WiscKey: Separating Keys from Values in SSD-Conscious Storage

wu

April 25, 2024

1 Abstract

WiscKey is a persistent LSM-tree-based key-value store with a performance-oriented data layout that **separates keys from values to minimize I/O amplification**.

2 Introduction

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:

1. 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
2. Second, SSDs have a large degree of internal parallelism; an LSM built atop an SSD must be carefully designed to harness said parallelism
3. Third, SSDs can wear out through repeated writes; the high write amplification in LSM-trees can significantly reduce device lifetime.

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.

The central idea behind WiscKey is the separation of keys and values; only keys are kept sorted in the LSM-tree, while values are stored separately in a log.

1. reduce write amplification by avoiding the unnecessary movement of values while sorting
2. decrease size of the LSM-tree

Separating keys from values introduces a number of challenges and optimization opportunities

1. 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.

2. 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.

3. separating keys and values makes crash consistency challenging;

   WiscKey leverages an interesting property in modern file systems, that appends never result in garbage data on a crash.

3 Background and Motivation

3.1 Write and Read Amplification

Table 1: Write and Read Amplification

Data Size	Write Amplification	Read Amplification
1GB	3.1	8.2
100GB	14	327

3.2 Fast Storage Hardware

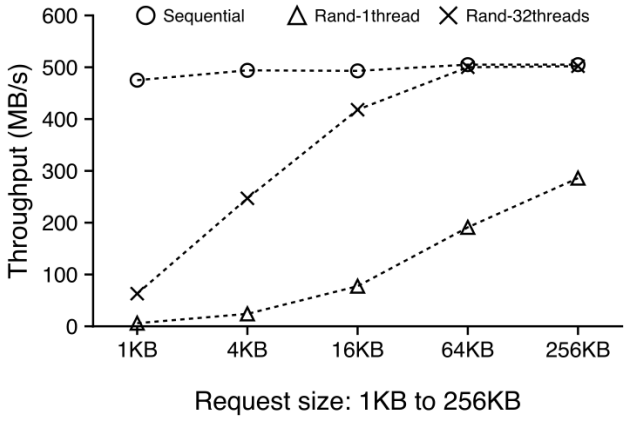

Figure 1: Sequential and Random Reads on SSD

4 WiscKey

To realize an SSD-optimized key-value store, WiscKey includes four critical ideas:

1. separate keys and values
2. to deal with unsorted values, WiscKey uses the parallel random-read characteristic of SSD devices as shown in Figure 1.
3. WiscKey utilizes unique crash-consistency and garbage-collection techniques to efficiently manage the value log.
4. removing the LSM-tree log without sacrificing consistency.

4.1 Design Goals

4.2 Key-Value Separation

Compaction only needs to sort keys, while values can be managed separately.

4.3 Challenges

4.3.1 Parallel Range Query

Based on Figure 1, parallel random reads with a fairly large request size can fully utilize the device's internal parallelism, getting performance similar to sequential reads.

To make range queries efficient, WiscKey leverages the parallel I/O characteristic of SSD devices to prefetch values from the vLog during range queries.

4.3.2 Garbage Collection

In WiscKey, only invalid keys are reclaimed by the LSM-tree compaction. Since WiscKey does not compact values, it needs a special garbage collector to reclaim free space in the vLog.

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 2: the tuple `(key size, value size, key, value)` is stored in the vLog.

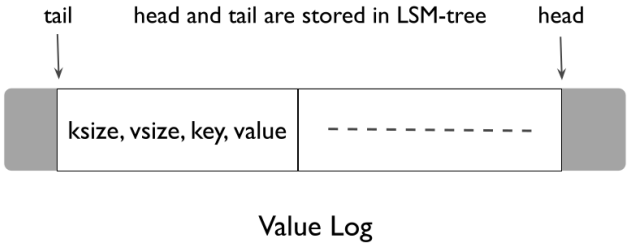

Figure 2: WiscKey New Data Layout for Garbage Collection

WiscKey's garbage collection aims to keep valid values in a contiguous range of tghe vLog, as shown in Figure 2. **head** always corresponds to the end of the vLog where new values will be appended. **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 searched during lookups.

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

1. After appending the valid values to the vLog, the garbage collection calls a `fsync()` on the vLog.

   > Calling `fsync()` does not necessarily ensure that the entry in the directory containing the file has also reached disk. For that an explicit `fsync()` on a file descriptor for the directory is also needed.

2. 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, tail-vLog-offset>`

3. the free space in the vLog is reclaimed.

4.3.3 Crash Consistency

WiscKey provides same crash guarantees by using an interesting property of modern file systems (ext4, btrfs, xfs).

Consider a file that contains the sequence of bytes $\langle b_1 b_2 \dots b_n \rangle$ and the user appends the sequence $\langle b_{n+1} b_{n+2} \dots b_{n+m} \rangle$ 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 $\langle b_1 \dots b_n \dots b_{n+1} \dots b_{n+x} \rangle$ where $x < m$ [**?**]. Since 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 future values 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 vLog before the crash, it will be garbage collected later.

- if the key could be found in the LSM-tree, an additional step is required to maintain consistency.

1. verifies 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
2. if the verification fails, 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.

4.4 Optimizations

4.4.1 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. Figure 3 shows the total time to sequentially write a 10-GB file in ext4.

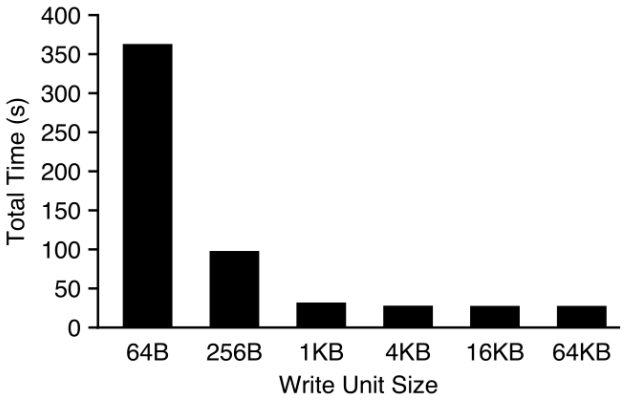

Figure 3: Impact of Write Unit Size

To reduce overhead, WiscKey buffers values in a userspace buffer, and flushes the buffer only when the buffer size exceeds a threshold or when the user requests a synchronous insertion.

TODO: how does leveldb handle crash

4.4.2 Optimizing the LSM-tree Log

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 in the previous section. 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. As to require scanning only a small portion of the vLog, WiscKey records the head of the vLog periodically in the LSM-tree, as a key-value pair `<head, head-vLog-offset>`

vLog is the WAL in essence.

5 Evaluation

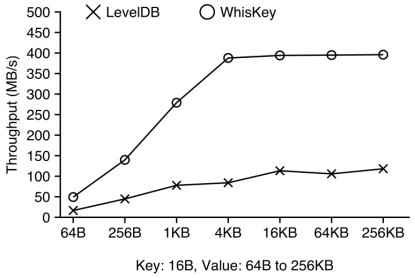

Figure 7: **Sequential-load Performance.** *This figure shows the sequential-load throughput of LevelDB and WiscKey for different value sizes for a 100-GB dataset. Key size is 16 B.*

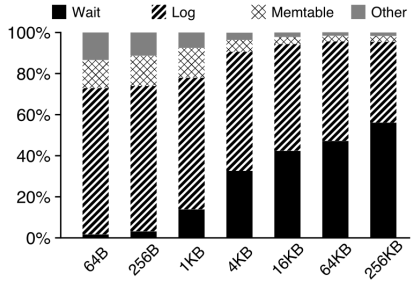

Figure 8: **Sequential-load Time Breakup of LevelDB.** *This figure shows the percentage of time incurred in different components during sequential load in LevelDB.*

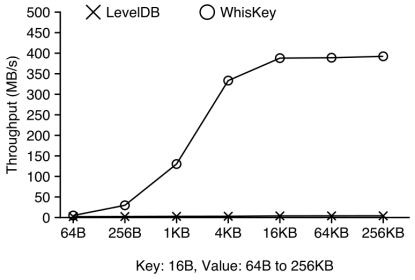

Figure 9: **Random-load Performance.** *This figure shows the random-load throughput of LevelDB and WiscKey for different value sizes for a 100-GB dataset. Key size is 16 B.*

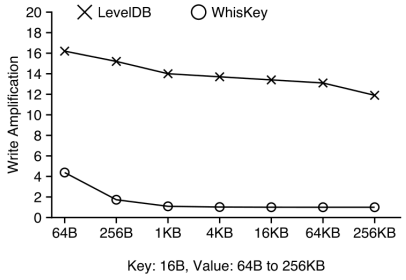

Figure 10: **Write Amplification of Random Load.** *This figure shows the write amplification of LevelDB and WiscKey for randomly loading a 100-GB database.*

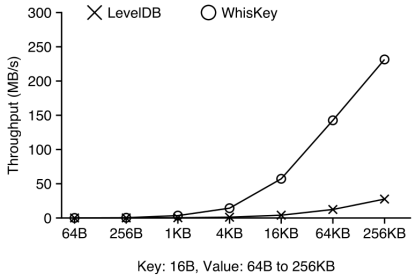

Figure 11: **Random Lookup Performance.** *This figure shows the random lookup performance for 100,000 operations on a 100-GB database that is randomly loaded.*

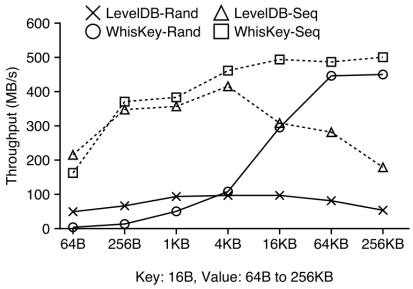

Figure 12: **Range Query Performance.** *This figure shows range query performance. 4 GB of data is queried from a 100-GB database that is randomly (Rand) and sequentially (Seq) loaded.*

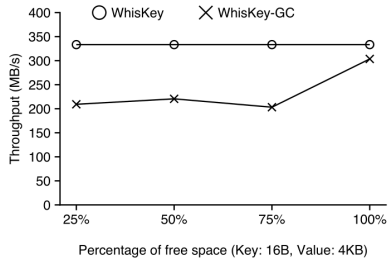

Figure 13: **Garbage Collection.** *This figure shows the performance of WiscKey under garbage collection for various free-space ratios.*

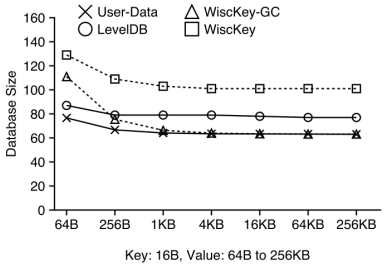

Figure 14: **Space Amplification.** *This figure shows the actual database size of LevelDB and WiscKey for a random-load workload of a 100-GB dataset. User-Data represents the logical database size.*

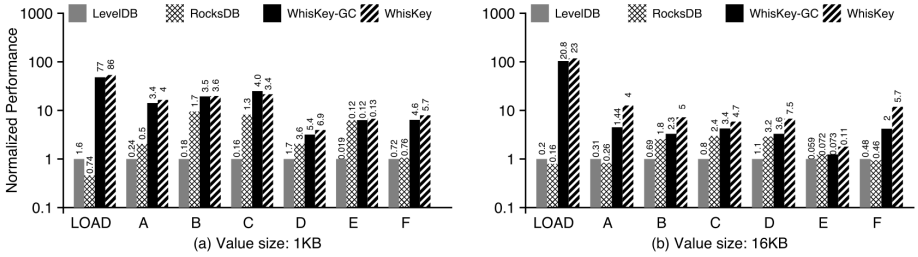

Figure 15: **YCSB Macrobenchmark Performance.** *This figure shows the performance of LevelDB, RocksDB, and WiscKey for various YCSB workloads. The X-axis corresponds to different workloads, and the Y-axis shows the performance normalized to LevelDB's performance. The number on top of each bar shows the actual throughput achieved (K ops/s). (a) shows performance under 1-KB values and (b) shows performance under 16-KB values. The load workload corresponds to constructing a 100-GB 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).*

5.1 Crash Consistency

Not good illustration

6 References

References

[PCA+14] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting Crash-Consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.