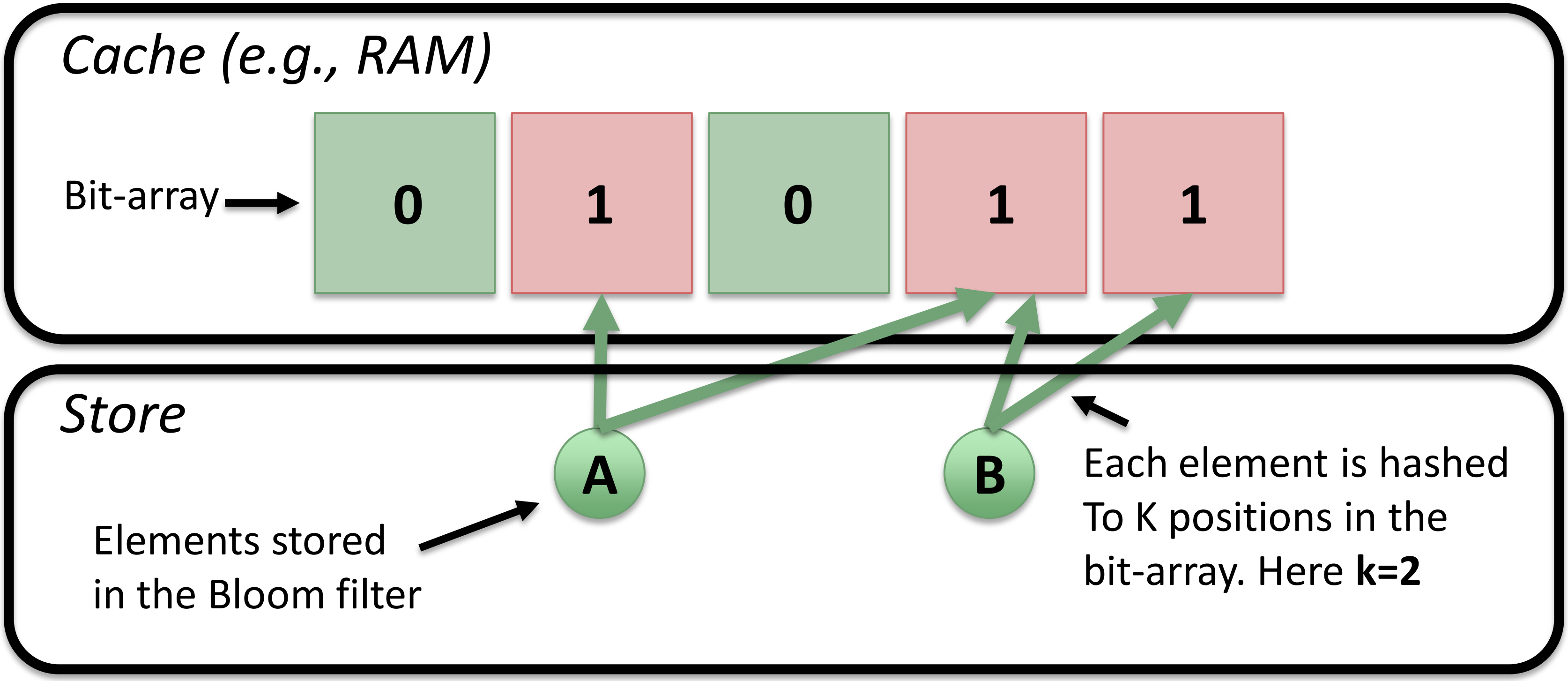
Don’t Thrash: How to Cache Your Hash in Flash

M.A. Bender, M. Farach-Colton, R. Johnson,
B.C. Kuszmaul, D. Medjedovic, **P. Montes**,
P. Shetty, **R. P. Spillane**, E. Zadok

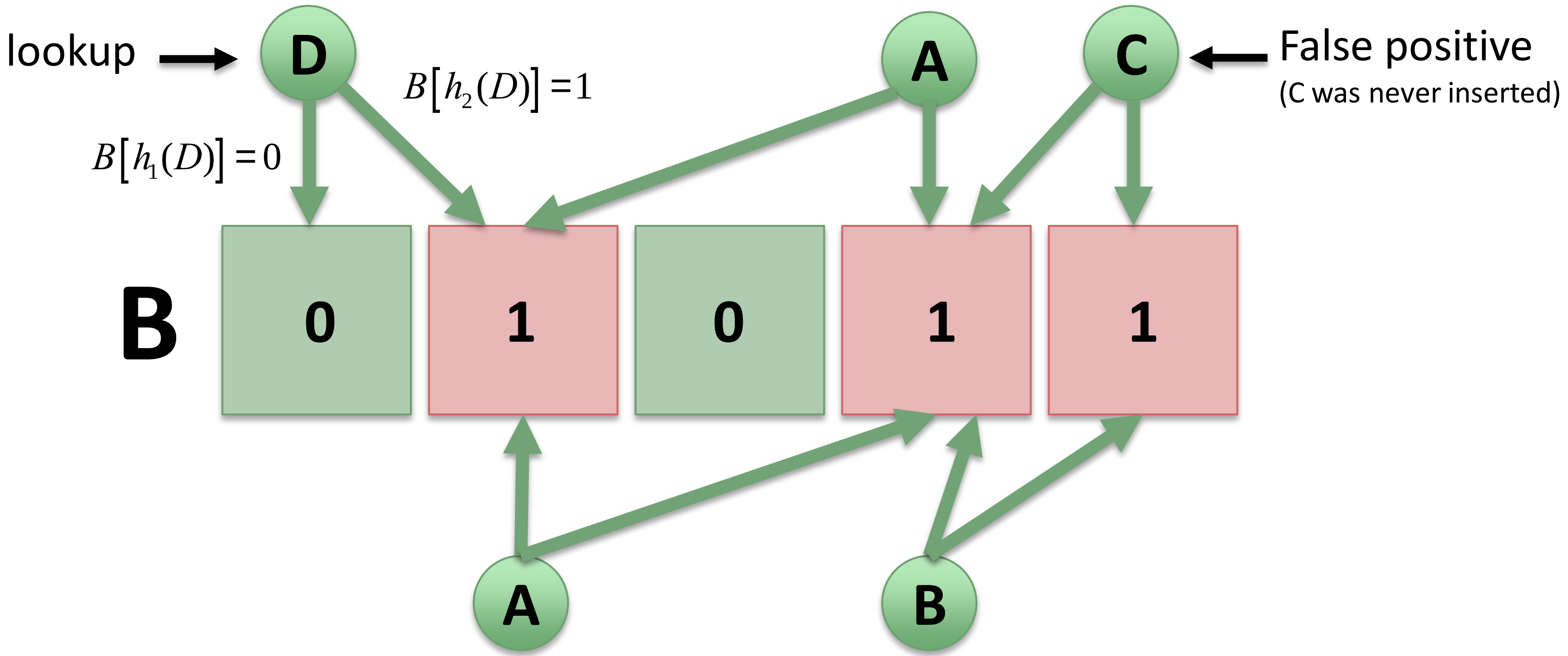
Stony Brook U., Rutgers U., MIT, TokuTek

Bloom Filter


Cache (e.g., RAM)
Bit-array
0
1
0
1
1
Store
A
B
Elements stored in the Bloom filter
Each element is hashed To K positions in the bit-array. Here **k=2**

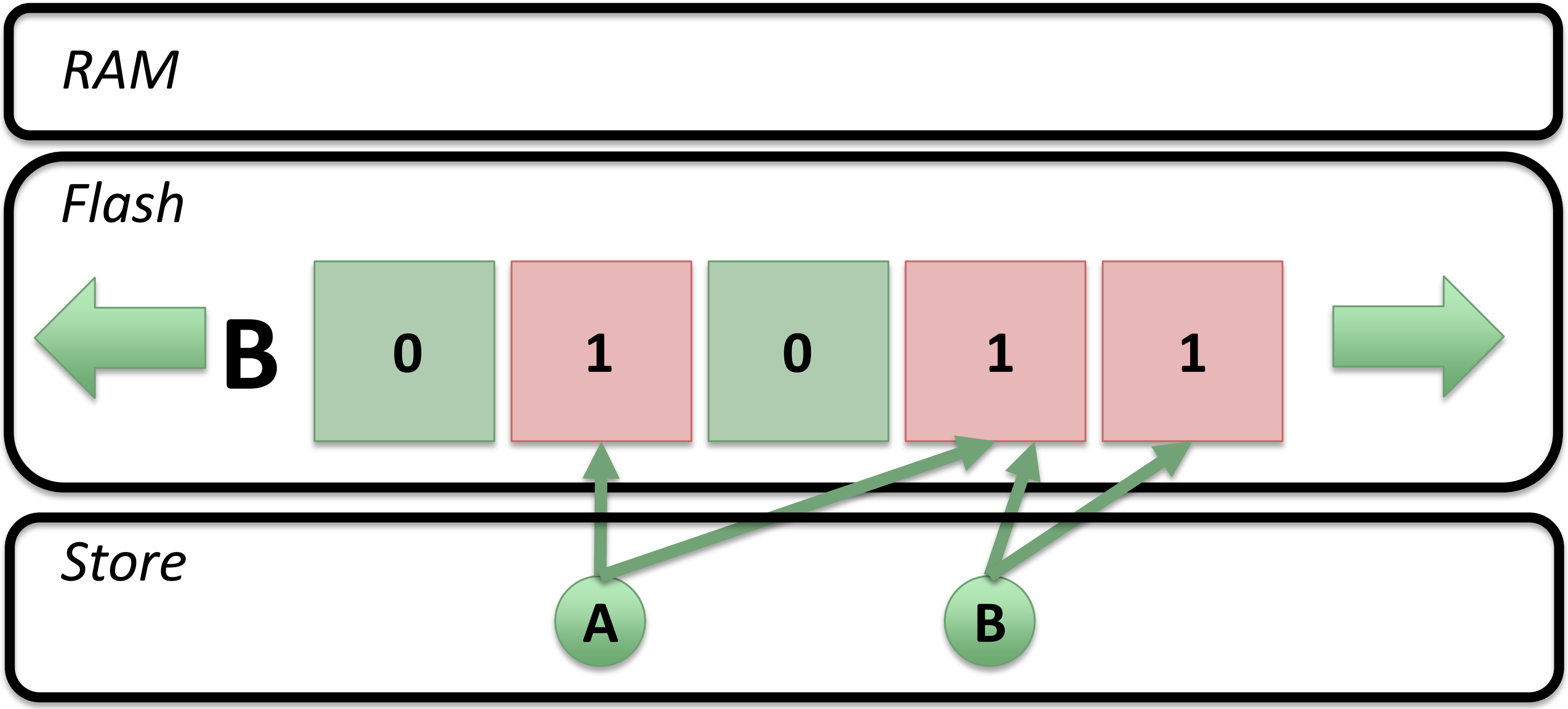
- A Bloom filter is a bit-array + k hash functions
- Storing a few bits per element lets the BF stay in RAM, even as the elements are too large

Bloom Filter Lookups & False Positives



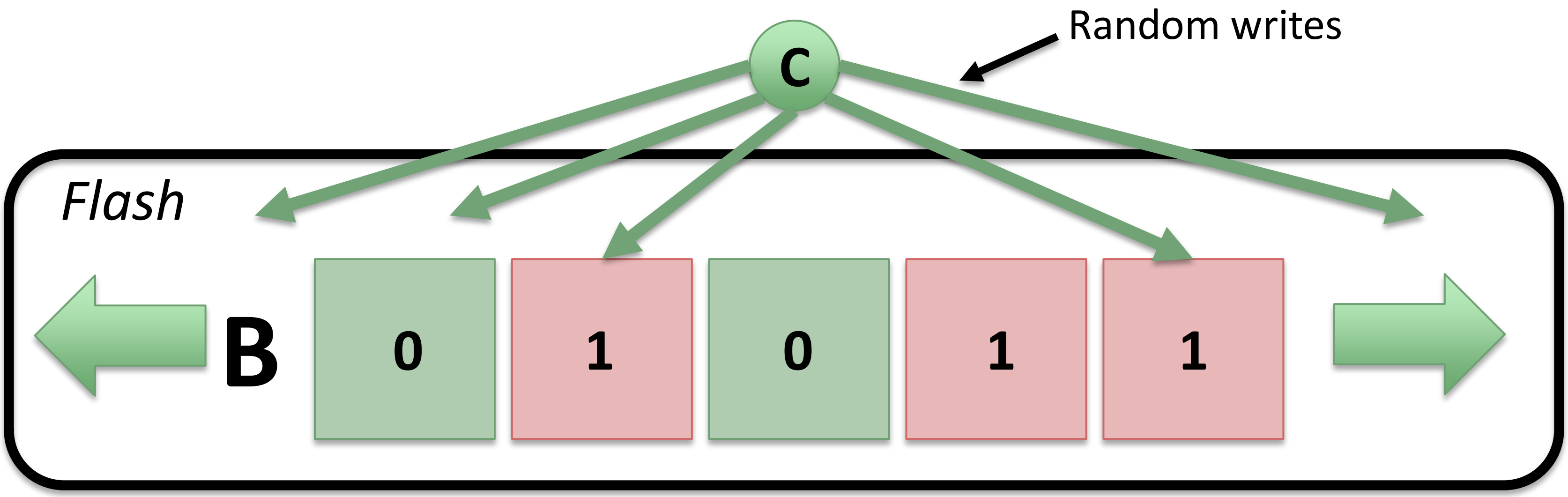
- False positives unlikely, $p_{FP}\left(x\right) \gg \left(1- e^{kn/m}\right)^{k}$
- No false negatives (no means no)
- Allowing false positives is what keeps the BF small

Flash



- Bigger & cheaper than RAM, faster than disk
- 8TB of 512B keys needs 16GB of RAM for a ~1% BF
- Flash is a good place to cheaply store large BFs

Thrashing

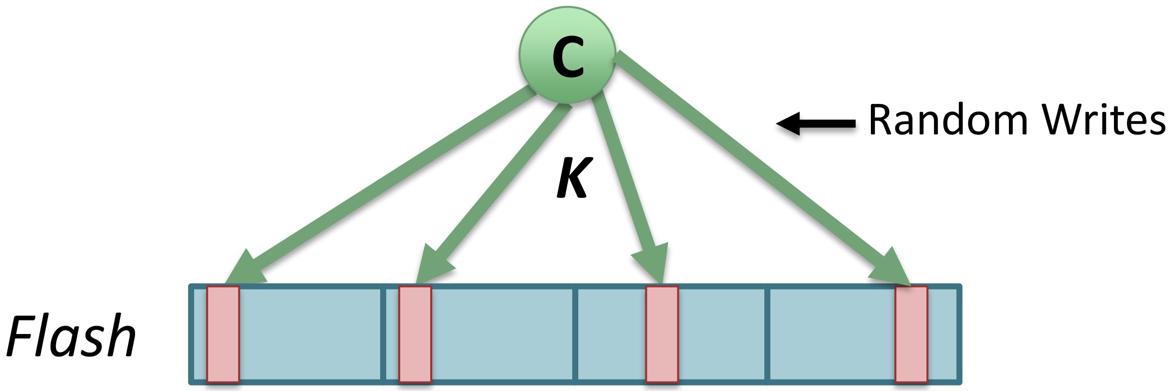


- Setting random bits to 1 causes random writes
- OK in RAM, not in Flash

Summary of Our Results

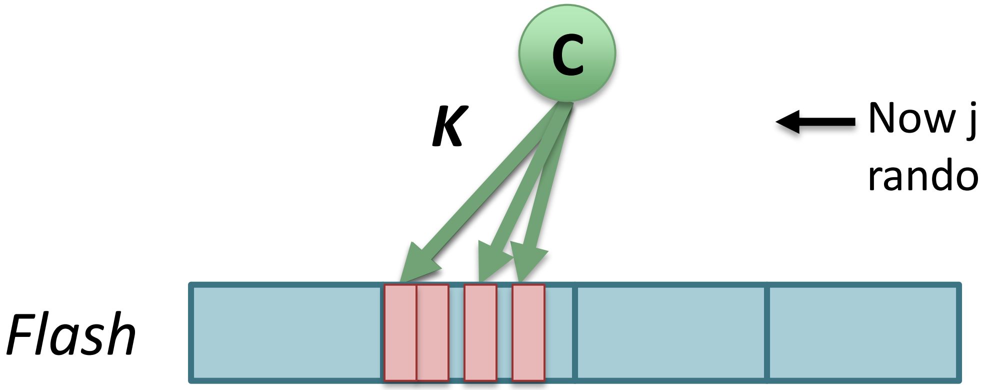
- **Cascade Filter** (CF), a BF replacement opt. for fast inserts on Flash
- Our performance
  - We do 670,000 inserts/sec (40x of other variants)
  - We do 530 lookups/sec (1/3x of other variants)
- We use **Quotient Filters** (QF) instead of Bloom Filters
  - They have better access locality
  - You can efficiently merge two QFs into a larger QF (w/ same FP rate)
- We use **merging techniques** to compose multiple QFs into a CF

Thrashing is the Problem



- Every insert, you write to K Flash pages
- Expensive to write to a Flash page
- We can't do fast insertions without working around this issue

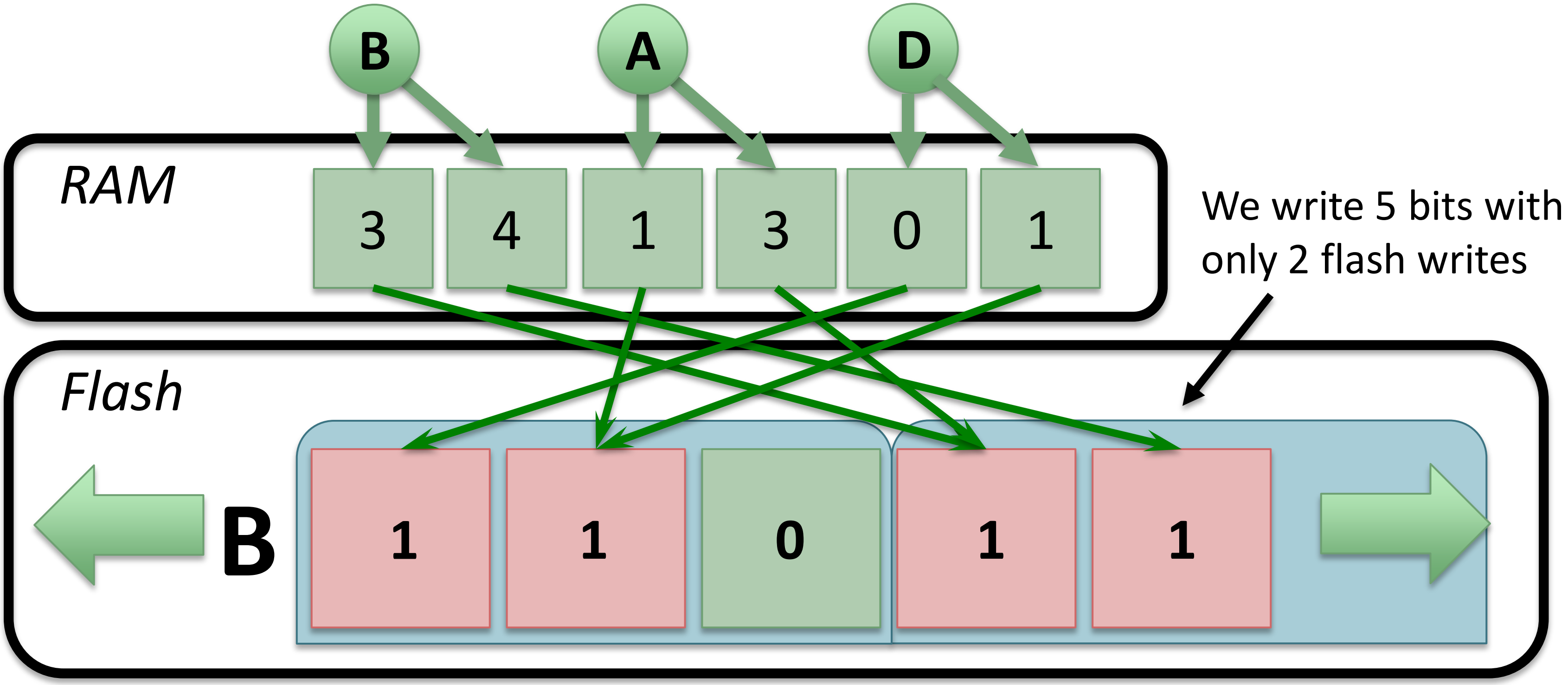
Shaving off K



Now just one random write, not K

- Now you only write one block for each insert instead of K blocks
- Two-step hash [Canim et. al., 2010]
- This helps a little

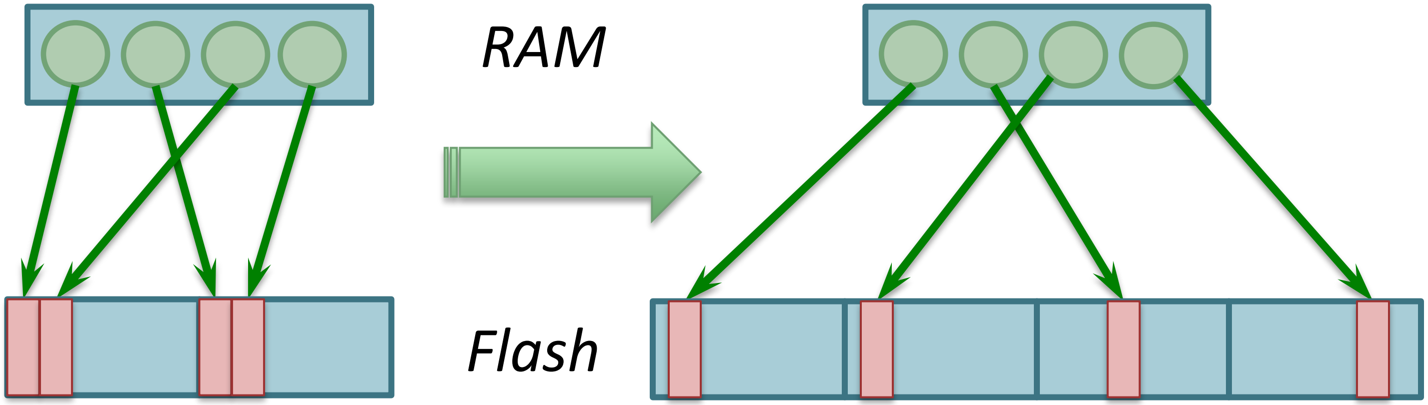
Queue Writes



We write 5 bits with only 2 flash writes

- This helps **a lot** [Canim et. al. 2010]
- Buffering gives bit-flips a chance to piggy-back
- How others have cached hashes in Flashes

We Need Help



- Buffering works when the queue is large
- Small queues insert ~1 element per flash write
- We're interested in large datasets, and fast insertions (i.e., when buffering doesn't work)

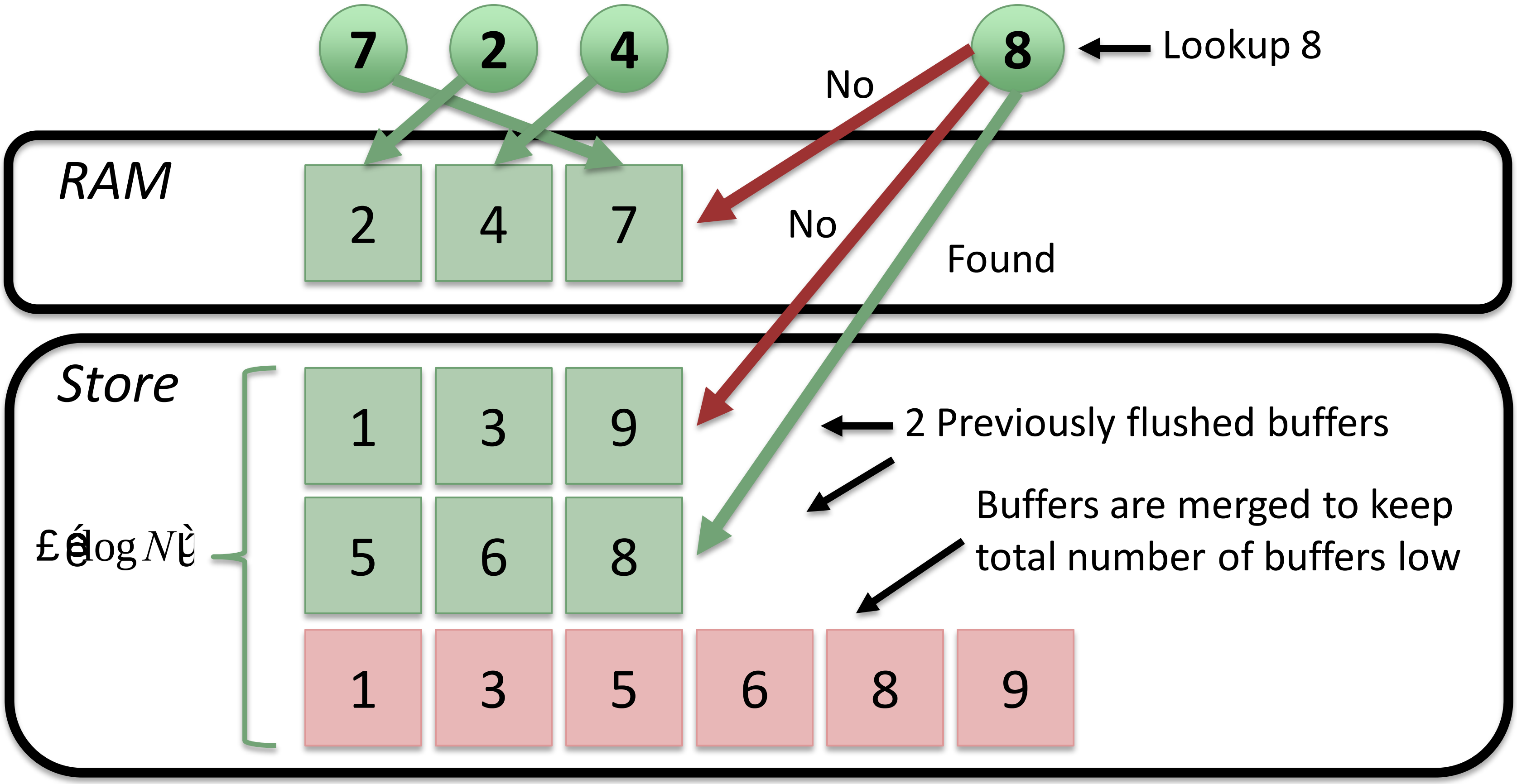
An Important Problem

- Many companies optimize their DBs for large data-sets and fast inserts
  - Bai-Du Hypertable
  - Facebook Cassandra
  - Google BigTable
  - TokuTek TokuDB
  - Yahoo! HBase
  - … and more!
- Scaling the trusty Bloom Filter to Flash would be a powerful tool for tackling these problems

Several data structures avoid RWs

- A list of the most common methods
  - Buffered Repository Trees
  - Cassandra
  - Cache Oblivious Look-ahead Arrays
  - Log-structured Merge Trees
  - …and more
- We can try to adapt the general method many of these structures use

The General Method


7
2
4
8
Lookup 8
No
RAM
2
4
7
No
Found
Store
1
3
9
2 Previously flushed buffers
£ éêlog Nùú
5
6
8
Buffers are merged to keep total number of buffers low
1
3
5
6
8
9

- Supports deletes
- Composed of many sorted lists
- We can use this technique to avoid random writes

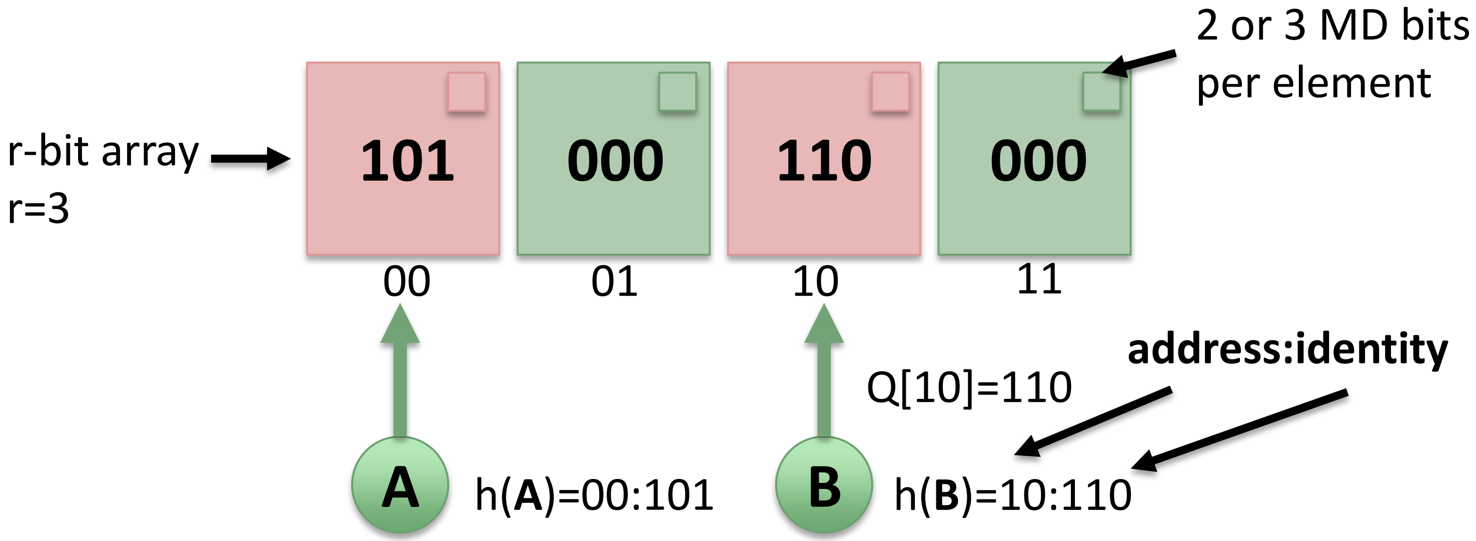
Problem: Elements not Bits

- This method is used with sorted lists of elements, **not Bloom filters**
- We need a data structure that
  - Supports insert + lookup
  - Is as space efficient as a Bloom filter
  - **Can be merged on Flash like a sorted list of elements**
  - ★ Bonus: supports always-working deletes
  - ★ Bonus: faster than BFs

Our Proposal: Quotient Filters

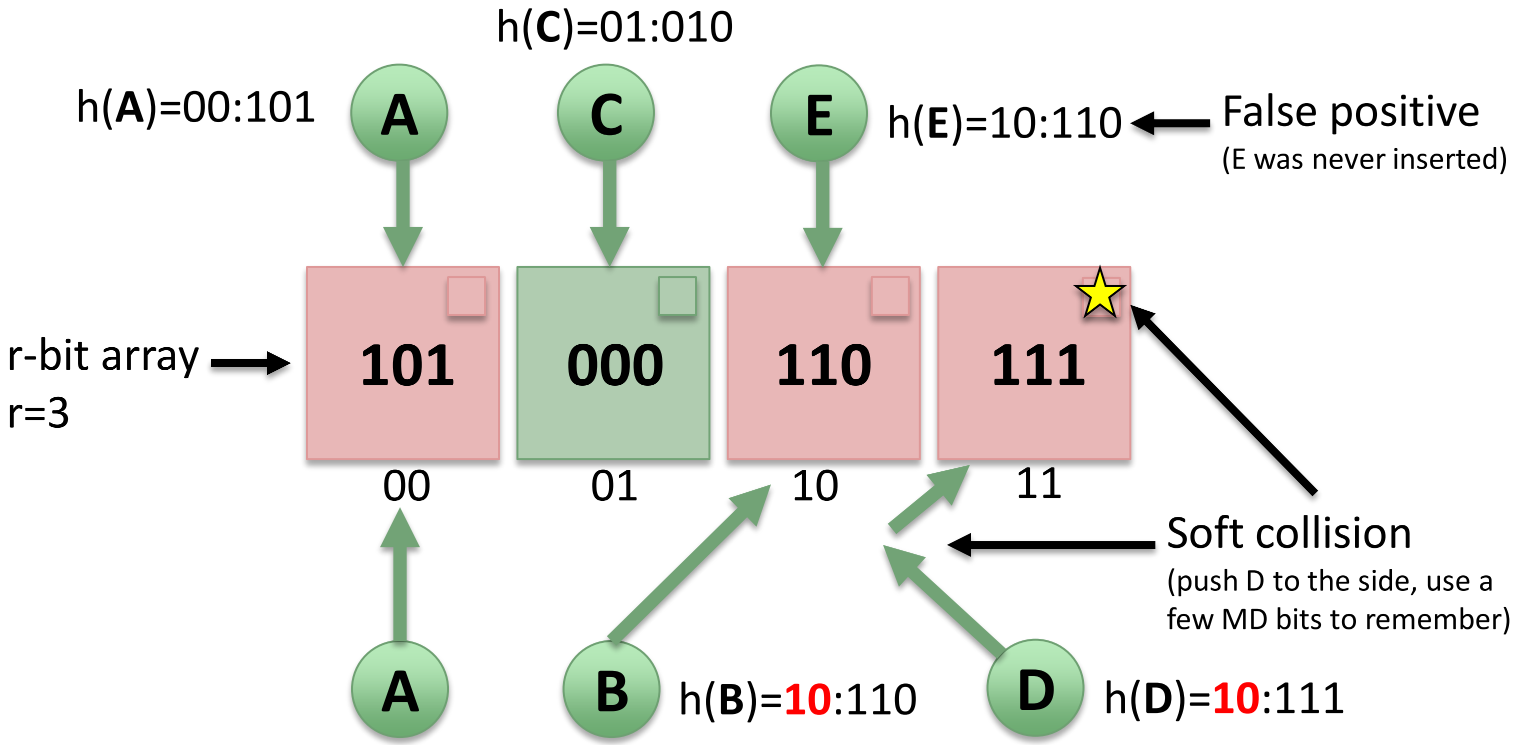
- Supports insert + lookup
- Compact like a Bloom filter
- **Two QFs can be merged into a larger QF**
- Supports always-working deletes
- Faster
- We can use this alternative to replace the sorted lists of elements in a write-opt. method

A Quotient Filter



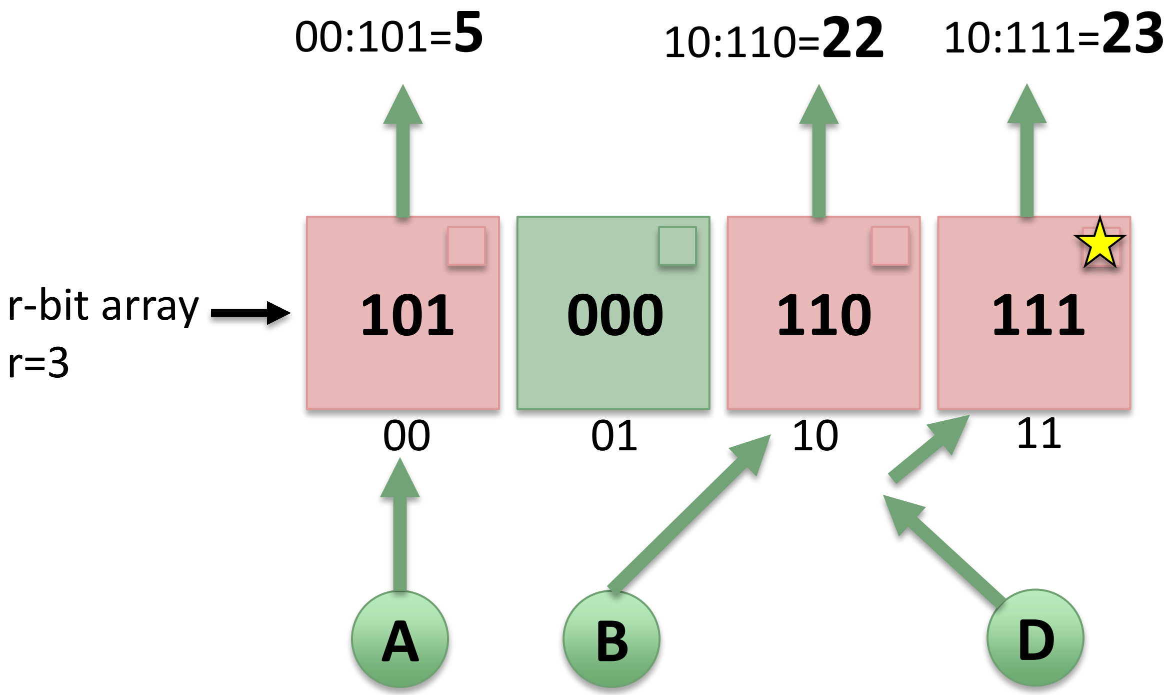
- fingerprints + **quotienting** to save space
- fingerprint: p-bit hash (p=5)
- Compact, only stores r+MD bits per element

A Quotient Filter



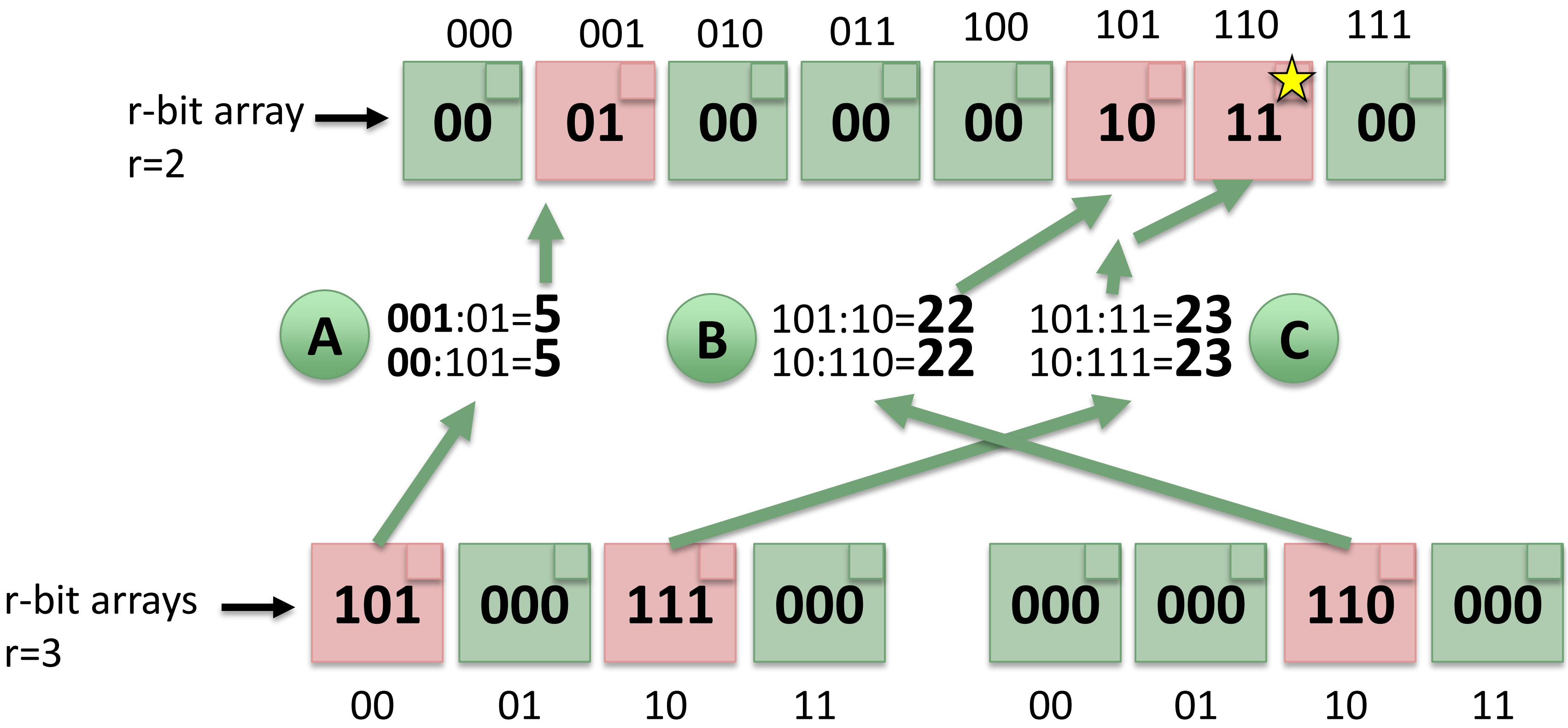
- False positive: fingerprint collision
- $p_{FP}(x) £ a \, {}^{1}/_{2^r}$, $size = a^{-1}(r+MD)2^q$, or ~1.2x a BF for ~0.1% FP-rate
- Quotient Filters also remain small by allowing false positives

But Will it Merge?



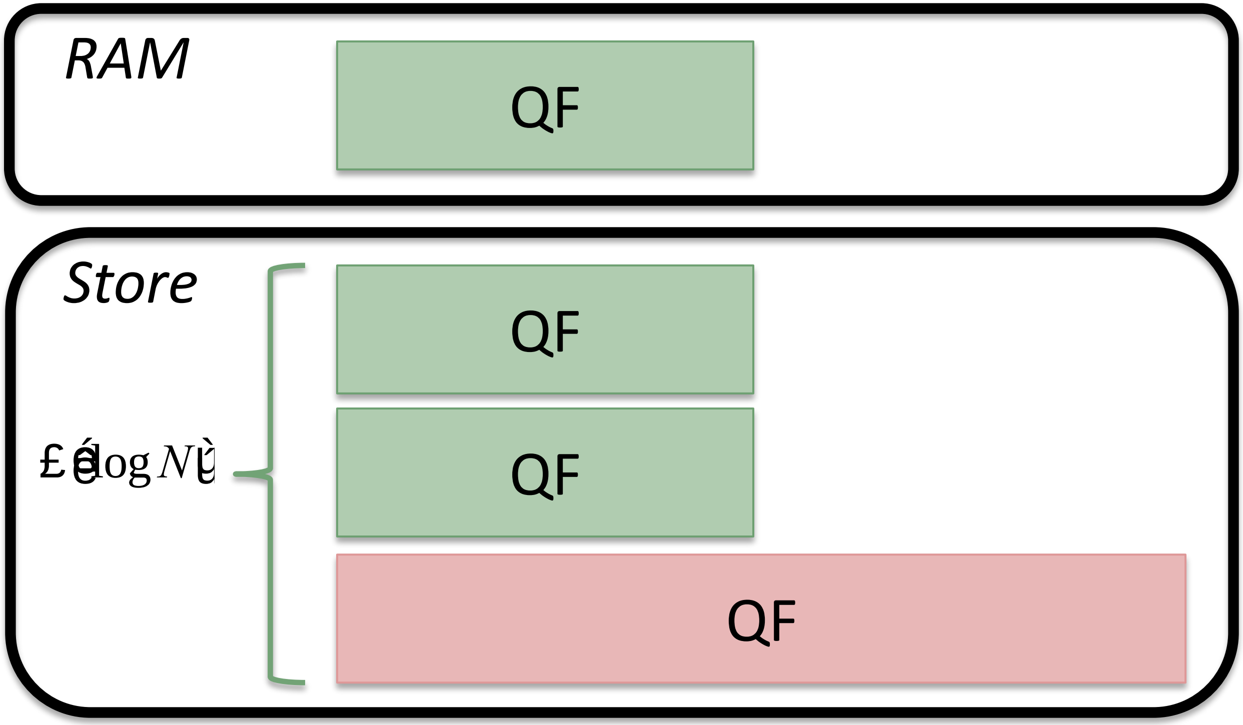
- Actually, a compact **sorted list of integers**

Merge as Integers, Then Insert



- QFs support Plug-n-Play with wrt.-opt. DSes

Cascade Filter

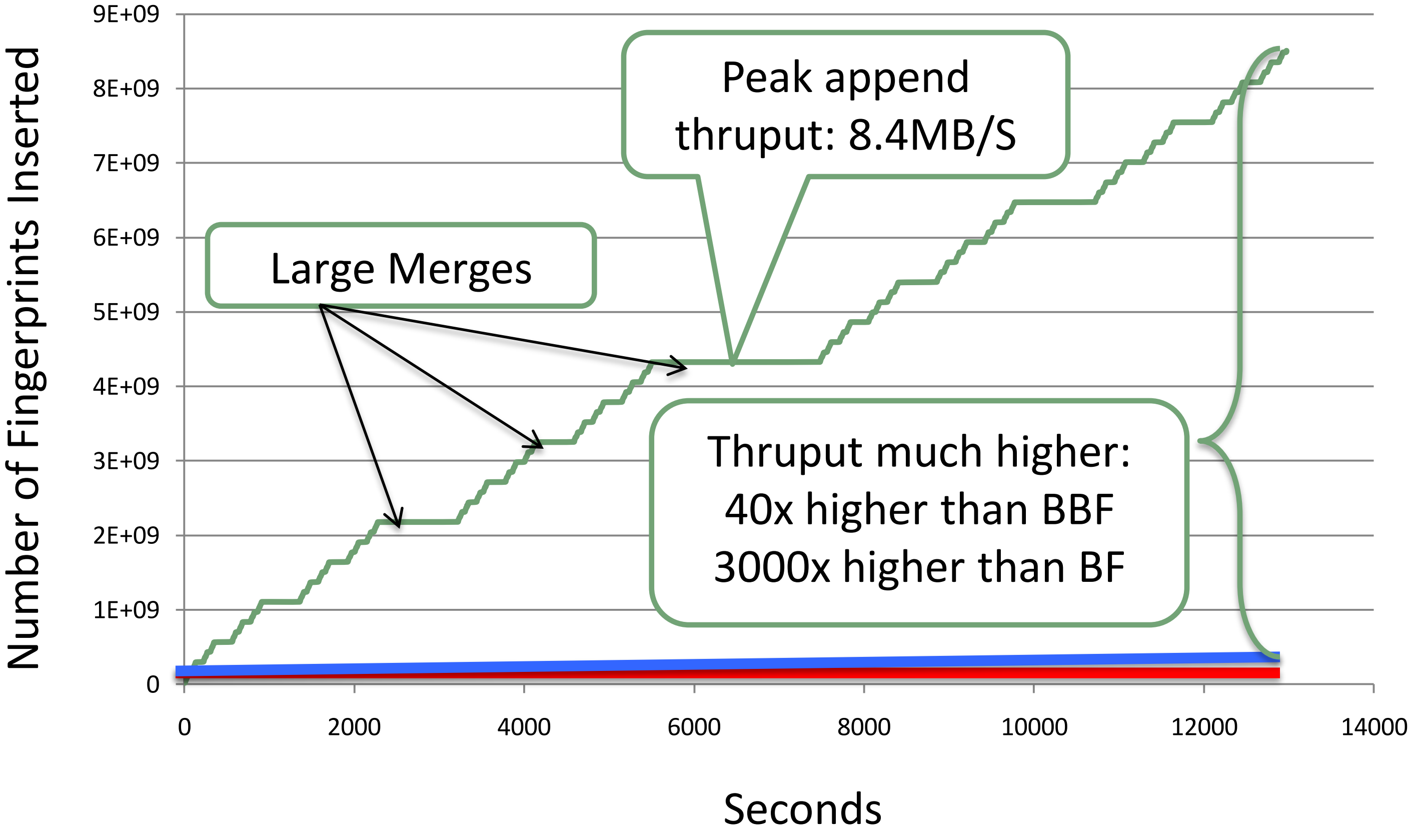


- Just substitute sorted lists of elements with Quotient Filters instead
- Now we have fast insertions and a compact representation **in Flash**

Experimental Setup

- Everything was the same (e.g., cache size)
- Inserted 8.4 billion hashes
- Randomly queried them

Insertion Throughput

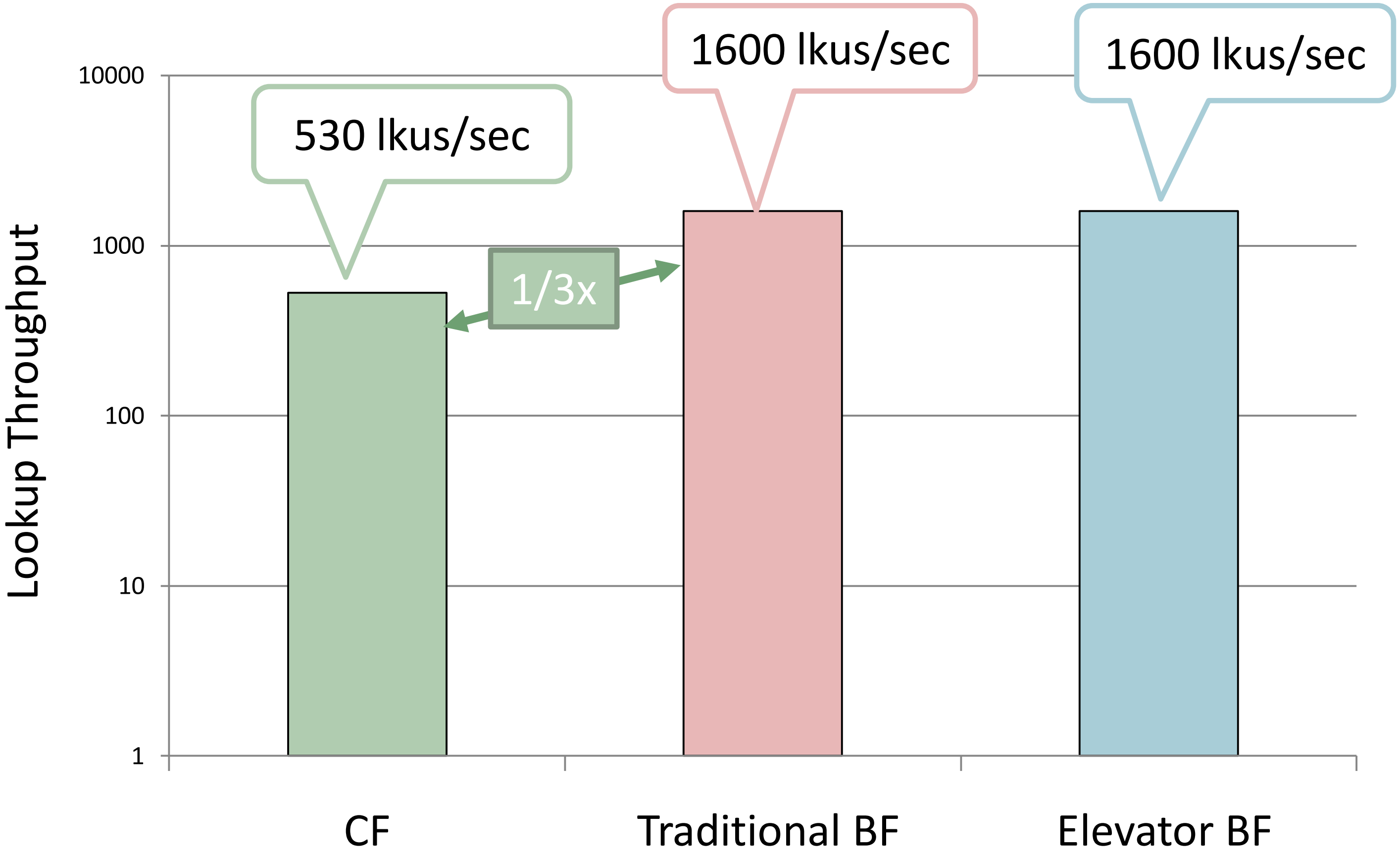


Peak append thruput: 8.4MB/S

Large Merges

Thruput much higher:
40x higher than BBF
3000x higher than BF

Lookup Throughput



Conclusions

- Quotient Filters outperform BFs in RAM
  - 3x faster inserts, same lookups
  - Support deletes
  - Can be dynamically resized
- Cascade Filters outperform BFs in Flash
  - All advantages of Quotient Filters (e.g., deletes)
  - 40x faster inserts, 1/3x lookups
  - CPU bound

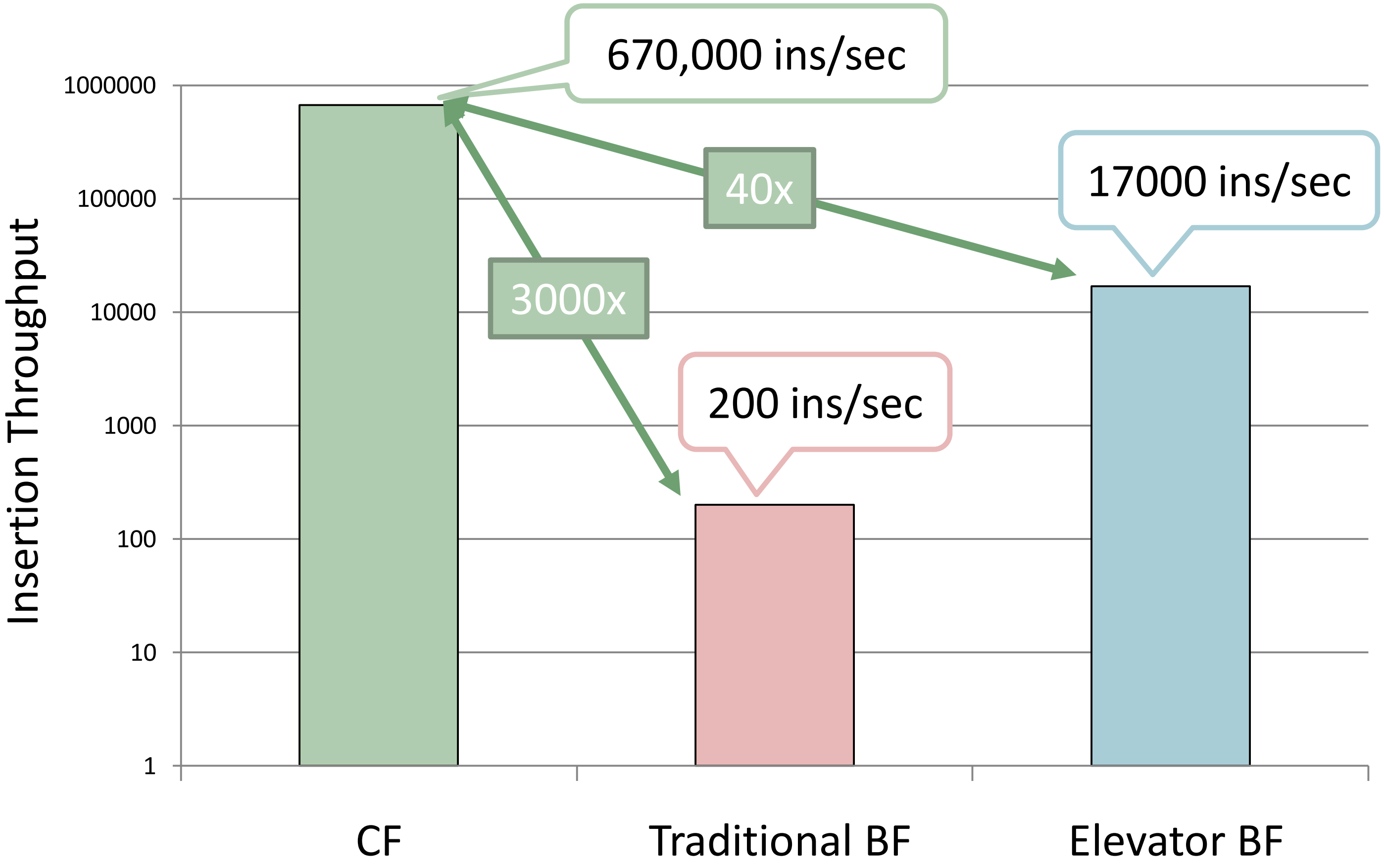
Future Work

- Tweak the CF to handle buffering as well
- Measure real index workloads
- Can a CF help a write-optimized DB?
- There are a lot of exciting boulevards to explore

And That is How…

- …you Don’t Thrash, when you Cache Your Hash in Flash
- Thank you for listening, Questions?
  - Pablo Montes: pmontes@cs.stonybrook.edu
  - Rick Spillane: rick@fsl.cs.sunysb.edu

Insertion Throughput



Experimental Setup

- Controls:
  - ~Equal DS cache size, BF given benefit of doubt
  - Equal RAM in all runs/tests
  - BF tests run in steady-state for 4+ hours
  - CF tests run for 8.4 billion insertions (~16GB CF)
  - Flash partition 60% of Intel X25-Mv2, 90GB
- Machine:
  - Quad-core 2.4GHz Xeon E5530 with 8MB cache
  - 24GB of RAM (booted with 0.994GB)
  - 159.4GB Intel X-25M SSD (second generation)

Future Work

- Measure CF effectiveness for read-optimized
- Measure real index workloads
- Can a CF help a write-optimized DB?
- Better CPU/GPU optimization
- There are a lot of exciting boulevards to explore