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

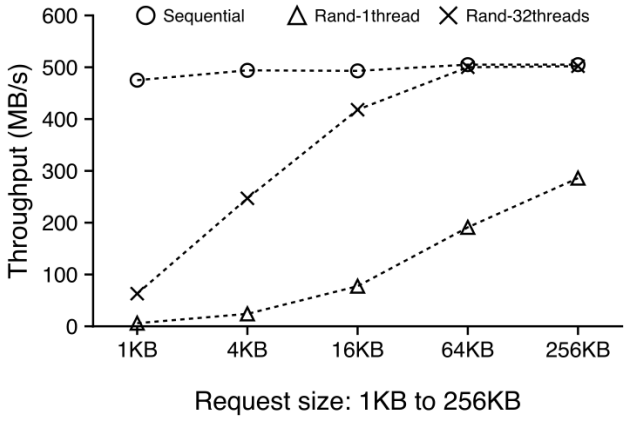

Figure 1: Sequential and Random Reads on SSD

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.

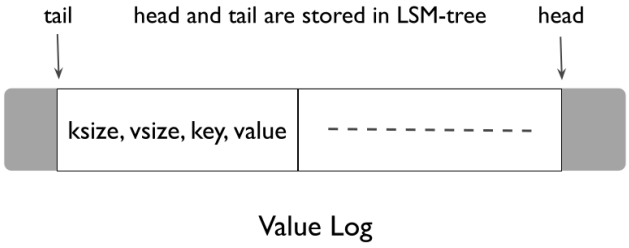

Figure 2: WiscKey New Data Layout for Garbage Collection

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.