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

key

value
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

LSM-tree \rightarrow Value Log
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

LSM-tree  Value Log
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

memory

disk

L0 (8MB)

L1 (10MB)

L2 (100MB)

L6 (ITB)

Log
LSM-trees: Insertion
LSM-trees: Insertion

memory
disk
LevelDB

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (1TB)

KV

Log
LSM-trees: Insertion

- Memory
- Disk
  - L0 (8MB)
  - L1 (10MB)
  - L2 (100MB)
  - L6 (ITB)

KV
Log
LSM-trees: Insertion

- **memory**
- **disk**
  - **LevelDB**
    - L0 (8MB)
    - L1 (10MB)
    - L2 (100MB)
    - L6 (ITB)

- **Log**
- **KV**
  - memT
LSM-trees: Insertion

memory

disk

LevelDB

L0 (8MB)

L1 (10MB)

L2 (100MB)

L6 (1TB)

Log

KV

1

2

3
LSM-trees: Insertion

LevelDB

memory

disk

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (ITB)

KV

Log

1
2
3
4
LSM-trees: Insertion

memory

disk

LevelDB

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (ITB)

memT

KV

Log

1

2

3

4

5
LSM-trees: Insertion

1. Write sequentially  
2. Sort data for quick lookups

LevelDB

memory

disk

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (1TB)
LSM-trees: Insertion

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

LevelDB

- L0 (8MB)
- L1 (10MB)
- L2 (100MB)
- L6 (ITB)

Memory

- memT

Disk

- Log
LSM-trees: Lookup

memory

disk

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (1TB)
LSM-trees: Lookup

LevelDB

Level 0 (8MB)

Level 1 (10MB)

Level 2 (100MB)

Level 6 (ITB)

Log

Memory and disk structure:

- L0 (8MB)
- L1 (10MB)
- L2 (100MB)
- L6 (1TB)
LSM-trees: Lookup

K

memory

disk

LevelDB

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (1TB)

Log
LSM-trees: Lookup
LSM-trees: Lookup

memory

disk

LevelDB

L0 (8MB)
L1 (10MB)
L2 (100MB)

Log

K

memT

1

2
LSM-trees: Lookup

memory

disk

LevelDB

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (ITB)

memT → K

Log

1

2
LSM-trees: Lookup

memory

disk

LevelDB

L0 (8MB)
L1 (10MB)
L2 (100MB)
L6 (1TB)

K

memT

1

2

3 L1 to L6

Log

1. Lookup in memory
2. Lookup in disk
3. Lookup in disk (L1 to L6)
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

Amplification Ratio

Write
Read

100 GB

1000
100
10
1

14
327
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

Amplification Ratio

1000
100
10
1

Write
Read

14
327

100 GB
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

SSD device

LSM-tree

Value Log
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

Throughput (MB/s)
Random Load

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

Throughput (MB/s)

LevelDB
WiscKey

load 100 GB database

64B 256B 1KB 4KB 16KB 64KB 256KB
Random Load

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

Throughput (MB/s)

0 50 100 150 200 250 300 350 400 450 500

64B 256B 1KB 4KB 16KB 64KB 256KB

- LevelDB
- WiscKey

Only 2 MB/s to 4.1 MB/s

Load 100 GB database
Random Load

Throughput (MB/s)

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

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

LevelDB

WiscKey
Random Load

Only 2 MB/s to 4.1 MB/s

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

Throughput (MB/s)

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

LevelDB

WiscKey

Load 100 GB database

Large write amplification (12 to 16) in LevelDB

Only 2 MB/s to 4.1 MB/s

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

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

- Only 2 MB/s to 4.1 MB/s

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

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

Load 100 GB database

Large write amplification (12 to 16) in LevelDB

Throughput (MB/s)

64B 256B 1KB 4KB 16KB 64KB 256KB

0 50 100 150 200 250 300 350 400 450 500
<table>
<thead>
<tr>
<th>limits of files</th>
<th>num of files</th>
</tr>
</thead>
<tbody>
<tr>
<td>L0</td>
<td>9</td>
</tr>
<tr>
<td>L1 (5)</td>
<td>30</td>
</tr>
<tr>
<td>L2 (50)</td>
<td>365</td>
</tr>
<tr>
<td>L3 (500)</td>
<td>2184</td>
</tr>
<tr>
<td>L4 (5000)</td>
<td>15752</td>
</tr>
<tr>
<td>L5 (50000)</td>
<td>23733</td>
</tr>
<tr>
<td>L6 (500000)</td>
<td>0</td>
</tr>
<tr>
<td>LevelDB</td>
<td>limits of files</td>
</tr>
<tr>
<td>--------</td>
<td>----------------</td>
</tr>
<tr>
<td>L0</td>
<td>9</td>
</tr>
<tr>
<td>L1 (5)</td>
<td>30</td>
</tr>
<tr>
<td>L2 (50)</td>
<td>365</td>
</tr>
<tr>
<td>L3 (500)</td>
<td>2184</td>
</tr>
<tr>
<td>L4 (5000)</td>
<td>15752</td>
</tr>
<tr>
<td>L5 (50000)</td>
<td>23733</td>
</tr>
<tr>
<td>L6 (500000)</td>
<td>0</td>
</tr>
<tr>
<td>limits of files</td>
<td>LevelDB num of files</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>L0</td>
<td>9</td>
</tr>
<tr>
<td>L1 (5)</td>
<td>30</td>
</tr>
<tr>
<td>L2 (50)</td>
<td>365</td>
</tr>
<tr>
<td>L3 (500)</td>
<td>2184</td>
</tr>
<tr>
<td>L4 (5000)</td>
<td>15752</td>
</tr>
<tr>
<td>L5 (50000)</td>
<td>23733</td>
</tr>
<tr>
<td>L6 (500000)</td>
<td>0</td>
</tr>
<tr>
<td>limits of files</td>
<td>LevelDB</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td>L0</td>
<td>9</td>
</tr>
<tr>
<td>L1 (5)</td>
<td>30</td>
</tr>
<tr>
<td>L2 (50)</td>
<td>365</td>
</tr>
<tr>
<td>L3 (500)</td>
<td>2184</td>
</tr>
<tr>
<td>L4 (5000)</td>
<td>15752</td>
</tr>
<tr>
<td>L5 (50000)</td>
<td>23733</td>
</tr>
<tr>
<td>L6 (500000)</td>
<td>0</td>
</tr>
</tbody>
</table>

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

Throughput (MB/s)

LevelDB, WiscKey

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

100,000 lookups on a randomly loaded 100 GB database

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

100,000 lookups on a randomly loaded 100 GB database

large read amplification in LevelDB

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

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

100,000 lookups on a randomly loaded 100 GB database

large read amplification in LevelDB

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

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

Throughput (MB/s)

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

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

Throughput (MB/s)

LevelDB-Rand

WiscKey-Rand
Range Query

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

read 4GB from a randomly loaded 100 GB database
Range Query

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

LevelDB-Rand
WiscKey-Rand

read 4GB from a randomly loaded 100 GB database

For large kv pairs, WiscKey can perform better
Range Query

Throughput (MB/s)

- LevelDB-Rand
- WiscKey-Rand

WiscKey is limited by SSD’s parallel random read performance.

For large kv pairs, WiscKey can perform better.

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

read 4GB from a randomly loaded 100 GB database.
Range Query

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.

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

read 4GB from a randomly loaded 100 GB database
Range Query

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

Throughput (MB/s)

- LevelDB-Rand
- WiscKey-Rand
- LevelDB-Seq
- WiscKey-Seq
Range Query

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

Throughput (MB/s)

LevelDB-Rand
WiscKey-Rand
LevelDB-Seq
WiscKey-Seq

read 4GB from a sequentially loaded 100 GB database
Range Query

![Graph showing throughput (MB/s) for different key sizes and values.]

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

**Throughput (MB/s)**

- **LevelDB-Rand**
- **LevelDB-Seq**
- **WiscKey-Rand**
- **WiscKey-Seq**

Both WiscKey and LevelDB read sequentially.

Read 4GB from a sequentially loaded 100 GB database.
Range Query

Sorted databases help WiscKey’s range query

Throughput (MB/s)

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

LevelDB-Rand
LevelDB-Seq
WiscKey-Rand
WiscKey-Seq

read 4GB from a sequentially loaded 100 GB database

Both WiscKey and LevelDB read sequentially

Sorted databases help WiscKey’s range query
Optimizations
Optimizations

SSD device

LSM-tree

Value Log
Optimizations

Online and light-weight garbage collection
- append (ksize, vsize, key, value) in value log

SSD device

<table>
<thead>
<tr>
<th>k, addr</th>
<th>value</th>
<th>value</th>
<th>value</th>
</tr>
</thead>
</table>

LSM-tree

Value Log
Optimizations

Online and light-weight garbage collection
- append \((ksize, vsize, key, value)\) in value log

SSD device

- LSM-tree
- Value Log
Optimizations

Online and light-weight garbage collection
- append \((ksize, vsize, key, value)\) in value log

SSD device

<table>
<thead>
<tr>
<th>k, addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>tail</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ksize, vsize, key, value</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
</tr>
</tbody>
</table>

LSM-tree

Value Log
Optimizations

Online and light-weight garbage collection
- append (ksize, vsize, key, value) in value log

Remove LSM-tree log in WiscKey

SSD device

LSM-tree

Value Log
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

SSD device

LSM-tree

Value Log
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 \((k, addr, v)\) 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

Key size: 16B, Value size: 1KB
YCSB Benchmarks

LevelDB    RocksDB    WiscKey-GC    WiscKey

Normalized Performance

48x-116x  2x-20x  1.5x-4x  6x-8x

6x-16x  2.6x-25x  1x-7x

LOAD  A  B  C  D  E  F

50% R 95% R 100% R 95% R 95% S 50% R
50% U 95% U 5% U 5% I 5% I 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