SILT: A Memory-Efficient, High-Performance Key-Value Store

Hyeontaek Lim, Bin Fan, David G. Andersen
Michael Kaminsky†

Carnegie Mellon University
†Intel Labs

2011-12-08
Key-Value Store

Clients

<table>
<thead>
<tr>
<th>PUT(key, value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>value = GET(key)</td>
</tr>
<tr>
<td>DELETE(key)</td>
</tr>
</tbody>
</table>

Key-Value Store Cluster

- E-commerce (Amazon)
- Web server acceleration (Memcached)
- Data deduplication indexes
- Photo storage (Facebook)
Flash-Based Key-Value Stores

• Many key-value store projects have examined **flash memory** as main storage
  – Faster than disk, cheaper than DRAM
  – 5+ papers in well-known systems/DB/networking conferences in the past 3 years
  – Several commercial products

• Frequently appearing design question: **reducing DRAM consumption** (why?)
Need for In-Memory Indexing

• Random flash reads/sec = 48,000 (1 SATA SSD)
  – Fast... enough?
  – E.g., Memcached (DRAM-only key-value store) achieves over 300,000 queries per second

• **DRAM** is used to **index** (locate) items on flash for efficient flash I/O
Small Data, Large Index?

• 1 TB of data to store on flash
• 4 bytes of DRAM per key-value pair (previous state-of-the-art)

Total index size = 1 TB / (key-value pair size) x 4 bytes
Scarcer DRAM in “Wimpy” Nodes

1.6 GHz Dual-core Atom
32-160 GB Flash SSD
Only 1 GB DRAM!
Three Metrics to Minimize

**Memory overhead** = Index size per entry
- Ideally 0 bytes/entry (no memory overhead)

**Read amplification** = Flash reads per query
- Limits query throughput
- Ideally 1 (no wasted flash reads)

**Write amplification** = Flash writes per entry
- Limits insert throughput
- Also reduces flash life expectancy
  - Must be small enough for flash to last a few years
Landscape: Where We Were

Read amplification

Memory overhead (bytes/entry)

- SkimpyStash (2011)
- HashCache (2009)
- BufferHash (2010)
- FlashStore (2010)
- FAWN-DS (2009)
Seesaw Game?

How can we improve?

SkimpyStash

Memory efficiency

High performance

FAWN-DS
FlashStore
HashCache
BufferHash
“SILT” Preview: (1) Three Stores with (2) New Index Data Structures

Queries look up stores in sequence (from right to left)

Writes only go to this store

Data are moved in background

Memory efficient

Bridging two extremes

Write friendly

Memory

Flash
LogStore: No Control over Data Layout

Naive Hashtable (48+ B/entry)
SILT Log Index (6.5+ B/entry)

Still need pointers: size $\geq \log N$ bits/entry (3~4 bytes in practice)

Memory overhead
6.5+ bytes/entry

Write amplification
1
SortedStore: Space-Optimized Layout

SILT Sorted Index (0.4 B/entry)

On-flash sorted array

Need to perform bulk-insert to amortize write amplification

Memory overhead
0.4 bytes/entry

Write amplification
High
Combining SortedStore and LogStore

Merge: (1) Sort log  (2) Sequentially merge two sorted data
Achieving both Low Memory Overhead and Low Write Amplification

Now we can achieve simultaneously:

Write amplification = 5.4 = 3 year flash life
Memory overhead = 1.3 B/entry

With “HashStores”, memory overhead = 0.7 B/entry!
SILT’s Design (Recap)

**<SortedStore>**
- SILT Sorted Index
  - On-flash sorted array

**<HashStore>**
- SILT Filters
  - On-flash hashtables

**<LogStore>**
- SILT Log Index
  - On-flash log

**Memory overhead**
- 0.7 bytes/entry

**Read amplification**
- 1.01

**Write amplification**
- 5.4
Landscape: Where We Are

Read amplification

Memory overhead (bytes/entry)

SILT

SkimpyStash (2011)
HashCache (2009)
BufferHash (2010)
FlashStore (2010)
FAWN-DS (2009)
Evaluation

1. Various combinations of indexing schemes
2. Background operations (merge/conversion)

<table>
<thead>
<tr>
<th>Experiment Setup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2.80 GHz (4 cores)</td>
</tr>
<tr>
<td>Flash drive</td>
<td>SATA 256 GB</td>
</tr>
<tr>
<td></td>
<td>(48 K random 1024-byte reads/sec)</td>
</tr>
<tr>
<td>Workload size</td>
<td>20-byte key, 1000-byte value, ≥ 50 M keys</td>
</tr>
<tr>
<td>Query pattern</td>
<td>Uniformly distributed (worst for SILT)</td>
</tr>
</tbody>
</table>
LogStore Alone: Too Much Memory

Workload: 90% GET (50-100 M keys) + 10% PUT (50 M keys)
LogStore+SortedStore: Still Much Memory

Workload: 90% GET (50-100 M keys) + 10% PUT (50 M keys)
Full SILT: Very Memory Efficient

Workload: 90% GET (50-100 M keys) + 10% PUT (50 M keys)
Small Impact from Background Operations

Workload: 90% GET (100~ M keys) + 10% PUT

Oops! bursty TRIM requests from ext4 FS
Conclusion

• SILT provides both memory-efficient and high-performance key-value store
  – New items are put into a write-friendly store; later migrated to a memory-efficient store
  – Two new compact index data structures

• Full source code is available
  – https://github.com/silt/silt