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/](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 $(s_1, s_2, \ldots, s_t)$

Suppose $T = 4$
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.

Suppose $T = 4$

Event!  
Report
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)
- Events are high-consequence real-life events
  - High throughput ingestion
  - 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)
- Events are high-consequence real-life events
- Very small reporting thresholds
- Very small reporting thresholds
- High throughput ingestion
- No false-negatives; few false-positives
- Timely reporting (real-time)
One-pass streaming has errors

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

- **Approximate solution**: report all items with count \( \geq \varphi N \), 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**: \( \Omega(1/\varepsilon) \)

Real time with false-positives!

Maintain count estimates in RAM

Misra & Gries ‘82
One-pass streaming has errors

- **Approximate solution**: report all items with count $\geq \varphi N$, 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, $\varphi N$ is a small constant (24), So $\Omega(1/\varepsilon)$ is very very large!!

Can’t solve in RAM for very small $\varphi$

Real time with false-positives!

Maintain count estimates in RAM

Misra & Gries ‘82
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

Scales to very small φ but offline!
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) [X]
- 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

Combine streaming and EM algorithms to solve real-time event detection problem
- **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 $M$, and data size $N$. 

---

**External memory model**  
Aggarwal+Vitter ‘08

---

![Diagram showing RAM and DISK with block sizes $B$ and $M$]
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

Efficient merging

Quotient filter

log(N/M)}

RAM

FLASH

M

0

1

L

N

Mr^L

Mr^1
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 $i-1$
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

\[ \log(\frac{N}{M}) \]

Quotient filter

Efficient merging

When RAM is full, items are flushed to the smallest level on disk \(i\) with space to insert items in level 0 to \(i-1\)
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
A query operation requires a lookup in each non-empty level
Cascade filter operations

<table>
<thead>
<tr>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
</tbody>
</table>
## Cascade filter operations

<table>
<thead>
<tr>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
</tbody>
</table>

$< 1$ I/O per observation
Cascade filter operations

<table>
<thead>
<tr>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
</tbody>
</table>

- **< 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!

<table>
<thead>
<tr>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
</tbody>
</table>

- < 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!

<table>
<thead>
<tr>
<th>Insert</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O \left( \frac{1}{B} \log \frac{N}{M} \right)$</td>
<td>$O \left( \log \frac{N}{M} \right)$</td>
</tr>
</tbody>
</table>

Traditional cascade filter doesn’t solve the problem! But we can use insights.

Observation 1
Observation 2
This talk: Leveled External-Memory Reporting Table (LERT)

- Given a stream of size $N$ and $\varphi N > \Omega(N/M)$ the amortized cost of solving real-time event detection is
  \[
  O \left( \left( \frac{1}{B} + \frac{1}{(\varphi-1/M)N} \right) \log \frac{N}{M} \right)
  \]

- For a constant time stretch in reporting, can support arbitrarily small thresholds $\varphi$ 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 $\phi N > \Omega(N/M)$ the amortized cost of solving real-time event detection is

$$\Omega\left(\left(\log \frac{N}{M} \right)^2\right)$$

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

arbitrarily small thresholds $\phi$ 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!
Given a stream of size $N$ and $\varphi N > \Omega(N/M)$ the amortized cost of solving real-time event detection is

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

For a constant time stretch in reporting, can support arbitrarily small thresholds $\varphi$ with amortized cost

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

$$\alpha = \frac{R_T}{F_T} - 1$$
For a \textbf{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 $a$.

\[ \alpha = \frac{R_T}{F_T} - 1 \]
Divide each level into $1 + 1/\alpha$, equal-sized bins.
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

\[ \log\left(\frac{N}{M}\right) \]

Last bin **flushed** to first bin of the next level
While flushing consolidate counts; report if hits threshold

Quotient filter

Last bin **flushed** to first bin of the next level
Main idea: item is not put on a deeper level until it’s "aged sufficiently"

Last bin **flushed** to first bin of the next level
Time-stretch LERT correctness

\[
\frac{1}{\alpha} \text{ bins of size } \frac{\alpha}{\alpha+1} \cdot r^i M
\]
Time-stretch LERT correctness

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

$log(N/M)$

1/α bins of size $\frac{\alpha}{\alpha - 1} \cdot r^i M$
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,

$$F_T \geq \frac{r^i M}{\alpha + 1}$$
Let $i + 1$ be the lowest level a key is at when it hits the threshold count.

Must have waited $\frac{1}{\alpha}$ 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

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

Optimal insert cost for Write-optimized data structure
Extra cost because we only move one bin during a flush. Constant loss for constant $\alpha$.

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
Average time stretch is 43% smaller than theoretical upper bound. Multithreading has negligible effect on the empirical time stretch.
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
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

https://prashantpandey.github.io
A counting filter is a lossy representation of a multiset

- Operations: insert, count, and delete
- False-positive errors \(\approx\) 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

- **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

- $b(x) = \text{location in the hash table}$
- $t(x) = \text{tag stored in the hash table}$
Storing fingerprints compactly

- $b(x) = \text{location in the hash table}$
- $t(x) = \text{tag stored in the hash table}$

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

- $b(x) = \text{location in the hash table}$
- $t(x) = \text{tag stored in the hash table}$

Collisions in the hash table?
- Linear probing.

$t(y)$ belongs to slots 4 or 5?
Resolving collisions in the QF

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

- 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

<table>
<thead>
<tr>
<th></th>
<th>t(u)</th>
<th>t(v)</th>
<th>t(v)</th>
<th>t(w)</th>
<th>t(x)</th>
<th>t(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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: $\sim 1.44 \log_2 (1/\varepsilon)$ bits/element.

Quotient filter: $\sim 2.125 + \log_2 (1/\varepsilon)$ bits/element.
Quotient filters use less space than Bloom filters for all practical configurations.

Bloom filter: $\sim 1.44 \log_2 (1/\varepsilon)$ bits/element.
Quotient filter: $\sim 2.125 + \log_2 (1/\varepsilon)$ bits/element.
Quotient filters perform better (or similar) to other non-counting filters

- 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**

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