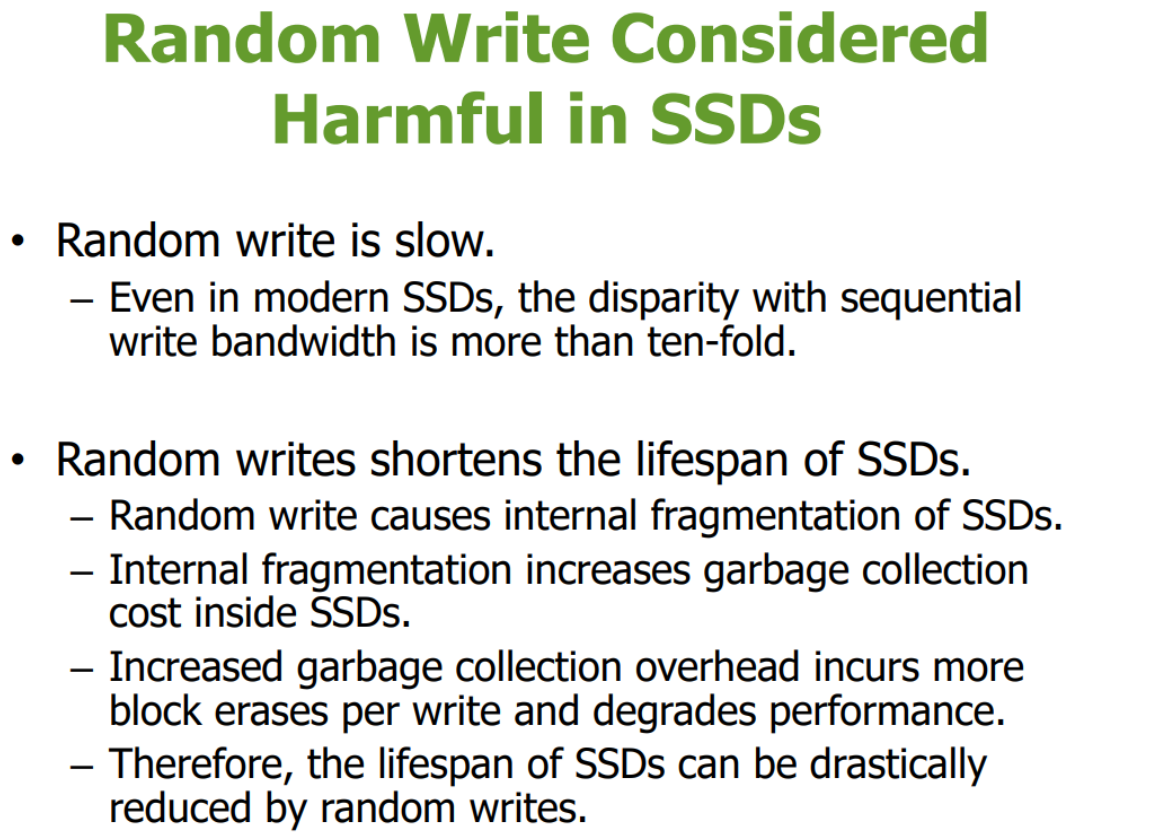
WiscKey -2016

BadgerDb is the Go implementation of WiscKey.

This paper aims to optimize LSM tree performance when SSDs are used as the storage device instead of HDDs.

The main advantage of LSMtrees over other indexing structures (such as B-trees) is that they maintain sequential access patterns for writes. 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

- **SSDs** use **flash memory chips** to store data electronically. This is similar to how data is stored on a USB drive.
- **HDDs** use **spinning magnetic platters** and a **read/write head** that moves across the platters to access data magnetically. This is a mechanical process.



The success of classic LSM-based technology is tied closely to its usage upon classic hard-disk drives (HDDs). In HDDs, random I/Os are over 100× slower than sequential ones;

thus, performing additional sequential reads and writes to continually sort keys and enable efficient lookups represents an excellent trade-off.

SSDs have a large degree of internal parallelism->

https://claude.ai/share/5f0451d0-7d31-4374-9ac8-f58a76887983

SSDs can wear out through repeated writes ; the high write amplification in LSMtrees can significantly reduce device lifetime.

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 WiscKey we separate the keys from the values , this leads to lower write amplification

As we only have to sort the keys and no longer have to sort the values.

However it affects the range query operations as values are not stored in sorted order anymore. We can solve this problem by using the parallelism which SSDs provide.

WiscKey also suggests that we use a garbage collector which works in the background to remove keys that have been deleted from our LSM tree.

WiscKey also removes the LSM tree log which is usually maintained resulting in a reduced number of system call overheads.

WiscKey provides the usual run of the mill APIs like Put(key, value), Get(key), Delete(key) and Scan(start, end).

The main design goals of WiscKey are:

Low write amplification: Write amplification introduces extra unnecessary writes. Even though 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, so as to improve workload performance and SSD 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 speedup 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 effectively utilized so that applications can fully utilize the device's 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., 16 B) , though value sizes can vary widely (e.g., 100 B to larger than 4 KB). WiscKey aims to provide high performance for this realistic set of key-value sizes

How WiscKey implements Key-Value Separation:

During compaction we do not need to compact values, as long as we keep the keys compacted the values can be maintained separately. Since the size of the keys are much smaller than the values , this improves performance significantly in terms of the amount of data that is needed during sorting.

In WiscKey the values are stored separately and only the location of the value is stored with the key. This leads to the LsmTree of WiscKey being much smaller than the corresponding tree of LevelDb and this leads to a much smaller write amplification.

Assuming a 16-B key, a 1- KB value, and a write amplification of 10 for keys (in the LSM-tree) and 1 for values, the effective write amplification of WiscKey is only (10 × 16 + 1024) / (16 + 1024) = 1.14.

This smaller write amplification improves the lifetime of the SSDs.

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. You 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 may search fewer levels of table files in the LSM-tree and 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 a lookup performance better than LevelDB.

In the WiscKey architecture keys are stored in the LSM tree and the values are mantained separately in a valueLog file (vLOG). The address of this file is stored along with the key in the LSM tree. When the user inserts a key-value pair in WiscKey , the value is first

appended in the vLog and the key is inserted along with the value's address (vLog-offset,value-size). Deleting a key simply deletes it from the LSM tree, without touching 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.

Challenges in this Architecture:

Range Queries:

For range queries, LevelDB provides the user with an iterator-based interface with Seek(key), Next(), Prev(), Key() and Value() operations. To scan a range of keyvalue 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.

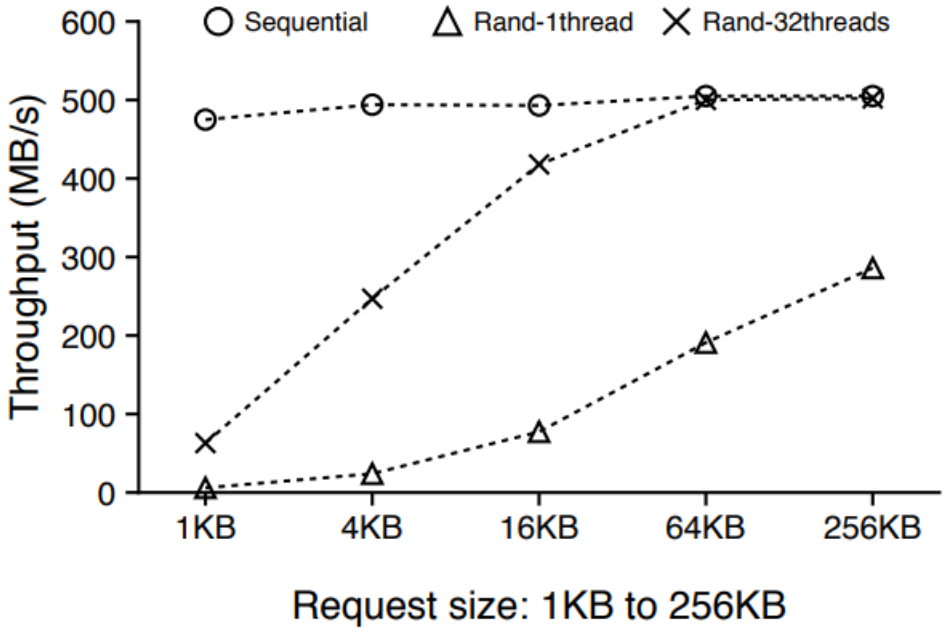


Figure 3: **Sequential and Random Reads on SSD.** *This figure shows the sequential and random read performance for various request sizes on a modern SSD device. All requests are issued to a 100-GB file on ext4.*

As we see in the above Figure, the random read performance of a single thread on SSD cannot match the sequential read performance. However, 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. The underlying idea is that, with SSDs, only keys require special attention for efficient retrieval. So long as keys are retrieved efficiently, range queries can use parallel random reads for efficiently retrieving values.

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 will fetch these addresses from the vLog concurrently in the background.

GarbageCollection in WiscKey

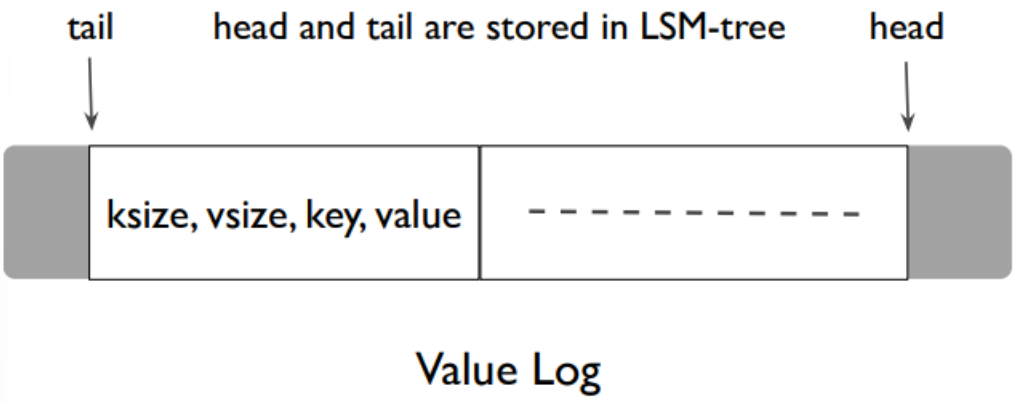


Figure 5: **WiscKey New Data Layout for Garbage Collection.** *This figure shows the new data layout of WiscKey to support an efficient garbage collection. A head and tail pointer are maintained in memory and stored persistently in the LSM-tree. Only the garbage collection thread changes the tail, while all writes to the vLog are append to the head.*

Since WiscKey does not compact values, it needs a special garbage collector to reclaim free space in the vLog.

WiscKey targets lightweight and online garbage collector. While storing values in the vLog, the corresponding key is also stored along with the value. The new data layout is shown in Figure 5.

WiscKey's garbage collection aims to keep valid values (that do not correspond to deleted keys) in a contiguous range of the vLog.

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 searched during lookups.

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

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 a fsync() on the vLog.

Then, it adds these new value's addresses and current tail to the LSMtree in a synchronous manner; the tail is stored in the LSM-tree as <''tail'', tail-vLog-offset>.

Finally, the free space in the vLog is reclaimed.