Courtesy: Kin Sum Liu and https://www.wordclouds.com/

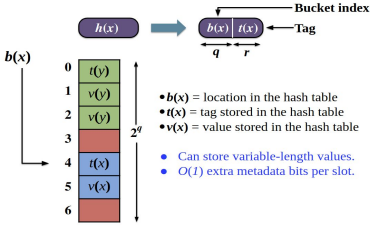
Fast and Space-Efficient Maps for Large Datasets

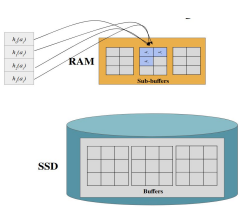
Prashant Pandey
Stony Brook University, NY

Advisors: Michael A. Bender and Rob Johnson

Thesis overview: data structures

Data structures



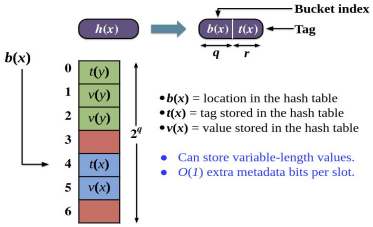


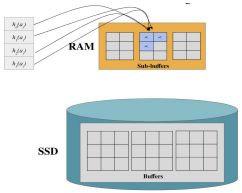
SIGMOD ‘17

ESA ‘18

Thesis overview: file system

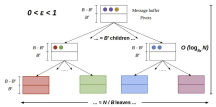
Data structures





SIGMOD ‘17

ESA ‘18



B$^\varepsilon$-tree

File systems

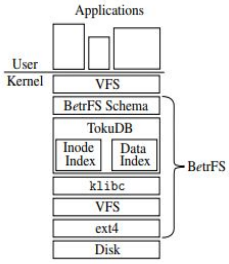
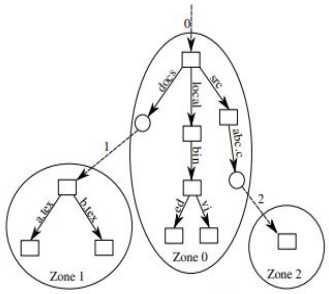


Figure 1: The BetrFS architecture.



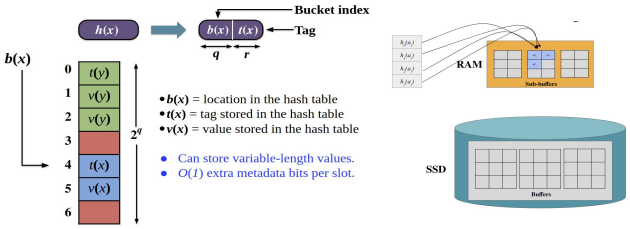
(a) An example zone tree in BetrFS 0.2.

FAST ‘15, TOS 15

FAST ‘16, TOS 16

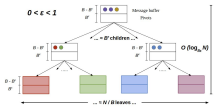
Thesis overview: computational biology

Data structures



SIGMOD ‘17

ESA ‘18



B^{ε}-tree

File systems

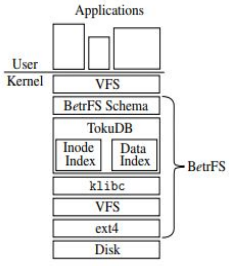
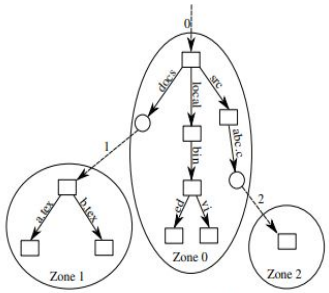


Figure 1: The BetrFS architecture.

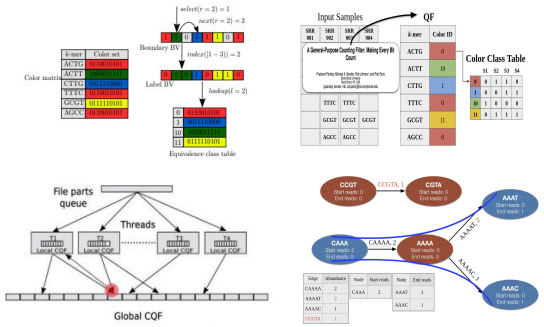


(a) An example zone tree in BetrFS 0.2.

FAST ‘15, TOS 15

FAST ‘16, TOS 16

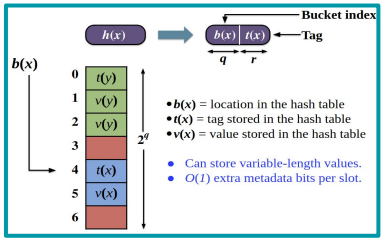
Computational biology

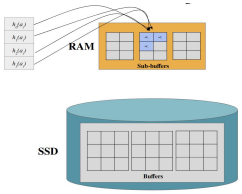


ISMB ‘17, BIOINFORMATICS ‘17, WABI ‘17, RECOMB ‘18, Cell Systems ‘18

Thesis overview: computational biology

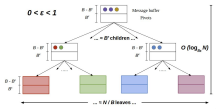
Data structures





SIGMOD ‘17

ESA ‘18



B$^{\varepsilon}$-tree

File systems

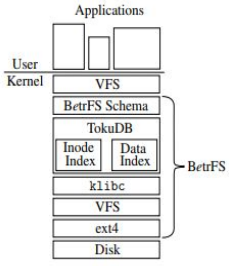
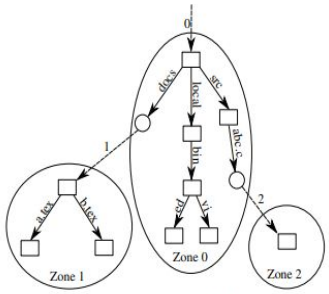
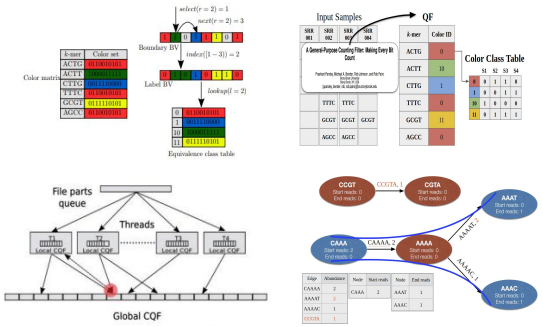


Figure 1: The BetrFS architecture.



(a) An example zone tree in BetrFS 0.2.

Computational biology



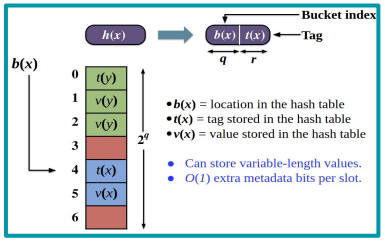
FAST ‘15, TOS 15

FAST ‘16, TOS 16

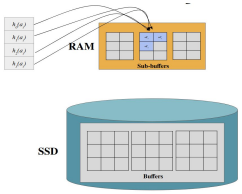
ISMB ‘17, BIOINFORMATICS ‘17, WABI ‘17, RECOMB ‘18, Cell Systems ‘18

Thesis overview: streaming

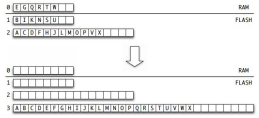
Data structures



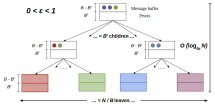
SIGMOD ‘17



ESA ‘18



Cascade filter



B$^\varepsilon$-tree

File systems

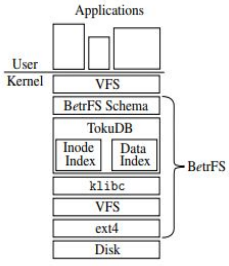
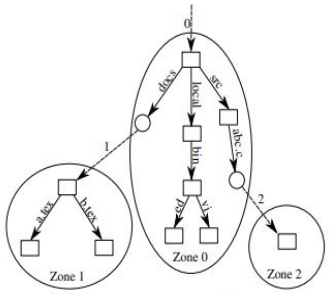


Figure 1: The BetrFS architecture.

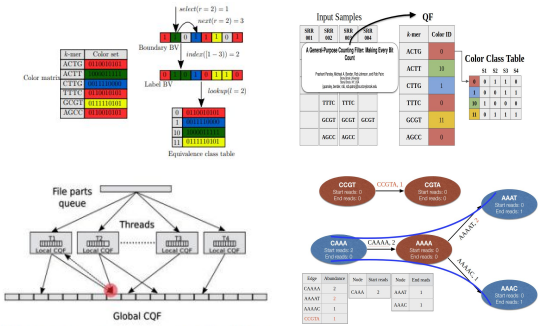


(a) An example zone tree in BetrFS 0.2.

FAST ‘15, TOS 15

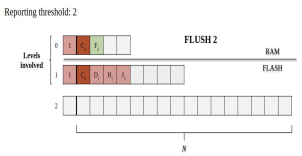
FAST ‘16, TOS 16

Computational biology



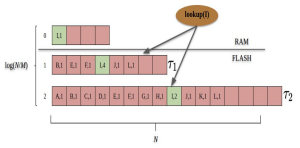
ISMB ‘17, BIOINFORMATICS ‘17, WABI ‘17, RECOMB ‘18, Cell Systems ‘18

Streaming



$$I_1 = 0, I_T = 4, I_R = 5$$

$$\alpha = \frac{5-1}{4-1} = 1.33$$

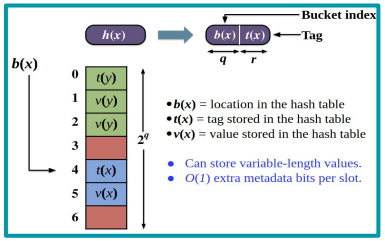


• The maximum count stretch ω would be:

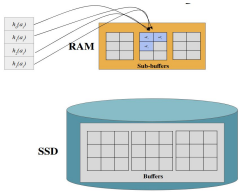
$$\omega = \frac{T + \sum_{i=1}^{L} \tau_i}{T}$$

Thesis overview: streaming

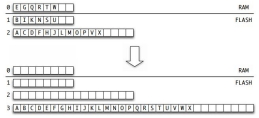
Data structures



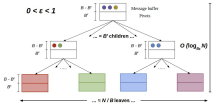
SIGMOD ‘17



ESA ‘18



Cascade filter



B$^\varepsilon$-tree

File systems

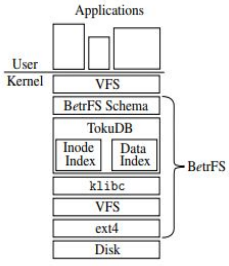
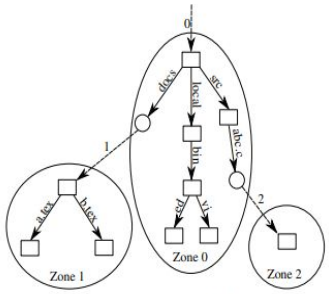


Figure 1: The BetrFS architecture.

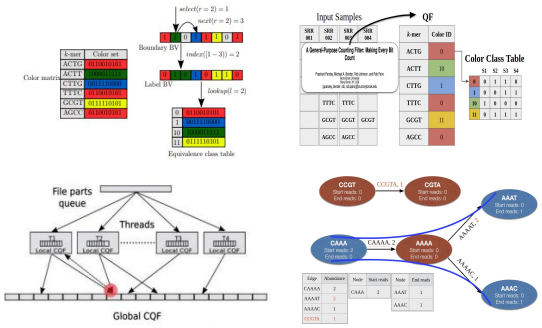


(a) An example zone tree in BetrFS 0.2.

FAST ‘15, TOS 15

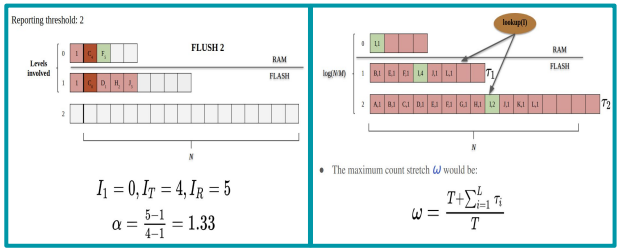
FAST ‘16, TOS 16

Computational biology



ISMB ‘17, BIOINFORMATICS ‘17, WABI ‘17, RECOMB ‘18, Cell Systems ‘18

Streaming

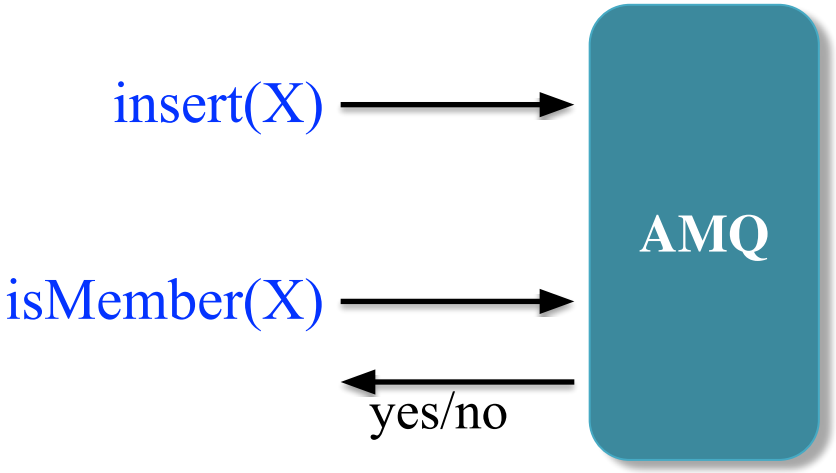


$$I_1 = 0, I_T = 4, I_R = 5$$

$$\alpha = \frac{5-1}{4-1} = 1.33$$

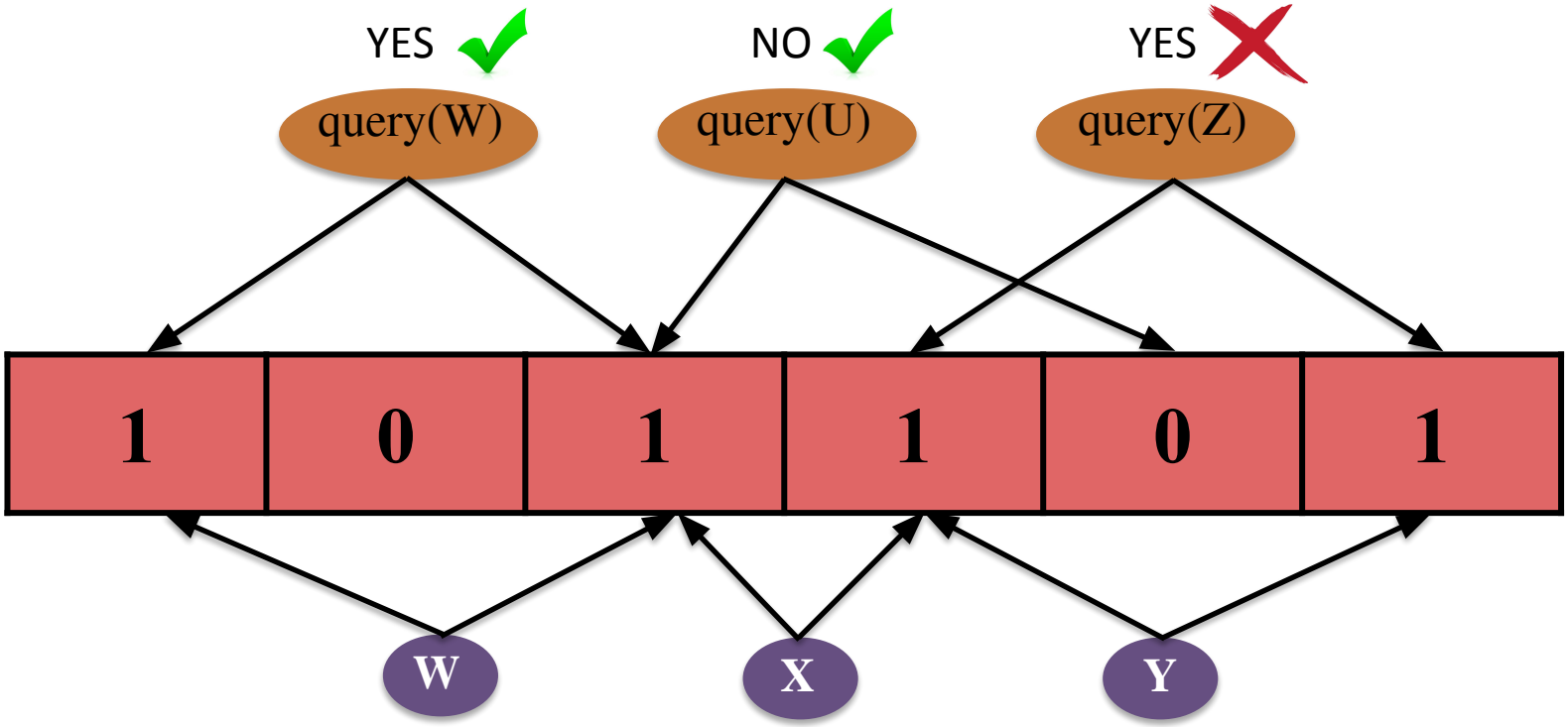
$$\omega = \frac{T + \sum_{i=1}^{L} \tau_i}{T}$$

Approximate Membership Query (AMQ)



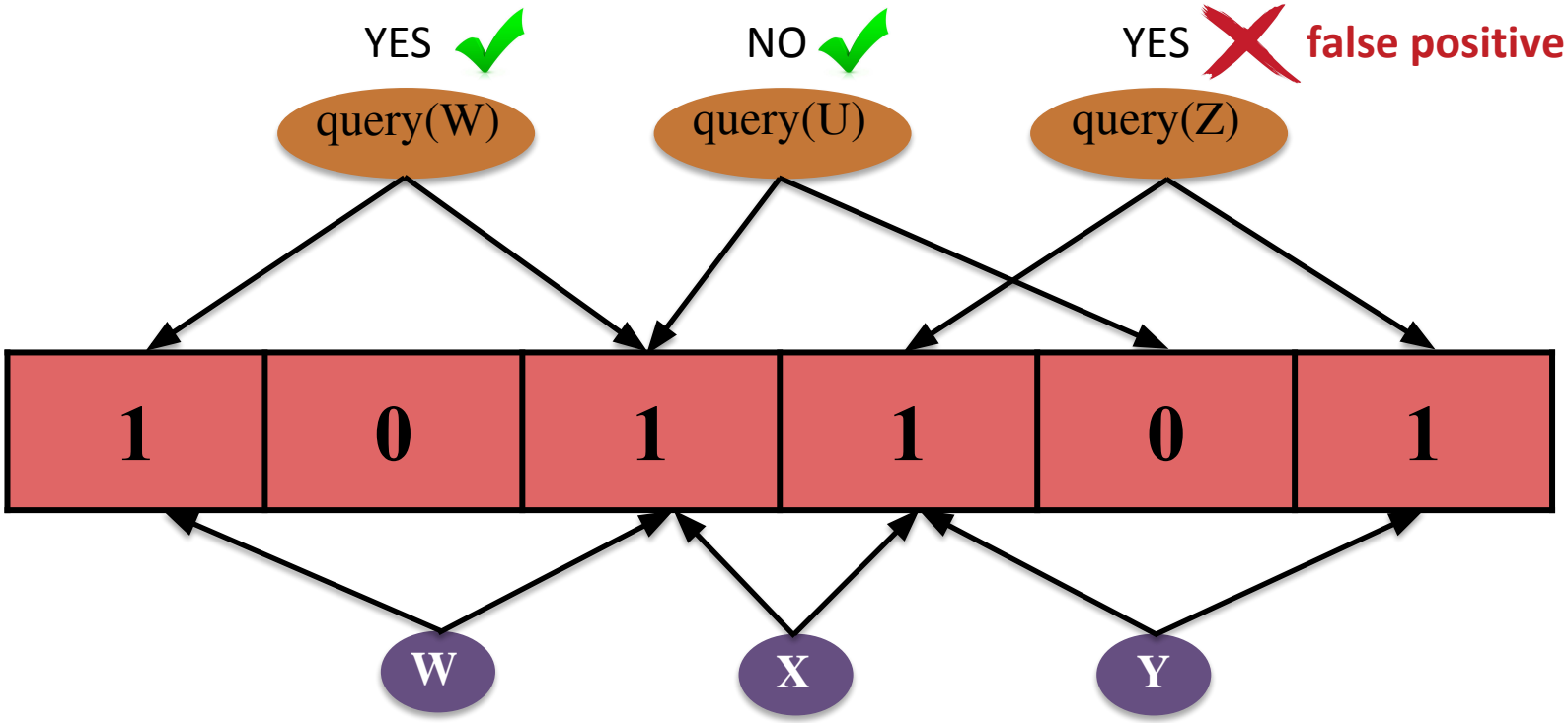
- **An AMQ is a lossy representation of a set.**
- **Operations: inserts and membership queries.**
- **Compact space:**
  - Often taking < 1 byte per item.
  - Comes at the cost of occasional false positives.

Bloom filters



The Bloom filter is a bit array + k hash functions.

Bloom filters have one-sided errors



The Bloom filter has a bounded false-positive rate.

Bloom filters are ubiquitous

Streaming applications

Networking

Databases

Computational biology

Storage systems

Application often must work around Bloom filter limitations.

Limitations	Workarounds
No deletions	Rebuild
No resizes	Guess, rebuild if wrong
No enumeration (merging)	???
No values	Combine with another data structure

Application often must work around Bloom filter limitations.

Limitations	Workarounds
No deletions	Rebuild
No resizes	Guess, rebuild if wrong
No enumeration (merging)	???
No values	Combine with another data structure

Bloom filter limitations increase complexity, waste space, and hurt application performance.

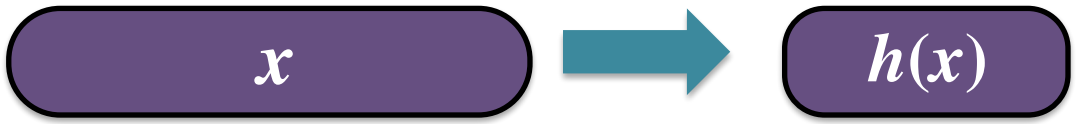
The Quotient filter (QF)

- A replacement for the (counting) Bloom filter.
- Space and computationally efficient.
- Can be used as a map for small key-value pairs.
- Uses variable-sized counts/values.

$$\text{QF space} \leq \text{BF space} + O(\sum |v(x)|)$$

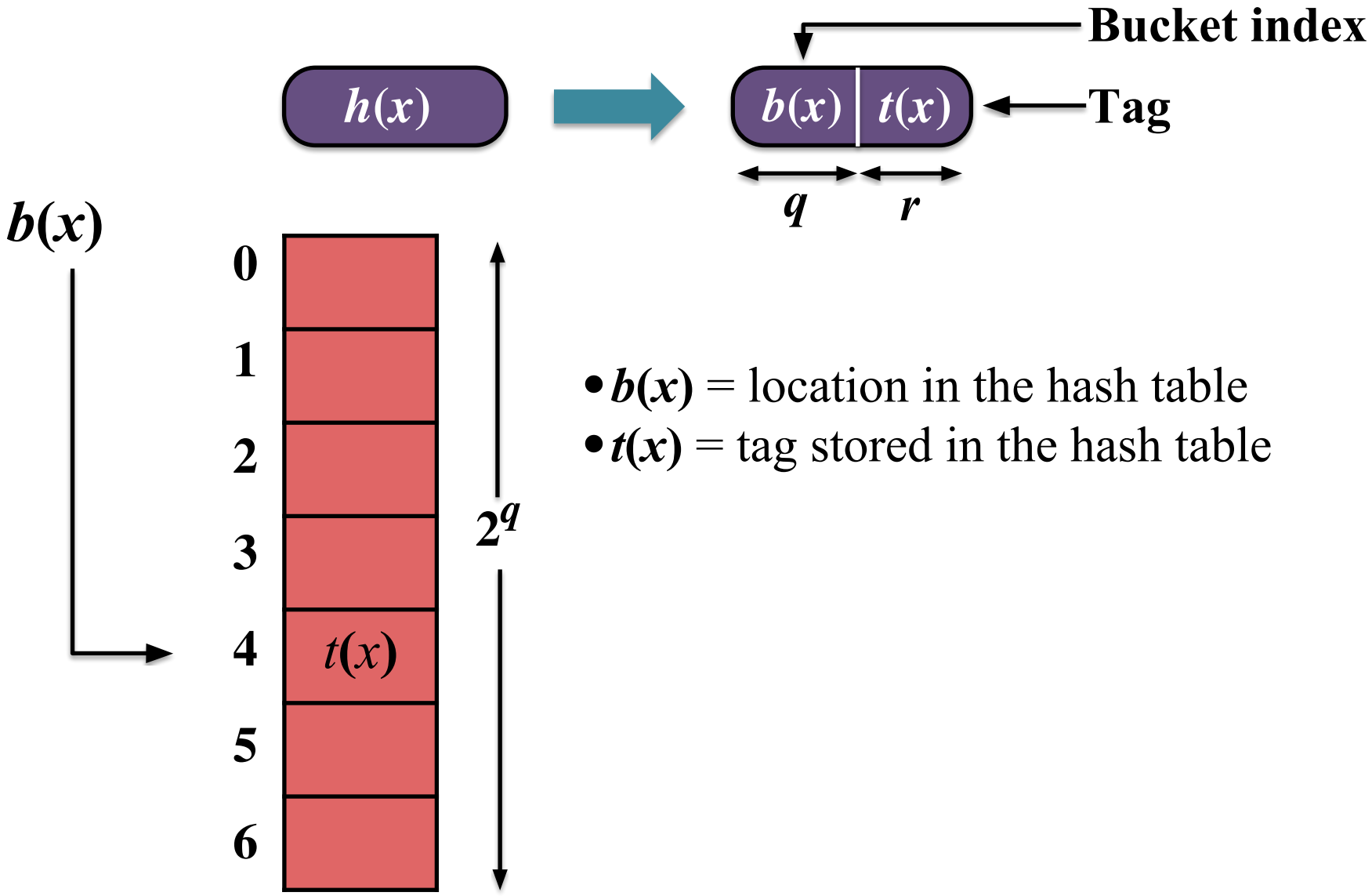
Quotienting: an alternative to Bloom filters [Knuth. *Sorting and Searching* volume 3, '97]

- **Store fingerprint compactly in a hash table.**
  - Take a fingerprint ***h(x)*** for each element ***x***.



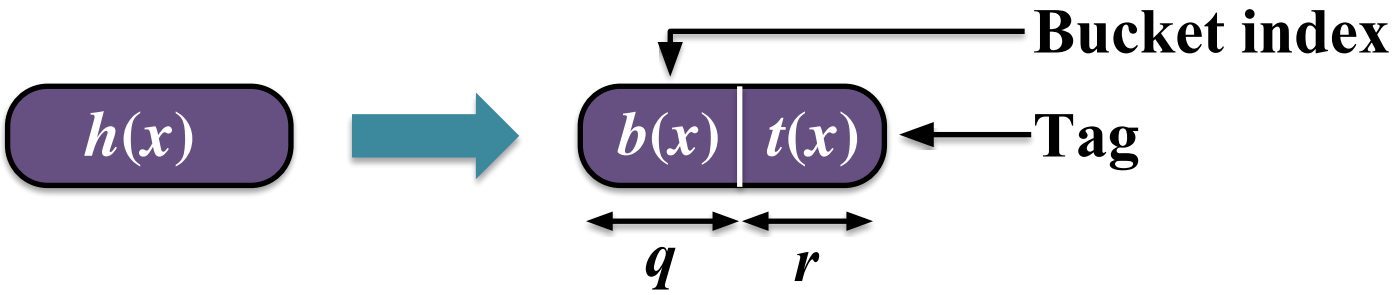
- **Only source of false positives:**
  - Two distinct elements ***x*** and ***y***, where ***h(x)*** = ***h(y)***.
  - If ***x*** is stored and ***y*** isn't, ***query(y)*** gives a false positive.

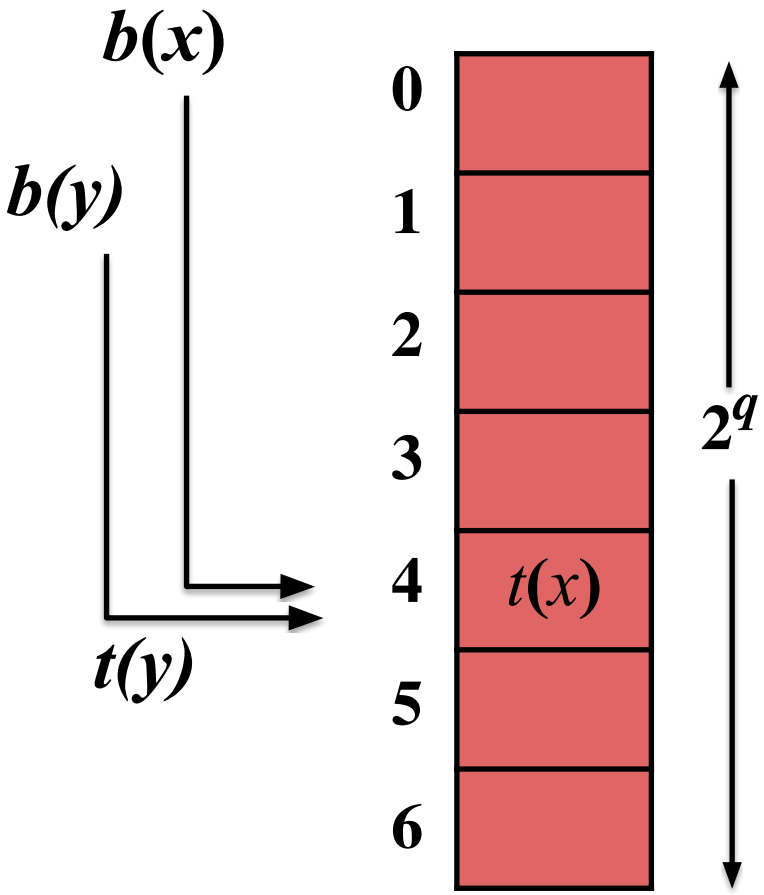
Storing fingerprints compactly



- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Storing fingerprints compactly

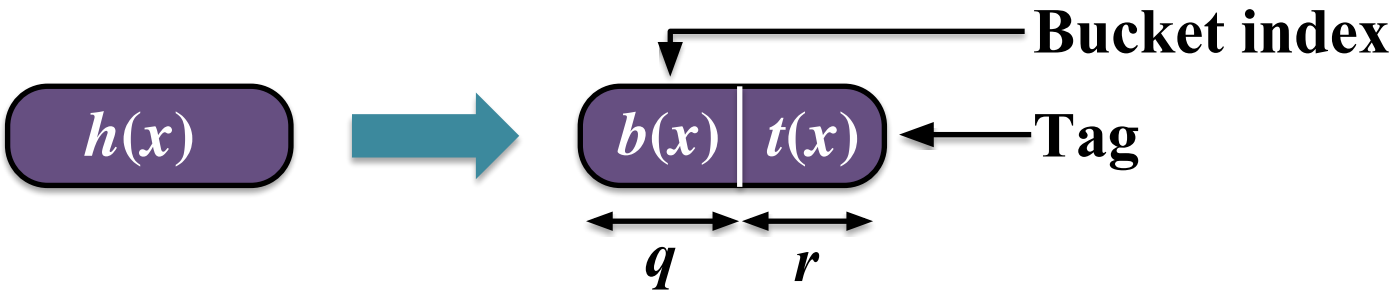


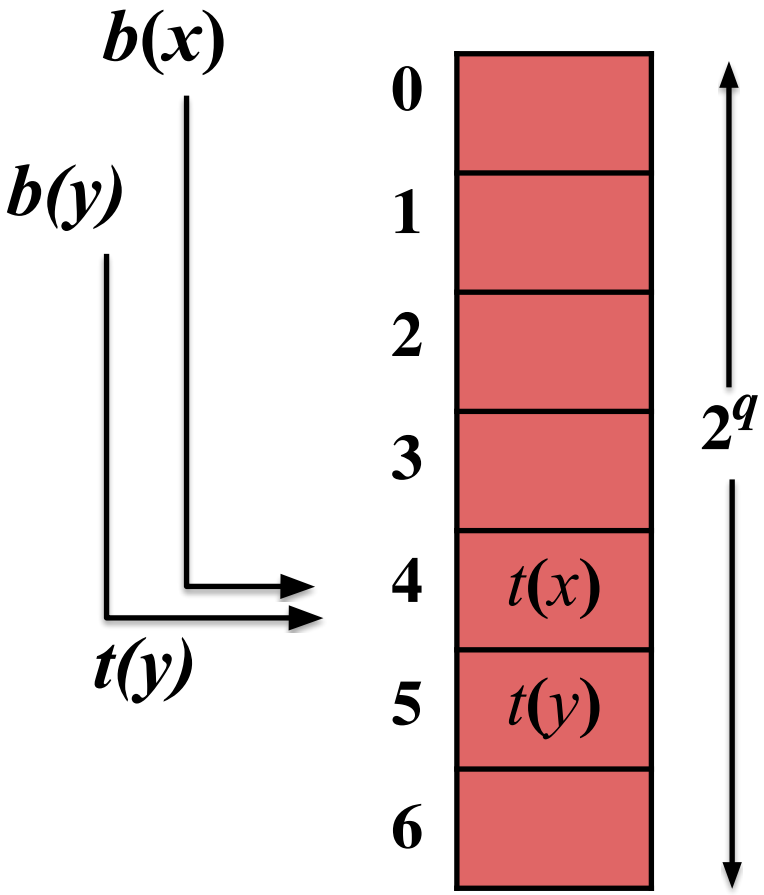


- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Collisions in the hash table?

Storing fingerprints compactly



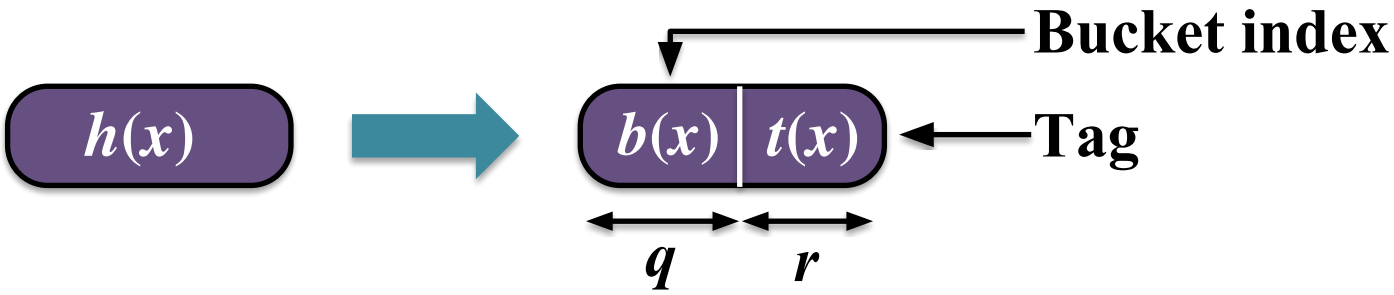


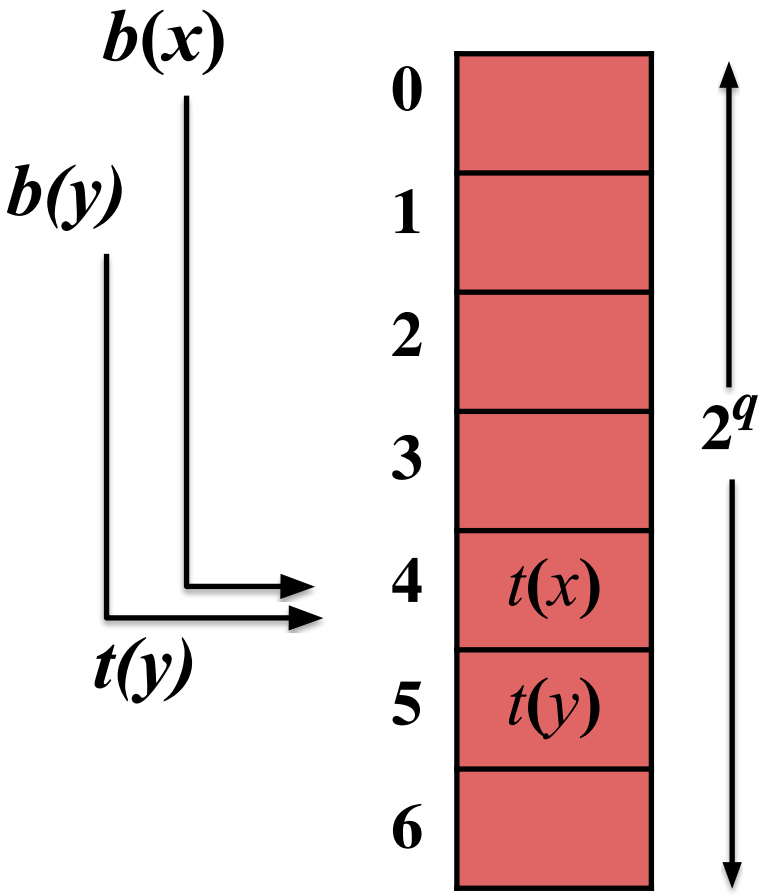
- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Collisions in the hash table?

- Linear probing.

Storing fingerprints compactly





- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

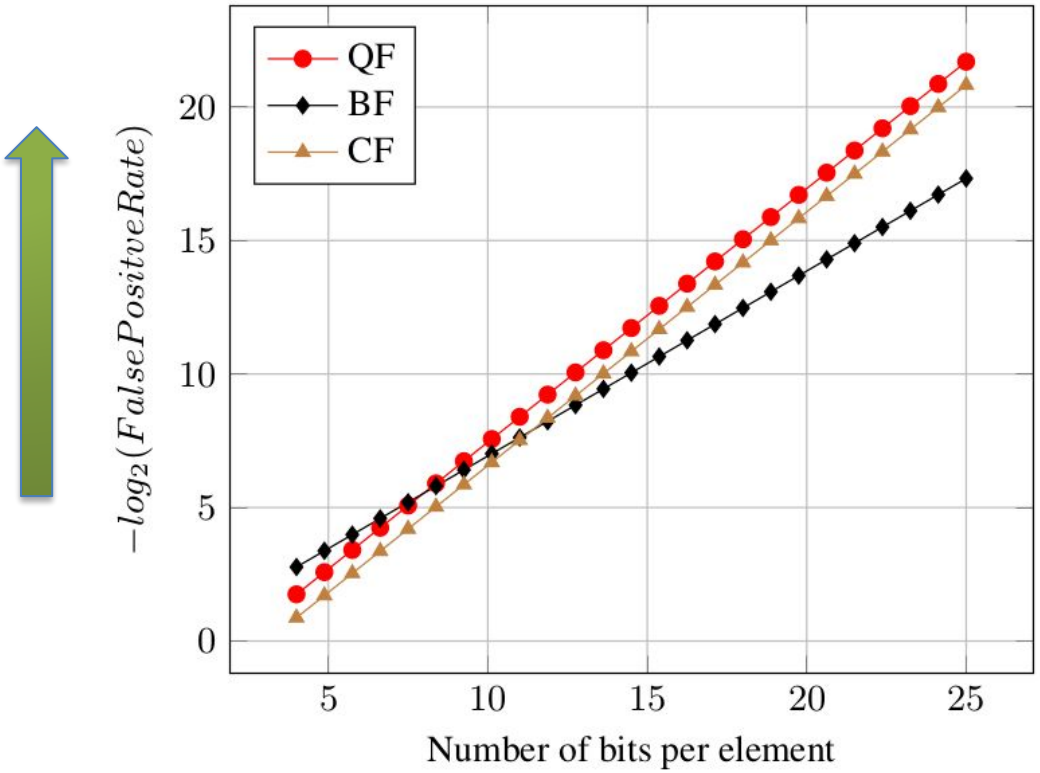
Collisions in the hash table?

- Linear probing.
- $O(1)$ metadata bits per slot.

Quotienting enables many features in the QF

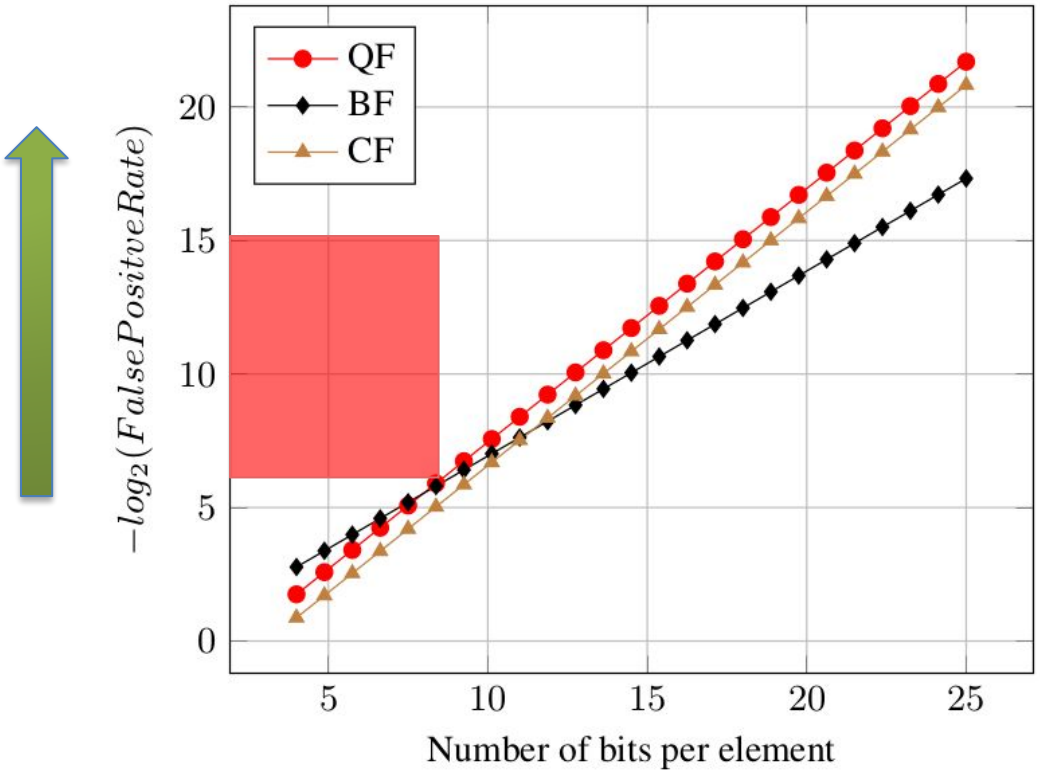
- **Good cache locality**
- **Efficient scaling out-of-RAM**
- **Deletions**
- **Enumerability/Mergeability**
- **Resizing**

Quotient filters (QF) use comparable space to Bloom filters (BF)



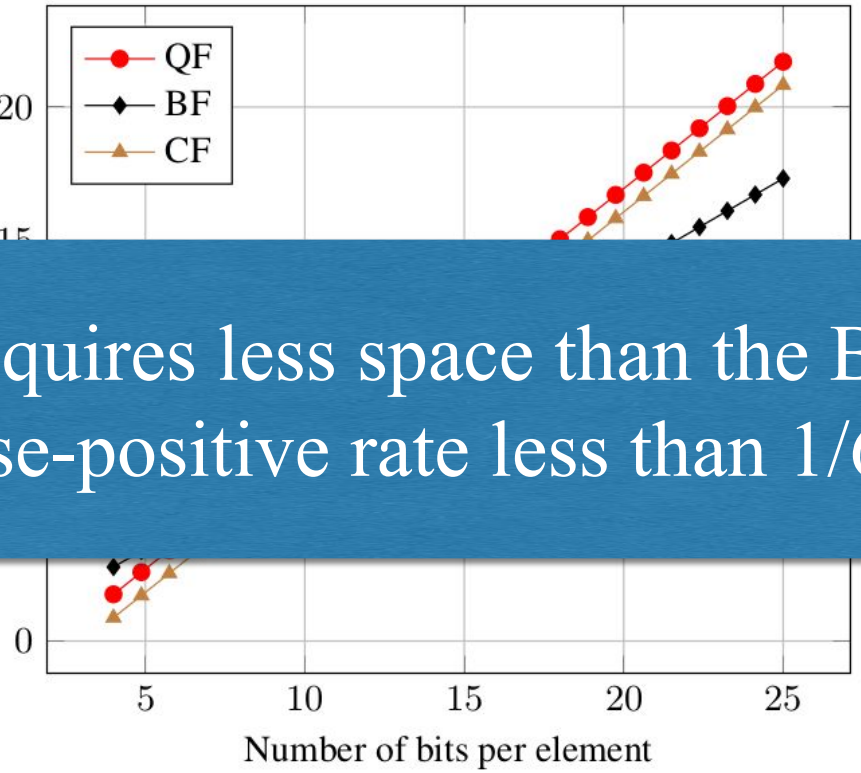
Bloom filters: ~$1.44log_2(1/\varepsilon)$ bits/element.
Quotient filters: ~$2.125 + log_2(1/\varepsilon)$ bits/element.

Quotient filters (QF) use comparable space to Bloom filters (BF)



Bloom filters: ~$1.44log_2(1/\varepsilon)$ bits/element.
Quotient filters: ~$2.125 + log_2(1/\varepsilon)$ bits/element.

Quotient filters (QF) use comparable space to Bloom filters (BF)

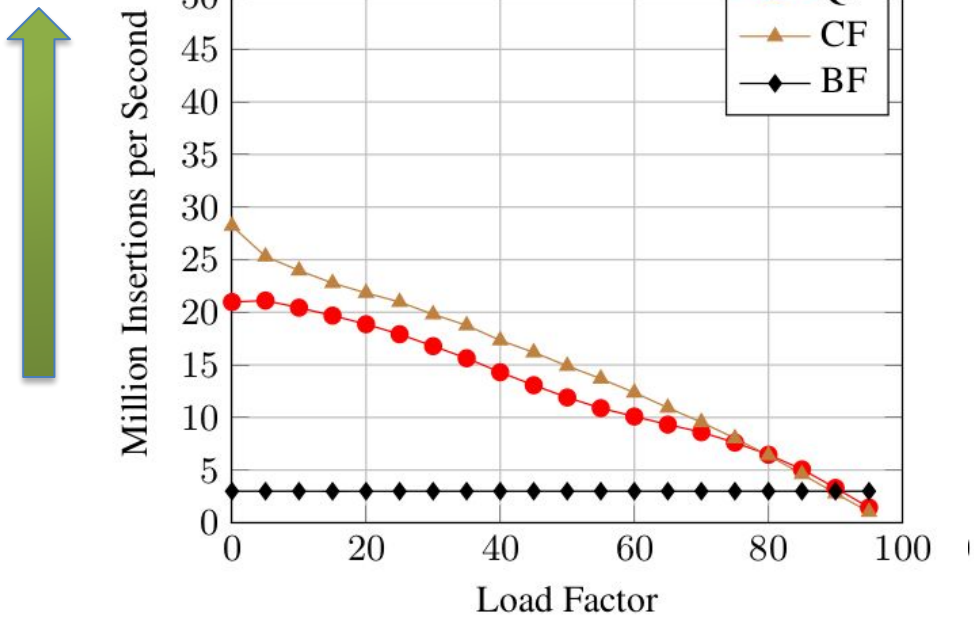


The QF requires less space than the BF for any false-positive rate less than 1/64

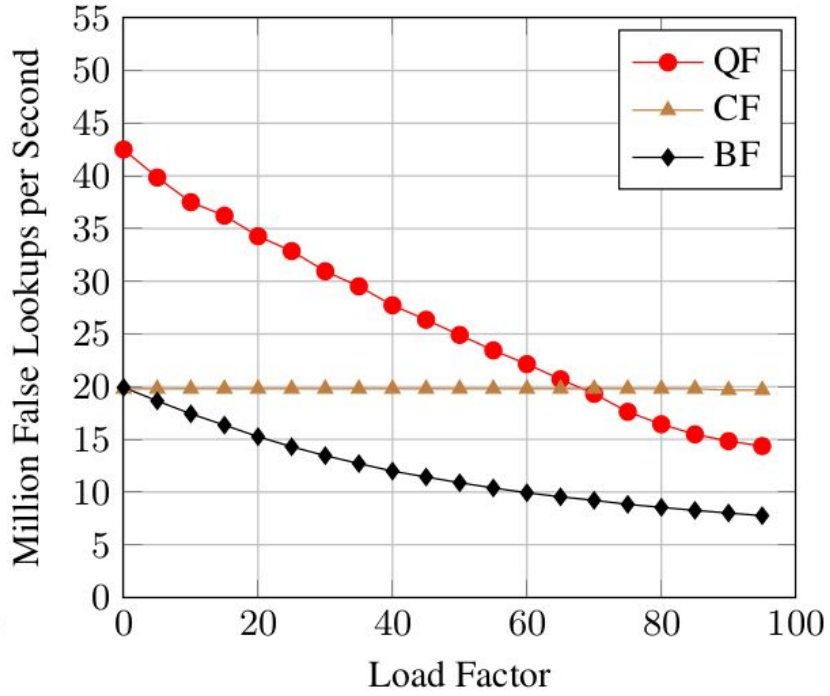
Bloom filters: $\sim 1.44 \log_2(1/\varepsilon)$ bits/element.
Quotient filters: $\sim 2.125 + \log_2(1/\varepsilon)$ bits/element.

Performance: In memory

Inserts



Lookups

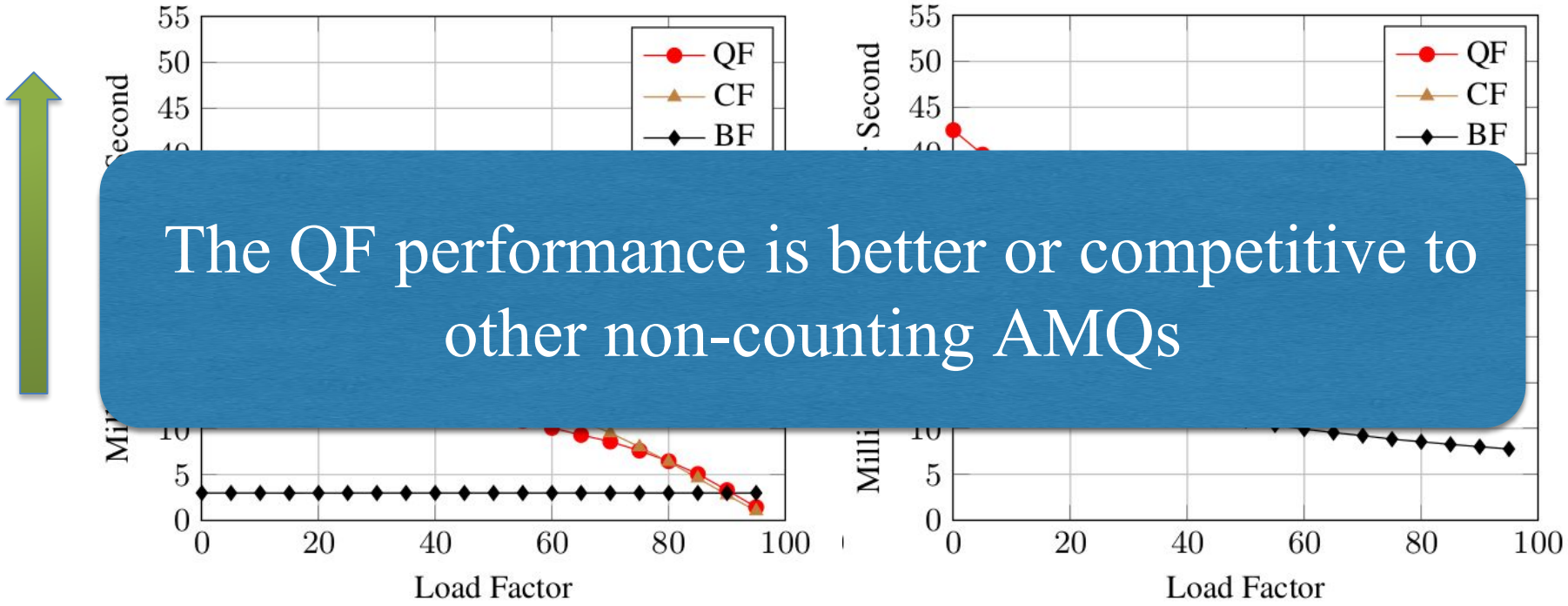


- The QF insert performance in RAM is similar to that of the state-of-the-art non-counting AMQ.
- The QF query performance is significantly fast at low load-factors and slightly slower at higher load-factors.

Performance: In memory

Inserts

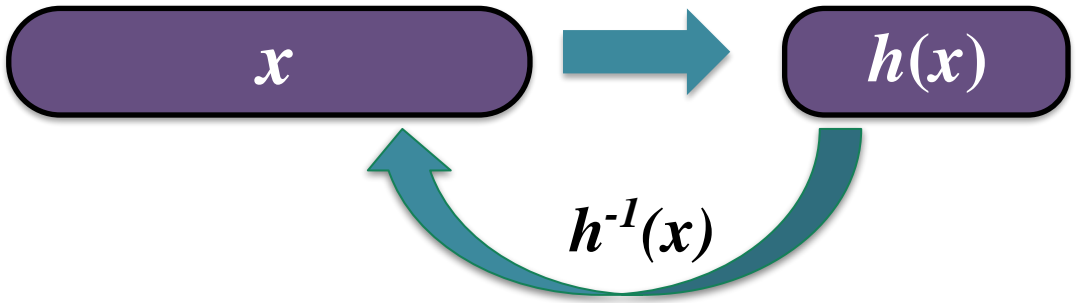
Lookups



- The QF insert performance in RAM is similar to that of the state-of-the-art non-counting AMQ.
- The QF query performance is significantly fast at low load-factors and slightly slower at higher load-factors.

Quotient filters can also be exact

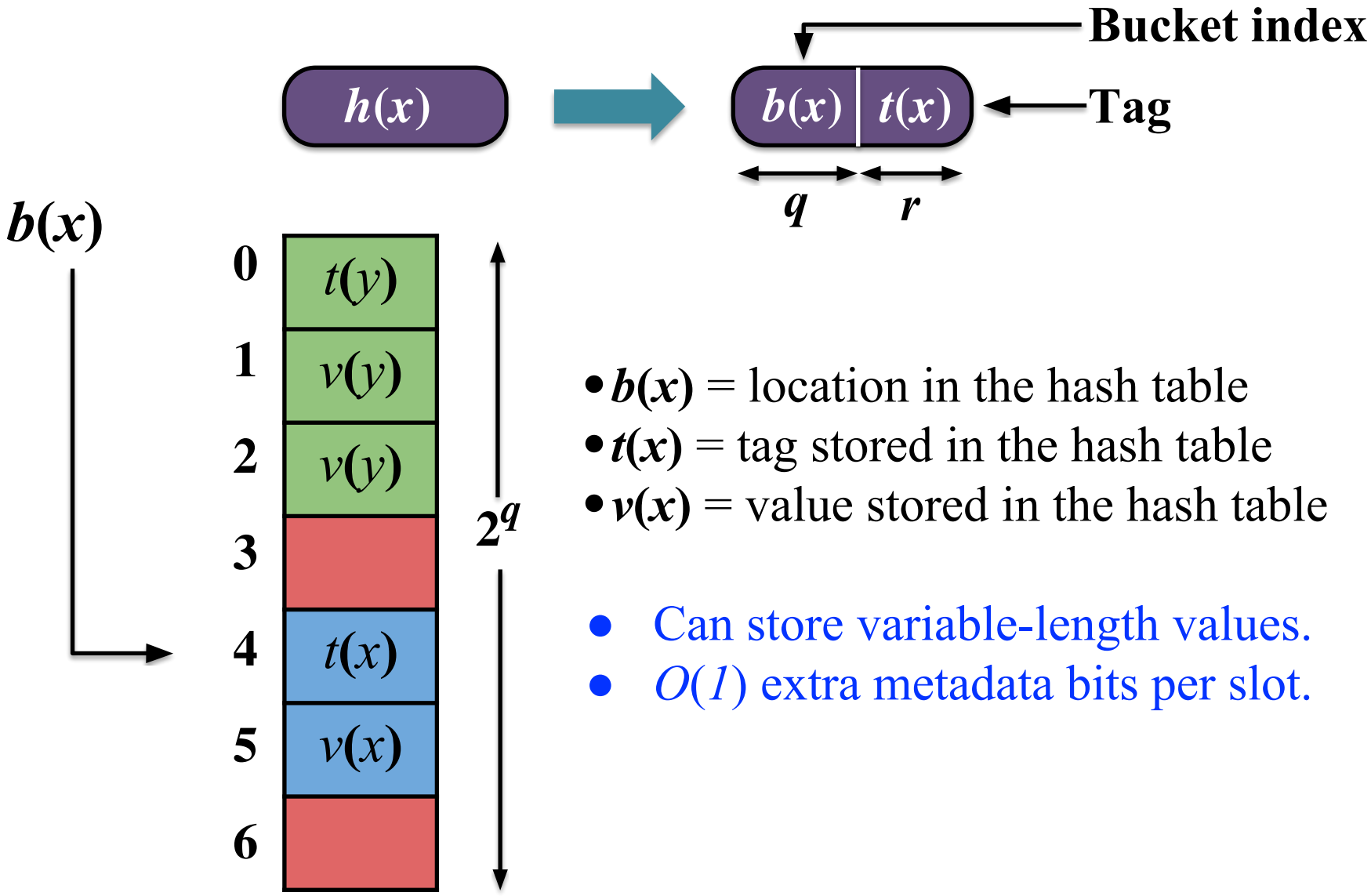
- **Quotient filters store *h(x)* exactly.**
- **To store *x* exactly, use an invertible hash function.**



- **For *n* elements and *p*-bit hash function:**

  Space usage: ~*p*-log$_2$*n* bits/element.

Storing key-value pairs

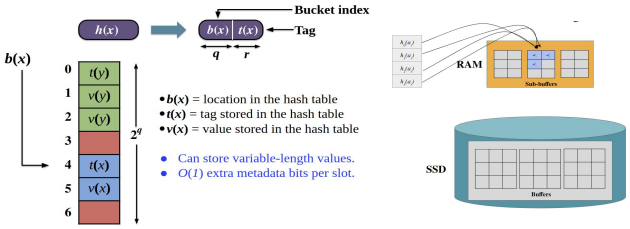


- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table
- $v(x)$ = value stored in the hash table

- Can store variable-length values.
- $O(1)$ extra metadata bits per slot.

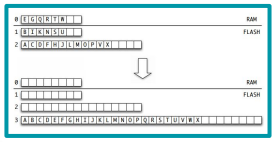
Thesis overview: cascade filter

Data structures

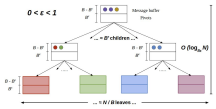


SIGMOD ‘17

ESA ‘18



Cascade filter



B$^\varepsilon$-tree

File systems

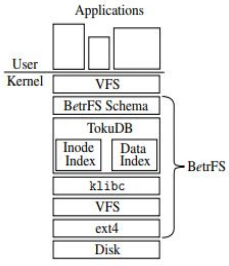
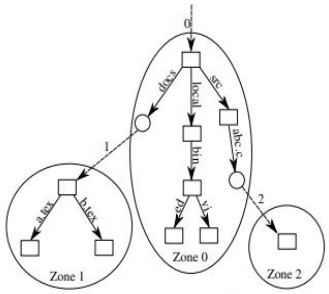


Figure 1: The BetrFS architecture.

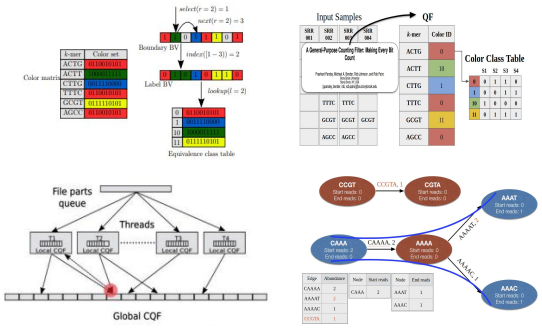


(a) An example zone tree in BetrFS 0.2.

FAST ‘15, TOS 15

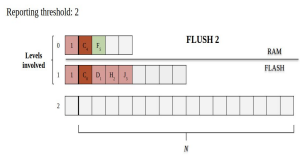
FAST ‘16, TOS 16

Computational biology



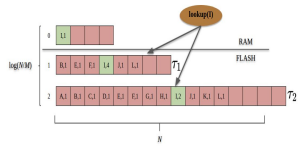
ISMB ‘17, BIOINFORMATICS ‘17, WABI ‘17, RECOMB ‘18, Cell Systems ‘18

Streaming



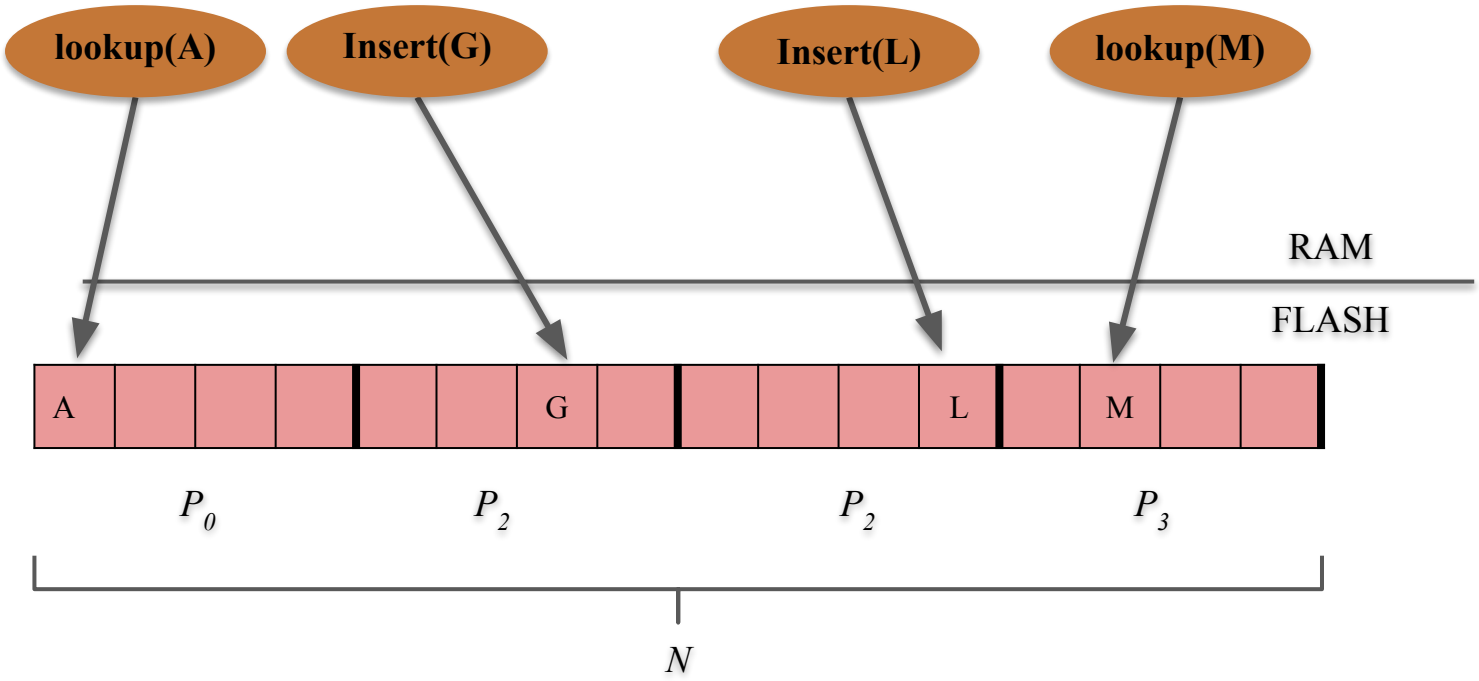
$I_1 = 0, I_T = 4, I_R = 5$

$\alpha = \frac{5-1}{4-1} = 1.33$



$\omega = \frac{T + \sum_{i=1}^{L} \tau_i}{T}$

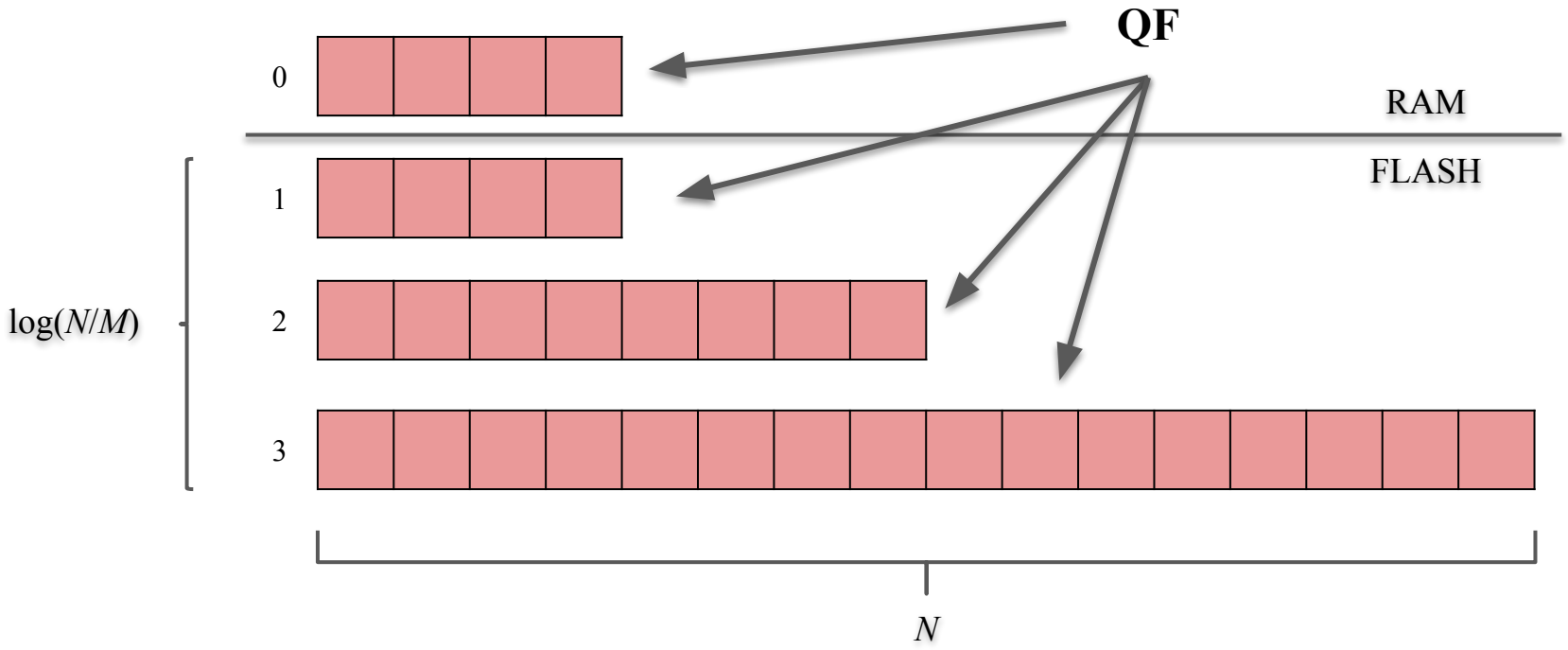
Quotient filters scale out-of-RAM



- Each operation, insert or lookup, can cost 1 I/O.
- However, a Bloom filter takes K I/Os for each operation.

Cascade filter uses write-optimization

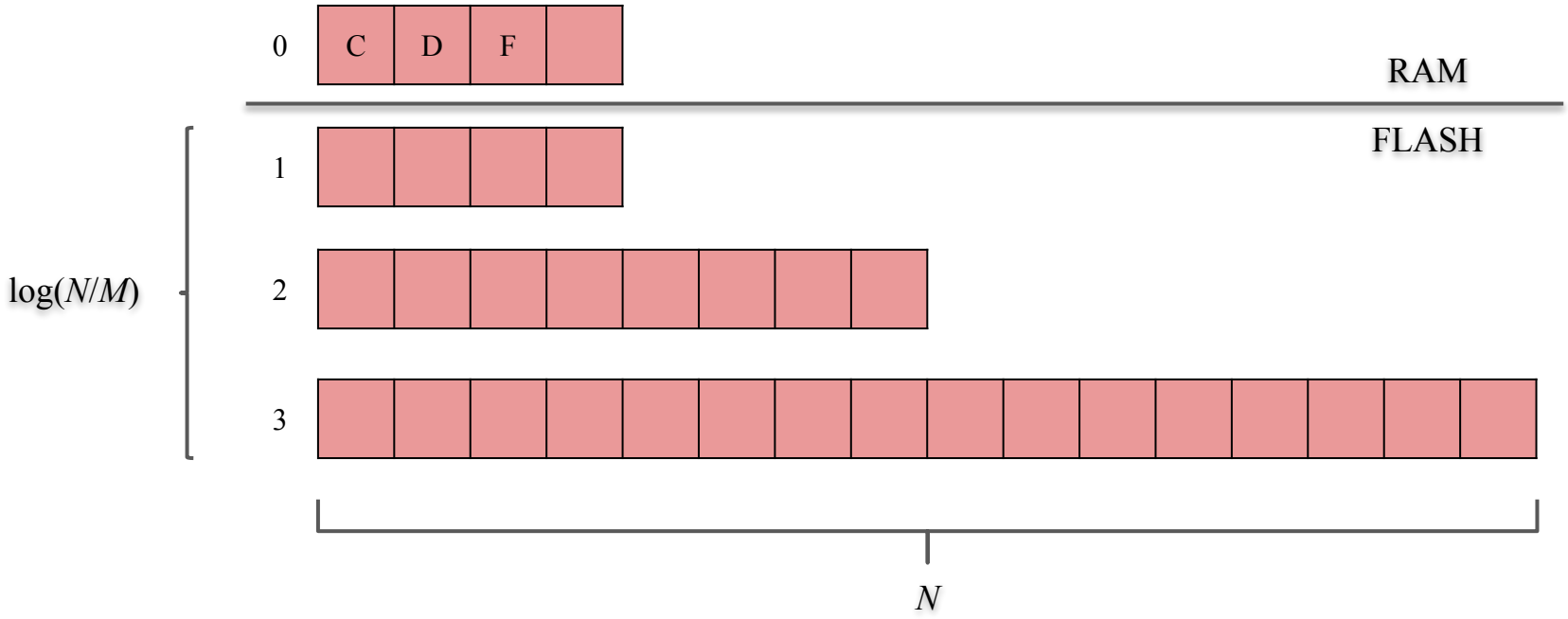
[Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok '12]



- The cascade filter is a write-optimized data structure and efficiently scales out of RAM.
- It greatly accelerates insertions at some cost to queries.

Cascade filter uses write-optimization

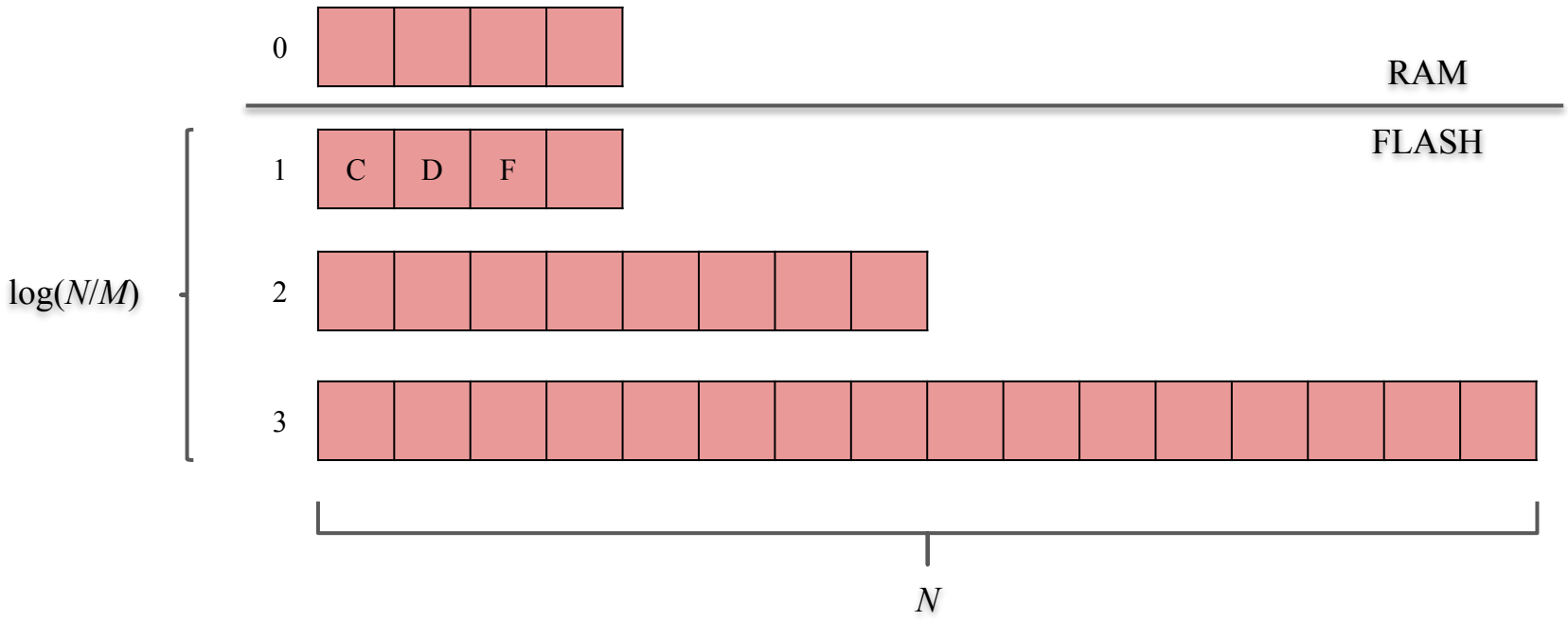
[Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok '12]



- Items are first inserted into the in-memory QF.
- When the in-memory QF reaches maximum load factor it flushes.
- Levels grow exponentially in size.

Cascade filter uses write-optimization

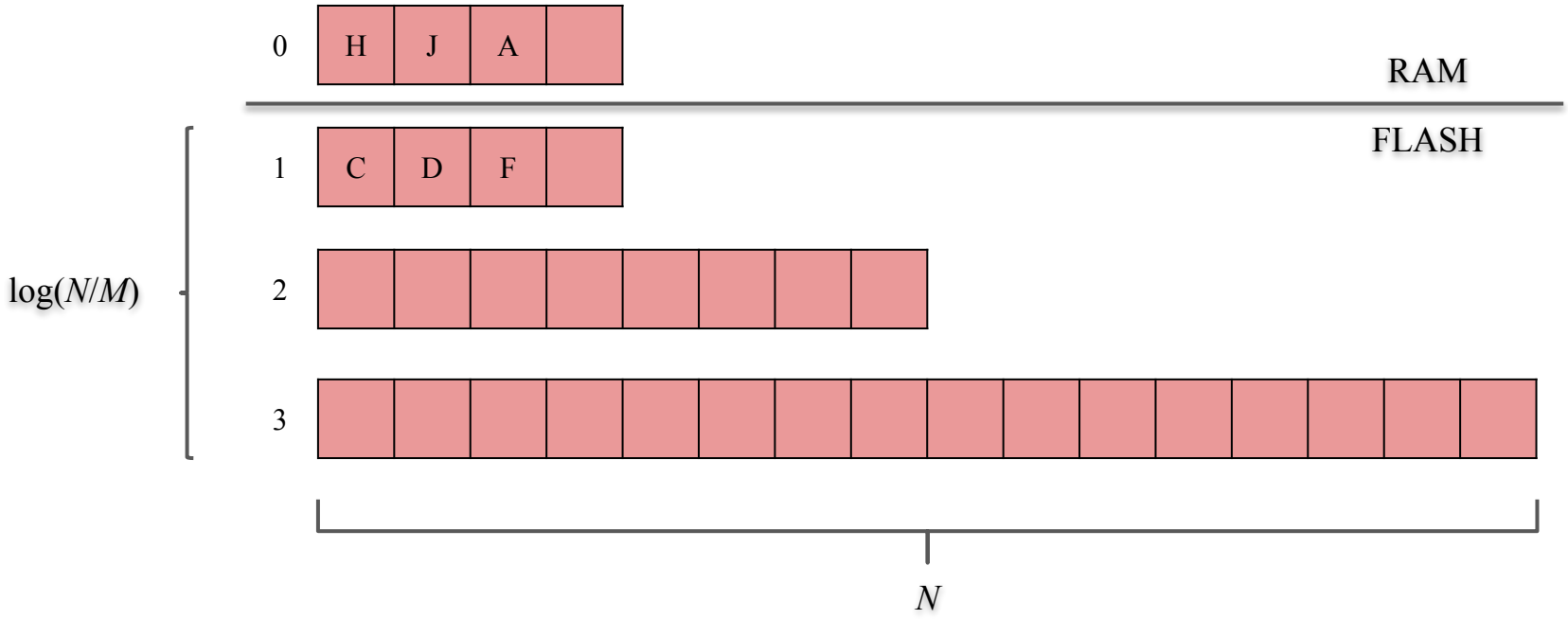
[Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok '12]



- During a flush, find the smallest i such that the items in $l_0, \ldots, l_i$ can be merged into level i.

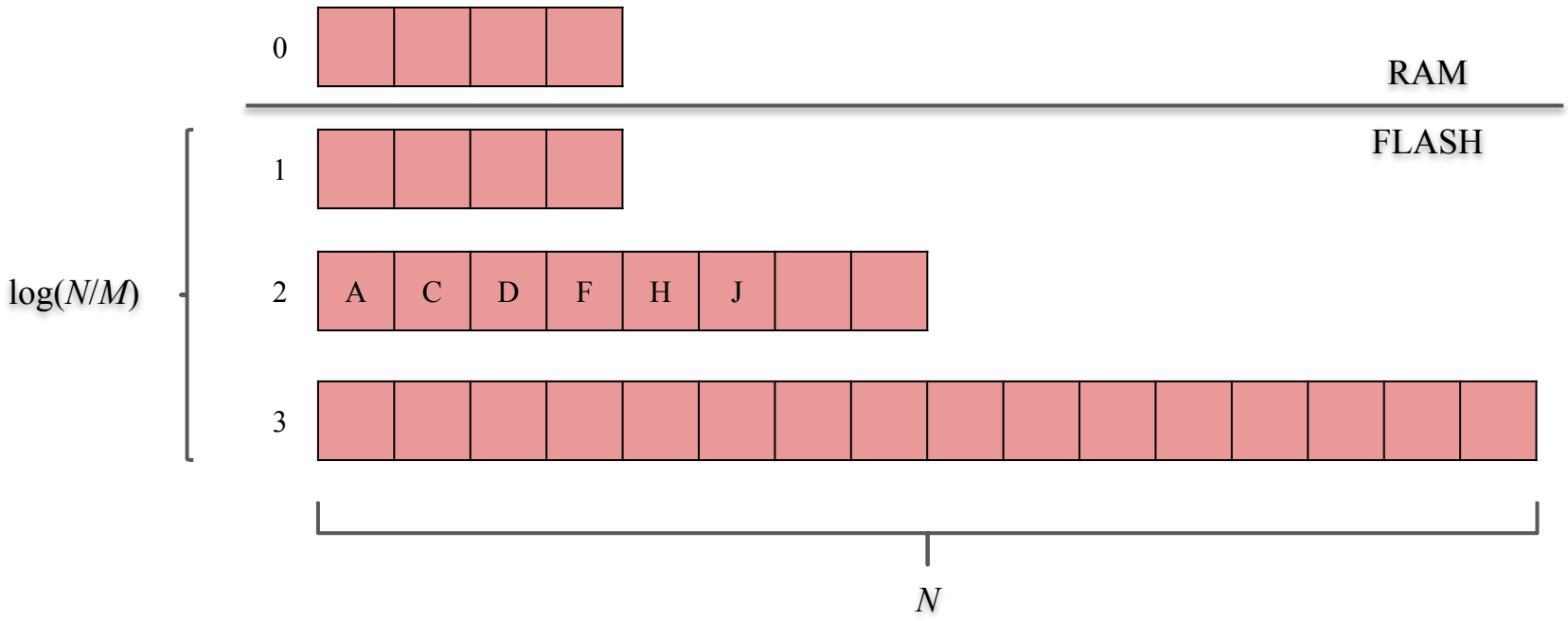
Cascade filter uses write-optimization

[Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok '12]

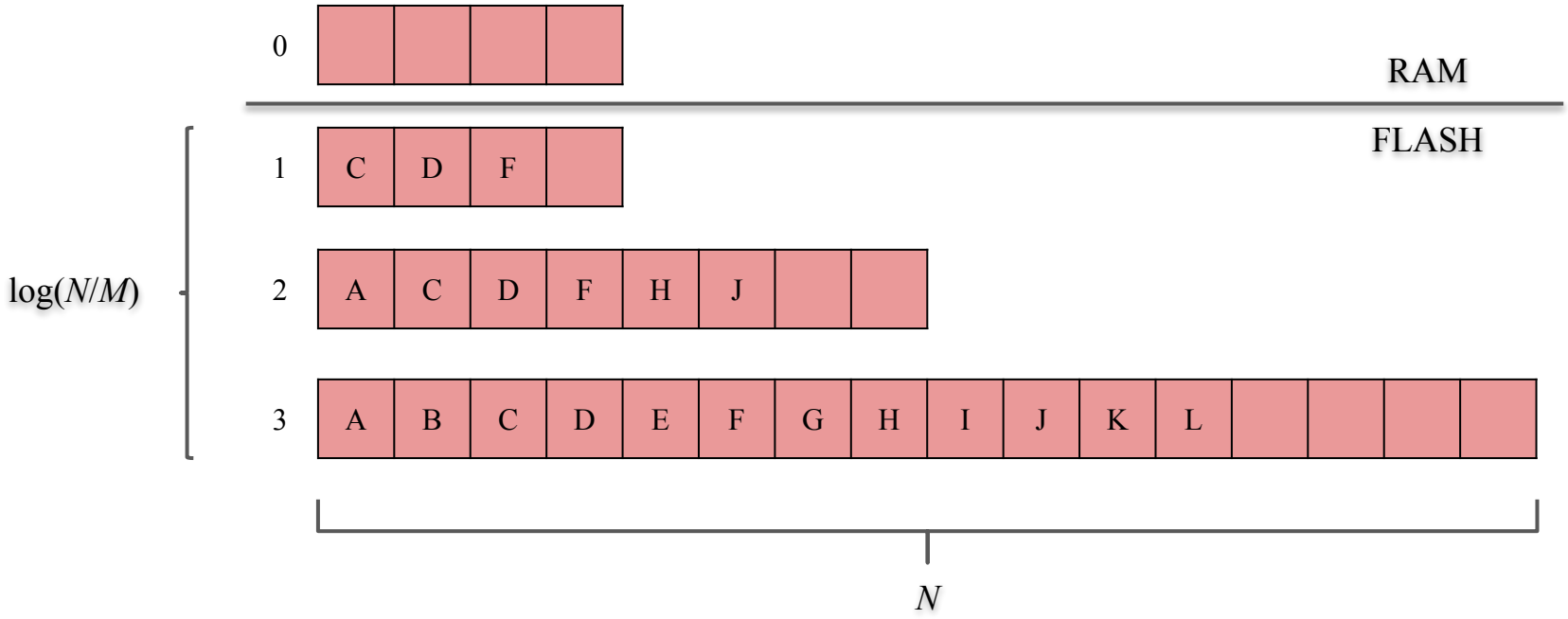


Cascade filter uses write-optimization

[Bender, Farach-Colton, Johnson, Kraner, Kuszmaul, Medjedovic, Montes, Shetty, Spillane, Zadok '12]

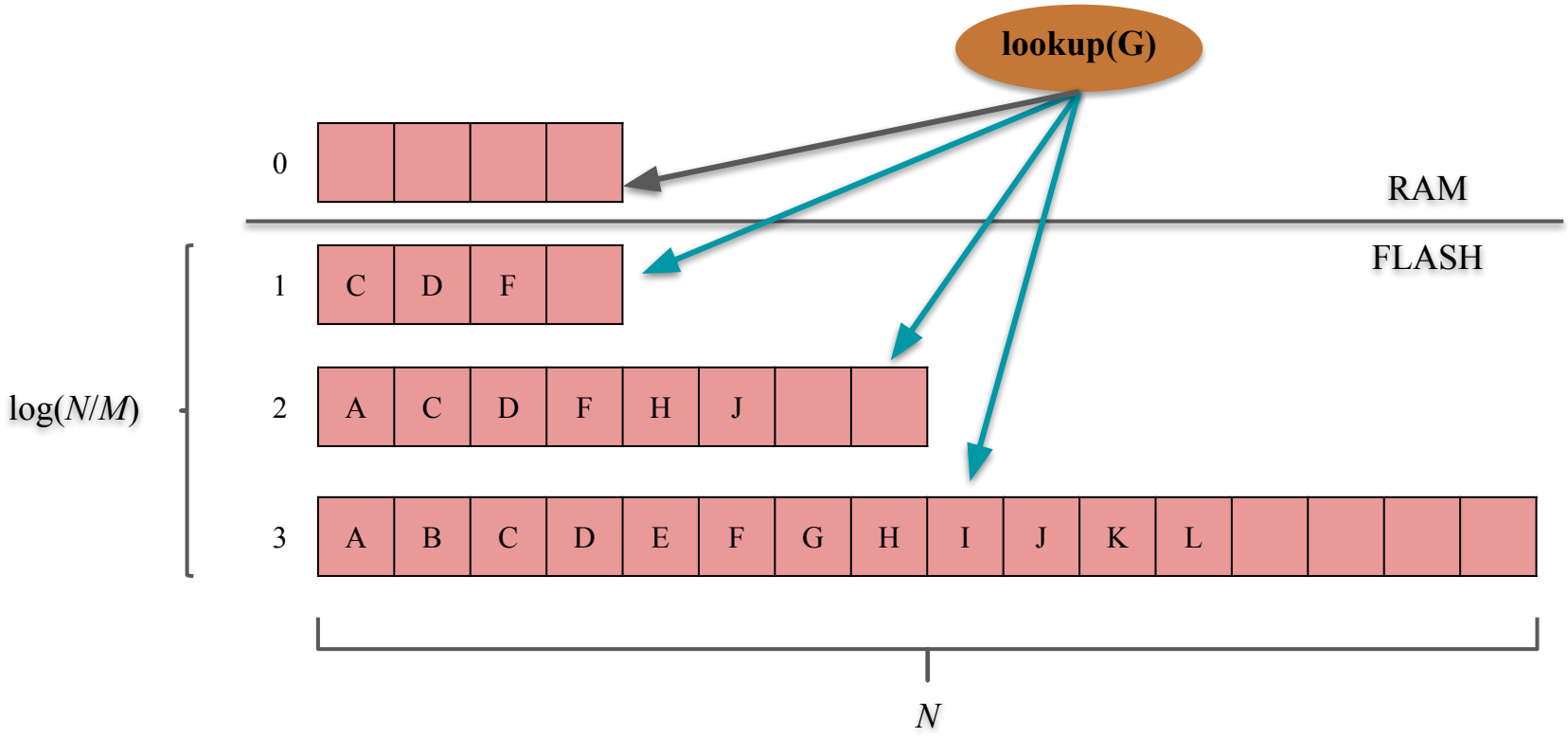


Cascade filter: insertions



Number of I/Os per item: $O(\log(\frac{N}{M})/B)$

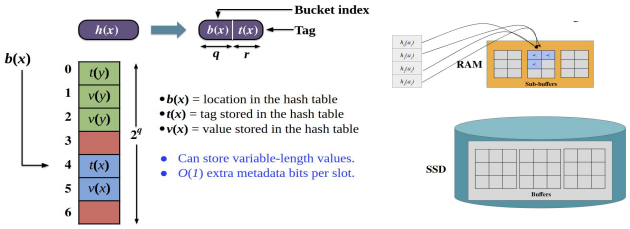
Cascade filter: lookups



Number of I/Os per item: $O(\log(\frac{N}{M}))$

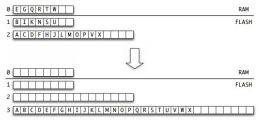
Thesis overview: streaming

Data structures

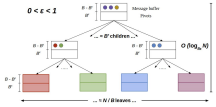


SIGMOD '17

ESA '18



Cascade filter



B^ε-tree

File systems

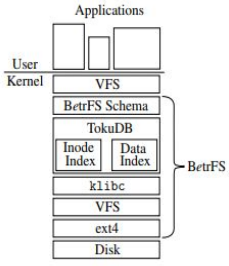
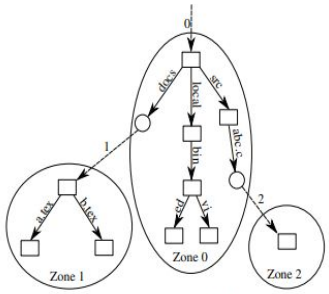


Figure 1: The BetrFS architecture.

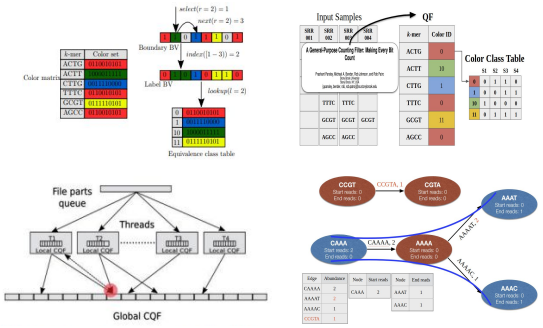


(a) An example zone tree in BetrFS 0.2.

FAST '15, TOS 15

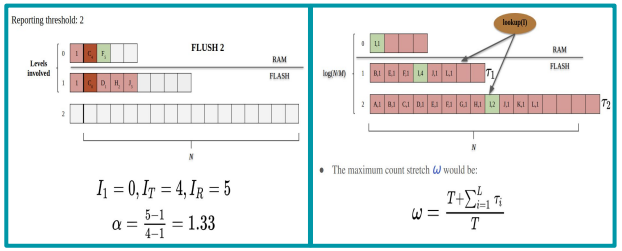
FAST '16, TOS 16

Computational biology



ISMB '17, BIOINFORMATICS '17, WABI '17, RECOMB '18, Cell Systems '18

Streaming



Open problem from Sandia National Lab

- A **high-speed stream** of key-value pairs arriving over time.
- **Too large** to fit in RAM. We can use SSD.
- **Goal**: report any key **as soon as** it appears 24 times **without any errors**.
- This is called the **online event-detection** problem in CS literature. (Will cover literature in the next couple of slides.)

Why Sandia cares about this problem

- Defense systems for cyber security[Berry et al. 09, Kezunovic 06, Litvinov 06], physical systems, such as water or power distribution[Sceller et al. 17, Raza et al. 13, Yan et al. 09].
- Automated systems take defensive action for every reported event.
- **No false positives**.

https://homeandkitchenaz.wordpress.com/2015/05/03/the-benefits-of-installing-a-power-monitoring-system/

The heavy hitters problem (HH(k))

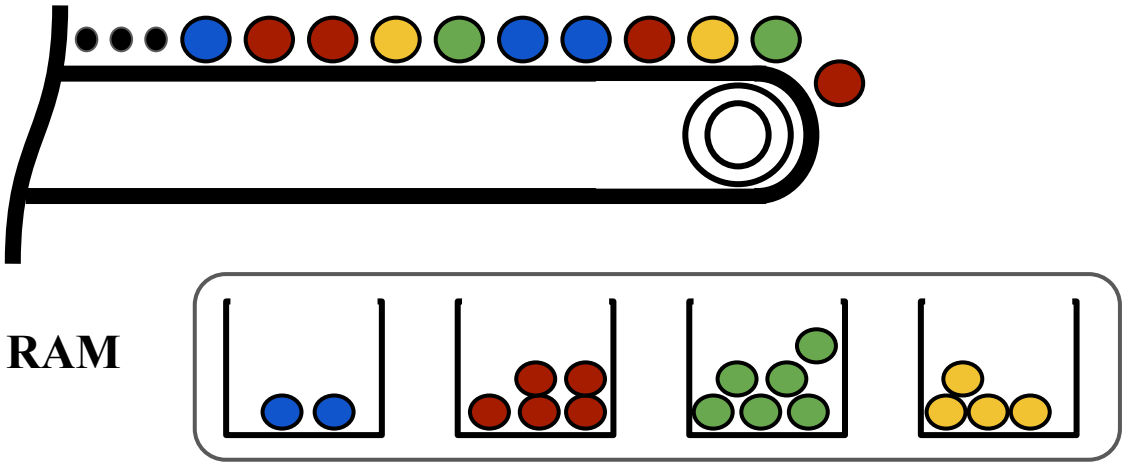
[Cormode et al. 05]

- Given stream of N items, report items whose frequency $\geq \varphi N$.
- General solution is "hard" in small space **(i.e., $\Omega(N)$ words)** and so people give approximation algorithms. **(But remember that Sandia doesn't want approximations.)**
- For Sandia application, φN is a constant (i.e., 24) and so φ is **very very small**.

The approximate heavy hitters problem (ε-HH(k))

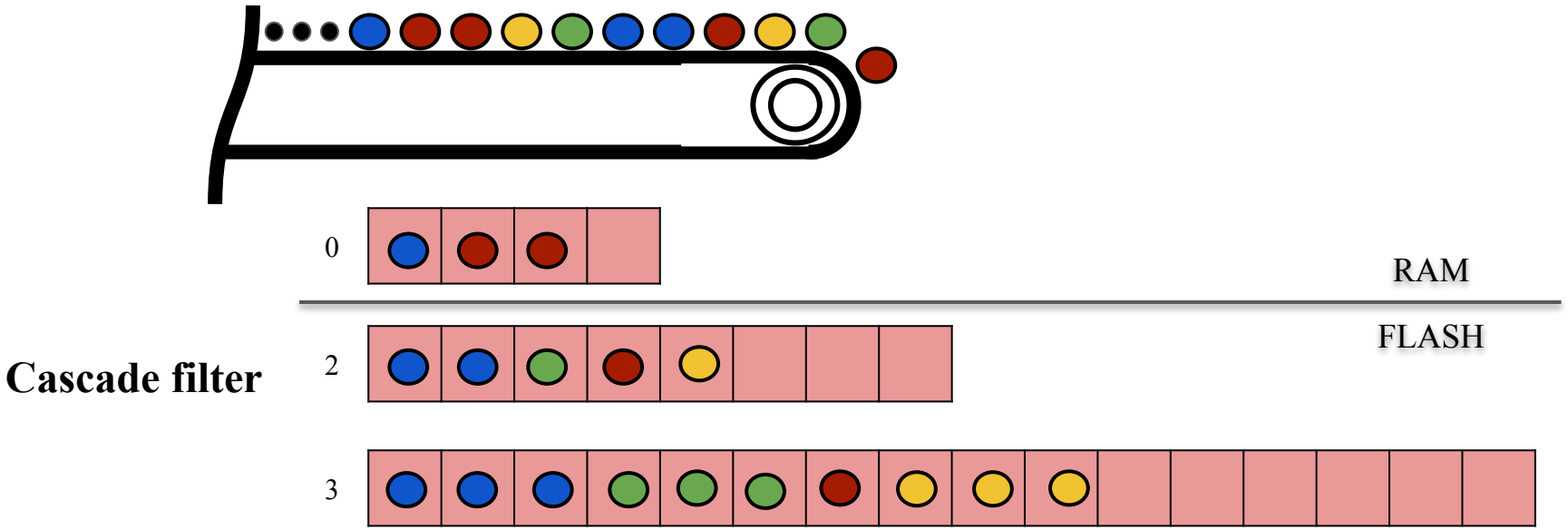
- Find all items with count $\geq \varphi N$, none with count $< (\varphi - \varepsilon)N$
- Related problem: **estimate each frequency with error $\pm \varepsilon N$**
- There is a rich literature that offers optimal solutions in RAM, i.e., for large ε. **(But remember that Sandia doesn't want approximations.)**

Alon et al. 96, Berinde et al. 10, Bhattacharyya et al. 16, Bose et al. 03, Braverman et al. 16, Charikar et al. 02, Cormode et al. 05, Demaine et al. 02, Dimitropoulos et al. 08, Larsen et al. 16, Manku et al. 02.



Can we use the cascade filter?

- Cascade filter supports fast insertions (**< 1 I/O per item**).
- But they do not help with queries (**log(*N*/*M*) I/Os per item**).
- I will tell you in the next few slides how we modify the cascade filter to solve the online event-detection problem.



A solution for online event-detection must have:

	Existing	Cascade filter	We want
Fast insertions	✓	✓	✓
Scalability	✗	✓	✓
Be exact	✗	✓	✓
Reporting timely	✓		✓

- A solution to the online event-detection problem must support all of the above features.

Online event-detection problem

[Bender, Berry, Farach-Colton, Johnson, Kroeger, Pandey, Phillips, Singh, '18]

- **Time-stretch filter**: online event-detection with a **bounded delay**.

$$O\left(\frac{\alpha+1}{\alpha}\log\left(\frac{N}{M}\right)/B\right) \text{ I/Os per item}$$

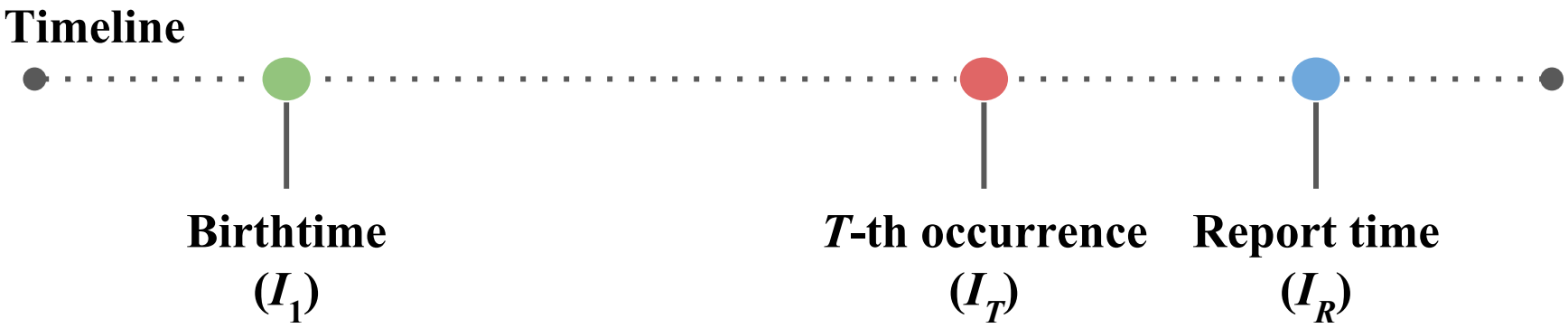
- **Popcorn filter**: online event-detection **immediate reporting**.

$$O\left(\left(\frac{1}{B}+\frac{1}{(\phi N-\gamma)^{\theta-1}}\right)\log\left(\frac{N}{M}\right)\right) \text{ I/Os per item}$$

We are much better than 1 I/O per item. Thus, effectively the same cost as the cascade filter, but queries are faster.

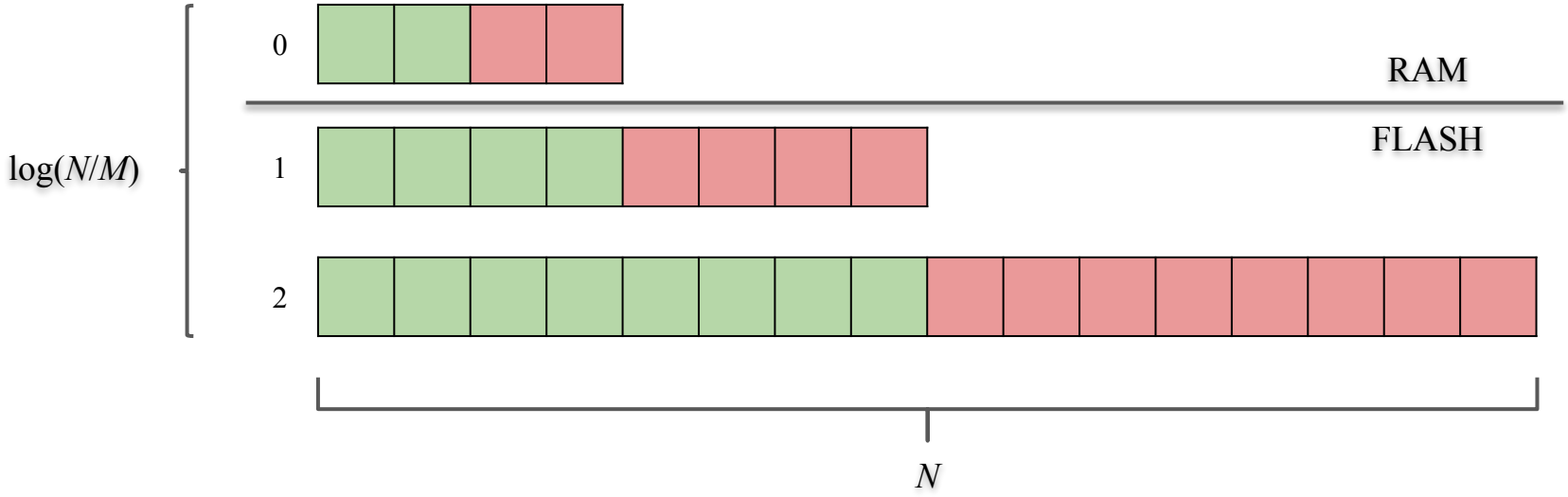
Time-stretch filter

A time-stretch of α, we must report an element a no later than time $I_1 + (1 + \alpha)F_T$, where I_1 is the time of the first occurrence of a, and F_T is the flow time of a.



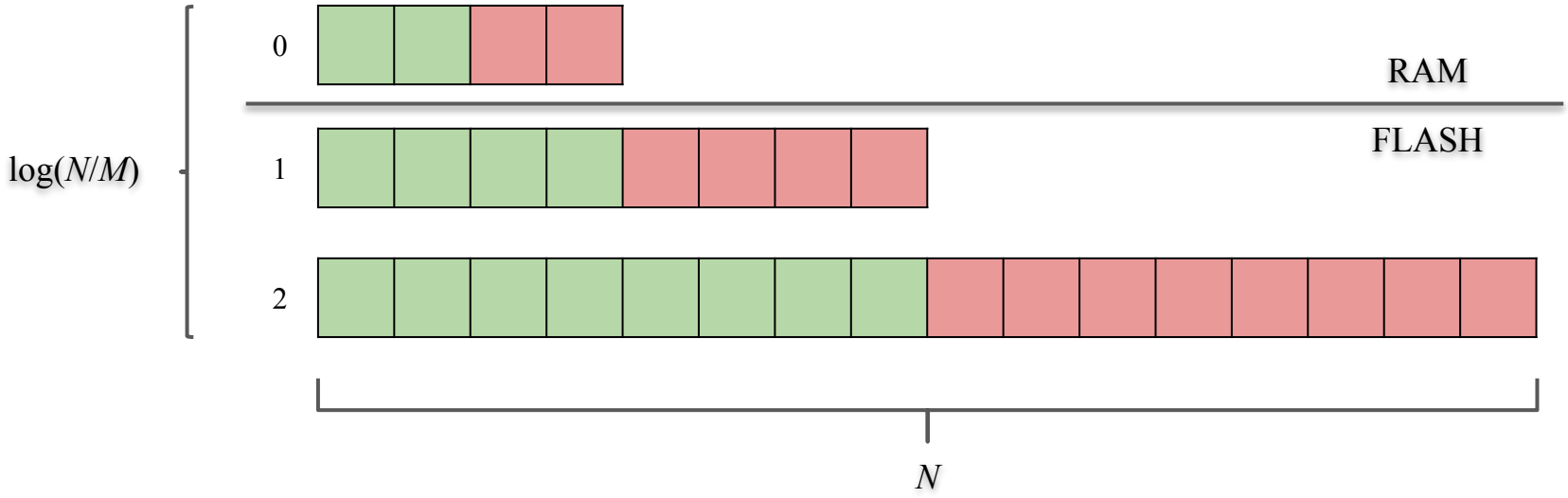
$$\alpha = \frac{I_R - I_1}{I_T - I_1}$$

Time-stretch filter



- Item can not go to level i until it is at least $r^i M$ time steps old.

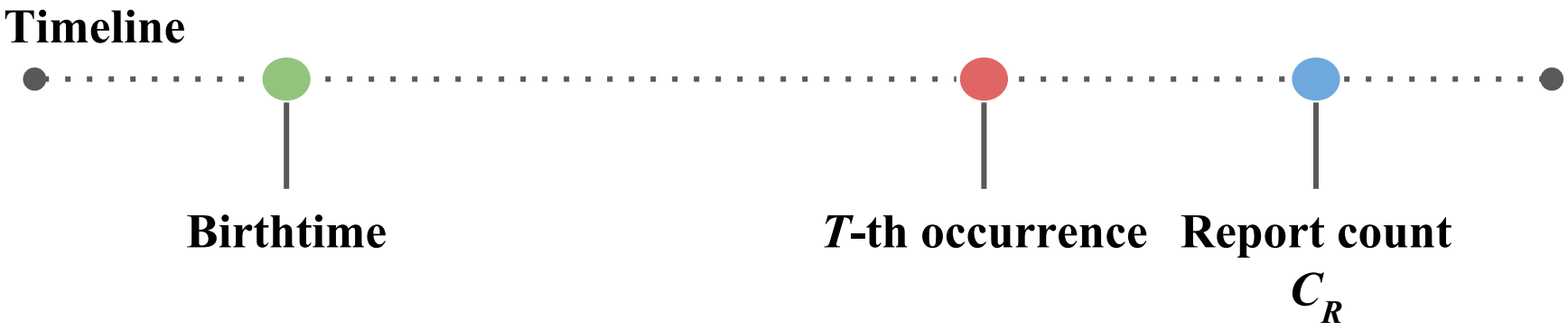
Time-stretch filter



- The QF at each level is split into $l = (\alpha+1)/\alpha$ equal sized bins, here $l = 2$.
- Flushes are performed at the granularity of bins and follow a fixed round-robin schedule for flushing bins.
- Each item inserted at level i spends at least $r^i M/\alpha$ time steps.

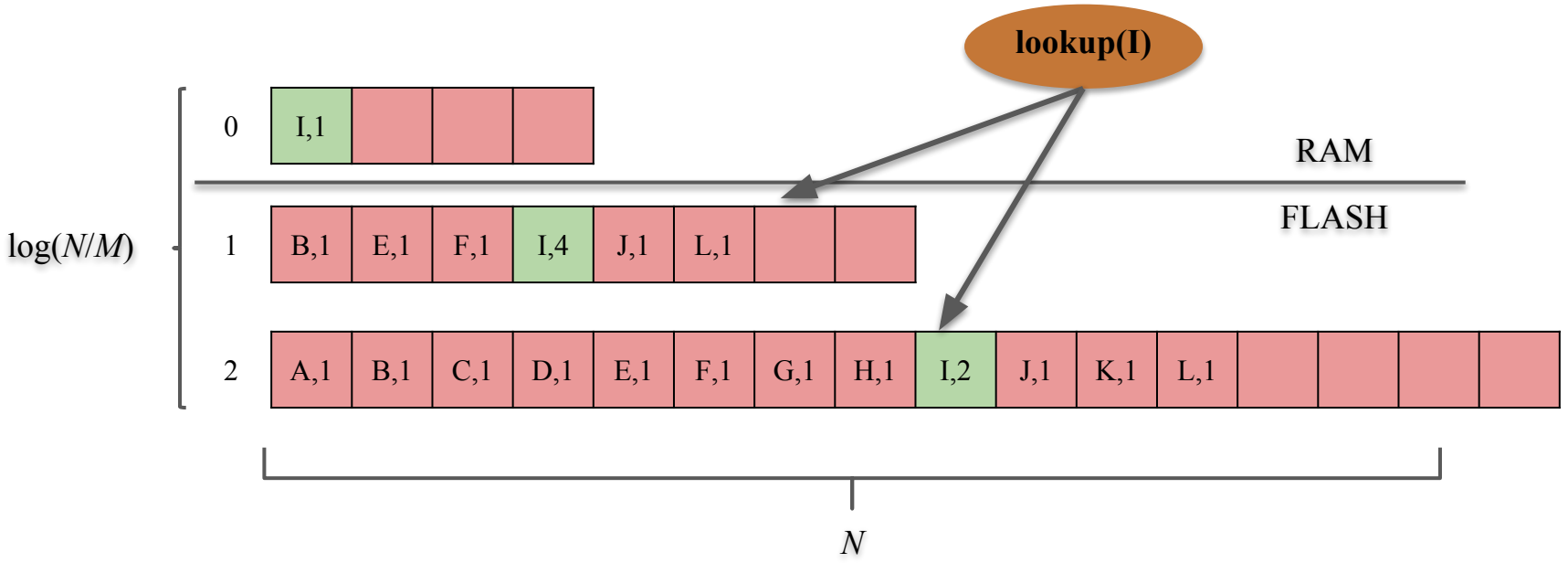
Popcorn filter

A count-stretch of ω, we must report an element a no later than when the count of a is ωT. In **immediate reporting** $\omega = 1$ & $\alpha = 1$.



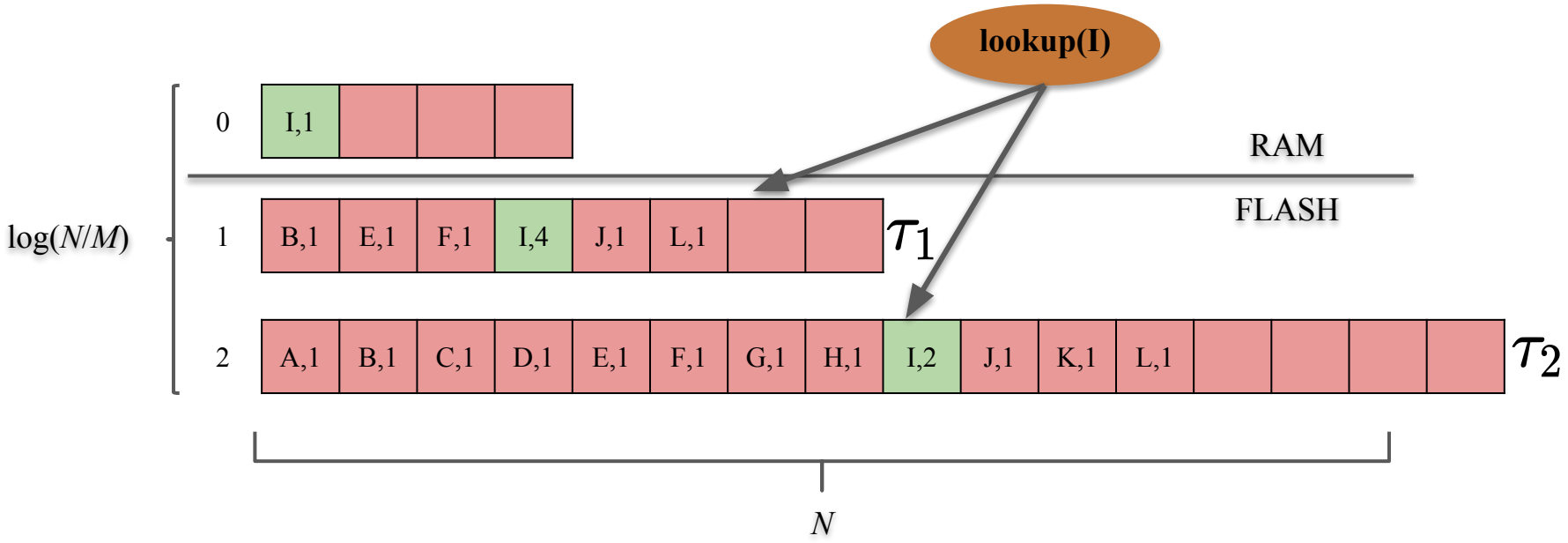
$$\omega = \frac{C_R}{T}$$

Popcorn filter



- To achieve immediate reporting, we need to perform multiple I/Os for every new item arriving in the RAM QF.

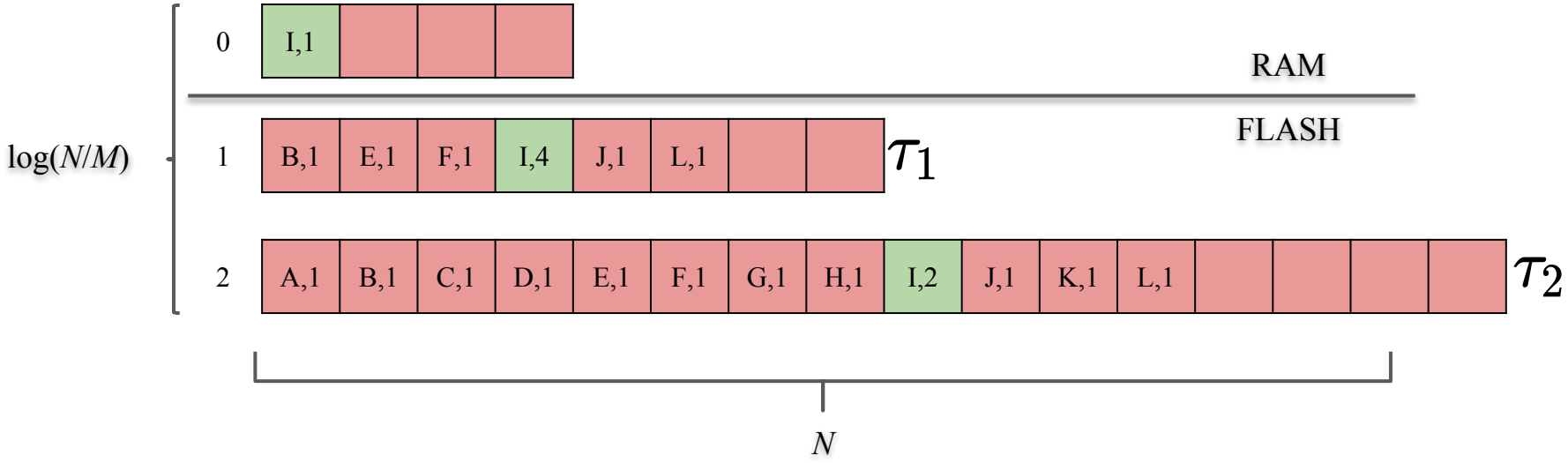
Popcorn filter: immediate reporting



- We can avoid a bunch of unnecessary I/Os if we can **upper bound the total instances on disk** of an item.

$$\text{Lookup } if: Ram_{count} = T - \left(\sum_{i=1}^{L} \tau_i\right)$$

Popcorn filter: count stretch



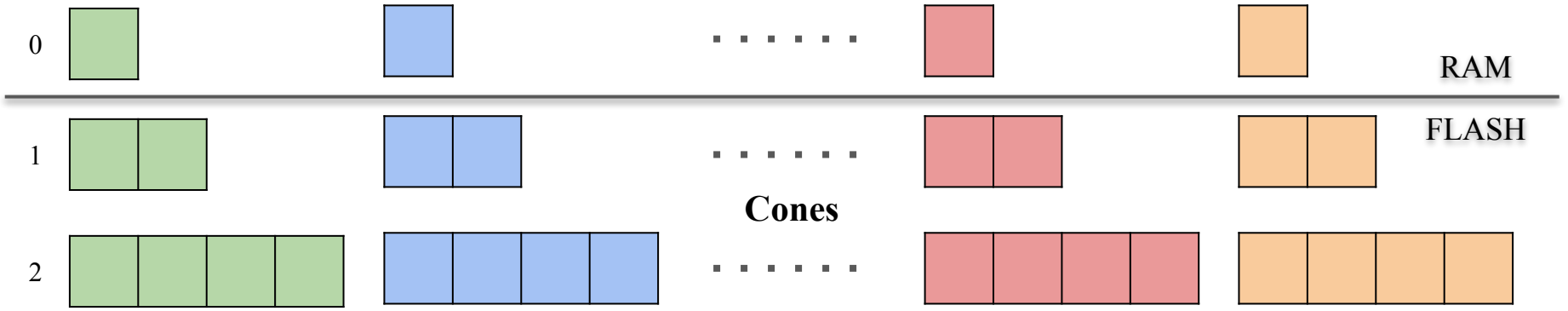
- The maximum count stretch ω would be:

$$\omega = \frac{T + \sum_{i=1}^{L} \tau_i}{T}$$

Results from this part

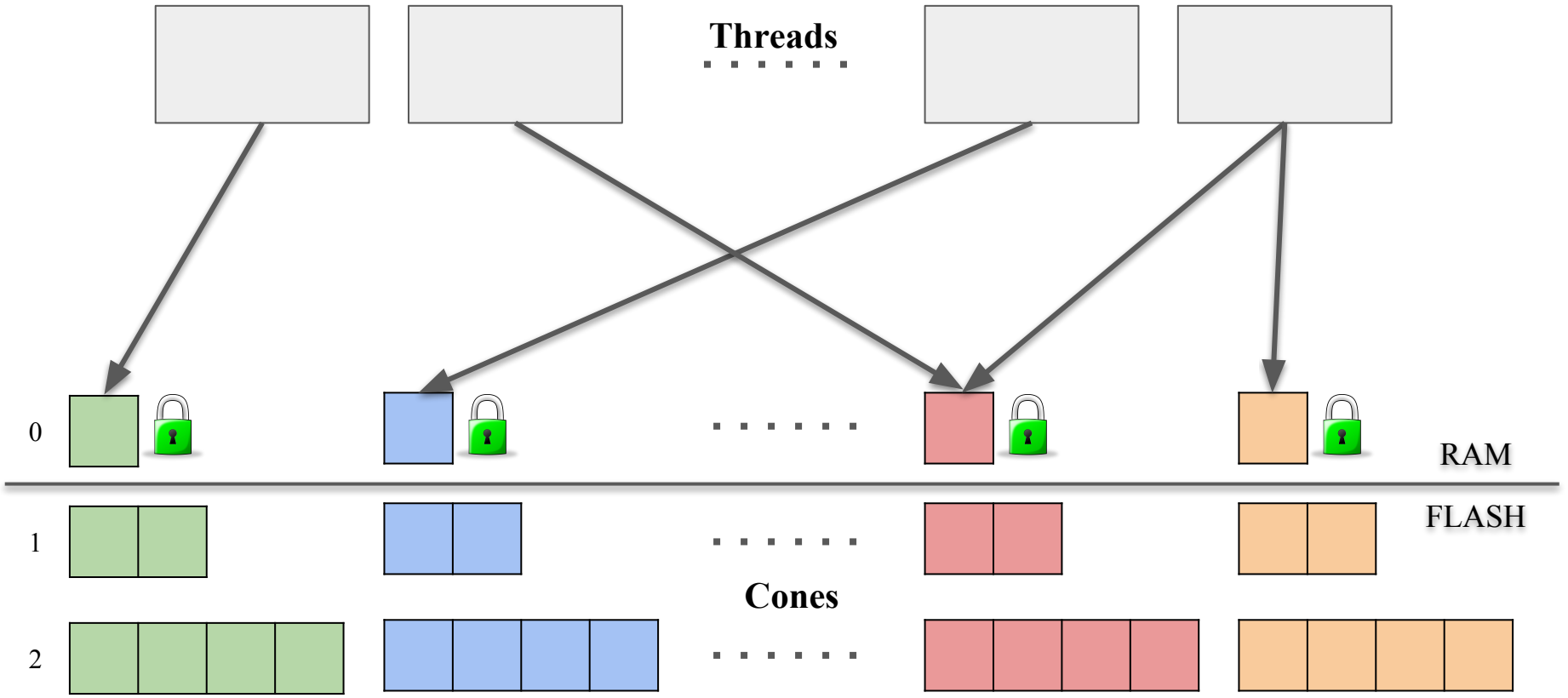
- **Implement** the time-stretch filter and popcorn filter.
- **Multi-threaded/deamortized version** of the time-stretch and popcorn filter.
- Evaluate:
  - Timeliness guarantees
  - I/O performance
  - Insertion throughput
  - Scalability with multiple threads

Multithreading/Deamortization



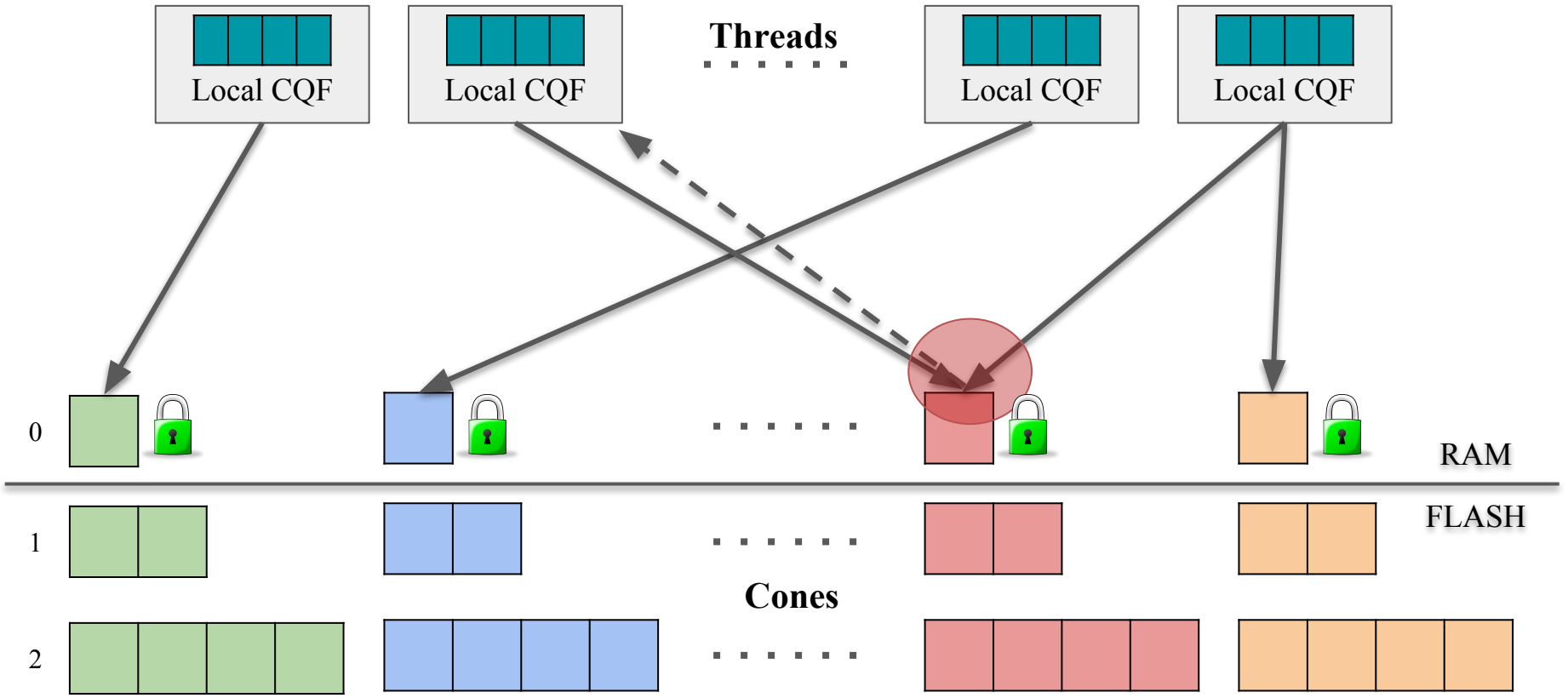
We divide the filter into multiple smaller filters called ***cones***, where each cone consists of same number of levels and growing exponentially.

Multithreading/Deamortization



Each thread operates by first taking a lock at the cone and then performing the insert operation.

Multithreading/Deamortization



If there is contention, the thread then inserts the item in its local buffer and continues.

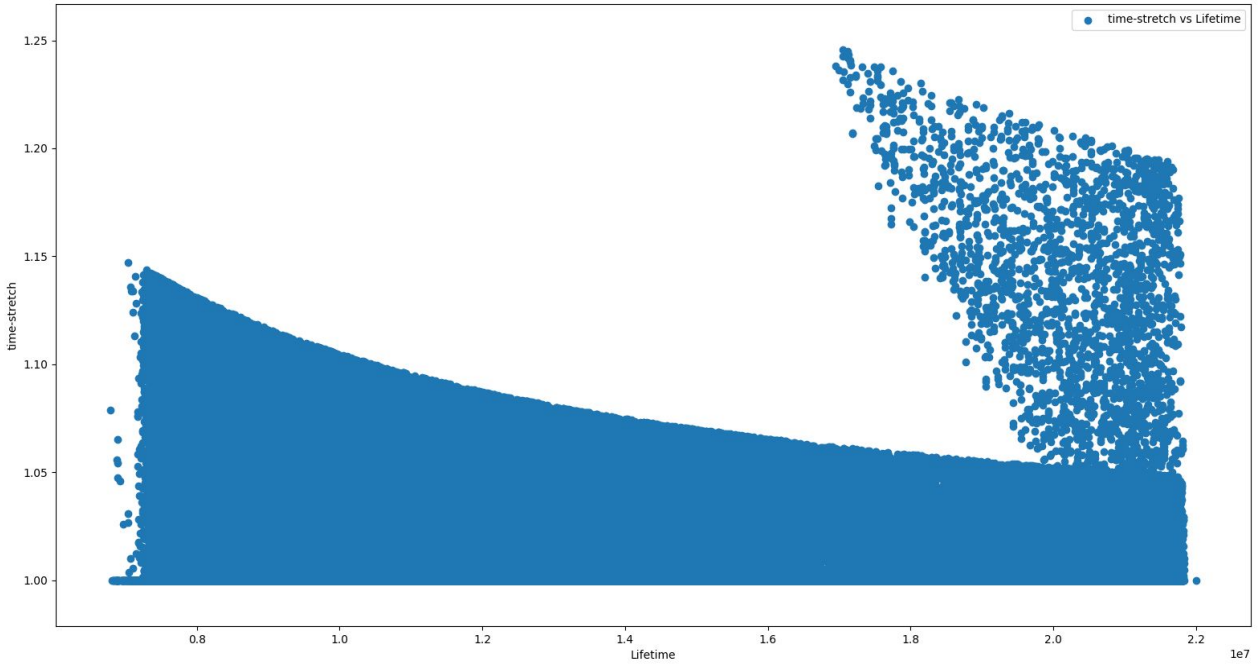
Timely reporting validation

Num age bits	Num bins	Alpha value
1	2	1
2	4	1.33
3	8	1.14
4	16	1.06

$$l = \frac{\alpha+1}{\alpha}$$

- This is the relationship between the number of bins and the **α** value in the time-stretch filter.

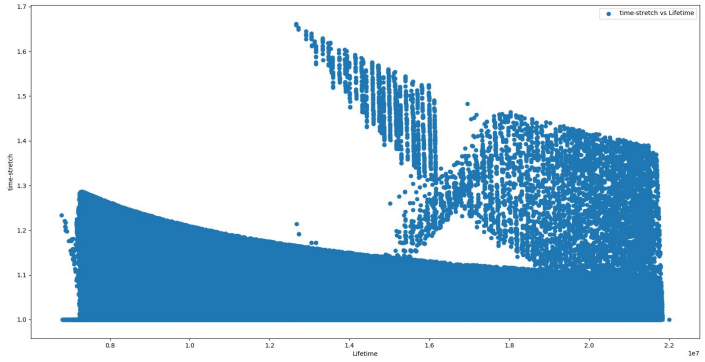
Time stretch: timely reporting validation



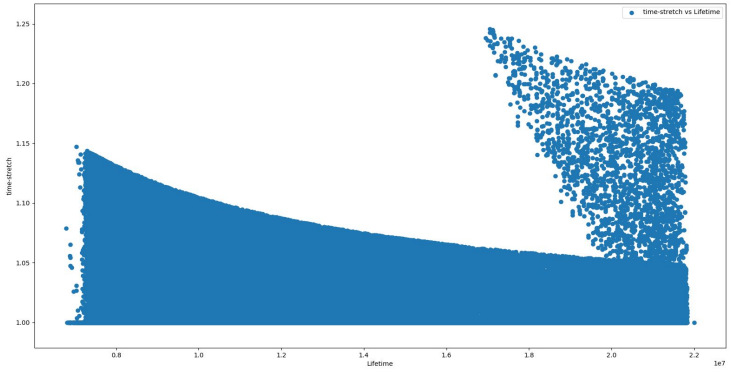
#bins: 4

- The time-stretch filter reports all items within the maximum allowed time stretch for time stretch 1.33.

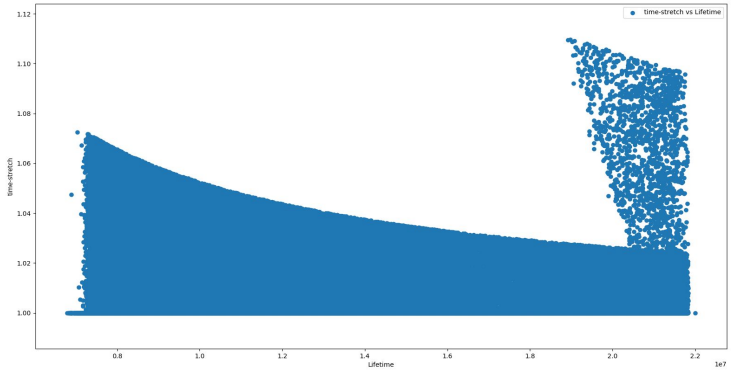
Time stretch: timely reporting validation



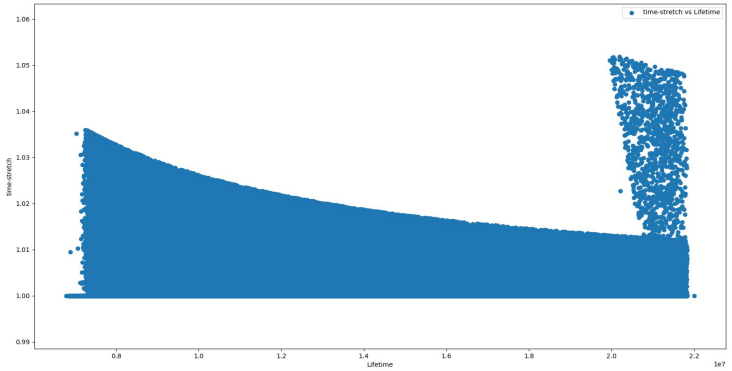
#bins: 2



#bins: 4



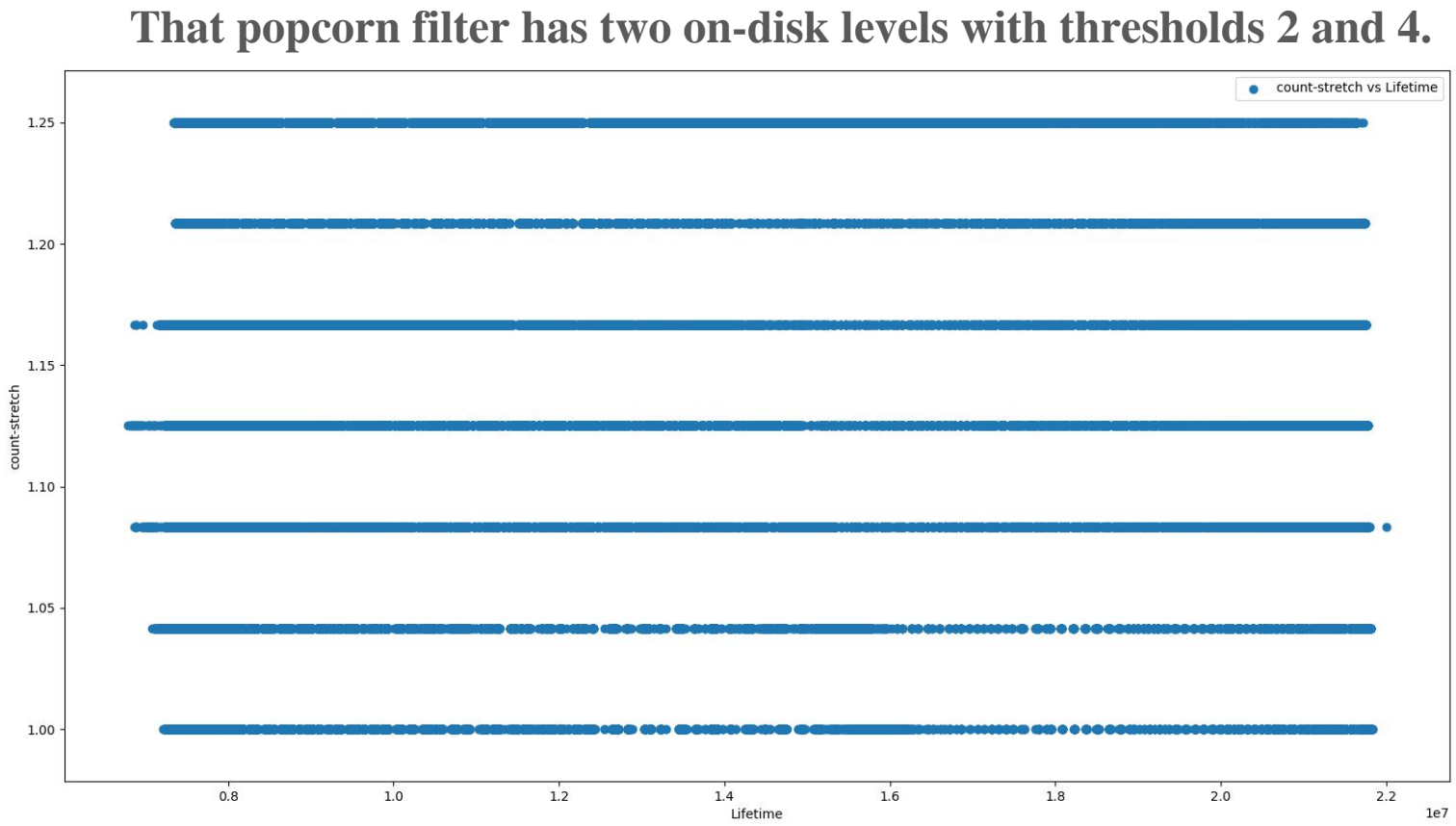
#bins: 8



#bins: 16

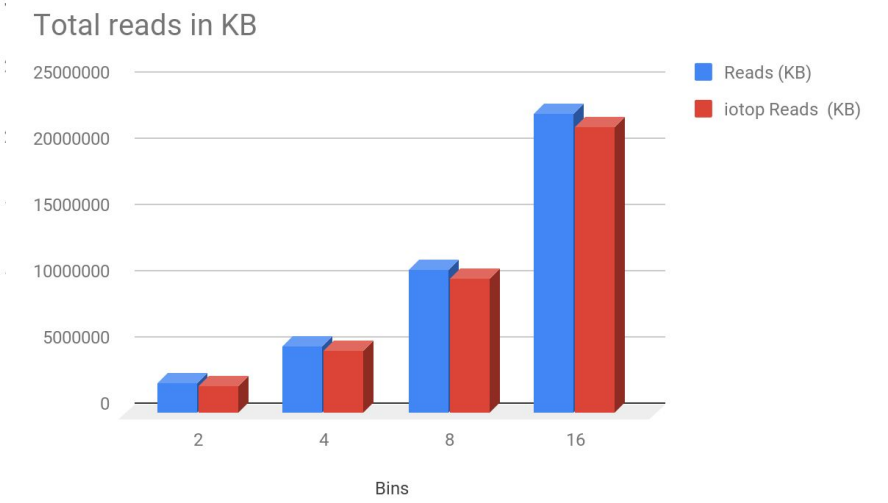
- The time-stretch filter reports all items within the maximum allowed time stretch for all the four time stretch values.

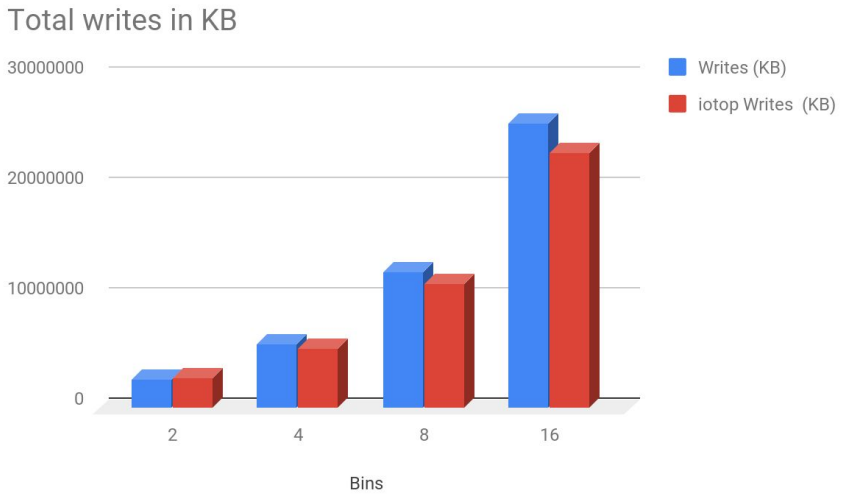
Count stretch: timely reporting validation



- The popcorn filter reports all items within the maximum allowed count stretch.

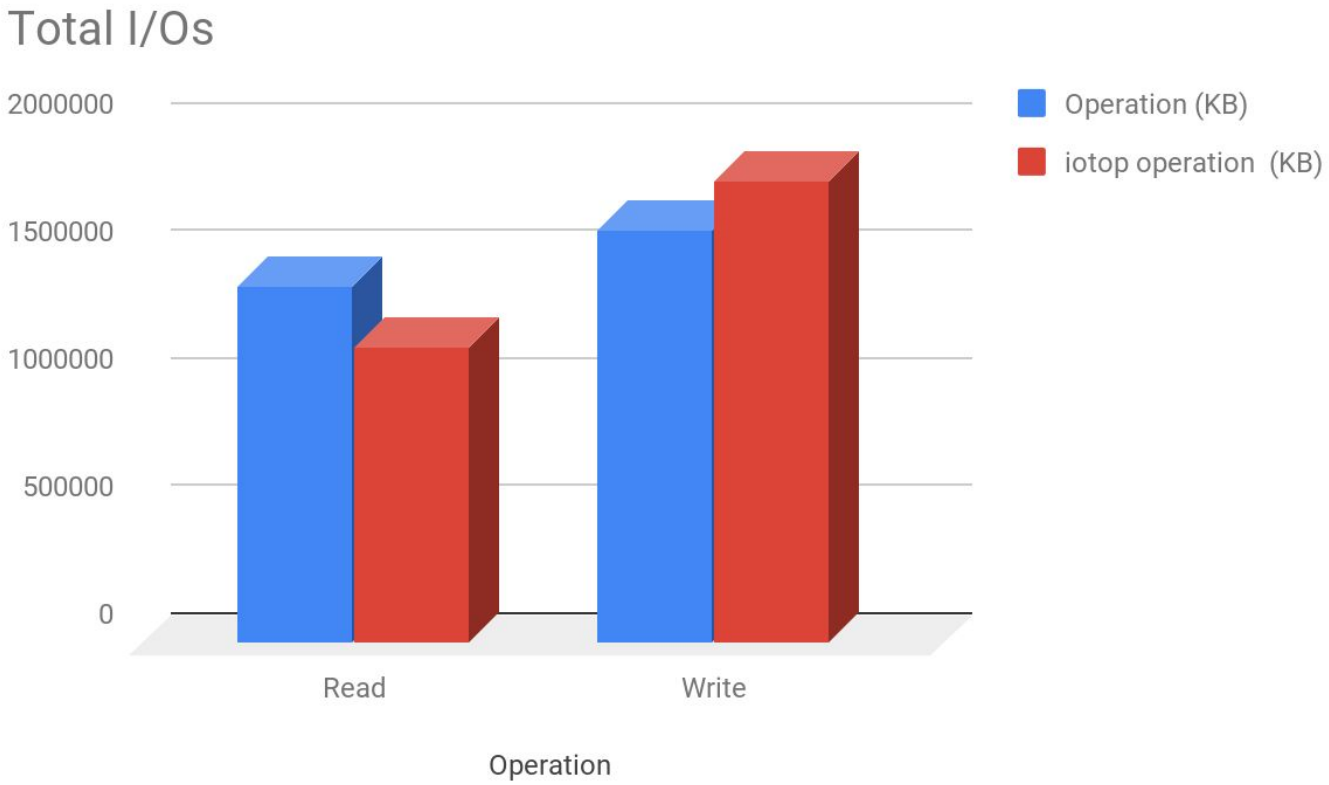
Time stretch: I/O cost





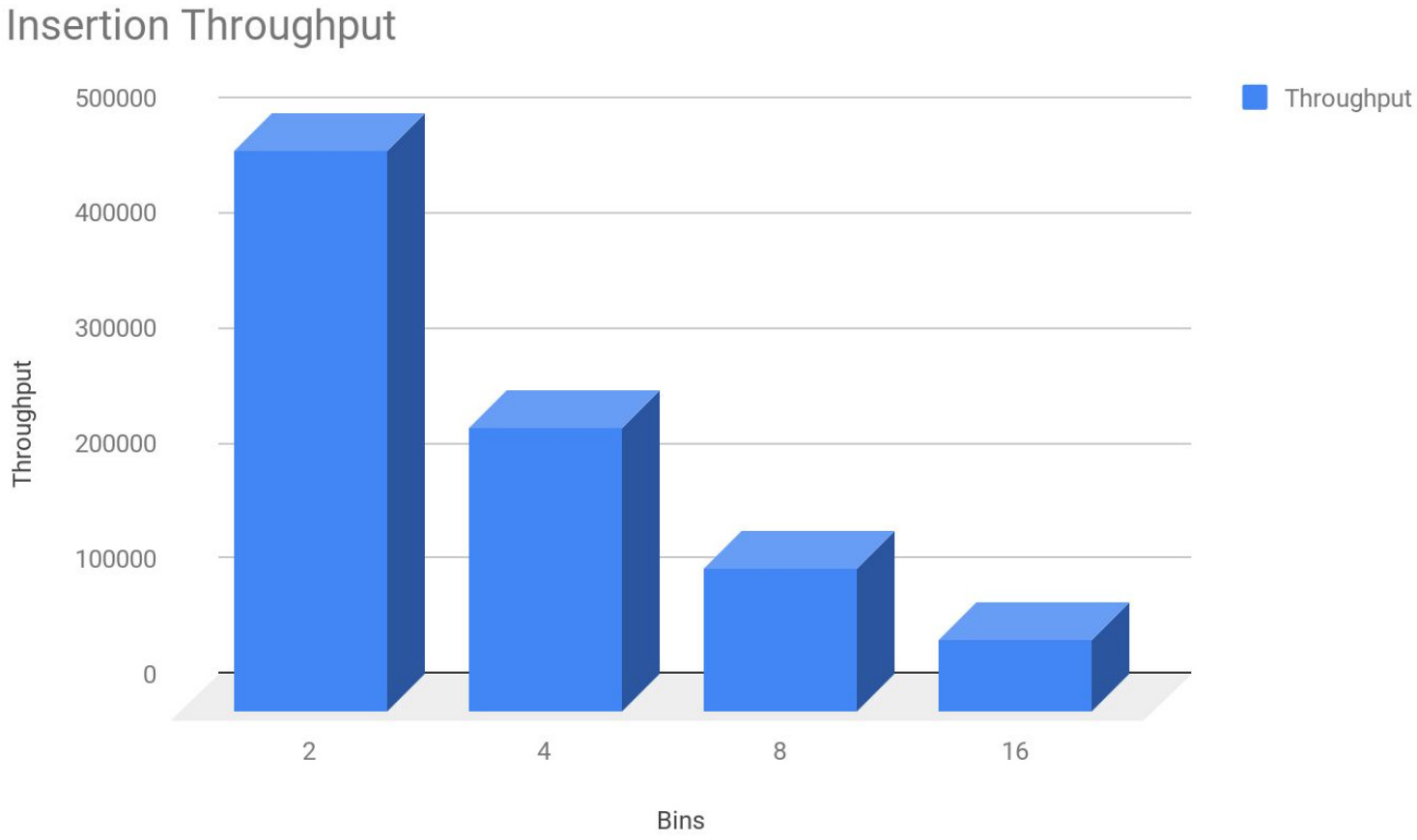
- The time-stretch filter empirically performs similar amount of I/Os as given by the theoretical bounds.

Count stretch: I/O cost



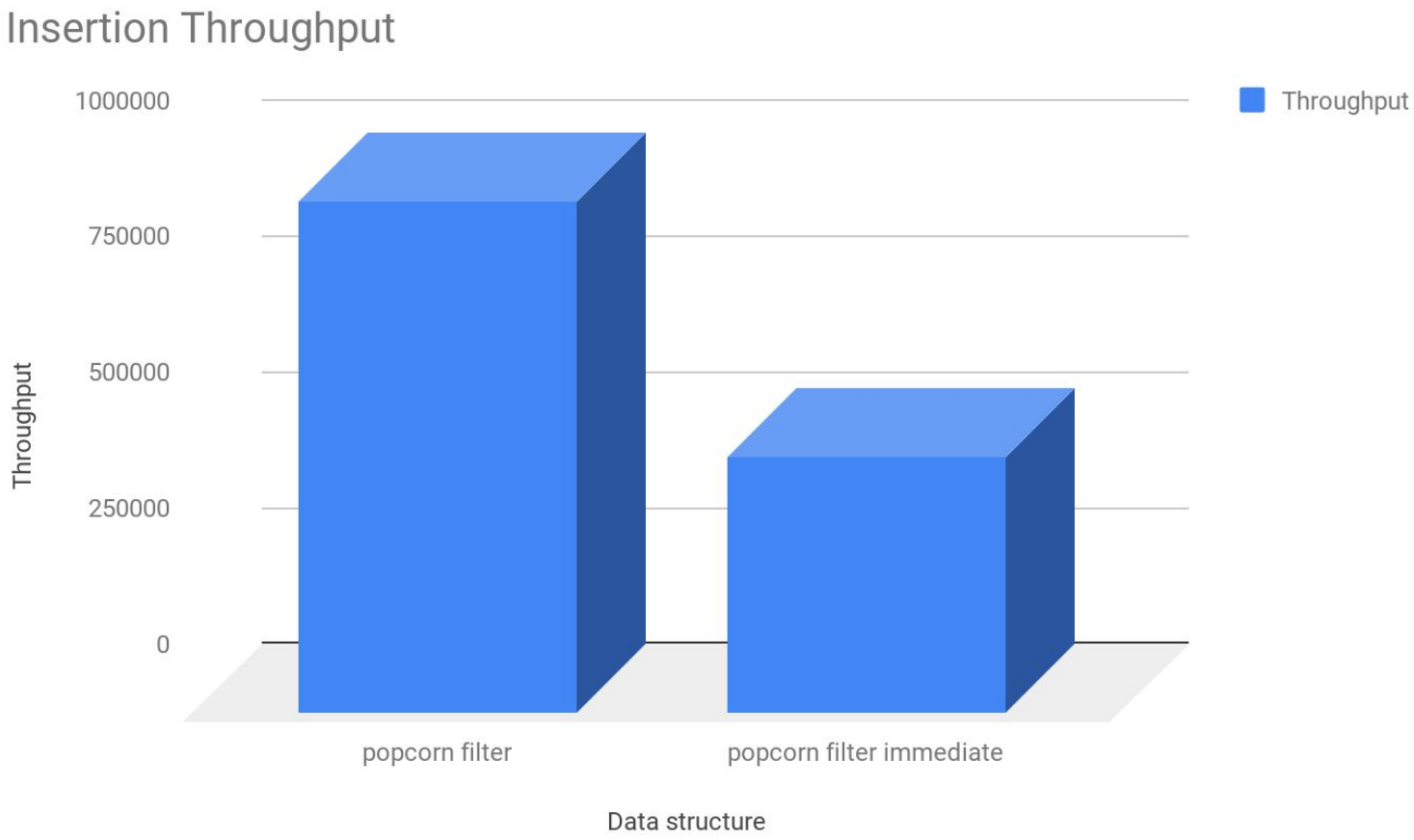
- The popcorn filter empirically performs similar amount of I/Os as given by the theoretical bounds.

Time stretch: Insertion throughput



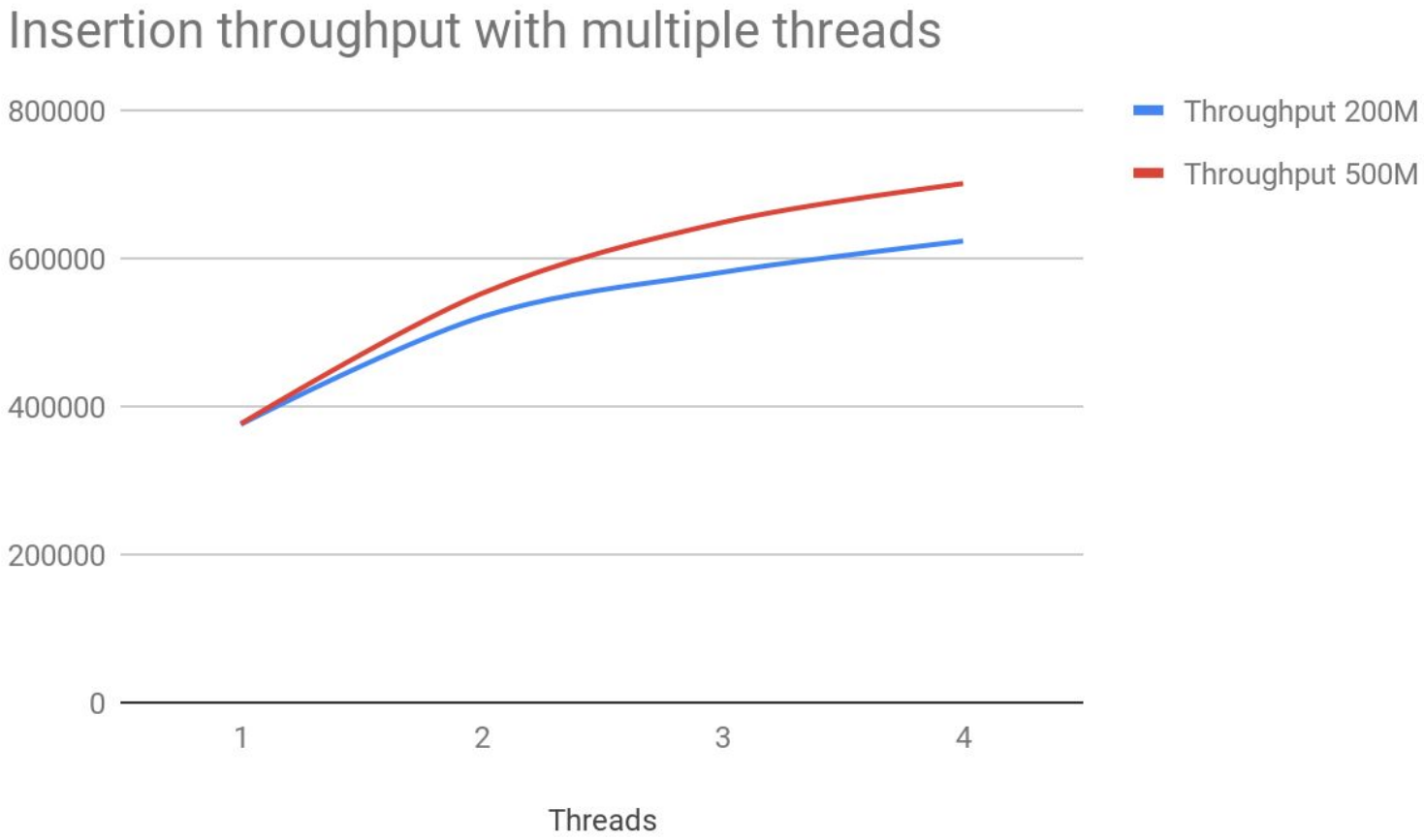
- The insertion throughput of the time-stretch filter is going down as we decrease the time-stretch.

Count stretch: Insertion throughput



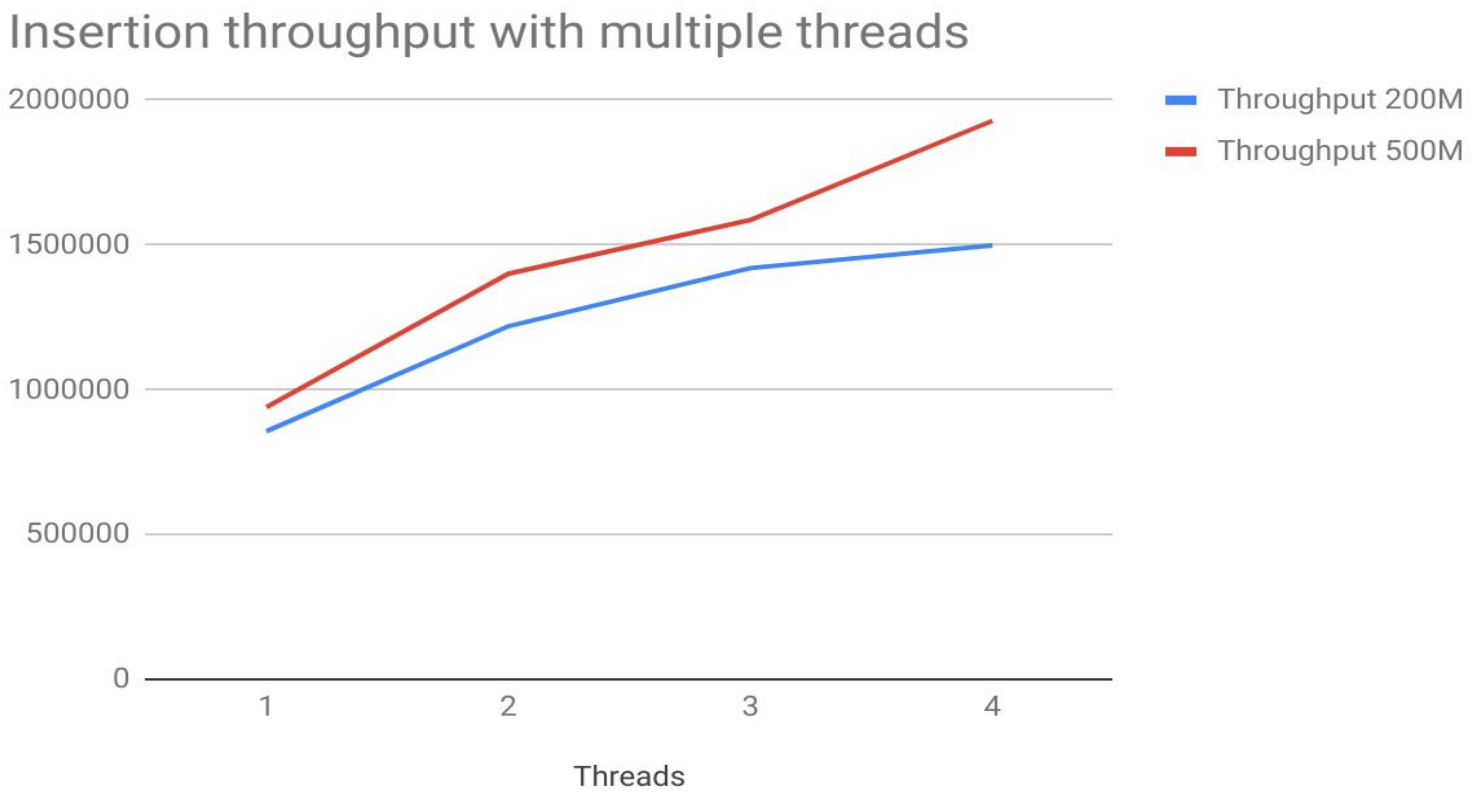
- The insertion throughput of the popcorn filter with immediate reporting is lower because of random I/Os for lookups.

Time stretch: scalability with threads



- The insertion throughput is going up with increasing number of threads.

Count stretch: scalability with threads

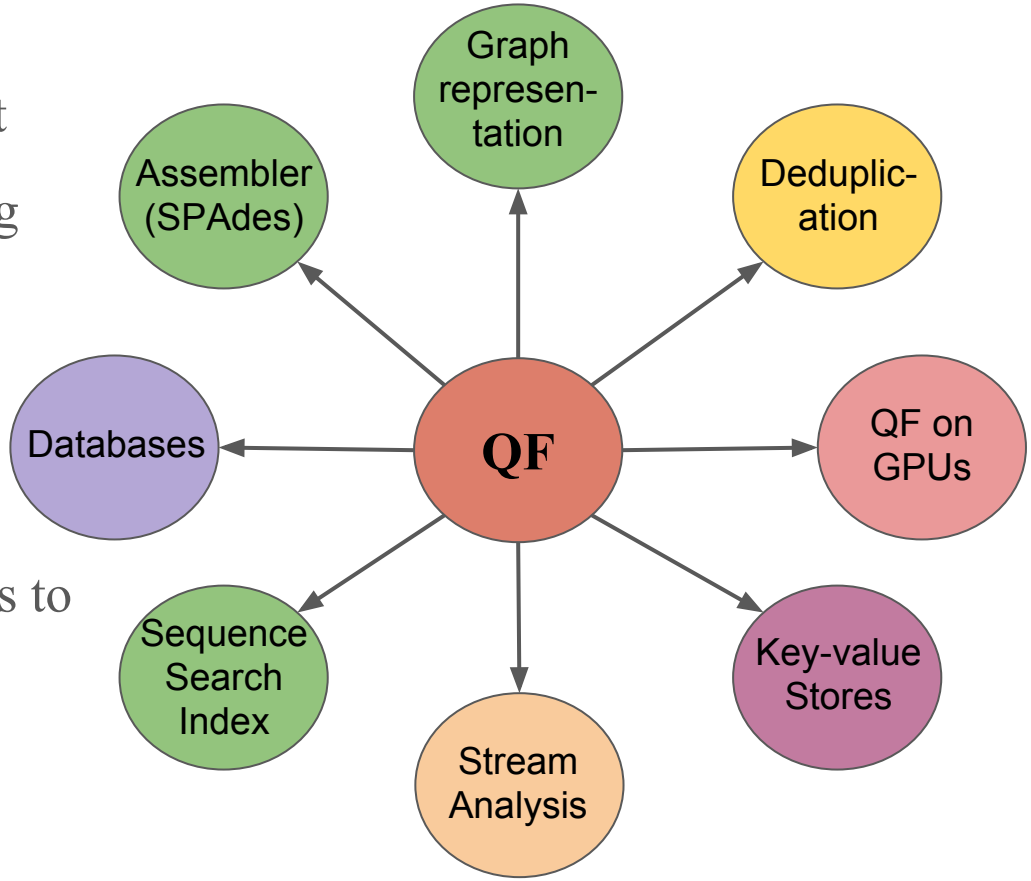


- The insertion throughput is going up with increasing number of threads.

Conclusion

- Space-efficient and fast compact data structures can help solve big data problems across subfields.

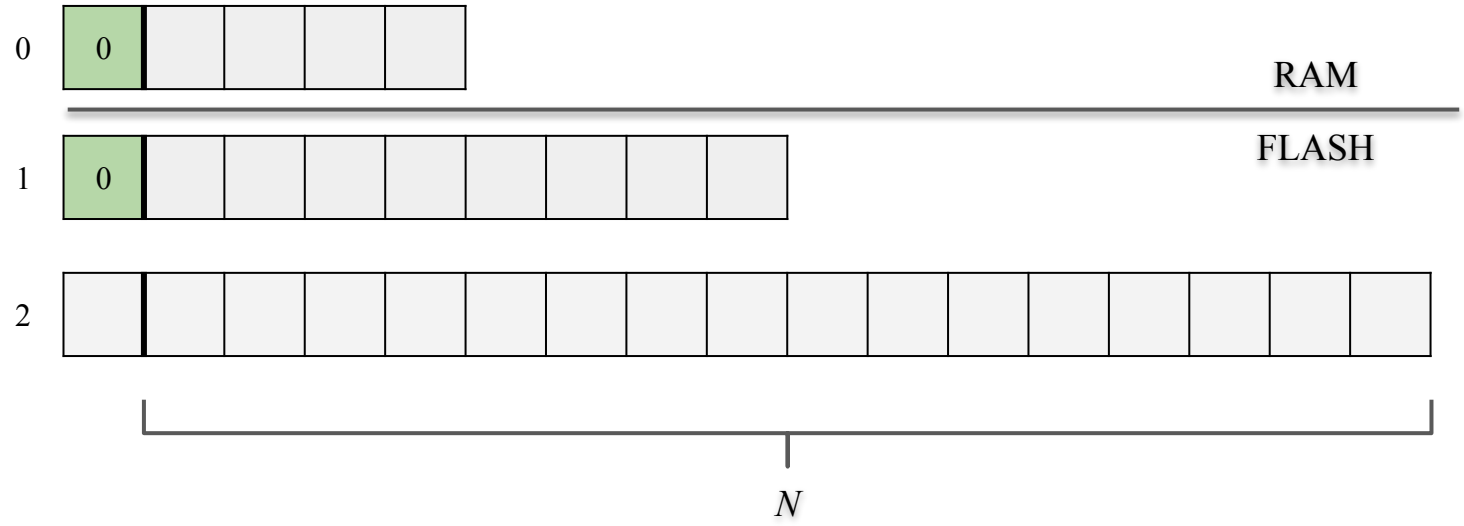
- We need to redesign applications to get complete benefits of today's feature-rich AMQs.



Source code: https://github.com/splatlab/


2018
RECOMB
Paris, France

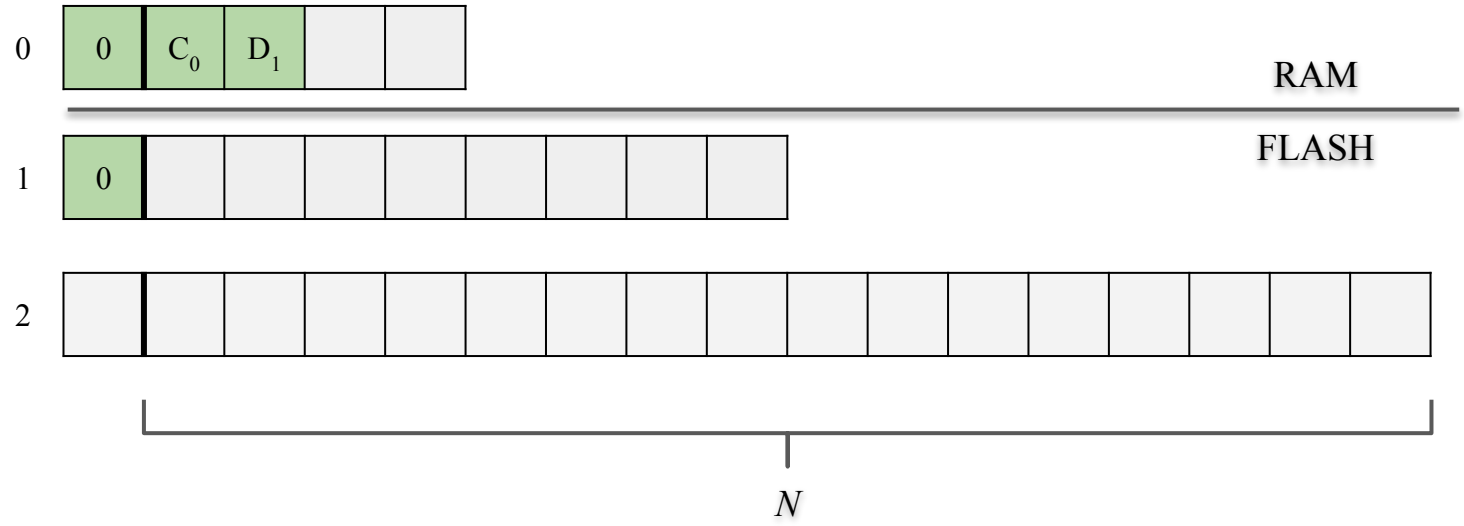
Time-stretch filter



- We store an age value of log(l) bits with each level and item
- At initialization time, every level gets an age 0.
- Here, we have 1-bit age, i.e., 0, 1.

Time-stretch filter

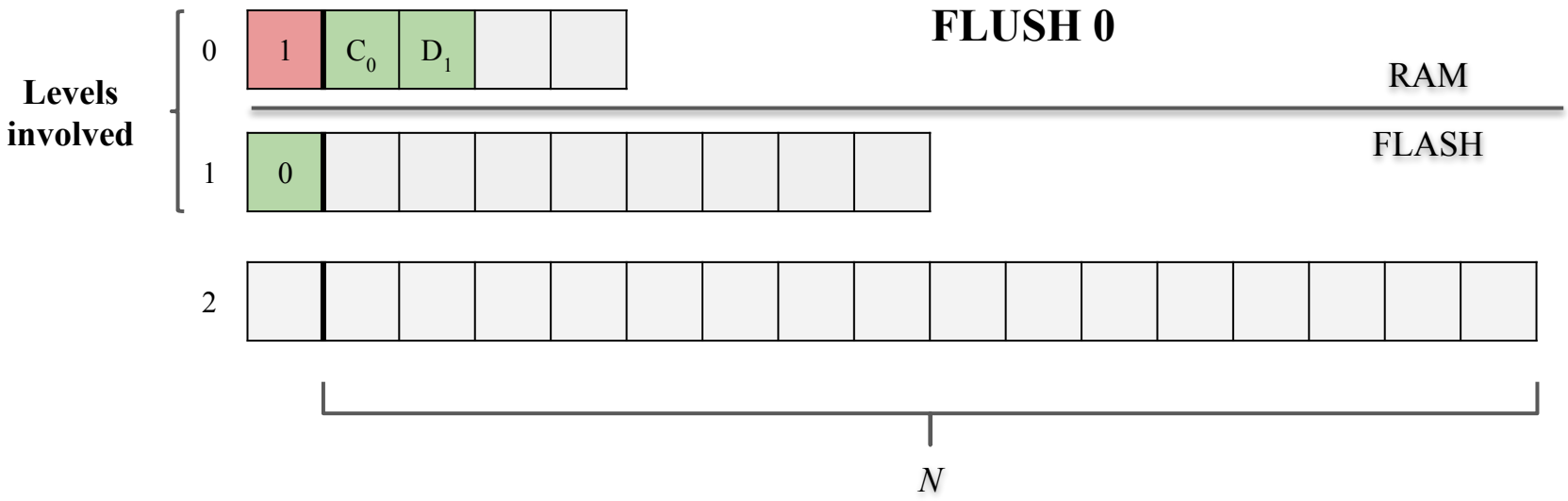
Reporting threshold: 2



- When an item arrives at a level it gets the current age of the level.
- We trigger a flush when a bin is full.

Time-stretch filter

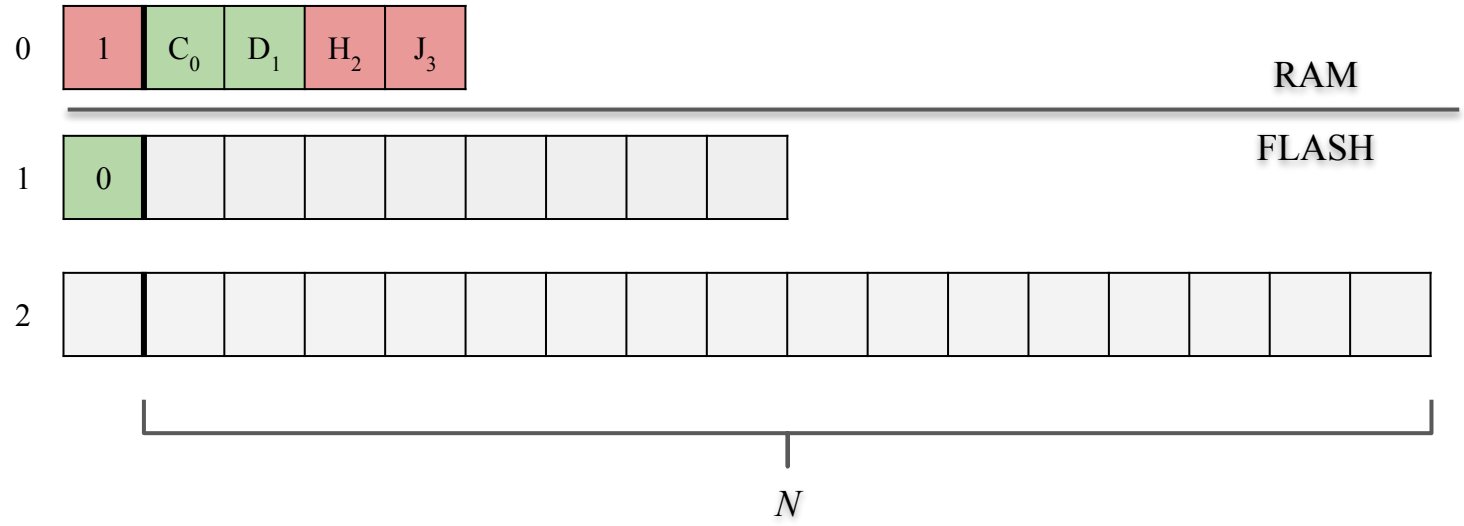
Reporting threshold: 2



- We increment the age of all levels that are flushing before the flush.
- Every item with the same age as the current age of level is flushable.
- Schedule: every *r*-th flush goes to the *r*-th level.

Time-stretch filter

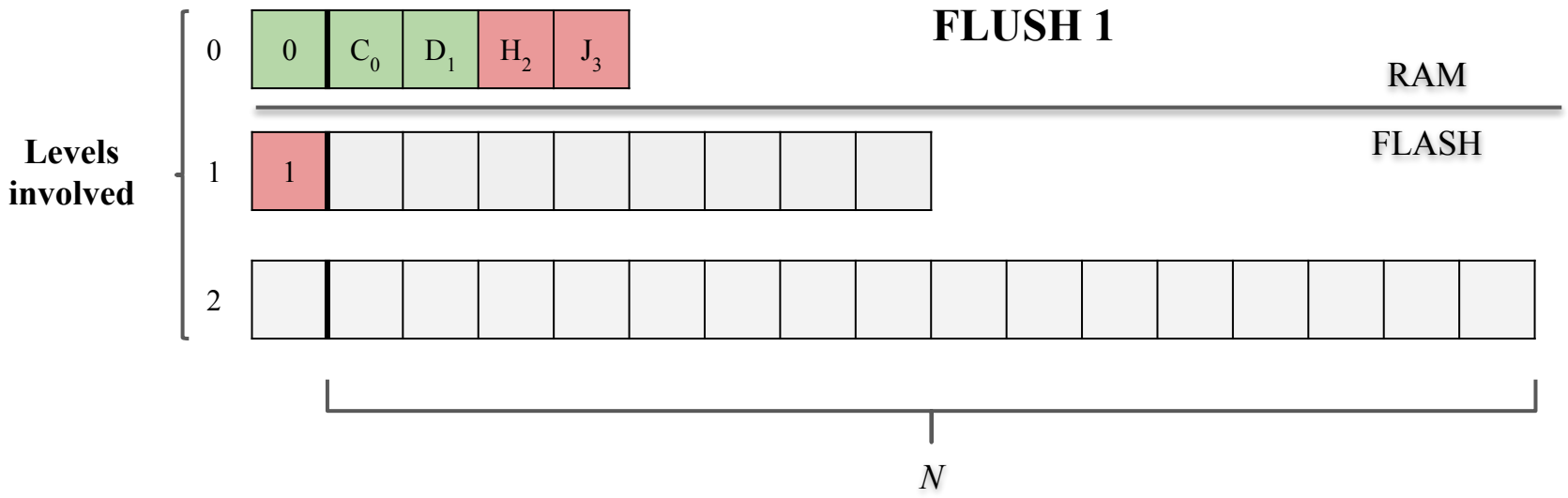
Reporting threshold: 2



- We increment the age of the level before a flush.
- Every item with the same age as the current age of level is flushable.

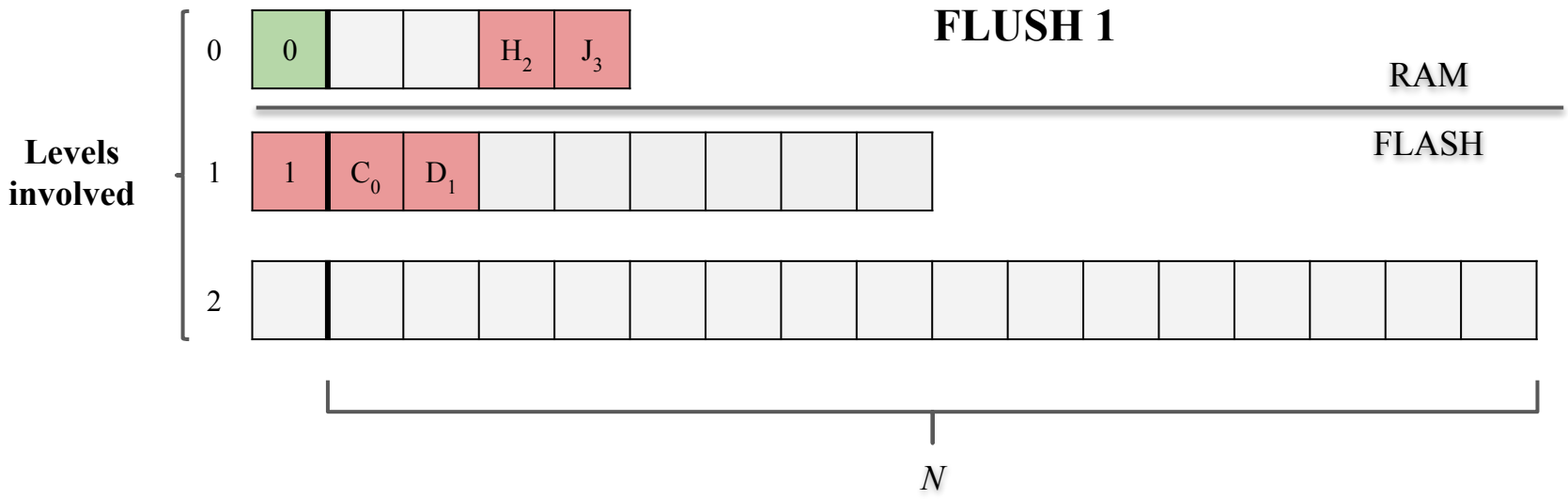
Time-stretch filter

Reporting threshold: 2



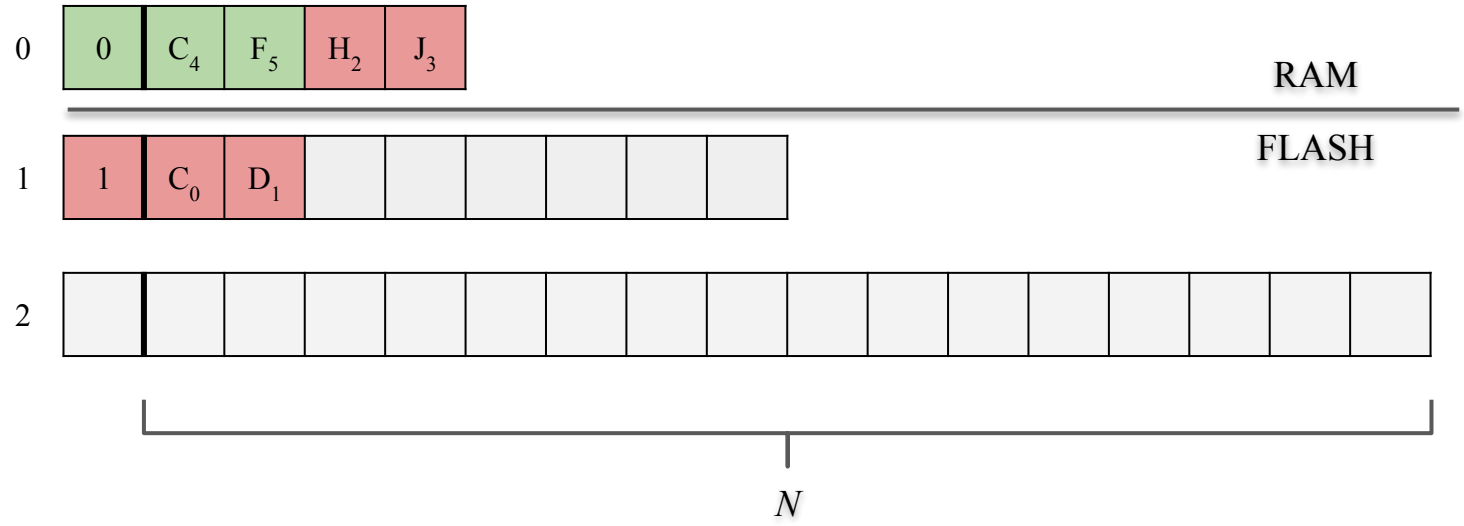
Time-stretch filter

Reporting threshold: 2



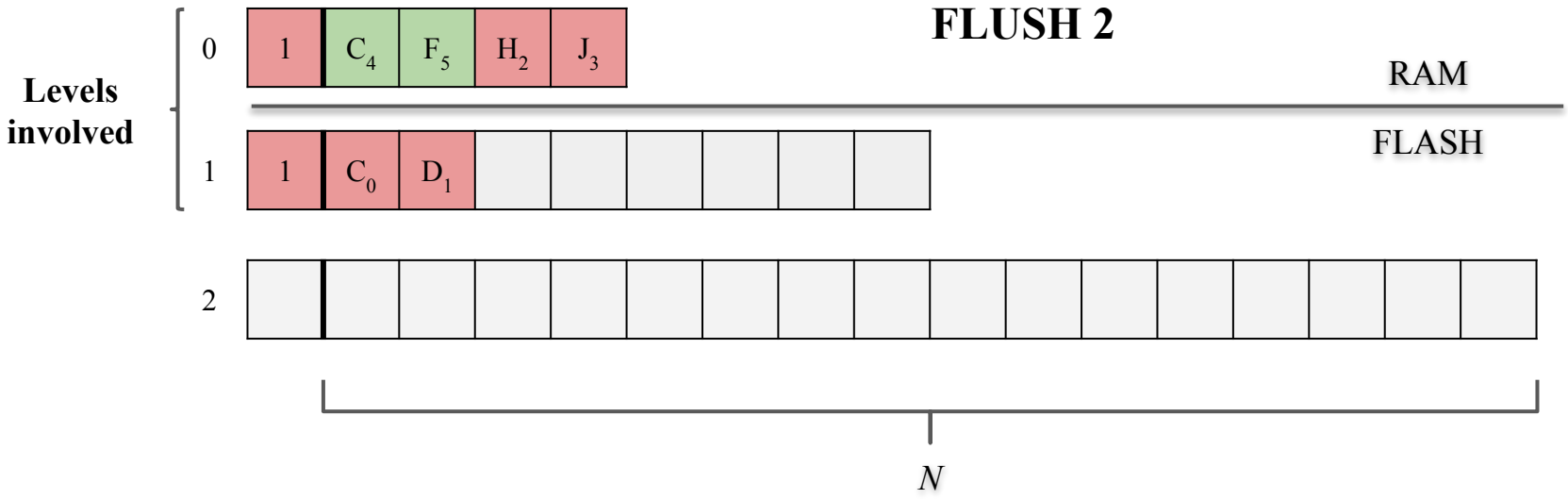
Time-stretch filter

Reporting threshold: 2



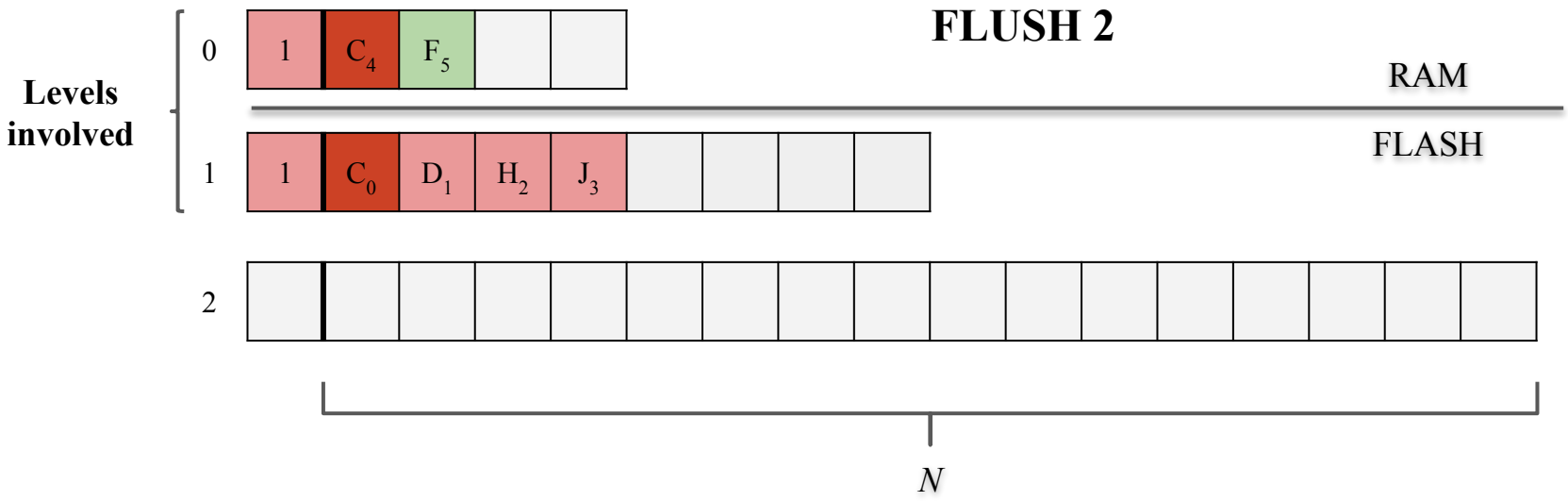
Time-stretch filter

Reporting threshold: 2



Time-stretch filter

Reporting threshold: 2



$$I_1 = 0, I_T = 4, I_R = 5$$

$$\alpha = \frac{5-1}{4-1} = 1.33$$