has to grow into four or larger number of levels when the store becomes large. Accordingly write amplification cannot be substantially reduced. Meanwhile, because memory demand, including MemTables and metadata, is about proportional to the number of shards, using many shards increase pressure on memory. In contrast, LSM-trie fundamentally addresses the issue by improving store growth pattern to minimize compaction cost without concerns of sharding.

Being aware of large compaction cost in LevelDB, VT-Tree opportunistically looks for any block at a level whose key range does not overlap with that of blocks at another level during merge-sorting of the two levels' KV items [30]. Effectiveness of this method relies on probability of having non-overlapping blocks. For workloads with small items, there are a large number of keys in a block, reducing the probability. Though it had been reported that this method can reduce write amplification by about $\frac{1}{3}$ to $\frac{2}{3}$, it is far from enough. In contrast, LSM-trie reduces the amplification by up to an order of magnitude.

While LSM-trie trades some disk space (around 5%) for much improved performance, Yu et al. proposed a method to improve performance of the disk array by trading capacity for performance [32]. They trade 50% of the disk space for a throughput improvement of 160%.

5 Conclusions

In this paper we describe LSM-trie, a key-value store designed to manage a very large data set in terms of both its data volume and KV item count. By introducing linear growth pattern, LSM-trie minimizes compaction cost for LSM-tree-based KV systems. As our extensive experiments demonstrate, LSM-trie can manage billions of KV items with a write amplification of only five. By design it can manage a store of up to 10 TB. LSM-trie can service a read request with only two SSD reads even when over 90% of the bloom-filters is not in the memory. Furthermore, with a second small SSD (only 20 GB) to store the bloom-filters, the overall throughput can reach the peak throughput of the raw device (50 K QPS vs. 52 K IOPS), and 99% of its read latency is below 1 ms.

6 Acknowledgments

This work was supported by US National Science Foundation under CAREER CCF 0845711 and CNS 1217948.

References

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

[2] Consider the apache cassandra database. http://goo.gl/tX37h3.

[3] How much text versus metadata is in a tweet? http://goo.gl/EBFIFs.

[4] Leveldb: A fast and lightweight key/value database library by google. https://code.google.com/p/leveldb/.

[5] mongodb. http://goo.gl/sQdYfo.

[6] Mysql cluster: Scalability. http://goo.gl/SIfvfe.

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

[8] Seagate kinetic HDD. http://goo.gl/pS9bs1.

[9] Silt: A memory-efficient, high-performance key-value store. https://github.com/silt/silt.

[10] Storing hundreds of millions of simple key-value pairs in redis. http://goo.gl/ieeU17.

[11] Under the hood: Building and open-sourcing rocksdb. http://goo.gl/9xulVB.

[12] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. Fawn: A fast array of wimpy nodes. *Commun. ACM 54*, 7 (July 2011), 101–109.

[13] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[14] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2010), OSDI'10, USENIX Association, pp. 1–8.

[15] Bender, M. A., Farach-Colton, M., Fineman, J. T., Fogel, Y. R., Kuszmaul, B. C., and Nelson, J. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2007), SPAA '07, ACM, pp. 81–92.

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

[17] Brantner, M., Florescu, D., Graf, D., Kossmann, D., and Kraska, T. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2008), SIGMOD '08, ACM, pp. 251–264.

[18] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[19] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[20] Debnath, B., Sengupta, S., and Li, J. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 1414–1425.

[21] Debnath, B., Sengupta, S., and Li, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 25–36.

[22] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[23] Kuszmaul, B. C. How fractal trees work. http://goo.gl/PG3kr4.

[24] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 1–13.

[25] Lomet, D. Replicated indexes for distributed data. In *In PDIS* (1996), pp. 108–119.

[26] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[27] 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.

[28] Pirzadeh, P., Tatemura, J., and Hacigümüs, H. Performance evaluation of range queries in key value stores. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May 2011 - Workshop Proceedings* (2011), pp. 1092–1101.

[29] Sears, R., and Ramakrishnan, R. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 217–228.

[30] Shetty, P., Spillane, R., Malpani, R., Andrews, B., Seyster, J., and Zadok, E. Building workload-independent storage with vt-trees. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2013), FAST'13, USENIX Association, pp. 17–30.

[31] Vo, H. T., Wang, S., Agrawal, D., Chen, G., and Ooi, B. C. Logbase: A scalable log-structured database system in the cloud. *Proc. VLDB Endow. 5*, 10 (June 2012), 1004–1015.

[32] Yu, X., Gum, B., Chen, Y., Wang, R. Y., Li, K., Krishnamurthy, A., and Anderson, T. E. Trading capacity for performance in a disk array. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4* (2000), USENIX Association, pp. 17–17.

[33] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 2–2.