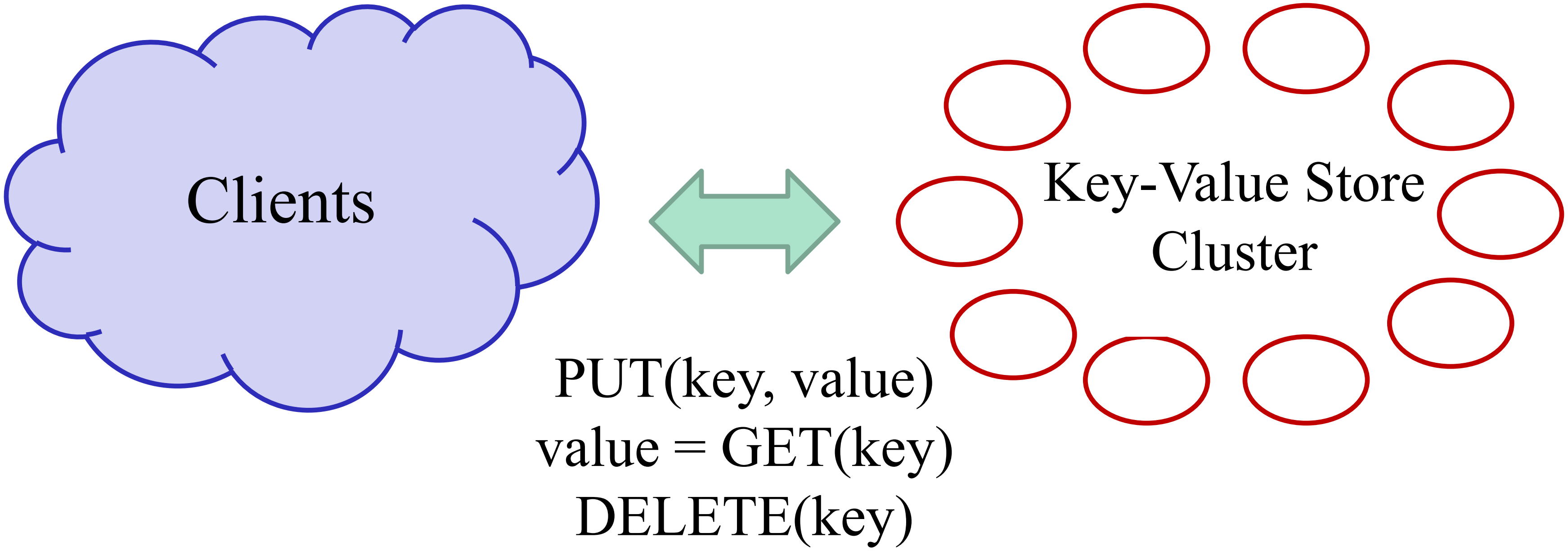
龙星计划课程: 文件系统和分布式数据管理系统

Building File Systems and Distributed Data Management Systems for Performance and Reliability

Lecture 3: Key-Value Data Management Systems

Key-Value Store



PUT(key, value)
value = GET(key)
DELETE(key)

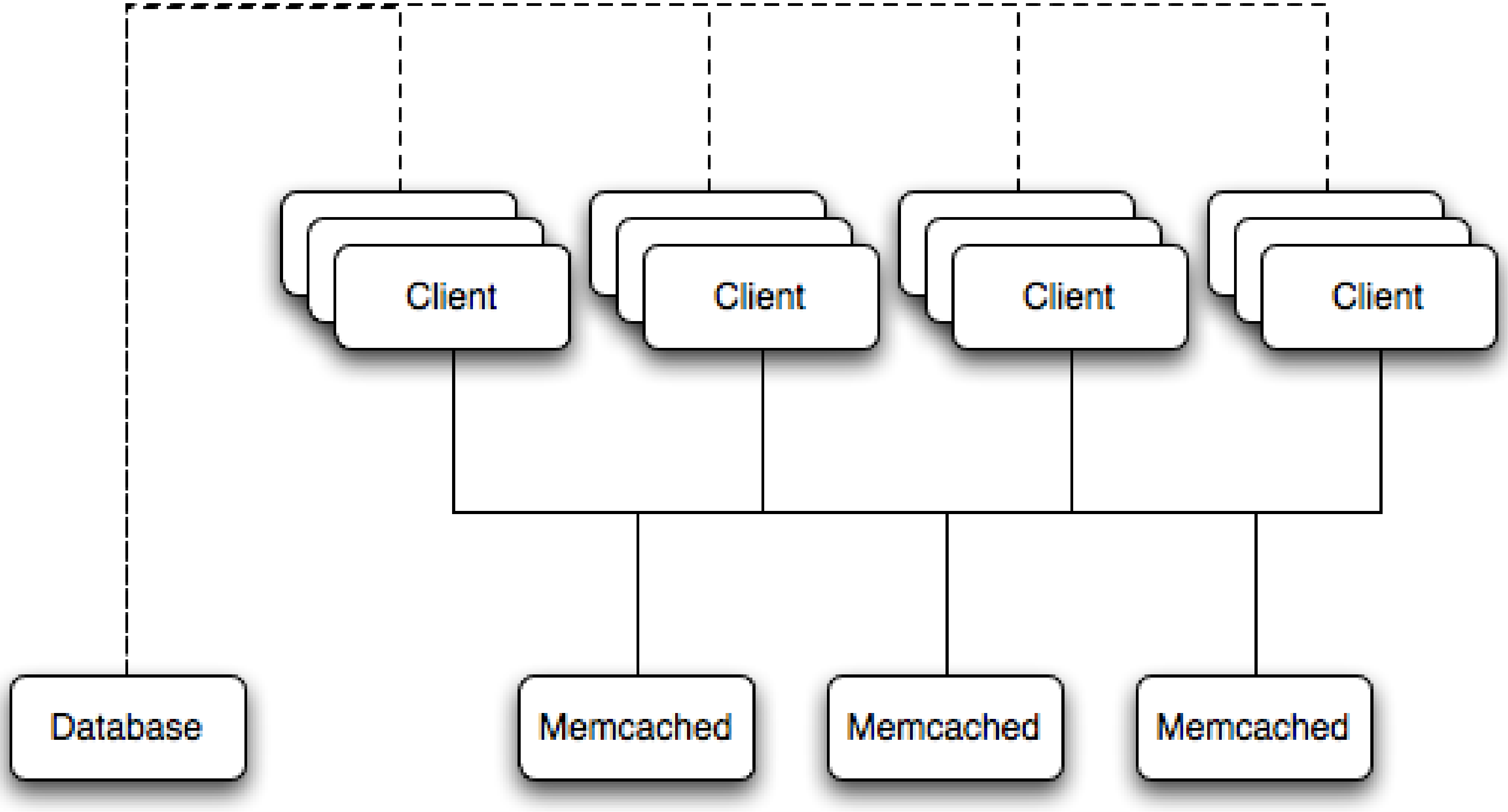
- Dynamo at Amazon
- BigTable (LevelDB) at Google
- Redis at GitHub, Digg, and Blizzard Interactive
- Memcached at Facebook, Zynga and Twitter
- Voldemort at Linkedin

NoSQL DB and KV Store

- ❑ A **NoSQL** or **Not Only SQL** database stores and organizes data differently from the tabular relations used in relational databases.
- ❑ Why NoSQL?
  - ✓ Simplicity of design, horizontal scaling, and finer control over availability.
- ❑ It can be classified as column, document, key-value, and graph store based on their data models.

- ❑ The key-value model is one of the simplest non-trivial data models, and richer data models are often implemented on top of it.
  - ✓ Applications store their data in a schema-less way.
- ❑ Array, linked list, binary search trees, B+ tree, or hash table …?

The First Example: Memcached



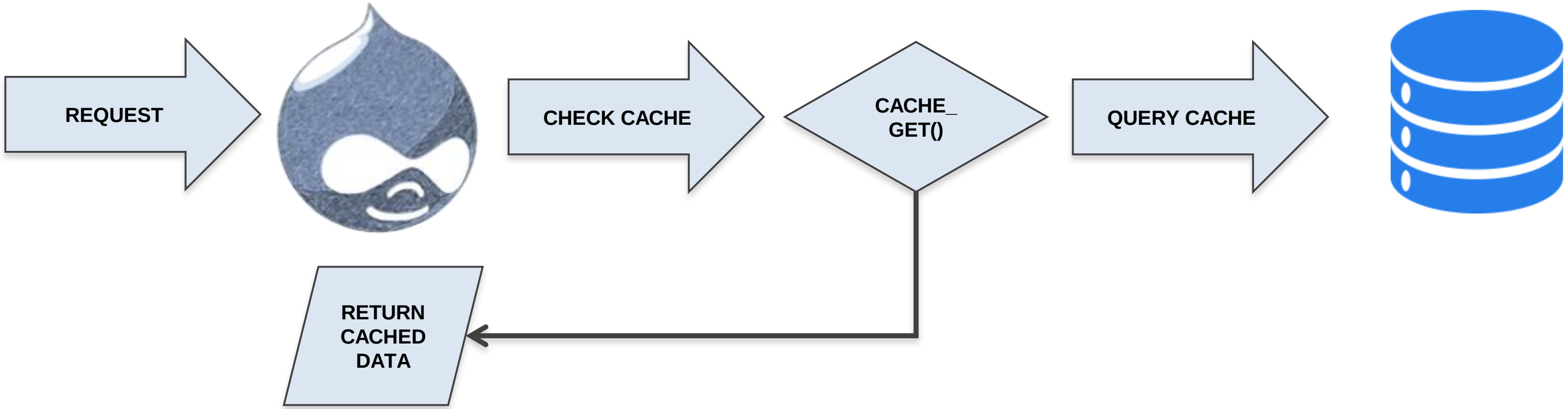
Memcached

- memcached is a high-performance, distributed in-memory object caching system, generic in nature
- It is a key-based cache daemon that stores data and objects wherever dedicated or spare RAM is available for very quick access
- It is a distributed hash table. It doesn’t provide redundancy, failover or authentication. If needed the client has to handle that.

Memcached

- To reduce the load on the database by caching data BEFORE it hits the database
- Can be used for more than just holding database results (objects) and improve the entire application's response time
- Feel the need for speed
  - Memcache is in RAM - much faster then hitting the disk or the database

Memcached



REQUEST

CHECK CACHE

CACHE_ GET()

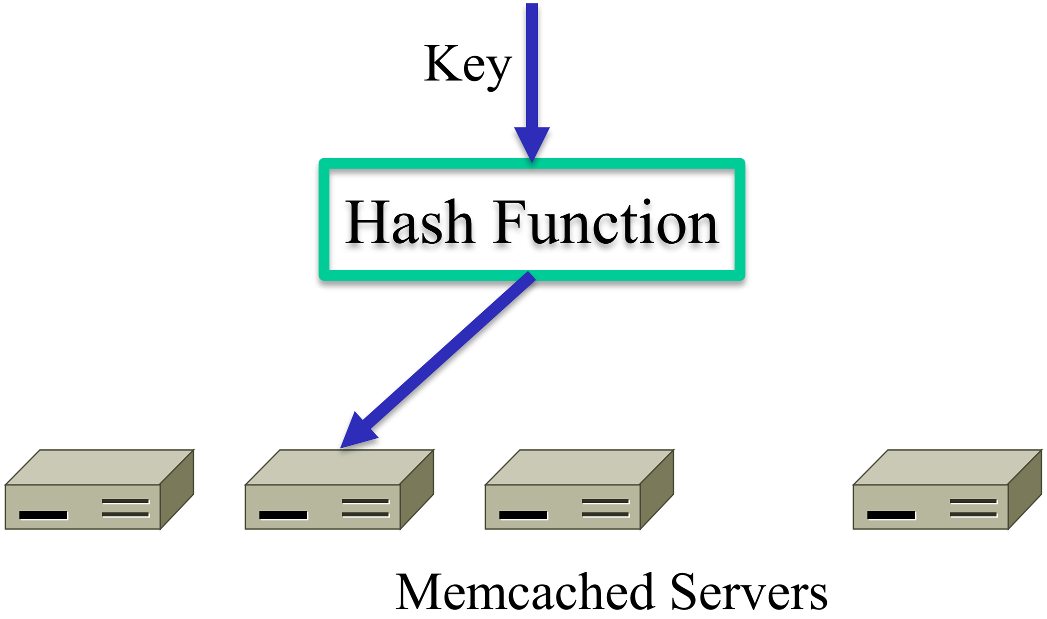
QUERY CACHE

RETURN CACHED DATA

7

Memcached

Distributed memory caching system as a cluser



8

Memcached Workload Pattern at Facebook

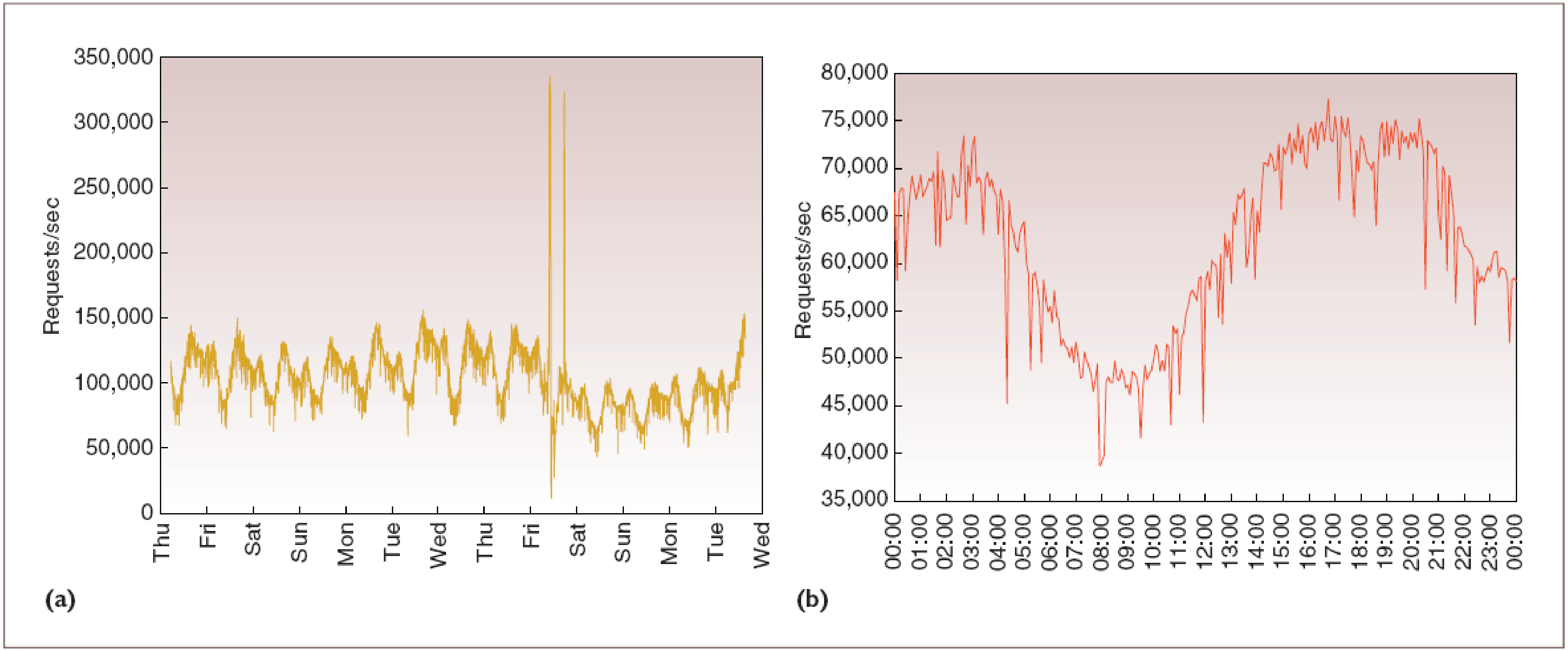


Figure 1. Request rates at different (a) days (USR) and (b) times of day (ETC, in Coordinated Universal Time [UTC]). Each data point counts the total number of requests in the preceding second.

Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, Mike Paleczny. **Workload Analysis of a Large-Scale Key-Value Store** . In *Proceedings of the SIGMETRICS'12*,

Memcached Workload Pattern at Facebook

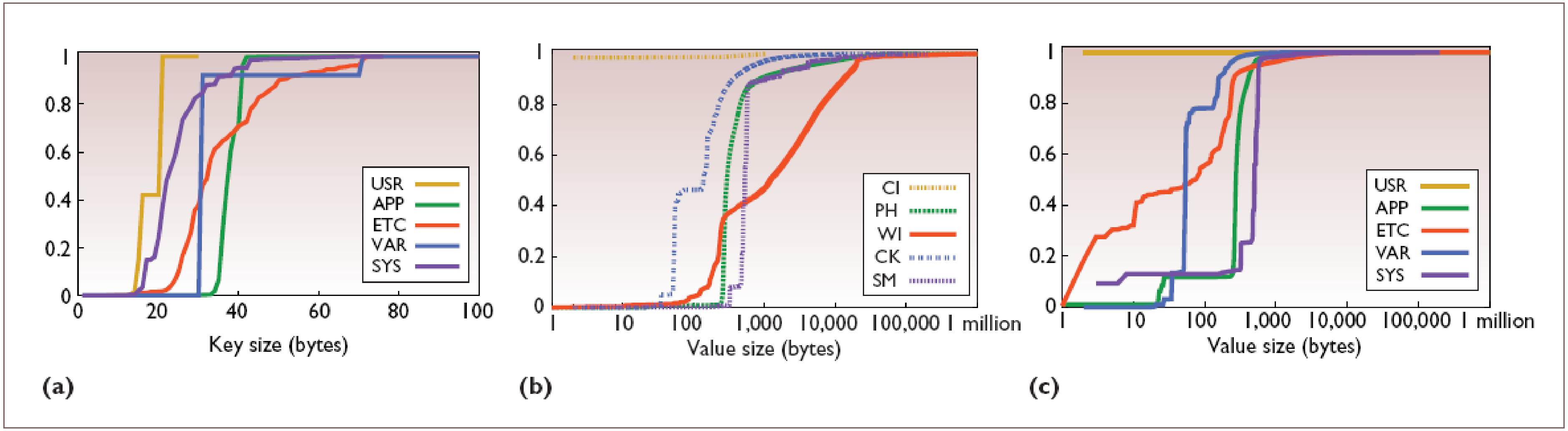


Figure 2. Key and value size distributions for all traces: (a) key size cumulative distribution function (CDF) by appearance, up to Memcached's limit of 250 bytes (not shown); (b) value size CDF by appearance; and (c) value size CDF by total amount of data used in the cache. For example, values under 320 bytes or so in pool SYS use virtually no space in the cache; 320-byte values weigh around 8 percent of the data; and values close to 500 bytes take up nearly 80 percent of the entire cache's allocation.

Memcached Workload Pattern at Facebook

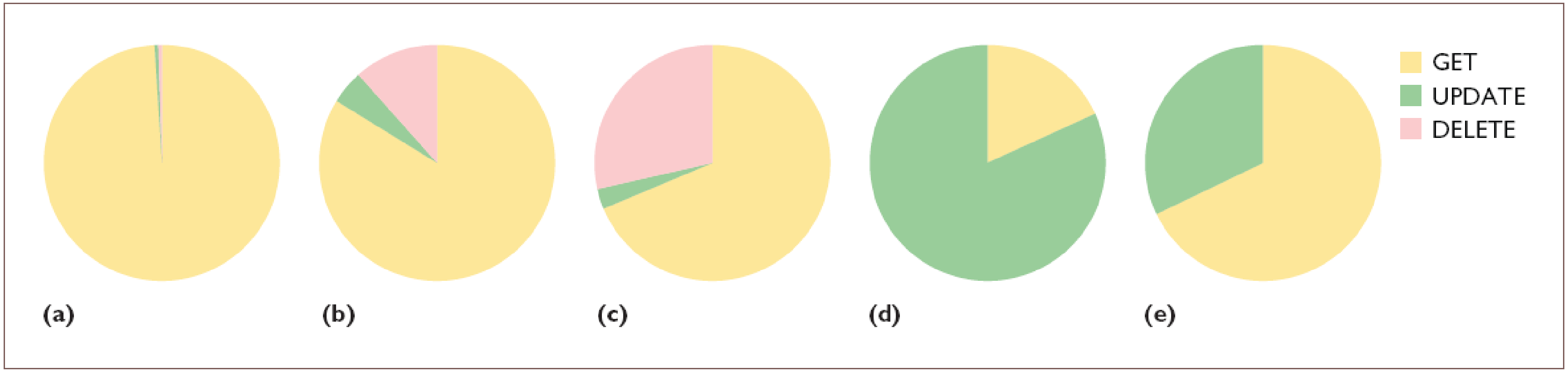
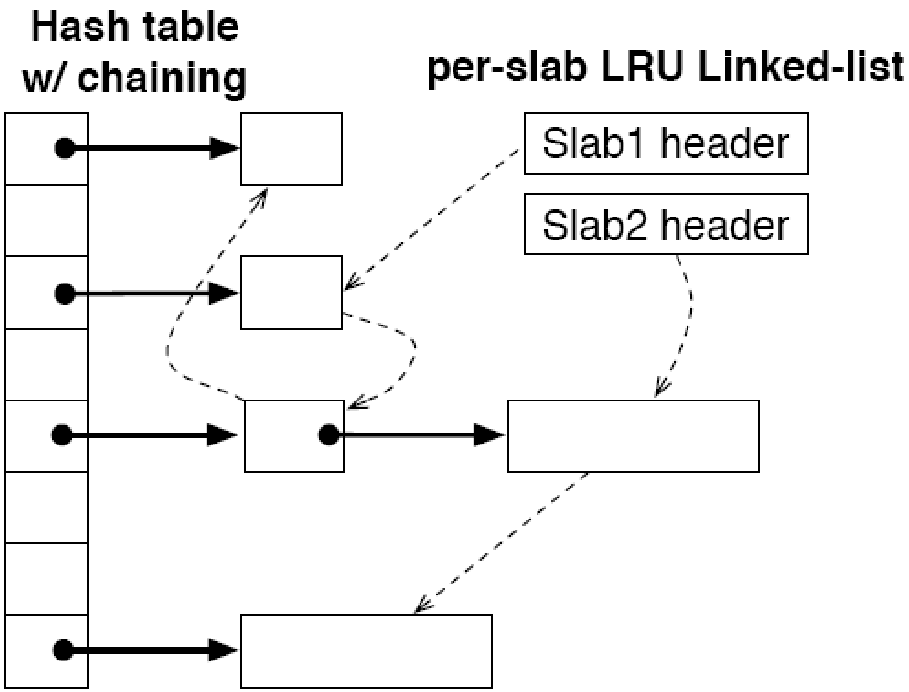


Figure 3. Distribution of request types per pool, over exactly seven days: (a) USR, n = 60.7 billion requests; (b) APP, n = 39.5 billion requests; (c) ETC, 30 billion requests; (d) VAR, 44.6 billion requests; and (e) SYS, 4.4 billion requests. UPDATE commands aggregate all non-DELETE writing operations.

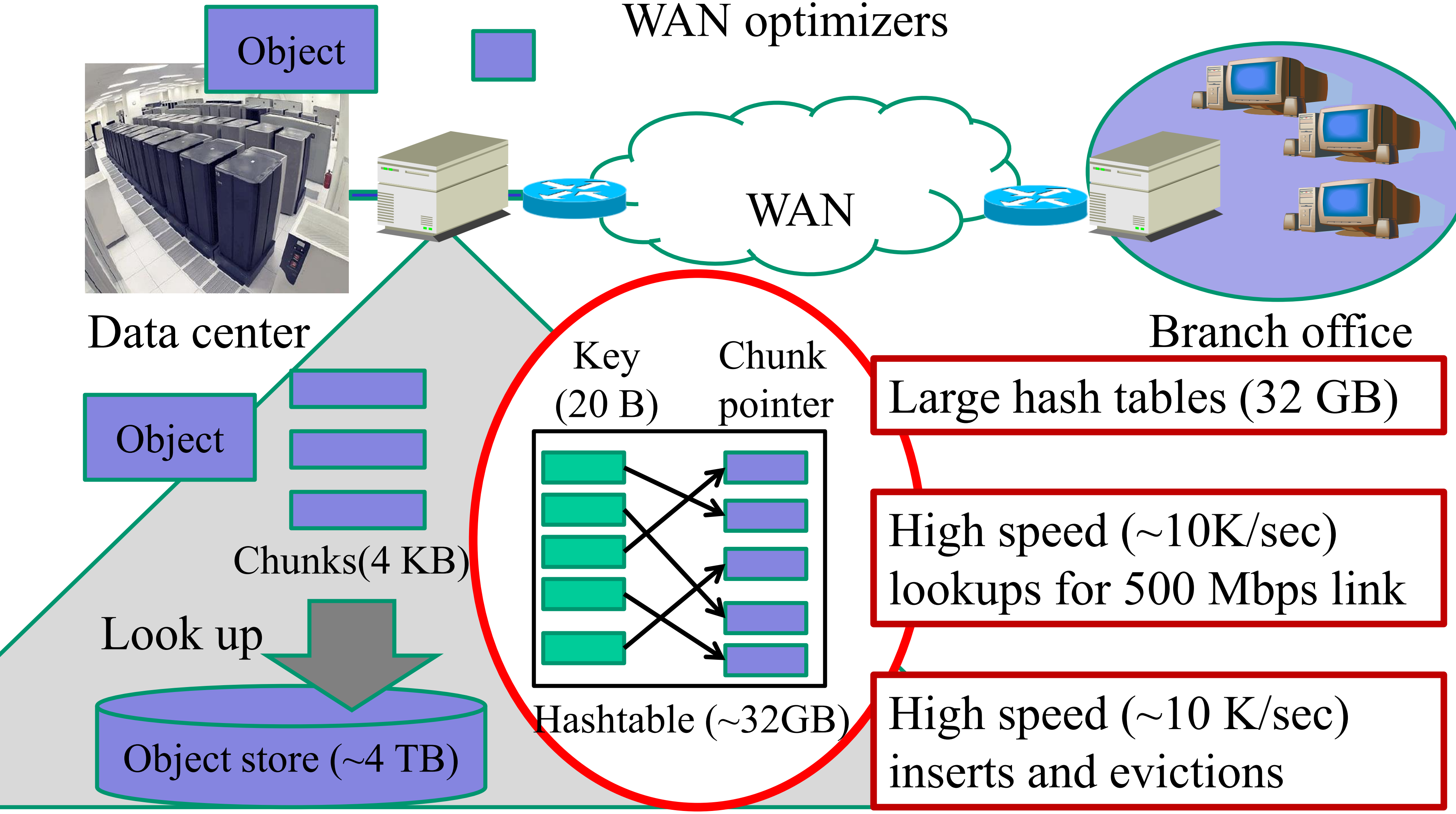
Why are update requests represent more expensive in memcached?

The Data Structure



- ❑ Slab-based memory allocation: memory is divided into 1 MB slabs consisting of fixed-length chunks. E.g, slab class 1 is 72 bytes and each slab of this class has 14563 chunks; slab class 43 is 1 MB and each slab has only one chunk. Least-Recently-Used (LRU) algorithm to select the items for replacement in each slab-class queue. (why slab?)
- ❑ Lock has to be used for integrity of the structure, which is very expensive (why?)

A Use Scenario of KV store: Data-intensive Networked Systems



WAN optimizers

Object

WAN

Data center

Branch office

Object

Chunks(4 KB)

Look up

Object store (~4 TB)

Key (20 B)

Chunk pointer

Hashtable (~32GB)

Large hash tables (32 GB)

High speed (~10K/sec) lookups for 500 Mbps link

High speed (~10 K/sec) inserts and evictions

Candidate options

Too slow

[+]Price statistics from 2008-09

Too expensive

	Random reads/sec	Random writes/sec	Cost (128 GB)
Disk	250	250	$30[+]
DRAM	300K	300K	$120K[+]
Flash-SSD	10K*	5K*	$225[+]

2.5 ops/sec/$

Slow writes

* Derived from latencies on Intel M-18 SSD in experiments

How to deal with slow writes of Flash SSD?

Flash/SSD primer

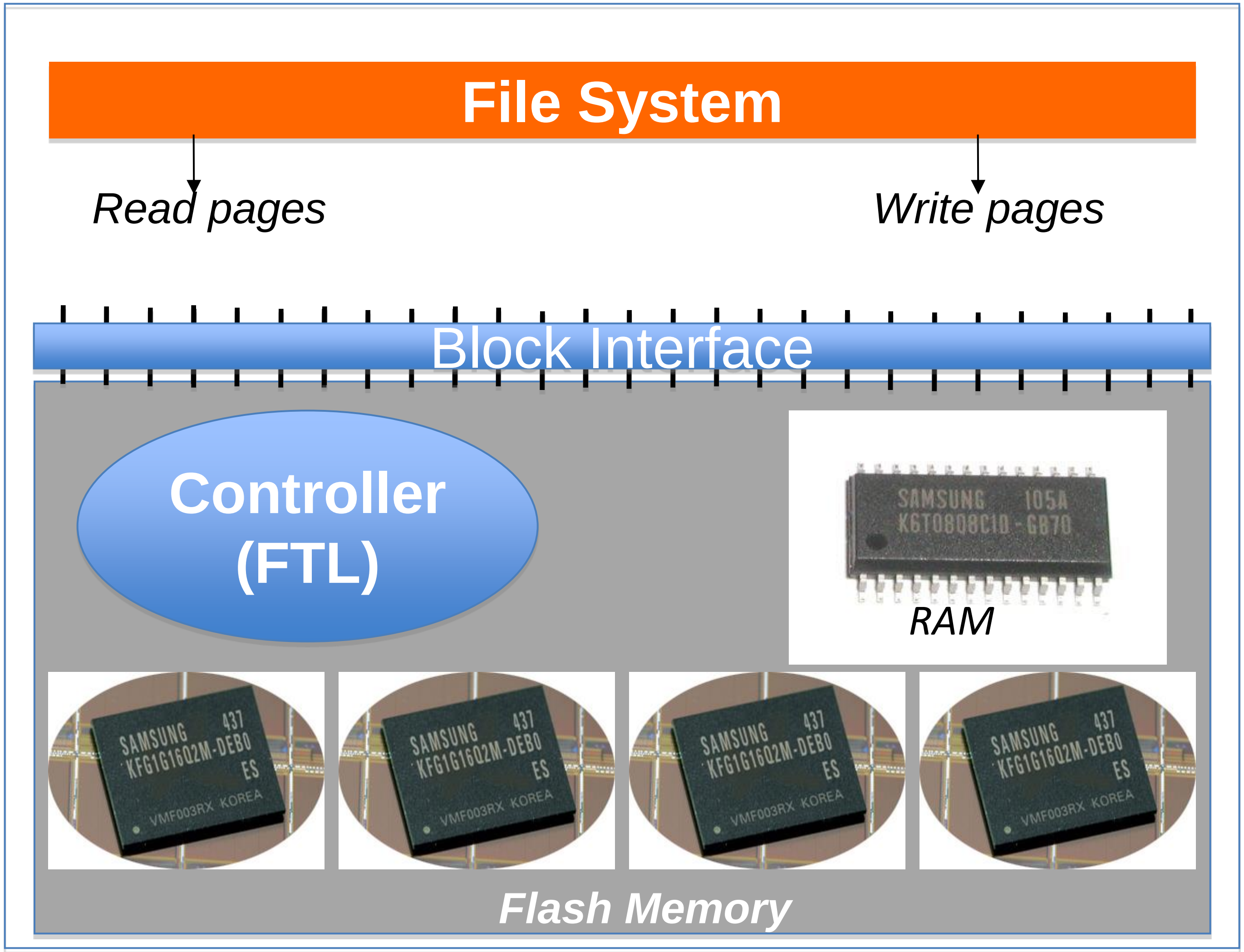
Random writes are expensive

- **Avoid random page writes**

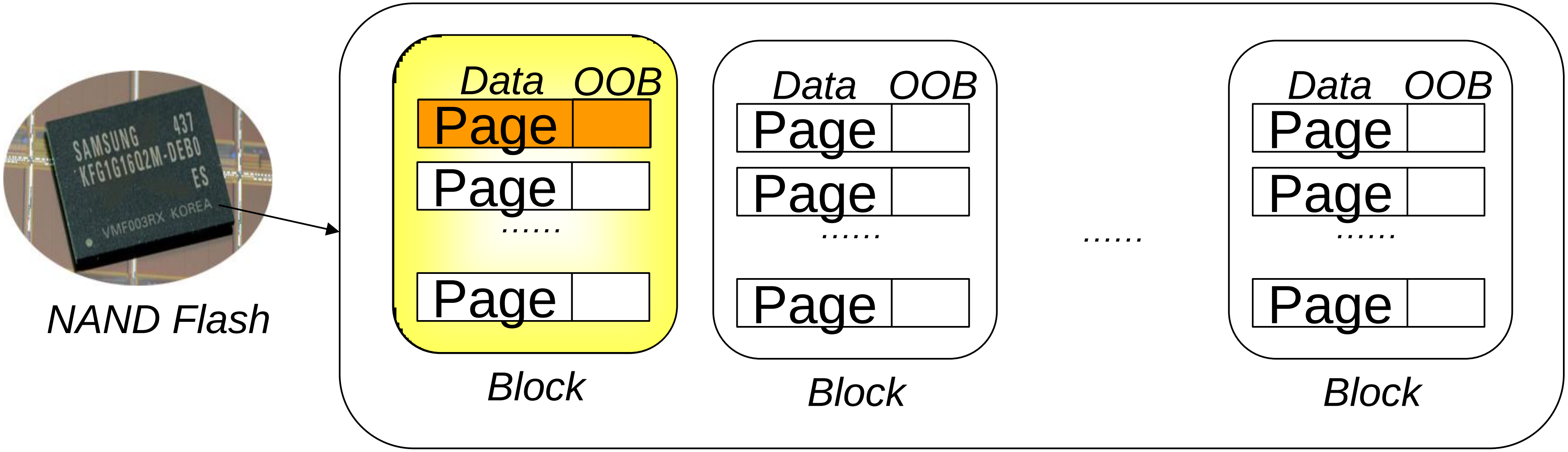
Reads and writes happen at the granularity of a flash page

- **I/O smaller than page should be avoided, if possible**

Flash Solid State Drive (SSD)



Basics of NAND Flash Memory



Three operations: read, write, erase

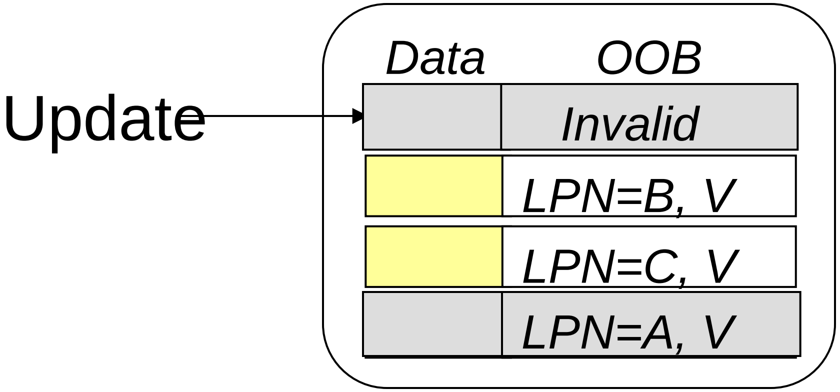
Reads and writes are done at the granularity of a **page** (2KB or 4KB)

Erases are done at the granularity of a **block**

- Block: A collection of physically contiguous pages (64 or 128)
- Block erase is the **slowest operation** requiring about 2ms

Writes can only be done on erased pages

Out-of-Place Updates



Data	OOB
	Invalid
	LPN=B, V
	LPN=C, V
	LPN=A, V

Block 0

LPN	PPN (PBN, Offset)
A	(0, 3)
B	(0, 1)
C	(0, 2)

Flash Mapping Table

Over-writes on the same location (page) are expensive

Updates are written to a free page

OOB area

- Keeps valid/free/invalid status
- Stores LPN, used to reconstruct mapping table upon power failure

Flash Translation Layer (FTL)

Flash Translation Layer

- Emulates a normal block device interface
- Hides the presence of erase operation/erase-before-write
- Address translation, garbage collection, and wear-leveling

Address Translation

- Mapping table present in small RAM within the flash device

SDF: A Customized SSD for Baidu

SSD deployment at a large scale since 2007

- •Purchase tens of thousands of SSDs each year.
- •Initially adopted SSDs in index servers in 2007
- •SSDs are commonly deployed in various servers.
- •SSDs support various services, such as indexing, CDN, SQL, and No-SQL(table, KV, object).

However, Baidu is increasingly challenged by the SSDs' inadequacies.

Jian Ouyang, Shiding Lin,1 Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang, **SDF: Software-Defined Flash for Web-Scale Internet Storage, Systems, in ASPLOS'14**

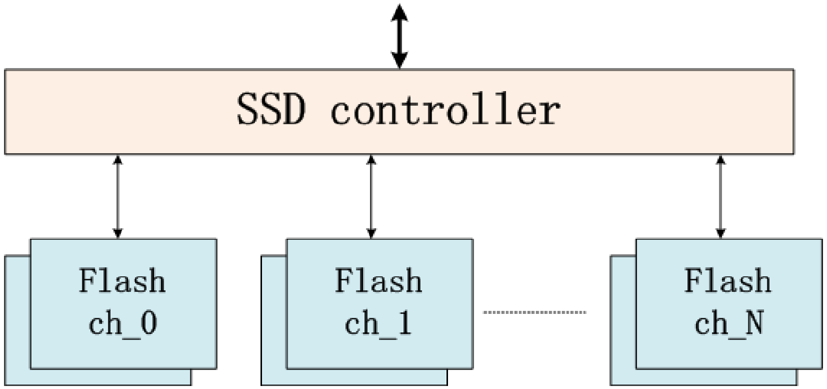
Background – SSD Limitations (1/3)

Low bandwidth utilization

- 70~80% for read and 40~50% for write
- 40% or less in real workload

Reasons

- Current architecture limits exploitation of hardware parallelism
  - Centralized SSD controller becomes the bottleneck.



- Access interface is not optimized for the flash
  - Write granularity (4KB vs. erase block size)

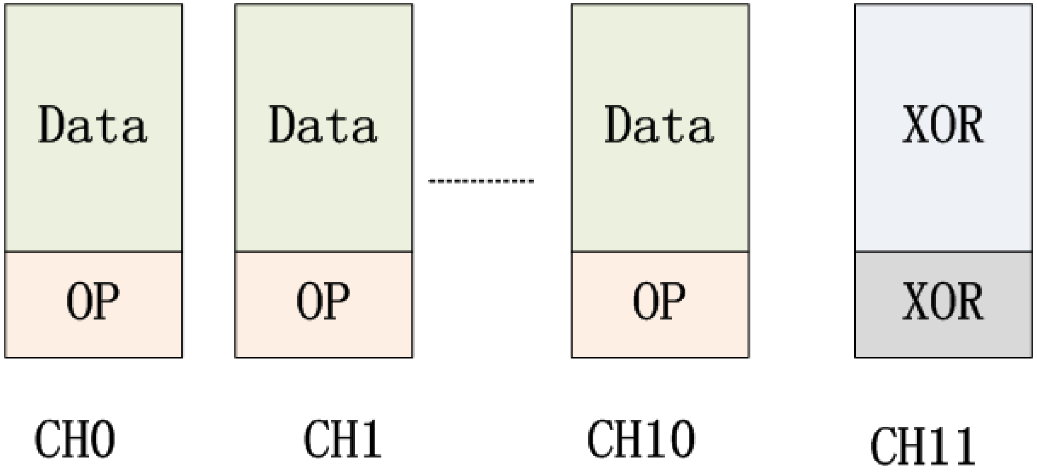
Background – SSD Limitations (2/3)

Limited capacity utilization

- Only 50%~70% for applications

Reasons

- 7%~50% raw capacity for over-provisioning (OP)
  - To accommodate out-of-place updates
- ~10% raw capacity for parity coding across flash channels
  - To recover from channel crash



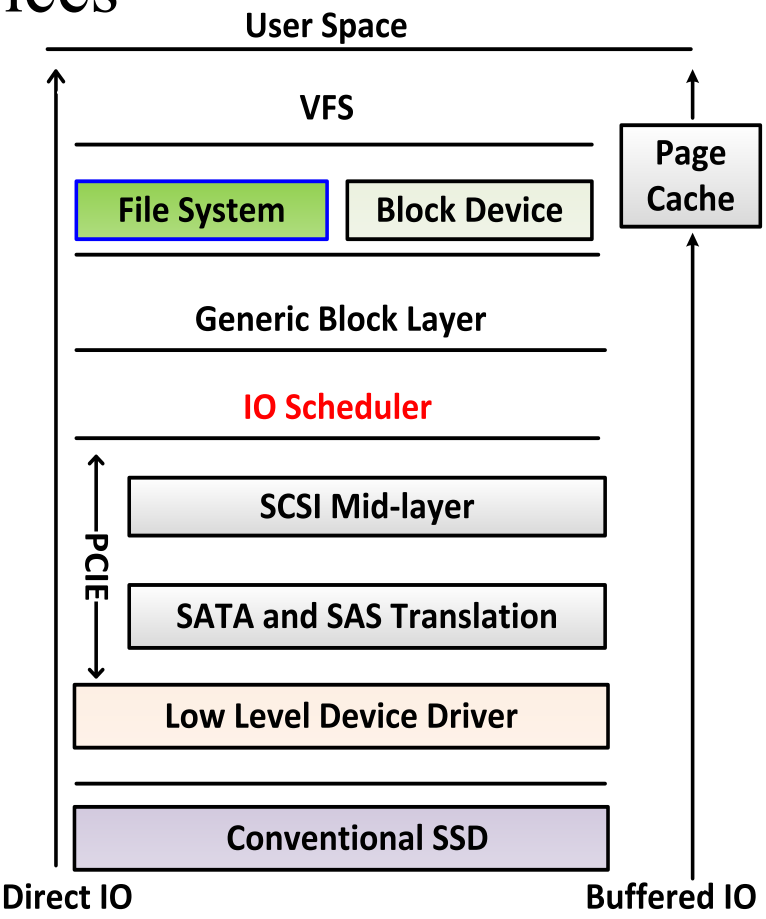
Background – SSD Limitations (3/3)

Less predictable performance

- Quality variation of online services

Reasons

- Garbage collection
  - Blocking the normal operations
- Linux IO stack
  - Difficult to configure
  - Difficult to debug



Background – challenges

Large-scale

- 10,000+ SSD deployment per year (10PB+ capacity)

Challenges

- Acquisition of extra devices
- Higher cost
  - Installation, cooling, and operational energy cost
  - Testing and debugging

Design Goals

An ideal SSD

- Delivers all raw hardware bandwidth to applications
- Makes all raw hardware capacity available to applications
- Provides predictable performance

Approaches

- Explore parallelism
  - Highly concurrent access
  - Keep all hardware channels busy
- Explore raw capacity
  - Remove spaces for over-provisioning and parity coding
- Eliminate redundant layers
  - Remove Linux IO stack and file system

Rethinking SSD in Data Center

Design principles

- HW/SW co-design
- Simplified hardware and system

The solution: Software-Defined Flash (SDF)

- Software defined
  - Expose low level hardware interface to software
  - Software can control hardware completely
- New hardware architecture
  - Expose hardware channels to software
  - Individual FTL controller for each channel
- New HW/SW interface
  - Write in the unit of erase block size
- Leverage global resource for data safety
  - Removes across-channel parity coding

Designs

Software/hardware interface

Software stack

Hardware implementations

Design - SW/HW Interface

Expose flash channels to applications

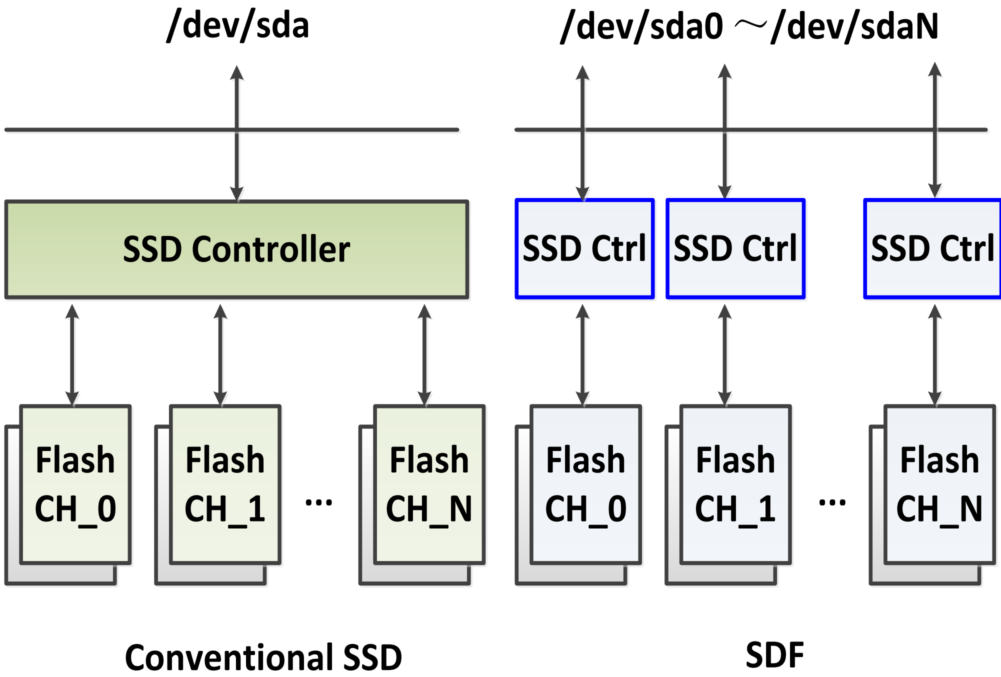
- Data placement determined by software
- Explores hardware parallelism to software
- Enforce synchronous IO

API

- Read: 8KB
- Write: 2MB aligned
- Erase: 2MB aligned

Erase conducted by software

- Erases before writes
- Scheduled by applications



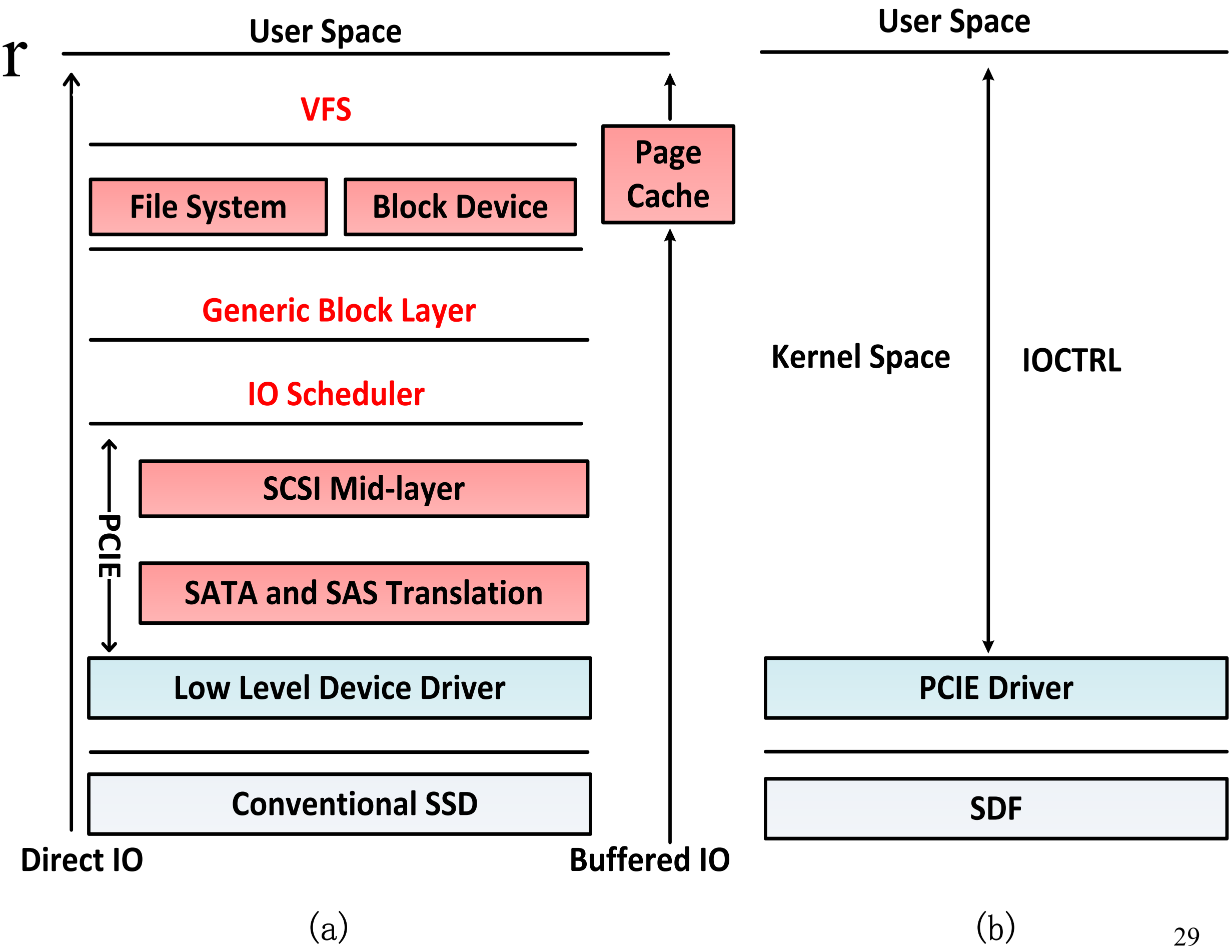
Design - Software Stack

Removing unnecessary software layers

- To reduce latency and CPU cycles
- To remove complexity of kernel configurations

User-defined scheduler

- Data layout
- Erase scheduling



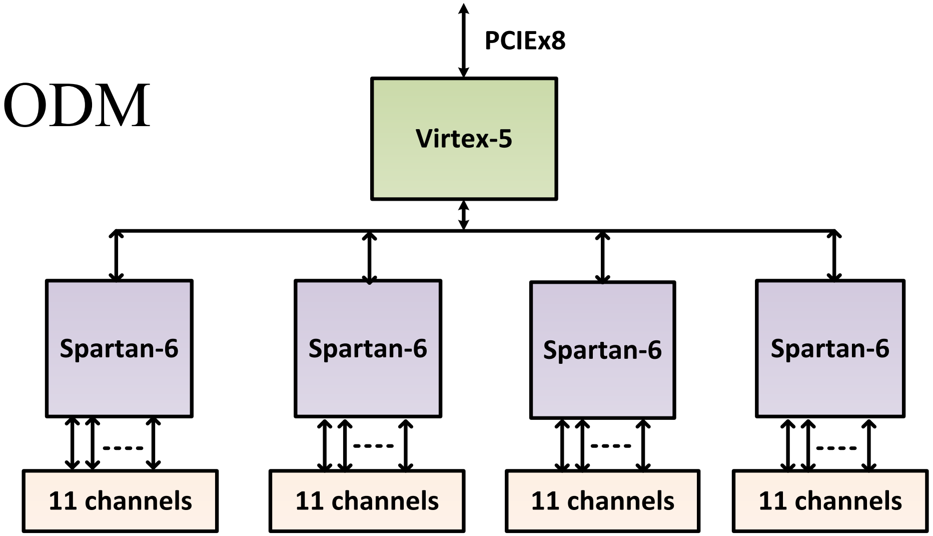
Design - Hardware

Specifications

- 25nm MLC NAND, 44 channels, ONFI 1.x asynchronous 40Mhz
- 5 FPGA, 4 Spartan-6 for FTL, Virtex-5 for PCIE

Development cost

- 12K lines of Verilog RTL (4K lines for FTL), 3K lines of C code for driver and file system
- 2 persons for 7 months
- Hardware board designed by ODM



Hardware Design

Simple hardware

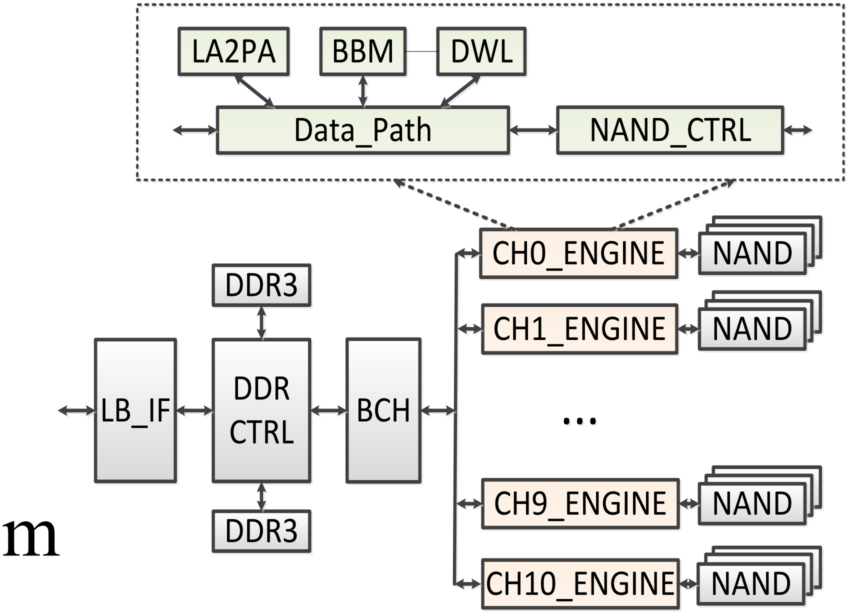
- No static wear-leveling and garbage collection
- No data stripping logic and parity coding
- No hardware cache and battery (or capacitor)

FTL

- Dynamic wear-leveling
- bad-block management
- block-level address mapping

Fault tolerant

- 30b BCH per 1KB block
- Reliance on system-level system
- Remove parity code
- Only 1 error detected with six months and recovered by system-level replication



Evaluations

Setup

- HOST
  - Intel E5620x2, 2.4GHz
  - 32GB memory
  - 2.6.32 Linux Kernel

- SSD
  - Huawei Gen3 PCIE based, 44 CH, 25nm MLC NAND, 16GB/CH
  - SDF is the same hardware configuration with SSD

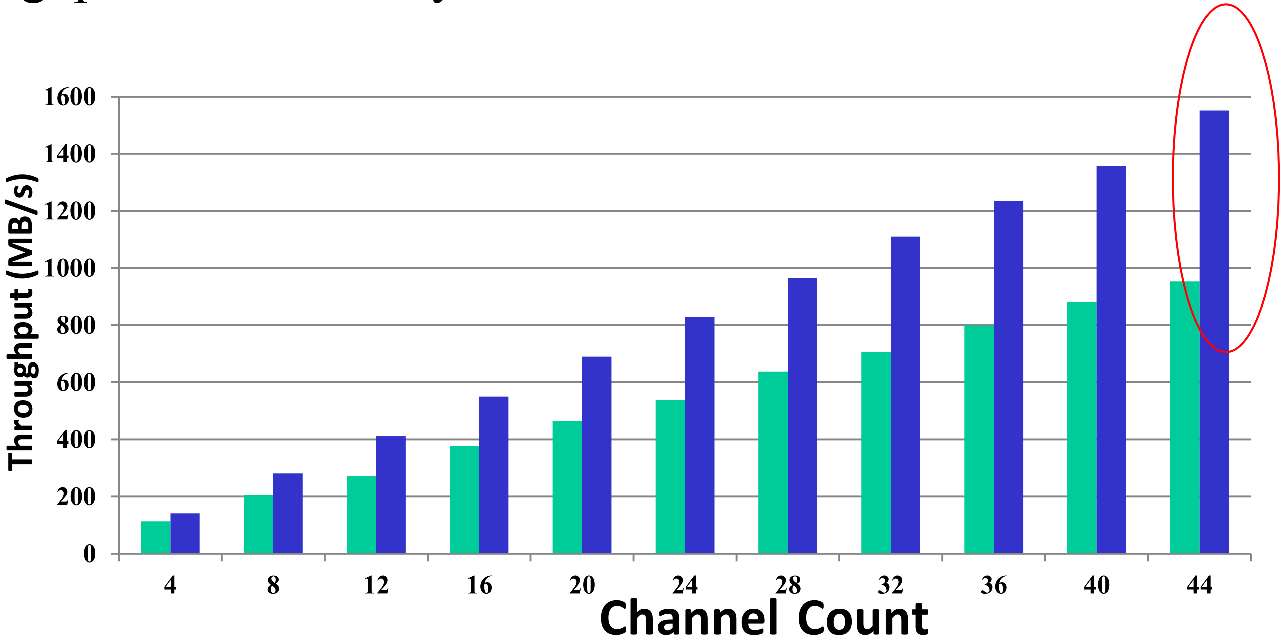
Evaluations

Micro benchmark: throughput

- 99% read bandwidth and 95% write bandwidth utilization

Operations	8 KB Read	16 KB Read	64 KB Read	8 MB Read	8 MB Write
Baudu SDF (GB/s)	1.23	1.42	1.51	1.59	0.96
Huawei Gen3 (GB/s)	0.92	1.02	1.15	1.20	0.67
Intel 320 (GB/s)	0.17	0.20	0.22	0.22	0.13

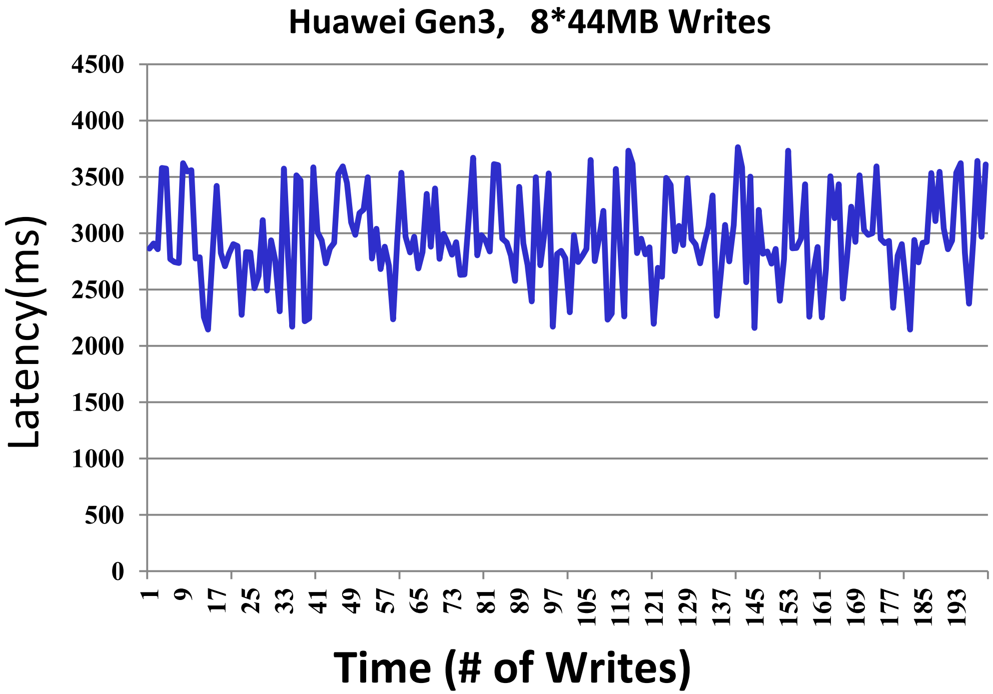
- Throughput scales linearly with channel count

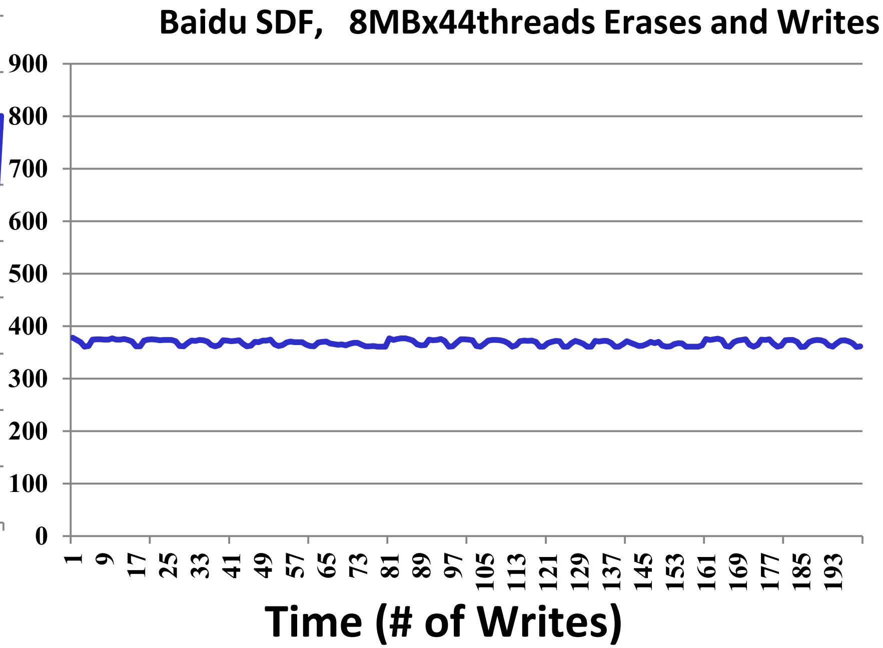


R:1.59GB/s
W:0.96GB/s

Throughput variation

- Setup: 8*44MB writes when SSD and SDF are 95% full
- SDF maintains a latency of 380ms.
- SSD's latency is 7.8x higher and highly variable.
- 95% of raw write bandwidth for SDF and 11.4% for SSD



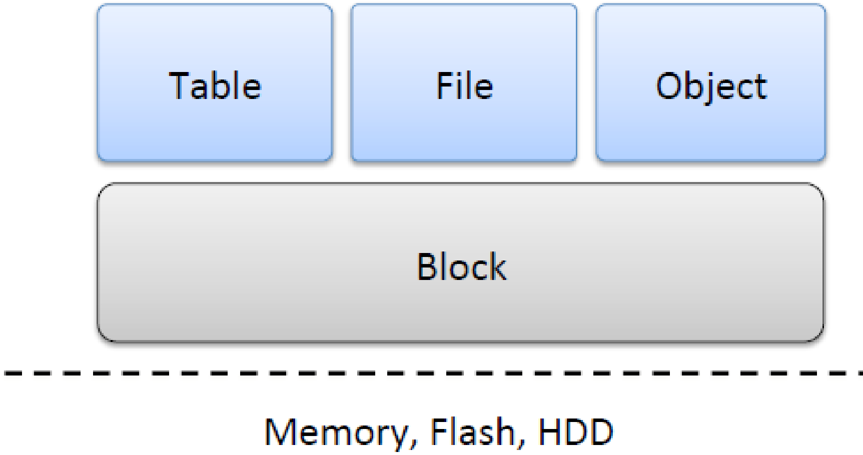


Production system

- Various data management systems implemented as Key-value stores
- Log-based merge tree
- 3000 SDF used in Baidu’s Table system (web page repository)
- Batch size: number of requests issued in one batch
- Slice: a KV store responsible for a given range of keys.

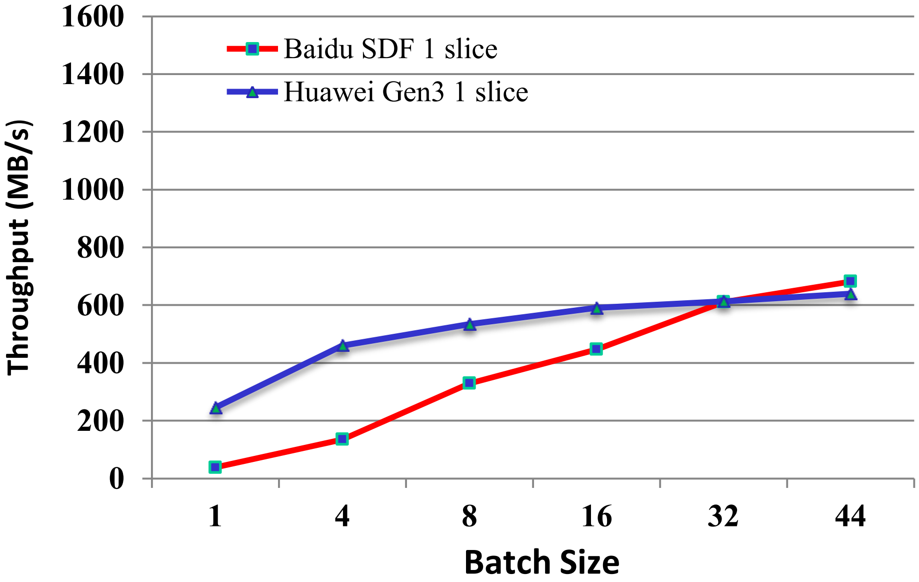
Experiments setup

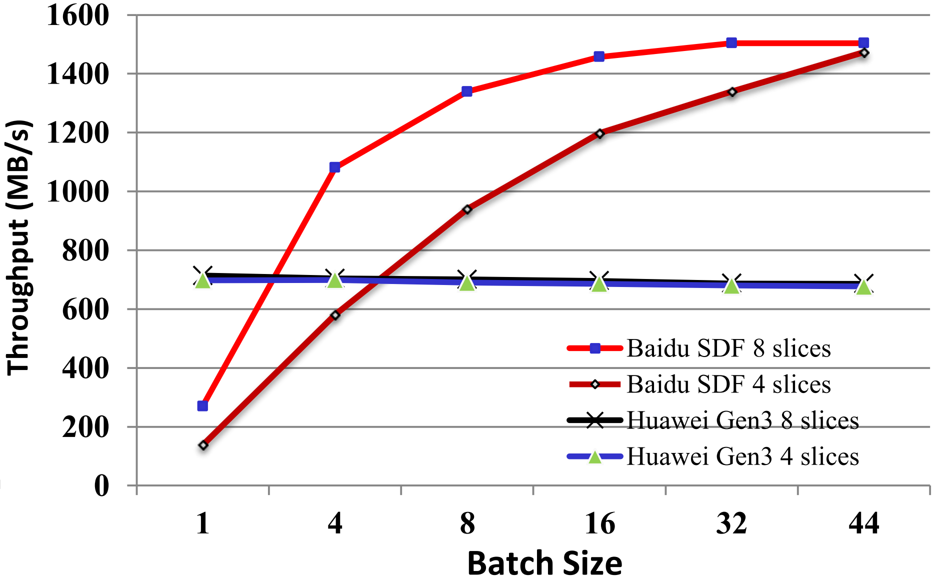
- Master node with 2 10Gbps NIC
- Huawei Gen3 SSD with 25% OP



512KB random reads

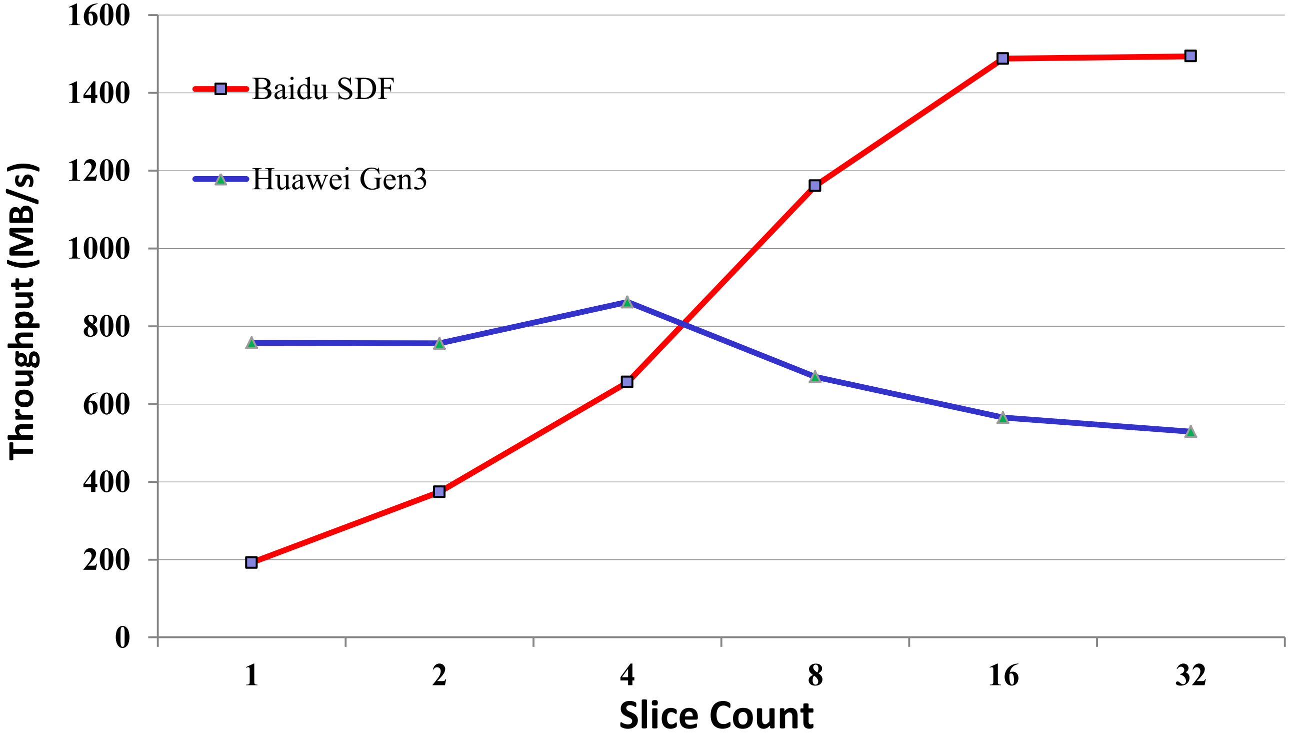
- Data mining job is random read intensive
- Each server contains 50~60 slices
- 1 slice: SSD is better when batch size is less than 32
- 8 slice: SDF is 3x better than SSD





Sequential read

- Building index generates mostly sequential reads
- Throughput of SSD degrades with the increase of read
- Throughput of SDF increases almost linearly with the slide count



Summary of SDF

Key ideas

- Exposes flash channels to software
- SW/HW co-design

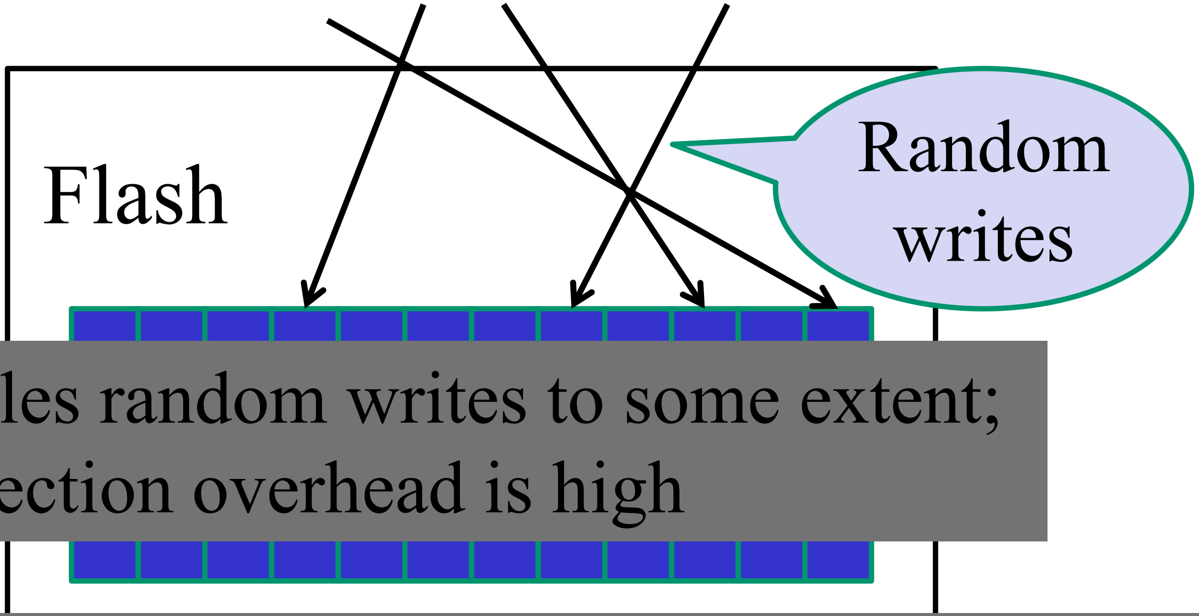
Results

- 95% write and 99% read bandwidth utilization
- 99% capacity utilization
- 50% cost reduction per GB compared with SSD for workload on the production systems

A Naïve Design: Hash Table on Flash/SSD

Cannot move entire hash table to the SSD:

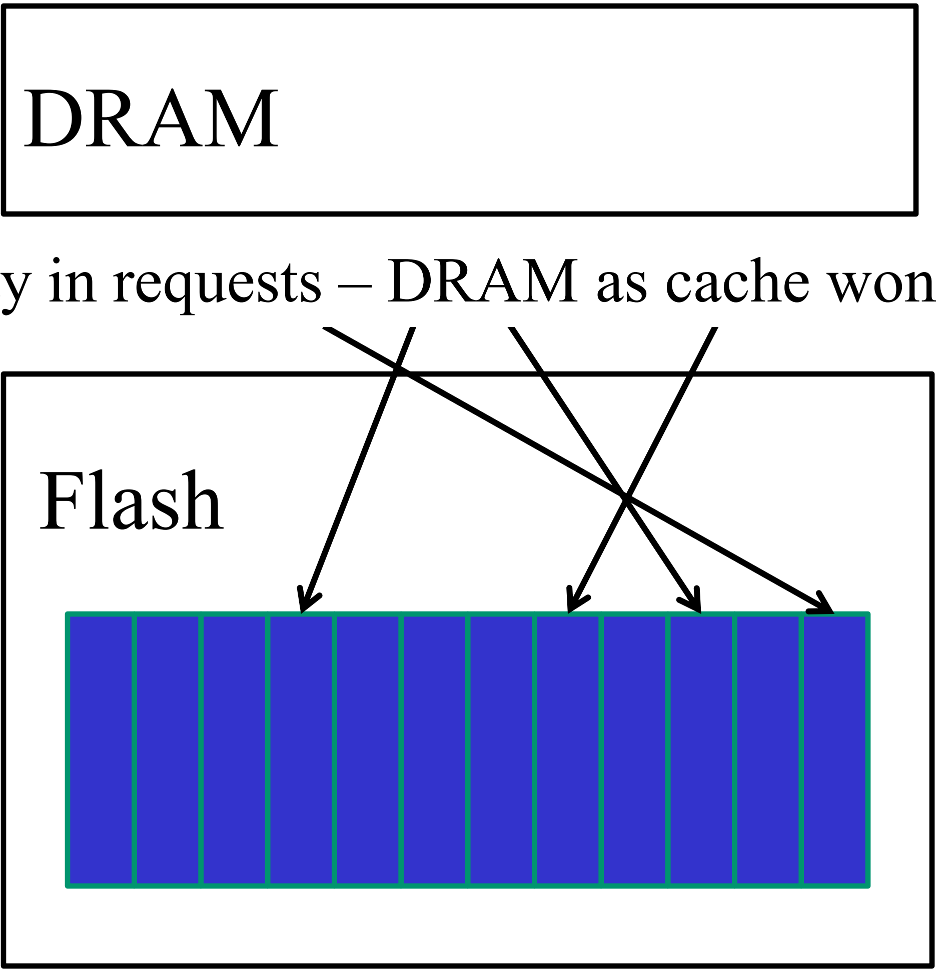
Keys are likely to hash to random locations



SSDs: FTL handles random writes to some extent;
But garbage collection overhead is high

~200 lookups/sec and ~200 inserts/sec with WAN optimizer workload, << 10 K/s and 5 K/s

A Naïve Design: Hash Table on Flash/SSD (con’d)



Can’t assume locality in requests – DRAM as cache won’t work

Three Metrics to Minimize

Memory overhead = Index size per entry

- Ideally 0 (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

Design I: FAWN for Fast Read and Fast Write

FAWN (Fast Array of Wimpy Nodes)

- A Key-Value Storage System
  - I/O intensive, not computation
  - Massive, concurrent, random , small-sized data access
- A new low-power cluster architecture
  - Nodes equipped with embedded CPUs + FLASH
  - A tenth of the power compared to conventional architecture

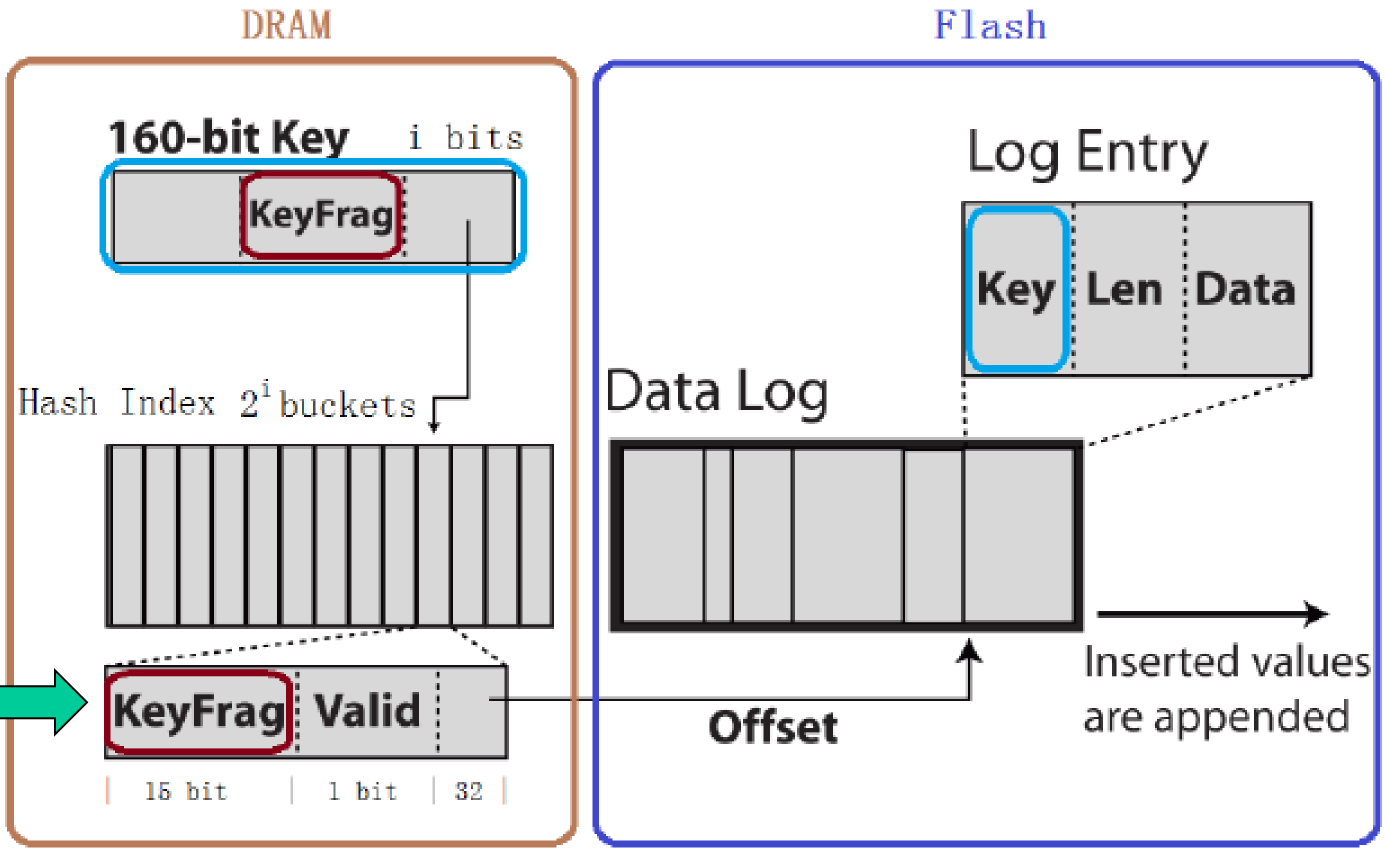
David G. Andersen, et al, **FAWN: a Fast Array of Wimpy Nodes, on SOPS'09**

FAWN System

- ❑ Hash Index to map 160-bit keys to a value stored in the data log;
- ❑ It stores only a fragment of the actual key in memory to find a location in the log => reduce memory use.
- ❑ How to store, lookup, update, and delete?
- ❑ How to do garbage collection?
- ❑ How to reconstruct after a crash, and how to speed up the reconstruction?
- ❑ Why is a Delete entry necessary? (hint: fault tolerance)

FAWN DataStore

Constrained DRAM	Avoid Random Write in Flash
Only maintain Hash Table in DRAM	Append-only log-structured filesystem



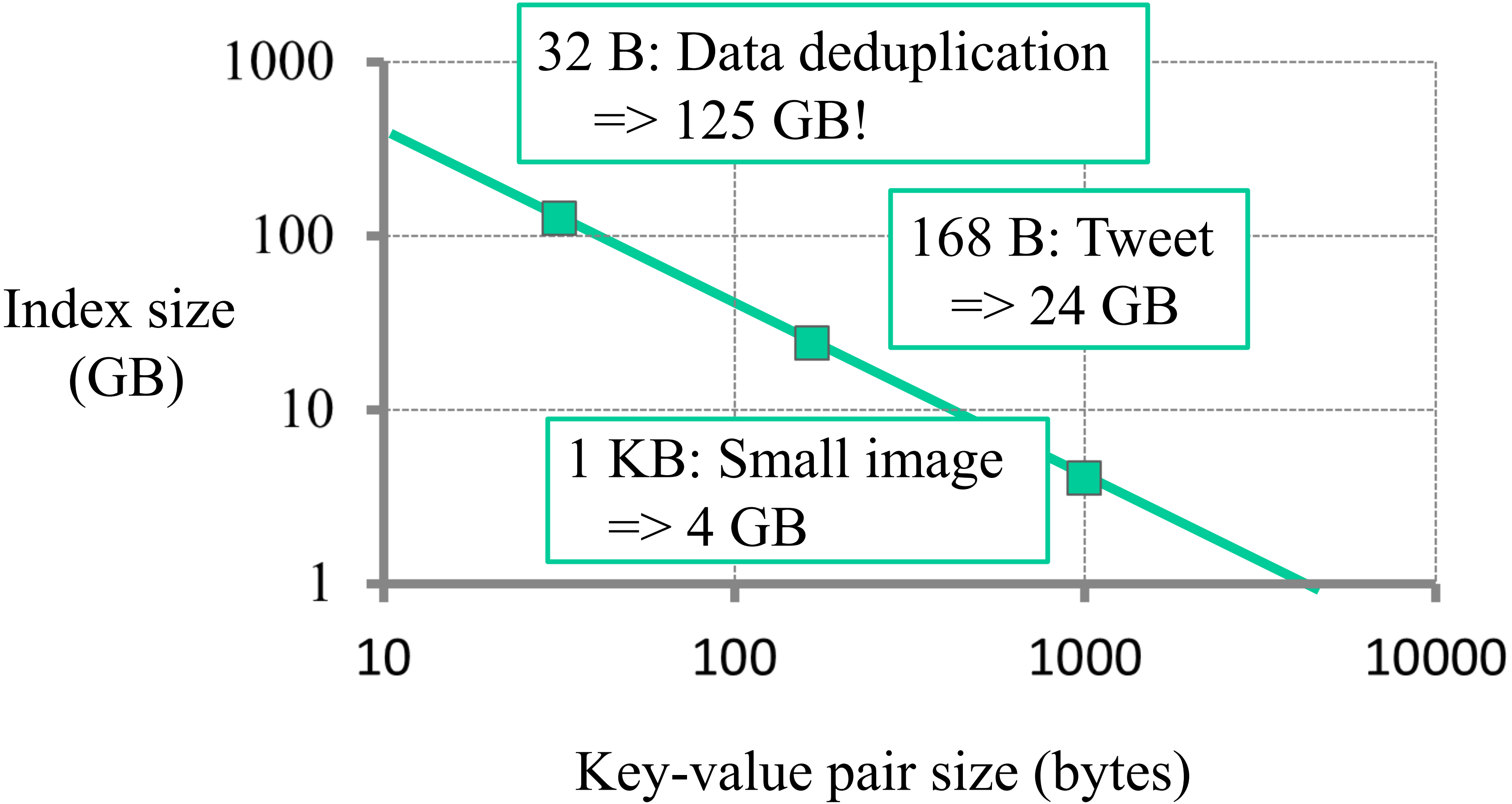
Chained hash entries in each bucket

DRAM Must be Used Efficiently

DRAM used for index (locate) items on flash

1 TB of data to store on flash

4 bytes of DRAM for key-value pair (previous state-of-the-art)



32 B: Data deduplication
=> 125 GB!

168 B: Tweet
=> 24 GB

1 KB: Small image
=> 4 GB

Design II: SkimpyStash for Small Memory Demand

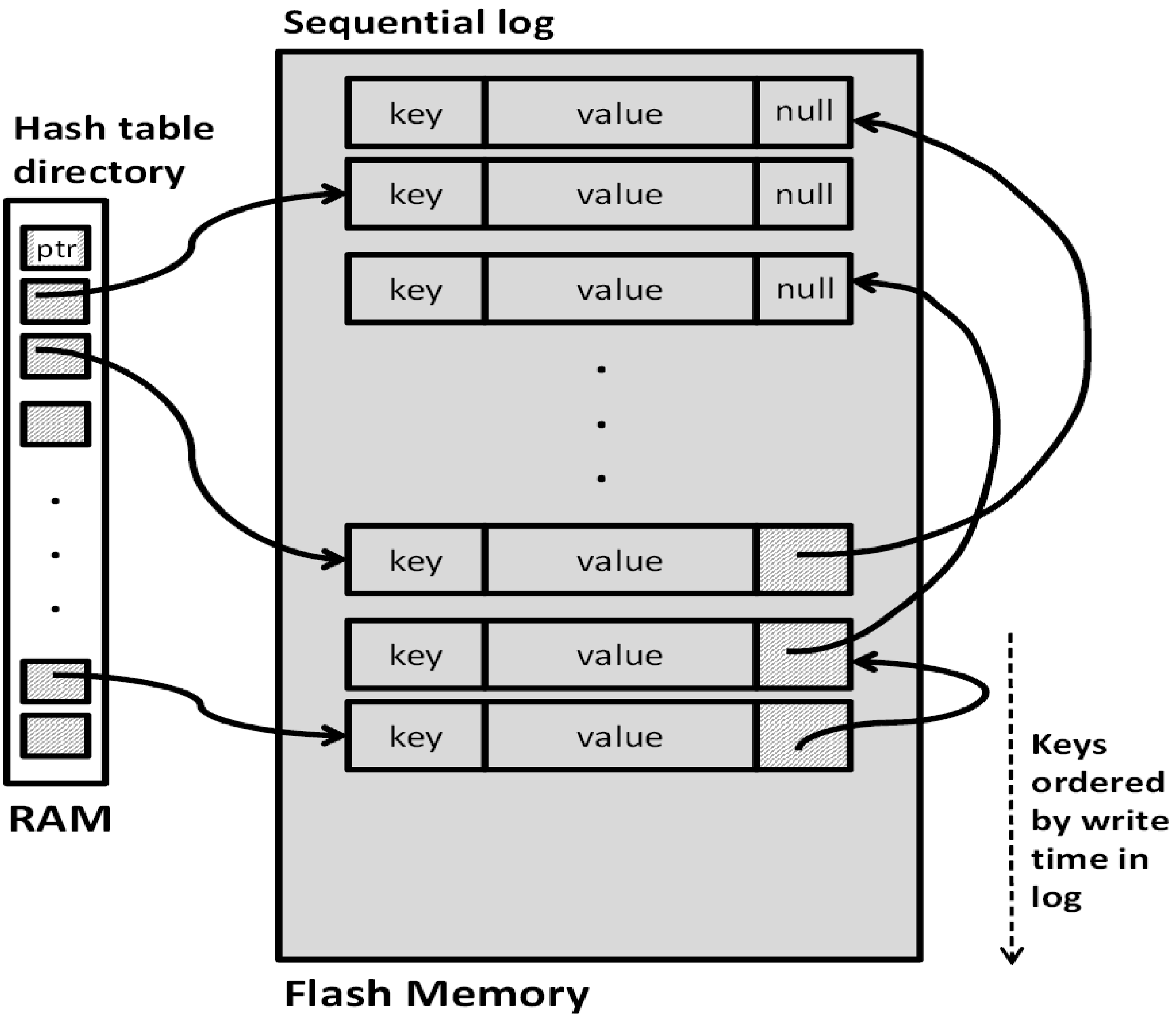
- ❑ Memory use is not proportional to the KV item count.
- ❑ Place the hash table buckets, including the links, to the flash.
- ❑ Only the first pointers to buckets (Hash table directory), are in memory.

❑ Debnath et al., SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage in SIGMOD'11

Design II: SkimpyStash for Small Memory Demand

- Use in-RAM write buffer to enable batched writes (timeout threshold to bound response time and concurrent write and flush)
- Basic operations:
  - lookup: HT directory in memory → bucket on the flash
  - insert: buffer in memory → batched write to the flash as a log and linked into HT directory.
  - delete: write a NULL entry

SkimpyStash Architecture



- ❑ Garbage collection in the log:
  - ➢ Start from the log tail (the currently written end).
  - ➢ Do page by page,
  - ➢ Cannot update predecessor's pointer in the bucket.
  - ➢ Compact and relocate whole bucket → leave orphans for garbage collection.
- ❑ The cost of write/delete/lookup

  In the worst case how many flash reads are needed for one lookup?
- ❑ The consequence of unbalanced buckets

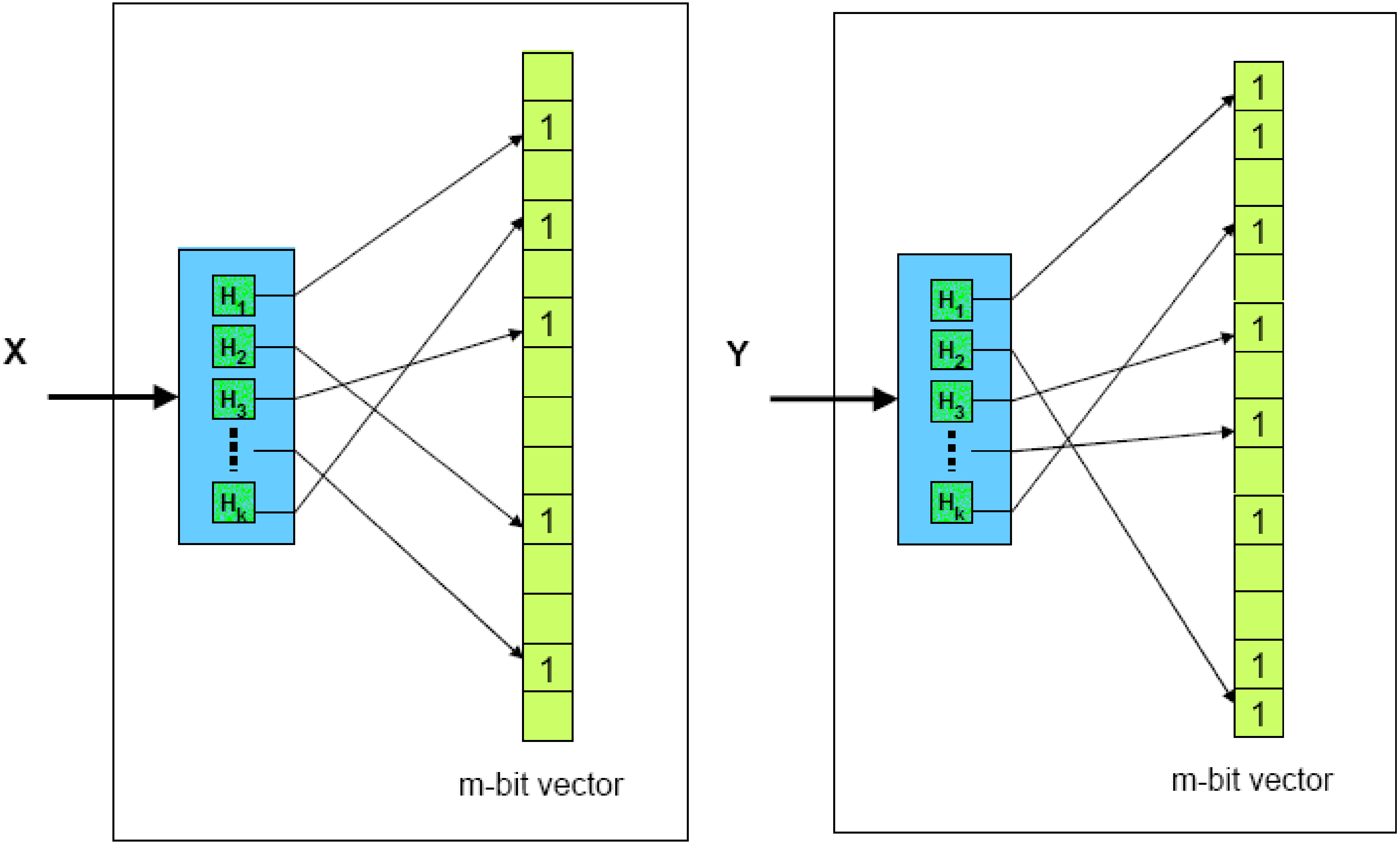
  Exceptionally long buckets → unacceptably long lookup time!
- ❑ Solution: two-choice-based hashing: each key would be hashed to two candidate HT directory buckets, using two hash functions h1 and h2, and inserted into the one that has currently fewer elements
- ❑ How to know in which bucket to search for a lookup?

  Use of Bloom filter for each bucket. The filter is dimensioned one byte per key and assume average number of items in each bucket.

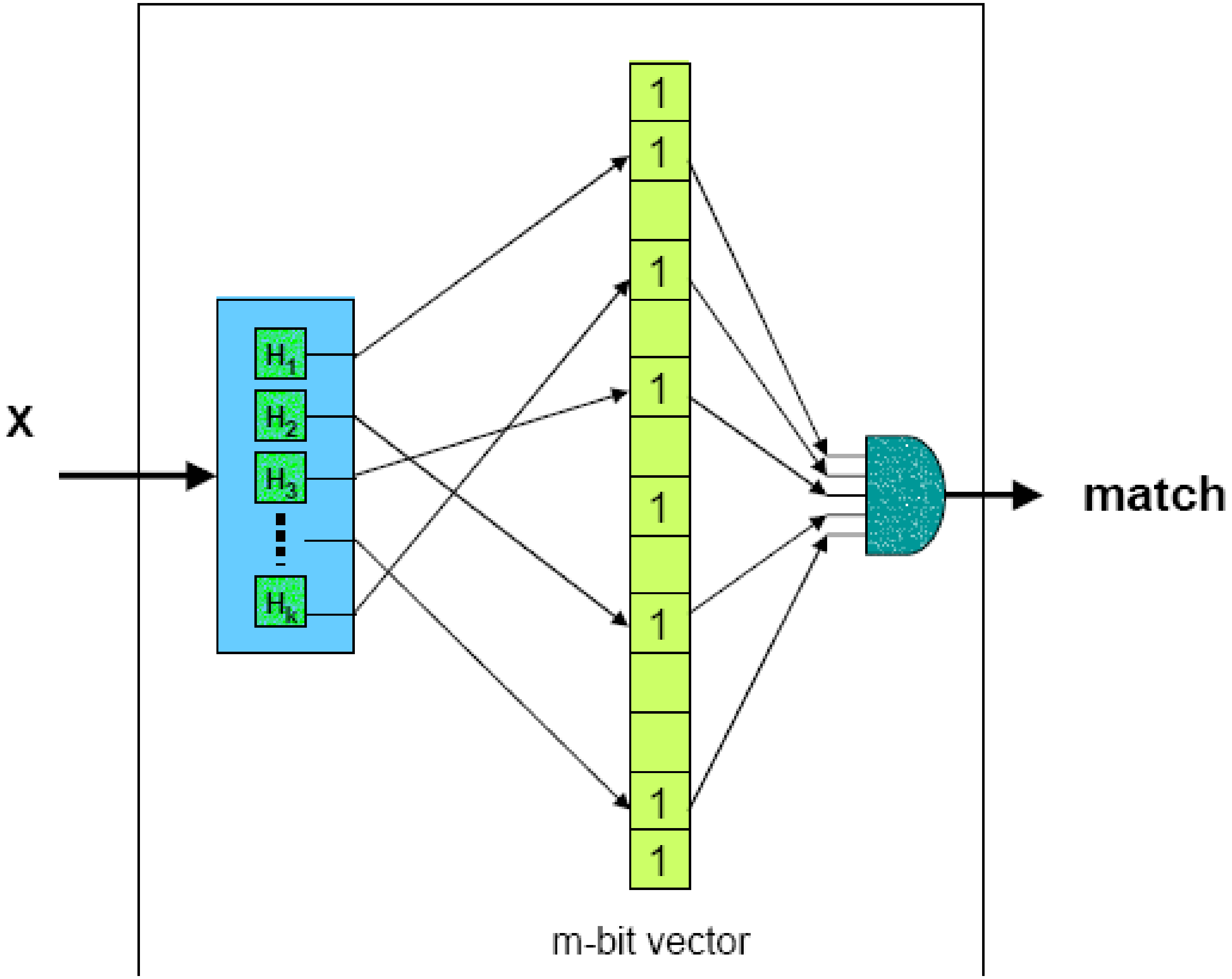
Background on Bloom Filter

- Data structure proposed by Burton Bloom
- Randomized data structure
  - Strings are stored using multiple hash functions
  - It can be queried to check the presence of a string
- Membership queries result in rare false positives but never false negatives
- Originally used for UNIX spell check
- Modern applications include :
  - Content Networks
  - Summary Caches
  - route trace-back
  - Network measurements
  - Intrusion Detection

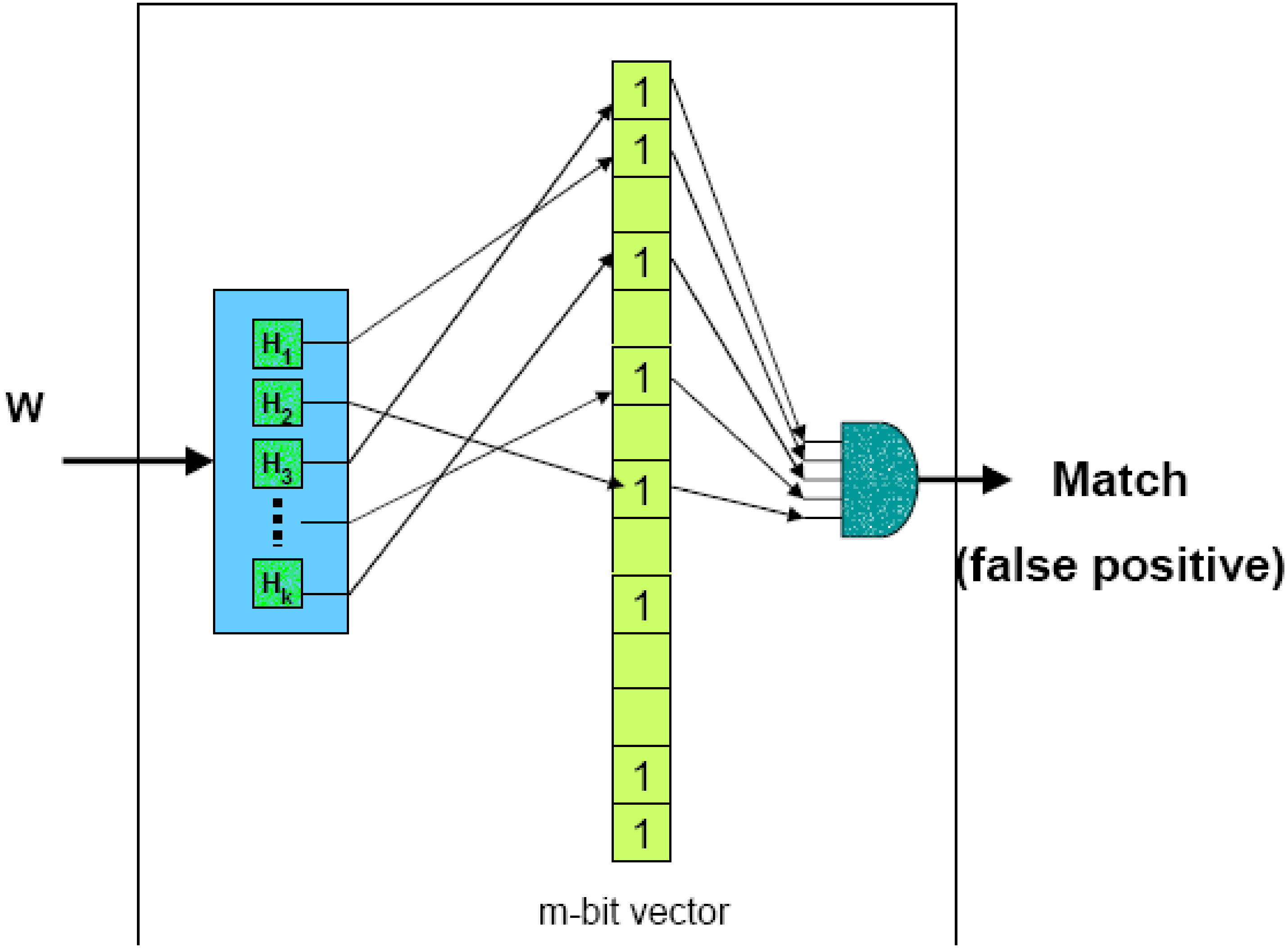
Programming a Bloom Filter



Querying a Bloom Filter



Querying a Bloom Filter (False Positive)



Optimal Parameters of a Bloom Filter

Bloom filter computes k hash functions on input

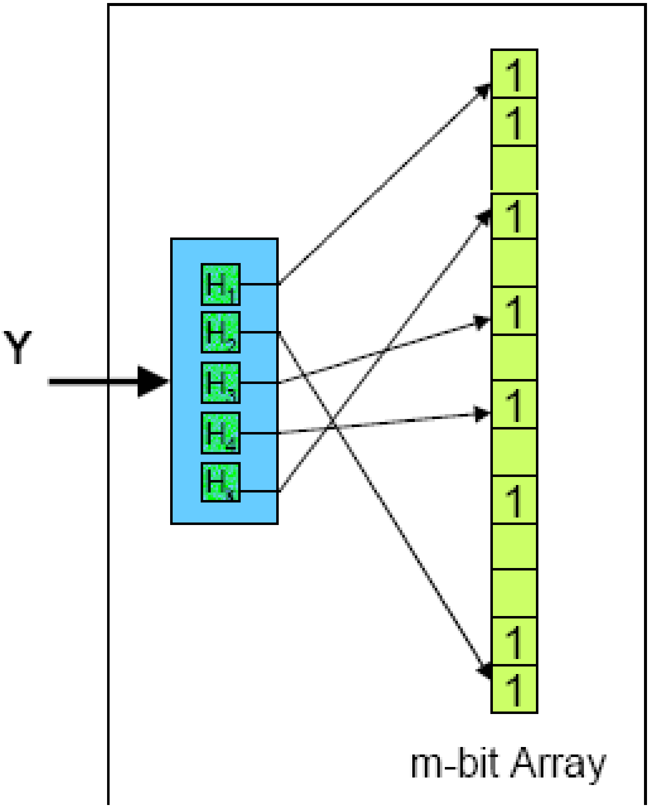
- *n* : number of strings to be stored
- *k* : number of hash functions
- *m* : the size of the bit-array (memory)

- The false positive probability

  $f = (\frac{1}{2})^k$

- The optimal value of hash functions, k, is

  $k = \ln 2 \times m/n = 0.693 \times m/n$

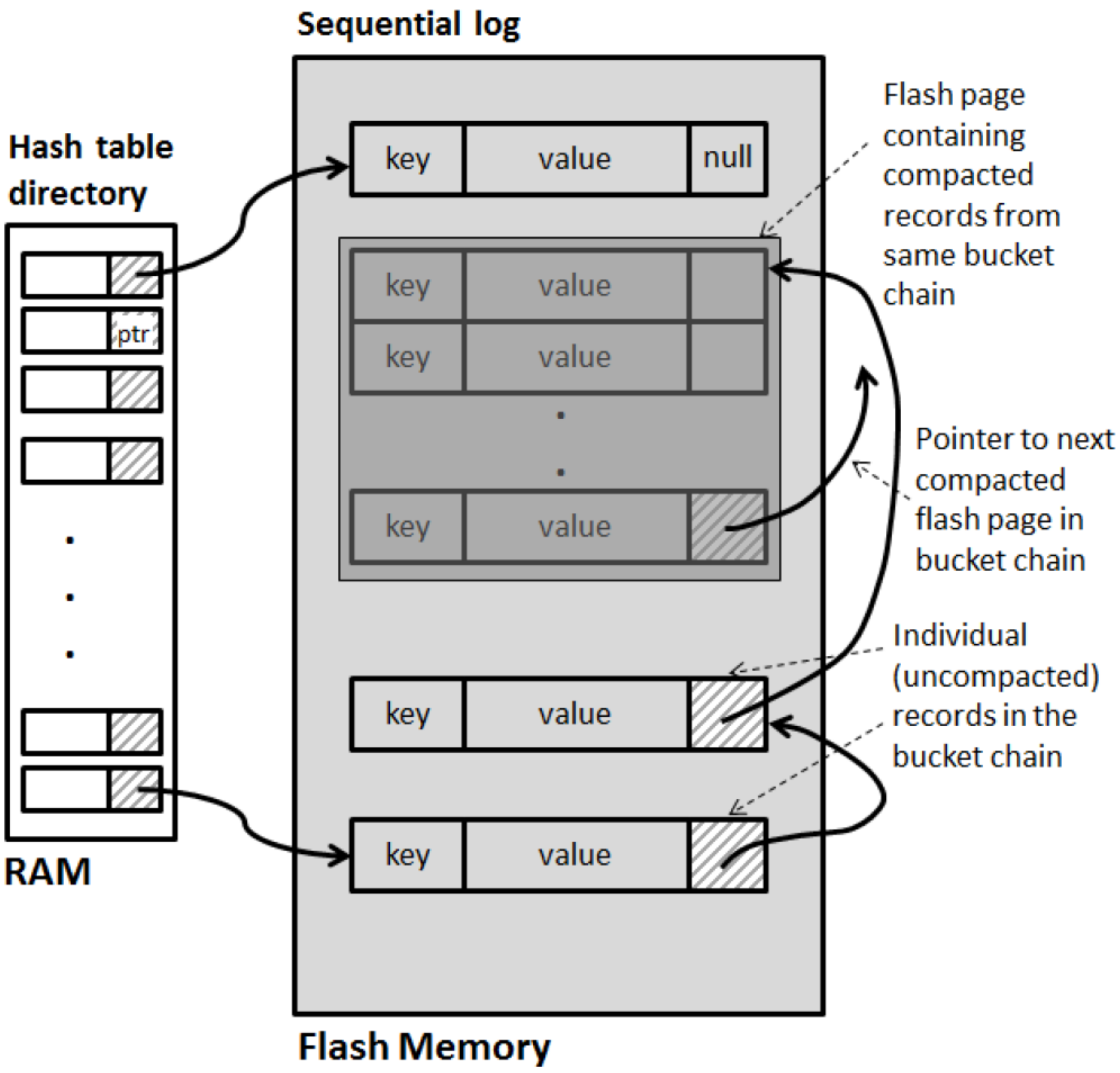


Key Point : false positive rate decreases exponentially with linear increase in number of bits per string (item)

Compaction in SkimpyStash

The consequence of spreading a chain of entries in a bucket across pages.

Use compaction to ameliorate the issue.


Sequential log
Hash table directory
ptr
RAM
key
value
null
key
value
key
value
key
value
key
value
key
value
Flash page containing compacted records from same bucket chain
Pointer to next compacted flash page in bucket chain
Individual (uncompacted) records in the bucket chain
Flash Memory

The Weaknesses of SkimpyStach

- Long/Unpredictable/unbounded lookup time.

Design III: BufferHash using Equally-sized Levels

- ❑ Move entire hash tables to the disk/flash
- ❑ The store consists of multiple levels and each is organized as a hash table.

Anand et al., **Cheap and Large CAMs for High Performance Data-Intensive Networked Systems** in NSDI'10

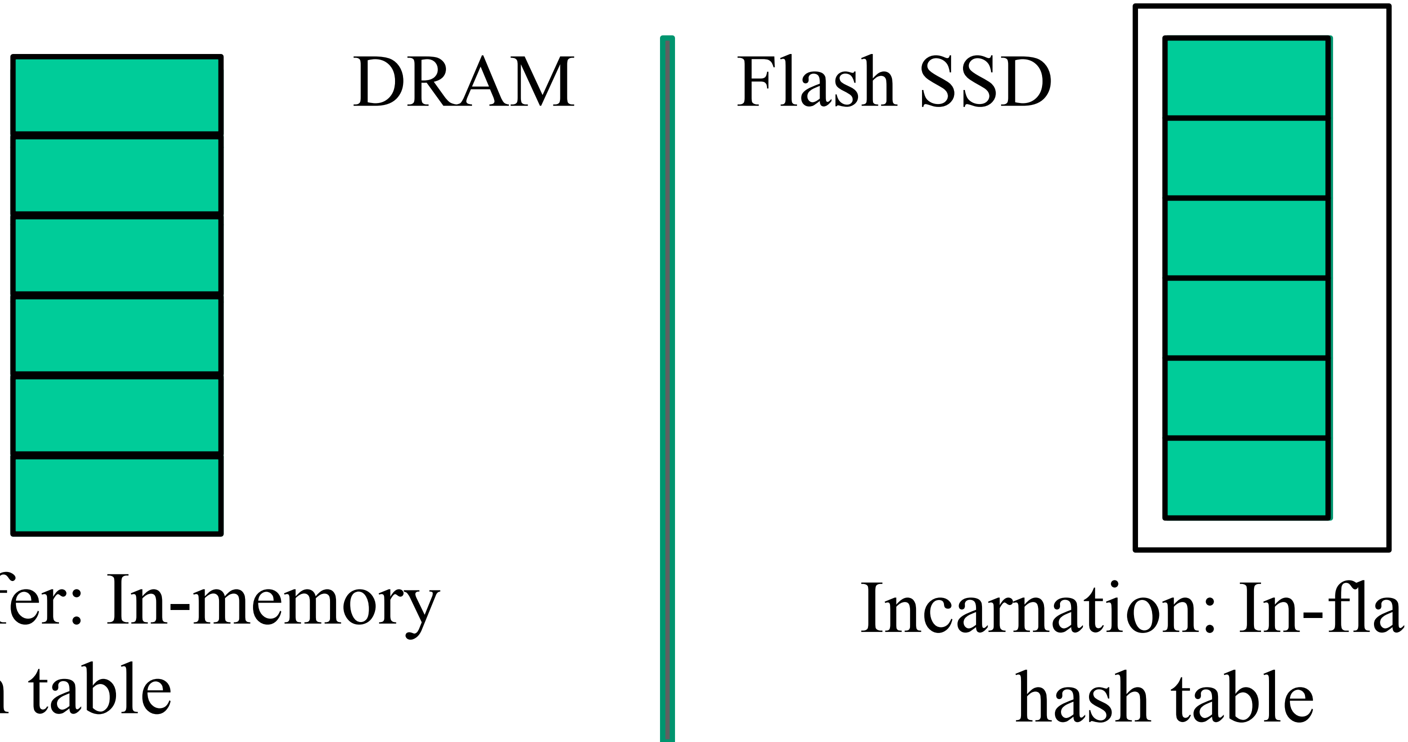
The approach: Buffering insertions

Control the impact of random writes

Maintain small hash table (**buffer**) in memory

As in-memory buffer gets full, write it to flash

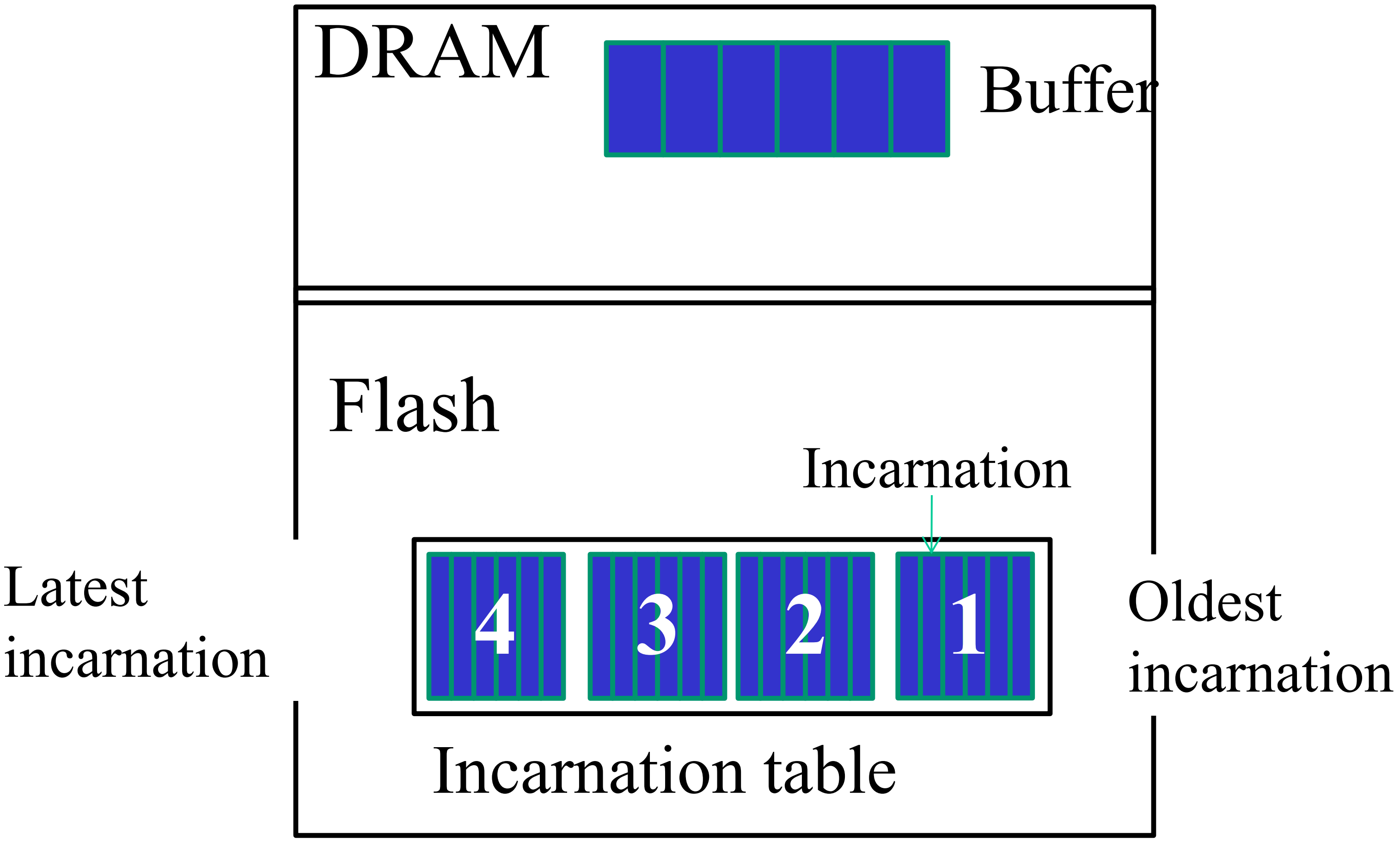
- We call in-flash buffer, **incarnation** of **buffer**



Buffer: In-memory hash table

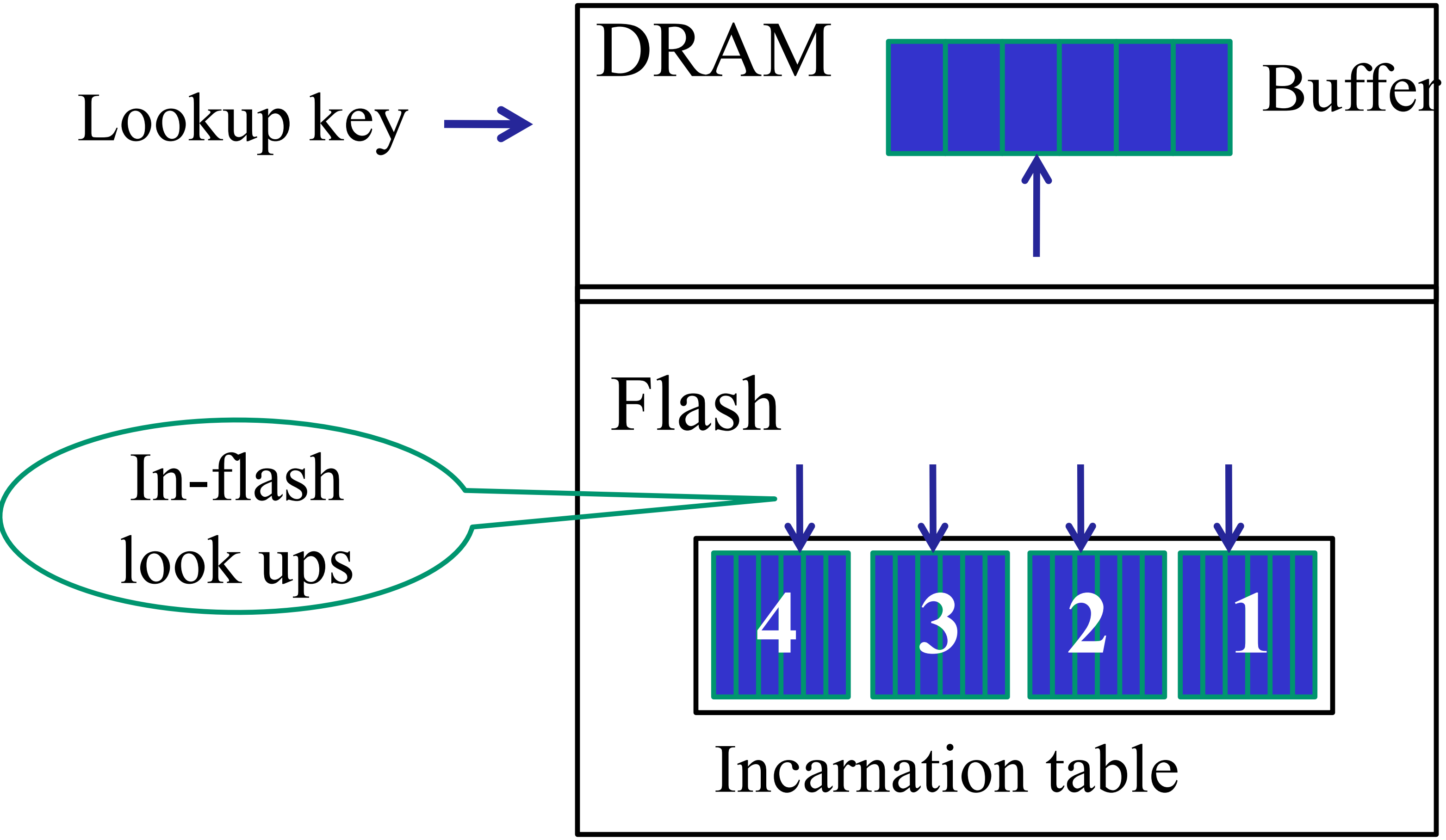
Incarnation: In-flash hash table

Two-level memory hierarchy



Net hash table is: buffer + all incarnations

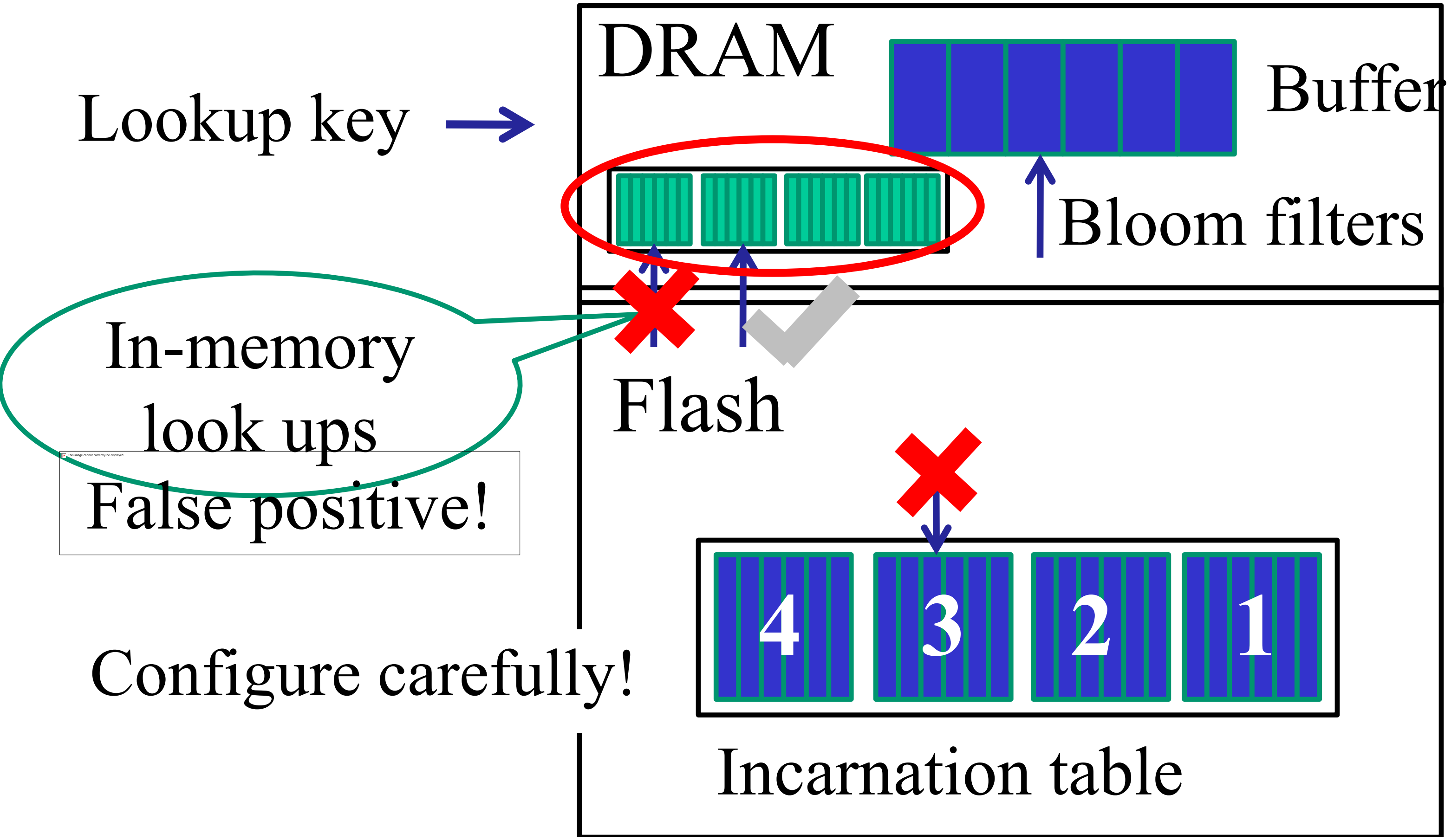
Lookups are impacted due to buffers



Multiple in-flash lookups. Can we limit to only one?

➔ *Use* ***Bloom Filters***

Bloom filters for optimizing lookups



Lookup key →

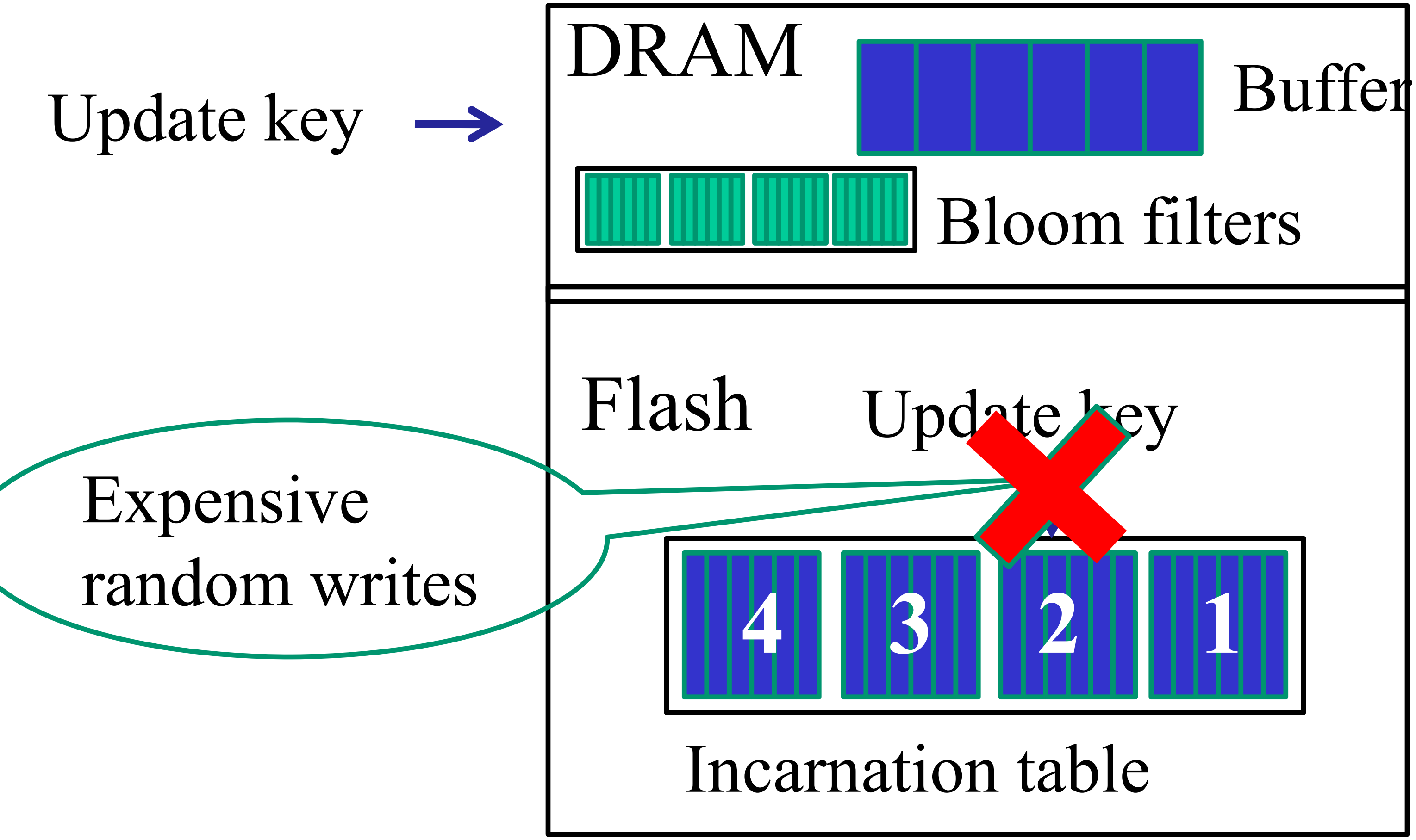
In-memory look ups

False positive!

Configure carefully!

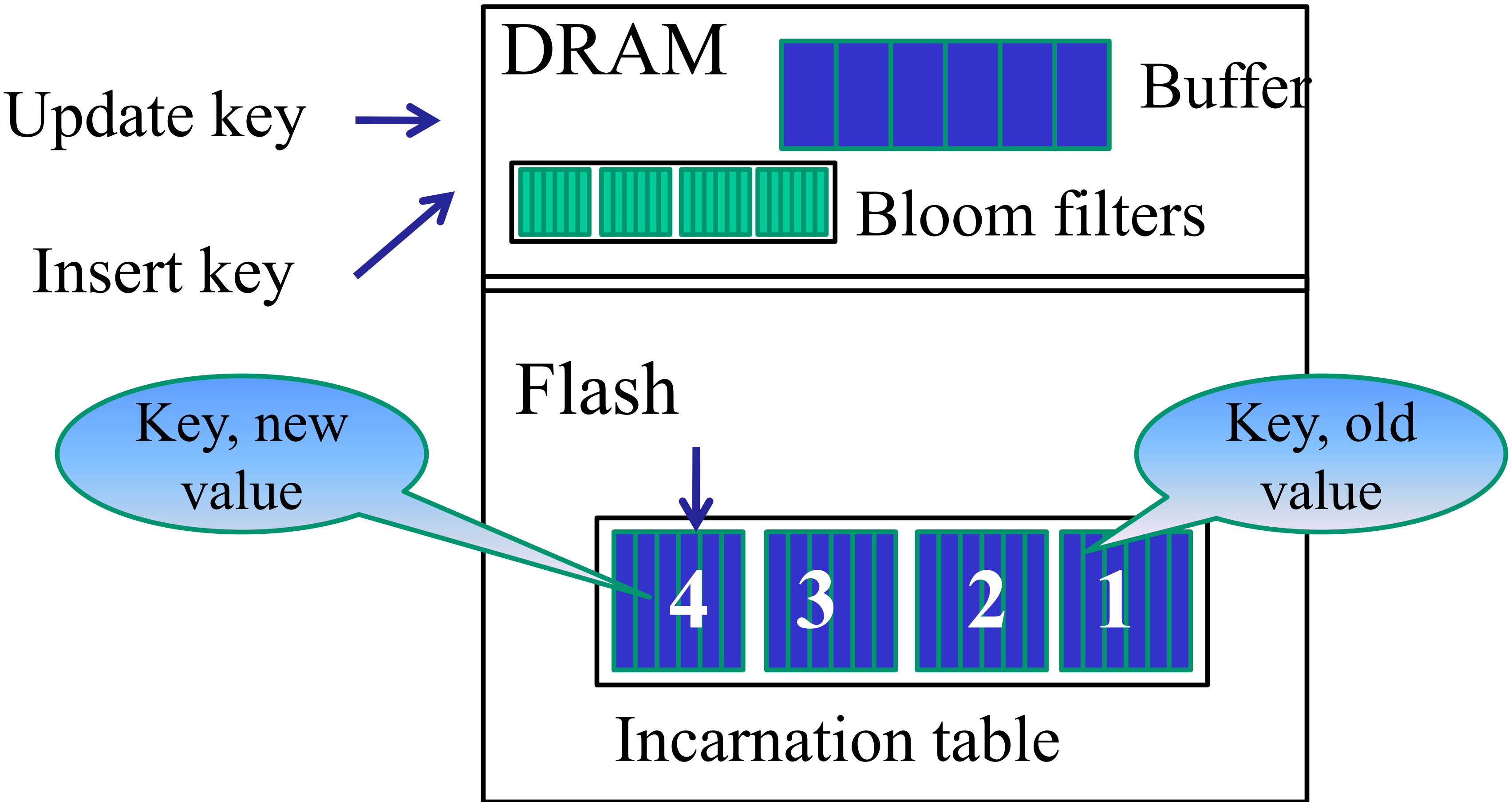
2 GB Bloom filters for 32 GB Flash for false positive rate < 0.01!

Update: naïve approach



Discard this naïve approach

Lazy updates

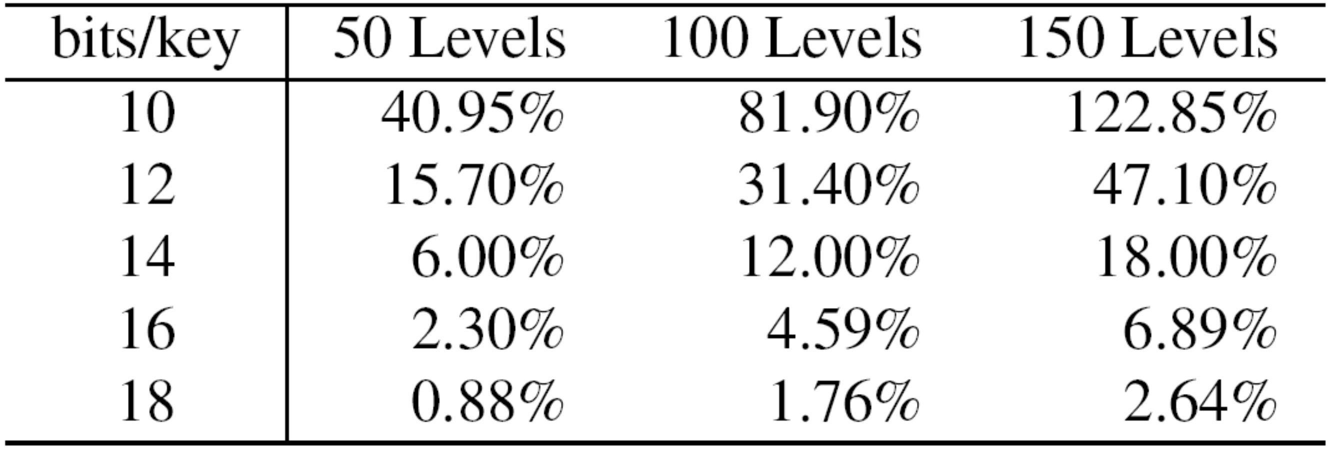


Lookups check latest incarnations first

Weaknesses of BufferHash

Excessively large number of (incarnations) levels makes BF less effective.

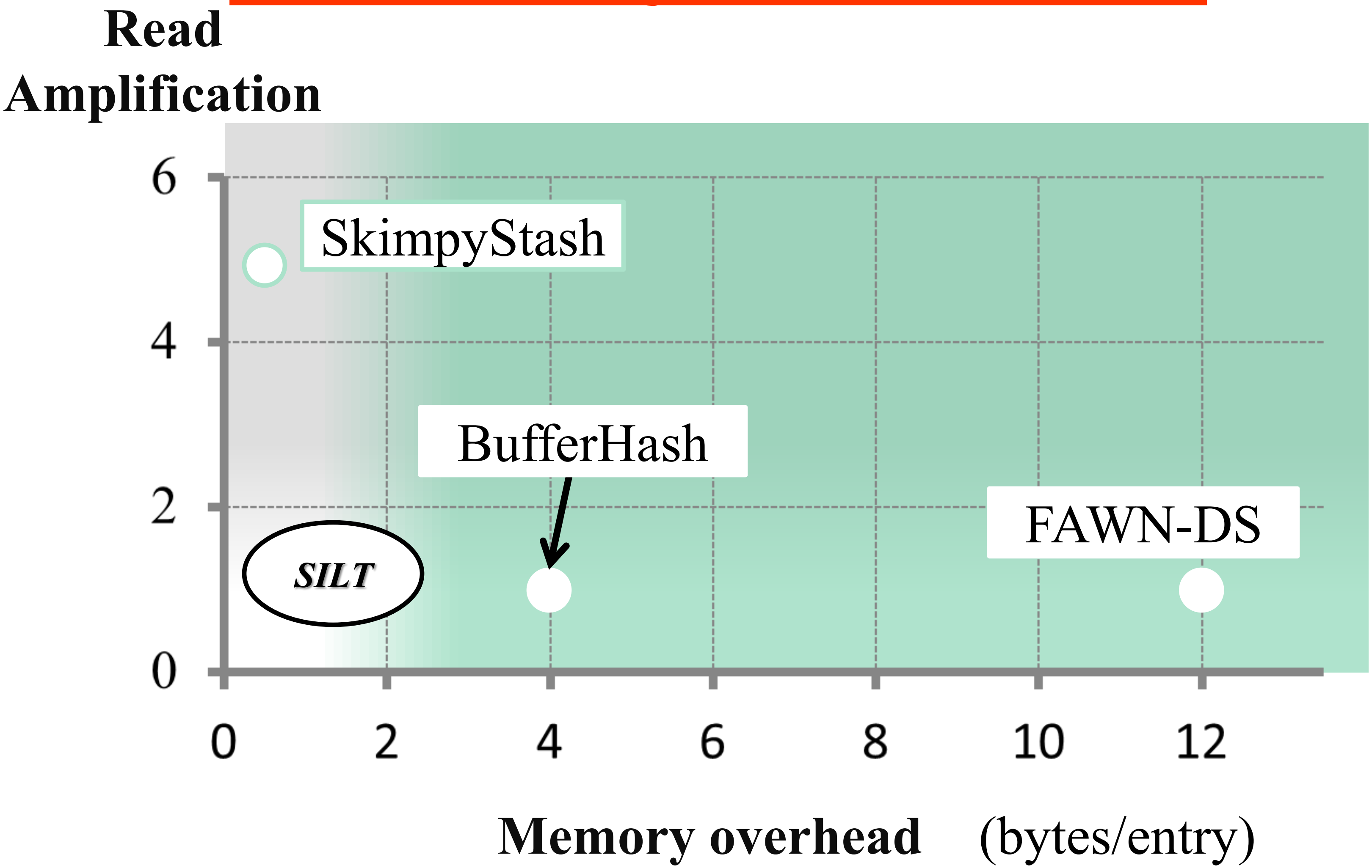
Searching in individual incarnations is not efficient.



bits/key	50 Levels	100 Levels	150 Levels
10	40.95%	81.90%	122.85%
12	15.70%	31.40%	47.10%
14	6.00%	12.00%	18.00%
16	2.30%	4.59%	6.89%
18	0.88%	1.76%	2.64%

Table 1: Bloom-filter false-positive rate.

Design IV: SILT with Levels of Dramatically-different Sizes



Hyeontaek Lim et al, **SILT: A Memory-Efficient, High-Performance Key-Value Store**, in SOSP'11.

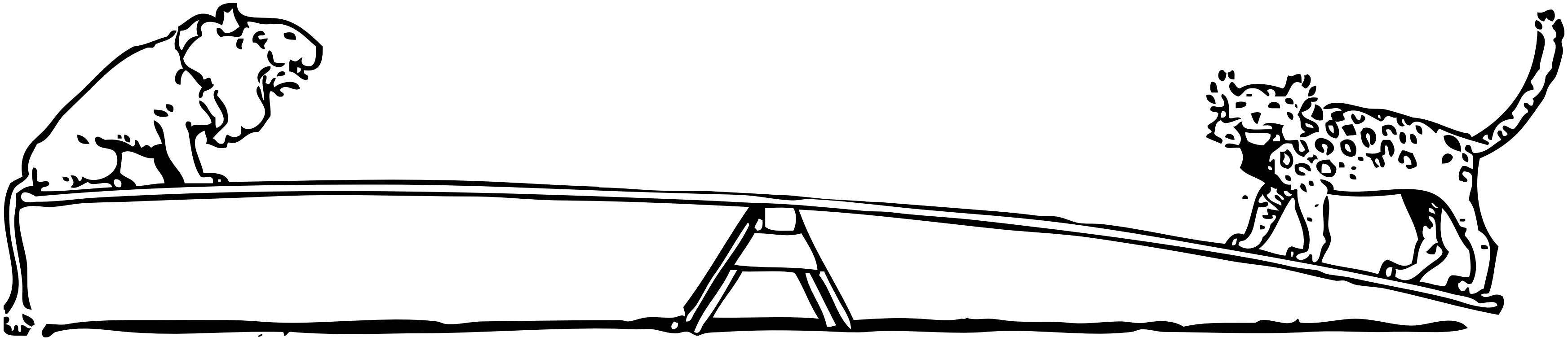
Seesaw Game?

FAWN-DS

How can we improve?

BufferHash

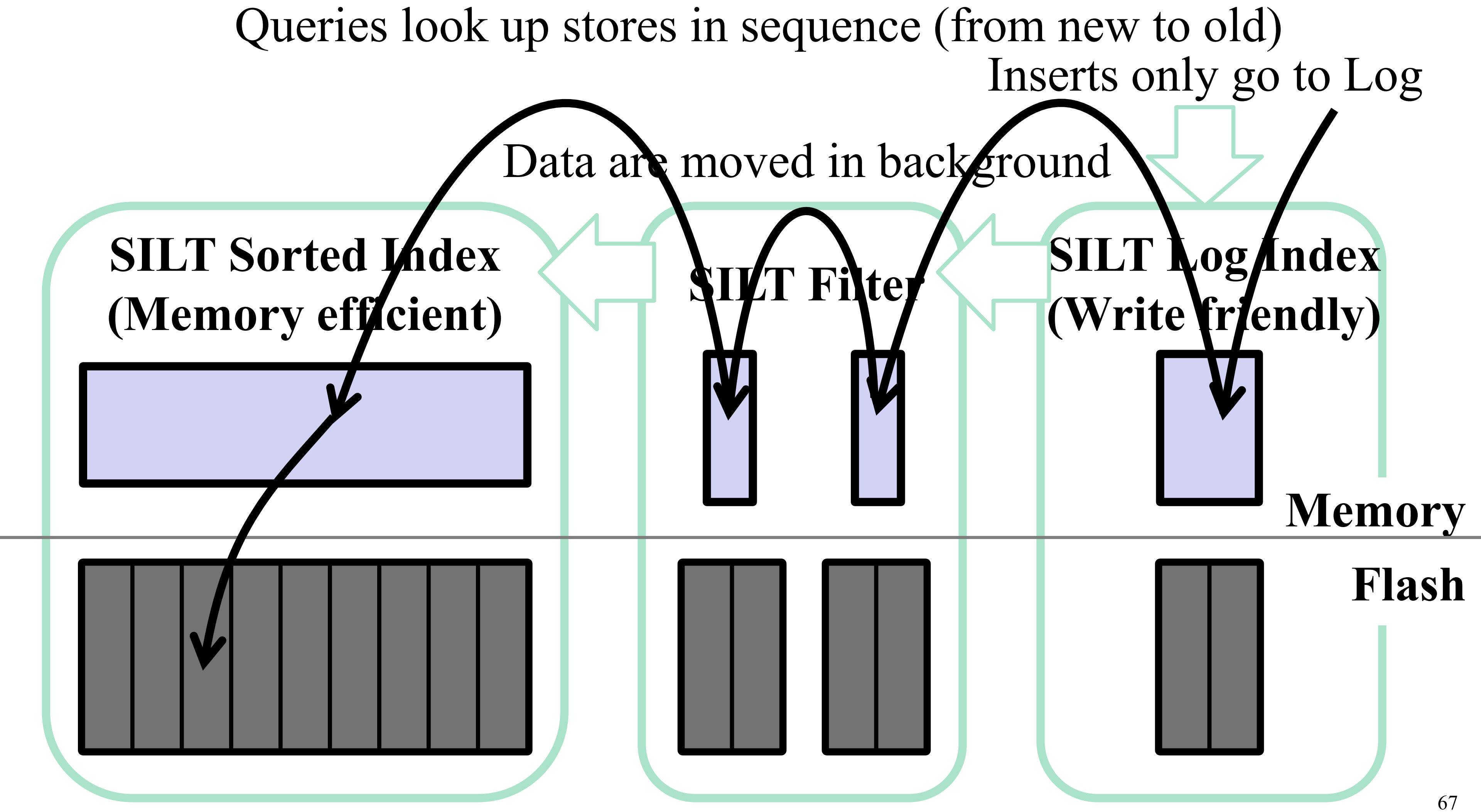
SkimpyStash



Memory efficiency

High performance

Solution Preview: (1) Three Stores with (2) New Index Data Structures



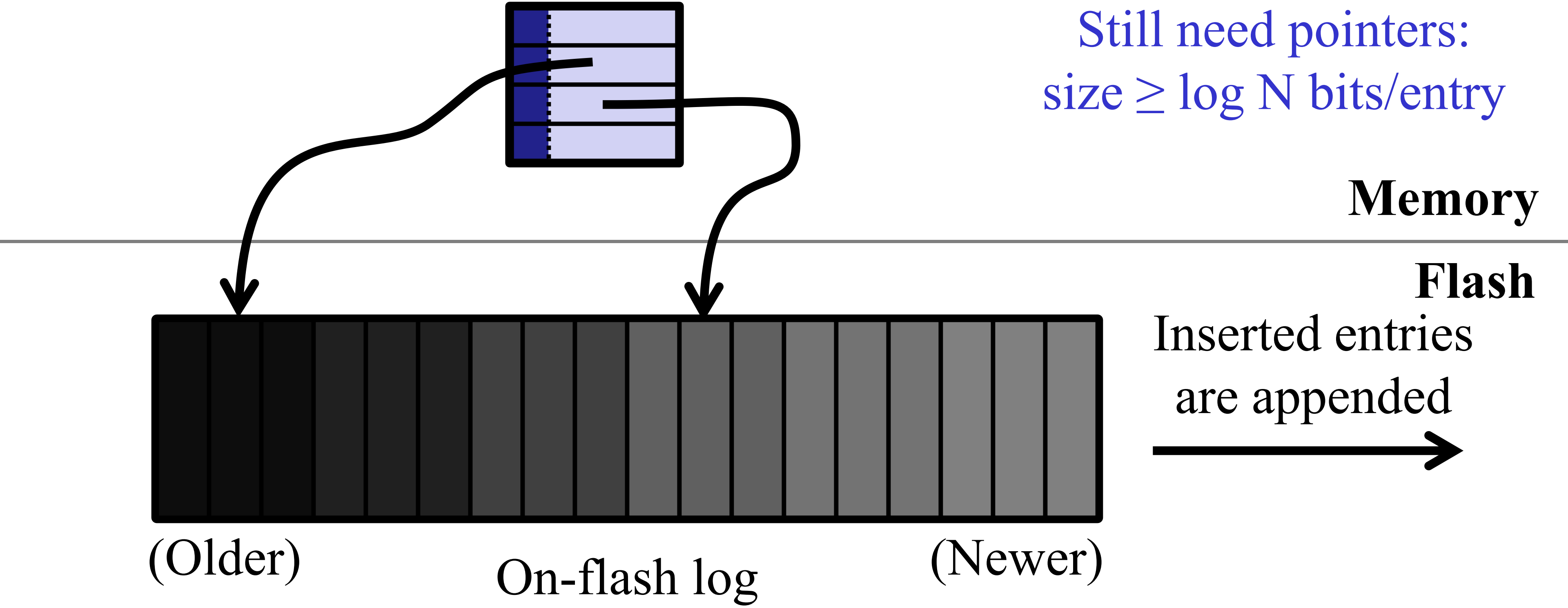
Queries look up stores in sequence (from new to old)

Inserts only go to Log

Data are moved in background

LogStore: No Control over Data Layout

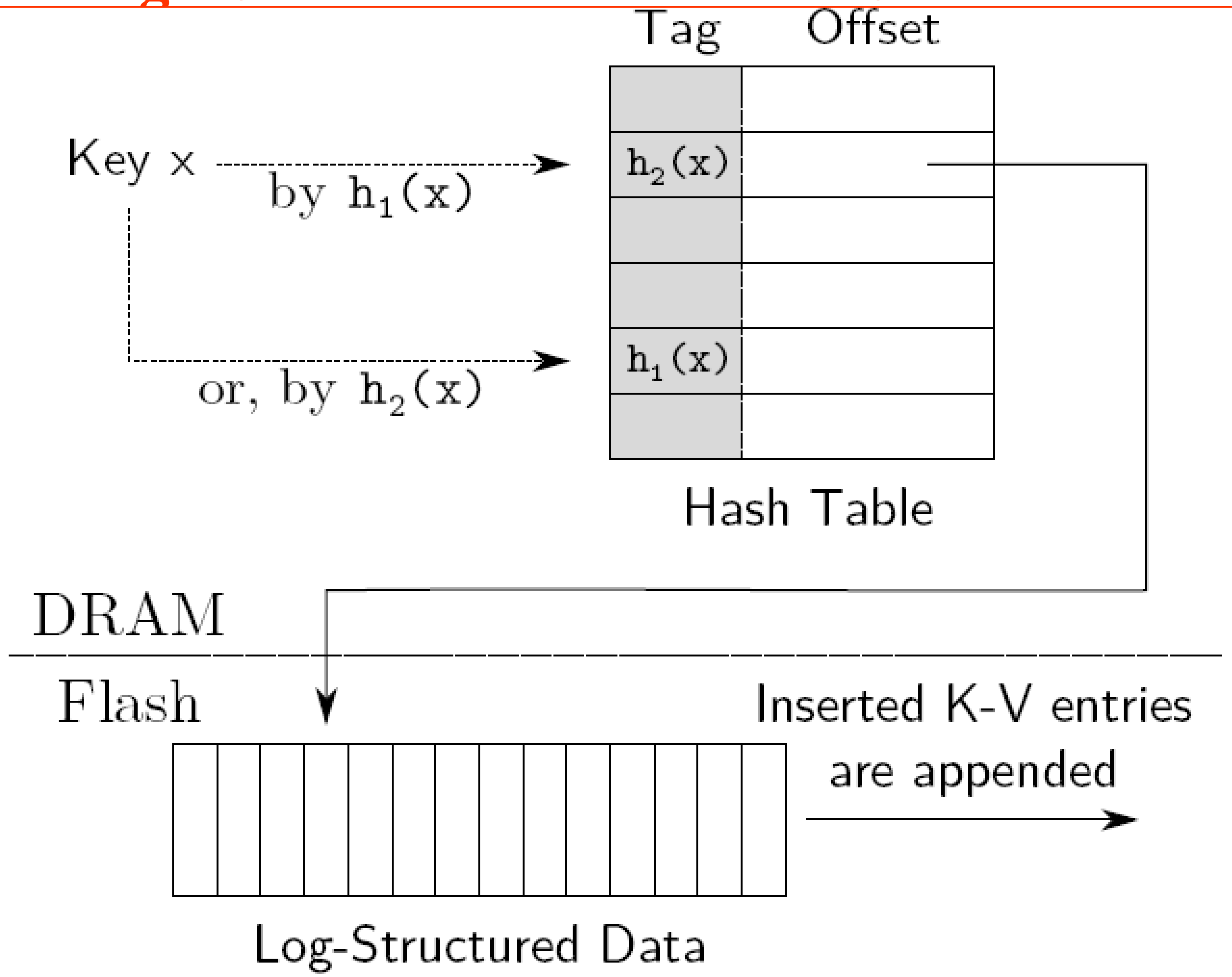
Naive Hashtable (48+ B/entry)
SILT Log Index (6.5+ B/entry)



Still need pointers:
size ≥ log N bits/entry

Memory overhead	Write amplification
6.5+ bytes/entry	**1**

LogStore: Using Cuckoo Hash to Embed Buckets into HT Directory



How to find the alternative slot for displacement by storing hash index in the tag?

HashStore: Remove in-memory HT (or the index)

HashStore saves memory over LogStore by eliminating the index and reordering the on-flash (key,value) pairs from insertion order to hash order.

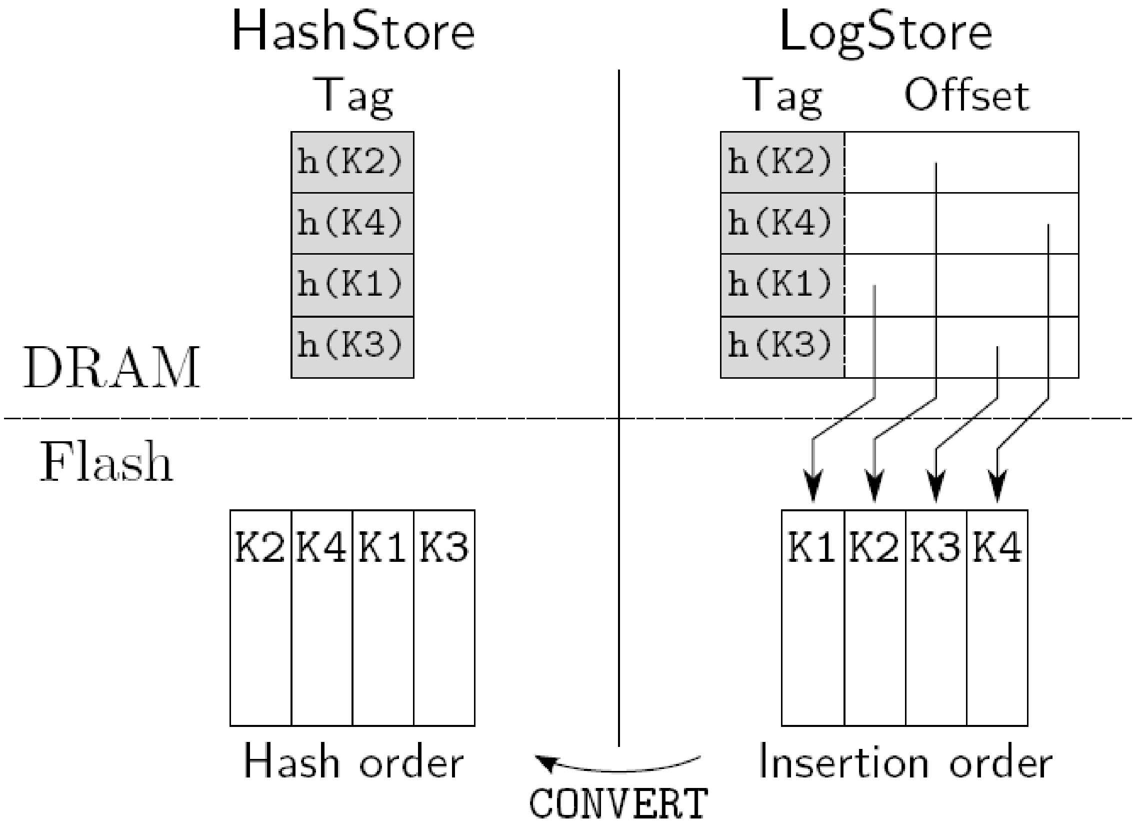
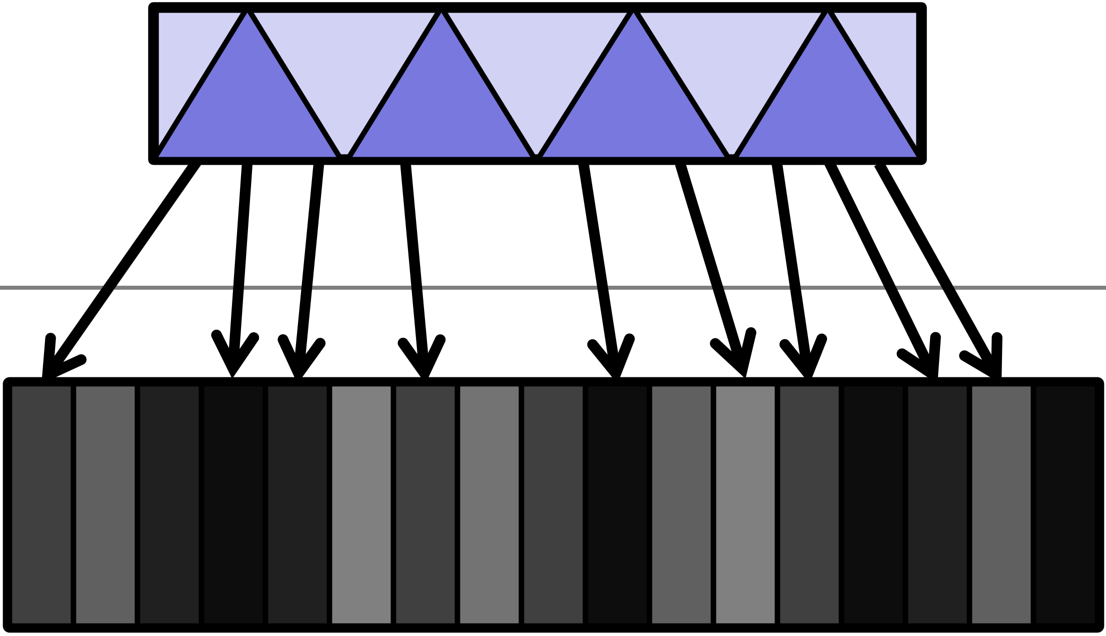


Figure 4: Convert a LogStore to a HashStore. Four keys K1, K2, K3, and K4 are inserted to the LogStore, so the layout of the log file is the insert order; the in-memory index keeps the offset of each key on flash. In HashStore, the on-flash data forms a hash table where keys are in the same order as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

SortedStore: Space-Optimized Layout

SILT Sorted Index (0.4 B/entry)



Memory

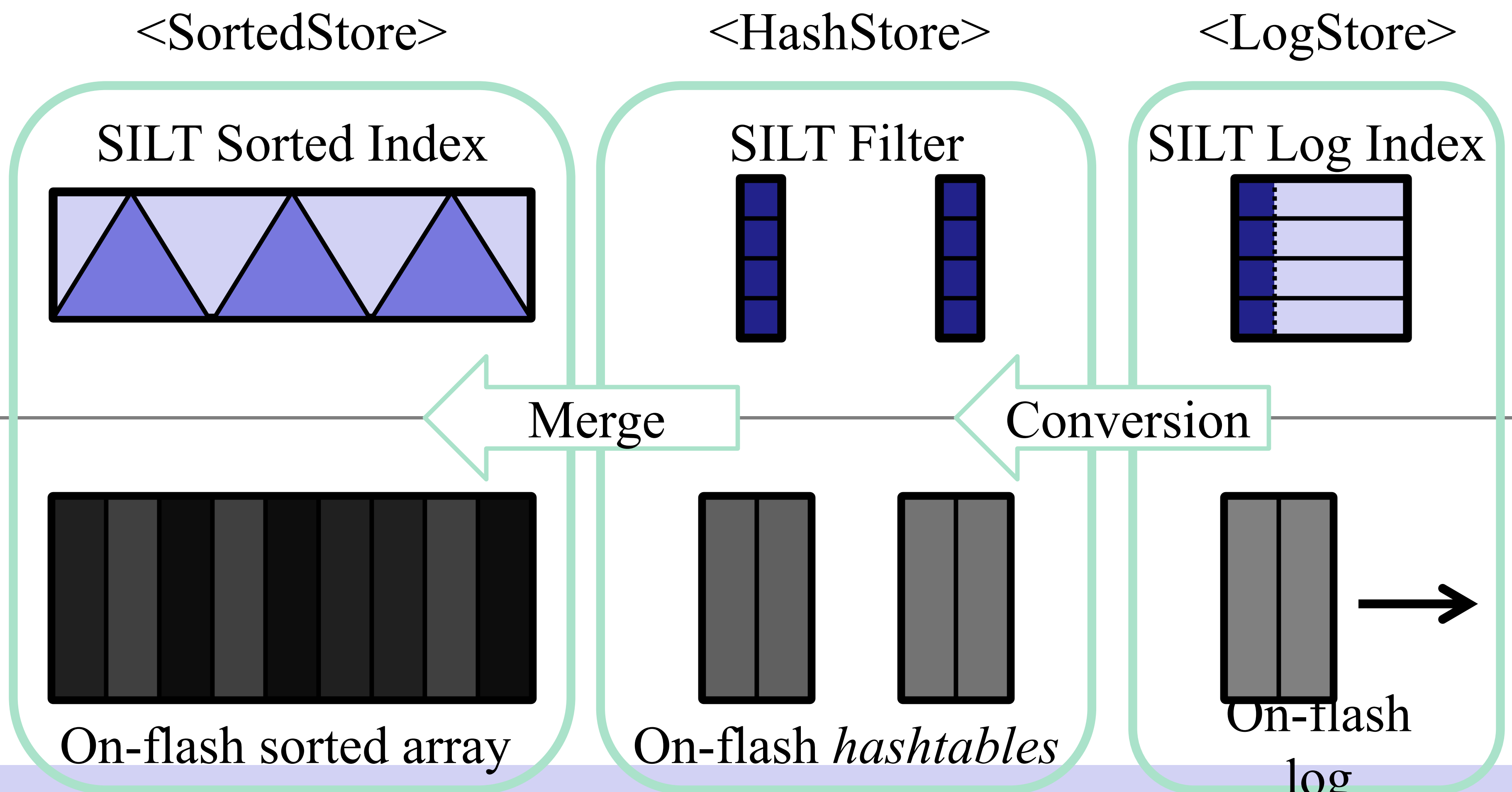
Flash

Need to perform bulk-insert to amortize cost

On-flash sorted array

To merge HashStore entries into the SortedStore, SILT must generate a new SortedStore

SILT's Design (Recap)



<SortedStore>

<HashStore>

<LogStore>

SILT Sorted Index

SILT Filter

SILT Log Index

Merge

Conversion

On-flash sorted array

On-flash *hashtables*

On-flash log

Memory overhead

0.7 bytes/entry

Read amplification

1.01

Write amplification

5.4

Any Issue with SILT?

- SILT provides both **memory-efficient** and **high-performance** key-value store
  - Multi-store approach
  - Entropy-coded tries
  - Partial-key cuckoo hashing

- The weakness: Write amplification is way too high!

Design V: Google’s BigTable and LevelDB

- A multi-layered LSM-tree structure
- Progressively sort data for small memory demand
- Small number of levels for effective BF use.

Chang, et al., **Bigtable: A Distributed Storage System for Structured Data** in OSDI’06.

Design V: Google's BigTable and LevelDB to Scale Data Store

Scale Problem

- Lots of data
- Millions of machines
- Different project/applications
- Hundreds of millions of users

Storage for (semi-)structured data

No commercial system big enough

- Couldn't afford if there was one

Low-level storage optimization helps performance significantly

Much harder to do when running on top of a database layer

Bigtable

Fault-tolerant, persistent

Scalable

- Thousands of servers
- Terabytes of in-memory data
- Petabyte of disk-based data
- Millions of reads/writes per second, efficient scans

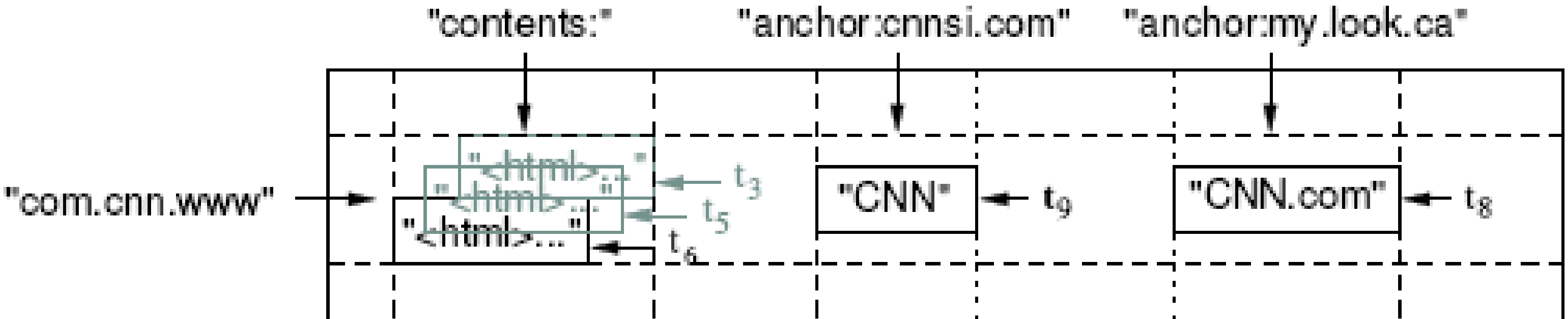
Self-managing

- Servers can be added/removed dynamically
- Servers adjust to load imbalance

Data model: a big map

- <Row, Column, Timestamp> triple for key
  - Each value is an uninterpreted array of bytes
- Arbitrary "columns" on a row-by-row basis
  - Column family:qualifier. a small number of families and large number of columns
  - Lookup, insert, delete API

Each read or write of data under a single row key is atomic



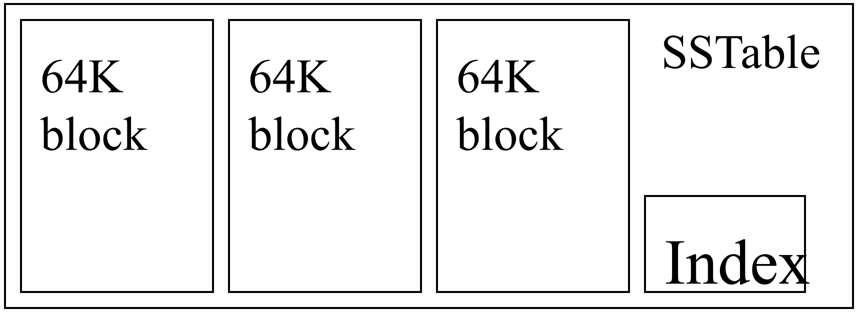
SSTable

Immutable, sorted file of key-value pairs

Chunks of data plus an index

- Index is of block ranges, not values
- Index loaded into memory when SSTable is opened
- Lookup is a single disk seek

Alternatively, client can load SSTable into memory

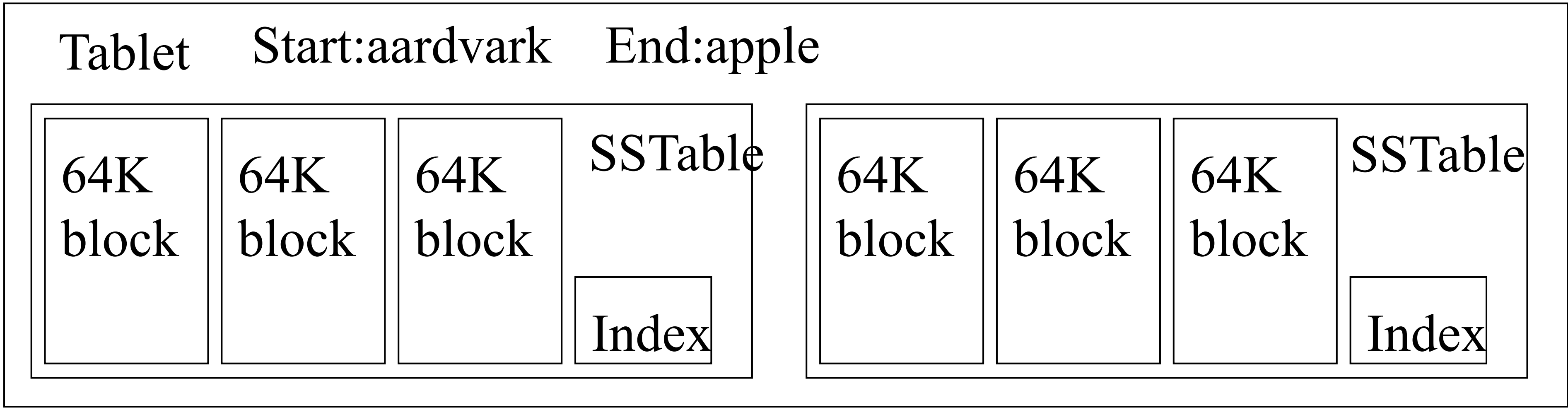


Tablet

Contains some range of rows of the table

Unit of distribution & load balance

Built out of multiple SSTables



Tablet Start:aardvark End:apple

64K block

64K block

64K block

SSTable

Index

64K block

64K block

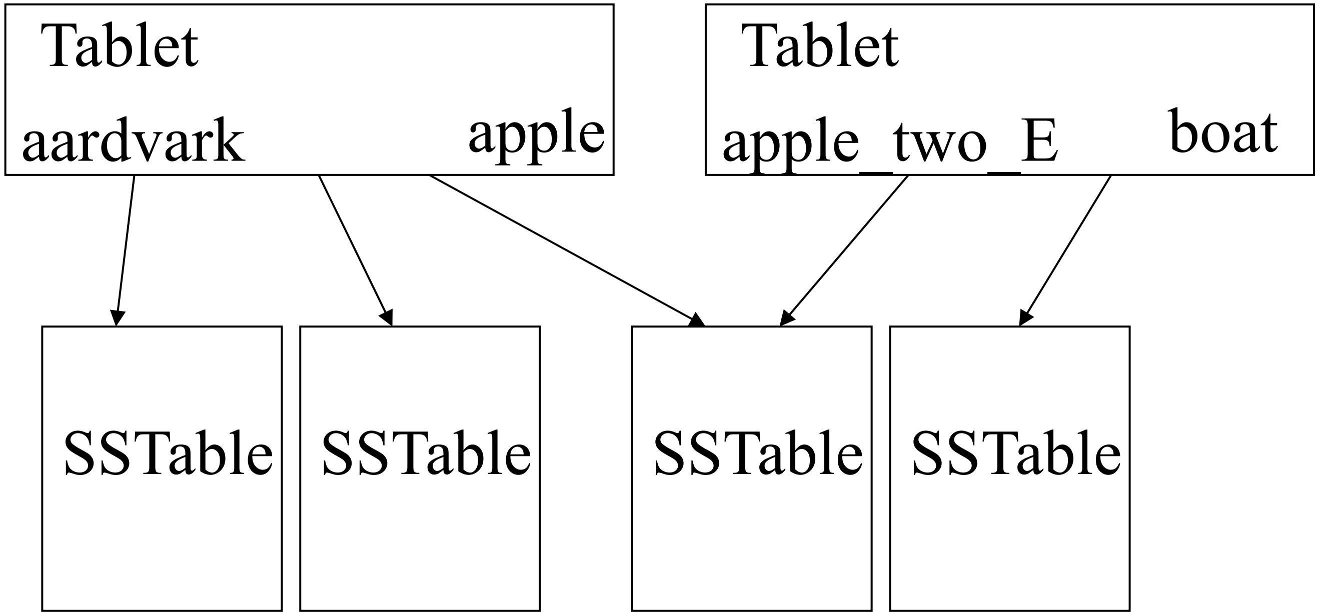
64K block

SSTable

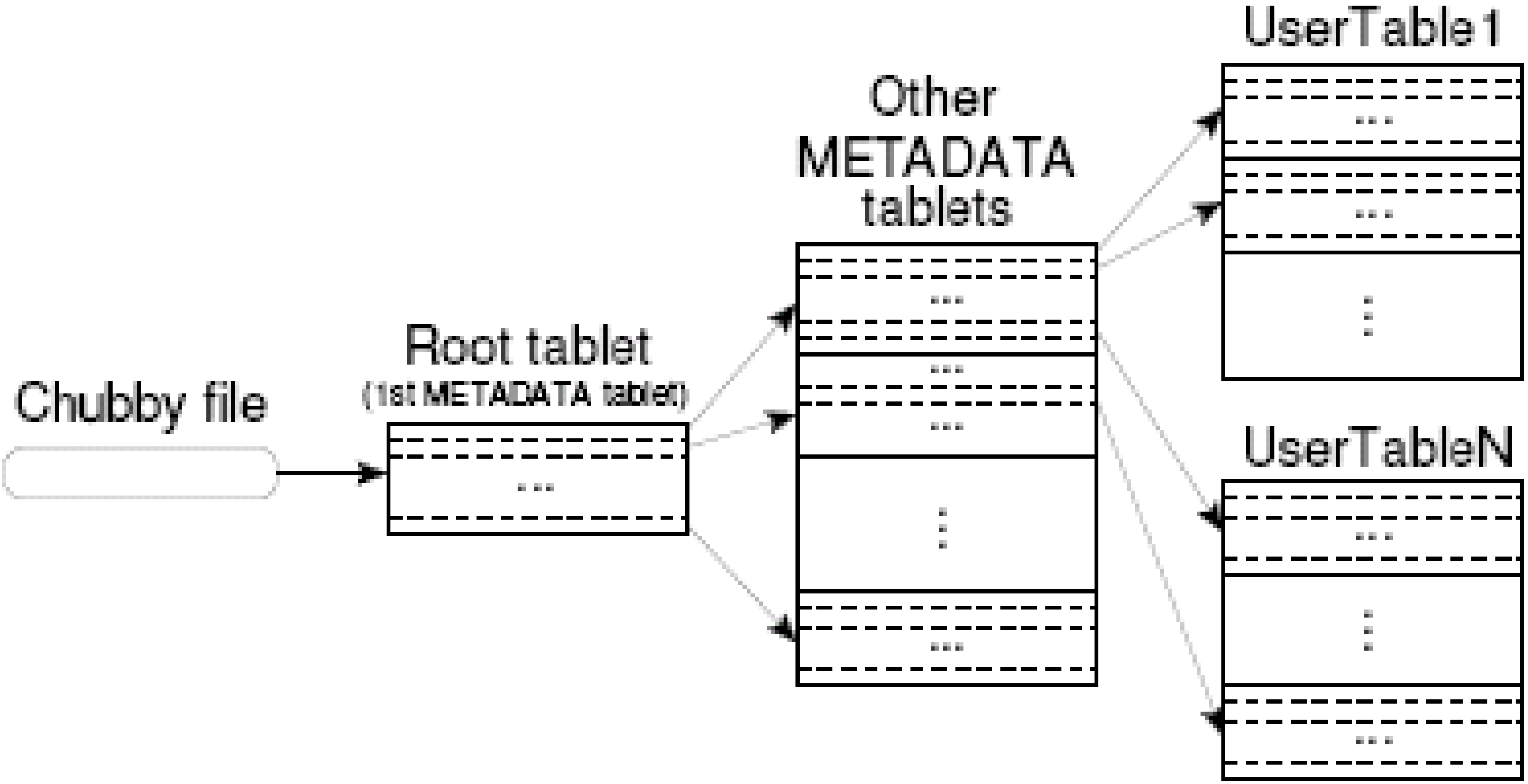
Index

Table

Multiple tablets make up the table



Finding a tablet



- Client library caches tablet locations

Servers

Tablet servers manage tablets, multiple tablets per server. Each tablet is 100-200 MBs

- Each tablet lives at only one server
- Tablet server splits tablets that get too big

Master responsible for load balancing and fault tolerance

- Use Chubby to monitor health of tablet servers, restart failed servers
- GFS replicates data. prefer to start tablet server on same machine that the data is already at

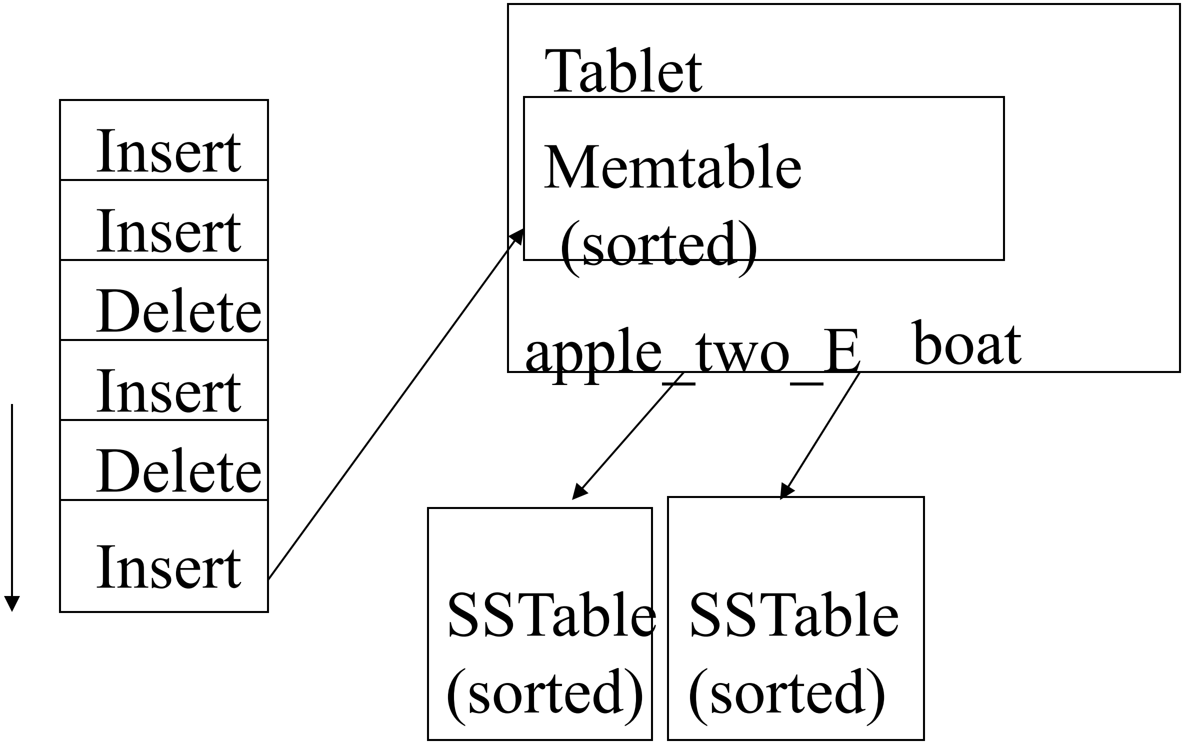
Editing/Reading a table

Mutations are committed to a commit log (in GFS)

Then applied to an in-memory version (memtable)

Reads applied to merged view of SSTables & memtable

Reads & writes continue during tablet split or merge



Bigtable Tablet

LevelDB is similar to a single Bigtable tablet

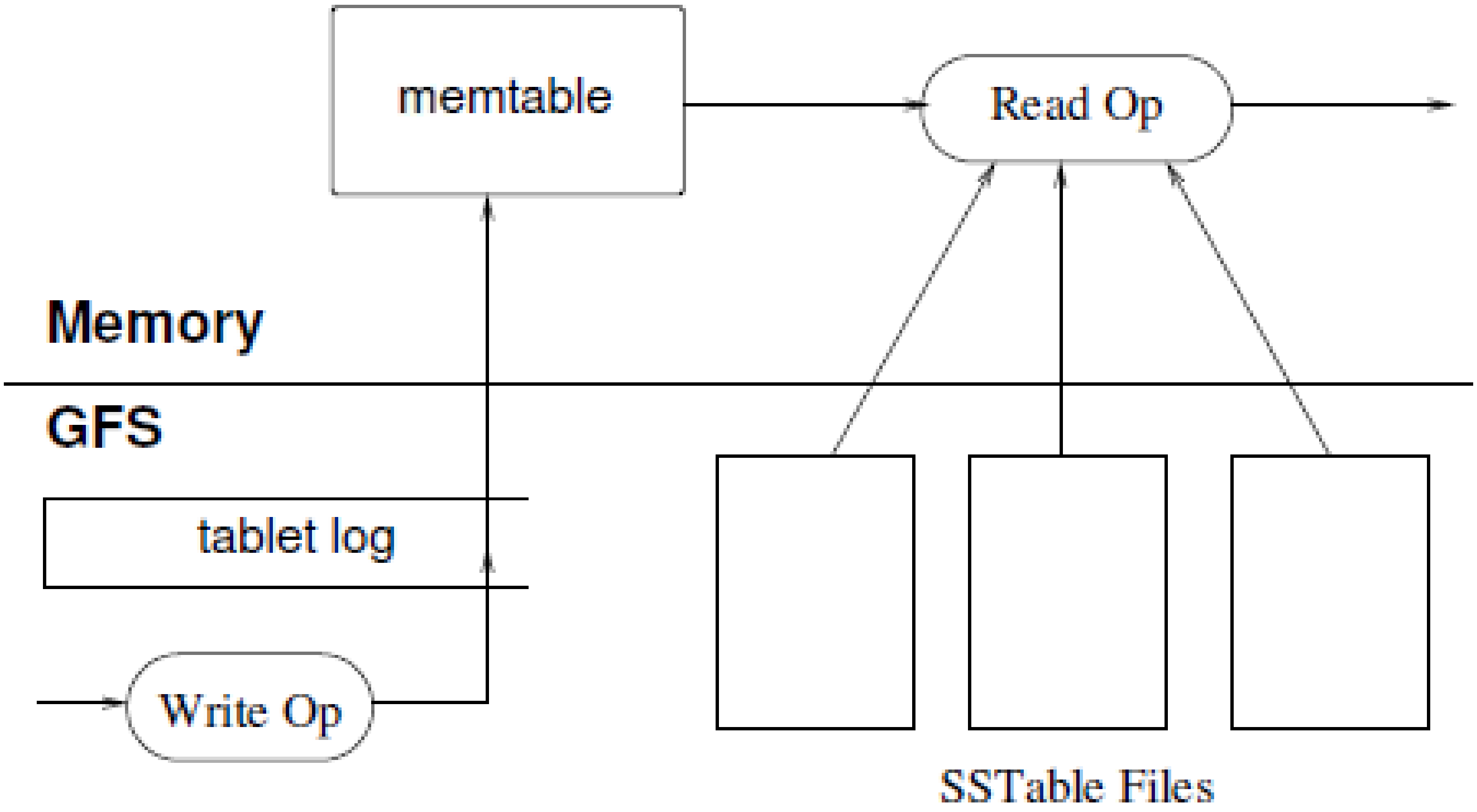


Figure 5: Tablet Representation

Compactions

Minor compaction – convert a full memtable into an SSTable, and start a new memtable

- Reduce memory usage
- Reduce log traffic on restart

Major compaction

- Merging compaction that results in only one SSTable
- No deletion records, only live data

Management of SSTable -- LSM-tree

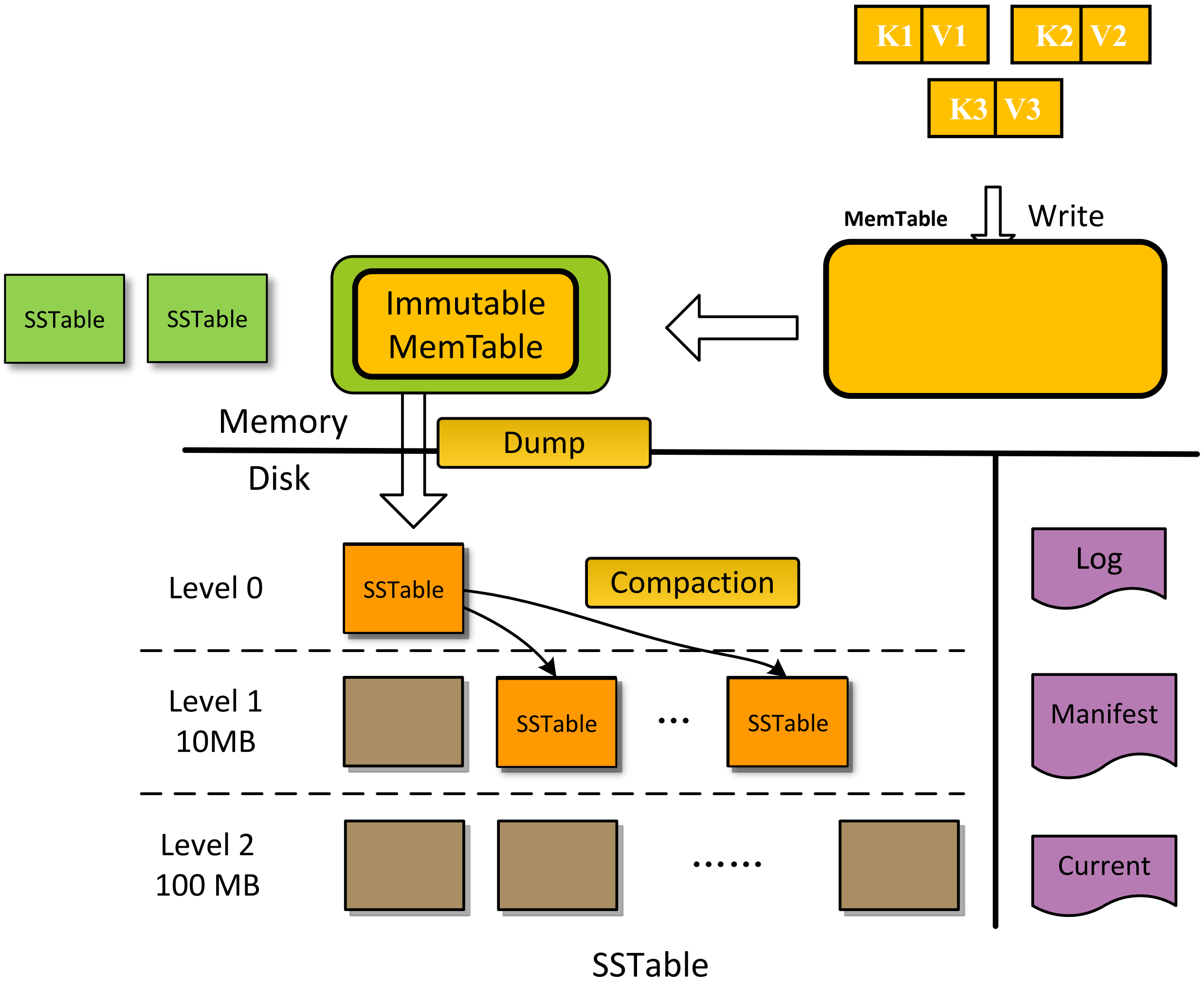
Log-Structured Merge-Tree (LSM-tree)

- Optimized for fast random updates, inserts and deletes with moderate read performance.

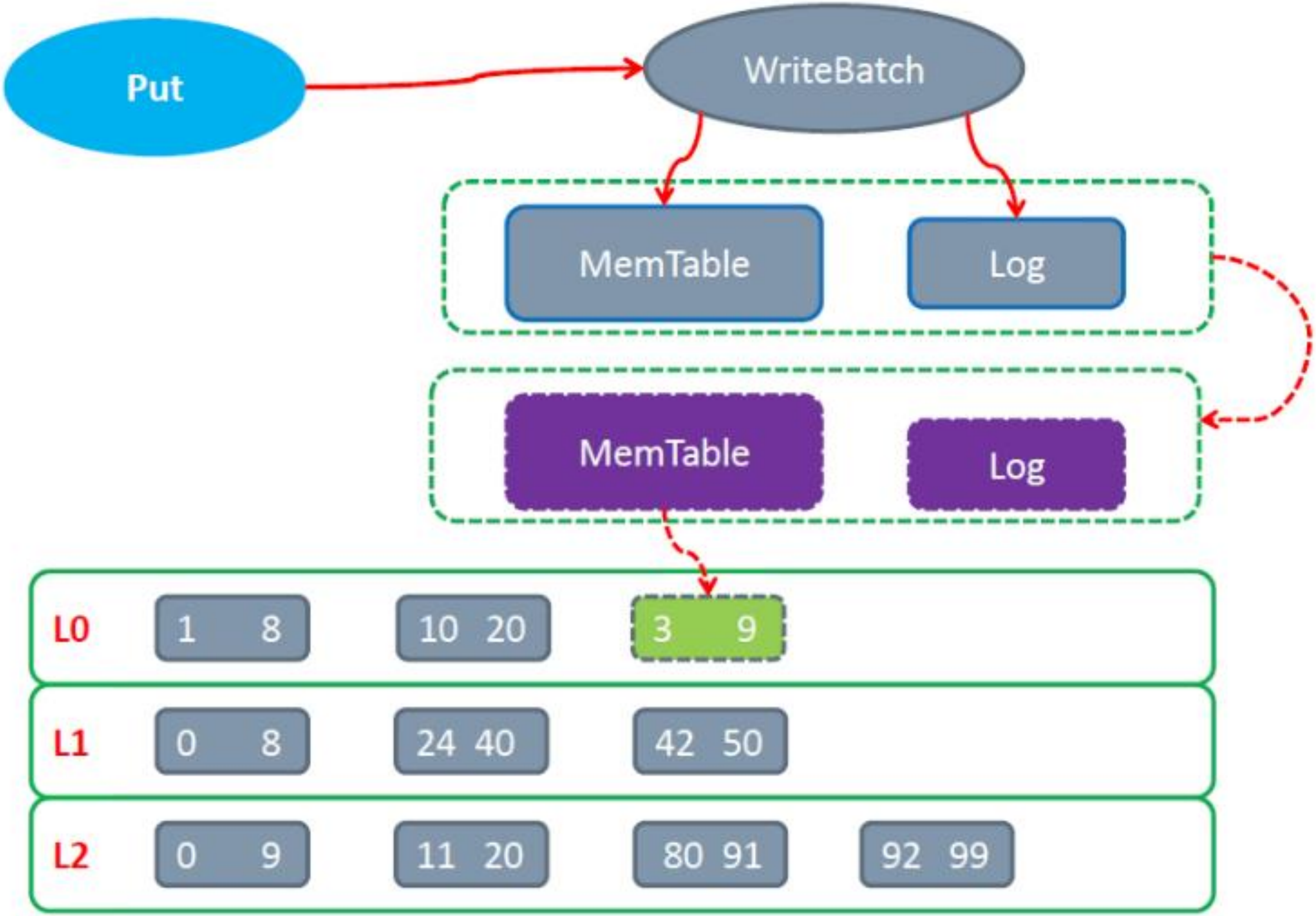
- Convert the random writes to sequential writes
  - Accumulate recent updates in memory
  - Flush the changes to disks sequentially in batches
  - Merge on-disk components periodically

- At the expense of read performance

Google's LevelDB: Progressively Increasing Level Size



LevelDB Write Flow



Put

WriteBatch

MemTable

Log

MemTable

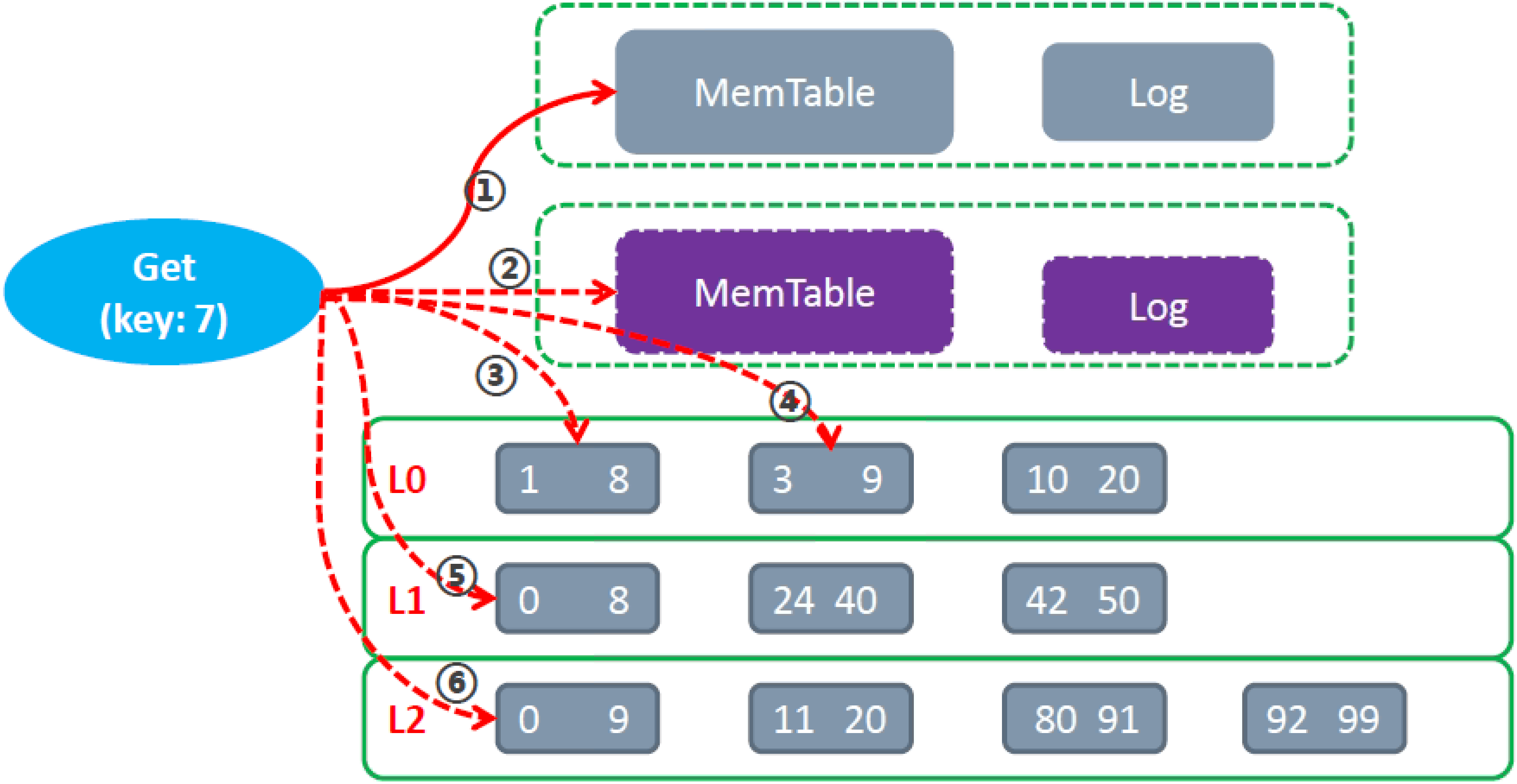
Log

L0 1 8 10 20 3 9

L1 0 8 24 40 42 50

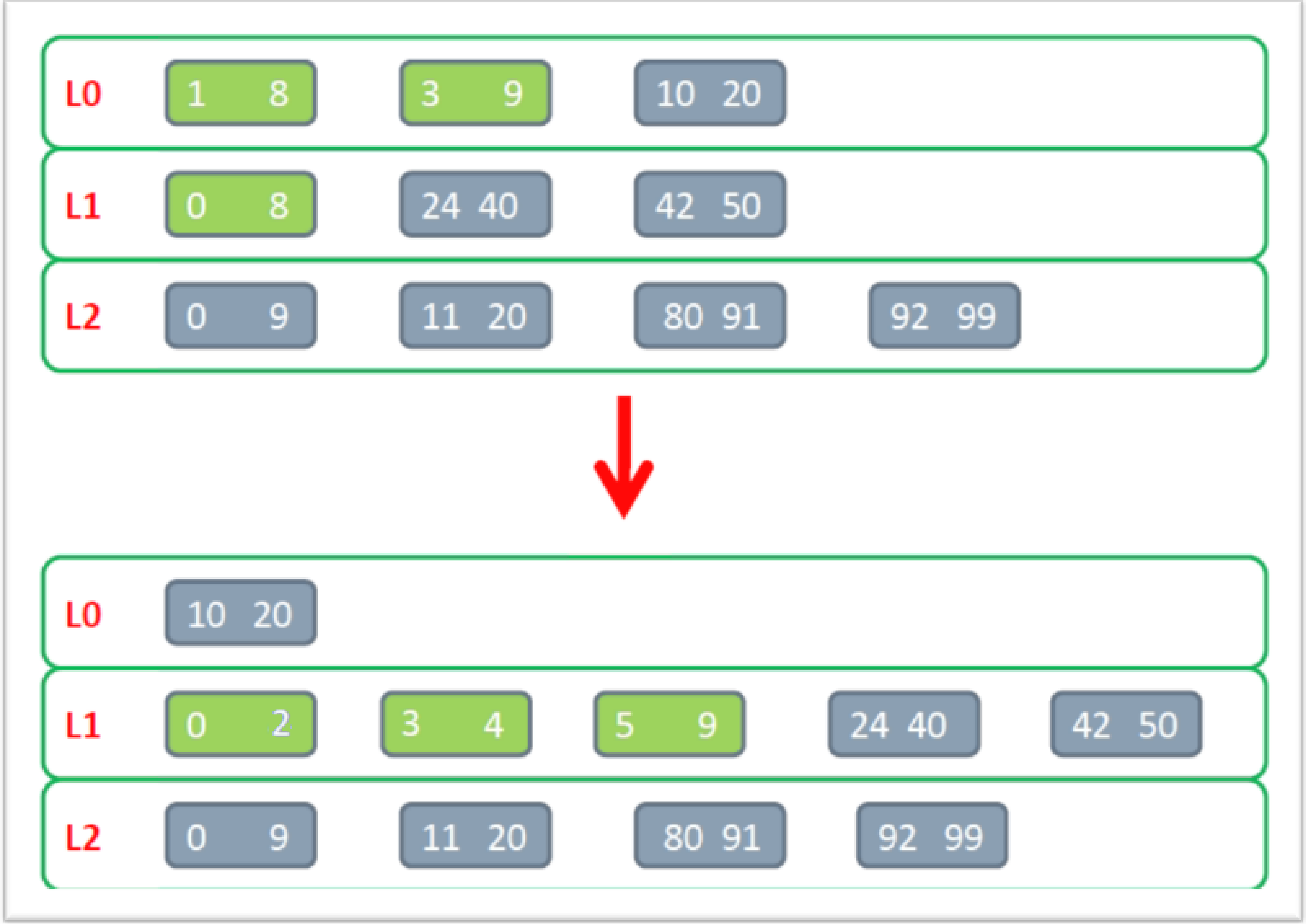
L2 0 9 11 20 80 91 92 99

LevelDB Read Flow

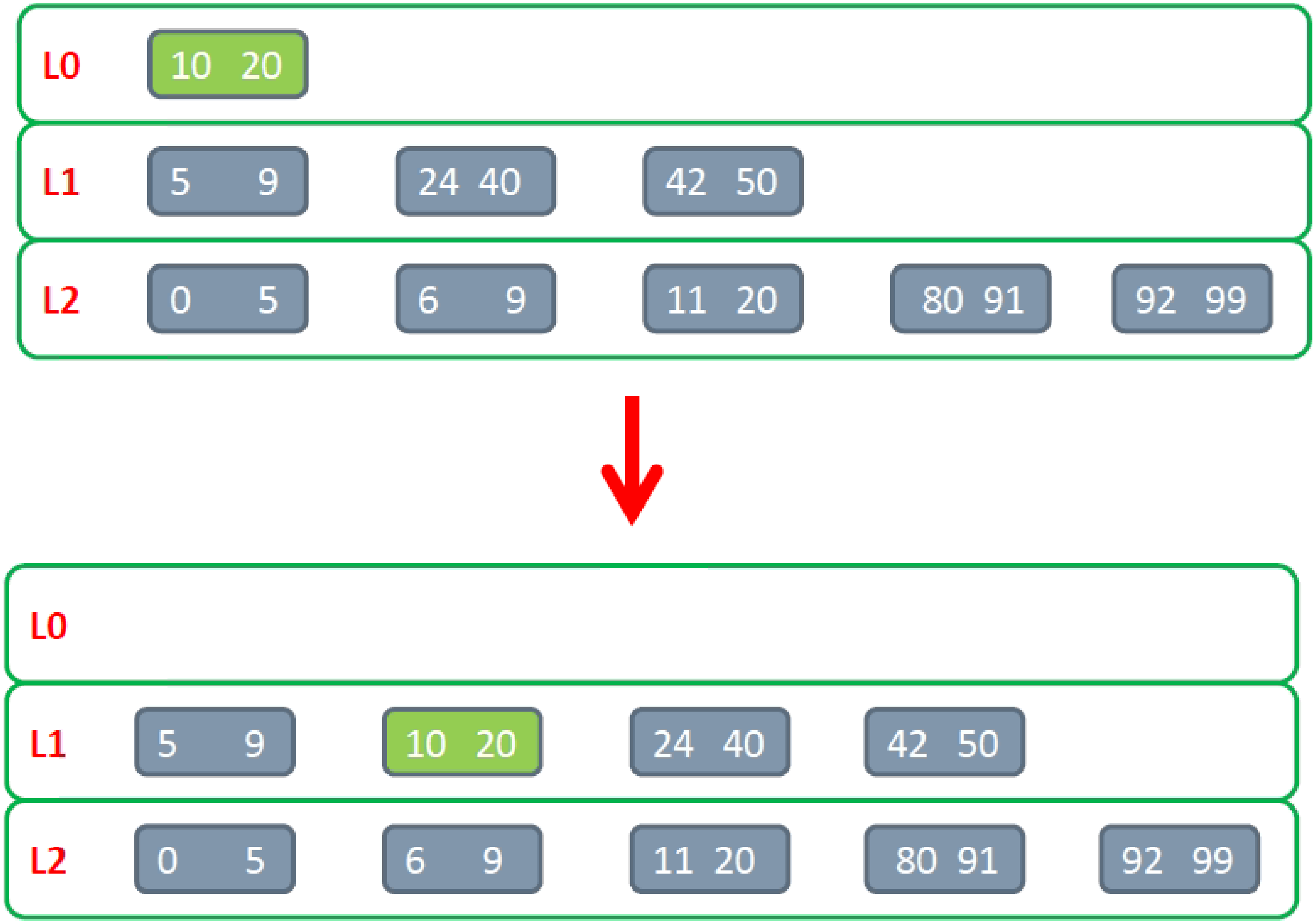


Bloom Filter

LevelDB Compact (L0/L1)



LevelDB Compact (L0/L1 Move)



Design VI: LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items

Current KV storage systems have one or more of the issues:

(1)very high data write amplifications;

(2)Large index set; and

(3)dramatic degradation of read performance with overspill index out of the memory.

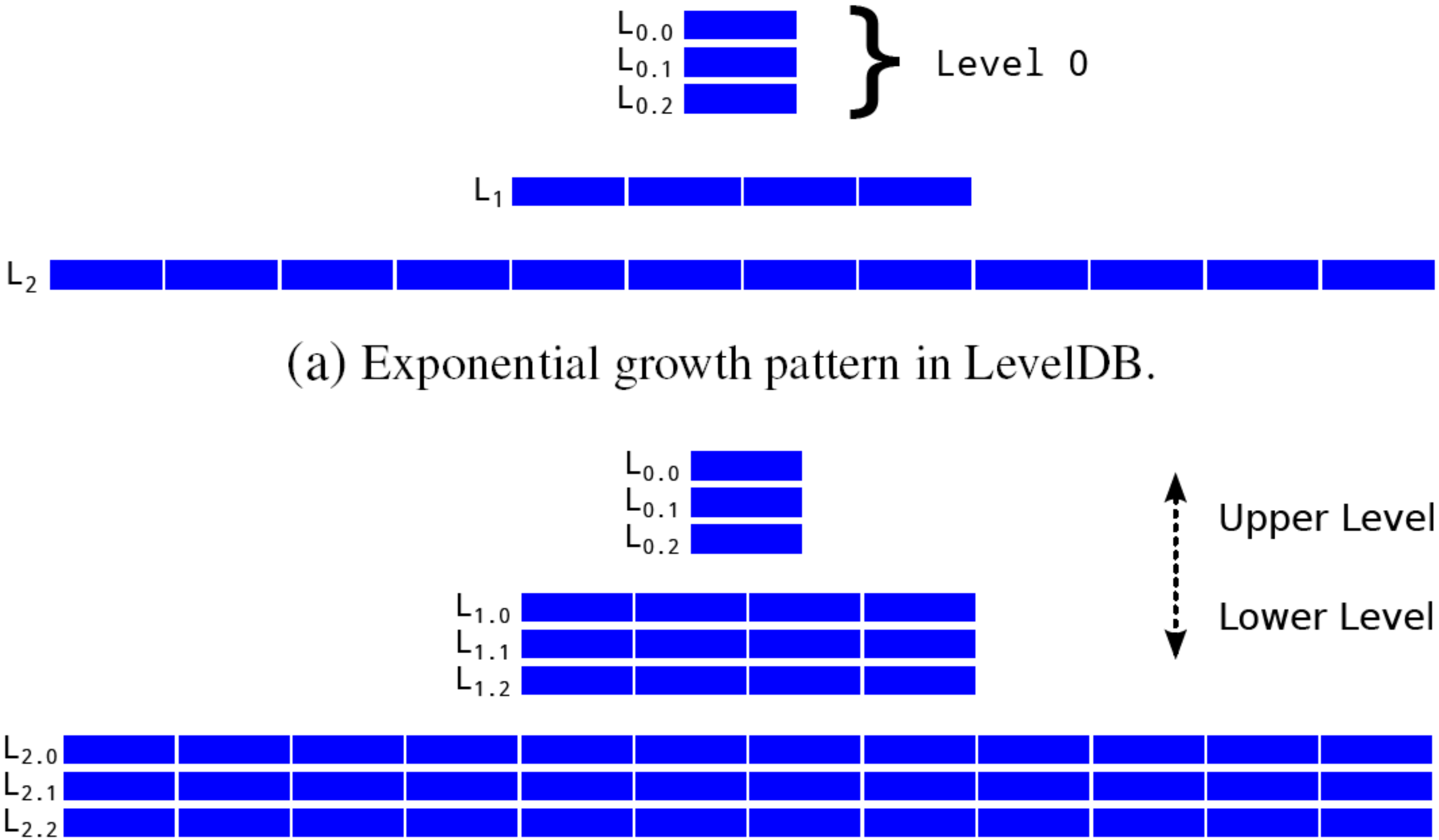
LSM-trie:

(1)substantially reduces metadata for locating items,

(2) reduces write amplification by an order of magnitude,

(3) needs only at most two disk accesses with each KV read even when only less than 10% of metadata (Bloom Filters) can be held in the memory

LSM-trie: A New Level Growth Pattern



(a) Exponential growth pattern in LevelDB.

(b) Alternating use of linear and exponential growth patterns.

Figure 1: Using multi-level structure to grow an LSM-tree store. Each solid rectangle represents an SSTable.

LSM-trie: Minimize Write Amplification

- To enable linear growth pattern, the SSTables in one column of sub-levels of a level must have the same key range.
- KV items are hashed into and organized in the store.
- 160b-Hash key is generated with SHA-1 for uniform distribution.

The Trie Structure

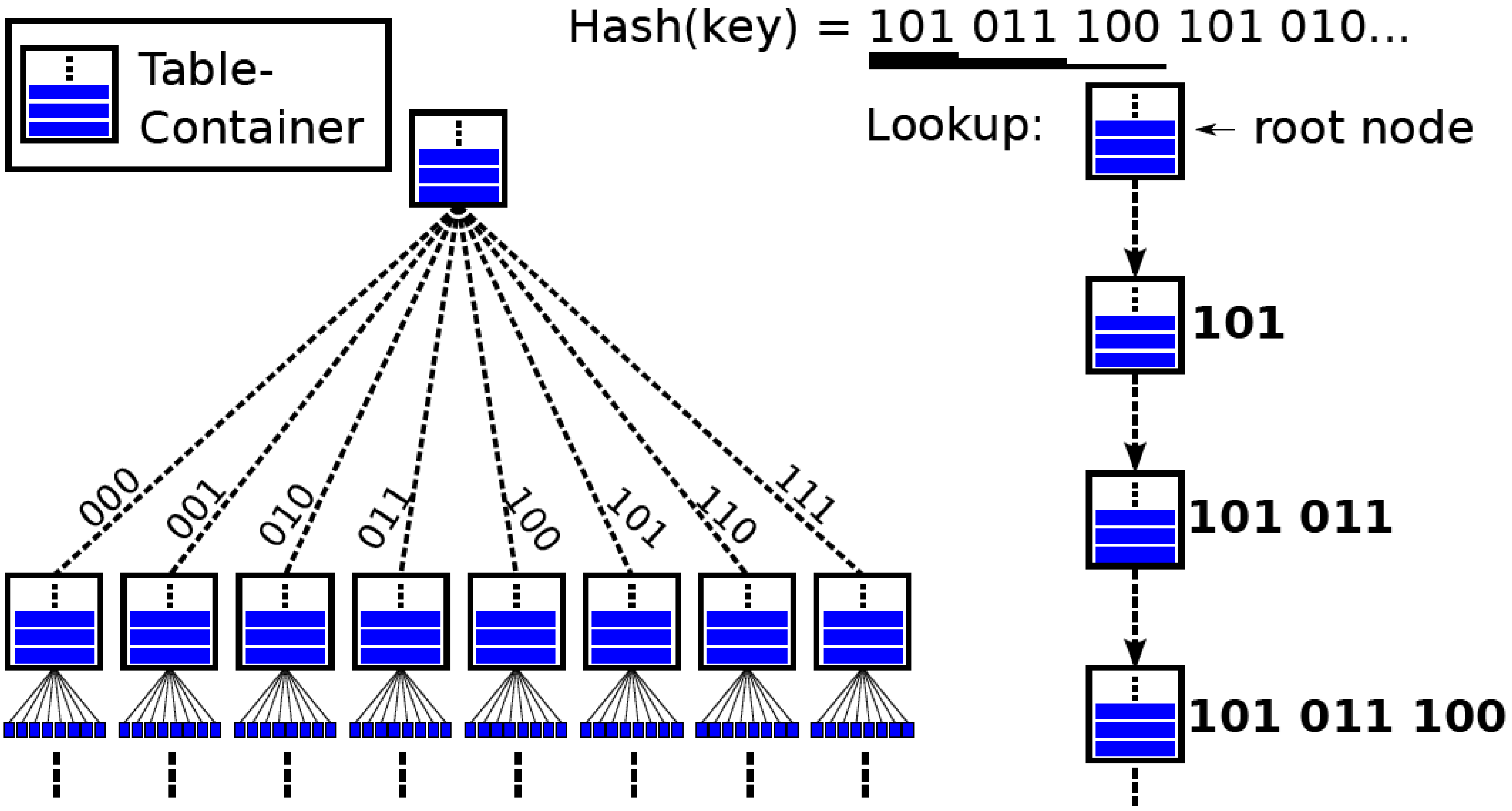


Figure 2: A trie structure for organizing SSTables. Each box represents a table container, which contains a pile of SSTables.

Compaction in LSM-trie

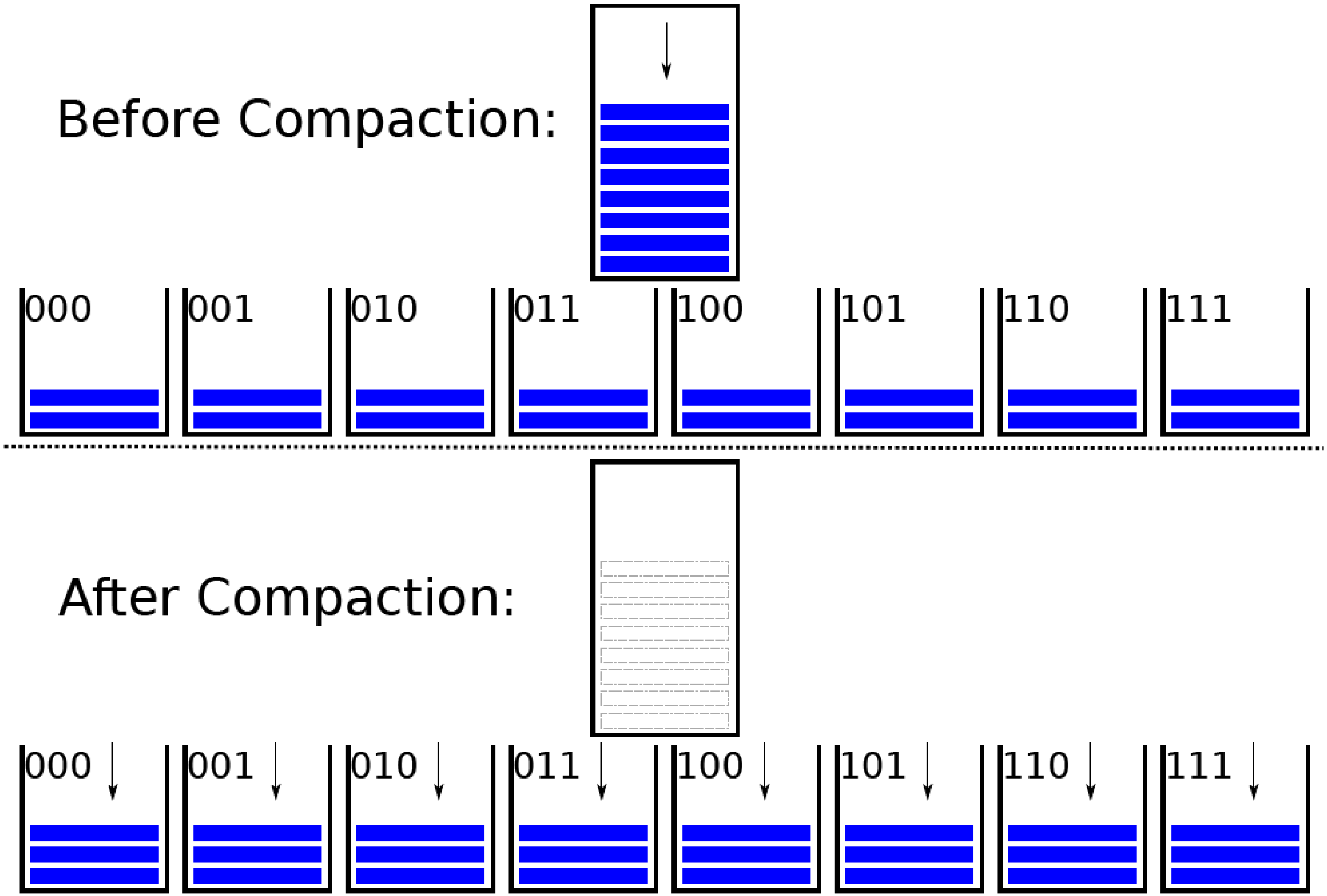


Figure 3: A compaction operation in the trie.

How about out-of-core the Bloom Filters?

- ❑ To scale the store to very large size in terms of both capacity and KV-item count (e.g, a 10 TB store containing 100 billion of 100-byte KV items). A big challenge on designing such a large-scale store is the management of its metadata that often have to be out of core (the DRAM).
- ❑ LSM-trie hashes keys into buckets within each SSTable (Htable).

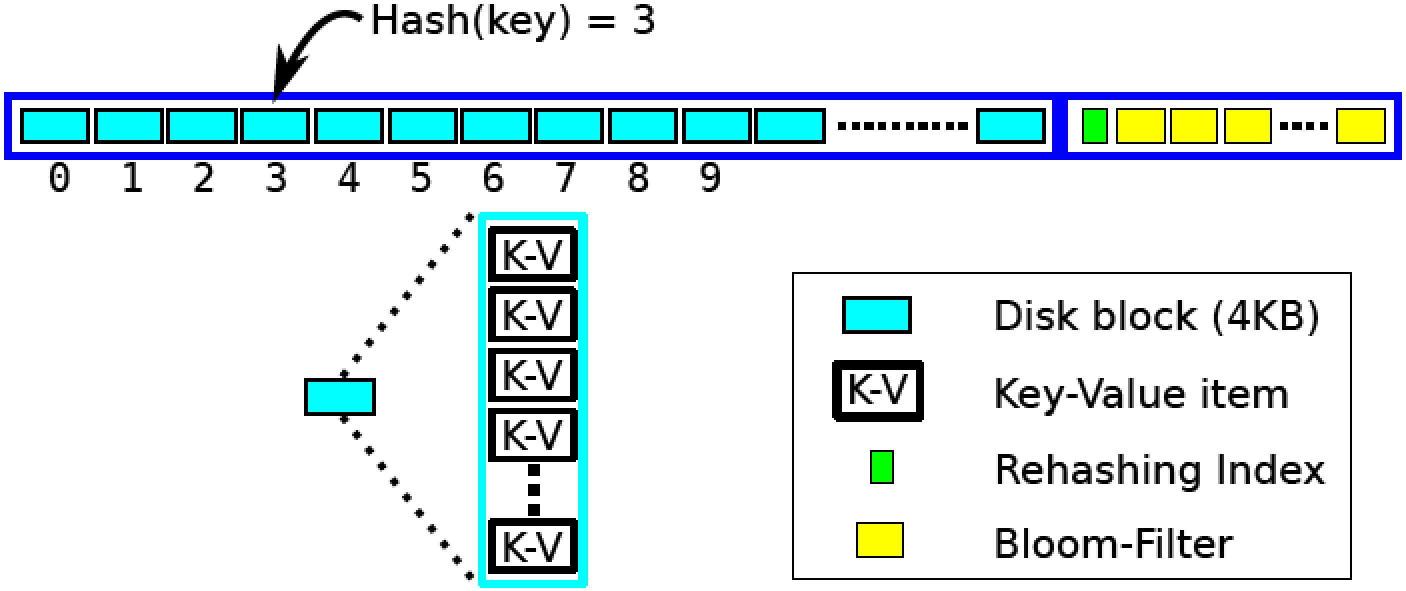
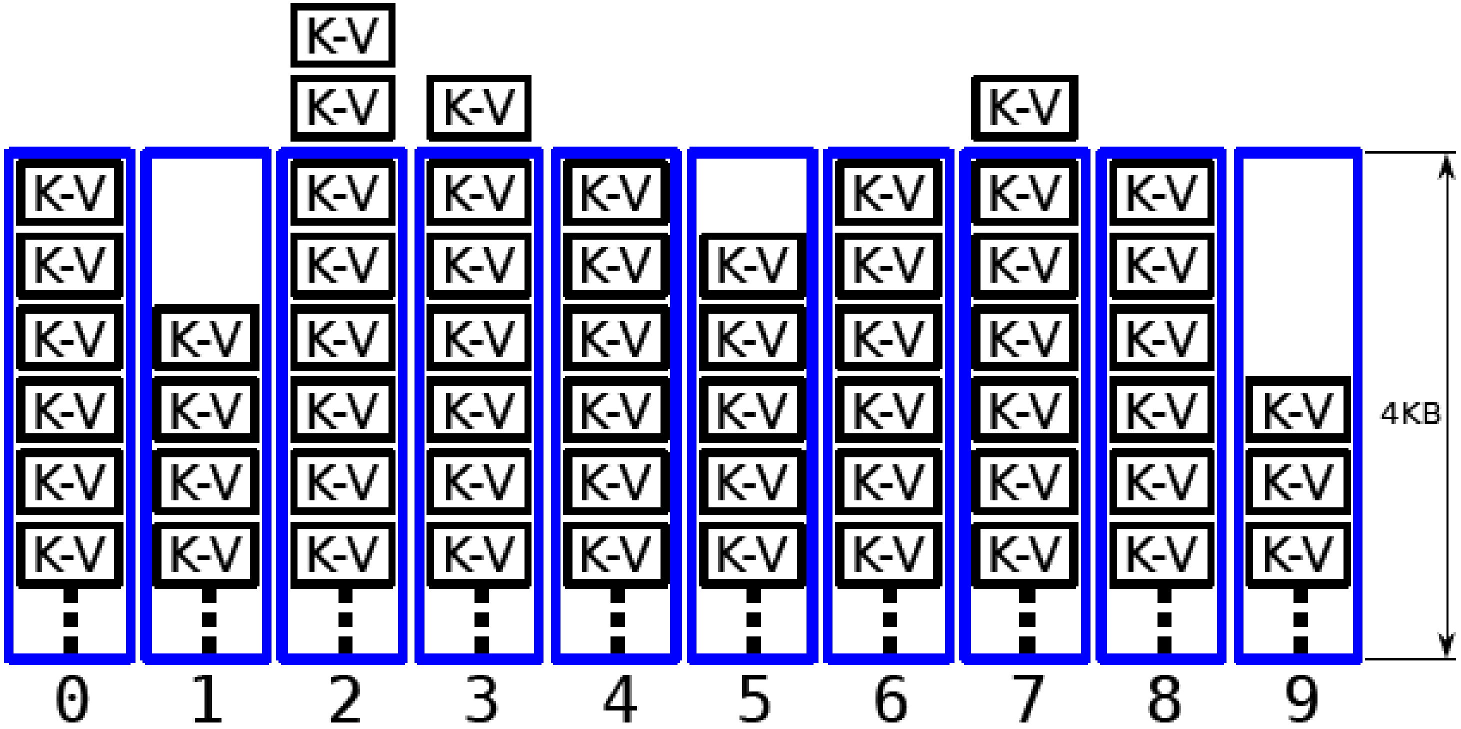


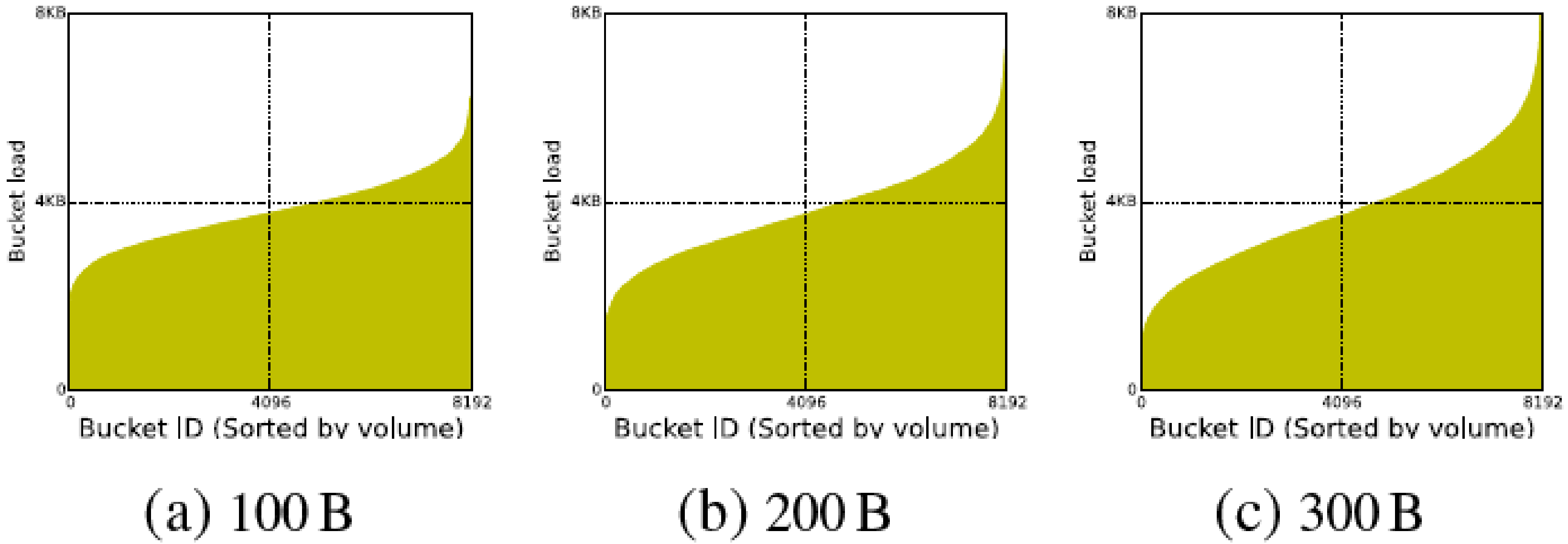
Figure 4: The structure of an HTable.

But the load on the buckets in an Htable may not be balanced.



(a) KV items are assigned to the buckets according to the hash function, causing unbalanced load distribution.

Load on the buckets in an Htable not be balanced.



(a) 100 B

(b) 200 B

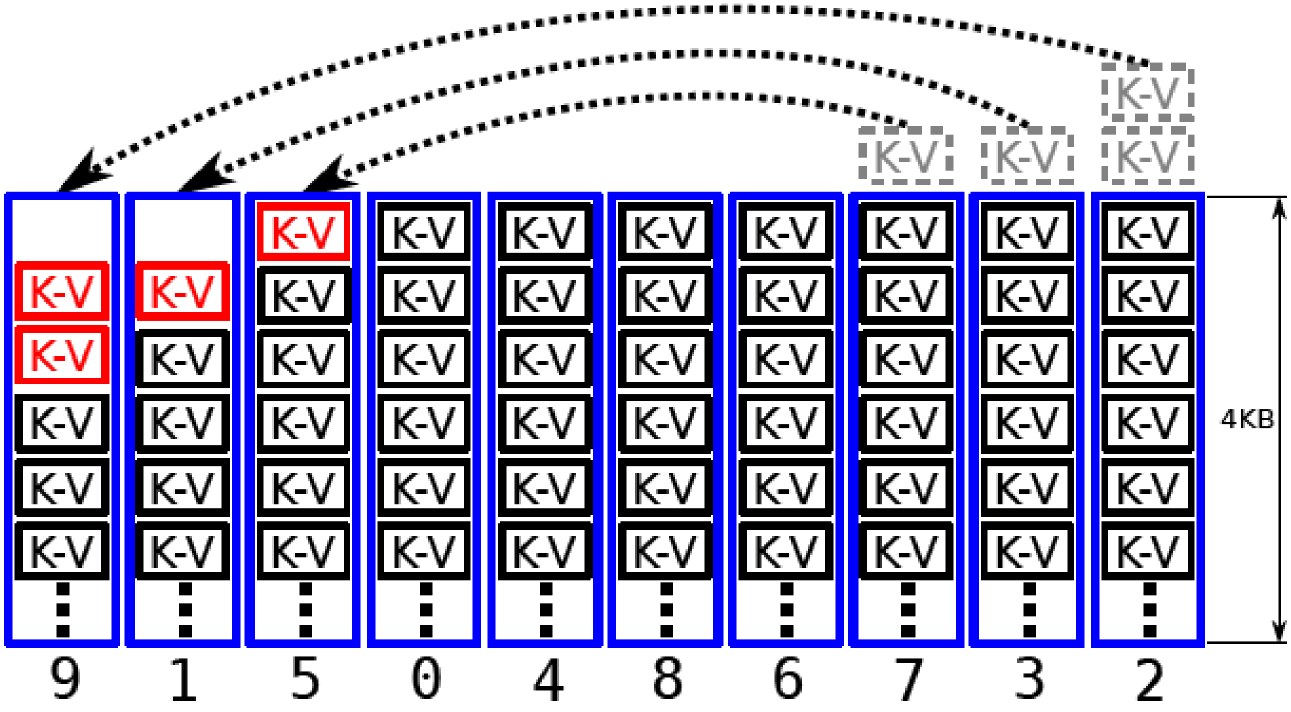
(c) 300 B

Figure 5: Distribution of bucket load across buckets of an HTable with a uniform distribution of KV-item size and an average size of 100 B (a), 200 B (b), and 300 B (c). The keys follow the Zipfian distribution. For each plot, the buckets are sorted according to their loads. The load is measured in terms of aggregate size of KV items in a bucket.

Balance the load using item migration

Migration Metadata:

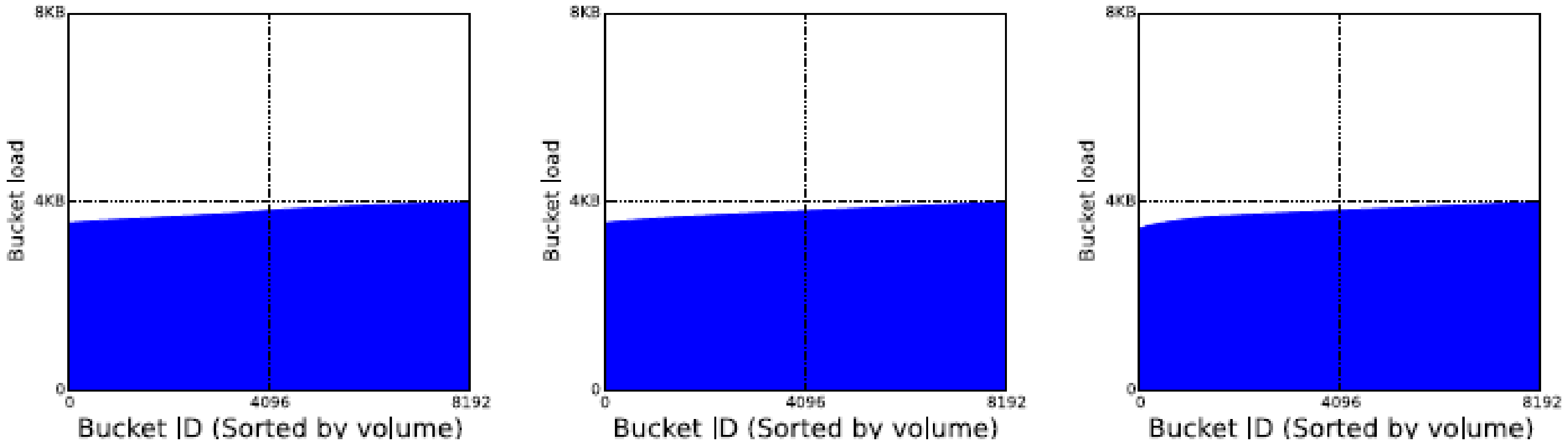
Src-ID	Dest-ID	HashMark
2	9	0xA953
3	1	0xD0C9
7	5	0xEE3F



(b) Buckets are sorted according to their loads and balanced by using a greedy algorithm.

- ❑ How to determine if an item has been migrated?
- ❑ Does BF still work after migration?

The load is balanced!



(a) 100 B

(b) 200 B

(c) 300 B

Figure 7: Bucket load distribution after retaining.

A Summary of the Use of 160b Hashkey

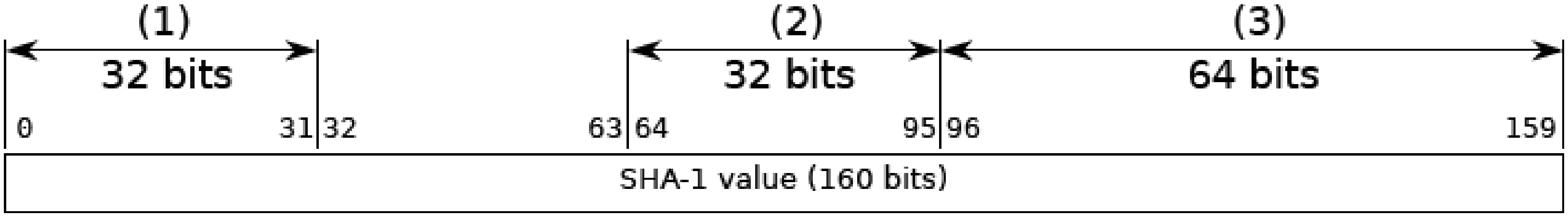


Figure 8: Use of a 160-bit SHA1 key. (1) The prefix is used for trie encoding. (2) The infix is used for sorting KV items in a bucket. (3) The suffix is used for locating the KV items in an HTable.

BF is clustered for at most one Access at each level

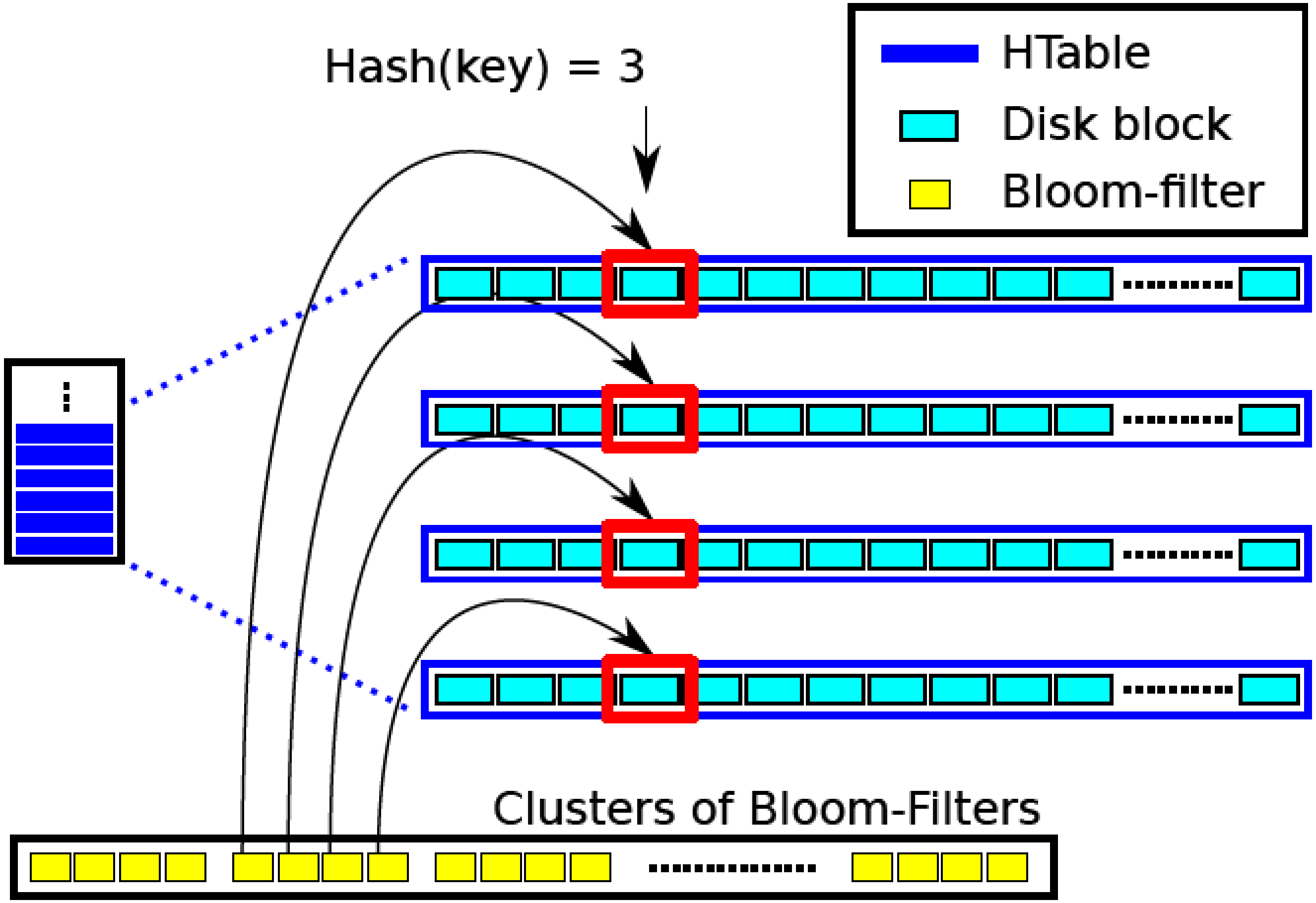


Figure 9: Clustering Bloom filters

Prototyped LSM-trie

- 32MB HTables and an amplification factor (AF) of 8.
- The store has five levels. In the first four levels, LSM-trie uses both linear and exponential growth pattern.
- All the Bloom filters for the first 32 sub-levels are of 4:5 GB, assuming a 64B average item size and 16 bit Bloom filter per key. Adding metadata about item migration within individual HTables (up to 0:5 GB), LSM-trie needs up to only 5GB memory to hold all necessary metadata
- At the fifth level, which is the last level, LSM-trie uses only linear growth pattern. As one sub-level at this level has a capacity of 128 G, it needs 8 such sub-levels for the store to reach 1 TB, and 80 such sub-levels to reach 10 TB.
- LSM-trie uses 16-bit-per-item Bloom filters, the false positive rate is only about 5% even for a 112-sub-level 10 TB KV store.

Write Throughput

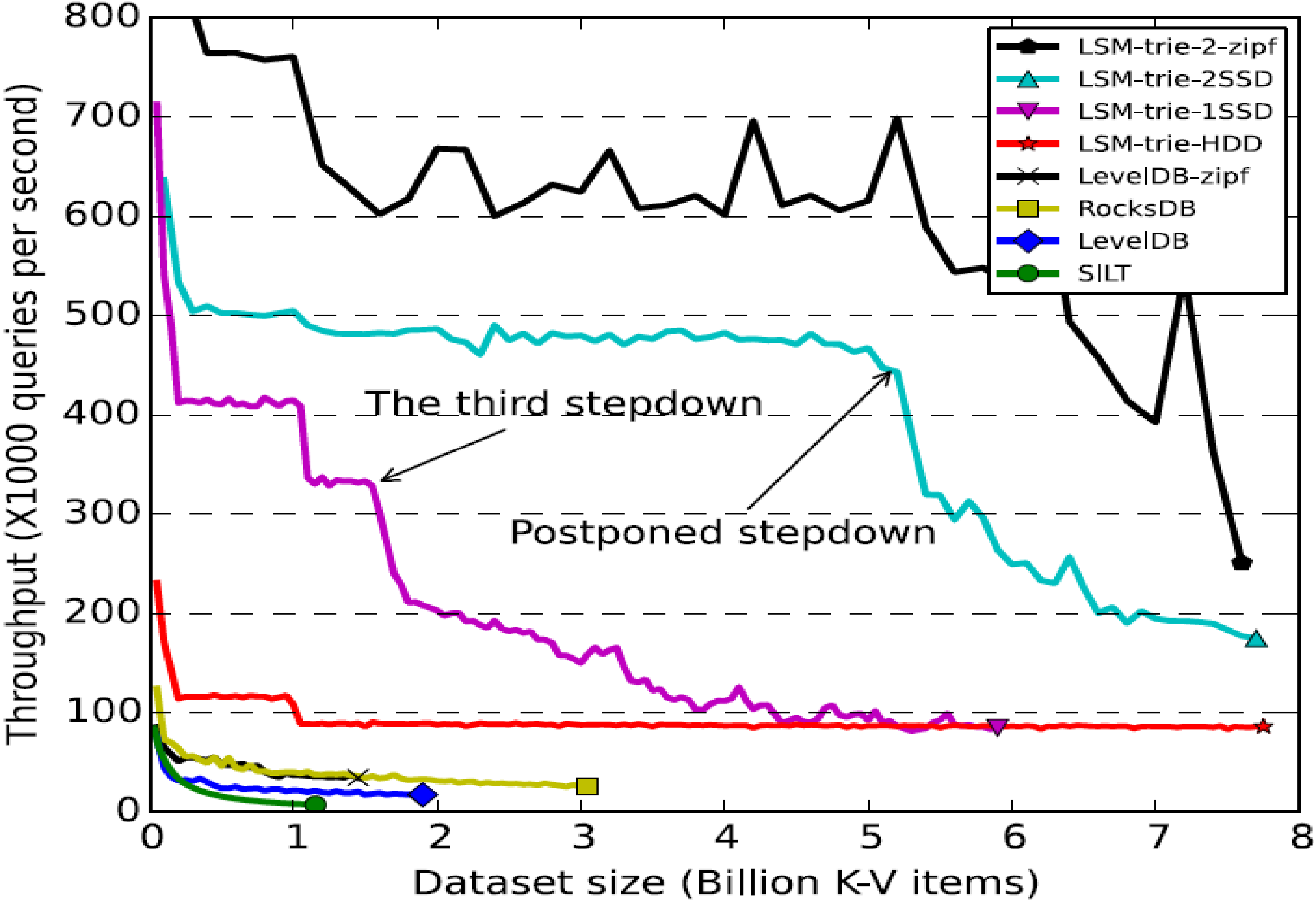


Figure 10: Write throughput of different schemes.

Write Amplification

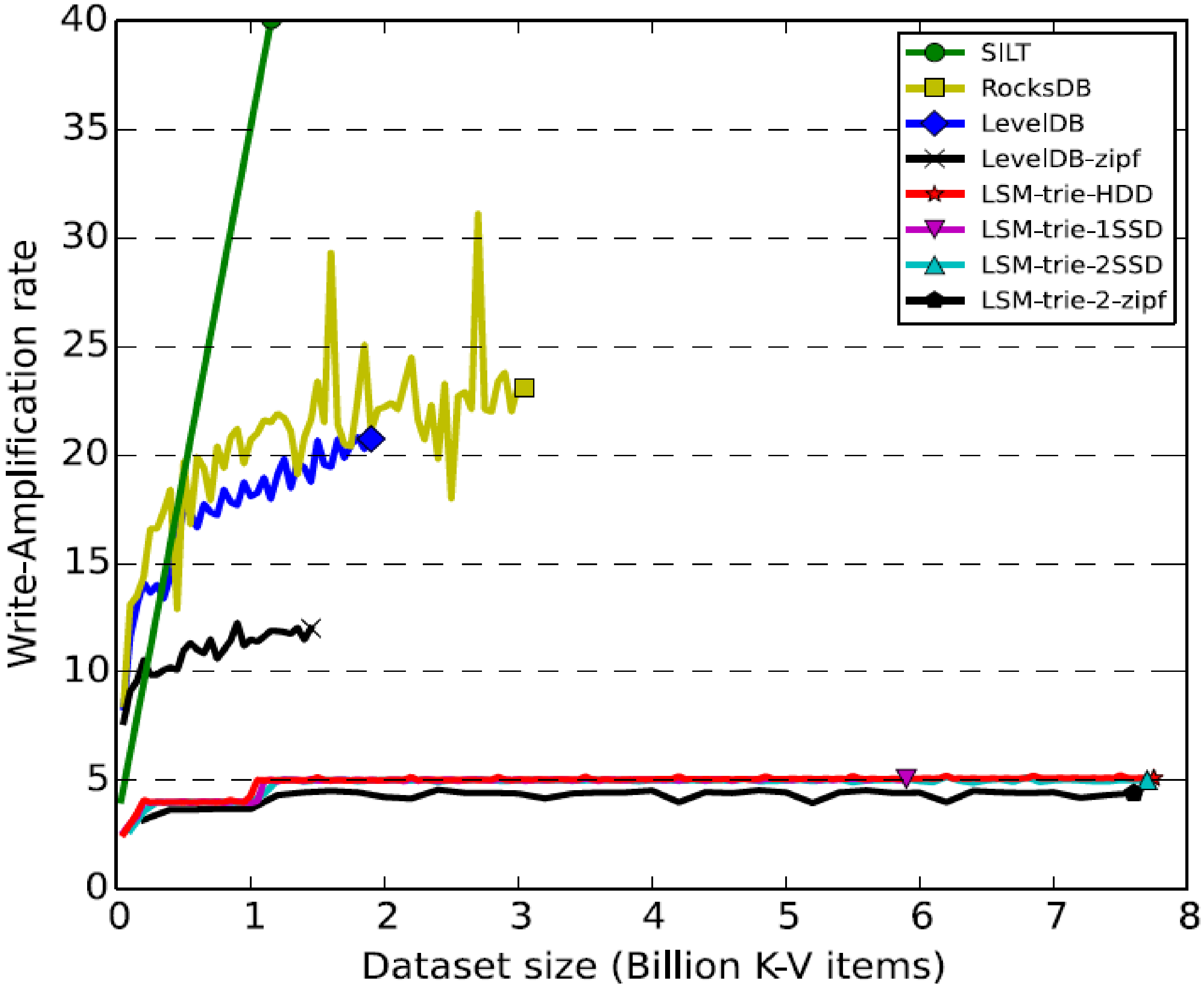


Figure 11: Write amplifications of different schemes.

Read Throughput

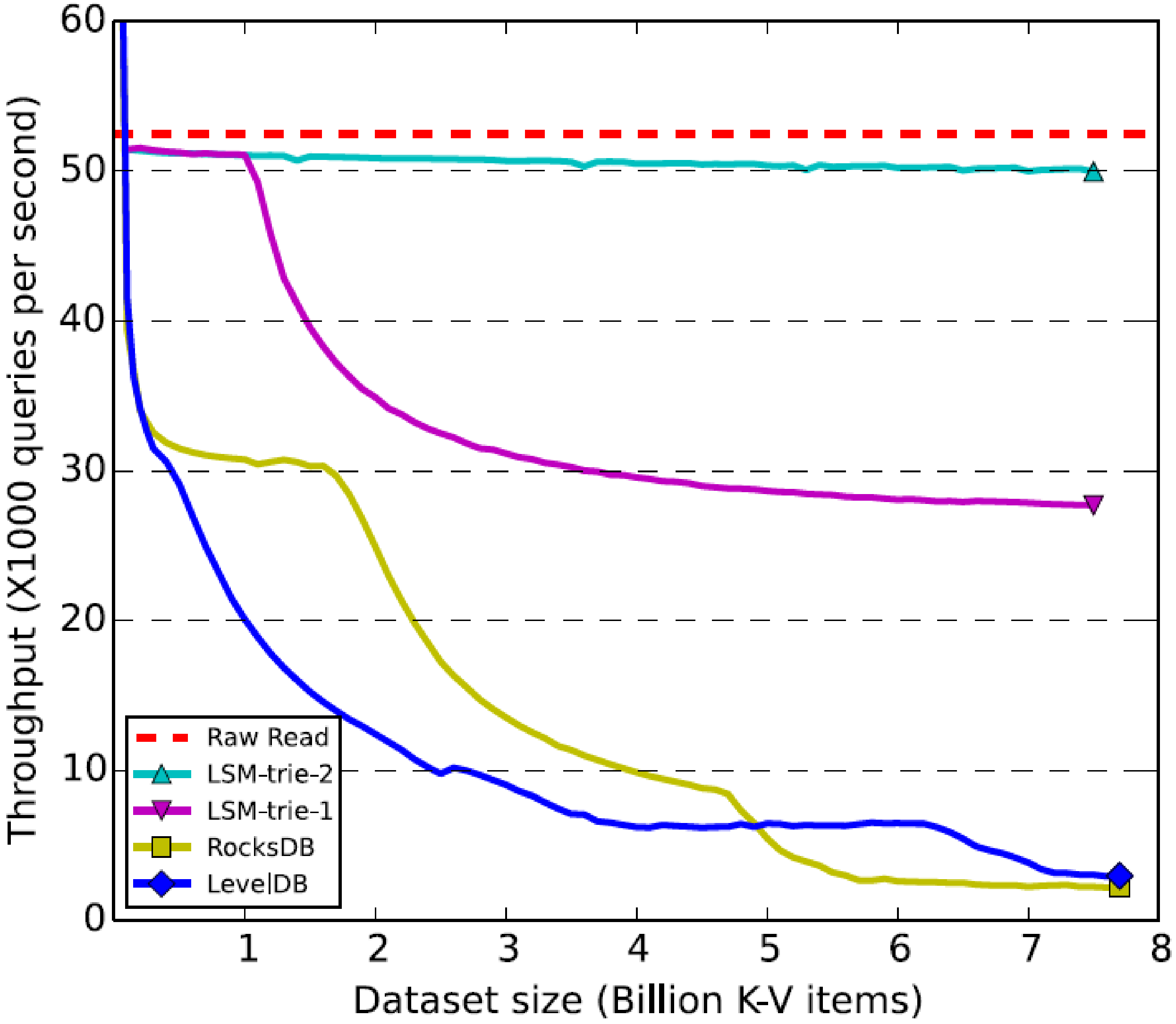


Figure 12: Read throughput of different schemes.

Summary

LSM-trie is designed to manage a large set of small data.

It reduces the write-amplification by an order of magnitude.

It delivers high throughput even with out-of-core metadata.

The LSM-trie source code can be downloaded at:

https://github.com/wuxb45/lsm-trie-release

Atlas: Baidu’s Key-value Storage System for Cloud Data

Cloud Storage Service

- Cloud storage services become increasingly popular.
  - Baidu Cloud has over 200 million users and 200PB user data.

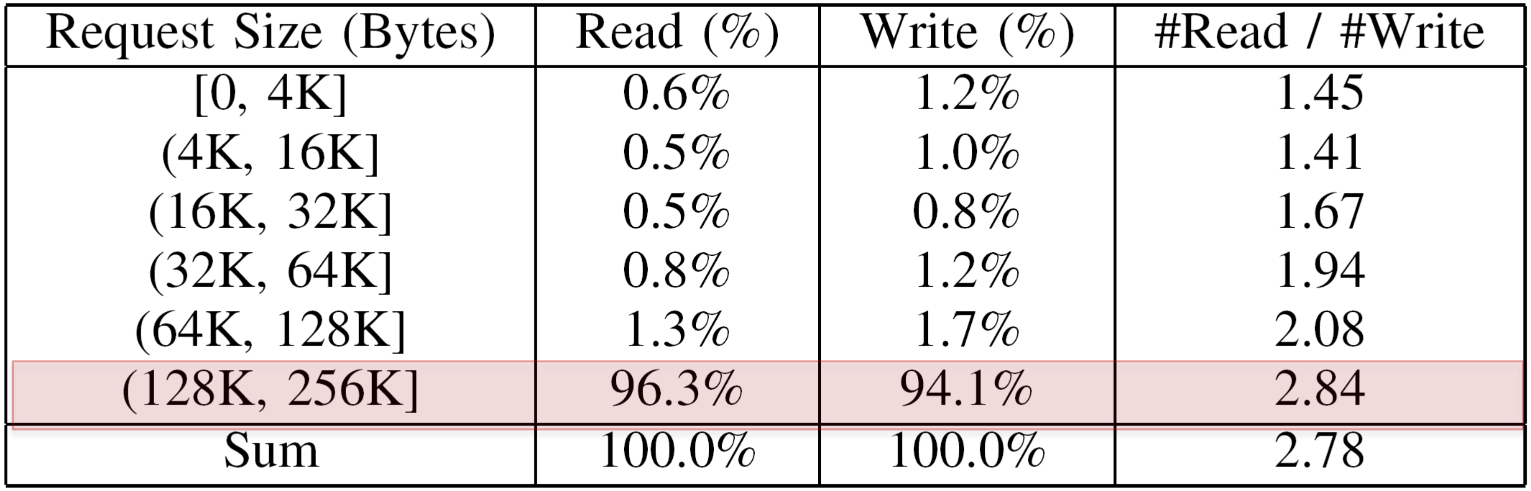
- To be attractive and competitive, they often offer large free space and price the service modestly.
  - Baidu offers 2TB free space for each user.

- The challenge is how to economically provision resources and also achieve service quality.
  - A large number of servers, each with local large storage space.
  - The data must be reliably stored with a high availability.
  - Requests for any data in the system should be served reasonably fast.

Challenges on Baidu’s System

- The workload
  - Request size is capped at 256KB for system efficiency.
  - Majority of the requests are for data between 128KB and 256KB.

Distribution of requests on a typical day in 2014.



Request Size (Bytes)	Read (%)	Write (%)	#Read / #Write
[0, 4K]	0.6%	1.2%	1.45
(4K, 16K]	0.5%	1.0%	1.41
(16K, 32K]	0.5%	0.8%	1.67
(32K, 64K]	0.8%	1.2%	1.94
(64K, 128K]	1.3%	1.7%	2.08
(128K, 256K]	96.3%	94.1%	2.84
Sum	100.0%	100.0%	2.78

- The Challenges
  - Can the X86 processors be efficiently used?
  - Can we use a file system to store data at each server?
  - Can we use an LSM-tree-based key-value store to store the data?

Challenge on Processor Efficiency

- ❑ The X86 processors (two 4-core 2.4GHz E5620) were consistently under-utilized
  - ➢ Less than 20% utilization rate with nine hard disks installed on a server.
  - ➢ Adding more disks is not an ultimate solution.

- ❑ The ARM processor (one 4-core 1.6GHz Cortex A9) can provide similar I/O performance.
  - ➢ The ARM processor is more than 10X cheaper and more energy-efficient.

- ❑ Baidu's customized ARM-based server.
  - ➢ Each 2U chassis has six 4-core Cortex A9 processors.
  - ➢ Each processor comes with four 3TB SATA disks.
- ❑ However, each processor can support only 4GB memory.
  - ➢ On each chassis only 24GB memory available for accessing data as large as 72TB data.

Challenge on Using a File System

- ❑ Memory cannot hold all metadata.
  - ➢ Most files would be of 128-256KB.
  - ➢ Access on the storage has little locality.
  - ➢ More than one disk accesses are often required to access a file.

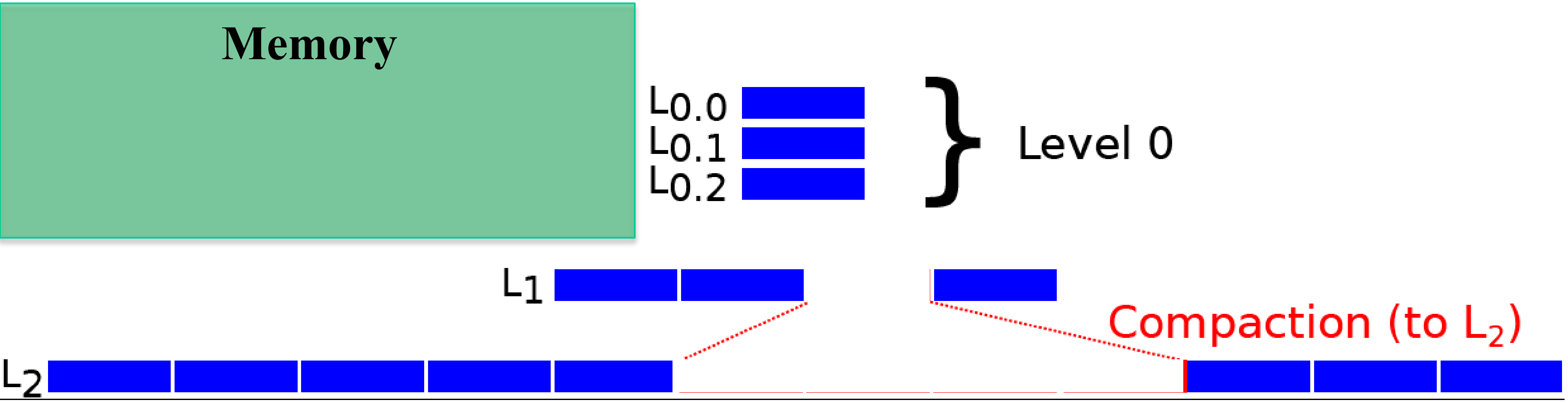
- ❑ The approach used in Facebook’s Haystack is not sufficient.
  - ➢ There are 3.3GB metadata for 16TB 128KB-data.
  - ➢ System software and buffer cache also compete for 4GB memory.

Challenge on Using LSM-tree Based Key-value Store

- ❑ LSM-tree-based KV store is designed for storing many small key-value items, represented by Google's LevelDB.

- ❑ The store is memory efficient.
  - ➢ The metadata is only about 320MB for 16TB 128KB-data.

- ❑ However, the store needs constant compaction operations to sort its data distributed across levels of the store.
  - ➢ For a store of 7 levels, the write amplification can be over 70.
  - ➢ Very limited I/O bandwidth is left for servicing frond-end user requests.

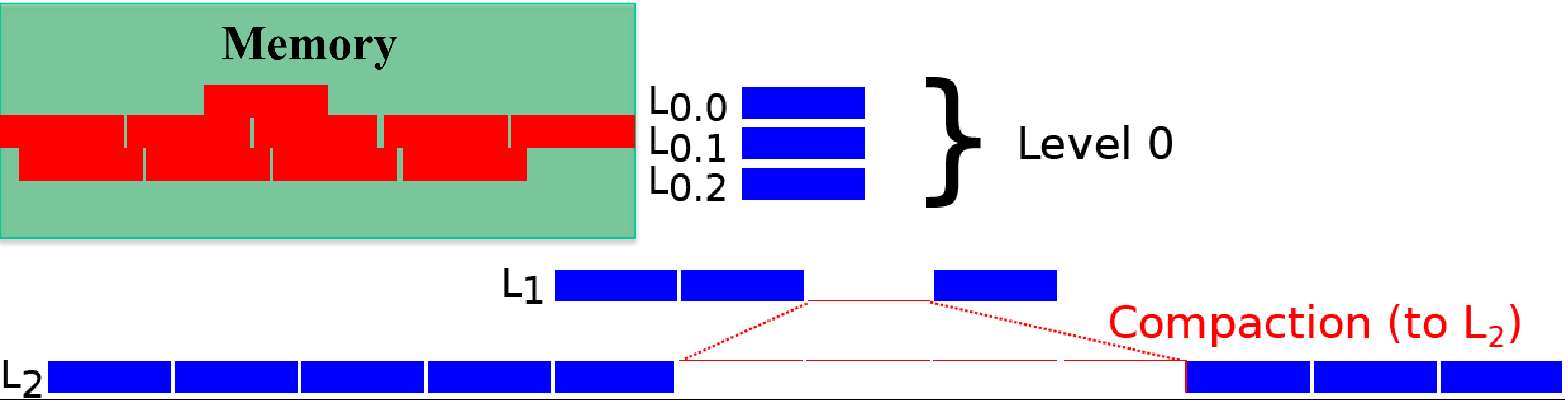


Challenge on Using LSM-tree Based Key-value Store

- ❑ LSM-tree-based KV store is designed for storing many small key-value items, represented by Google's LevelDB.

- ❑ The store is memory efficient.
  - ➢ The metadata is only about 320MB for 16TB 128KB-data.

- ❑ However, the store needs constant compaction operations to sort its data distributed across levels for such a small metadata.
  - ➢ For a store of 7 levels, the write amplification can be over 70.
  - ➢ Very limited I/O bandwidth is left for servicing frond-end user requests.

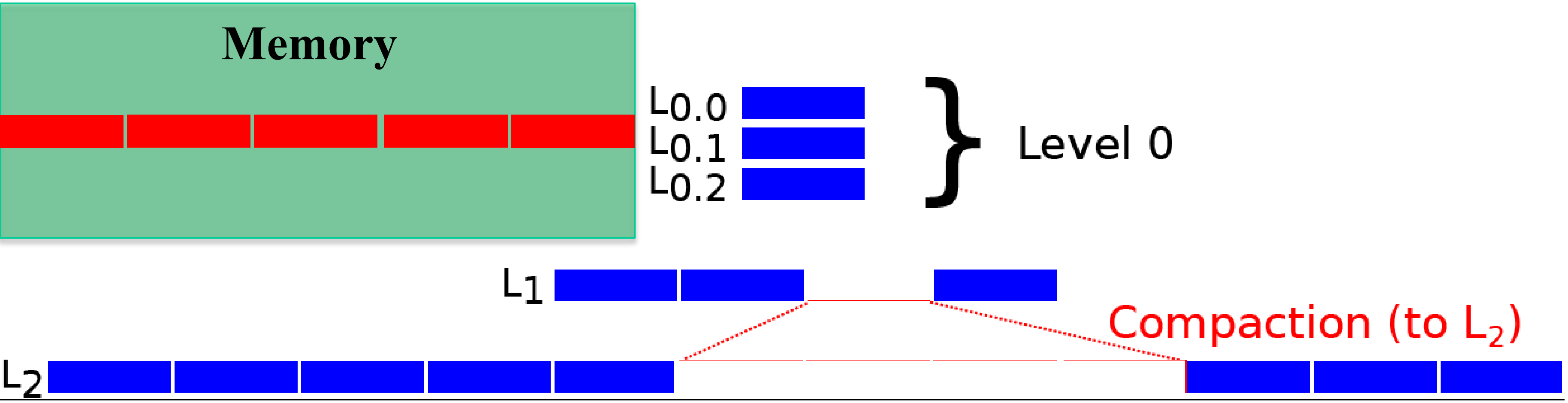


Challenge on Using LSM-tree Based Key-value Store

- ❑ LSM-tree-based KV store is designed for storing many small key-value items, represented by Google’s LevelDB.

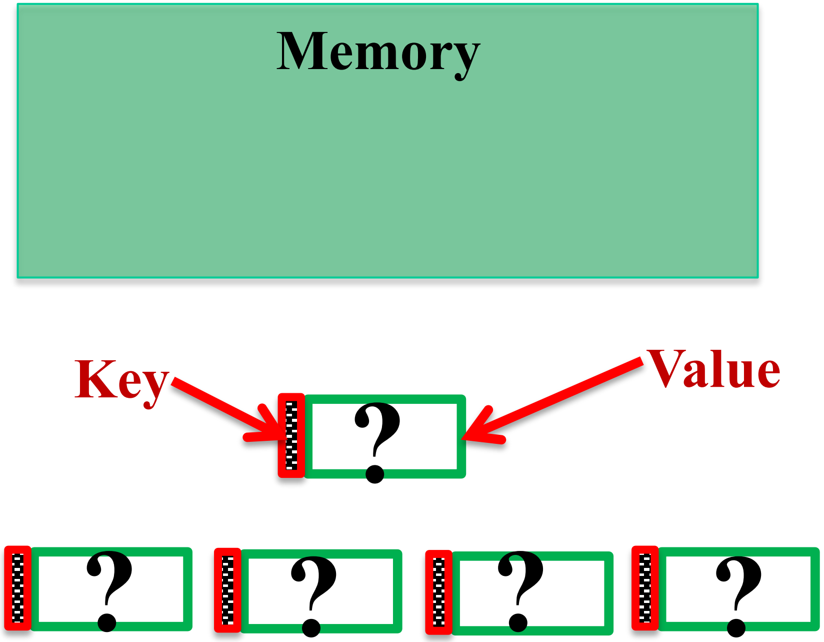
- ❑ The store is memory efficient.
  - ➢ The metadata is only about 320MB for 16TB 128KB-data.

- ❑ However, the store needs constant compaction operations to sort its data distributed across levels for such a small metadata.
  - ➢ For a store of 7 levels, the write amplification can be over 70.
  - ➢ Very limited I/O bandwidth is left for servicing frond-end user requests.



Reducing Compaction Cost

- ❑ In a KV item, value is usually much larger than the key.
- ❑ Values are not necessary to be involved in compactions.
- ❑ Move and place the values in a fixed-size container (block), and replace the values with pointers in KV items.

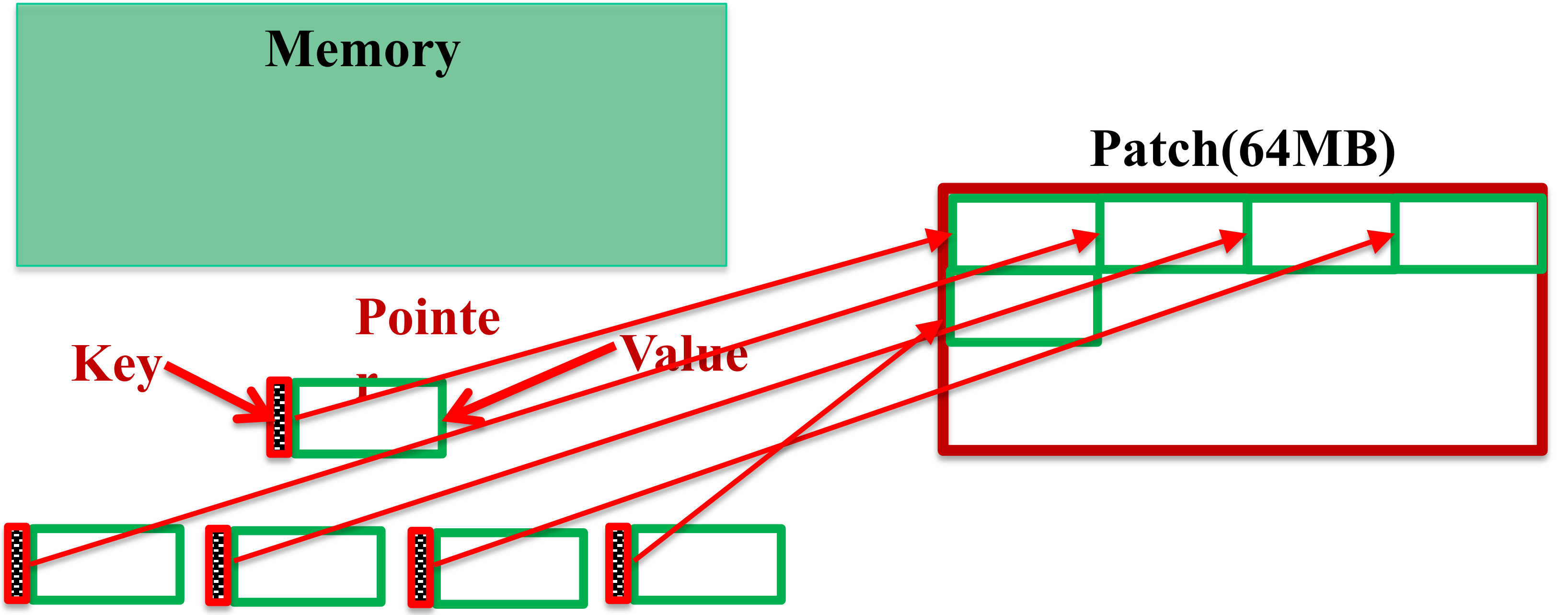


Reducing Compaction Cost

❑ In a KV item, value is usually much larger than the key.

❑ Values are not necessary to be involved in compactions.

❑ Move and place the values in a fixed-size container (block), and replace the values with pointers in KV items.

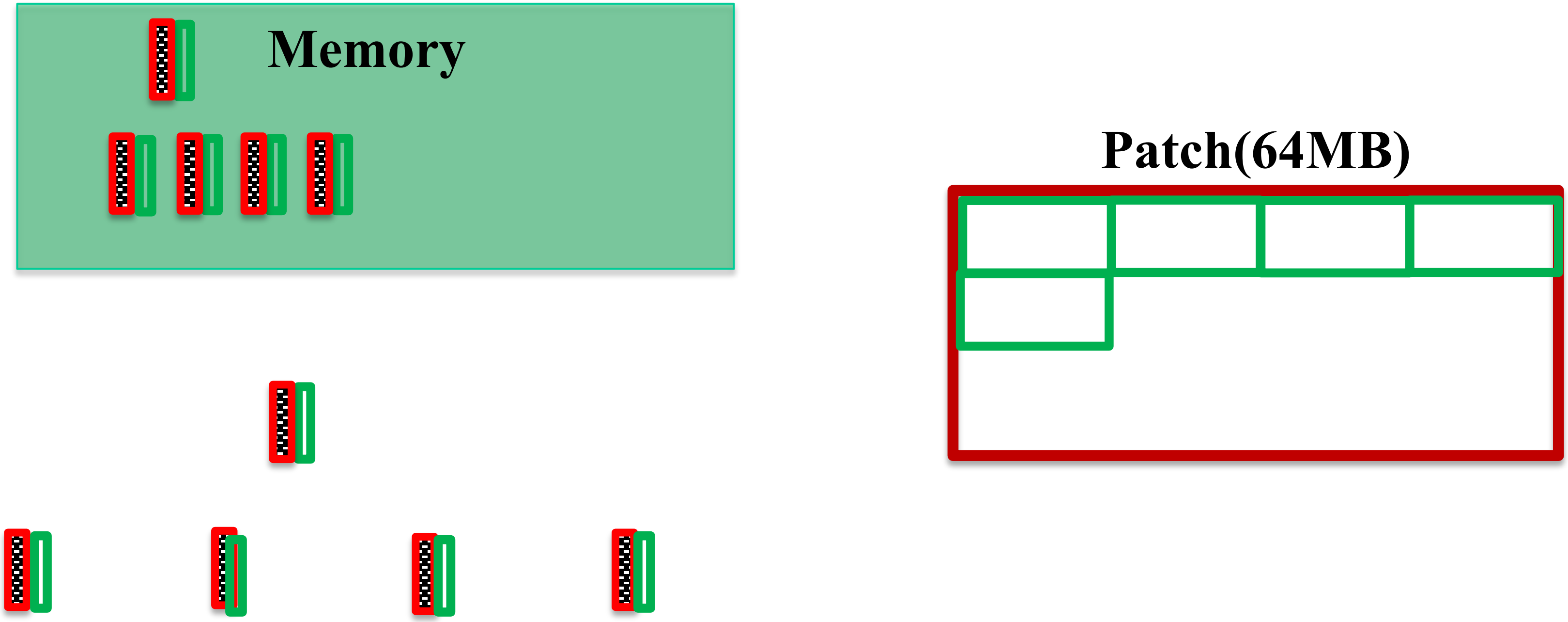


Reducing Compaction Cost

- In a KV item, value is usually much larger than the key.

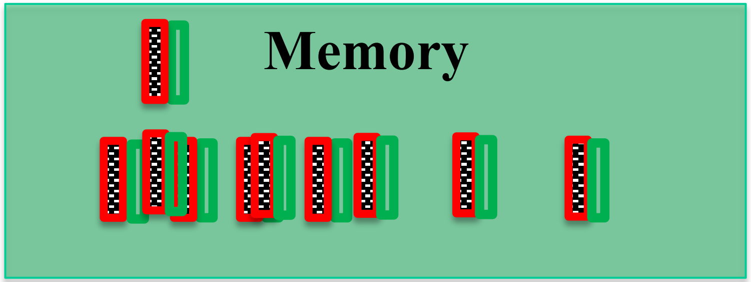
- Values are not necessary to be involved in compactions.

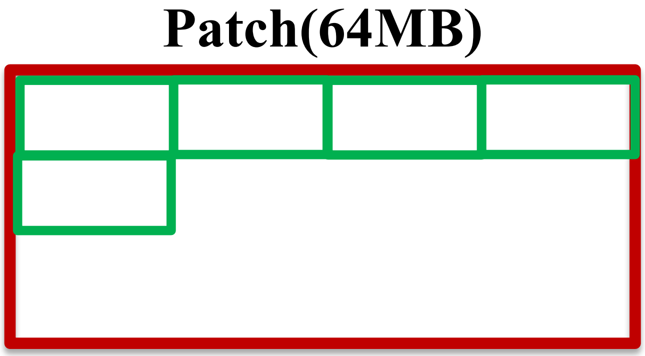
- Move and place the values in a fixed-size container (block), and replace the values with pointers in KV items.



Reducing Compaction Cost

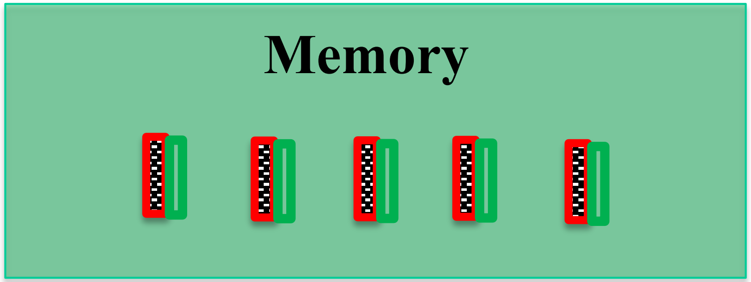
- ❑ In a KV item, value is usually much larger than the key.
- ❑ Values are not necessary to be involved in compactions.
- ❑ Move and place the values in a fixed-size container (block), and replace the values with pointers in KV items.

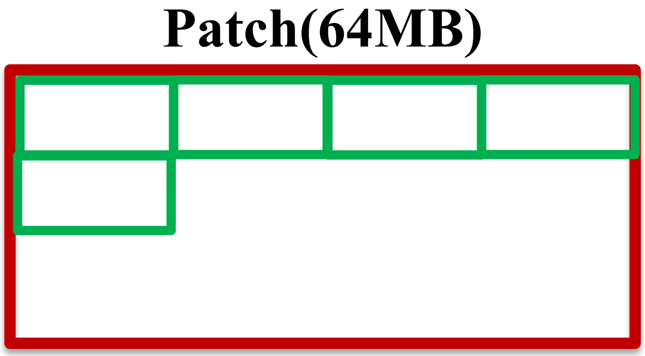




Reducing Compaction Cost

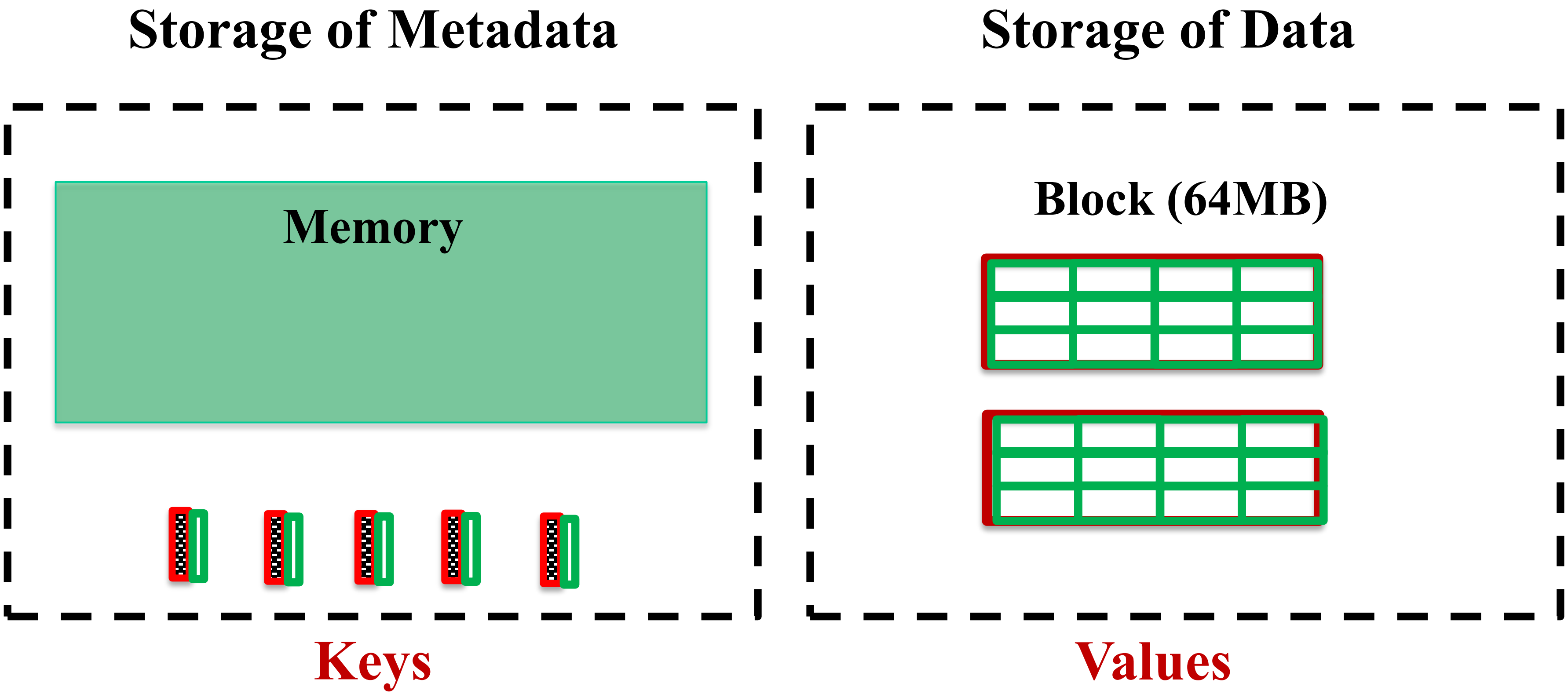
- ❑ In a KV item, value is usually much larger than the key.
- ❑ Values are not necessary to be involved in compactions.
- ❑ Move and place the values in a fixed-size container (block), and replace the values with pointers in KV items.



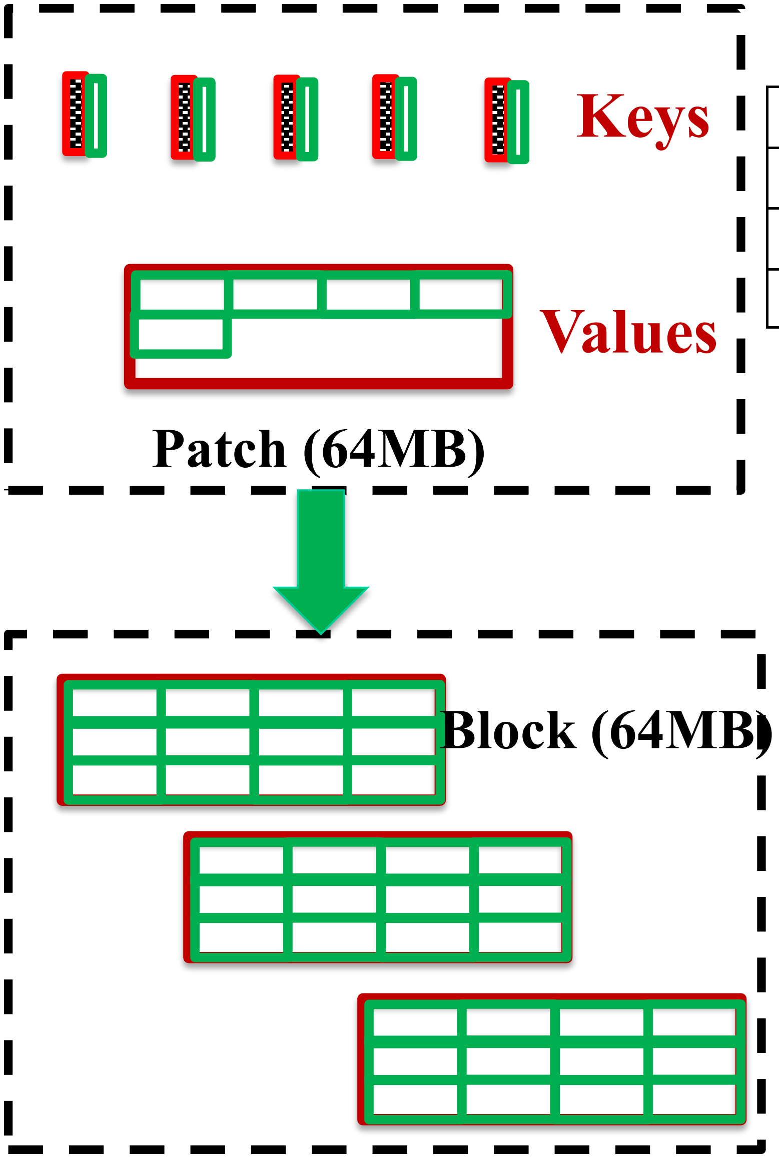


Features of Baidu's Cloud Storage System (Atlas)

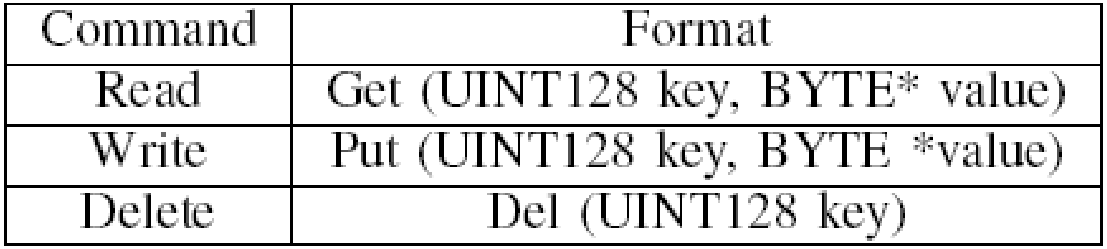
- ❑ A hardware and software co-design with customized low-power servers for high resource utilization
- ❑ Separate metadata (keys and offsets) and data (value blocks) management systems.
- ❑ Data are efficiently protected by erasure coding.



Big Picture of the Atlas System

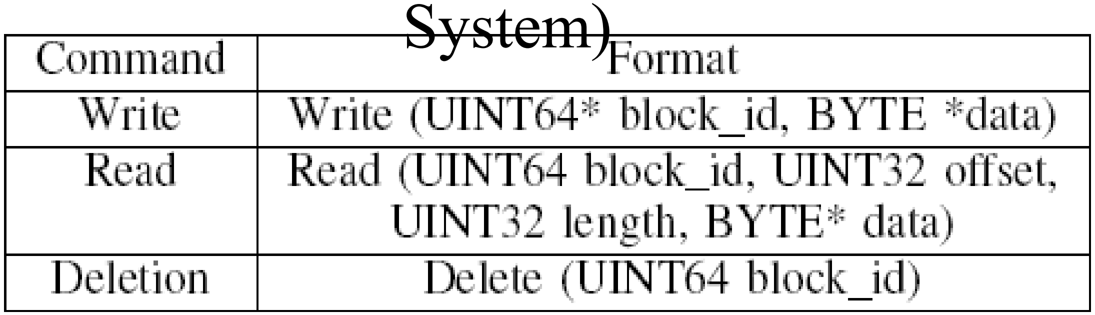


PIS (Patch and Index System)



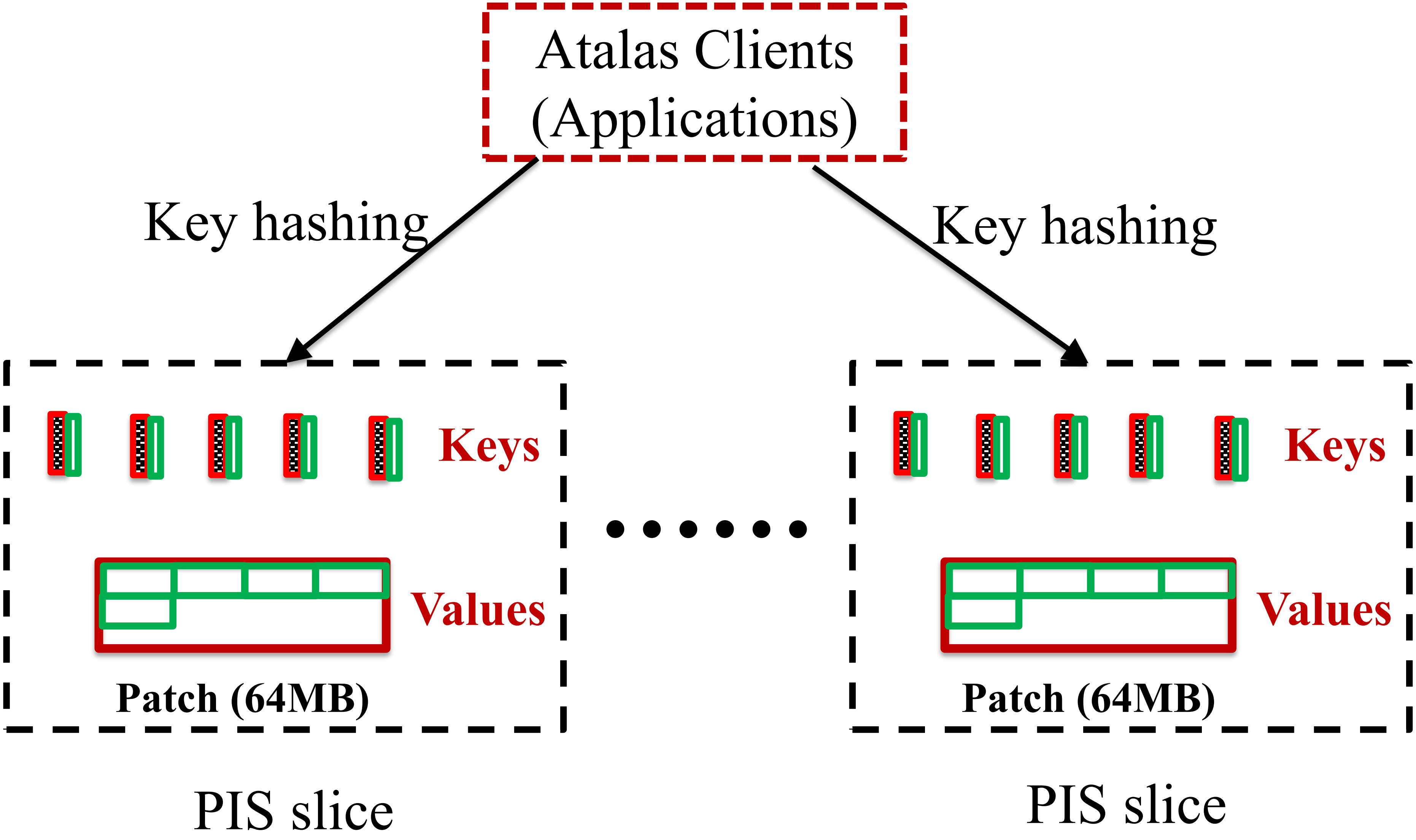
Command	Format
Read	Get (UINT128 key, BYTE* value)
Write	Put (UINT128 key, BYTE *value)
Delete	Del (UINT128 key)

RBS (RAID-like Block System)



Command	Format
Write	Write (UINT64* block_id, BYTE *data)
Read	Read (UINT64 block_id, UINT32 offset, UINT32 length, BYTE* data)
Deletion	Delete (UINT64 block_id)

Distribution of User Requests



Atalas Clients (Applications)

Key hashing

Key hashing

Keys

Values

Patch (64MB)

Keys

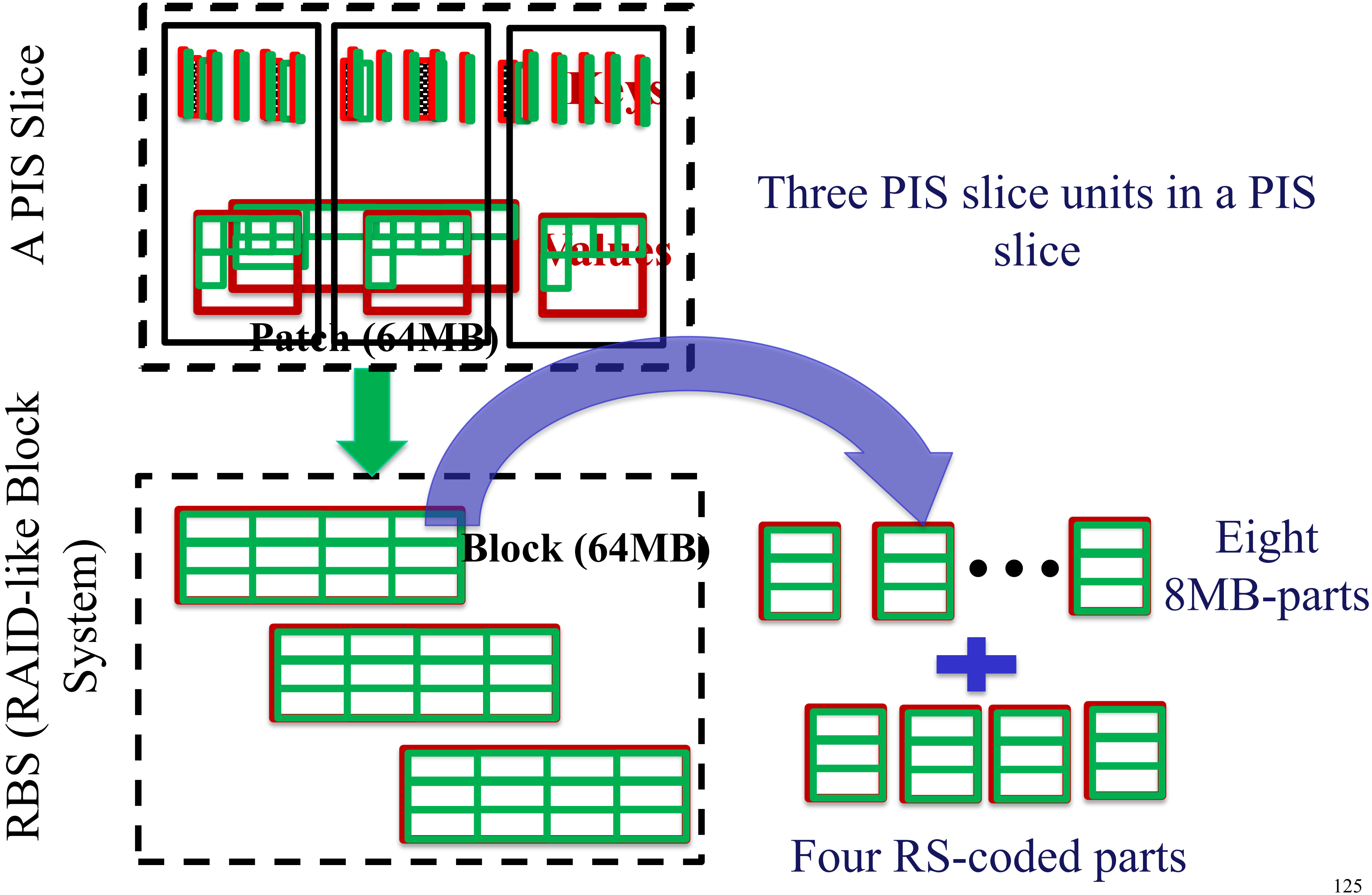
Values

Patch (64MB)

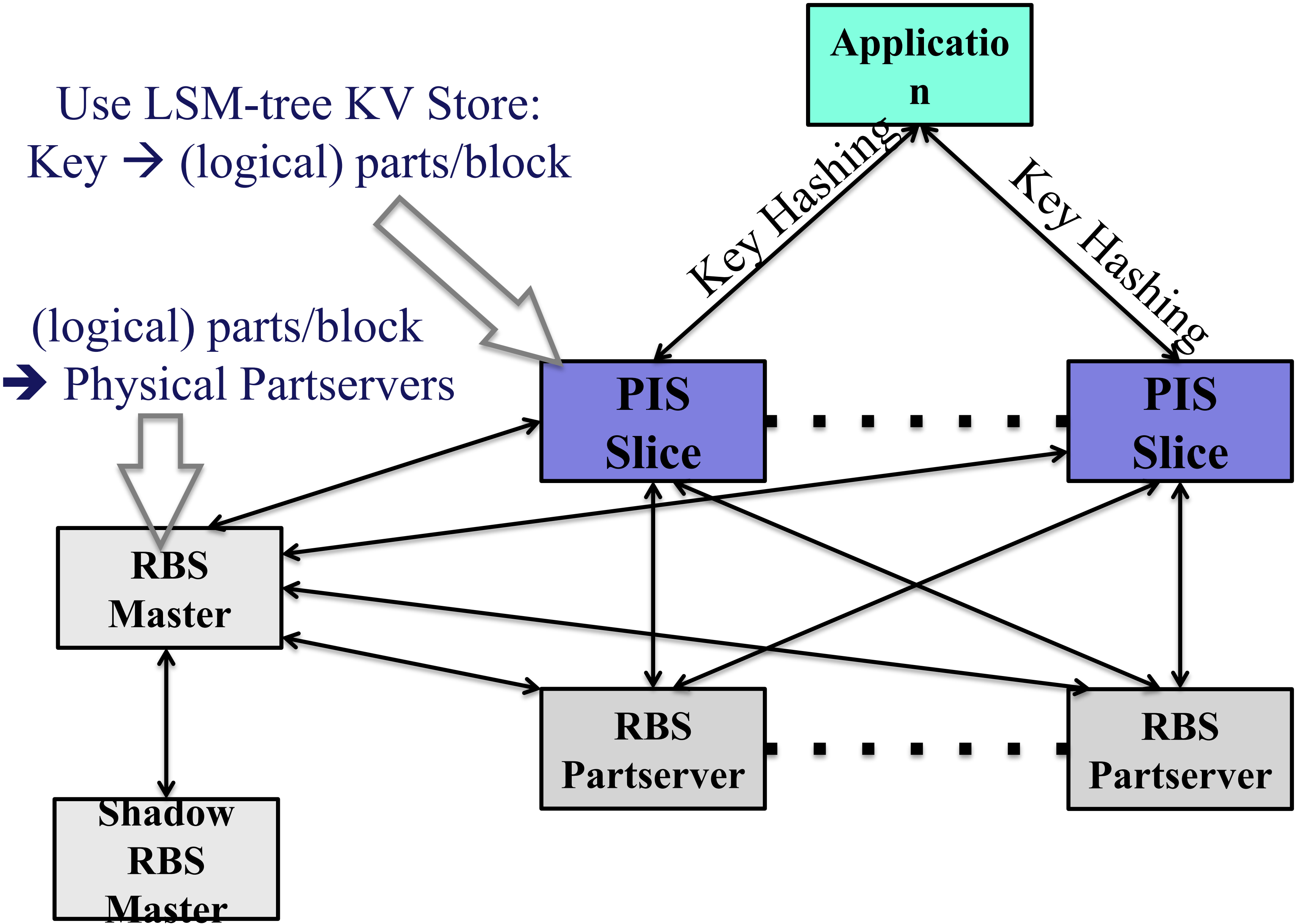
PIS slice

PIS slice

Redundancy for Protecting KV items



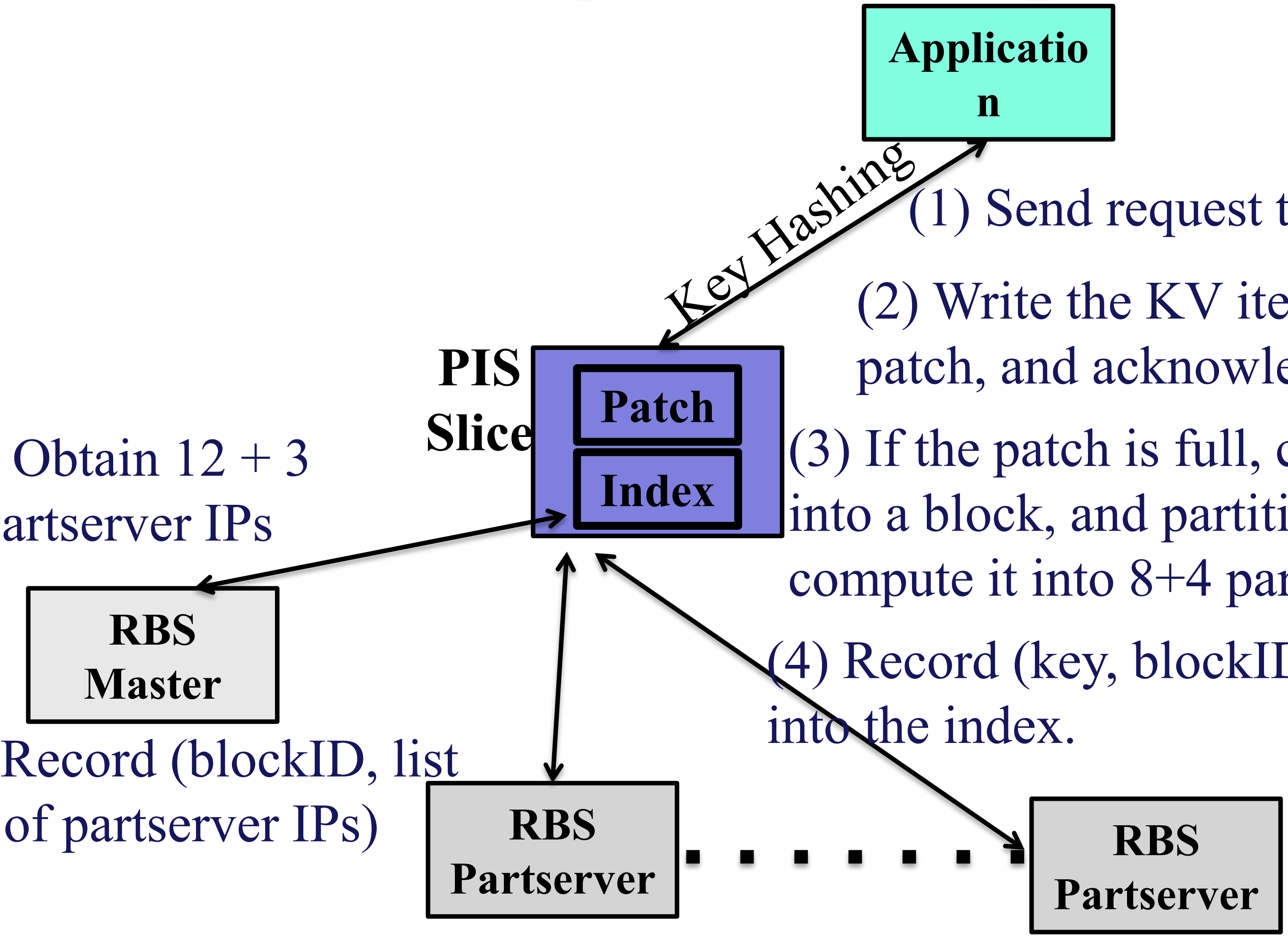
The Architecture of Atlas



Use LSM-tree KV Store:
Key → (logical) parts/block

(logical) parts/block
→ Physical Partservers

Serving a Write Request



(1) Send request to a PIS slice.

(2) Write the KV item in the patch, and acknowledge client;

(3) If the patch is full, convert it into a block, and partition and compute it into 8+4 parts.

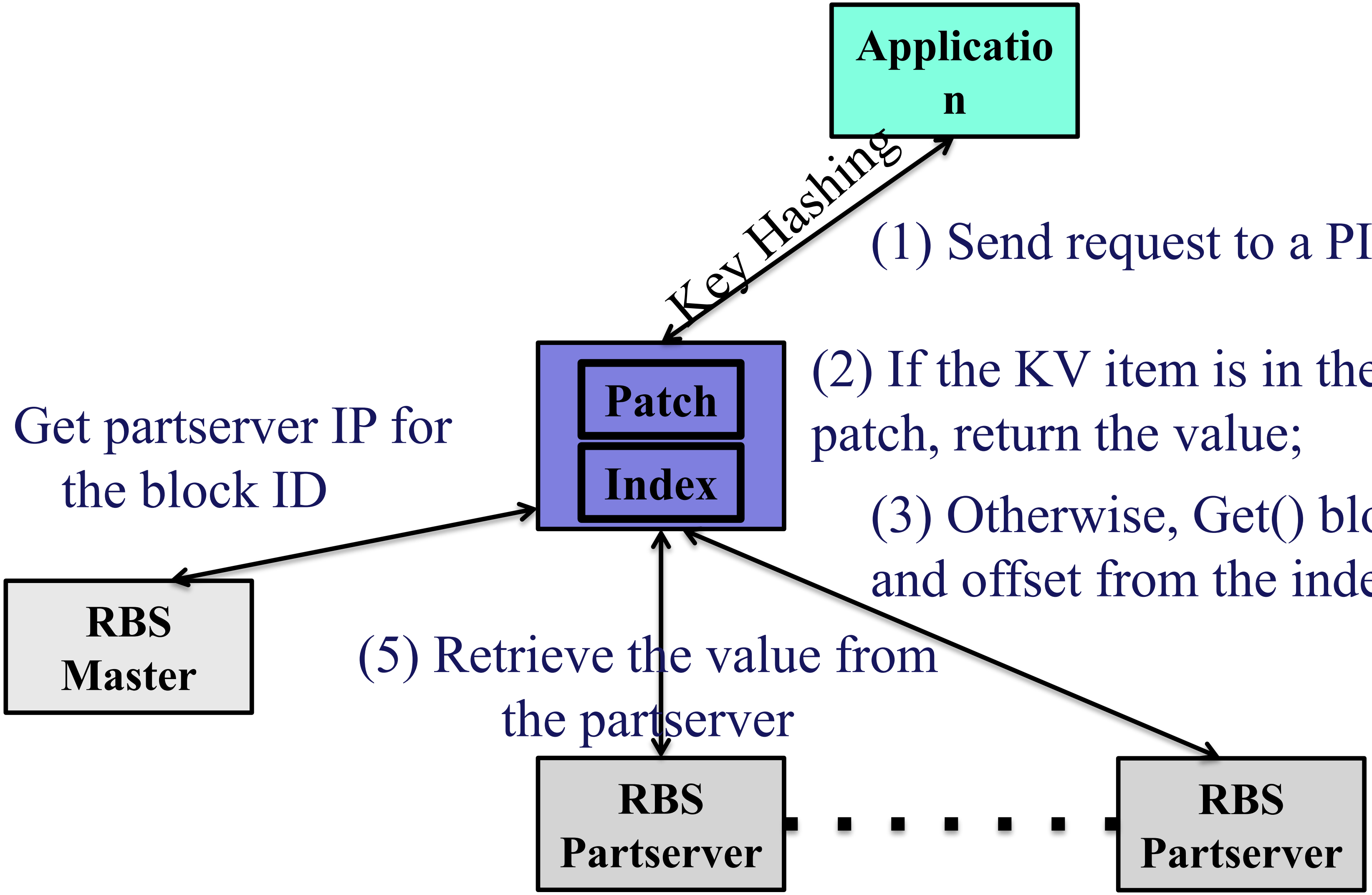
(4) Record (key, blockID, offset) into the index.

(4) Obtain 12 + 3 partserver IPs

6) Record (blockID, list of partserver IPs)

(5) Write the parts to the partsevers.

Serving a Read Request



(1) Send request to a PIS slice.

(2) If the KV item is in the patch, return the value;

(3) Otherwise, Get() block ID and offset from the index.

(4) Get partserver IP for the block ID

(5) Retrieve the value from the partserver

(6) Part recovery is initiated if it is a failure

Serving Delete/Overwrite Requests

❑ KV pairs stored in Atlas are immutable.

❑ Blocks in Atlas are also immutable.

❑ A new KV item is written into the system to service a delete/overwritten request.

❑ Space occupied by obsolete items are reclaimed in a garbage collection (GC) process.

❑ Periodically two questions are asked about a block in the RBS subsystem, and positive answers to both lead to a GC.

1) Is the block created earlier than a threshold (such as one week ago)?
2) Is the ratio of valid data in the block smaller than a threshold (such as 80%)?

Atlas's Advantages on Hardware Cost and Power

- ❑ Atlas saves about 70% of hardware cost per GB storage
  - Using ARM servers to replace x86 servers
  - Using erasure coding to replace 3-copy replication.
- ❑ Power consumption is reduced by about 53% per GB storage.
  - The ARM processors are more power efficient.
  - The ARM server racks are more space efficient, reducing energy cost for power supply and thermal dissipation.

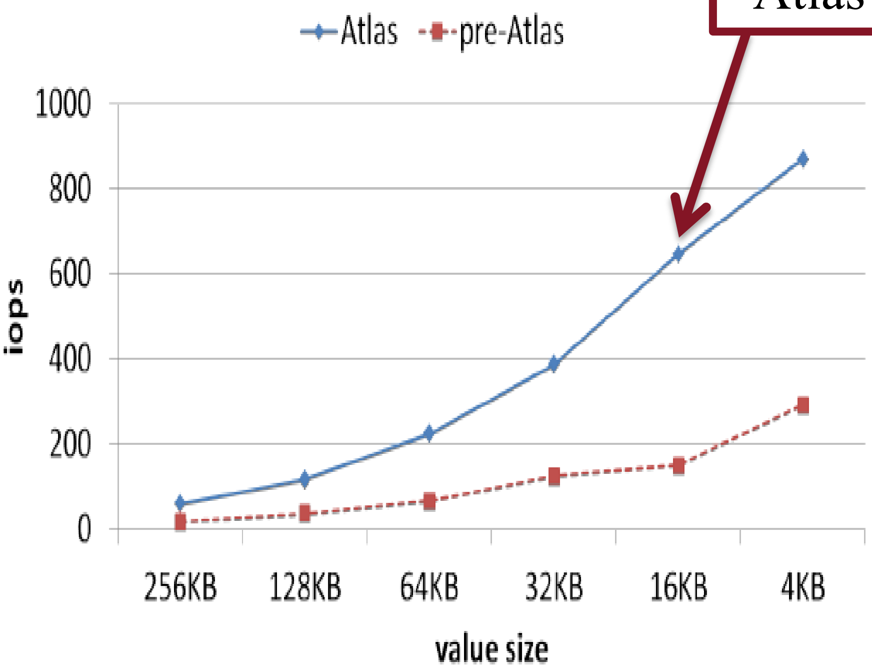
Comparison with the Prior System

❑Reference system (pre-Atlas)

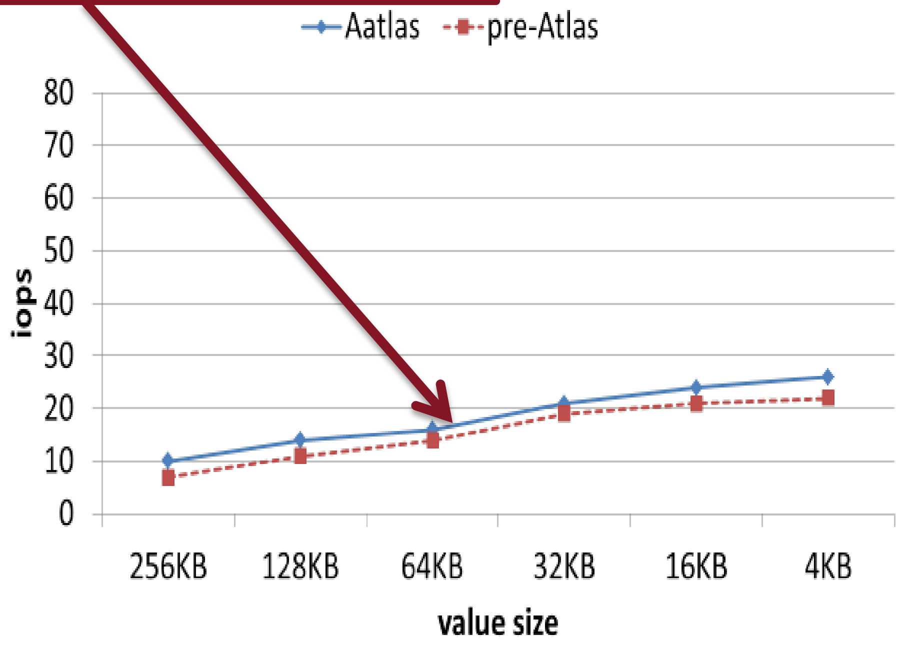
- Similar PIS subsystem.
- All data are managed solely by the LSM-tree-based KV store.

❑Run on a 12-server X86 cluster.

Atlas’s throughput at one node



All writes

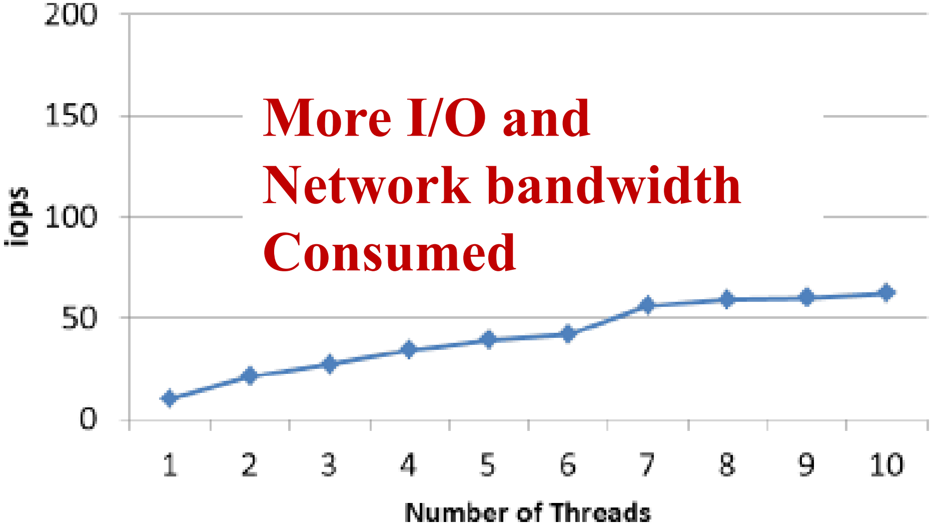


Read : Write = 3:1

Atlas on a Customized ARM cluster

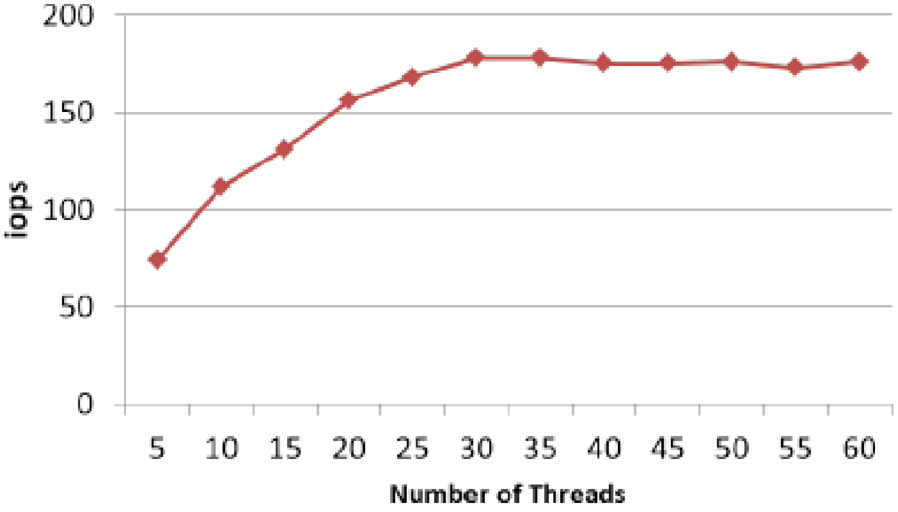
- A cluster of 12 ARM servers.
- Each hosts multiple PIS slices and RBS partservers.
- Each server has a 4-core Marvell processor, 4GB memory, four 3TB disks.
- 1Gbps full-duplex Ethernet adapter.
- Request size is 256KB.

Throughput at One Node with Diff. Request Types

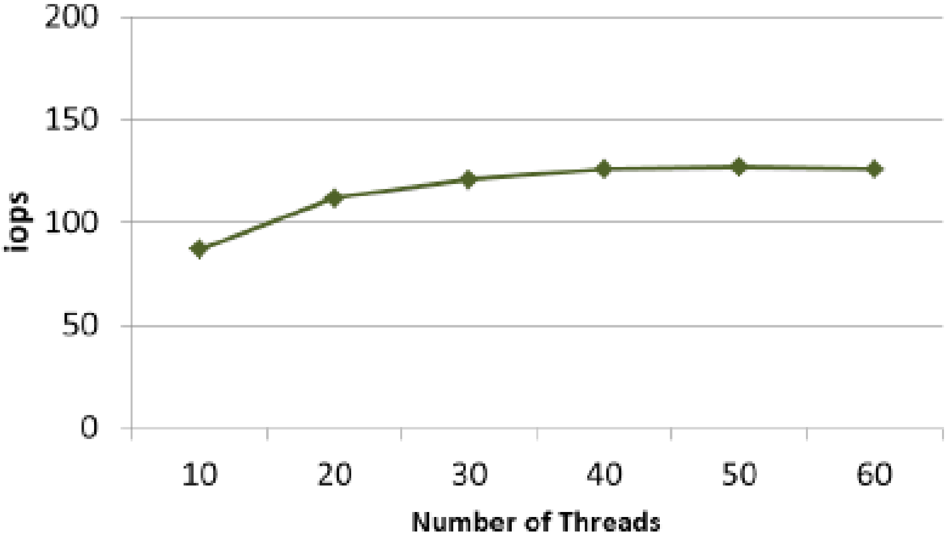


More I/O and Network bandwidth Consumed

All writes

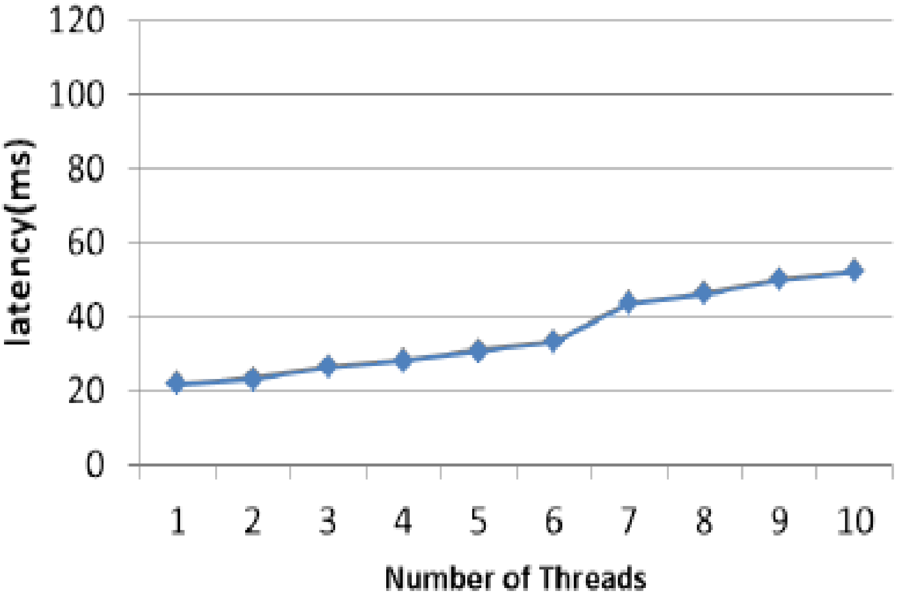


All Reads

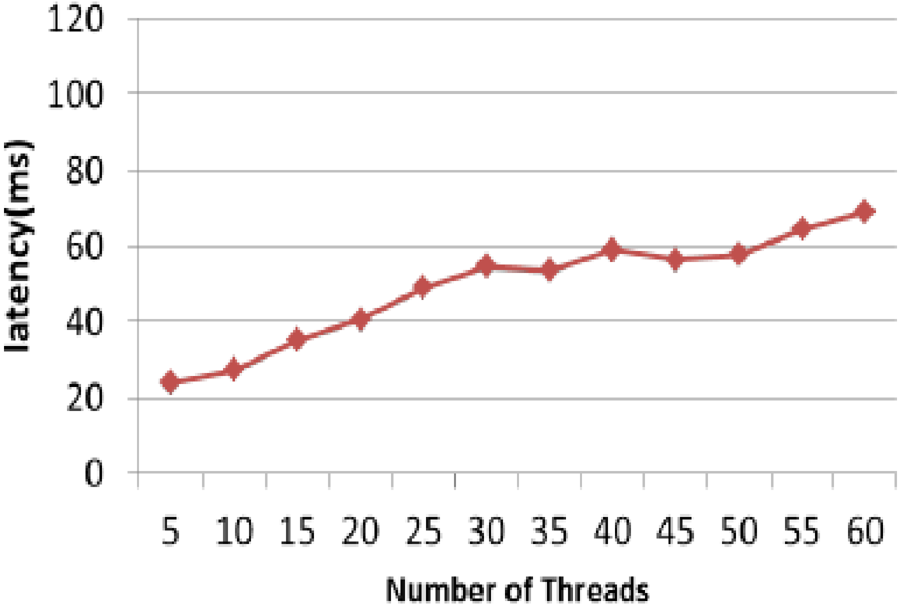


Read : Write = 3:1

Latencies with Diff. Request Types

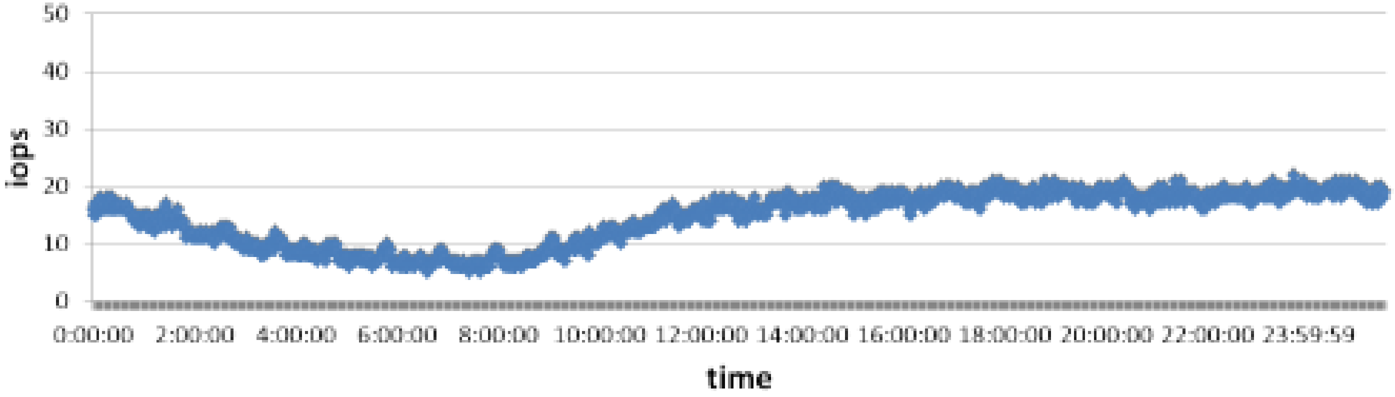


All writes

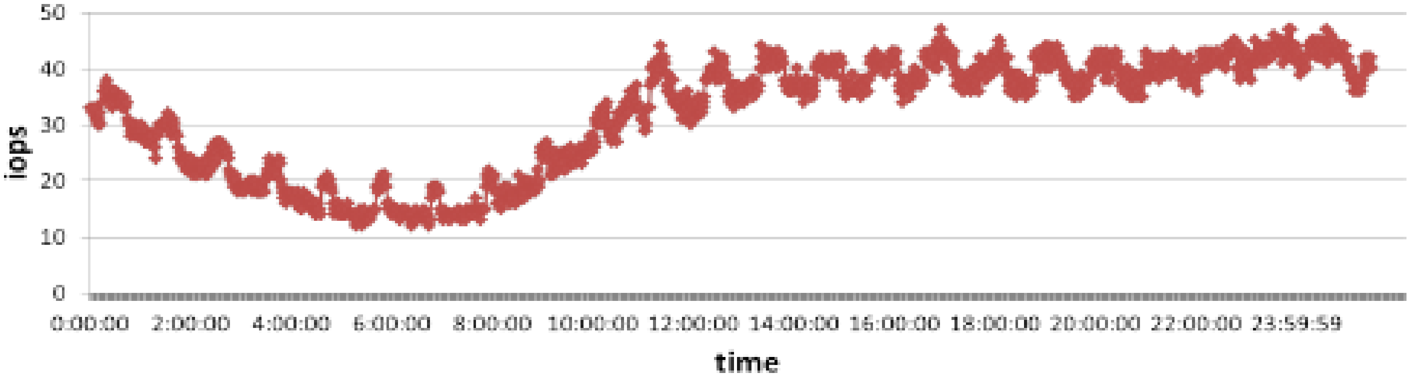


All Reads

Throughput at one Node of a Production System

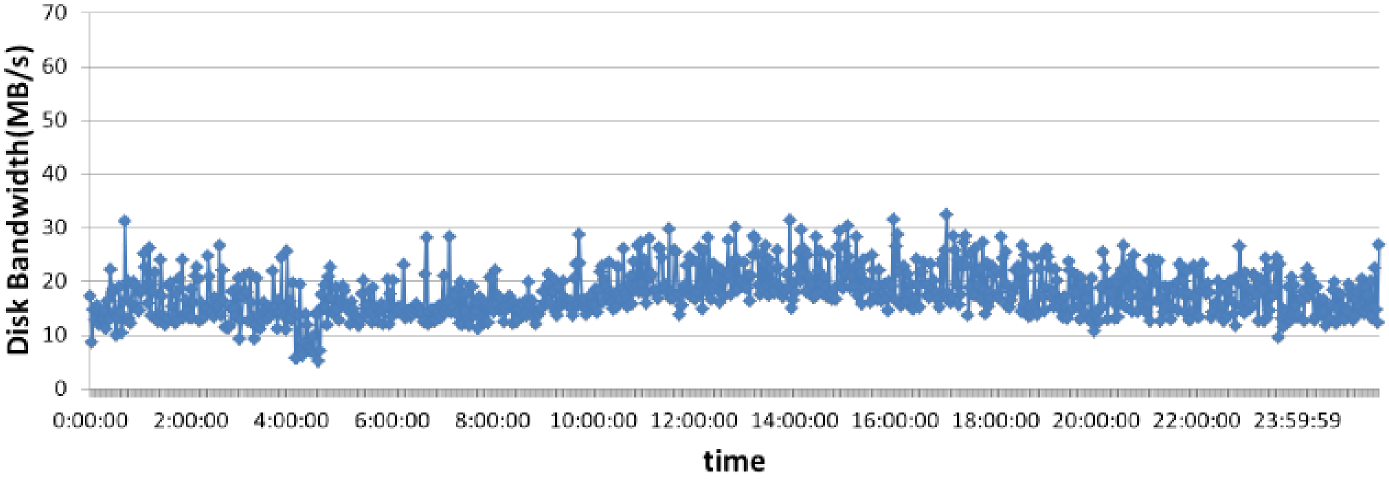


write

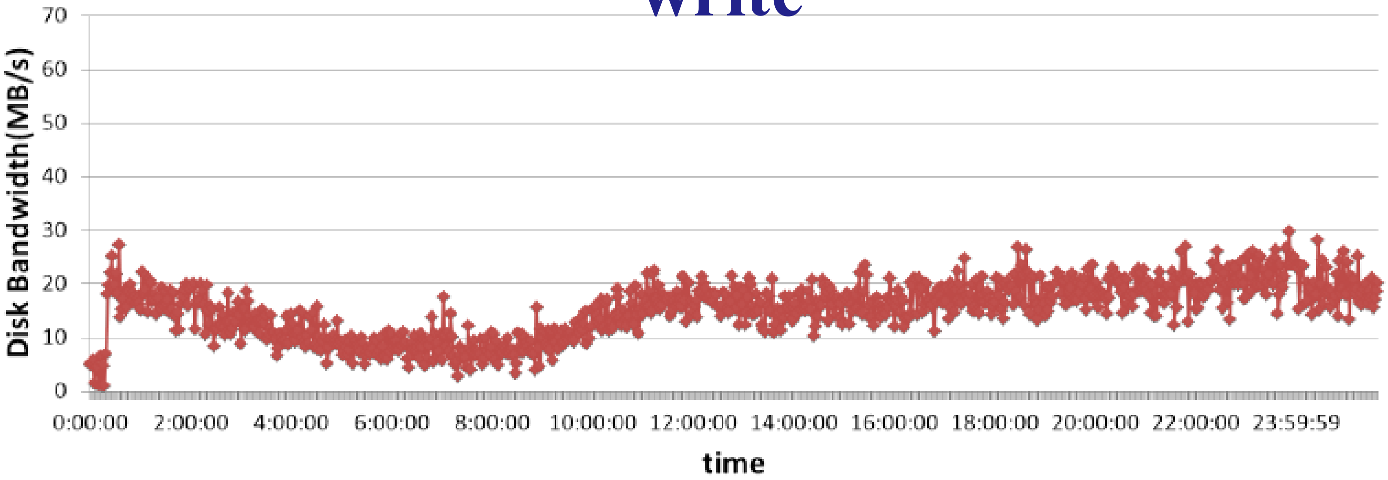


Reads

Disk Bandwidth at one Node of a Production Svstem



write



Reads

Summary

❑ Atlas is an object store using a two-tier design separating the managements of keys and values.

❑ Atlas uses a hardware-software co-design for high cost-effectiveness and energy efficiency.

❑ Atlas adopts the erasure coding technique for space-efficient data protection.