WiscKey: Separating Keys from Values in SSD-Conscious Storage

LANYUE LU, THANUMALAYAN SANKARANARAYANA PILLAI,
HARIHARAN GOPALAKRISHNAN, ANDREA C. ARPACI-DUSSEAU,
and REMZI H. ARPACI-DUSSEAU, University of Wisconsin, Madison

We present WiscKey, a persistent LSM-tree-based key-value store with a performance-oriented data layout that separates keys from values to minimize I/O amplification. The design of WiscKey is highly SSD optimized, leveraging both the sequential and random performance characteristics of the device. We demonstrate the advantages of WiscKey with both microbenchmarks and YCSB workloads. Microbenchmark results show that WiscKey is 2.5× to 111× faster than LevelDB for loading a database (with significantly better tail latencies) and 1.6× to 14× faster for random lookups. WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads.

CCS Concepts: • **General and reference → Performance**; • **Information systems → Key-value stores**; • **Software and its engineering → File systems management**; *Operating systems*; *Input/output;*

Additional Key Words and Phrases: LevelDB, WiscKey, flash-based SSDs

ACM Reference Format:
Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating keys from values in SSD-conscious storage. ACM Trans. Storage 13, 1, Article 5 (March 2017), 28 pages.
DOI: http://dx.doi.org/10.1145/3033273

1. INTRODUCTION

Persistent key-value stores play a critical role in a variety of modern data-intensive applications, including web indexing [Chang et al. 2006; Sanjay Ghemawat and Jeff Dean 2011], e-commerce [DeCandia et al. 2007], data deduplication [Anand et al. 2010; Debnath et al. 2010], photo stores [Beaver et al. 2010], cloud data [Lai et al. 2015], social networking [Armstrong et al. 2013; Dong 2015; Sumbaly et al. 2012], online gaming [Debnath et al. 2011], messaging [George 2011; Harter et al. 2014], and online advertising [Cooper et al. 2008]. By enabling efficient insertions, point lookups,

The research herein was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, as well as generous donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Seagate, Samsung, Veritas, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

This article is an extended version of a FAST '16 paper by Lu et al. [2016]. The additional material here includes a detailed study of tail latency (for both reads and writes) and LSM-tree size, more thorough descriptions of LSM-tree operation and basic APIs, more graphics depicting key data structures to aid in understanding, improved citations and related work, new future work and conclusions, and many other small edits and updates.

Authors' addresses: L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, 1210 W. Dayton St., Madison, WI 53706; emails: lu.lanyue@gmail.com, madthanu@cs.wisc.edu, hgopalakris2@wisc.edu, dusseau@cs.wisc.edu, remzi@cs.wisc.edu.

DOI: http://dx.doi.org/10.1145/3033273

and range queries, key-value stores serve as the foundation for this growing group of important applications.

For write-intensive workloads, key-value stores based on Log-Structured Merge-Trees (LSM-trees) [ONeil et al. 1996] have become the state of the art. Various distributed and local stores built on LSM-trees are widely deployed in large-scale production environments, such as BigTable [Chang et al. 2006] and LevelDB [Sanjay Ghemawat and Jeff Dean 2011] at Google; Cassandra [Lakshman and Malik 2009], HBase [Harter et al. 2014], and RocksDB [Dong 2015] at Facebook; PNUTS [Cooper et al. 2008] at Yahoo!; and Riak [Redmond 2013] at Basho. The main advantage of LSM-trees over other indexing structures (such as B-trees) is that they maintain sequential access patterns for writes [Athanassoulis et al. 2015]. Small updates on B-trees may involve many random writes and are hence not efficient on either solid-state storage devices or hard-disk drives.

To deliver high write performance, LSM-trees batch key-value pairs and write them sequentially. Subsequently, to enable efficient lookups (for both individual keys and range queries), LSM-trees continuously read, sort, and write key-value pairs in the background, thus maintaining keys and values in sorted order. As a result, the same data is read and written multiple times throughout its lifetime; as we show later (Section 2), this I/O amplification in typical LSM-trees can reach a factor of 50× or higher [Harter et al. 2014; Marmol et al. 2015; Wu et al. 2015].

The success of LSM-based technology is tied closely to its usage in classic hard-disk drives (HDDs) [Arpaci-Dusseau and Arpaci-Dusseau 2014]. In HDDs, random I/Os are over 100× slower than sequential ones [Arpaci-Dusseau and Arpaci-Dusseau 2014; ONeil et al. 1996]; thus, performing additional sequential reads and writes to continually sort keys and enable efficient lookups represents an excellent tradeoff.

However, the storage landscape is quickly changing, and modern solid-state storage devices (SSDs) are supplanting HDDs in many important use cases [Arpaci-Dusseau and Arpaci-Dusseau 2014]. As compared to HDDs, SSDs are fundamentally different in their performance and reliability characteristics; when considering key-value storage system design, we believe the following three differences are of paramount importance. First, the difference between random and sequential performance is not nearly as large as with HDDs; thus, an LSM-tree that performs a large number of sequential I/Os to reduce later random I/Os may be wasting bandwidth needlessly. Second, SSDs have a large degree of internal parallelism; an LSM built atop an SSD must be carefully designed to harness said parallelism [Wang et al. 2014]. Third, SSDs can wear out through repeated writes [Lee et al. 2015; Min et al. 2012]; the high write amplification in LSM-trees can significantly reduce device lifetime. As we will show in this article (Section 4), the combination of these factors greatly impacts LSM-tree performance on SSDs, reducing throughput by 90% and increasing write load by a factor over 10. While replacing an HDD with an SSD underneath an LSM-tree does improve performance, with current LSM-tree technology, the SSD's true potential goes largely unrealized.

In this article, we present WiscKey, an SSD-conscious persistent key-value store derived from the popular LSM-tree implementation, LevelDB. The central idea behind WiscKey is the separation of keys and values [Nyberg et al. 1994]; only keys are kept sorted in the LSM-tree, while values are stored separately in a log. In other words, we decouple key sorting and garbage collection in WiscKey, whereas LevelDB bundles them together. This simple technique can significantly reduce write amplification by avoiding the unnecessary movement of values while sorting. Furthermore, the size of the LSM-tree is noticeably decreased, leading to fewer device reads and better caching during lookups. WiscKey retains the benefits of LSM-tree technology, including excellent insert and lookup performance, but without excessive I/O amplification.

Separating keys from values introduces a number of challenges and optimization opportunities. First, range query (scan) performance may be affected because values are not stored in sorted order anymore. WiscKey solves this challenge by using the abundant internal parallelism of SSD devices. Second, WiscKey needs garbage collection to reclaim the free space used by invalid values. WiscKey proposes an online and lightweight garbage collector, which only involves sequential I/Os and impacts the foreground workload minimally. Third, separating keys and values makes crash consistency challenging; WiscKey leverages an interesting property in modern file systems, whose appends never result in garbage data on a crash, to realize crash consistency correctly and efficiently. As a result, WiscKey delivers high performance while providing the same consistency guarantees as found in modern LSM-based systems.

We compare the performance of WiscKey with LevelDB [Sanjay Ghemawat and Jeff Dean 2011] and RocksDB [Dong 2015], two popular LSM-tree key-value stores. For most workloads, WiscKey performs significantly better. With LevelDB's own microbenchmark, WiscKey is 2.5× to 111× faster than LevelDB for loading a database (with notably better tail latencies), depending on the size of the key-value pairs; for random lookups, WiscKey is 1.6× to 14× faster than LevelDB. WiscKey's performance is not always better than standard LSM-trees; if small values are written in random order and a large dataset is range-queried sequentially, WiscKey performs worse than LevelDB. However, this workload does not reflect real-world use cases (which primarily use shorter range queries) and can be improved by log reorganization. Under YCSB macrobenchmarks [Cooper et al. 2010] that reflect real-world use cases, WiscKey is faster than both LevelDB and RocksDB in all six YCSB workloads, and follows a trend similar to the load and random lookup microbenchmarks.

The rest of the article is organized as follows. We first describe background and motivation (Section 2). We then explain the design of WiscKey (Section 3) and analyze its performance (Section 4). We next describe related work (Section 5) and finally conclude (Section 6).

2. BACKGROUND AND MOTIVATION

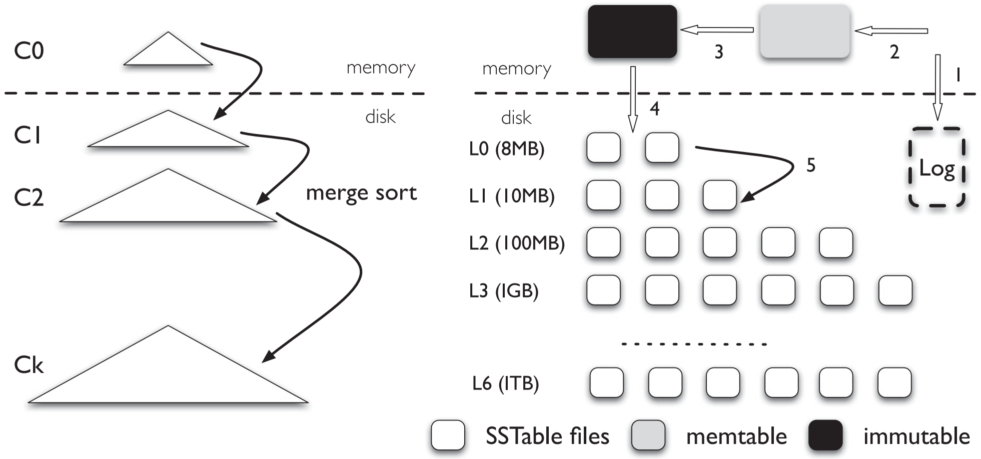
In this section, we first describe the concept of a Log-Structured Merge-tree (LSM-tree). Then, we explain the design of LevelDB, a popular key-value store based on LSM-tree technology. We investigate read and write amplification in LevelDB. Finally, we describe the characteristics of modern storage hardware.

2.1. Log-Structured Merge-Tree

An LSM-tree is a persistent structure that provides efficient indexing for a key-value store with a high rate of inserts and deletes [ONeil et al. 1996]. An LSM-tree defers and batches data writes into large chunks to use the high sequential bandwidth of hard drives. Since random writes are nearly two orders of magnitude slower than sequential writes on hard drives, LSM-trees provide better write performance than traditional B-trees, which require random accesses.

An LSM-tree consists of a number of components of exponentially increasing sizes, C_0 to C_k, as shown in Figure 1. The C_0 component is a memory-resident update-in-place sorted tree, while the other components C_1 to C_k are disk-resident append-only B-trees.

During an insert in an LSM-tree, the inserted key-value pair is appended to an on-disk sequential log file, so as to enable recovery in case of a crash. Then, the key-value pair is added to the in-memory C_0, which is sorted by keys; C_0 allows efficient lookups and scans on recently inserted key-value pairs. Once C_0 reaches its size limit, it will be merged with the on-disk C_1 in an approach similar to merge sort; this process is known as *compaction*. The newly merged tree will be written to disk sequentially, replacing the old version of C_1. Compaction (i.e., merge sorting) also happens for on-disk components, when each C_i reaches its size limit. Note that compactions are only performed between adjacent levels (C_i and C_{i+1}), and they can be executed asynchronously in the background.



(a) LSM-tree

(b) LevelDB

Fig. 1. LSM-tree and LevelDB architecture. This figure shows the standard LSM-tree and LevelDB architecture. For LevelDB, inserting a key-value pair goes through many steps: (1) the log file, (2) the memtable, (3) the immutable memtable, (4) an SSTable in L0, and (5) compacted to further levels.

To serve a lookup operation, LSM-trees may need to search multiple components. Note that C_0 contains the freshest data, followed by C_1, and so on. Therefore, to retrieve a key-value pair, the LSM-tree searches components starting from C_0 in a cascading fashion until it locates the desired data in the smallest component C_i. Compared with B-trees, LSM-trees may need multiple reads for a point lookup. Hence, LSM-trees are most useful when inserts are more common than lookups [Athanassoulis et al. 2015; ONeil et al. 1996].

2.2. LevelDB

LevelDB is a widely used key-value store based on LSM-trees and inspired by BigTable [Chang et al. 2006; Ghemawat and Dean 2011]. LevelDB supports range queries, snapshots, and other features that are useful in modern applications. In this section, we briefly describe the core design of LevelDB.

The LevelDB interface is a basic key-value store API, with certain richer functions including methods to create batches, snapshots, and iterators. The main methods are `Put()`, `Delete()`, `Get()`, `Write()`, `NewIterator()`, `GetSnapshot()`, and `CompactRange()`. The `Write()` operation is used to batch multiple writes together; internally, `Put()` and `Delete()` operations are implemented as batched writes. Snapshots are used to save previous versions of values for different keys. On each write, a sequence number is generated; garbage collection for earlier (nonlive) snapshots is performed during compaction. The compaction only retains key-value pairs with sequence numbers that are greater than or equal to the oldest live snapshot (i.e., the snapshot that was marked by `GetSnapshot` and not yet released).

Range queries are implemented through an iterator-based interface, with `Next()` and `Prev()` methods for scanning and `Seek()`, `SeekToFirst()`, and `SeekToLast()` methods for jumping to specific keys. The `CompactRange()` operation gives users the ability to trigger compactions before the automated compaction (discussed later) begins.

The overall architecture of LevelDB is shown in Figure 1. The main data structures in LevelDB are an on-disk log file, two in-memory sorted skiplists (*memtable* and *immutable memtable*), and seven levels (L_0 to L_6) of on-disk Sorted String Table (*SSTable*)

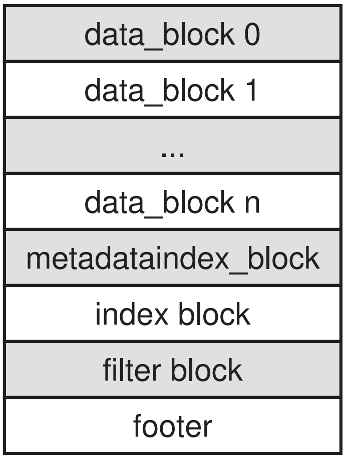


Fig. 2. SSTable structure. The figure depicts the logical layout of an SSTable file.

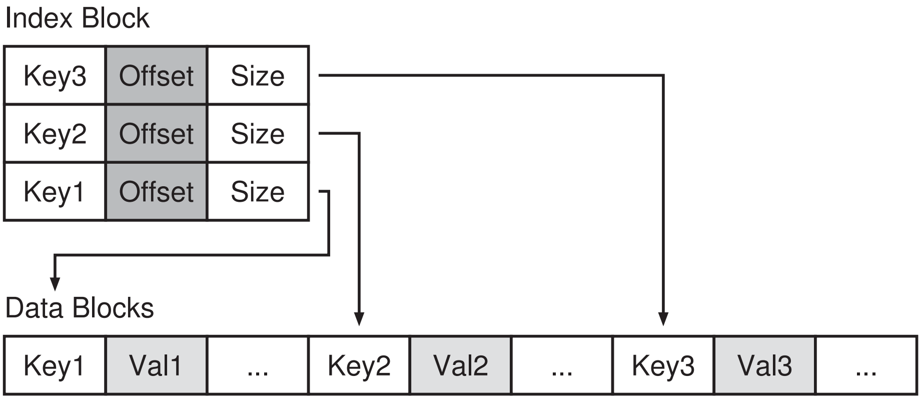


Fig. 3. Data and index blocks. Structure of the data and index blocks of an SSTable file.

files. LevelDB initially stores inserted key-value pairs in a log file and the in-memory *memtable*. Once the memtable is full, LevelDB switches to a new memtable and log file to handle further inserts from the user; in the background, the previous memtable is converted into an immutable memtable, and a compaction thread then flushes it to disk, generating a new SSTable file at level 0 (L_0) with a rough size of 2MB; the previous log file is subsequently discarded.

The SSTable file organizes all data in the form of a sequence of triplets: `<block data, type, CRC>`. The `type` refers to the compression status of the data. While this sequence of triplets is the actual layout of the file, the logical contents of these files, as shown in Figure 2, are a set of data blocks, a meta-index block, an index block, an optional filter block, and a footer.

Data blocks consist of a prefix-compressed set of key-value pairs as shown in Figure 3. For a fixed number of key-value pairs (16 by default), LevelDB stores only a suffix of each key, ignoring the prefix it shares with the previous key. Each such sequence is followed by a restart point where the full key is stored. The end of the data block consists of the offsets to all the restart points, enabling a quick binary-search-based lookup for keys. Index blocks consist of `<key, offset, size>` triplets for each data block, where the key points to the first key in the block. Since entries in the SSTable are in key order, this provides an easy way to identify which block to read during lookup. The meta-index block consists of a pointer (i.e., an offset) to the filter block. The filter block consists of a Bloom filter for the file; it can indicate absence of a key and removes the need for expensive seeks and scans through the index and data blocks. Finally, the footer block consists of pointers to the meta-index and index blocks.

The size of all files in each level is limited and increases by a factor of 10 with the level number. For example, the size limit of all files at L_1 is 10MB, while the limit of L_2 is 100MB. To maintain the size limit, once the total size of a level L_i exceeds its limit, the compaction thread will choose one file from L_i, merge sort with all the overlapped files of L_{i+1}, and generate new L_{i+1} SSTable files. The compaction thread continues until all levels are within their size limits. During compaction, LevelDB ensures that all files in a particular level (except L_0) do not overlap in their key ranges; keys in files of L_0 can overlap with each other since they are directly flushed from the memtable.

To serve a lookup operation, LevelDB searches the memtable first and immutable memtable next, and then files L_0 to L_6 in order. For levels other than 0, a MANIFEST file stores the smallest and largest key in each SSTable file in those levels. This is used during lookup to identify the file that contains a key in a specific level. The number of file searches required to locate a random key is bounded by the maximum number of levels, since keys do not overlap between files within a single level, except in L_0. Since files in L_0 can contain overlapping keys, a lookup may search multiple files at L_0. To avoid a large lookup latency, LevelDB throttles foreground write traffic if the number of files at L_0 is bigger than eight, in order to wait for the compaction thread to compact some files from L_0 to L_1.

2.3. Write and Read Amplification

Write and read amplification are major problems in LSM-trees such as LevelDB. Write (read) amplification is defined as the ratio between the amount of data written to (read from) the underlying storage device and the amount of data requested by the user. In this section, we analyze the write and read amplification in LevelDB.

To achieve mostly sequential disk access, LevelDB writes more data than necessary (although still sequentially); that is, LevelDB has high write amplification. Since the size limit of L_i is 10 times that of L_{i-1}, when merging a file from L_{i-1} to L_i during compaction, LevelDB may read up to 10 files from L_i in the worst case and write back these files to L_i after sorting. Therefore, the write amplification of moving a file across two levels can be up to 10. For a large dataset, since any newly generated table file can eventually migrate from L_0 to L_6 through a series of compaction steps, write amplification can be over 50 (10 for each gap between L_1 to L_6).

Read amplification has been a major problem for LSM-trees due to tradeoffs made in the design. There are two sources of read amplification in LevelDB. First, to look up a key-value pair, LevelDB may need to check multiple levels. In the worst case, LevelDB needs to check eight files in L_0 and one file for each of the remaining six levels: a total of 14 files. Second, to find a key-value pair within an SSTable file, LevelDB needs to read multiple metadata blocks within the file. Specifically, the amount of data actually read is given by `(index block + bloom-filter blocks + data block)`. For example, to look up a 1KB key-value pair, LevelDB needs to read a 16KB index block, a 4KB Bloom filter block, and a 4KB data block—in total, 24KB. Therefore, considering the 14 SSTable files in the worst case, the read amplification of LevelDB is $24 \times 14 = 336$. Smaller key-value pairs will lead to an even higher read amplification.

To measure the amount of amplification seen in practice with LevelDB, we perform the following experiment. We first load a database with 1KB key-value pairs and then look up 100,000 entries from the database; we use two different database sizes for the

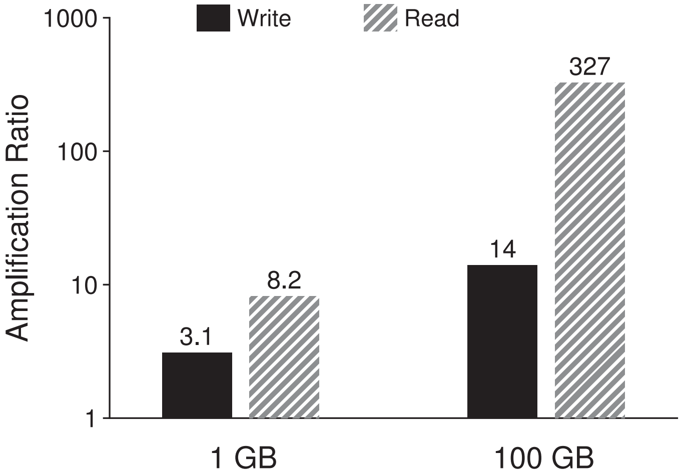


Fig. 4. Write and read amplification. This figure shows the write amplification and read amplification of LevelDB for two different database sizes, 1GB and 100GB. Key size is 16B and value size is 1KB.

initial load and choose keys randomly from a uniform distribution. Figure 4 shows write amplification during the load phase and read amplification during the lookup phase. For a 1GB database, write amplification is 3.1, while for a 100GB database, write amplification increases to 14. Read amplification follows the same trend: 8.2 for the 1GB database and 327 for the 100GB database. The reason write amplification increases with database size is straightforward. With more data inserted into a database, the key-value pairs will more likely travel further along the levels; in other words, LevelDB will write data many times when compacting from low levels to high levels. However, write amplification does not reach the worst case predicted previously, since the average number of files merged between levels is usually smaller than the worst case of 10. Read amplification also increases with the dataset size, since for a small database, all the index blocks and Bloom filters in SSTable files can be cached in memory. However, for a large database, each lookup may touch a different SSTable file, paying the cost of reading index blocks and Bloom filters each time.

It should be noted that the high write and read amplifications are a justified tradeoff for hard drives. As an example, for a given hard drive with a 10ms seek latency and a 100MB/s throughput, the approximate time required to access a random 1K of data is 10ms, while that for the next sequential block is about 10μs—the ratio between random and sequential latency is 1,000:1. Hence, compared to alternative data structures such as B-trees that require random write accesses, a sequential-write-only scheme with write amplification less than 1,000 will be faster on a hard drive [ONeil et al. 1996; Sears and Ramakrishnan 2012]. On the other hand, the read amplification for LSM-trees is still comparable to B-trees. For example, considering a B-tree with a height of five and a block size of 4KB, a random lookup for a 1KB key-value pair would require accessing six blocks, resulting in a read amplification of 24.

2.4. Fast Storage Hardware

Many modern servers adopt SSD devices to achieve high performance. Similar to hard drives, random writes are considered harmful also in SSDs [Arpaci-Dusseau and Arpaci-Dusseau 2014; Kim et al. 2012; Lee et al. 2015; Min et al. 2012] due to their unique erase-write cycle and expensive garbage collection. Although initial random-write performance for SSDs is good, the performance can significantly drop after the

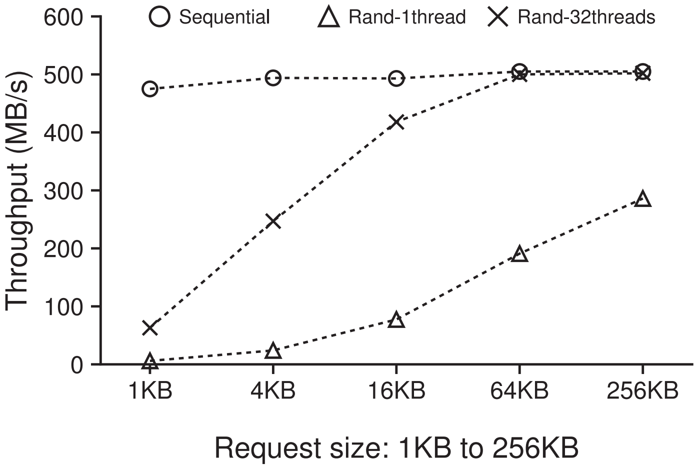


Fig. 5. Sequential and random reads on SSD. This figure shows the sequential and random read performance for various request sizes on a modern SSD device. All requests are issued to a 100GB file on ext4.

reserved blocks are utilized. The LSM-tree characteristic of avoiding random writes is hence a natural fit for SSDs; many SSD-optimized key-value stores are based on LSM-trees [Dong 2015; Shetty et al. 2013; Wang et al. 2014; Wu et al. 2015].

However, unlike hard drives, the relative performance of random reads (compared to sequential reads) is significantly better on SSDs; furthermore, when random reads are issued concurrently in an SSD, the aggregate throughput can match sequential throughput for some workloads [Chen et al. 2011]. As an example, Figure 5 shows the sequential and random read performance of a 500GB Samsung 840 EVO SSD, for various request sizes. For random reads by a single thread, the throughput increases with the request size, reaching half the sequential throughput for 256KB. With concurrent random reads by 32 threads, the aggregate throughput matches sequential throughput when the size is larger than 16KB. For more high-end SSDs, the gap between concurrent random reads and sequential reads is much smaller [Fusion-IO 2015; Marmol et al. 2015].

As we showed in this section, LSM-trees have a high write and read amplification, which is acceptable for hard drives. Using LSM-trees on a high-performance SSD may waste a large percentage of device bandwidth with excessive writing and reading. In this article, our goal is to improve the performance of LSM-trees on SSD devices to efficiently exploit device bandwidth.

3. WISCKEY

The previous section explained how LSM-trees maintain sequential I/O access by increasing I/O amplification. While this tradeoff between sequential I/O access and I/O amplification is justified for traditional hard disks, LSM-trees are not optimal for modern hardware utilizing SSDs. In this section, we present the design of WiscKey, a key-value store that minimizes I/O amplification on SSDs.

To realize an SSD-optimized key-value store, WiscKey includes four critical ideas. First, WiscKey separates keys from values, keeping only keys in the LSM-tree and the values in a separate log file. Second, to deal with unsorted values (which necessitate random access during range queries), WiscKey uses the parallel random-read characteristic of SSD devices. Third, WiscKey utilizes unique crash consistency and garbage collection techniques to efficiently manage the value log. Finally, WiscKey optimizes performance by removing the LSM-tree log without sacrificing consistency, thus reducing system-call overhead from small writes.

3.1. Design Goals

WiscKey is a single-machine persistent key-value store, derived from LevelDB. It can be deployed as the storage engine for a relational database (e.g., MySQL) or a distributed key-value store (e.g., MongoDB). It provides the same API as LevelDB, including `Put(key, value)`, `Get(key)`, `Delete(key)`, and `Scan(start, end)`. The design of WiscKey follows these main goals.

Low write amplification. Write amplification introduces extra unnecessary writes. Although SSD devices have higher bandwidth compared to hard drives, large write amplification can consume most of the write bandwidth (over 90% is not uncommon) and decrease the SSD's lifetime due to limited erase cycles. Therefore, it is important to minimize write amplification to improve performance and device lifetime.

Low read amplification. Large read amplification causes two problems. First, the throughput of lookups is significantly reduced by issuing multiple reads for each lookup. Second, the large amount of data loaded into memory decreases the efficiency of the cache. WiscKey targets a small read amplification to speed up lookups.

SSD optimized. WiscKey is optimized for SSD devices by matching its I/O patterns with the performance characteristics of SSD devices. Specifically, sequential writes and parallel random reads are employed so that applications can fully utilize device bandwidth.

Feature-rich API. WiscKey aims to support modern features that have made LSM-trees popular, such as range queries and snapshots. Range queries allow scanning a contiguous sequence of key-value pairs. Snapshots allow capturing the state of the database at a particular time and then performing lookups on the state.

Realistic key-value sizes. Keys are usually small in modern workloads (e.g., 16B) [Anand et al. 2010; Andersen et al. 2009; Atikoglu et al. 2015; Debnath et al. 2010; Lim et al. 2011], though value sizes can vary widely (e.g., 100B to larger than 4KB) [Ahn et al. 2016; Atikoglu et al. 2015; Debnath et al. 2010; Golan-Gueta et al. 2015; Lai et al. 2015; Sears and Ramakrishnan 2012]. WiscKey aims to provide high performance for this realistic set of key-value sizes.

3.2. Key-Value Separation

The major performance cost of LSM-trees is the compaction process, which constantly sorts SSTable files. During compaction, multiple files are read into memory, sorted, and written back, which could significantly affect the performance of foreground workloads. However, sorting is required for efficient retrieval; with sorting, range queries (i.e., scans) result mostly in sequential access to multiple files, while point queries require accessing at most one file at each level.

WiscKey is motivated by a simple revelation. Compaction only needs to sort keys, while values can be managed separately [Nyberg et al. 1994]. Since keys are usually smaller than values, compacting only keys could significantly reduce the amount of data needed during the sorting. In WiscKey, only the location of the value is stored in the LSM-tree with the key, while the actual values are stored elsewhere in an SSD-friendly fashion. With this design, for a database with a given size, the size of the LSM-tree of WiscKey is much smaller than that of LevelDB. The smaller LSM-tree can remarkably reduce the write amplification for modern workloads that have a moderately large value size. For example, assuming a 16B key, a 1KB value, and a write amplification of 10 for keys (in the LSM-tree) and 1 for values, the effective write

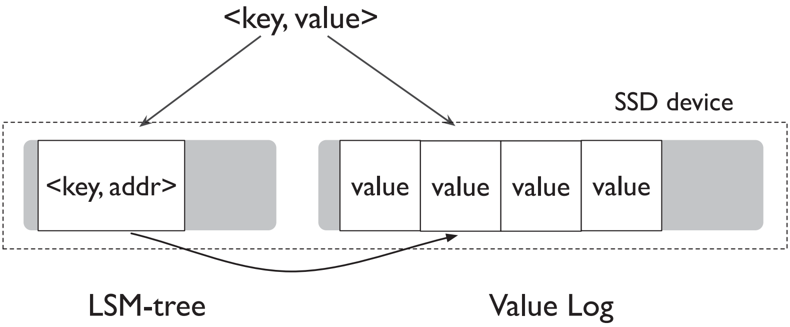


Fig. 6. WiscKey data layout on SSD. This figure shows the data layout of WiscKey on a single SSD device. Keys and values' locations are stored in LSM-tree while values are appended to a separate value log file.

amplification of WiscKey is only (10 × 16 + 1024) / (16 + 1024) = **1.14**. In addition to improving the write performance of applications, the reduced write amplification also improves an SSD's lifetime by requiring fewer erase cycles.

WiscKey's smaller read amplification improves lookup performance. During lookup, WiscKey first searches the LSM-tree for the key and the value's location; once found, another read is issued to retrieve the value. Readers might assume that WiscKey will be slower than LevelDB for lookups, due to its extra I/O to retrieve the value. However, since the LSM-tree of WiscKey is much smaller than LevelDB (for the same database size), a lookup will likely search fewer levels of table files in the LSM-tree; furthermore, a significant portion of the LSM-tree can be easily cached in memory. Hence, each lookup only requires a single random read (for retrieving the value) and thus achieves better lookup performance than LevelDB. For example, assuming 16B keys and 1KB values, if the size of the entire key-value dataset is 100GB, then the size of the LSM-tree is roughly 2GB (assuming a 12B cost for a value's location and size), which readily fits into main memory on modern systems.

WiscKey's architecture is shown in Figure 6. Keys are stored in an LSM-tree while values are stored in a separate value-log file, the *vLog*. The artificial value stored along with the key in the LSM-tree is the address of the actual value in the vLog.

When the user inserts a key-value pair in WiscKey, the value is first appended to the vLog, and the key is then inserted into the LSM-tree along with the value's address (`<vLog-offset, value-size>`). Deleting a key simply deletes it from the LSM tree, without accessing the vLog. All valid values in the vLog have corresponding keys in the LSM-tree; the other values in the vLog are invalid and will be garbage collected later, as we discuss later (Section 3.3.2).

When the user queries for a key, the key is first searched for in the LSM-tree, and if found, the corresponding value's address is retrieved. Then, WiscKey reads the value from the vLog. Note that this process is applied to both point queries and range queries.

3.3. Challenges

Although the idea behind key-value separation is simple, it leads to many challenges and optimization opportunities. For example, the separation of keys and values makes range queries require random I/O. Furthermore, the separation makes both garbage collection and crash consistency challenging. We now explain how we address these challenges.

3.3.1. Parallel Range Query. Range queries are an important feature of modern key-value stores, allowing users to scan a range of key-value pairs. Relational databases [Facebook 2015], local file systems [Jannen et al. 2015; Ren and Gibson

2013; Shetty et al. 2013], and even distributed file systems [Mai and Zhao 2015] use key-value stores as their storage engines, and range queries are a core API requested in these environments.

As previously described, for range queries, LevelDB provides the user with an iterator-based interface including `Seek()`, `Next()`, `Prev()`, `Key()`, and `Value()` operations. To scan a range of key-value pairs, users can first `Seek()` to the starting key, then call `Next()` or `Prev()` to search keys one by one. To retrieve the key or the value of the current iterator position, users call `Key()` or `Value()`, respectively.

In LevelDB, since keys and values are stored together and sorted, a range query can sequentially read key-value pairs from SSTable files. However, since keys and values are stored separately in WiscKey, range queries require random reads, and are hence not as efficient. As we saw previously in Figure 5, random read performance of a single thread on SSD is lower than sequential read performance. However, parallel random reads with a fairly large request size can fully utilize SSD internal parallelism, obtaining performance on par with sequential reads.

Thus, to make range queries efficient, WiscKey leverages the parallel I/O characteristic of SSD devices to prefetch values from the vLog during range queries. The underlying idea is that, with SSDs, *only* keys require special attention for efficient retrieval. As long as keys are retrieved efficiently, range queries can use parallel random reads to efficiently retrieve values.

The prefetching framework can easily fit with the current range query interface. In the current interface, if the user requests a range query, an iterator is returned to the user. For each `Next()` or `Prev()` requested on the iterator, WiscKey tracks the access pattern of the range query. Once a contiguous sequence of key-value pairs is requested, WiscKey starts reading a number of following keys from the LSM-tree sequentially. The corresponding value addresses retrieved from the LSM-tree are inserted into a queue; multiple threads fetch these values from the vLog concurrently.

3.3.2. Garbage Collection. Key-value stores based on standard LSM-trees do not immediately reclaim free space when a key-value pair is deleted or overwritten. Rather, during compaction, if data relating to a deleted or overwritten key-value pair is found, the data is discarded and space is reclaimed. In WiscKey, only invalid keys are reclaimed by LSM-tree compaction. Since WiscKey does not compact values, it needs a special garbage collector to reclaim free space in the vLog.

Because we only store the values in the vLog file (Section 3.2), a naive way to reclaim free space from the vLog is to first scan the LSM-tree to retrieve all the valid value addresses; then, all the values in the vLog without any valid reference from the LSM-tree can be viewed as invalid and reclaimed. However, this method is too heavyweight and is only usable for offline garbage collection.

WiscKey targets a lightweight and online garbage collector. To make this possible, we introduce a small change to WiscKey's basic data layout: while storing values in the vLog, we also store the corresponding key along with the value. The new data layout is shown in Figure 7: the tuple `<key size, value size, key, value>` is stored in the vLog.

WiscKey's garbage collection aims to keep valid values (that do not correspond to deleted keys) in a contiguous range of the vLog, as shown in Figure 7. One end of this range, the *head*, always corresponds to the end of the vLog where new values will be appended. The other end of this range, known as the *tail*, is where garbage collection starts freeing space whenever it is triggered. Only the part of the vLog between the head and the tail contains valid values and will be accessed during lookups.

During garbage collection, WiscKey first reads a chunk of key-value pairs (e.g., several megabytes) from the tail of the vLog, then finds which of those values are valid

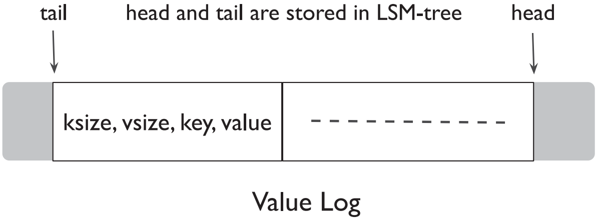


Fig. 7. WiscKey new data layout for garbage collection. This figure shows the new data layout of WiscKey to support an efficient garbage collection. A head and tail pointer are maintained in memory and stored persistently in the LSM-tree. Only the garbage collection thread changes the tail, while all writes to the vLog are appended to the head.

(not yet overwritten or deleted) by querying the LSM-tree. WiscKey then appends valid values back to the head of the vLog. Finally, it frees the space occupied previously by the chunk and updates the tail accordingly.

To avoid losing any data if a crash happens during garbage collection, WiscKey has to make sure that the newly appended valid values and the new tail are persistent on the device before actually freeing space. WiscKey achieves this using the following steps. After appending the valid values to the vLog, the garbage collection calls an `fsync()` on the vLog. Then, it adds these new values' addresses and current tail to the LSM-tree in a synchronous manner; the tail is stored in the LSM-tree as `<tail-marker, tail-vLog-offset>`. Finally, the free space in the vLog is reclaimed.

WiscKey can be configured to initiate and continue garbage collection periodically or until a particular threshold is reached. The garbage collection can also run in offline mode for maintenance. Garbage collection can be triggered rarely for workloads with few deletes and for environments with overprovisioned storage space.

3.3.3. Crash Consistency. On a system crash, LSM-tree implementations usually guarantee atomicity of inserted key-value pairs and in-order recovery of inserted pairs. Because WiscKey's architecture stores values separately from the LSM-tree, obtaining the same crash guarantees might appear complicated. However, WiscKey provides the same crash guarantees by using an interesting property of modern file systems (such as ext4, btrfs, and xfs). Consider a file that contains the sequence of bytes $\langle b_1 b_2 b_3 ... b_n \rangle$, and the user appends the sequence $\langle b_{n+1} b_{n+2} b_{n+3} ... b_{n+m} \rangle$ to it. If a crash happens, after file system recovery in modern file systems, the file will be observed to contain the sequence of bytes $\langle b_1 b_2 b_3 ... b_n b_{n+1} b_{n+2} b_{n+3} ... b_{n+x} \rangle \; \exists \; x < m$; that is, only some prefix of the appended bytes will be added to the end of the file during file system recovery [Pillai et al. 2014]. It is not possible for random bytes or a nonprefix subset of the appended bytes to be added to the file. Because values are appended sequentially to the end of the vLog file in WiscKey, the aforementioned property conveniently translates as follows: if a value X in the vLog is lost in a crash, all values inserted after X are lost too.

When the user queries a key-value pair, if WiscKey cannot find the key in the LSM-tree because the key had been lost during a system crash, WiscKey behaves exactly like traditional LSM-trees: even if the value had been written in the vLog before the crash, it will be garbage collected later. If the key could be found in the LSM tree, however, an additional step is required to maintain consistency. In this case, WiscKey verifies first whether the value address retrieved from the LSM-tree falls within the current valid range of the vLog, and then whether the value found corresponds to the queried key. If the verifications fail, WiscKey assumes that the value was lost during a system crash, deletes the key from the LSM-tree, and informs the user that the key was not found. Because each value added to the vLog has a header including the corresponding

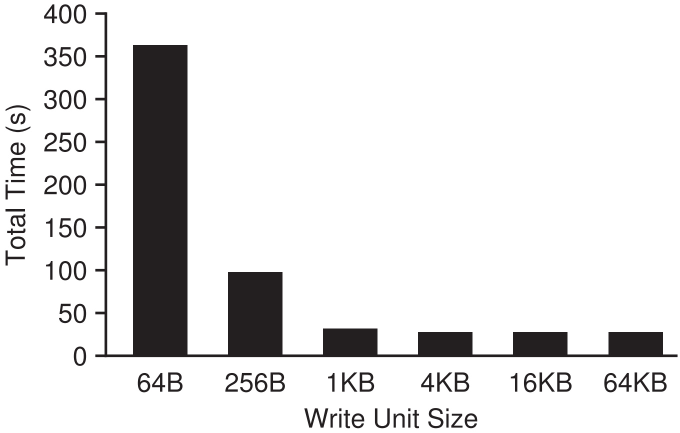


Fig. 8. Impact of write unit size. This figure shows the total time to write a 10GB file to an ext4 file system on an SSD device, followed by an `fsync()` at the end. We vary the size of each `write()` system call.

key, verifying whether the key and the value match is straightforward; if necessary, a magic number or checksum can be easily added to the header.

LSM-tree implementations also guarantee the user durability of key-value pairs after a system crash if the user specifically requests synchronous inserts. WiscKey implements synchronous inserts by flushing the vLog before performing a synchronous insert into its LSM-tree.

3.4. Optimizations

Separating keys from values in WiscKey provides an opportunity to rethink how the value log is updated and the necessity of the LSM-tree log. We now describe how these opportunities can lead to improved performance.

3.4.1. Value-Log Write Buffer. For each `Put()`, WiscKey needs to append the value to the vLog by using a `write()` system call. However, for an insert-intensive workload, issuing a large number of small writes to a file system can introduce a noticeable overhead, especially on a fast storage device [Caulfield et al. 2010; Peter et al. 2014]. Figure 8 shows the total time to sequentially write a 10GB file in ext4 (Linux 3.14). For small writes, the overhead of each system call aggregates significantly, leading to a long runtime. With large writes (larger than 4KB), device throughput is fully utilized.

To reduce overhead, WiscKey buffers values in a user-space buffer and flushes the buffer only when the buffer size exceeds a threshold or when the user requests a synchronous insertion. Thus, WiscKey only issues large writes and reduces the number of `write()` system calls. For a lookup, WiscKey first searches the vLog buffer, and if not found there, actually reads from the vLog. Obviously, this mechanism might result in some data (that is buffered) being lost during a crash; the crash consistency guarantee obtained is similar to LevelDB.

3.4.2. Optimizing the LSM-Tree Log. As shown in Figure 1, a log file is usually used in LSM-trees. The LSM-tree tracks inserted key-value pairs in the log file so that, if the user requests synchronous inserts and there is a crash, the log can be scanned after reboot and the inserted key-value pairs recovered.

In WiscKey, the LSM-tree is only used for keys and value addresses. Moreover, the vLog also records inserted keys to support garbage collection as described previously. Hence, writes to the LSM-tree log file can be avoided without affecting correctness.

If a crash happens before the keys are persistent in the LSM-tree, they can be recovered by scanning the vLog. However, a naive algorithm would require scanning the entire vLog for recovery. To reduce the amount of scanning required, WiscKey records the head of the vLog periodically in the LSM-tree, as a key-value pair `<head-marker, head-vLog-offset>`. When a database is opened, WiscKey starts the vLog scan from the most recent head position stored in the LSM-tree and continues scanning until the end of the vLog. Because the head is stored in the LSM-tree and the LSM-tree inherently guarantees that keys inserted into the LSM-tree will be recovered in the inserted order, this optimization is crash consistent. Therefore, removing the LSM-tree log of WiscKey is a safe optimization and improves performance, especially when there are many small insertions.

3.5. Implementation

WiscKey is based on LevelDB 1.18. WiscKey creates a vLog when creating a new database and manages the keys and value addresses in the LSM-tree. The vLog is internally accessed by multiple components with different access patterns. For example, a lookup is served by randomly reading the vLog, while the garbage collector sequentially reads from the tail and appends to the head of the vLog file. We use `posix_fadvise()` to predeclare access patterns for the vLog under different situations.

For range queries, WiscKey maintains a background thread pool with 32 threads. These threads sleep on a thread-safe queue, waiting for new value addresses to arrive. When prefetching is triggered, WiscKey inserts a fixed number of value addresses to the worker queue and then wakes up all the sleeping threads. These threads will start reading values in parallel, caching them in the buffer cache automatically.

To efficiently garbage collect the free space of the vLog, we use the hole-punching functionality of modern file systems (`fallocate()`). Punching a hole in a file can free the physical space allocated and allows WiscKey to elastically use the storage space. The maximal file size on modern file systems is big enough for WiscKey to run a long time without wrapping back to the beginning of the file; for example, the maximal file size is 64TB on ext4, 8EB on xfs, and 16EB on btrfs. The vLog can be trivially adapted into a circular log if necessary.

4. EVALUATION

In this section, we present evaluation results that demonstrate the benefits of the design choices of WiscKey. Specifically, we seek to answer the following fundamental performance questions about WiscKey:

(1) Does key-value separation result in lower write and read amplification, and how does it impact performance (throughput, tail latency) and device endurance?
(2) Does the parallel range query in WiscKey work efficiently with modern SSDs?
(3) What is the effect of garbage collection on WiscKey's performance?
(4) Does WiscKey maintain crash consistency, and how long does it take to recover after a crash?
(5) What are the CPU overheads of WiscKey?
(6) How does WiscKey perform on real workloads, compared to its peers?

4.1. Experimental Setup

All experiments are run on a testing machine with two Intel Xeon CPU E5-2667 v2 @ 3.30GHz processors and 64GB of memory. The operating system is 64-bit Linux 3.14, and the file system used is ext4. The storage device used is a 500GB Samsung 840 EVO SSD, which has 500MB/s sequential-read and 400MB/s sequential-write maximal performance; random read performance is shown in Figure 5.

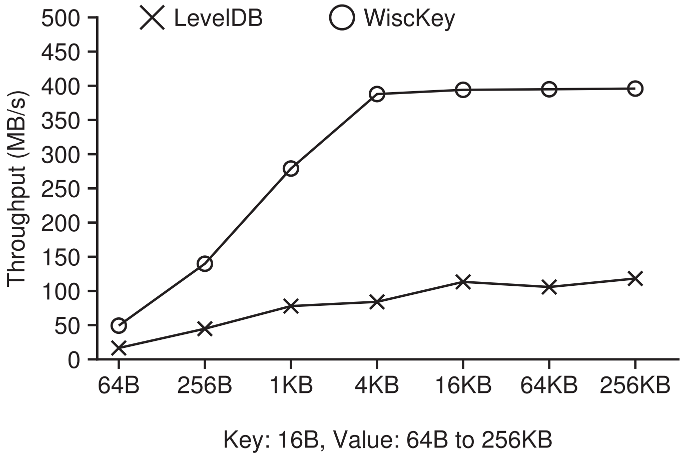


Fig. 9. Sequential-load performance. This figure shows the sequential-load throughput of LevelDB and WiscKey for different value sizes for a 100GB dataset. Key size is 16B.

4.2. Microbenchmarks

We use db_bench (the default microbenchmarks in LevelDB) to evaluate LevelDB and WiscKey. We always use a key size of 16B but perform experiments for different value sizes. We disable data compression for easier understanding and analysis of performance.

4.2.1. Load Performance. We now describe the results for the sequential-load and random-load microbenchmarks. The former benchmark constructs a 100GB database by inserting keys in a sequential order, while the latter inserts keys in a uniformly distributed random order. Note that the sequential-load benchmark does not cause compaction in either LevelDB or WiscKey, while the random load does.

Figure 9 shows the sequential-load throughput of LevelDB and WiscKey for a wide range of value sizes: the throughput of both stores increases with the value size. However, even for the largest value size considered (256KB), LevelDB's throughput is far below peak device bandwidth. To analyze this result further, Figure 10 presents a breakdown of how time is spent in different components during each run of the benchmark. As seen in the figure, time is spent in three major parts: writing to the log file, inserting to the memtable, and waiting for the memtable to be flushed to the device. For small key-value pairs, writing to the log file accounts for the most significant percentage of the total time, due to the inefficiency of small writes (seen previously in Figure 8). For larger pairs, log writing and the memtable sorting are more efficient, while waiting for memtable flushes becomes the bottleneck. Unlike LevelDB, WiscKey reaches the full device bandwidth for value sizes more than 4KB. Because WiscKey does not write to the LSM-tree log and buffers append to the vLog, it is 3× faster even for small values.

Figure 11 shows the random-load throughput of LevelDB and WiscKey for different value sizes. LevelDB's throughput ranges from only 2MB/s (64B value size) to 4.1MB/s (256KB value size), while WiscKey's throughput increases with the value size, reaching the peak device write throughput after the value size is larger than 4KB. WiscKey's throughput is 46× and 111× of LevelDB for the 1KB and 4KB value sizes, respectively. LevelDB has low throughput because compaction both consumes a large percentage of the device bandwidth and also slows down foreground writes (to avoid overloading the L_0 of the LSM-tree, as described earlier (Section 2.2). In WiscKey, compaction only

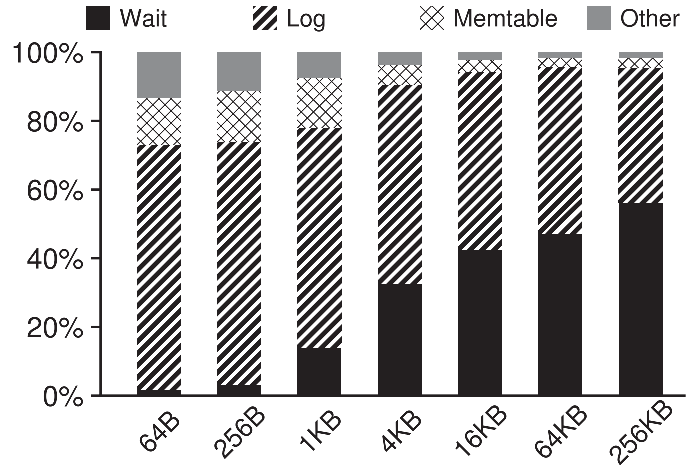


Fig. 10. Sequential-load time breakdown of LevelDB. This figure shows the percentage of time incurred in different components during sequential load in LevelDB. Time is broken down into time spent waiting for the memtable to be flushed (Wait), writing to the log file (Log), inserting data into the memtable (Memtable), and other time (Other).

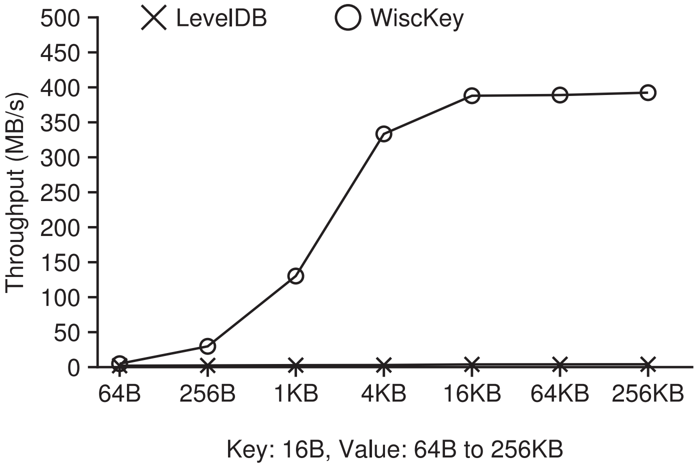


Fig. 11. Random-load performance. This figure shows the random-load throughput of LevelDB and WiscKey for different value sizes for a 100GB dataset. Key size is 16B.

introduces a small overhead, leading to the full device bandwidth being effectively utilized.

To analyze this result further, Figure 12 presents the write amplification of LevelDB and WiscKey. The write amplification of LevelDB is always more than 12, while that of WiscKey decreases quickly to nearly one when the value size reaches 1KB, because the LSM-tree of WiscKey is significantly smaller.

Finally, we examine the latency of writes for one particular value size under random writes. Figure 13 shows a latency CDF for 4KB random writes for both LevelDB and WiscKey. The distribution shows that WiscKey generally has (much) lower latencies, and is far more predictable in its performance, than LevelDB (note: the x-scale is logarithmic). This result is due to the reduced effect that compaction has on the foreground

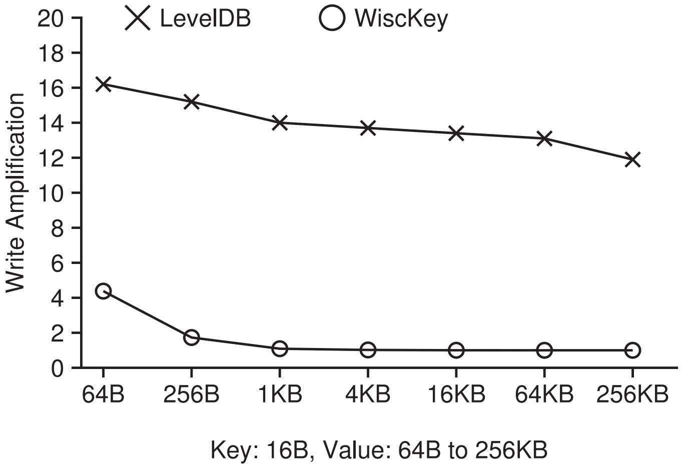


Fig. 12. Write amplification of random load. This figure shows the write amplification of LevelDB and WiscKey for randomly loading a 100GB database.

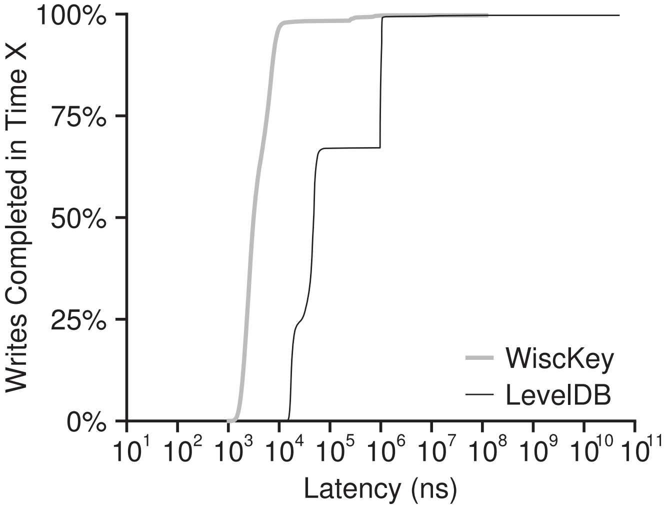


Fig. 13. Random write latencies (CDF). This figure shows the cumulative distribution function for random writes with 4KB value sizes.

workload; with a smaller tree, WiscKey performs fewer compactions and thus delivers higher performance.

To demonstrate the differences between the LSM-trees of WiscKey and LevelDB, we also measured the sizes of the SStable files that comprise each system's LSM-tree; the results are shown in Figure 14. After the random-write workload has completed, WiscKey's tree is almost entirely compressed into just two levels (2 and 3), whereas the LSM-tree of LevelDB spans six levels (0 through 5). Furthermore, the total size difference between the two trees is enormous; the approximate total file size for WiscKey is 358MB, whereas for LevelDB it is roughly 100GB.

4.2.2. Query Performance. We now compare the random lookup (point query) and range query performance of LevelDB and WiscKey. Figure 15 presents the random lookup results of 100,000 operations on a 100GB random-loaded database. Even though a

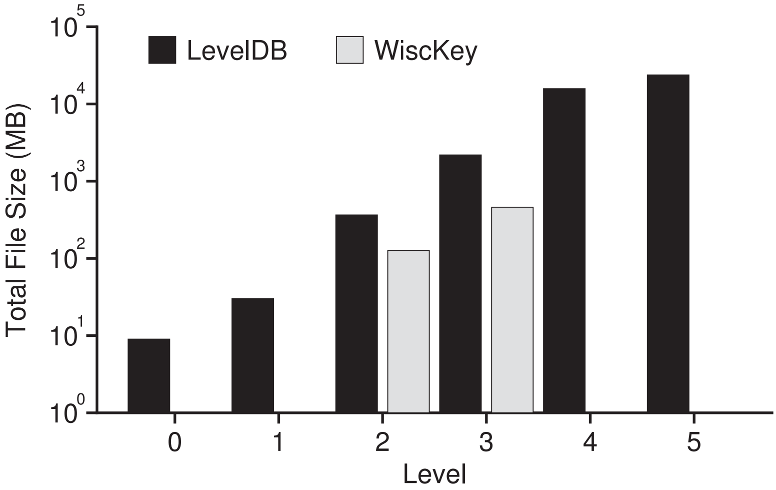


Fig. 14. Table size breakdown. The figure shows the amount of data in each of the levels of LevelDB and WiscKey.

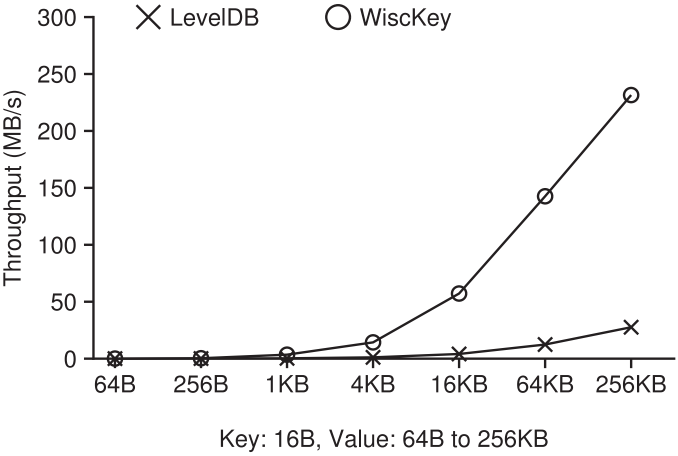


Fig. 15. Random lookup performance. This figure shows the random lookup performance for 100,000 operations on a 100GB database that is randomly loaded.

random lookup in WiscKey needs to check both the LSM-tree and the vLog, the throughput of WiscKey is still much better than LevelDB: for 1KB value size, WiscKey's throughput is 12× that of LevelDB. For large value sizes, the throughput of WiscKey is only limited by the random read throughput of the device, as shown in Figure 5. LevelDB has low throughput due to high read amplification (Section 2.3). WiscKey performs significantly better because the read amplification is lower due to a smaller LSM-tree. Another reason for WiscKey's performance is that the compaction process in WiscKey is less I/O intensive, avoiding many background reads and writes.

We delve further into these results by presenting the latency breakdown of 4KB reads. Figure 16 presents the cumulative distribution of lookup latencies for both WiscKey and LevelDB. As you can see from the figure, WiscKey achieves notably lower latencies for roughly half of its requests; caching of the LSM-tree and repeated access to cached values explains this performance advantage. However, some reads are indeed

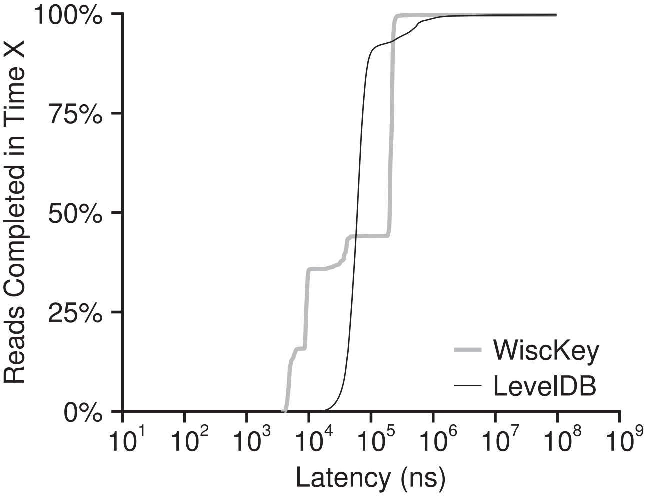


Fig. 16. Random read latencies (CDF). This figure shows the cumulative distribution function for random reads with 4KB value sizes.

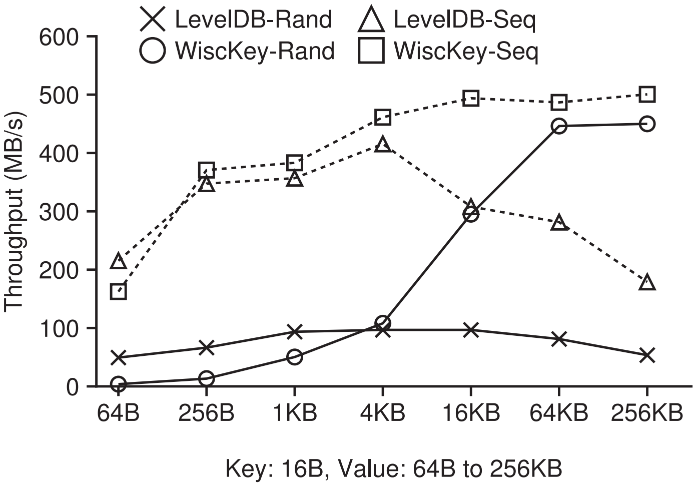


Fig. 17. Range query performance. This figure shows range query performance; 4GB of data is queried from a 100GB database that is randomly (Rand) and sequentially (Seq) loaded.

slower in WiscKey than in LevelDB, due to the cost of (randomly) accessing values from the vLog.

Figure 17 shows the range query (scan) performance of LevelDB and WiscKey. For a randomly loaded database, LevelDB reads multiple files from different levels, while WiscKey requires random accesses to the vLog (but WiscKey leverages parallel random reads). As can be seen from Figure 17, the throughput of LevelDB initially increases with the value size for both databases. However, beyond a value size of 4KB, since an SSTable file can store only a small number of key-value pairs, the overhead is dominated by opening many SSTable files and reading the index blocks and Bloom filters in each file. For larger key-value pairs, WiscKey can deliver the device's sequential bandwidth, up to 8.4× of LevelDB. However, WiscKey performs 12× worse than LevelDB for 64B key-value pairs due to the device's limited parallel random-read throughput for

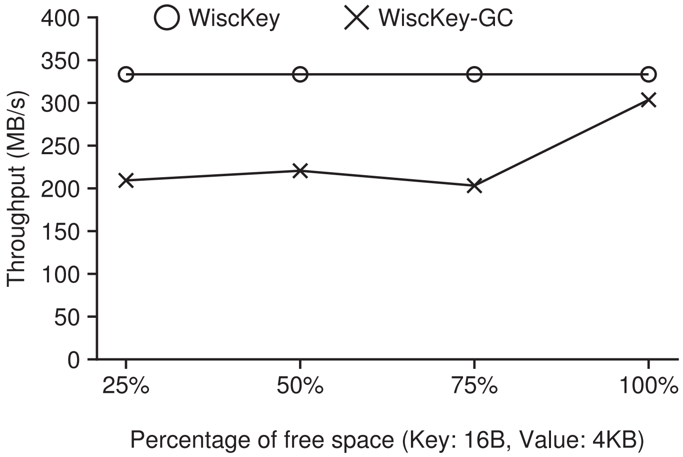


Fig. 18. Garbage collection. This figure shows the performance of WiscKey under garbage collection for various free-space ratios.

small request sizes; WiscKey's relative performance is better on high-end SSDs with higher parallel random-read throughput [Fusion-IO 2015]. Furthermore, this workload represents a worst case where the database is randomly filled and the data is unsorted in the vLog.

Figure 17 also shows the performance of range queries when the data is sorted, which corresponds to a sequentially loaded database; in this case, both LevelDB and WiscKey can sequentially scan through data. Performance for sequentially loaded databases follows the same trend as randomly loaded databases; for 64B pairs, WiscKey is 25% slower because WiscKey reads both the keys and the values from the vLog (thus wasting bandwidth), but WiscKey is 2.8× faster for large key-value pairs. Thus, with small key-value pairs, log reorganization (sorting) for a random-loaded database can make WiscKey's range-query performance comparable to LevelDB's performance.

4.2.3. Garbage Collection. We now investigate WiscKey's performance while garbage collection is performed in the background. The performance can potentially vary depending on the percentage of free space found during garbage collection, since this affects the amount of data written and the amount of space freed by the garbage collection thread. We use random-load (the workload that is most affected by garbage collection) as the foreground workload and study its performance for various percentages of free space. Our experiment specifically involves three steps: we first create a database using random-load, then delete the required percentage of key-value pairs, and finally, we run the random-load workload and measure its throughput while garbage collection happens in the background. We use a key-value size of 4KB and vary the percentage of free space from 25% to 100%.

Figure 18 shows the results: if 100% of data read by the garbage collector is invalid, the throughput is only 10% lower. Throughput is only marginally lower because garbage collection reads from the tail of the vLog and writes only valid key-value pairs to the head; if the data read is entirely invalid, no key-value pair needs to be written. For other percentages of free space, throughput drops about 35% since the garbage collection thread performs additional writes. Note that, in all cases, while garbage collection is happening, WiscKey is at least 70× faster than LevelDB.

4.2.4. Crash Consistency. Separating keys from values necessitates additional mechanisms to maintain crash consistency. We verify the crash consistency mechanisms of

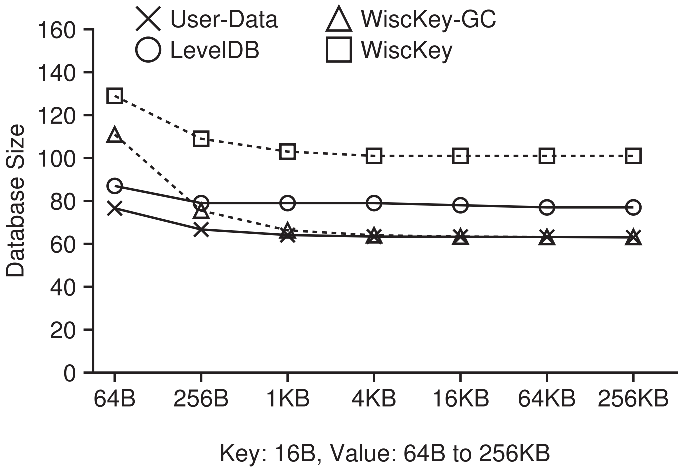


Fig. 19. Space amplification. This figure shows the actual database size of LevelDB and WiscKey for a random-load workload of a 100GB dataset. User-Data represents the logical database size.

WiscKey by using the ALICE tool [Alagappan et al. 2016; Pillai et al. 2014]; the tool chooses and simulates a comprehensive set of system crashes that have a high probability of exposing inconsistency. We use a test case that invokes a few asynchronous and synchronous `Put()` calls. When configured to run tests for ext4, xfs, and btrfs, ALICE checks more than 3,000 selectively chosen system crashes and does not report any consistency vulnerability introduced by WiscKey.

The new consistency mechanism also affects WiscKey's recovery time after a crash, and we design an experiment to measure the worst-case recovery time of WiscKey and LevelDB. LevelDB's recovery time is proportional to the size of its log file after the crash; the log file exists at its maximum size just before the memtable is written to disk. WiscKey, during recovery, first retrieves the head pointer from the LSM-tree and then scans the vLog file from the head pointer till the end of the file. Since the updated head pointer persists on disk when the memtable is written, WiscKey's worst-case recovery time also corresponds to a crash happening just before then. We measured the worst-case recovery time induced by the situation described so far; for 1KB values, LevelDB takes 0.7 seconds to recover the database after the crash, while WiscKey takes 2.6 seconds. Note that WiscKey can be configured to persist the head pointer more frequently if necessary.

4.2.5. Space Amplification. When evaluating a key-value store, most previous work focused only on read and write amplification. However, space amplification is important for flash devices because of their expensive price-per-gigabyte compared with hard drives. Space amplification is the ratio of the actual size of the database on disk to the logical size of the database [Balasundaram et al. 2015]. For example, if a 1KB key-value pair takes 4KB of space on disk, then the space amplification is 4. Compression decreases space amplification, while extra data (garbage, fragmentation, or metadata) increases space amplification. Compression is disabled to make the discussion simple.

For a sequential-load workload, the space amplification can be near one, given that the extra metadata in LSM-trees is minimal. For a random-load or overwrite workload, space amplification is usually more than one when invalid pairs are not garbage collected fast enough.

Figure 19 shows the database size of LevelDB and WiscKey after randomly loading a 100GB dataset (the same workload as Figure 11). The space overhead of LevelDB

Workloads	**Sequential Load**	**Random Load**	**Random Lookup**	**Range Query**
LevelDB	10.6%	6.3%	7.9%	11.2%
WiscKey	8.2%	8.9%	11.3%	30.1%

Fig. 20. CPU usage of LevelDB and WiscKey. This table compares the CPU usage of four workloads on LevelDB and WiscKey. Key size is 16B and value size is 1KB. Sequential load and random load sequentially and randomly load a 100GB database, respectively. Given a 100GB random-filled database, random lookup issues 100K random lookups, while range query sequentially scans 4GB of data.

arises due to invalid key-value pairs that are not garbage collected when the workload is finished. The space overhead of WiscKey includes the invalid key-value pairs and the extra metadata (pointers in the LSM-tree and the tuple in the vLog as shown in Figure 7). After garbage collection, the database size of WiscKey is close to the logical database size when the extra metadata is small compared to the value size.

No key-value store can minimize read amplification, write amplification, and space amplification at the same time. Tradeoffs among these three factors are balanced differently in various systems. In LevelDB, the sorting and garbage collection are coupled together. LevelDB trades higher write amplification for lower space amplification; however, the workload performance can be significantly affected. WiscKey consumes more space to minimize I/O amplification when the workload is running; because sorting and garbage collection are decoupled in WiscKey, garbage collection can be performed later, thus minimizing its impact on foreground performance.

4.2.6. CPU Usage. We now investigate the CPU usage of LevelDB and WiscKey for various workloads shown in previous sections. The CPU usage shown here includes both the application and operating system usage.

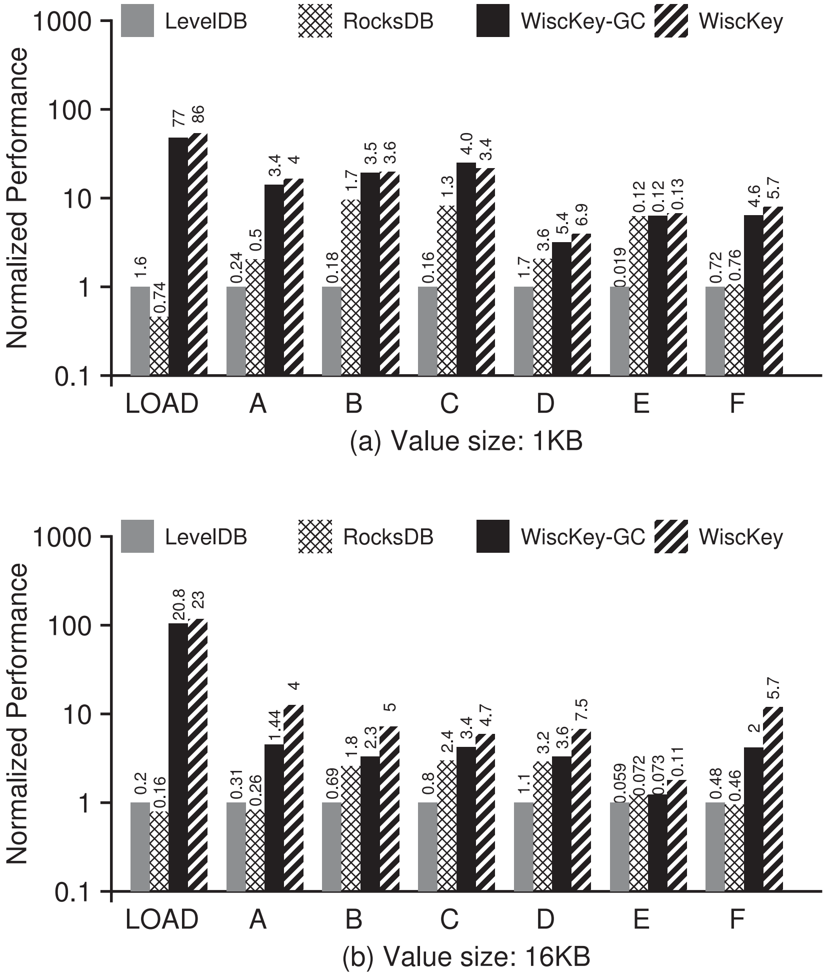
As shown in Figure 20, LevelDB has higher CPU usage for the sequential-load workload. As we explained in Figure 10, LevelDB spends a large amount of time writing key-value pairs to the log file. Writing to the log file involves encoding each key-value pair, which has high CPU cost. Since WiscKey removes the log file as an optimization, WiscKey has lower CPU usage than LevelDB. For the range query workload, WiscKey uses 32 background threads to do the prefetch; therefore, the CPU usage of WiscKey is much higher than LevelDB.

Overall, we find that CPU is not a bottleneck for both LevelDB and WiscKey in our setup. However, our workloads stress single-thread I/O performance and not multi-threaded lock contention. For more detail on how to scale LevelDB, see related work on RocksDB [Dong 2015].

4.3. YCSB Benchmarks

The YCSB benchmark [Cooper et al. 2010] provides a framework and a standard set of six workloads for evaluating the performance of key-value stores. We use YCSB to compare LevelDB, RocksDB [Dong 2015], and WiscKeyon a 100GB database. In addition to measuring the usual-case performance of WiscKey, we also run WiscKey with garbage collection always happening in the background so as to measure its worst-case performance. RocksDB is an SSD-optimized version of LevelDB with many performance-oriented features, including multiple memtables and background threads for compaction. We use RocksDB with the default configuration parameters. We evaluated the key-value stores with two different value sizes, 1KB and 16KB (data compression is disabled).

WiscKey performs significantly better than LevelDB and RocksDB, as shown in Figure 21. For example, during load, for 1KB values, WiscKey performs at least 50× faster than the other databases in the usual case, and at least 45× faster in the worst



(a) Value size: 1KB

(b) Value size: 16KB

Fig. 21. YCSB macrobenchmark performance. This figure shows the performance of LevelDB, RocksDB, and WiscKey for various YCSB workloads. The x-axis corresponds to different workloads, and the y-axis shows the performance normalized to LevelDB's performance. The number on top of each bar shows the actual throughput achieved (K ops/s). (a) shows performance under 1KB values and (b) shows performance under 16KB values. The load workload corresponds to constructing a 100GB database and is similar to the random-load microbenchmark. Workload A has 50% reads and 50% updates, Workload B has 95% reads and 5% updates, and Workload C has 100% reads; keys are chosen from a Zipf, and the updates operate on already-existing keys. Workload D involves 95% reads and 5% inserting new keys (temporally weighted distribution). Workload E involves 95% range queries and 5% inserting new keys (Zipf), while Workload F has 50% reads and 50% read-modify-writes (Zipf).

case (with garbage collection); with 16KB values, WiscKey performs 104× better, even under the worst case.

For reads, the Zipf distribution used in most workloads allows popular items to be cached and retrieved without incurring device access, thus reducing WiscKey's advantage over LevelDB and RocksDB. Hence, WiscKey's relative performance (compared to and RocksDB) is better in Workload A (50% reads) than in Workload B (95% reads) and Workload C (100% reads). However, RocksDB and LevelDB still do not match WiscKey's performance in any of these workloads. The worst-case performance of WiscKey (with garbage collection switched on always, even for read-only workloads) is better than LevelDB and RocksDB. However, the impact of garbage collection on performance is markedly different for 1KB and 16KB values. Garbage collection repeatedly selects and cleans a 4MB chunk of the vLog; with small values, the chunk will include many key-value pairs, and thus garbage collection spends more time accessing the LSM-tree to verify the validity of each pair. For large values, garbage collection spends less time on the verification, and hence aggressively writes out the cleaned chunk, affecting foreground throughput more. Note that, if necessary, garbage collection can be throttled to reduce its foreground impact.

Unlike the microbenchmark considered previously, Workload E has multiple small range queries, with each query retrieving between 1 and 100 key-value pairs. Since the workload involves multiple range queries, accessing the first key in each range resolves to a random lookup—a situation favorable for WiscKey. Hence, WiscKey performs better than RocksDB and LevelDB even for these relatively small 1KB values.

5. RELATED WORK

Various key-value stores based on hash tables have been proposed for SSD devices. FAWN [Andersen et al. 2009] keeps key-value pairs in an append-only log on the SSD and uses an in-memory hash table index for fast lookups. FlashStore [Debnath et al. 2010] and SkimpyStash [Debnath et al. 2011] follow the same design but optimize the in-memory hash table; FlashStore uses cuckoo hashing and compact key signatures, while SkimpyStash moves a part of the table to the SSD using linear chaining. BufferHash [Anand et al. 2010] uses multiple in-memory hash tables, with Bloom filters to choose which hash table to use for a lookup. SILT [Lim et al. 2011] is highly optimized for memory and uses a combination of log structure, hash table, and sorted table layouts. WiscKey shares the log-structure data layout with these key-value stores. However, these stores use hash tables for indexing, and thus do not support modern features that have been built atop LSM-tree stores, such as range queries or snapshots. WiscKey instead targets a feature-rich key-value store that can be used in various situations.

Much work has gone into optimizing the original LSM-tree key-value store [ONeil et al. 1996]. bLSM [Sears and Ramakrishnan 2012] presents a new merge scheduler to bound write latency, thus maintaining a steady write throughput, and also uses Bloom filters to improve performance. VT-tree [Shetty et al. 2013] avoids sorting any previously sorted key-value pairs during compaction by using a layer of indirection. WiscKey instead directly separates values from keys, significantly reducing write amplification regardless of the key distribution in the workload. LOCS [Wang et al. 2014] exposes internal flash channels to the LSM-tree key-value store, which can exploit the abundant parallelism for a more efficient compaction. Atlas [Lai et al. 2015] is a distributed key-value store based on ARM processors and erasure coding, and stores keys and values on different hard drives. WiscKey is a stand-alone key-value store, where the separation between keys and values is highly optimized for SSD devices to achieve significant performance gains. LSM-trie [Wu et al. 2015] uses a trie structure to organize keys and proposes a more efficient compaction based on the trie; however, this design sacrifices LSM-tree features such as efficient support for range queries. RocksDB, described previously, still exhibits high write amplification due to its design being fundamentally similar to LevelDB; RocksDB's optimizations are orthogonal to WiscKey's design.

Walnut [Chen et al. 2012] is a hybrid object store that stores small objects in an LSM-tree and writes large objects directly to the file system. IndexFS [Ren et al. 2014] stores its metadata in an LSM-tree with the column-style schema to speed up the throughput of insertion. Purity [Colgrove et al. 2015] also separates its index from data tuples by only sorting the index and storing tuples in time order. All three systems use similar techniques as WiscKey. However, we solve this problem in a more generic and complete manner, and optimize both load and lookup performance for SSD devices across a wide range of workloads.

Key-value stores based on other data structures have also been proposed. TokuDB [Bender et al. 2007; Buchsbaum et al. 2000] is based on fractal-tree indexes, which buffer updates in internal nodes; the keys are not sorted, and a large index has to be maintained in memory for good performance. ForestDB [Ahn et al. 2016] uses a HB+-trie to efficiently index long keys, improving the performance and reducing the space overhead of internal nodes. NVMKV [Marmol et al. 2015] is an FTL-aware key-value store that uses native FTL capabilities, such as sparse addressing, and transactional supports. Vector interfaces that group multiple requests into a single operation are also proposed for key-value stores [Vasudevan et al. 2012]. Since these key-value stores are based on different data structures, they each have different tradeoffs relating to performance; instead, WiscKey proposes improving the widely used LSM-tree structure.

Many proposed techniques seek to overcome the scalability bottlenecks of in-memory key-value stores, such as Masstree [Mao et al. 2012], MemC3 [Fan et al. 2013], Memcache [Nishtala et al. 2013], MICA [Lim et al. 2014], and cLSM [Golan-Gueta et al. 2015]. These techniques may be adapted for WiscKey to further improve its performance.

6. CONCLUSIONS

Key-value stores have become a fundamental building block in data-intensive applications. In this article, we propose WiscKey, a novel LSM-tree-based key-value store that separates keys and values to minimize write and read amplification. The data layout and I/O patterns of WiscKey are highly optimized for SSD devices. Our results show that WiscKey can significantly improve performance for most workloads.

We believe there are many avenues for future work. For example, in WiscKey, garbage collection is done by a single background thread. The thread reads a chunk of key-value pairs from the tail of the vLog file; then, for each key-value pair, it checks the LSM-tree for validity; finally, the valid key-value pairs are written back to the head of the vLog file. We can improve garbage collection in two ways. First, lookups in the LSM-tree are slow since multiple random reads may be required. To speed up this process, we can use multiple threads to perform the lookup concurrently for different key-value pairs. Second, we can make garbage collection more effective by maintaining a bitmap of invalid key-value pairs in the vLog file. When the garbage collection is triggered, it will first reclaim the chunk with the highest percentage of free space.

Another interesting direction to scale LevelDB or WiscKey is sharding the database. Many components of LevelDB are single threaded due to a single shared database. As we discussed before, there is a single memtable to buffer writes in memory. When the memtable is full, the foreground writes will be stalled until the compaction thread flushes the memtable to disk. In LevelDB, only a single writer can be allowed to update the database. The database is protected by a global mutex. The background compaction thread also needs to grab this mutex when sorting the key-value pairs, competing with the foreground writes. For a multiple-writer workload, this architecture can unnecessarily block concurrent writes. One solution is to partition the database and related memory structures into multiple smaller shards. Each shard's keys will not overlap with others. Under this design, writes to different key-value ranges can be done concurrently to different shards. A random lookup can also be distributed to one target shard, without searching all shards. This new design may make lookups faster because of a smaller dataset to search. Our initial work on a prototype implementation of sharded WiscKey has yielded substantial benefits, and thus is another clear direction for future work.

Overall, key-value stores are an increasingly important component in modern scalable systems. New techniques to make them deliver higher and more consistent performance will thus likely continue to be a focus in the coming years. As new, faster media supplant SSDs, it is likely that even further software innovation will be required.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Ethan Miller (our shepherd) for their feedback. We thank the members of the ADSL research group, the RocksDB team (FaceBook), Yinan Li (Microsoft Research), and Bin Fan (Tachyon Nexus) for their suggestions and comments on this work at various stages.

REFERENCES

Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Trans. Comput.* 65, 3 (March 2016), 902–915.

Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI'16)*.

Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. 2010. Cheap and large cams for high-performance data-intensive networked systems. In *Proceedings of the 7th Symposium on Networked Systems Design and Implementation (NSDI'10)*.

David Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*.

Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*.

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2014. *Operating Systems: Three Easy Pieces* (0.9 ed.). Arpaci-Dusseau Books.

Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, and Mark Callaghan. 2015. Designing Access Methods: The RUM Conjecture. In *The 19th International Conference on Extending Database Technology (EDBT'15)*.

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2015. Workload analysis of a large-scale key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*.

Radheshyam Balasundaram, Igor Canadi, Yueh-Hsuan Chiang, Siying Dong, Leonidas Galanis, Lei Jin, Xing Jin, Venkatesh Radhakrishnan, Dmitri Smirnov, Yi Wu, and Feng Zhu. 2015. RocksDB Blog. Retrieved from http://rocksdb.org/blog/.

Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*.

Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan Fogel, Bradley Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming b-trees. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'07)*.

Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. 2000. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*.

Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. 205–218.

Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA'11)*.

Jianjun Chen, Chris Douglas, Michi Mutsuzaki, Patrick Quaid, Raghu Ramakrishnan, Sriram Rao, and Russell Sears. 2012. Walnut: A unified cloud object store. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*.

John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*.

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!s hosted data serving platform. In *Proceedings of the VLDB Endowment (PVLDB'08)*.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'10)*.

Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High throughput persistent key-value store. In *Proceedings of the 36th International Conference on Very Large Databases (VLDB'10)*.

Biplob Debnath, Sudipta Sengupta, and Jin Li. 2011. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*.

Guiseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*.

Siying Dong. 2015. RocksDB: Challenges of LSM-Trees in Practice. http://www.hpts.ws/papers/2015/rocksdb_hpts_website.pdf.

Facebook. 2015. RocksDB 2015 H2 Roadmap. http://rocksdb.org/blog/2015/rocksdb-2015-h2-roadmap/.

Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI'13)*.

Fusion-I. O. 2015. Fusion-I. O. ioDrive2. Retrieved from http://www.fusionio.com/products/iodrive2.

Lars George. 2011. *HBase: The Definitive Guide*. O'Reilly.

Sanjay Ghemawat and jeff Dean. 2011. LevelDB. Retrieved from http://code.google.com/p/leveldb.

Guy Golan-Gueta, Edward Bortnikov, Eshcar Hillel, and Idit Keidar. 2015. Scaling concurrent log-structured data stores. In *Proceedings of the EuroSys Conference (EuroSys'15)*.

Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Analysis of HDFS under HBase: A Facebook messages case study. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST'14)*.

William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*.

Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. 2012. Revisiting storage for smartphones. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*.

Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidus key-value storage system for cloud data. In *Proceedings of the 31st IEEE Conference on Massive Data Storage (MSST'15)*.

Avinash Lakshman and Prashant Malik. 2009. Cassandra – a decentralized structured storage system. In *The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware*.

Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*.

Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI'14)*.

Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*.

Haohui Mai and Jing Zhao. 2015. Scaling HDFS to manage billions of files with key value stores. In *The 8th Annual Hadoop Summit*.

Yandong Mao, Eddie Kohler, and Robert Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the EuroSys Conference (EuroSys'12)*.

Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A scalable, lightweight, ftl-aware key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*.

Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*.

Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI'13)*.

Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. 1994. AlphaSort: A RISC machine sort. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD'94)*.

Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996), 351–385.

Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, and Thomas Anderson. 2014. Arrakis: The operating system is the control plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*.

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*.

Eric Redmond. 2013. A Little Riak Book. Retrieved from http://www.littleriakbook.com.

Kai Ren and Garth Gibson. 2013. TABLEFS: Enhancing metadata efciency in the local file system. In *Proceedings of the USENIX Annual Technical Conference (USENIX'13)*.

Kai Ren, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*.

Russell Sears and Raghu Ramakrishnan. 2012. bLSM: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD'12)*.

Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST'13)*.

Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. 2012. Serving large-scale batch computed data with project Voldemort. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*.

Vijay Vasudevan, Michael Kaminsky, and David G. Andersen. 2012. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'12)*.

Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the EuroSys Conference (EuroSys'14)*.

Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based ultra-large key-value store for small data. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*.

Received December 2016; accepted December 2016