WiscKey: Separating Keys from Values in SSD-Conscious Storage

Lanyue Lu, Thanumalayan Pillai,

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin-Madison

2019-05-29

presented by **삼삼하조 (Hwajung Kim, Sunggon Kim)**

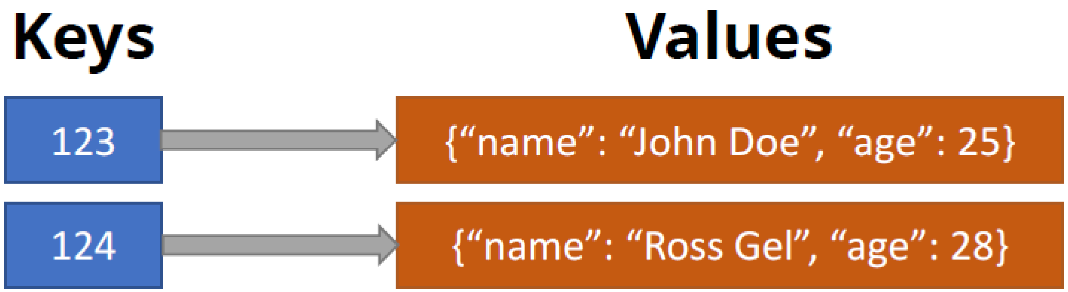
Paper in a nutshell

- **Motivation**
  - Large write/read amplification on LSM-trees of Key-Value Stores
  - LSM-trees are optimized for HDDs; not optimal for SSDs
- **Main Idea**: Separating Keys from Values
- **Challenges and Optimizations**
  - Parallel Range Query
  - Garbage Collection
  - Crash Consistency
- **Performance**
  - 2.5x to 111x for random loading, 1.6x to 14x for random lookups
- **Limitation**
  - Random lookup performance is limited to device's parallel random-read performance
  - Performance is worse when request size (value size) is small
  - High space amplification

Table of Contents

Key-Value Store

- Store any arbitrary value for a given key

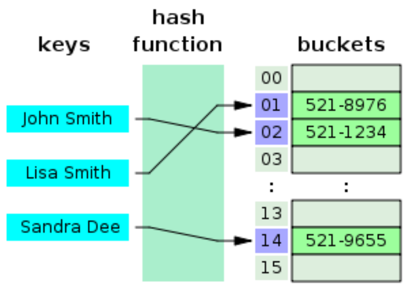


- Insertions: put (key, value)
- Point lookups: get (key)
- Range queries: get_range (key1, key2)

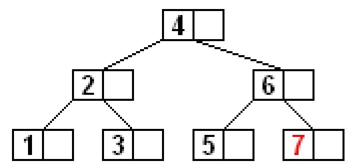
Source: https://www.sigops.org/s/conferences/sosp/2017/slides/pebblesdb-sosp17-slides.pdf

Key-Value Store

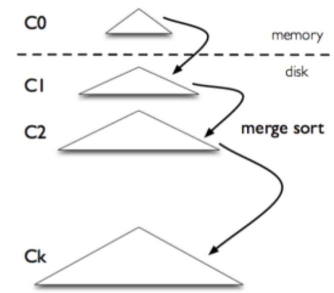
- Key-value stores are important
  - web indexing, e-commerce, social networks
  - various key-value stores



Hash Table



B-Tree



LSM-Tree

- LSM-tree based key-value stores are popular
  - optimize for write-intensive workloads
  - widely deployed

BigTable

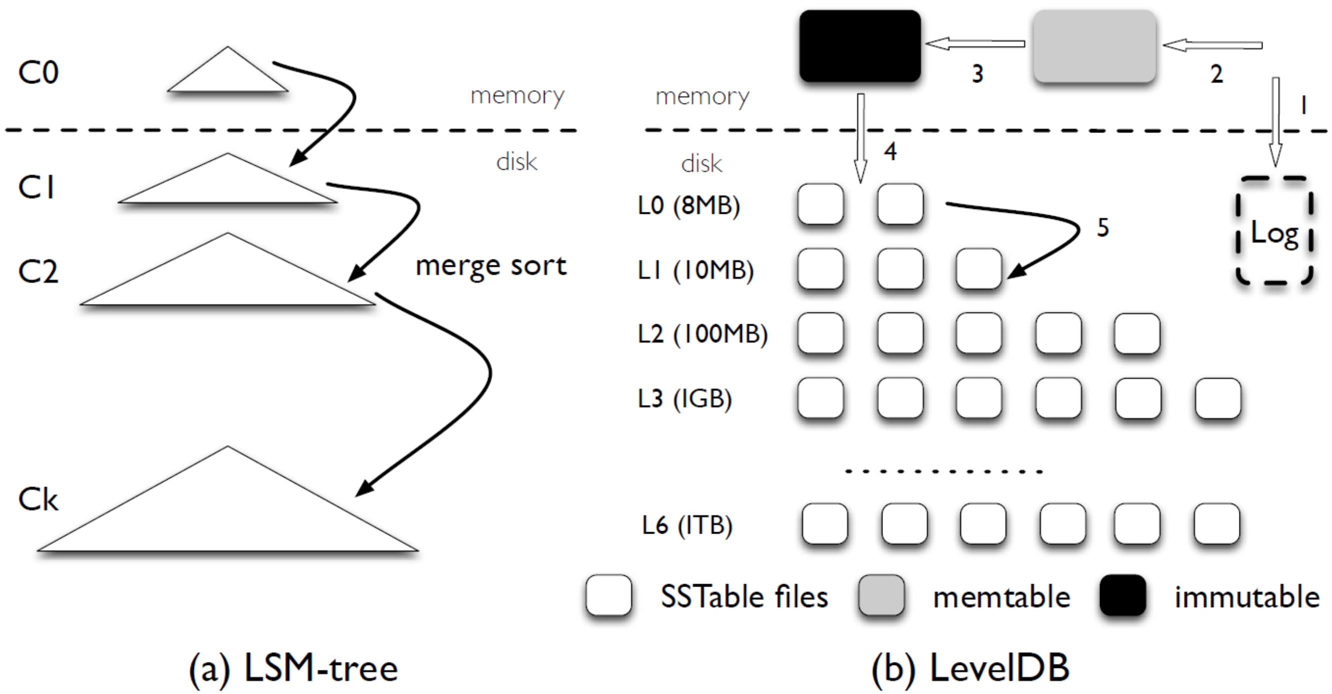
Why LSM-trees?

- Good for hard drives 
  - batch and sequential write
  - high sequential throughput
  - sequential access up to 1000x faster than random access

- Not optimal for SSDs 
  - large write/read amplification
    → wastes device resources
  - unique characteristics of SSDs
    → fast random reads due to the internal parallelism

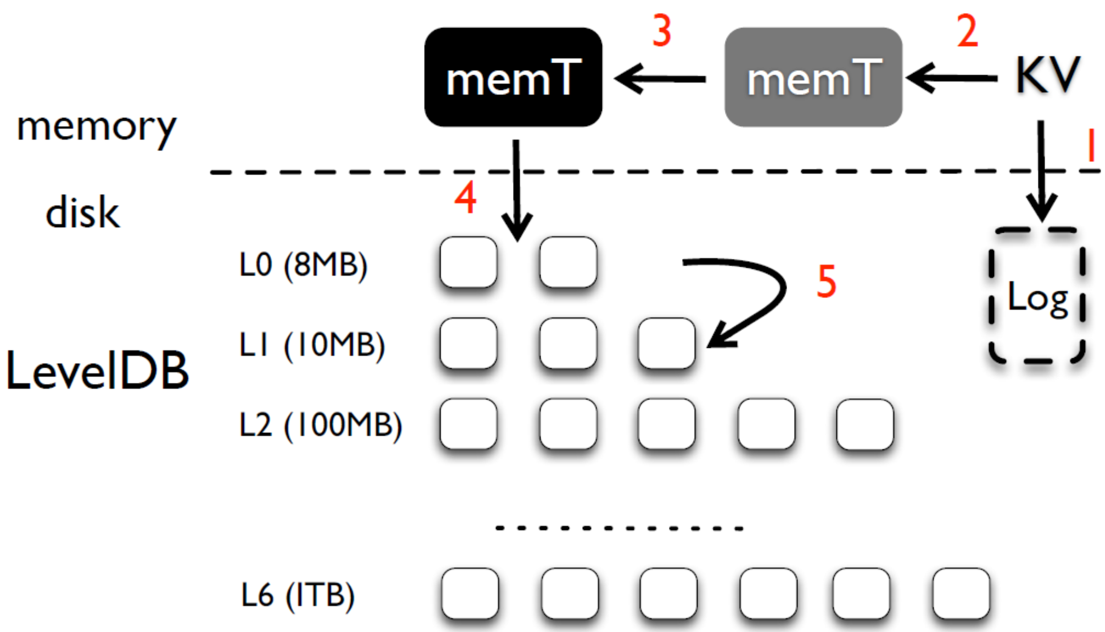
Log-Structured Merge-Tree

- persistent structure that provides efficient indexing for a key-value store
- defers and batches data writes into large chunks
- consists of a number of components of exponentially increasing sizes, C_0 to C_k
  - C_0 : memory-resident update-in-place sorted tree
  - C_1 to C_k : disk-resident append-only B-trees



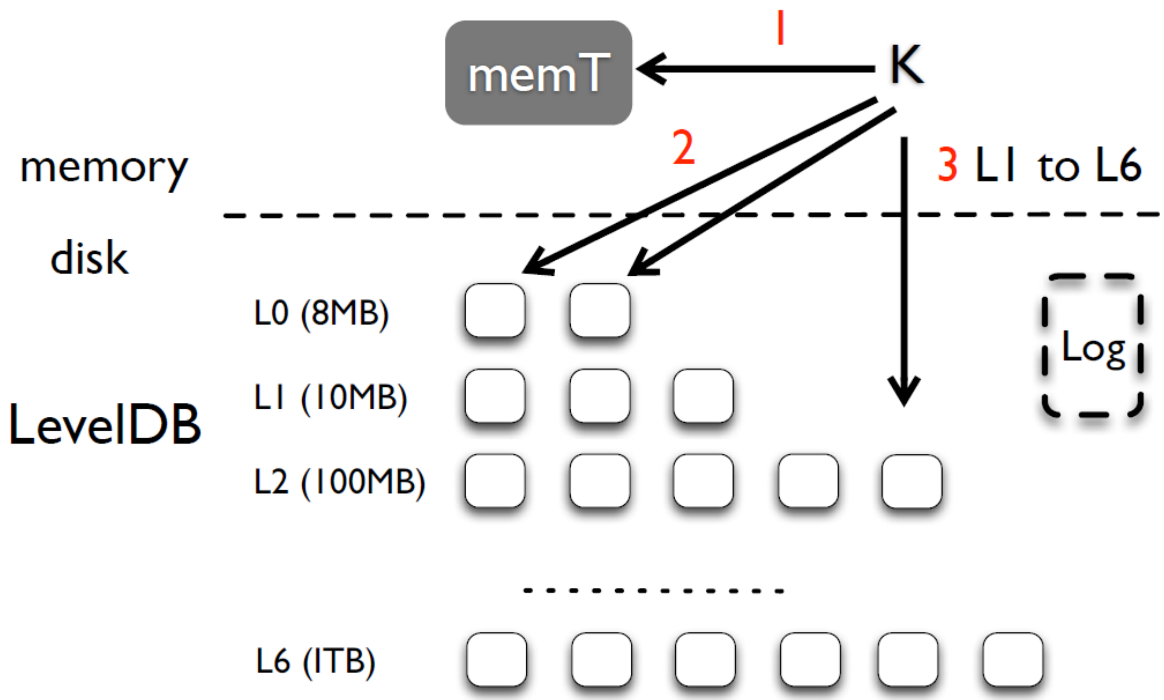
LSM-trees: Insertion

- Write ahead logging (WAL)
- Sort data for quick lookups
- Write sequentially on disk (flush)
- Merge and sort unsorted keys periodically (compaction)
- Sorting and garbage collection are coupled

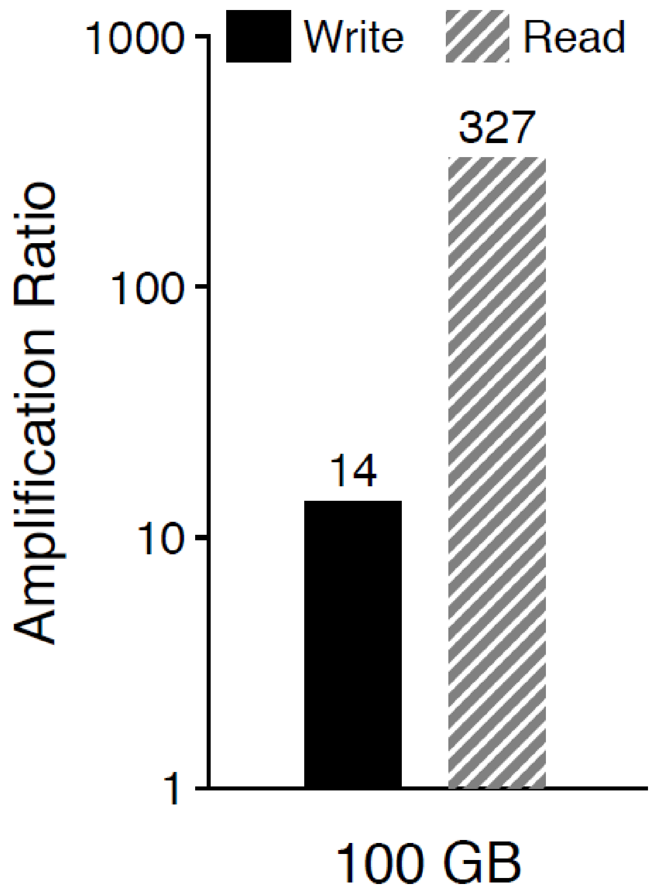


LSM-trees: Lookup

- Searches all files reside in Level 0
- If not found, searches higher levels until found
- Travel many levels for a large LSM-tree



I/O Amplification in LSM-trees



Random load:
a 100GB database

Random lookup:
100,000 lookups

Problems:

large write amplification

large read amplification

Why is write amplification bad?

- Reduces the write throughput
- Flash devices wear out after limited write cycles

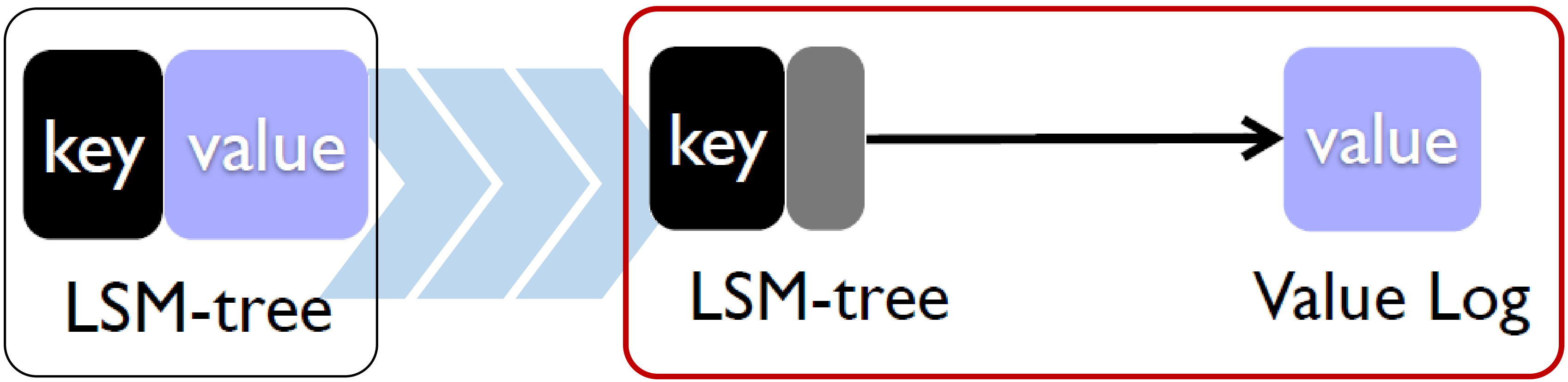
Intel SSD DC P4600
can last ~5 years assuming **~5TB** write per day

RocksDB
can write **~500GB** of user data per day
to a SSD
to last **1.25 years**

Source: https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p4600-series/dc-p4600-1-6tb-2-5inch-3d1.html

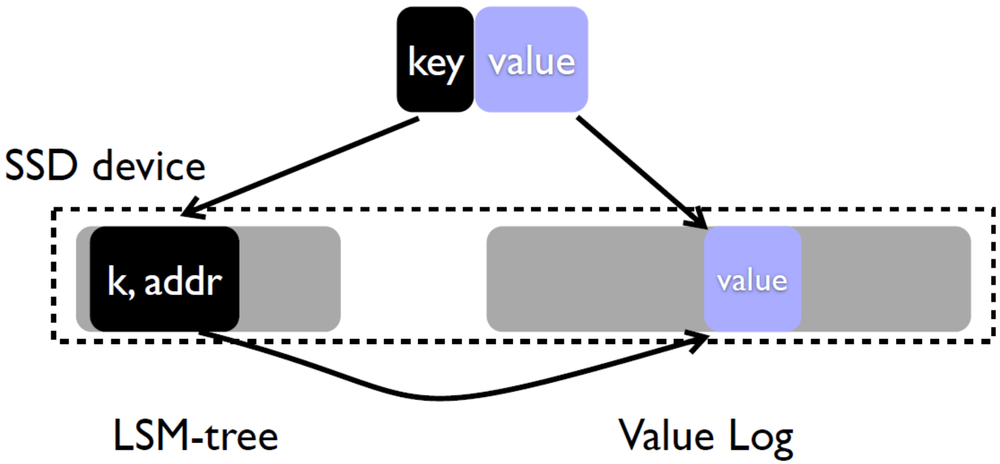
WiscKey: Key-Value Separation

- **Separate keys from values**
  - decouple sorting and garbage collection



WiscKey: Key-Value Separation

- Main idea: only keys are required to be sorted
- Decouple sorting and garbage collection



improves
write performance of applications
SSD's lifetime by requiring fewer erase cycles

- Example) 16B key, 1KB value, write-amplification: 10

	LSM-tree (existing)	WiscKey (optimized)
Write Amount	10*(16 + 1024)	10*16 + 1024
Write Amplification	10	**1.14**

WiscKey: Smaller LSM-tree

load 100 GB database

	LevelDB
limits of files	**num of files**
L0	9
L1 (5)	30
L2 (50)	365
L3 (500)	2184
L4 (5000)	15752
L5 (50000)	23733
L6 (500000)	0

Large LSM-tree:

Intensive compaction
- ➡ repeated reads/writes
- ➡ stall foreground I/Os

Many levels
- ➡ travel several levels for each lookup

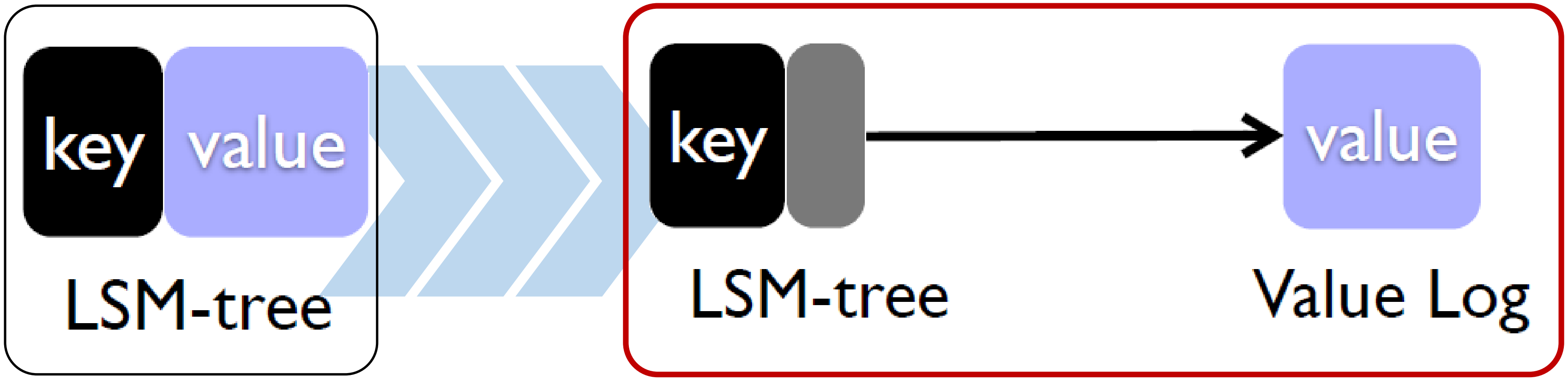
WiscKey: Smaller LSM-tree

load 100 GB database

	LevelDB	WiscKey
limits of files	**num of files**	**num of files**
L0	9	7
L1 (5)	30	11
L2 (50)	365	127
L3 (500)	2184	460
L4 (5000)	15752	0
L5 (50000)	23733	0
L6 (500000)	0	0

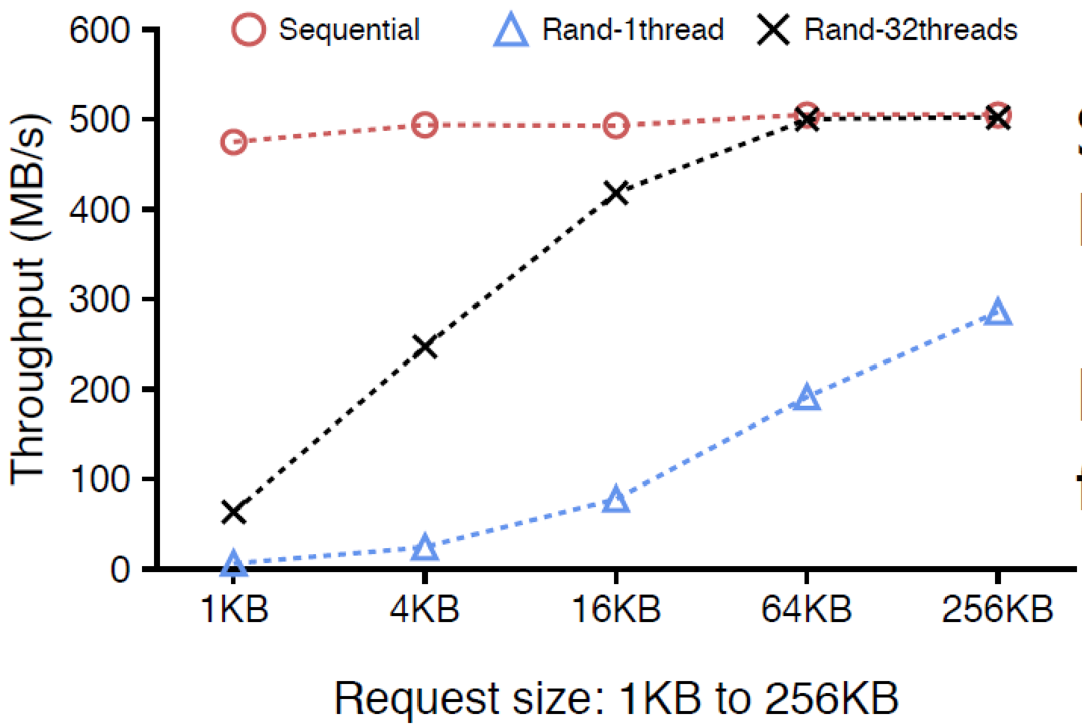
Small LSM-tree: less compaction, fewer levels to search, and better caching

- **Challenges due to decoupling Keys from Values**
  - Slow lookup performance
    → utilize SSD's internal parallelism for range queries
  - Additional garbage collection operation
    → online and light-weight garbage collection
  - Crash consistency
    → minimize I/O amplification and crash consistent



SSD Read Performance

- SSD Read Performance
  - Sequential, Random, Parallel

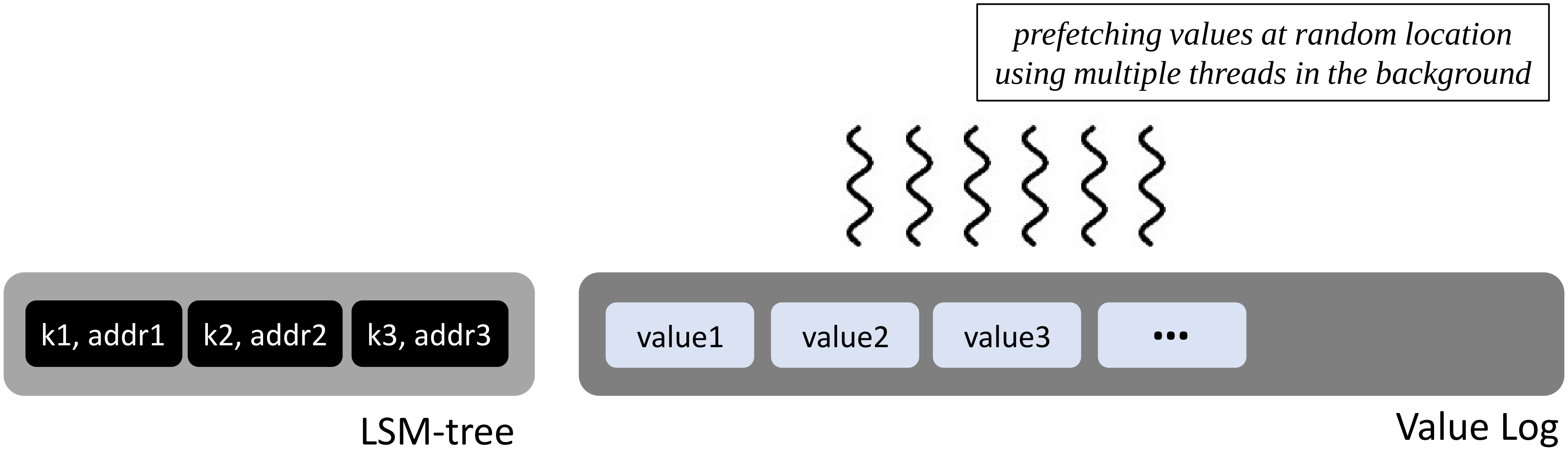


SSD: Samsung 840 EVO 500GB

Reads on a 100GB file on ext4

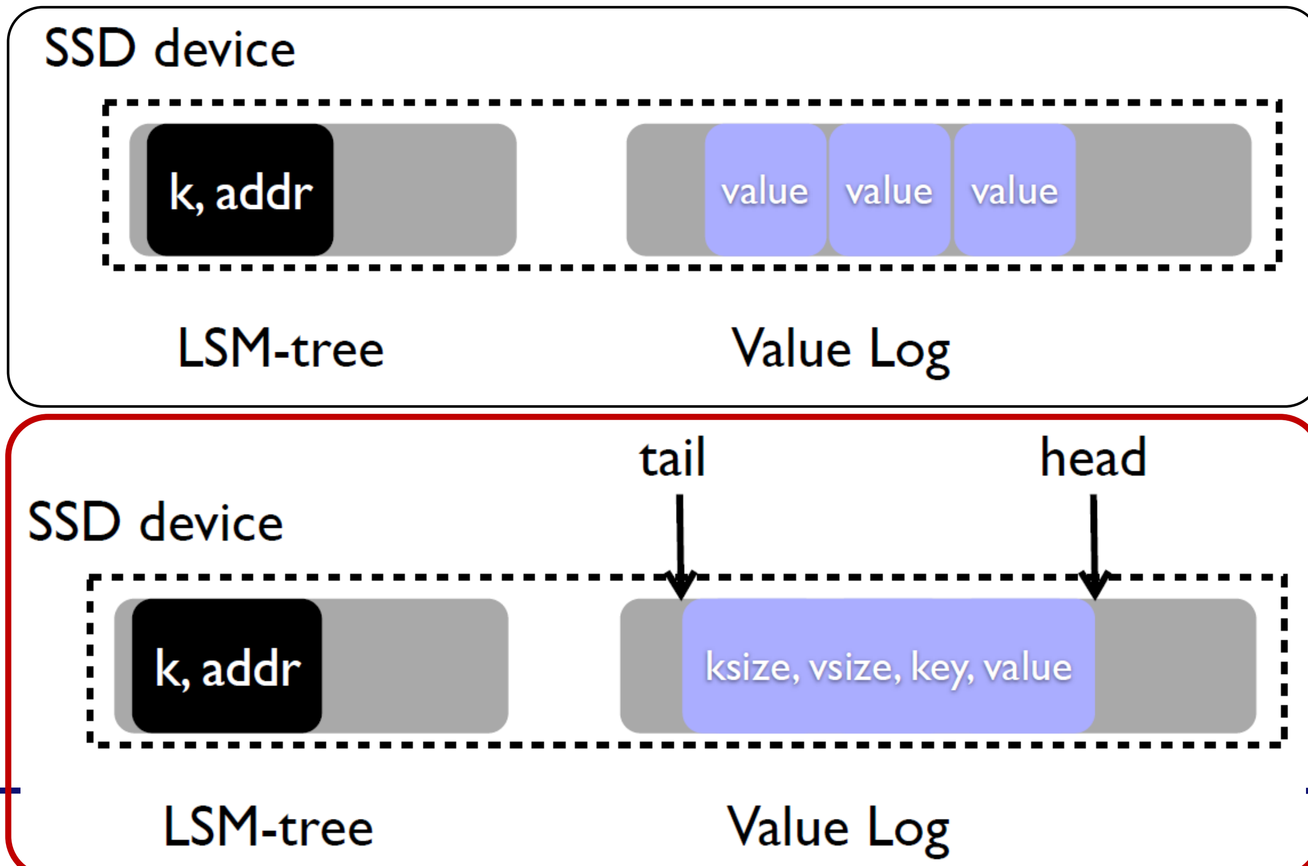
Parallel Range Query

- Parallel range query
  - leverage parallel random reads of SSDs
  - prefetch key-value fairs in advance
    - range query interface: `seek(), next(), prev()`
    - detect a sequential pattern
    - prefetch concurrently in background



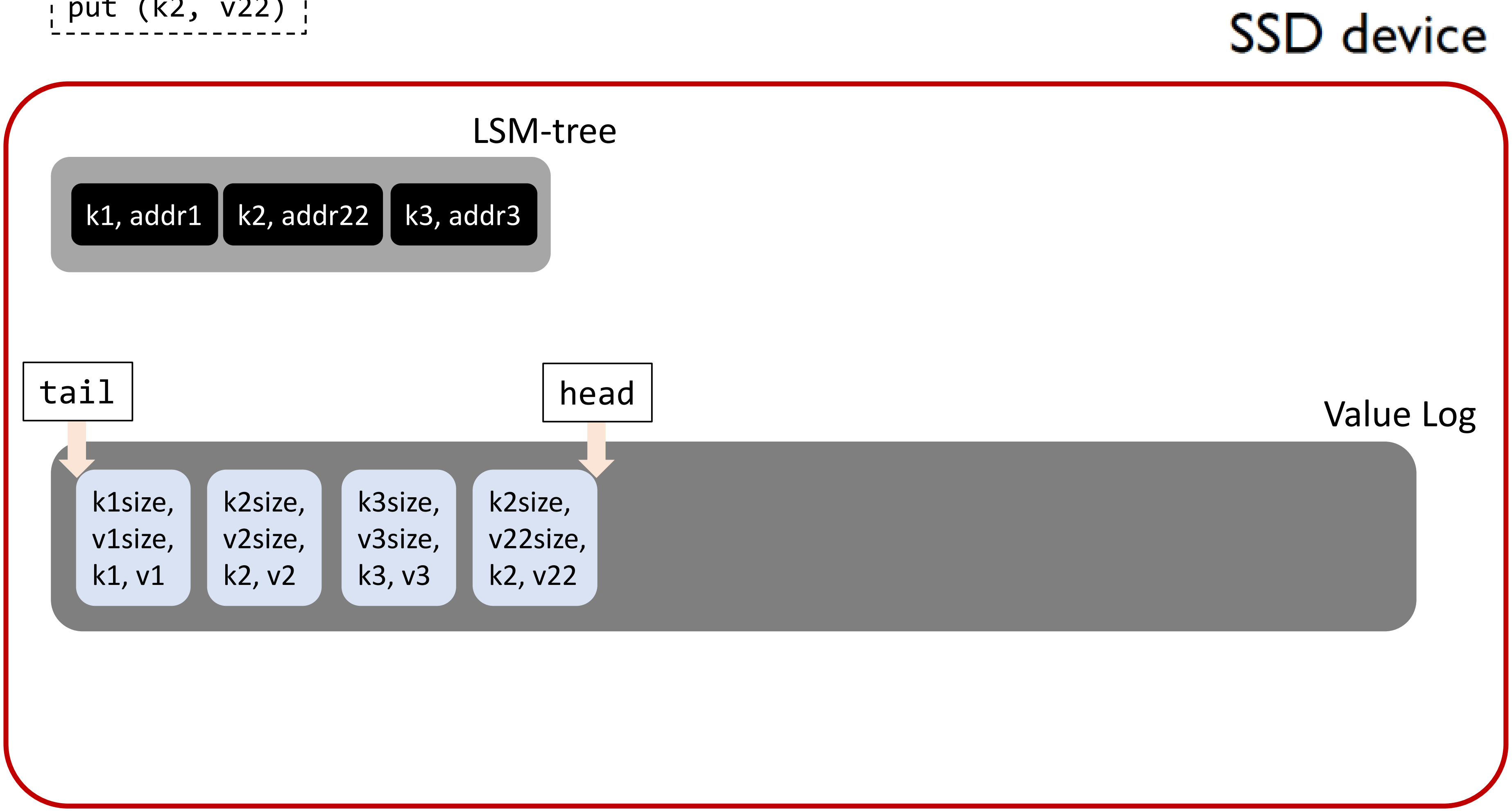
Garbage Collection

- Online and light-weight garbage collection
  - append (ksize, vsize, key, value) in value log
- Remove LSM-tree log in WiscKey
  - store head in LSM-tree periodically
  - scan the value log from the head to recover



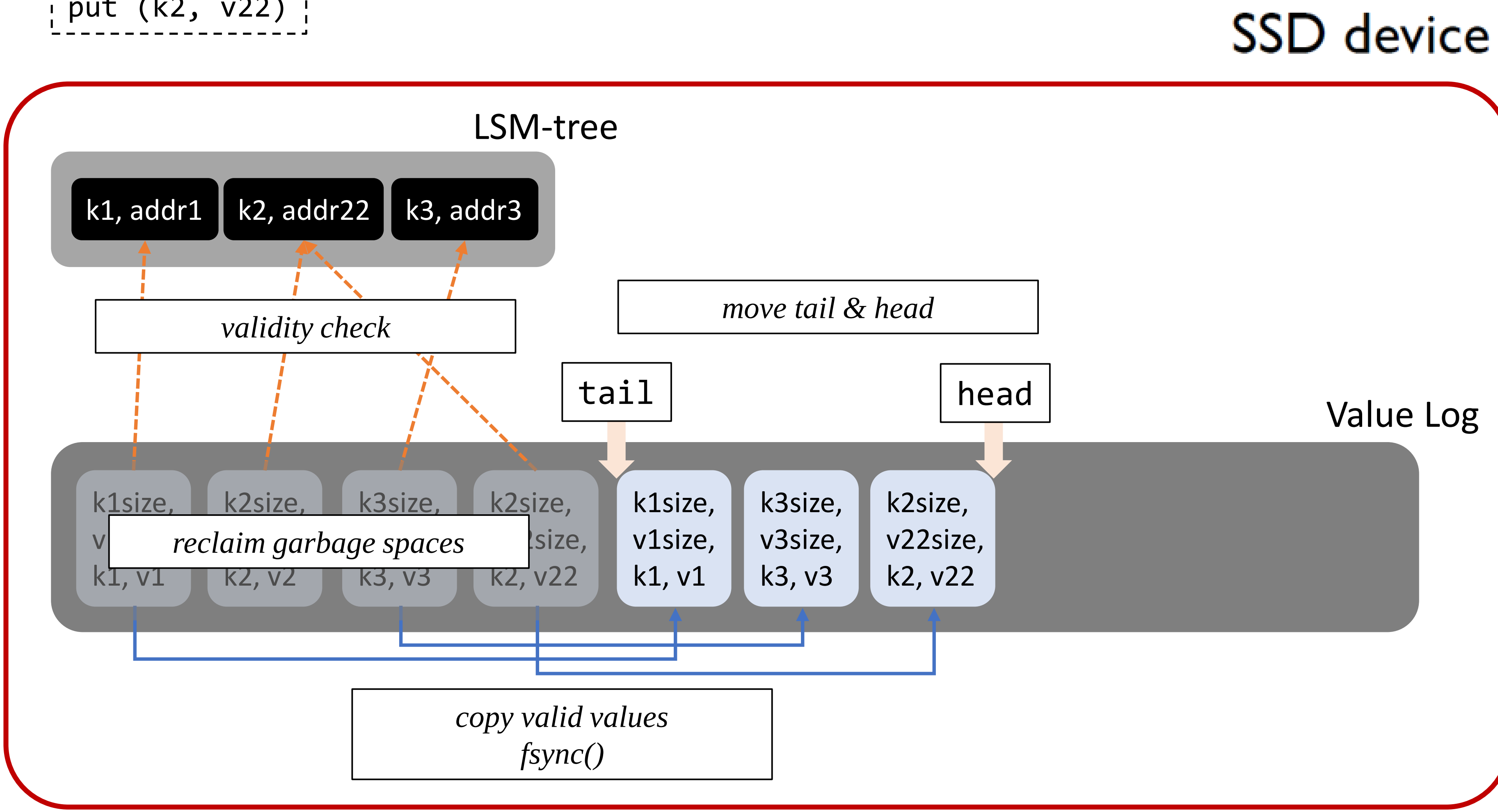
Garbage Collection


```
put (k1, v1)
put (k2, v2)
put (k3, v3)
put (k2, v22)
```



Garbage Collection


```
put (k1, v1)
put (k2, v2)
put (k3, v3)
put (k2, v22)
```



Crash Consistency

- exploit the property of modern file systems (ext4, btrfs, xfs)
- not possible for random bytes or a non-prefix subset of the appended bytes to be added to the file

a file

$\langle b_1 b_2 b_3 \ldots b_n \rangle$

user appends sequence $<b_{n+1} \ldots b_{n+m}>$

$\langle b_1 b_2 b_3 \ldots b_n \rangle$ $\langle b_{n+1} b_{n+2} b_{n+3} \ldots b_{n+m} \rangle$

crash happens

a file contains following sequence

$\langle b_1 b_2 b_3 \ldots b_n b_{n+1} b_{n+2} b_{n+3} \ldots b_{n+x} \rangle \; \exists \; x < m$

File System Property

if a value X in the vLog (Value Log) is lost in a crash, all future values (inserted after X) are lost too

- Value that has no corresponding key → garbage collected later
- Value that has corresponding key → verifies the value in the valid vLog range

Implementation

- Based on LevelDB v1.18
  - a separate vLog file for values
  - modify I/O paths to separate keys and values
  - leverages most of high-quality LevelDB source code

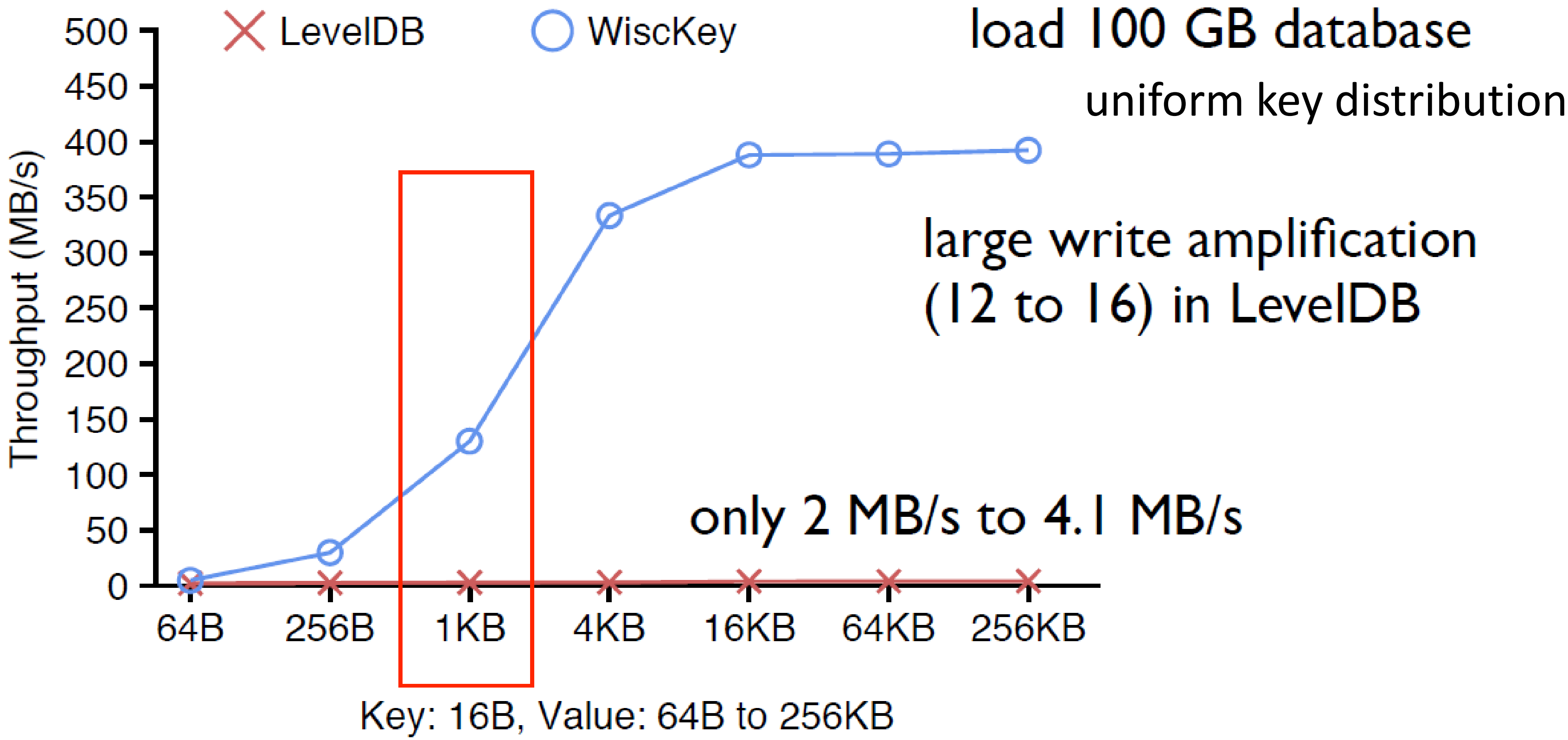
- Range query
  - thread pool *(32 threads)* launches queries in parallel
  - detect sequential pattern with the Iterator interface

- File-system support
  - `fadvise()` to predeclare access patterns
    - random read vLog *for lookup*, sequential read *for garbage collector*
  - hole-punching to free space (`fallocate()`)

Experimental Setup

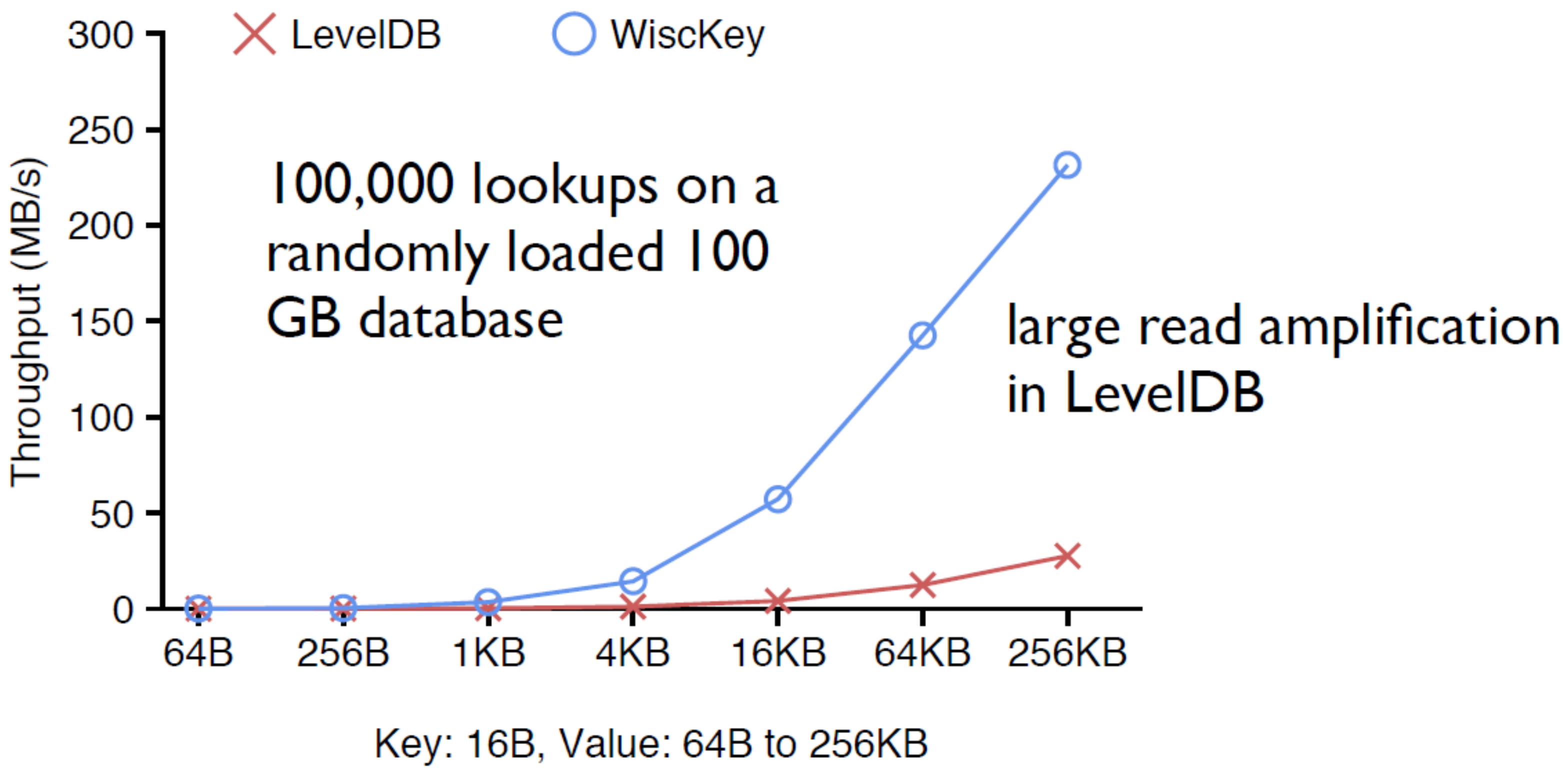
- Testing machine
  - CPU: 2 Intel Xeon CPU E5-2667 v2 @ 3.30GHz *(16-core/32-threads total)*
  - Memory: 64 GB
  - Storage: 500GB Samsung 840 EVO SSD
    - 500MB/s sequential-read, 400MB/s sequential-write
    - 500MB/s random-read *(32-threads, 256KB request)*
  - Operating System: 64-bit Linux 3.14
  - File System: EXT4
- Microbenchmark
  - db_bench: the default microbenchmarks in LevelDB
  - 16B key size, various value sizes (compression disabled)
- YCSB Benchmarks
  - LevelDB, RocksDB and wiskKey, on a 100 GB database
  - 16B key size, 1KB and 16KB value sizes (compression disabled)

Microbenchmark: Random Load



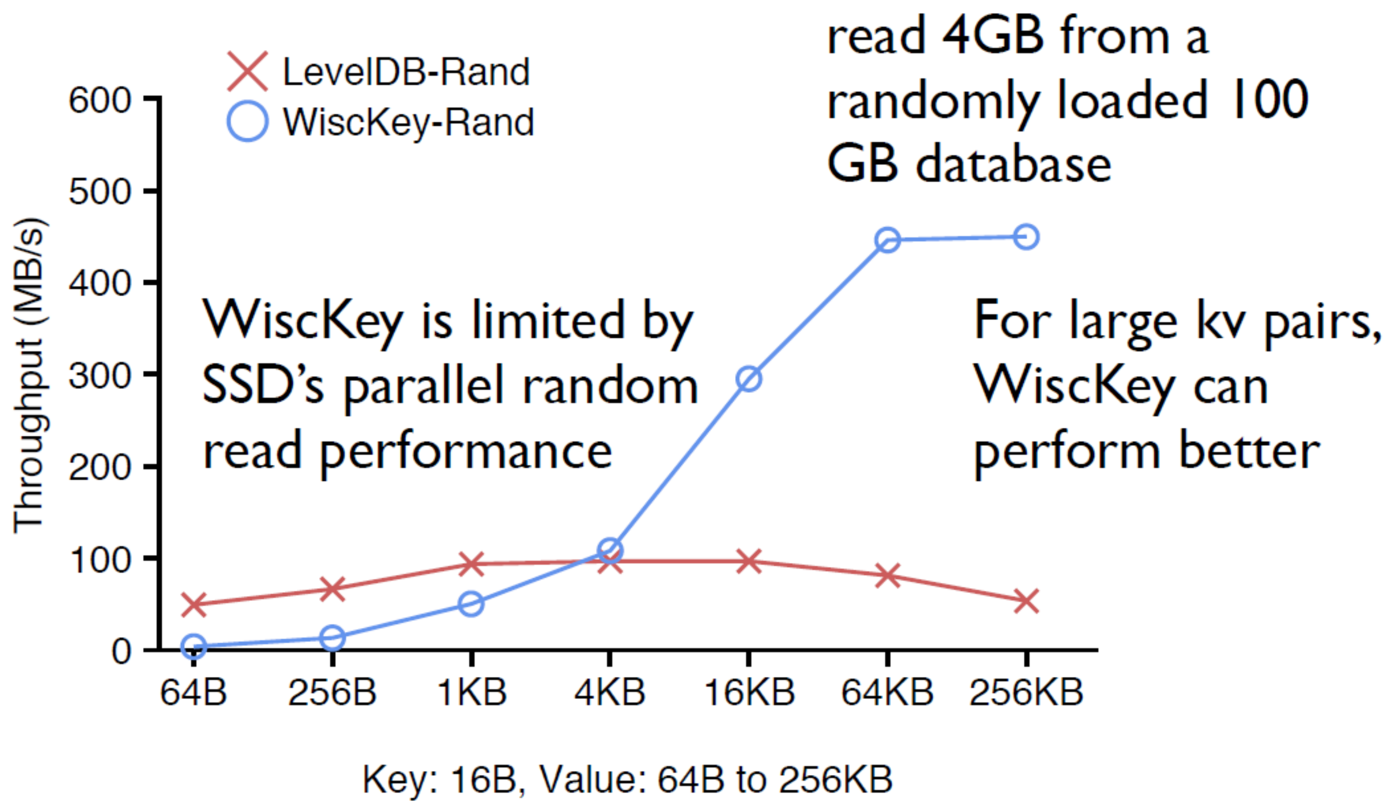
Small write amplification in WiscKey due to key-value separation (up to 111x in throughput)

Microbenchmark: Random Lookup



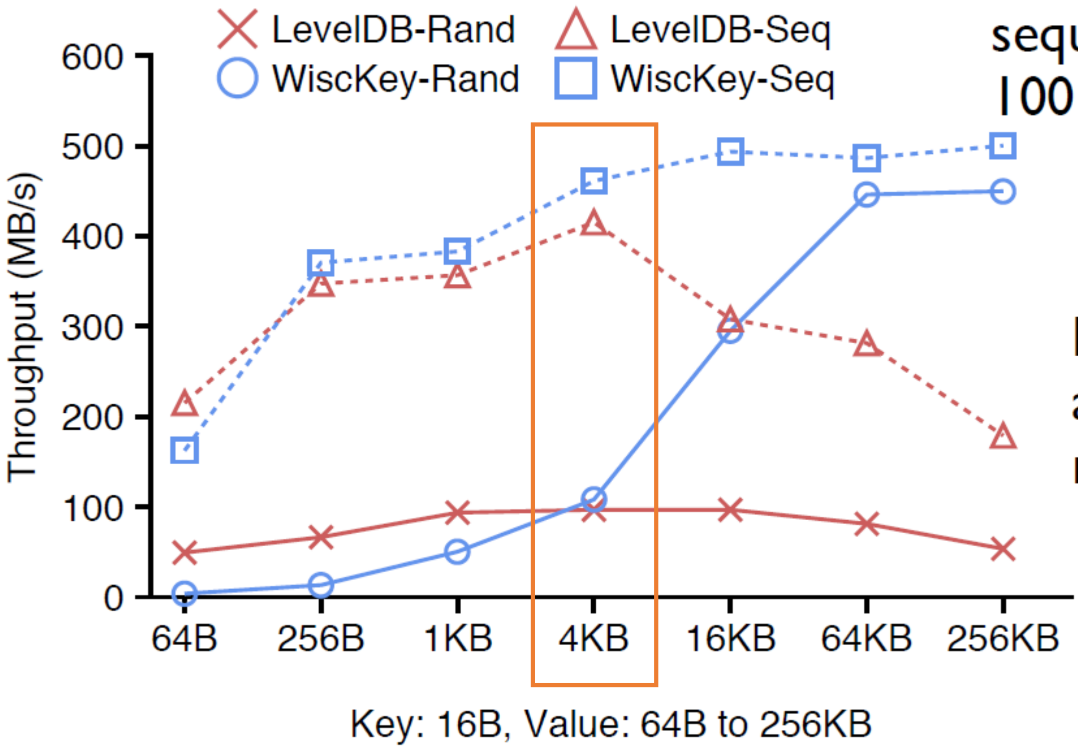
Smaller LSM-tree in WiscKey leads to better lookup performance (1.6x - 14x)

Microbenchmark: Range Query



Better for large kv pairs, but worse for small kv pairs on an unsorted database

Microbenchmark: Range Query



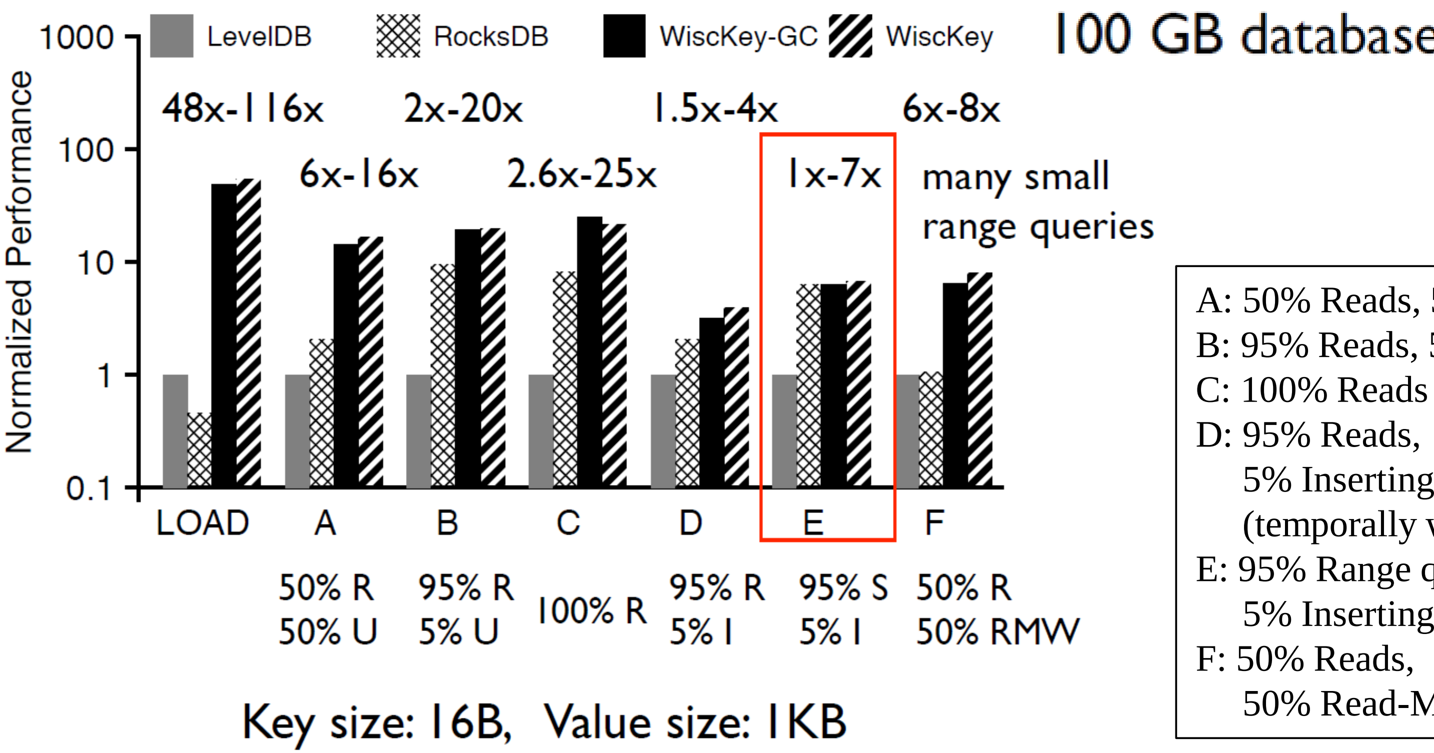
read 4GB from a sequentially loaded 100 GB database

Both WiscKey and LevelDB read sequentially

Sorted databases help WiscKey's range query

YCSB Benchmarks

- Yahoo! Cloud Serving Benchmark
  - Industry standard macro-benchmark
- WiscKey-GC: worst-case performance, GC always happening in the background
- RocksDB: SSD-optimized version of LevelDB with many optimizations



A: 50% Reads, 50% Updates
B: 95% Reads, 5% Updates
C: 100% Reads (Zipf)
D: 95% Reads,
 5% Inserting New Keys
 (temporally weighted distribution)
E: 95% Range queries,
 5% Inserting New Keys (Zipf)
F: 50% Reads,
 50% Read-Modify-Writes (Zipf)

Related Work

- Tucana: Design and Implementation of Fast and Efficient scale-up Key-value Store (ATC '16)
  - Uses Copy-one-write to achieve crash consistency without log (SSD)
- SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-sorted Data (VLDB '17)
  - Uses semi-sorted data structure to reduce lookup overhead (2-level lookup rather than N level of levelDB)
- Falcon: Scaling IO performance in Multi-SSD Volumes (ATC '17)
  - Optimizes IO stack to issue IO requests in a batch manner per volume (rather than fine-tuning number of threads for parallelism)
- HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems (ATC '17)
  - Utilizes NVM to store persistent index and DRAM to support range scan

- PebblesDB (SOSP'17), NoveLSM (ATC '18), SLM-DB (FAST '19)

Conclusion

- WiscKey: a LSM-tree based key-value store
  - **decouple** sorting and garbage collection by **separating keys from values**
  - SSD-conscious designs
  - significant performance gain

- Transition to new storage hardware
  - understand and leverage existing software
  - explore new designs to utilize the new hardware
  - get the best of two worlds