All-Flash Array Key-Value Cache for Large Objects

Jinhyung Koo, Jinwook Bae, Minjeong Yuk, Sungkyun Oh, Jungwoo Kim

Jung-soo Park[1], Eunji Lee[2], Bryan S. Kim[3], and Sungjin Lee

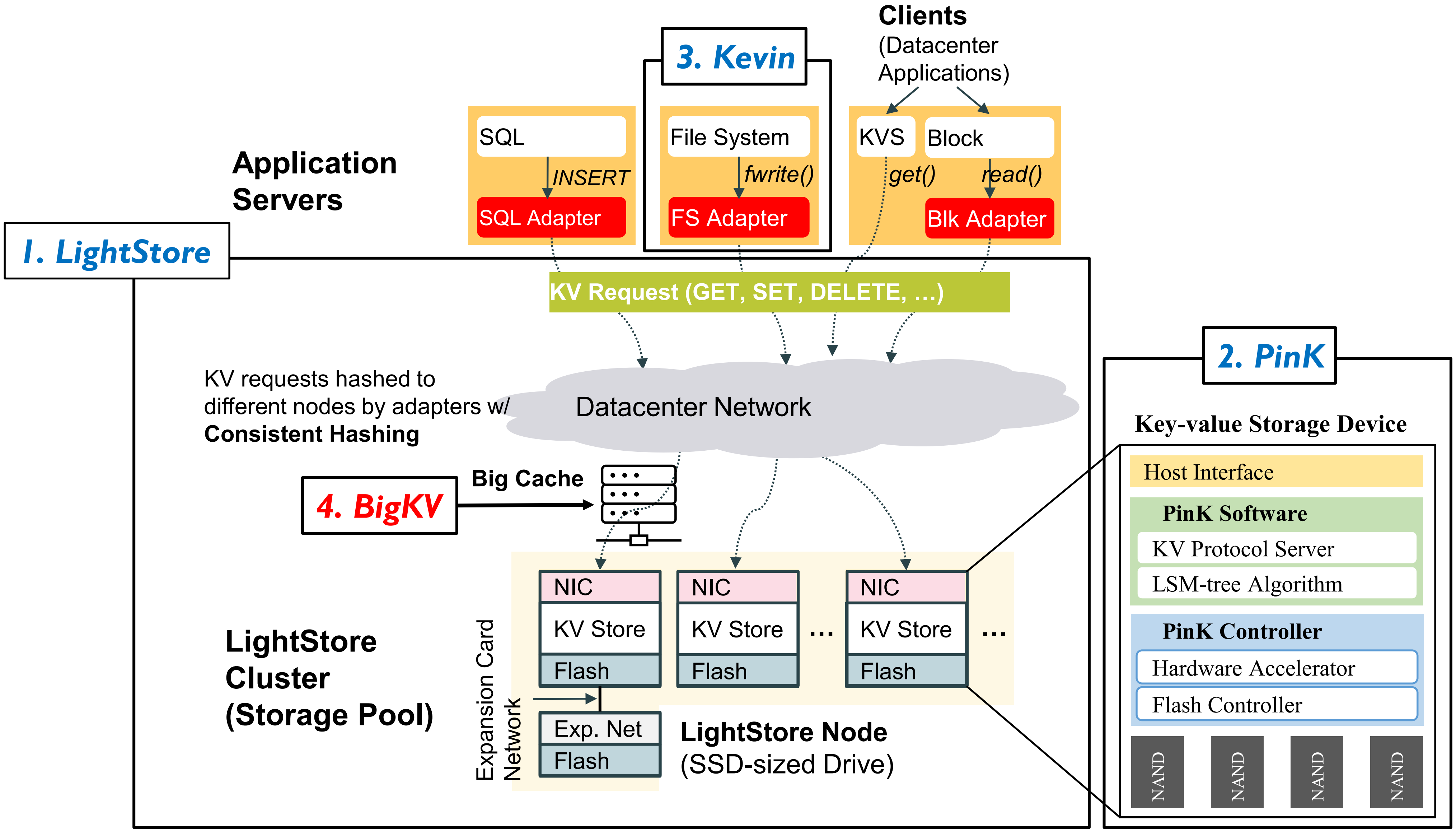
DGIST [1]WineSoft [2]Soongsil University [3]Syracuse University

Operating System Support for Next Generation Large Scale NVRAM (NVRAMOS'23)

(Presented at 18th ACM European Conference on Computer Systems)

2023. 10. 20

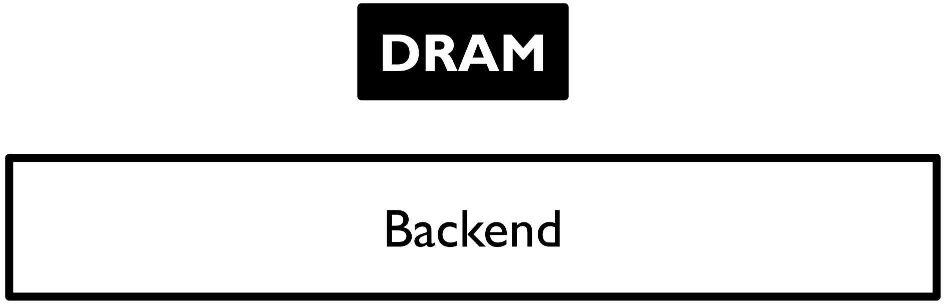
Today's Presentation

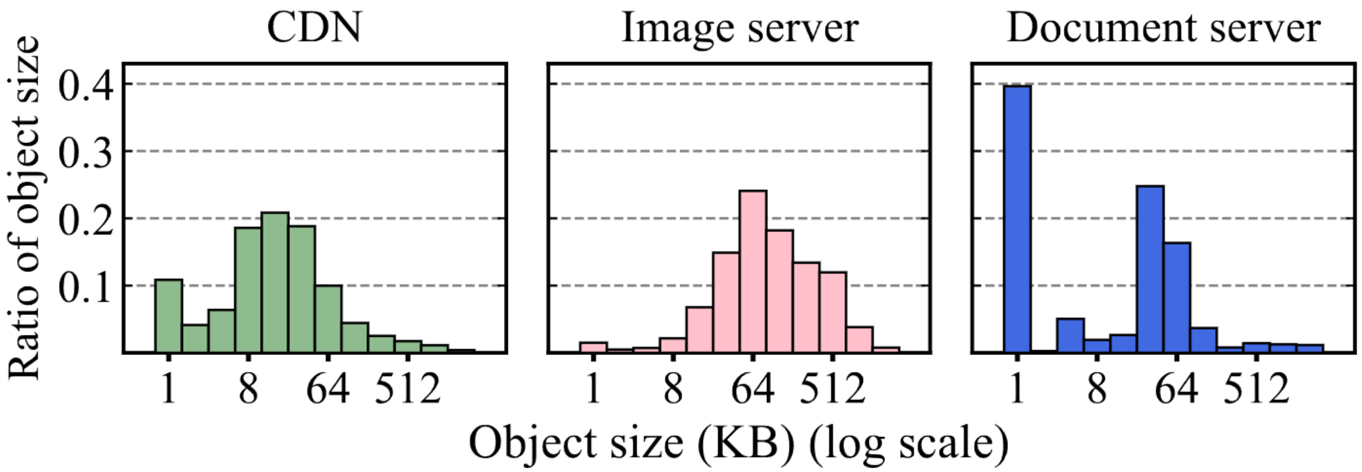


Limitations of Using DRAM as a Key-value Cache

- KV (Key-Value) cache
  - Reduce user-perceived latency and backend loads

- DRAM as a KV cache
  - Fit for caching small objects, but too costly for large objects!





Need for Caching Large Objects

Distributed caching for large objects

Asked 10 years, 8 months ago   Modified 9 years, 3 months ago   Viewed 6k times

11

I want to share a very large object e.g. in orders of megabytes or even several gigabytes, between a set of machines. The object will be written once but may be read many times. Maybe a naive approach is to use a ceneteralized storage like [redis](). However, it may become a single point of failure and too many requests may make a DOS attack on redis. Then, a distributed solution is much more promising. But, the main concern is replicating the structure to all machines. If the

1 Answer   Sorted by: Highest score (default)

16

Caching large objects in NoSQL stores is generally not a good idea, because it is expensive in term of memory and network bandwidth. I don't think NoSQL solutions shine when it comes to storing large objects. Redis, memcached, and most other key/value stores are clearly not designed for this.

If you want to store large objects in NoSQL products, you need to cut them in small pieces, and store the pieces as independent objects. This is the approach retained by 10gen for gridfs (which is part of the standard MongoDB distribution):

Redis for caching image files?

Asked 7 years, 4 months ago   Modified 7 years, 4 months ago   Viewed 17k times   Part of AWS Collective

10

I am using Amazon S3 for storing and retrieving images for an image storing website. The trouble is that multiple users have to retrieve same image multiple times.

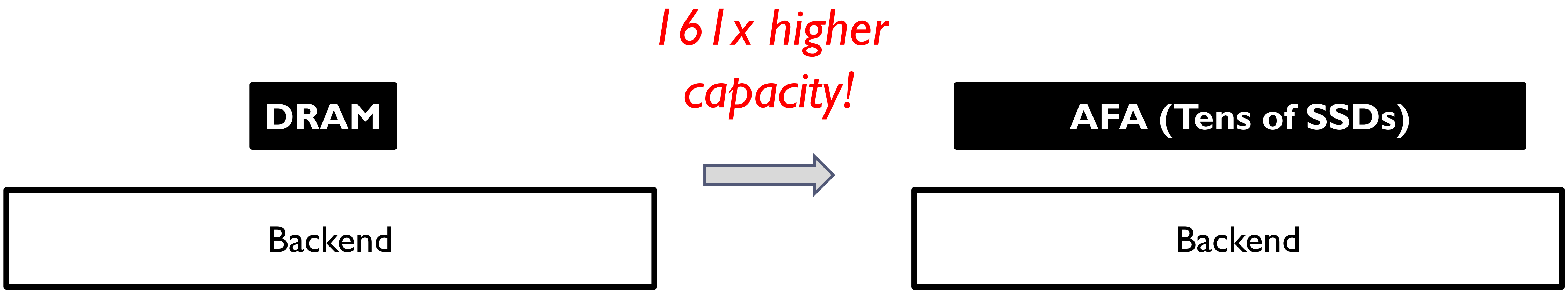
Is it suggested to use Redis or memcached for caching image files by storing them directly onto it.

1

Storing images in Redis seems like a terrible idea since it will quickly fill up the available RAM on the Redis server. Also your statement that "S3 pricing for data transfer is much higher than compared to serving images via Redis" sounds incorrect to me. I think you are missing something there. The standard way to cache images is to use a CDN such as CloudFront,

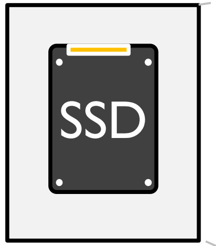
AFA (All-Flash Array) as an Alternative

- Flash-based SSD provides an order of magnitude higher GB/$ than DRAM
- Satisfy the demand for caching large objects at a lower cost
  - 36x cost-effective



Lack of Prior Studies for AFA KV caches

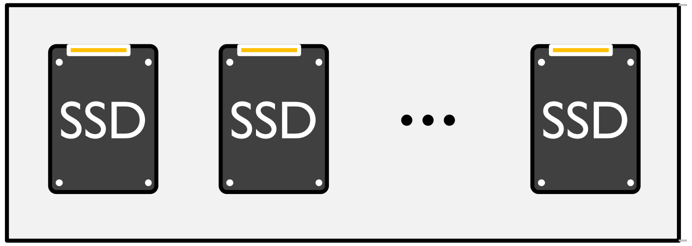
SSD KV cache



- **Indexing**
  - Kangaroo (SOSP '21)
  - FlashShield (NSDI '19)
  - SlickCache (SoCC '18)
- **Cross-layer optimization**
  - uDepot (FAST '19)
  - DIDACache (FAST '17)

  …

AFA KV cache



- **No prior studies**

What is the Difference b/w SSD and AFA KV caches?

Enough resources to manage SSDs

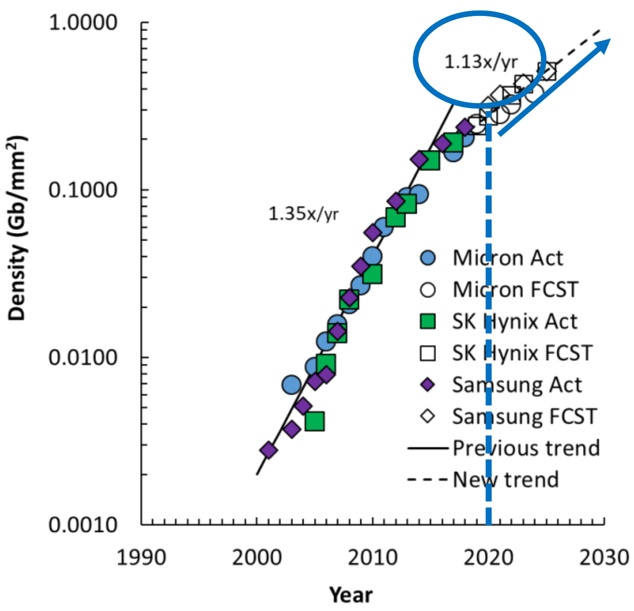
+ High performance
+ Huge capacity
-- Small amount of DRAM
-- Many SSDs to manage

SSD KV cache

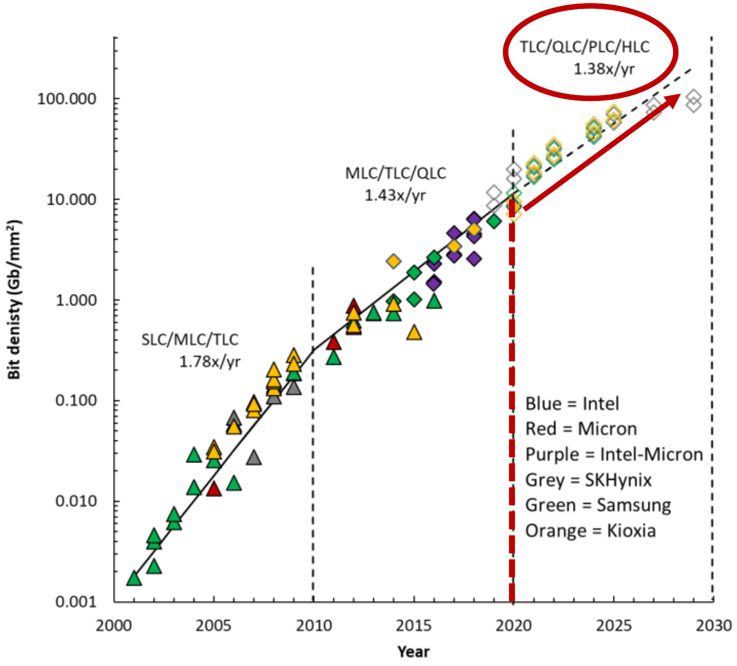
AFA KV cache

What is the Difference b/w SSD and AFA KV caches?

Capacity growth*: 1.38x (SSD) > 1.13x (DRAM)
# of SSD slots > # of DRAM slots



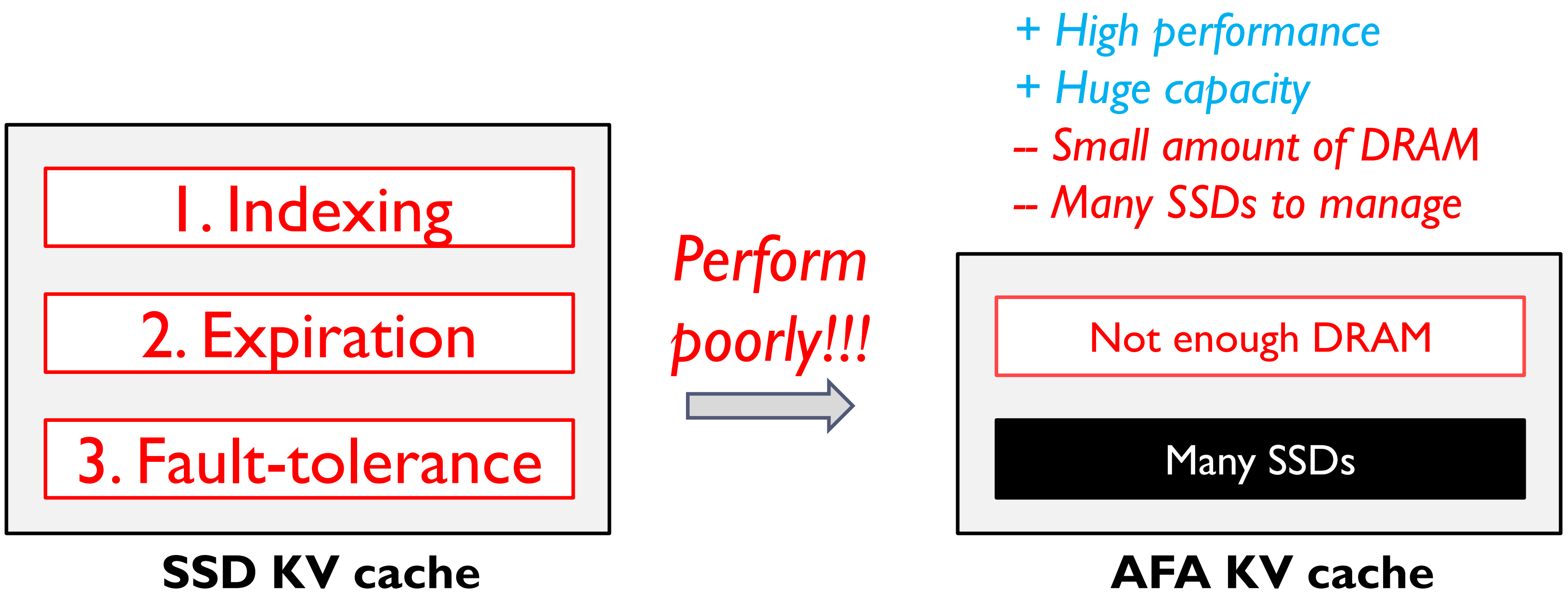
DRAM bit density



NAND Flash bit density

* Technology and Cost Trends at Advanced Nodes (2020)

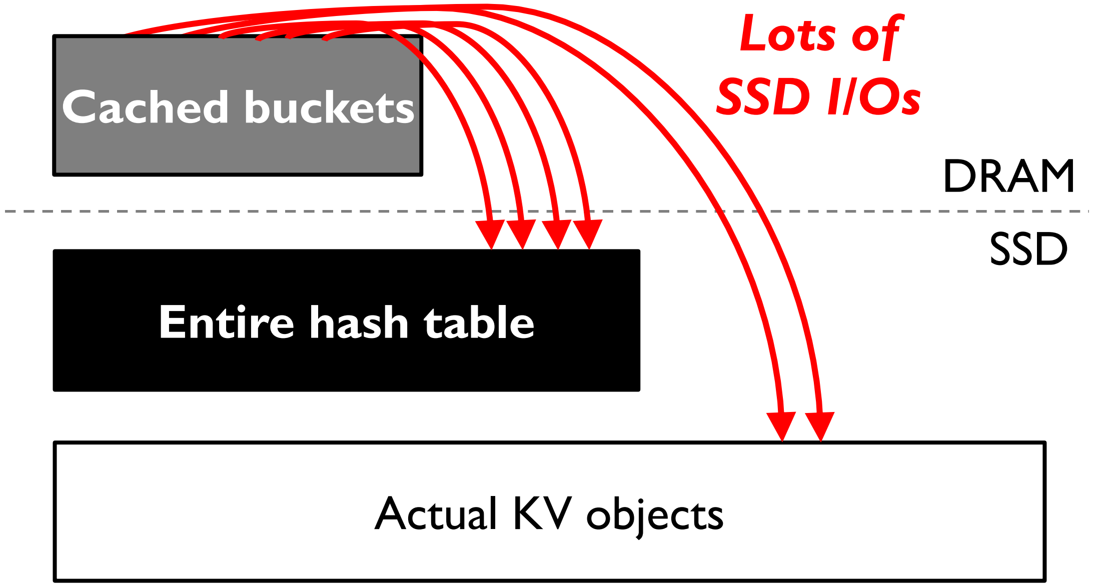
Key Challenges of Existing SSD KV caches

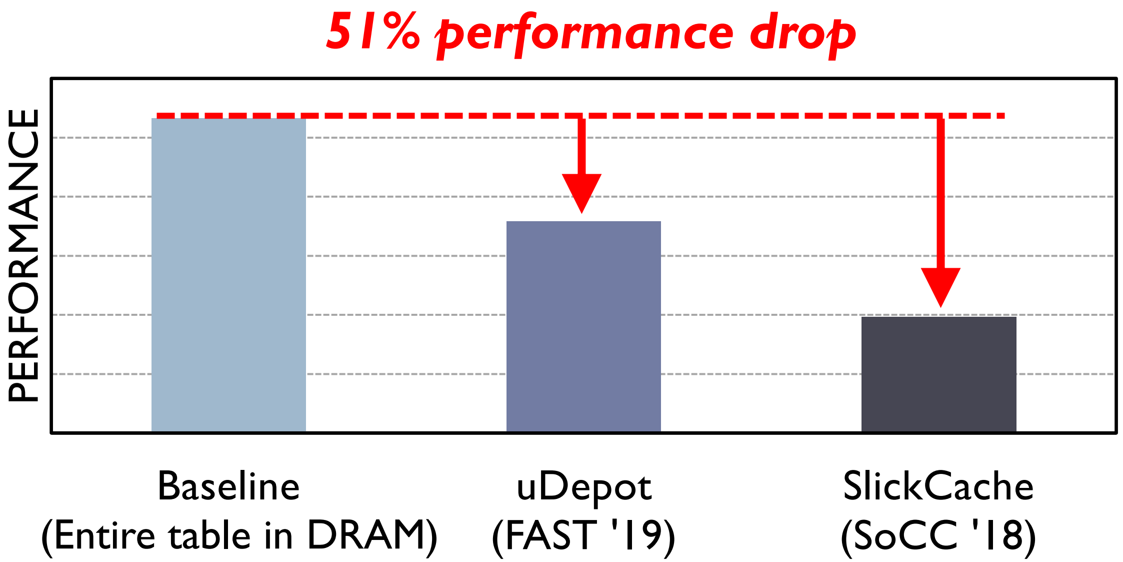


Challenge #1: Performance Drop of Existing Hashing

- The huge hash table does not fit in the AFA's DRAM
  → Must be stored both in DRAM and SSD

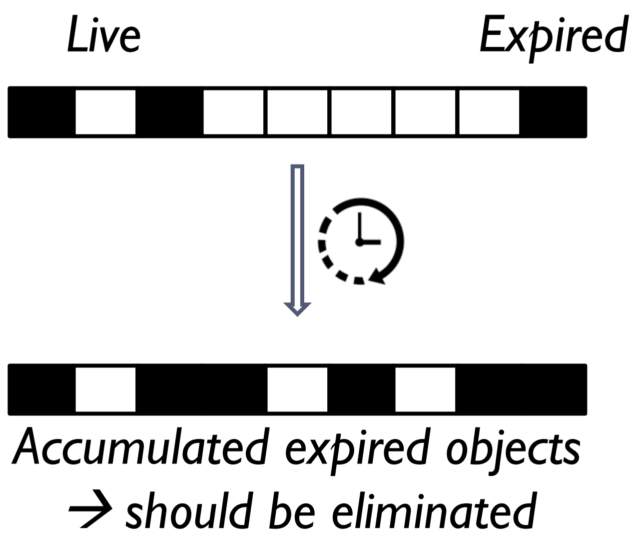
- Accesses to hash buckets and objects in SSD incur significant I/O overhead

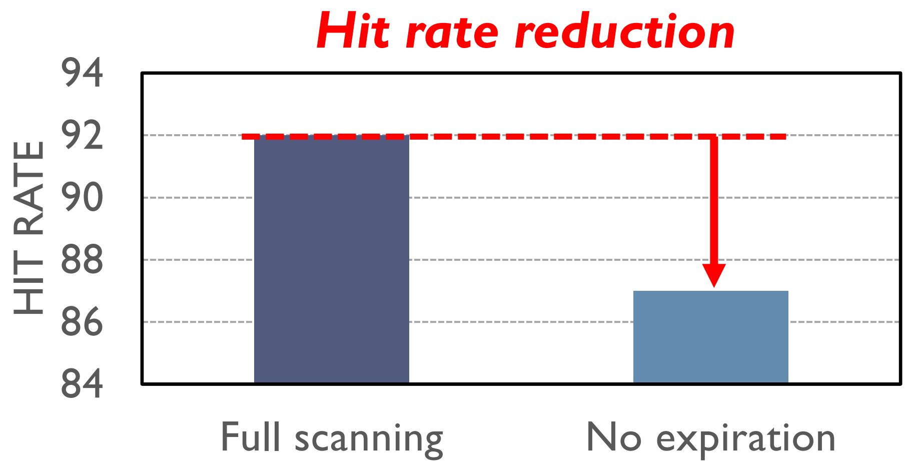




Challenge #2: Capacity & I/O Overhead for Expired Objects

- Expired objects accumulate in the AFA space, resulting in the hit rate reduction
- Full-scanning for removal incurs a huge amount of I/Os

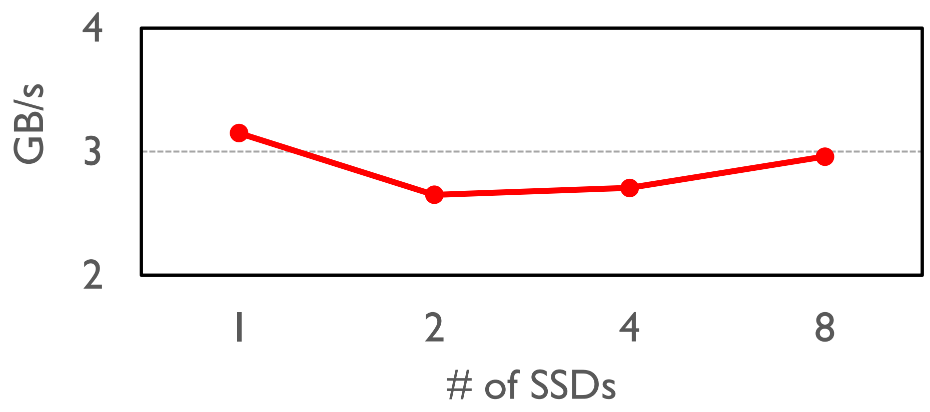




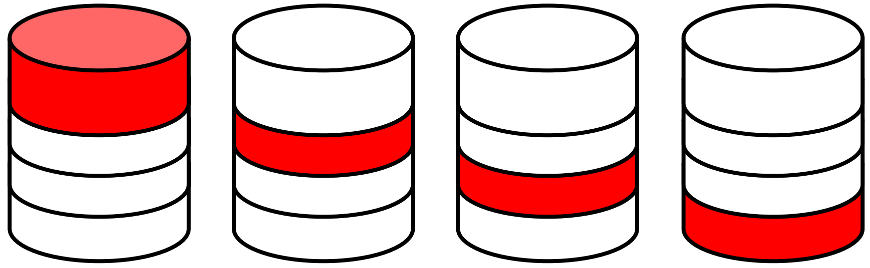
Challenge #3: Poor Scalability of RAID

- RAID always protects data with parity blocks
- Unacceptable performance and capacity penalty in AFA using multiple SSDs

Not scalable performance



25% capacity waste for parity



Motivation Summary: Performance and Capacity Penalty!

1. Indexing: two-level hash table
   - ➡ Performance degradation

2. Expiration: Do nothing or full-scanning
   - ➡ Hit rate reduction or costly scanning I/Os

3. Fault-tolerance: RAID
   - ➡ Scalability problem

✗ { *+ High performance* / *+ Huge capacity* }

✓ { *-- Small amount of DRAM* / *-- Many SSDs to manage* }

Not enough DRAM

Many SSDs

AFA KV cache

Is there a common factor in the challenges?

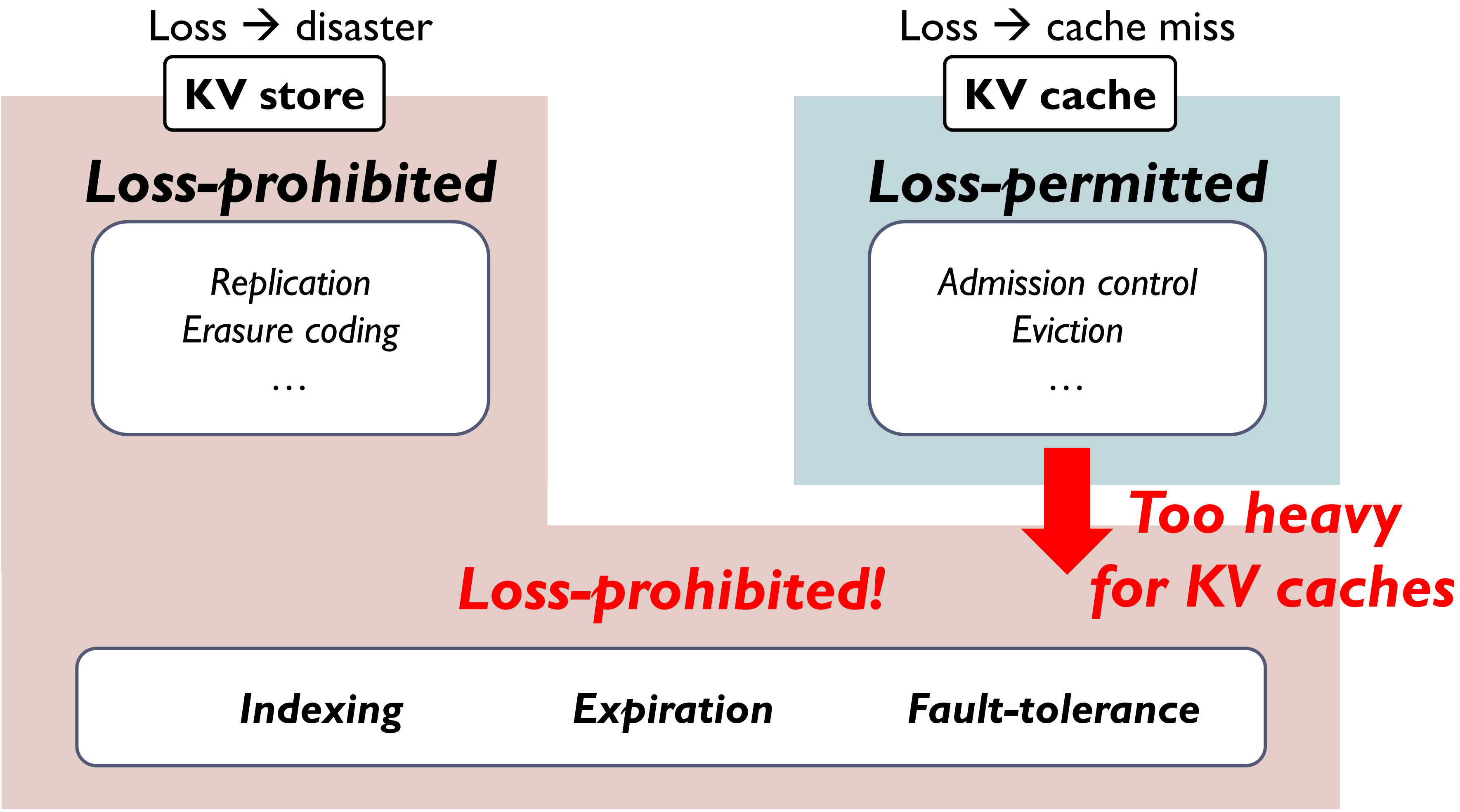
Our Approach: Data Loss

- The existing techniques all take a ***loss-prohibited*** approach

- ***Loss-prohibited***
  - Maintain all objects without any data loss

- ***Loss-permitted***
  - May lose objects when processing a task

Is the Loss-prohibited Design Mandatory for KV Caches? NO!



BigKV "Drawing the Line: Clearing Up the Differences"

Loss → disaster

KV store

Loss-prohibited

Replication
Erasure coding
…

Loss-prohibited

Indexing
Expiration
Fault-tolerance

Loss → cache miss

BigKV

Loss-permitted

Admission control
Eviction
…

Loss-permitted

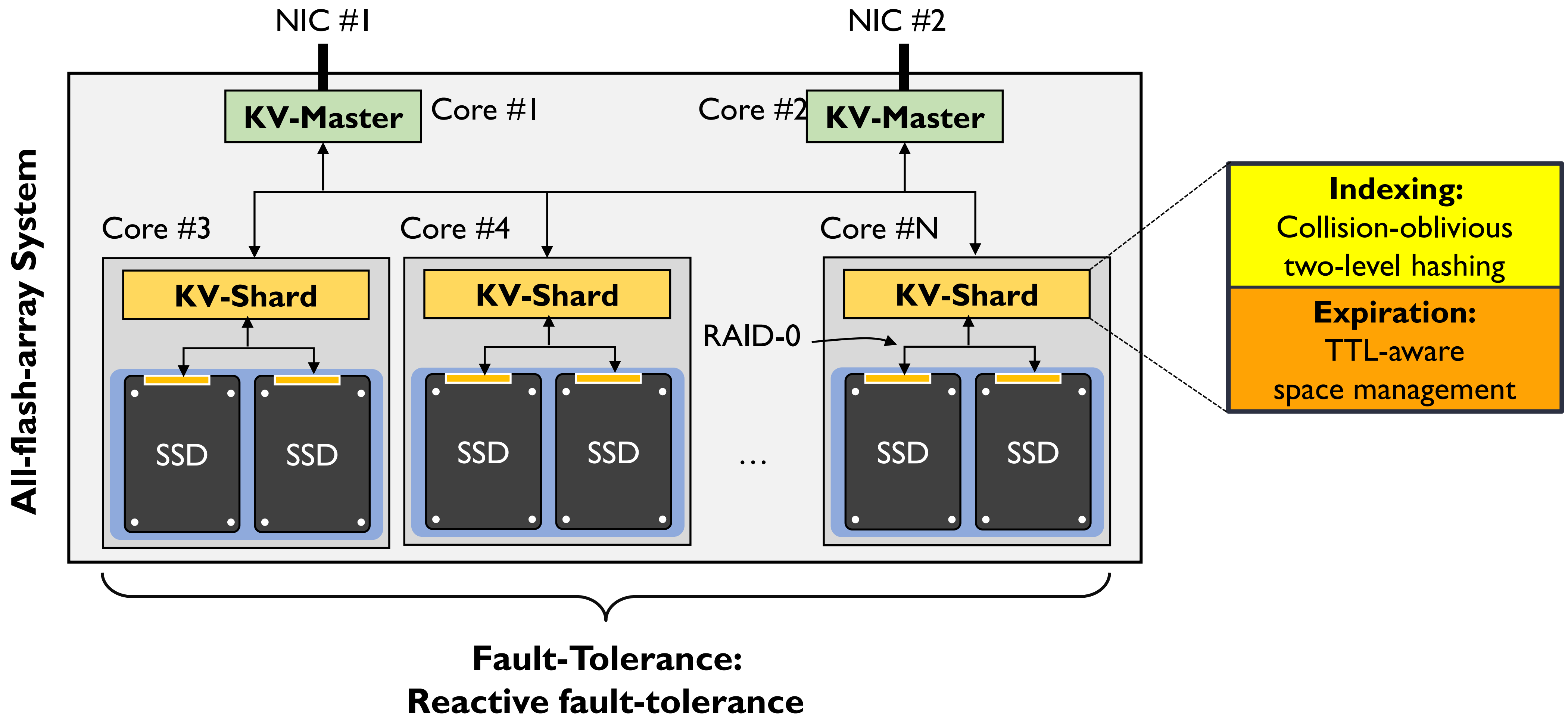
Indexing
Expiration
Fault-tolerance

Fully utilize AFA's SSDs

BigKV Design Overview

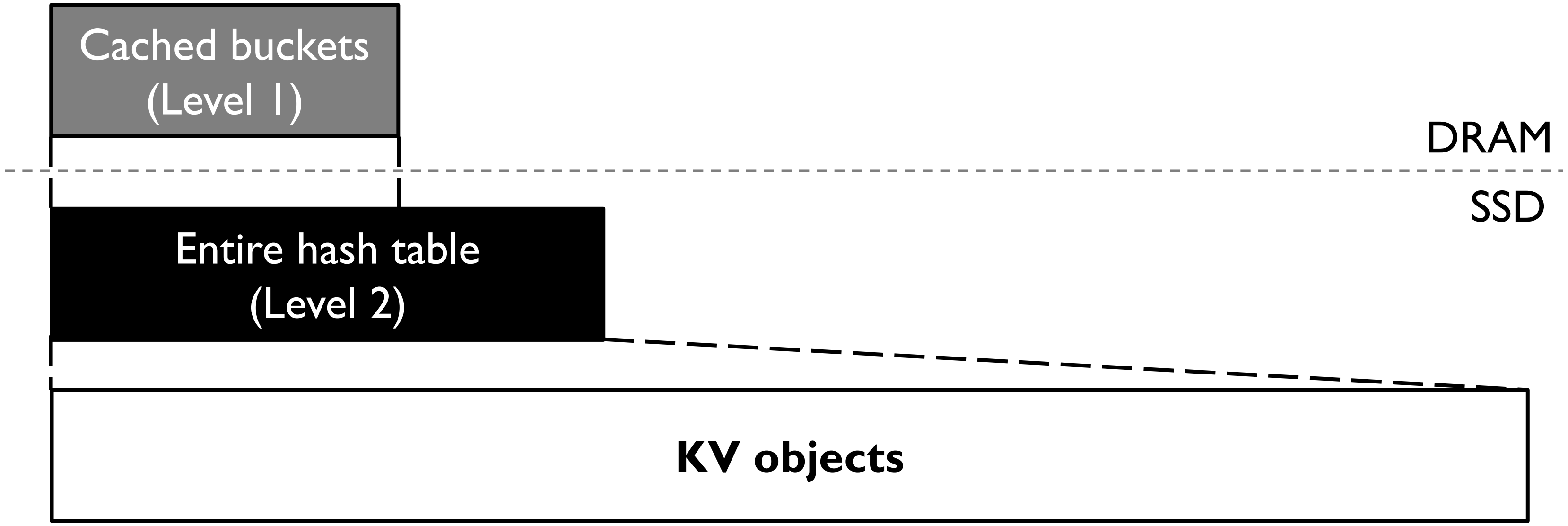
1. Collision-oblivious two-level hashing
   - Collision-oblivious object update
   - Bounded object lookup
   - Metadata eviction
2. TTL-aware space management
   - TTL-aware grouping
   - TTL approximation
   - Zombie object eviction
3. Reactive fault-tolerance
   - Reactive fault-tolerance with sharding
   - Metadata persistence

BigKV Design Overview



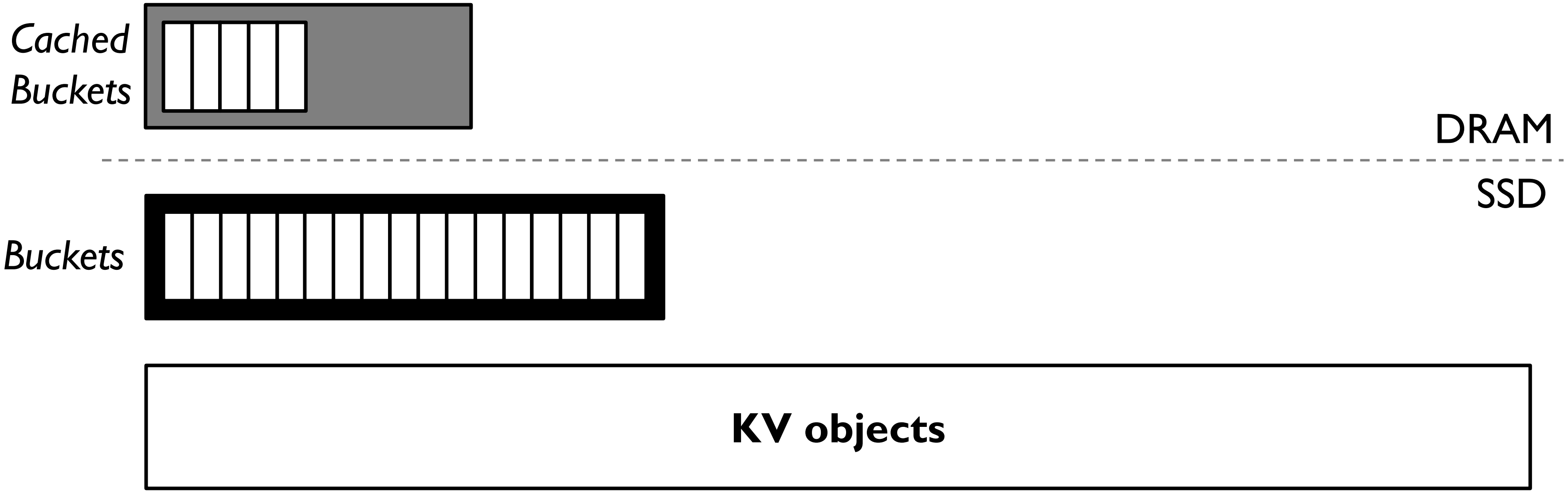
Performance Problem of Existing Indexing

- Level 1 in DRAM: recently-accessed hash buckets
- Level 2 in SSD: entire hash table
- Each hash buckets point to actual KV objects



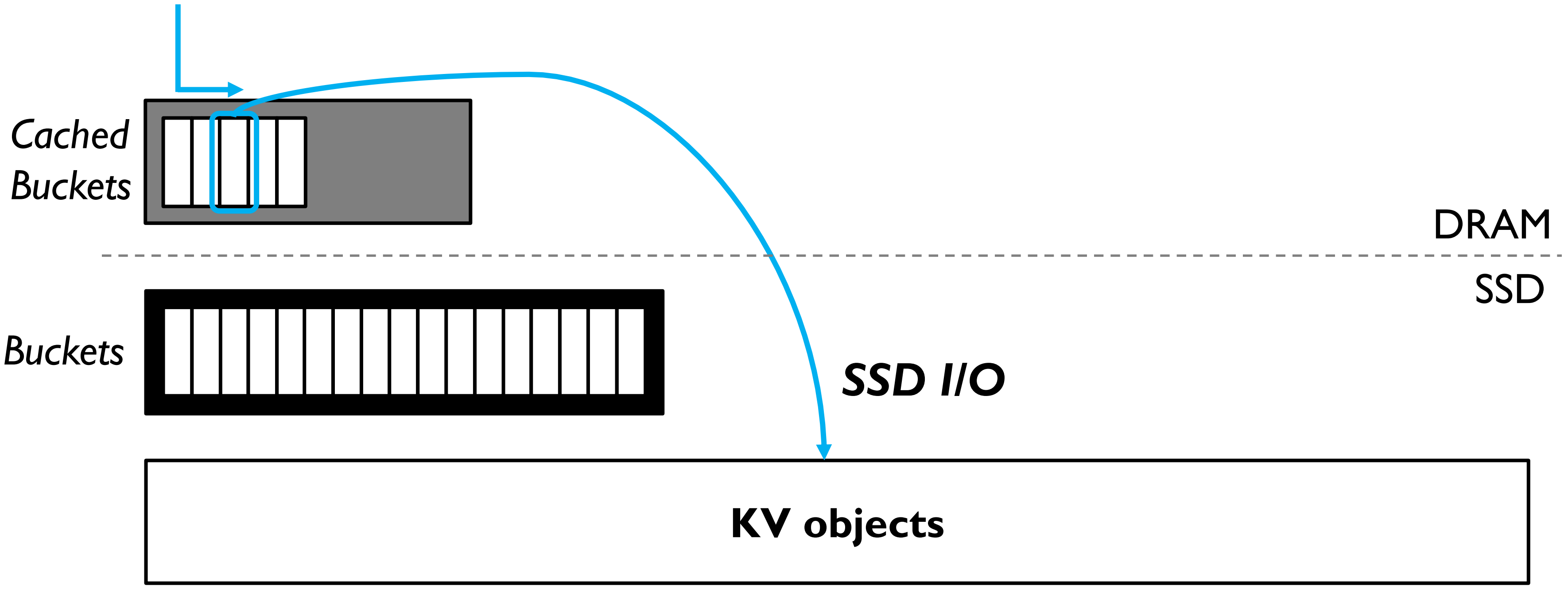
Performance Problem of Existing Indexing

Execution time = *Hit time* + *Miss rate* × *Miss time*



Performance Problem of Existing Indexing

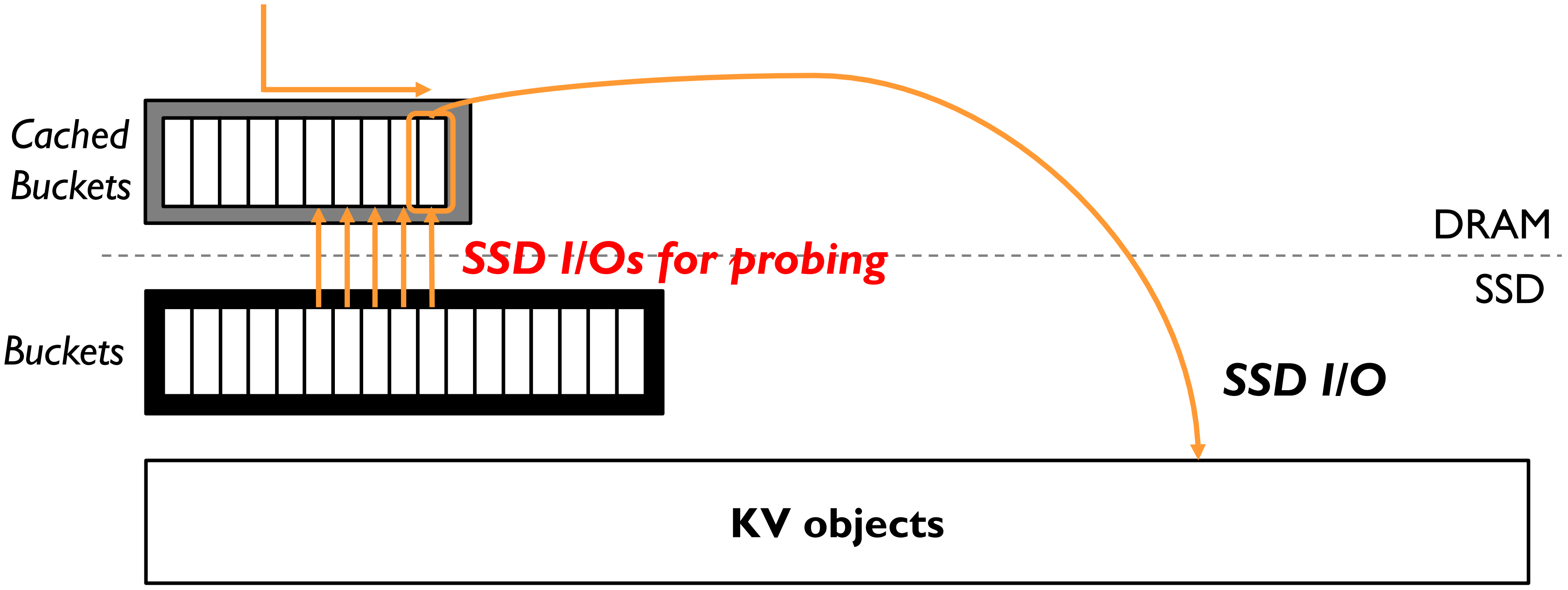
Execution time = *Hit time* + *Miss rate* × *Miss time*



Performance Problem of Existing Indexing

Execution time = *Hit time* + *Miss rate* × *Miss time*

-- several probing I/Os

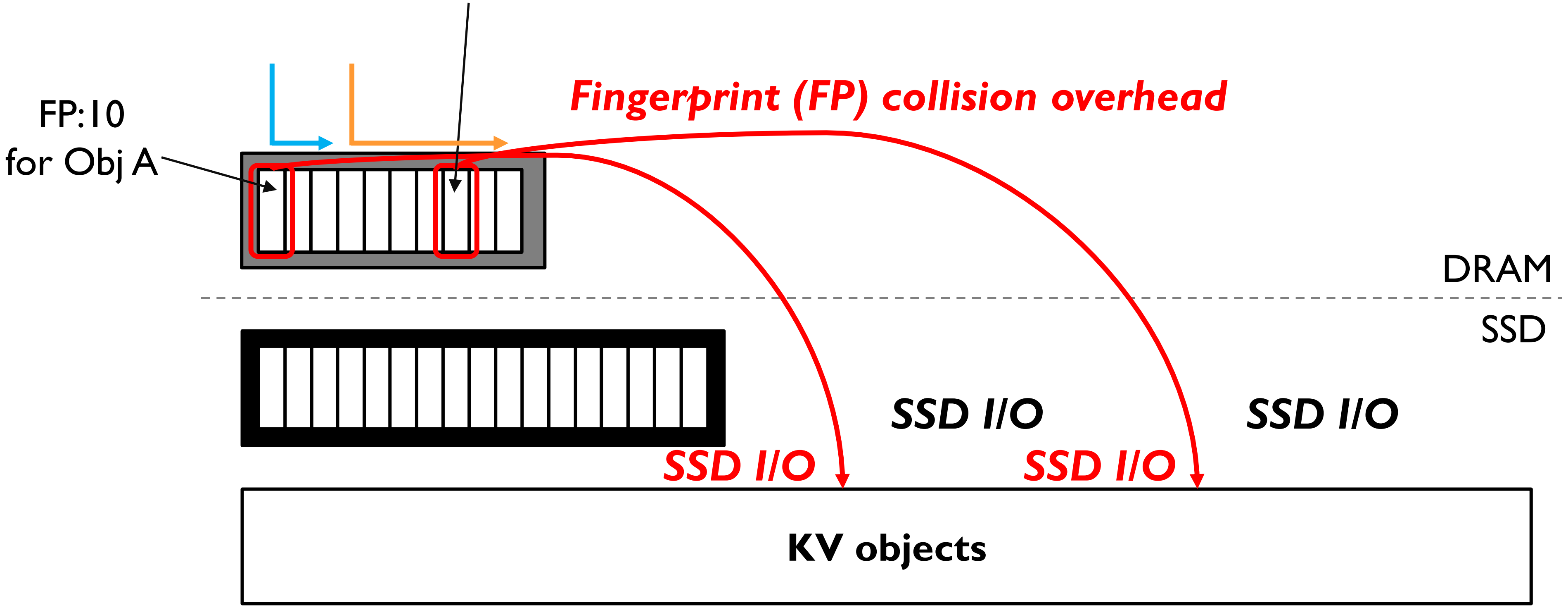


Performance Problem of Existing Indexing

Execution time = *Hit time* + *Miss rate* × *Miss time*

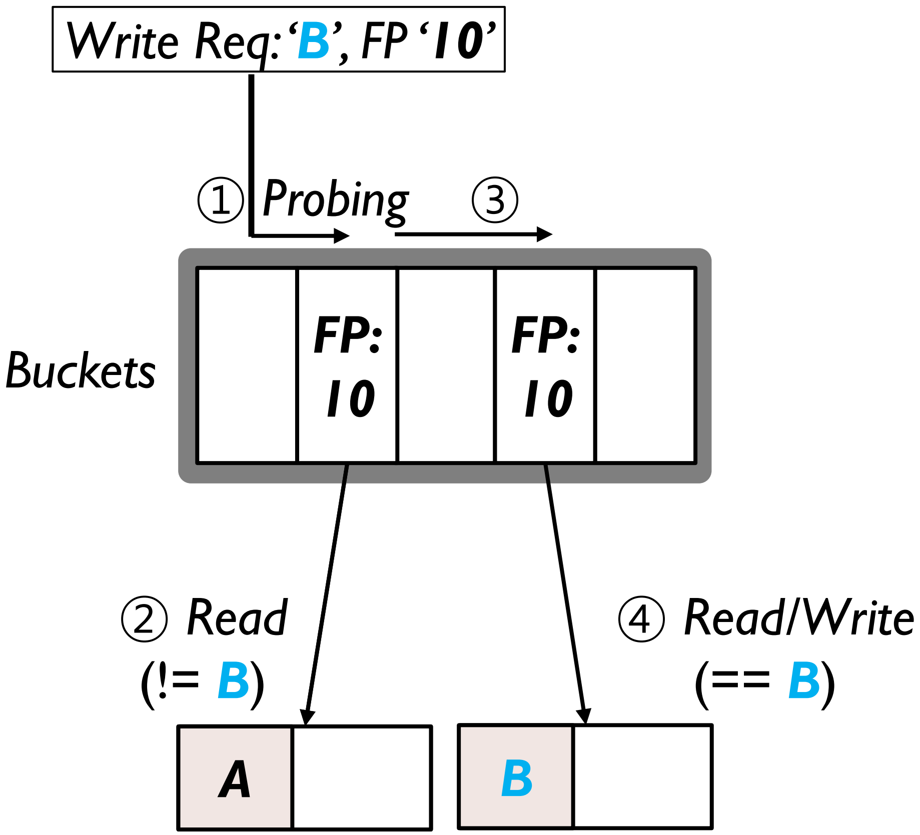
-- FP collision I/Os

-- several probing I/Os
-- FP collision I/Os



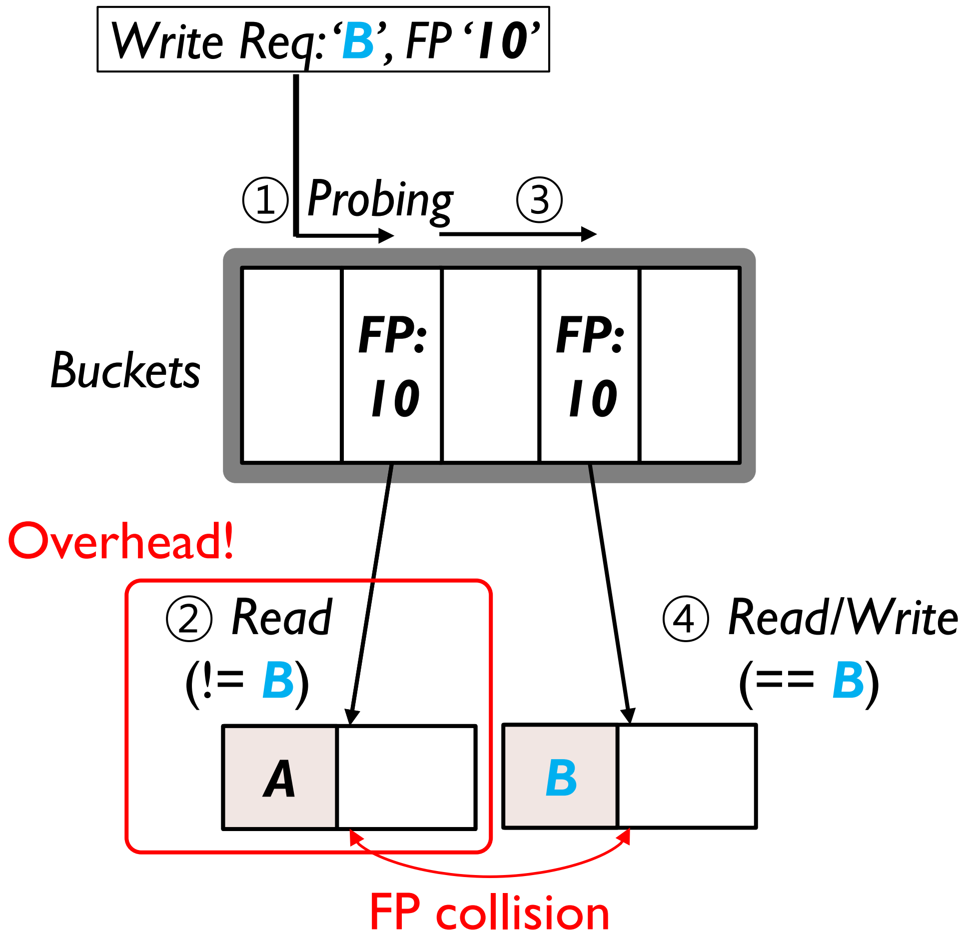
Fingerprint Collision

- Fingerprint (FP)
  - Integer obtained by hashing an object's full-key
- FP collision
  - Same FP, different full-keys



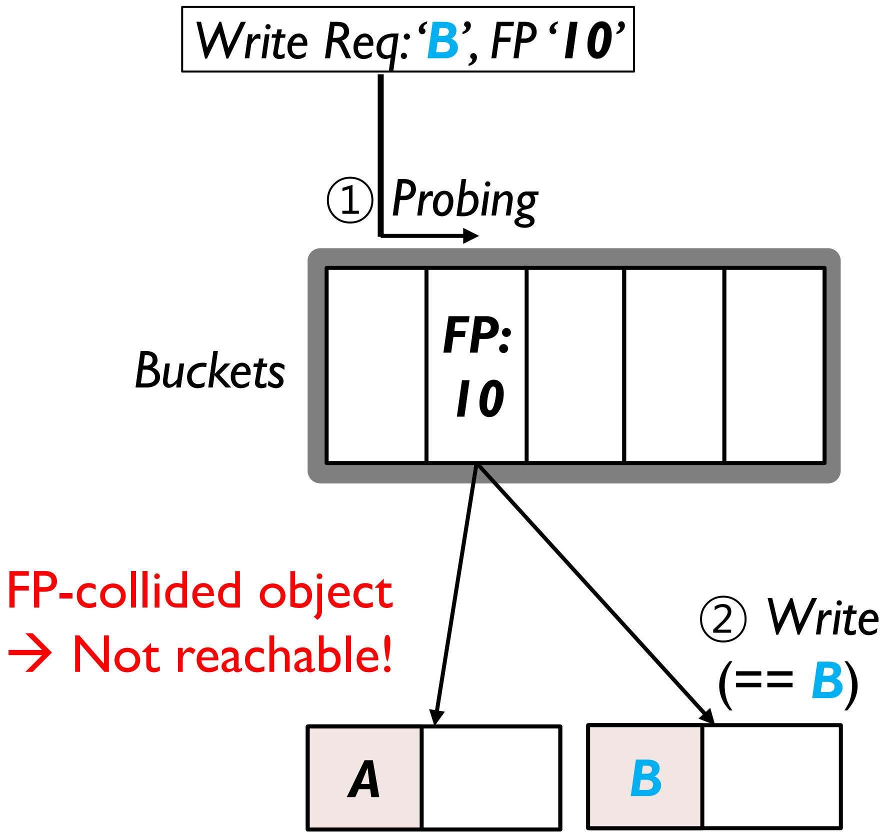
Fingerprint Collision

- Fingerprint (FP)
  - Integer obtained by hashing an object's full-key
- FP collision
  - Same FP, different full-keys
  - Incur additional object reads
- ***Loss-prohibited*** indexing
  - Store all objects, ignoring FP collisions



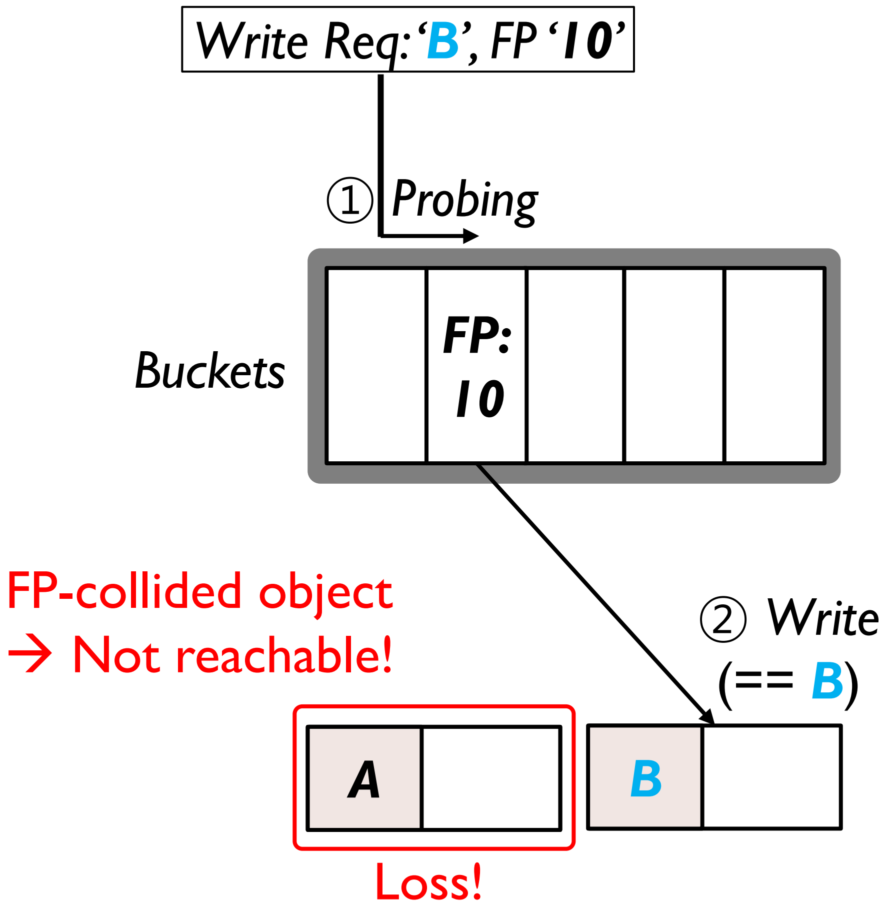
Collision-oblivious Hashing of BigKV

- No FP collisions
  - When writing an object, simply overwrite the FP-matched bucket
  - No additional object reads



Collision-oblivious Hashing of BigKV

- No FP collisions
  - When writing an object,
    simply overwrite the FP-matched bucket
  - No additional object reads
- ***Loss-permitted*** indexing
  - Lose the old FP-collided object
- Data loss penalty?
  - Minimized by optimizations
    - Large FP size, hash table organization
    - → 5 misses out of 400M requests
    - → Minimal drop in cache hit ratio

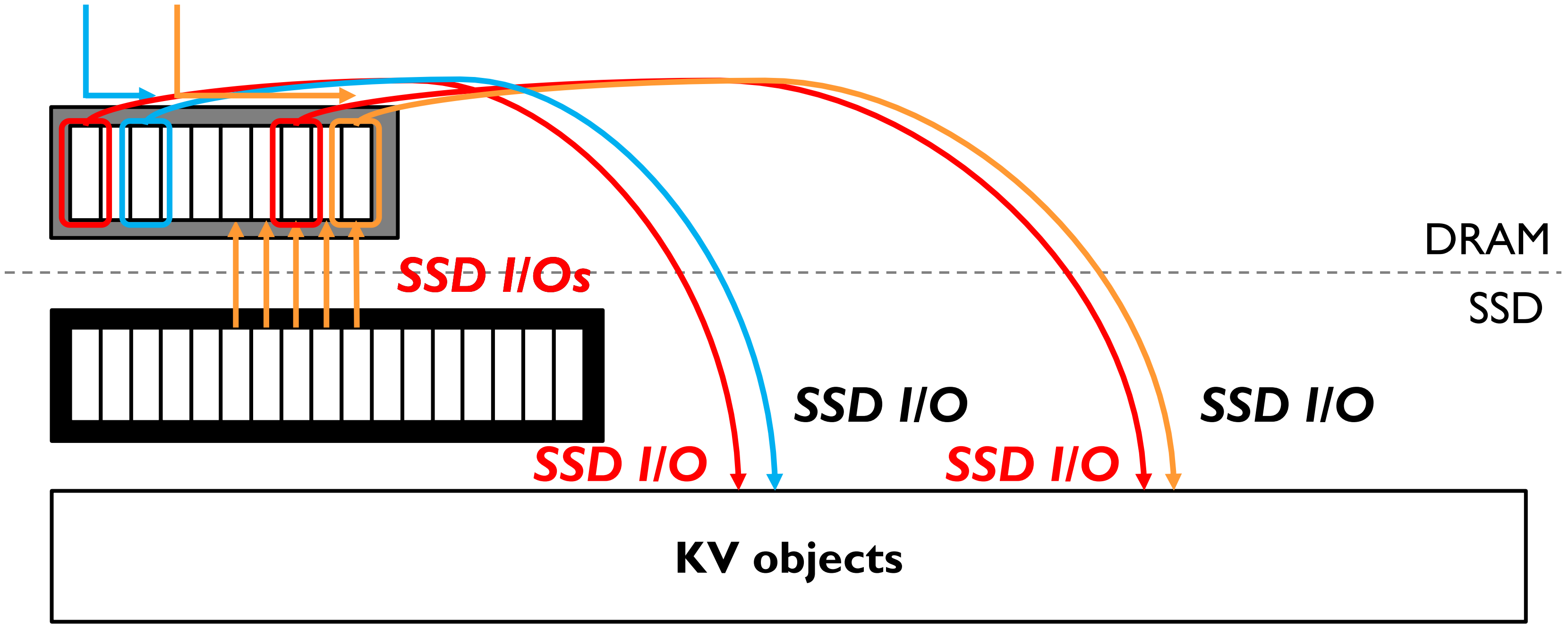


BigKV: Eliminate I/O Overhead!

Execution time = *Hit time* + *Miss rate* × *Miss time*

-- FP collision I/Os

-- several probing I/Os
-- FP collision I/Os



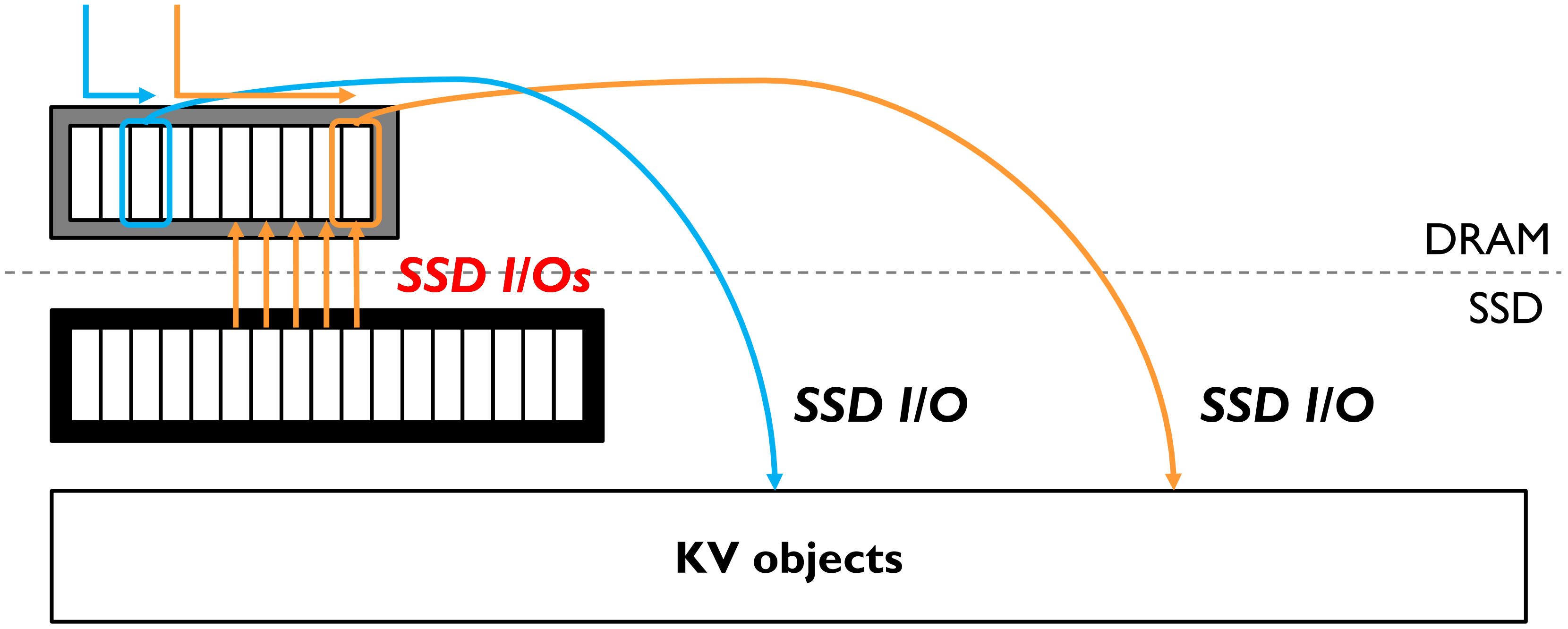
BigKV: Eliminate I/O Overhead!

Execution time = *Hit time* + *Miss rate* × *Miss time*

-- FP collision I/Os

-- several probing I/Os

-- FP collision I/Os



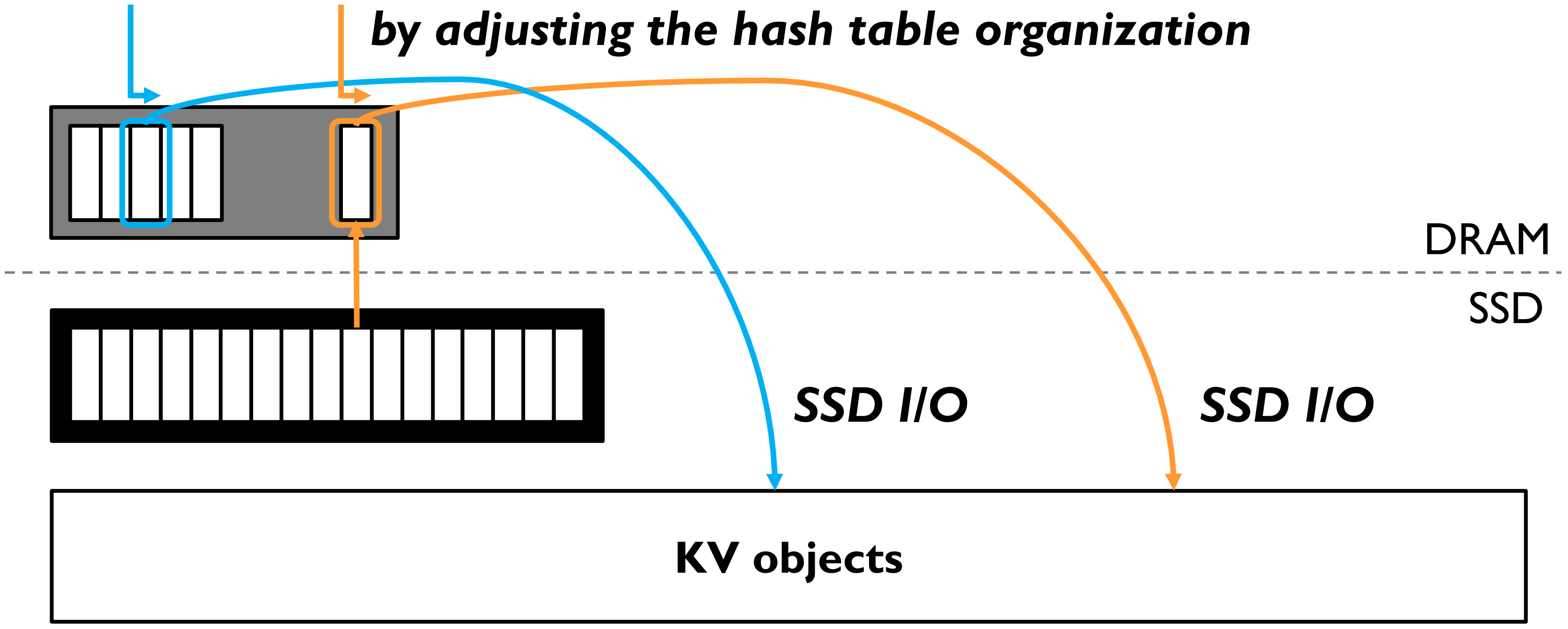
BigKV: Eliminate I/O Overhead!

Execution time = *Hit time* + *Miss rate* × *Miss time*

-- FP collision I/Os

-- several probing I/Os
-- FP collision I/Os
+ Only one probing I/O

Limit the probing distance
by adjusting the hash table organization



BigKV: Eliminate I/O Overhead!

Execution time = Hit time + Miss rate × Miss time

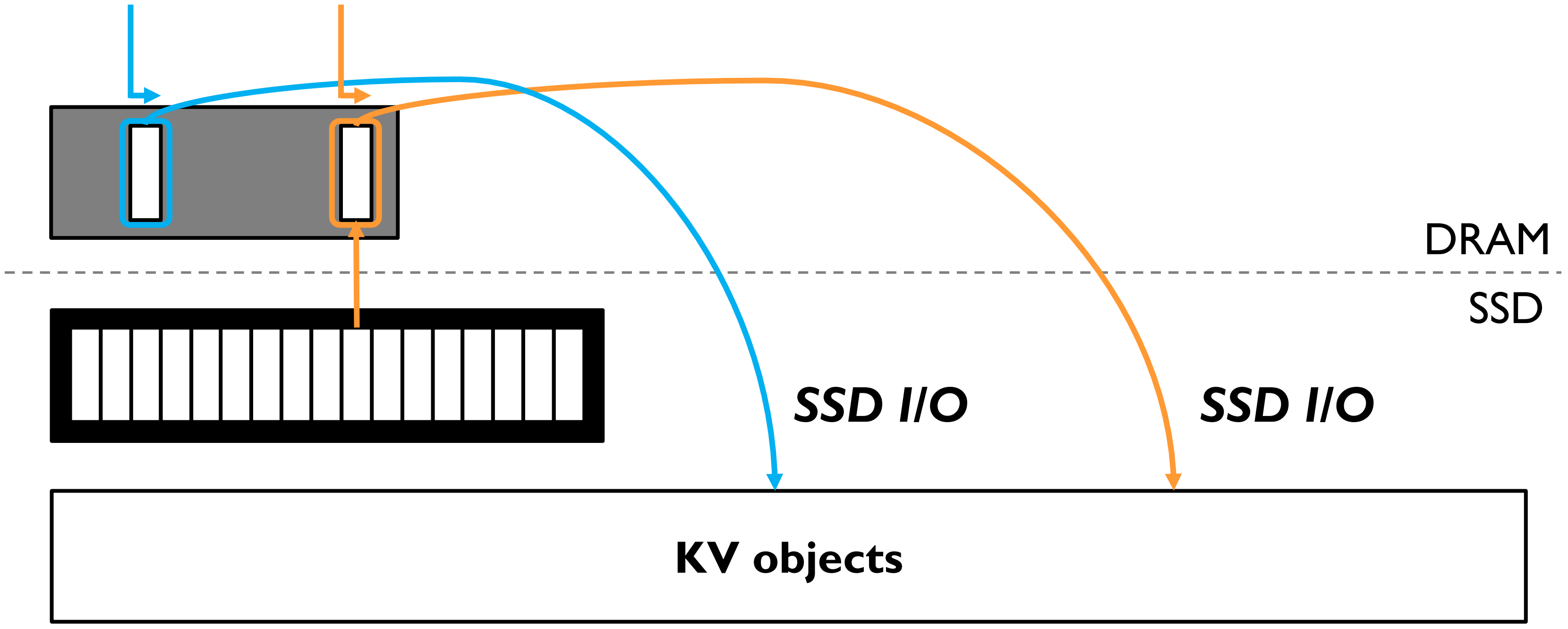
-- FP collision I/Os + Hot bucket caching -- several probing I/Os

-- FP collision I/Os

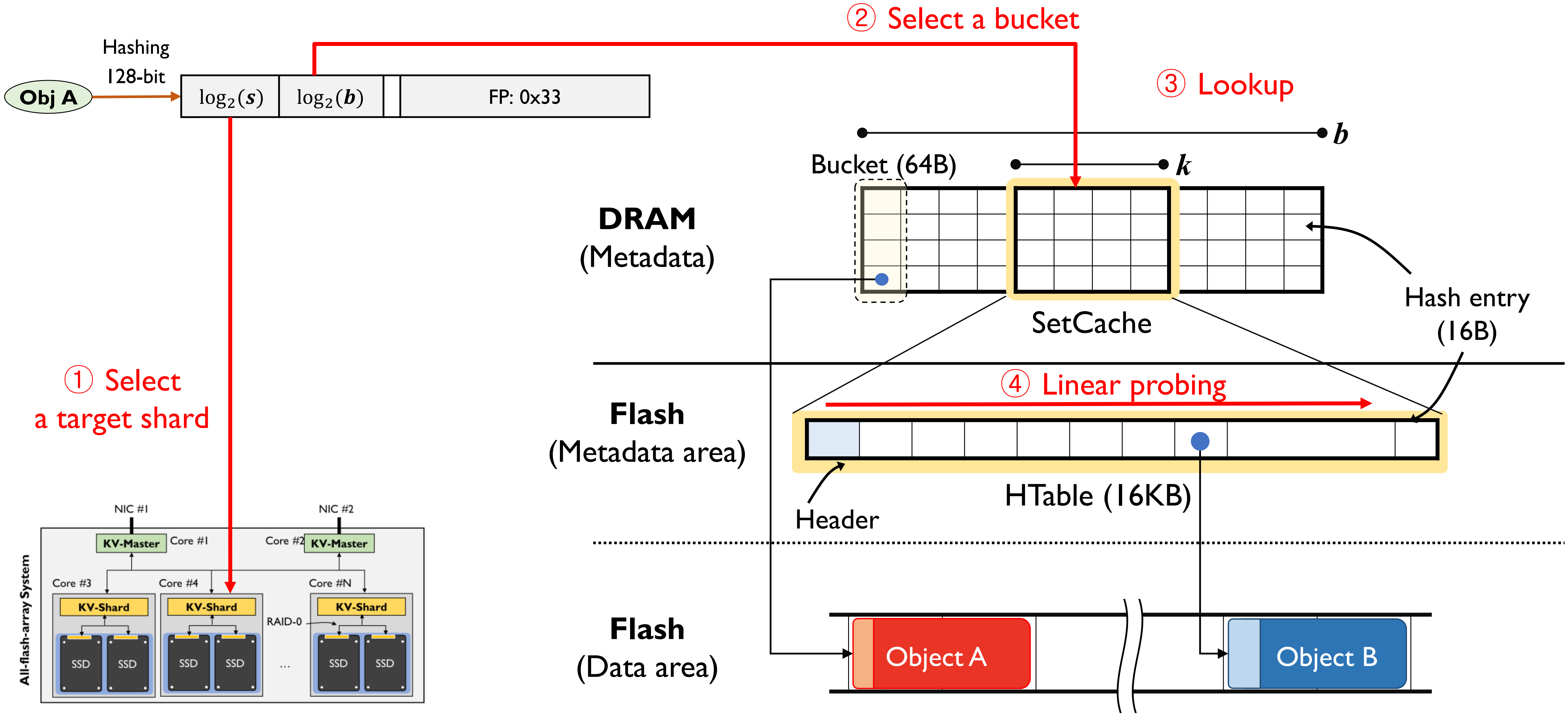
+ Only one probing I/O

Only caching hot buckets

→ reduce the bucket miss rate



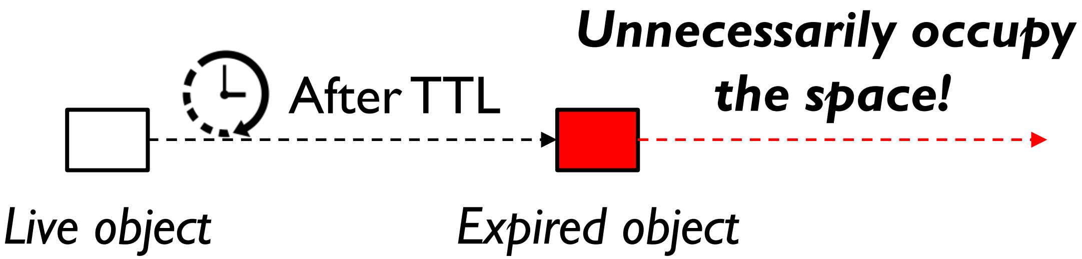
BigKV: Two-level Metadata Indexing (Detail)



Time-to-live (TTL)

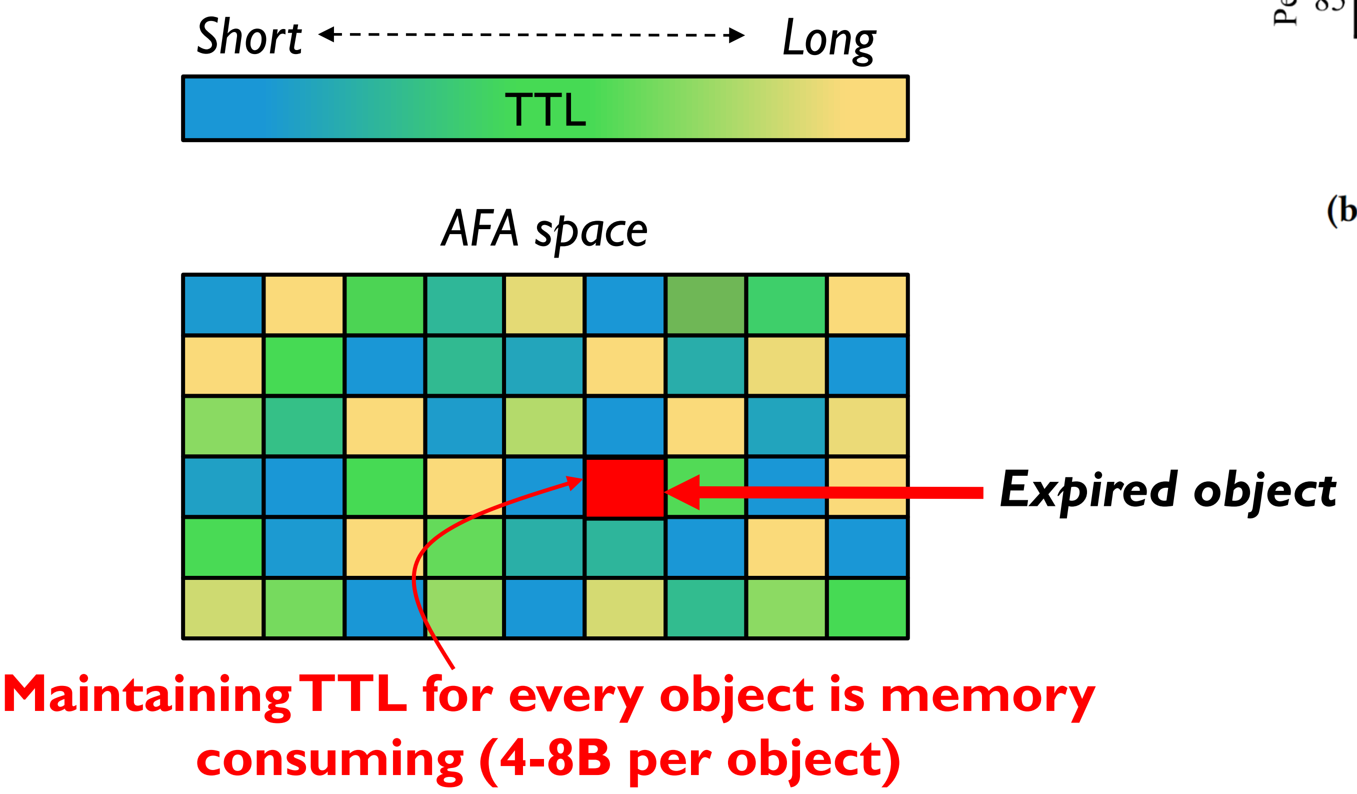
- TTL → object's lifetime

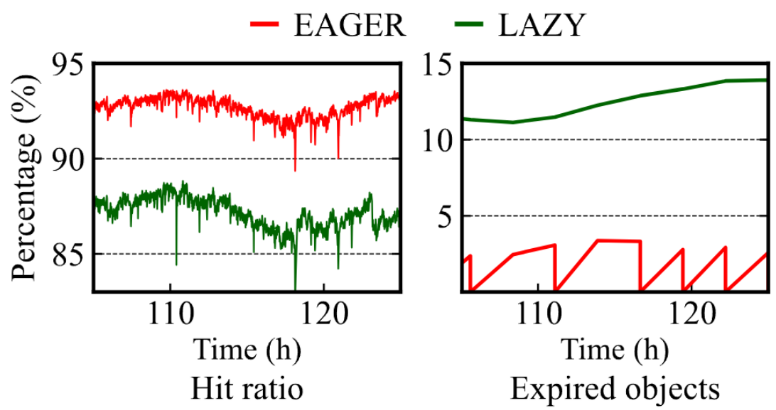
- Expired object
  - Unnecessarily occupy the space
  - Should be eliminated ASAP for the high hit rate



Loss-prohibited Expiration is Too Costly!

- ***Loss-prohibited*** expiration
  - Remove only expired objects exactly

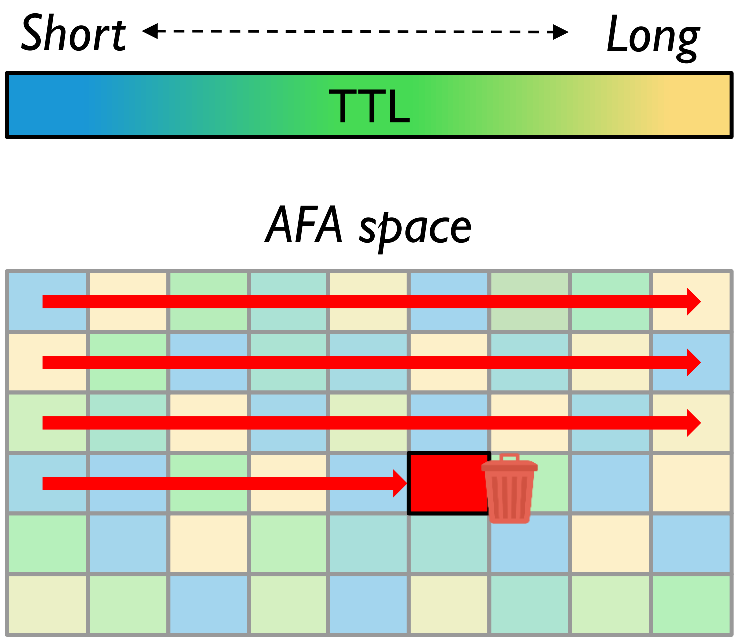




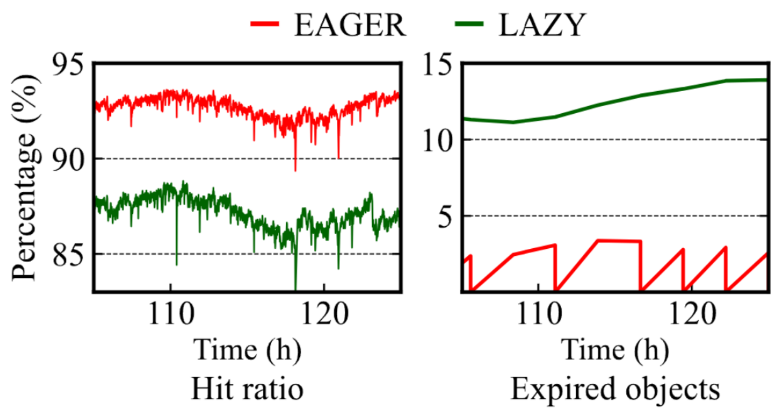
(b) Cache hit ratio and % of expired objects

Loss-prohibited Expiration is Too Costly!

- ***Loss-prohibited*** expiration
  - Remove only expired objects exactly



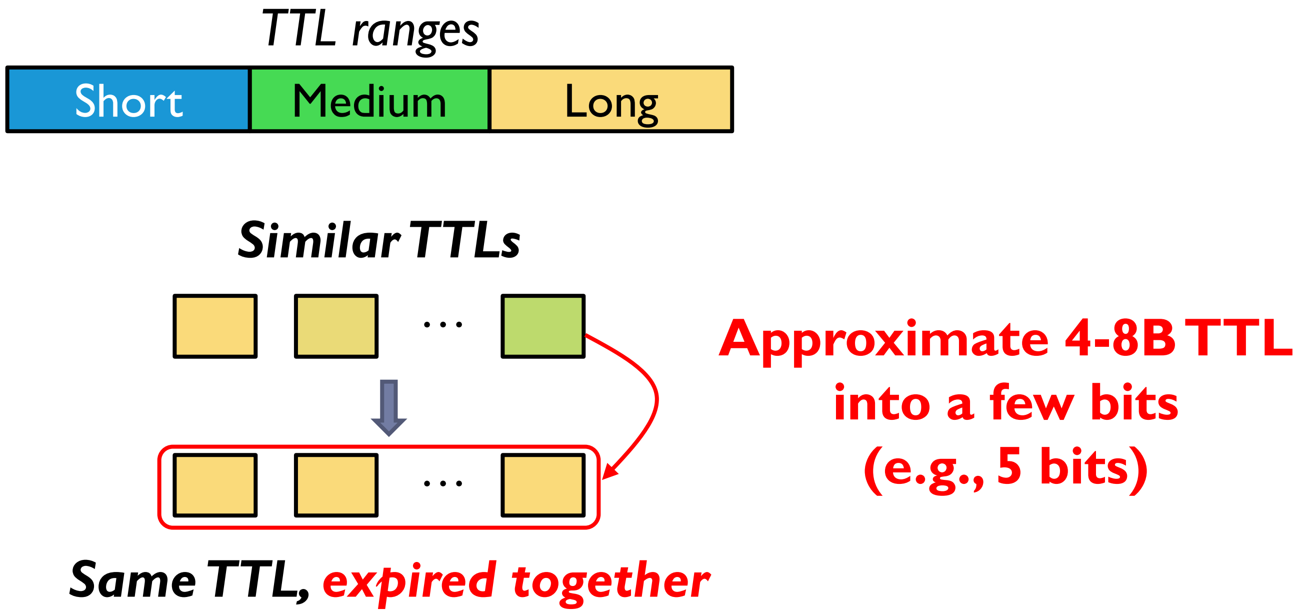
Lots of scanning I/Os
for the loss-prohibited expiration



(b) Cache hit ratio and % of expired objects

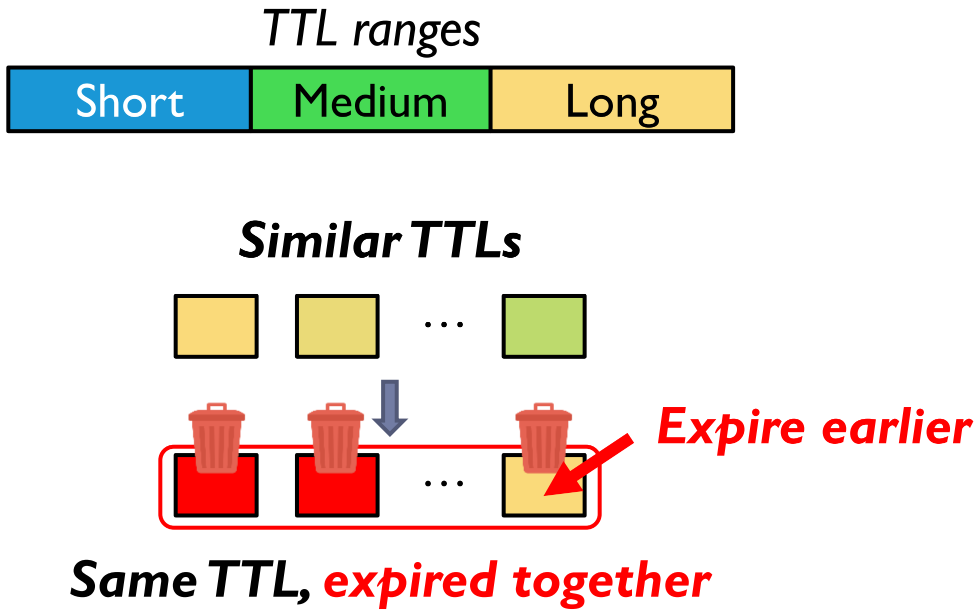
TTL-aware object grouping of BigKV

- Group and expire objects which have similar TTLs together
- ***Loss-permitted*** expiration
  - May remove still-alive objects



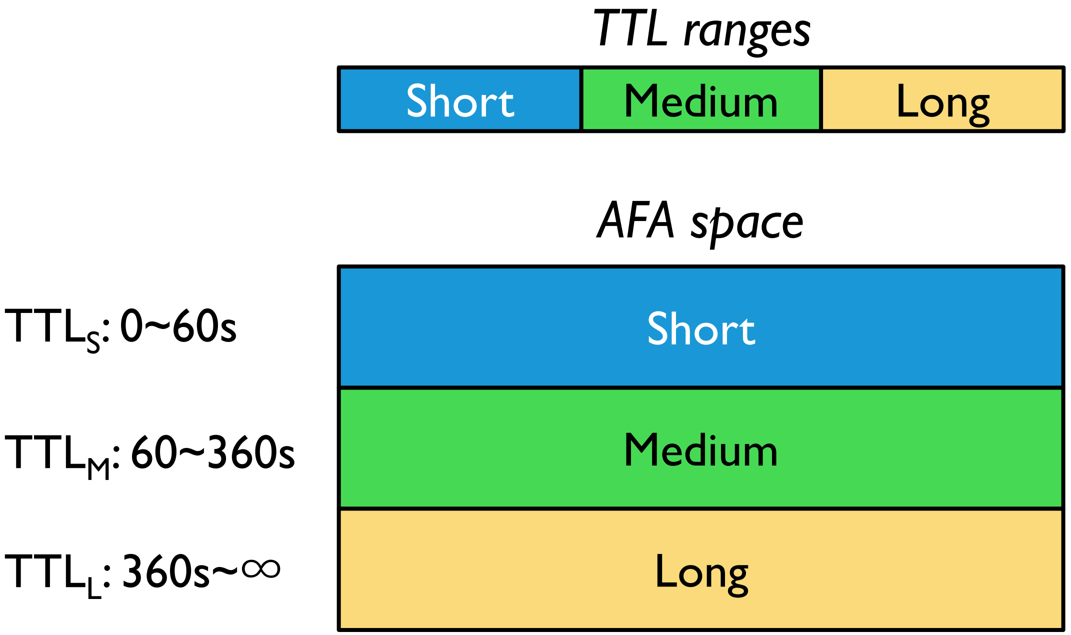
TTL-aware object grouping of BigKV

- Group and expire objects which have similar TTLs together
- ***Loss-permitted*** expiration
  - May remove still-alive objects



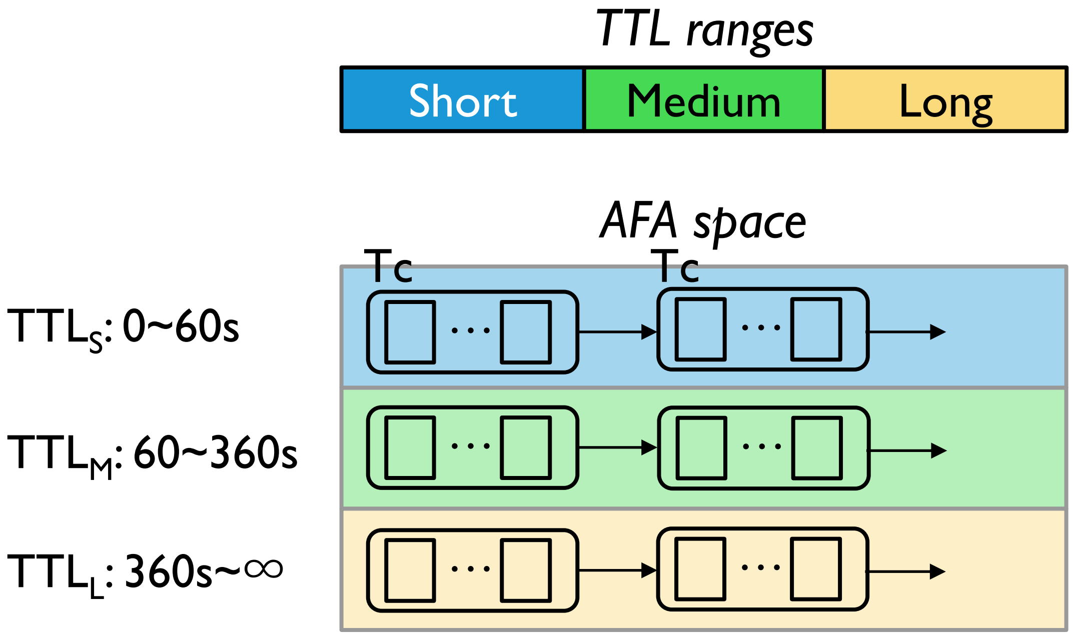
TTL-aware Space Management

- Proactively remove expired objects with near-zero overhead



TTL-aware Space Management

- Proactively remove expired objects with near-zero overhead

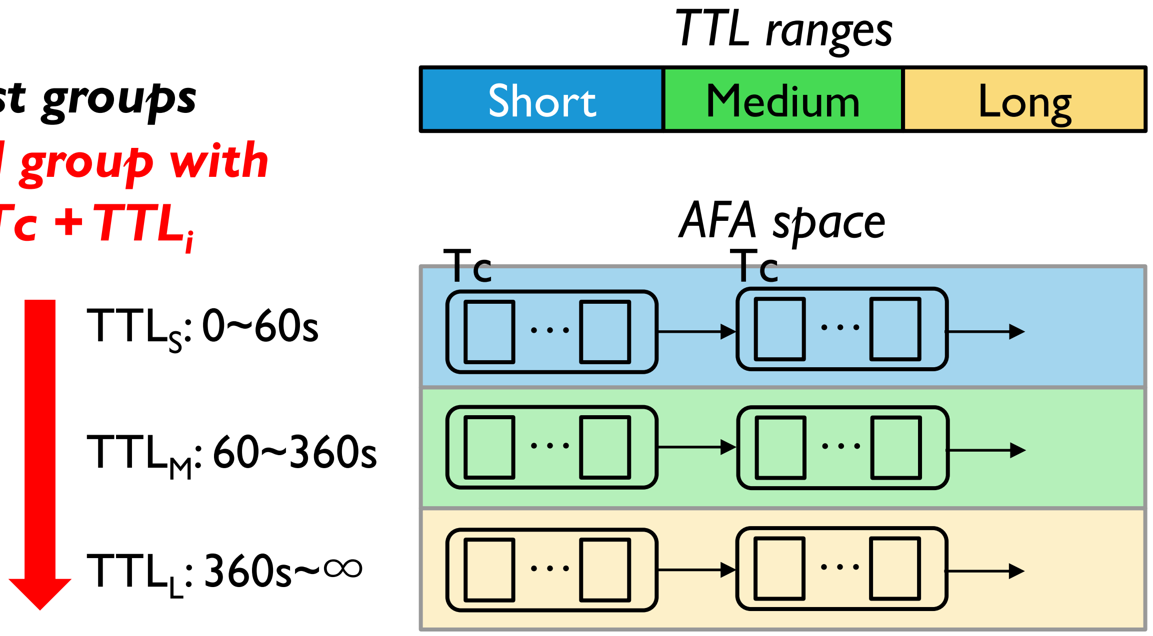


TTL-aware Space Management

- Proactively remove expired objects with near-zero overhead

Check the oldest groups

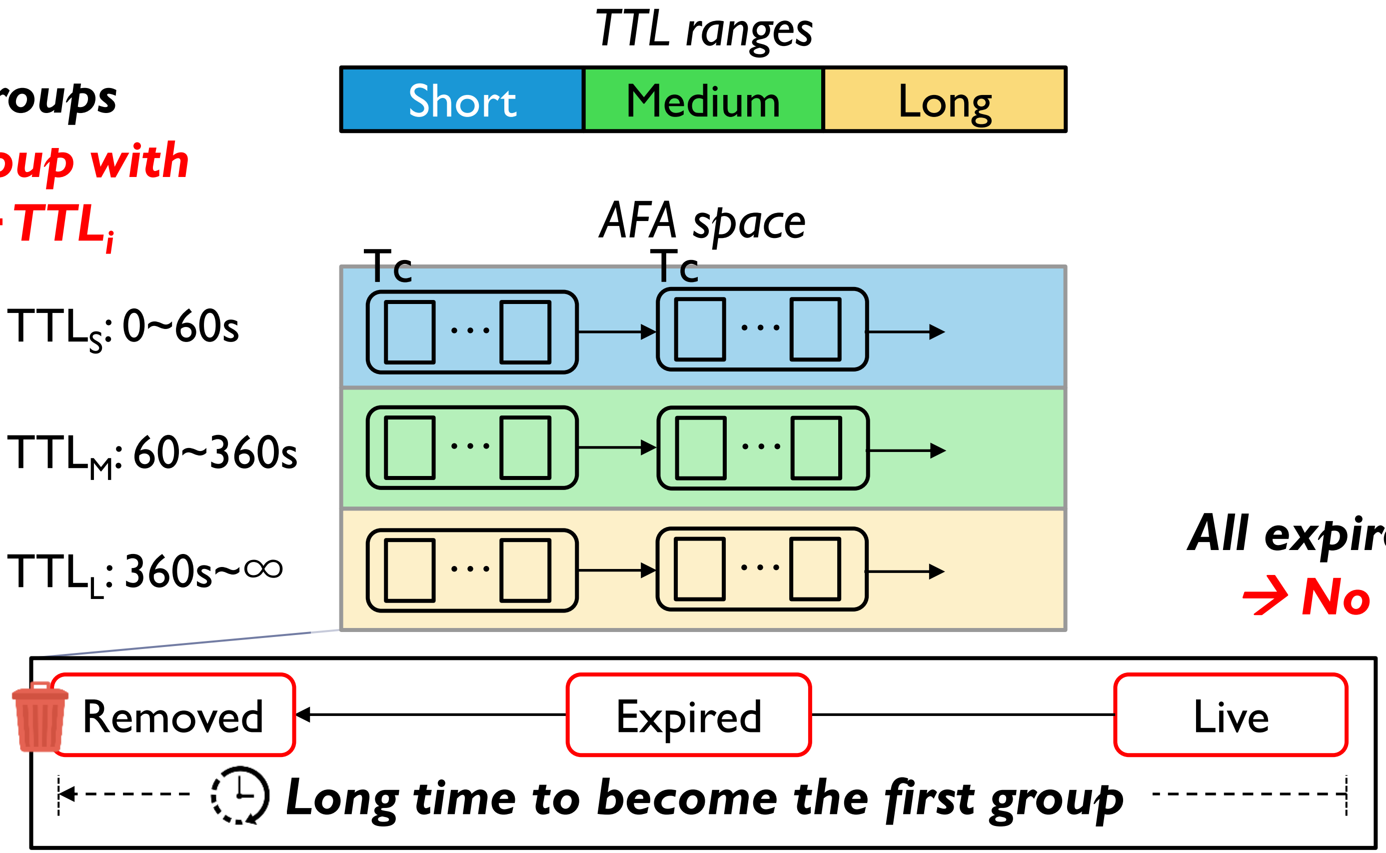
→ *Remove expired group with current time > Tc + TTL$_i$*



TTL-aware Space Management

- Proactively remove expired objects with near-zero overhead

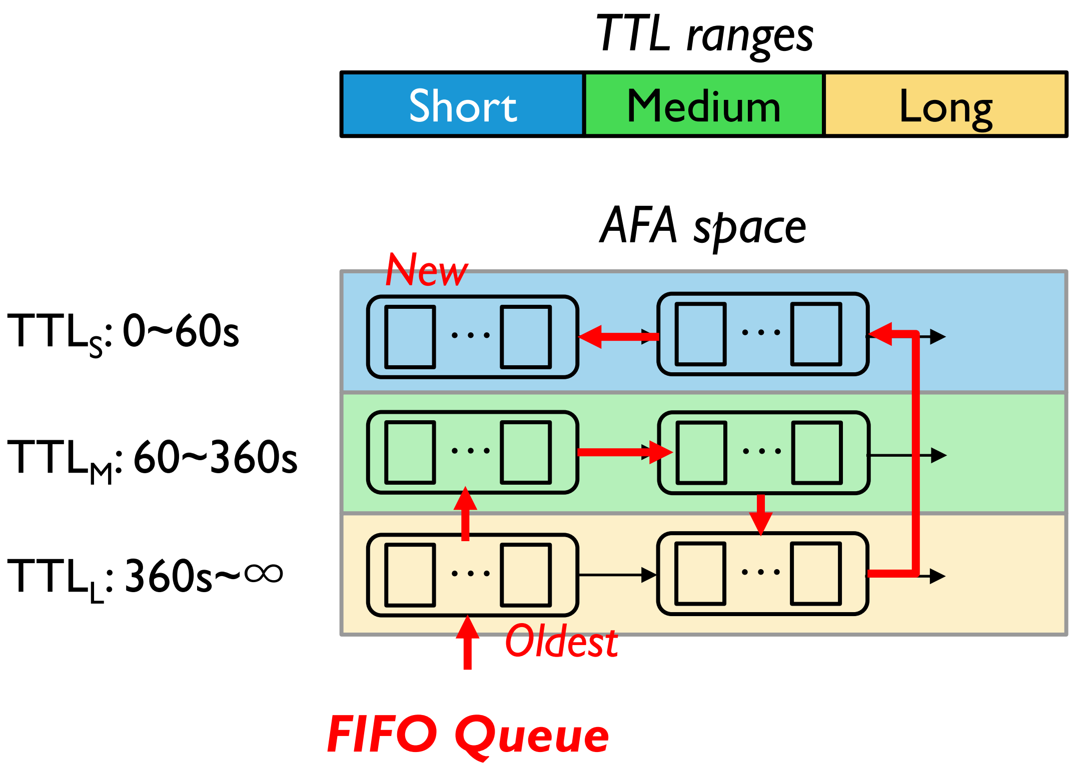
Check the oldest groups
→ Remove expired group with current time > $Tc + TTL_i$



All expired over a long time
→ No data loss penalty

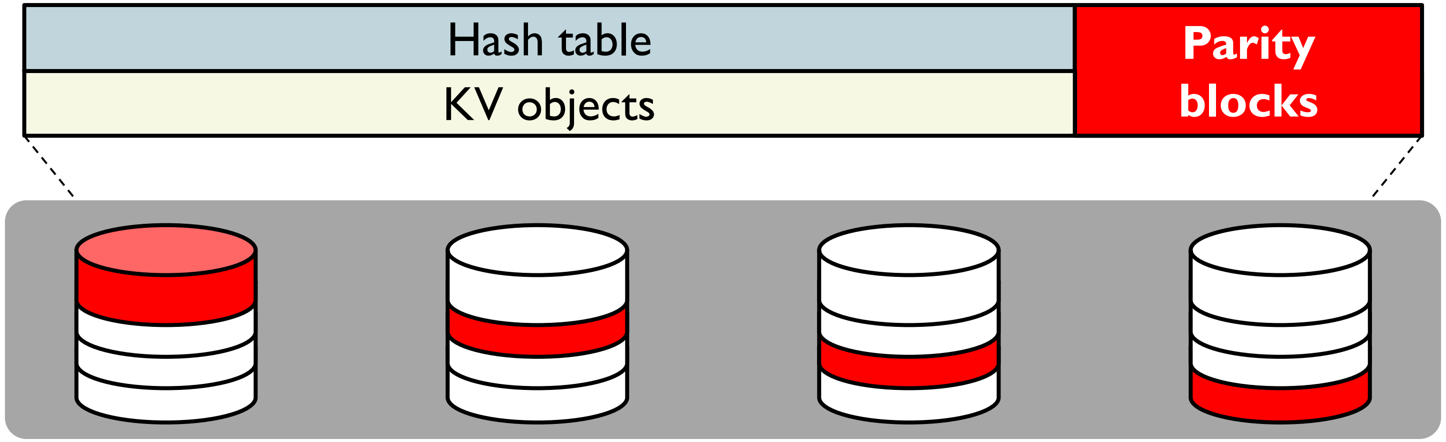
No Expired Group?

- If there is no expired group, choose the oldest group



Problem of RAID for Fault-tolerance

- ***Loss-prohibited*** RAID
  - Always protect data
- Performance/capacity overheads due to parity blocks

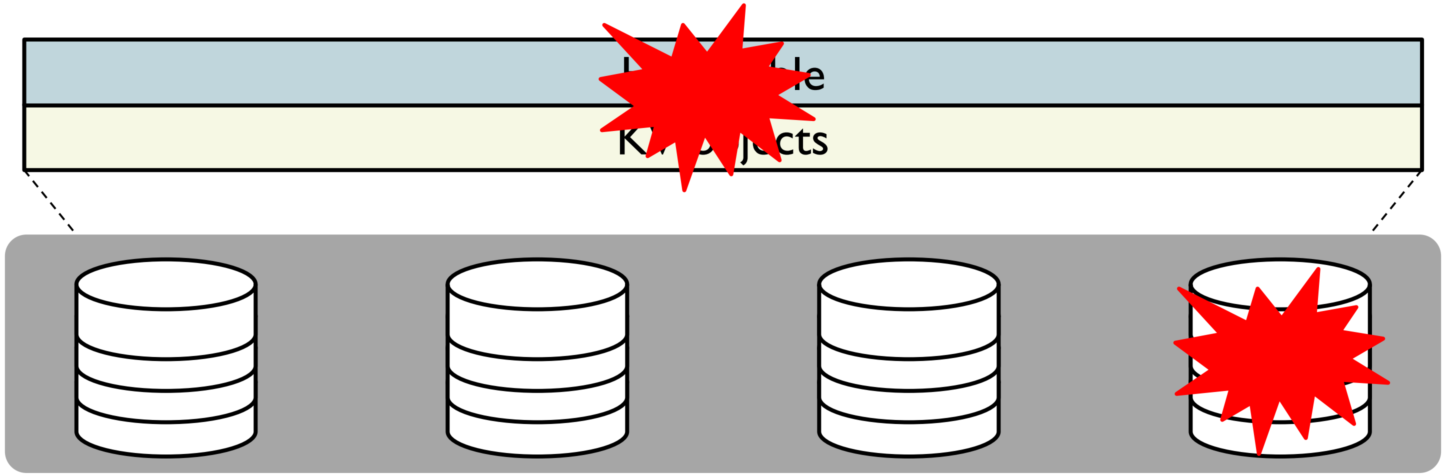


RAID-parity

BigKV: Reactive Fault-tolerance with Sharding

- ***Loss-permitted*** fault-tolerance
  - High scalability, but losing objects
- Sharding rather than striping
  - Isolating loss penalty

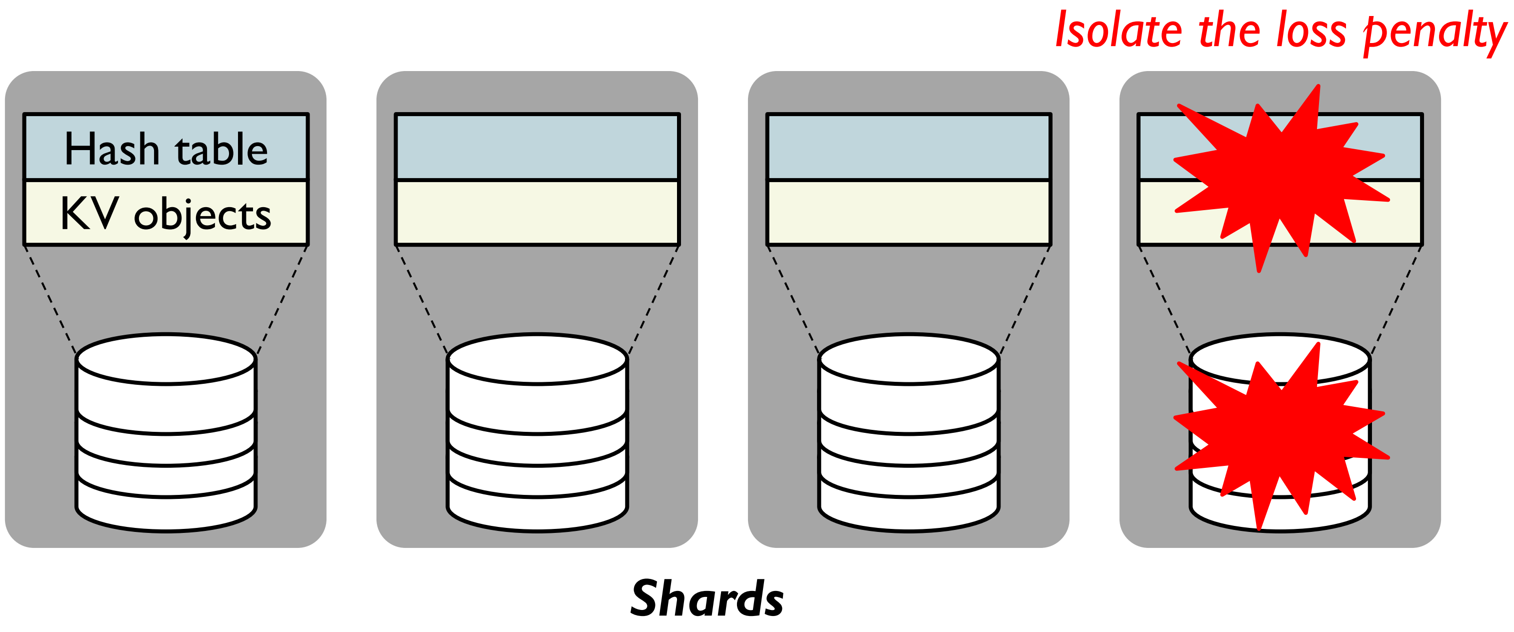
High scalability but losing all data!



RAID-stripe?

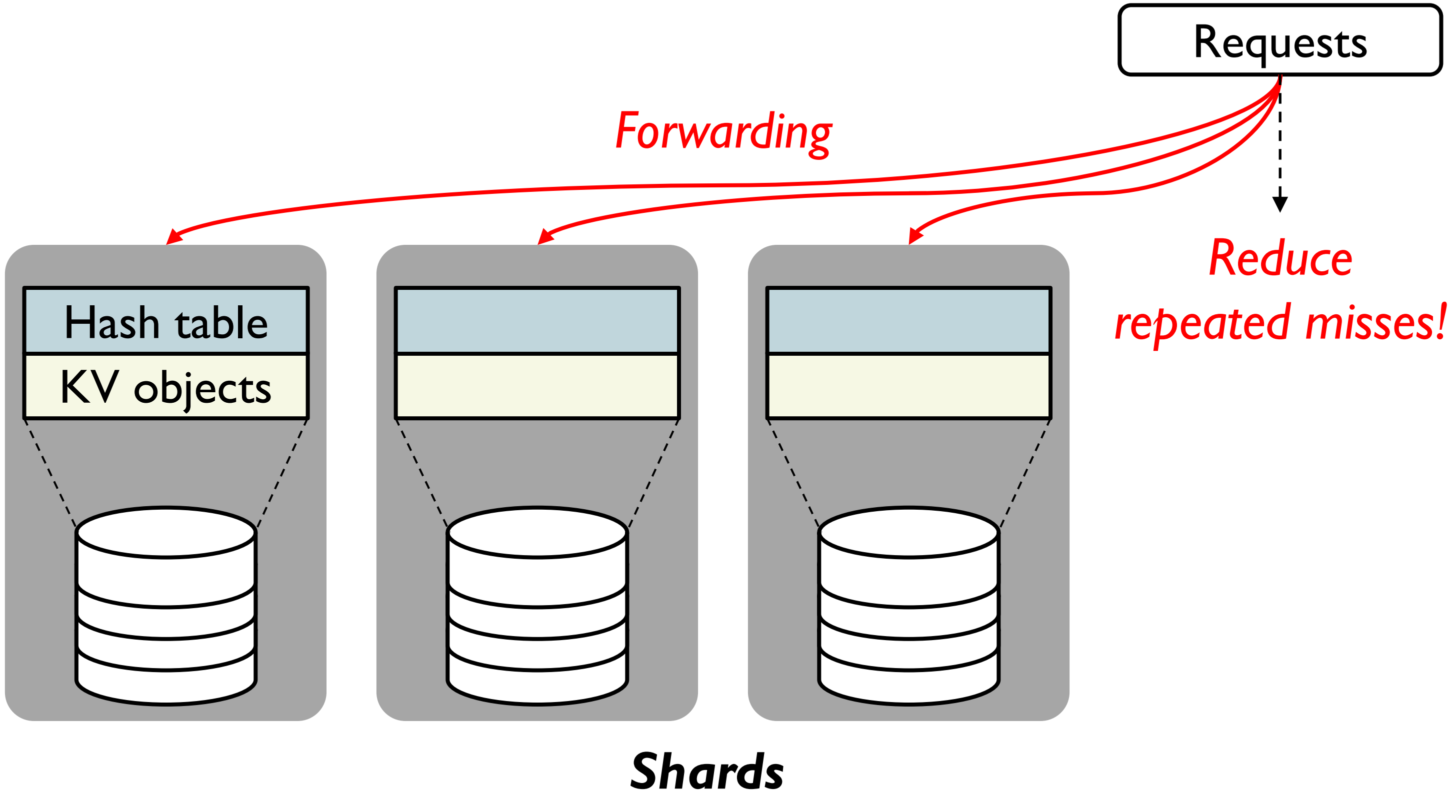
BigKV: Reactive Fault-tolerance with Sharding

- ***Loss-permitted*** fault-tolerance
  - High scalability, but losing objects
- Sharding rather than striping
  - Isolating loss penalty



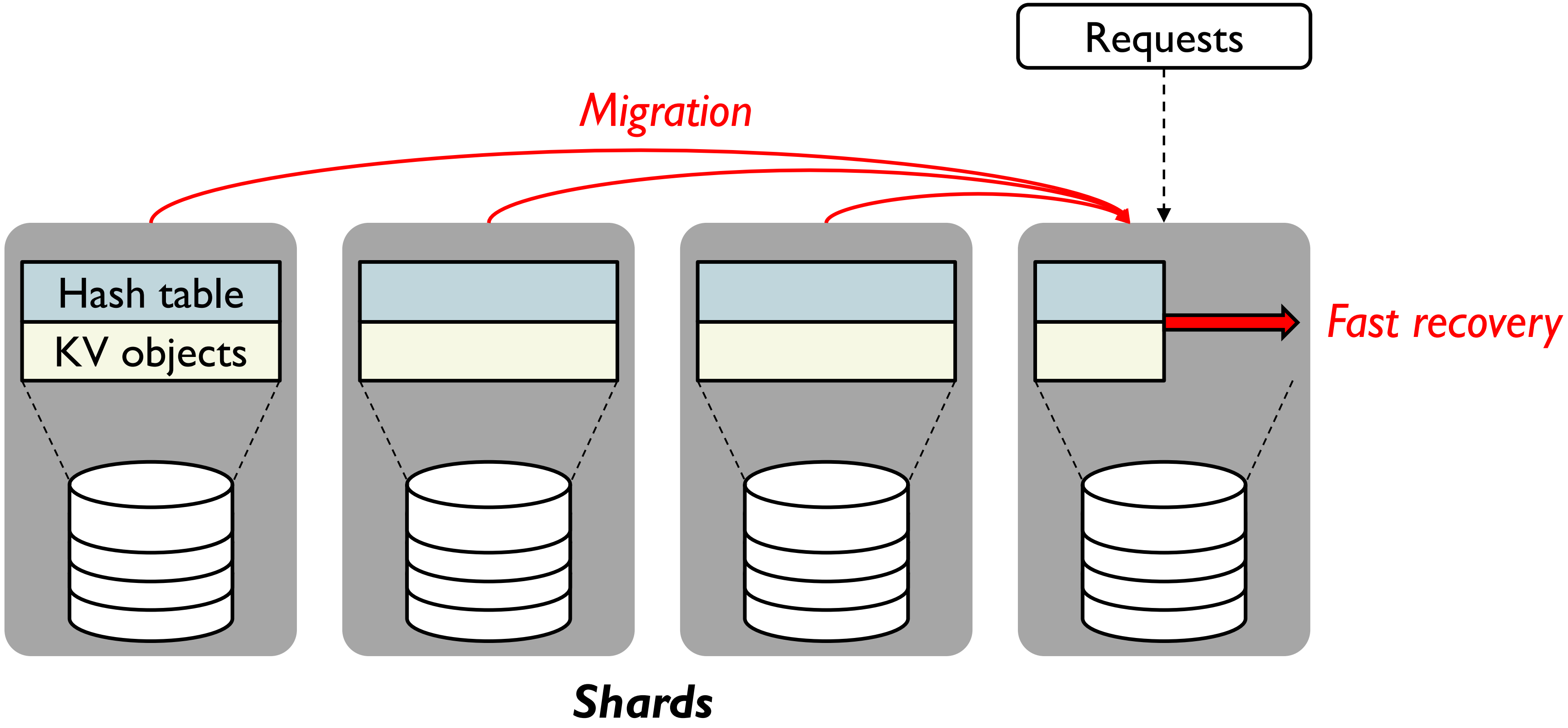
BigKV: Reactive Fault-tolerance with Sharding (cont.)

- Reactive fault-tolerance on SSD failures mitigate the loss penalty
  - Request forwarding – prevent further cache misses
  - Fast recovery – migrate objects to the replaced shard



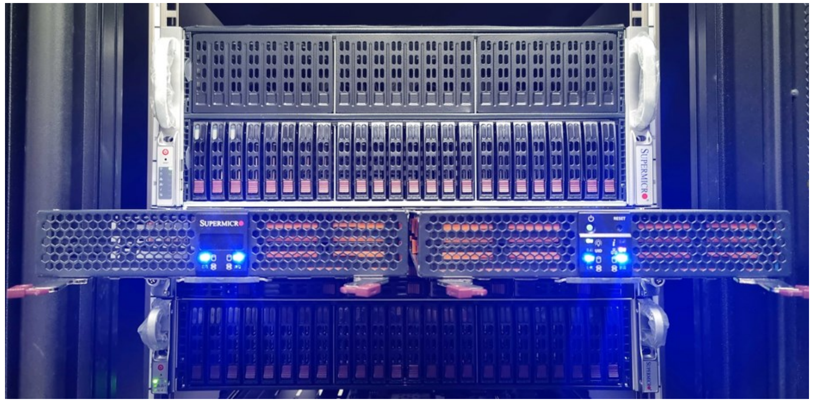
BigKV: Reactive Fault-tolerance with Sharding (cont.)

- Reactive fault-tolerance on SSD failures mitigate the loss penalty
  - Request forwarding – prevent further cache misses
  - Fast recovery – migrate objects to the replaced shard



Experimental Setup

- Implemented on an AFA machine
  - 64GB DRAM / 8x 3.84TB SSD
- Evaluation
  - Overall performance
  - Hit rate
  - Fault-tolerance
- Benchmarks
  - YCSB
  - Cache traces

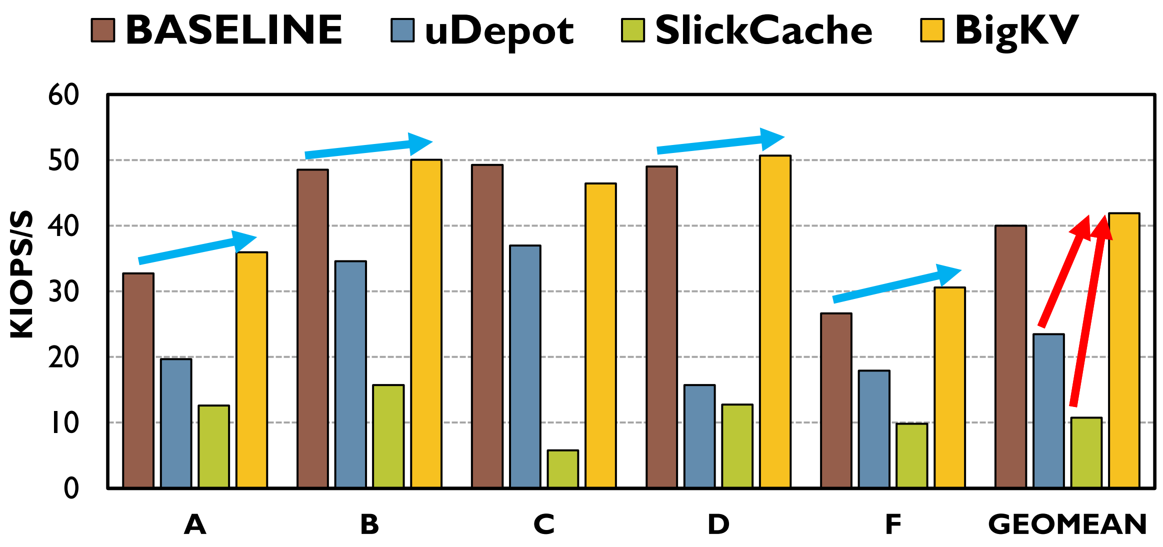


Results: Performance with YCSB

- Baseline: entire table in DRAM / the others: two-level hash table
- Outperform the existing SSD KV caches by removing I/O overheads

3.1x improvement

Outperform the baseline by ignoring FP collisions



Throughput

68% shorter latency

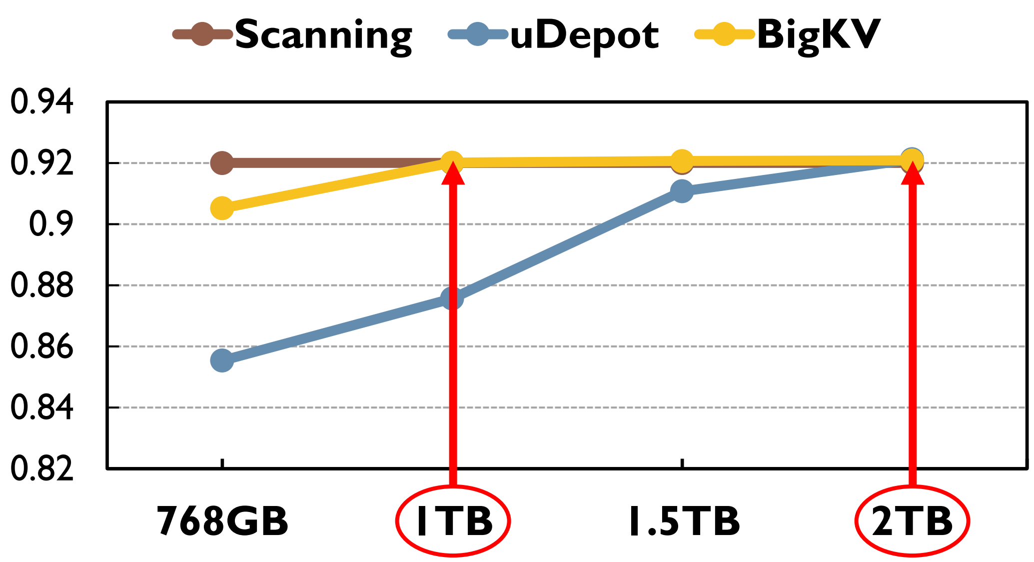
%	BASELINE	uDepot	SlickCache	BigKV
99	88	183	201	96 (49% shorter)
99.9	118	247	1,655	167 (73% shorter)
99.99	208	1,115	1,993	208 (86% shorter)

Lookup tail latency (us)

Results: Hit Rate with Traces

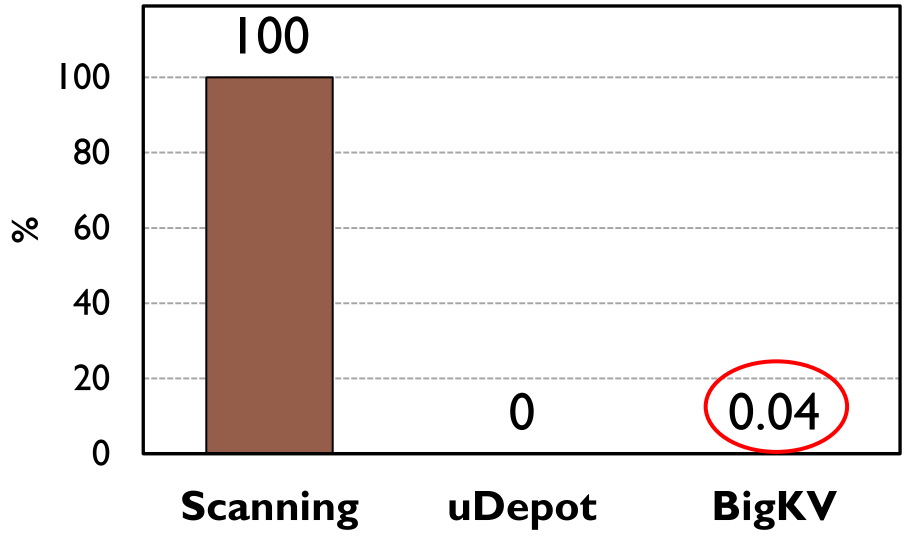
- Achieve the target hit rate with 2x smaller space with near-zero I/O overhead
  - Proactively remove expired objects

2x larger effective-capacity
(1TB BigKV == 2TB uDepot)



Hit rates over various capacities

Near-zero I/O overhead

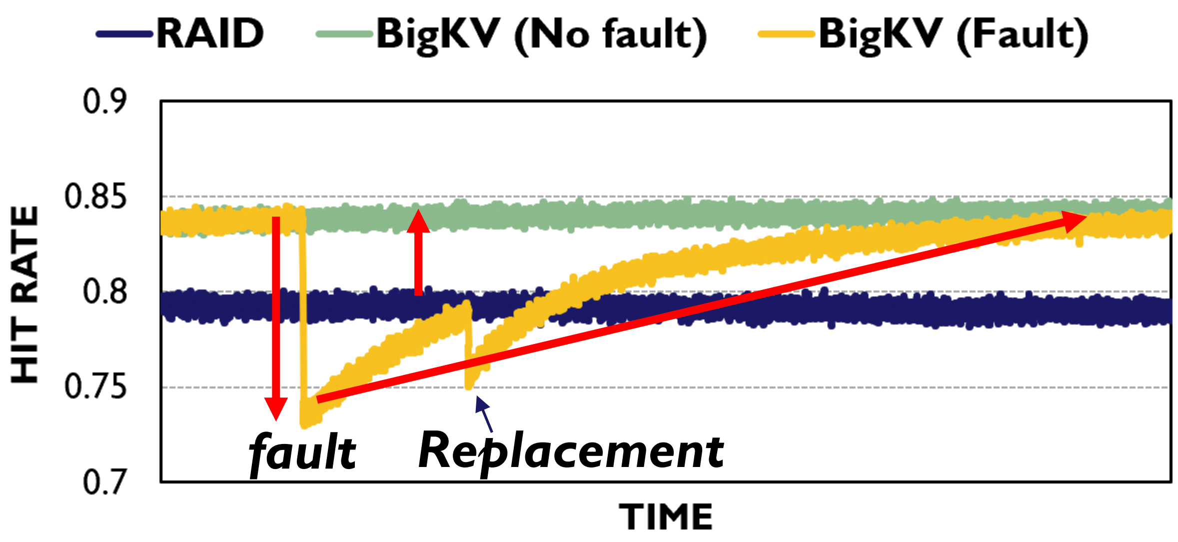


I/O overhead for reclaiming free space

Results: Fault-tolerance

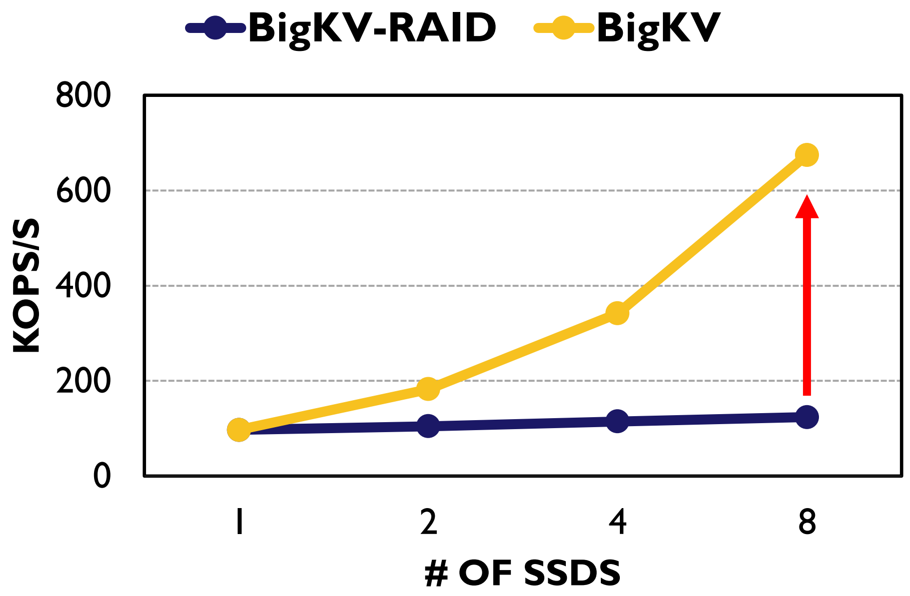
- Hit rate & performance improvement without parity overhead

***Improved hit rate* in normal cases**
Hit rate recovery



Hit rate

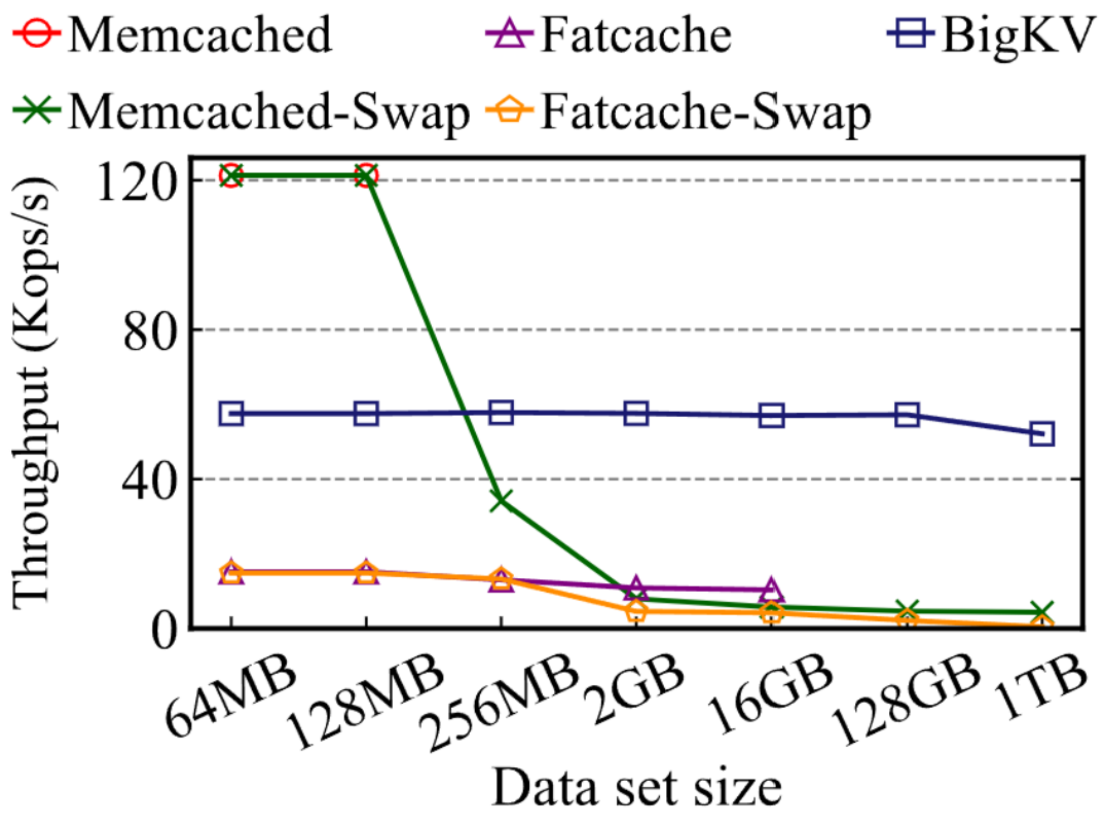
High scalability
***5.4x higher throughput* with 8 SSDs**



Performance scalability

Results: Comparison with Memcached and Fatcache

- Memcached & Fatcache stop working after metadata cannot be kept in DRAM
- Swap versions still work, but provide terrible throughput
- BigKV provides consistent throughput, regardless of input data set sizes



Conclusion

- Current: An AFA is a cost-effective alternative for caching large objects
- Motivation: Existing loss-prohibited techniques cannot fully leverage the AFA
- Solution: BigKV efficiently utilizes the AFA with loss-permitted techniques
  1. Collision-oblivious two-level hashing
  2. TTL-aware space management
  3. Reactive fault-tolerance
- Results
  - 3.1x higher throughput, 68% shorter latency
  - 2x larger effective-capacity
  - High scalability

Thank You !

Sungjin Lee (sungjin.lee@dgist.ac.kr)