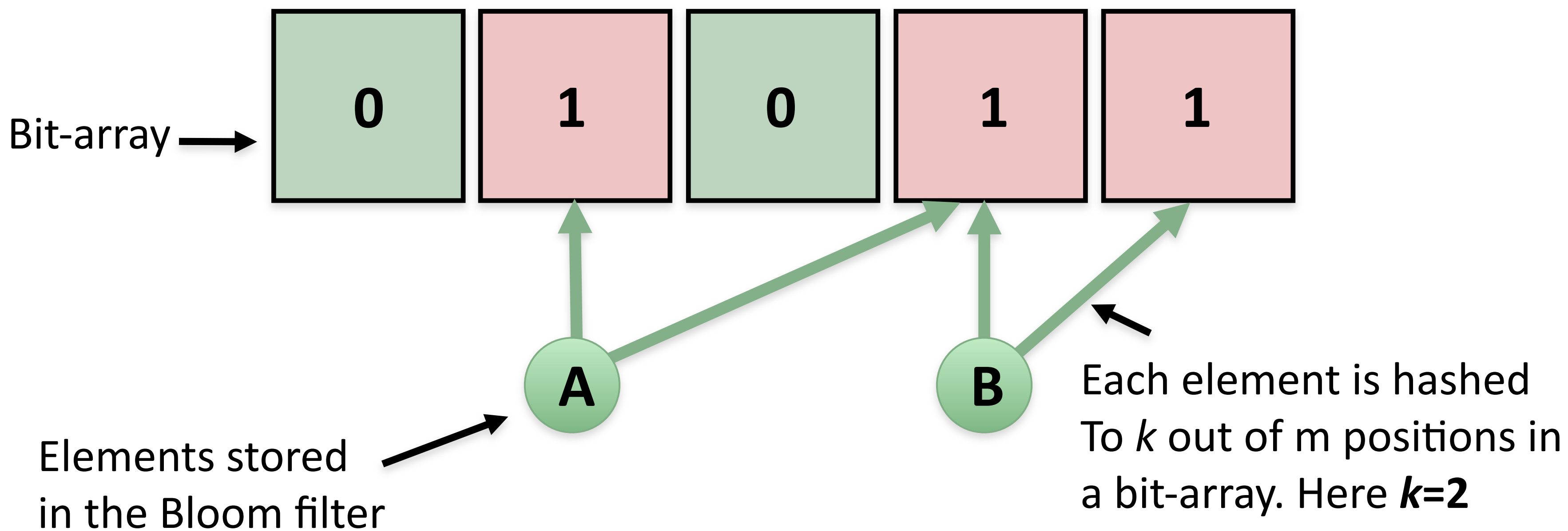
Don’t Thrash: How to Cache Your Hash in Flash

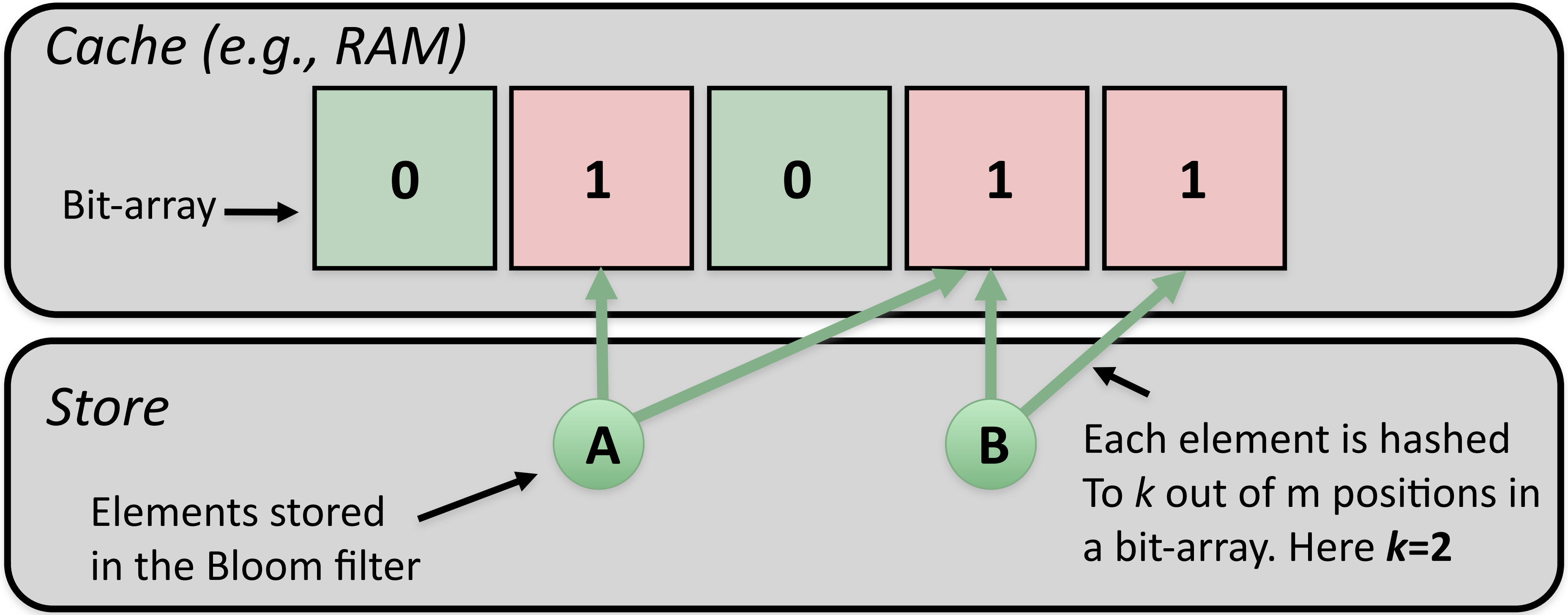
Michael A. Bender
Bradley C. Kuszmaul
Pradeep Shetty
Martin Farach-Colton
Dzejla Medjedovic
Richard P. Spillane
Rob Johnson
Pablo Montes
Erez Zadok

Bloom Filter (approx membership queries)



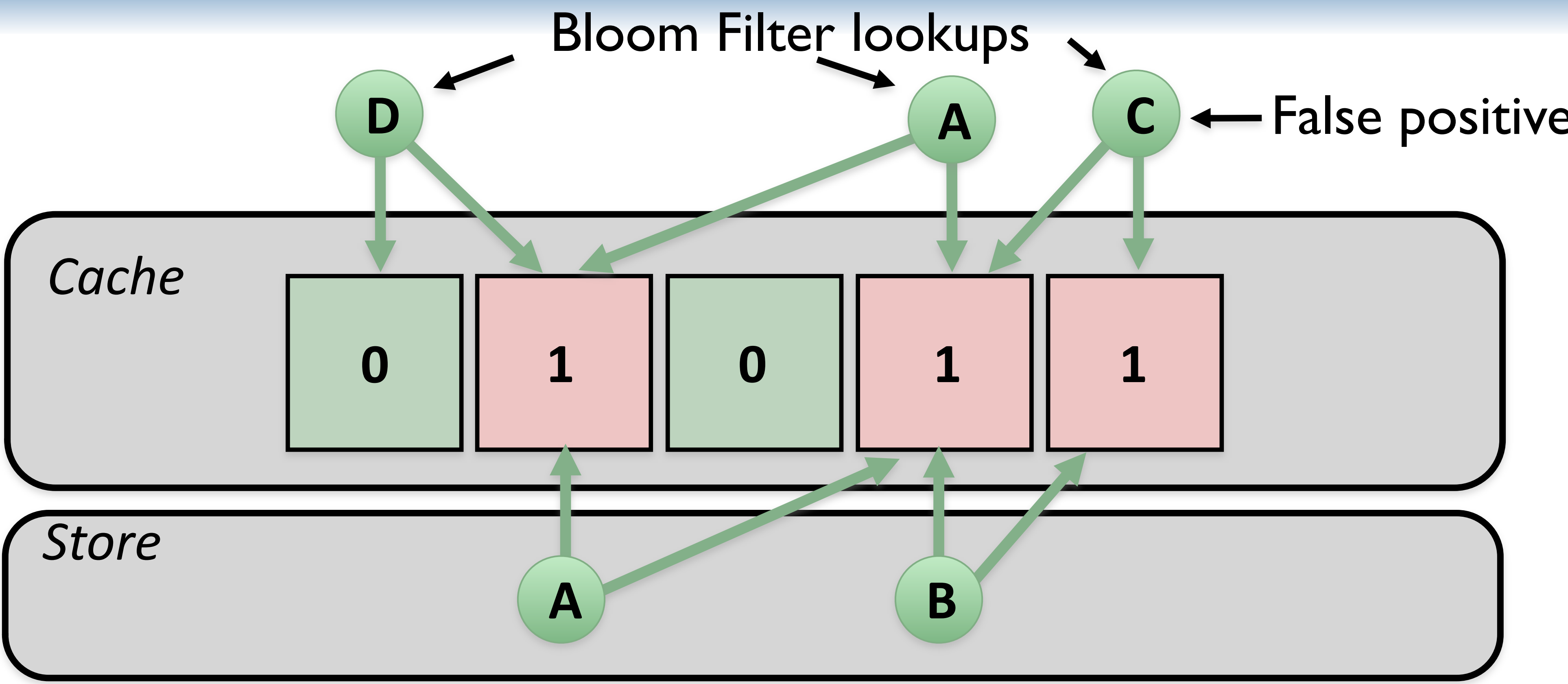
- A BF is a bit-array + *k* hash functions
- The BF stays in RAM because it only stores a few bits per element independent of their sizes.

Bloom Filter (approx membership queries)



- A BF is a bit-array + *k* hash functions
- The BF stays in RAM because it only stores a few bits per element independent of their sizes.

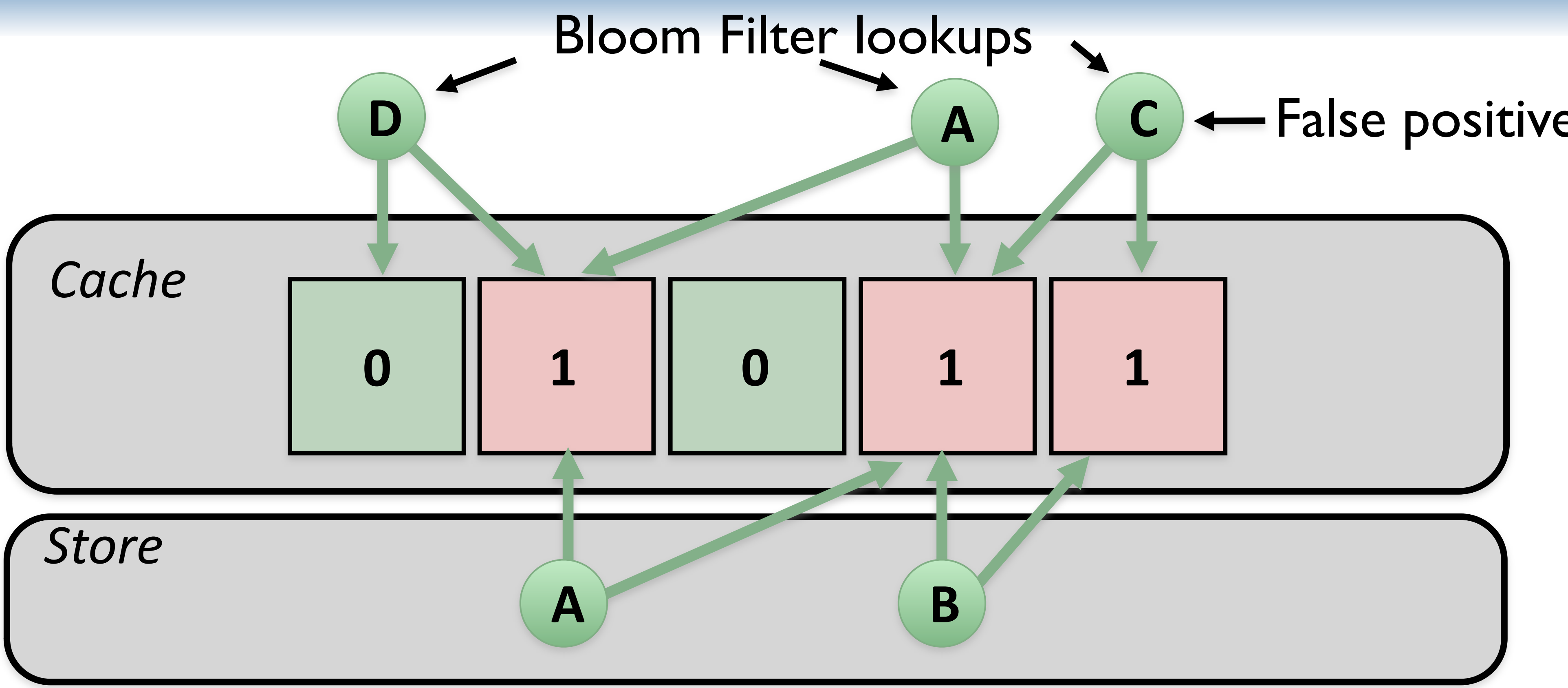
Don't Thrash: **How to Cache Your Hash** in Flash



Bounded rate of false positives: $p_{FP}(x) \approx \left(1 - e^{-kn/m}\right)^k$

No false negatives.

Don't Thrash: **How to Cache Your Hash** in Flash



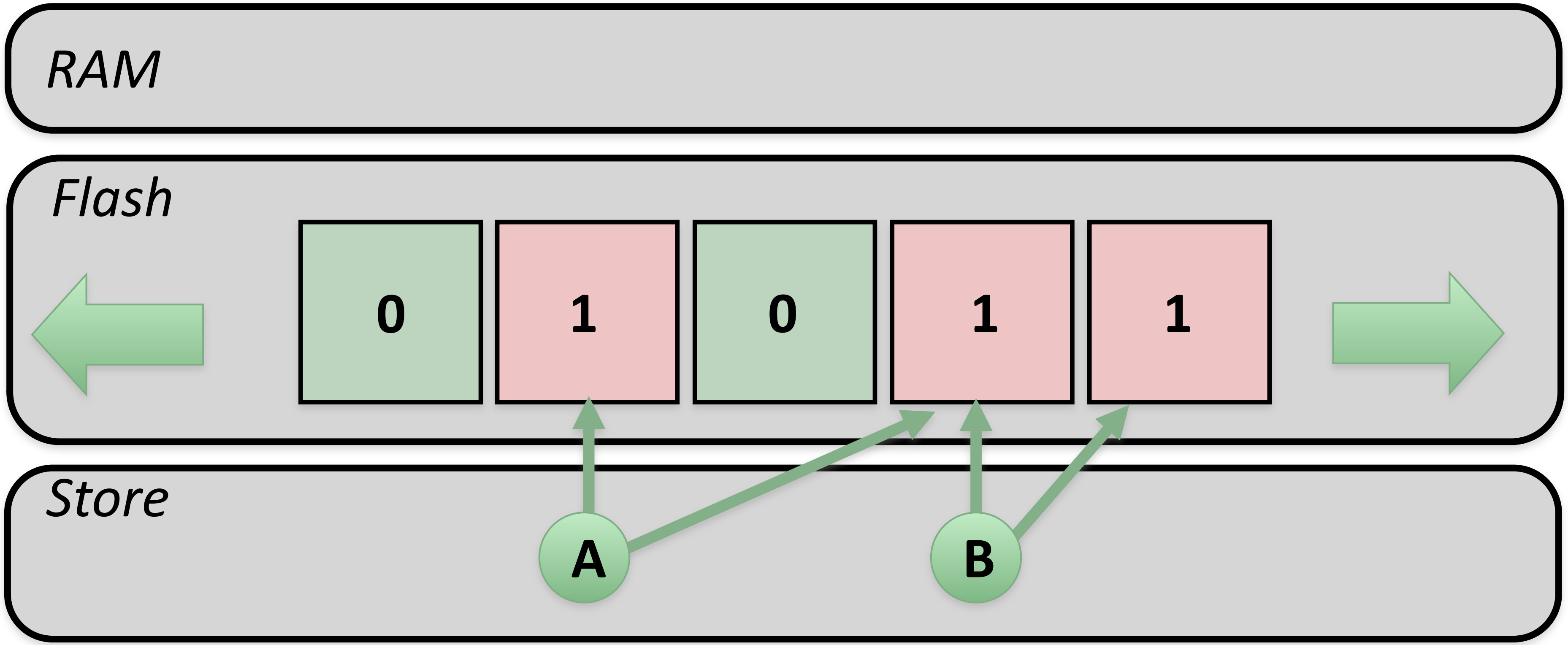
Bounded rate of false positives:

$$p_{FP}(x) \approx \left(1 - e^{kn/m}\right)^k$$

No false negatives.

Fast way to say "No" without going to disk.
The Bloom filter is a well-loved hammer in a system-builders toolbox.

Don't Thrash: How to Cache Your Hash in **Flash**



Flash (e.g., SSD) is bigger & cheaper than RAM, faster than disk.

- Good place to store Bloom filter, in between disk and RAM.

Bloom filters even for moderate-sized data sets may be too large for RAM.

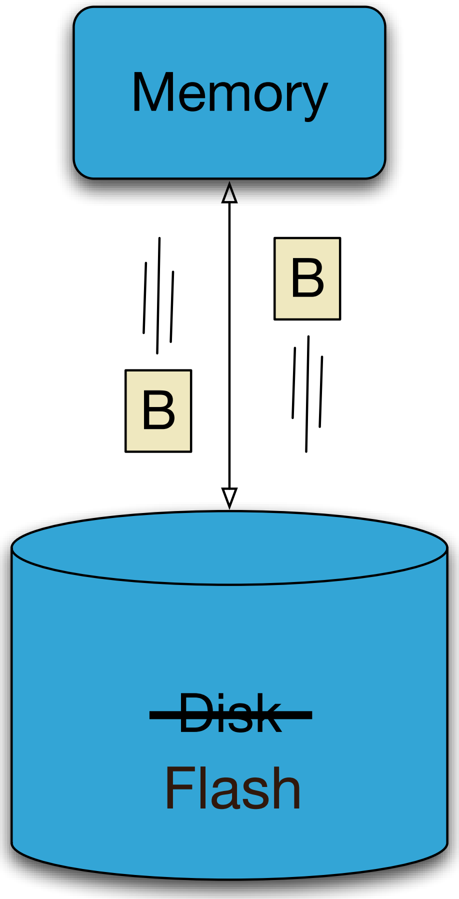
- Example: 8TB of 512B keys needs 16GB of RAM for a ~1% BF.

We should think about Flash the way we think about disk, except that random I/O is faster.

- Two-levels of memory.
- Two parameters:
  block-size B, memory-size M.

Metric: # of block transfers.

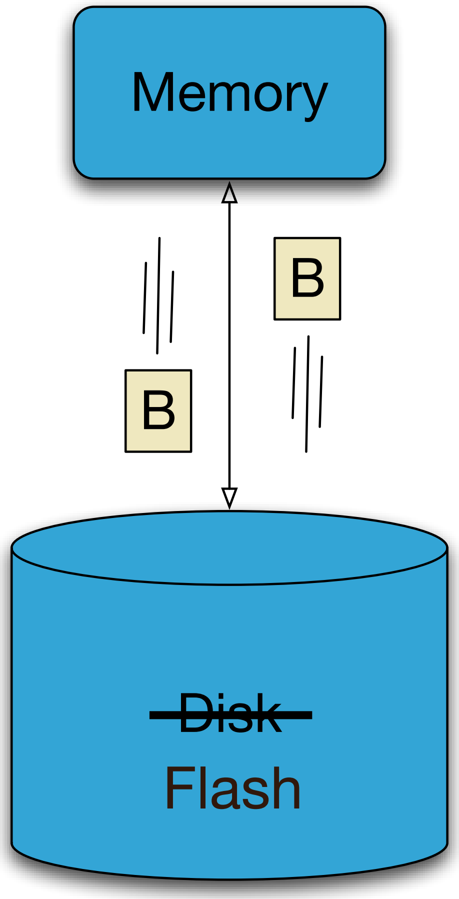
- Block transfers are faster in flash than disk, but the issues are similar.



Don't Thrash: How to Cache Your Hash in **Flash**

We should think about Flash the way we think about disk, except that random I/O is faster.

- Two-levels of memory.
- Two parameters:
  block-size B, memory-size M.



Metric: # of block transfers.

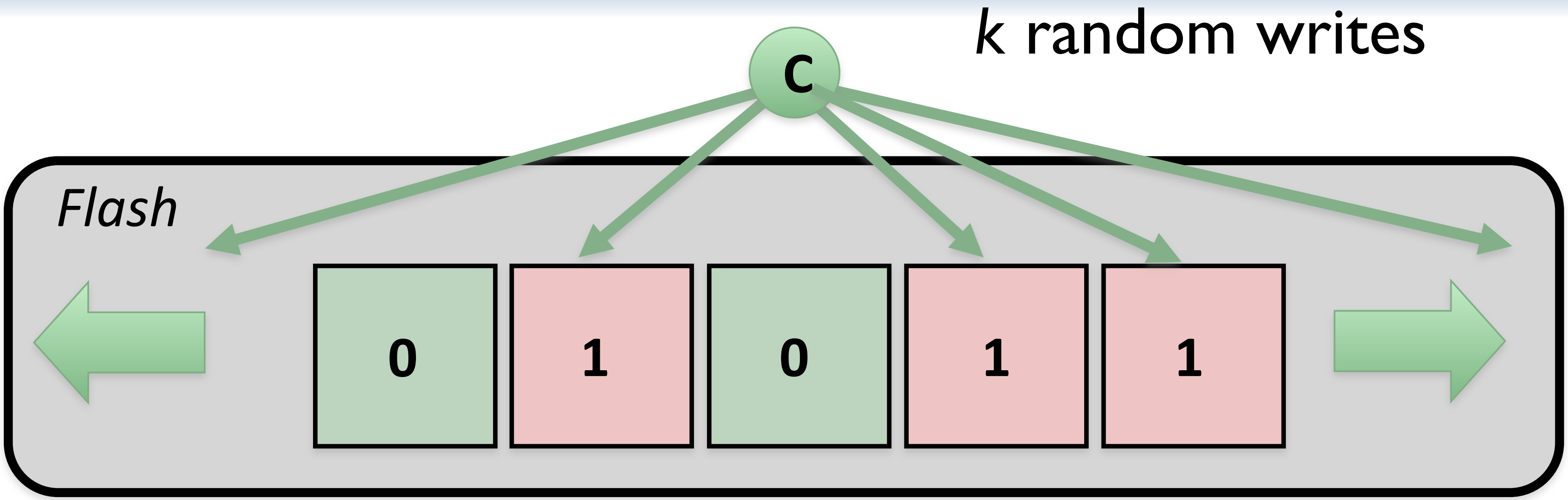
- Block transfers are faster in flash than disk, but the issues are similar.

(More accurate SSD models exist

[Ajwani, Beckmann, Jacob, Meyer, and Moruz 2009]

but aren't necessary here.)

Don't **Thrash**: How to Cache Your Hash in Flash



Thrashing in Bloom filter implemented in Flash.

- Setting *k* random bits to 1 causes O(*k*) random writes.
- OK in RAM, expensive in Flash.

We cannot have an efficient data structure without working around this issue.

Don't Thrash: How to Cache Your Hash in Flash (**Results**)

Cascade Filter (CF), a BF replacement optimized for fast insertions/deletions on flash.

Write-optimized performance:

- 670,000 inserts/sec (3000x of Bloom, 40x best alternative)
- 530 lookups/sec (1/3x of best alternative)

The Cascade Filter is based upon Quotient Filters (QFs) instead of BF

- QFs have better access locality.
- QFs support deletes.

We can efficiently merge two QFs into a larger QF, which is important for fast insertions.

Don't Thrash: How to Cache Your Hash in Flash (**Results**)

We just presented this result at HotStorage.

- Not an algorithms conference.
- The storage community is grappling with difficult I/O issues. Feeling tooth pain.
- Community understands that I/O-efficient algorithms can help.

Don't Thrash: How to Cache Your Hash in Flash (**Results**)

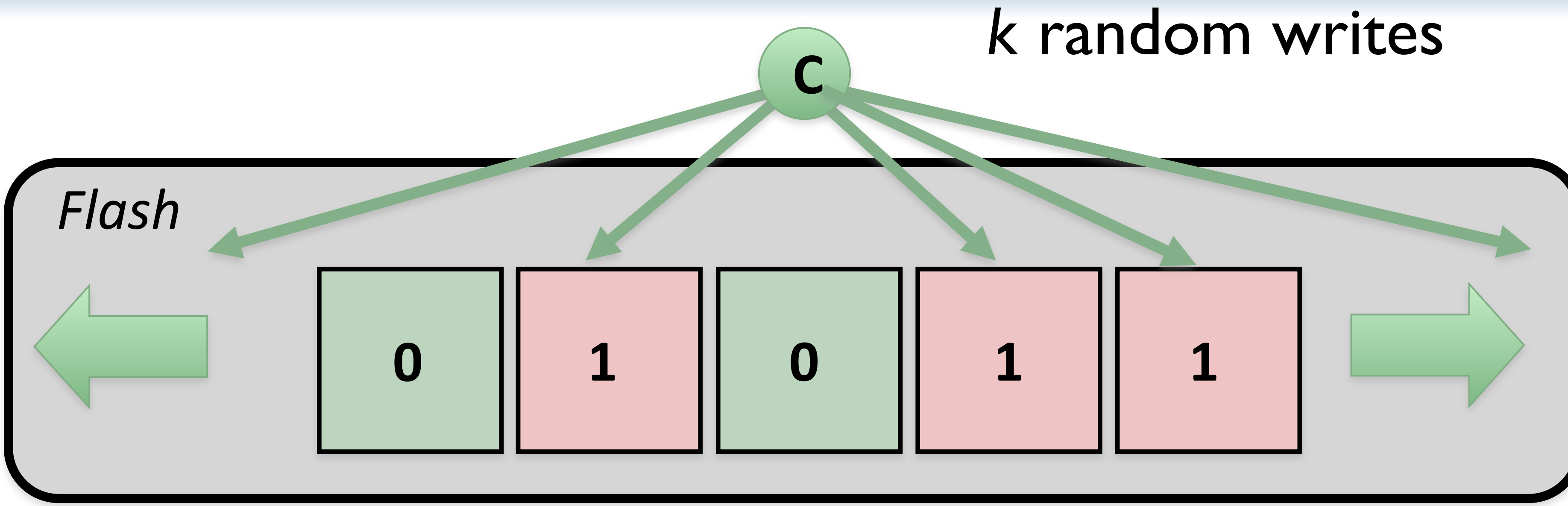
We just presented this result at HotStorage.

- Not an algorithms conference.
- The storage community is grappling with difficult I/O issues. Feeling tooth pain.
- Community understands that I/O-efficient algorithms can help.

Of interest to several write-optimized storage systems

- Apache: HBase
- Facebook: Cassandra
- Google: BigTable
- Tokutek: TokuDB

Don't **Thrash**: How to Cache Your Hash in Flash

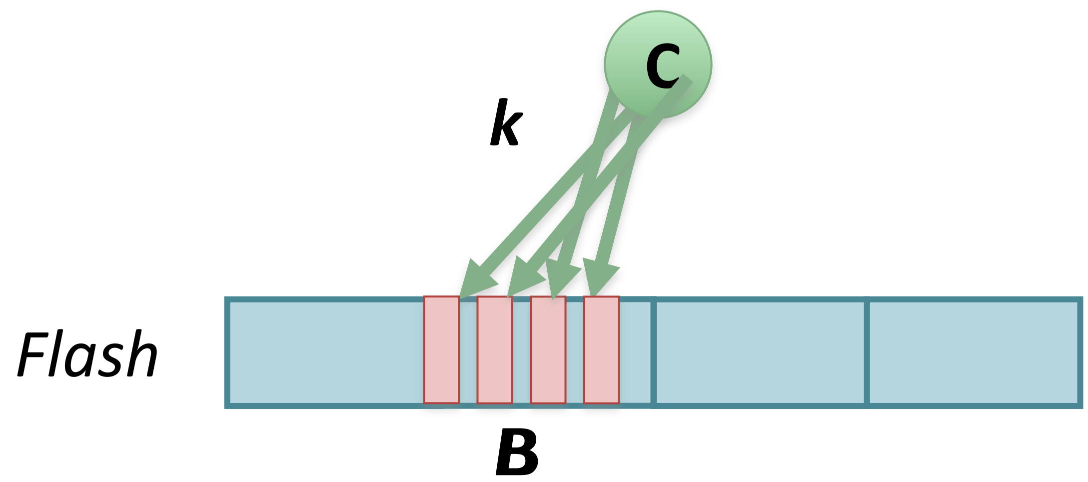


Thrashing in Bloom filter implemented in Flash.

- Setting *k* random bits to 1 causes O(*k*) random writes.
- OK in RAM, expensive in Flash.

We cannot have an efficient data structure without working around this issue.

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



Use a two-step hash to shave off *k*

[Canim, Mihaila, Bhattacharjee, Lang, Ross, 2010]

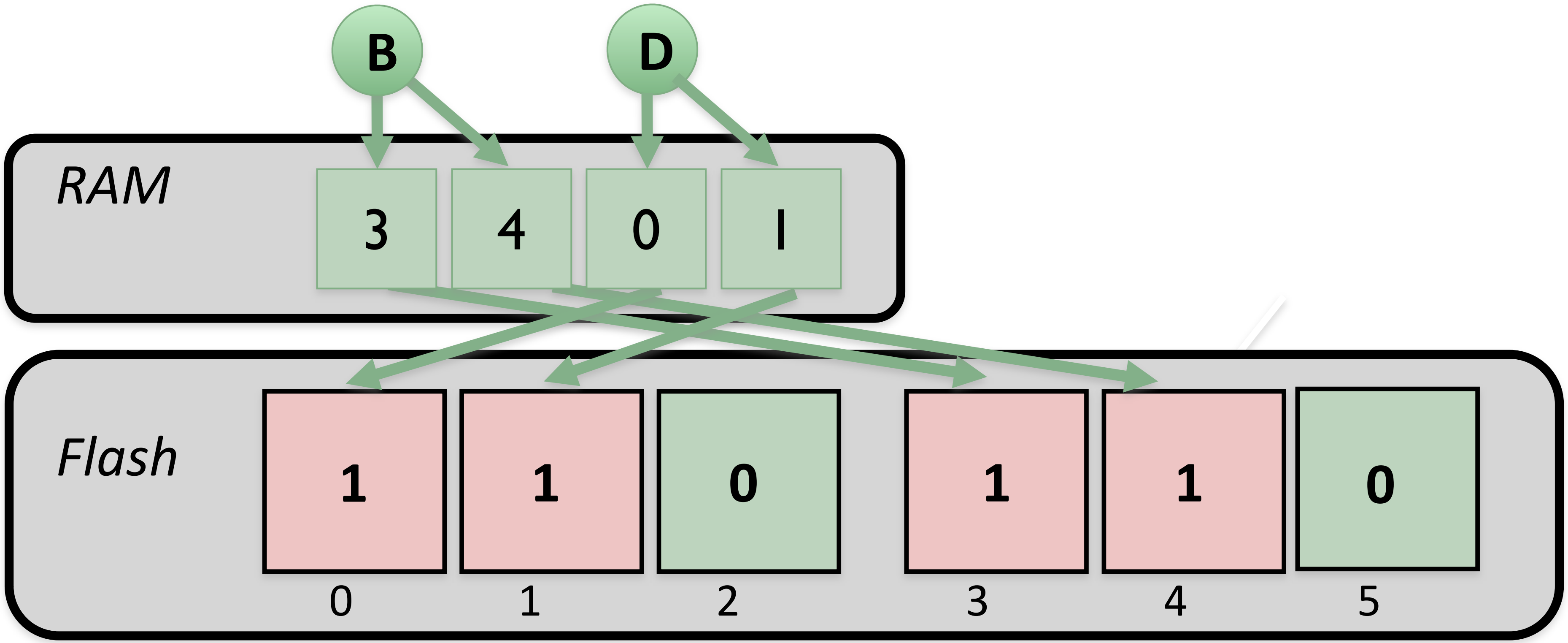
- Store a separate Bloom filter in each flash page.
- First hash to a particular page/Bloom filter.
- Perform the *k* writes.

This helps a little.

- 1 I/O per insertion. We want <1 I/O per insert.

***Don't* Thrash**: How to Cache Your Hash in Flash

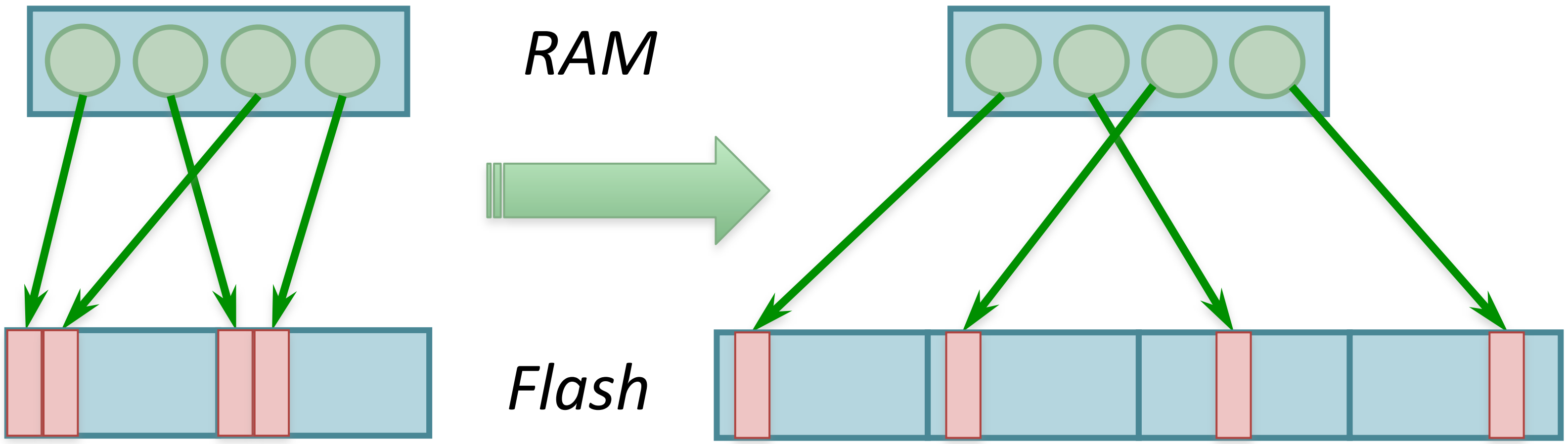
Buffer Writes



Amortize the I/O cost over many bit writes.

- This helps a lot [Canim, Mihaila, Bhattacharjee, Lang, Ross, 2010]

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



- **Buffering works well when Flash isn't too large compared to RAM.**
- **Buffering degrades as flash size grows, approaching ~1 I/O per insert.**
- **We're interested in large datasets and fast insertions (i.e., when buffering doesn't work)**

Don't Thrash: How to Cache Your Hash in Flash (**Results**)

Cascade Filter (CF), a BF replacement optimized for fast insertions/deletions on flash.

Write-optimized performance:

- 670,000 inserts/sec (40x of other variants)
- 530 lookups/sec (1/3x of other variants)

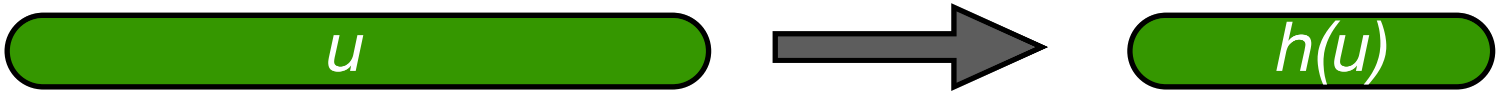
The Cascade Filter is based upon Quotient Filters (QFs) instead of BF

- QFs have better access locality.
- QFs support deletes.

We can efficiently merge two QFs into a larger QF, which is important for fast insertions.

Idea of Quotienting

- Take a fingerprint $h(u)$ for each element u.
- Store fingerprints *compactly* in a hashtable.
  (Next slide: how to store compactly.)



Quotienting [Knuth]
(Alternative to Bloom Filters)

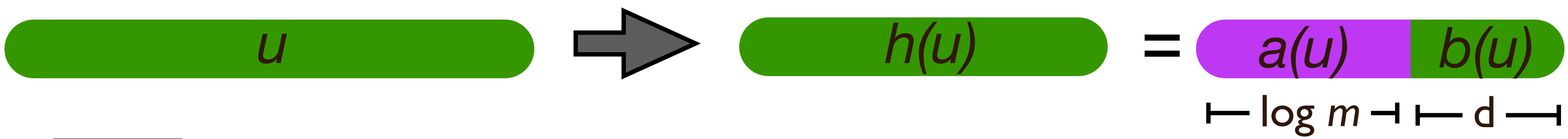
Idea of Quotienting

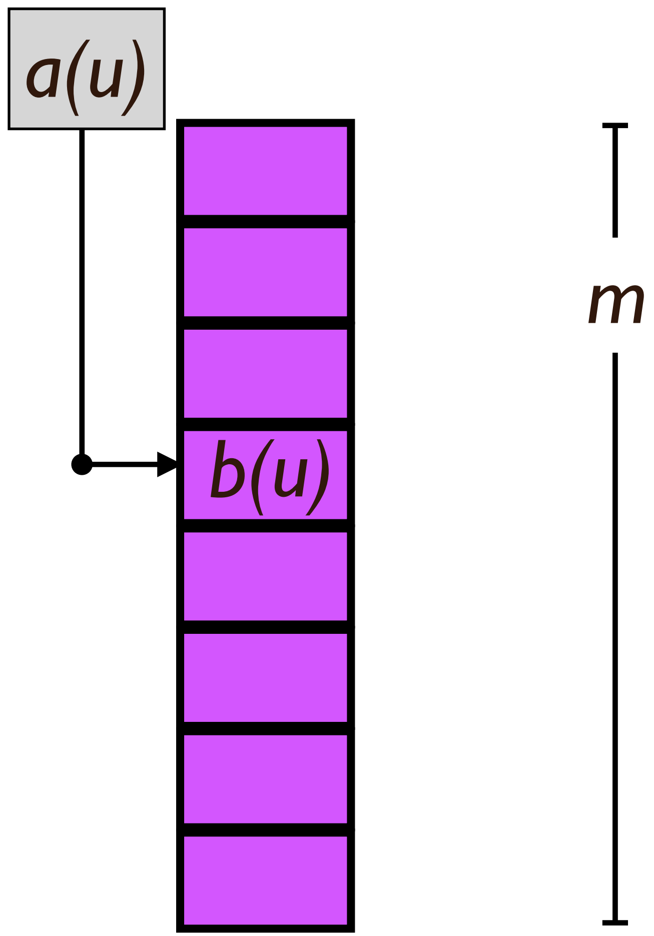
- Take a fingerprint $h(u)$ for each element u.
- Store fingerprints *compactly* in a hashtable.
(Next slide: how to store compactly.)

Only source of False Positives

- Two distinct elements u and v, where $h(u)=h(v)$.
- If u is stored and v isn't, Query(v) gives a false positive.

Quotienting [Knuth]
(Alternative to Bloom Filters)

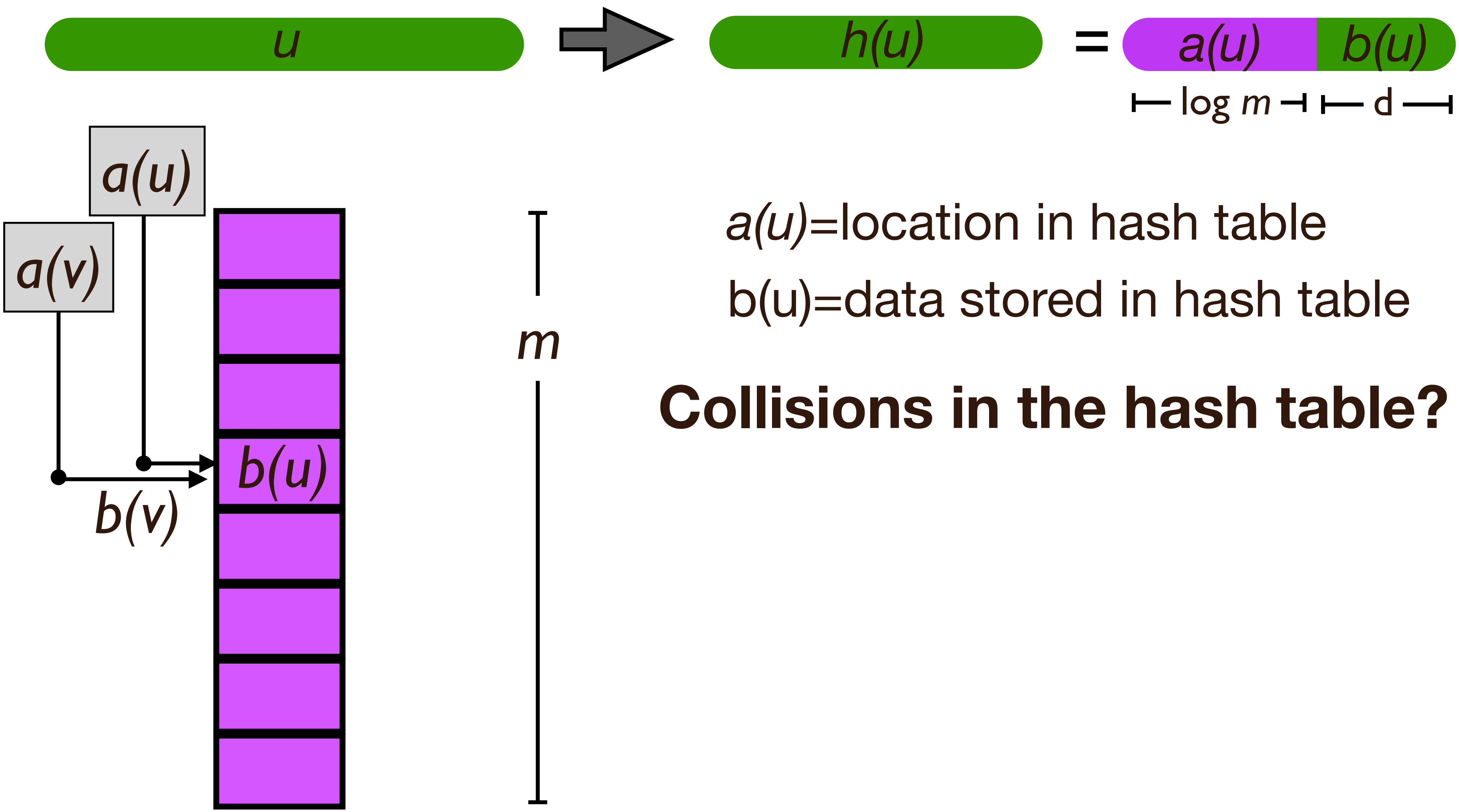




a(u)=location in hash table

b(u)=data stored in hash table

Quotienting [Knuth]
(Alternative to Bloom Filters)

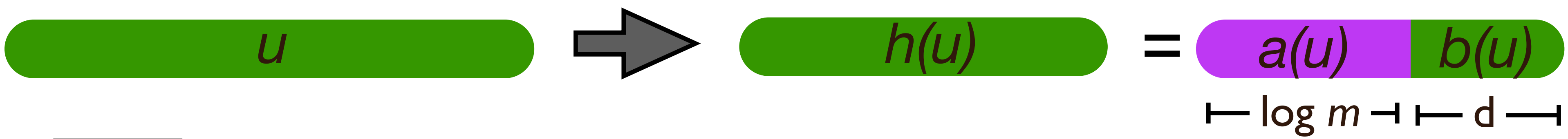


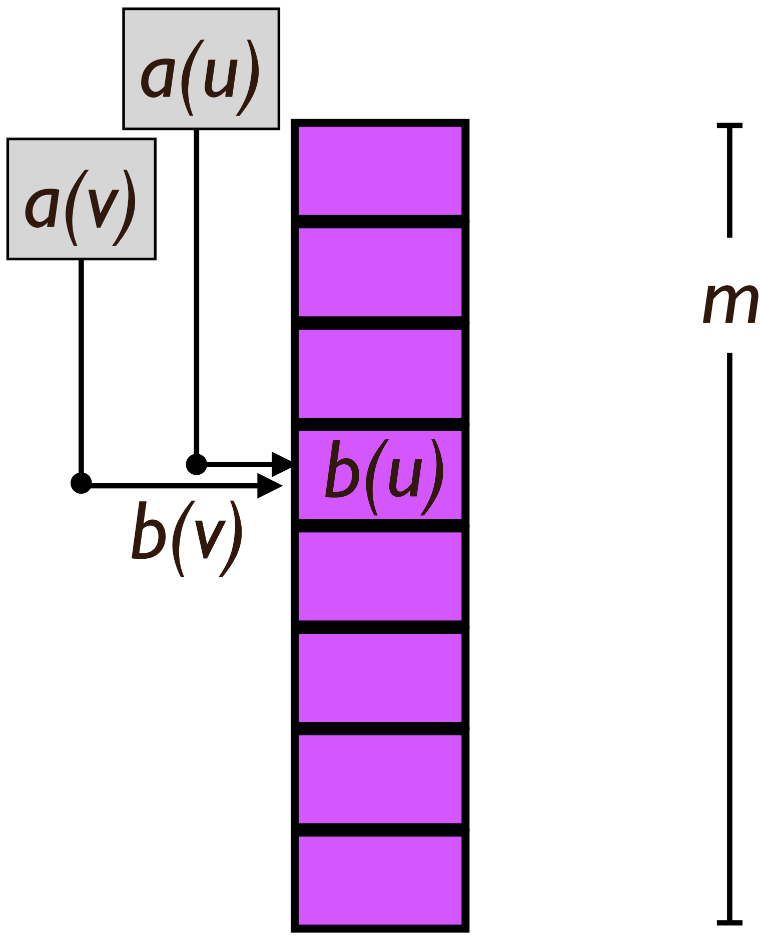
a(u)=location in hash table

b(u)=data stored in hash table

Collisions in the hash table?

Quotienting [Knuth]
(Alternative to Bloom Filters)





a(u)=location in hash table

b(u)=data stored in hash table

Collisions in the hash table?

- Linear probing?
- Yes, but be careful.

Quotienting +Linear Probing?

Ex: 6 bit hash. 3 bits for address, 3 for data.

000	001
001	011
010	
011	010
100	111
101	
110	100
111	

Quotienting +Linear Probing?

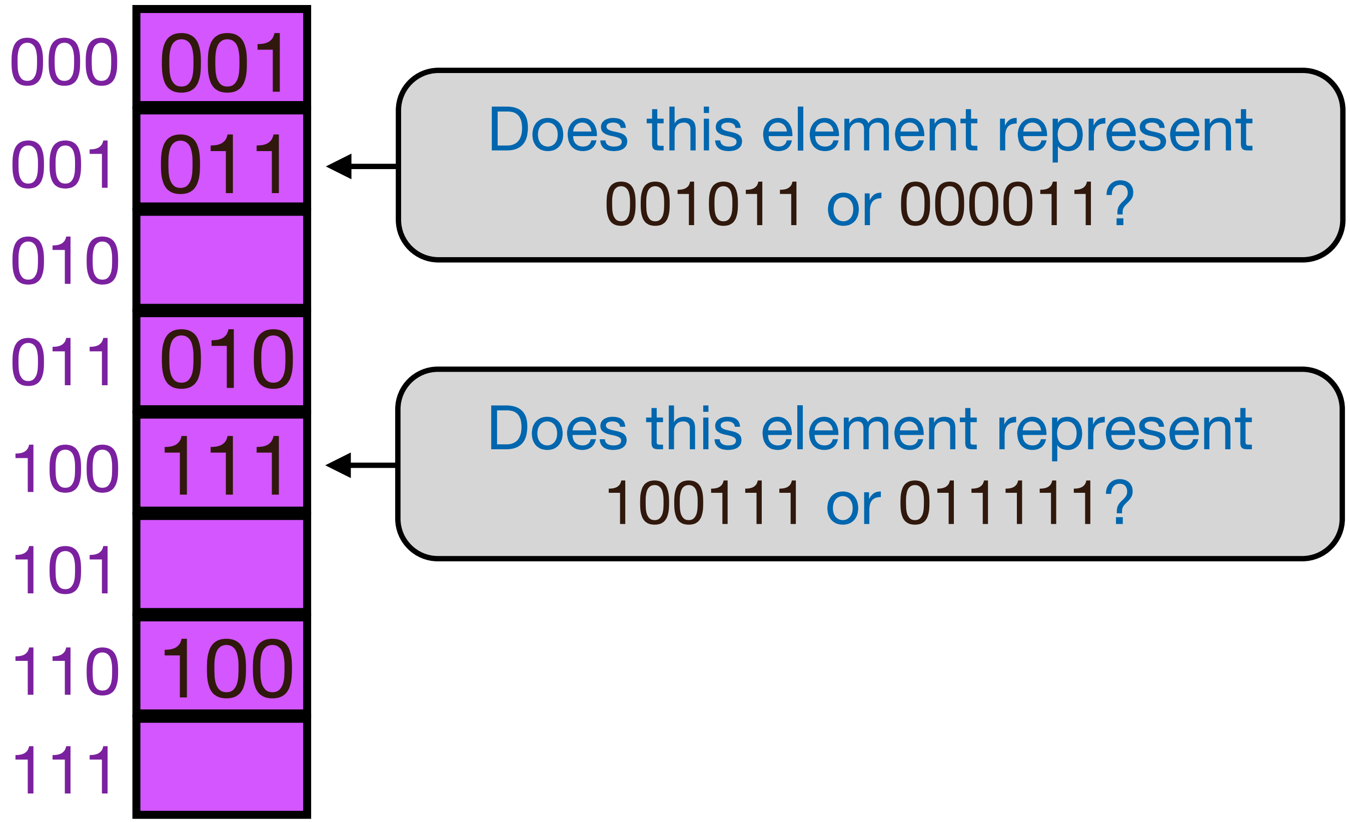
Ex: 6 bit hash. 3 bits for address, 3 for data.

000	001
001	011
010	
011	010
100	111
101	
110	100
111	

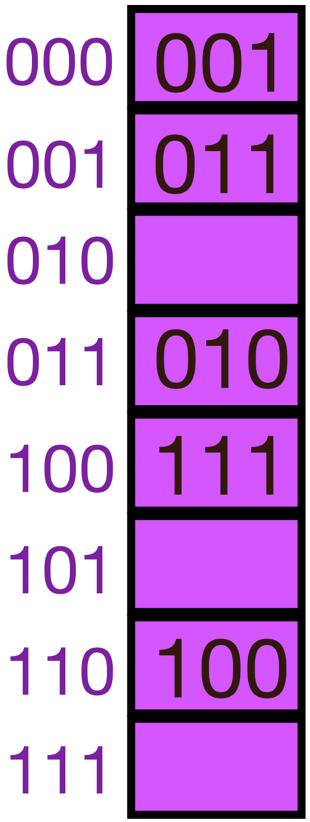
Does this element represent 001011 or 000011?

Quotienting +Linear Probing?

Ex: 6 bit hash. 3 bits for address, 3 for data.



Quotienting +Linear Probing?



Need control bits to

- store distance Δ between “target” and stored location in the hash table;
- Indicate which array positions are filled.

Solutions:

- Naive: O(loglog *n*), since Δ=O(log *n*) w.h.p.
- Can do with 2 control bits, but our implementation was too slow.
- We chose 3 bits.
- How much better can we do than 2?

QF: alternative to Bloom filter.

- Supports deletes as well as inserts.
- 1 I/O per insert/delete/lookup in expectation
- Just a compact, and easily implementable data structure for storing hashes.

Next slide: merge exponentially increasing Quotient Filters to generate the Cascade Filter.

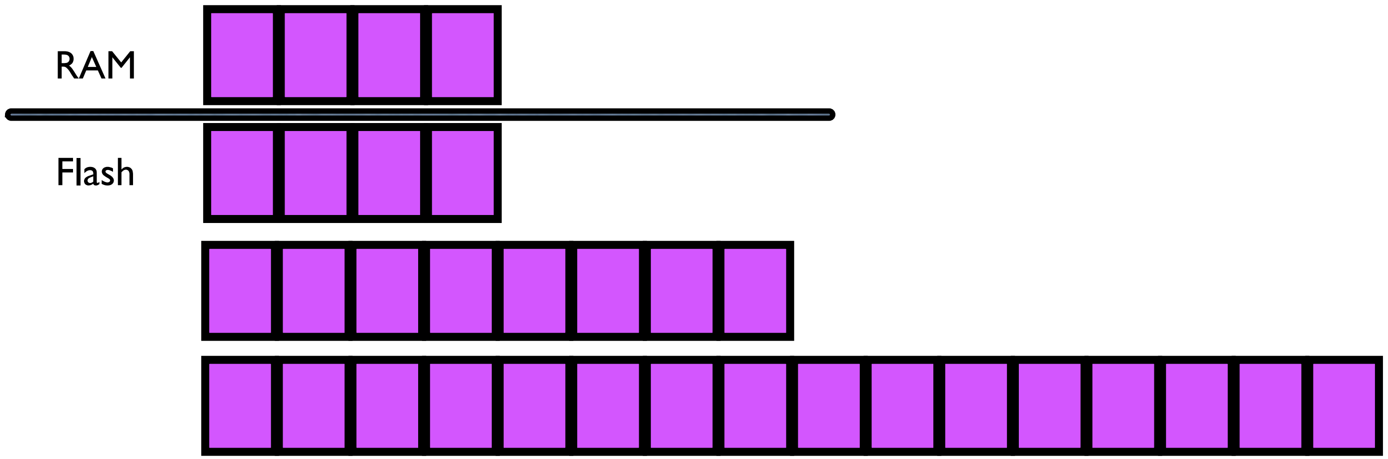
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use a standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



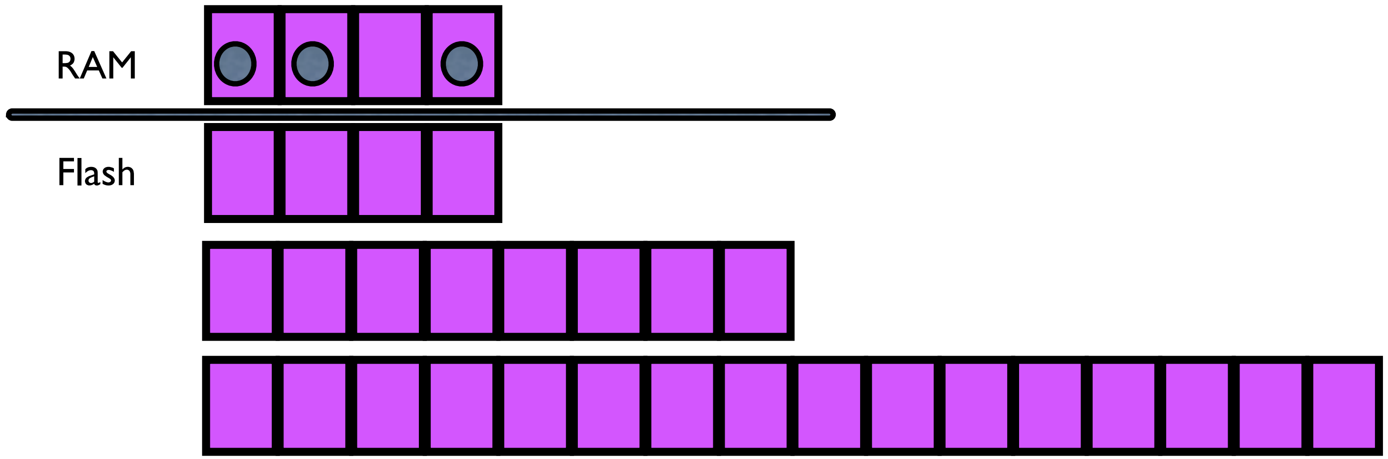
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



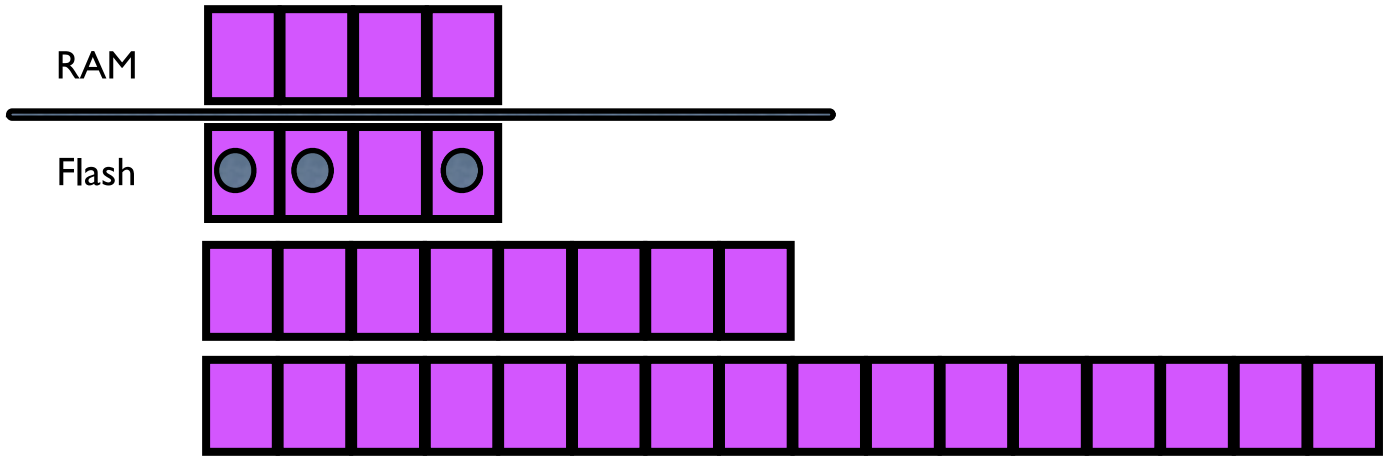
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



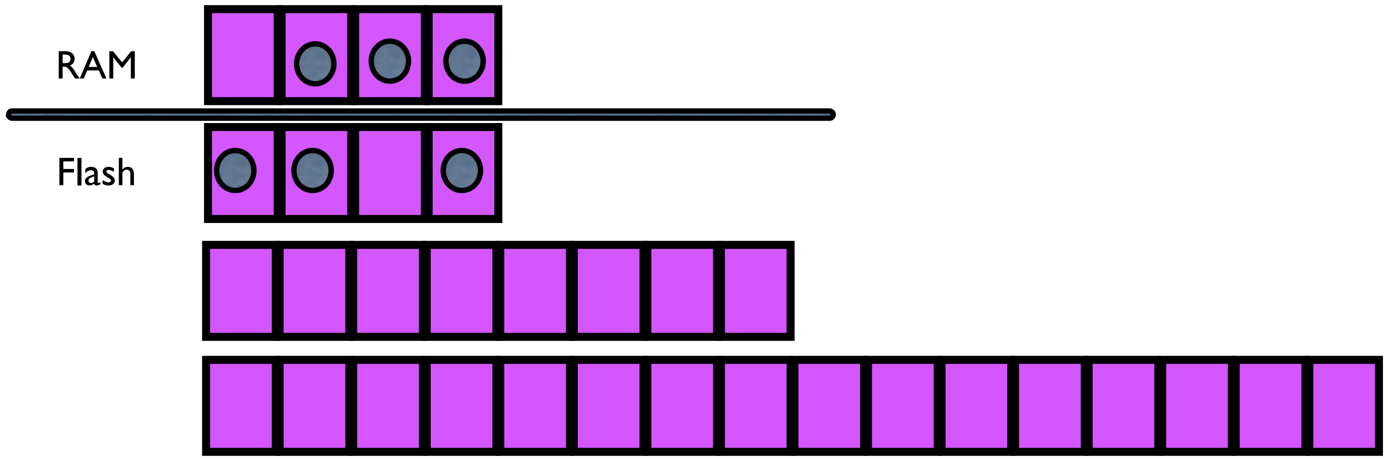
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



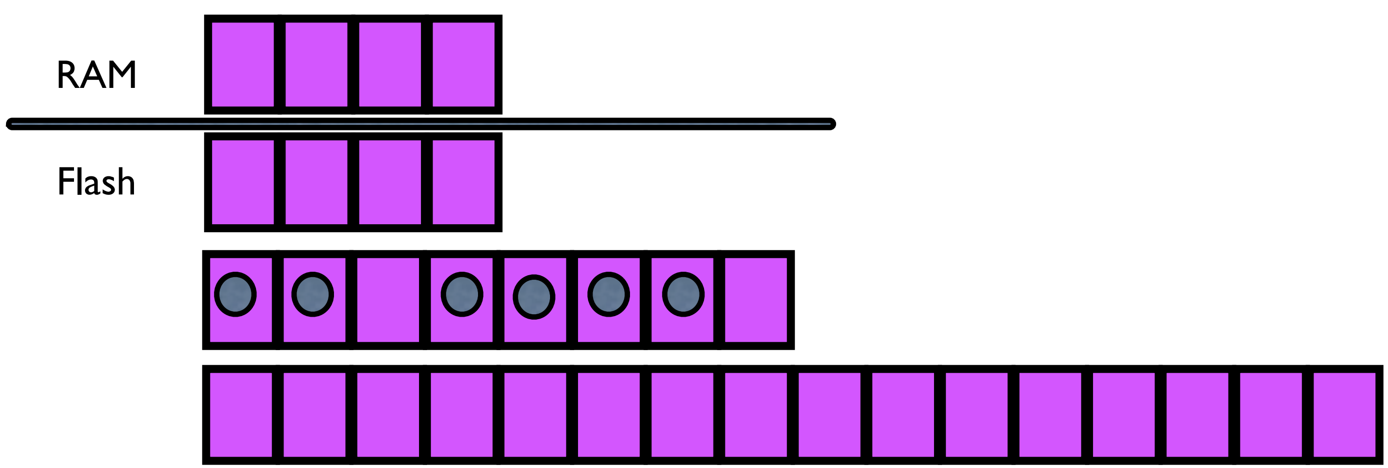
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



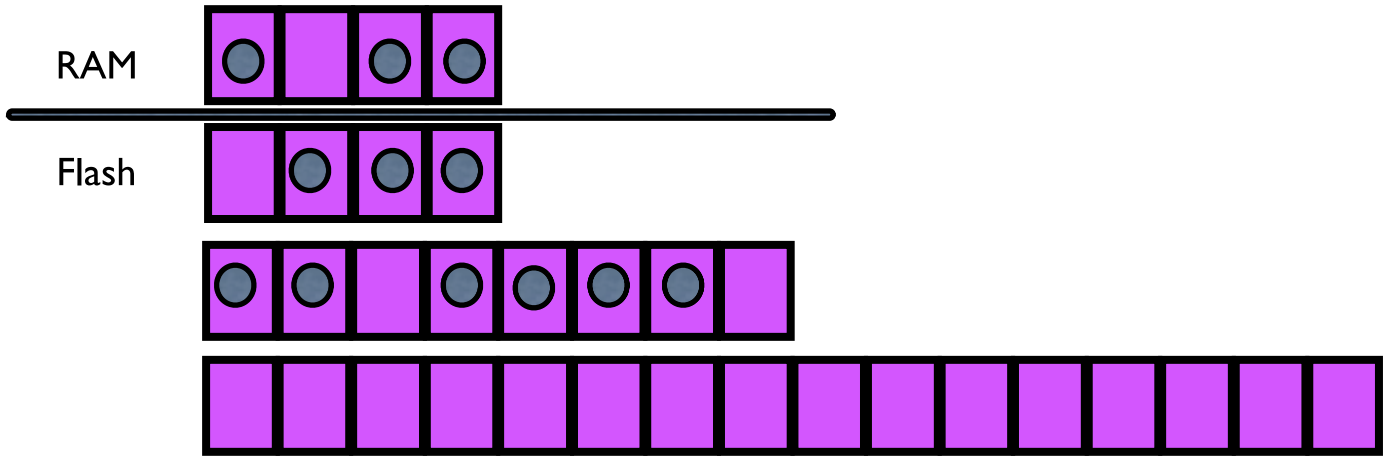
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



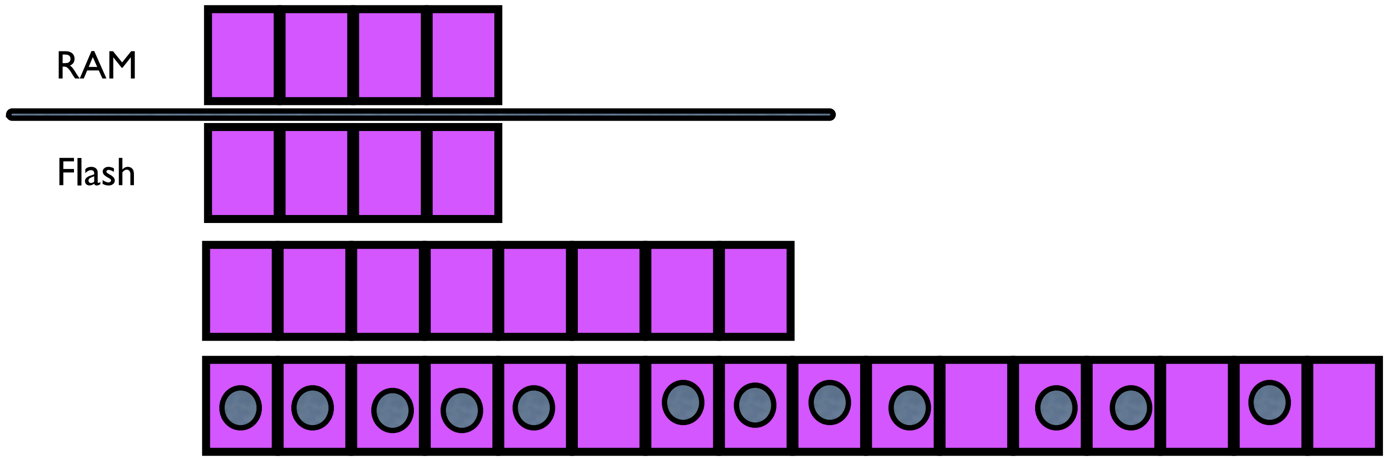
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



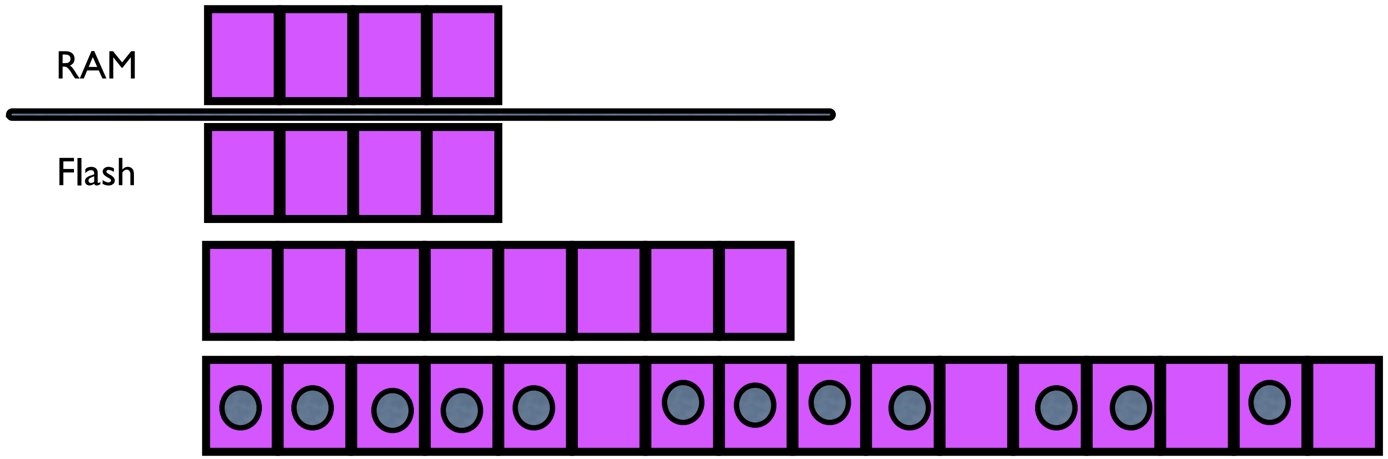
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



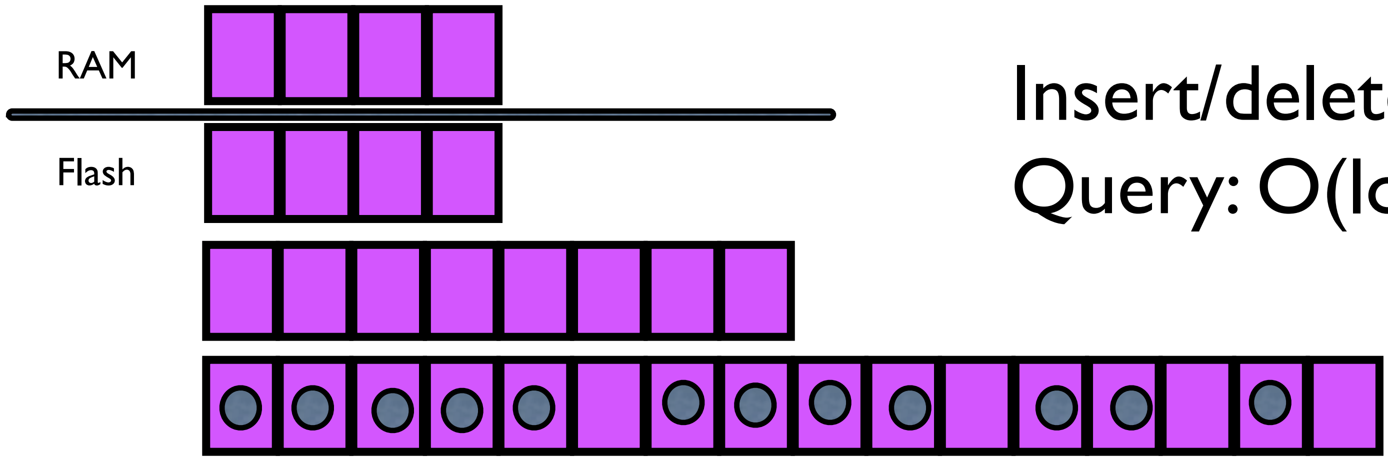
Cascade Filter

Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

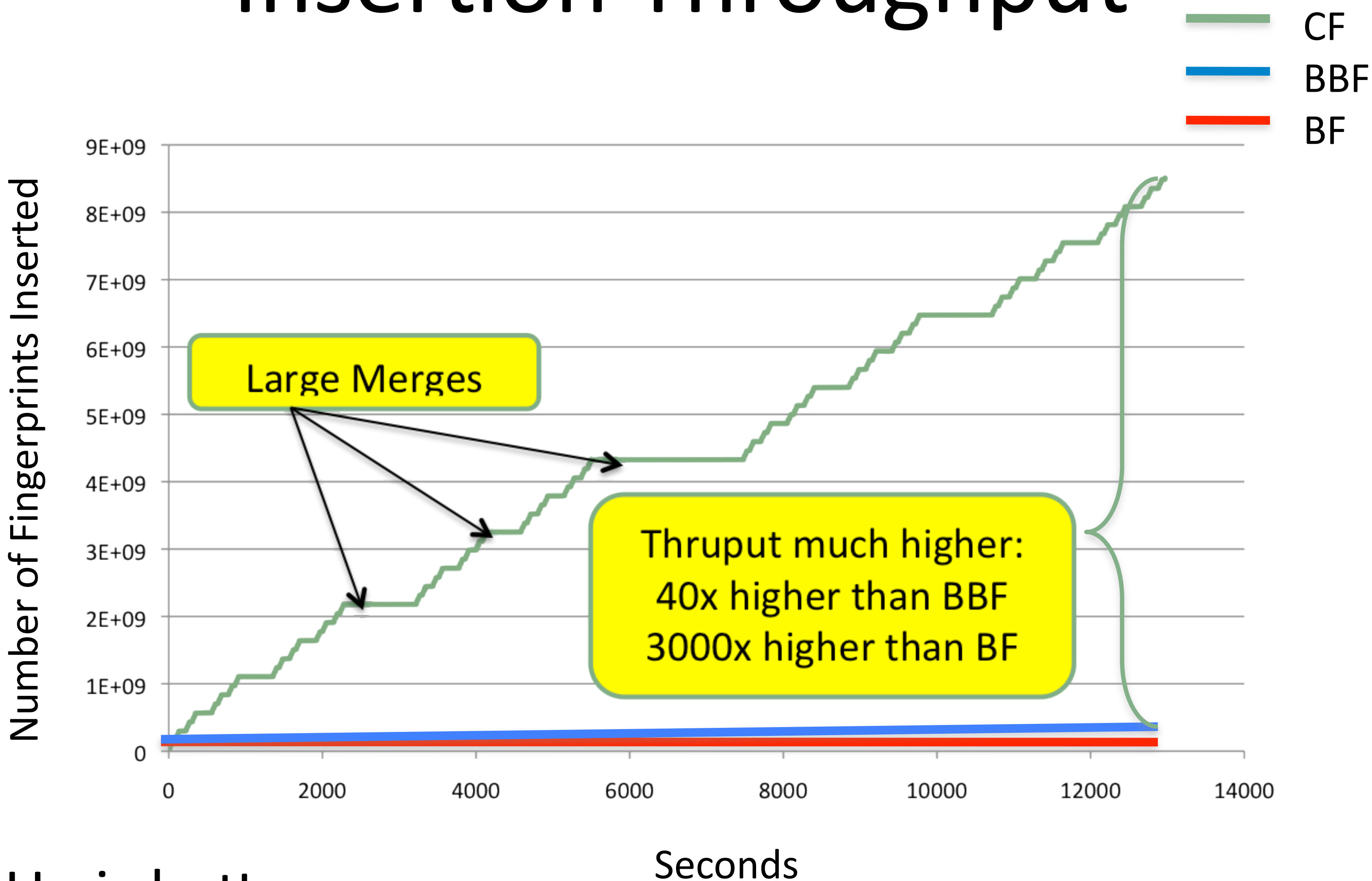
[O'Neil,Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach,Fineman,Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans



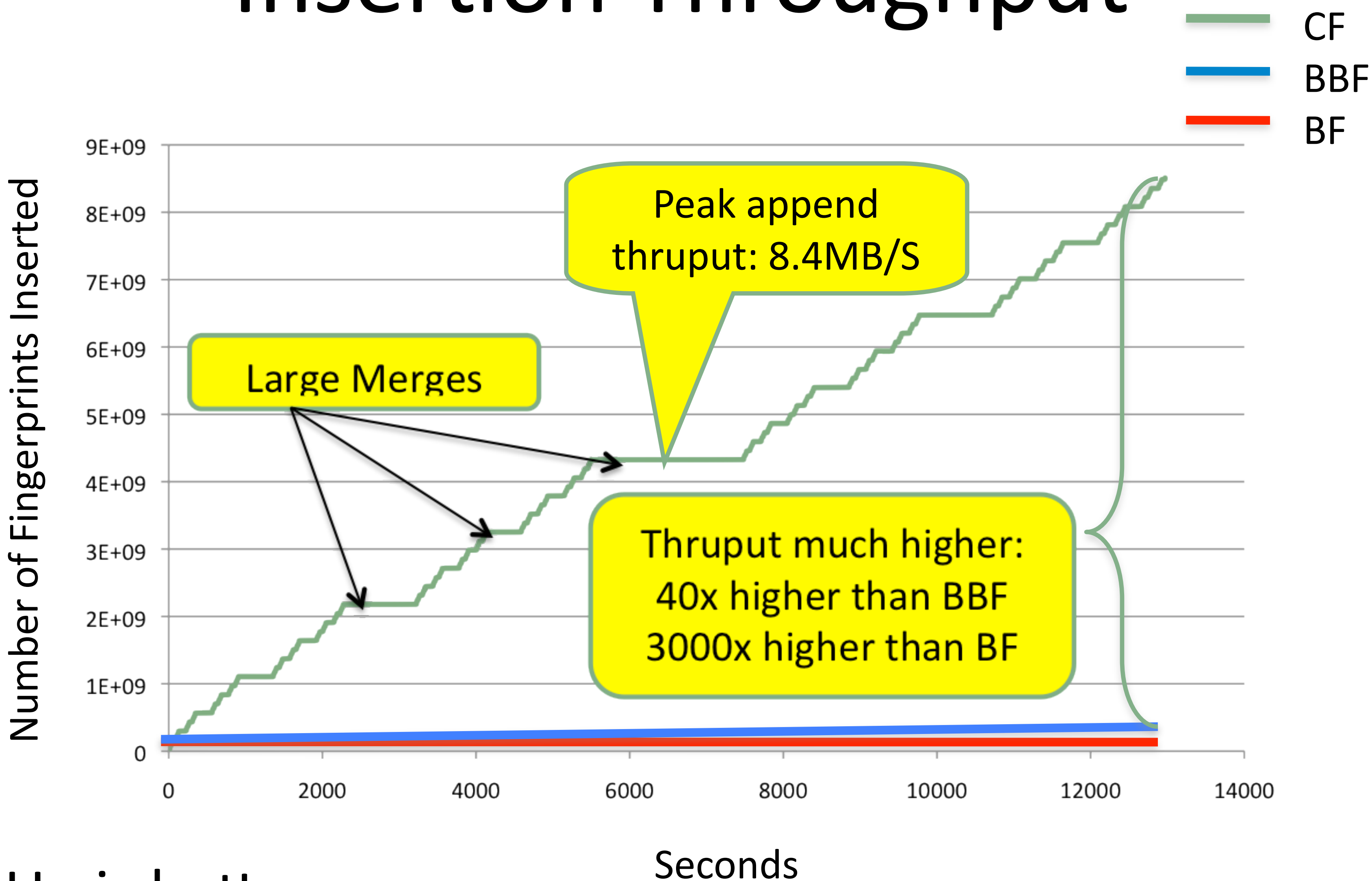
Insert/delete: $O((\log N/M)/B)$
Query: $O(\log N/M)$

Insertion Throughput



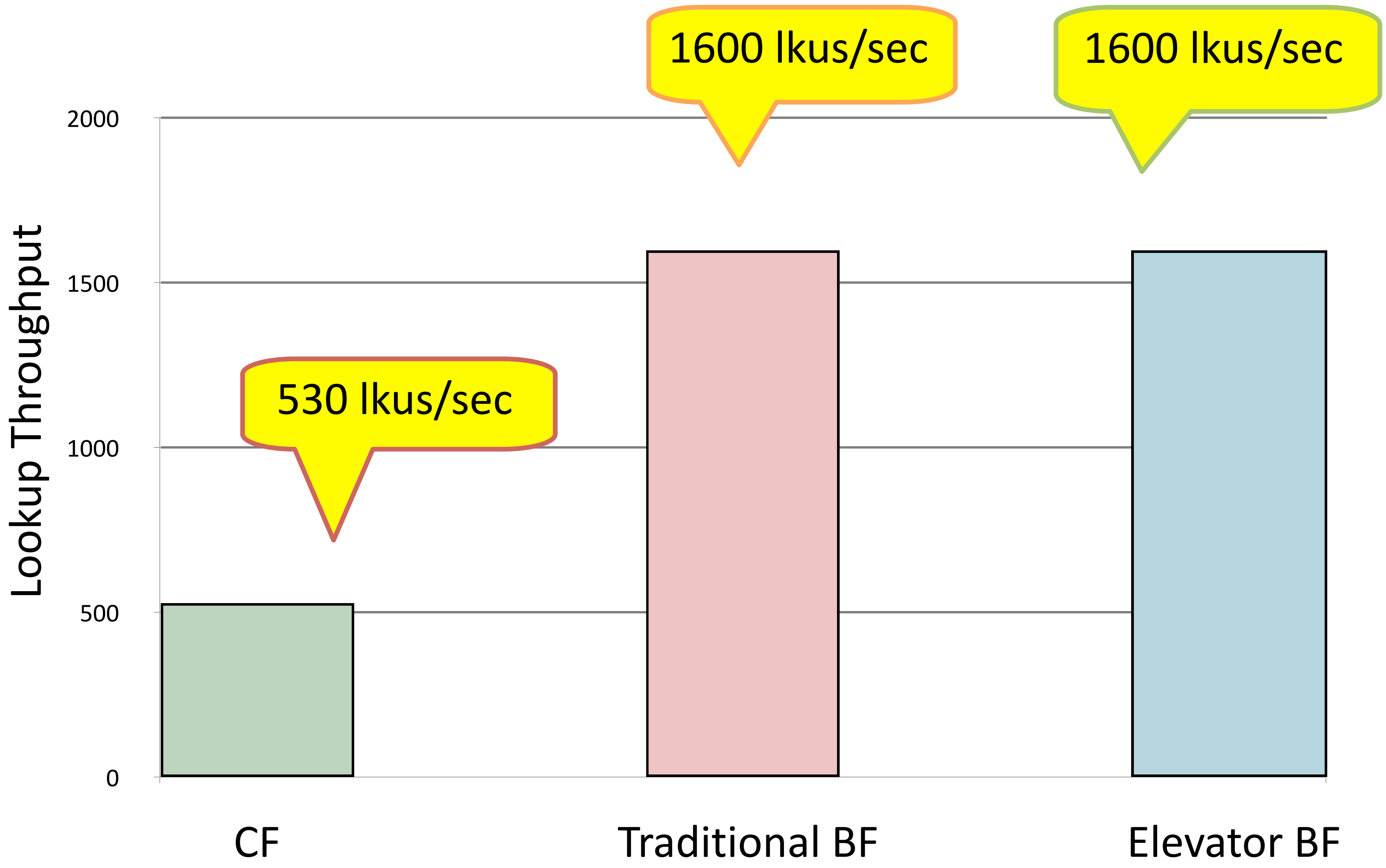
Up is better

Insertion Throughput



Up is better

Lookup Throughput



Don't Thrash: How to Cache Your Hash in Flash (**Results**)

Cascade Filter (CF), a BF replacement optimized for fast insertions/deletions on flash.

Write-optimized performance:

- 670,000 inserts/sec (3000x of Bloom, 40x best alternative)
- 530 lookups/sec (1/3x of best alternative)

The Cascade Filter is based upon Quotient Filters (QFs) instead of BF

- QFs have better access locality.
- QFs support deletes.

We can efficiently merge two QFs into a larger QF, which enables fast insertions.

Conclusion

Write optimized data structures are solving problems where people fill pain

- indexing in databases
- metadata maintenance on cloud/parallel file systems
- creating of tens of thousands of microfiles/sec/disk
- deduplication

Cascade filters (Approx membership queries) helps many of these applications

- queries on write-optimized systems
- insertions when there are uniqueness constraints