Timely Reporting of Heavy Hitters using External Memory

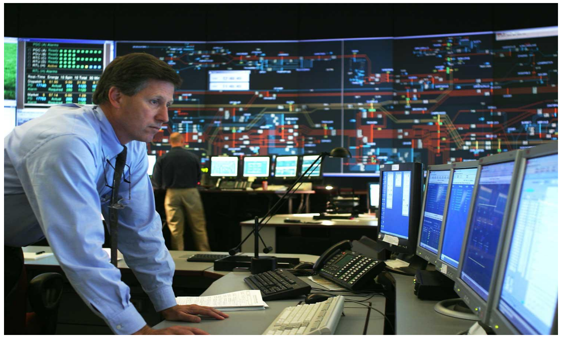
Prashant Pandey, Shikha Singh, Michael A. Bender, Jonathan W. Berry,
Martin Farach-Colton, Rob Johnson, Thomas M. Kroeger,
Cynthia A. Phillips

Open problem from Sandia National Labs

- A **high-speed stream** of key-value pairs arriving over time
- **Goal:** report every key **as soon as** it appears **24 times** without missing any

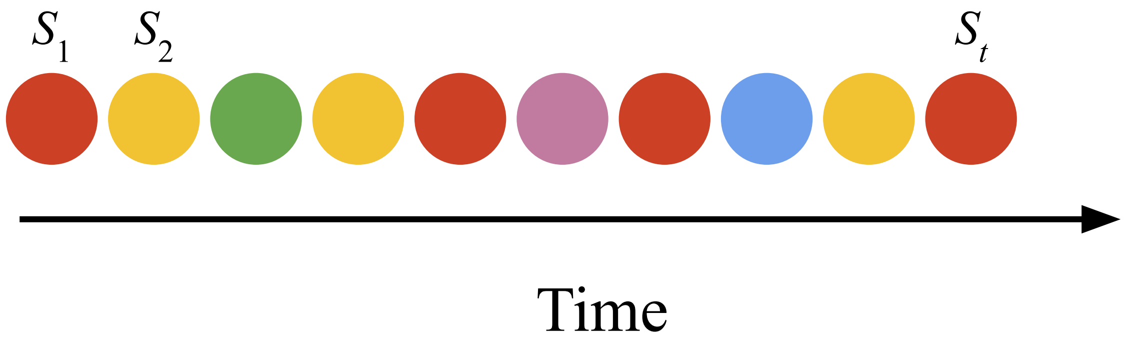
Why should we care about this problem

- Defense systems for cyber security monitor high-speed stream
- Malicious traffic forms a small portion of the stream
- Automated systems take defensive actions for every reported event.
- Firehose benchmark simulates the stream
    - https://firehose.sandia.gov/



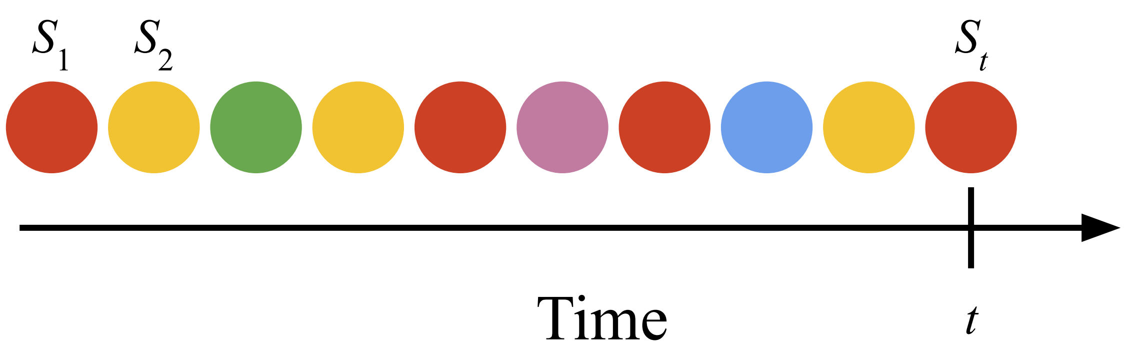
Timely event detection problem

- Stream of elements arrive over time



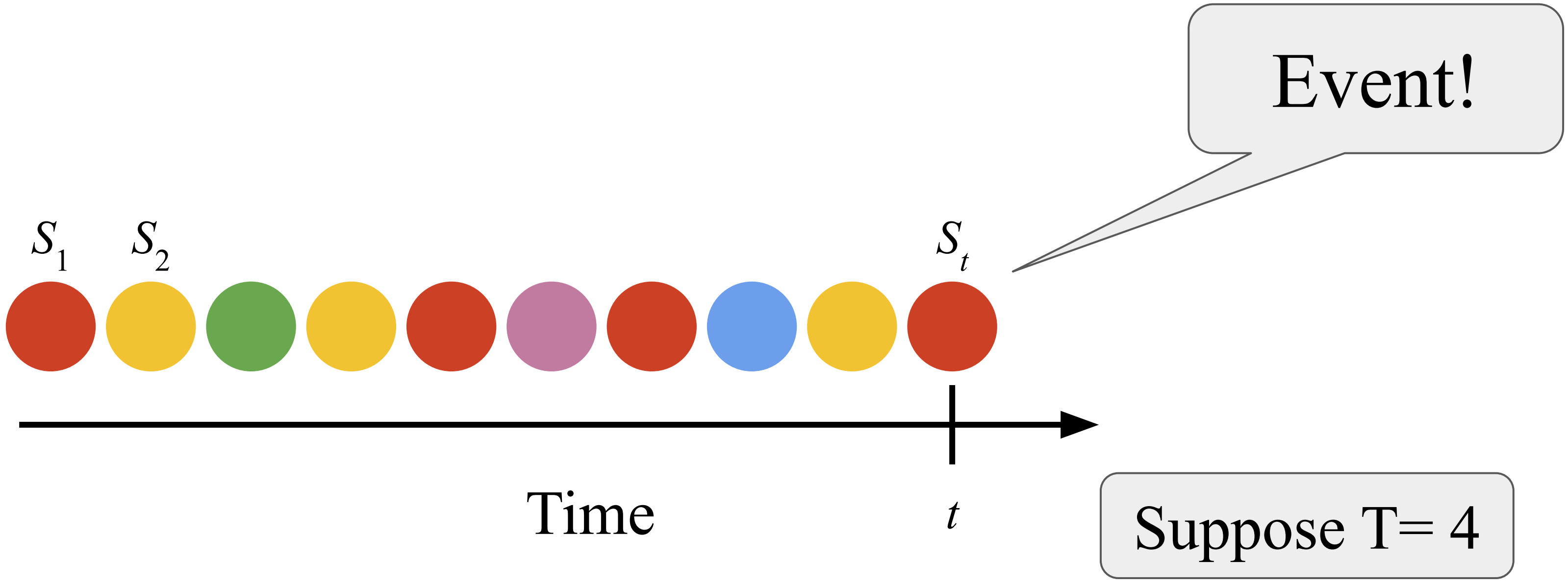
Timely event detection problem

- Stream of elements arrive over time
- An **event** occurs at time t if S_t occurs exactly T times in $(s_1, s_2 \ldots . s_t)$



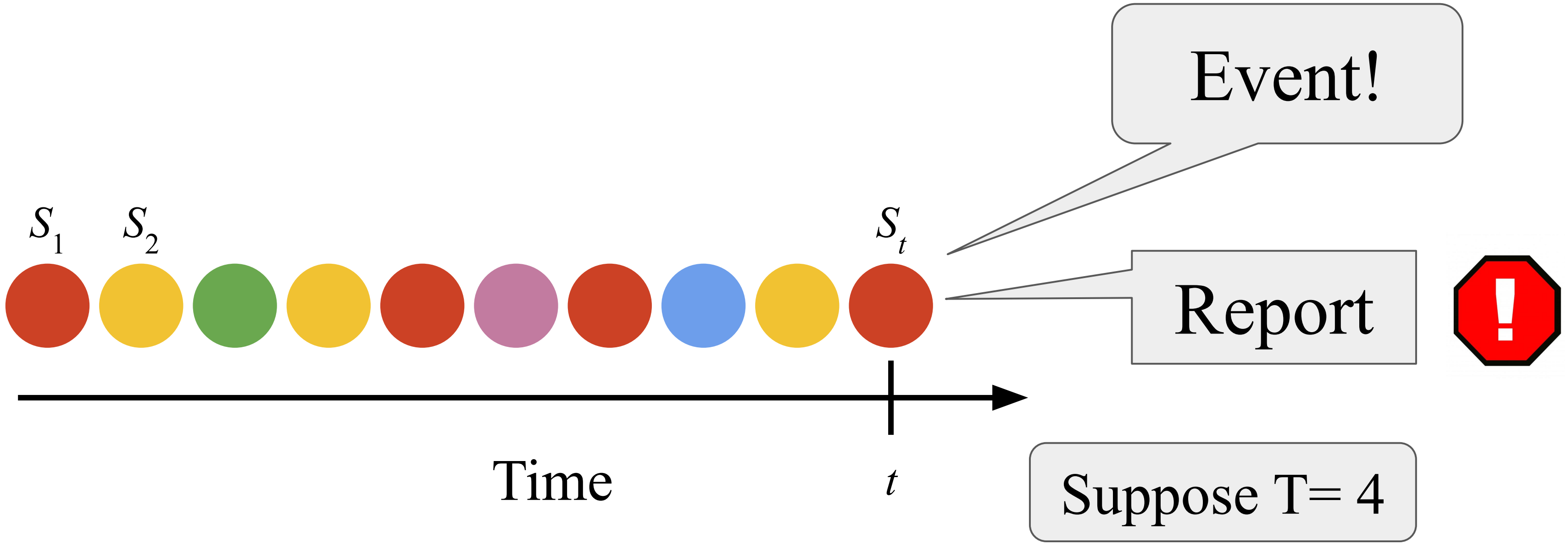
Timely event detection problem

- Stream of elements arrive over time
- An **event** occurs at time t if S_t occurs exactly T times in $(s_1, s_2 \ldots s_t)$



Timely event detection problem

- Stream of elements arrive over time
- An **event** occurs at time t if S_t occurs exactly T times in $(s_1, s_2 \ldots . s_t)$
- In **timely event-detection problem (TED)**, we want to report all events shortly after they occur.



Features we need in the solution

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput ingestion

Features we need in the solution

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput ingestion

- Events are high-consequence real-life events

No false-negatives; few false-positives

Timely reporting (real-time)

Features we need in the solution

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput ingestion

- Events are high-consequence real-life events

No false-negatives; few false-positives

Timely reporting (real-time)

- Very small reporting threshold $T << N$ (stream size)

Very small reporting thresholds

Features we need in the solution

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput i

- Events are high-co

No false-negatives

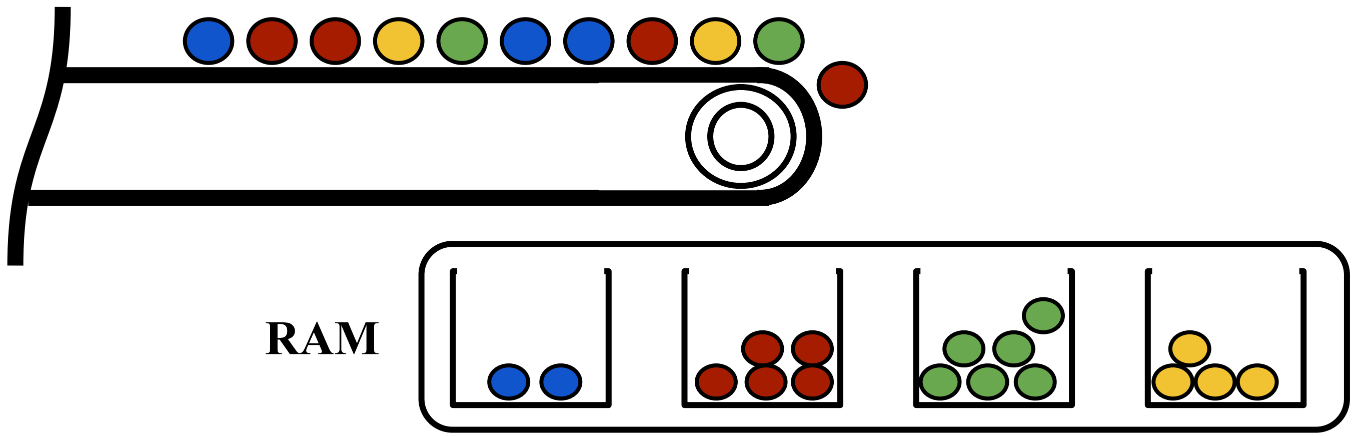
Timely reporting (

- Very small reportin size)

Very small reporting thresholds

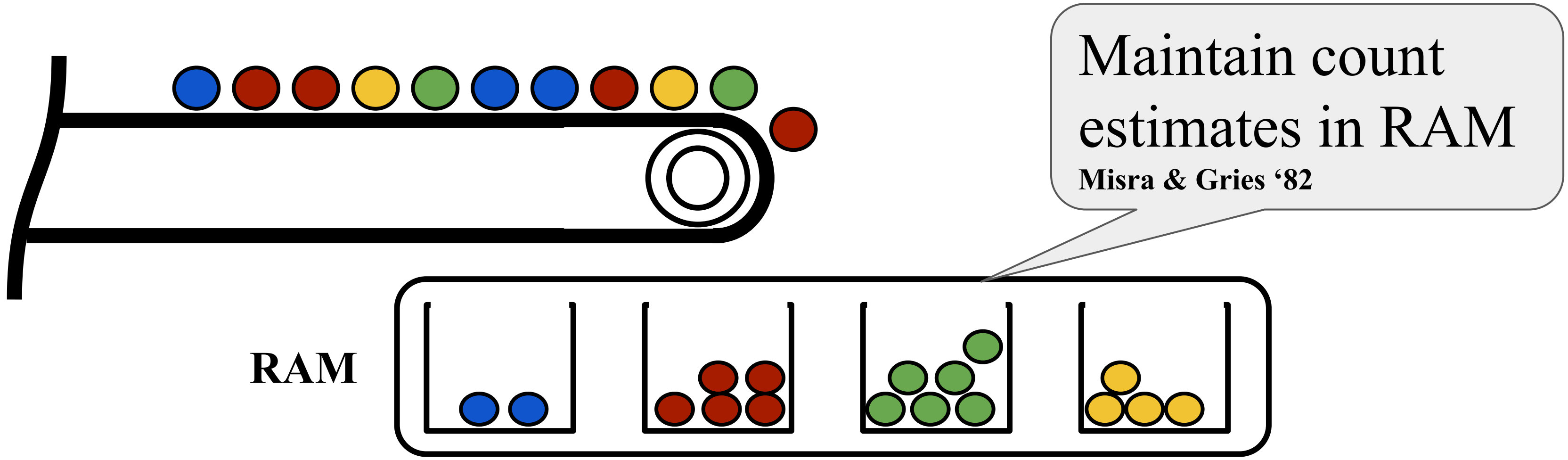
One-pass streaming has errors

- **Heavy hitter problem:** report items whose frequency $\geq \varphi N$
- Exact one-pass solution solution requires $\Omega(N)$ space



One-pass streaming has errors

- **Approximate solution**: report all items with count **≥ φ*N*,** none with **< *(φ−ε)N*** [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.]
- Approximate solutions requires: **Ω(1/ε)**

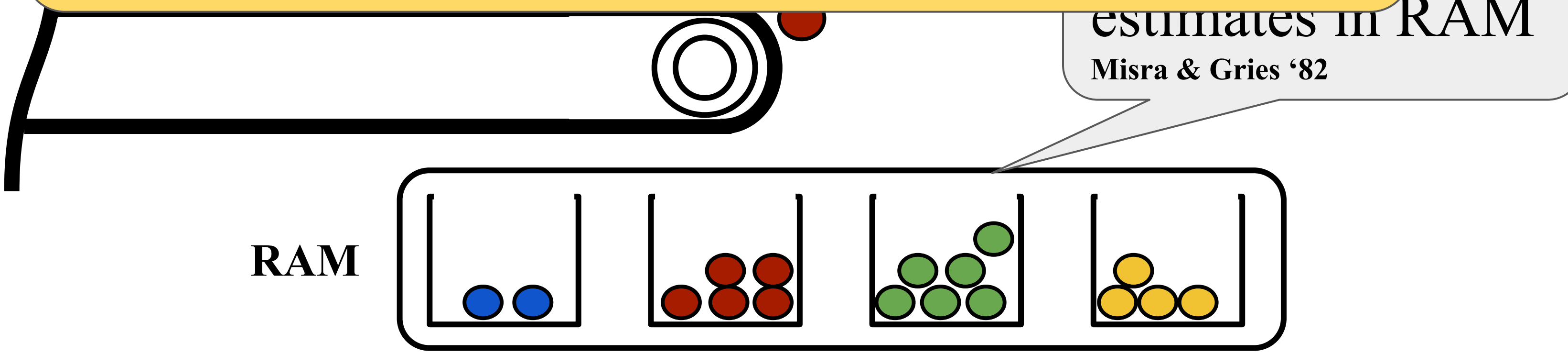


Real time with false-positives!

One-pass streaming has errors

- **Approximate solution**: report all items with count ≥ φN, none with < $(\varphi-\varepsilon)N$ [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.]
- Approximate solutions requires: $\Omega(1/\varepsilon)$

For Sandia, φN is a small constant (24),
So $\Omega(1/\varepsilon)$ is very very large!!
Can’t solve in RAM for very small φ



Real time with false-positives!

One-pass solution has:

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput ingestion ✓

- Events are high-consequence real-life events

No false-negatives; few false-positives ✓

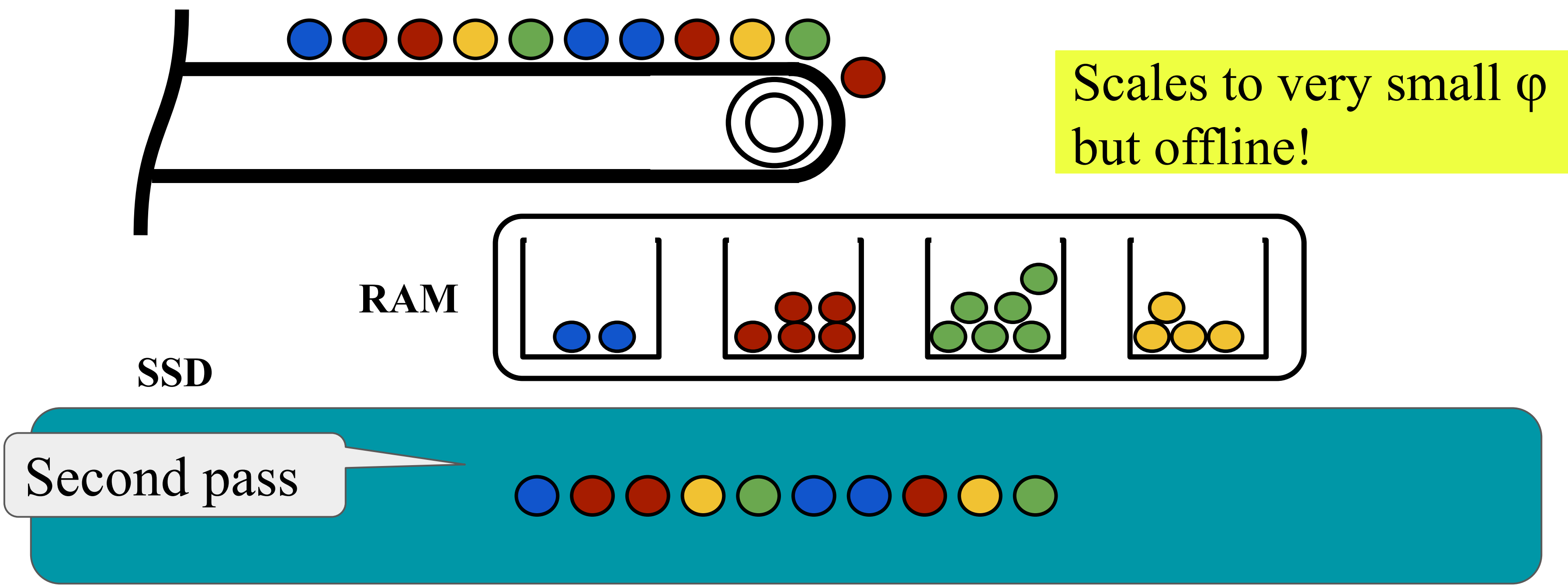
Timely reporting (real-time) ✓

- Very small reporting threshold $T << N$ (stream size)

Very small reporting thresholds ✗

Two-pass streaming isn't real-time

- A second pass over the stream can get rid of errors
- Store the stream on SSD and access it later



Two-pass solution has:

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput ingestion

- Events are high-consequence real-life events

No false-negatives; few false-positives

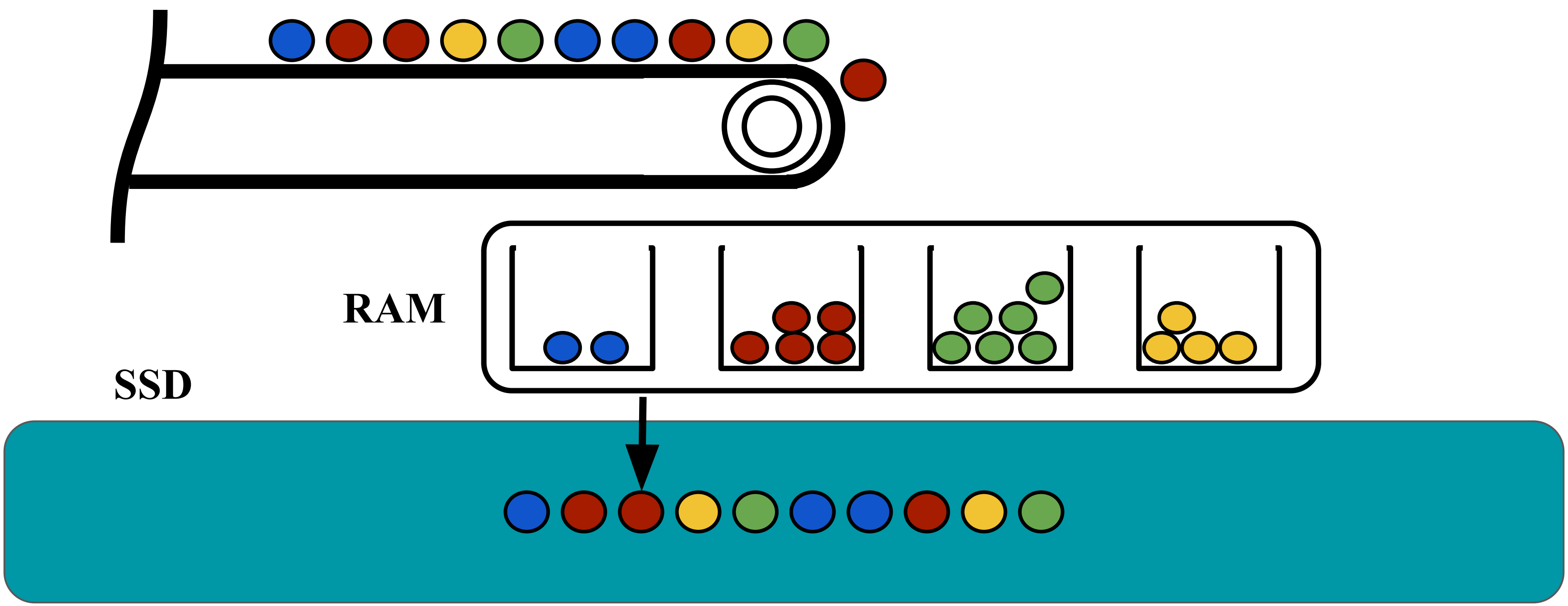
Timely reporting (real-time)

- Very small reporting threshold $T << N$ (stream size)

Very small reporting thresholds

If data is stored: why not access it?

Why wait for second pass?



Our contribution

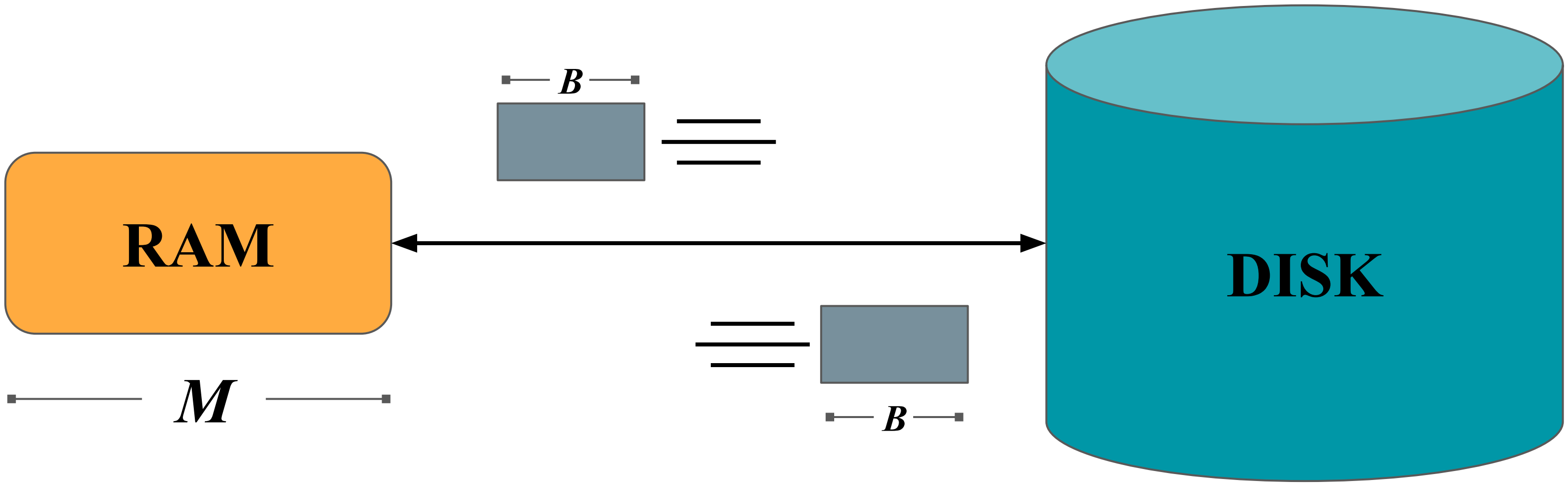
Streaming model

External memory algorithms

Combine streaming and EM algorithms to solve real-time event detection problem

External memory model Aggarwal+Vitter ‘08

- **How computations work:**
  - Data is transferred in blocks between RAM and disk.
  - The number of block transfers dominate the running time.
- **Goal: Minimize number of block transfers**
  - Performance bounds are parameterized by block size B, memory size M, data size N.



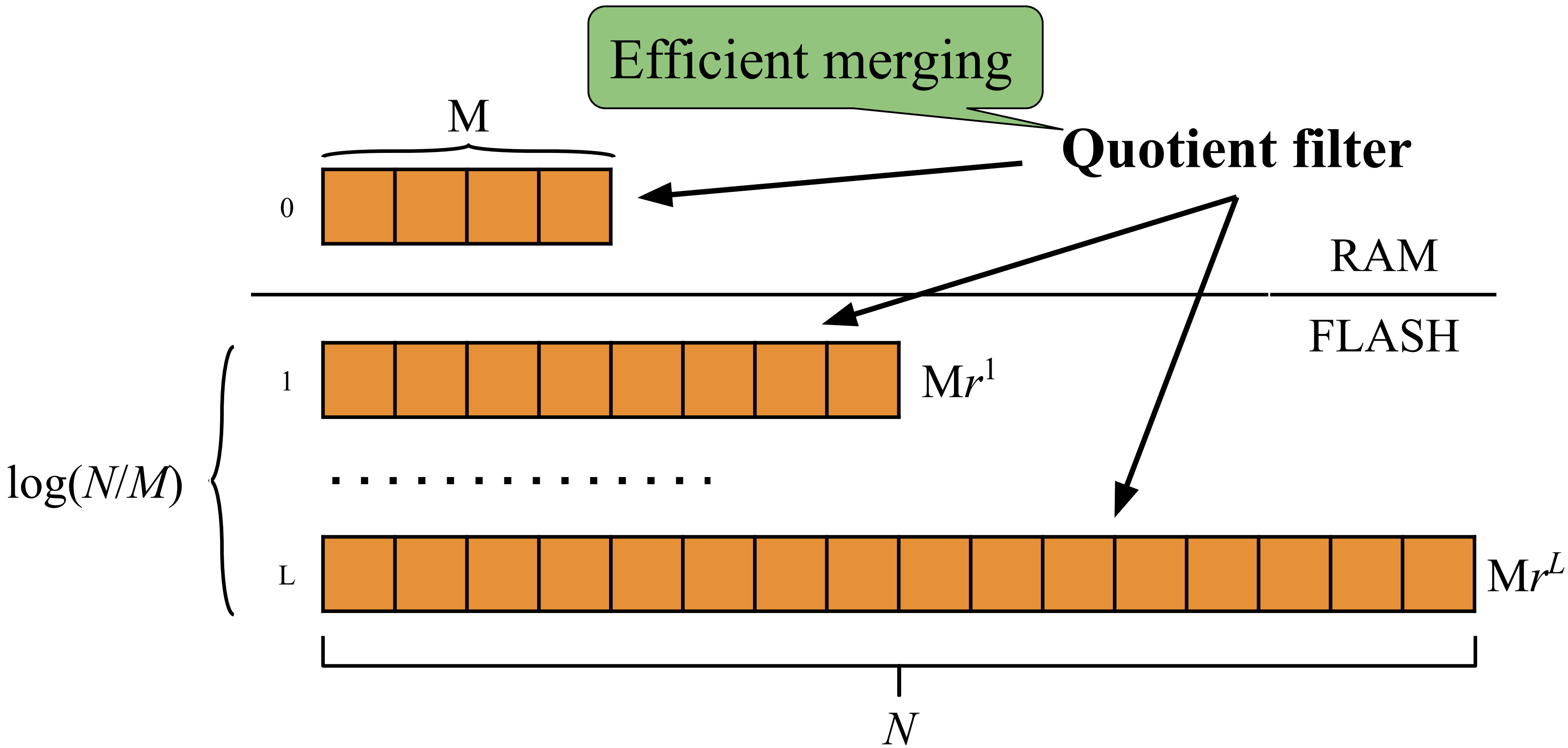
Counting quotient filter Pandey et al. '17

- **Maintains item counts using a variable length encoding**
  - Asymptotically optimal space: $O(\sum |C(x)|)$
- **Good cache locality**
- **Enumerability/Mergeability**
- **Efficient scaling out-of-RAM**
- **Deletions**

We build an efficient EM counting data structure using the quotient filter.

Cascade filter: write-optimized quotient filter

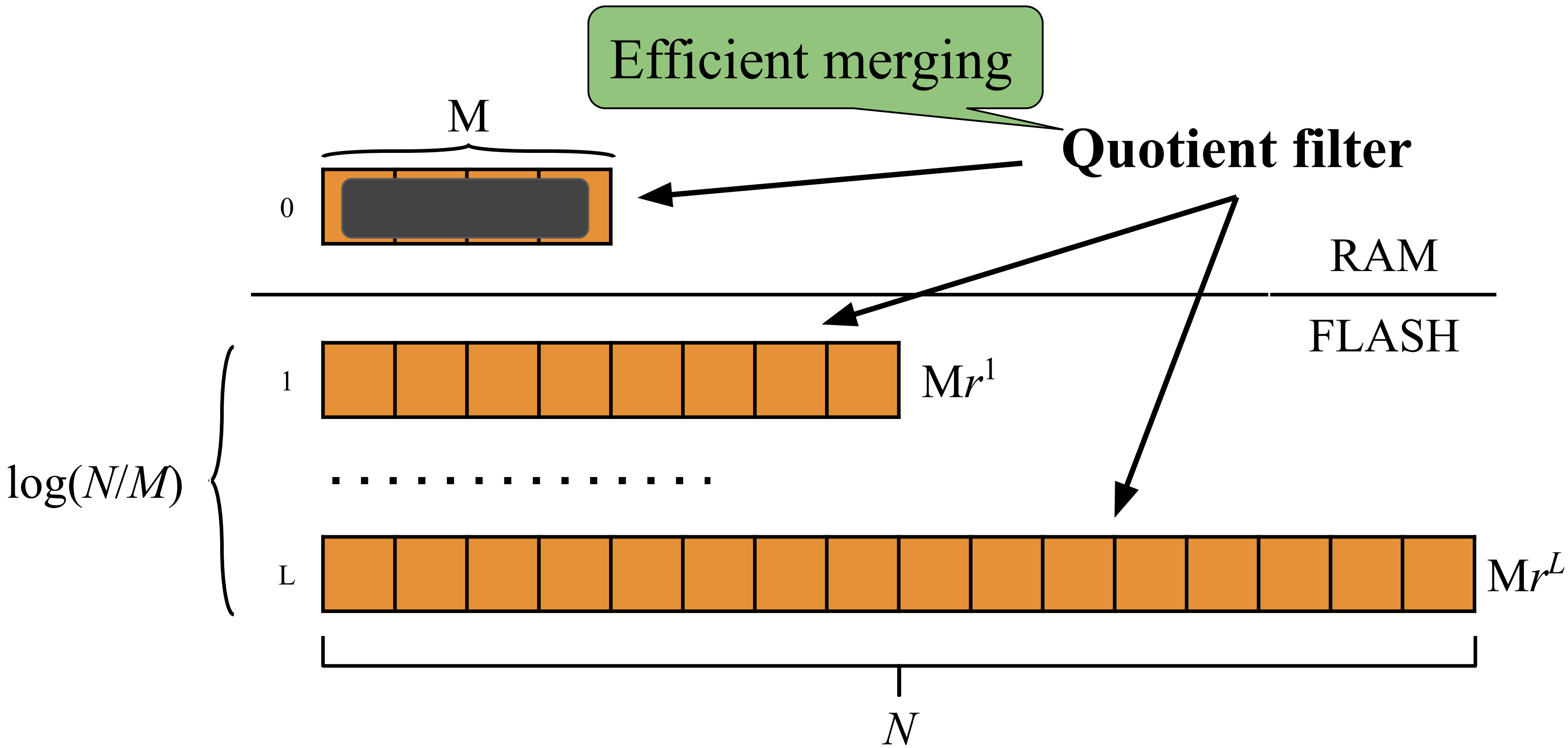
Bender et al. '12, Pandey et al. '17



- The Cascade filter efficiently scales out-of-RAM
- It accelerates insertions at some cost to queries

Cascade filter: flushing

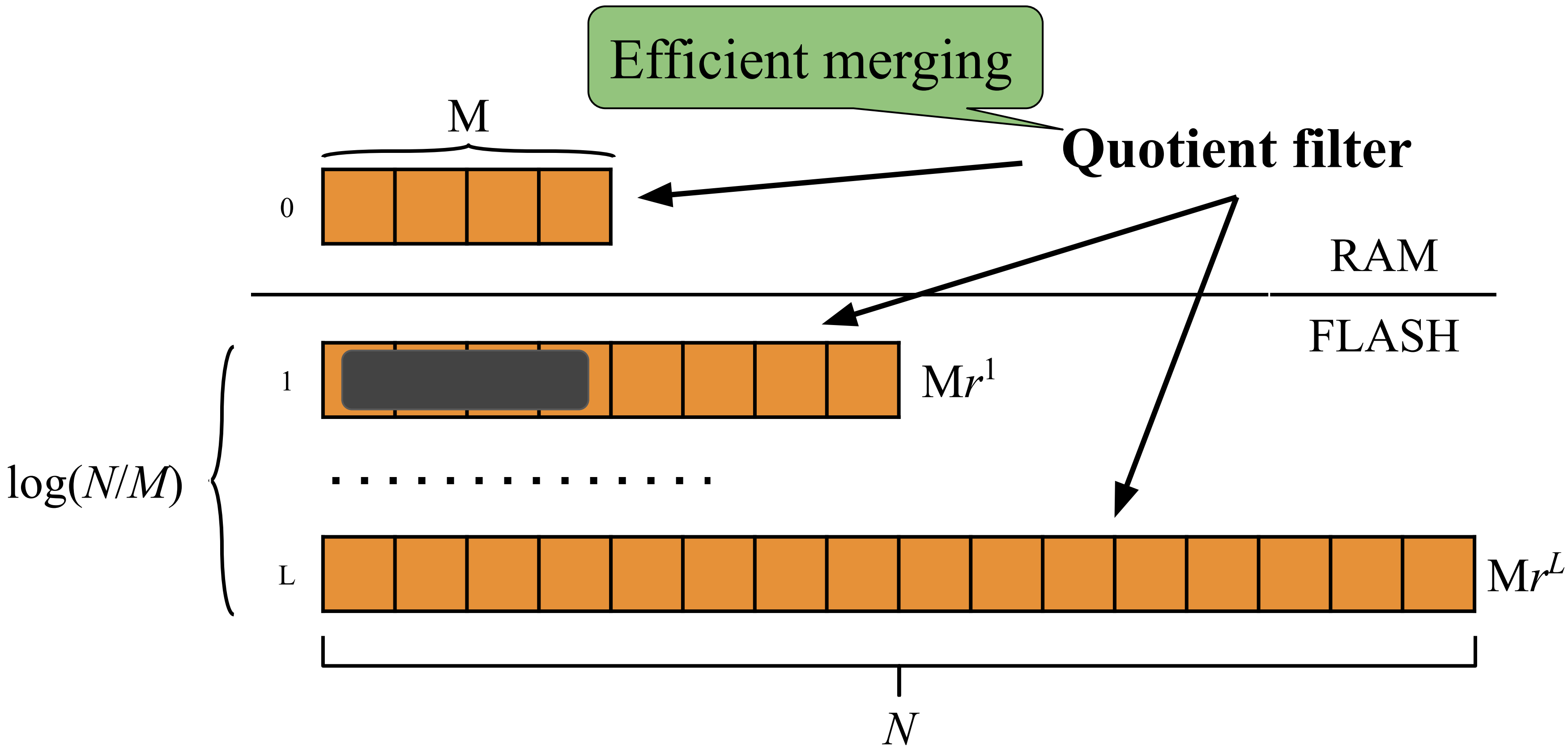
Bender et al. ‘12, Pandey et al. ‘17



Items are initially inserted in the RAM level

Cascade filter: flushing

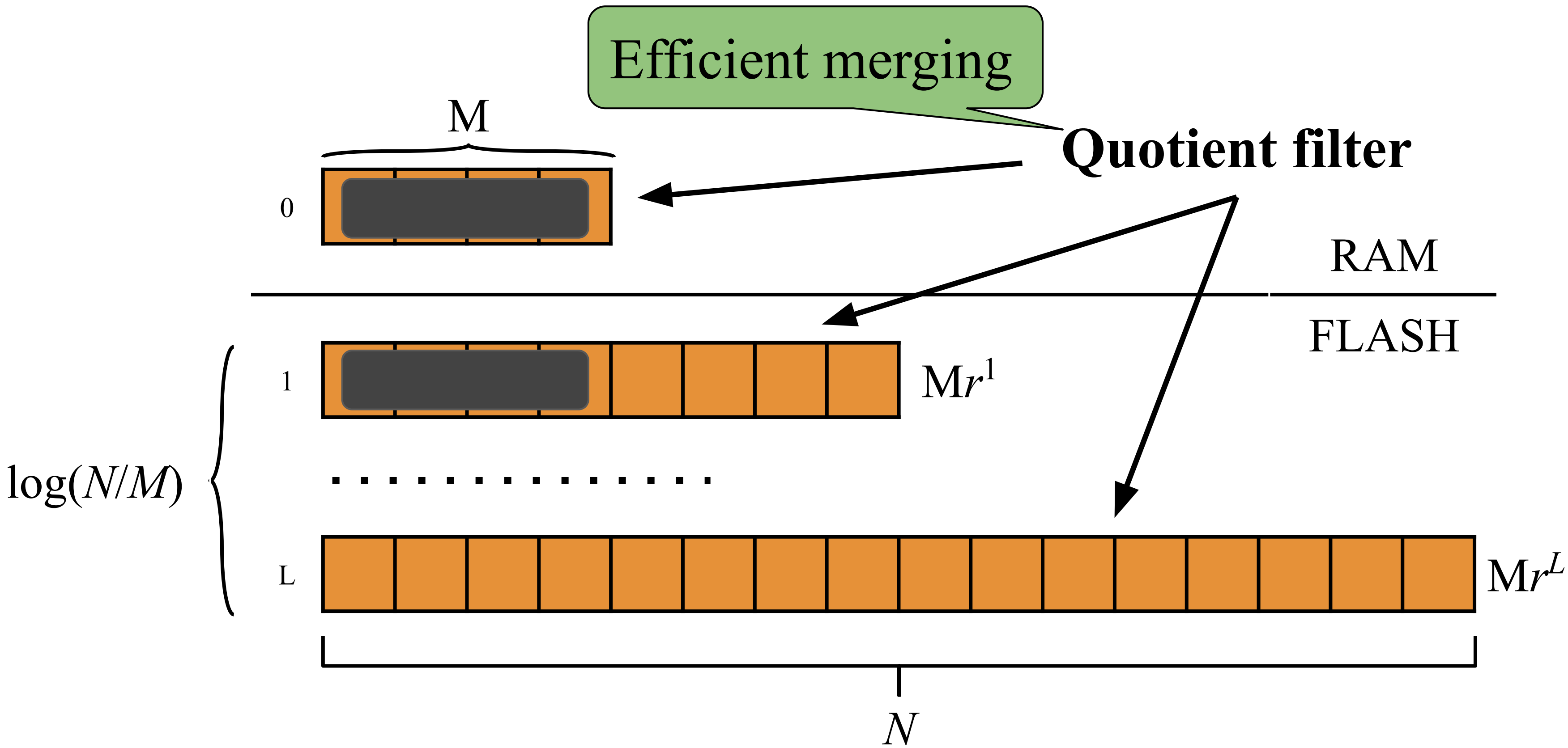
Bender et al. '12, Pandey et al. '17



When RAM is full, items are flushed to the smallest level on disk ***i*** with space to insert items in level **0** to ***i*-1**

Cascade filter: flushing

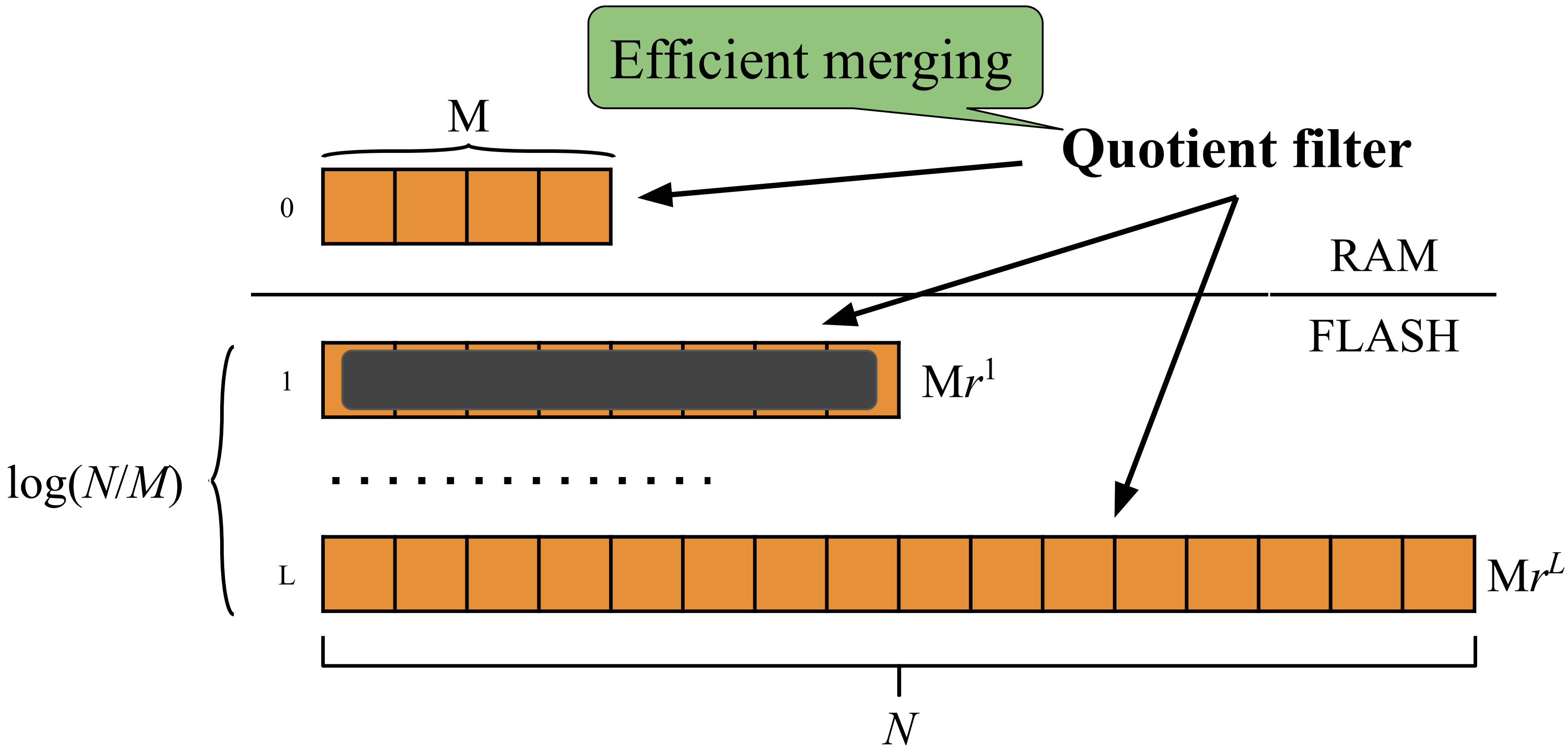
Bender et al. ‘12, Pandey et al. ‘17



When RAM is full, items are flushed to the smallest level on disk i with space to insert items in level **0** to i**-1**

Cascade filter: flushing

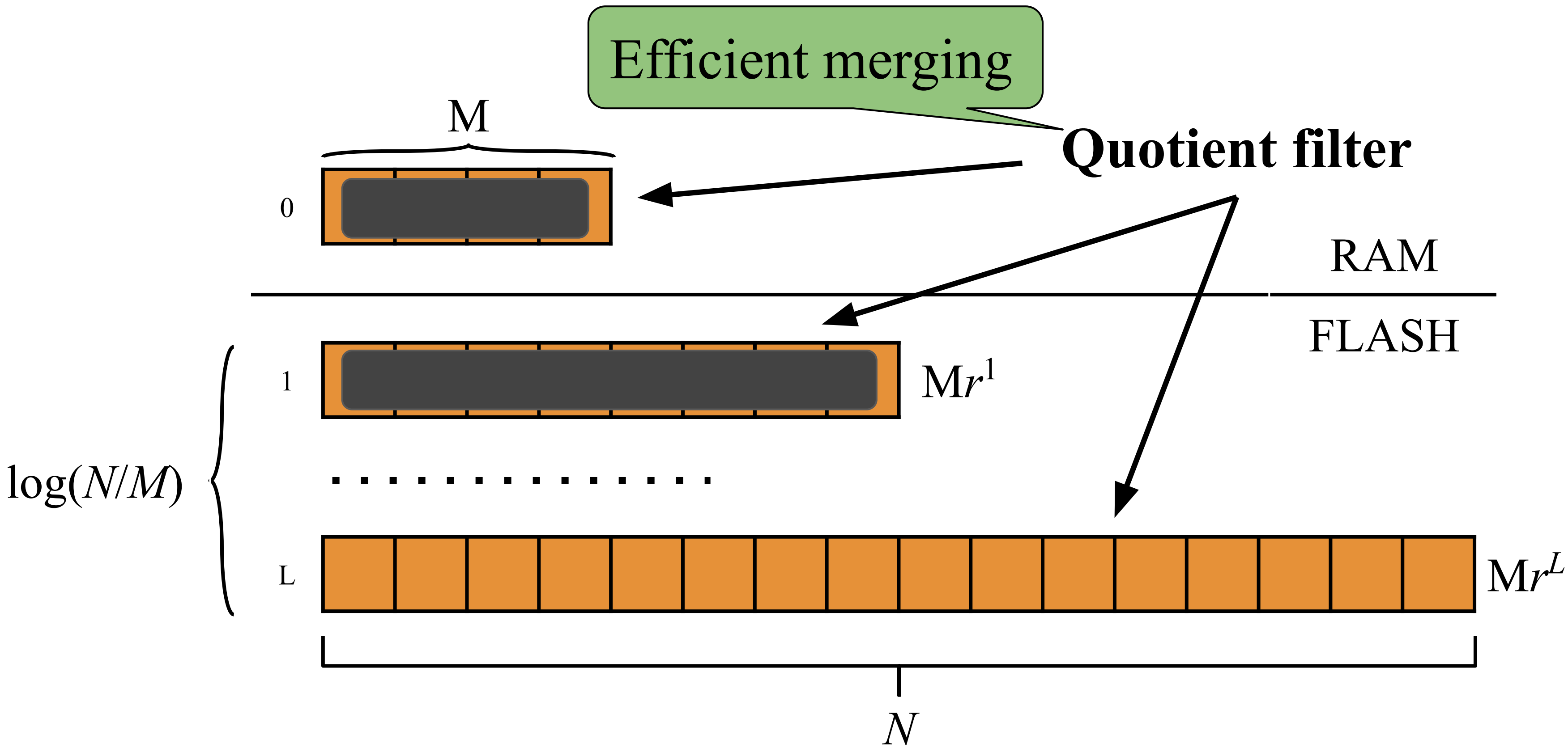
Bender et al. '12, Pandey et al. '17



When RAM is full, items are flushed to the smallest level on disk ***i*** with space to insert items in level **0** to ***i*-1**

Cascade filter: flushing

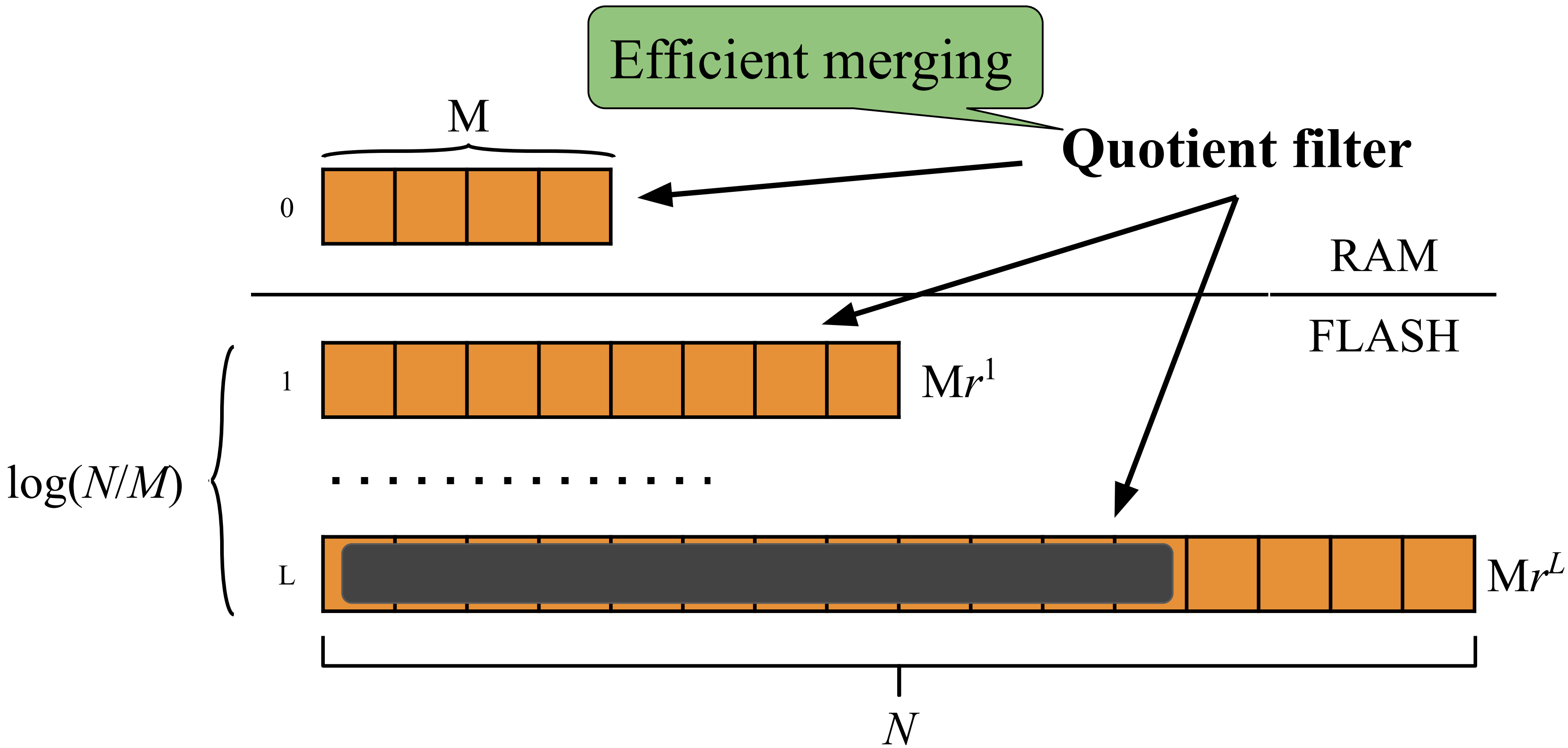
Bender et al. ‘12, Pandey et al. ‘17



When RAM is full, items are flushed to the smallest level on disk i with space to insert items in level **0** to i-**1**

Cascade filter: flushing

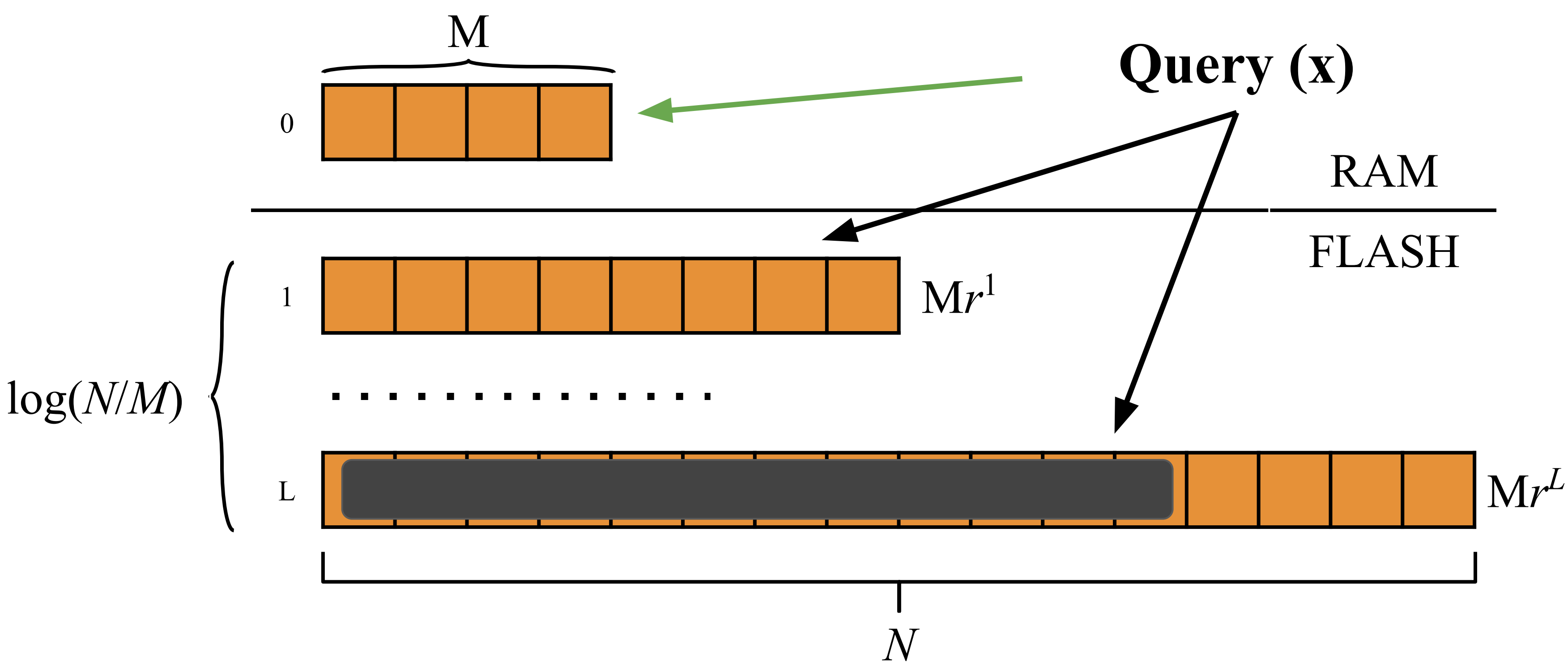
Bender et al. ‘12, Pandey et al. ‘17



When RAM is full, items are flushed to the smallest level on disk ***i*** with space to insert items in level **0** to ***i*-1**

Cascade filter: query

Bender et al. ‘12, Pandey et al. ‘17



A query operation requires a lookup in each non-empty level

Cascade filter operations

Insert	Query
$O\left(\frac{1}{B}\log\frac{N}{M}\right)$	$O\left(\log\frac{N}{M}\right)$

Cascade filter operations

Insert	Query
$O\left(\frac{1}{B}\log\frac{N}{M}\right)$	$O\left(\log\frac{N}{M}\right)$

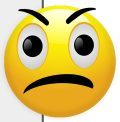
< 1 I/O per observation

Cascade filter operations

Insert	Query
$O\left(\frac{1}{B}\log\frac{N}{M}\right)$	$O\left(\log\frac{N}{M}\right)$

<1 I/O per observation

>1 I/O per observation



Cascade filter doesn't have real-time reporting

But every insert is also a query in real-time reporting!

Insert	Query
$O\left(\frac{1}{B}\log\frac{N}{M}\right)$	$O\left(\log\frac{N}{M}\right)$

< 1 I/O per observation

> 1 I/O per observation

Cascade filter doesn't have real-time reporting

But every insert is also a query in real-time reporting!

Insert	Query
$O\left(\frac{1}{B}\log\frac{N}{M}\right)$	$O\left(\log\frac{N}{M}\right)$

Traditional cascade filter doesn't solves the problem! But we can use insights

observation 

observation 

This talk: Leveled External-Memory Reporting Table (LERT)

- Given a stream of size N and $\varphi N > \mathbf{\Omega(N/M)}$ the amortized cost of solving real-time event detection is

$$O\left(\left(\frac{1}{B} + \frac{1}{(\phi - 1/M)N}\right) \log \frac{N}{M}\right)$$

- For a **constant time stretch** in reporting, can support arbitrarily small thresholds φ with amortized cost

$$O\left(\frac{1}{B} \log \frac{N}{M}\right)$$

Takeaway: Online reporting comes at the cost of throughput but almost online reporting is essentially free!

This talk: Leveled External-Memory Reporting Table (LERT)

- Given a stream of size N and $\varphi N > \boldsymbol{\Omega(N/M)}$ the amortized cost of solving real-time event detection is

Can achieve timely reporting at effectively the optimal insert cost; no query cost

- arbitrarily small thresholds φ with amortized cost

$$O\left(\frac{1}{B}\log\frac{N}{M}\right)$$

Takeaway: Online reporting comes at the cost of throughput but almost online reporting is essentially free!

This talk: Leveled External-Memory Reporting Table (LERT)

- Given a stream of size N and $\varphi N > \boldsymbol{\Omega(N/M)}$ the amortized cost of solving real-time event detection is

$$O\left(\left(\frac{1}{B} + \frac{1}{(\phi - 1/M)N}\right)\log\frac{N}{M}\right)$$

- For a **constant time stretch** in reporting, can support arbitrarily small thresholds φ with amortized cost

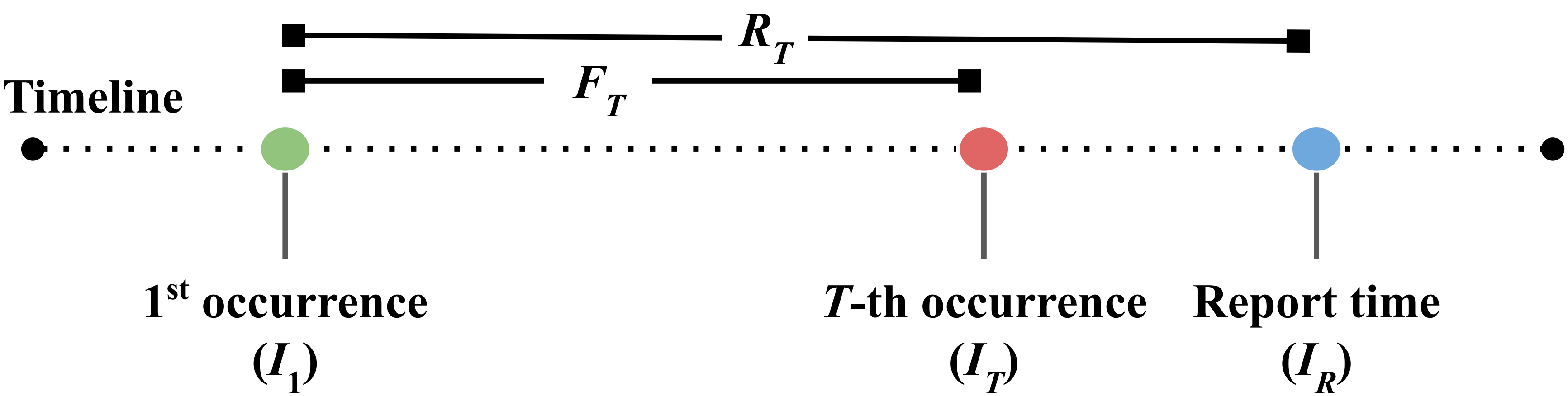
$$O\left(\frac{1}{B}\log\frac{N}{M}\right)$$

This talk!

ne reporting comes at the cost of throughput but almost online reporting is essentially free!

Time stretch

For a **time-stretch** of $1+\alpha$, we must report an element a no later than time $I_1 + (1+\alpha)F_T$, where F_T is the flow time of a

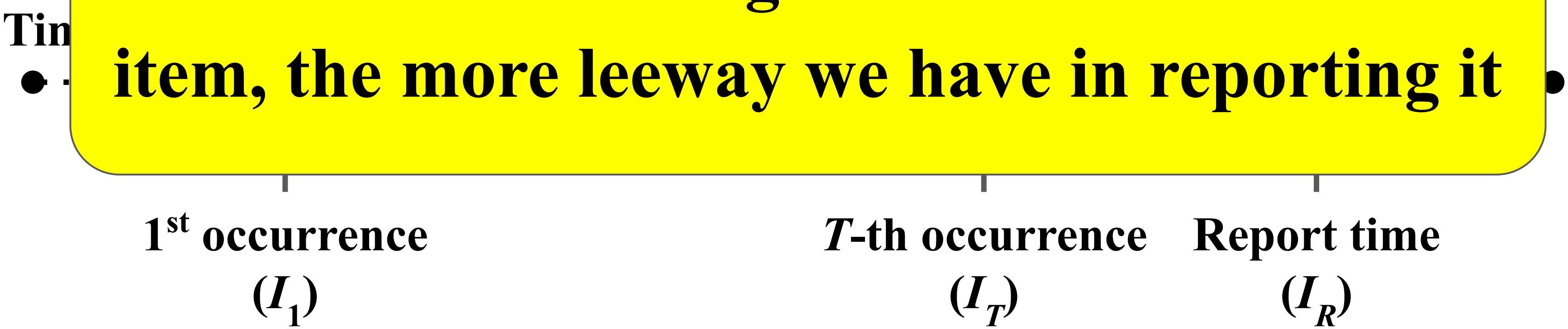


$$\alpha = \frac{R_T}{F_T} - 1$$

Time stretch

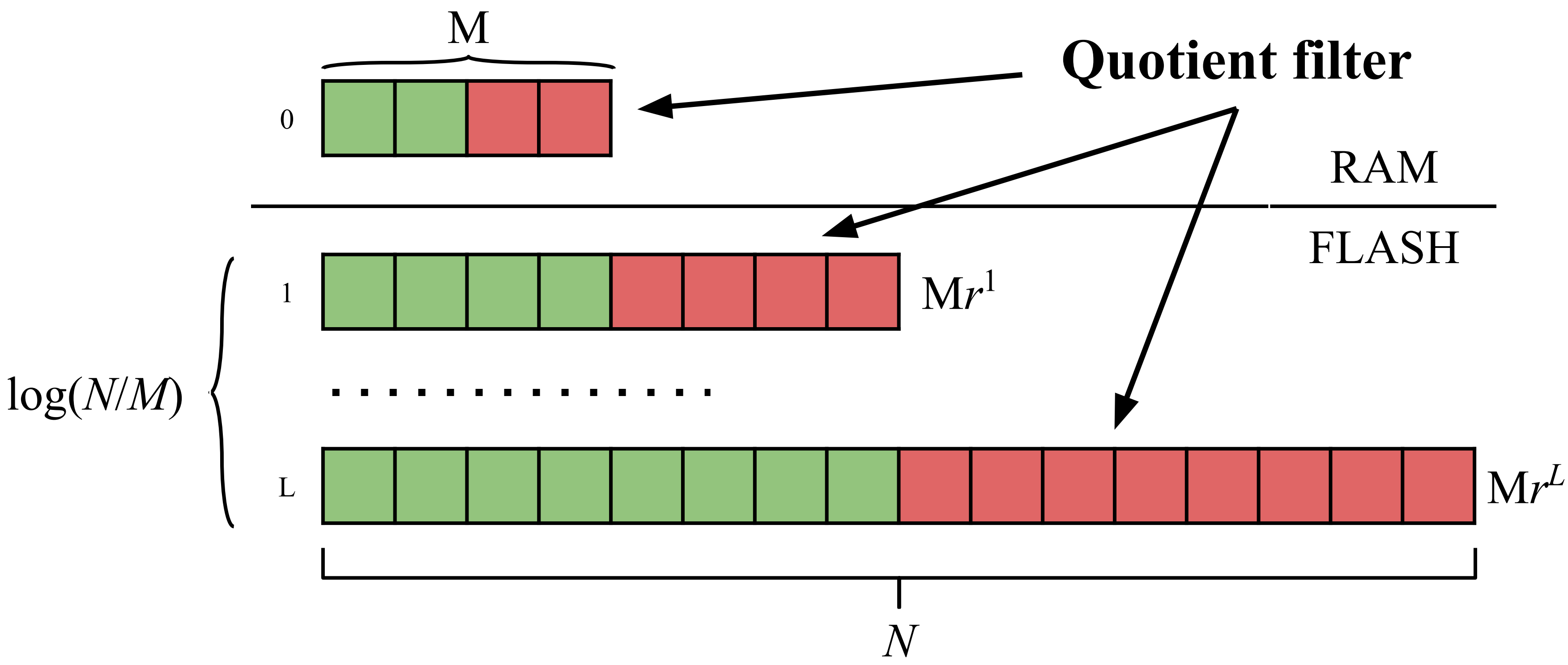
For a **time-stretch** of $1+\alpha$, we must report an element a no later than time $I_1 + (1+\alpha)F_T$, where F_T is the flow time of a

Main idea: the longer the flow time of an item, the more leeway we have in reporting it



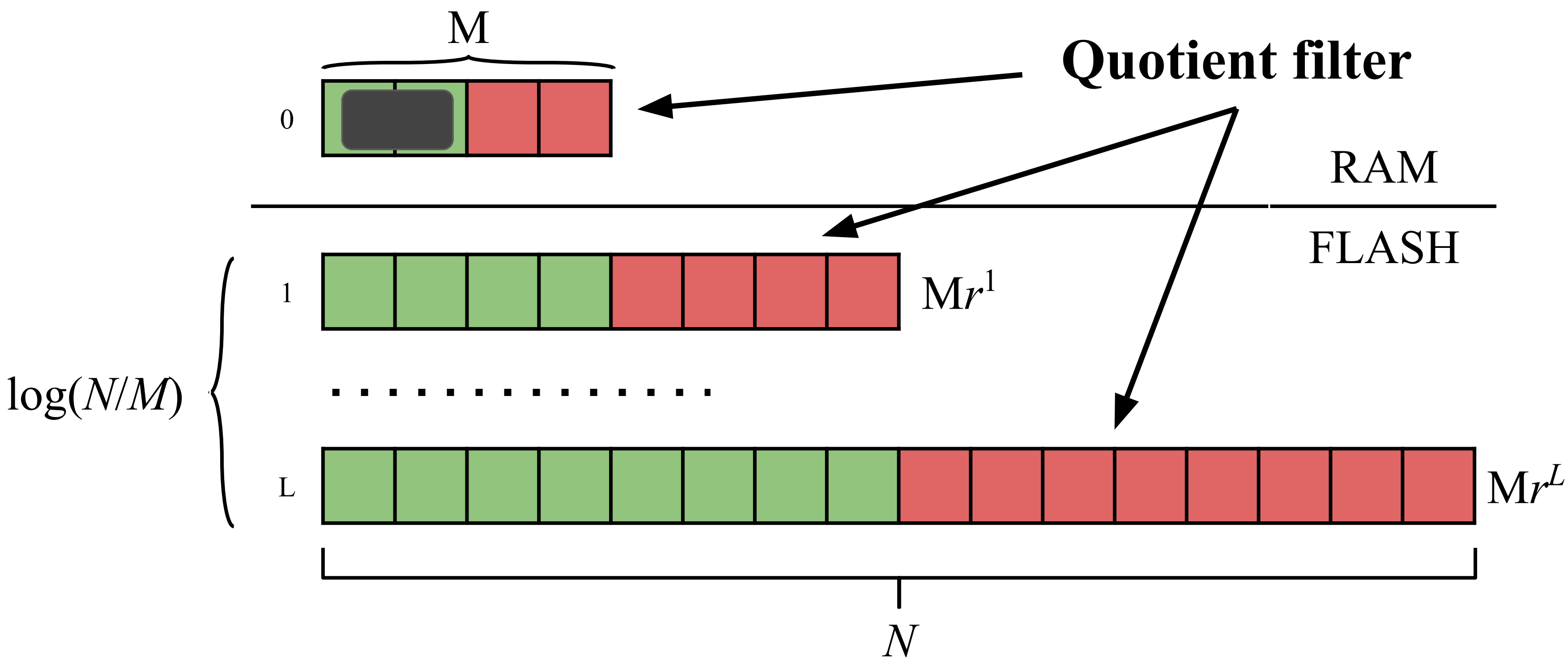
$$\alpha = \frac{R_T}{F_T} - 1$$

Time-stretch LERT



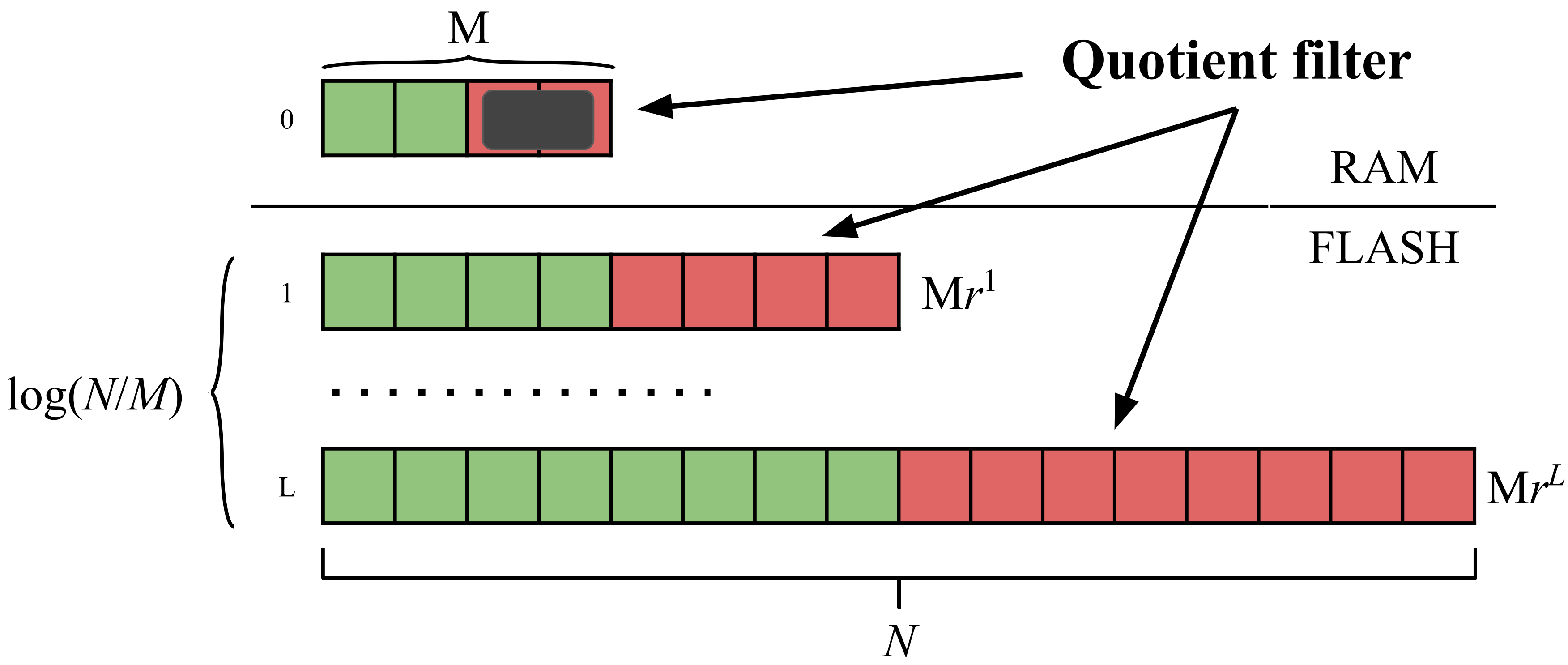
Divide each level into $1+1/\alpha$, equal-sized bins.

Time-stretch LERT



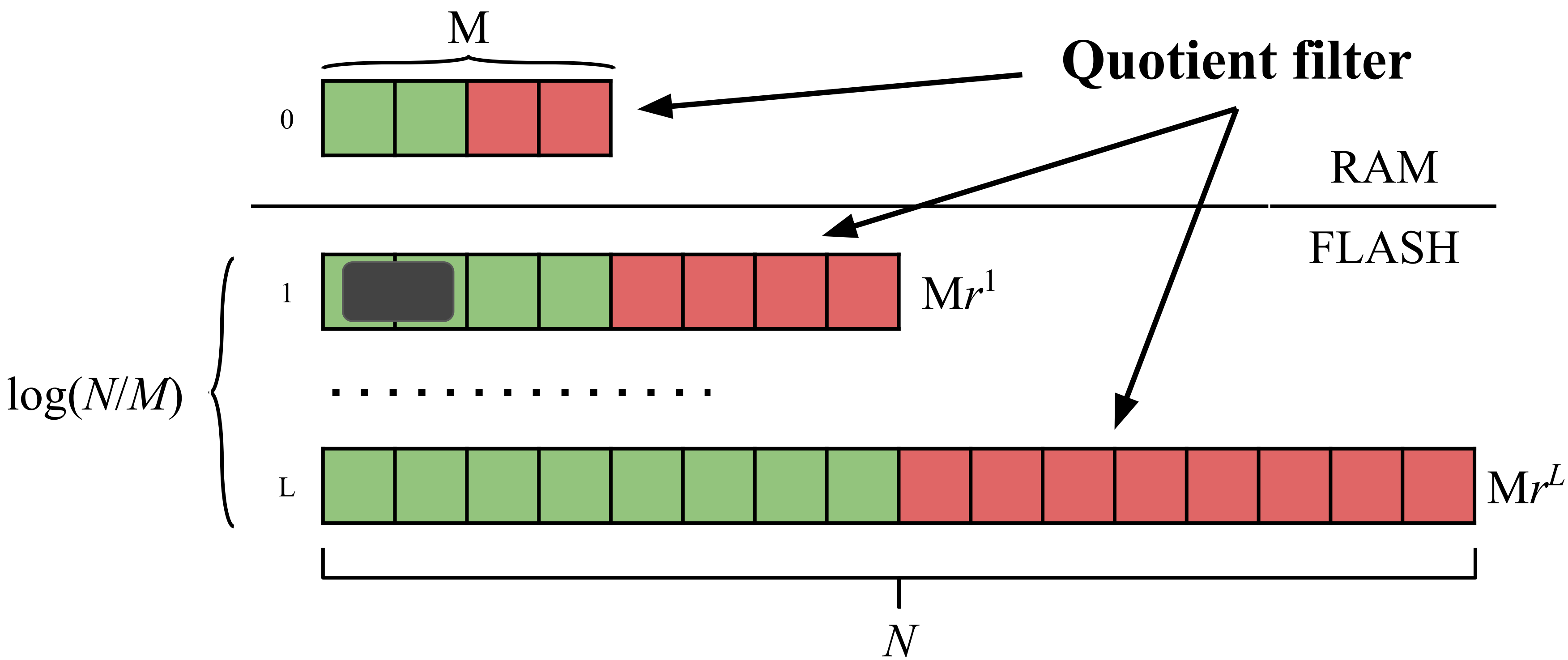
When a bin is full, items move to the adjacent bin

Time-stretch LERT



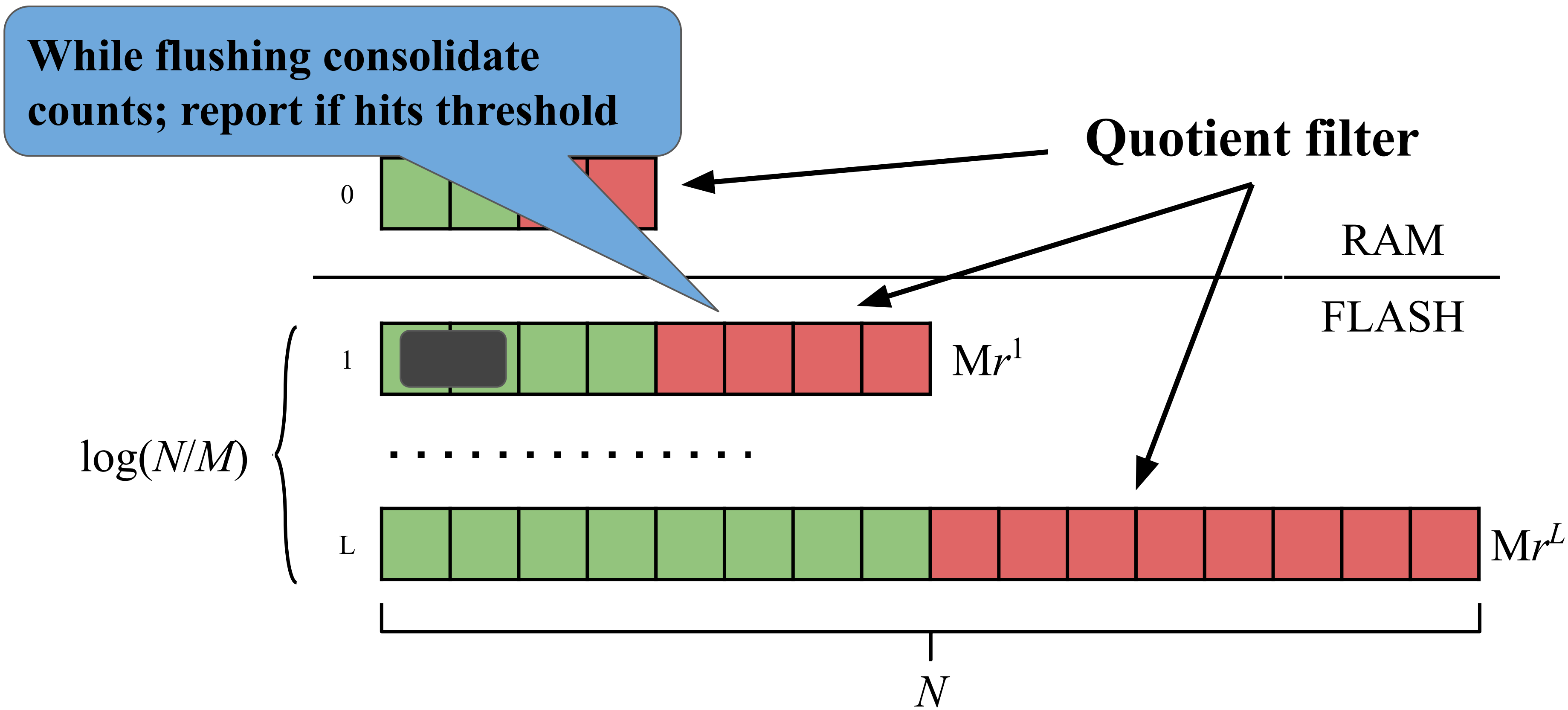
When a bin is full, items move to the adjacent bin

Time-stretch LERT



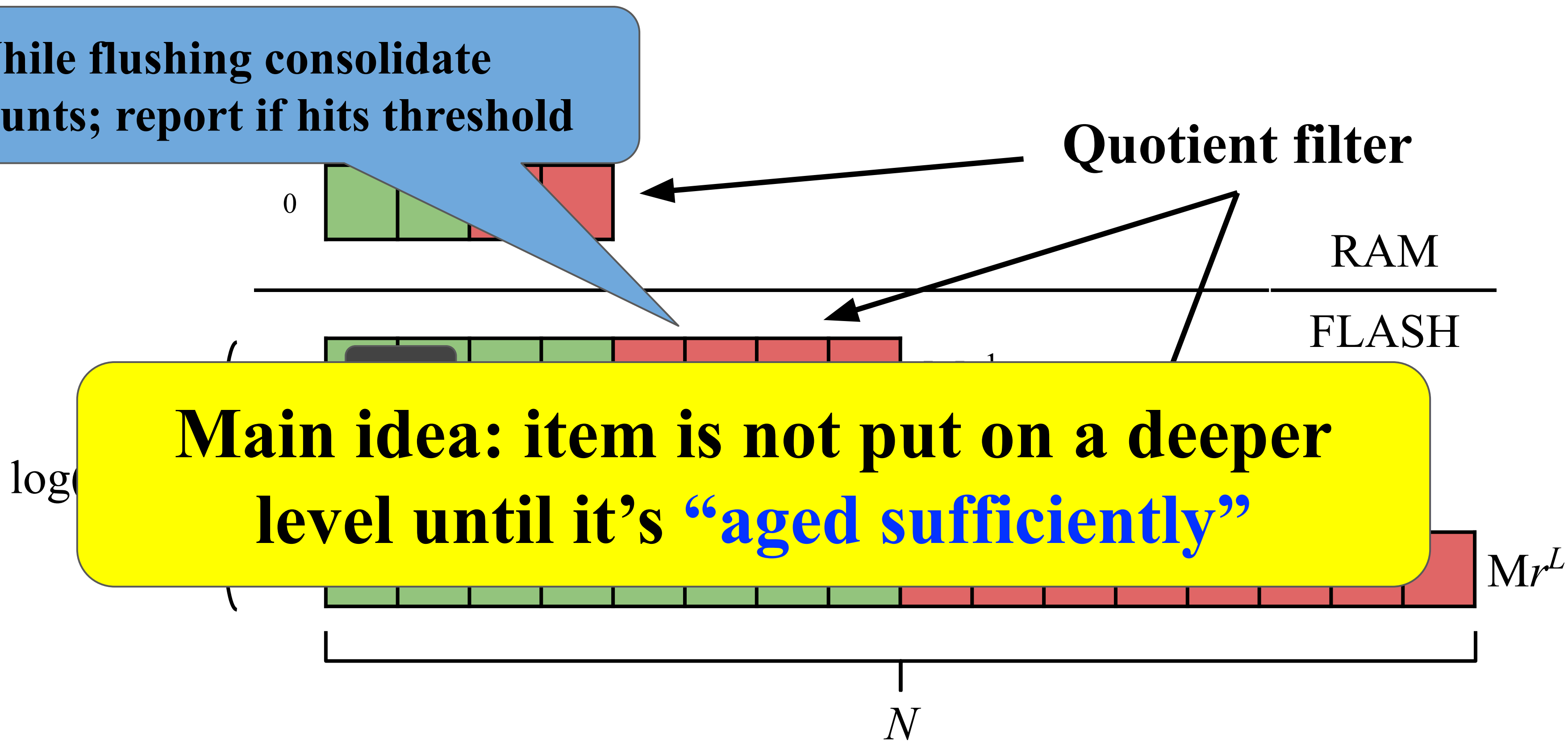
Last bin **flushed** to first bin of the next level

Time-stretch LERT



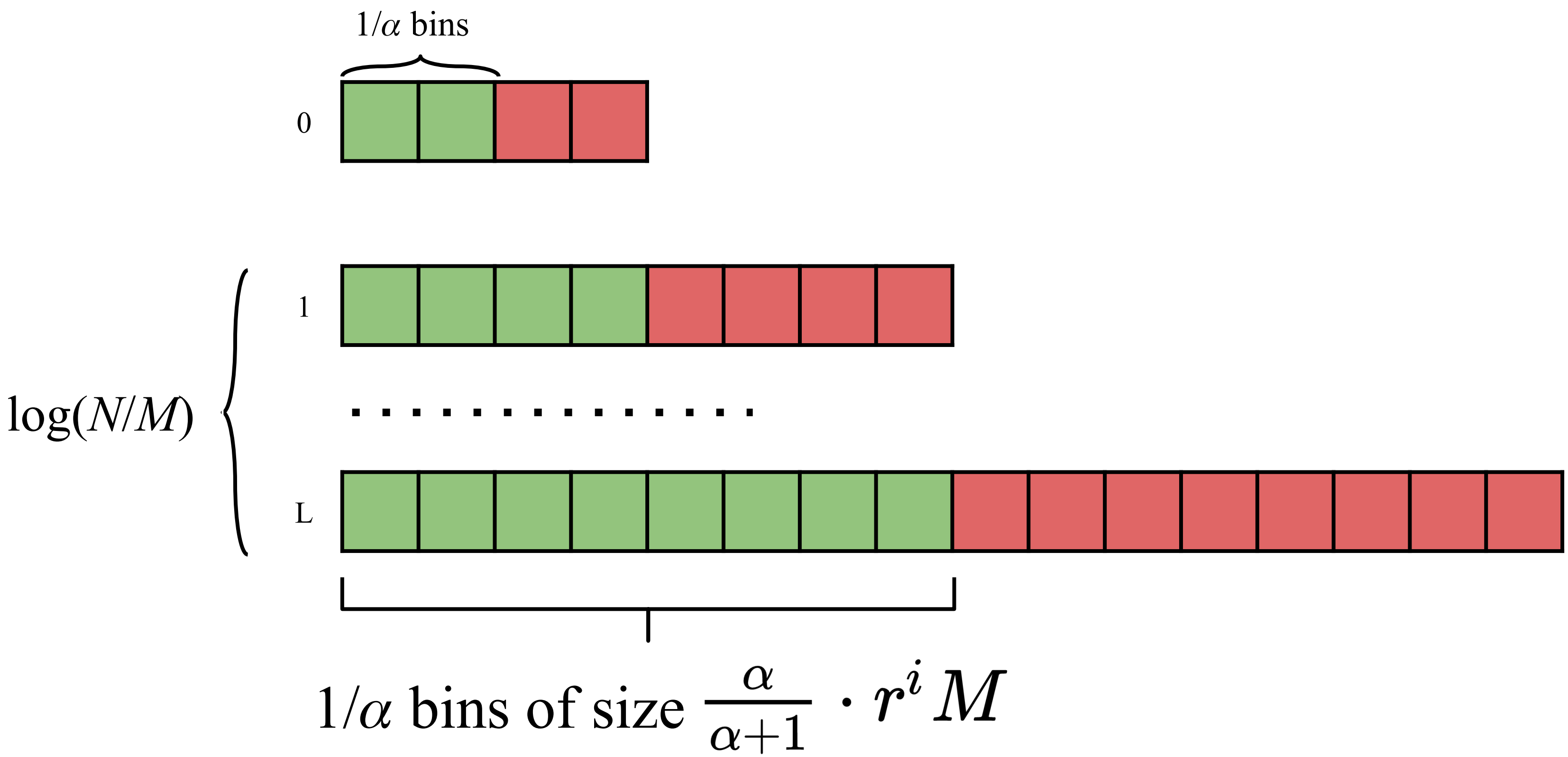
Last bin **flushed** to first bin of the next level

Time-stretch LERT



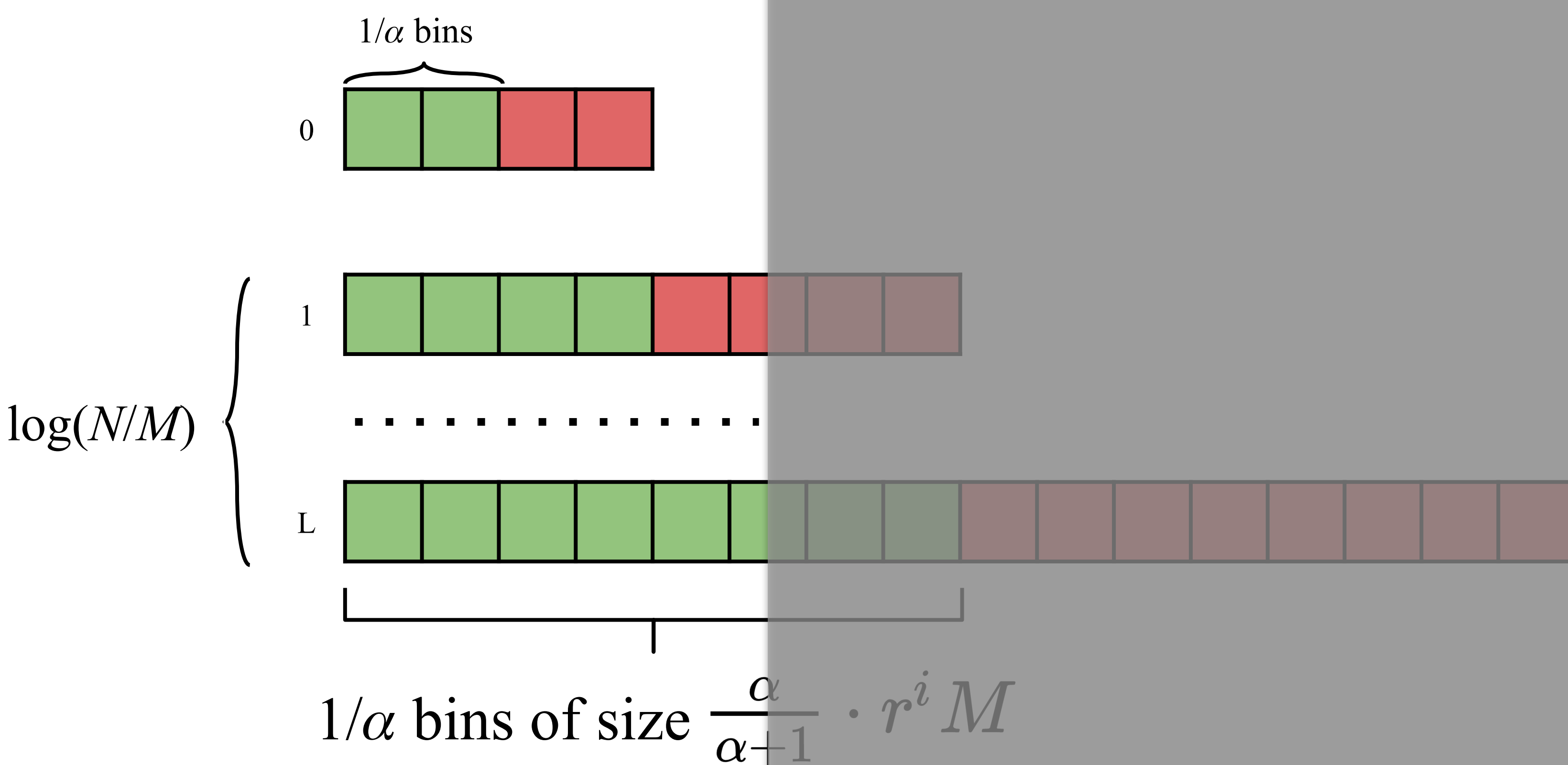
Last bin **flushed** to first bin of the next level

Time-stretch LERT correctness

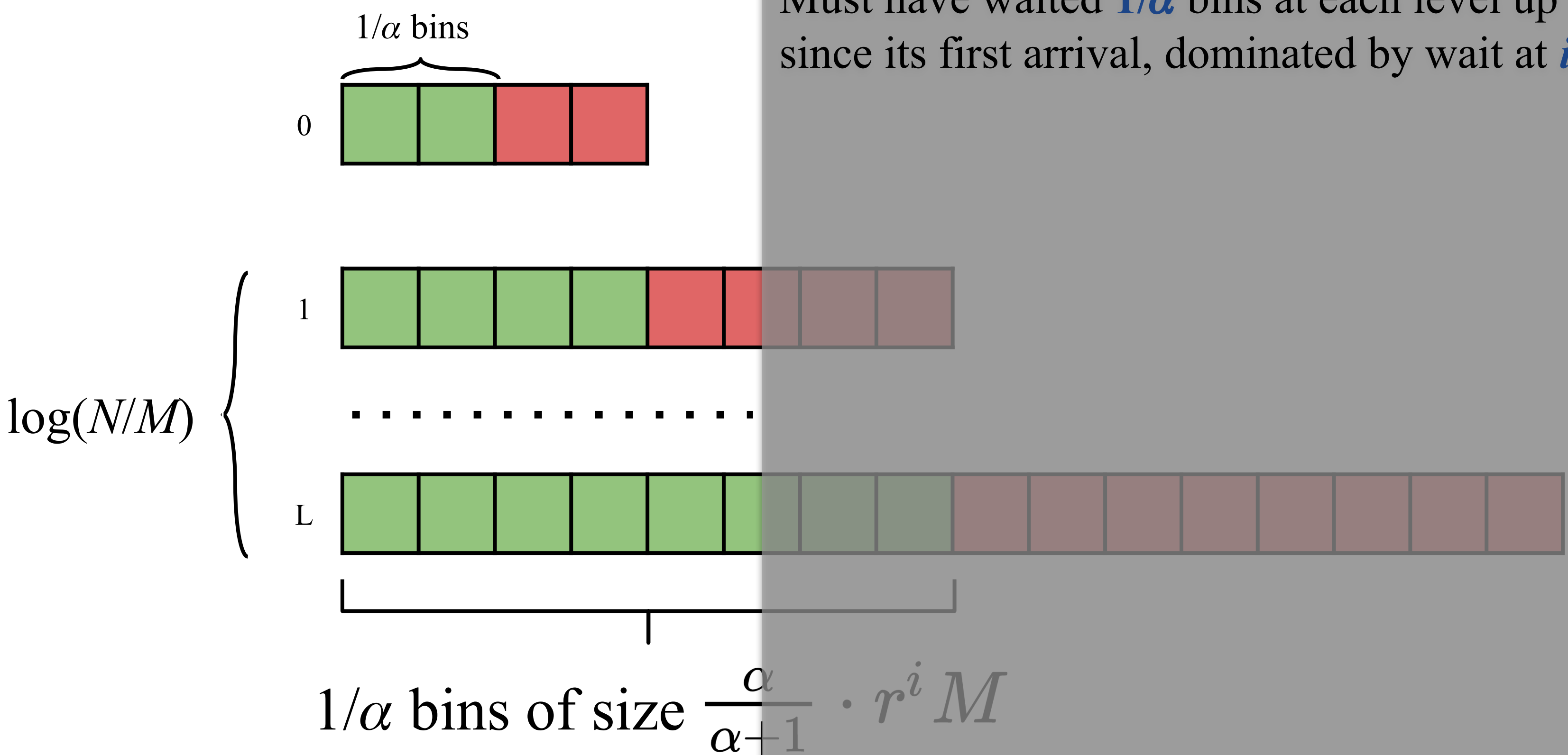


Time-stretch LERT correctness

Let ***i* + 1** be the lowest level a key is at when it hits the threshold count



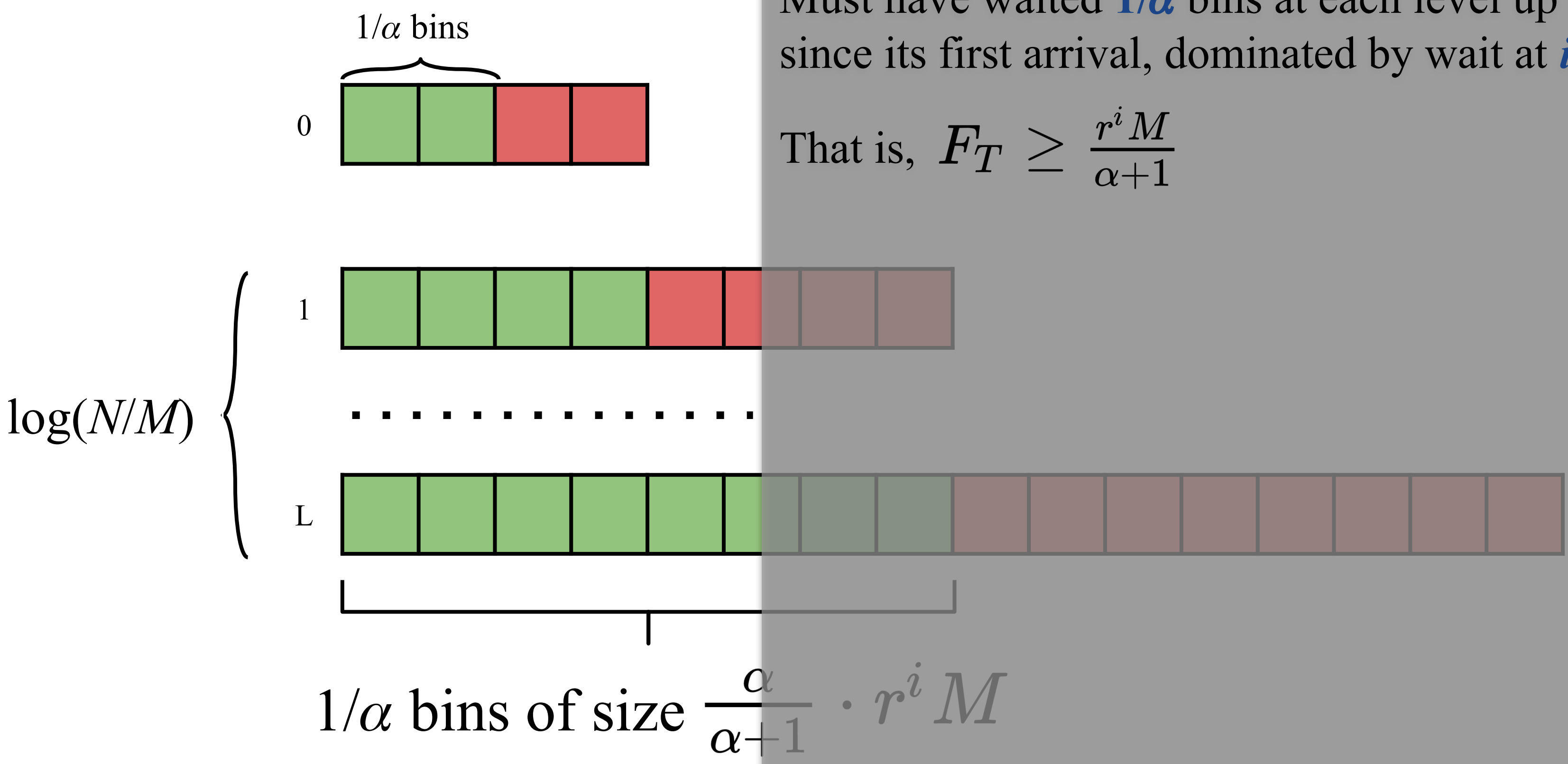
Time-stretch LERT correctness



Let $i + 1$ be the lowest level a key is at when it hits the threshold count

Must have waited $1/\alpha$ bins at each level up to i since its first arrival, dominated by wait at i

Time-stretch LERT correctness

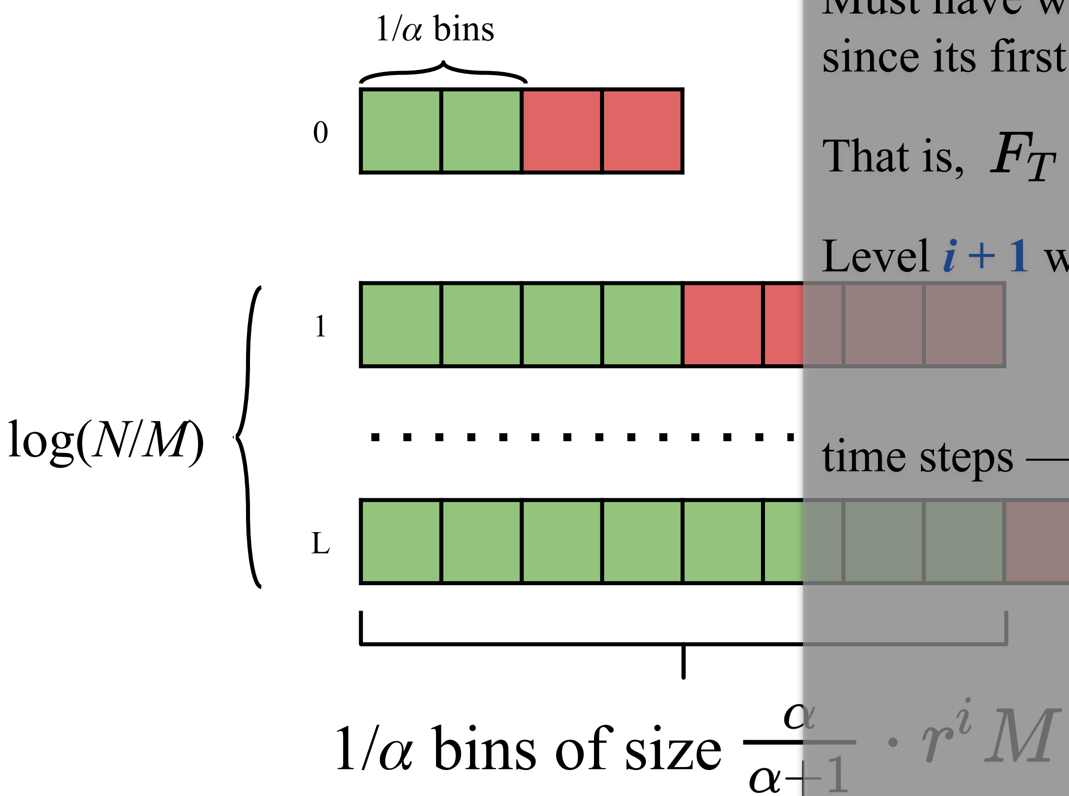


Let $i + 1$ be the lowest level a key is at when it hits the threshold count

Must have waited $1/\alpha$ bins at each level up to i since its first arrival, dominated by wait at i

That is, $F_T \geq \frac{r^i M}{\alpha+1}$

Time-stretch LERT correctness



Let $i + 1$ be the lowest level a key is at when it hits the threshold count

Must have waited $1/\alpha$ bins at each level up to i since its first arrival, dominated by wait at i

That is, $F_T \geq \frac{r^i M}{\alpha+1}$

Level $i + 1$ will participate in a flush again in

$$\frac{\alpha r^i M}{\alpha+1} \leq \alpha F_T$$

time steps — key will be reported

Time-stretch LERT I/O complexity

$$O\left(\left(\frac{\alpha+1}{\alpha}\right)\boxed{\frac{1}{B}\log\frac{N}{M}}\right)$$

Optimal insert cost for Write-optimized data structure

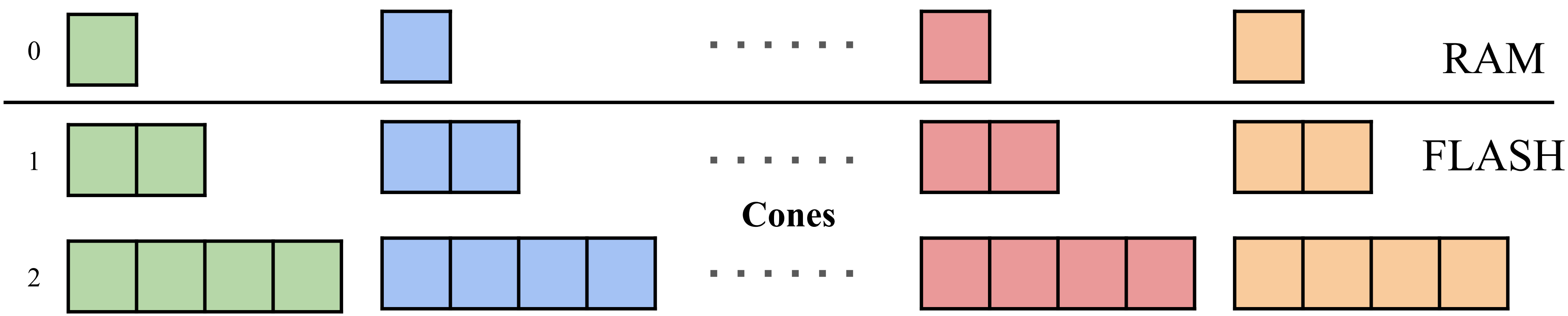
Time-stretch LERT I/O complexity

$$O\left(\left(\frac{\alpha+1}{\alpha}\right)\frac{1}{B}\log\frac{N}{M}\right)$$

Extra cost because we only move one bin during a flush. Constant loss for constant α

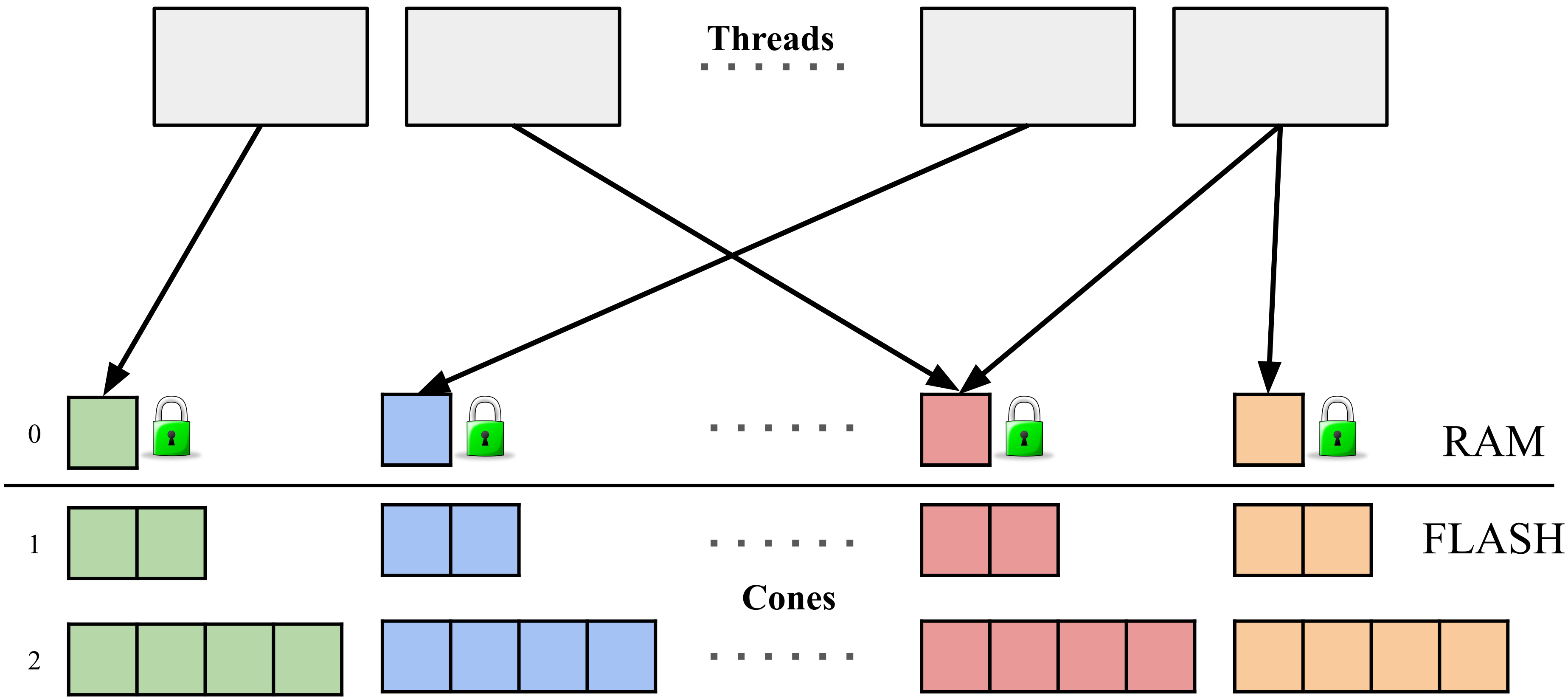
Optimal insert cost for Write-optimized data structure

Supporting high ingestion throughput



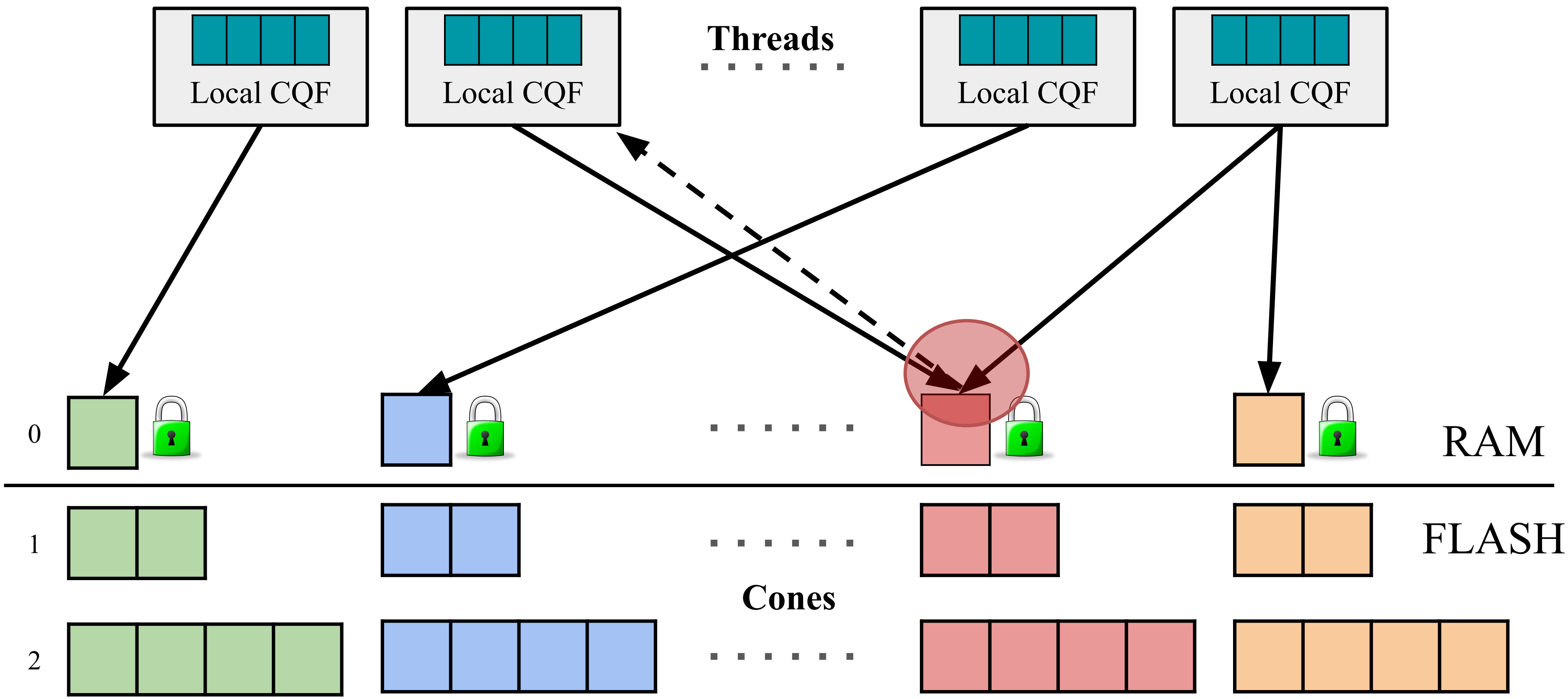
Divide into multiple smaller LERTs called *cones*, each with the same number of levels and growth factor.

Supporting high ingestion throughput



Use uniform-random hashing to route items to cones.
Each thread first acquires a lock on the cone and then performs insertion.

Avoiding contention for skewed distributions

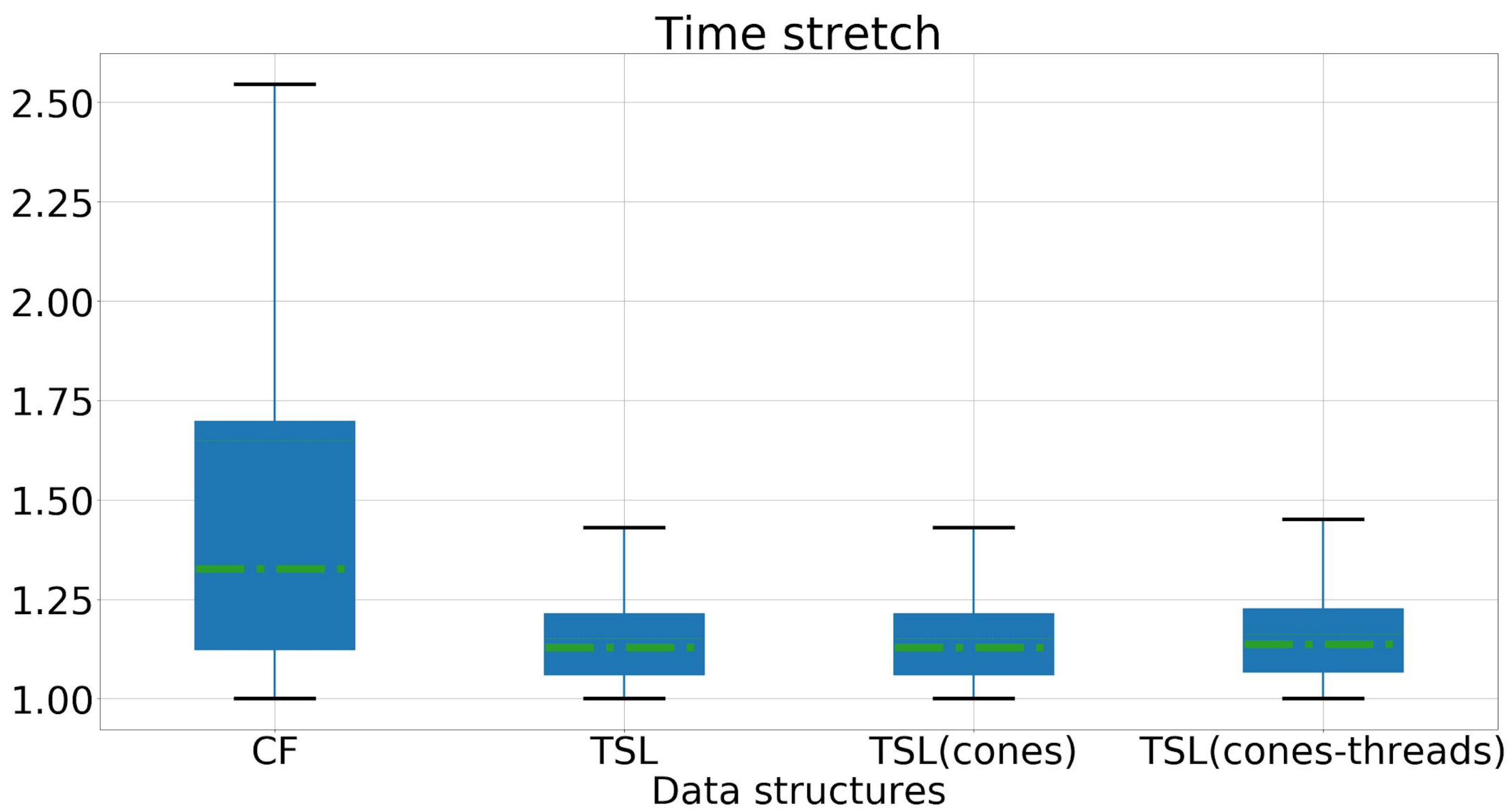


If there is contention, threads make progress by inserting items in the local buffer.
Local buffer is flushed at regular intervals.

Evaluation

- Empirical timeliness
- Insertion throughput
- Effect of cones/threads on instantaneous throughput
- Scalability with threads

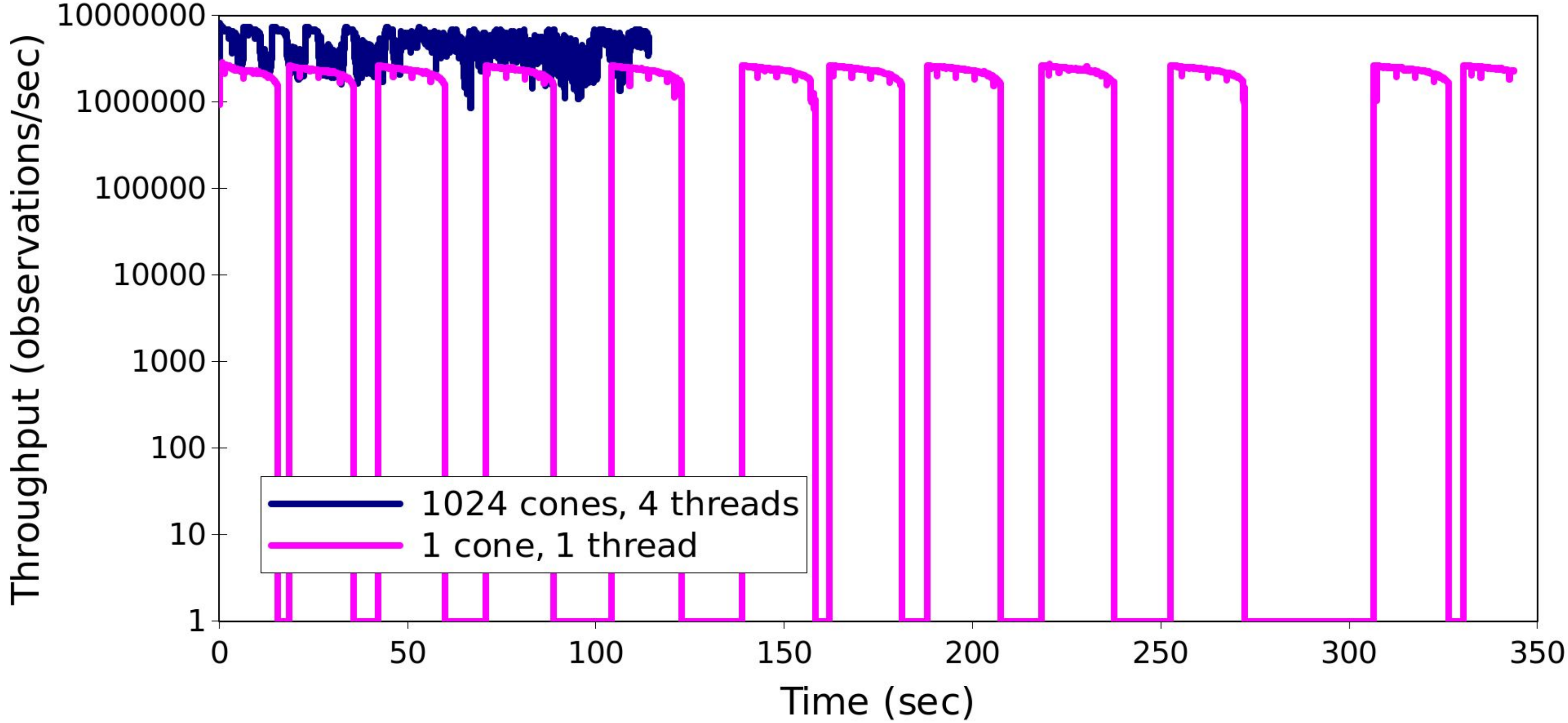
Evaluation: empirical time stretch



Average time stretch is 43% smaller than theoretical upper bound.

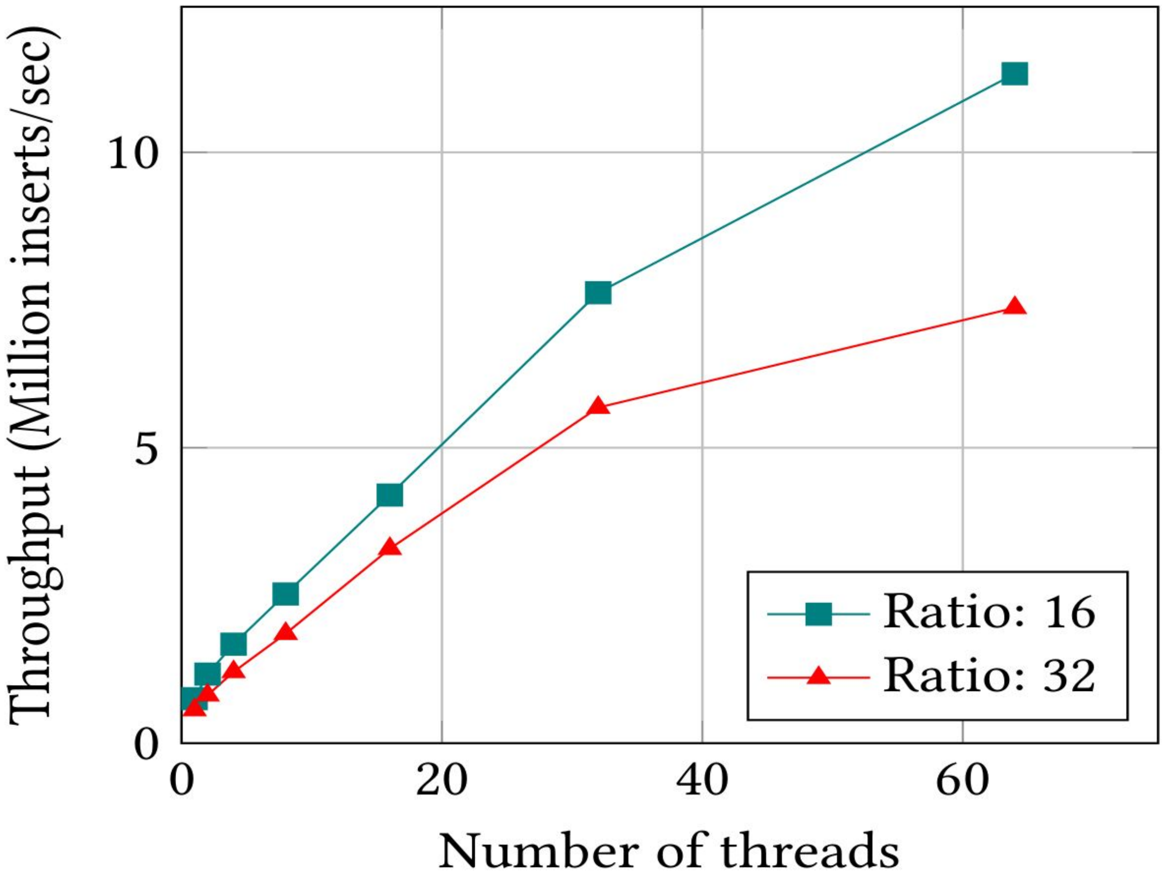
Multithreading has negligible effect on the empirical time stretch.

Evaluation: instantaneous throughput



Multithreading achieves smoother throughput with any jitters.
Cones and multithreading improve both instantaneous throughput and average throughput.

Evaluation: scalability



The insertion throughput increases as we add more threads.

We can achieve > 11M insertions/sec.

LERT: supports scalable and real-time reporting

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput ingestion ✓

- Events are high-consequence real-life events

No false-negatives; few false-positives ✓

Timely reporting (real-time) ✓

- Very small reporting threshold $T << N$ (stream size)

Very small reporting thresholds ✓

Conclusion

- This work bridges the gap between streaming & external memory.
- We can solve timely event detection problem at a level of precision that is not possible in the streaming model.
- What other streaming problems can be solved in external memory at comparable speed?
- What is the right model for streaming in modern external memory?

Acknowledgements

Kingsford Group

Carl Kingsford
Guillaume Marcais
Natalie Sauerwald
Cong Ma
Laura Tung
Hongyu Zheng
Yihang Shen
Yutong Qiu
Minh Hoang
Mohsen Ferdosi

Funding

Sandia National Laboratories

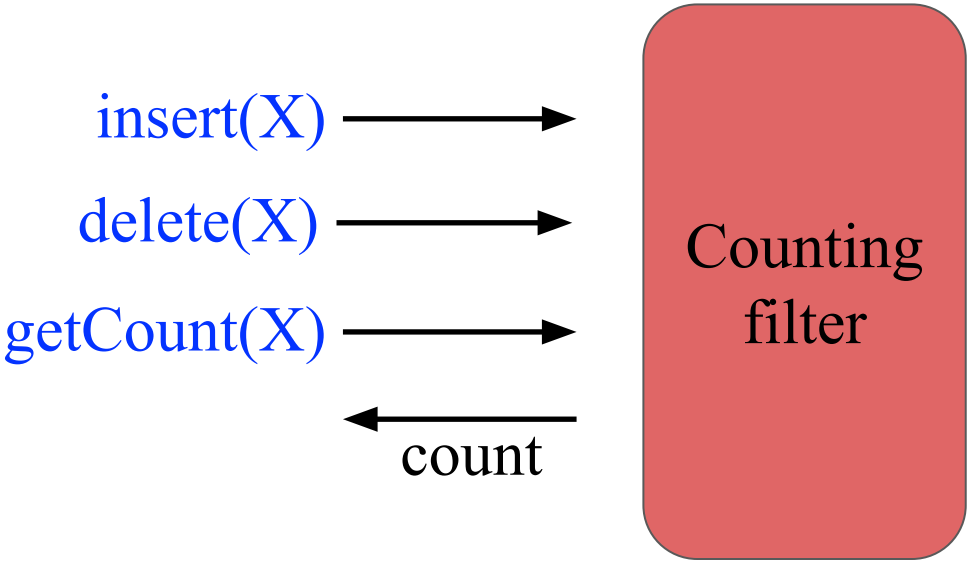
GORDON AND BETTY MOORE FOUNDATION

NIH National Institutes of Health

The Shurl and Kay Curci Foundation

https://prashantpandey.github.io

Counting filters



- **A counting filter is a lossy representation of a multiset**
- **Operations: insert, count, and delete**
- **False-positive errors ≈ Over counts**

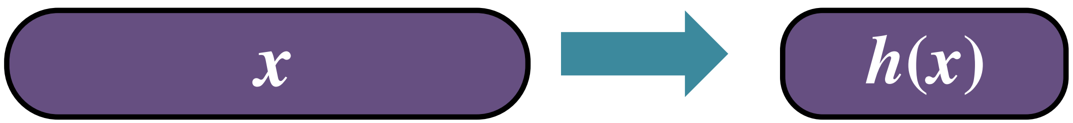
The quotient filter (QF)

- Maintains count estimates
- Space and computationally efficient
- Can be used as a map for small key-value pairs
- Uses variable-sized encoding for counts
  - Asymptotically optimal space: $O(\sum |C(x)|)$

QF uses Quotienting

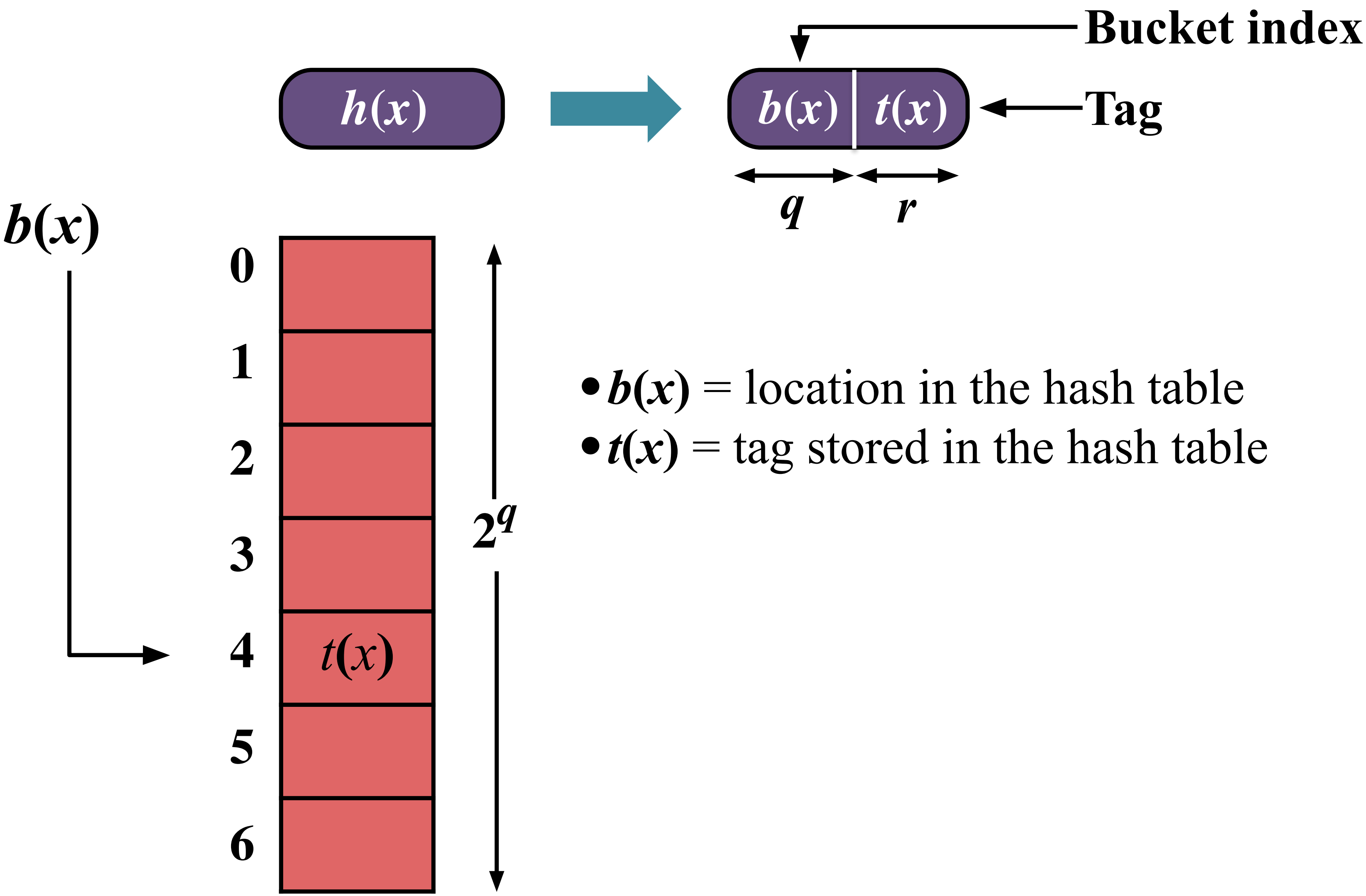
Knuth. Searching and Sorting Vol. 3, ‘97

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



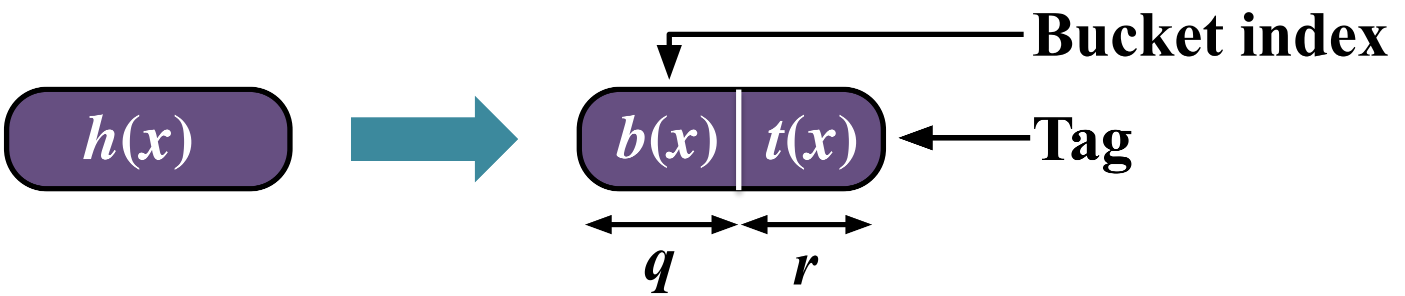
- **Only source of false positives:**
  - Two distinct elements x and y, where $h(x) = h(y)$
  - If x is stored and y isn’t, query(y) gives a false positives

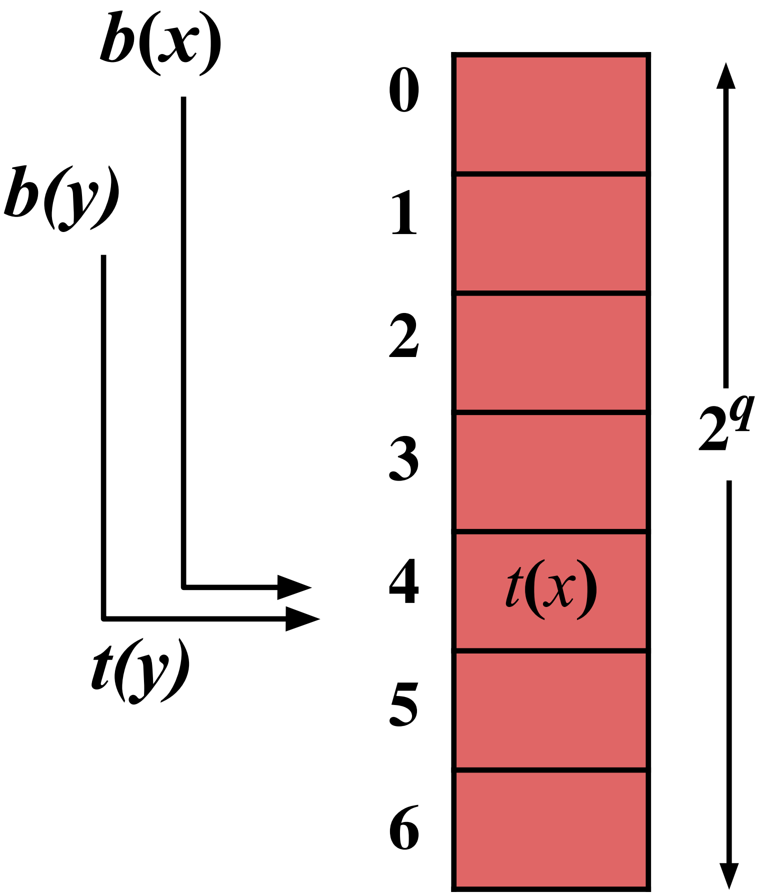
Storing fingerprints compactly



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

Storing fingerprints compactly

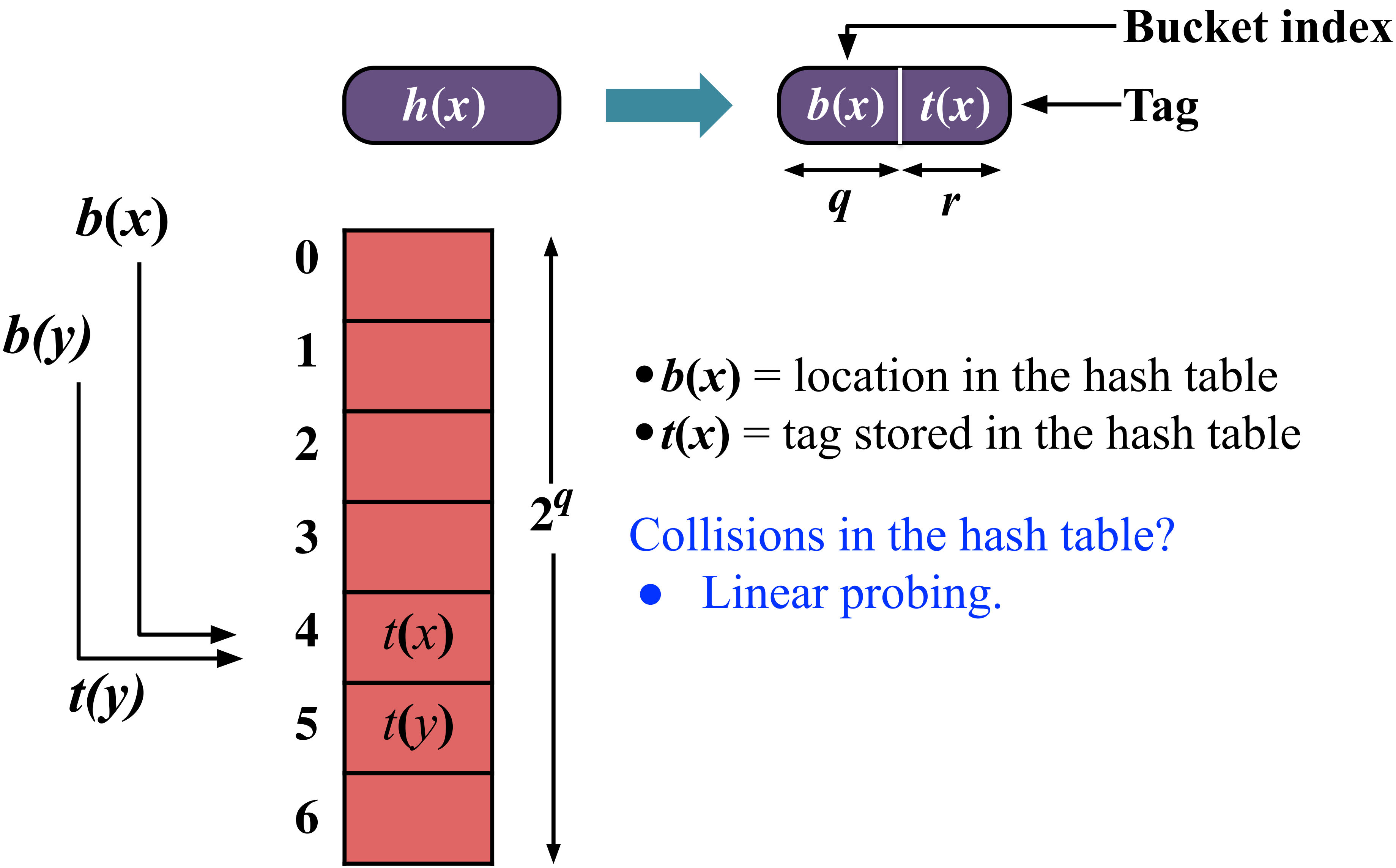




- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

Collisions in the hash table?

Storing fingerprints compactly

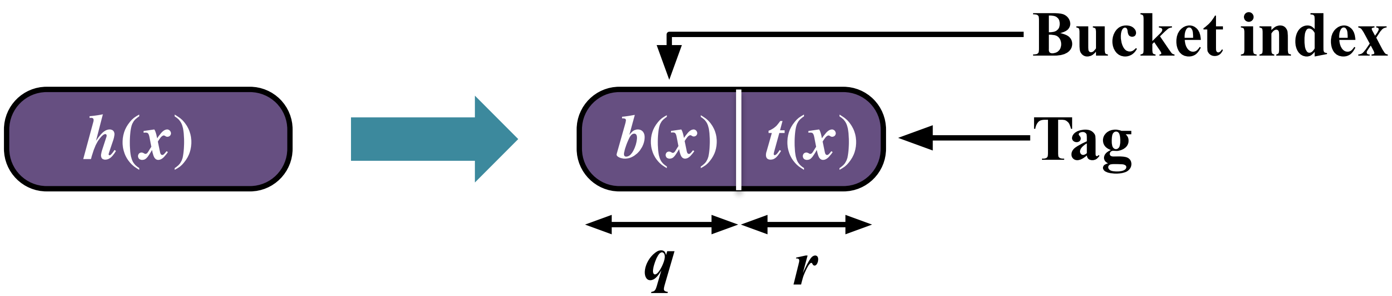


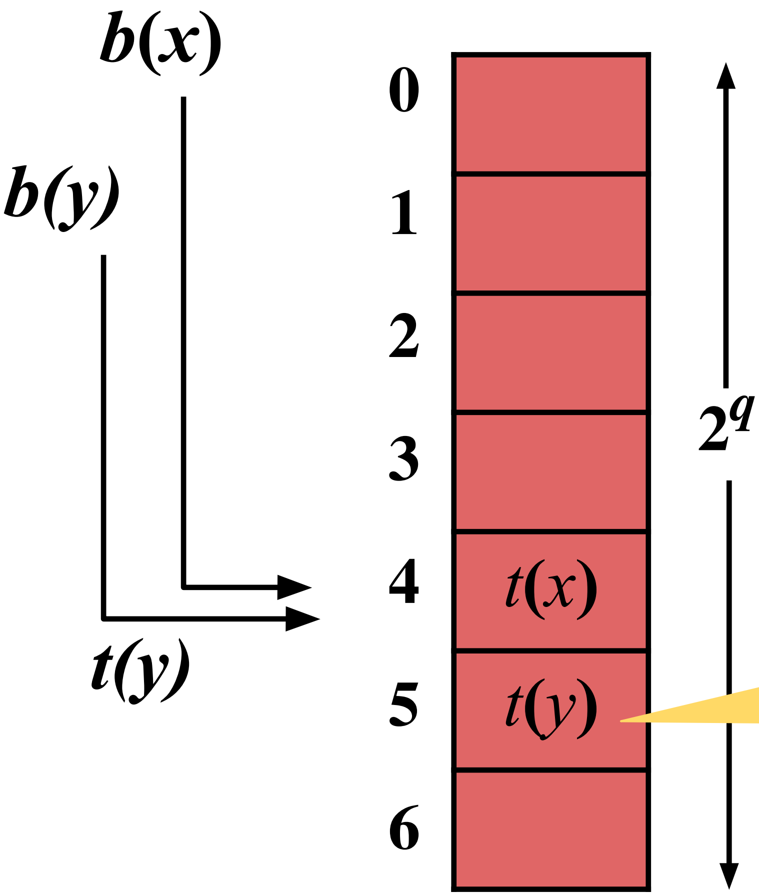
- $b(x)$ = location in the hash table
- $t(x)$ = tag stored in the hash table

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?

- Li

$t(y)$ belongs to slots 4 or 5?

Resolving collisions in the QF

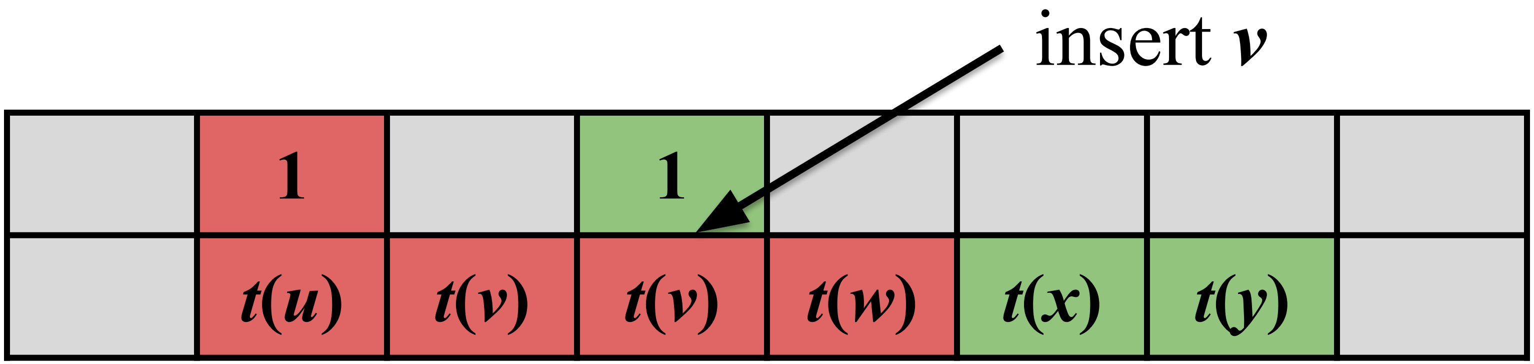
- QF uses two metadata bits to resolve collisions and identify home bucket

	1		1				
	$t(u)$	$t(v)$	$t(w)$	$t(x)$	$t(y)$		

- The metadata bits group tags by their home bucket

Resolving collisions in the QF

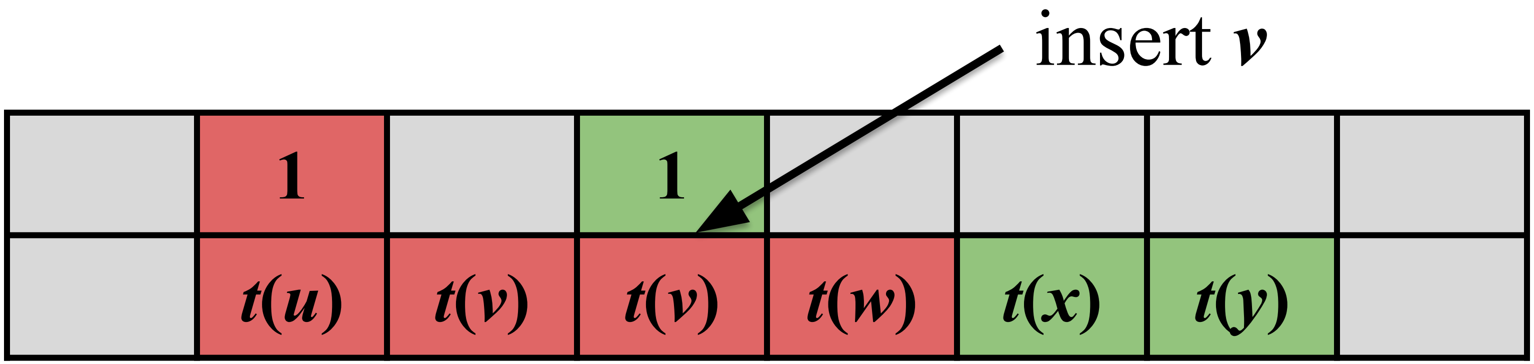
- QF uses two metadata bits to resolve collisions and identify home bucket



- The metadata bits group tags by their home bucket

Resolving collisions in the QF

- QF uses two metadata bits to resolve collisions and identify home bucket



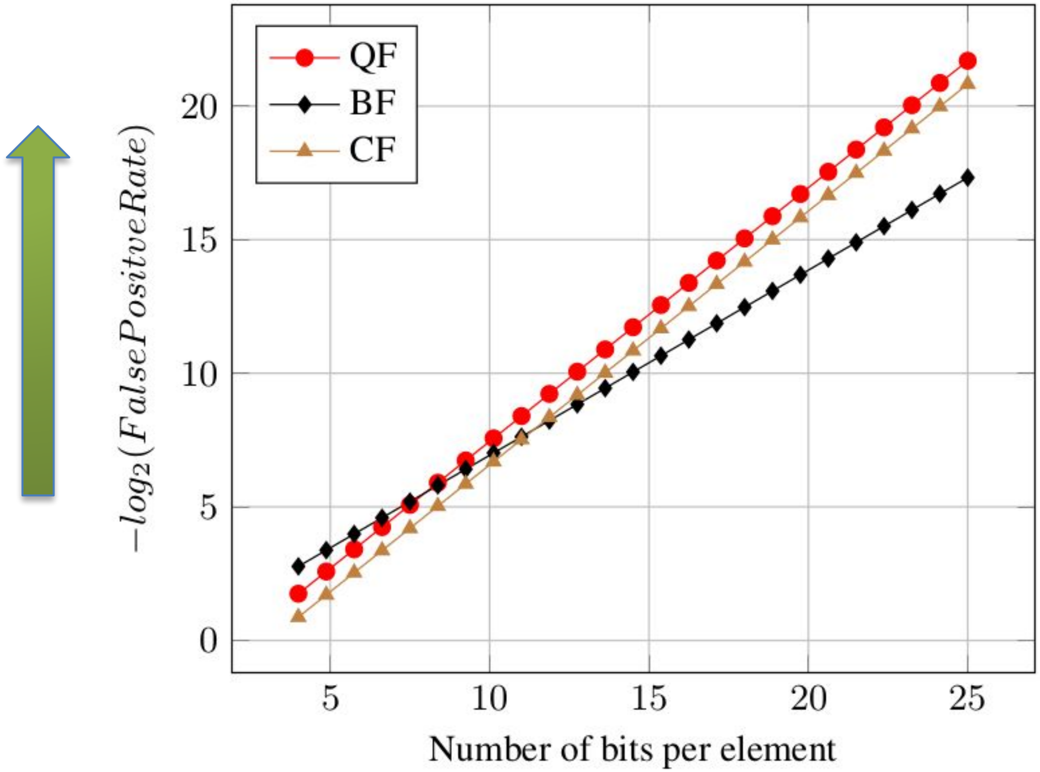
- The metadata bits group tags by their home bucket

The metadata bits enable us to identify the slots holding the contents of each bucket.

Quotienting enables many features in the QF

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

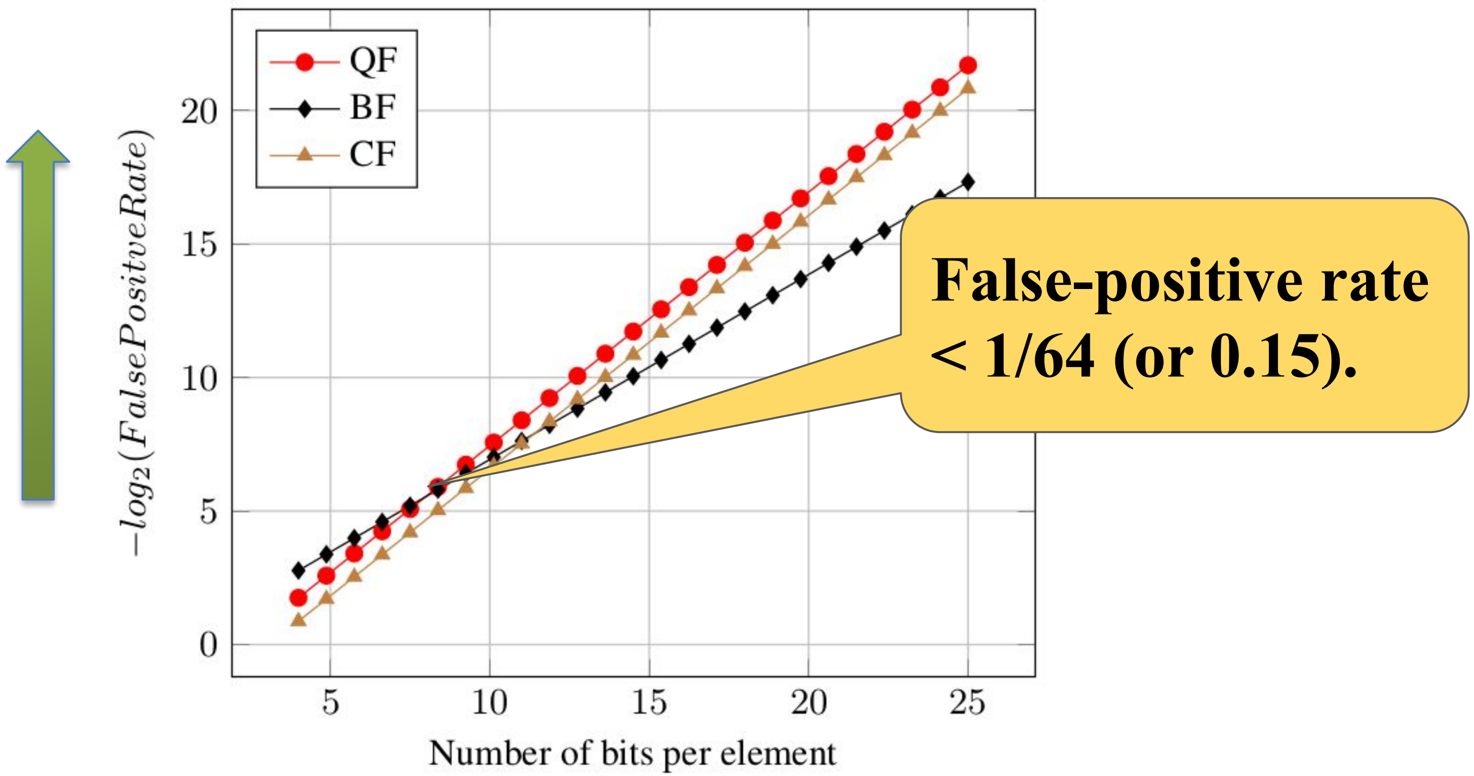
Quotient filters use less space than Bloom filters for all practical configurations



Bloom filter: ~*1.44log$_2$(1/ε)* bits/element.

Quotient filter: ~*2.125 + log$_2$(1/ε)* bits/element.

Quotient filters use less space than Bloom filters for all practical configurations

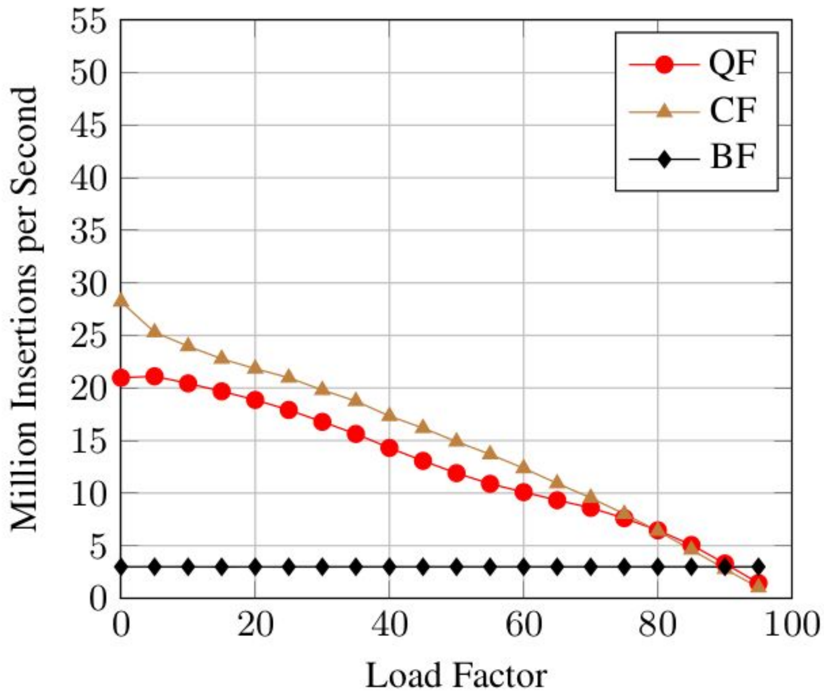


Bloom filter: ~*1.44log$_2$(1/ε)* bits/element.

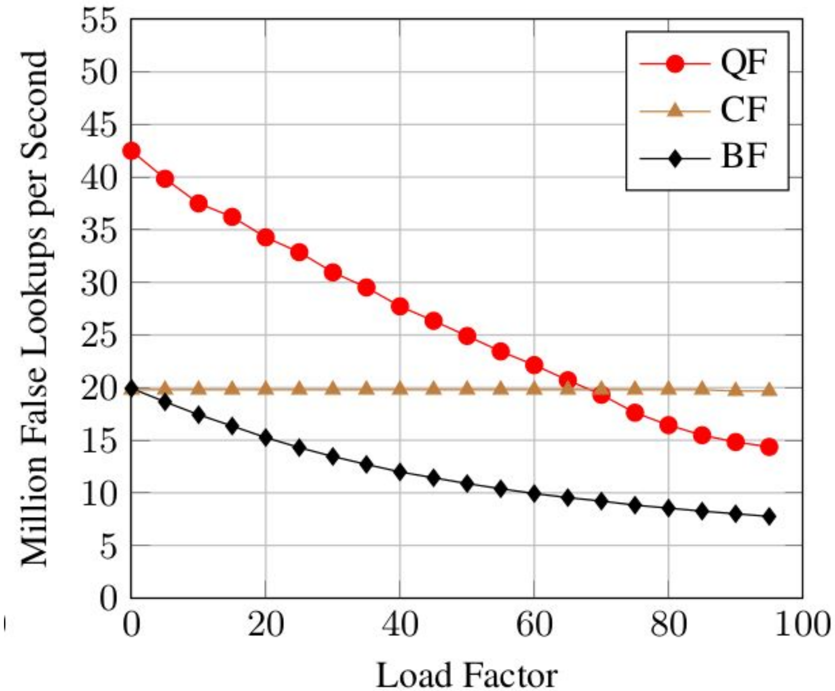
Quotient filter: ~*2.125 + log$_2$(1/ε)* bits/element.

Quotient filters perform better (or similar) to other non-counting filters

Inserts



Lookups



- Insert performance is similar to the state-of-the-art non-counting filters
- Query performance is significantly fast at low load-factors and slightly slower at higher load-factors

Cascade filter doesn't have real-time reporting

- Stream is large (in terabytes) and high-speed (millions/sec)

High throughput ingestion

- Events are high-consequence real-life events

No false-negatives; few false-positives

Timely reporting (real-time)

- Very small reporting threshold $T << N$ (stream size)

Very small reporting thresholds