BloomStore

Bloom-Filter based Memory-efficient Key-Value Store for Indexing of Data Deduplication

Guanlin Lu, EMC^2
Youngjin Nam, Daegu Univ., Korea
David H.C. Du, Univ. of Minnesota, Twin-cities

Overview of Key-Value Store

- Key-Value (KV) store
  - efficiently supports simple operations: Key lookup & KV pair insertion
  - replaces traditional relational DBs for its superior scalability & perf.
  - often implemented through an index structure, mapping Key → Value
- Popular management (index + storage) solution for large volume of records, with the applications like
  - social networks, online shopping, online multi-player gaming
  - data deduplication*

*Indexing & storing billions of KV pairs persistently, as well as providing high-throughput access
(e.g., each single node KV store offers >10,000 key lookups/sec)

Motivation(1/2)

EMC² where information lives®

- KV store in a deduplication system should provide high access throughput (> 10,000 key lookups/sec)

Scalability challenge: available memory space limits the maximum number of stored KV pairs

Q1: Can we overcome the space limitation?

Motivation(2/2)

EMC² where information lives®

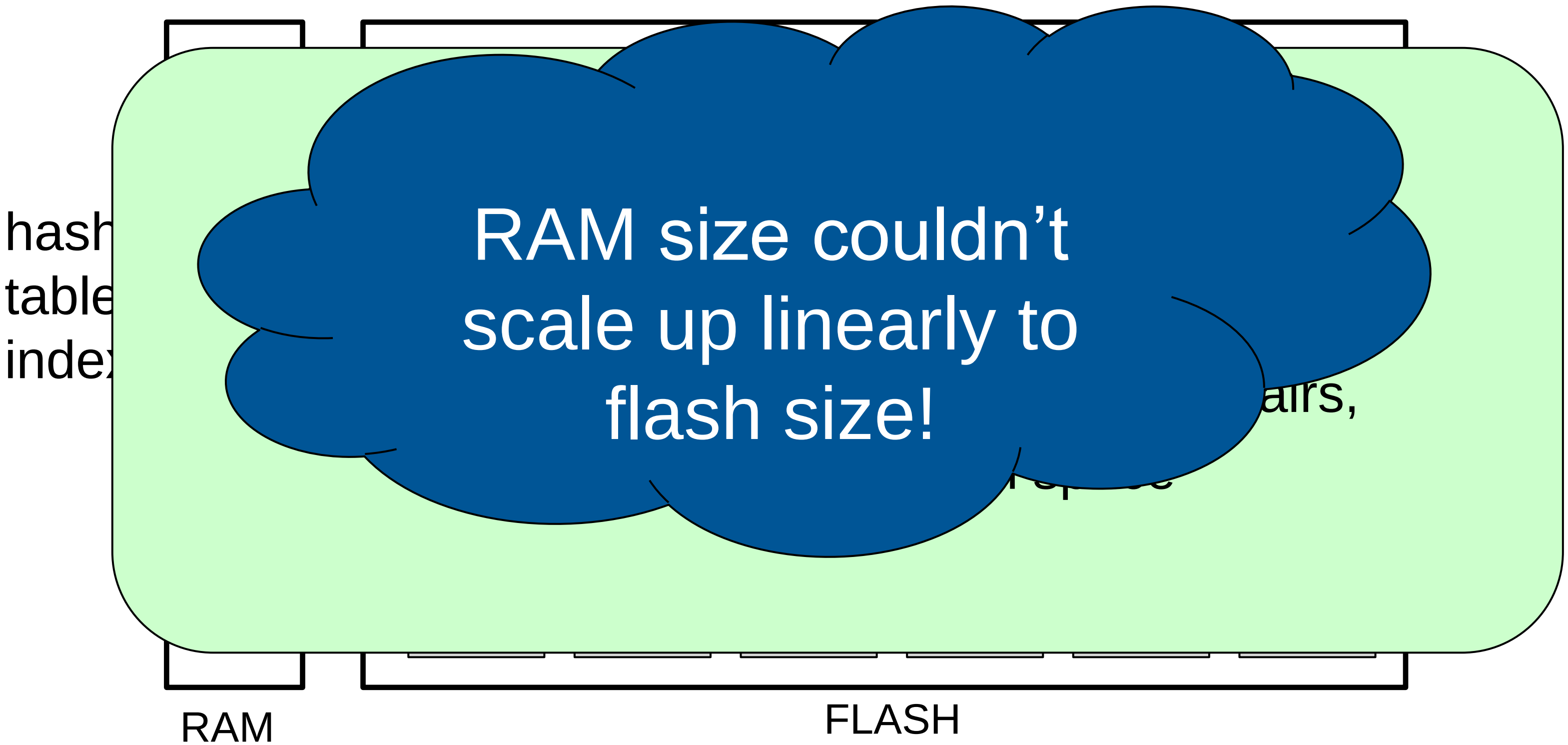
- To meet high throughput demand, the performance of index access and KV pair (data) access is critical
  - index access : search the KV pair associated with a given “key”
  - KV pair access: get/put the actual KV pair

Using in-RAM index structure can only address index access performance demand

Q2: How to optimize both index & KV pair accesses in KV Store?

Existing Approach to Speed up Index & KV pair Accesses

- Store KV pairs into SSD for faster data access
- Maintain the index structure in RAM to map each key to its KV pair on SSD



Handling Scalability Challenge with SSD

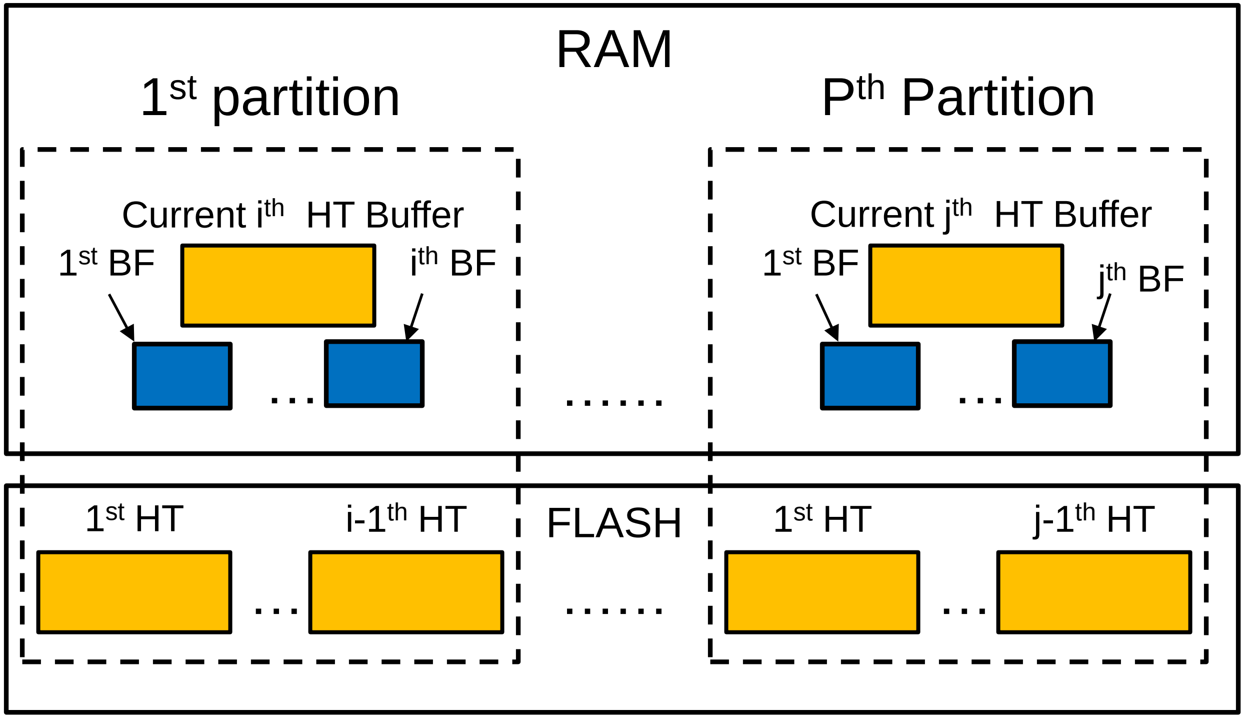
- Keep the minimum index structure in RAM, while storing the rest of the index structure in SSD

- On-flash Index structure should be designed carefully:
  - read/write by page
  - write data only into clean (erased) pages
  - sequential write is multiple times faster than random write
  - erase by block (much slower than read/write)
  - overwrite is inefficient
  - a limited erase count per cell (10K – 100K)

BufferHash [Anand'10]

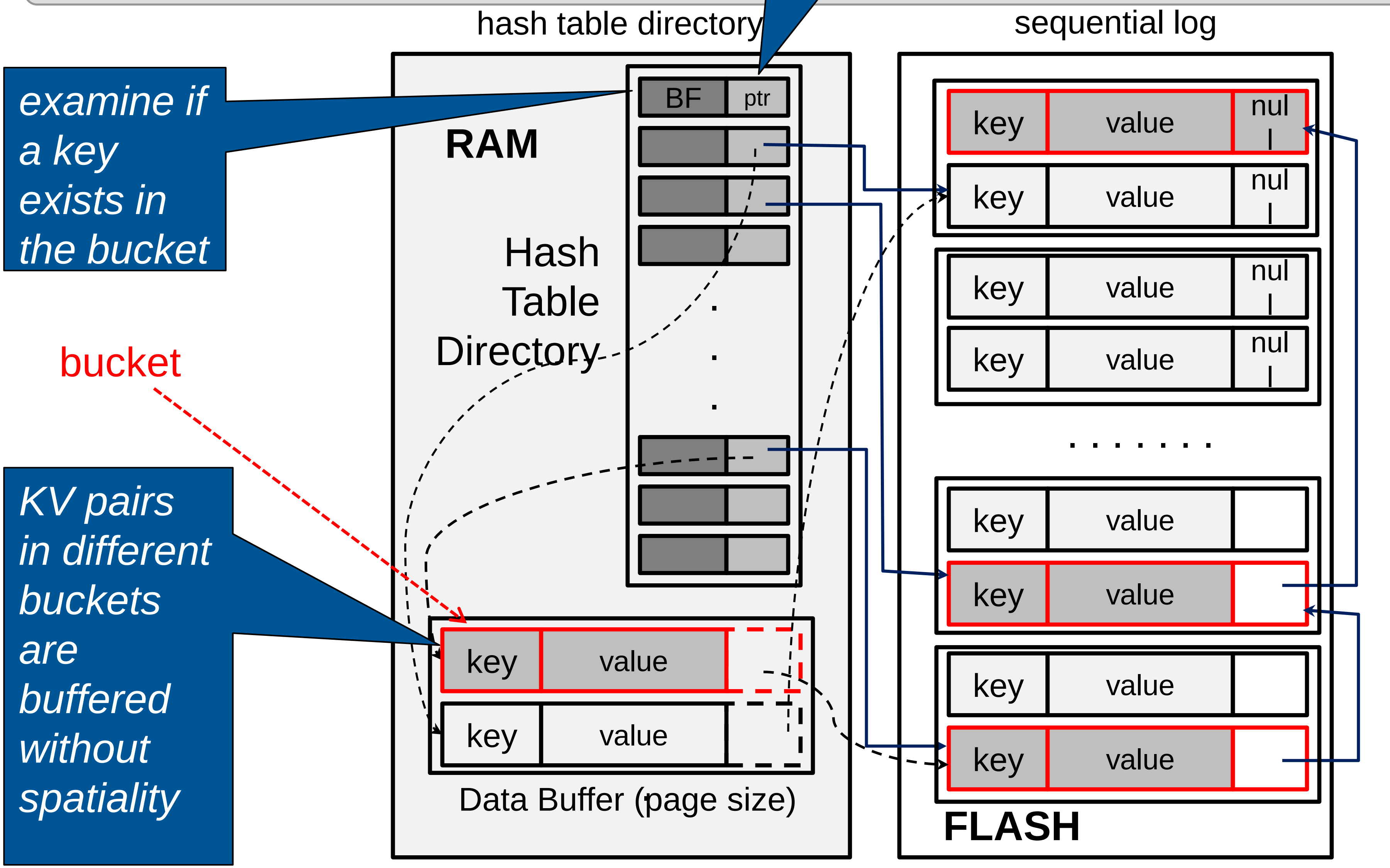
EMC² where information lives®

- Keeps all BFs & the current HT in RAM, while keeping other HTs in flash



use hash table both as index structure and data container for KV pairs

SkimpyStash [Debnath'11]

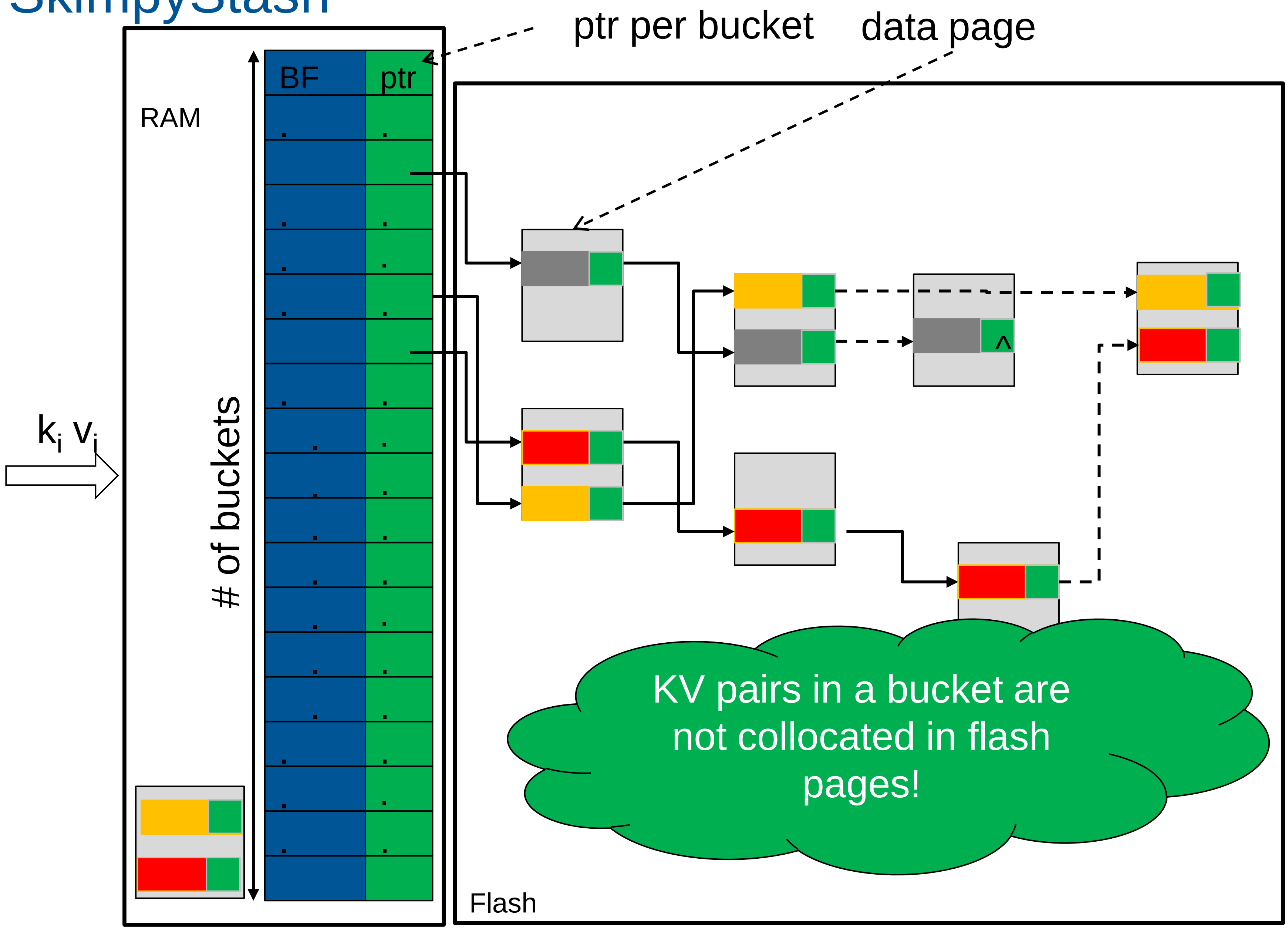


Limitation of SkimpyStash

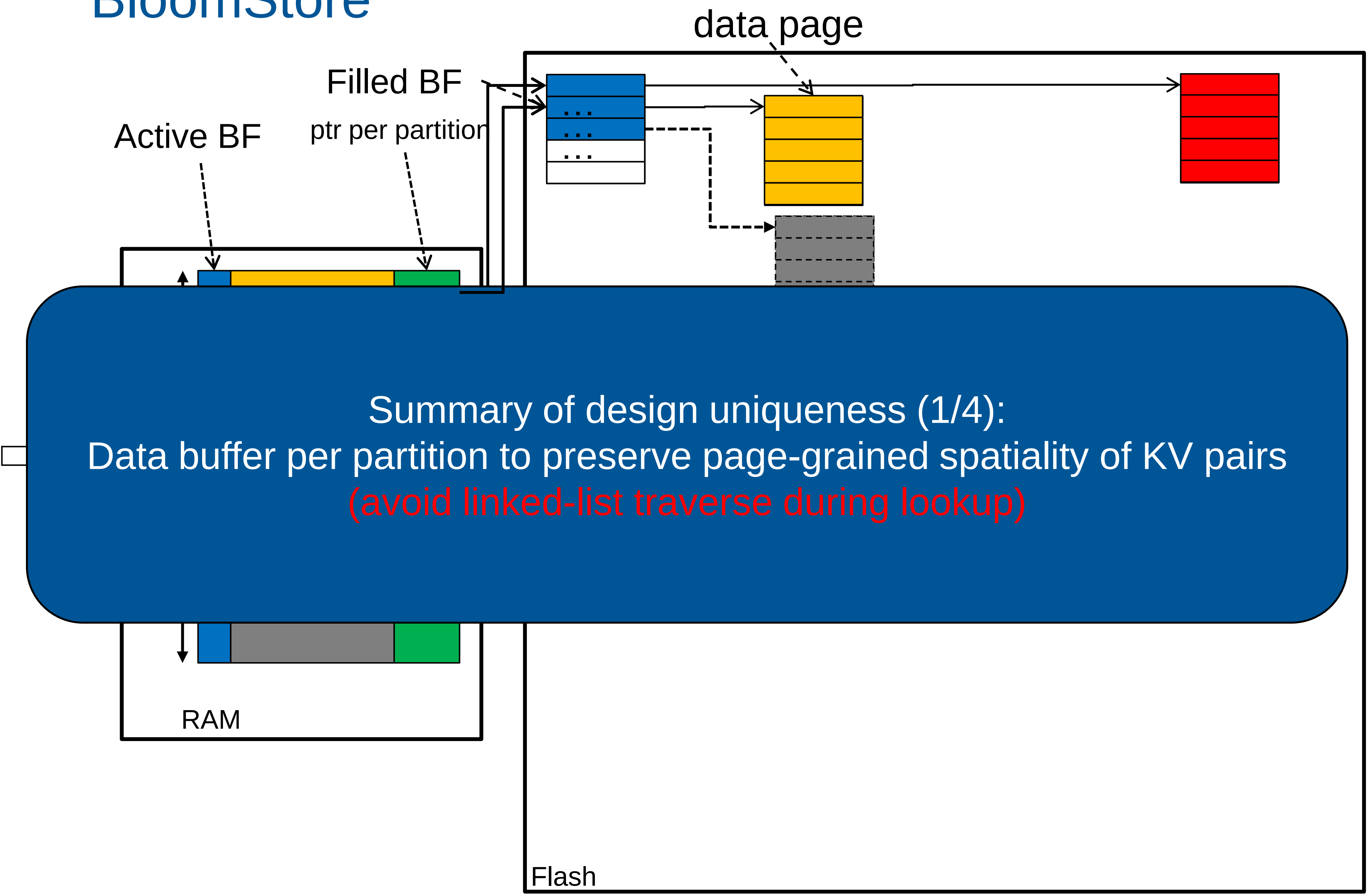
EMC² where information lives®

- Each false positive error causes all KV pairs in the corresponding bucket to be searched “in vain”
  - to improve lookup performance, they have to either increase the BF size, or reduce the bucket length → both increase the RAM usage!
  - 1-byte in-RAM BF footprint per key
  - RAM overhead per key = 1+ 4/(avg_bucket_length) bytes
- Key lookup time increases linearly as the bucket length grows
  - avg # of flash page reads in each key lookup operation equals to half of the (average) bucket length

SkimpyStash

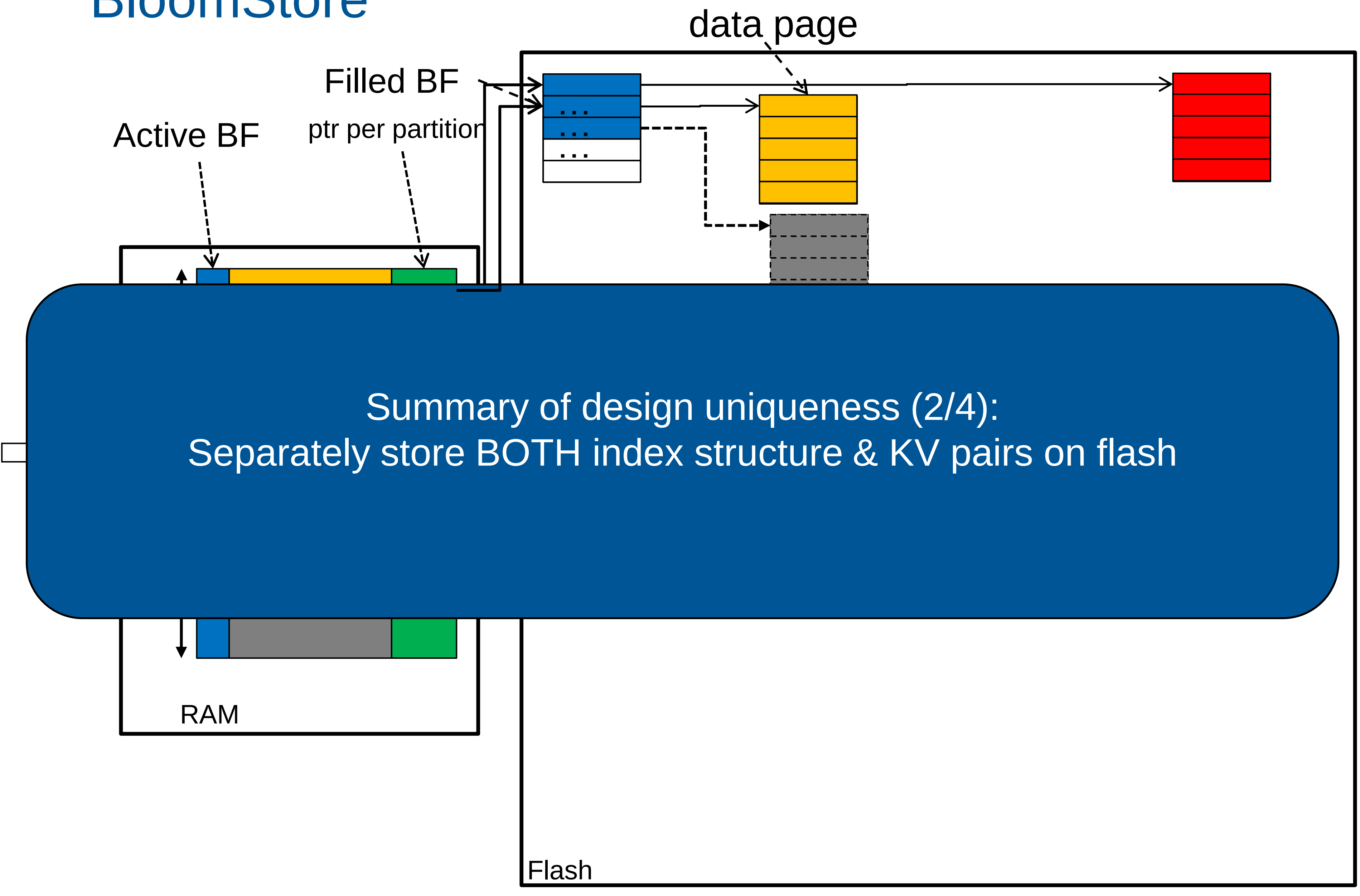


BloomStore



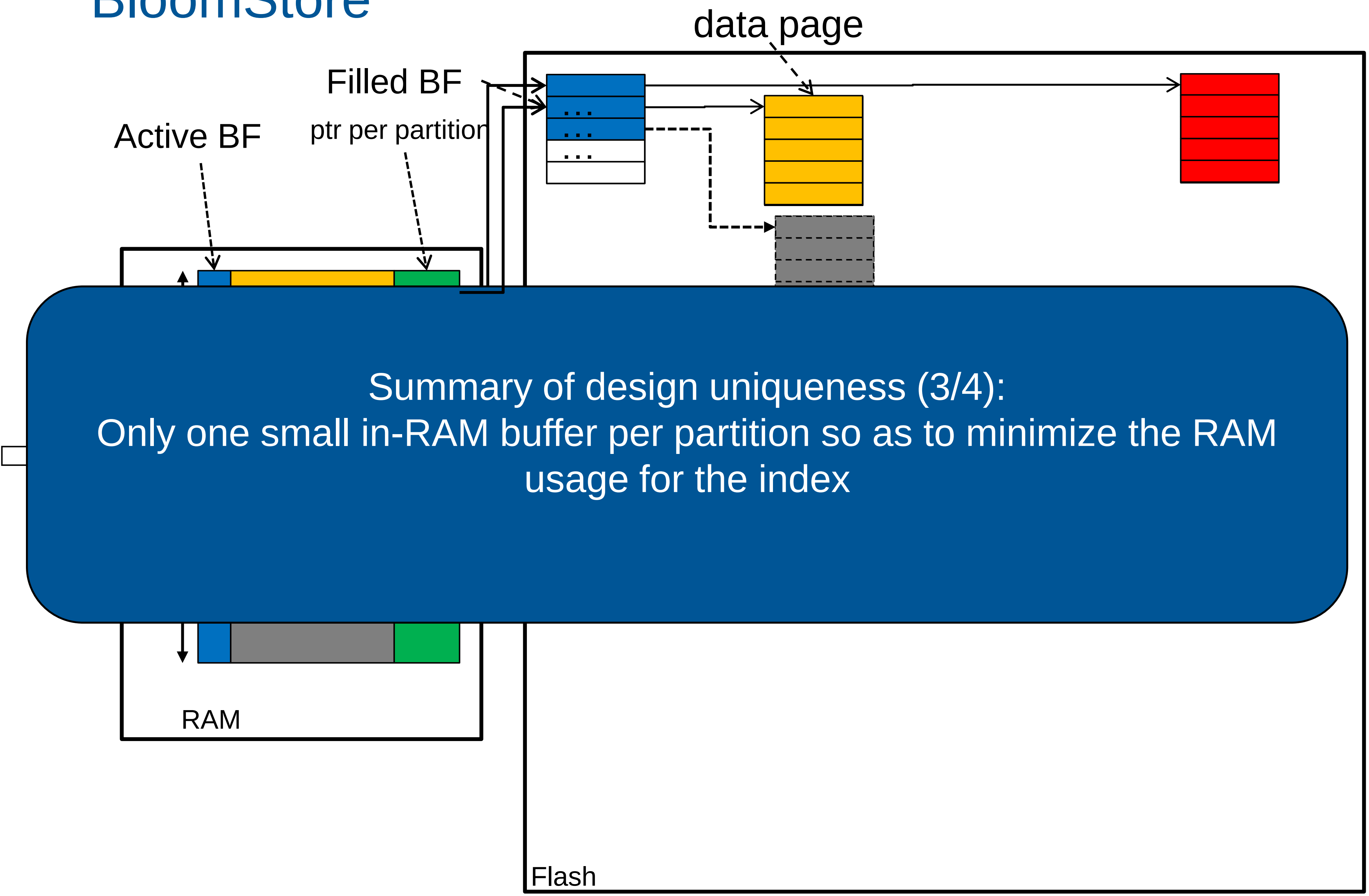
Summary of design uniqueness (1/4):
Data buffer per partition to preserve page-grained spatiality of KV pairs
(avoid linked-list traverse during lookup)

BloomStore



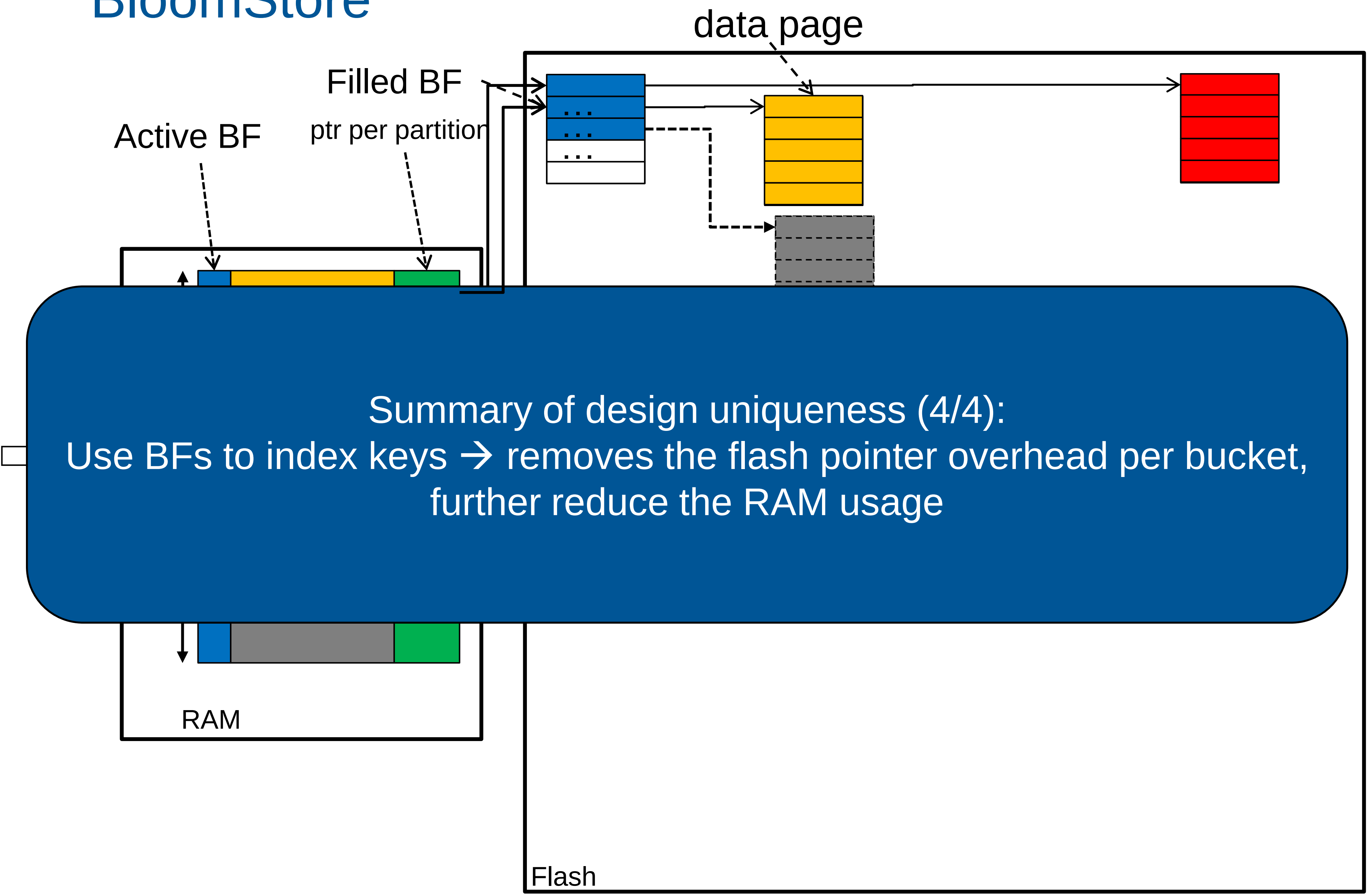
Summary of design uniqueness (2/4):
Separately store BOTH index structure & KV pairs on flash

BloomStore



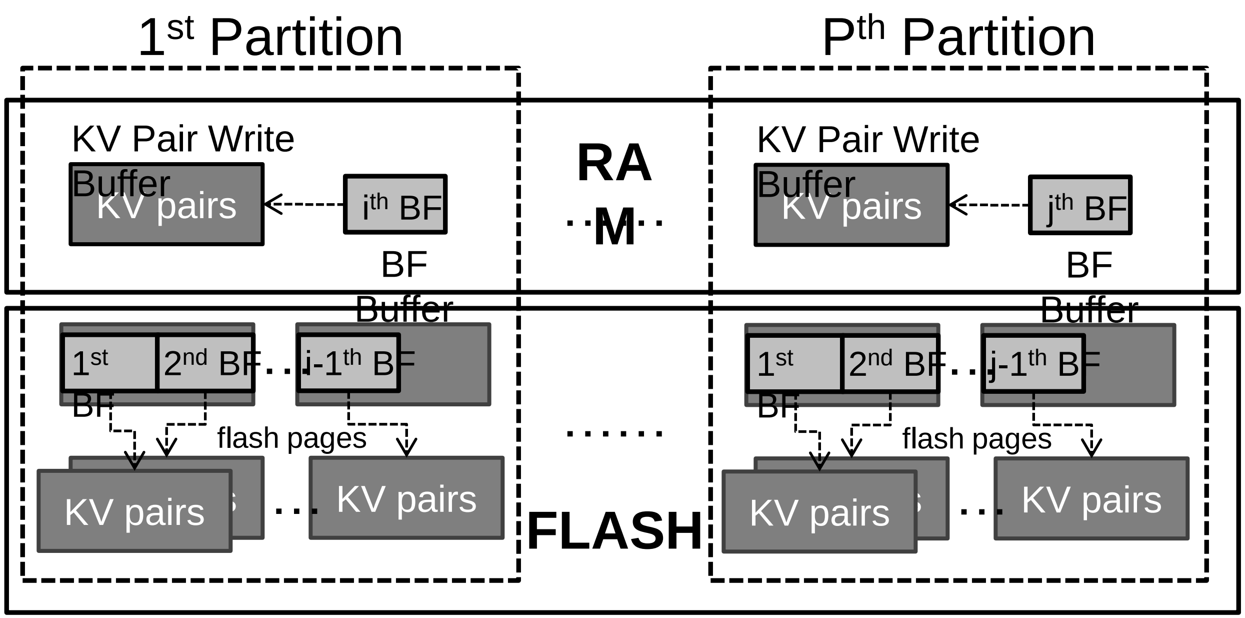
Summary of design uniqueness (3/4):
Only one small in-RAM buffer per partition so as to minimize the RAM usage for the index

BloomStore



Summary of design uniqueness (4/4):
Use BFs to index keys → removes the flash pointer overhead per bucket, further reduce the RAM usage

BloomStore Architecture



BloomStore – Performance Enhancements

EMC² where information lives®

- Multi-BF Buffering
  - Each BloomStore instance holds the active BF plus a number of BFs whose data flash pages of KV pairs have been already written into the flash in its BF buffer.
- Pre-filter
  - Why need a pre-filter?
  - Solution: keeping a fix-sized pre-filter in RAM to filter out large portion of lookups for the nonexistent keys before reading a BF chain from the flash.
    - Use a Bloom Filter as our pre-filter for (1) BF is free of false negative errors; (2) with fairly small memory footprint (4 bits/key), the BF is able to identify and filter out a significant amount of non-existent keys.

Experimental Setup

- Two I/O traces: backup (linux) & primary storage (vx)

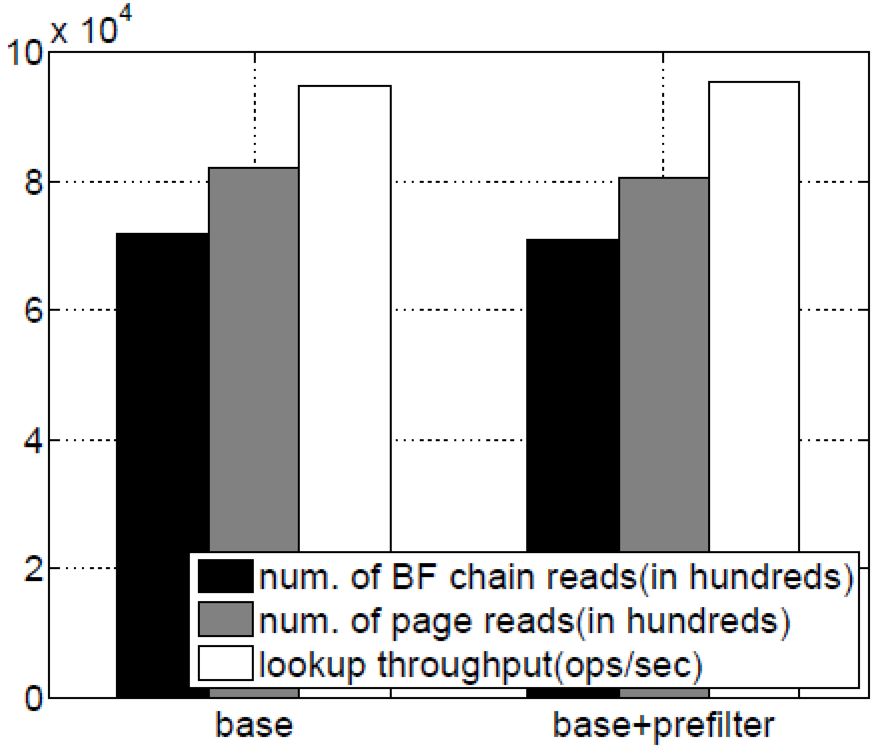
Workload name	Lookup & insert operations #	Lookup:insert ratio	Key/value size (byte)
Linux	12,427,697	4.1 : 1	20/44
vx	14,628,873	1.6 : 1	20/44

- BloomStore settings:
  - partition size: 96 flash pages per partition
  - BF chain size: 10KB (for vx) and 12KB (for Linux)

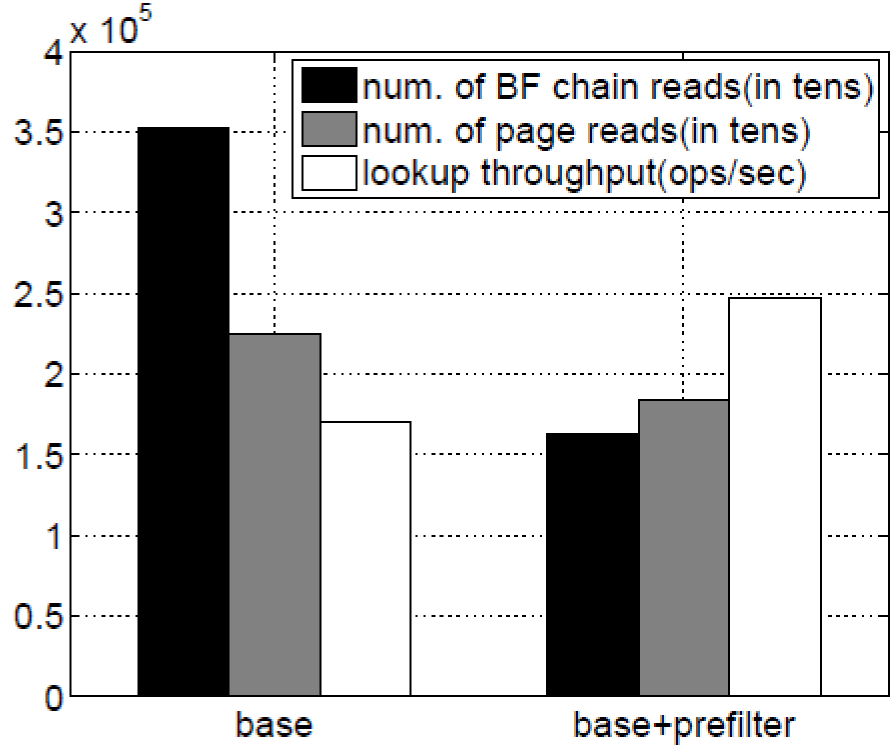
Experimental Result: Impact of Pre-filter

RAM usage decomposition	BF buffer	prefilter overhead	KV pair write buffer
base	1,302	0	1,648
base+prefilter	807	495	1,648

RAM usage decomposition	BF buffer	prefilter overhead	KV pair write buffer
base	3,066	0	3,840
base+prefilter	186	2,880	3,840



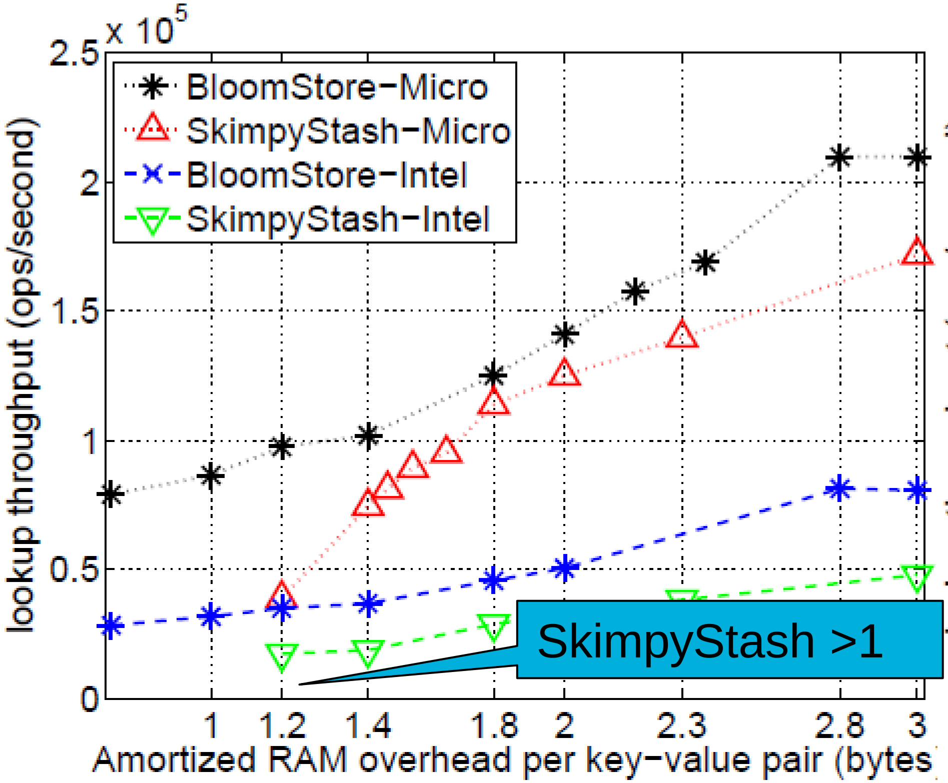
Backup (linux)



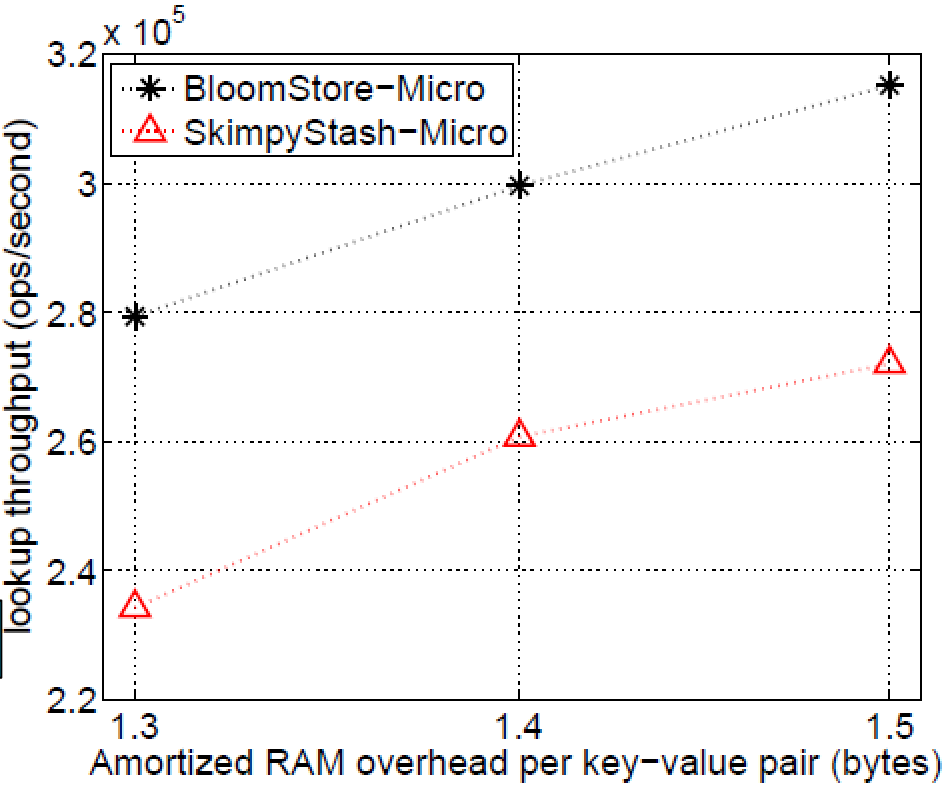
Primary storage (vx)

Experimental Result: Key Lookup T-put

EMC² where information lives®



Backup (linux)



Primary storage (vx)

Summary

EMC² where information lives®

- We designed BloomStore, a novel KV store on flash
  - utilizes very limited RAM space combined with much large flash space to support high throughput, low latency lookup/insertion ops.
  - achieves the design goal of sub-byte-level RAM overhead per key-value pair, which is significantly lower than other designs
- Compared with the state-of-the-art (SkimpyStash)
- Achieved better key lookup performance with lower RAM usage on backup & primary dedupe workloads

EMC²

where information lives®

Thanks & Questions?