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

Key-Value Stores

Key-Value Stores

Key-value stores are important

- web indexing, e-commerce, social networks
- various key-value stores
  - hash table, b-tree
  - log-structured merge trees (LSM-trees)

Key-Value Stores

Key-value stores are important

- web indexing, e-commerce, social networks
- various key-value stores
  - hash table, b-tree
  - log-structured merge trees (LSM-trees)

LSM-tree based key-value stores are popular

- optimize for write intensive workloads
- widely deployed
  - BigTable and LevelDB at Google
  - HBase, Cassandra and RocksDB at FaceBook

Why LSM-trees ?

Why LSM-trees ?

Good for hard drives

- batch and write sequentially
- high sequential throughput
- sequential access up to 1000x faster than random

Why LSM-trees ?

Good for hard drives

- batch and write sequentially
- high sequential throughput
- sequential access up to 1000x faster than random

Not optimal for SSDs

- large write/read amplification
  - wastes device resources

Why LSM-trees ?

Good for hard drives

- batch and write sequentially
- high sequential throughput
- sequential access up to 1000x faster than random

Not optimal for SSDs

- large write/read amplification
  - wastes device resources
- unique characteristics of SSDs
  - fast random reads
  - internal parallelism

Our Solution: WiscKey

Our Solution: WiscKey

Separate keys from values

Our Solution: WiscKey

Separate keys from values

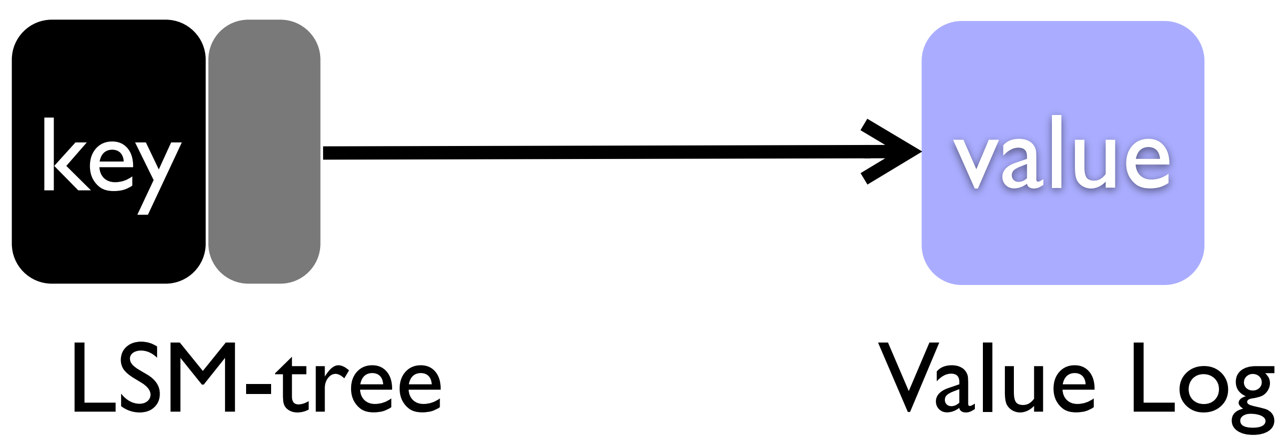
- ➡ decouple sorting and garbage collection

LSM-tree

Our Solution: WiscKey

Separate keys from values

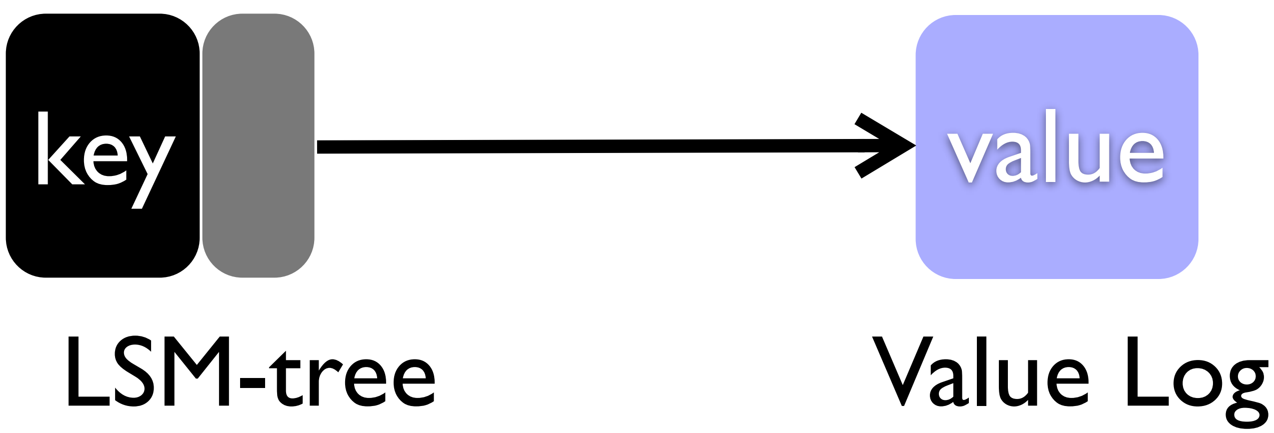
- ➡ decouple sorting and garbage collection



Our Solution: WiscKey

Separate keys from values

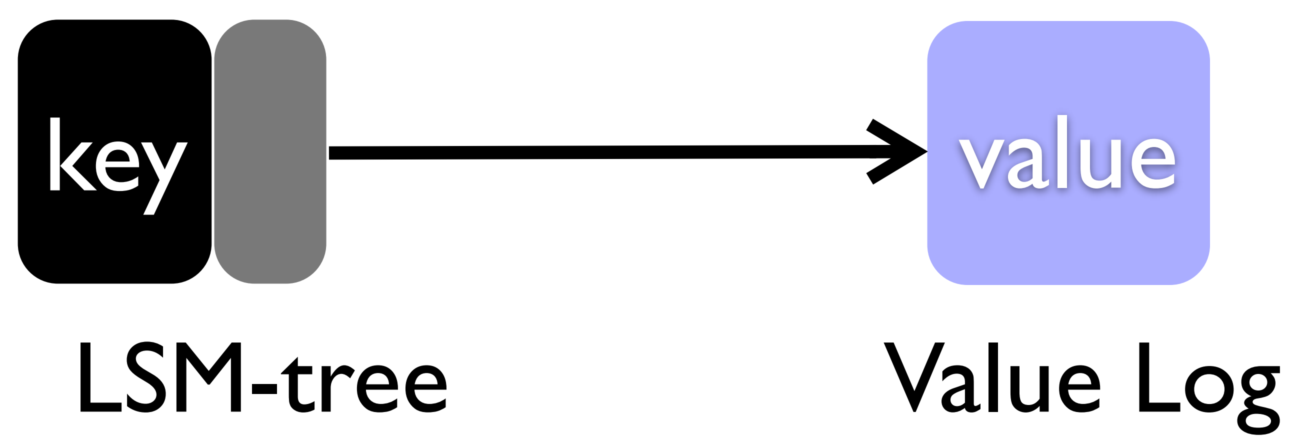
- decouple sorting and garbage collection
- harness SSD's internal parallelism for range queries



Our Solution: WiscKey

Separate keys from values

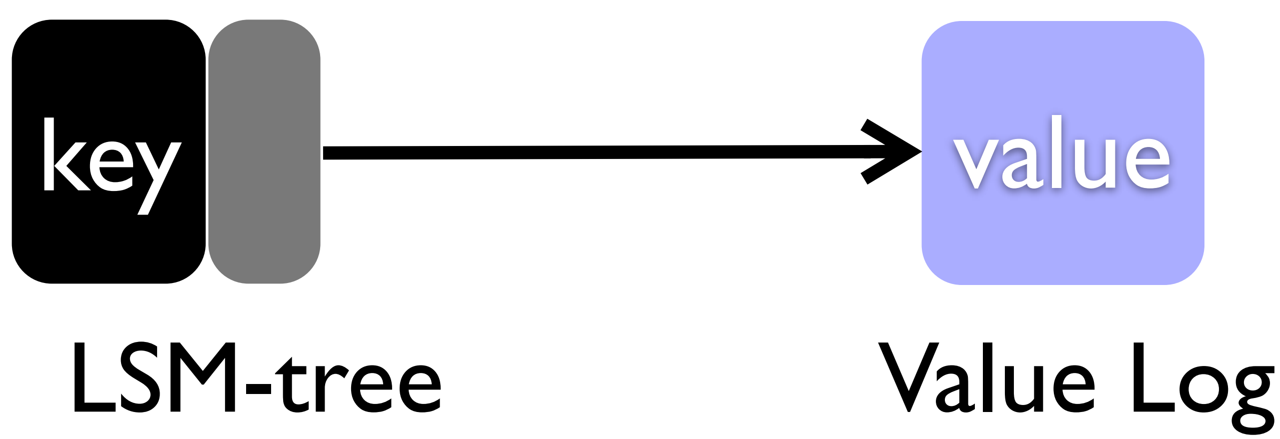
- decouple sorting and garbage collection
- harness SSD's internal parallelism for range queries
- online and light-weight garbage collection



Our Solution: WiscKey

Separate keys from values

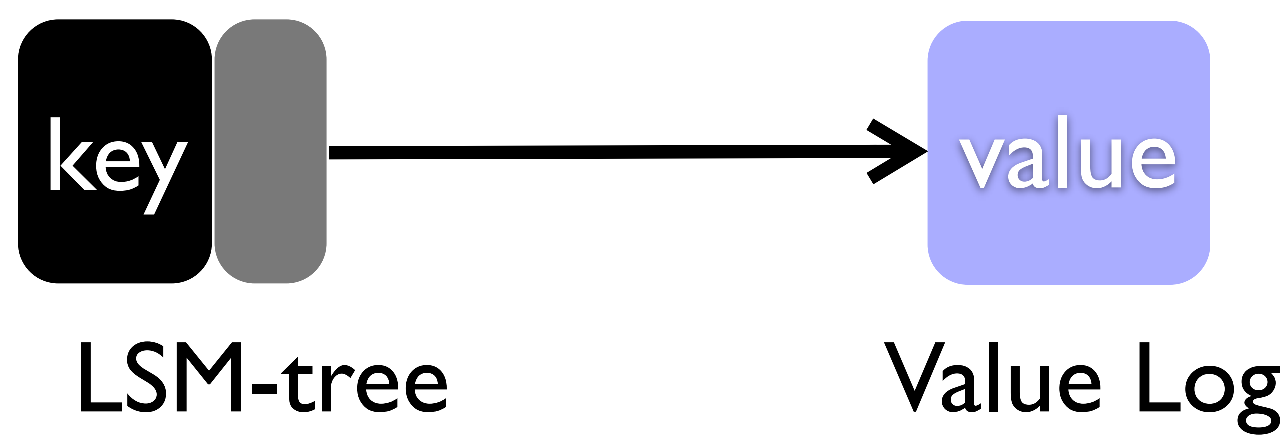
- decouple sorting and garbage collection
- harness SSD's internal parallelism for range queries
- online and light-weight garbage collection
- minimize I/O amplification and crash consistent



Our Solution: WiscKey

Separate keys from values

- decouple sorting and garbage collection
- harness SSD's internal parallelism for range queries
- online and light-weight garbage collection
- minimize I/O amplification and crash consistent



Performance of WiscKey

- 2.5x to 111x for loading, 1.6x to 14x for lookups

Background

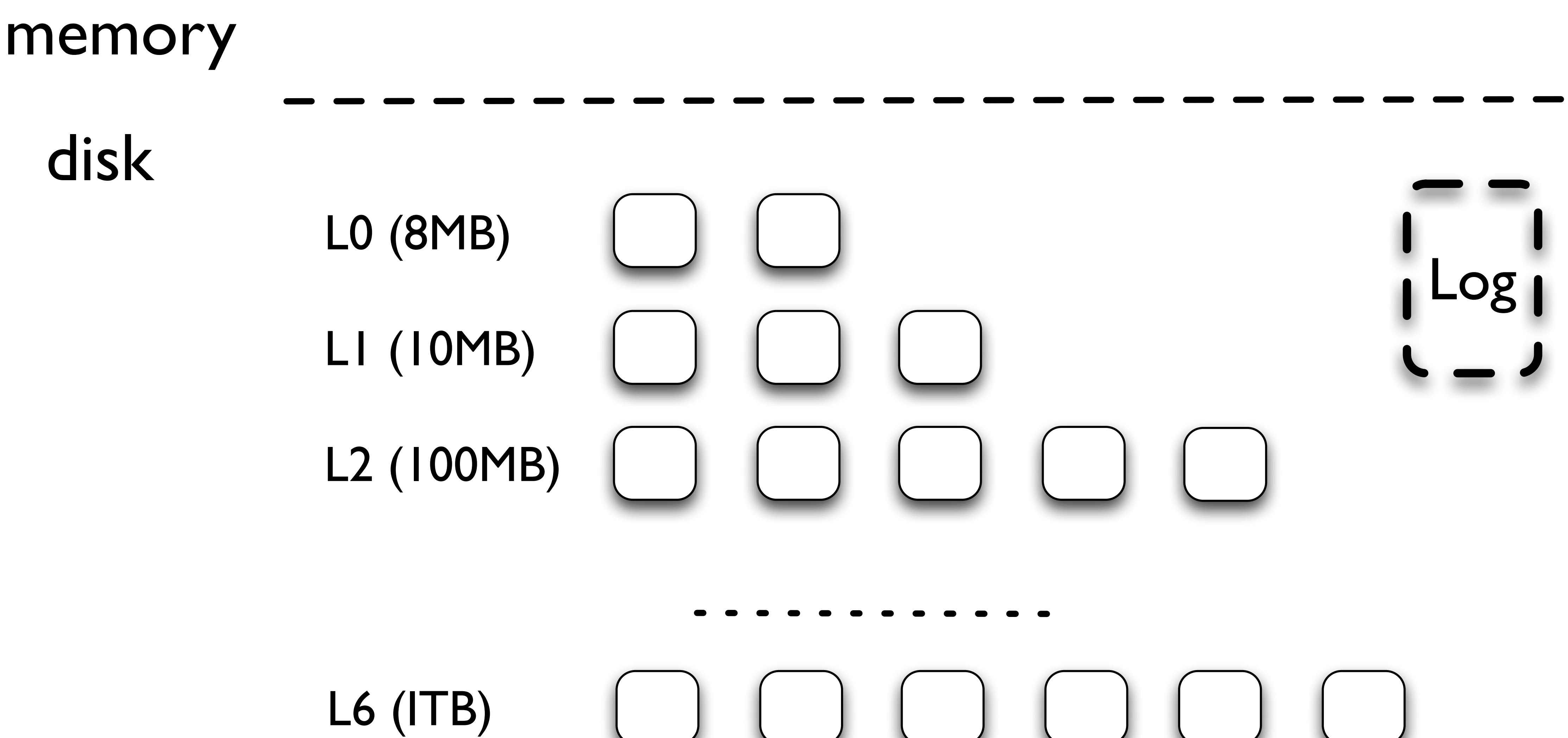
Key-Value Separation

Challenges and Optimizations

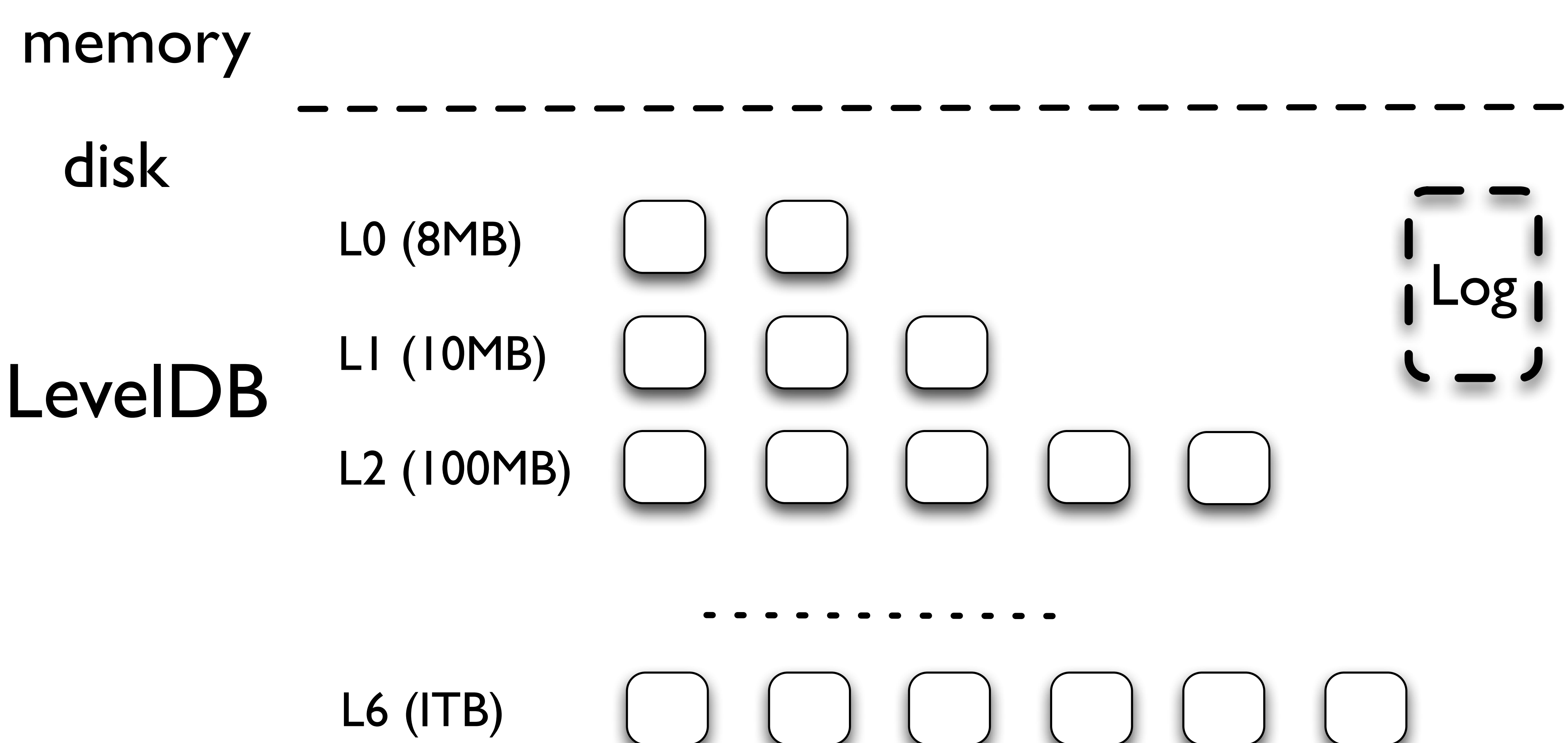
Evaluation

Conclusion

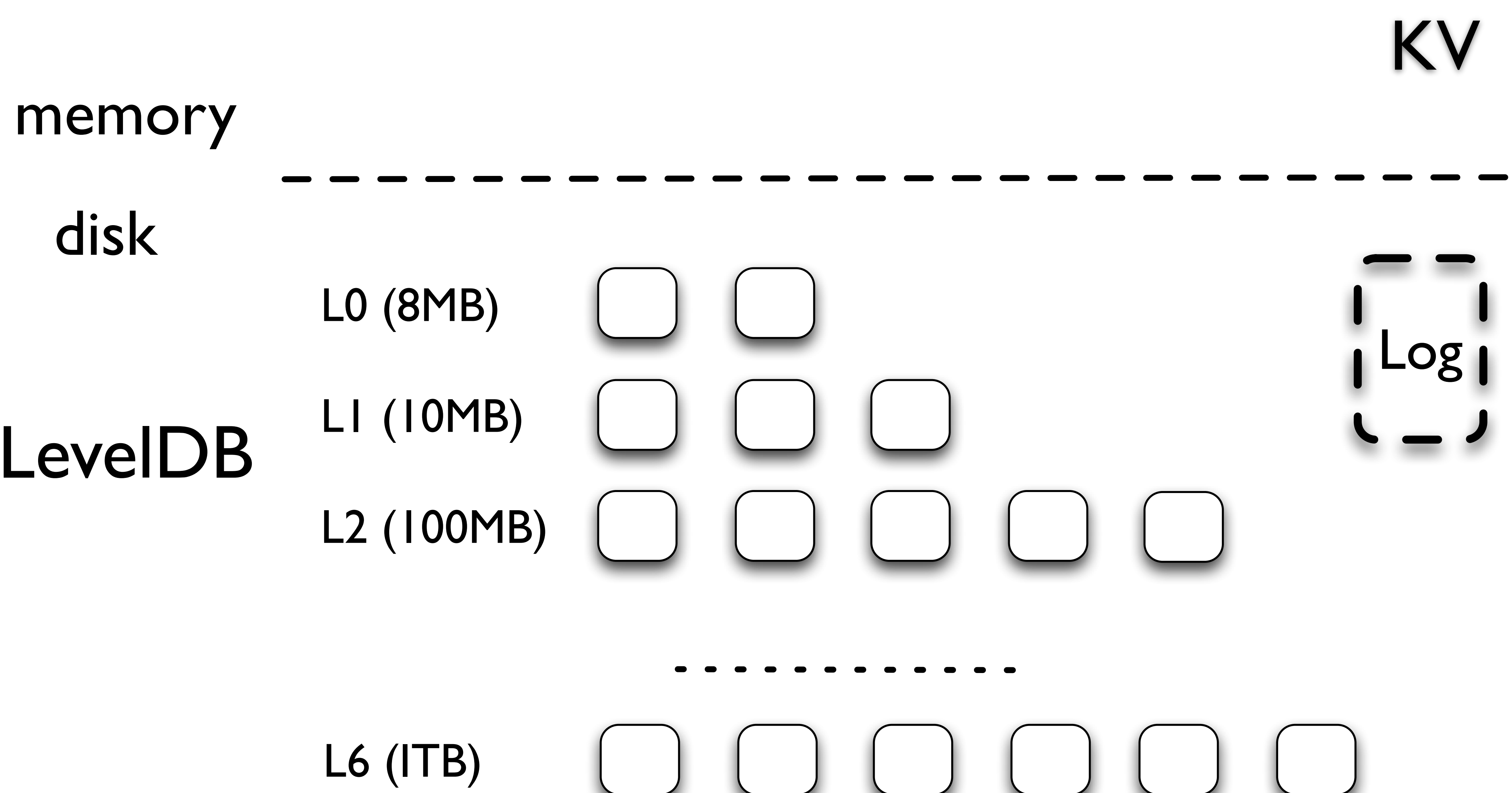
LSM-trees: Insertion



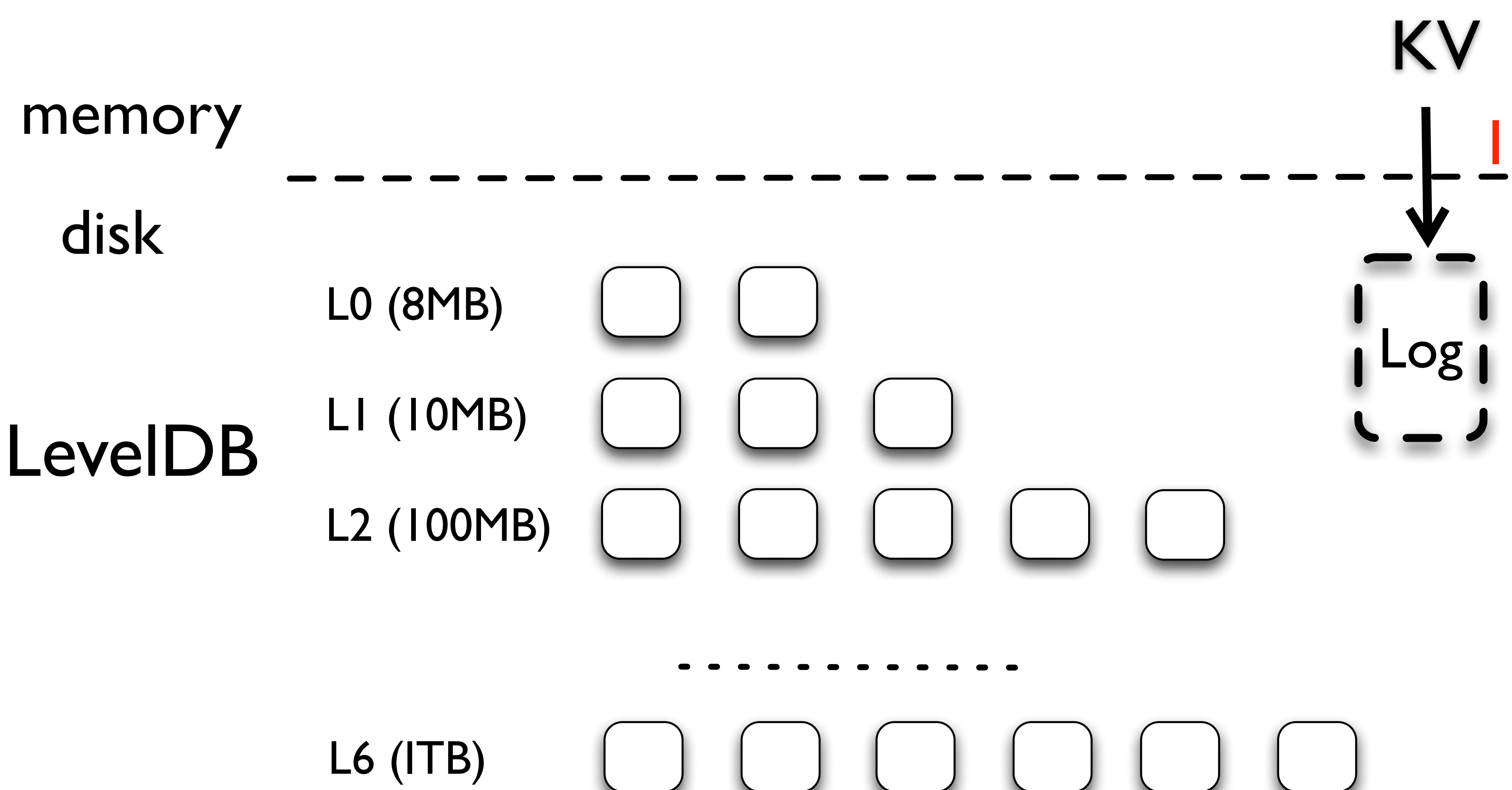
LSM-trees: Insertion



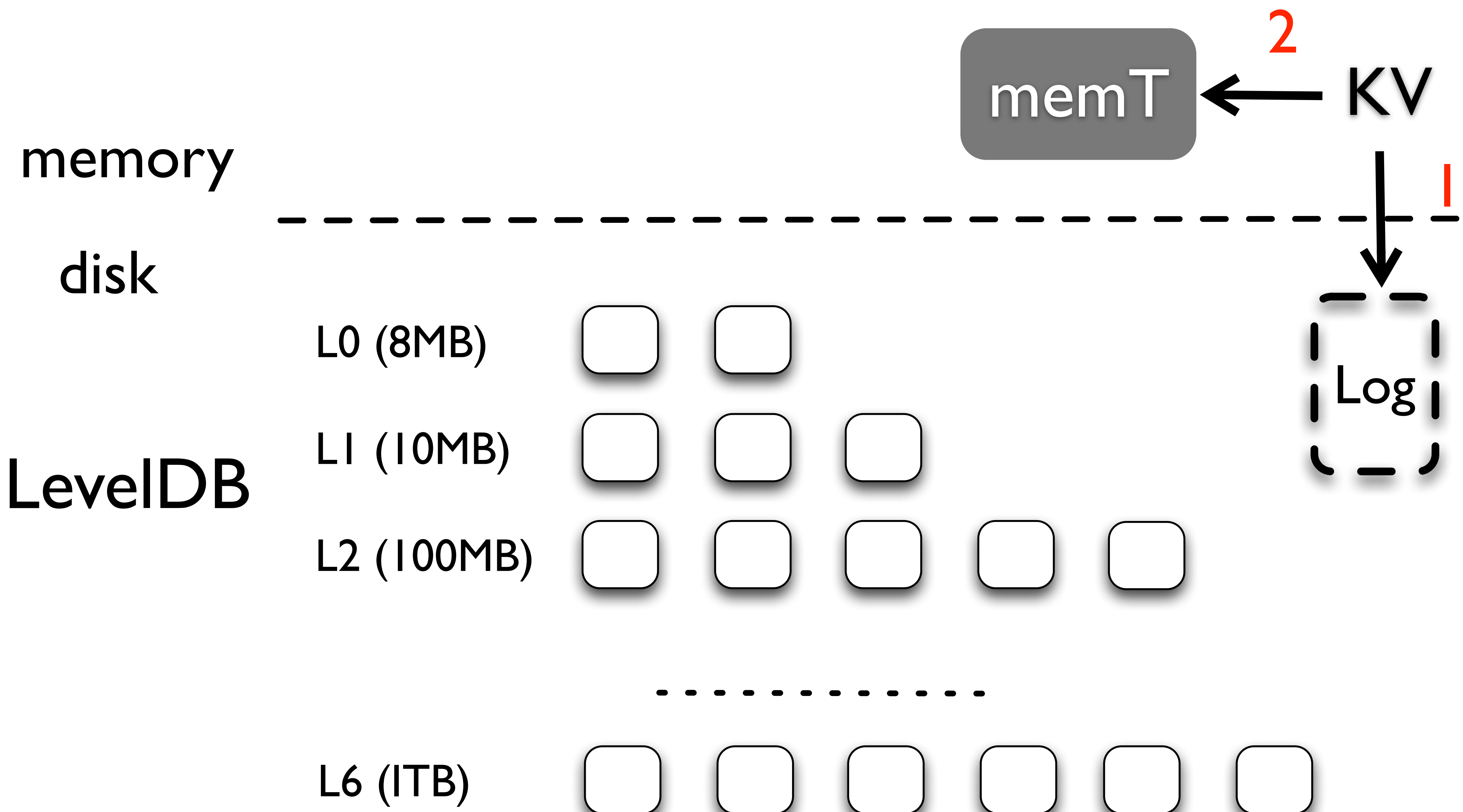
LSM-trees: Insertion



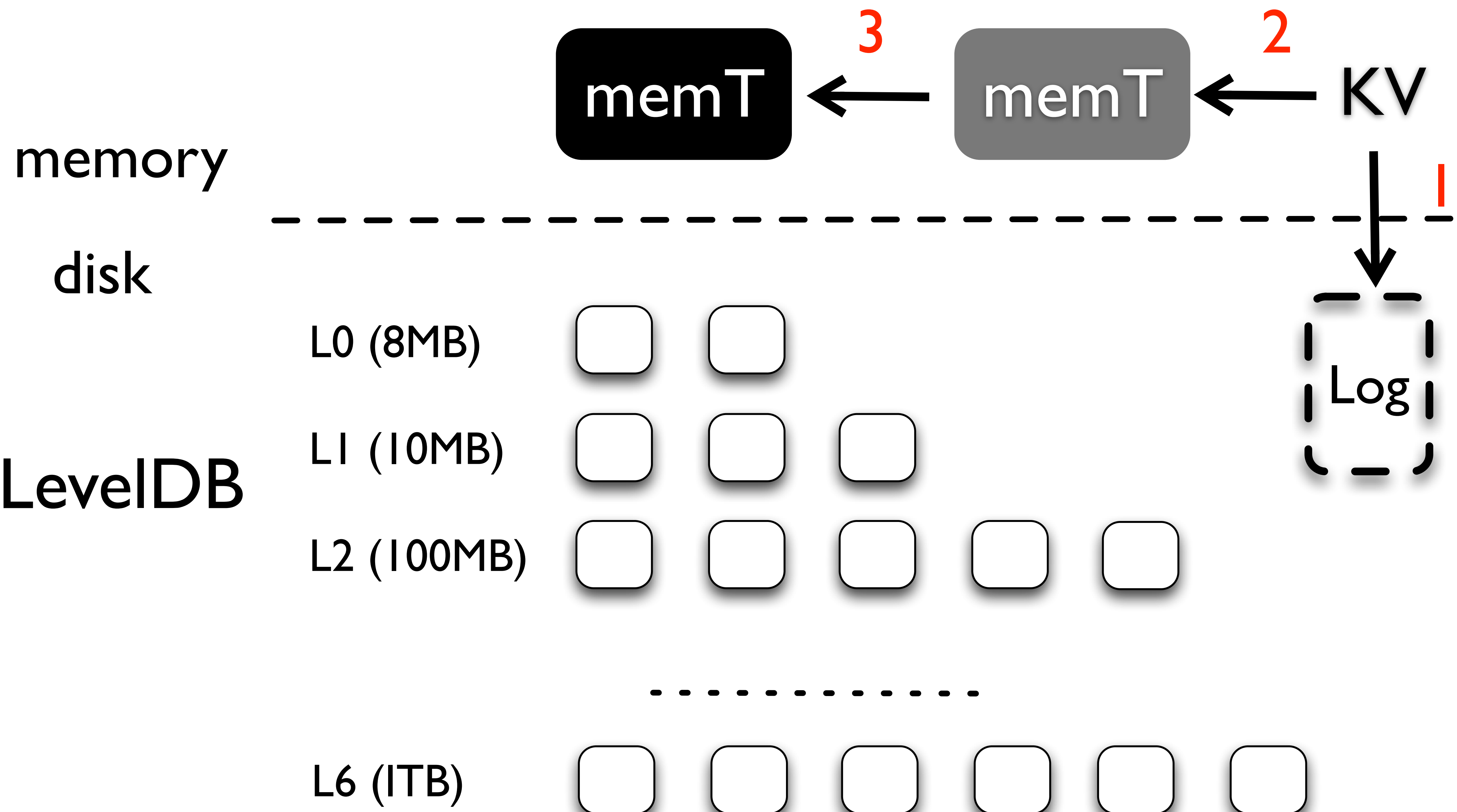
LSM-trees: Insertion



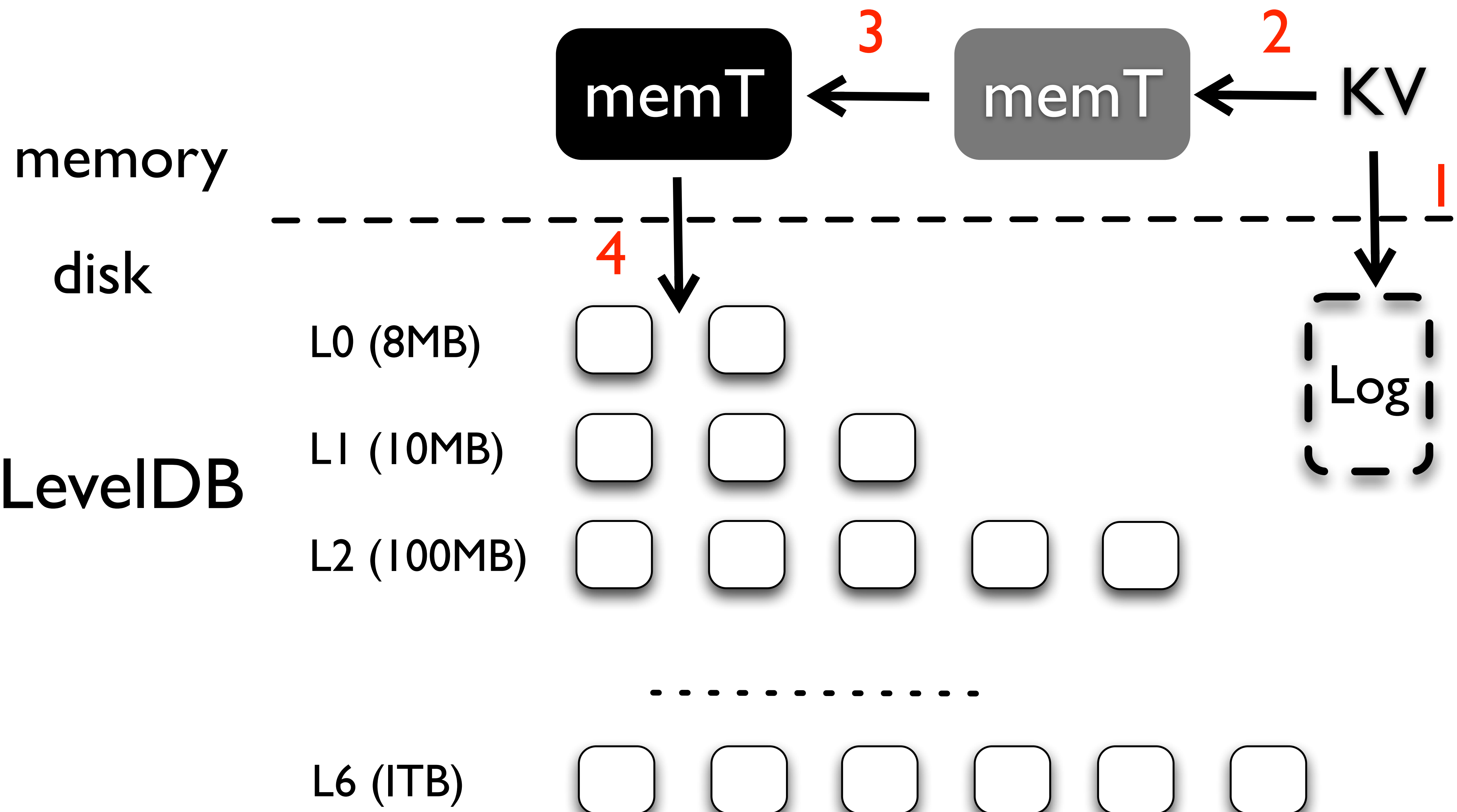
LSM-trees: Insertion



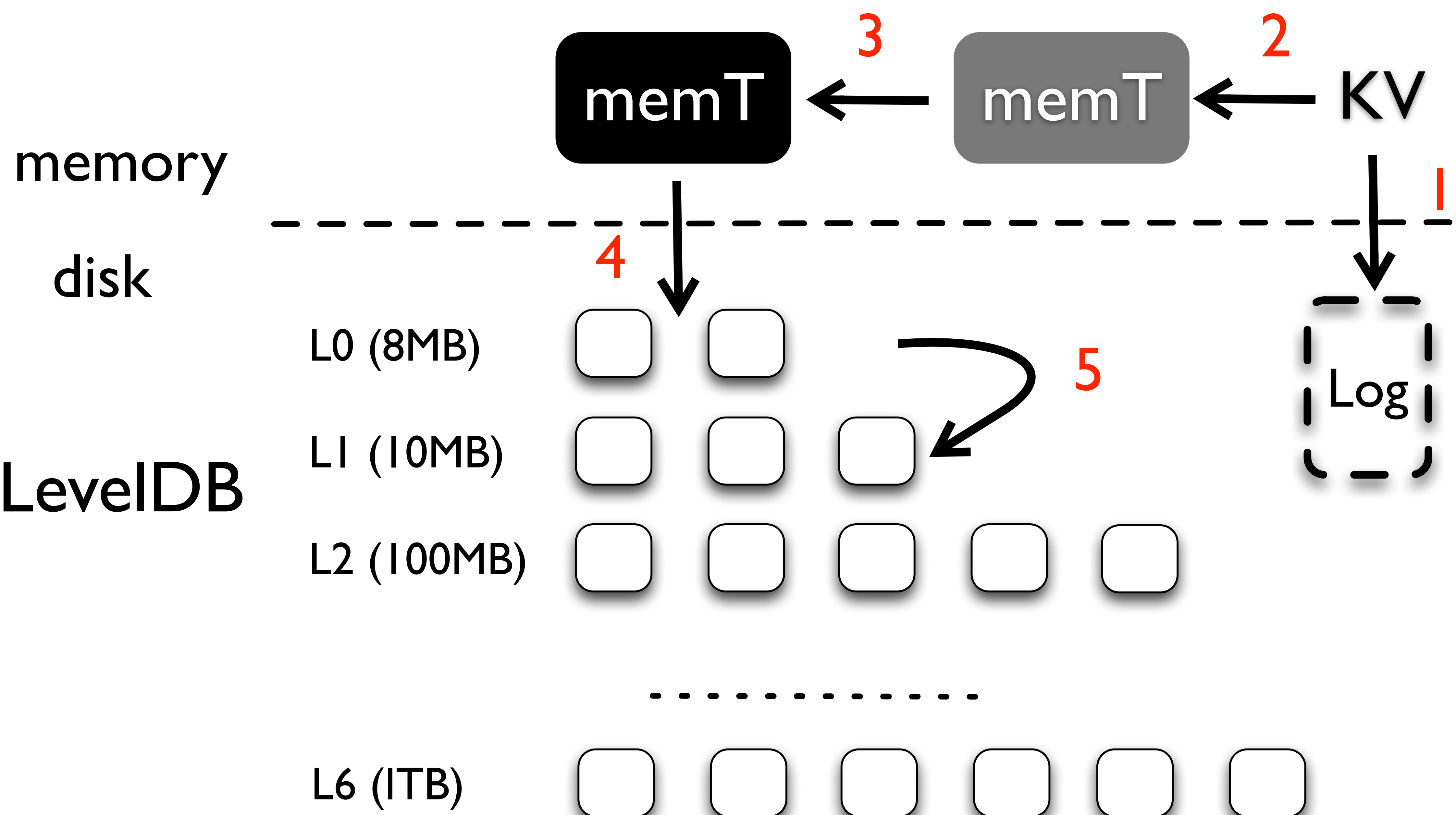
LSM-trees: Insertion



LSM-trees: Insertion

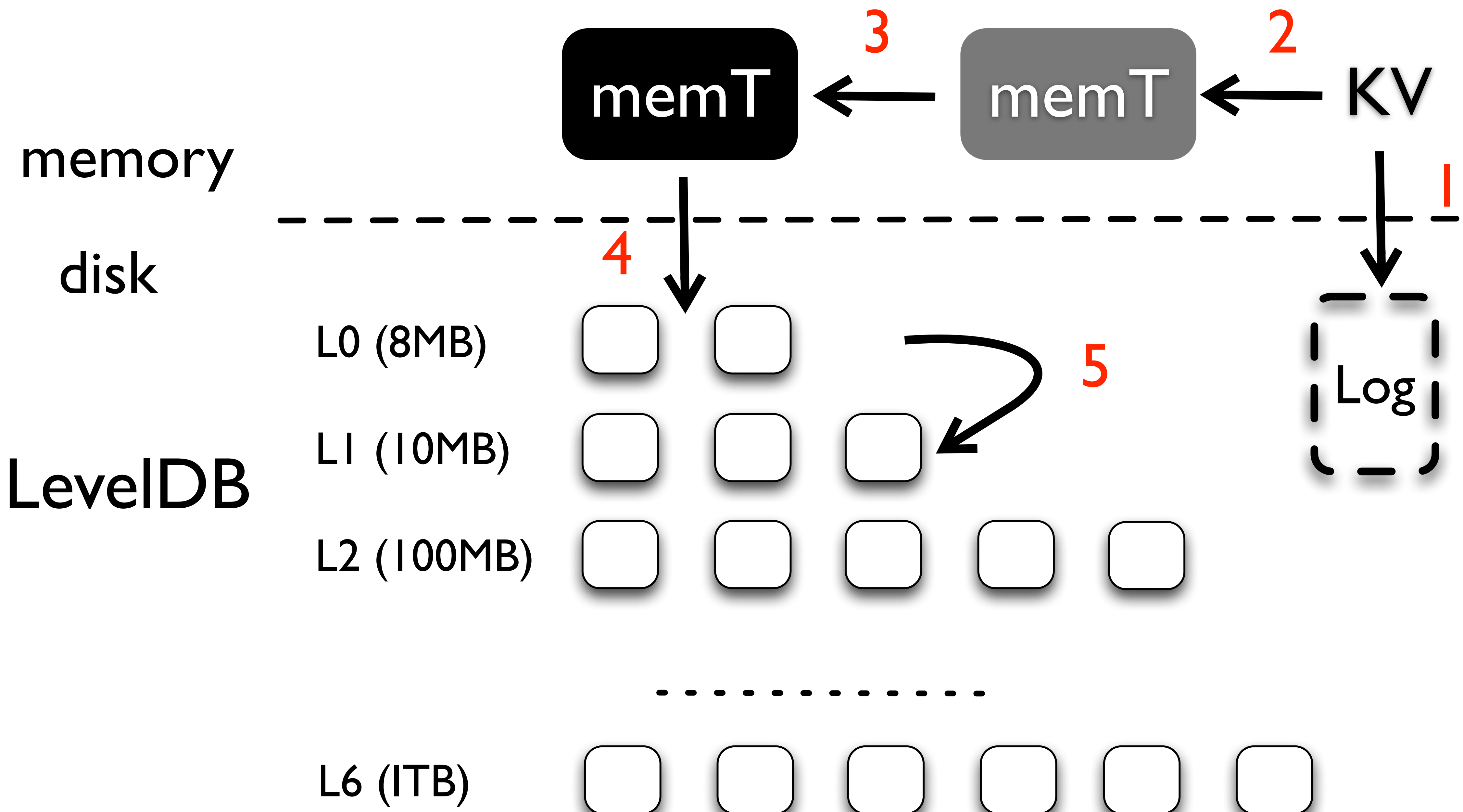


LSM-trees: Insertion



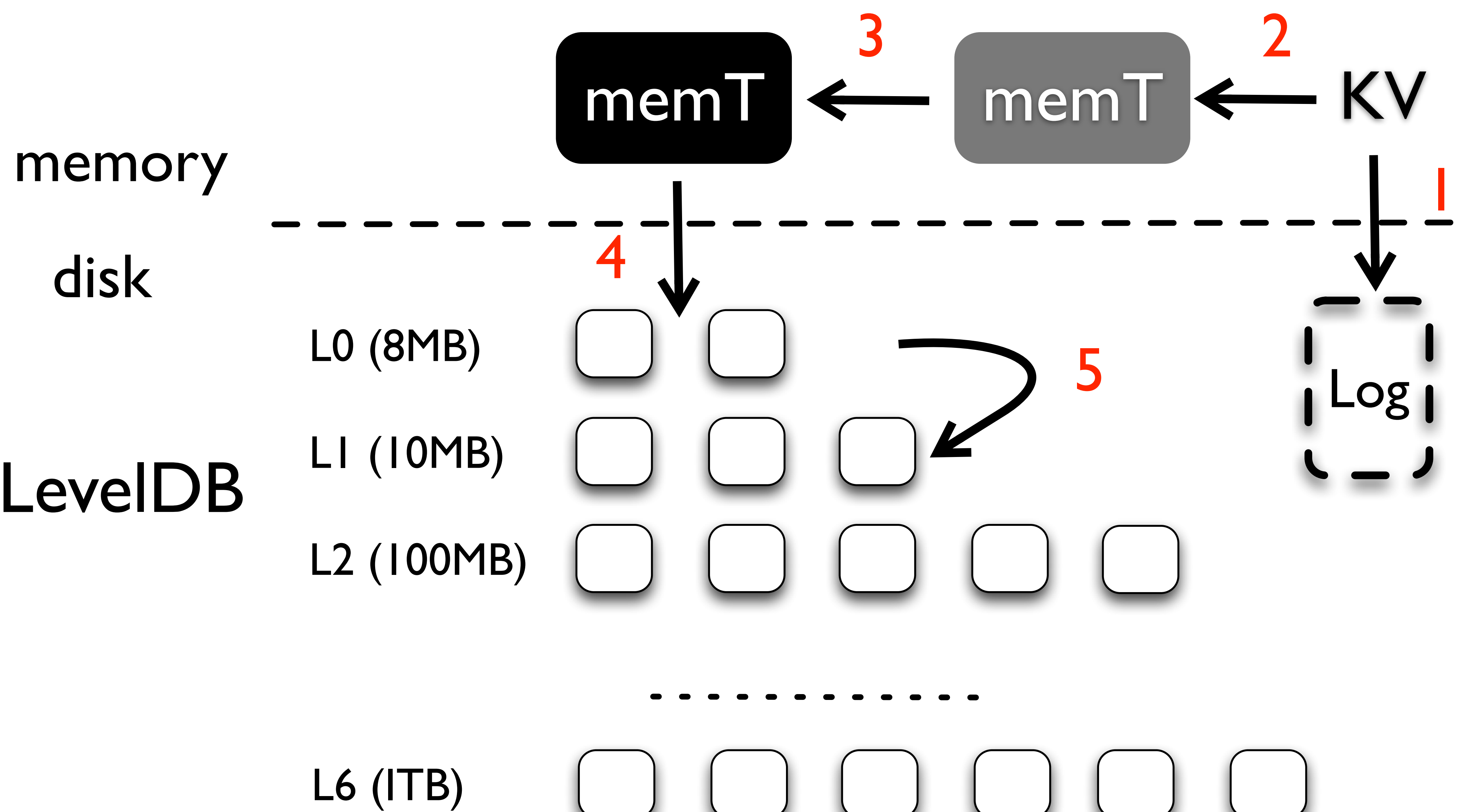
LSM-trees: Insertion

1. Write sequentially 2. Sort data for quick lookups

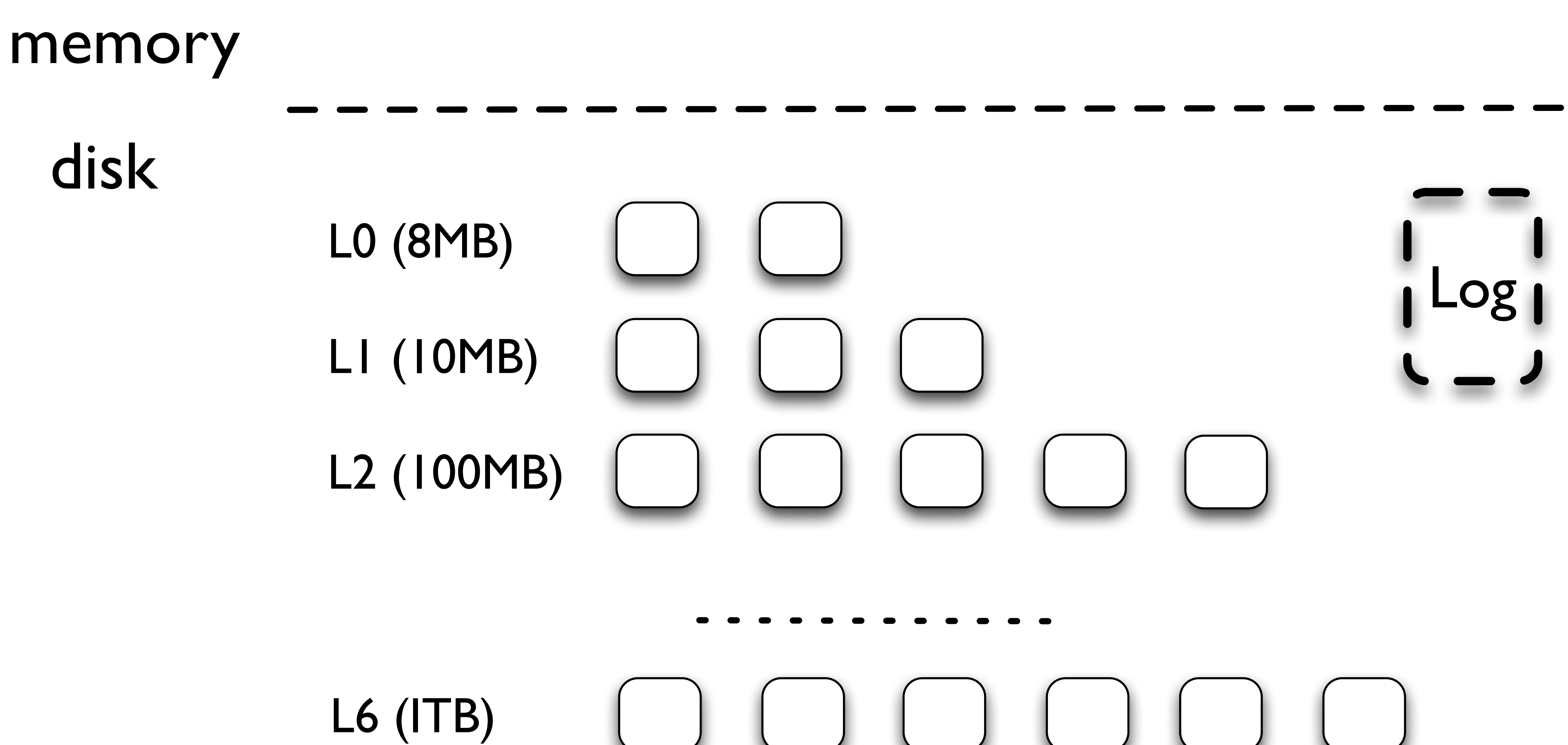


LSM-trees: Insertion

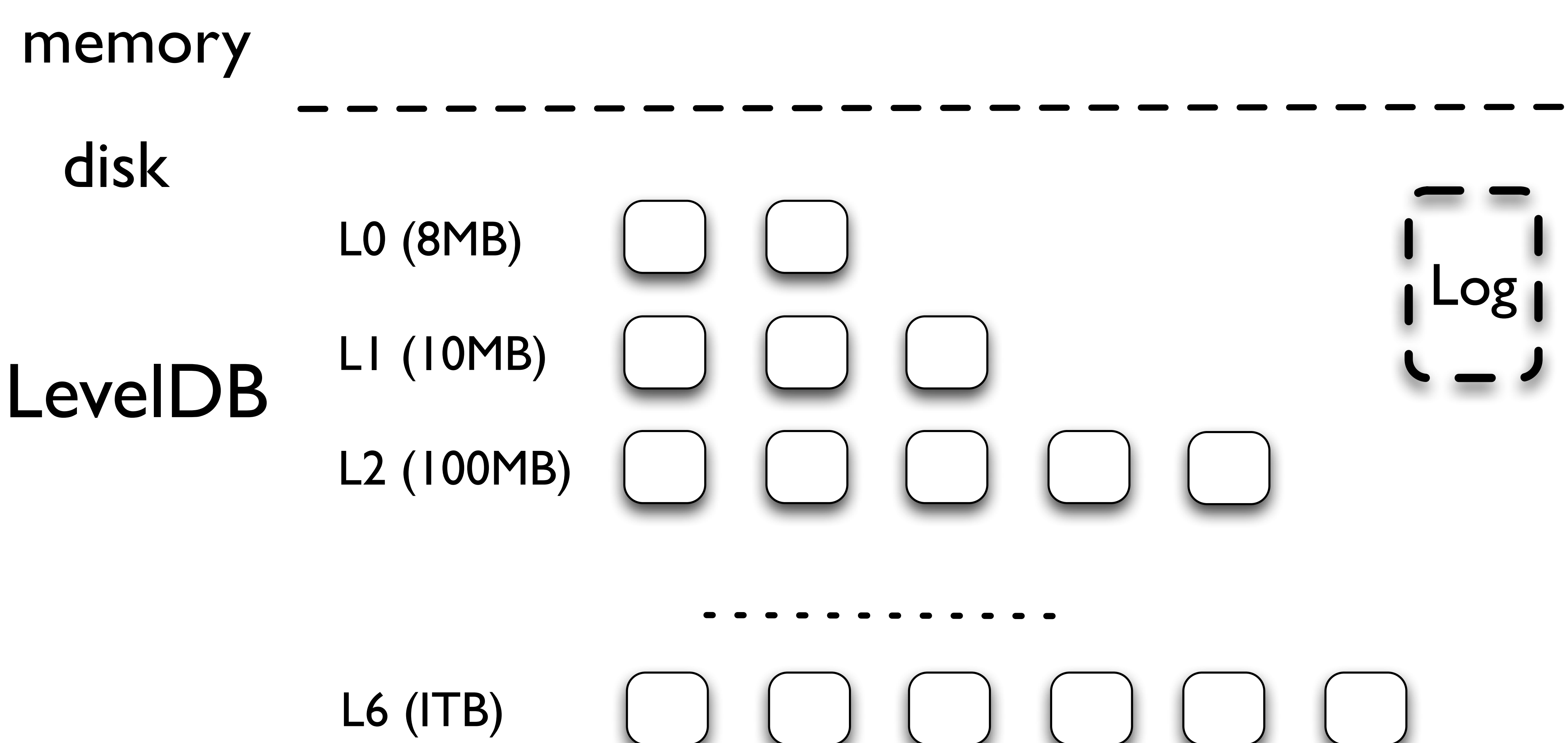
1. Write sequentially 2. Sort data for quick lookups
3. Sorting and garbage collection are coupled



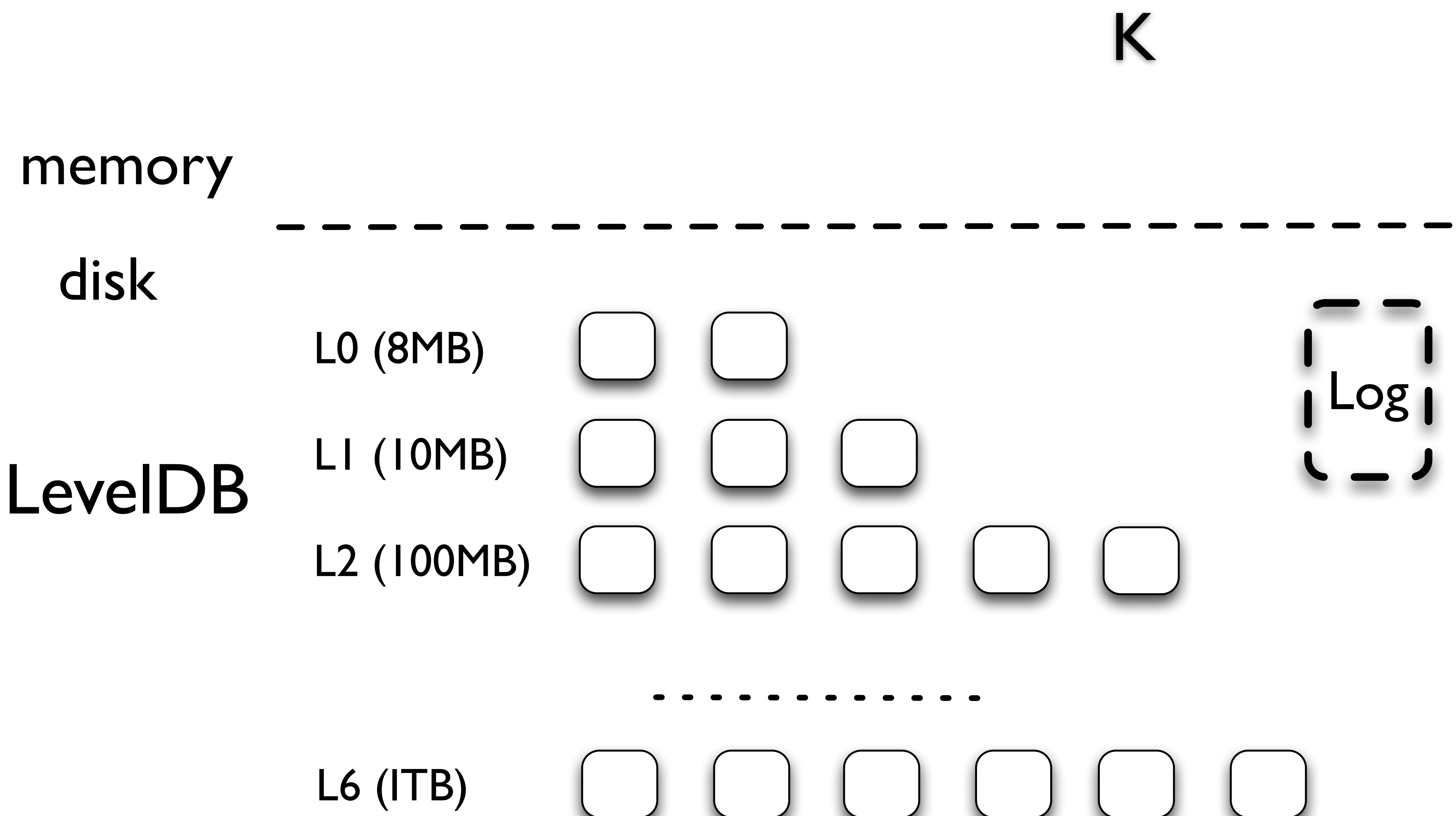
LSM-trees: Lookup



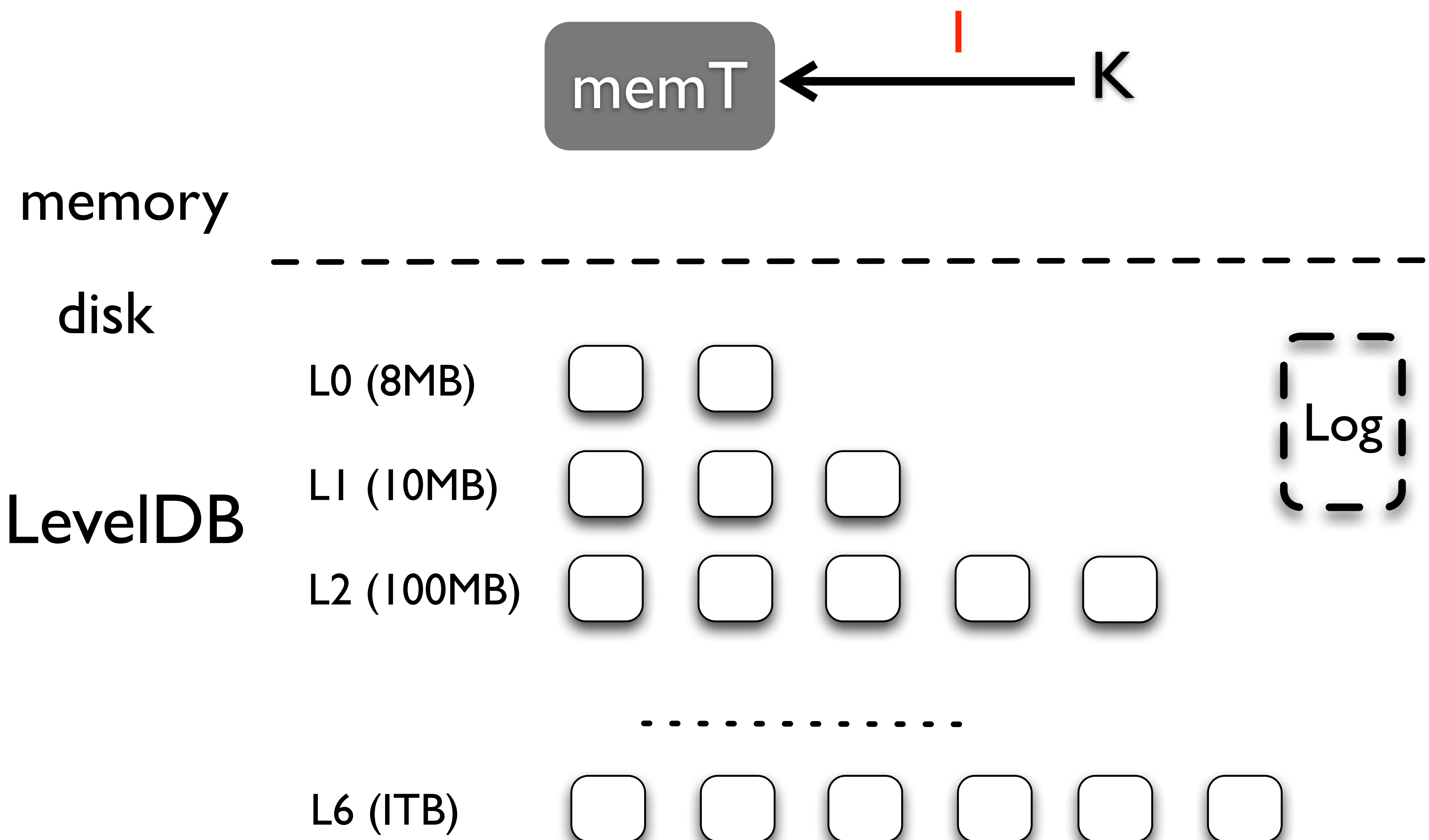
LSM-trees: Lookup



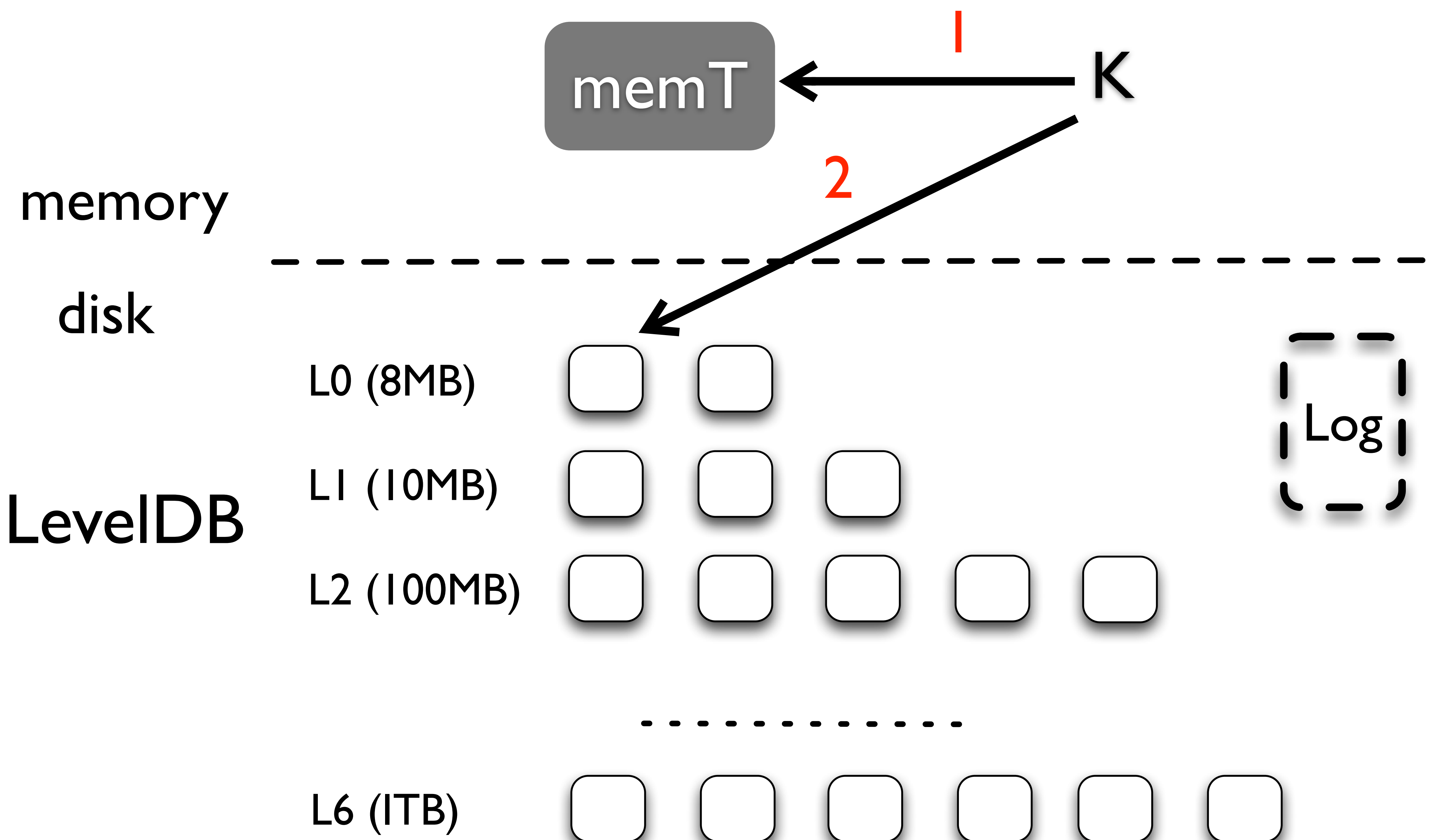
LSM-trees: Lookup



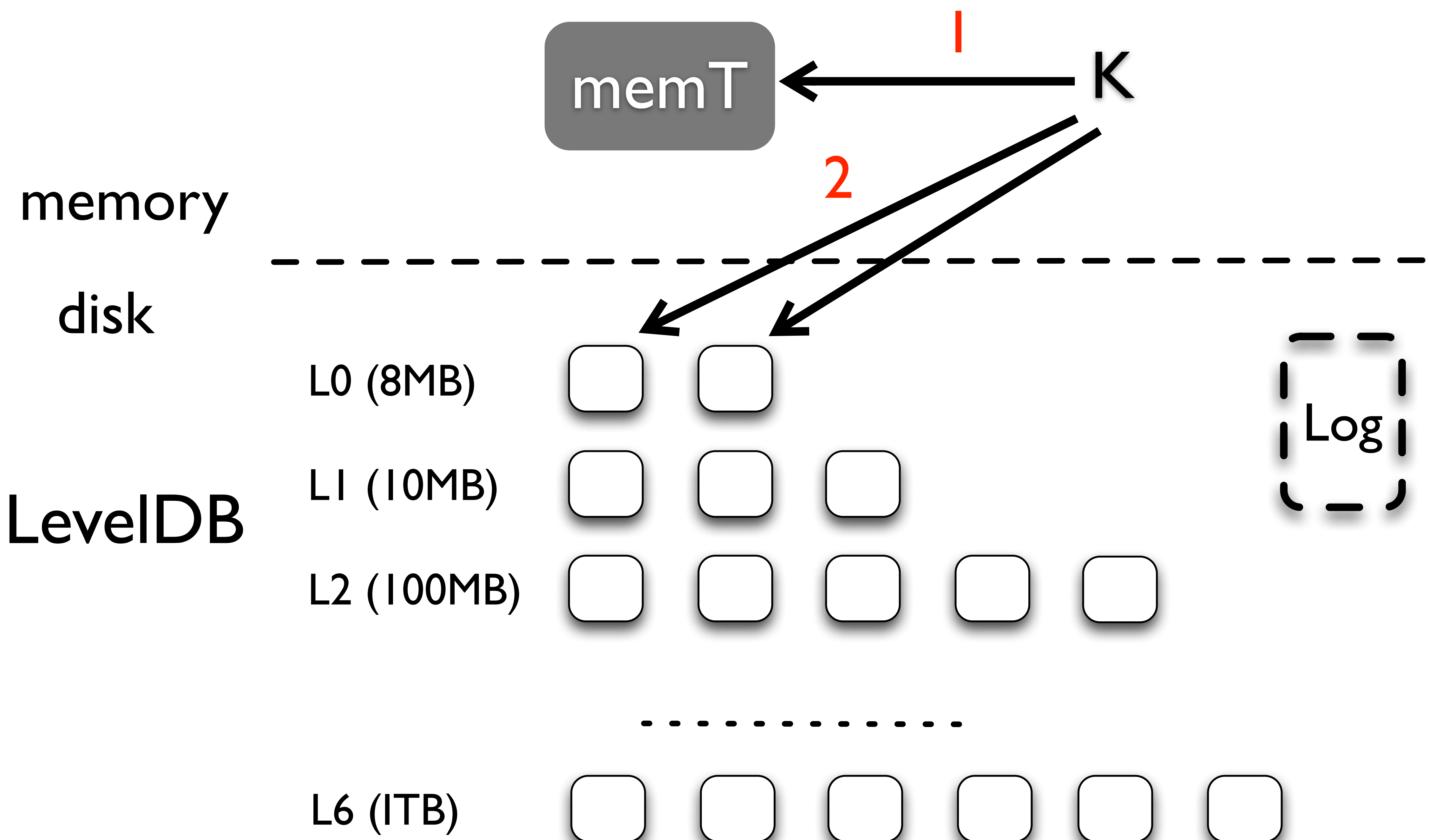
LSM-trees: Lookup



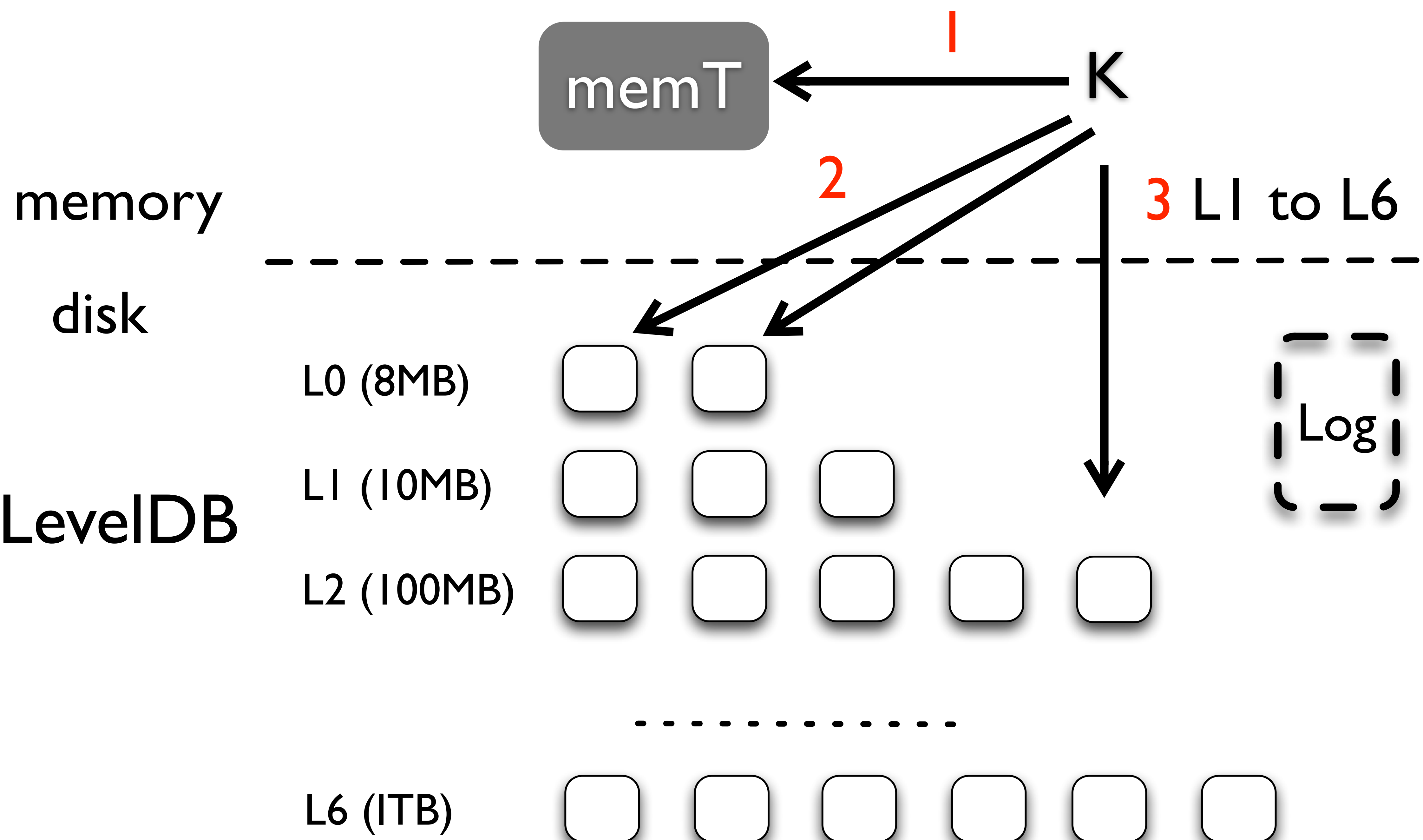
LSM-trees: Lookup



LSM-trees: Lookup

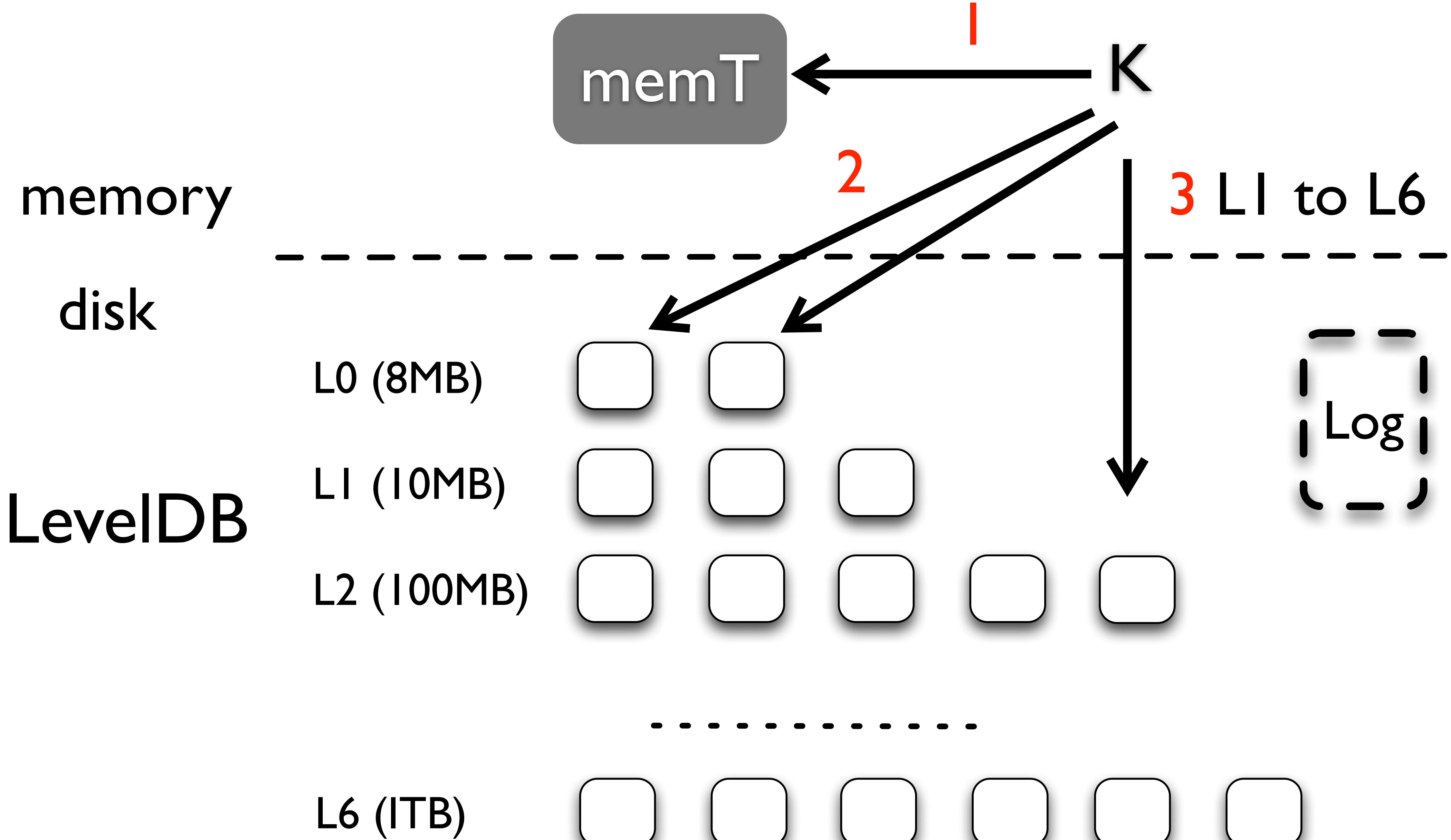


LSM-trees: Lookup



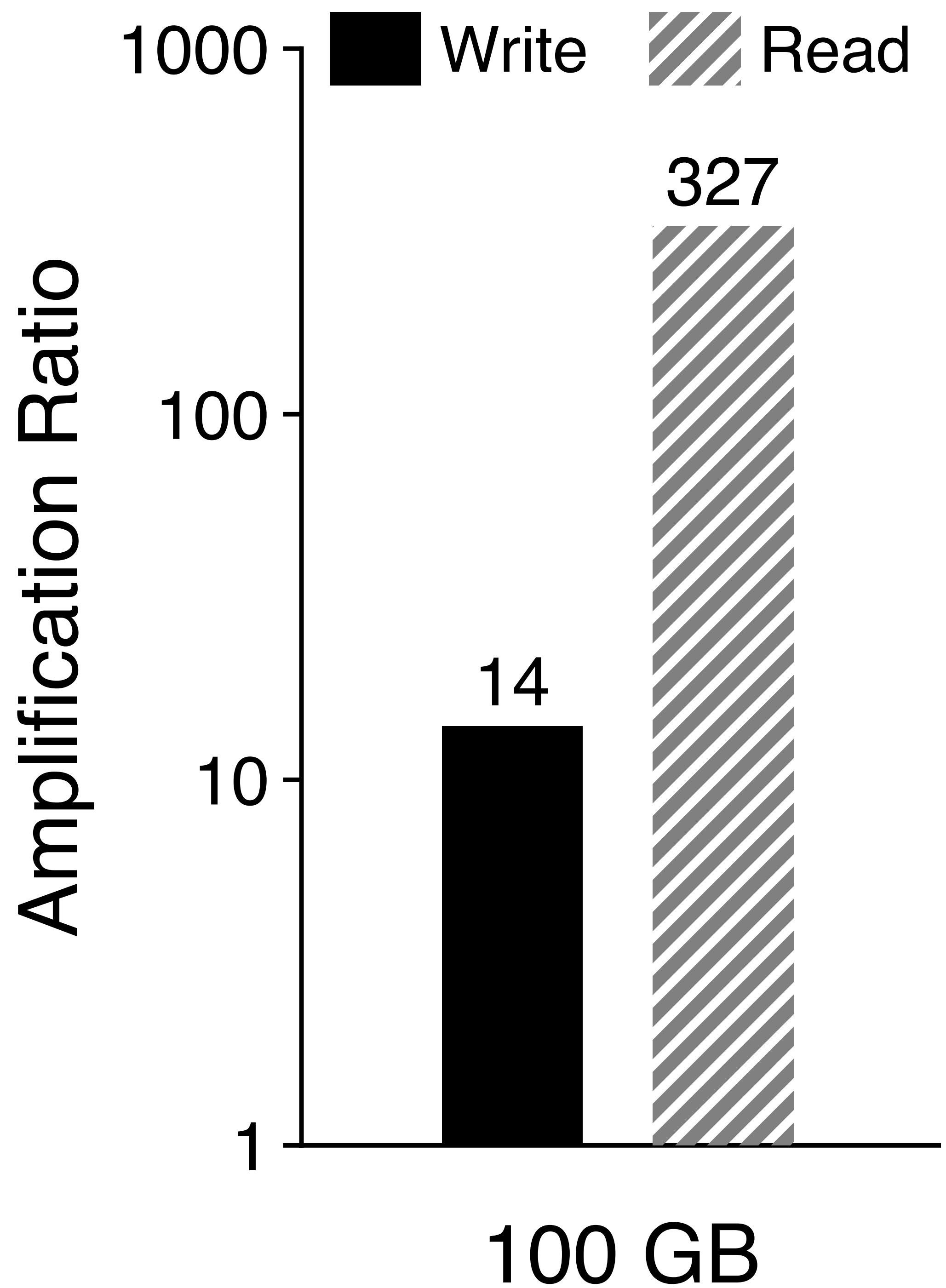
LSM-trees: Lookup

1. Random reads
2. Travel many levels for a large LSM-tree

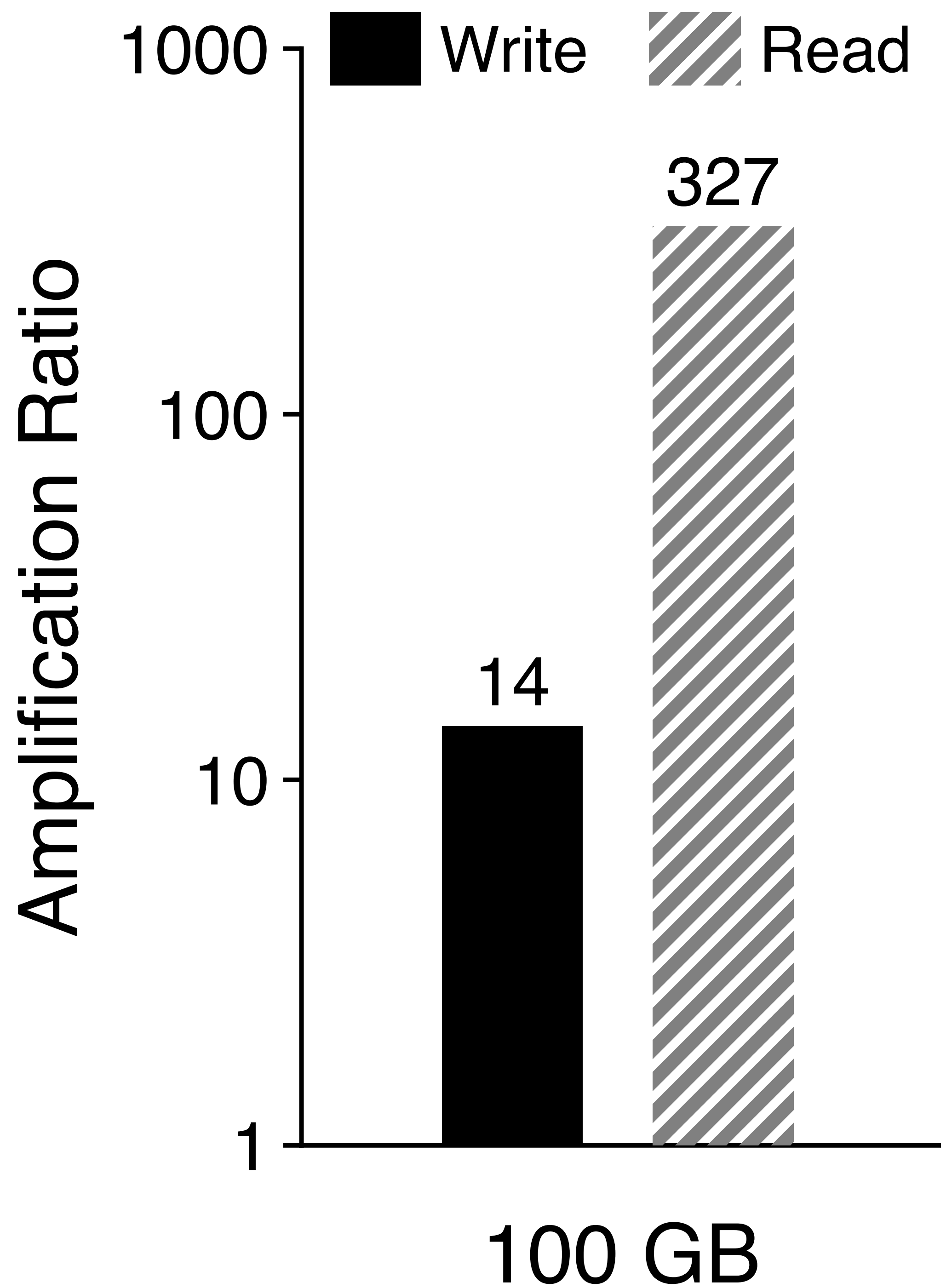


I/O Amplification in LSM-trees

I/O Amplification in LSM-trees



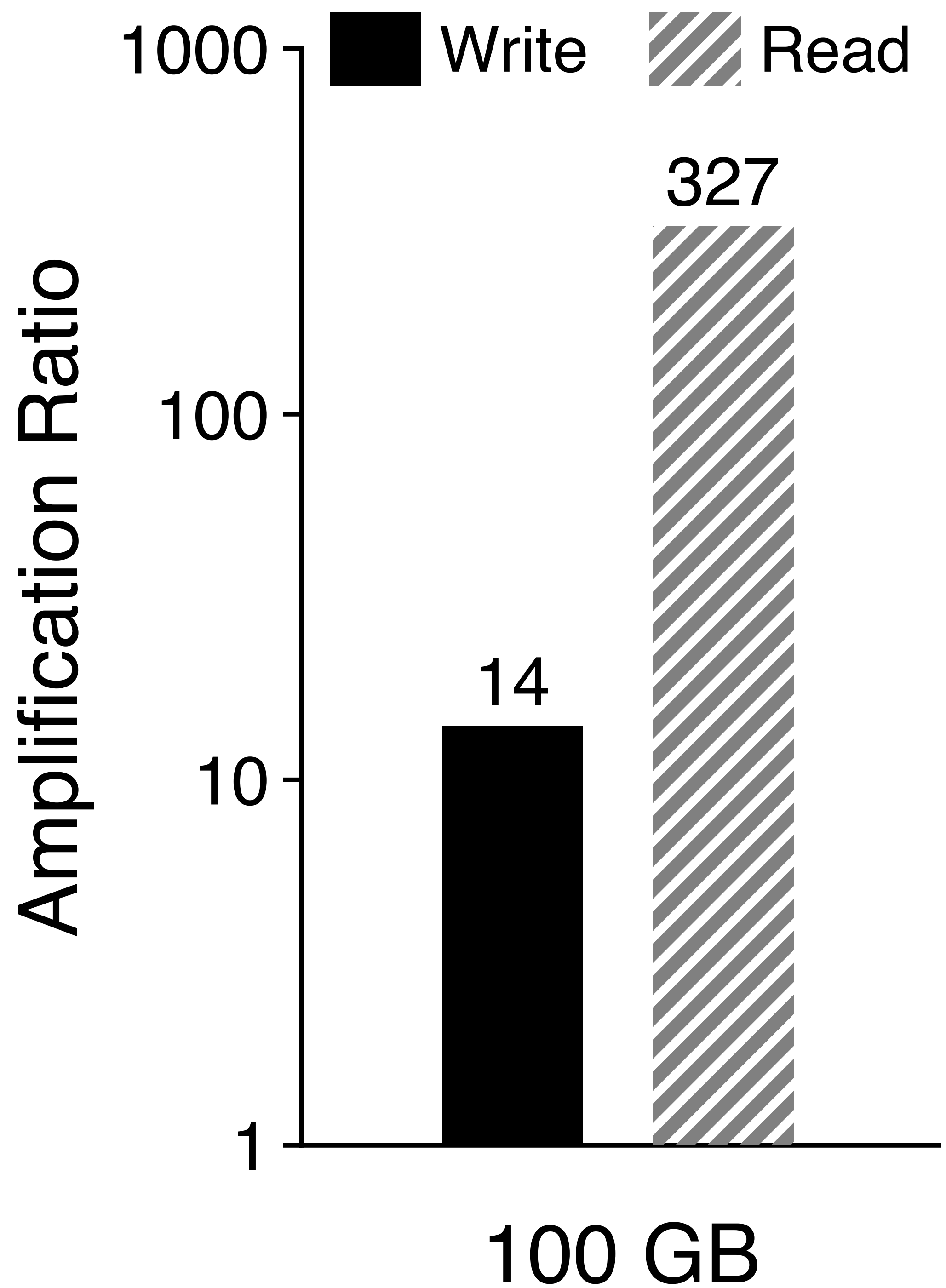
I/O Amplification in LSM-trees



Random load:
a 100GB database

Random lookup:
100,000 lookups

I/O Amplification in LSM-trees



Random load:
a 100GB database

Random lookup:
100,000 lookups

Problems:

large write amplification

large read amplification

Background

Key-Value Separation

Challenges and Optimizations

Evaluation

Conclusion

Key-Value Separation

Key-Value Separation

Main idea: only keys are required to be sorted

Key-Value Separation

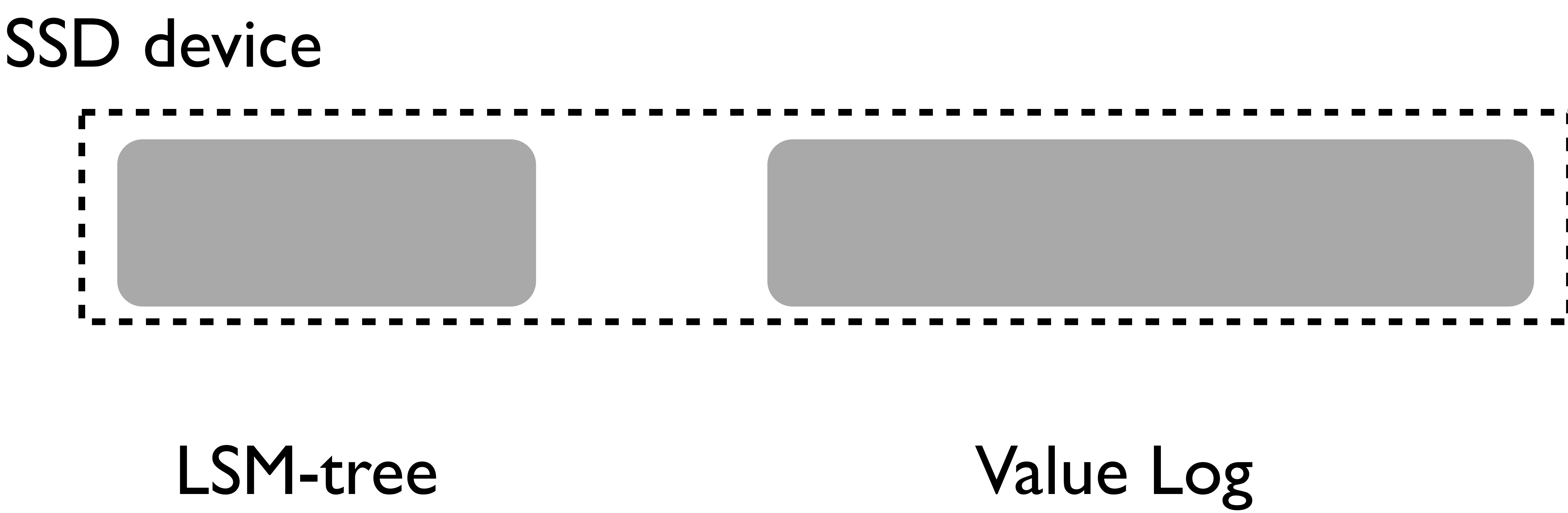
Main idea: only keys are required to be sorted

Decouple sorting and garbage collection

Key-Value Separation

Main idea: only keys are required to be sorted

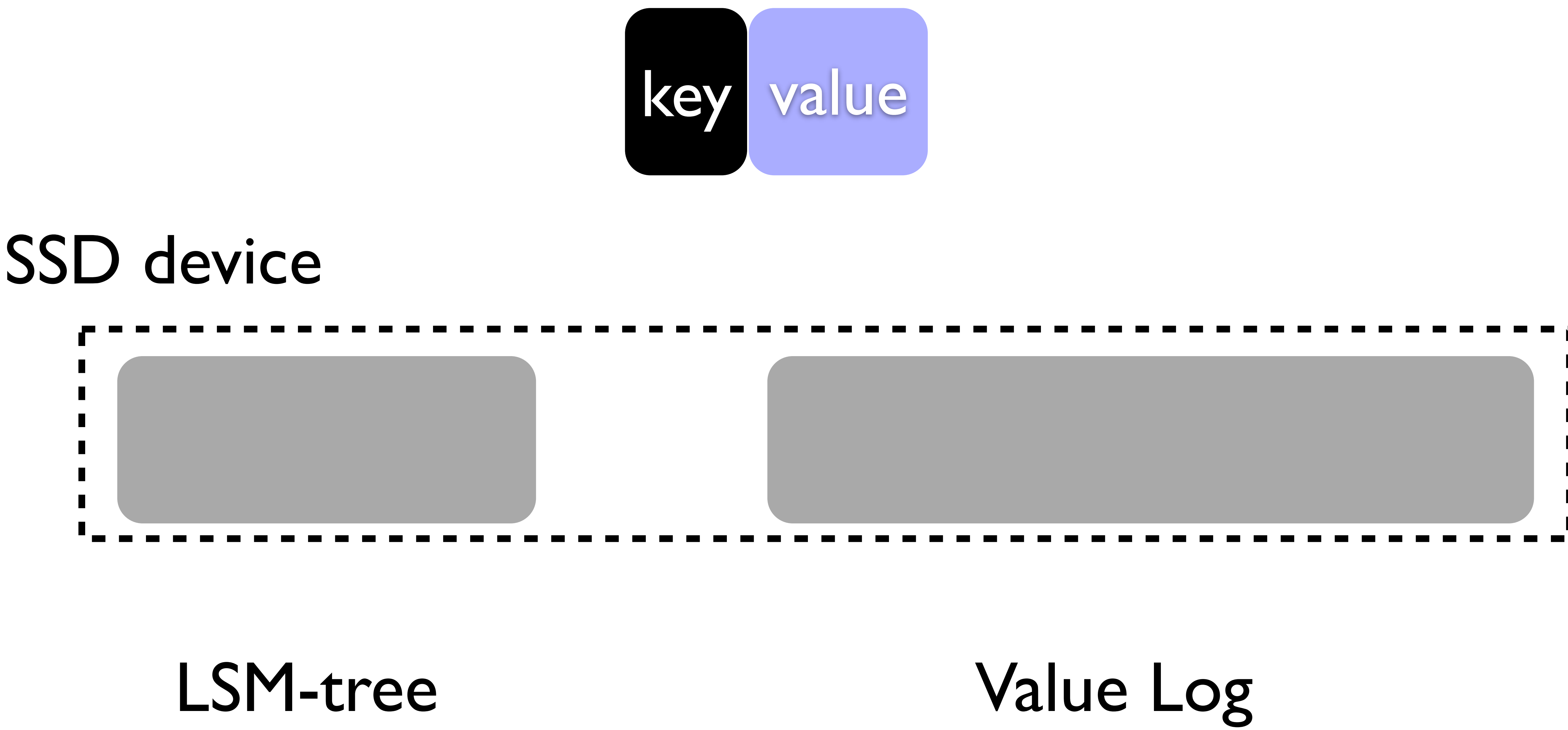
Decouple sorting and garbage collection



Key-Value Separation

Main idea: only keys are required to be sorted

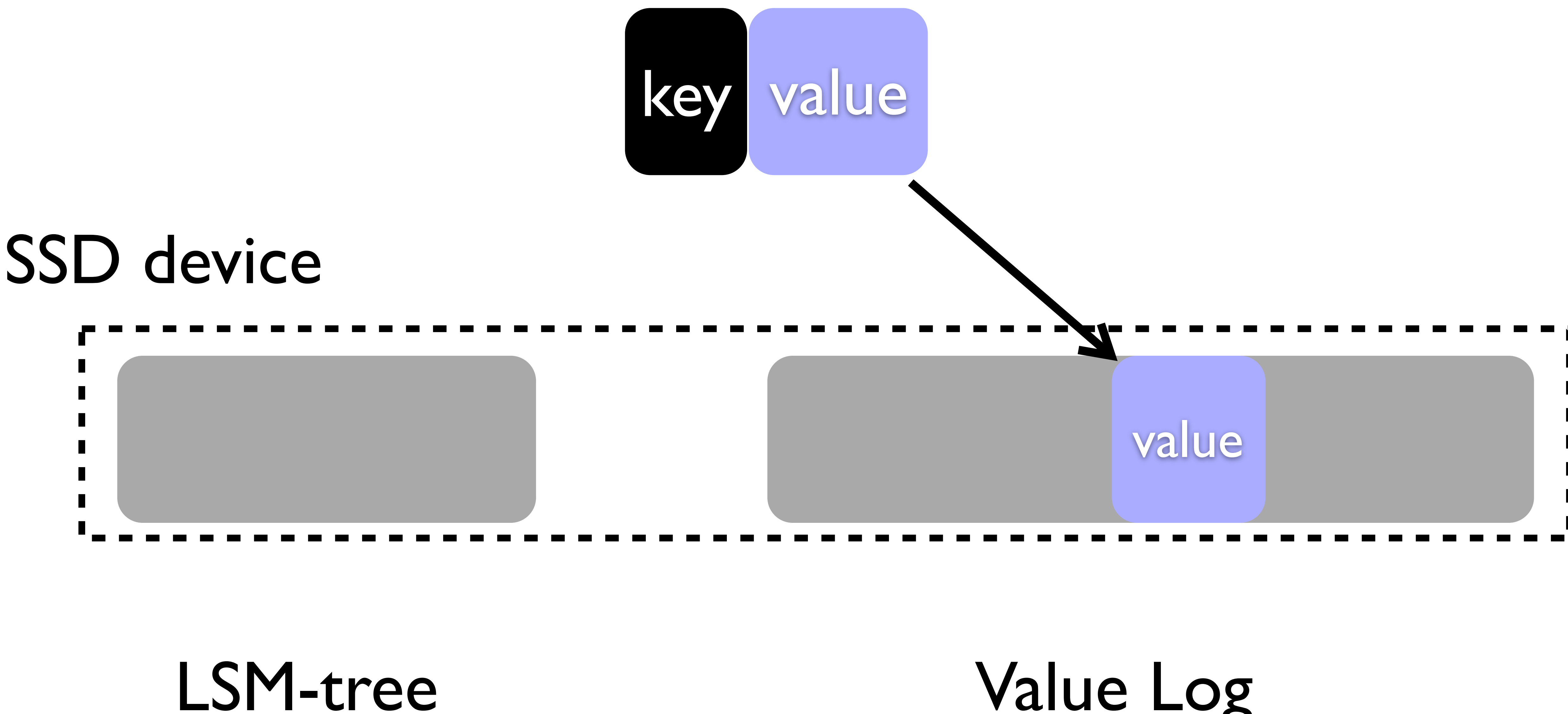
Decouple sorting and garbage collection



Key-Value Separation

Main idea: only keys are required to be sorted

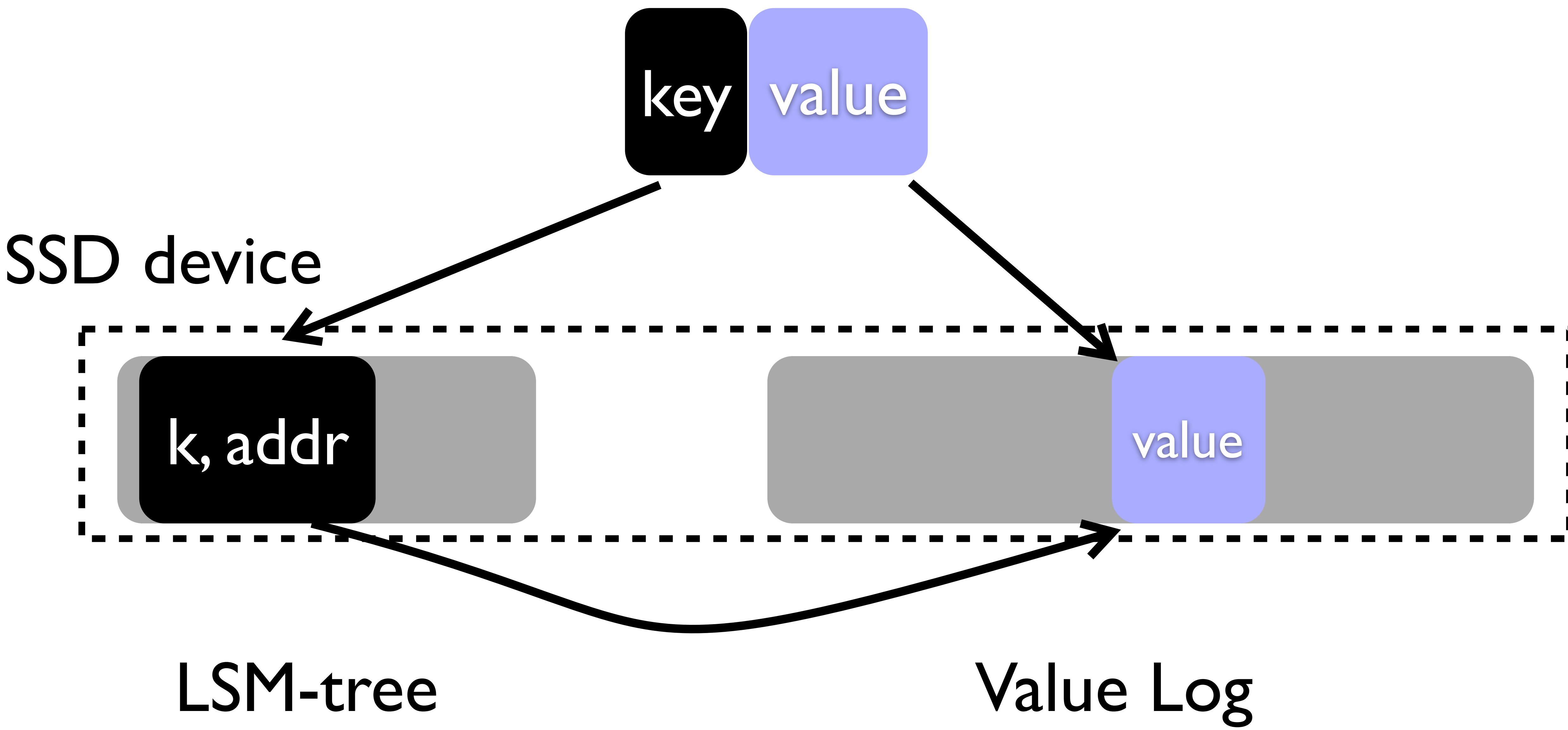
Decouple sorting and garbage collection



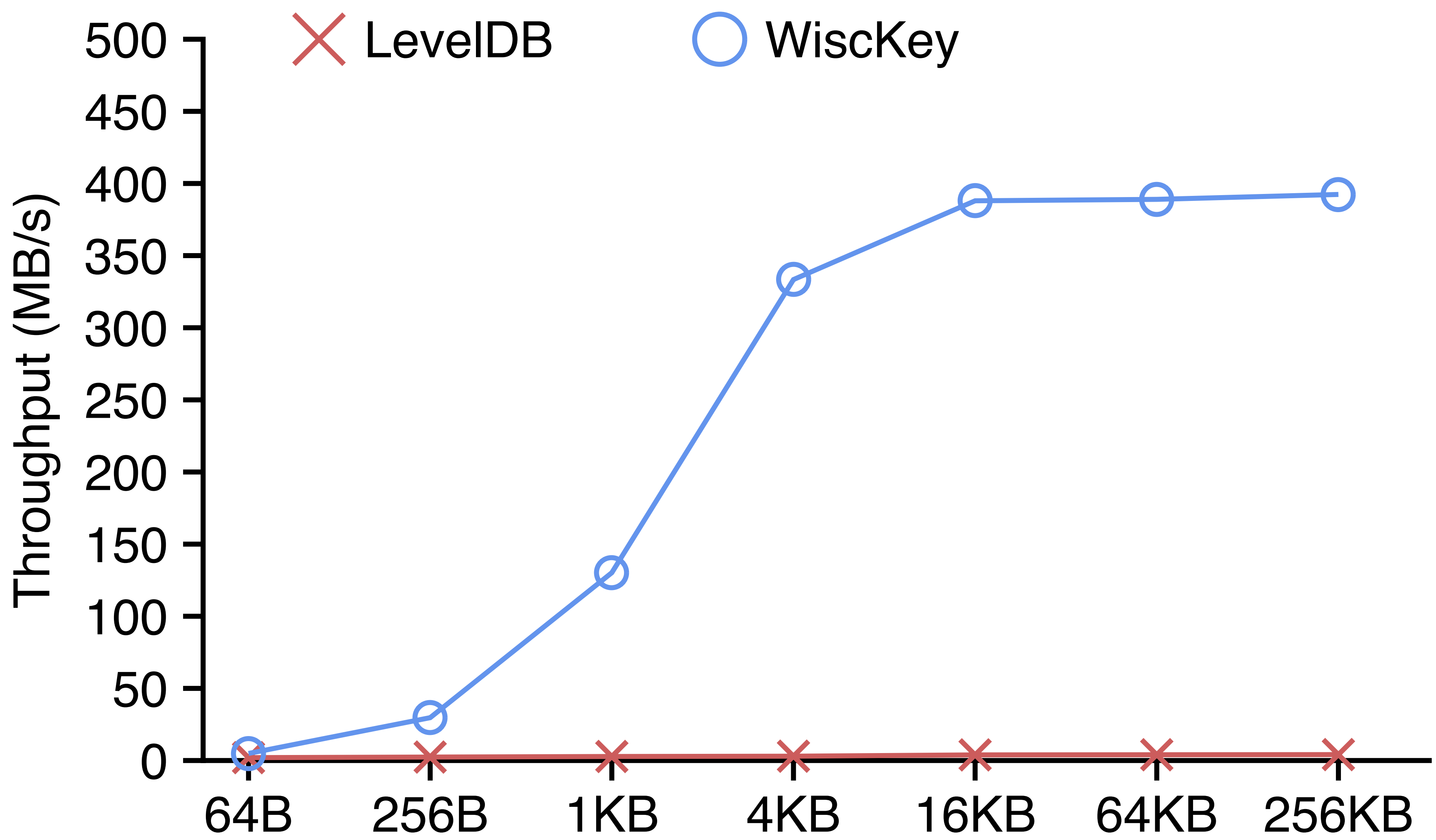
Key-Value Separation

Main idea: only keys are required to be sorted

Decouple sorting and garbage collection

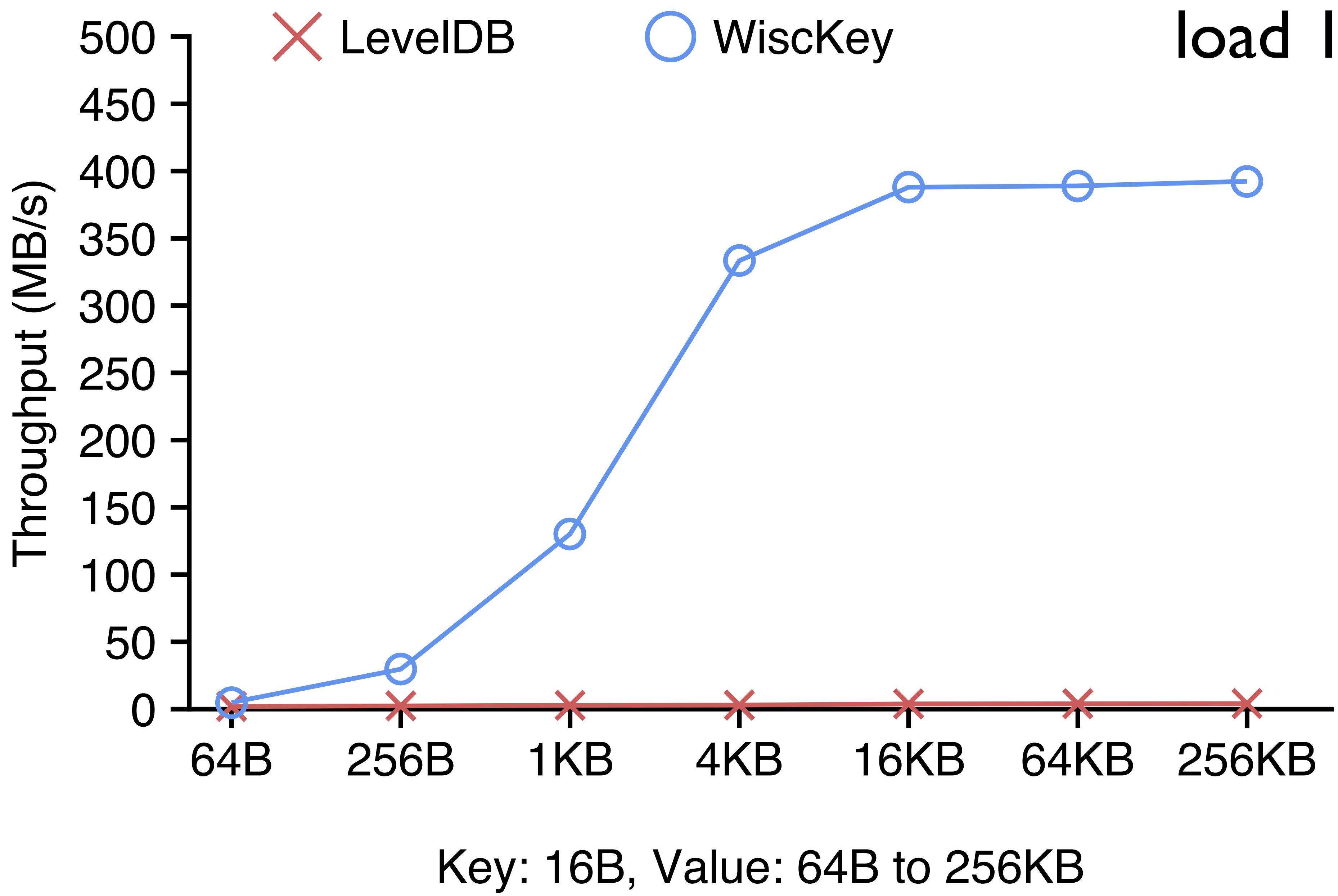


Random Load



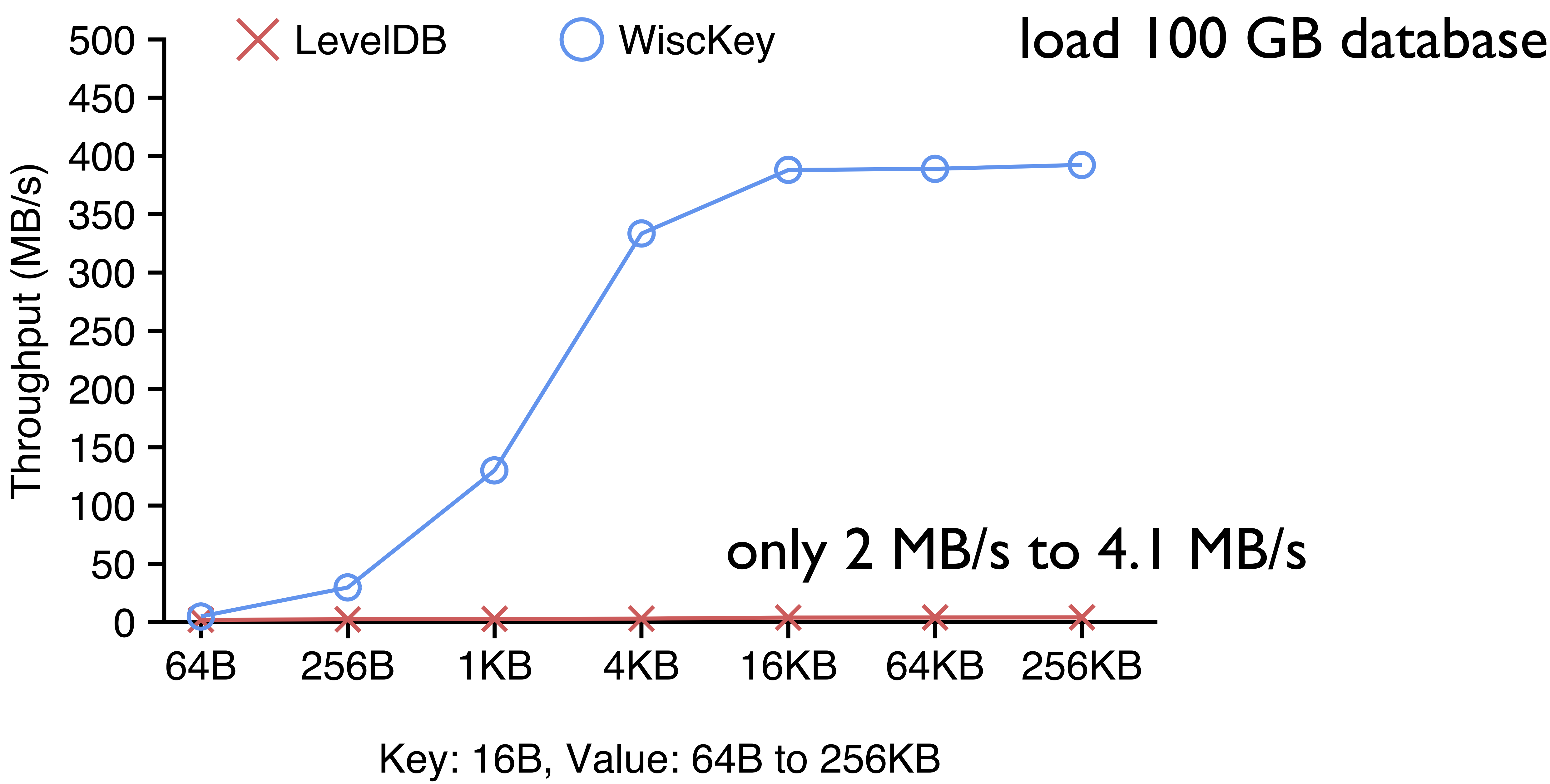
Key: 16B, Value: 64B to 256KB

Random Load



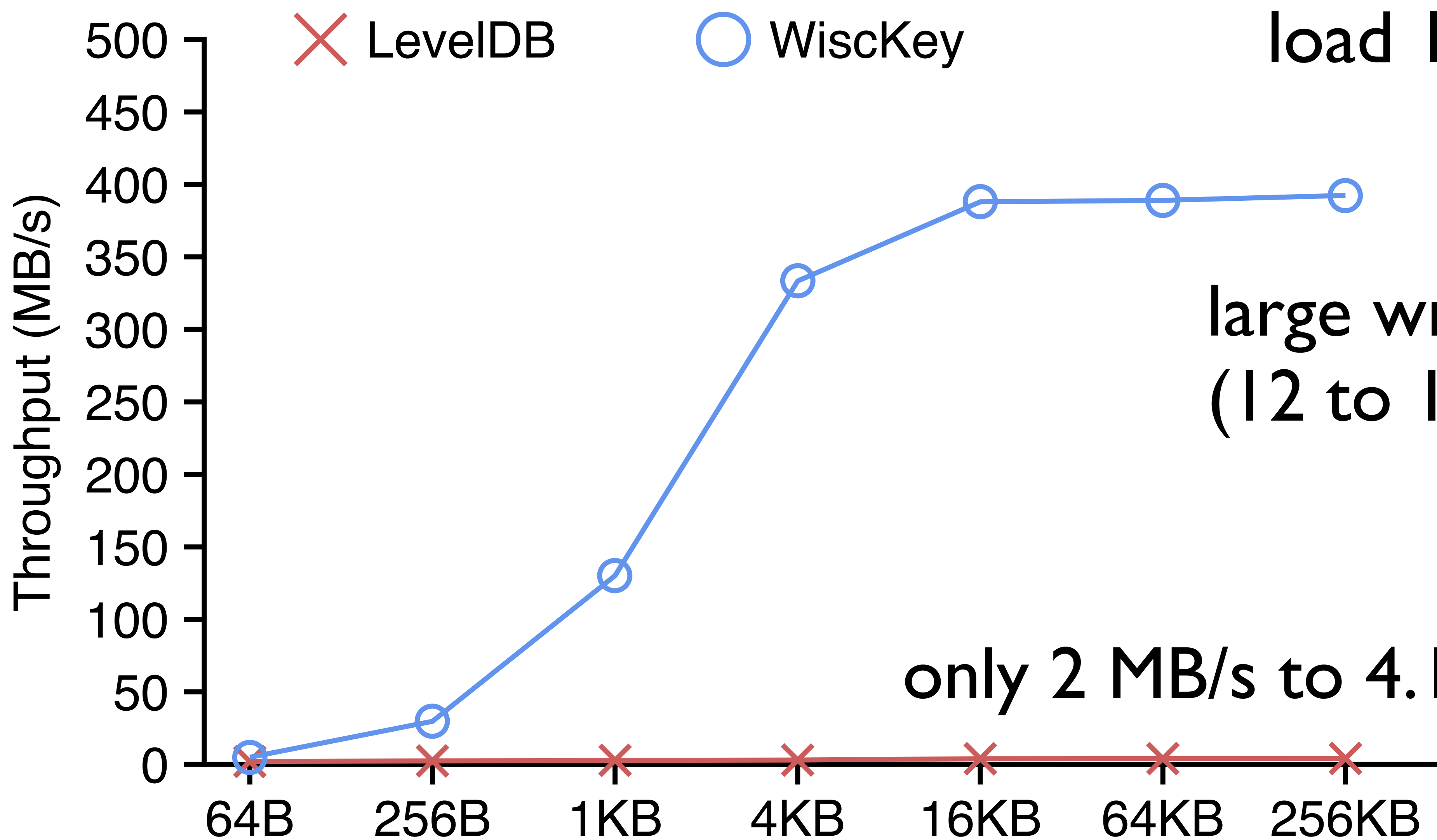
load 100 GB database

Random Load



LevelDB
WiscKey
load 100 GB database
Throughput (MB/s)
500
450
400
350
300
250
200
150
100
50
0
only 2 MB/s to 4.1 MB/s
64B
256B
1KB
4KB
16KB
64KB
256KB
Key: 16B, Value: 64B to 256KB

Random Load



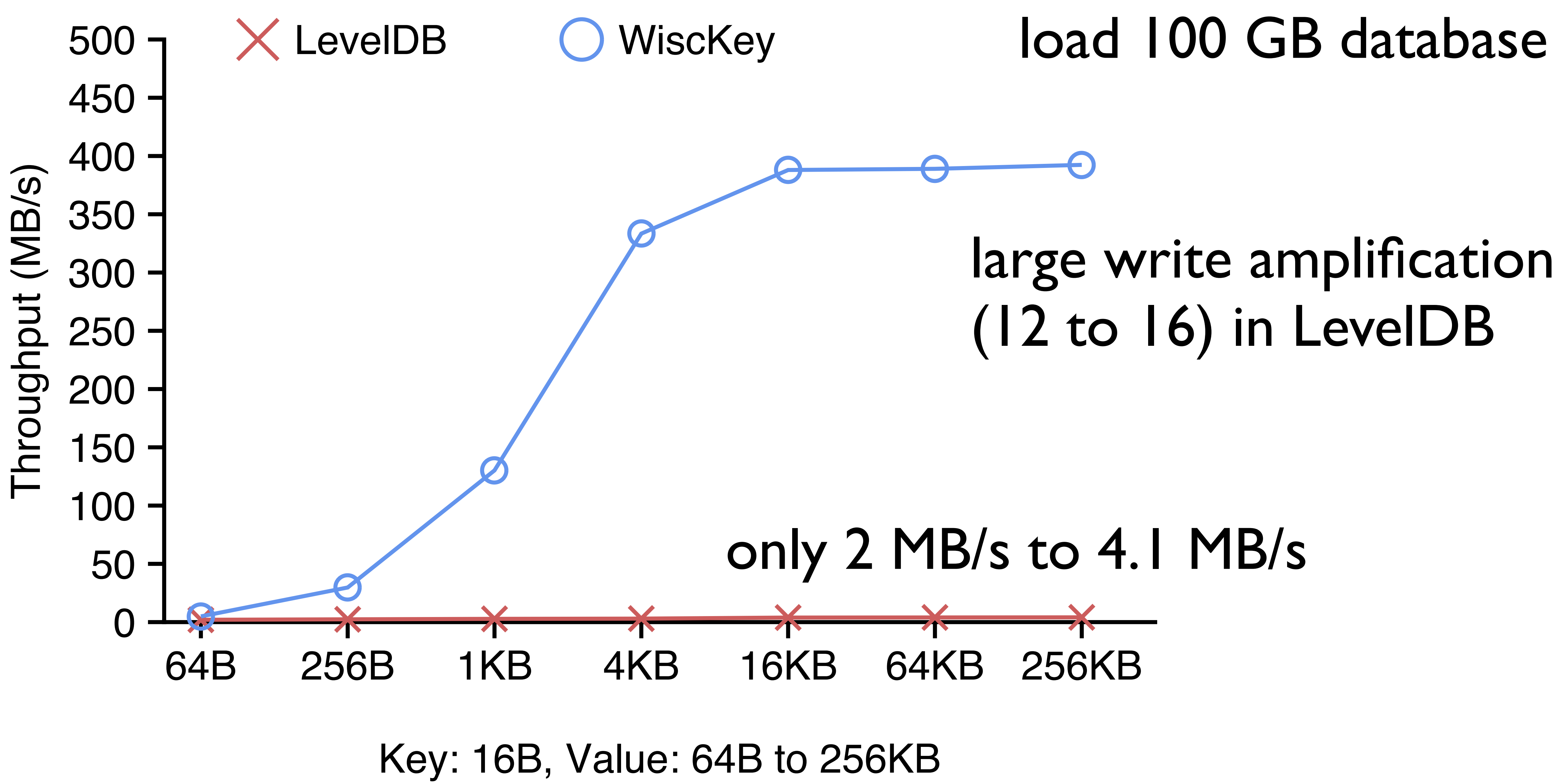
Key: 16B, Value: 64B to 256KB

load 100 GB database

large write amplification (12 to 16) in LevelDB

only 2 MB/s to 4.1 MB/s

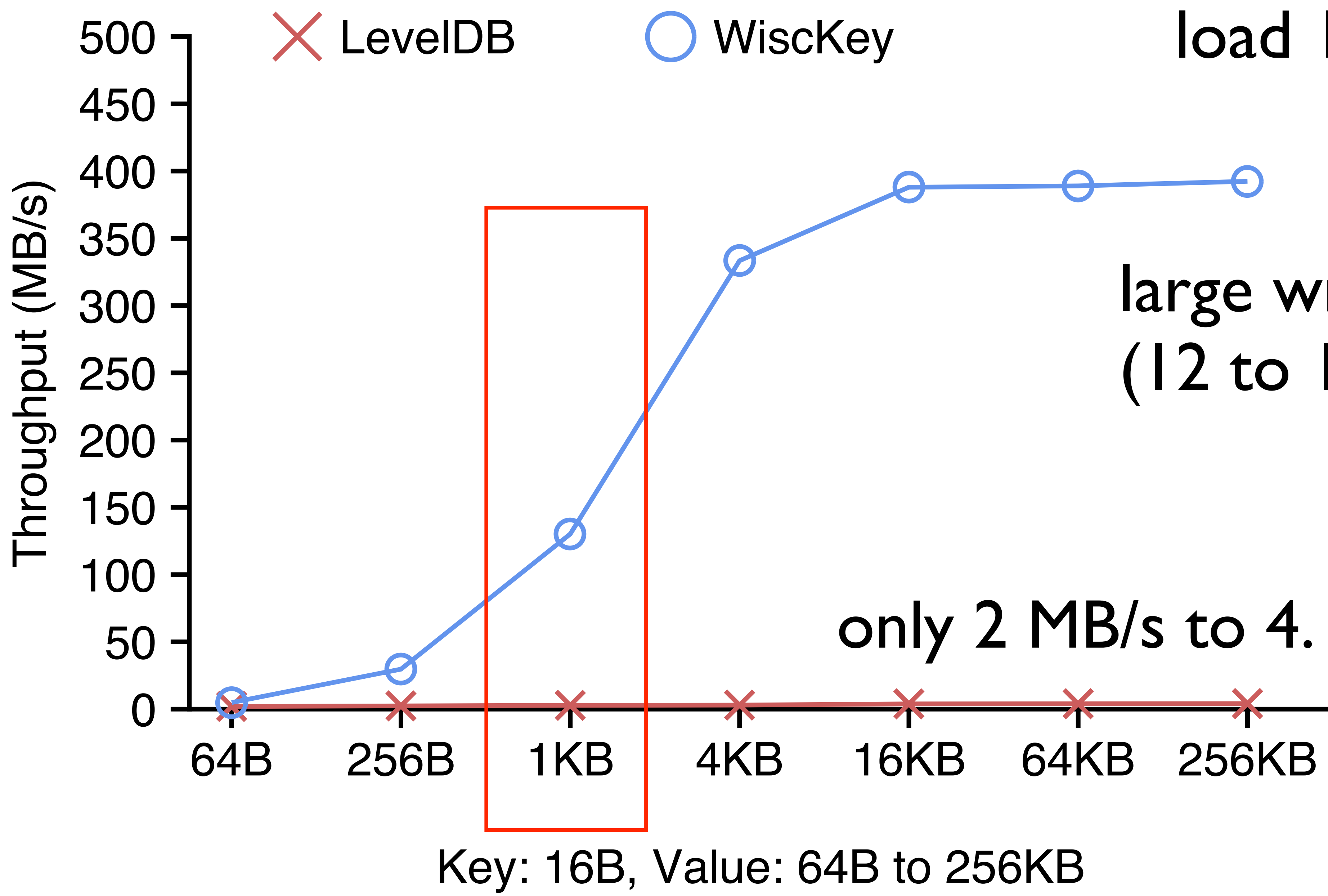
Random Load



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

Random Load

load 100 GB database



large write amplification (12 to 16) in LevelDB

only 2 MB/s to 4.1 MB/s

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

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

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

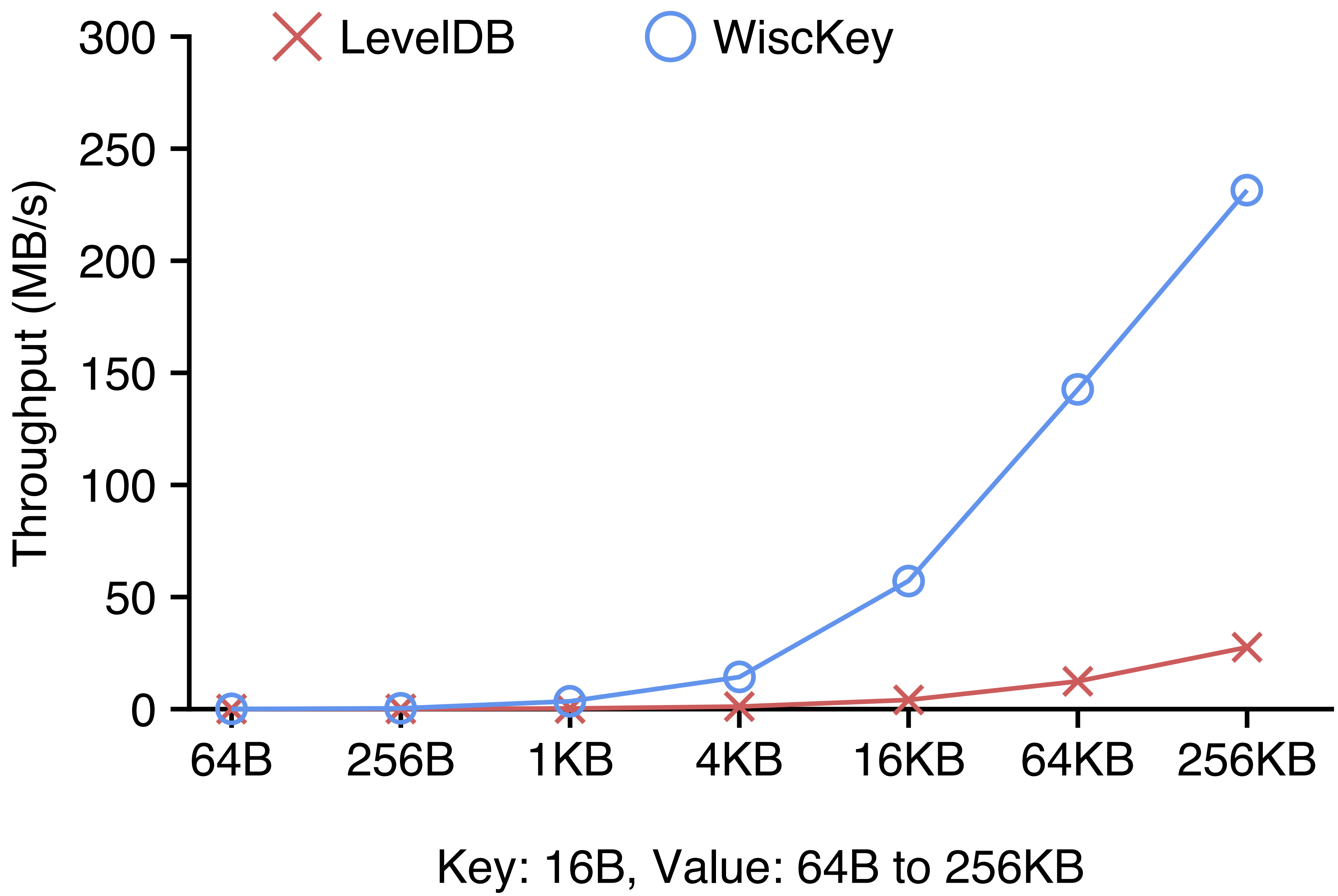
➡ travel several levels for each lookup

	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

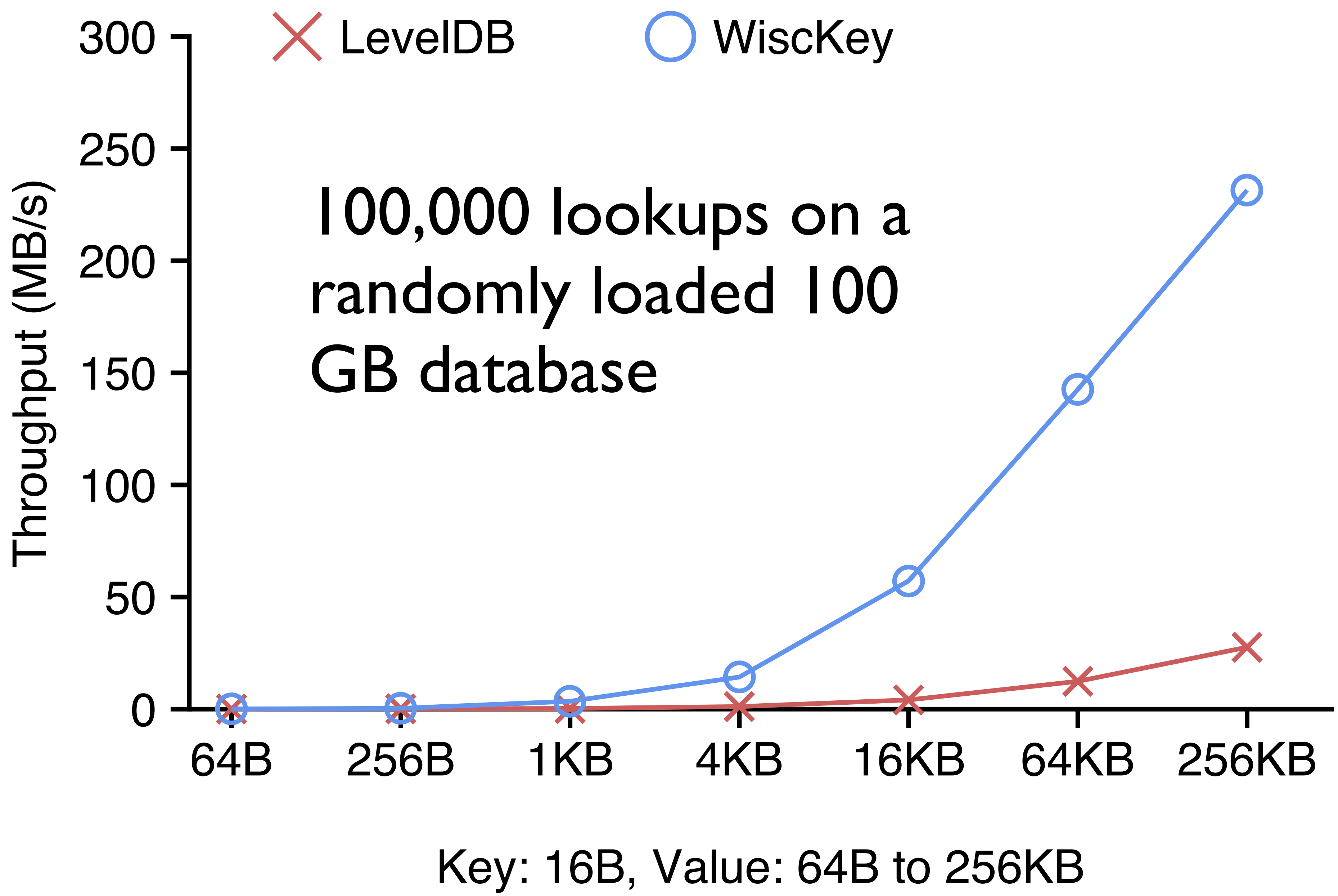
	LevelDB	WiscKey
limits of files	**num of files**	**num of files**
L0	9	7
L1 (5)	30	11
L2 (50)	365	127
L3 (500)	2184	460
L4 (5000)	15752	0
L5 (50000)	23733	0
L6 (500000)	0	0

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

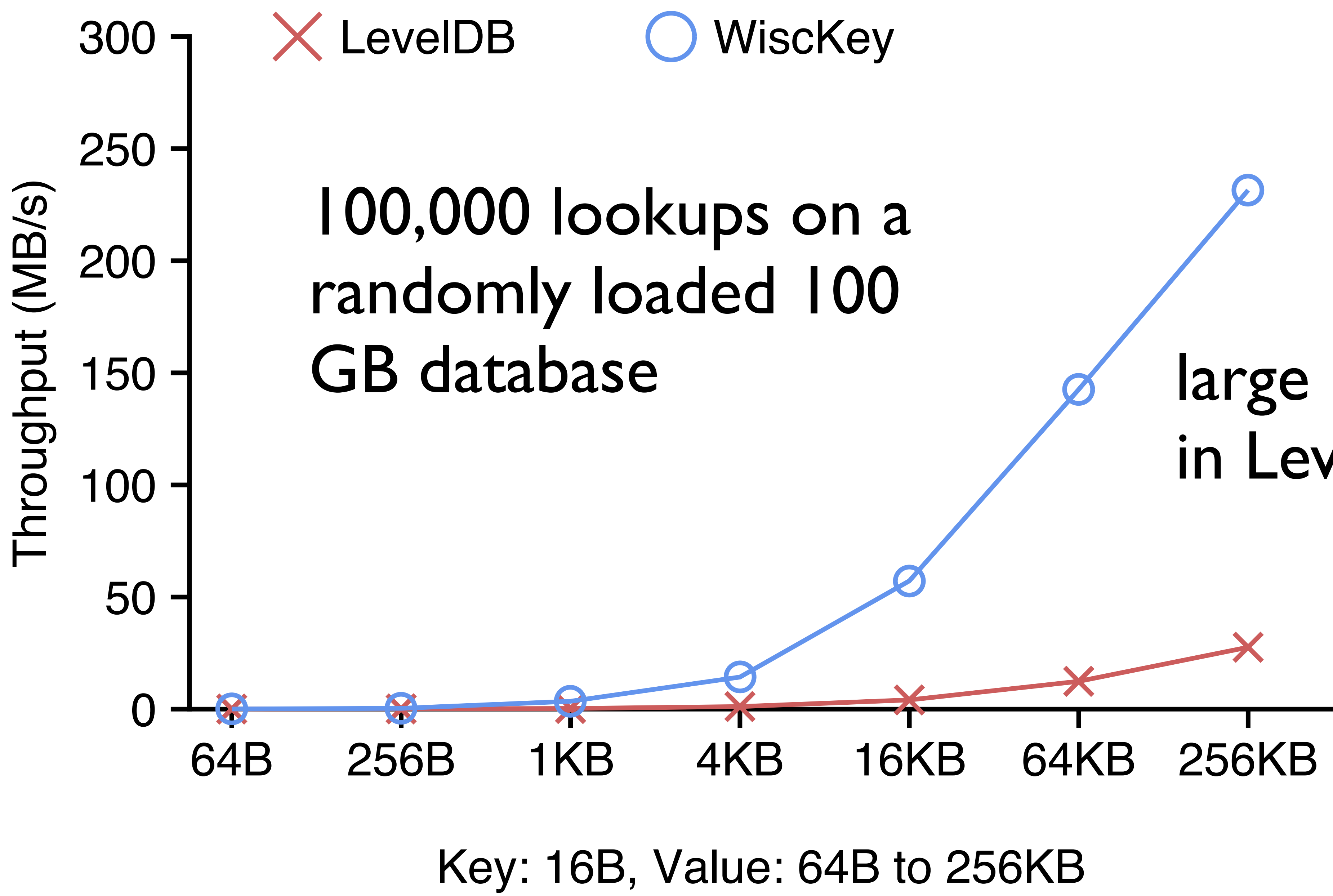
Random Lookup



Random Lookup



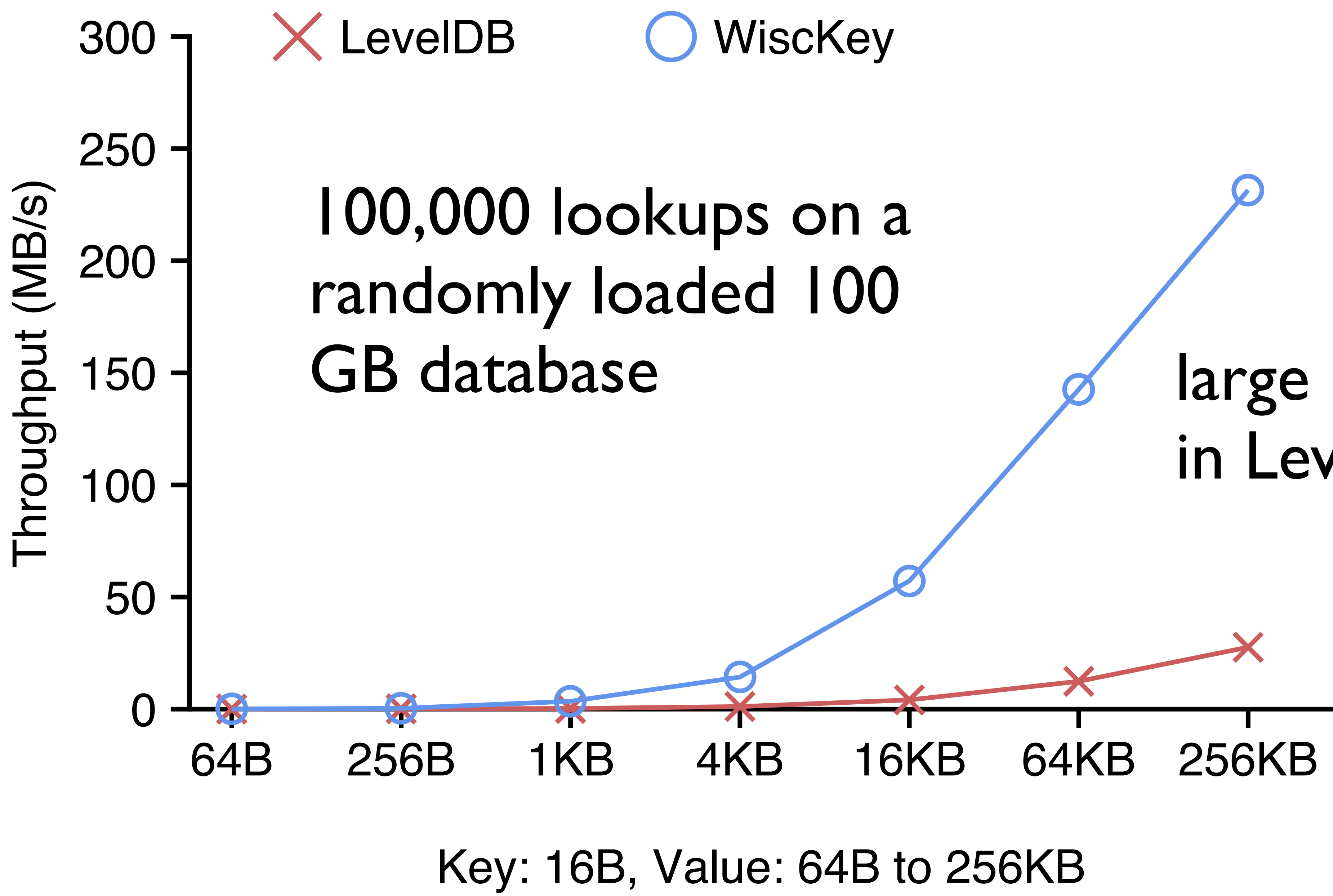
Random Lookup



100,000 lookups on a randomly loaded 100 GB database

large read amplification in LevelDB

Random Lookup



large read amplification in LevelDB

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

Background

Key-Value Separation

Challenges and Optimizations

- Parallel range query
- Garbage collection
- LSM-log

Evaluation

Conclusion

Parallel Range Query

Parallel Range Query

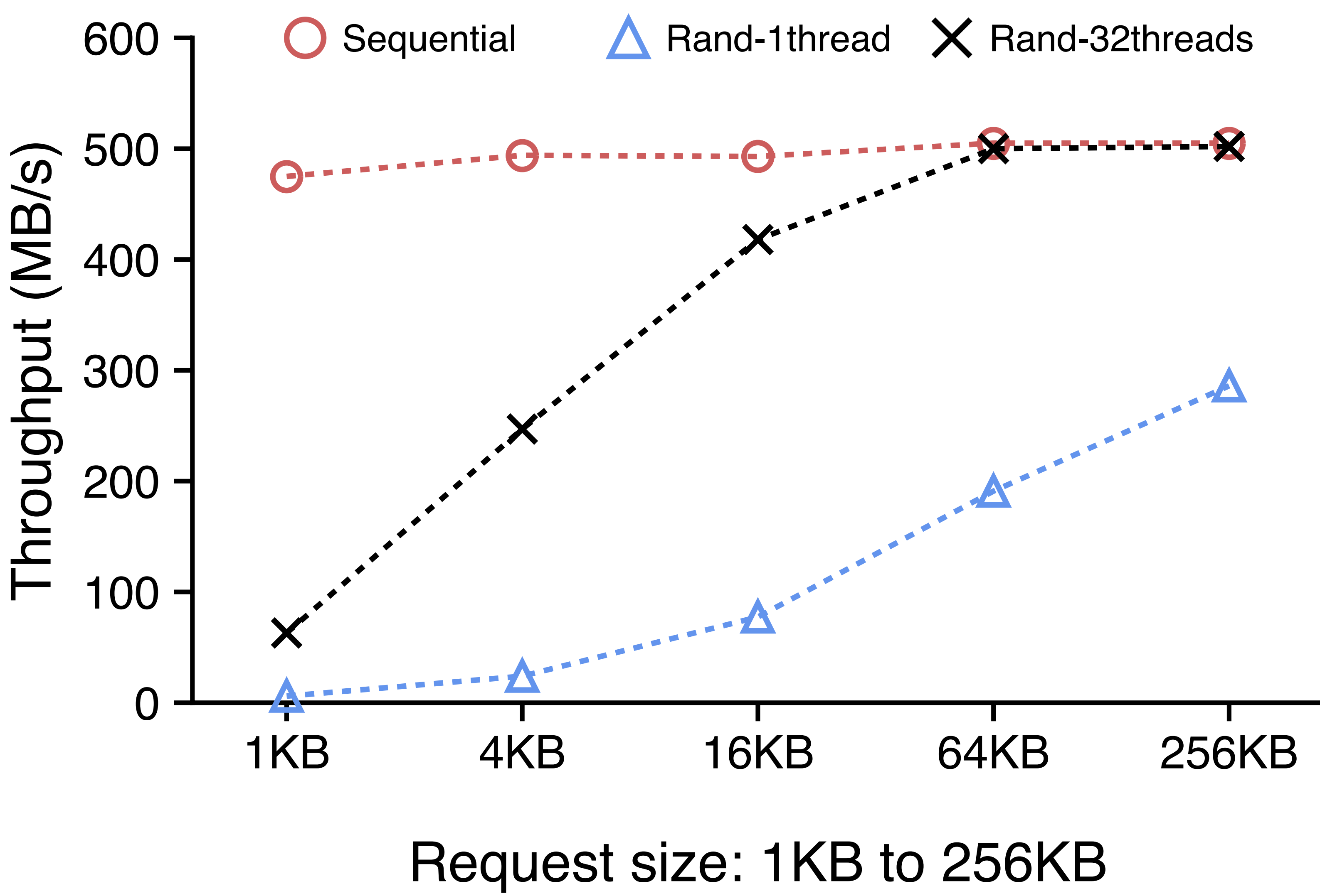
SSD read performance

- sequential, random, parallel

Parallel Range Query

SSD read performance

- sequential, random, parallel



SSD: Samsung 840 EVO 500GB

Reads on a 100GB file on ext4

Parallel Range Query

Parallel Range Query

Challenge

- sequential reads in LevelDB
- read keys and values separately in WiscKey

Parallel Range Query

Challenge

- sequential reads in LevelDB
- read keys and values separately in WiscKey

Parallel range query

- leverage parallel random reads of SSDs

Parallel Range Query

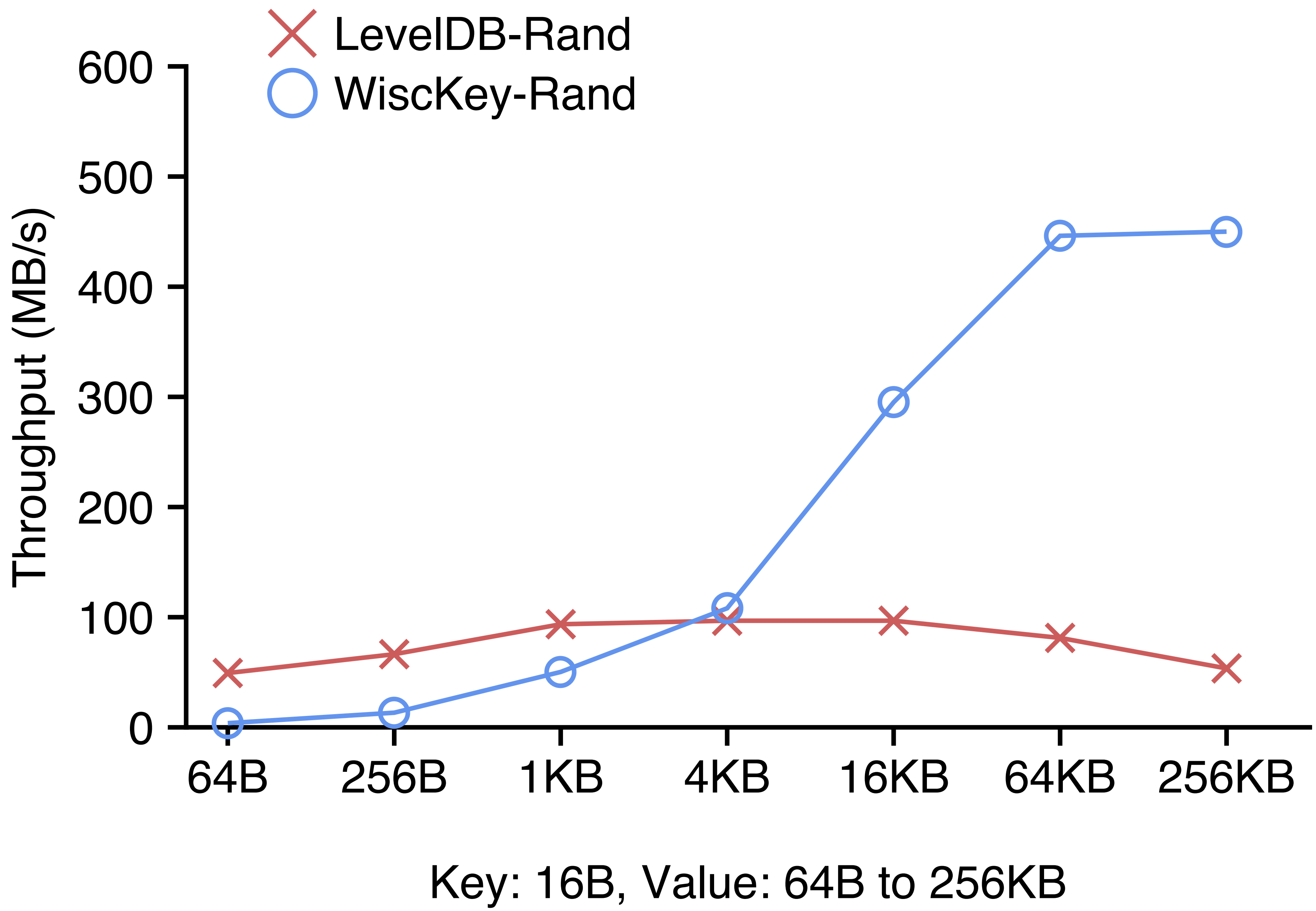
Challenge

- sequential reads in LevelDB
- read keys and values separately in WiscKey

Parallel range query

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

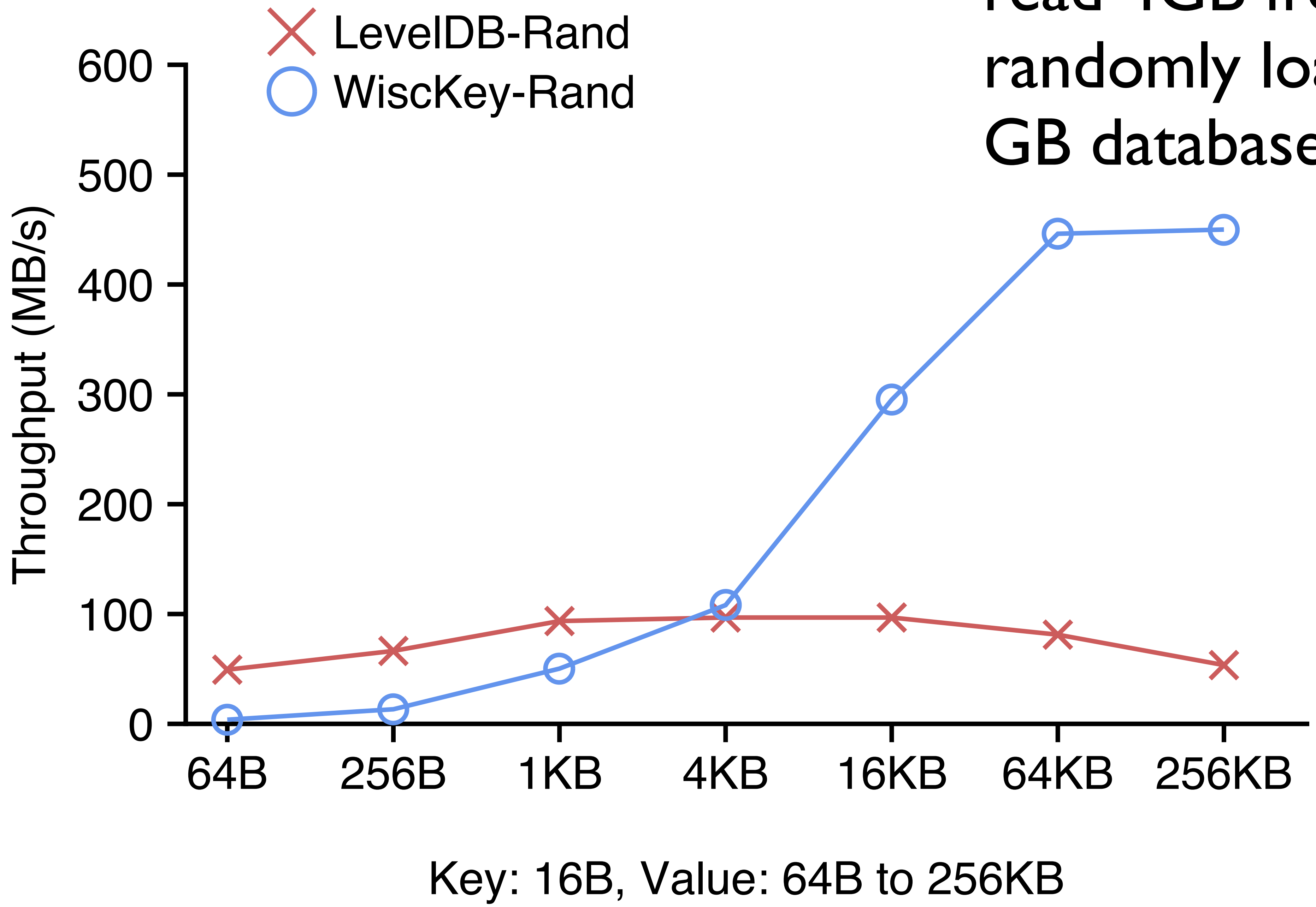
Range Query



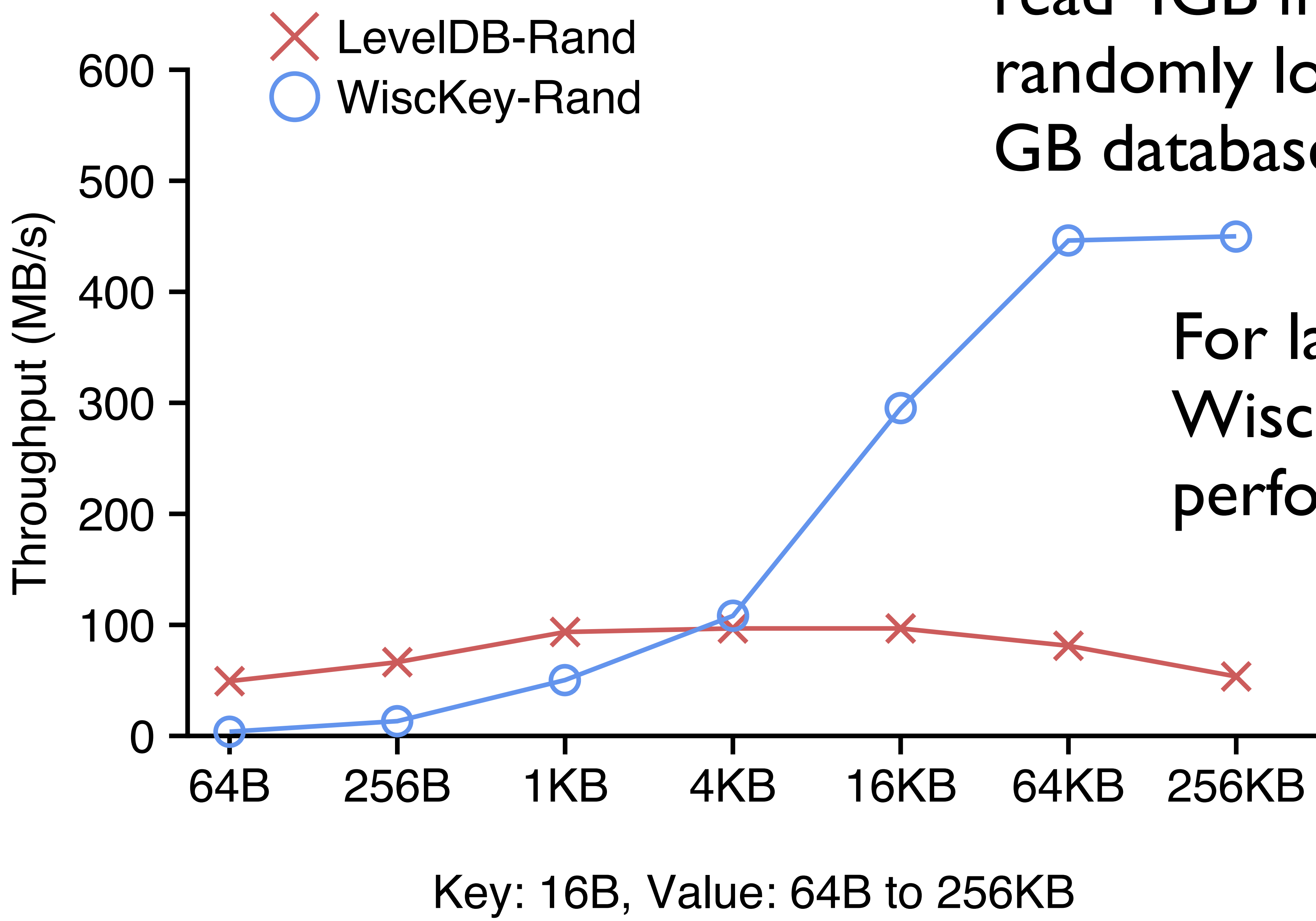
LevelDB-Rand
WiscKey-Rand
Throughput (MB/s)
600
500
400
300
200
100
0
64B
256B
1KB
4KB
16KB
64KB
256KB
Key: 16B, Value: 64B to 256KB

Range Query

read 4GB from a randomly loaded 100 GB database



Range Query

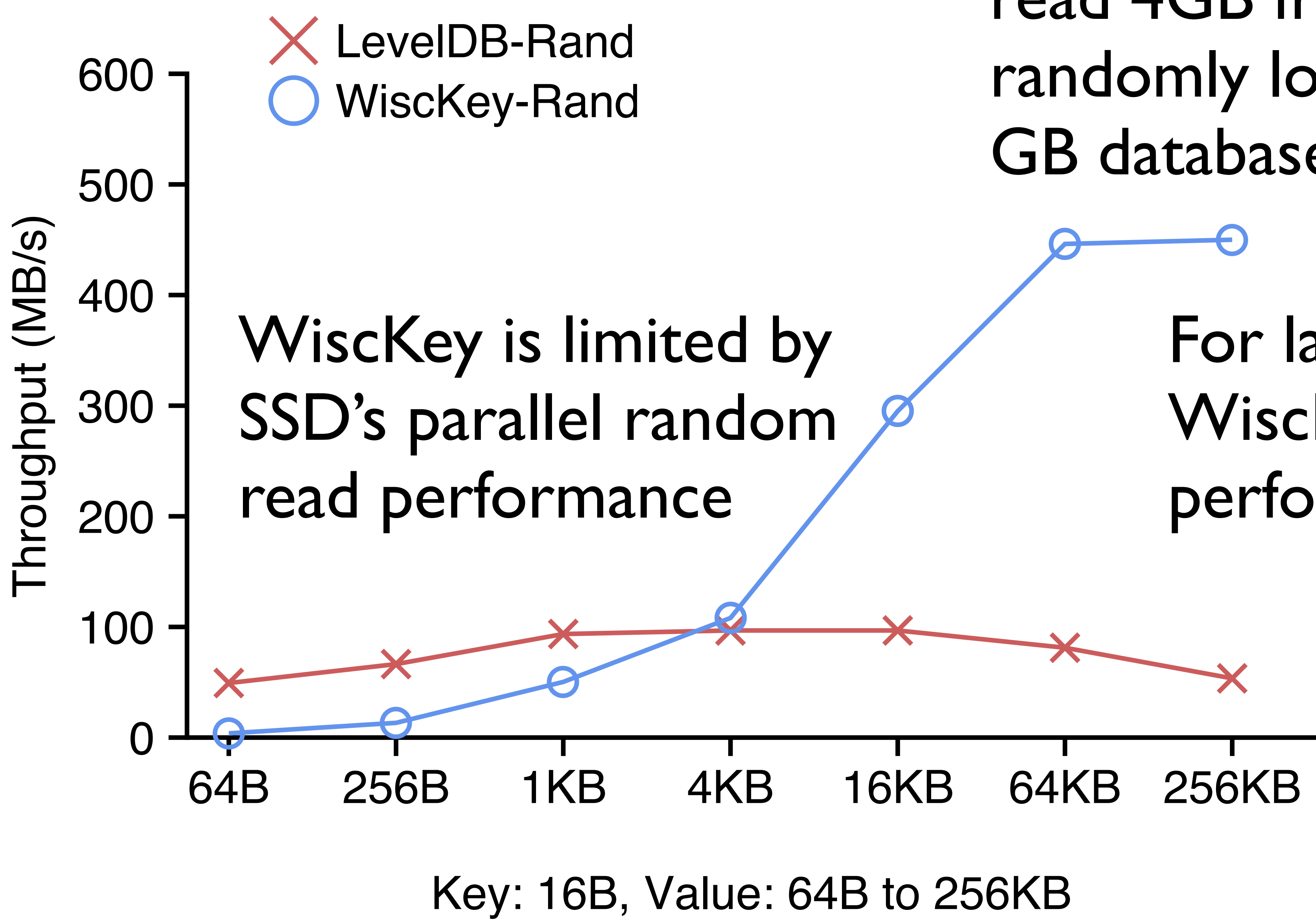


read 4GB from a randomly loaded 100 GB database

For large kv pairs, WiscKey can perform better

Range Query

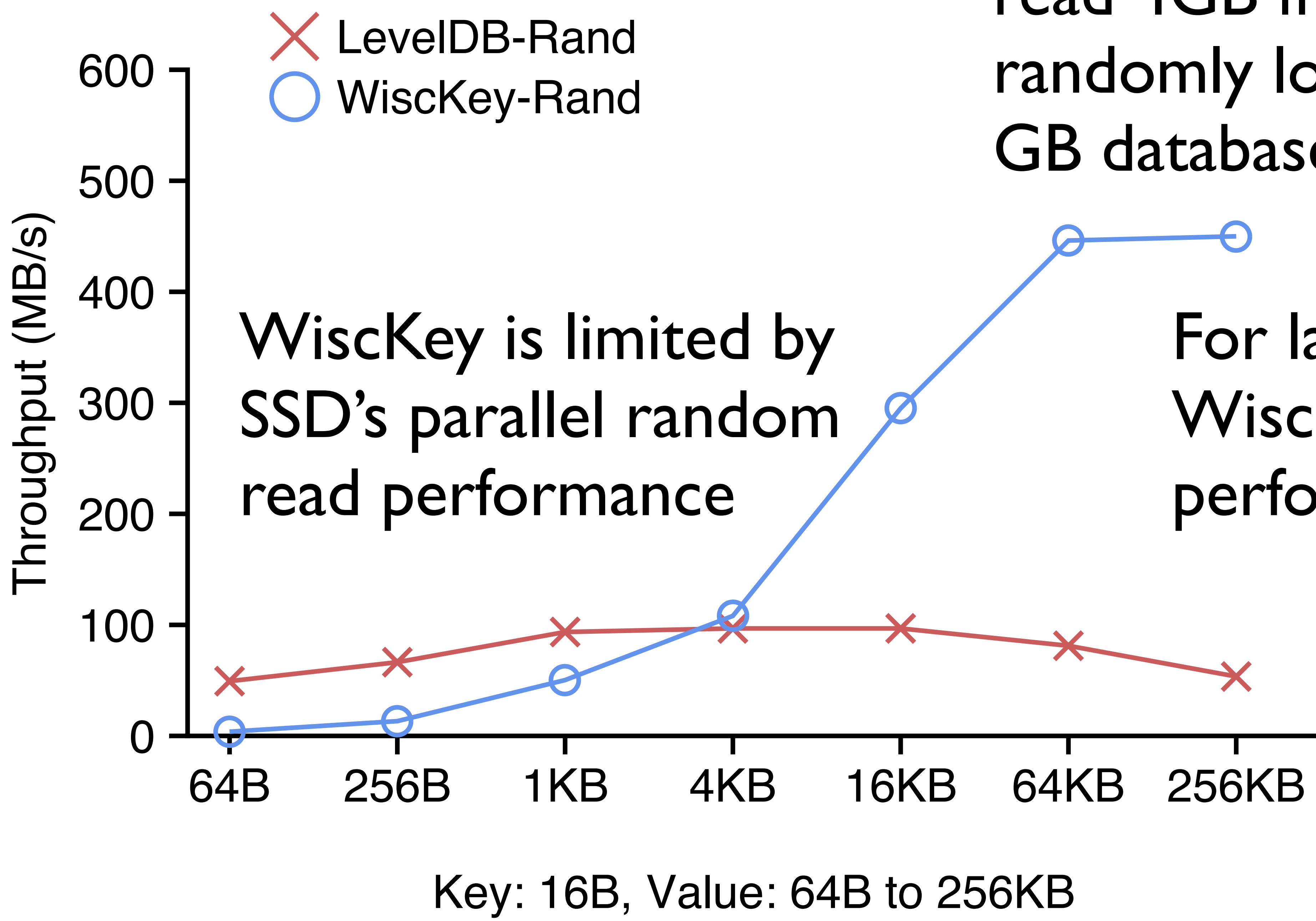
read 4GB from a randomly loaded 100 GB database



WiscKey is limited by SSD's parallel random read performance

For large kv pairs, WiscKey can perform better

Range Query



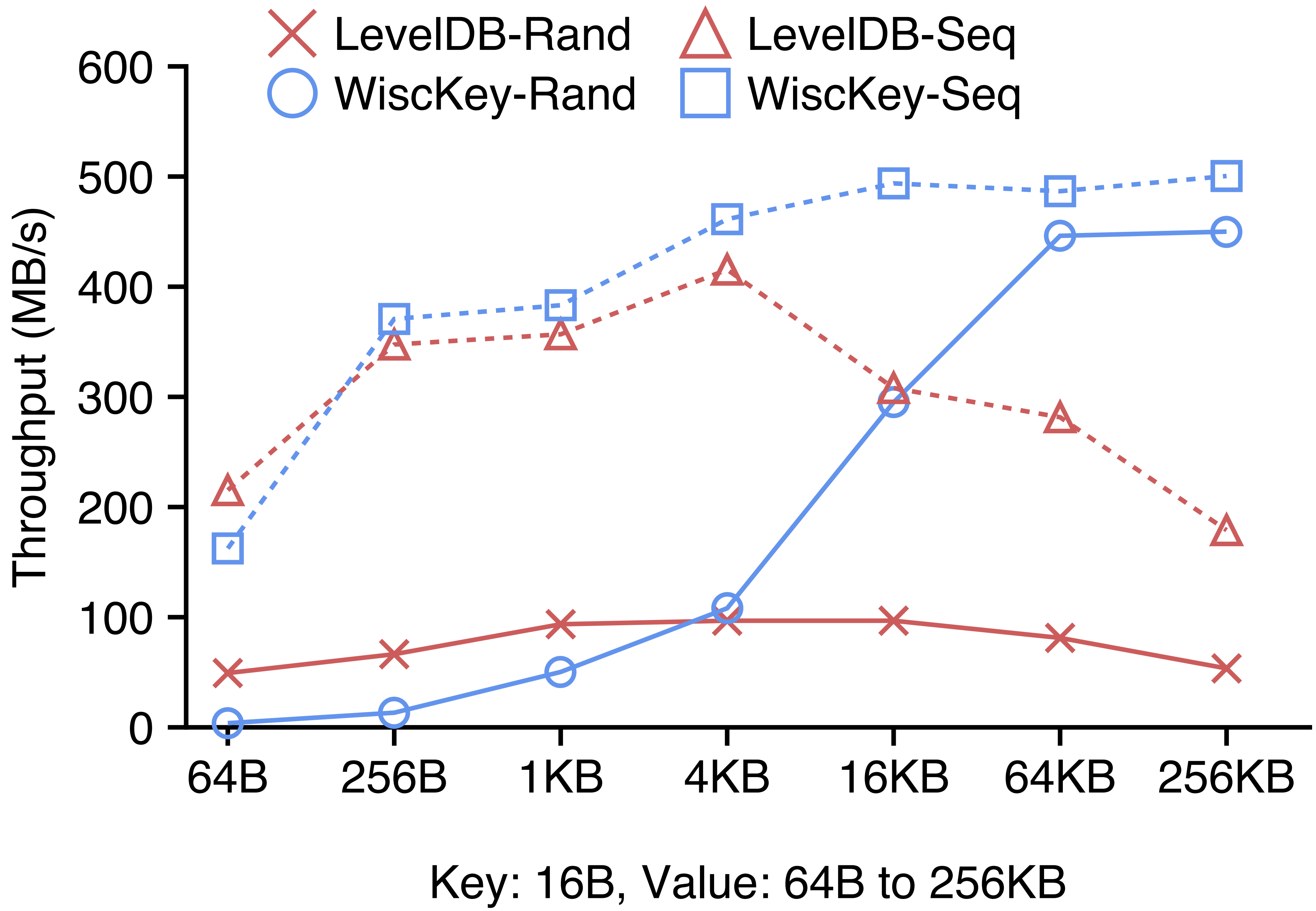
read 4GB from a randomly loaded 100 GB database

WiscKey is limited by SSD's parallel random read performance

For large kv pairs, WiscKey can perform better

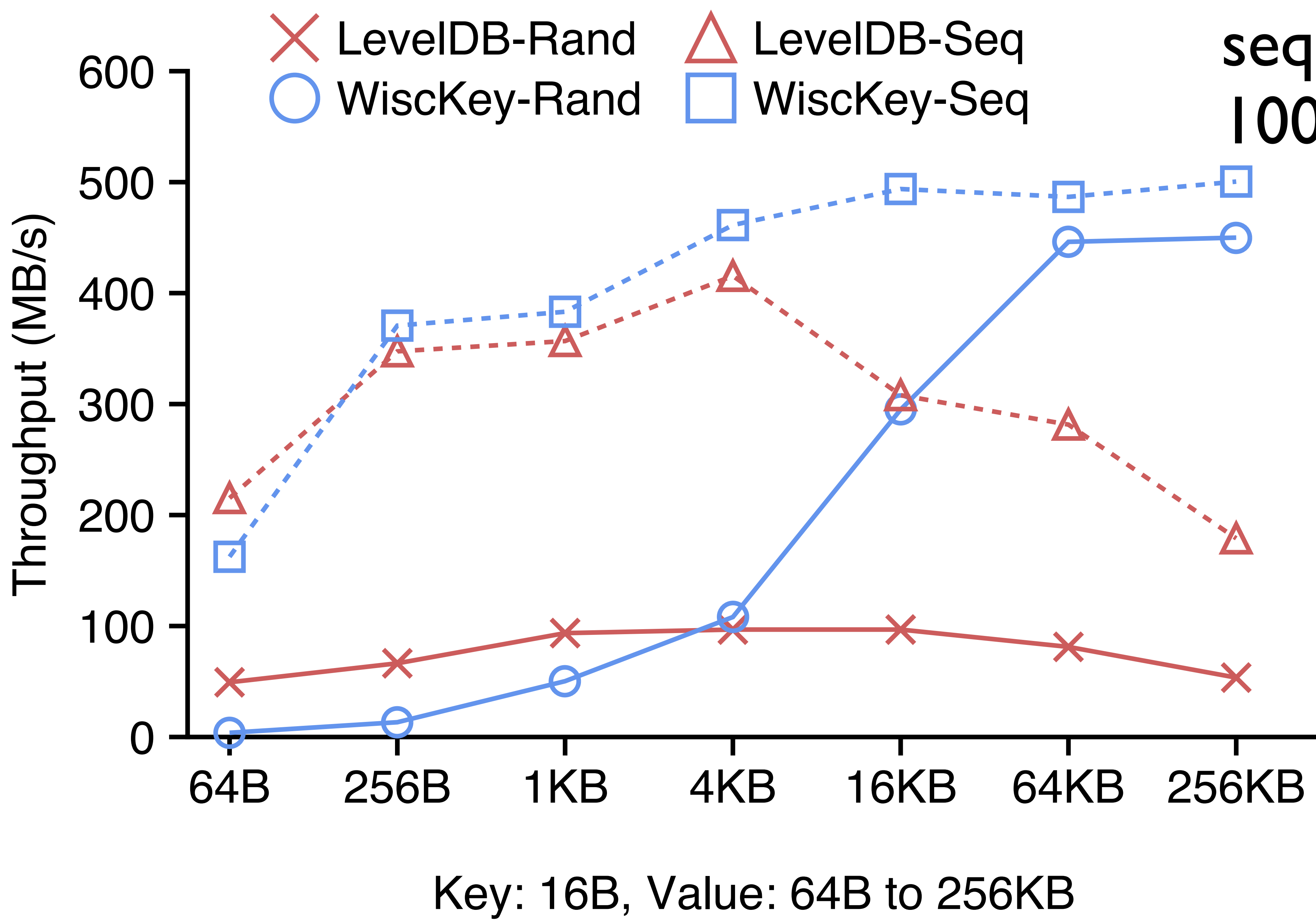
Better for large kv pairs, but worse for small kv pairs on an unsorted database

Range Query



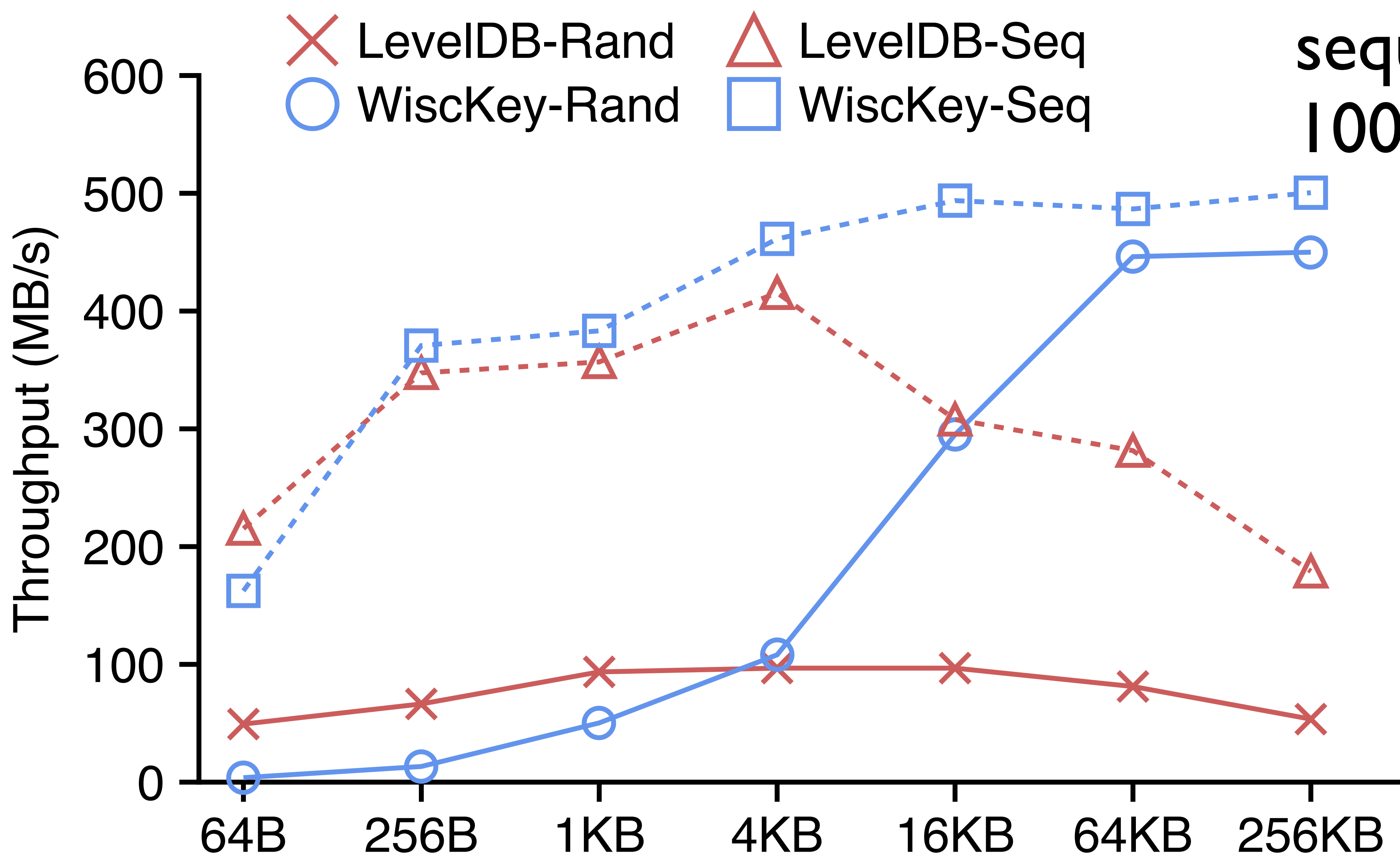
LevelDB-Rand
LevelDB-Seq
WiscKey-Rand
WiscKey-Seq
Throughput (MB/s)
600
500
400
300
200
100
0
64B
256B
1KB
4KB
16KB
64KB
256KB
Key: 16B, Value: 64B to 256KB

Range Query



read 4GB from a sequentially loaded 100 GB database

Range Query

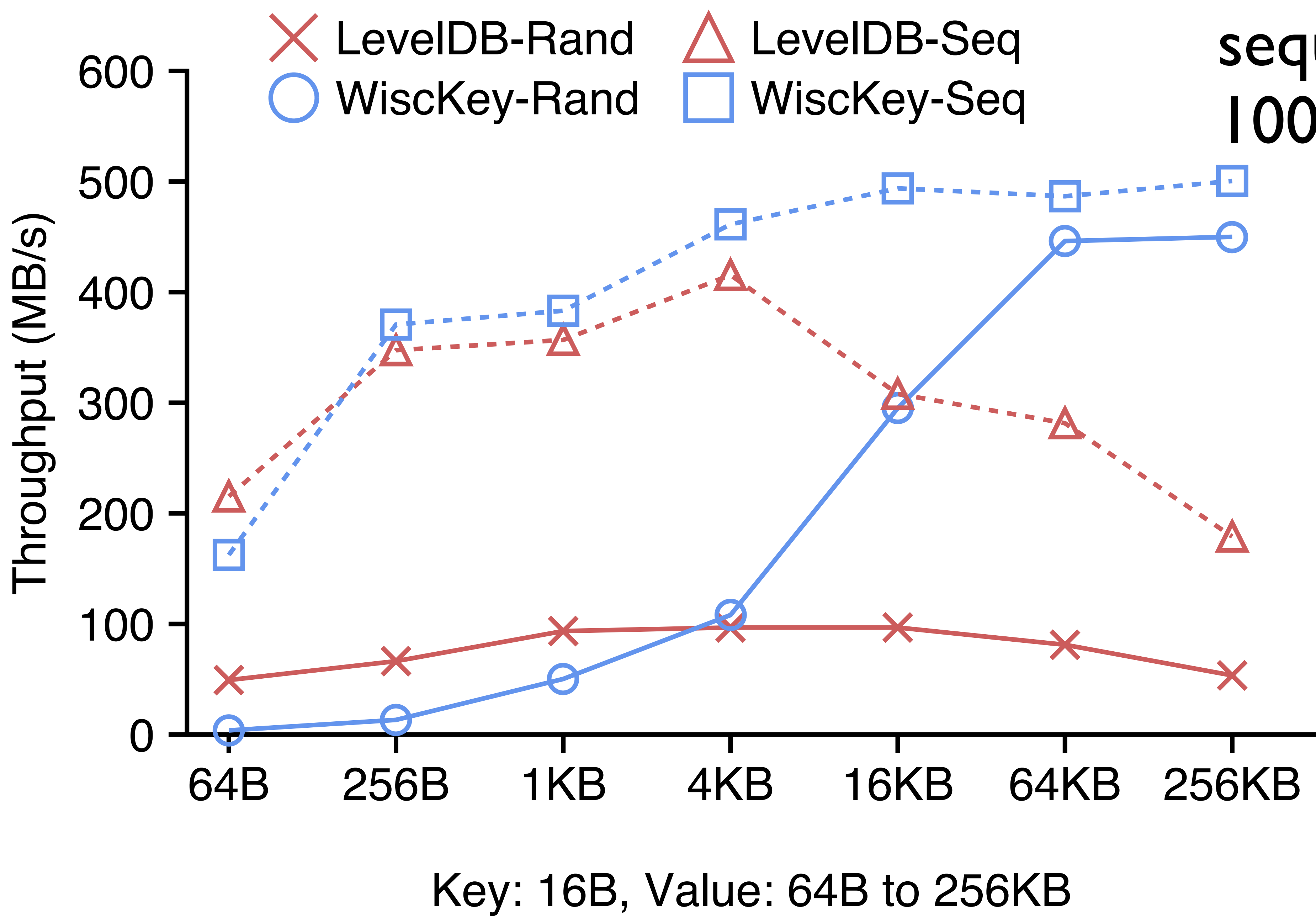


Key: 16B, Value: 64B to 256KB

read 4GB from a sequentially loaded 100 GB database

Both WiscKey and LevelDB read sequentially

Range Query



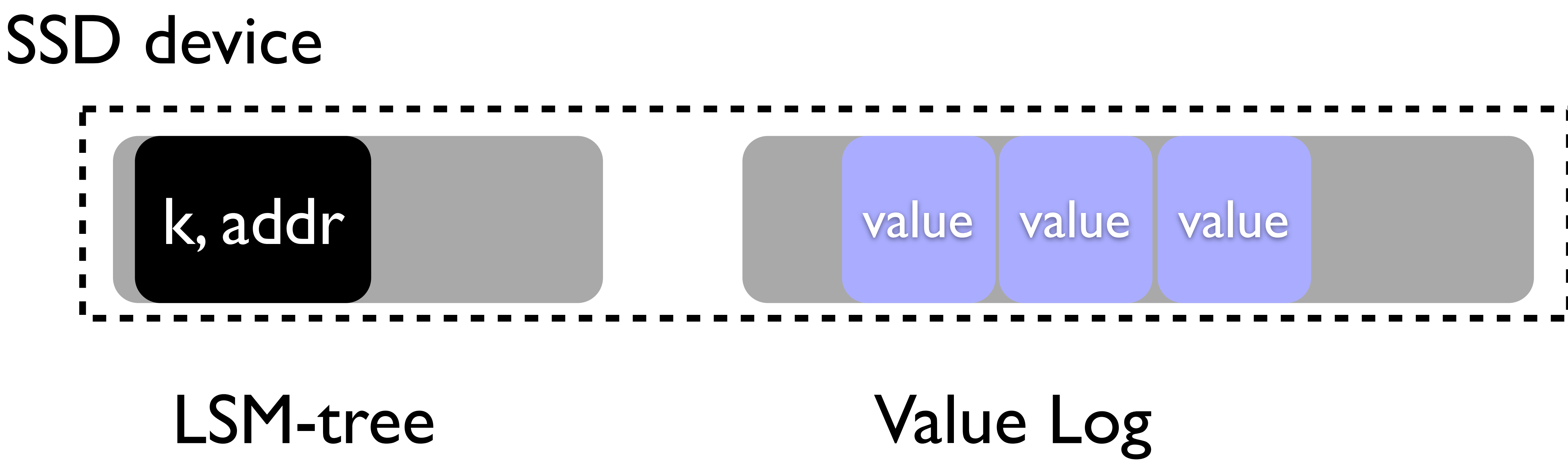
read 4GB from a sequentially loaded 100 GB database

Both WiscKey and LevelDB read sequentially

Sorted databases help WiscKey's range query

Optimizations

Optimizations

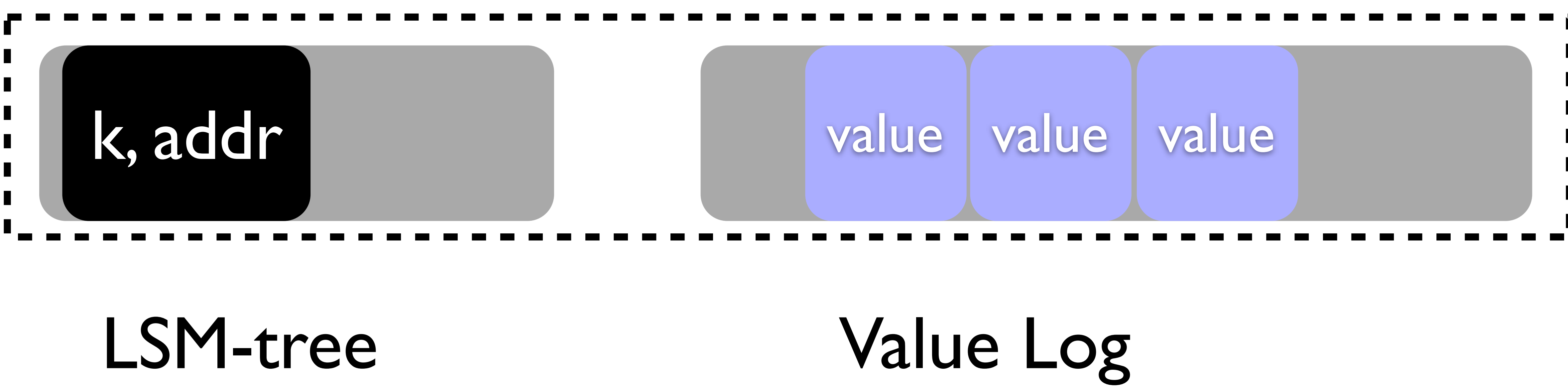


Optimizations

Online and light-weight garbage collection

- append (ksize, vsize, key, value) in value log

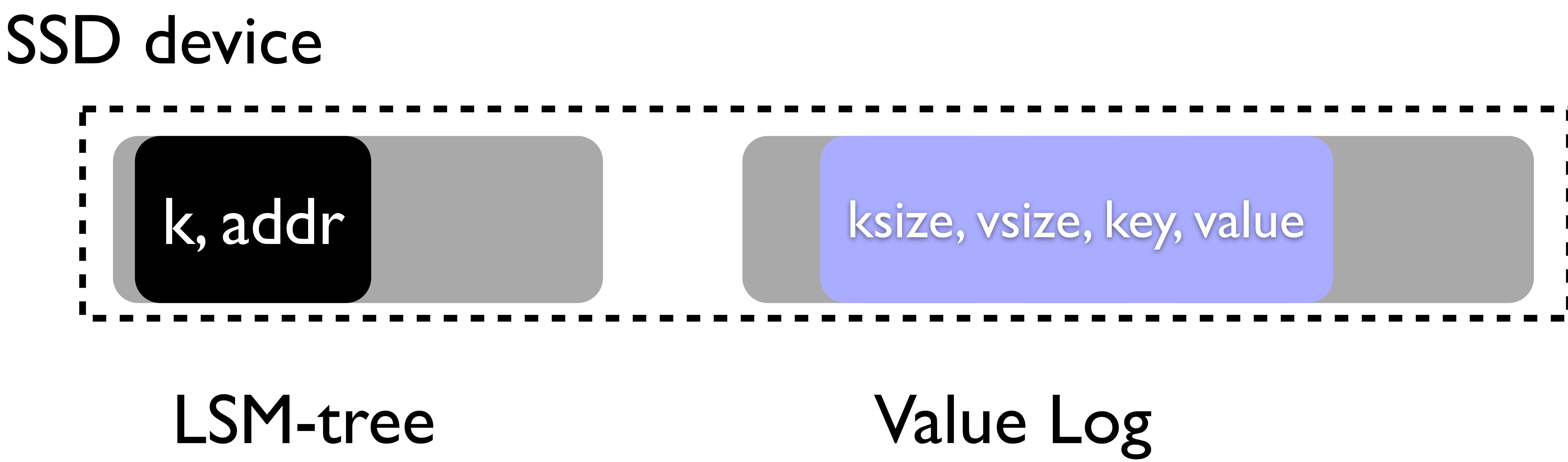
SSD device



Optimizations

Online and light-weight garbage collection

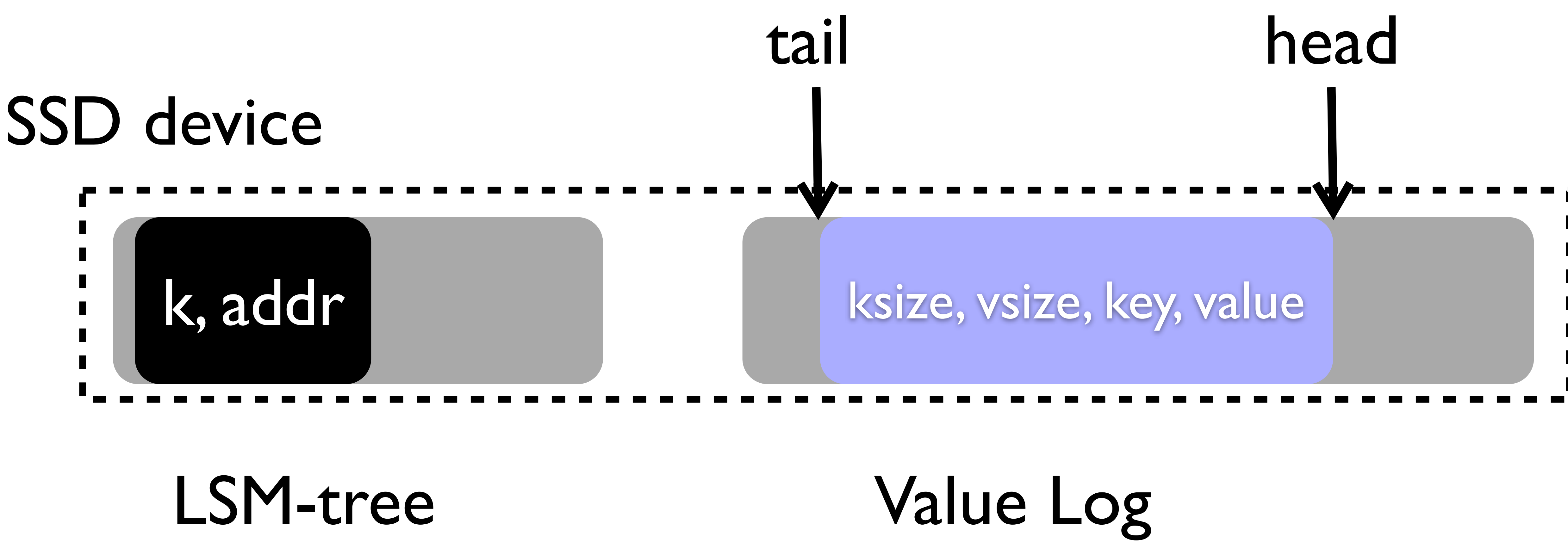
- append (ksize, vsize, key, value) in value log



Optimizations

Online and light-weight garbage collection

- append (ksize, vsize, key, value) in value log

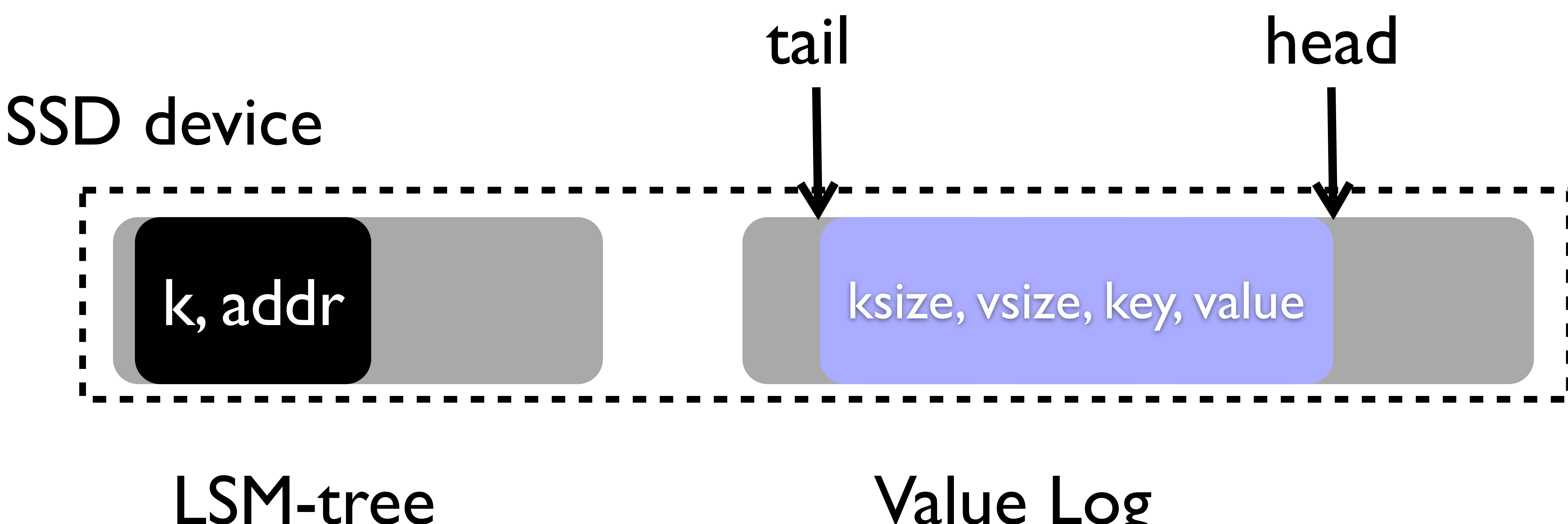


Optimizations

Online and light-weight garbage collection

- append (ksize, vsize, key, value) in value log

Remove LSM-tree log in WiscKey



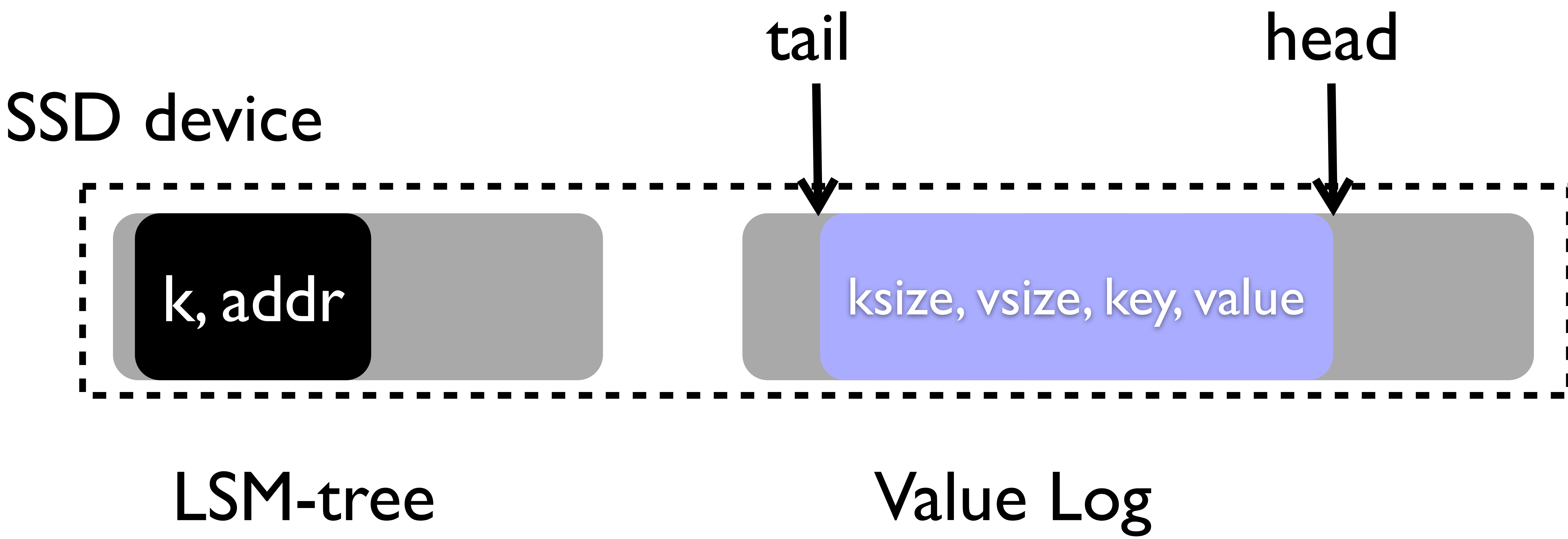
Optimizations

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



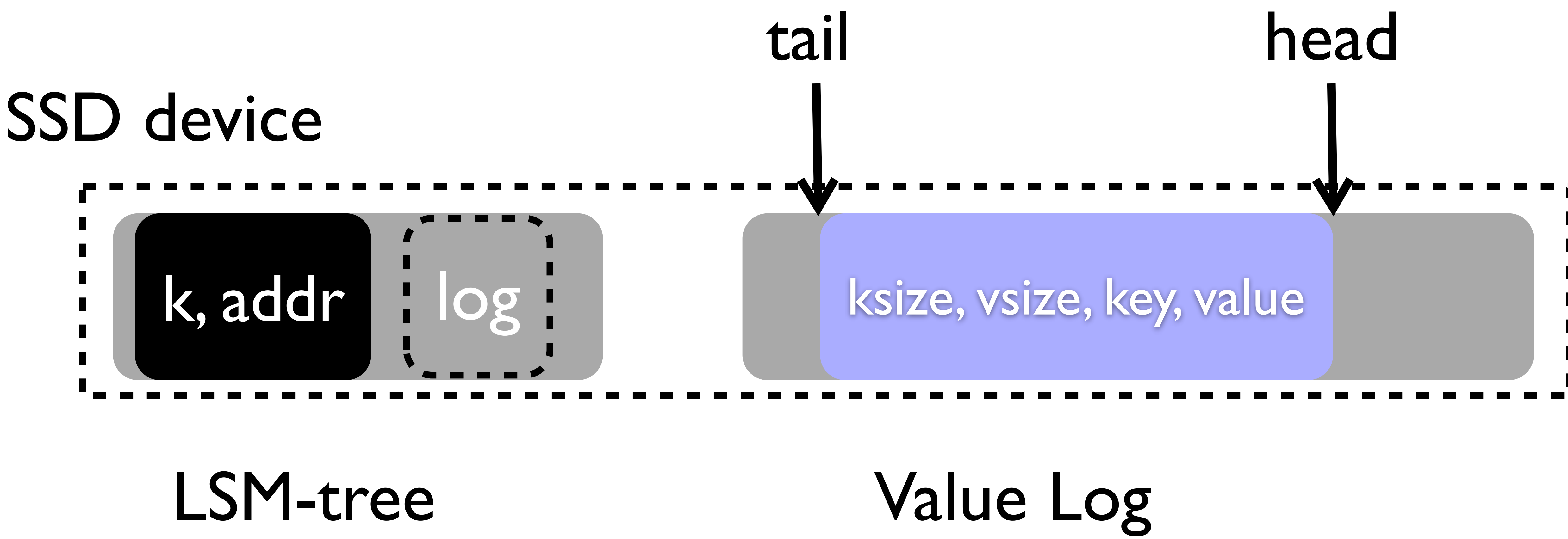
Optimizations

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



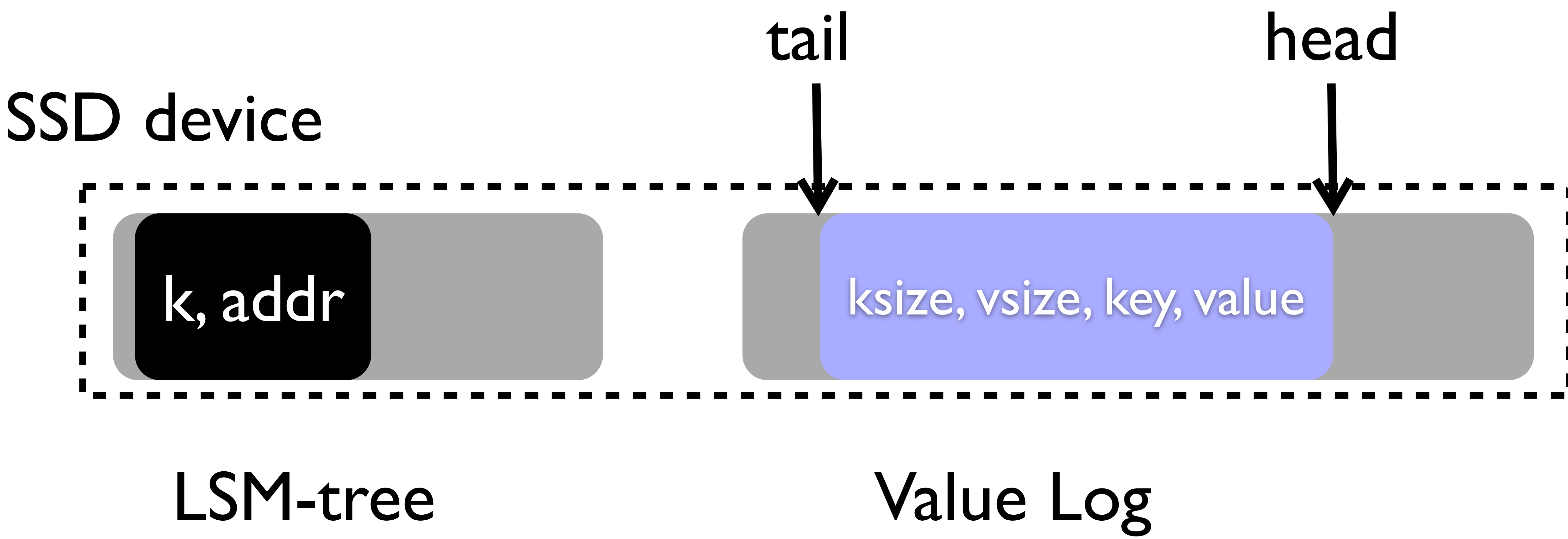
Optimizations

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



WiscKey Implementation

WiscKey Implementation

Based on LevelDB

- ➡ a separate vLog file for values
- ➡ modify I/O paths to separate keys and values
- ➡ leverages most of high-quality LevelDB source code

WiscKey Implementation

Based on LevelDB

- 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 launches queries in parallel
- detect sequential pattern with the Iterator interface

WiscKey Implementation

Based on LevelDB

- 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 launches queries in parallel
- detect sequential pattern with the Iterator interface

File-system support

- fadvise to predeclare access patterns
- hole-punching to free space

Background

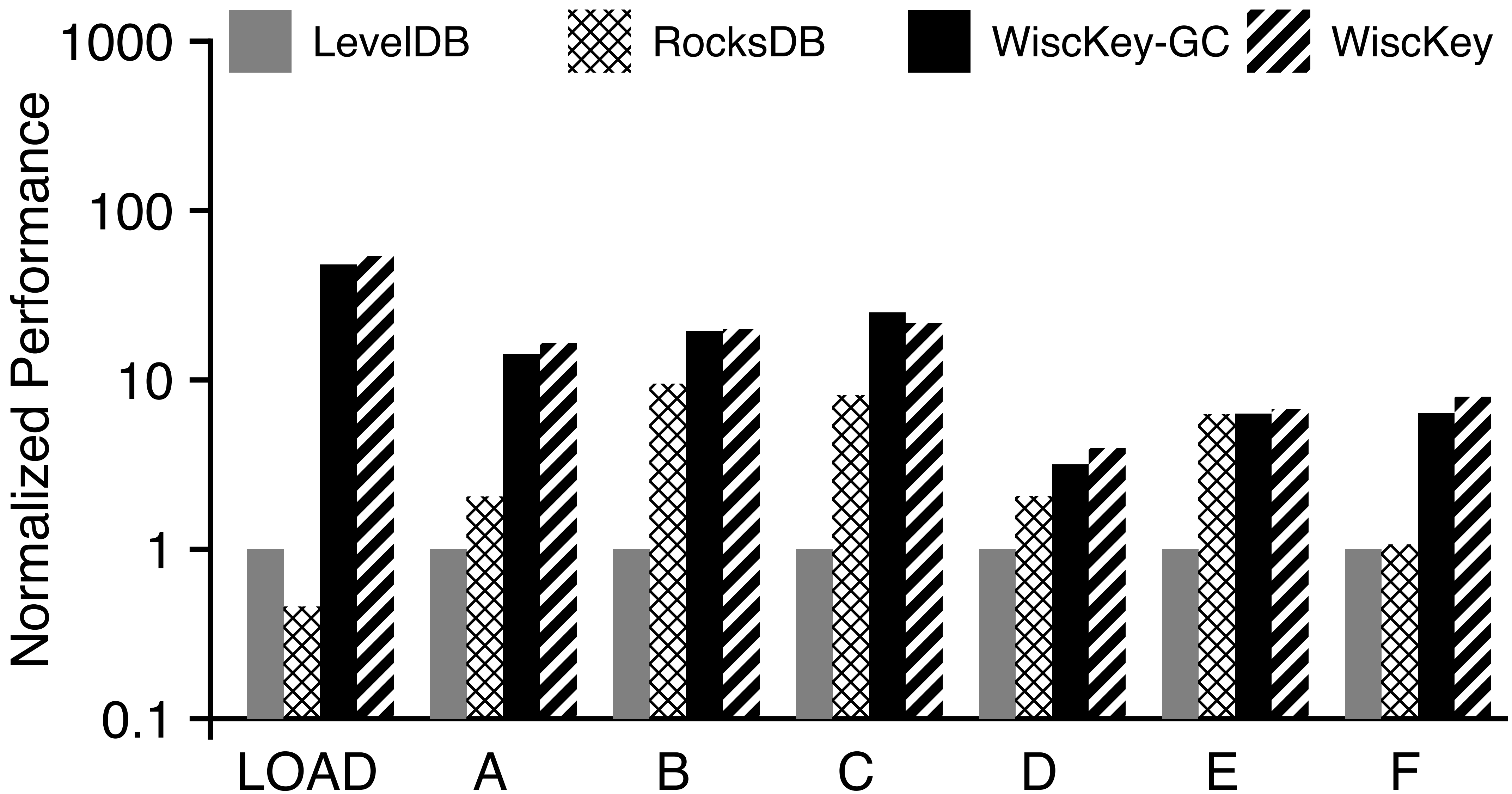
Key-Value Separation

Challenges and Optimizations

Evaluation

Conclusion

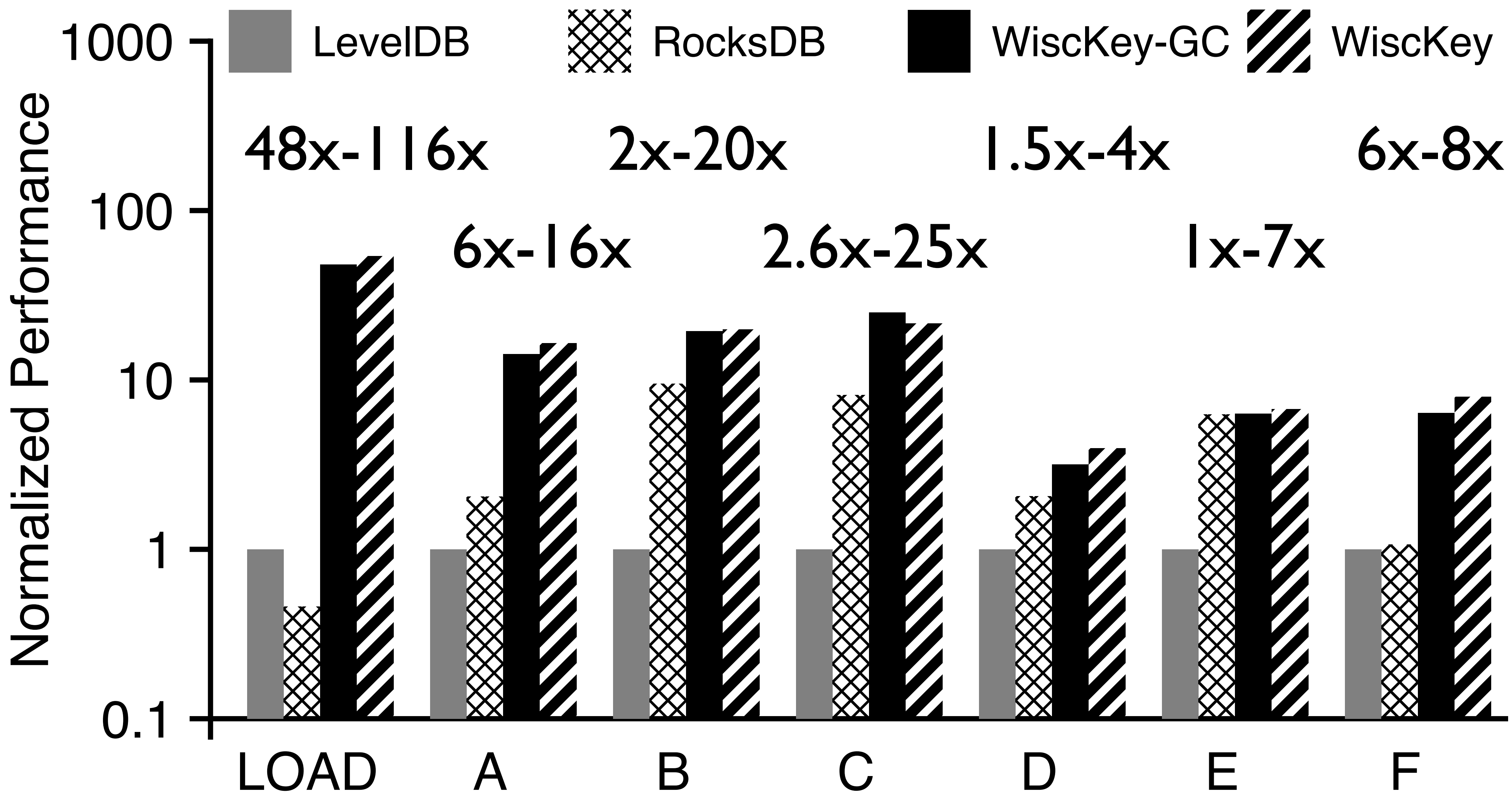
YCSB Benchmarks



A	B	C	D	E	F
50% R 50% U	95% R 5% U	100% R	95% R 5% I	95% S 5% I	50% R 50% RMW

Key size: 16B, Value size: 1KB

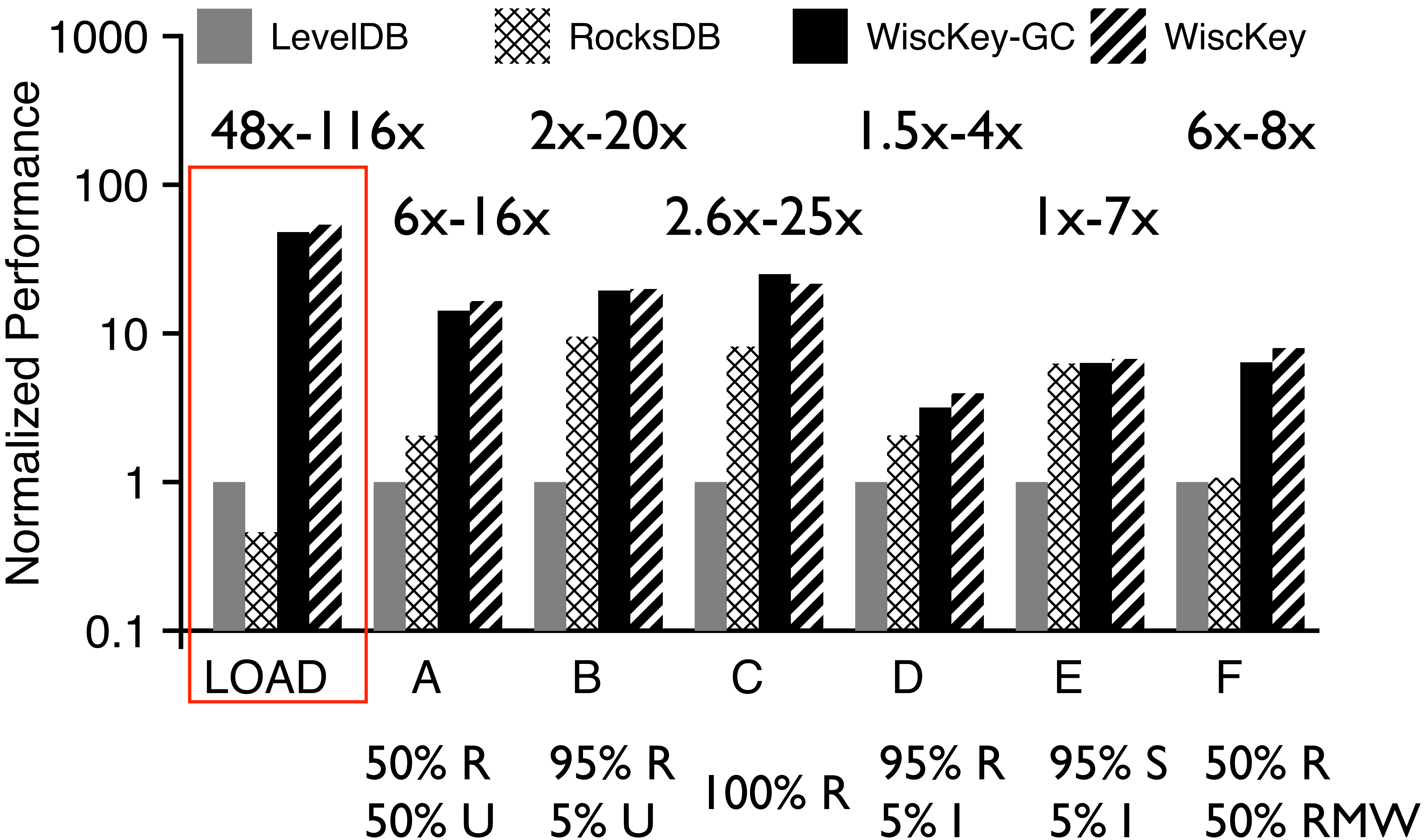
YCSB Benchmarks



A	B	C	D	E	F
50% R 50% U	95% R 5% U	100% R	95% R 5% I	95% S 5% I	50% R 50% RMW

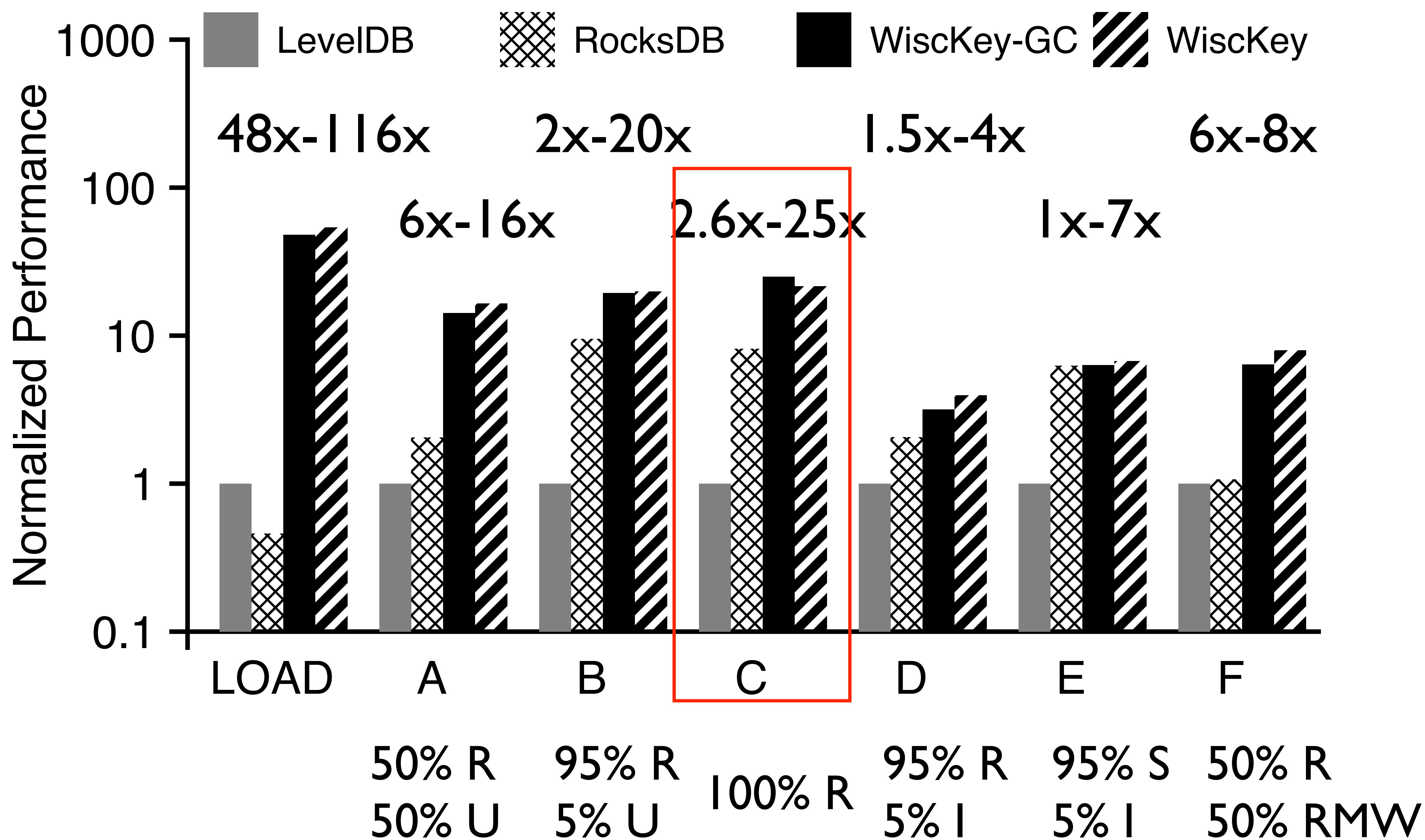
Key size: 16B, Value size: 1KB

YCSB Benchmarks



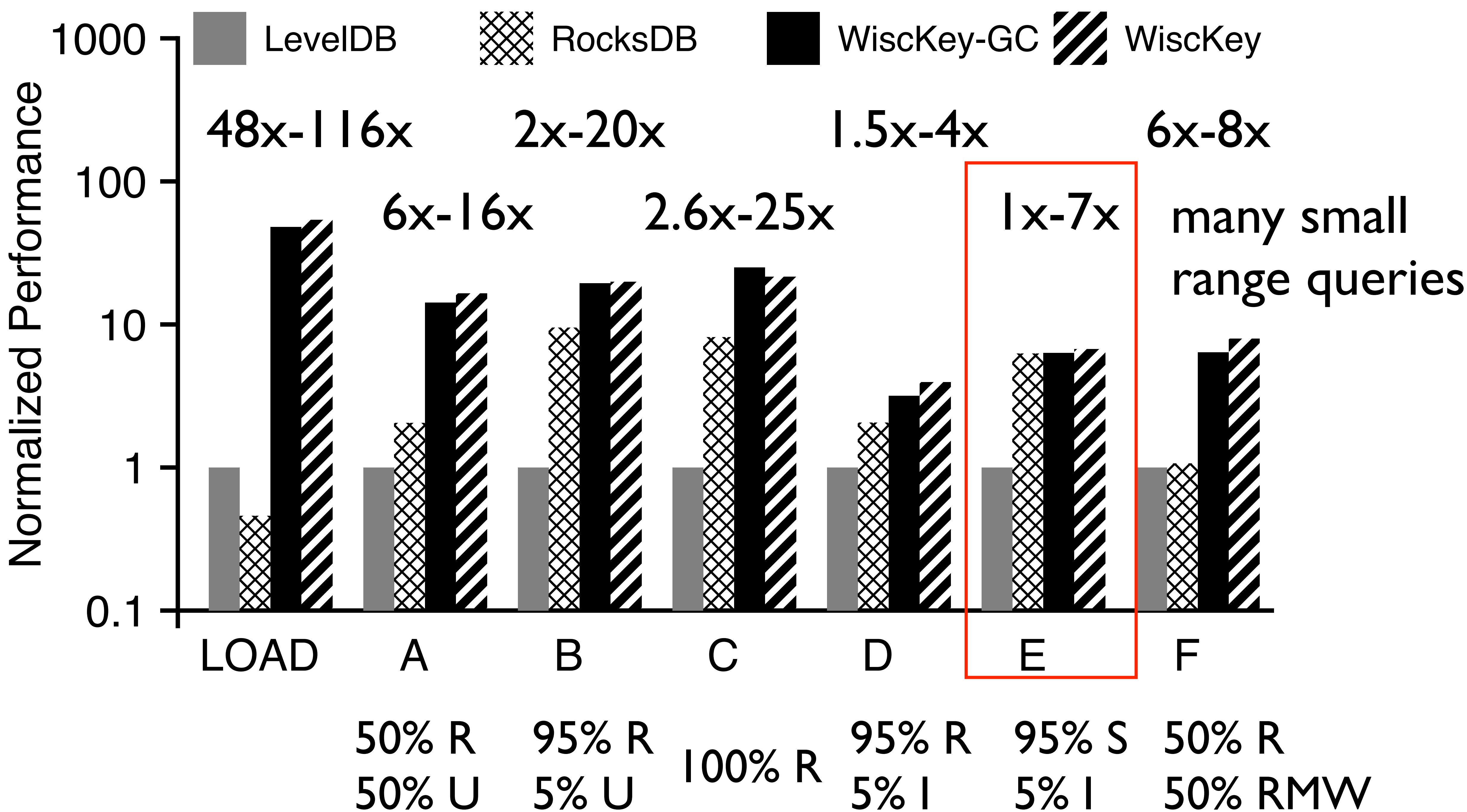
Key size: 16B, Value size: 1KB

YCSB Benchmarks



Key size: 16B, Value size: 1KB

YCSB Benchmarks



Key size: 16B, Value size: 1KB

Conclusion

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

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