SAND2020-3526C

Timely Reporting of Heavy Hitters using External Memory

Prashant Pandey[*]
ppandey2@cs.cmu.edu
Carnegie Mellon University

Shikha Singh[*]
shikha.singh@wellesley.edu
Wellesley College

Michael A. Bender
bender@cs.stonybrook.edu
Stony Brook University

Jonathan W. Berry
jberry@sandia.gov
Sandia National Laboratories

Martín Farach-Colton
farach@cs.rutgers.edu
Rutgers University

Rob Johnson
robj@vmware.com
VMware Research

Thomas M. Kroeger
tmkroeg@sandia.gov
Sandia National Laboratories

Cynthia A. Phillips
caphill@sandia.gov
Sandia National Laboratories

ABSTRACT

Given an input stream of size N, a ***ϕ-heavy hitter*** is an item that occurs at least ϕN times in S. The problem of finding heavy-hitters is extensively studied in the database literature.

We study a real-time heavy-hitters variant in which an element must be reported shortly after we see its $T = \phi N$-th occurrence (and hence becomes a heavy hitter). We call this the Timely Event Detection (TED) Problem. The TED problem models the needs of many real-world monitoring systems, which demand accurate (i.e., no false negatives) and timely reporting of all events from large, high-speed streams, and with a low reporting threshold (high sensitivity).

Like the classic heavy-hitters problem, solving the TED problem without false-positives requires large space ($\Omega(N)$ words). Thus in-RAM heavy-hitters algorithms typically sacrifice accuracy (i.e., allow false positives), sensitivity, or timeliness (i.e., use multiple passes).

We show how to adapt heavy-hitters algorithms to external memory to solve the TED problem on large high-speed streams while guaranteeing accuracy, sensitivity, and timeliness. Our data structures are limited only by I/O-bandwidth (not latency) and support a tunable trade-off between reporting delay and I/O overhead. With a small bounded reporting delay, our algorithms incur only a logarithmic I/O overhead.

We implement and validate our data structures empirically using the Firehose streaming benchmark. Multi-threaded versions of our structures can scale to process 11M observations per second before becoming CPU bound. In comparison, a naive adaptation of the standard heavy-hitters algorithm to external memory would be limited by the storage device's random I/O throughput, i.e., ≈ 100K observations per second.

[*]Both authors contributed equally to this research.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; **Streaming, sublinear and near linear time algorithms**.

KEYWORDS

Dictionary data structure; streaming algorithms; external-memory algorithms

ACM Reference Format:
Prashant Pandey, Shikha Singh, Michael A. Bender, Jonathan W. Berry, Martín Farach-Colton, Rob Johnson, Thomas M. Kroeger, and Cynthia A. Phillips. 2020. Timely Reporting of Heavy Hitters using External Memory. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3380598

1 INTRODUCTION

Real-time monitoring of high-rate data streams, with the goal of detecting and preventing malicious events, is a critical component of defense systems for cybersecurity [46, 56, 58] as well as for physical systems, e.g., for water or power distribution [15, 44, 47]. In such a monitoring system, the stream elements represent the changes to the state of the system. Each detected/reported event triggers an intervention. Analysts use more specialized tools to gauge the actual threat level. Newer systems are even beginning to take defensive actions, such as blocking a remote host, automatically based on detected events [38, 50]. Accuracy (i.e., few false-positives and no false-negatives) and timeliness of event detection are essential to these systems.

Central to these applications is the problem of timely reporting of ***heavy-hitters***. In the heavy-hitters problem, we are given a stream $S = (s_1, \ldots, s_N)$ and a reporting threshold $T = \phi N$, and we need to report all elements that occur more than T times in S. In the real-time version, we must report each heavy hitter soon after its T-th occurrence, where the acceptable reporting delay is defined by the application. We call the problem of reporting heavy-hitters with bounded delay the Timely Event Detection (TED) problem.

In network-security monitoring applications, N is huge (effectively infinite) and T can be very small, even a constant. This is because anomalies in network streams are often small-sized events that develop slowly, appearing normal in the midst of large amounts of legitimate traffic [48, 57]. As an example of the demands placed on event-detection systems, the US Department of Defense (DoD) and Sandia National Laboratories developed the Firehose streaming benchmark suite [1, 5] to measure the performance of TED algorithms. In the FireHose benchmark, the reporting threshold is preset to the representative value of $T = 24$, i.e., $\phi = 24/N = o(1)$.

The classic streaming algorithms for reporting heavy-hitters were designed assuming that only an in-RAM data structure can keep up with high-speed streams. The challenge of detecting events entirely within RAM has inspired a deep and beautiful literature on streaming algorithms and database systems [4, 14, 17, 18, 20, 21, 28, 33–36, 45, 49].

However, streaming algorithms sacrifice accuracy in order to get solutions that can fit in RAM. First, most streaming heavy-hitter algorithms only work for high reporting thresholds, e.g., T is a constant fraction of N. Second, they let through false positives. Third, many streaming algorithms perform some kind of sampling, and these also let through false negatives. These inaccuracies are not the fault of the streaming algorithms. They are an inherent limitation when you have a large stream and a much smaller RAM size.

This paper. This paper challenges the assumption that only in-RAM data structures can keep up with real-world streams. We show that, by using modern storage devices and building upon recent advances in external-memory dictionaries, we can design on-disk data structures that can process millions of stream events per second.

We prove our results in the external-memory model [3]. In the external-memory model, RAM has fixed size M, and accessing it is free. The disk has unbounded size and accessing it costs an I/O. An I/O transfers data between RAM and disk in blocks of size B. The algorithmic advantage of external memory is that there is unbounded storage. The algorithmic challenge is that I/Os are expensive.

External-memory enables us to overcome longstanding limitations in accuracy (i.e. no false-positives or negatives) and sensitivity (i.e. small ϕ) while maintaining timeliness in event reporting, but necessitates developing new heavy-hitters algorithms that use I/Os efficiently.

Our algorithms support both exact and approximate reporting of heavy hitters. Specifically, our TED algorithms can be generalized to solve the ***(ε, ϕ)-approximate heavy hitters*** problem—where every item that occurs $\geq \phi N$ times must be reported, no item that occurs $\leq (\phi - \varepsilon)N$ times should be reported. Items with count in between $(\phi - \varepsilon)N$ and ϕN may be reported and these are false positives.

Timeliness, not ingestion, is the challenge in external memory. Stream ingestion is *not* the bottleneck for on-disk data structures. Optimal external-memory (EM) dictionaries (including write-optimized dictionaries such as B^ε-trees [9, 11, 24], COLAs [10], xDicts [23], buffered repository trees [25], write-optimized skip lists [13], log structured merge trees [54], and optimal external-memory hash tables [30, 41]) can ingest new observations at a significant fraction of disk bandwidth. The fastest can index using $O\left(\frac{1}{B} \log \frac{N}{M}\right)$ I/Os per stream item, which is far less than one I/O per item. In practice, this means that even a system with just a single disk can ingest hundreds of thousands to millions of items per second.

For example, prior work at SuperComputing 2017 showed that a single computer can easily maintain an on-disk B^ε-tree [24] index of all connections on a 600 gigabit/sec network [8]. The system could efficiently answer offline queries. What the system could not do was detect events online.

Existing external-memory data structures do not solve the TED problem because queries are too slow. For example, consider a straw-man solution in which we use an external-memory dictionary to implement the standard heavy-hitters algorithm, Misra Gries [52]. Since Misra-Gries performs a query for each stream observation, this approach is bottlenecked on the dictionary searches. Once the dictionary is larger than RAM, for a random stream, most queries will miss the cache and require an I/O, and hence will be bottlenecked on the latency of the storage device.

In this paper, we show how to perform timely event detection for essentially the same cost as simply inserting the data into a B^ε-tree or other optimal external-memory dictionary. Even so, we manage to answer the standing heavy-hitter query for each new stream element.

1.1 Results

In this paper, we present external-memory algorithms for the TED problem. We evaluate these algorithms theoretically and empirically. In both cases, we show that these algorithms perform much less than one I/O per query and are limited only by I/O bandwidth (not latency). Furthermore, we show how to provide a tradeoff between reporting delay and I/O cost. We call these data structures ***leveled external-memory reporting tables (LERTs)***.

We begin by presenting the ***Misra-Gries LERT***, which adapts the Misra-Gries heavy-hitter algorithm to solve the TED problem in external-memory with ***immediate reporting***. In particular, the Misra-Gries LERT reports each ϕ-event as soon as it occurs (no delay) at an amortized cost of $O((1/B)\log(N/B))$ I/Os, for sufficiently large ϕ. The guarantees of the Misra-Gries LERT hold for any input distribution; see Corollary 1.

The Misra-Gries LERT serves as the basis of our main algorithms that support much smaller ϕ, but permit some delay in reporting. We define two types of delay: time stretch and count stretch. We say an event-detection algorithm has ***time stretch*** $1+\alpha$ if each item s is reported at most αL_s time steps after s's Tth occurrence, where L_s is the number of time steps between s's first and Tth occurrences. We say that an event-detection algorithm has ***count stretch*** $1+\omega$, if each item is reported before the item's count reaches $(1+\omega)T$.

We design a data structure, the ***time-stretch LERT***, that solves the TED problem for any input stream and any $\phi > 1/M$ and with time stretch $1+\alpha$ at an amortized cost of $O\left(\frac{\alpha+1}{\alpha}\frac{1}{B}\log\frac{N}{M}\right)$ I/Os per stream item. For constant α, this is asymptotically as fast as simply ingesting and indexing the data [10, 24, 25]. The time-stretch LERT guarantees hold for any input distribution; see Corollary 2.

In our evaluations, the time-stretch LERT with stretch of 2 can ingest at $\approx$ 500K insertions/sec using a single thread. We also observed that the average empirical time stretch is 43% smaller than the theoretical upper bound.

We design the ***count-stretch LERT*** which is tailored to guarantee count-stretch on input distributions, where the count of items is drawn from a power-law distribution. We say the item counts in the stream follow a power-law distribution with exponent θ if the probability that an item has count c is proportional to $c^{-(\theta-1)}$.

Given an input stream with item counts distributed according to a power-law with parameter θ, where θ is in the typical range of 2 to 2.96 [2, 14, 22, 29, 53], and parameters T and ω, such that $\omega T > 2.5(N/M)^{\frac{1}{\theta-1}}$, we show that the ***count-stretch LERT*** solves the TED problem with count stretch $1+\omega$ at an amortized I/O cost $O\left(\frac{1}{B}\log\frac{N}{M}\right)$ per stream item with high probability (w.h.p.). Thus, the count-stretch LERT avoids expensive point queries, matching the ingestion rate of write-optimized data structures. In our evaluations, we find that the count-stretch LERT with stretch 1.583 can ingest at $\approx 1M$ insertions/sec using a single thread. With multi-threading and de-amortization, the count-stretch LERT scales to more than 11M insertions/sec, and the variance of the instantaneous throughput goes down by several orders of magnitude relative to the amortized, single-threaded version; see Figure 5c. Moreover, the average empirical count stretch is 21% smaller than the theoretical upper bound.

Finally, we show how to modify the count-stretch LERT to support immediate reporting. We call the resulting data structure the ***immediate-report LERT*** and show that it solves the TED problem much faster than the Misra-Gries LERT for input streams drawn from power-law distributions; see Theorem 8 for the formal I/O cost. In our evaluation, we find that the immediate-report LERT can ingest at $\approx$ 500K insertions/sec using a single thread.

Additional Related Work

Heavy-hitter algorithms. The heavy-hitter problem has been extensively studied in the database literature; we refer readers to the survey by Cormode and Hadjieleftheriou [31].

There are two main strategies that have been used: deterministic counter-based approaches [18, 35, 43, 49, 51, 52] and randomized sketch-based ones [28, 32]. The first is based on the classic Misra and Gries (MG) algorithm [52], which generalizes the Boyer-Moore majority finding algorithm [19].

Randomized sketch-based algorithms such as count-min sketch [32] maintain a small sketch of the frequency vectors using compact hash functions.

Database iceberg queries. The TED problem is related to the problem of answering ***iceberg queries*** in databases [16, 37, 39, 40]. An iceberg query computes an aggregate function over some database attribute and reports the values that are above some predetermined threshold. The main distinctions between the two problems is: (a) iceberg queries are offline, i.e., performed on a static dataset, and (b) the number of reported results in iceberg queries is usually small; while the number of reported events can be large in the TED.

Database continuous queries. The TED problem is also closely related to answering ***continuous*** or ***standing queries*** over a database [6, 7, 27]. A continuous query, once issued, runs as the database is updated through inserts and deletes. The system reports new query matches as the database is updated. In TED, the database D consists of the items from the stream seen so far, and the continuous query over D is whether there is an item with count exactly $\lceil \phi N \rceil$.

2 PRELIMINARIES

We review several building blocks of our data structures: the Misra-Gries heavy-hitters algorithm [52], counting quotient filters (CQF) [55], and cascade filters (CF) [10].

The Misra-Gries frequency estimator. The Misra-Gries (MG) algorithm estimates the frequency of items in a stream. Given an estimation error ε and a stream S of N items from a universe $\mathcal{U}$, the MG algorithm uses a single pass over S to construct a table C with at most $\lceil 1/\varepsilon \rceil$ entries. Each table entry is an item $s \in \mathcal{U}$ with a count, denoted $C[s]$. For each $s \in \mathcal{U}$ not in table C, let $C[s] = 0$. Let f_s be the number of occurrences of item s in stream S. The MG algorithm guarantees that $C[s] \leq f_s < C[s] + \varepsilon N$ for all $s \in \mathcal{U}$.

MG initializes C to an empty table and processes items in the stream as described below. For each s_i in S,

- If $s_i \in C$, increment counter $C[s_i]$.
- If $s_i \notin C$ and $|C| < \lceil 1/\varepsilon \rceil$, insert s_i into C. Set $C[s_i] \leftarrow 1$.
- If $s_i \notin C$ and $|C| = \lceil 1/\varepsilon \rceil$, then for each $x \in C$ decrement $C[x]$ and delete its entry if $C[x]$ becomes 0.

We argue that $C[s] \leq f_s < C[s] + \varepsilon N$. We have $C[s] \leq f_s$ because $C[s]$ is incremented only for an occurrence of s in the stream. We can bound how many counts are lost through decrements: in a single decrement step caused by an item s_i, $\lceil 1/\varepsilon \rceil + 1$ counts are decremented at once: one for each item in the table and an instance of s_i. There can be at most $\lfloor N/\lceil 1/\varepsilon + 1 \rceil \rfloor < \varepsilon N$ such steps. Thus, $f_s < C[s] + \varepsilon N$.

The MG algorithm can be used to solve the ***(ε, ϕ)-heavy hitters problem*** as follows. Run the MG algorithm on the stream with error parameter ε. Then iterate over the set C and report any item s with $C[s] > (\phi - \varepsilon)N$.

Analogous to the (ε, ϕ)-heavy hitters problem, we define the ***approximate TED problem*** as: report all ϕ-events soon after they occur, do not report any item with count $\leq (\phi - \varepsilon)N$. Reported items with count in between are false positives.

Counting Quotient Filter. The counting quotient filter (CQF) [55] can be viewed as a hash table based on Robin-Hood hashing [26]. The CQF consists of an array Q of 2^q ***slots*** and a hash function h mapping stream elements to p-bit integers, where $p \geq q$. Robin-Hood hashing is a variant of linear probing in which we try to place an element a in slot $h(a)/2^{p-q}$, but shift elements down when there are collisions. Furthermore, Robin-Hood hashing maintains the invariant that, if $h(a) < h(a')$, then a will be in an earlier slot than a'.

The CQF supports efficient insertions, queries, updates, and deletions, just like any Robin-Hood hash table. Thus, it is straightforward to implement the Misra-Gries algorithm on top of a CQF, by using the CQF to store the table C.

Cascade Filter. The cascade filter (CF) is a write-optimized data structure based on the CQF [55] and the COLA [10]. The CF consists of multiple levels with exponentially increasing sizes where each level is a CQF. The first level Q_0 is in RAM and the rest are on SSD. There are $L = \log_r(N/M) + O(1)$ levels, where M is the size of RAM, N is the size of the dataset, and r is the factor by which levels grow in size.

Since the cascade filter is also a map, we can use it as the basis for an EM Misra-Gries algorithm. The total table size is $N = \Theta(1/\varepsilon)$. The amortized I/O cost to update the table for each stream element is $O\big(\frac{1}{B}\log_r\big(\frac{1}{\varepsilon M}\big)\big)$. However, if we want to support immediate reporting in a CF, then a query is triggered after each insert which costs $O(\log_r(1/\varepsilon M))$ I/Os. Thus the overall algorithm is bottlenecked on the queries performed for each stream element.

3 IMMEDIATE REPORTING

In this section, we first design an efficient external-memory version of the core Misra-Gries frequency estimator and then extend our external-memory Misra-Gries algorithm to solve the TED problem with immediate reporting.

When $\varepsilon = o(1/M)$, then simply running the standard Misra-Gries algorithm can result in a cache miss for every stream element, incurring an amortized cost of $\Omega(1)$ I/Os per element. Our construction reduces this to $O\big(\frac{1}{B}\log\big(\frac{1}{\varepsilon M}\big)\big)$, which is $o(1)$ when $B = \omega\big(\log\big(\frac{1}{\varepsilon M}\big)\big)$.

External-memory Misra-Gries. Our external-memory Misra-Gries data structure is a sequence of Misra-Gries tables, $C_0, \ldots, C_{L-1}$, where $L = 1 + \lceil \log_r(1/(\varepsilon M)) \rceil$ and r (> 1) is a parameter we set later. The size of the table C_i at level i is $r^i M$, so the size of the last level is at least $1/\varepsilon$.

Each level acts as a Misra-Gries data structure. Level 0 receives its input from the stream. Level $i > 0$ receives its input from level $i-1$, the level above. Whenever the standard Misra-Gries algorithm for the table C_i at level i would decrement an item count, the external-memory MG data structure decrements that item's count by one on level i and sends one instance of that item to the level below $(i+1)$. The decrements from the last level L are deleted.

We call the process of decrementing the counts of all the items at level i and incrementing all the corresponding item counts at level $i+1$ a ***flush***.

Lemma 1 shows that every prefix of levels $C_0, \ldots, C_j$ in the external-memory MG data structure is a MG frequency estimator, with the accuracy of the estimates increasing with j.

Lemma 1. *Let $\widehat{C_j}[x] = \sum_{i=0}^{j} C_i[x]$ (where $C_i[x] = 0$ if $x \notin C_i$). Then, the following holds:*

- $\widehat{C_j}[x] \leq f_x < \widehat{C_j}[x] + (N/(r^j M))$*, and,*
- $\widehat{C_{L-1}}[x] \leq f_x < \widehat{C_{L-1}}[x] + \varepsilon N$.

Thus, to report (ε, ϕ)-heavy hitters (at the end of the stream), we can iterate over the sets C_i and report any element x with counter $\widehat{C_{L-1}}[x] > (\phi - \varepsilon)N$.

For the analysis, we assume that each level of the external-memory MG structure is implemented as a B-tree.

Lemma 2. *Given $\varepsilon \geq 1/N$, the amortized I/O cost of insertion in the external-memory MG data structure is $O(\frac{1}{B}\log\frac{1}{\varepsilon M})$.*

When no false positives are allowed, that is, $\varepsilon = 1/N$, the I/O complexity is $O(\frac{1}{B}\log\frac{N}{M})$.

Misra-Gries LERT. We extend our external-memory MG data structure to support immediate reporting. In particular, we show that for a threshold ϕ that is sufficiently large, we can report ϕ-events as soon as they occur.

A first attempt to add immediate reporting is to compute $\widehat{C}_{L-1}[s_i]$ for each stream event s_i and report s_i as soon as $\widehat{C}_{L-1}[s_i] > (\phi - \varepsilon)N$. However, this requires querying C_i for $i = 0, \ldots, L-1$ for every stream item and can cost up to $O(\log(1/\varepsilon M))$ I/Os per stream item.

We avoid these expensive queries by using the properties of the in-memory MG estimates C_0. If $C_0[s_i] \leq (\phi - 1/M)N$, then we know that $f_{s_i} \leq \phi N$ and we therefore do not have to report s_i, regardless of the count for s_i in the lower levels of the external-memory data structure.

We describe the new data-structure, the ***Misra-Gries LERT***. Whenever we increment $C_0[s_i]$ from a value that is at most $(\phi - 1/M)N$ to a value that is greater than $(\phi - 1/M)N$, we compute $\widehat{C}_{L-1}[s_i]$ and report s_i if $\widehat{C}_{L-1}[s_i] = \lceil(\phi - \varepsilon)N\rceil$. For each entry $C_0[x]$, we store a bit indicating whether we have performed a query for $\widehat{C}_{L-1}[x]$. As in our external-memory MG structure, if the count for an entry $C_0[x]$ becomes 0, we delete that entry. This means we might query for the same item more than once; as we see below, this has no effect on the overall I/O cost of the algorithm.[1]

In order to avoid reporting the same item more than once, we can maintain, with each entry in C_i, a bit indicating whether that item has already been reported.

For the analysis, we assume that the levels of the data structure are implemented as sorted arrays with fractional cascading, and thus computing $\widehat{C}_{L-1}[x]$ requires $O(L)$ I/Os.

Theorem 3. *Given a stream of size N and parameters ε and ϕ, where $\varepsilon \in [1/N, \phi)$ and $\phi \in (1/M, 1)$, the approximate TED problem can be solved with immediate reporting at amortized I/O cost $O\big(\big(\frac{1}{B} + \frac{M}{(\phi M - 1)N}\big) \log \frac{1}{\varepsilon M}\big)$ per stream item.*

Proof. The amortized cost of performing insertions is $O((1/B)\log(1/\varepsilon M))$. To analyze the query costs, let $\varepsilon_0 = 1/M$, i.e., the frequency error of the in-memory level. Since we perform at most one query each time an item's count in C_0 goes from 0 to $(\phi - \varepsilon_0)N$, the total number of queries is at most $N/((\phi - \varepsilon_0)N) = 1/(\phi - \varepsilon_0) = M/(\phi M - 1)$. Since each query costs $O(\log(1/\varepsilon M))$ I/Os, the overall amortized I/O complexity of the queries is $O\big(\big(\frac{M}{(\phi M - 1)N}\big) \log \frac{1}{\varepsilon M}\big)$. □

To solve the problem exactly, that is, with no false positives we set $\varepsilon = 1/N$ in Theorem 3, and get the following corollary.

Corollary 1. *Given a stream of size N and $\phi \in (1/M, 1)$, the TED problem can be solved with immediate reporting at amortized I/O cost $O\big(\big(\frac{1}{B} + \frac{M}{(\phi M - 1)N}\big) \log \frac{N}{M}\big)$ per stream item.*

[1]It is possible to prevent repeated queries for an item but we allow it as it does not hurt the asymptotic performance.

Summary. The Misra-Gries LERT supports a throughput at least as fast as optimal write-optimized dictionaries [10, 11, 13, 23–25], while estimating the counts as well as if it had an enormous RAM. It maintains count estimates at different granularities across the levels. Not all estimates are actually needed, but given a small number of levels, we can refine the estimates by looking in only a few additional locations.

4 TIME STRETCH

The MG LERT described in Section 3 reports events immediately, albeit at a higher amortized I/O cost for each stream item. In this section, we show that if we allow a bounded reporting delay proportional to the time it takes an item to become a ϕ-event, we can significantly improve the I/O performance—in particular, we can perform timely event detection asymptotically as cheaply as if we reported all events only at the end of the stream.

In particular, our data structure guarantees a time-stretch of $1 + \alpha$, that is, it reports an item x no later than time $t_1 + (1 + \alpha)F_t = t_2 + \alpha F_t$, where t_1 is the time of the first occurrence of x, t_2 is the time of the ϕNth occurence of x and $F_t = t_2 - t_1$ is the ***flow time*** of x.

Time-stretch LERT. We design a data structure to guarantee time-stretch, the ***time-stretch LERT***. Similar to the Misra-Gries LERT, the time-stretch LERT consists of $L = \log_r(1/(\varepsilon M))$ levels $C_0, \ldots, C_{L-1}$. The ith level has size $r^i M$. Items are flushed from lower to higher levels.

Unlike the Misra-Gries LERT, all events are detected during the flush operations. Thus, we never need to perform point queries. This means: (1) we can use simple sorted arrays to represent each level and, (2) we don't need to maintain the invariant that level 0 is a MG data structure on its own.

Layout and flushing schedule. We split the table at each level i into $c = (\alpha + 1)/\alpha$ equal-sized ***bins*** $b_1^i, \ldots, b_c^i$, each of size $\frac{\alpha}{\alpha+1}(r^i M)$. The capacity of a bin is defined by the sum of the counts of the items in that bin, i.e., a bin at level i can become full because it contains $\frac{\alpha}{\alpha+1}(r^i M)$ items, each with count 1, or 1 item with count $\frac{\alpha}{\alpha+1}(r^i M)$.

We maintain a strict flushing schedule to obtain the time-stretch guarantee. The flushes are performed at the granularity of bins (rather than entire levels). Each stream item is inserted into b_1^0. Whenever a bin b_1^i becomes full (i.e., the sum of the counts of the items in the bin is equal to its size), we shift all the bins on level i over by one, and we move all the items in b_c^i into bin b_1^{i+1}. Since the bins in level $i + 1$ are r times larger than the bins in level i, bin b_1^{i+1} becomes full after exactly r flushes from b_c^i. When this happens, we perform a flush on level $i + 1$ and so on.

Finally, during a flush involving levels $0, \ldots, i$, where $i \leq L - 1$, we scan these levels and for each item k, we sum its counts. If the total count is greater than $(\phi - \varepsilon)N$, and (we have not reported it before[2]) then we report k.

We show that our data structure guarantees time stretch.

Lemma 4. *The time-stretch LERT reports each ϕ-event s_t occurring at time t by $t + \alpha F_t$, where F_t is the flow time of s_t.*

Proof. Consider an item s_t with flow time F_t. Let ℓ be the largest level containing an instance of s_t at time t when it hits the threshold count of ϕN. The flushing schedule guarantees that, for each level $i < \ell$, the item s_t must have waited $1/\alpha$ bins of size $\frac{\alpha r^i M}{\alpha+1}$ on that level before being inserted to level ℓ. This is dominated by waiting time on level $\ell - 1$. That is,

$$F_t \geq \frac{1}{\alpha} \cdot \frac{\alpha r^{\ell-1} M}{\alpha + 1} = \frac{r^{\ell-1} M}{\alpha + 1}.$$

This level participates in a flush again after $\frac{\alpha r^{\ell-1} M}{\alpha+1} \leq \alpha F_t$ inserts. Thus, s_t is reported at most αF_t time steps after t. □

For the analysis, we treat each level as a sorted array.

Theorem 5. *Given a stream of size N and parameters ε, $\phi \in (1/M, 1)$ and $\alpha > 0$, where $\varepsilon \in [1/N, \phi)$, the approximate TED problem can be solved with time-stretch $1+\alpha$ at amortized I/O cost $O(\frac{\alpha+1}{\alpha}(\frac{1}{B}\log\frac{1}{\varepsilon M}))$ per stream item.*

For exact reporting (no false positives), we set $\varepsilon = 1/N$.

Corollary 2. *Given a stream of size N, $\alpha > 0$, and $\phi \in (1/M, 1)$, the TED problem can be solved with time stretch $1+\alpha$ at amortized I/O cost $O(\frac{\alpha+1}{\alpha}(\frac{1}{B}\log\frac{N}{M}))$ per stream item.*

Summary. By allowing a little delay, we can solve the timely event-detection problem at the same asymptotic cost as simply indexing our data [10, 11, 13, 23–25].

Thus, our results on TED problem with immediate reporting and with time stretch show that there is a spectrum between completely online and completely offline, and it is tunable with little I/O cost.

5 POWER-LAW DISTRIBUTIONS

Our results in Section 3 and Section 4 hold for *worst-case input streams.* In this section, we design TED algorithms tailored to perform well on practical input streams, in particular where the item-counts follow a power-law distribution. Note that the order of arrivals can still be adversarial.

The item counts in the stream follow a power-law distribution with exponent θ if the probability that an item has count c is equal to $Z \cdot c^{-\theta}$, where Z is the normalization constant.

Berinde et al. [14] consider streams where the item counts follow a Zipfian distribution. A stream follows a Zipfian distribution with exponent α if and only if it follows a power-law distribution with exponent $\theta = 1 + 1/\alpha$ [2]. They show that for Zipfian distributions with $\alpha > 1$ (power-law distributions with $\theta \leq 2$), the MG algorithm can solve the ε-approximate heavy hitter problem using only $\varepsilon^{-1/\alpha}$ words. Alternatively, on such Zipfian distributions, the MG algorithm achieves an improved error bound ε^α using $1/\varepsilon$ words. Our data structures based on the MG algorithm automatically inherit these improved bounds.

In the rest of this section, we study the *exact* (error-free) TED problem and design algorithms tailored for power-law distributions with exponent $2 < \theta < 2.96$, which is representative of power-law distributions observed in practice [53].

Preliminaries. We use the continuous power-law definition [53]: the count of an item with a power-law distribution has a probability $p(x)\,dx$ of taking a value in the interval from x to $x + dx$, where $p(x) = Z \cdot x^{-\theta}$, where $\theta > 1$ and Z is the normalization constant.[3]

5.1 Immediate-report LERT

First, we present the layout of our data structure, the ***immediate-report LERT***, and then we present its main algorithms, ***shuffle merges*** and ***immediate-reporting queries***. Finally we analyze its correctness and I/O performance.

Layout. The immediate-report LERT consists of a cascade of tables, where M is the size of the table in RAM and there are $L = \log_r(N/M)$ levels on disk, where $N = \left(\frac{\theta-1}{\theta-2}\right)U$, the size of the stream S. The size of level i is $N/(r^{L-i})$.

Each level on disk has an *explicit upper bound* on the number of instances of an item that can be stored on that level. This is different from the MG algorithm, where this upper bound is implicit: based on the level's size. In particular, each level i in the immediate-report LERT has a ***level threshold*** τ_i for $1 \leq i \leq L$, $(\tau_1 \geq \tau_2 \geq \ldots \geq \tau_L)$, indicating that the maximum count on level i can be τ_i.

Threshold invariant. We maintain the invariant that at most τ_i instances of an item can be stored on level i. Later, we show how to set τ_i's based on the power-law exponent θ.

Shuffle merge. The Misra-Gries LERT and time-stretch LERT use two different flushing strategies. Here we present a third strategy called the shuffle merge.

The level in RAM receives inputs from the stream one at a time. When attempting to insert to a level i that is at capacity, instead of flushing items to the next level, we find the smallest level $j > i$, which has enough empty space to hold all items from levels $0, 1, \ldots, i$. We aggregate the count of each item k on levels $0, \ldots, j$, resulting in a consolidated count c_k^j. If $c_k^j \geq (\phi - \varepsilon)N$, we report k. Otherwise, we pack instances of k in a bottom-up fashion on levels j to 0, while maintaining

[2]For each reported item, we set a flag that indicates it has been reported, to avoid duplicate reporting of events.

[3] In general, the power-law distribution may hold above some value $c_{\min}$. For simplicity, we let $c_{\min} = 1$—for this choice $Z = \theta - 1$ and $\theta > 1$.

the threshold invariants. In particular, we place $\min\{c_k^j, \tau_j\}$ instances of k on level j, and $\min\{c_k^j - (\sum_{\ell=y+1}^{j} \tau_y), \tau_y\}$ instances of k on level y for $0 \leq y \leq j-1$. Because items can end up in higher levels (compared to their level before), we call this operation a shuffle-merge instead of a merge.

Notice that the threshold invariant prevents us from flushing too many counts of an item down. Thus, items can get *packed* at a level and cannot be flushed down. Specifically, we say an item is ***packed at level ℓ*** if its count exceeds $\sum_{i=L}^{\ell+1} \tau_i$.

Too many packed items at a level can clog the data structure. In Lemma 6, we show that given a power-law stream with exponent θ, we can set the thresholds based on θ in a way that no level has too many packed items.

Lemma 6. *Let the count of U distinct items in the stream of size N follow a power-law distribution with exponent $\theta > 2$. Let $\tau_i = r^{\frac{1}{\theta-1}} \tau_{i+1}$ for $1 \leq i \leq L-1$ and $\tau_L = r^{\frac{1}{\theta-1}}$. The number of keys packed in level i is at most $\frac{\theta-2}{\theta-1}$ times size of level i.*

Immediate reporting. As soon as the count of an item k in RAM (level 0) reaches a threshold of $\phi N - \sum_{i=L}^{1} \tau_i$, the structure triggers an ***immediate-reporting query***, which performs a sweep of all the L levels, consolidates the count estimates of k at all levels and reports if the consolidated count reaches threshold $T = \phi N$. Reported items are remembered, so that each event gets reported exactly once.

Analysis. Next, we prove correctness of the immediate-report LERT and analyze its I/O complexity. We set $r = e$, which we minimizes the insertion cost (in Theorem 8).

First, we prove that the immediate-report LERT reports all ϕ-events as soon as they occur.

Lemma 7. *Let S be a stream of size N where the count of items follow a power-law distribution with exponent $2 < \theta < 2.96$. W.h.p. the immediate-report LERT solves the TED problem with immediate reporting on S.*

Next, we analyze the I/O complexity of the immediate-report LERT. Similar to Section 3, we assume each level is implemented as a B-tree.

Theorem 8. *Let S be a stream of size N where the count of items follow a power-law distribution with exponent $2 < \theta < 2.96$. Given ϕ such that $\phi N > 2.5(N/M)^{\frac{1}{\theta-1}} = \gamma$, the TED problem on S can be solved at an amortized I/O complexity $O\big(\big(\frac{1}{B} + \frac{1}{(\phi N-\gamma)^{\theta-1}}\big) \log \frac{N}{M}\big)$ per stream item w.h.p.*

Remark on scalability. The immediate-report LERT allows for strictly smaller thresholds ϕ compared to Corollary 1 and Corollary 2, for power-law streams with $\theta > 2 + 1/(\log_{2.5}(N/M))$. This is because $\gamma = 2.5(N/M)^{\frac{1}{\theta-1}} < \frac{N}{M}$ when $\theta > 2 + 1/(\log_{2.5}(N/M))$.

5.2 Count-stretch LERT

In this section, we show that if we eliminate expensive immediate-reporting queries from the immediate-report LERT, the data structure still supports bounded-delay reporting with a count-dependent delay. We say that an TED problem algorithm has ***count stretch*** $1+\omega$ if it reports each key by the time its count hits $(1+\omega)\phi N$. In particular, the notion of count stretch relaxes the reporting threshold, which leads to reduced random disk accesses.

The ***count-stretch LERT*** is the following modification of the immediate-report LERT: we eliminate immediate-reporting queries and report an item when its count in RAM hits ϕN. The data structure layout, thresholds and shuffle-merges are the same as in the immediate-report LERT.

A count-stretch guarantee does not imply any time-stretch guarantee. This is because the item's arrival distribution may be irregular: a sudden burst may get a item up to a count of ϕN quickly and it could take much longer to get from ϕNth occurrence to the $(1+\omega)\phi N$th occurrence.

Theorem 9. *Given a stream of size N where the item-counts follow a power-law distribution with exponent $2 < \theta < 2.96$, and parameters ϕ, ω such that $\phi N \cdot \omega > 2.5(N/M)^{\frac{1}{\theta-1}}$, the count-stretch LERT solves the TED problem with count stretch $1+\omega$ at amortized I/O cost $O(\frac{1}{B} \log \frac{N}{M})$ per stream item w.h.p.*

Remark on dynamically setting thresholds. If the power-law exponent θ is not known ahead of time, but a feasible setting of level thresholds exist, then we can dynamically update them so as to not "clog" the data structure.

Note that by Lemma 6, for $\theta < 3$, it is sufficient to ensure that the number of items packed at any level i do not exceed $\frac{\theta-2}{\theta-1} \leq \frac{1}{2}$ its size. We incrementally update the level thresholds to satisfy this condition as follows. Initially, $\tau_i = 0$ for each level i. During a shuffle merge involving the first j levels on disk, we set τ_j to the minimum value such that the number of keys packed at level $j+1$ is no more than half its size. Thus, we increment τ's monotonically from 0 to their feasible settings, without relying on the exponent θ.

Summary. With a power law distribution, we can support a much lower threshold ϕ for the TED problem. In the Misra-Gries LERT (Section 3), the upper bounds on the counts at each level are implicit. We show that for power-law distributions, we can achieve better performance by explicitly setting these bounds in the form of thresholds.

6 IMPLEMENTATION

We now describe our implementation of the immediate-reporting, count-stretch, and time-stretch LERT. We represent each level in the LERT as an exact counting quotient filter [55]. In addition to the count, we store a small value (usually a few bits) with each key, which serves different purposes in the different structures. In the time-stretch LERT, the value bits track the age of items. In the count-stretch LERT, these bits mark whether an item has its absolute count at a level (its aggregate count across all the levels).

Time-stretch LERT. Recall that in the time-stretch LERT (Section 4), we split each level into $c = (\alpha + 1)/\alpha$ equal-sized bins. In our implementation, instead of actually splitting levels into physical bins we assign a value (i.e., age of the item) of size $\log c$ bits to each item which determines its bin. The age of the item on a level determines whether the item is ready to be flushed down from that level during a flush.

We also assign an age to each level, initialized to 0. For each level involved in a flush its age is incremented before the flush. The age of a level gets wrapped around back to 0 after c increments. The age of the level during the flush determines which items are eligible to be flushed down — if an item's age is same as the age of the level then the item has survived c flushes on that level. When an item is inserted in a level it gets the current age of the level as its age. However, if the level already has an instance of the item then it takes the age of the instance on that level.

We follow a fixed schedule for flushes. A flush is performed every $(\frac{\alpha}{\alpha+1})M$-th stream observation. Every r^{i-1}-th flush involves level i. After every $r - 1$ flushes to level i, level $i + 1$ is involved in the next flush. To determine the number of levels involved in a flush, we maintain a counter per level for the number of times the level has been involved in a flush.

Consolidating item-counts during a flush is implemented as a k-way merge sort. We first aggregate the count of an item across all the levels involved in the flush. We then decide based on the age of the instance of the item in the last level whether to move it to the next level. If the instance of the item in the last level is aged then we insert the item with the aggregate count in the next level. Otherwise, we update the count of the instance in the last level to the aggregate count.

Count-stretch and Immediate-report LERT. We describe the implementation details of the count-stretch and immediate-report LERT, including further optimizations.

Similar to the flush schedule in the time-stretch LERT we follow a fixed shuffle-merge schedule. A shuffle-merge is invoked from RAM after every M observations. The level thresholds determine how many instances of an item can be stored at that level. To satisfy threshold constraints, during a shuffle merge, we first aggregate the count of each item and then smear it across all levels involved in the shuffle-merge in a bottom-up fashion without violating the thresholds.

Our implementation of the count-stretch LERT further reduces I/O costs by following a "greedy" flushing schedule instead of a fixed schedule. We only invoke a shuffle-merge only if it is needed, i.e., when the RAM is at capacity.

The CQF uses a variable-length encoding for storing counts and uses much less space compared to a unary-encoding. Therefore, the actual number of slots needed for storing M observations can be much smaller than M slots, if there are duplicates in the stream. Especially, in case of streams such as the one from Firehose, where counts have a power-law distribution, the space needed to store M observations is much smaller than M slots. The greedy shuffle-merge schedule avoids unnecessary I/Os that a fixed schedule would incur during shuffle-merges.

As explained in Section 5, in the immediate-report LERT we perform an immediate-reporting query when the count in RAM reaches $T - \sum_{i=1}^{L} \tau_i$. To compute the aggregate count we perform point queries to each level on disk and sum the counts. If the aggregate count in RAM and on disk is T we report the item, else we insert the aggregate count in RAM and set a bit, ***the absolute bit***, that indicates that all the counts for the item has been found. This avoids unnecessary point queries to disk later on. We use a lazy policy to delete the instances of items from disk. They get garbage collected during the next shuffle merge.

7 DEAMORTIZATION TO SUPPORT CONSISTENT INGESTION RATES

The LERTs consider observation t to occur exactly one time step before observation $t + 1$. In practice, however, observation t might trigger a significant rebuild of the data structure, delaying observation $t+1$. In a high-speed streaming context, that observation, and potentially millions after it, would be dropped while a rebuild is going on.

To mitigate this problem, we now describe how to deamortize LERTs. Our deamortization strategy works in serial, and also provides the foundation of the multithreading strategy we introduce in Section 8.

To deamortize, we decompose the data structure into C independent parts called ***cones*** that partition the space of hashed items. Each stream item is mapped to exactly one of these cones using a uniform-random hash function. A cone is an independent instance of the LERT with the same expansion factor r and the same number of levels, each of which is $1/C$-th the size of the corresponding complete level.

Each cone is independent, following its own merge schedule. Incoming items are routed to the appropriate cone for independent insertion and potential reporting. Thus, given uniform-random hashing, each cone accounts for roughly $1/C$-th of the aggregate I/O.

Deamortization timeliness guarantees. We consider the timeliness guarantees for the deamortized serial version of the count-stretch and time stretch LERT. When streams are split into substreams based on hash values, we must revisit these guarantees. We note that count-stretch is unaffected:

LEMMA 10. *A deamortized count-stretch LERT provides the same count stretch guarantee as the original count-stretch LERT when run on the same input stream.*

LEMMA 11. *There exists an input stream for which the deamortized time-stretch LERT provides no global time stretch.*

PROOF. We construct an arrival distribution that causes an arbitrarily long time stretch for an item in a deamortized time-stretch LERT. It begins with $T - 1$ observations of an item I followed by enough distinct items that all go to the item I's cone (C) to cause a flush in cone C. The sequence then has one more observation of item I followed by an arbitrarily long sequence of observations, none of which go to cone C. Thus, cone C has an arbitrary delay before its next merge and item I has an unbounded reporting delay. □

THEOREM 12. *Consider a random stream where each item maps to a cone via a fixed probability distribution. If cone i runs a time-stretch LERT guaranteeing a time stretch of $(1+\alpha)$, then the deamortized time-stretch LERT will have a time stretch of $(1 + \alpha)$ in expectation with respect to the full stream.*

8 MULTI-THREADING

We now describe thread-safe versions of the deamortized count-stretch and time-stretch LERT. A thread-safe implementation enables ingesting observations using multiple threads. This is crucial for two reasons: (1) we can scale the ingestion throughput to support high-speed streams, and (2) multiple threads performing I/Os simultaneously can utilize the full SSD bandwidth which would be wasted otherwise.

We use two types of locks in our design, a cone-level lock and a CQF-level lock. The cone-level lock is a distributed readers-writer lock implemented using a partitioned counter (i.e., a per-CPU counter). This ensures that readers do not thrash on the cache line containing the count of the number of readers holding the lock. The CQF-level lock is a spin lock as described by Pandey et al. [55].

We assign a small local insertion buffer to each thread. Each insertion thread performs the same set of operations. It starts by first receiving a packet of observations over a network port or reading a small chunk (usually 1024) of observations from an input file. Each observation is then processed one-by-one.

Each thread must acquire two locks to do an insertion: one read lock on the item's cone and one lock on the region of the CQF (i.e., the RAM level of the cone) to which the item hashes. It tries once to acquire each lock. It does not spin or sleep upon failing to acquire either lock. If it does not get either of the locks in the first attempt then it releases any acquired lock, inserts the observation in its local insertion buffer, and continues to the next observation. When the local buffer is full, items in the buffer are dumped into respective cones. When dumping a buffer, the threads wait for the locks.

If it acquires both the locks in the first attempt, then it performs the insertion and releases the lock on the relevant region of the CQF. It then checks whether the cone needs to perform a flush or shuffle-merge. If so, it first releases the read lock and then tries to acquire a write lock on the cone. If it gets the write lock in the first attempt then it performs the flush/shuffle-merge. If it fails to acquire the write lock in its first attempt, then some other thread is already performing a flush/shuffle-merge. This thread can continue.

We avoid heavy contention among threads via the local buffers, even when every thread tries to lock the same cone. Our method scales well with increasing number of insertion threads even for streams with skewed distributions. We show this empirically in Section 9.8.

Using readers-writer locks at the cone level enables multiple threads simultaneously insert in different regions of the RAM CQF of a cone by acquiring a read lock. A thread upgrades to a write lock when it needs to do a flush/shuffle-merge. Readers-writer locks allows us to use more threads than cones. Even if all cones flush simultaneously, there would still be threads processing incoming observations.

8.1 Timeliness with multi-threading

We now discuss the effect of multithreading on the timeliness guarantees of the count-stretch and time-stretch LERT.

Measuring time. One issue that immediately arises when trying to analyze time- and count-stretch in the multi-threaded case is: how do we measure time? In the single-threaded case, we measure time in terms of the number of stream observations that the process has ingested, i.e., in each time step, the algorithm gets to read one stream observation, perform an arbitrary amount of computation and I/O, and generate an arbitrary number of reports. We say all reports generated during the i-th time step occur at time i.

We generalize this in the multi-threaded model: when a thread reports items, it uses the index of the last observation pulled by any thread as the reporting time. This can cause the reporting index of an item be much higher compared to the single-threaded case because multiple threads pull a chunk (usually 1024) of observations simultaneously. Therefore, multi-threading adds an extra delay to the timeliness guarantees of the time-stretch and count-stretch LERT. We analyze this delay empirically in Section 9.4.

Count stretch. The multi-threaded count-stretch LERT has only one new source of delay: the time that an item might spend sitting in a thread's local buffers. In the worst case, an item could accumulate up to $T - 1$ occurrences in each thread's local buffer, in addition to $T - 1$ occurrences in the main data structure, so that it doesn't get reported until it reaches a count of $(T-1)(P-1)+1$.

To limit this pathological case, we implement a policy to upper bound the total count that an item can have in a thread's local buffer. For example, we enforce that no thread can hold more than $\frac{T}{P}$ instances of an item in its local buffer. Whenever the count of an item in the local buffer equals $\frac{T}{P}$ the thread must move that item from the thread's local buffer to the main data structure. This way we can bound the maximum count of an item when it is reported.

Lemma 13. *Given ω and T such that $T \geq P$, where P is the number of threads, a multi-threaded count-stretch LERT guarantees a count stretch of $2+\omega$.*

Proof. Whenever an item reaches a count of $(2+\omega)T$ it will be reported. Because the maximum count of an item in a thread local buffer can be $\frac{T}{P}$ and for P threads it can $\frac{T}{P} \times P = T$. Therefore, maximum count in the cone can be $(2+\omega)T - T = (1+\omega)T$ which is the stretch guaranteed by the count-stretch LERT. □

Time stretch. It is comparatively harder to provide a time stretch guarantee with multiple threads compared to the count stretch. Because time stretch depends on the arrival distribution of other items in the stream unlike count stretch which is independent of that.

When multiple threads are simultaneously performing ingestion, each thread can pick a chunk of observations from the stream. These observations can be inserted in the data structure out-of-order based on the contention among threads. To guarantee a time stretch with multiple threads we need to global ordering on the observations.

Model. In each time step, a thread gets to read one observation from the stream and perform all the work on that observation. The work includes taking a lock and inserting the observation in the cone, inserting the observation in the local buffer, and performing a flush/shuffle-merge on the cone. As above, we constrain how long a thread can go before dumping its local buffer. We also constraint that every thread has to dump their local buffer after every t time steps.

Based on the above model and constraints, we can now guarantee that the time stretch in the multi-threading case will not be much worse than the single-threaded case.

Corollary 3. *In a multi-threaded time-stretch LERT in which each thread dumps its local buffer every t time steps, we can guarantee a time stretch of $(1+\alpha)F_T + t$, where F_T is the time an item takes to reach a count of T.*

9 EVALUATION

In this section, we evaluate our implementations of the time-stretch LERT (TSL), count-stretch LERT (CSL), and immediate-report LERT (IRL) for timeliness, robustness to input distributions, I/O performance, insertion throughput, and scalability with multiple threads.

We compare our implementations against Bender et al.'s cascade filter [12] as a baseline for timeliness. This baseline is an external-memory data structure with no timeliness guarantee. We show that reporting delays can be quite large when data structures take no special steps to ensure timeliness.

We also evaluate an implementation of the Misra-Gries data structure as a baseline for in-memory insertion throughput. We implement the Misra-Gries data structure with an exact counting data structure (counting quotient filter) to forbid false positives. This gives an upper bound on the insertion throughput one can achieve in-memory while performing immediate event-detection. The objective of this baseline is to evaluate the effect of disk accesses during flushes/shuffle-merges in our implementations of the TSL, CSL, and IRL.

We address the following performance questions for the time-stretch, count-stretch and immediate-report LERT:

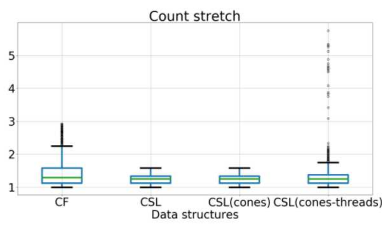
(1) How does the empirical timeliness of reported items compare to the theoretical bounds?
(2) How robust is the time-stretch LERT to different input distributions?
(3) How does deamortization and multi-threading affect the empirical timeliness of reported items?
(4) How does the buffering strategy affect count stretch and throughput?
(5) How does LERT total I/O compare to theoretical bounds?
(6) What is the insertion throughput of the time-stretch, count-stretch and immediate-report LERT?
(7) How does deamortization and multiple threads affect instantaneous throughput?
(8) How does insertion throughput scale with number of threads?

9.1 Experimental setup

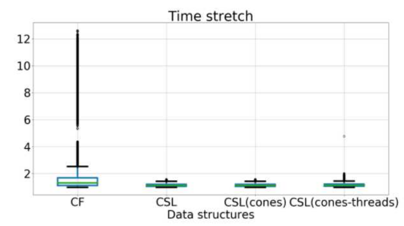
We describe how we designed experiments to answer the questions above. We describe our workloads, and how we validated timeliness and measured I/O performance.

Workload. Firehose [42] is a suite of benchmarks simulating a network-event monitoring workload. A Firehose benchmark consists of a *generator* that feeds keys to the *analytic*, being benchmarked. The analytic must detect and report each key that has 24 observations. .

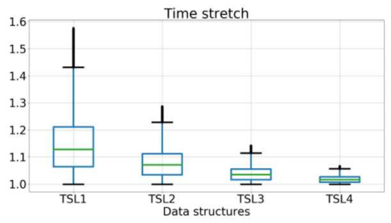
Firehose includes two generators: the power-law generator selects from a static ground set of 100,000 keys according to a power-law distribution, while the active-set generator allows the ground set to drift over an infinite key space. We use the active-set generator because an infinite key space more closely matches many real world streaming workloads.



(a) Distribution of count stretch of different data structures.

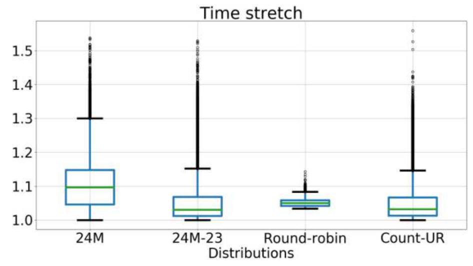


(b) Distribution of time stretch of different data structures.

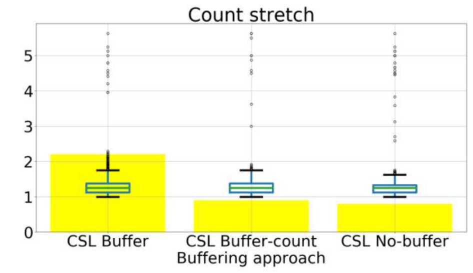


(c) Distribution of time stretch in the time-stretch LERT for different α values.

Figure 1: Data structure configuration: RAM level: 8388608 slots in the CQF, levels: 4, growth factor: 4, level thresholds: (2, 4, 8), cones: 8, threads: 8, number of observations: 512M. Data structures: Cascade filter (CF), count-stretch LERT (CSL), time-stretch LERT (TSL), (CSL and TSL) with cones, (CSL and TSL) with cones and threads. Time-stretch LERT with age bits 1 (TSL1) $\alpha = 1$, 2 (TSL2) $\alpha = 0.33$, 3 (TSL3) $\alpha = 0.14$, and 4 (TSL4) $\alpha = 0.06$.

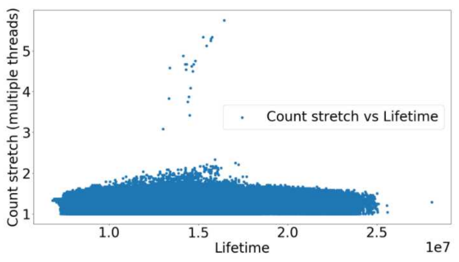


(a) Distribution of time stretch for different distributions. These distributions are described in Section 9.1.

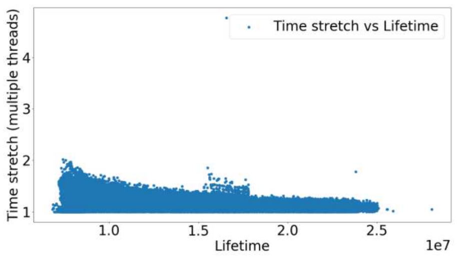


(b) Distribution of count stretch with different buffering strategies. Bars show the average insertion throughput (Million insertions/sec) for each buffering strategy. Average insertion throughput when no-buffer is used is $2.7\times$ lower compared to when buffers are used.

Figure 2: Data structure configuration: RAM level: 8388608 slots in the CQF, levels: 4, growth factor: 4, level thresholds for on-disk level: (2, 4, 8), cones: 8, threads: 8, number of observations: 512M.



(a) Distribution of count stretch vs lifetime of reported items in a CSL with 8 cones and 8 threads.



(b) Distribution of time stretch vs lifetime of reported items in a TSL with 8 cones and 8 threads.

Figure 3: Data structure configuration: RAM level: 8388608 slots in the CQF, levels: 4, growth factor: 4, level thresholds for on-disk level: (2, 4, 8), cones: 8, threads: 8, number of observations: 512M.

To simulate a stream of keys drawn from a huge key-space we increase the key space of the active set to one million.

Other workloads. Apart from Firehose, we use four other simulated workloads to evaluate the empirical stretch in the time-stretch LERT. These four workloads are generated to show the robustness of the data structure to non-power-law distributions. In the first distribution, M (where M is the size of the level in RAM) keys appear with a count between 24–50 and rest of the keys are chosen uniformly at random from a big universe. In the second, M keys appear 24 times and the rest of the keys appear 23 times. In the third, M keys appear round robin each with a count > 24. In the fourth, for each key we pick the count uniformly at random between 1–25.

Reporting. During insertion, we record each reported item and the index in the stream at which it is reported by the data structure. We record by inserting the reported item in an exact CQF (anomaly CQF) and encoding the index as the count of the item in the anomaly CQF. We also use the anomaly CQF to check if an incoming item has already been reported. We only insert the item if it is not reported yet. This prevents duplicate reports.

Timeliness. For the timeliness evaluation, we measure the reporting delay after its Tth occurrence. We have two measures of timeliness: time stretch and count stretch.

The time-stretch LERT upper bounds the reporting delay of an item based on its lifetime (i.e. time between its first and Tth instance). To validate the timeliness of the time-stretch LERT, we first perform an offline analysis of the stream and calculate the lifetime of each reportable item. Given a reporting threshold T, we record the index of the first occurrence of the item (I_0) and the index of the T-th occurrence of the item (I_T). During ingestion, we record the index (I_R) at which the time-stretch LERT reports the item. We calculate the time stretch (ts) for each reported item as $ts = (I_R - I_0)/(I_T - I_0)$ and verify that $ts \leq (1 + \alpha)$.

Multiple threads process chunks of 1024 observations from the input stream. We consider all reports a thread generates while processing the ith observation to occur at time i. Due to concurrency, two observations of the same key may be inserted into the data structure in a different order than they are pulled off of the input stream. This may introduce some noise in our time-stretch measurements. However, our experimental results with and without multi-threading were nearly identical, indicating that the noise is small.

In the count-stretch LERT, the upper bound is on the count of the item when it is reported. To validate timeliness, we first record indexes at which items are reported by the count-stretch LERT (I_R). We then perform an offline analysis to determine the count of the item at index I_R (C_{I_R}) in the stream. We then calculate the count stretch (cs) as $cs = C_{I_R}/T$ and validate that $cs \leq (T + \sum_{i=1}^{L} \tau_i)/T$.

To perform the offline analysis of the stream we first generate the stream from the active-set generator and dump it in a file. We then read the stream from the file for the analysis and for streaming it to the data structure. For timeliness validation experiments we use a stream of 512 Million observations from the active-set generator.

I/O performance. In our implementation of the time-stretch, count-stretch and immediate-report LERT, we allocate space for the data structure by mmap-ing each level (i.e., the CQF) to a file on SSD. To force the data structure to keep all levels except the first one on SSD we limit the RAM available to the insertion process using the "cgroups" utility in linux. We calculate the total RAM needed by the insertion process to only keep the first level in RAM by adding the size of the first level, the space used by the anomaly CQF to record reported keys, the space used by thread-local buffers, and a small amount of extra space to read the stream sequentially from SSD. We then provision the RAM to the next power-of-two of the total sum.

To measure the total I/O performed by the data structure we use the "iotop" utility in linux. Using `iotop` we can measure the total amount of reads and writes in KB performed by the process doing insertions.

To validate, we calculate the total amount of I/O performed by the data structure based on the number of merges (shuffle-merges in case of the count-stretch LERT) and time-stretch LERT and sizes of levels involved in those merges.

Similar to validation experiments, we first dump the stream to a file and then feed the stream to the data structure by streaming it from the file. We use a stream of 64 Million observations from the active-set generator.

Average insertion throughput and scalability. To measure the average insertion throughput, we first generate the stream from the active-set generator and dump it in a file. We then feed the stream to the data structure by streaming it from the file and measure the total time.

To evaluate scalability, we measure how data-structure throughput changes with increasing number of threads. We evaluate power-of-2 thread counts between 1 and 64.

To deamortize the data structures we divide them into 2048 cones. We use a stream of 4 Billion observations from the active-set generator. We evaluate the insertion performance and scalability for three values (16, 32 and 64) of the DatasetSize-to-RAM-ratio (i.e., the ratio of the data set size to the available RAM).

Instantaneous insertion throughput. We also evaluate the instantaneous throughput of the data structure when run using either a single cone and thread or multiple cones and threads. We approximate instantaneous throughput by calculating throughput (using system timestamps) every κ observations. In our evaluation, we fix $\kappa = 2^{17}$.

Machine specifications. The OS for all experiments was 64-bit Ubuntu 18.04 running Linux kernel 4.15.0-34-generic The machine for all timeliness and I/O performance benchmarks had an Intel Skylake CPU (Core(TM) i7-6700HQ CPU @ 2.60GHz with 4 cores and 6MB L3 cache) with 32 GB RAM and a 1TB Toshiba SSD. The machine for all scalability benchmarks had an Intel Xeon(R) CPU (E5-2683 v4 @ 2.10GHz with 64 cores and 20MB L3 cache) with 512 GB RAM and a 1TB Samsung 860 SSD.

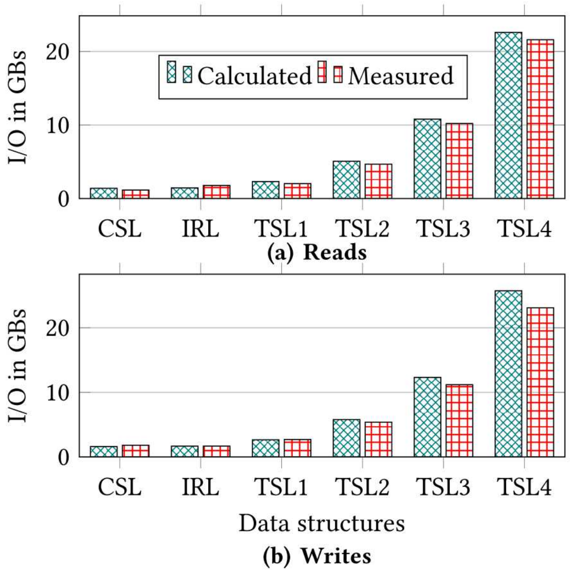


Figure 4: Total I/O performed by the count-stretch, time-stretch and immediate report LERT. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of observations: 64M. Immediate-report LERT (IRL).

For all the experiments, we use a reporting threshold of 24 since it is the default in the Firehose benchmarking suite.

9.2 Timely reporting

Cascade filter. Figures 1a and 1b show the distribution of count stretch and time stretch of reported items in the cascade filter. The cascade filter's maximum count-stretch is 3.0 and maximum time stretch is > 12, much higher than any single-threaded count-stretch or time-stretch LERT.

Count-stretch LERT. Figure 1a validates worst-case count stretch for the count-stretch LERT. The total on-disk count for an element is 14, so the maximum possible count when reported is 38 (i.e., 24 + 14), for a maximum count stretch of 1.583. The maximum reported count stretch is 1.583.

Time-stretch LERT. Figure 1b shows the time-stretch LERT meets the time-stretch requirements. The maximum reported time stretch is 1.59 which is smaller than the maximum allowable time stretch of 2. Figure 1c shows the distribution of empirical time stretches with changing α values. The time stretch of any reported element is always smaller than the maximum allowable time stretch. As the number of age bits increases, α decreases and the time stretch decreases.

9.3 Robustness with input distributions

Figure 2a shows the robustness of empirical time stretch (ETS) on four input distributions other than the Firehose power-law distribution. The ETS is less than 2, the theoretical limit of the data structure for all input distributions.

9.4 Effect of deamortization/threading

Figures 1a and 1b show the effect of deamortization and multi-threading on timeliness in the count-stretch LERT and time-stretch LERT.

Using 8 cones instead of one does not change the timeliness of any reported item. This is because the distribution of items in the stream is random (see Section 9.1) and we use a uniform-random hash function to distribute items to each cone. Each cone gets a similar number of items and the cones perform shuffle-merges in sync (refer to Section 7).

Running the count-stretch and time-stretch LERT with 8 cones and 8 threads does affect timeliness of reported items. Some items are reported later than the theoretical upper bound. The reported maximum time- and count-stretch is $>$ 5. This is because each thread inserts items into a local buffer when it can not immediately acquire the cone lock. We empty local buffers only when they are full. The maximum delay happens when an item's lifetime is similar to the time it takes for a cone to incur a full flush involving all levels of the data structure. Figure 3 shows the stretch of reported items and their lifetime. The maximum-stretch items have a lifetime $\approx$ 16M observations which is the number of observations it takes for a cone to incur a full flush.

9.5 Effect of buffering

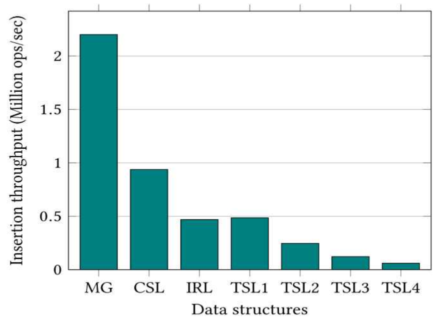
Figure 2b shows the empirical count stretch with three different buffering strategies. In the first, we use buffers without any constraint on the count of a key inside a buffer. We dump the buffer into the main data structure when it is full. In the second, we constrain the maximum count a key can have in a buffer to T/P (for $T = 24$ and $P = 8$ the max count is 3). In the third, we don't use buffers. Threads try to acquire the lock on the cone and wait if the lock is not available.

The empirical stretch is smallest without buffers. However, not using the buffers increases contention among threads and reduces insertion throughput. Using the buffers is 2.5× faster compared to not using the buffer.

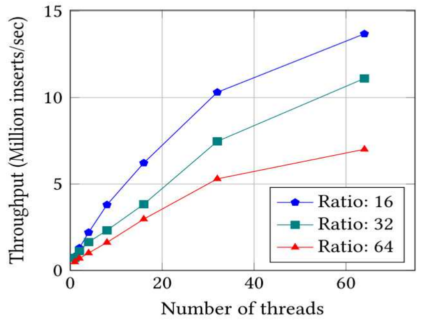
9.6 I/O performance and throughput

Figure 4 shows the total amount of I/O performed by the count-stretch, time-stretch and immediate-report LERT while ingesting a stream. For all data structures, the total I/O calculated and total I/O measured using `iotop` is similar.

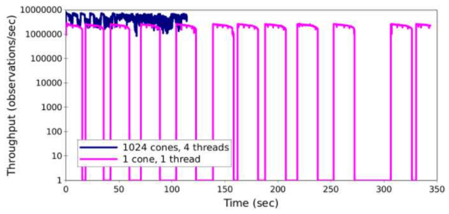
The count-stretch LERT does the least I/O because it performs the fewest shuffle-merges. The I/O for the time-stretch



(a) **Items inserted per second by the CSL, TSL, IRL and Misra-Gries (MG) data structure. MG is in-memory.**



(b) **Insertion throughput with increasing number of threads for the count-stretch LERT on 4 Billion observations.**



(c) **Instantaneous throughput of the count-stretch LERT with 1 cone and 1 thread and 1024 cones and 4 threads.**

Figure 5: Data structure configuration for (a): RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of observations: 64M. DatasetSize-to-RAM-ratio: 12.5. For (b): RAM level: 67108864 slots in the CQF, levels: 4, growth factor: 4, level thresholds for on-disk level($\ell_3 \ldots \ell_1$): (2, 4, 8), cones: 2048 with greedy flushing, DatasetSize-to-RAM-ratio: 16, 32, and 64. For (c): Same as Figure 1a.

LERT grows by a factor of two as the number of bins increases, as predicted by the theory. The I/O for immediate-report LERT is similar to that of the time-stretch LERT with stretch 2. This shows that when item counts follow a power-law distribution, we can achieve immediate reporting with the same amount of I/O as with a time stretch of 2.

Insertion throughput. Figure 5a shows insertion throughput using the same configuration and stream as the total-I/O experiments. The count-stretch LERT has the highest throughput because it performs the fewest I/Os. The immediate-report LERT has lower throughput because it performs extra random point queries. The time-stretch LERT throughput decreases as we add bins and decrease the stretch.

The **Misra-Gries data structure** throughput is 2.2 Million ops/sec in-memory. This acts a baseline for in-memory insertion throughput. The in-memory MG data structure is only twice as fast as the on-disk count-stretch LERT.

9.7 Instantaneous throughput

Figure 5c shows the instantaneous throughput of the count-stretch LERT. De-amortization and multi-threading improve both average throughput and throughput variance. With one thread and one cone, the data structure periodically stops processing inputs to perform flushes, causing throughput to crash to 0. With 1024 cones and four threads, the system has much smoother throughput, never stops processing inputs, and has about 3× greater average throughput.

9.8 Scaling with multiple threads

Figure 5b shows count-stretch LERT throughput with increasing number of threads. The scalability will follow for other variants since they all have the same insertion and SSD access pattern. The insertion throughput increases with thread count. We used three values of DatasetSize-to-RAM-ratio: 16, 32, and 64. All have similar scalability curves.

10 CONCLUSION

This work bridges external-memory and streaming algorithms. By taking advantage of external memory, we can solve timely event detection problems at a level of precision that is not possible in the streaming model, and with little or no sacrifice in terms of the timeliness of reports.

Even though streaming algorithms, such as Misra-Gries, were developed for a space-constrained setting, we show that they can be made efficient in the external-memory setting, where storage is plentiful but accessing the data is expensive.

Acknowledgments

We thank Tyler Mayer for helpful discussions. We gratefully acknowledge support from NSF grants CCF 805476, CCF 822388, CCF 1724745, CCF 1715777, CCF 1637458, IIS 1541613, CRII 1947789, CNS 1408695, CNS 1755615, CCF 1439084, CCF 1725543, CSR 1763680, CCF 1716252, CCF 1617618, CNS 1938709, IIS 1247726, CNS-1938709. This research is funded in part by the Gordon and Betty Moore Foundation's Data-Driven Discovery Initiative through Grant GBMF4554, the US National Institutes of Health (R01GM122935), VMware, and the Laboratory-Directed Research-and-Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] [n. d.]. FireHose Streaming Benchmarks. www.firehose.sandia.gov. Accessed: 2018-12-11.

[2] LA Adamic. 2008. Zipf, Power law, Pareto: a ranking tutorial. HP Research. http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html.

[3] Alok Aggarwal and Jeffrey Vitter. 1988. The input/output complexity of sorting and related problems. *Commun. ACM* 31, 9 (1988), 1116–1127.

[4] Noga Alon, Yossi Matias, and Mario Szegedy. 1996. The space complexity of approximating the frequency moments. In *Proc. 28th Annual ACM Symposium on Theory of Computing (STOC)*. 20–29.

[5] Karl Anderson and Steve Plimpton. 2015. *FireHose Streaming Benchmarks.* Technical Report. Sandia National Laboratory.

[6] Shivnath Babu and Jennifer Widom. 2001. Continuous queries over data streams. *ACM SIGMOD Record* 30, 3 (2001), 109–120.

[7] Daniel Barbará. 1999. The characterization of continuous queries. *International Journal of Cooperative Information Systems* 8, 04 (1999), 295–323.

[8] Michael A. Bender, Jonathan W. Berry, Martin Farach-Colton, Justin Jacobs, Rob Johnson, Thomas M. Kroeger, Tyler Mayer, Samuel McCauley, Prashant Pandey, Cynthia A. Phillips, Alexandra Porter, Shikha Singh, Justin Raizes, Helen Xu, and David Zage. 2018. *Advanced Data Structures for Improved Cyber Resilience and Awareness in Untrusted Environments: LDRD report.* Technical Report SAND2018-5404. Sandia National Laboratories.

[9] Michael A. Bender, Alex Conway, Martin Farach-Colton, William Jannen, Yizheng Jiao, Rob Johnson, Eric Knorr, Sara McAllister, Nirjhar Mukherjee, Prashant Pandey, Donald E. Porter, Jun Yuan, and Yang Zhan. 2019. Small Refinements to the DAM Can Have Big Consequences for Data-Structure Design. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 265–274.

[10] Michael A Bender, Martin Farach-Colton, Jeremy T Fineman, Yonatan R Fogel, Bradley C Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming B-trees. In *Proc. 19th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 81–92.

[11] Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, Jun Yuan, and Yang Zhan. 2015. An Introduction to B$^{\varepsilon}$-Trees and Write-Optimization. *:login; magazine* 40, 5 (October 2015), 22–28.

[12] Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. 2012. Don't Thrash: How to Cache Your Hash on Flash. *PVLDB* 5, 11 (2012), 1627–1637.

[13] Michael A Bender, Martín Farach-Colton, Rob Johnson, Simon Mauras, Tyler Mayer, Cynthia A Phillips, and Helen Xu. 2017. Write-Optimized Skip Lists. In *Proc. 36th Symposium on Principles of Database Systems (PODS)*. ACM, 69–78.

[14] Radu Berinde, Piotr Indyk, Graham Cormode, and Martin J Strauss. 2010. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems* 35, 4 (2010), 26.

[15] Jonathan Berry, Robert D. Carr, William E. Hart, Vitus J. Leung, Cynthia A. Phillips, and Jean-Paul Watson. 2009. Designing Contamination Warning Systems for Municipal Water Networks Using Imperfect Sensors. *Journal of Water Resources Planning and Management* 135 (2009). Issue 4.

[16] Kevin Beyer and Raghu Ramakrishnan. 1999. Bottom-up computation of sparse and iceberg cube. In *ACM SIGMOD Record*, Vol. 28. 359–370.

[17] Arnab Bhattacharyya, Palash Dey, and David P Woodruff. 2016. An optimal algorithm for l1-heavy hitters in insertion streams and related problems. In *Proc. 35th ACM Symposium on Principles of Database Systems (PODS)*. 385–400.

[18] Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. 2003. Bounds for Frequency Estimation of Packet Streams.. In *SIROCCO*. 33–42.

[19] Robert S Boyer and J Strother Moore. 1981. *A fast majority vote algorithm.* SRI International. Computer Science Laboratory.

[20] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, Jelani Nelson, Zhengyu Wang, and David P Woodruff. 2016. BPTree: an ℓ_2 heavy hitters algorithm using constant memory. *arXiv preprint arXiv:1603.00759* (2016).

[21] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, and David P Woodruff. 2016. Beating CountSketch for heavy hitters in insertion streams. In *Proc. 48th Annual Symposium on Theory of Computing (STOC)*. ACM, 740–753.

[22] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. Annual Joint Conference of the IEEE Computer and Communications Societies*, Vol. 1. 126–134.

[23] Gerth Stølting Brodal, Erik D. Demaine, Jeremy T. Fineman, John Iacono, Stefan Langerman, and J. Ian Munro. 2010. Cache-Oblivious Dynamic Dictionaries with Update/Query Tradeoffs. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 1448–1456.

[24] Gerth Stølting Brodal and Rolf Fagerberg. 2003. Lower Bounds for External Memory Dictionaries. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 546–554.

[25] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. 2000. On external memory graph traversal. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 859–860.

[26] Pedro Celis, Per-Ake Larson, and J Ian Munro. 1985. Robin hood hashing. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*. IEEE, 281–288.

[27] Sirish Chandrasekaran and Michael J Franklin. 2002. Streaming queries over streaming data. In *Proc. 28th International conference on Very Large Data Bases*. VLDB Endowment, 203–214.

[28] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*. 693–703.

[29] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.

[30] Alex Conway, Martin Farach-Colton, and Philip Shilane. 2018. Optimal Hashing in External Memory. In *Proc. 45th International Colloquium on Automata, Languages, and Programming (ICALP)*. 39:1–39:14.

[31] Graham Cormode and Marios Hadjieleftheriou. 2010. Methods for finding frequent items in data streams. *The VLDB Journal* 19, 1 (2010), 3–20.

[32] Graham Cormode and S Muthukrishnan. 2004. An improved data stream summary: The count-min sketch and its applications. In *Proc. Latin American Symposium on Theoretical Informatics*. 29–38.

[33] Graham Cormode and S Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[34] Graham Cormode and S Muthukrishnan. 2005. What's hot and what's not: tracking most frequent items dynamically. *ACM Transactions on Database Systems* 30, 1 (2005), 249–278.

[35] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. 2002. Frequency estimation of internet packet streams with limited space. In *Proc. European Symposium on Algorithms (ESA)*. Springer, 348–360.

[36] Xenofontas Dimitropoulos, Paul Hurley, and Andreas Kind. 2008. Probabilistic lossy counting: an efficient algorithm for finding heavy hitters. *ACM SIGCOMM Computer Communication Review* 38, 1 (2008), 5–5.

[37] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D Ullman. 1998. Computing Iceberg Queries Efficiently.. In *Proc. 24rd International Conference on Very Large Databases (VLDB)*. 299–310.

[38] Jose M. Gonzalez, Vern Paxson, and Nicholas Weaver. 2007. Shunting: A Hardware/Software Architecture for Flexible, High-performance Network Intrusion Prevention. In *Proc. 14th ACM Conference on Computer and Communications Security (CCS)*. 139–149.

[39] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. 2001. Efficient computation of iceberg cubes with complex measures. In *ACM SIGMOD Record*, Vol. 30. 1–12.

[40] John Hershberger, Nisheeth Shrivastava, Subhash Suri, and Csaba D Tóth. 2005. Space complexity of hierarchical heavy hitters in multi-dimensional data streams. In *Proc. 24th Symposium on Principles of Database Systems (PODS)*. ACM, 338–347.

[41] John Iacono and Mihai Pătraşcu. 2012. Using hashing to solve the dictionary problem. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 570–582.

[42] Steve Plimpton Karl Anderson. 2013. FireHose. http://firehose.sandia.gov/. [Online; accessed 19-Dec-2015].

[43] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. 2003. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems* 28, 1 (2003), 51–55.

[44] M. Kezunovic. 2006. Monitoring of Power System Topology in Real-Time. In *Proc. 39th Annual Hawaii International Conference on System Sciences*, Vol. 10. 244b–244b. https://doi.org/10.1109/HICSS.2006.355

[45] Kasper Green Larsen, Jelani Nelson, Huy L Nguyen, and Mikkel Thorup. 2016. Heavy hitters via cluster-preserving clustering. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 61–70.

[46] Quentin Le Sceller, ElMouatez Billah Karbab, Mourad Debbabi, and Farkhund Iqbal. 2017. SONAR: Automatic Detection of Cyber Security Events over the Twitter Stream. In *Proc. 12th International Conference on Availability, Reliability and Security*.

[47] E. Litvinov. 2006. Real-time Stability in Power Systems: Techniques for Early Detection of the Risk of Blackout [Book Review]. *IEEE Power and Energy Magazine* 4, 3 (May 2006), 68–70. https://doi.org/10.1109/MPAE.2006.1632456

[48] Jianning Mai, Chen-Nee Chuah, Ashwin Sridharan, Tao Ye, and Hui Zang. 2006. Is sampled data sufficient for anomaly detection?. In *Proc. 6th ACM SIGCOMM conference on Internet measurement*. 165–176.

[49] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. In *Proc. 28th International Conference on Very Large Data Bases*. VLDB Endowment, 346–357.

[50] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. 2010. Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *Proc. 19th USENIX Conference on Security*.

[51] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *Proc. International Conference on Database Theory*. Springer, 398–412.

[52] Jayadev Misra and David Gries. 1982. Finding repeated elements. *Science of computer programming* 2, 2 (1982), 143–152.

[53] Mark EJ Newman. 2005. Power laws, Pareto distributions and Zipf's law. *Contemporary physics* 46, 5 (2005), 323–351.

[54] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[55] Prashant Pandey, Michael A. Bender, Rob Johnson, and Robert Patro. 2017. A General-Purpose Counting Filter: Making Every Bit Count. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 775–787. https://doi.org/10.1145/3035918.3035963

[56] Shahid Raza, Linus Wallgren, and Thiemo Voigt. 2013. SVELTE: Real-time intrusion detection in the Internet of Things. *Ad Hoc Networks* 11, 8 (2013), 2661–2674. http://dblp.uni-trier.de/db/journals/adhoc/adhoc11.html#RazaWV13

[57] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. *Department of Electrical and Computing Engineering*.

[58] H. Yan, R. Oliveira, K. Burnett, D. Matthews, L. Zhang, and D. Massey. 2009. BGPmon: A Real-Time, Scalable, Extensible Monitoring System. In *2009 Cybersecurity Applications Technology Conference for Homeland Security*. 212–223. https://doi.org/10.1109/CATCH.2009.28