Cascade Mapping: Optimizing Memory Efficiency for Flash-based Key-value Caching

Kefei Wang and Feng Chen
Louisiana State University

SoCC '18 Carlsbad, CA

Key-value Systems in Internet Services

LEVELDB

cassandra

DynamoDB

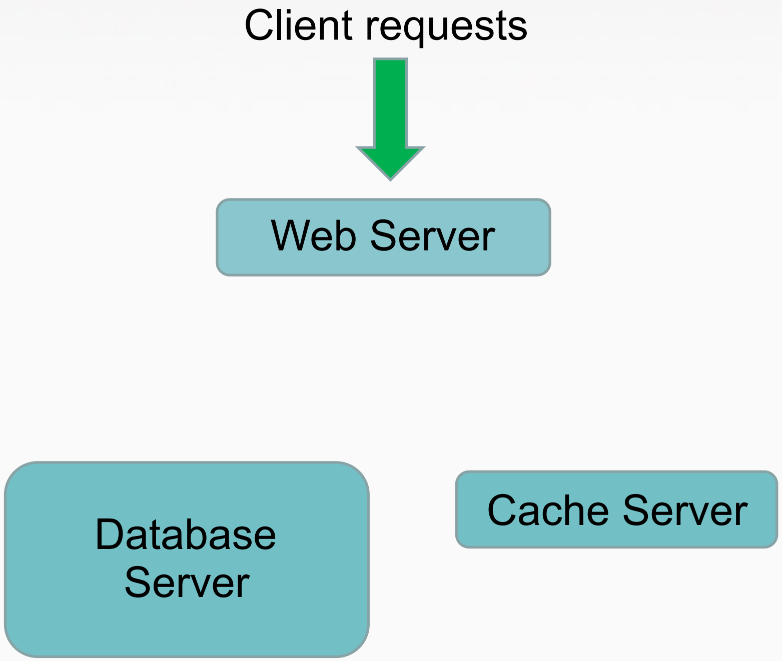
- Key-value systems are widely used today
  - Online shopping
  - Social media
  - Cloud storage
  - Big data

Key	Value
Product_ID	Product_Name
URL	Image

Key-value Caching

“First line of defense” in today’s Internet service

- High throughput
- Low latency



Operations:

SET

GET

DELETE

Key-value Caching

"First line of defense" in today's Internet service

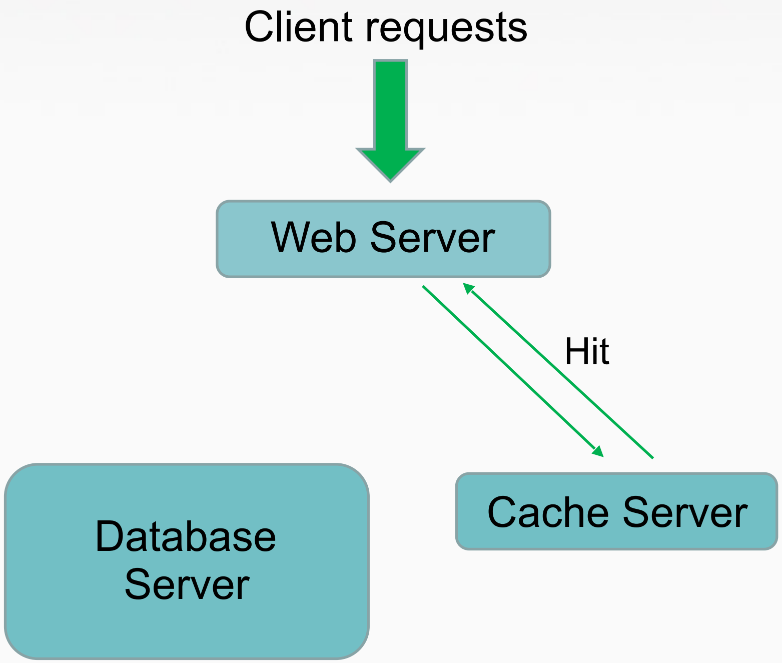
- High throughput
- Low latency

Operations:

SET

GET

DELETE



Client requests

Web Server

Hit

Cache Server

Database Server

Key-value Caching

“First line of defense” in today’s Internet service

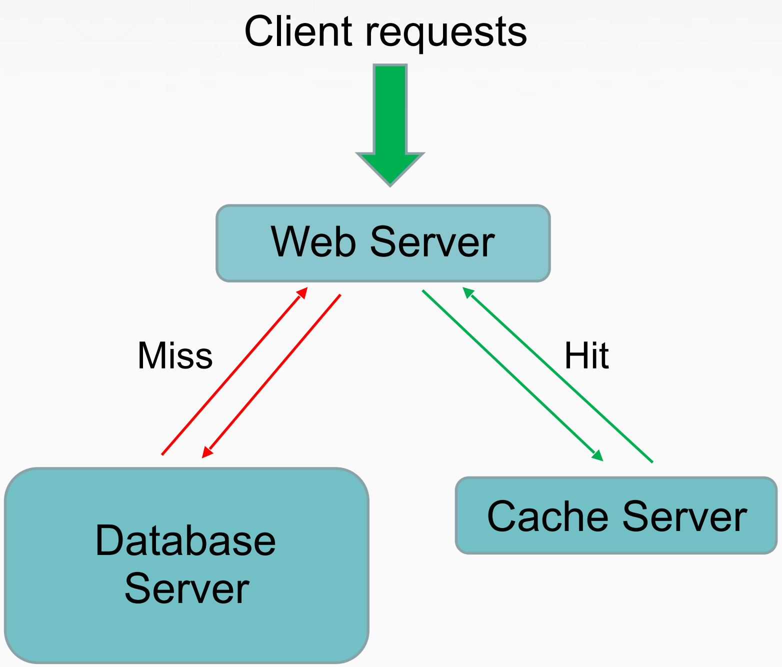
- High throughput
- Low latency

Operations:

SET

GET

DELETE

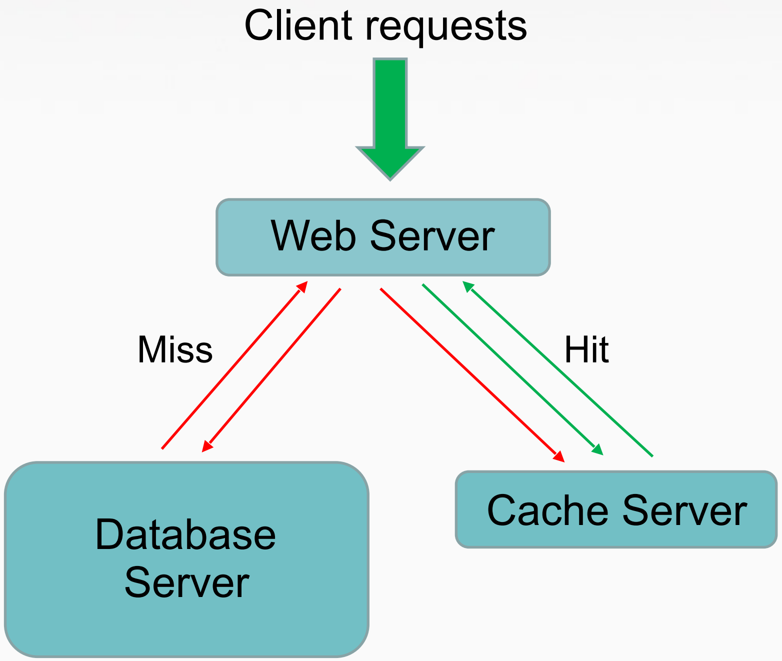


Key-value Caching

“First line of defense” in today’s Internet service

- High throughput
- Low latency

MEMCACHED



Operations:

SET

GET

DELETE

Flash-based Key-value Caching

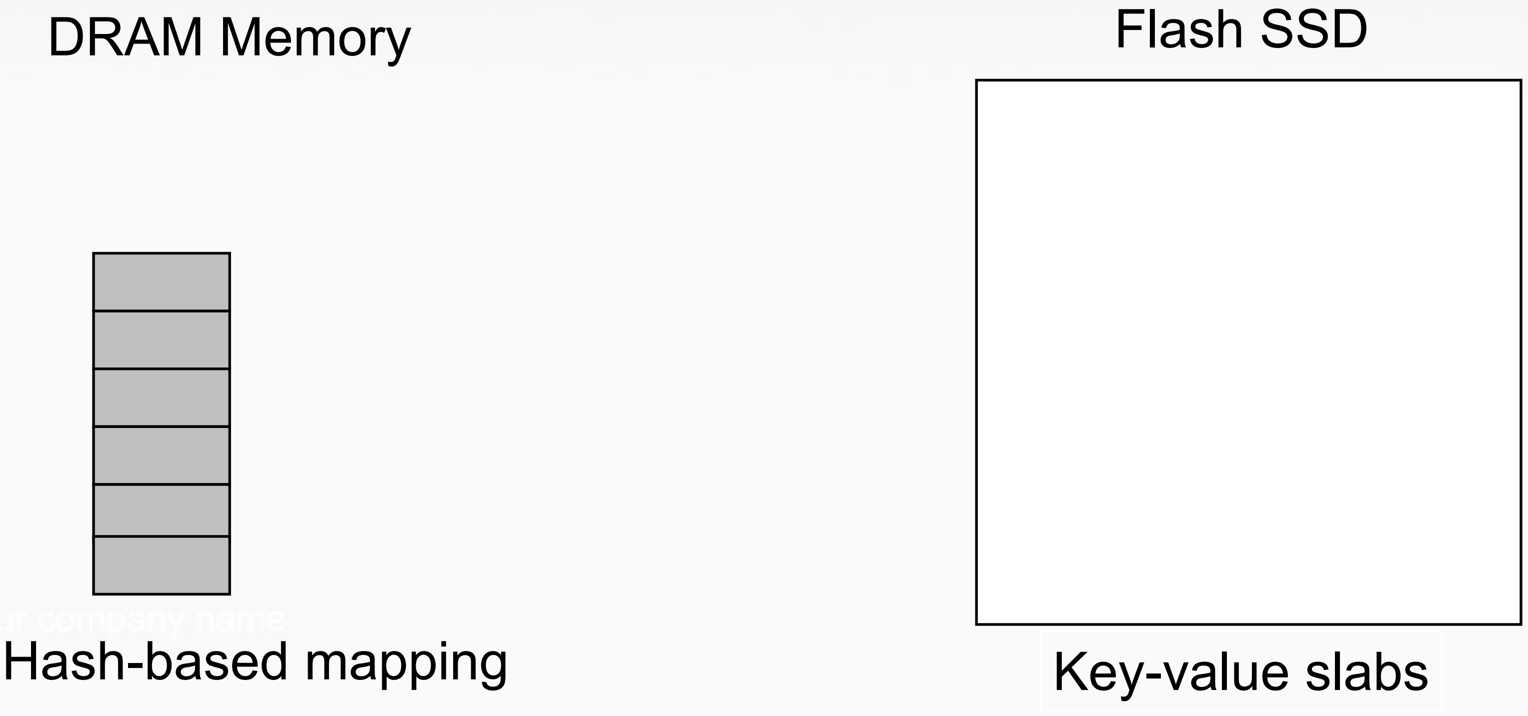
- In-flash key-value caches
  - Key-values are stored in commercial flash SSDs
  - Example: Facebook’s McDipper, Twitter’s Fatcache
- Key features
  - Memcached compatible (*SET*, *GET*, *DELETE*)
  - Advantages: low cost and high performance
    - McDipper: reduce 90% deployed servers, 90% GETs < 1ms*

	Speed	**Power**	**Cost**	**Capacity**	**Persistency**
DRAM	High	High	High	Low	No
Flash	Low-	**Low+**	**Low+**	**High+**	**Yes+**

*https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/

Flash-based Key-value Caching

Data stored in flash and all the mappings in DRAM



DRAM Memory

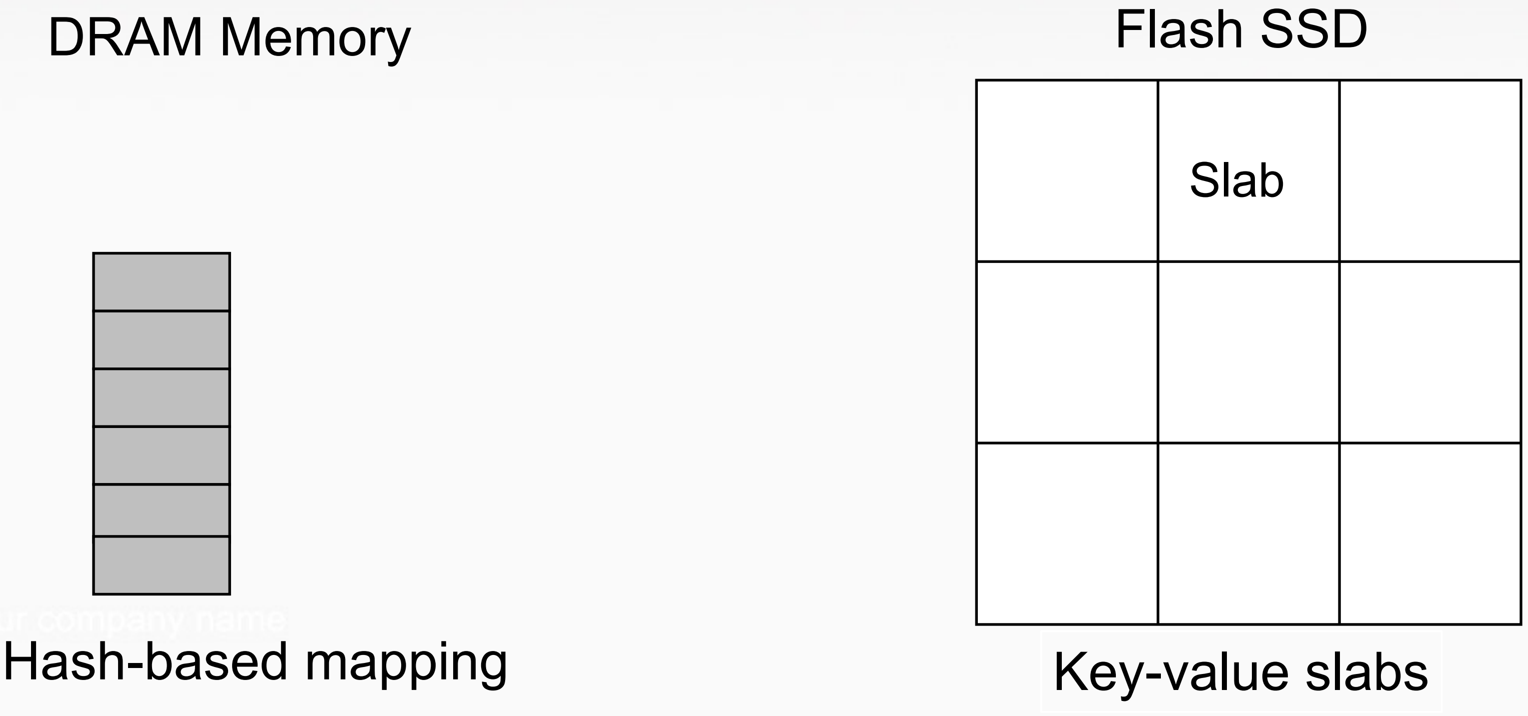
Flash SSD

Hash-based mapping

Key-value slabs

Flash-based Key-value Caching

Data stored in flash and all the mappings in DRAM



DRAM Memory

Hash-based mapping

Flash SSD

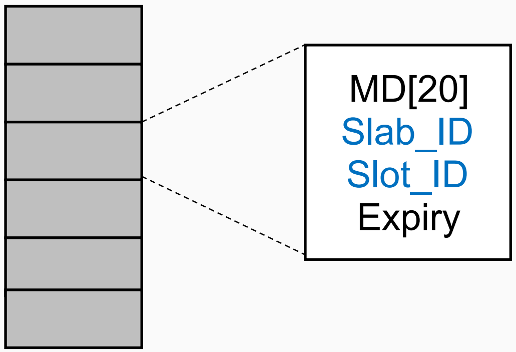
Slab

Key-value slabs

Flash-based Key-value Caching

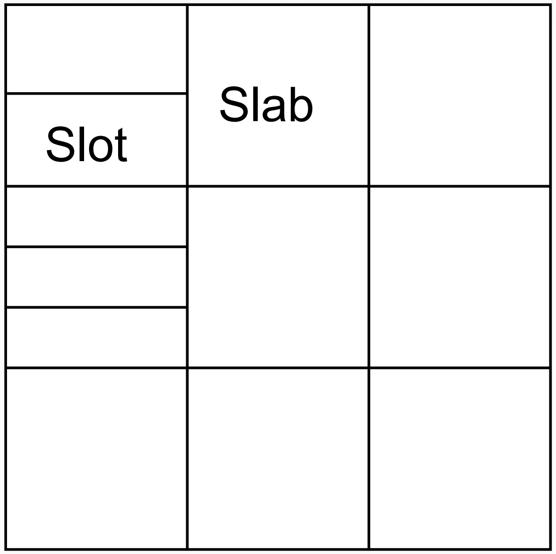
Data stored in flash and all the mappings in DRAM

DRAM Memory



Hash-based mapping

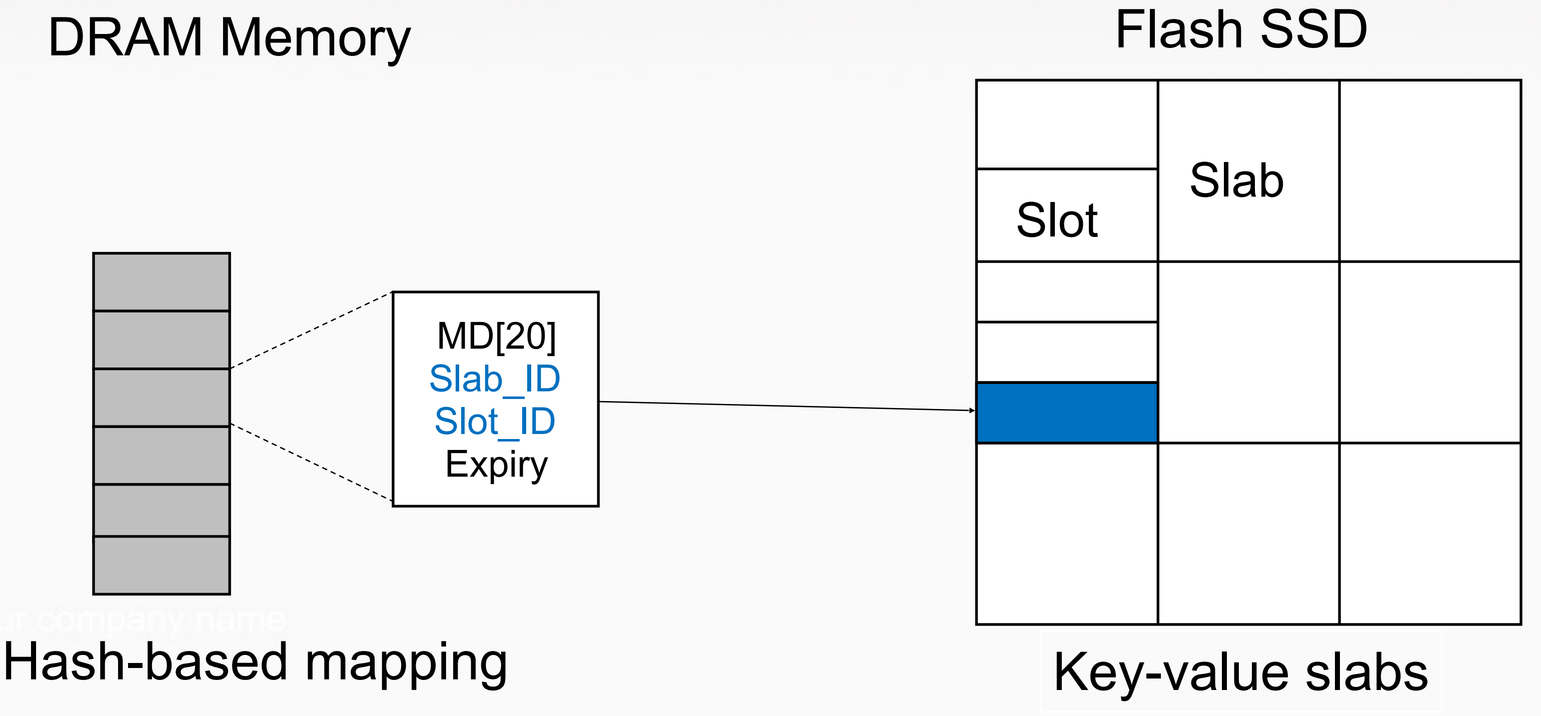
Flash SSD



Key-value slabs

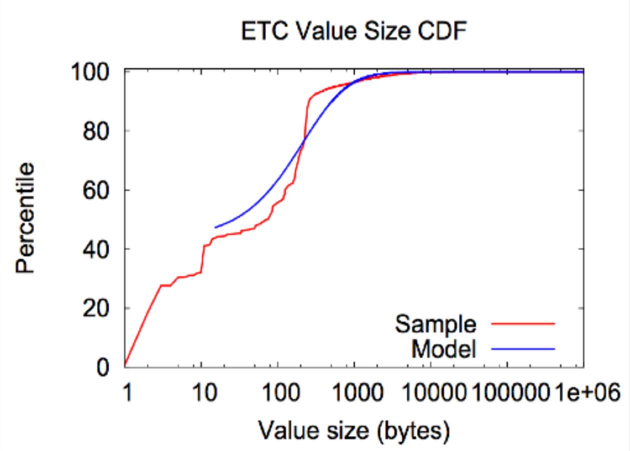
Flash-based Key-value Caching

Data stored in flash and all the mappings in DRAM



Scalability Challenge

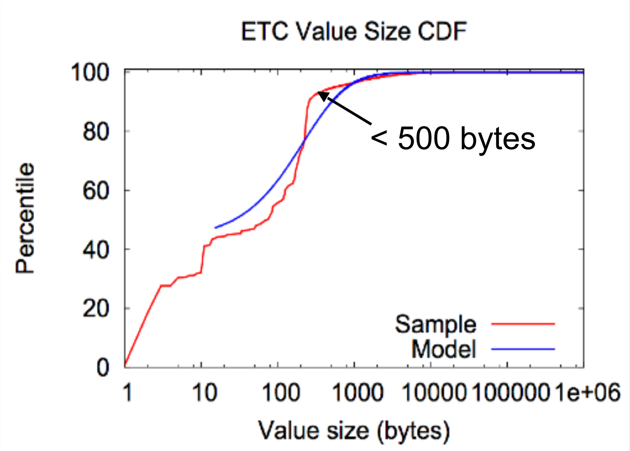
- High Index-to-data Ratio
  - Key-value cache is dominated by small items (90% < 500 bytes)
  - Key-value mapping entry size: 44 bytes in Fatcache



Atikoglu et al., "*Workload Analysis of A Large-scale Key-value Store*", in SIGMETRICS'12.

Scalability Challenge

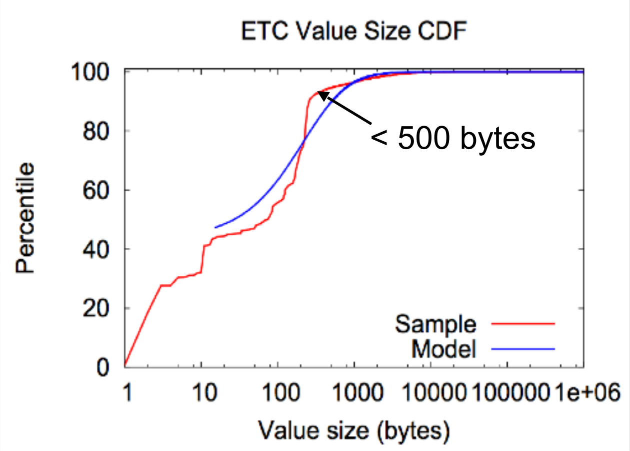
- High Index-to-data Ratio
  - Key-value cache is dominated by small items (90% < 500 bytes)
  - Key-value mapping entry size: 44 bytes in Fatcache



Atikoglu et al., “*Workload Analysis of A Large-scale Key-value Store*”, in SIGMETRICS’12.

Scalability Challenge

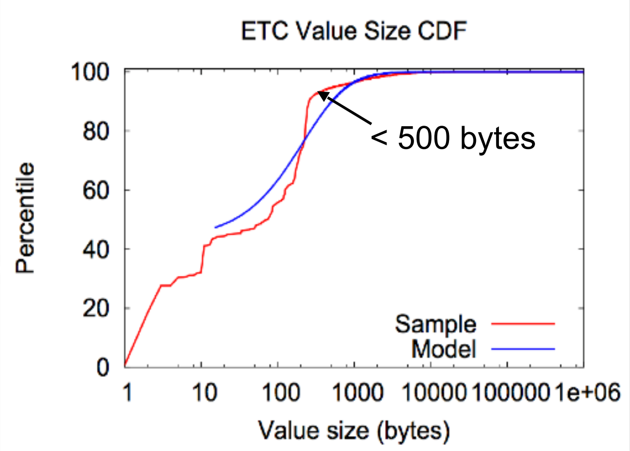
- High Index-to-data Ratio
  - Key-value cache is dominated by small items (90% < 500 bytes)
  - Key-value mapping entry size: 44 bytes in Fatcache
- Flash memory vs. DRAM memory
  - Capacity: Flash cache is 10-100x larger than memory-based cache
  - Price: 1-TB flash ($200-500), 1-TB DRAM (>$10,000)
  - Growth: flash (50-60% per year), DRAM (25-40% per year)



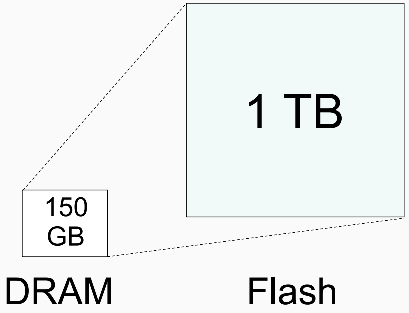
Atikoglu et al., "*Workload Analysis of A Large-scale Key-value Store*", in SIGMETRICS'12.

Scalability Challenge

- High Index-to-data Ratio
  - Key-value cache is dominated by small items (90% < 500 bytes)
  - Key-value mapping entry size: 44 bytes in Fatcache
- Flash memory vs. DRAM memory
  - Capacity: Flash cache is 10-100x larger than memory-based cache
  - Price: 1-TB flash ($200-500), 1-TB DRAM (>$10,000)
  - Growth: flash (50-60% per year), DRAM (25-40% per year)



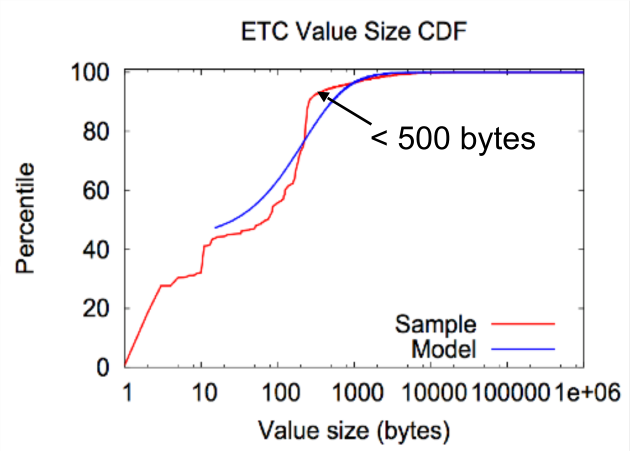
Atikoglu et al., "*Workload Analysis of A Large-scale Key-value Store*", in SIGMETRICS'12.



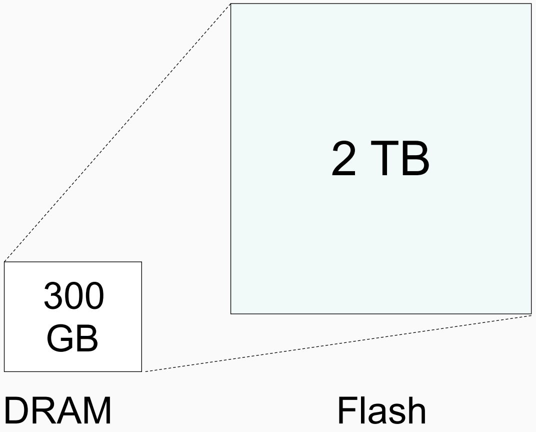
Assume average key-value size is 300 bytes

Scalability Challenge

- High Index-to-data Ratio
  - Key-value cache is dominated by small items (90% < 500 bytes)
  - Key-value mapping entry size: 44 bytes in Fatcache
- Flash memory vs. DRAM memory
  - Capacity: Flash cache is 10-100x larger than memory-based cache
  - Price: 1-TB flash ($200-500), 1-TB DRAM (>$10,000)
  - Growth: flash (50-60% per year), DRAM (25-40% per year)



Atikoglu et al., “*Workload Analysis of A Large-scale Key-value Store*”, in SIGMETRICS’12.



300 GB

2 TB

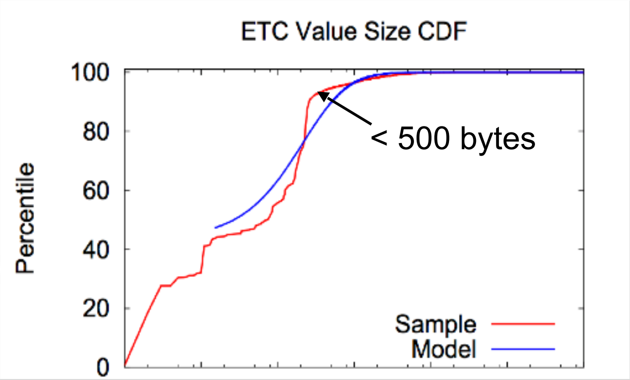
DRAM

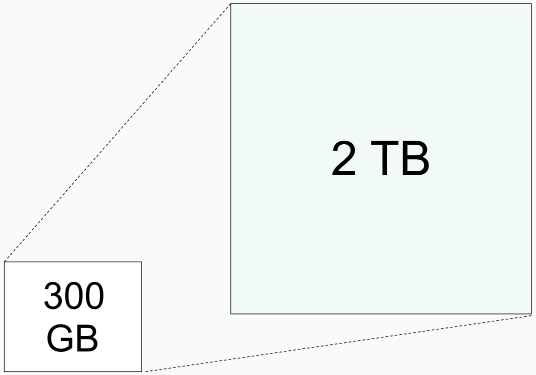
Flash

Assume average key-value size is 300 bytes

Scalability Challenge

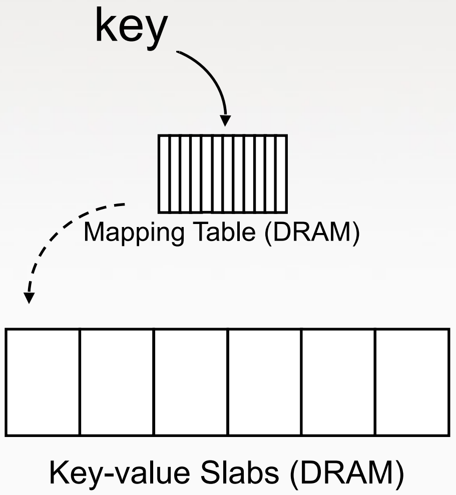
- High Index-to-data Ratio
  - Key-value cache is dominated by small items (90% < 500 bytes)
  - Key-value mapping entry size: 44 bytes in Fatcache
- Flash memory vs. DRAM memory
  - Capacity: Flash cache is 10-100x larger than memory-based cache
  - Price: 1-TB flash ($200-500), 1-TB DRAM (>$10,000)
  - Growth: flash (50-60% per year), DRAM (25-40% per year)





A **technical dilemma**: We have a lot of flash space to cache the data, but we don’t have enough DRAM to index the data

Evolution of Key-value Caching

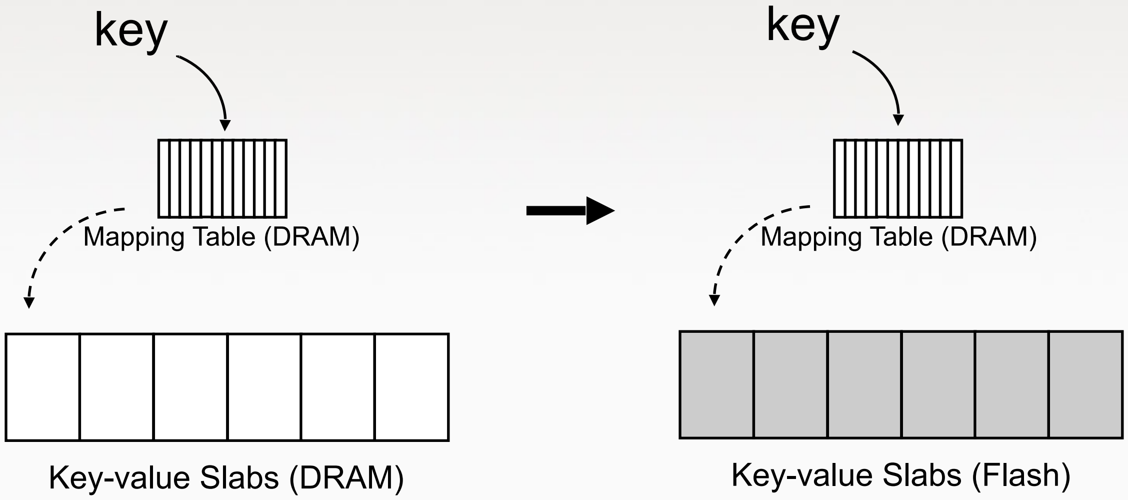


key

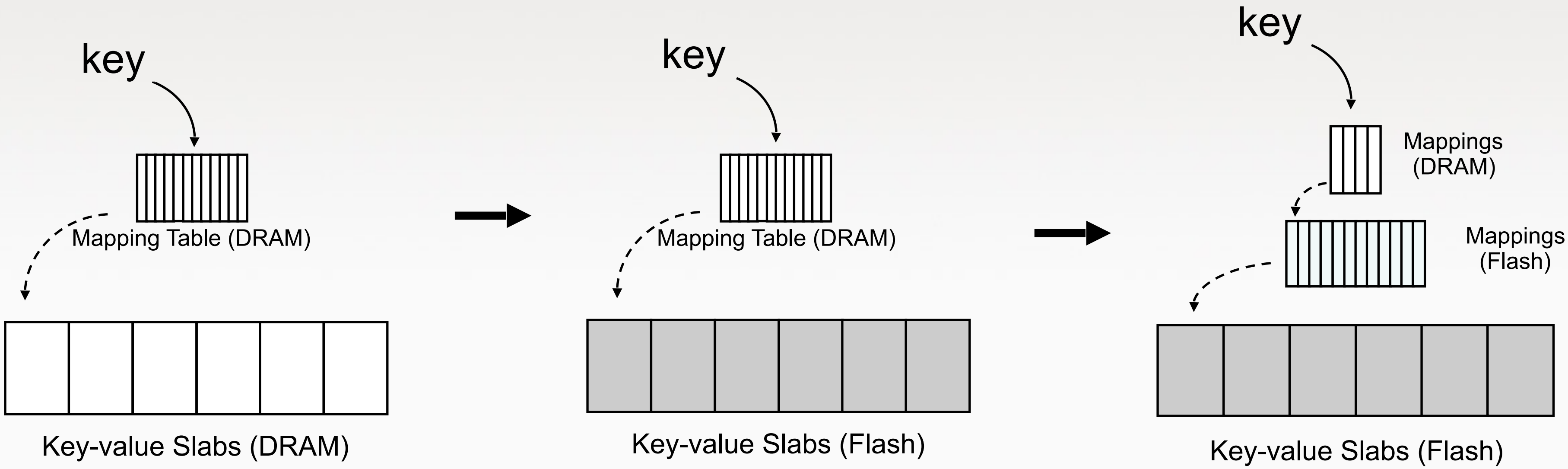
Mapping Table (DRAM)

Key-value Slabs (DRAM)

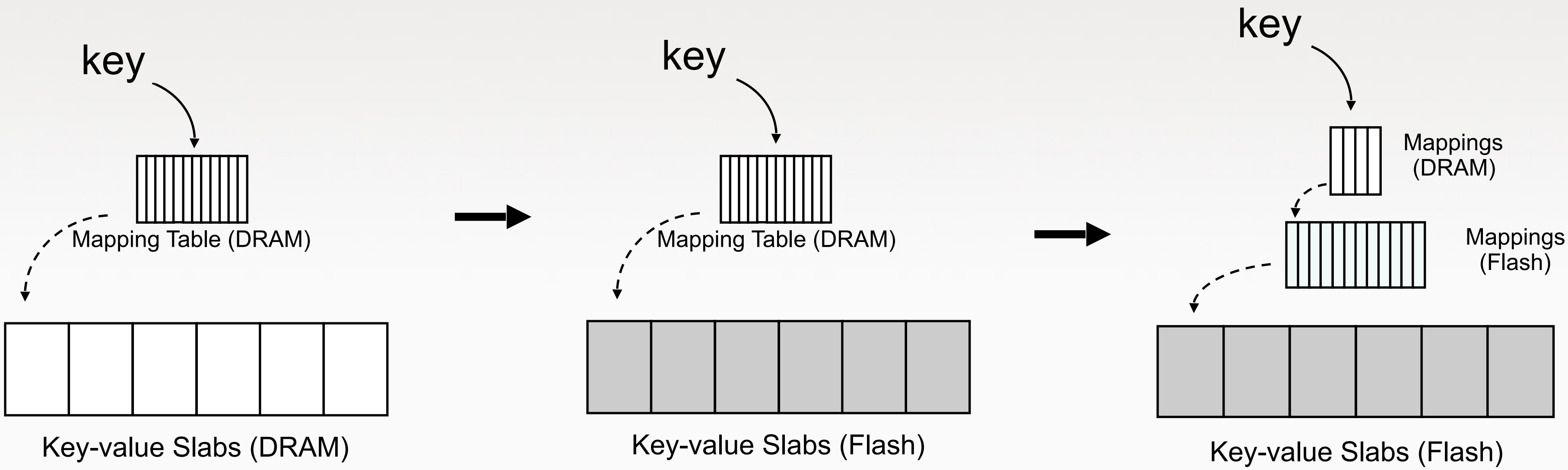
Evolution of Key-value Caching



Evolution of Key-value Caching

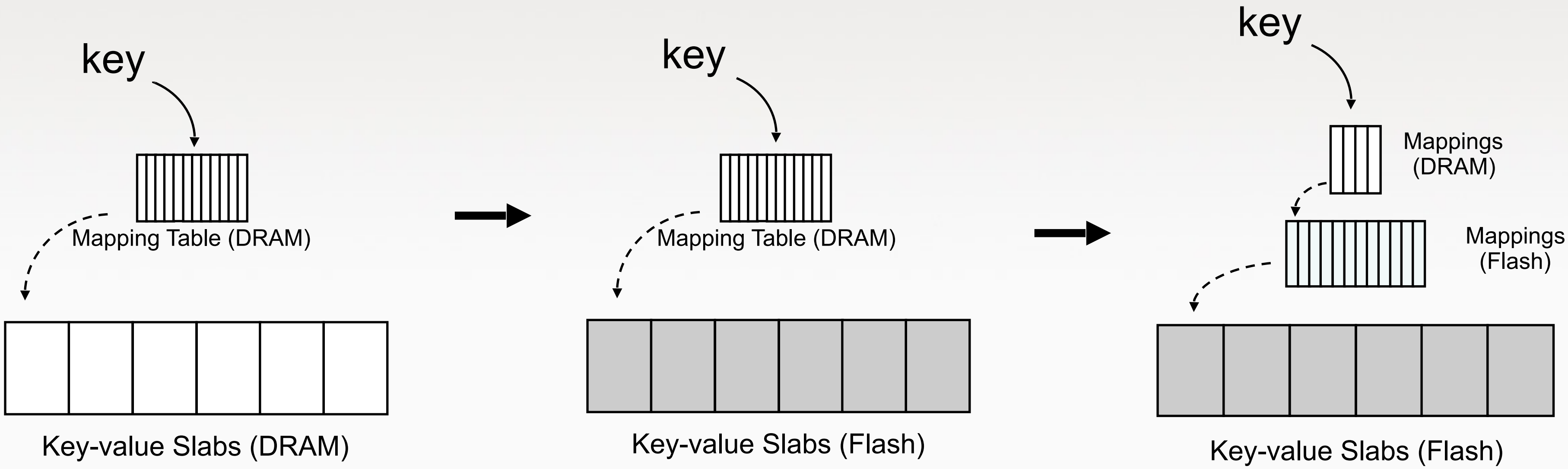


Evolution of Key-value Caching



Zero Flash I/O

Evolution of Key-value Caching



key

Mapping Table (DRAM)

Key-value Slabs (DRAM)

Zero Flash I/O

key

Mapping Table (DRAM)

Key-value Slabs (Flash)

One Flash I/O

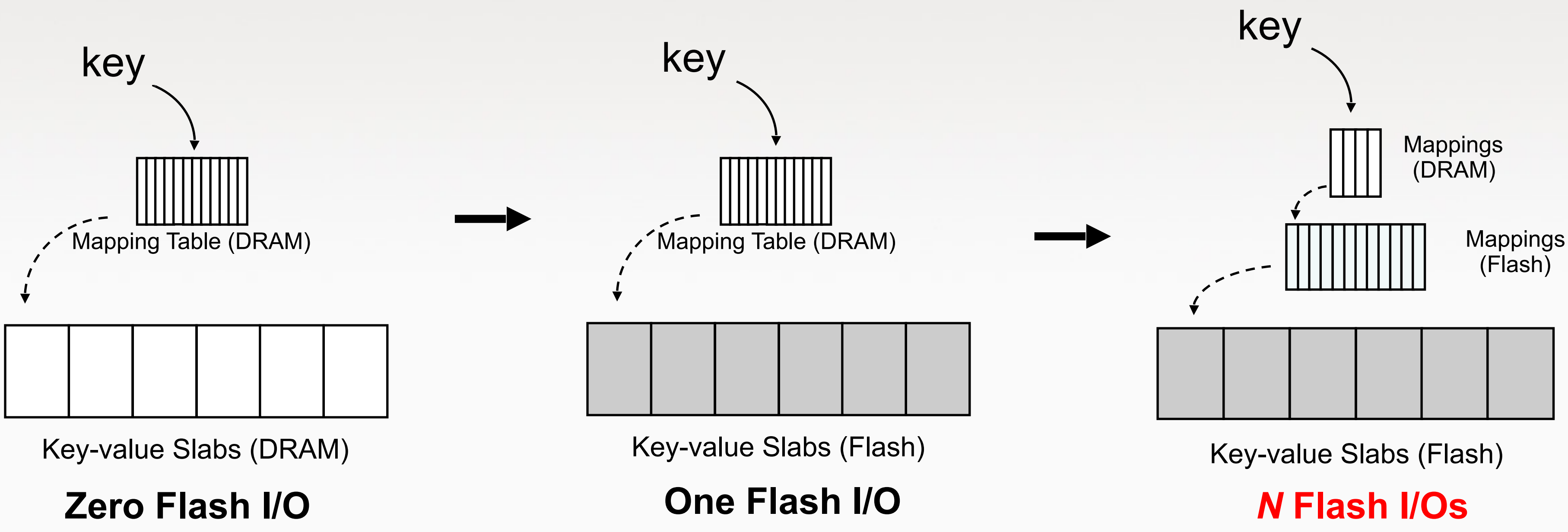
key

Mappings
(DRAM)

Mappings
(Flash)

Key-value Slabs (Flash)

Evolution of Key-value Caching



key

Mapping Table (DRAM)

Key-value Slabs (DRAM)

Zero Flash I/O

key

Mapping Table (DRAM)

Key-value Slabs (Flash)

One Flash I/O

key

Mappings (DRAM)

Mappings (Flash)

Key-value Slabs (Flash)

***N* Flash I/Os**

- Leverage the strong locality to differentiate hot and cold mappings
  - Hold the most popular mappings in a small in-DRAM mapping structure
  - Leave the majority mappings in a large in-flash mapping structure

Outline

- Cascade mapping design
- Optimizations
- Evaluation results
- Conclusions

Cascade Mapping

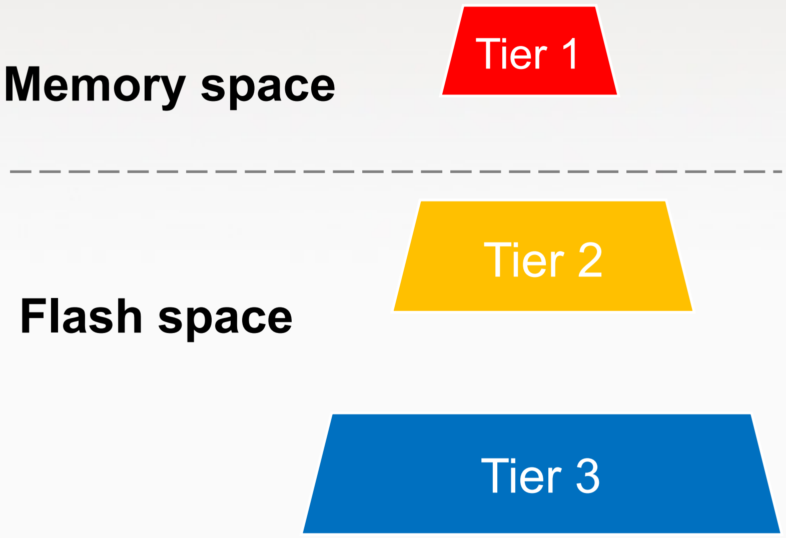
Hierarchical Mapping Structure

- Tier 1 – Hot mappings
  - Hash index based search in memory
- Tier 2 – Warm mappings
  - High-bandwidth quick scan in flash
- Tier 3 – Cold mappings
  - Efficient linked-list structure in flash

Cascade Mapping

Hierarchical Mapping Structure

- Tier 1 – Hot mappings
  - Hash index based search in memory
- Tier 2 – Warm mappings
  - High-bandwidth quick scan in flash
- Tier 3 – Cold mappings
  - Efficient linked-list structure in flash



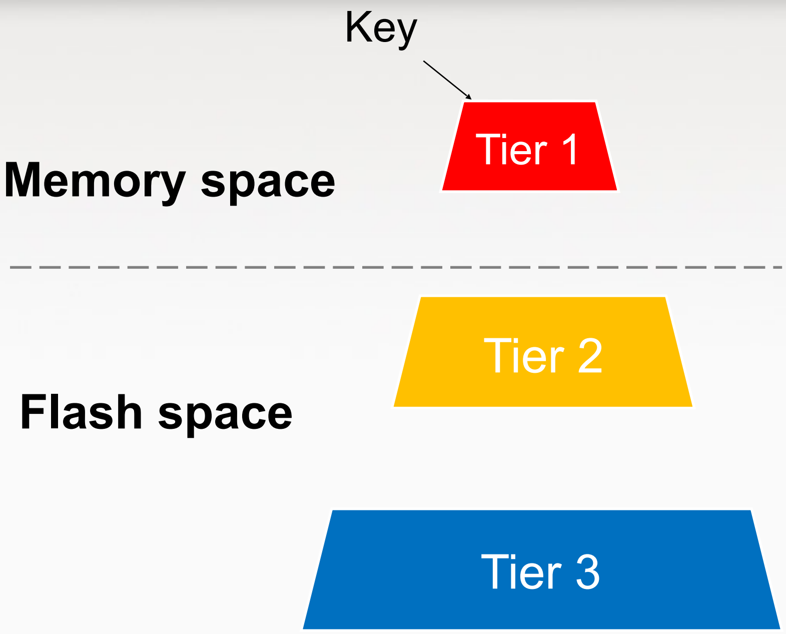
Memory space

Flash space

Cascade Mapping

Hierarchical Mapping Structure

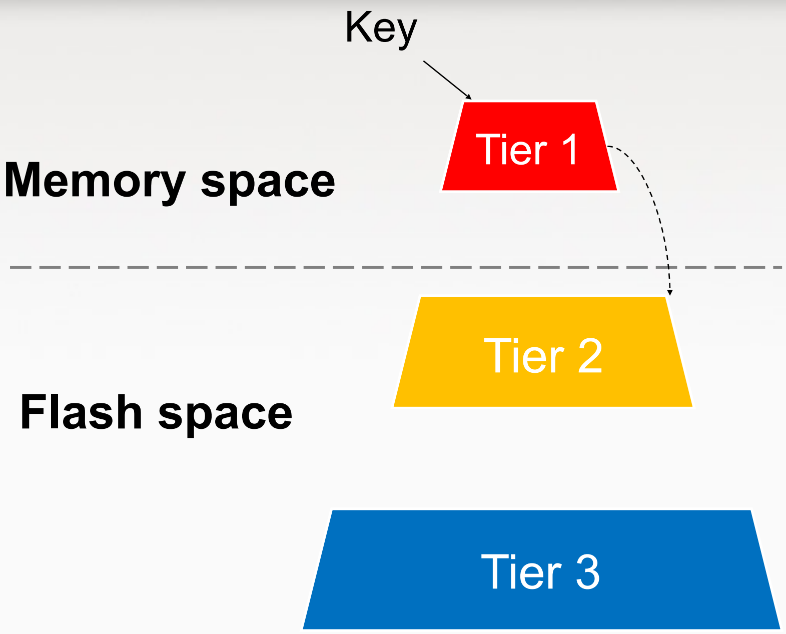
- Tier 1 – Hot mappings
  - Hash index based search in memory
- Tier 2 – Warm mappings
  - High-bandwidth quick scan in flash
- Tier 3 – Cold mappings
  - Efficient linked-list structure in flash



Cascade Mapping

Hierarchical Mapping Structure

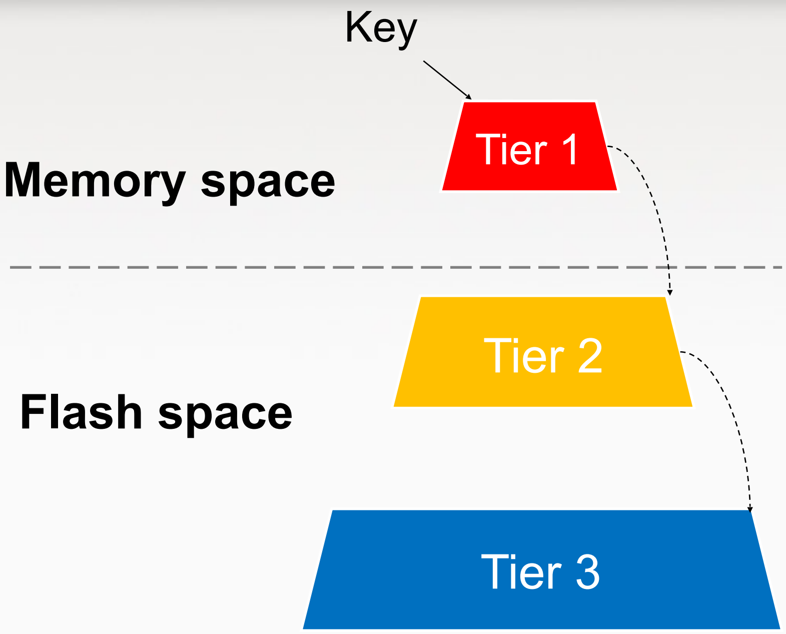
- Tier 1 – Hot mappings
  - Hash index based search in memory
- Tier 2 – Warm mappings
  - High-bandwidth quick scan in flash
- Tier 3 – Cold mappings
  - Efficient linked-list structure in flash



Cascade Mapping

Hierarchical Mapping Structure

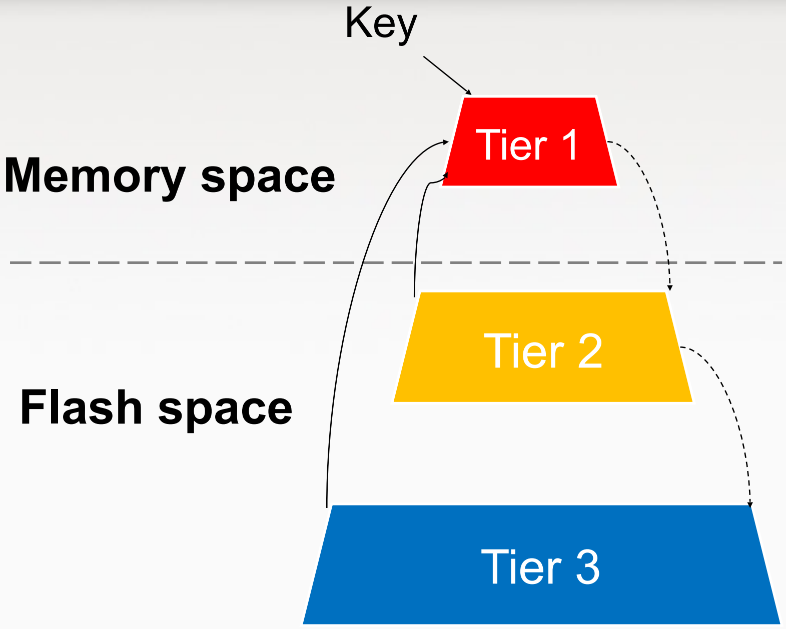
- Tier 1 – Hot mappings
  - Hash index based search in memory
- Tier 2 – Warm mappings
  - High-bandwidth quick scan in flash
- Tier 3 – Cold mappings
  - Efficient linked-list structure in flash



Cascade Mapping

Hierarchical Mapping Structure

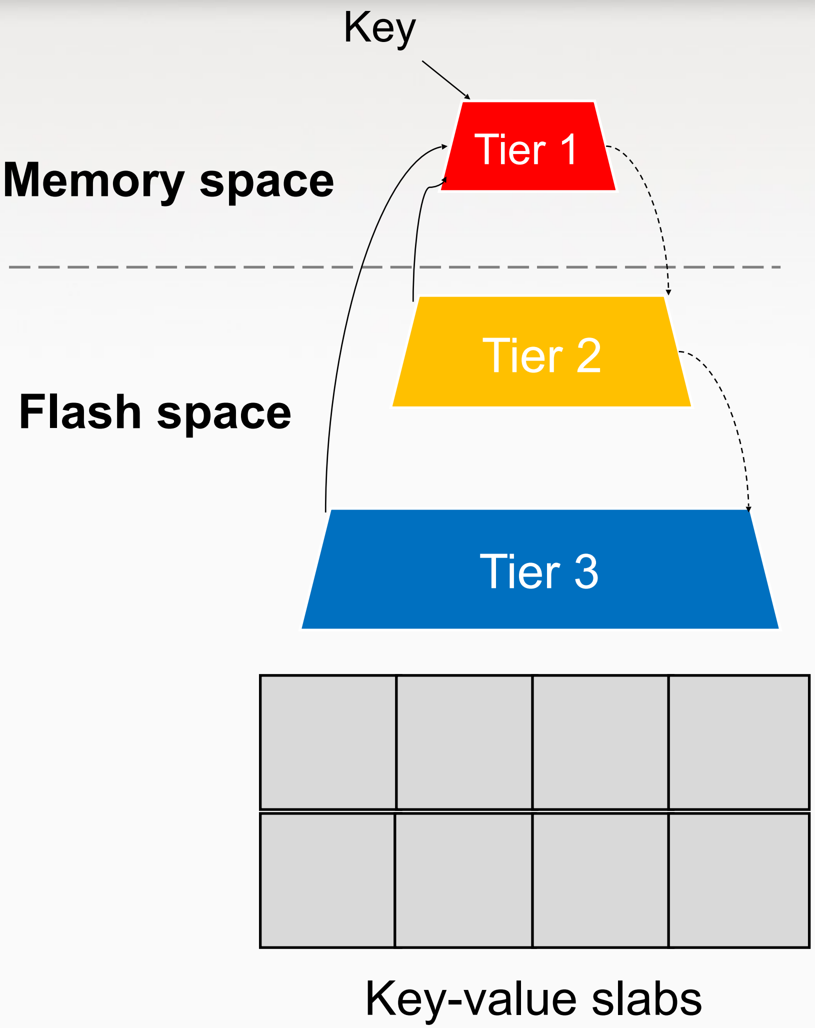
- Tier 1 – Hot mappings
  - Hash index based search in memory
- Tier 2 – Warm mappings
  - High-bandwidth quick scan in flash
- Tier 3 – Cold mappings
  - Efficient linked-list structure in flash



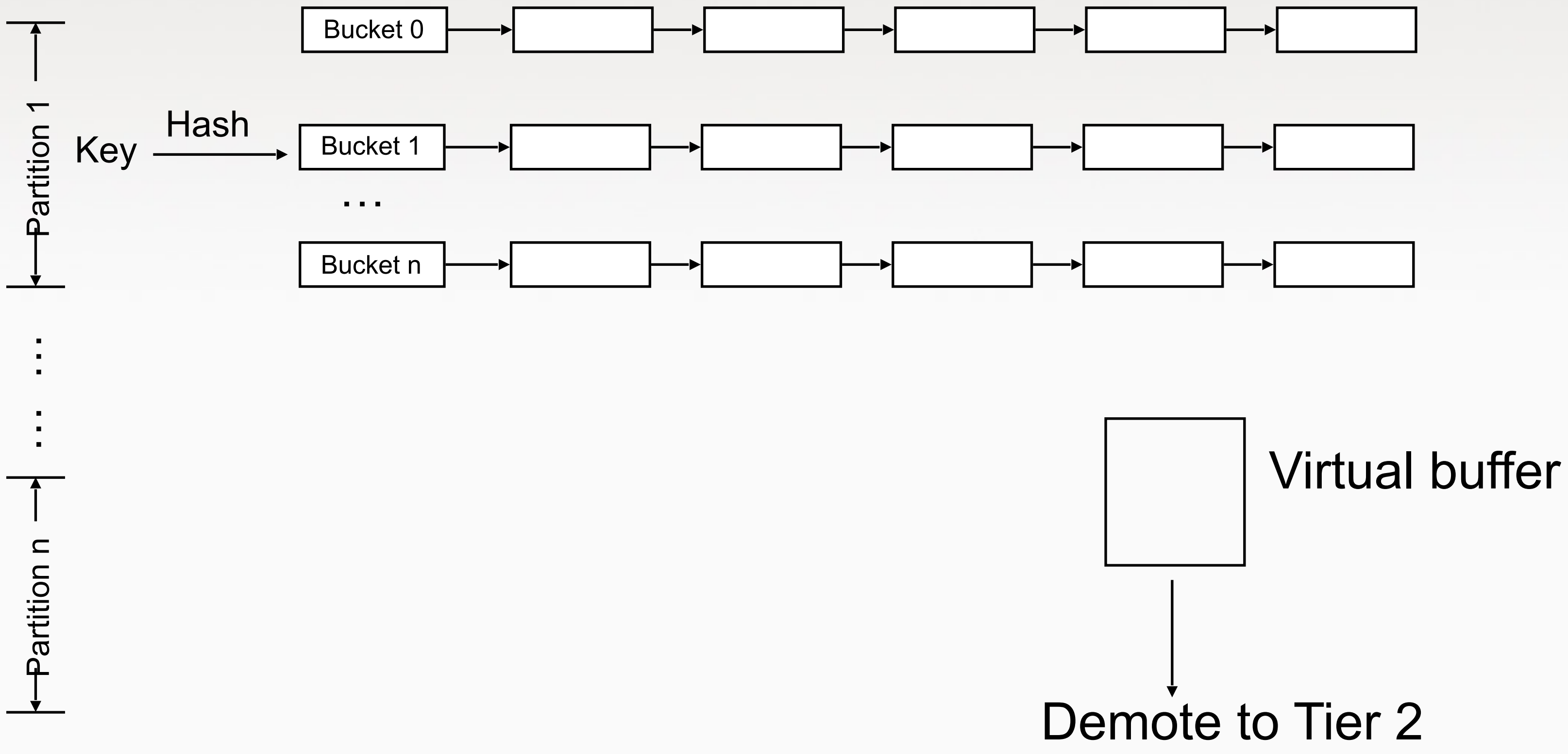
Cascade Mapping

Hierarchical Mapping Structure

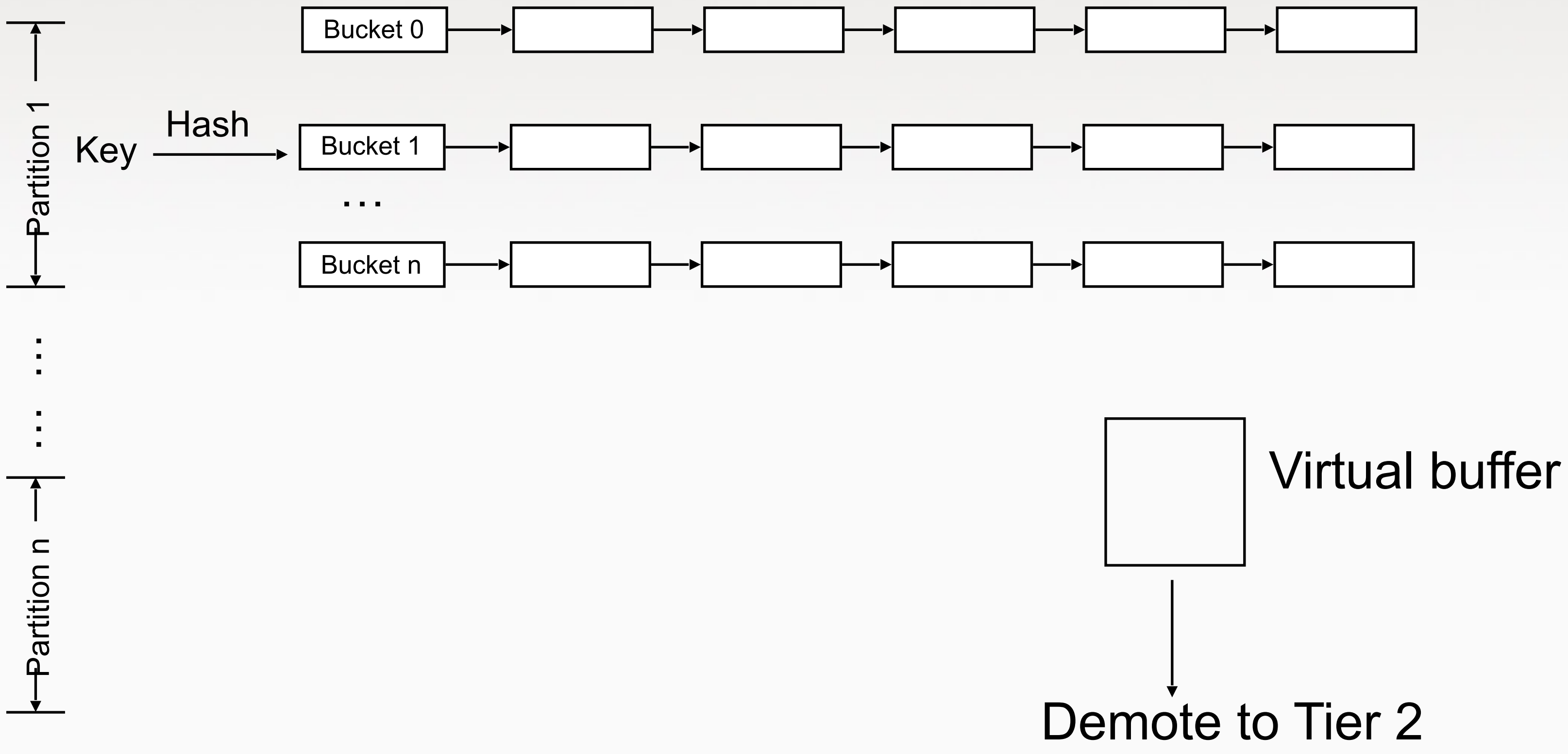
- Tier 1 – Hot mappings
  - Hash index based search in memory
- Tier 2 – Warm mappings
  - High-bandwidth quick scan in flash
- Tier 3 – Cold mappings
  - Efficient linked-list structure in flash



Tier 1: A Mapping Table in Memory



Tier 1: A Mapping Table in Memory



Partition 1

Key

Hash

Bucket 0

Bucket 1

…

Bucket n

…

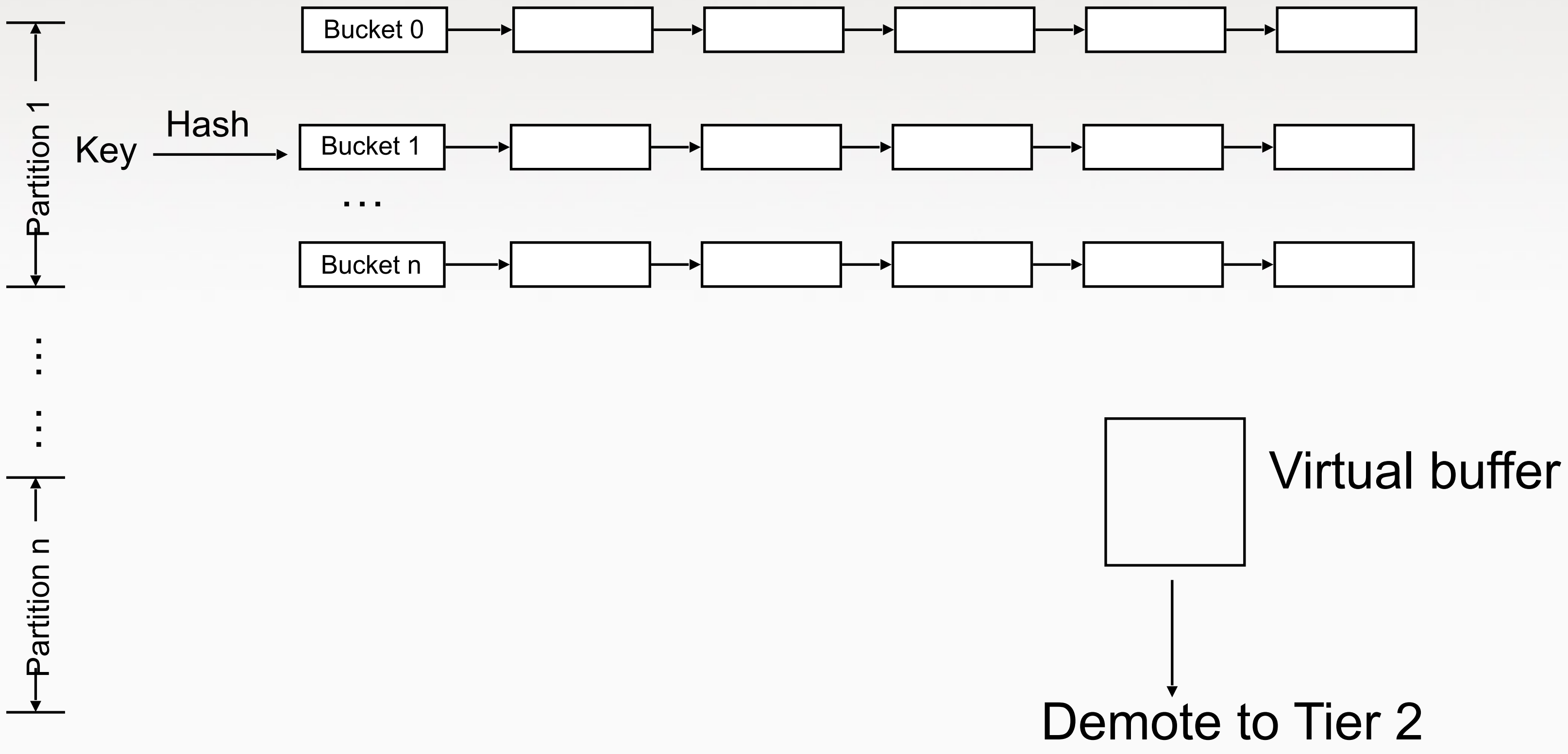
…

Partition n

Virtual buffer

Demote to Tier 2

Tier 1: A Mapping Table in Memory



Partition 1

Key

Hash

Bucket 0

Bucket 1

…

Bucket n

…

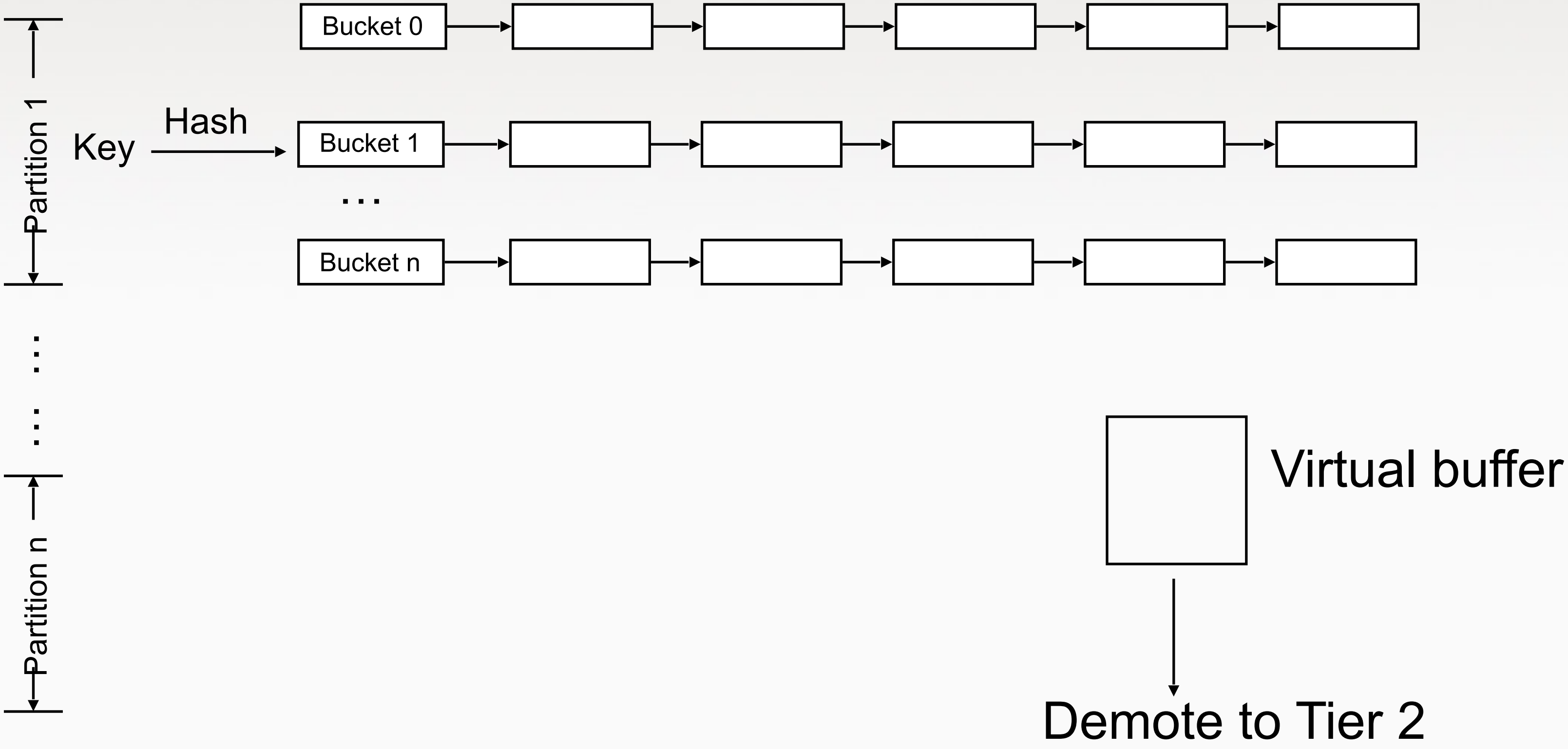
…

Partition n

Virtual buffer

Demote to Tier 2

Tier 1: A Mapping Table in Memory



Partition 1

Key

Hash

Bucket 0

Bucket 1

…

Bucket n

…

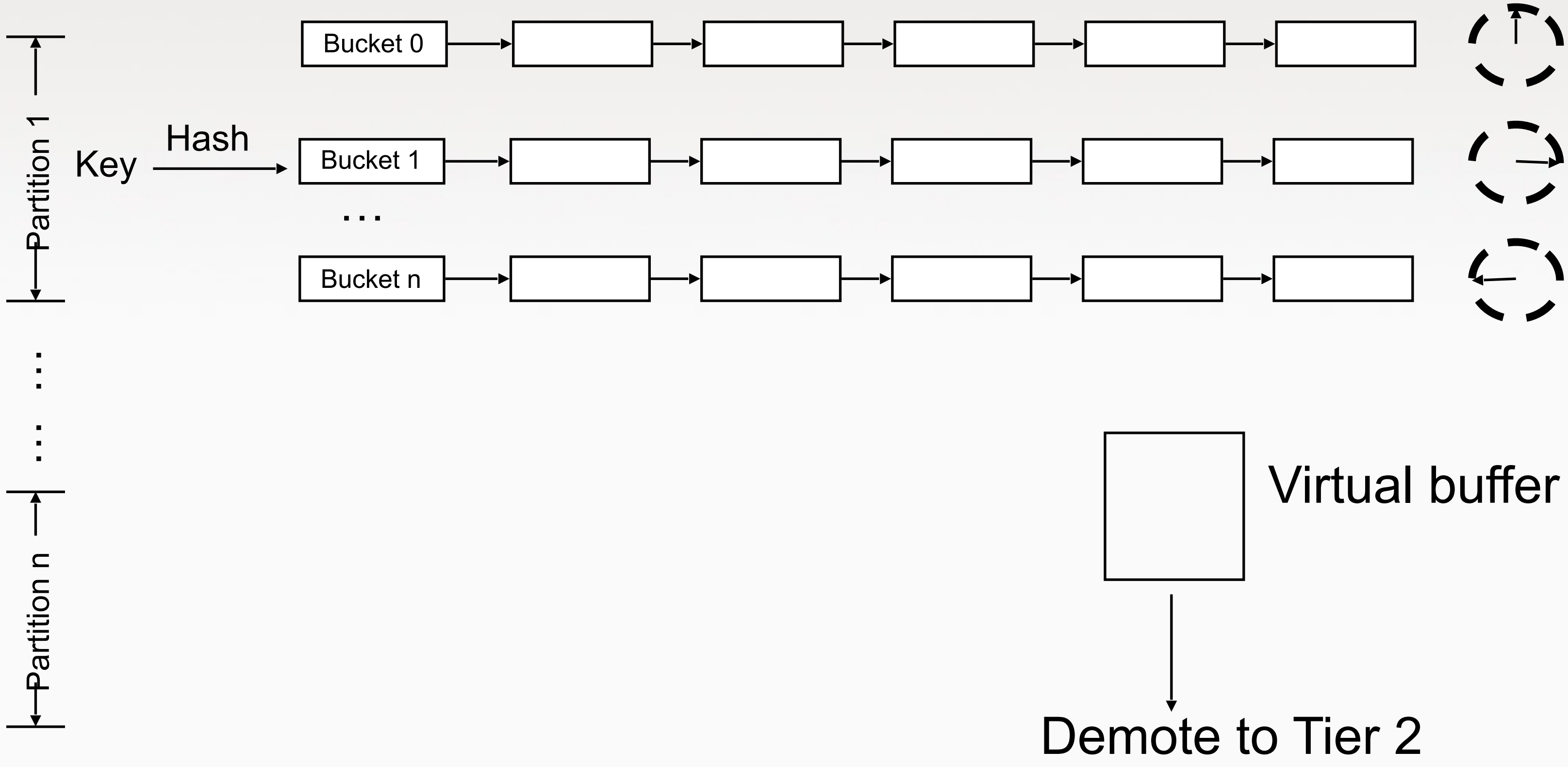
…

Partition n

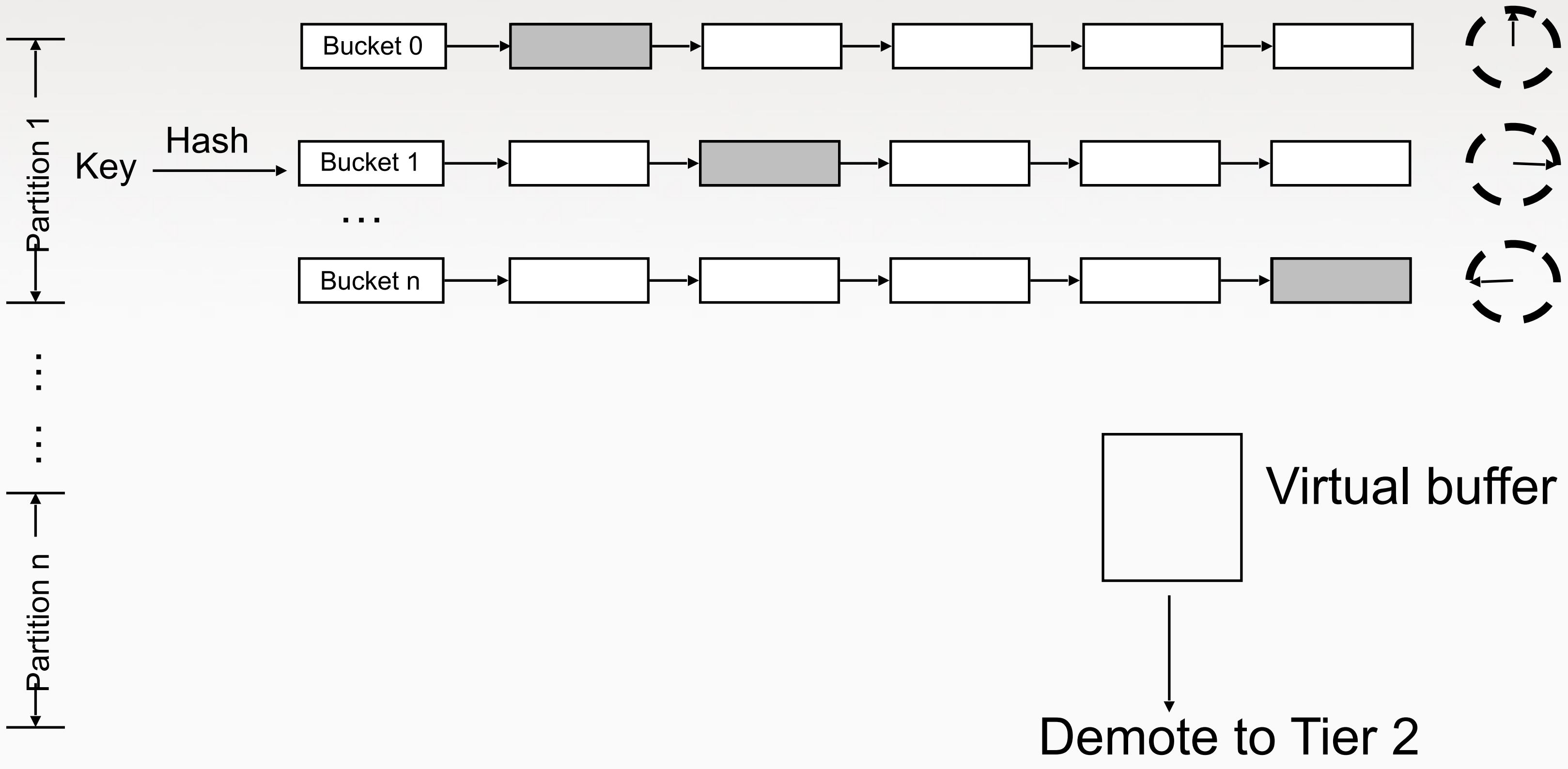
Virtual buffer

Demote to Tier 2

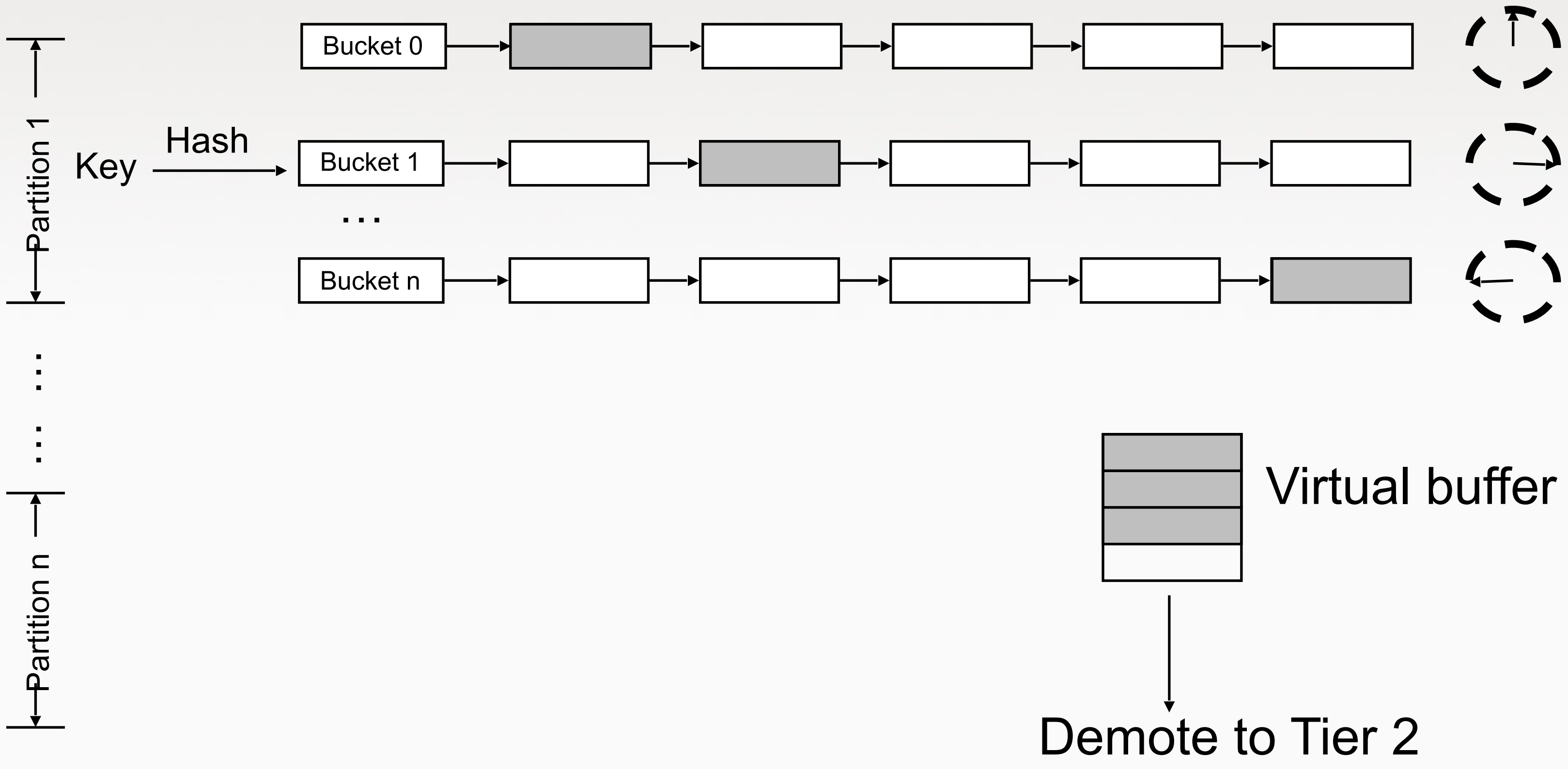
Tier 1: A Mapping Table in Memory



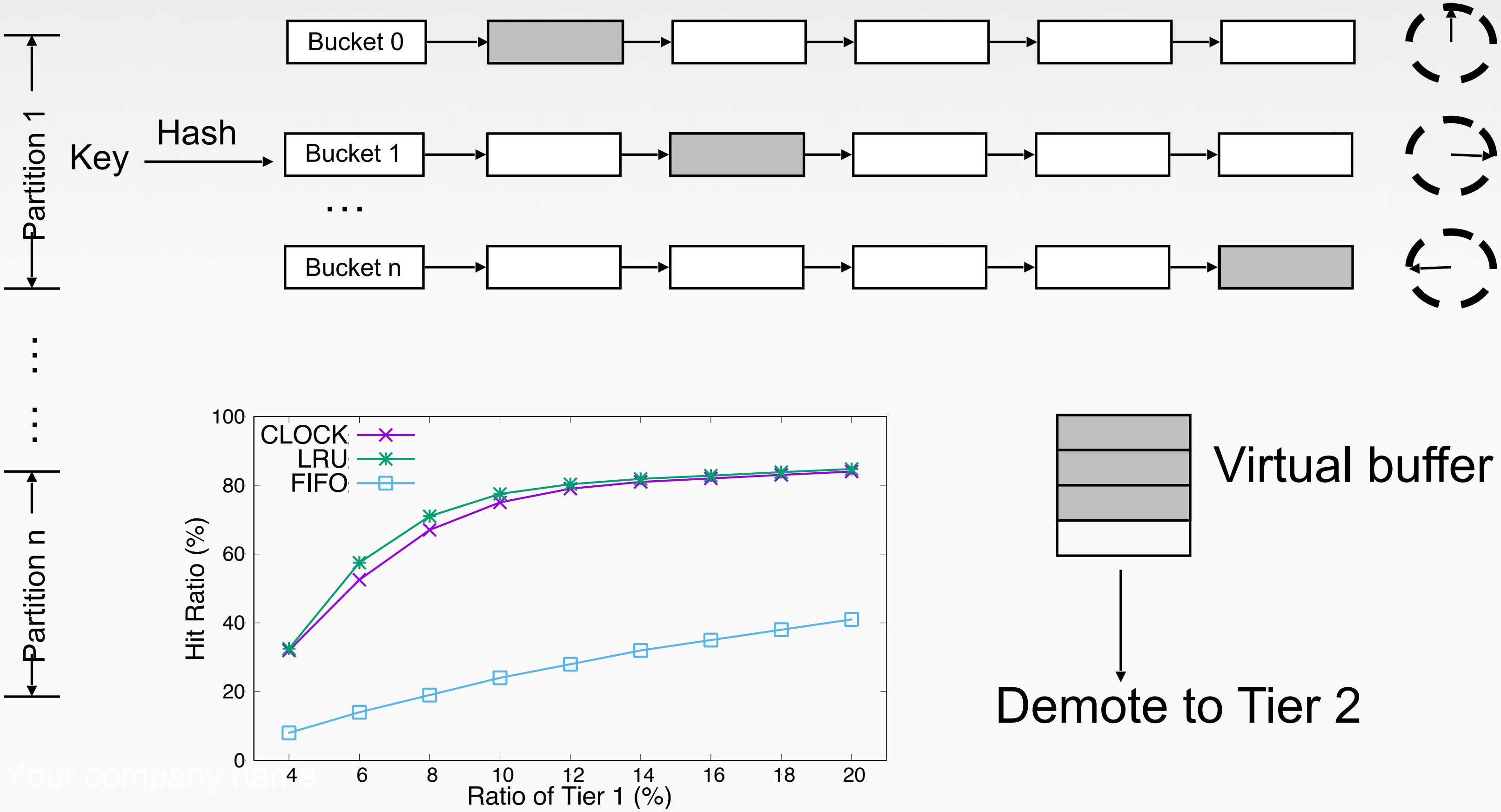
Tier 1: A Mapping Table in Memory



Tier 1: A Mapping Table in Memory



Tier 1: A Mapping Table in Memory

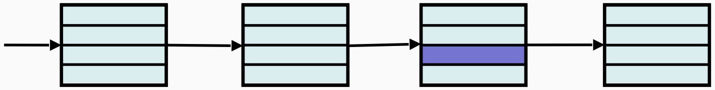


Tier 2: Direct Indexing in Flash

- Direct mapping block
  - A set of mapping entries demoted from Tier 1

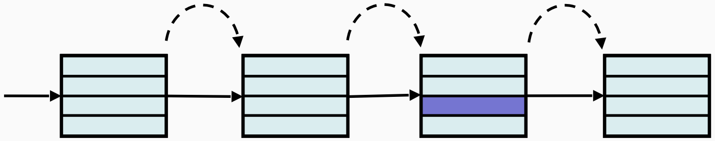
Tier 2: Direct Indexing in Flash

- Direct mapping block
  - A set of mapping entries demoted from Tier 1



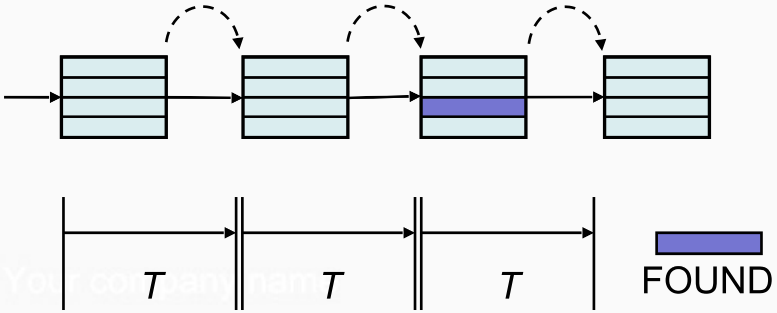
Tier 2: Direct Indexing in Flash

- Direct mapping block
  - A set of mapping entries demoted from Tier 1



Tier 2: Direct Indexing in Flash

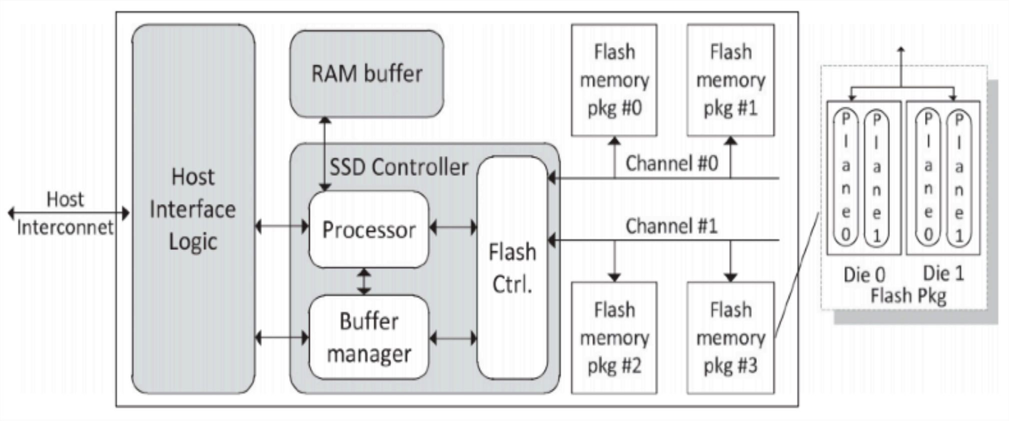
- Direct mapping block
  - A set of mapping entries demoted from Tier 1



Serial Search: 3x *T*

Tier 2: Direct Indexing in Flash

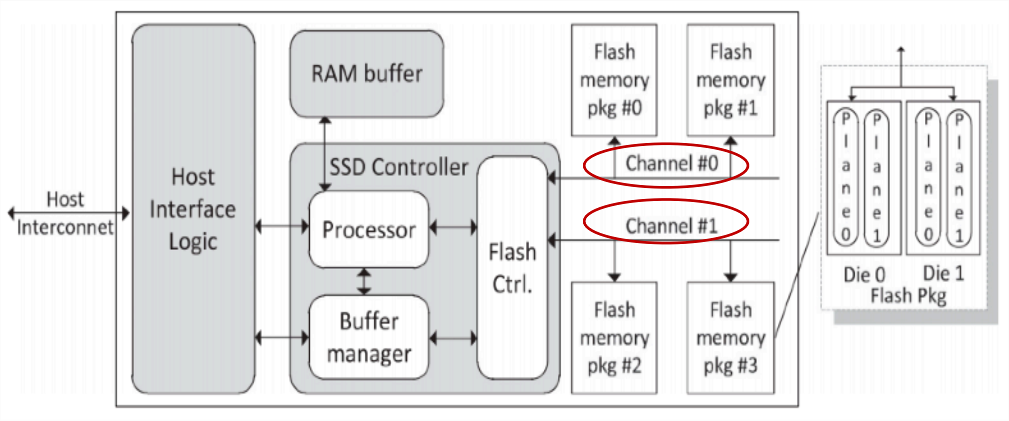
- Direct mapping block
  - A set of mapping entries demoted from Tier 1



Chen et al., "Internal Parallelism of Flash-based Solid State Drives", *ACM Transactions on Storage*, 12:3, May 2016

Tier 2: Direct Indexing in Flash

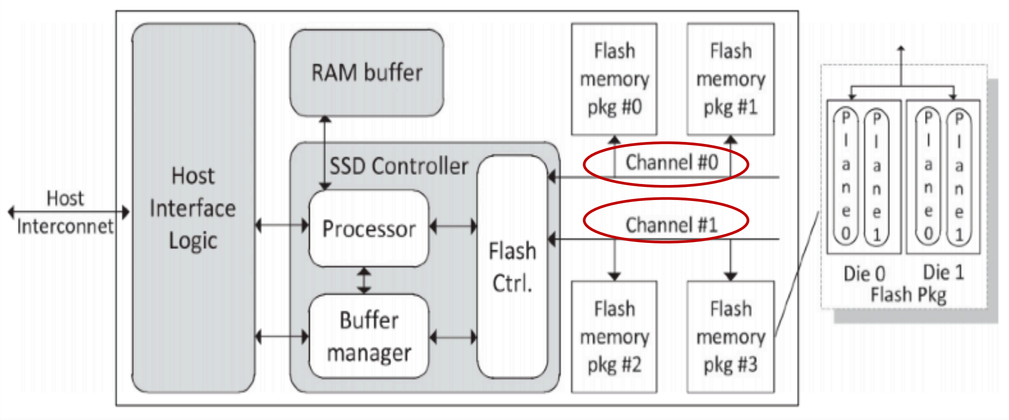
- Direct mapping block
  - A set of mapping entries demoted from Tier 1



Chen et al., “Internal Parallelism of Flash-based Solid State Drives”, *ACM Transactions on Storage*, 12:3, May 2016

Tier 2: Direct Indexing in Flash

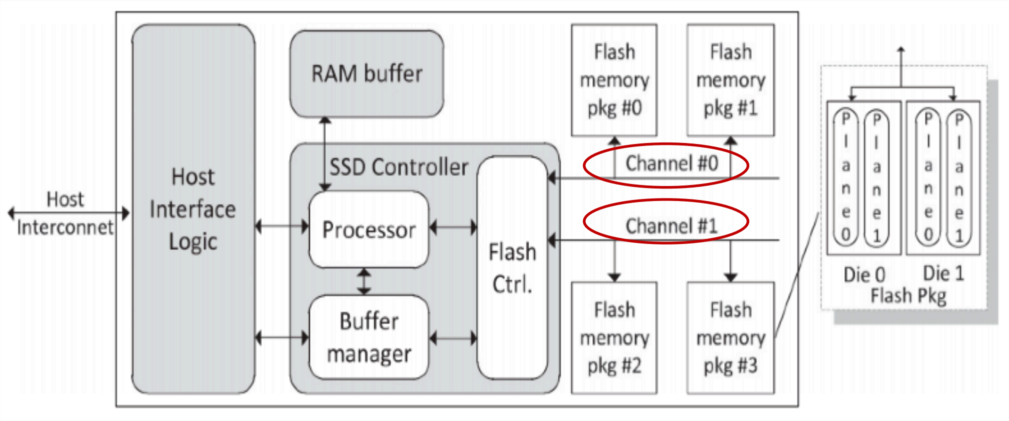
- Direct mapping block
  - A set of mapping entries demoted from Tier 1
- **An FIFO array of blocks**
  - The most recent version is always in the latest position
- **Parallelized Batch Search**
  - Parallel I/Os to load multiple mapping blocks into memory
  - Scan and find the most recent version of the data in one I/O time



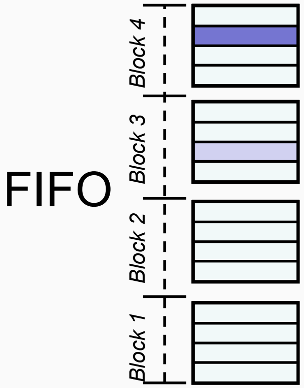
Chen et al., "Internal Parallelism of Flash-based Solid State Drives", *ACM Transactions on Storage*, 12:3, May 2016

Tier 2: Direct Indexing in Flash

- Direct mapping block
  - A set of mapping entries demoted from Tier 1
- **An FIFO array of blocks**
  - The most recent version is always in the latest position
- **Parallelized Batch Search**
  - Parallel I/Os to load multiple mapping blocks into memory
  - Scan and find the most recent version of the data in one I/O time

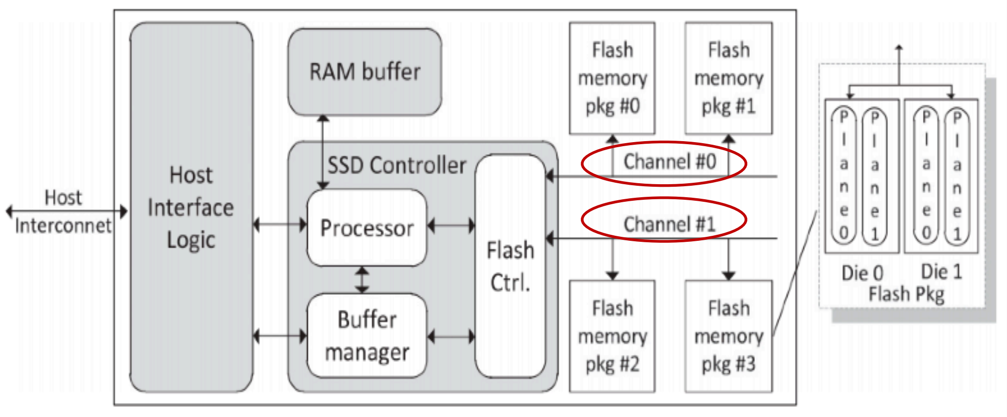


Chen et al., “Internal Parallelism of Flash-based Solid State Drives”, *ACM Transactions on Storage*, 12:3, May 2016

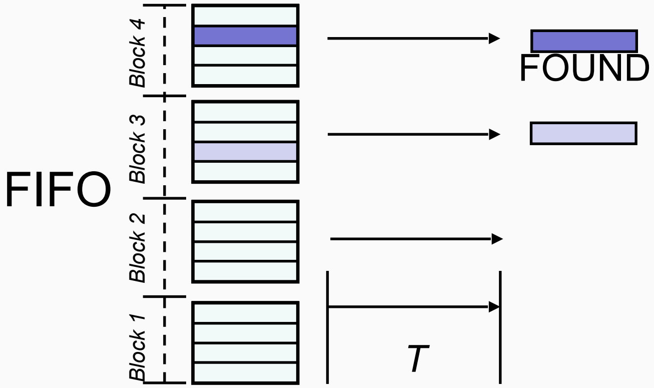


Tier 2: Direct Indexing in Flash

- Direct mapping block
  - A set of mapping entries demoted from Tier 1
- **An FIFO array of blocks**
  - The most recent version is always in the latest position
- **Parallelized Batch Search**
  - Parallel I/Os to load multiple mapping blocks into memory
  - Scan and find the most recent version of the data in one I/O time

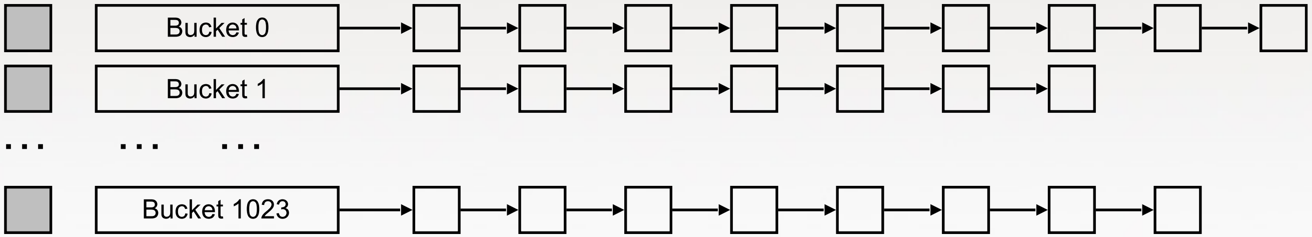


Chen et al., "Internal Parallelism of Flash-based Solid State Drives", *ACM Transactions on Storage*, 12:3, May 2016

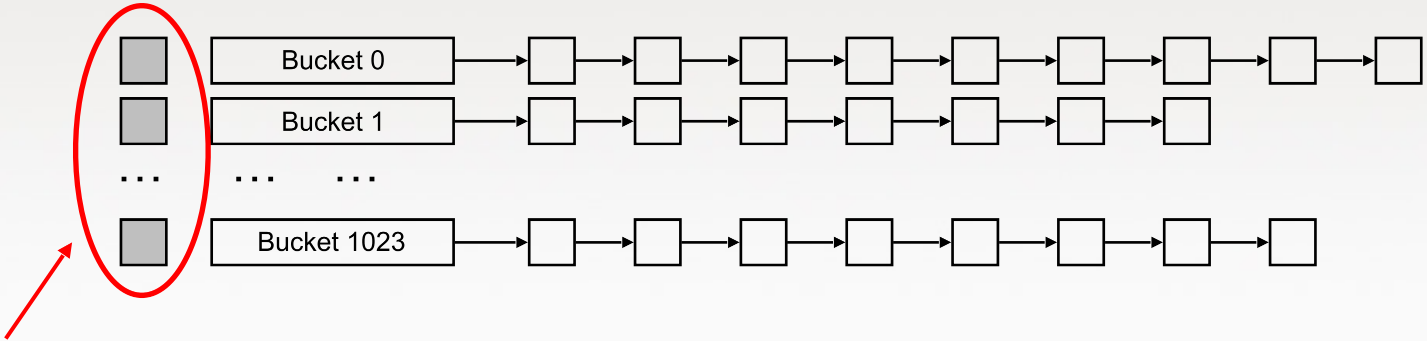


Parallel Search: 1x *T*

Tier 3: Hash Table List Designs

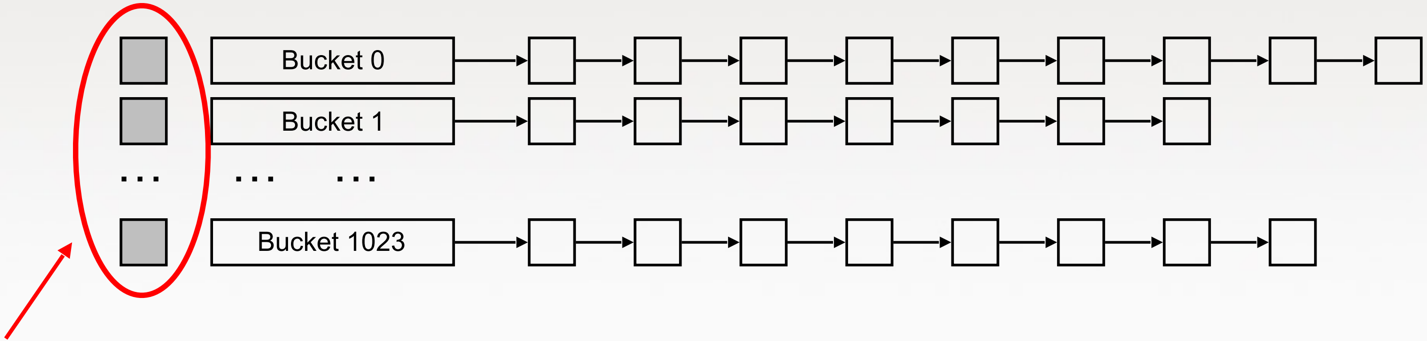


Tier 3: Hash Table List Designs



Memory buffers

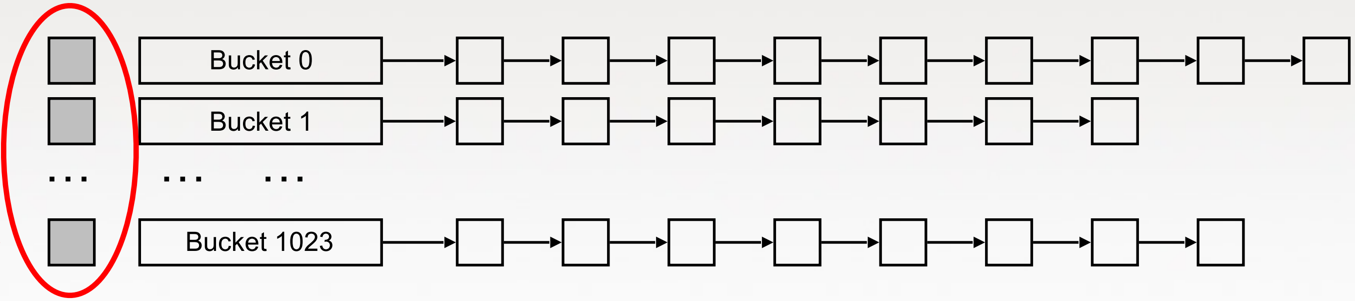
Tier 3: Hash Table List Designs



Memory buffers

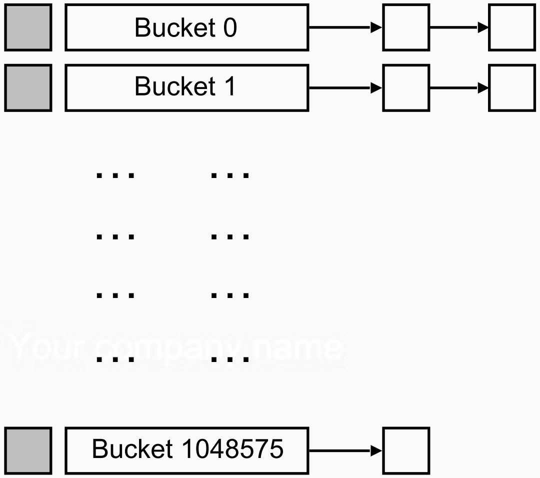
- “Narrow” hash table
  - **Long list** to walk through
  - Need **less memory** buffers (e.g., 128MB)

Tier 3: Hash Table List Designs

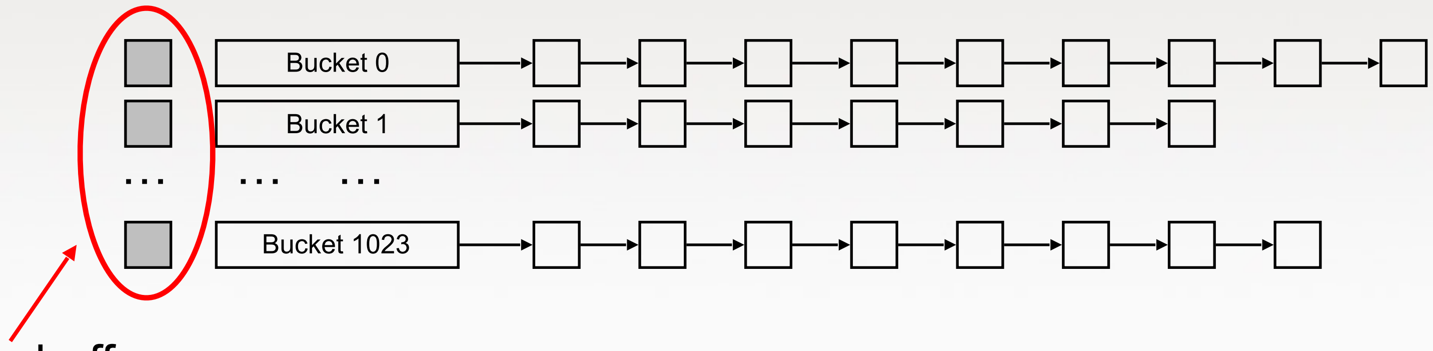


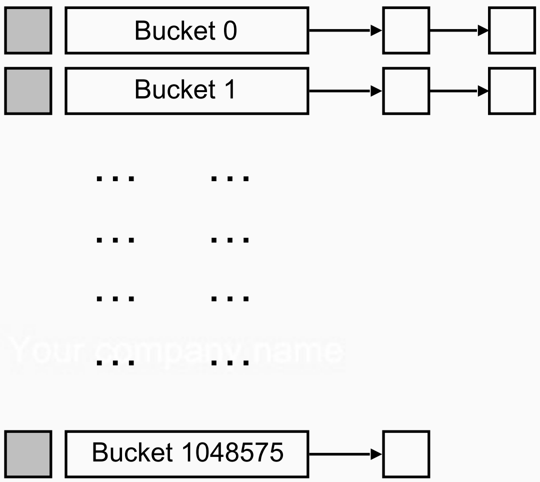
Memory buffers

- “Narrow” hash table
  - **Long list** to walk through
  - Need **less memory** buffers (e.g., 128MB)



Tier 3: Hash Table List Designs

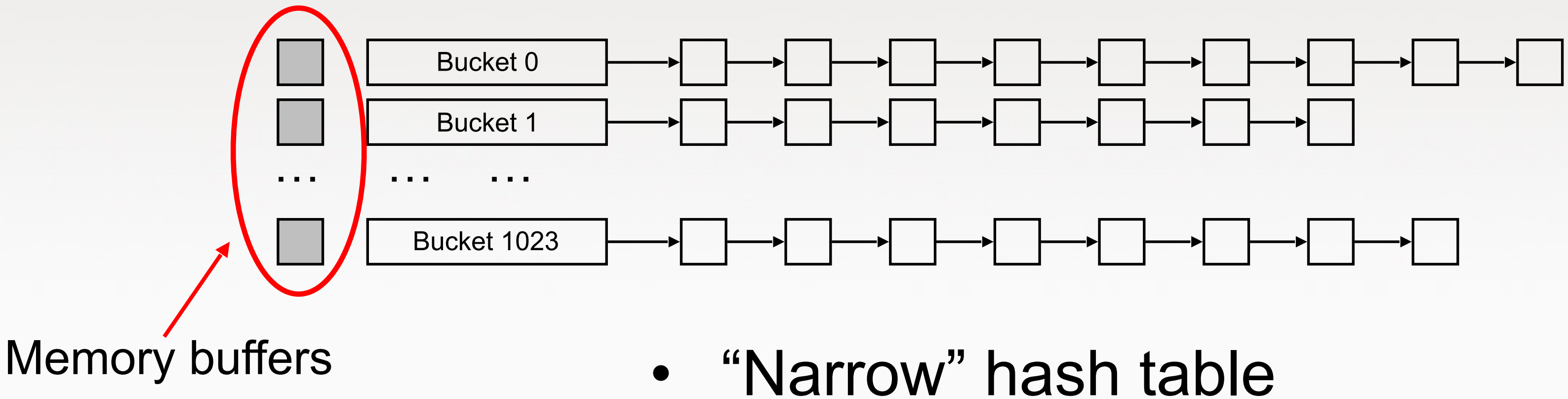


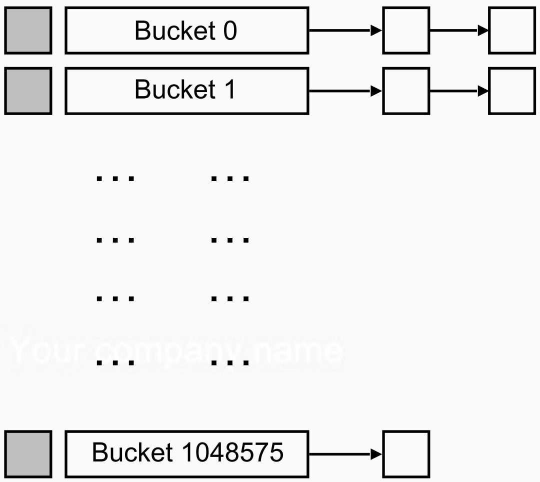


- “Narrow” hash table
  - **Long list** to walk through
  - Need **less memory** buffers (e.g., 128MB)

Memory efficiency v.s. I/O efficiency

Tier 3: Hash Table List Designs

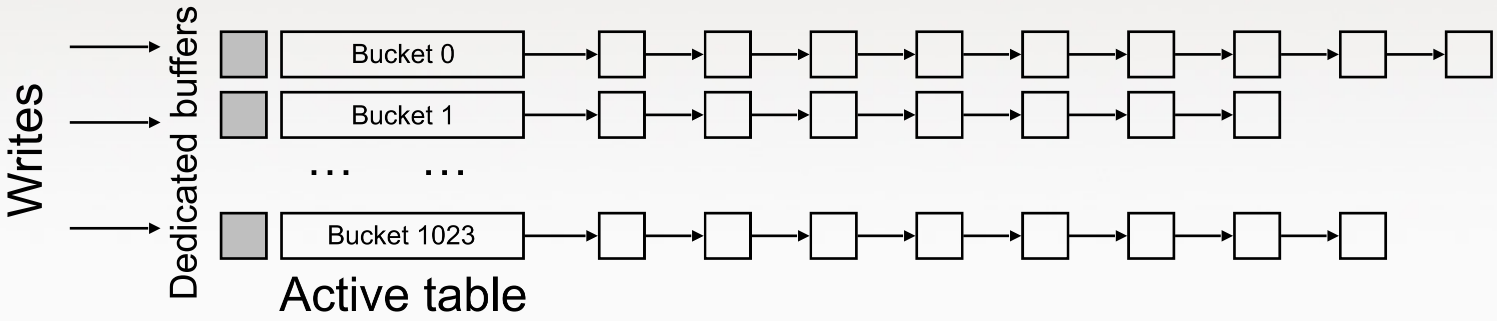


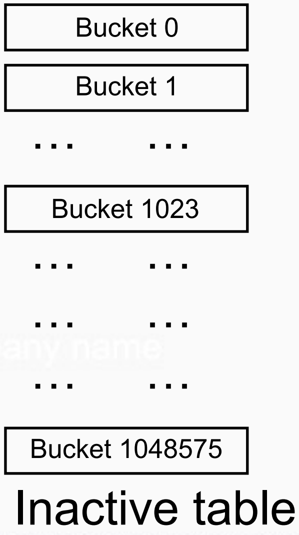


- “Narrow” hash table
  - **Long list** to walk through
  - Need **less memory** buffers (e.g., 128MB)
- “Wide” hash table
  - **Short list** to walk through
  - Need **more memory** buffers (e.g., 128GB)

Memory efficiency v.s. I/O efficiency

Tier 3: Dual-mode Hash Table

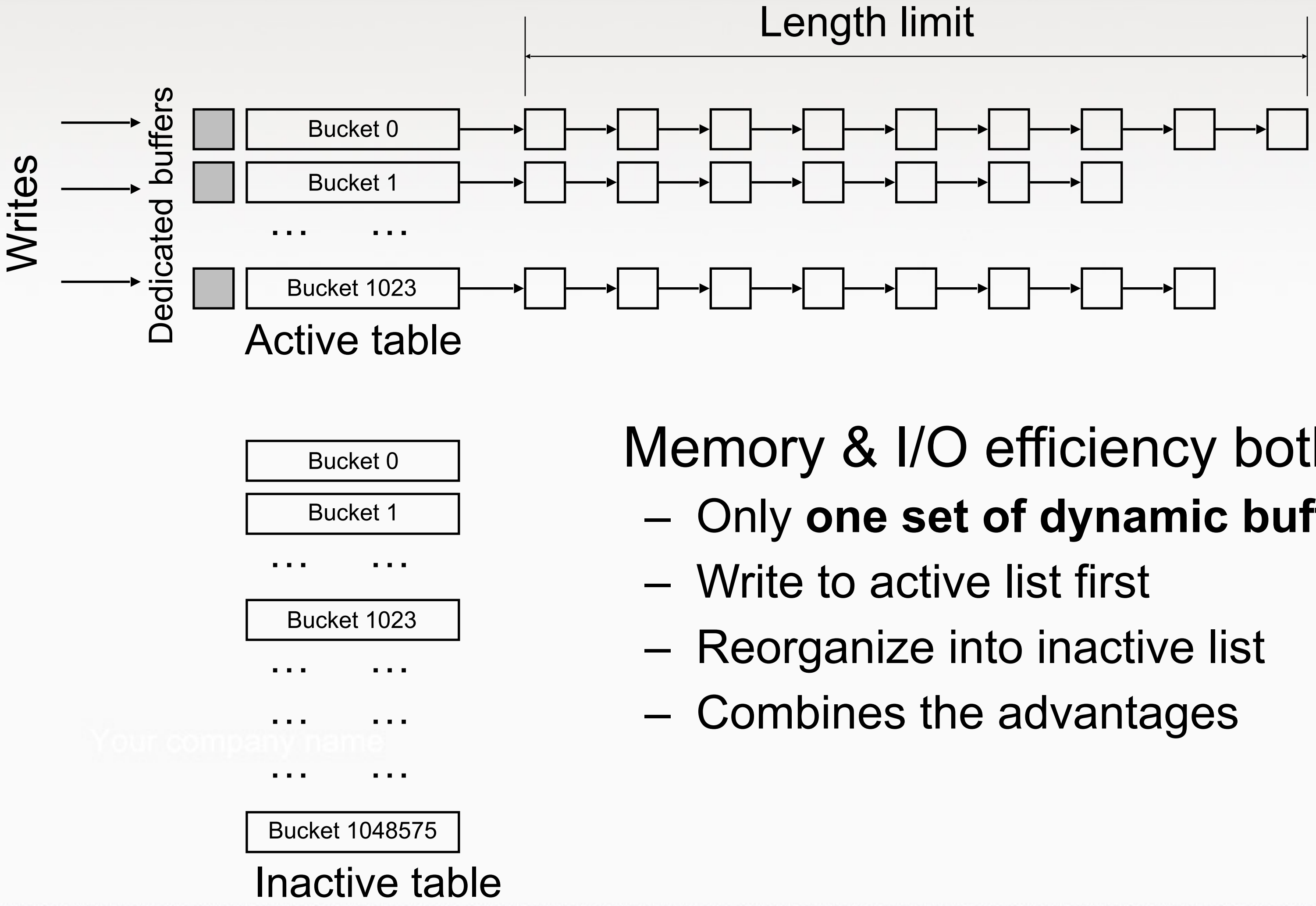




Memory & I/O efficiency both achieved

- Only **one set of dynamic buffers**
- Write to active list first
- Reorganize into inactive list
- Combines the advantages

Tier 3: Dual-mode Hash Table



Length limit

Writes

Dedicated buffers

Bucket 0

Bucket 1

… …

Bucket 1023

Active table

Bucket 0

Bucket 1

… …

Bucket 1023

… …

… …

… …

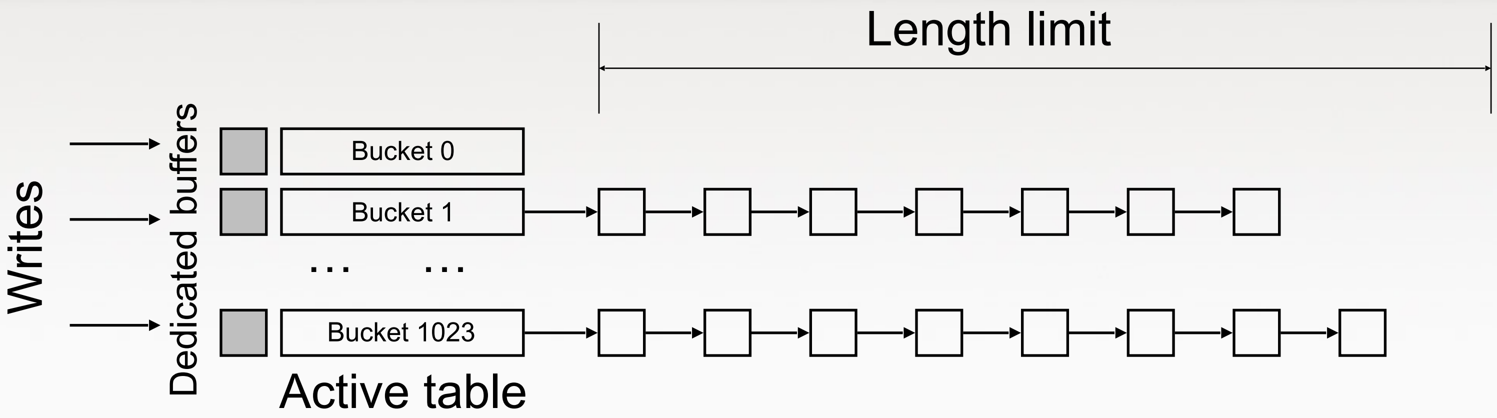
Bucket 1048575

Inactive table

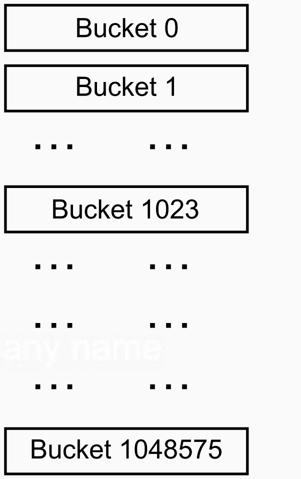
Memory & I/O efficiency both achieved

- Only **one set of dynamic buffers**
- Write to active list first
- Reorganize into inactive list
- Combines the advantages

Tier 3: Dual-mode Hash Table



Active table

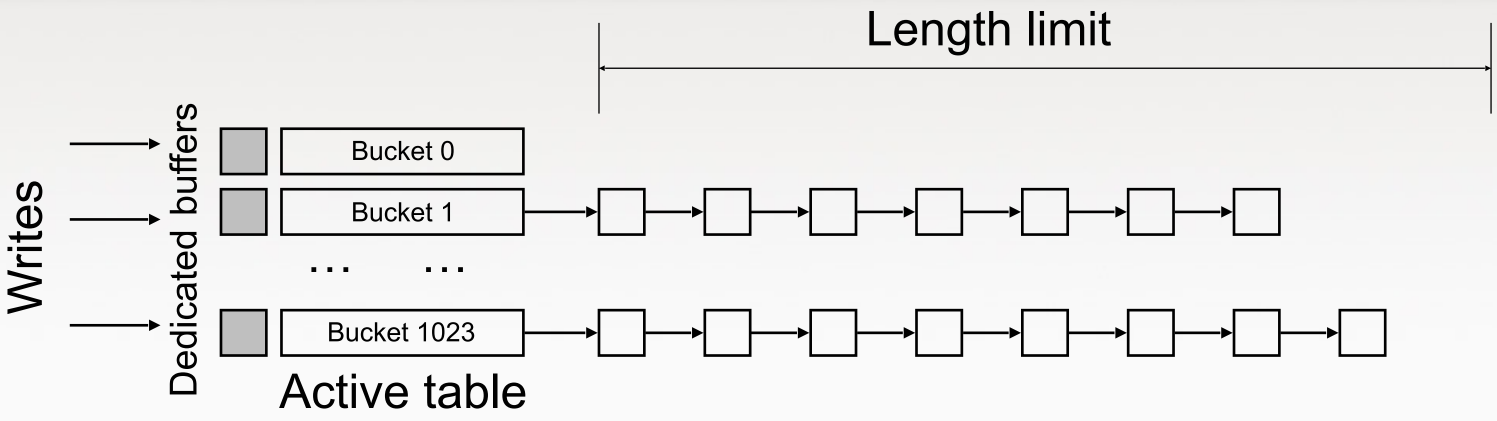


Inactive table

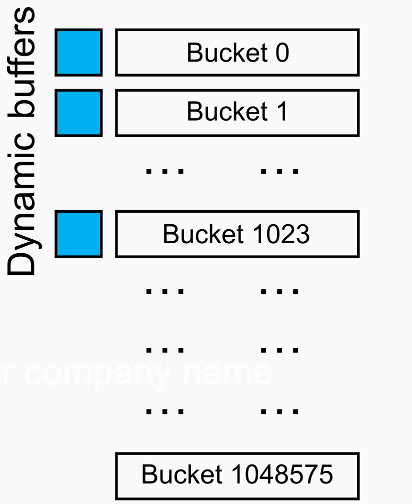
Memory & I/O efficiency both achieved

- Only **one set of dynamic buffers**
- Write to active list first
- Reorganize into inactive list
- Combines the advantages

Tier 3: Dual-mode Hash Table



Active table

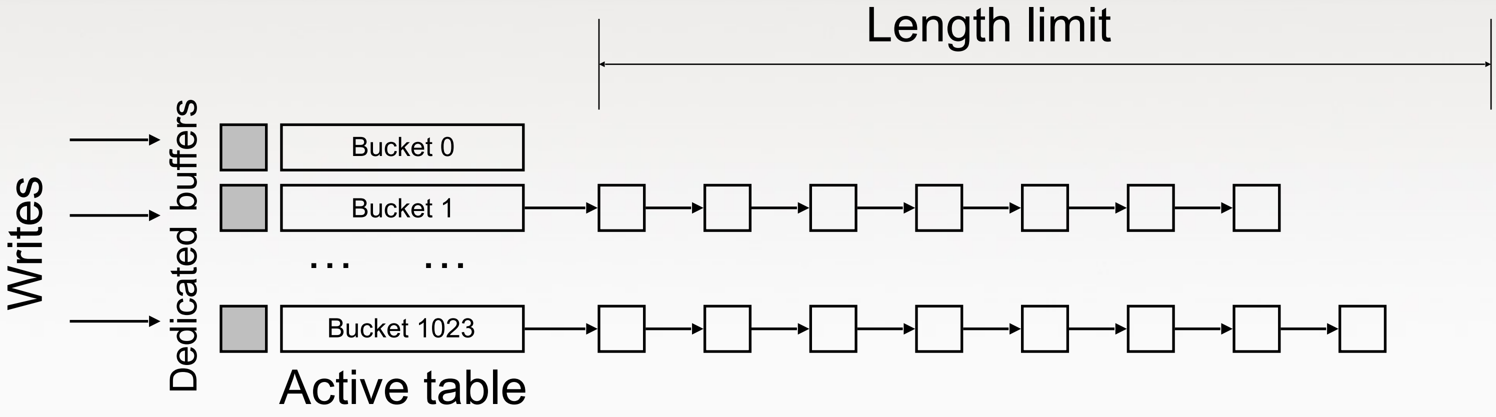


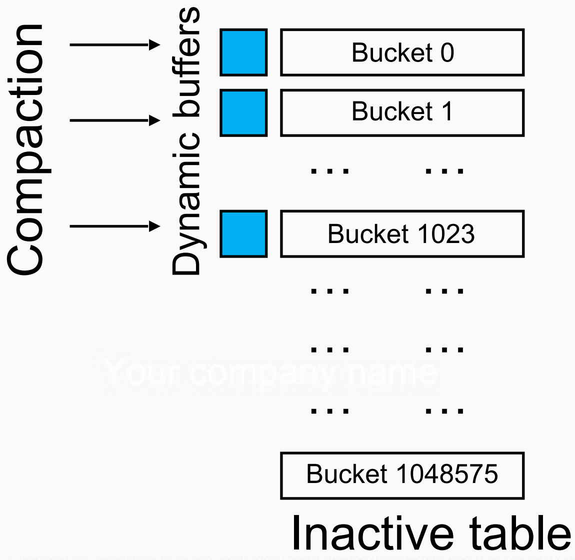
Inactive table

Memory & I/O efficiency both achieved

- Only **one set of dynamic buffers**
- Write to active list first
- Reorganize into inactive list
- Combines the advantages

Tier 3: Dual-mode Hash Table

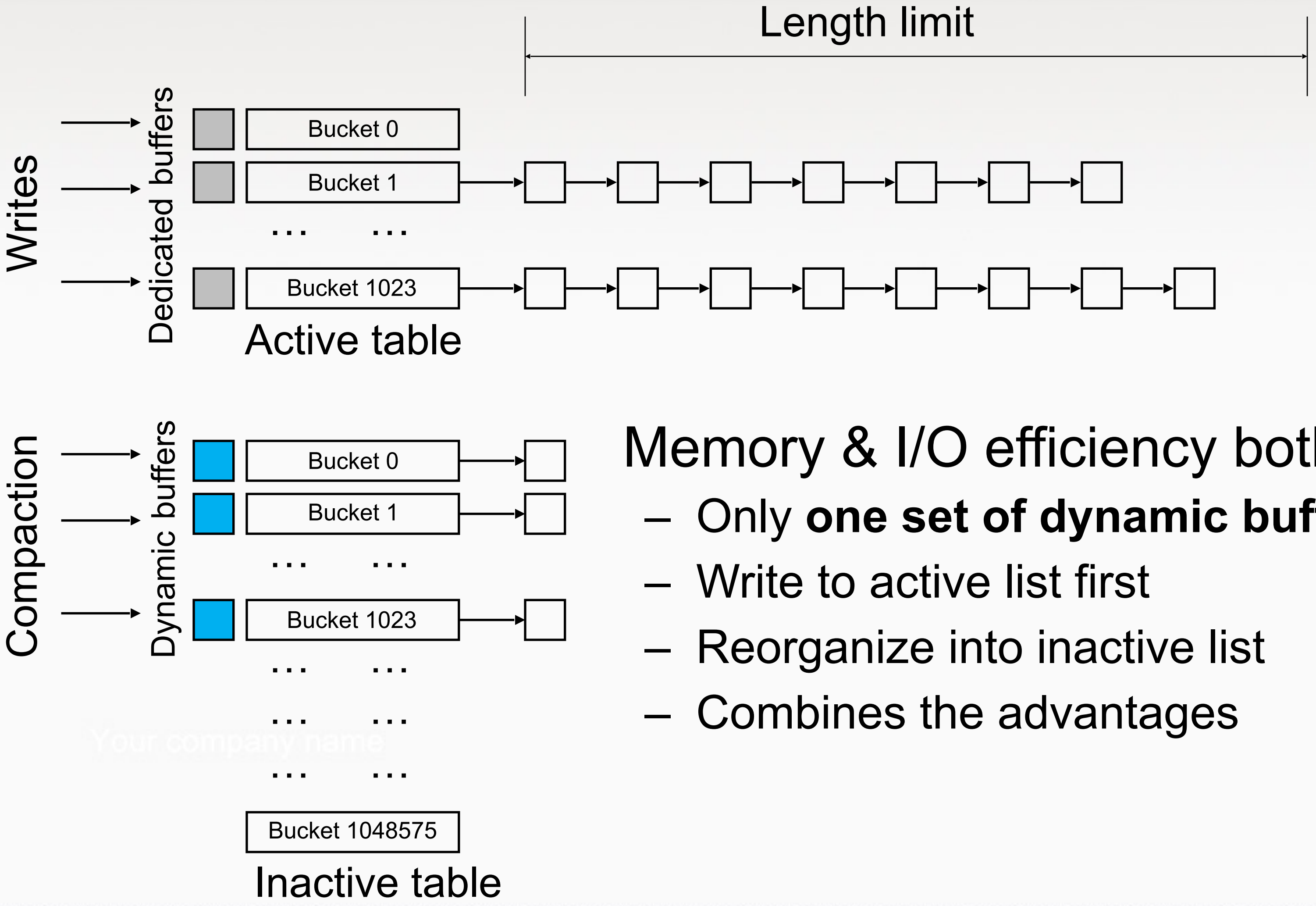




Memory & I/O efficiency both achieved

- Only **one set of dynamic buffers**
- Write to active list first
- Reorganize into inactive list
- Combines the advantages

Tier 3: Dual-mode Hash Table



Memory & I/O efficiency both achieved

- Only **one set of dynamic buffers**
- Write to active list first
- Reorganize into inactive list
- Combines the advantages

Outline

- Cascade mapping design
- Optimizations
- Evaluation results
- Conclusions

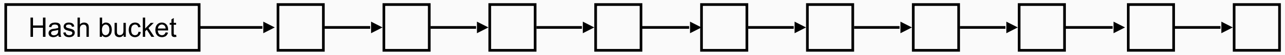
Optimization Techniques

- Partition the hash space to create multiple demotion I/O streams
- Adopt a memory-efficient CLOCK-based demotion policy
- Organize an array of direct mapping blocks in the FIFO order
- Parallel batch search to quickly complete a one-to-one scan
- Use a dual-mode hash table for both memory and I/O efficiency
- A jump list by using Bloom filters to skip impossible blocks
- Make the FIFO-based eviction policy locality aware
- Use slab sequence counter to realize zero-I/O demapping
- Leverage the FIFO nature of slabs for efficient crash recovery

Optimization Techniques

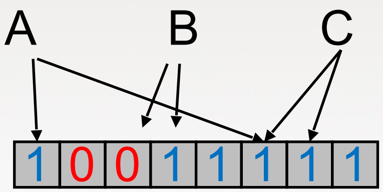
- Partition the hash space to create multiple demotion I/O streams
- Adopt a memory-efficient CLOCK-based demotion policy
- Organize an array of direct mapping blocks in the FIFO order
- Parallel batch search to quickly complete a one-to-one scan
- Use a dual-mode hash table for both memory and I/O efficiency
- A jump list by using Bloom filters to skip impossible blocks
- Make the FIFO-based eviction policy locality aware
- Use slab sequence counter to realize zero-I/O demapping
- Leverage the FIFO nature of slabs for efficient crash recovery

Optimization: Jump List



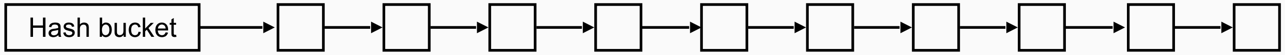
One single long list

Optimization: Jump List



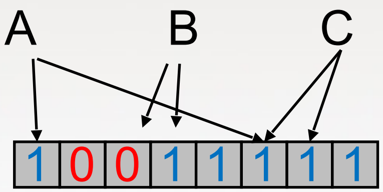
Bloom filter: to test whether an element is in a set

- A query returns either *possibly in set* or *definitely not in set*
- False positive is possible, but false negative is impossible
- Elements can be added to the set, but not removed



One single long list

Optimization: Jump List

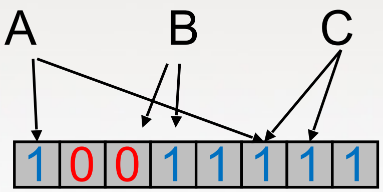


Bloom filter: to test whether an element is in a set

- A query returns either *possibly in set* or *definitely not in set*
- False positive is possible, but false negative is impossible
- Elements can be added to the set, but not removed

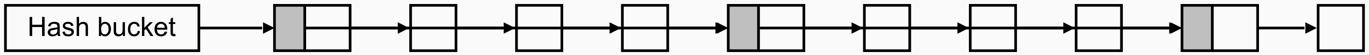
One single long list

Optimization: Jump List



Bloom filter: to test whether an element is in a set

- A query returns either *possibly in set* or *definitely not in set*
- False positive is possible, but false negative is impossible
- Elements can be added to the set, but not removed

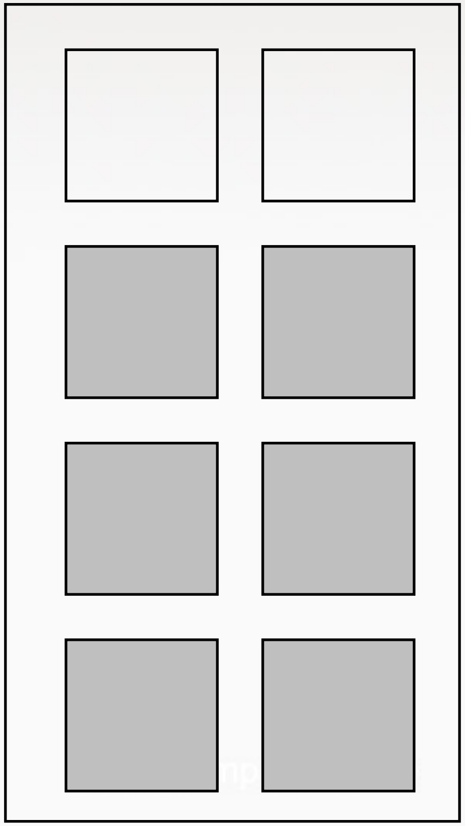


One single long list → Several short lists connected by hops

Bloom filters are used to avoid unnecessary tier-3 I/Os

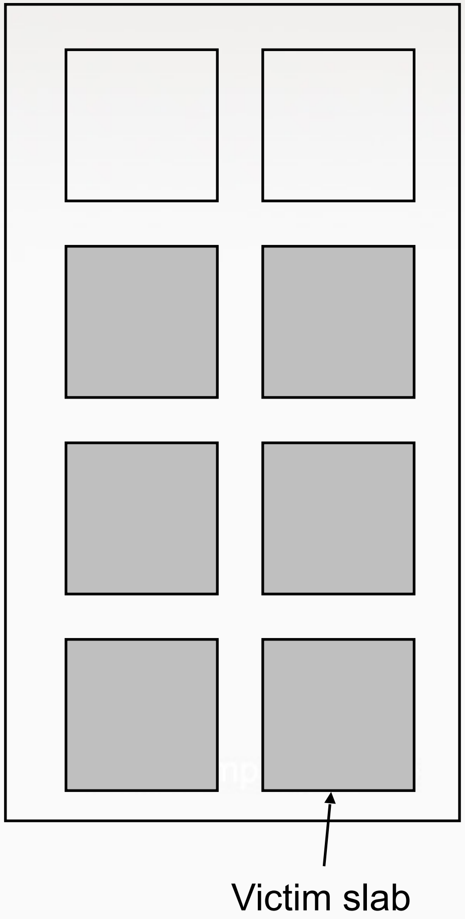
- Bloom filters are stored in flash together with regular mapping blocks
- Indicate whether a mapping can be found **within next several blocks**
- If returns negative, **jump to the next Bloom filter block**

Optimization: Garbage Collection



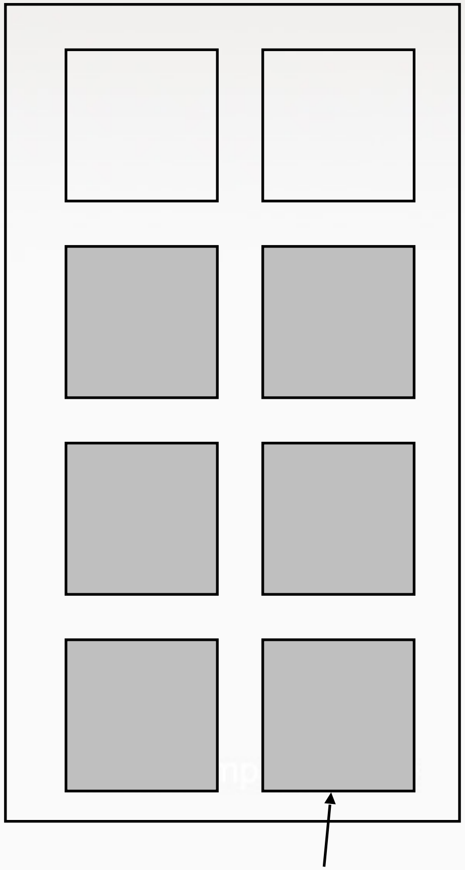
- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes

Optimization: Garbage Collection



- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes

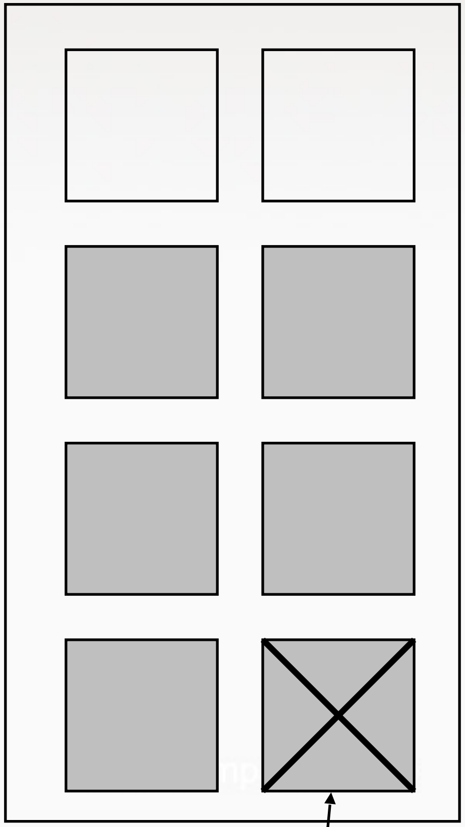
Optimization: Garbage Collection



Victim slab

- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data

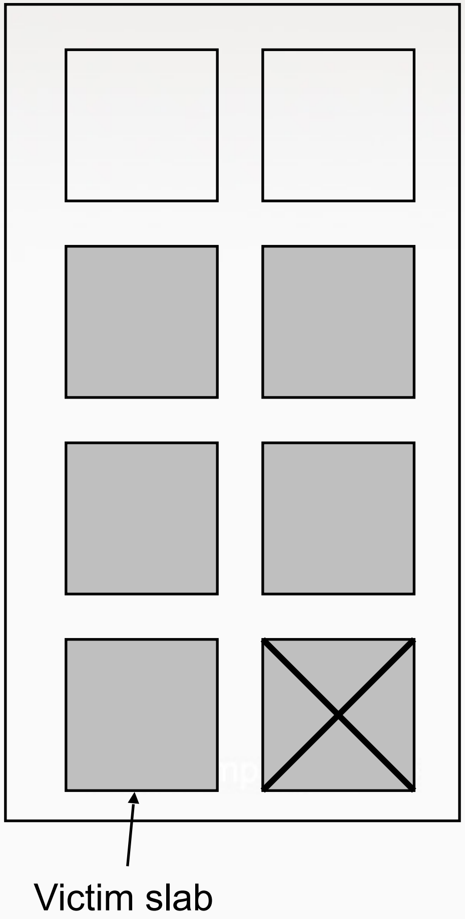
Optimization: Garbage Collection



Victim slab

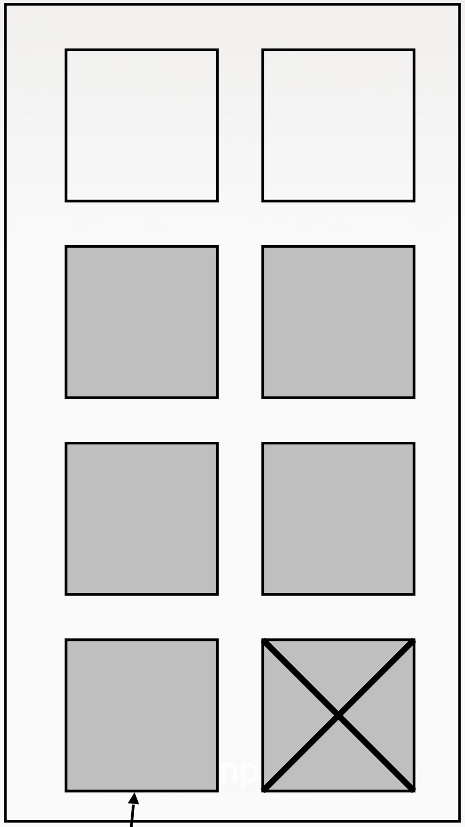
- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data

Optimization: Garbage Collection



- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data

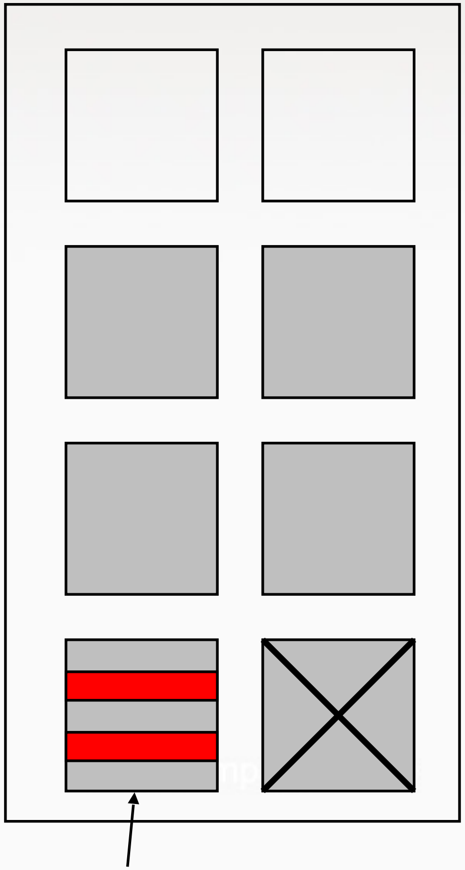
Optimization: Garbage Collection



Victim slab

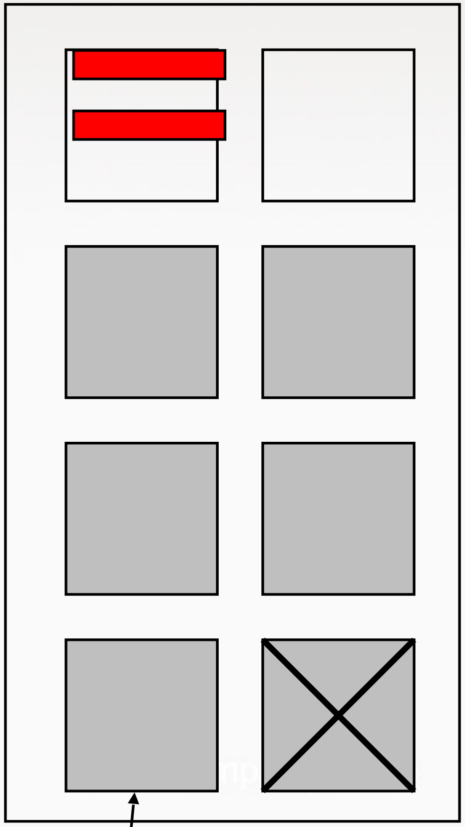
- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data
- Our solution: Keep hot data in cache
  - If a k-v item’s mapping is in tier 1, indicating it is hot data
  - Rewrite hot data to a new slab, then erase victim slab

Optimization: Garbage Collection



- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data
- Our solution: Keep hot data in cache
  - If a k-v item’s mapping is in tier 1, indicating it is hot data
  - Rewrite hot data to a new slab, then erase victim slab

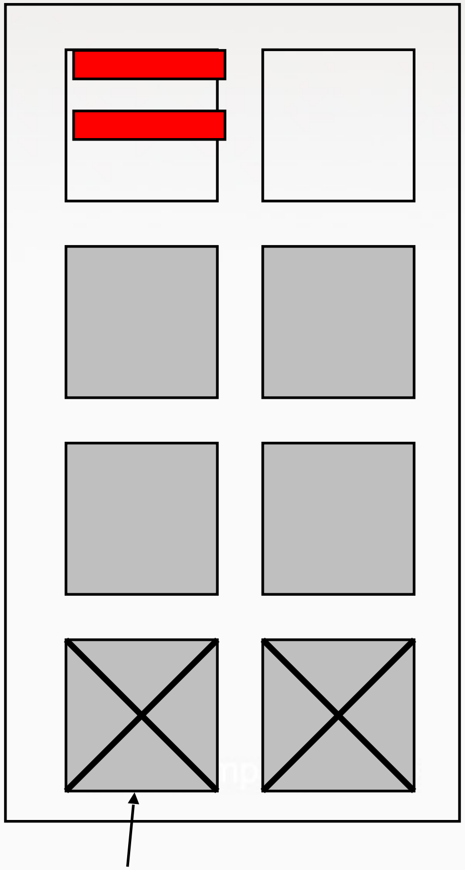
Optimization: Garbage Collection



Victim slab

- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data
- Our solution: Keep hot data in cache
  - If a k-v item’s mapping is in tier 1, indicating it is hot data
  - Rewrite hot data to a new slab, then erase victim slab

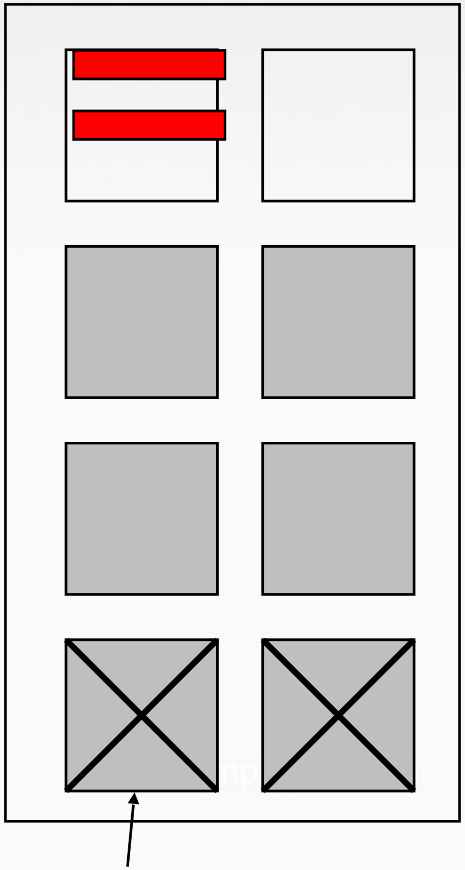
Optimization: Garbage Collection



Victim slab

- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data
- Our solution: Keep hot data in cache
  - If a k-v item’s mapping is in tier 1, indicating it is hot data
  - Rewrite hot data to a new slab, then erase victim slab

Optimization: Garbage Collection



- GC is a must-have for key-value systems
  - To reclaim flash space
  - To organize large sequential writes
- Traditional: Free up space immediately
  - Erase entire victim slab based on FIFO order
  - Reclaim space quickly, but may delete hot data
- Our solution: Keep hot data in cache
  - If a k-v item’s mapping is in tier 1, indicating it is hot data
  - Rewrite hot data to a new slab, then erase victim slab
- Adaptive two-phase GC
  - If free flash space is too low, perform fast space reclaim
  - Keep hot data when system under moderate pressure

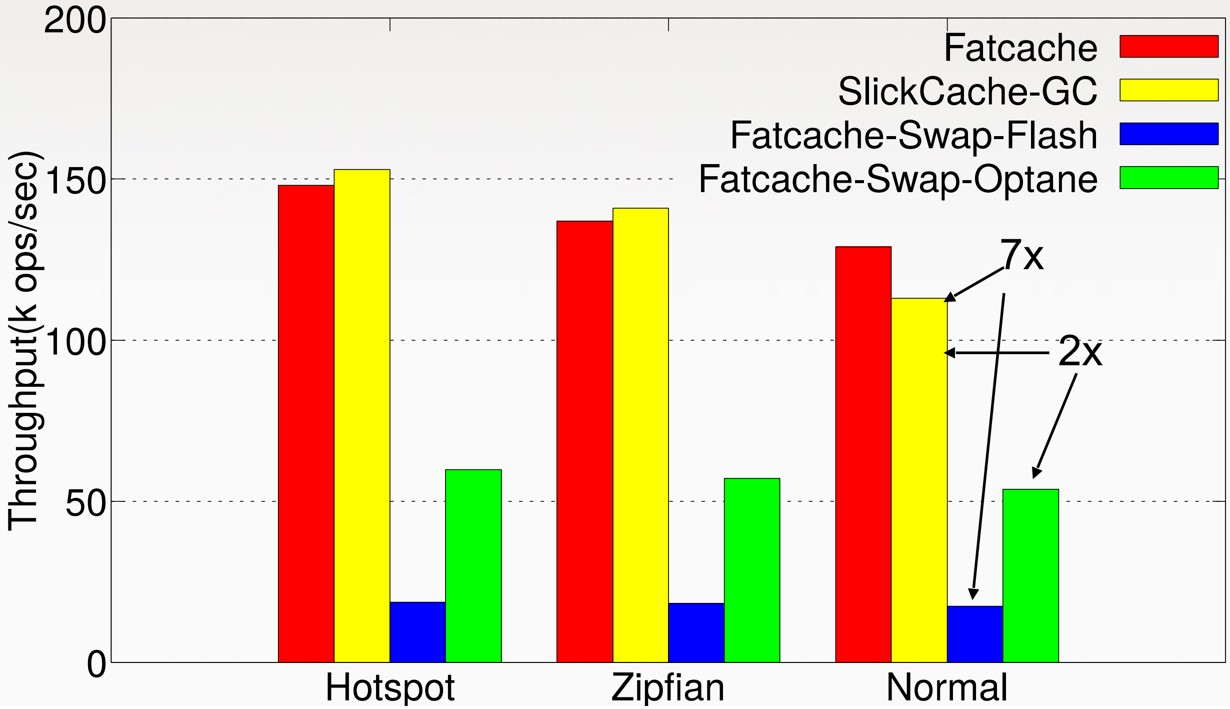
Outline

- Cascade mapping design
- Optimizations
- Evaluation results
- Conclusions

Experimental Setup

- Implementation
  - SlickCache: 3,800 lines of C code added to Twitter’s Fatcache
- Hardware environment
  - Lenovo ThinkServers: 4-core Intel Xeon 3.4 GHz with 16 GB DRAM
  - 240-GB Intel 730 SSD as cache device
  - 280-GB Intel Optane 900P SSD as swapping device
  - 7,200 RPM Seagate 2-TB HDD as database device
- Software environment
  - Ubuntu 16.04 with Linux kernel 4.12 and Ext4 file system
  - MongoDB 3.4 for backend database
- Workloads
  - Yahoo! Cloud Serving Benchmark (YCSB)
  - Popular distributions: Hotspot, Zipfian, and Normal

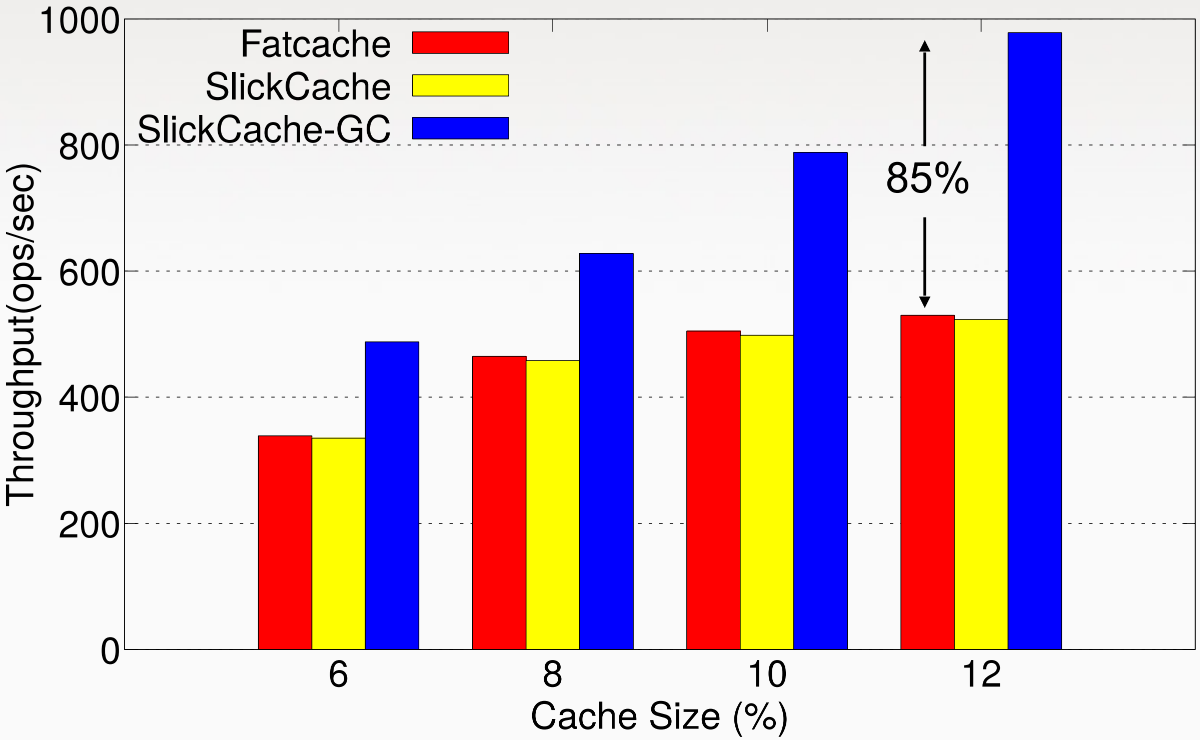
Evaluation Results



Comparison with Fatcache and system swapping

Fatcache-Swap-Flash and Fatcache-Swap-Optane are both configured with **10%** of physical memory, allowed to swap on flash SSD and Optane SSD respectively.

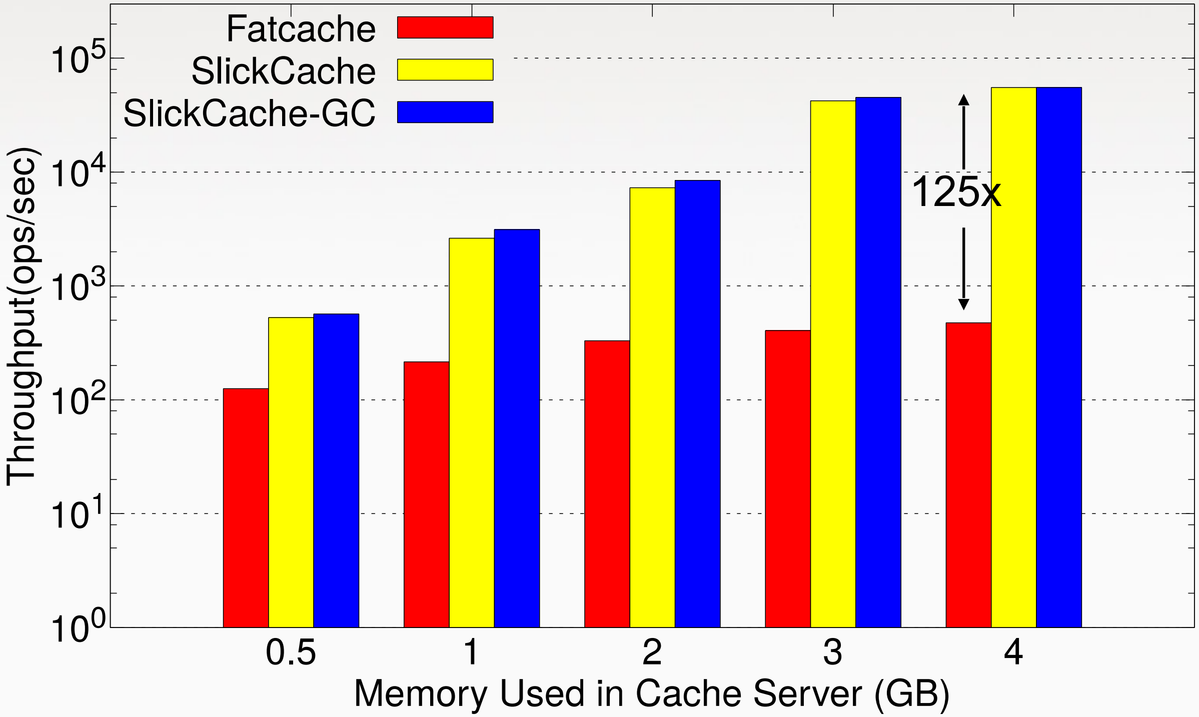
Evaluation Results



Cache effectiveness (Fixed cache size)

SlickCache only uses 10% of the memory used by Fatcache, achieves comparable performance.
SlickCache-GC increases throughput by up to **85%** due to the optimized GC policy.

Evaluation Results



Cache effectiveness (Fixed memory size)

SlickCache is able to index a **10 times** larger flash cache with the same amount of memory, which in turn increases the hit ratio by up to **8.2 times** and the throughput by up to **125 times**.

Conclusions

Cascade Mapping for flash-based key-value caching

- A hierarchical mapping structure for flash-based key-value cache
- A set of optimizations to improve performance
- Use less memory while performs better than current design

Thanks!

And Questions?