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

Presenter:
Andrzej Findeisen

Based on paper of H. Lim, B. Fan, D.G. Andersen, and M. Kaminsky:
“SILT: A Memory-Efficient, High-Performance Key-Value Store”
in Proceedings ACM SOSP 2011

Introduction

Abstract

- SILT = Small Index Large Table
- System for storing key-value pairs
- Extremely efficient in terms of memory and flash disk usage
  - 0.7 bytes of DRAM per entry
  - 1.01 flash reads per entry retrieval
  - Writing optimized for flash specific characteristics
- Assumption: uniformly distributed 160-bit keys

Rationale

Metric	2008 → 2011	Increase
CPU transistors	731 → 1,170 M	60 %
DRAM capacity	0.062 → 0.153 GB/$	147 %
Flash capacity	0.134 → 0.428 GB/$	219 %
Disk capacity	4.92 → 15.1 GB/$	207 %

Table 1: From 2008 to 2011, flash and hard disk capacity increased much faster than either CPU transistor count or DRAM capacity.

Moreover DRAM is:

- 8 times more expensive than flash disk
- 25 times power consuming than flash disk

Comparison

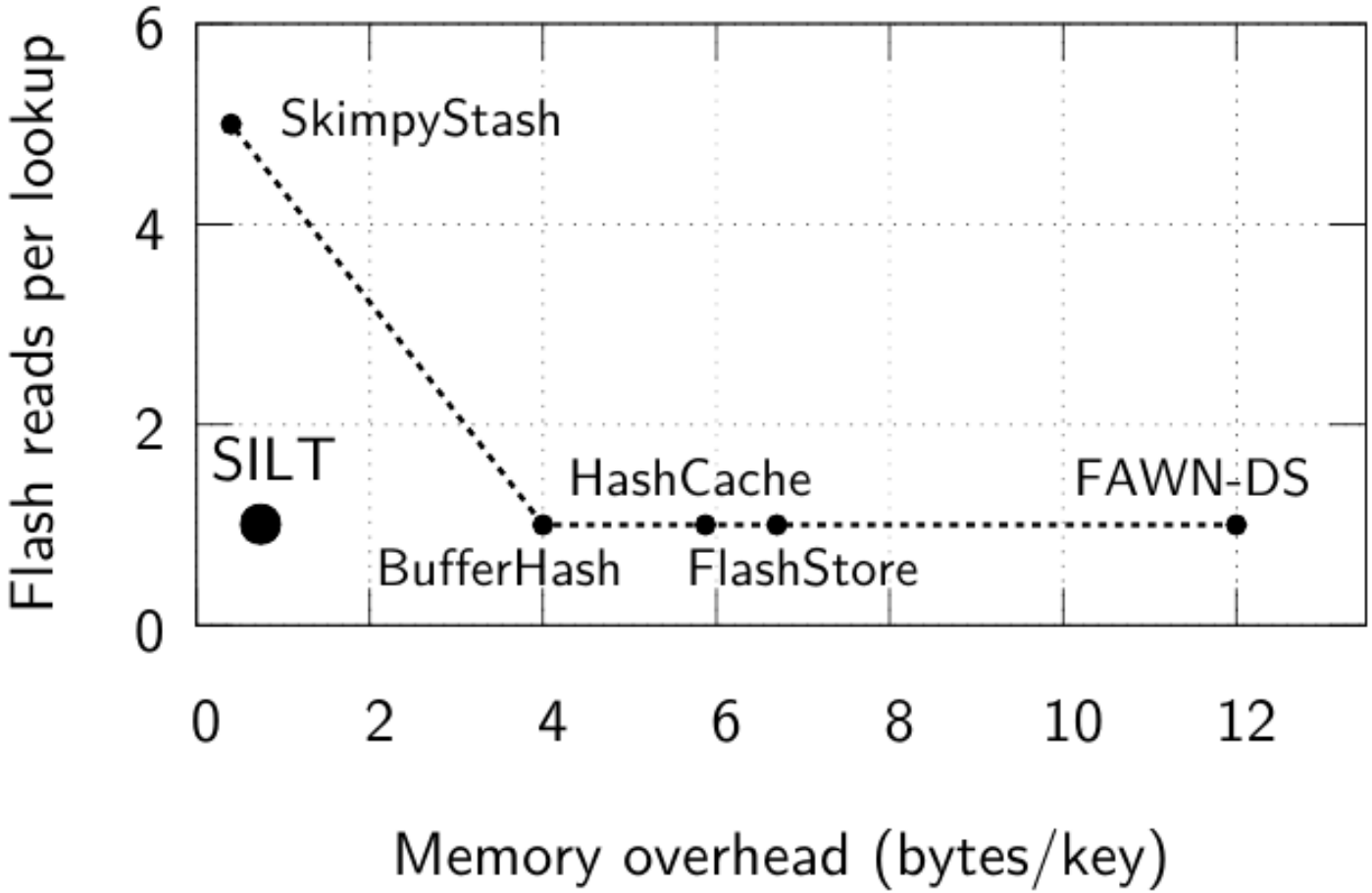


Figure 1: The memory overhead and lookup performance of SILT and the recent key-value stores. For both axes, smaller is better.

Goals

- **Low read amplification:**
  Issue at most 1 + ε flash reads for a single GET where ε is configurable and small (e.g. 0.01).
- **Controllable write amplification and favoring sequential writes:**
  The system should issue flash-friendly, large writes.
- **Memory-efficient indexing:**
  SILT should use as little memory as possible.
- **Computation-efficient indexing:**
  SILT's indexes should be fast enough to let the system saturate the flash I/O.
- **Effective use of flash space:**
  Some data layout options use the flash space more sparsely to improve lookup or insertion speed, but the total space overhead of any such choice should remain small – less than 20% or so.

Design

Architecture

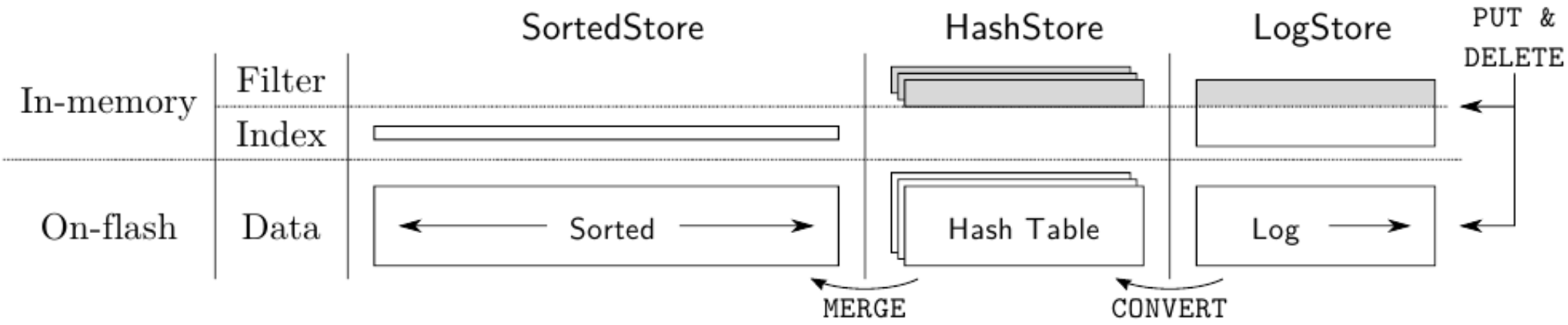


Figure 2: Architecture of SILT.

	SortedStore (§3.3)	**HashStore** (§3.2)	**LogStore** (§3.1)
Mutability	Read-only	Read-only	Writable
Data ordering	Key order	Hash order	Insertion order
Multiplicity	1	≥ 0	1
Typical size	$> 80\%$ of total entries	$< 20\%$	$< 1\%$
DRAM usage	0.4 bytes/entry	2.2 bytes/entry	6.5 bytes/entry

Table 2: Summary of basic key-value stores in SILT.

LogStore

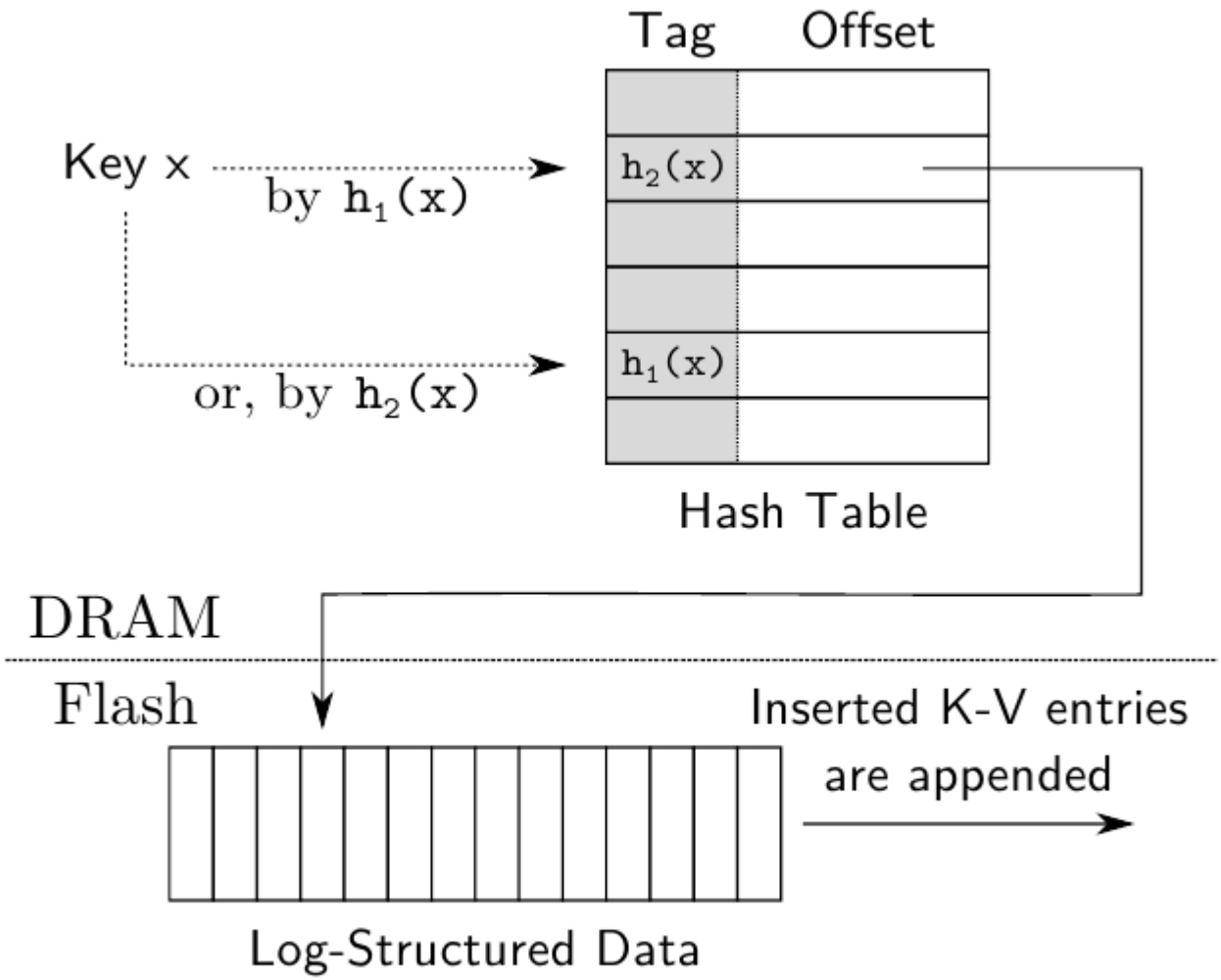


Figure 3: Design of LogStore: an in-memory cuckoo hash table (index and filter) and an on-flash data log.

HashStore

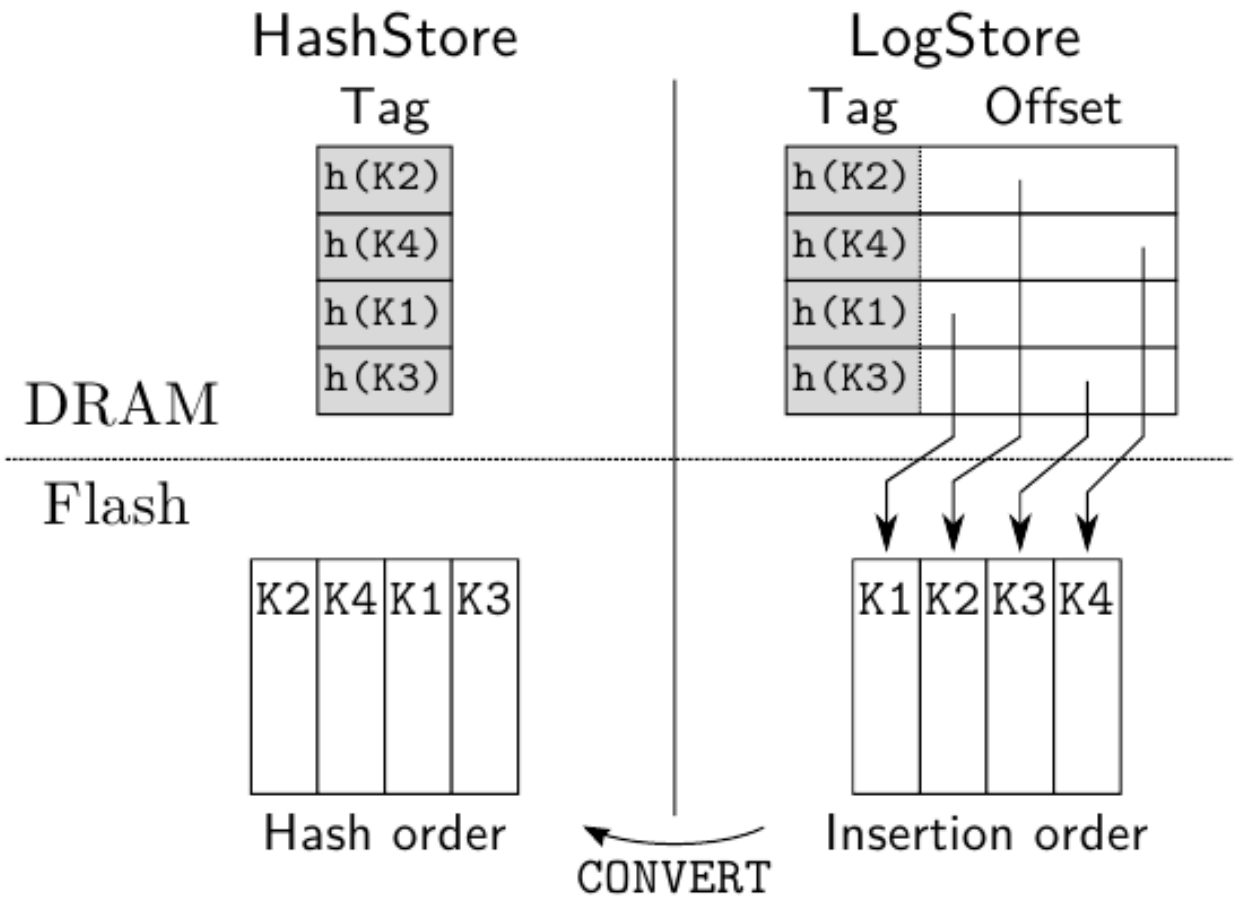


Figure 4: Convert a LogStore to a HashStore. Four keys K1, K2, K3, and K4 are inserted to the LogStore, so the layout of the log file is the insert order; the in-memory index keeps the offset of each key on flash. In HashStore, the on-flash data forms a hash table where keys are in the same order as the in-memory filter.

SortedStore I

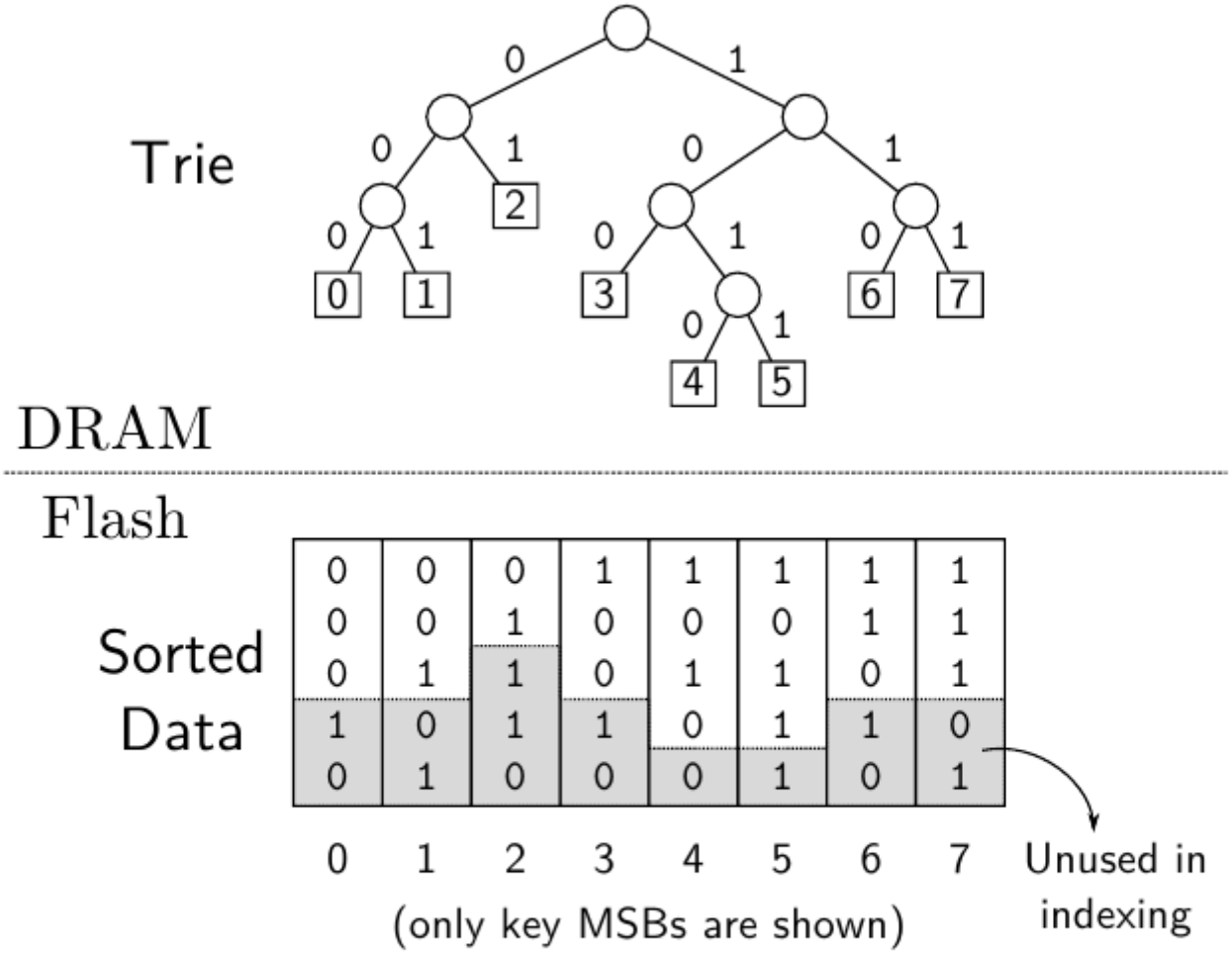


Figure 5: Example of a trie built for indexing sorted keys. The index of each leaf node matches the index of the corresponding key in the sorted keys.

SortedStore II

Trie representation

- Repr(T) := |L| Repr(L) Repr(R)
- Entropy coding
- Worst case: scanning whole trie
  - Therefore 2^k buckets are introduced (ex. k = 10 for 2^{16} items)

```
# @param  key    lookup key
# @param  trepr  trie representation
# @return        index of the key
#                in the original array
def lookup(key, trepr):
  (thead, ttail) = (trepr[0], trepr[1:])
  if thead == -1:
    return 0
  else:
    if key[0] == 0:
      # Recurse into the left subtrie
      return lookup(key[1:], ttail)
    else:
      # Skip the left subtrie
      ttail = discard_subtrie(ttail)
      # Recurse into the right subtrie
      return thead + lookup(key[1:], ttail)

# @param  trepr  trie representation
# @return        remaining trie representation
#                with the next subtrie consumed
def discard_subtrie(trepr):
  (thead, ttail) = (trepr[0], trepr[1:])
  if thead == -1:
    return ttail
  else:
    # Skip both subtries
    ttail = discard_subtrie(ttail)
    ttail = discard_subtrie(ttail)
    return ttail
```

Algorithm 2: Key lookup on a trie representation.

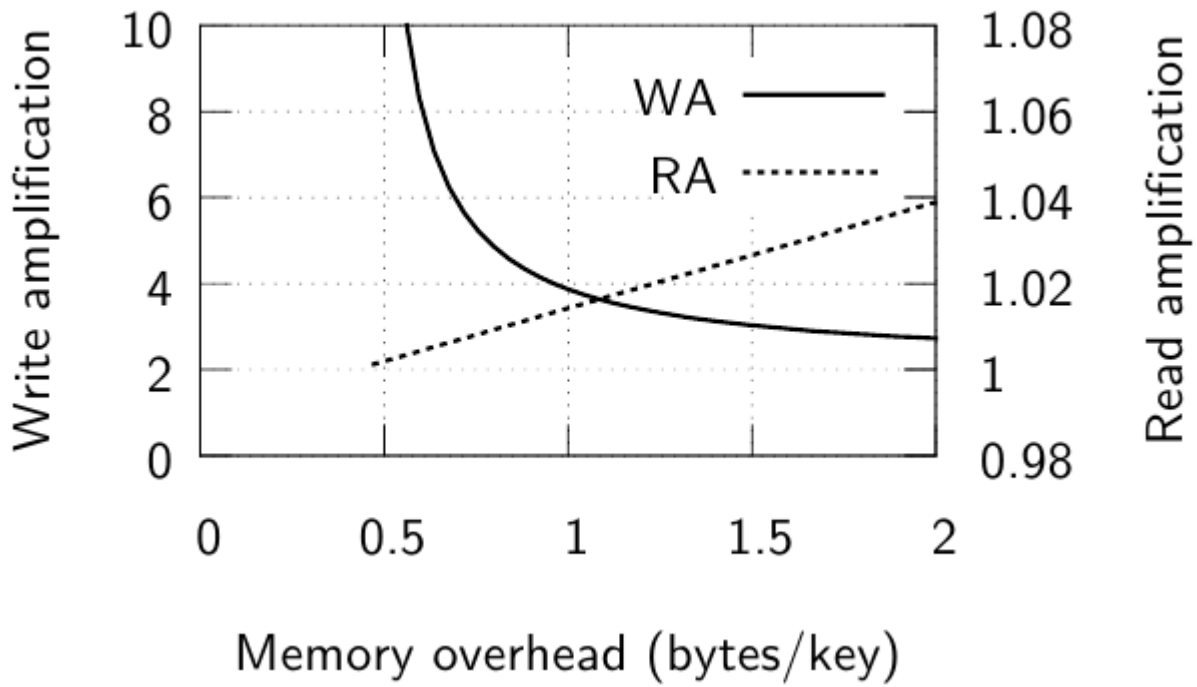
Performance

Overheads

$$\mathrm{WA} = \frac{\text{data written to flash}}{\text{data written by application}},$$

$$\mathrm{RA} = \frac{\text{data read from flash}}{\text{data read by application}},$$

$$\mathrm{MO} = \frac{\text{total memory consumed}}{\text{number of items}}.$$



Flash disk usage

- Data written to flash in bigger chunks
- Example
  - 256 GB disk, 10k erase cycles, 100M items, 4 SILT instances, 7.5M entries to merge, 1KiB entries, 5k updates per second
  - WA = 5.4, 31 of HashStores (RA = 1.008), 102.4GB of data, 73MB of DRAM
  - Device will last 3 years

Memory usage and throughput

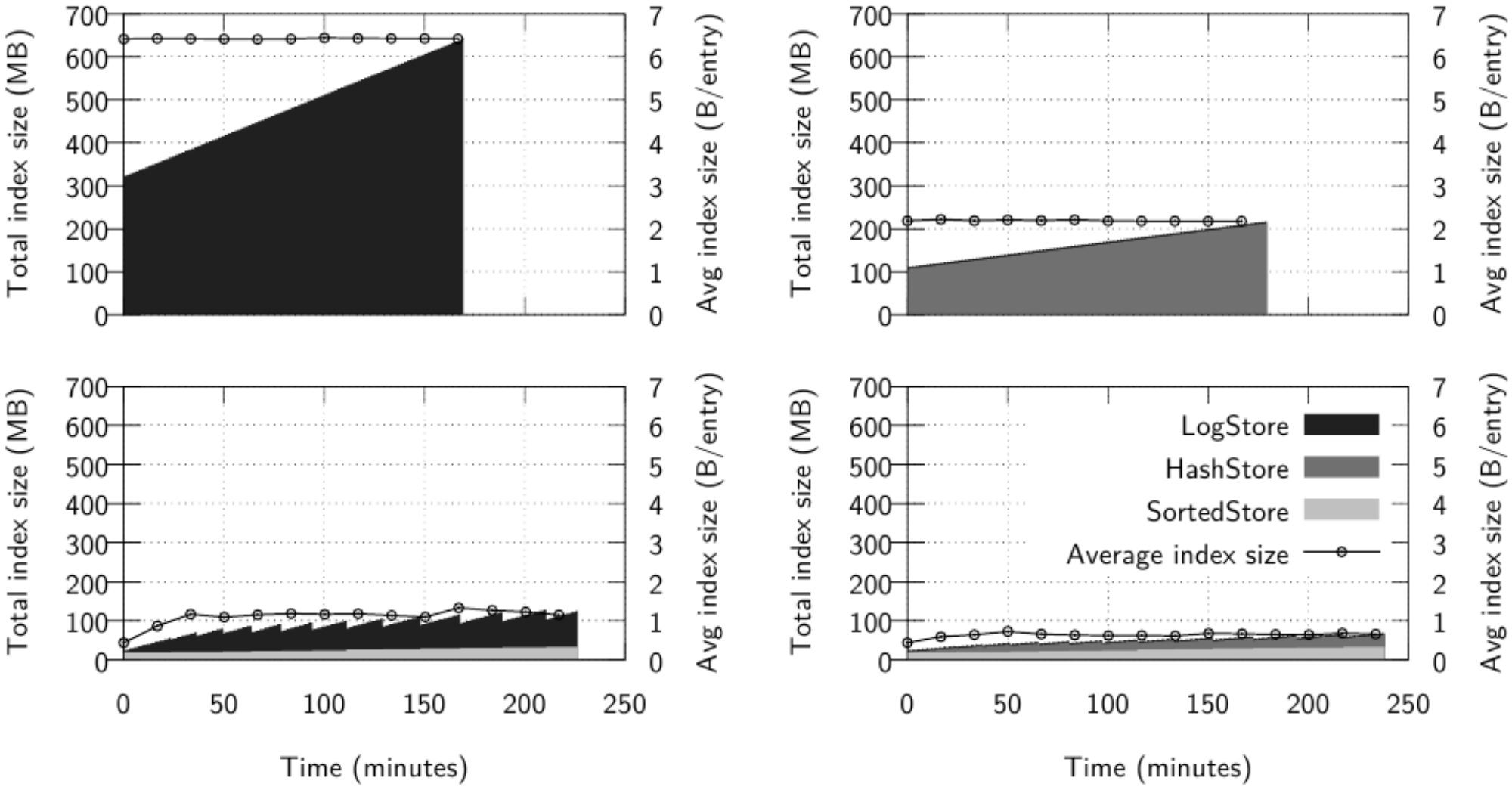


Figure 10: Index size changes for four different store combinations while inserting new 50 M entries.

Latency

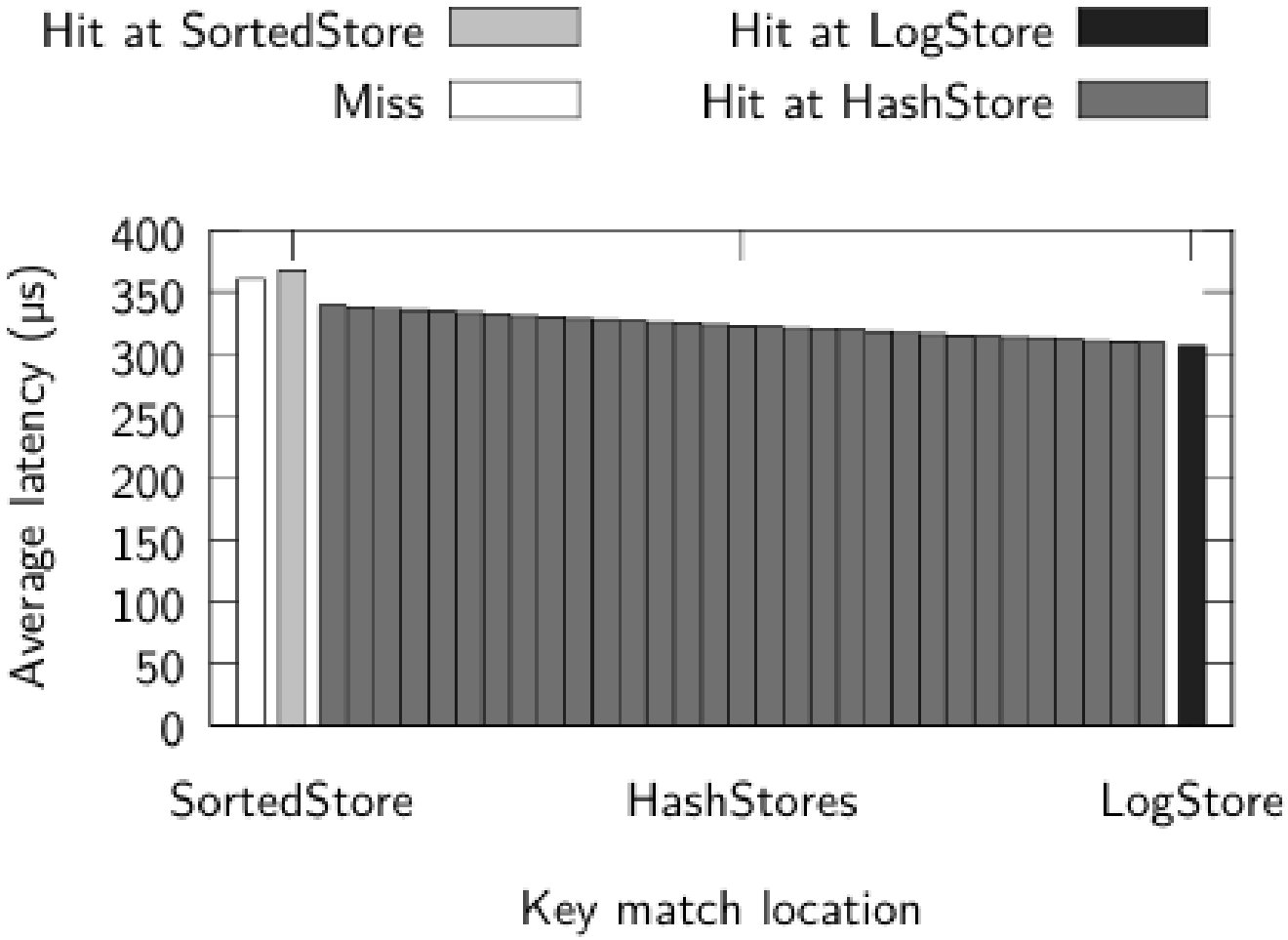


Figure 11: `GET` query latency when served from different store locations.

Throughput

Type	Speed (K keys/s)
LogStore (by `PUT`)	204.6
HashStore (by `CONVERT`)	67.96
SortedStore (by `MERGE`)	26.76

Table 6: Construction performance for basic stores. The construction method is shown in the parentheses.

Type	SortedStore (K ops/s)	HashStore (K ops/s)	LogStore (K ops/s)
`GET` (hit)	46.57	44.93	46.79
`GET` (miss)	46.61	7264	7086

Table 7: Query performance for basic stores that include in-memory and on-flash data structures.

Thank you

Q&A