Timely Reporting of Heavy Hitters using External Memory

SHIKHA SINGH*, Williams College
PRASHANT PANDEY*, Lawrence Berkeley National Laboratories and University of California Berkeley
MICHAEL A. BENDER, Stony Brook University
JONATHAN W. BERRY, Sandia National Laboratories
MARTÍN FARACH-COLTON, Rutgers University
ROB JOHNSON, VMware Research
THOMAS M. KROEGER, Sandia National Laboratories
CYNTHIA A. PHILLIPS, Sandia National Laboratories

Given an input stream S 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 it 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 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., $\approx$ 100K observations per second.

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

Additional Key Words and Phrases: Dictionary data structure; streaming algorithms; external-memory algorithms

*Both authors contributed equally to this research.

A preliminary version of this article appeared in SIGMOD '20 [58], where it earned a reproducibility badge.
Authors' addresses: Shikha Singh, shikha@cs.williams.edu, Williams College; Prashant Pandey, ppandey@berkeley.edu, Lawrence Berkeley National Laboratories and University of California Berkeley; 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.

XXXX-XXXX/2021/1-ART1 $15.00
https://doi.org/10.1145/3472392

ACM Reference Format:
Shikha Singh, Prashant Pandey, Michael A. Bender, Jonathan W. Berry, Martín Farach-Colton, Rob Johnson, Thomas M. Kroeger, and Cynthia A. Phillips. 2021. Timely Reporting of Heavy Hitters using External Memory. 1, 1, Article 1 (January 2021), 35 pages. https://doi.org/10.1145/3472392

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 [47, 59, 62] as well as for physical systems, e.g., for water or power distribution [8, 45, 48]. In such a monitoring system, the stream elements represent the changes to the state of the system. Each detected/reported event could trigger 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 [40, 51]. 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 must report all elements that occur at least T times in S. In the preliminary version of this paper [58], we introduced the real-time version of the heavy-hitters problem called the Timely Event Detection (TED) problem. In the TED problem, each heavy hitter must be reported soon after its Tth occurrence, where the acceptable reporting delay is defined by the application.

In network-security monitoring applications, N is huge and T can be very small. 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 [49, 61]. 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 [4, 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 [3, 17, 19, 20, 22, 23, 30, 35–38, 46, 50].

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 allow false positives. Third, many streaming algorithms perform some kind of sampling, which leads to 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. See Section 9 for a motivating application, where any of these three limitations would lead to failure.

Combining streaming and external memory. This work challenges the assumption that only in-RAM data structures can keep up with real-world streams and shows 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.

In particular, we present algorithms in the external-memory model that support both exact and approximate reporting of heavy hitters. In the external-memory model [2], 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 contributions. In this paper, we present I/O-efficient external-memory algorithms that support both exact and approximate reporting of heavy hitters. Specifically, our TED algorithms can be generalized to solve the ***(ε, ϕ)-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.

The present paper serves as the journal version for [58], and it also contains technical improvements. Our first contribution is theoretical. We include proofs for all lemmas and theorems, unlike [58], which, for space reasons, omitted essentially all proofs. Explaining and proving these results has more than doubled the length of the paper, and thus includes results that are indeed reproducible. Furthermore, we generalize the results for power-law streams presented in [58] by specifying the precise relationship between the reporting threshold ϕ and the power-law exponent θ; for details see Section 5.

Our second contribution is experimental. We include all of the experiments from [58] along with additional ones. In particular, we give empirical analysis of the birthtime versus lifetime of items in the active-set generator of the Firehose streaming benchmark [4, 5]. Also, in the interest of reproducibility, we have included pseudocode for all data structures and algorithms.

Finally, we provide detailed explanation of how the constraints of the TED problem are motivated from practice. In particular, in Section 9, we discuss the national-security application that motivates the Firehose benchmark [4, 5], and how the TED problem captures the main computational bottleneck of this application.

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 [11, 13, 14, 26], COLAs [12], xDicts [25], buffered repository trees [27], write-optimized skip lists [16], log structured merge trees [56], and optimal external-memory hash tables [32, 43]) 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 [26] index of all connections on a 600 gigabit/sec network [10]. 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 [54]. 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 formally defining an event that must be reported in the TED problem. Given a stream $S = (s_1, s_2, \ldots, s_N)$, a ***$\phi$-heavy hitter*** is an element that occurs at least ϕN times in S. The heavy-hitters problem is to report all ϕ-heavy-hitters in S.

In the TED problem, we say that there is a ***ϕ-event at time step t*** if stream element s_t occurs exactly $\lceil \phi N \rceil$ times in $(s_1, s_2, \ldots, s_t)$. Thus for each ϕ-heavy hitter there is a single ϕ-event, which occurs when the element's count reaches the ***reporting threshold*** $T = \lceil \phi N \rceil$. In the TED problem, the goal is to report ϕ-events as soon as they occur.

Our first data structure, the ***Misra-Gries LERT***, 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 $O((1/B)\log(N/M))$ 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 αF_s time steps after s's Tth occurrence, where F_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 $\phi > 1/M$ 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 [12, 26, 27]. The time-stretch LERT guarantees hold for any input distribution; see Corollary 2.

In our evaluations, the time-stretch LERT with stretch 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.

Our ***count-stretch LERT*** is tailored to guarantee count-stretch on input stream distributions where the count for each item is drawn from a power-law distribution. In particular, given an input stream with item counts distributed according to a power-law with parameter $\theta > 2$, which is the typical range [1, 17, 24, 31, 55], and parameters T and ω, such that $\omega T > \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1}\left(\frac{N}{M}\right)^{\frac{1}{\theta-1}}$, 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 9. 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 with element counts drawn from

power-law distributions; see Theorem 9 for the formal I/O cost. In our evaluation, we find that the immediate-report LERT can ingest at ≈ 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 [33].

Two main strategies have been used: deterministic counter-based approaches [20, 37, 44, 50, 52, 54] and randomized sketch-based approaches [30, 34]. The first is based on the classic Misra and Gries (MG) algorithm [54], which generalizes the Boyer-Moore majority finding algorithm [21].

Randomized sketch-based algorithms such as count-min sketch [34] maintain a small sketch of the frequency vectors using compact hash functions.

More recent work has focused on generalizations of the heavy-hitters problem. Ting [60] considers aggregating subset sums, rather than counts, and Ben-Basat et al. [9] generalize the heavy hitter problem to sliding windows. Multiple researchers [53, 63, 64] have designed heavy-hitter algorithms for detecting top flows in networking applications.

Database iceberg queries. The TED problem is related to the problem of answering ***iceberg queries*** in databases [18, 39, 41, 42]. 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 an instance of a ***continuous*** or ***standing query*** over a database [6, 7, 29]. 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 formalize our model and review several building blocks of our data structures: the Misra-Gries heavy-hitters algorithm [54], counting quotient filters (CQF) [57], and cascade filters (CF) [12].

TED problem and model. The TED problem is: given stream $S = (s_1, s_2, \ldots, s_N)$, for each i, if there is a ϕ-event at time i, report s_i before time j, such that the reporting delay, $j - i$, is within an acceptable degree of tolerance. In the Misra-Gries LERT in Section 3.2, there is no reporting delay. In the time-stretch LERT in Section 4, the reporting delay is dependent on the flow time of the item (the time it takes for the item's count to go from zero to $T = \phi N$), and in the count-stretch LERT in Section 5.2, the reporting delay is count-dependent.

We measure time in terms of the number of stream observations. That is, in each time step, the algorithm reads one stream observation, performs an arbitrary amount of computation and I/O, and generates an arbitrary number of reports. We say all reports generated during the ith time step occur at time i.

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.

To see why this algorithm ensures that $C[s] \leq f_s < C[s] + \varepsilon N$ for all $s \in \mathcal{U}$, note that a $C[s]$ is incremented only for an occurrence of s in S. Thus $C[s] \leq f_s$. For the upper bound, whenever we decrement $C[s]$, then $\lceil 1/\varepsilon \rceil$ other items have their count decremented. This can happen at most $\lfloor N/\lceil 1/\varepsilon + 1 \rceil \rfloor$ times. 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 $C[s] > (\phi - \varepsilon)N$.

For a frequency estimation error of ε, Misra-Gries uses $O(\lceil 1/\varepsilon \rceil)$ words of storage, assuming each stream item and each count occupy $O(1)$ words.

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) [57] can be viewed as a hash table based on Robin-Hood hashing [28]. 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) [15] is a write-optimized data structure based on the CQF [57] and the COLA [12]. 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 $O\left(\frac{1}{B}\log_r\left(\frac{1}{\varepsilon M}\right)\right)$. 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 $O\left(\frac{1}{B}\log\left(\frac{1}{\varepsilon M}\right)\right)$, which is $o(1)$ when $B = \omega\left(\log\left(\frac{1}{\varepsilon M}\right)\right)$.

3.1 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.

The external-memory MG algorithm processes the input stream by inserting each item in the stream into C_0. To insert an item x into level i, do the following:

- If $x \in C_i$, then increment $C_i[x]$.
- If $x \notin C_i$, and $|C_i| \leq r^i M - 1$, then $C_i[x] \leftarrow 1$.
- If $x \notin C_i$ and $|C_i| = r^i M$, then, for each x' in C_i, decrement $C_i[x']$; remove x' from C_i if $C_i[x']$ becomes 0. If $i < L - 1$, recursively insert x' into C_{i+1}.

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 an 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,*
- $\widehat{C}_{L-1}[x] \leq f_x < \widehat{C}_{L-1}[x] + \varepsilon N$.

Proof. Decrementing the count for an element x in level $i < j$ and inserting it on the next level does not change $\widehat{C}_j[x]$. This means that $\widehat{C}_j[x]$ changes only when we insert an item x from the input stream into C_0 or when we decrement the count of an element in level j. Thus, as in the MG algorithm, $C_j[x]$ is only incremented when x occurs in stream, and is decremented only when the counts for $r^j M$ other elements are also decremented. The first inequality follows from this and the MG analysis. The second inequality follows from the first, and the fact that $r^{L-1} M \geq 1/\varepsilon$. □

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 I/O analysis, we assume that each level of the external-memory MG structure is implemented as a cascade filter [15].

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

Proof. A flush from level i to level $i + 1$ in a cascade filter is implemented by scanning both both levels, which can be done in $O(r^{i+1} M/B)$ I/Os. Each such flush moves at least $r^i M$ stream elements down one level, so the amortized cost to move one stream element down one level is $O(\frac{r^{i+1} M}{B} / (r^i M)) = O(r/B)$ I/Os. Each stream element can be moved down at most L levels. Thus, the overall amortized I/O cost is $O(rL/B) = O((r/B) \log_r(1/(\varepsilon M)))$, which is minimized at $r = e$. □

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

3.2 Misra-Gries LERT

We extend our external-memory MG data structure to support immediate reporting. That is, we show that for a threshold ϕ that is sufficiently large, it 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 $\widehat{C}_{L-1}[x]$, along with a second count $\mathcal{D}_0[x]$ that stores the number of occurrences of x needed to hit reporting threshold $\lceil(\phi - \varepsilon)N\rceil$. We set $\mathcal{D}_0[x]$ appropriately whenever we compute $\widehat{C}_{L-1}[x]$ without reporting x. When an instance of x arrives, $C_0[x]$ is incremented as in external-memory MG, and if the search bit is set, then we also decrement $\mathcal{D}_0[x]$; if a decrement of $\mathcal{D}_0[x]$ causes it to become zero then we report x. As in our external-memory MG structure, if the count for an entry $C_0[x]$ becomes 0, we delete that entry (along with its metadata). 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.

Whenever we report a item x, we set the "reported" bit in $C_0[x]$. Whenever we flush an item from level i to level $i+1$, we set the bit for that item on level $i+1$ if it is set on level i. When we delete the entry for a item that has the bit set on level $L-1$, we add an entry for that item on a new level C_L. This new level contains only items that have already been reported. When we are checking whether to report a item during a query, we stop checking further and omit reporting as soon as we reach a level where the bit is set.

I/O complexity. 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 $O\big(\big(\frac{M}{(\phi M-1)N}\big)\log\frac{1}{\varepsilon M}\big)$. □

Exact reporting. 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.*

Remark 1. *The following example shows that the analysis of the Misra-Gries LERT is asymptotically tight. In particular, when the RAM threshold is reached for an item (its count in RAM reaches $(\phi -$*

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

$1/M)N$), then the item's counts are spread across all L levels of the data structure, requiring a full sweep to consolidate its count and report; and that the total number of such queries can be $\Omega(M/(\phi M - 1)))$.

Let $N = 2^\ell M$ and $r = 2$, so the Misra-Gries LERT has $L = \ell + 1$ levels. Let the threshold $\phi N = 2^\ell + 1$, which satisfies the condition that $\phi N > N/M$, and let $\varepsilon = 1/N$. Consider the stream S defined below.

$$\begin{aligned} S = (&a, x_{11}, x_{12}, \ldots, x_{1M}, \\ &a, x_{21}, x_{22}, \ldots, x_{2M}, \\ &a, x_{31}, x_{32}, \ldots, x_{3M}, \\ &\ldots \\ &a, x_{k1}, x_{k2}, \ldots, x_{kM}, \\ &a, a, y_1, \ldots, y_{M-2^\ell-1}) \end{aligned}$$

where $k = 2^\ell - 1$, all the x_{ij}'s and y_i's are all distinct and not equal to item a. For this stream, every $(2^i M + 1)^{th}$ unique element causes a decrement to the count of all items at the ith level, pushing instances of item a down to level $i + 1$. When item a reaches the reporting threshold of ϕN during the last phase, its instances occur all the way down to the last level in the Misra-Gries LERT.

Thus, when the instance of a that triggers a report enters the system, we must collect at least one instance of a from every single level to recognize the need to report. Furthermore, every unique element of the stream triggers a query in RAM (since the RAM threshold is $\phi N - N/M = 1$), and there are $\Omega(N)$ such queries.

Summary. The Misra-Gries LERT supports a throughput at least as fast as optimal write-optimized dictionaries [12, 14, 16, 25–27], 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.

The external-memory MG algorithm helps us solve the TED problem. The smallest MG sketch (which fits in memory) is the most important estimator here, because it serves to sparsify queries to the rest of the structure. When such a query gets triggered, we need the total counts from the remaining $\log \frac{N}{M}$ levels for the (exact) online event-detection problem but only $\log \frac{1}{\varepsilon M}$ levels when approximate thresholds are permitted. In the next two sections, we exploit other advantages of this cascading technique to support much lower ϕ without sacrificing I/O efficiency.

4 TIME STRETCH

The MG LERT described in Section 3.2 reports events immediately, albeit at a high amortized I/O cost to perform queries to recognize the need for reporting. 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.

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.

4.1 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.

Data structure layout. We split the table at each level i into $c = 1 + \lceil 1/\alpha \rceil$ equal-sized ***bins*** $b_1^i, \ldots, b_c^i$, each of size $r^i M/c$. 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 $r^i M/c$ items, each with count 1, or 1 item with count $r^i M/c$, or any other such combination. See Figure 1.

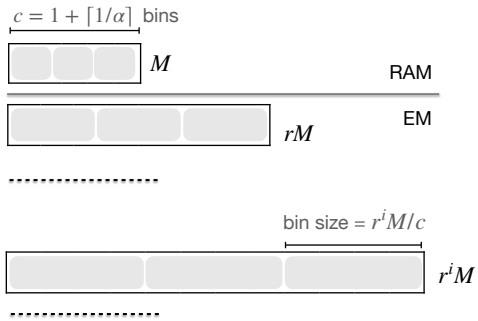


Fig. 1. A depiction of bins at each level of the Time-stretch LERT; α is the time-stretch parameter. EM stands for external memory. All bins are equal sized.

Flushing schedule. 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). The scheduling algorithm is described below.

- Let $b_1^i, \ldots, b_c^i$ be the bins (in order) on level i, where level 0 is RAM, and $0 \le i \le L-1$.
- Each stream item is inserted into b_1^0, the first bin in RAM.
- Whenever a bin b_1^i becomes full, we shift all the bins on level i over by one, that is, we move the contents of bin b_j^i to the adjacent bin b_{j+1}^i. The elements of the last bin at level i, b_c^i, are moved to b_1^{i+1}, the first bin on the next level.

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 shift and flush the last bin on level $i+1$ and so on.

Count consolidation. Finally, during a flush involving levels $0, \ldots, i$, where $i \le L-1$, 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.

4.2 Analysis of Time-stretch LERT

Correctness. We show that our data structure guarantees time stretch.

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

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

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 $c-1$ bins of size $r^i M/c$ on that level before being inserted to level ℓ, where $c = 1 + \lceil 1/\alpha \rceil$. This is dominated by waiting time on level $\ell - 1$. That is,

$$F_t \geq (c-1) \cdot \frac{r^{\ell-1}M}{c} \tag{1}$$

Level ℓ participates in a flush again after $\frac{r^{\ell-1}M}{c}$ inserts, which is the number of observations that fill up a bin on level $\ell - 1$. Using Eq(1), we get that $\frac{r^{\ell-1}M}{c} \leq \frac{F_t}{c-1} = \frac{F_t}{\lceil 1/\alpha \rceil} \leq \alpha F_t$. Thus, s_t is reported at most αF_t time steps after t. □

I/O complexity. For the analysis, we treat each level as a sorted array.

Theorem 5. *Given a stream of size N and parameters $\varepsilon, \phi \in (1/M, 1)$ 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.*

Proof. A flush from level i to $i+1$ costs $O(\frac{r^{i+1}M}{B})$ I/Os, and moves $r^i M/c$ stream items down one level, where $c = 1 + \lceil 1/\alpha \rceil$. Thus, the amortized cost to move one stream item down one level is $O(\frac{r^{i+1}M}{B} / \frac{\alpha}{\alpha+1} r^i M) = O(\frac{\alpha+1}{\alpha}\frac{r}{B})$ I/Os.

Each stream item can be moved down at most L levels, thus the overall amortized I/O cost of an insert is $O(\frac{\alpha+1}{\alpha}\frac{rL}{B}) = O\left(\frac{\alpha+1}{\alpha}\frac{r}{B}\log_r \frac{1}{\varepsilon M}\right)$, which is minimized at $r = e$. □

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.*

4.3 Implementation of time-stretch LERT

We implement each level in the time-stretch LERT as an exact counting quotient filter [57]. In addition to the count, we store a few additional bits with each item to keep track of its age.

Algorithm 1 Time-stretch LERT flush schedule

```
1:  max_age ← 2^num_age_bits                          ▷ num_age_bits is the number of bits to represent the age of a level
2:  flush_size ← levels[0].size/max_age               ▷ flush_size is the number of observations after which a flush is invoked
3:  procedure NEED_FLUSH(num_obs)
4:      if num_obs mod flush_size = 0 then
5:          return TRUE
6:      return FALSE
7:
8:  procedure FIND_NUM_LEVELS
9:      for i in 1 . . . (TOTAL_LEVELS − 1) do        ▷ r is the ratio of the sizes of level i + 1 and i.
10:         if levels[i].flushes < r − 1 then ▷ levels[i].flushes is the number of flushes from level i − 1 since the last flush to level i + 1
11:             break
12:     return i
13:
14: procedure PERFORM_MERGE_IF_NEEDED
15:     if NEED_FLUSH(num_obs) then
16:         num_levels ← FIND_NUM_LEVELS()
17:         for i in 1 . . . num_levels do            ▷ Increment the age for each level involved in the flush before the flush
18:             levels[i].age ← (levels[i].age + 1) mod max_age   ▷ levels is the array of levels
19:         MERGE(num_levels)
20:         for i in 1 . . . num_levels do            ▷ Increment the flush counter for each level involved in the flush
21:             levels[i].flushes ← (levels[i].flushes + 1) mod r
```

Algorithm 2 Time stretch LERT flush

```
1:  procedure IS_AGED(age, level)                ▷ If the age of an item is the same as the level age then it is ready to be flushed
2:      if age = level.age then
3:          return true
4:      return false
5:
6:  procedure MERGE(nlevels)                     ▷ nlevels is the number of levels involved in a flush
7:      elem ← −∞
8:      while true do
9:          elem ← MERGE_STEP(nlevels, elem)
10:         if elem = ∞ then
11:             break
12:
13: procedure MERGE_STEP(nlevels, prev)          ▷ Aggregate count of the smallest key across nlevels and flush it
14:     min ← ∞
15:     count ← 0
16:     age ← 0
17:     level ← 0
18:     next ← [ ]                               ▷ an array of {key, age, level, count}
19:     for i in 0 . . . nlevels do
20:         next[i] ← levels[i].SUCC(prev)       ▷ SUCC returns the smallest key larger than prev in level i
21:         if min > next[i] then
22:             min ← next[i]
23:     if min = ∞ then
24:         return ∞
25:     for i in 0 . . . nlevels do
26:         if min = next[i] then    ▷ Aggregate count. Record the lowest level where the key is and the age of the key on that level
27:             count ← count +next[i].count
28:             level ← i
29:             age ← next[i].age
30:             levels[i].DELETE(min)            ▷ DELETE remove key from level i
31:     if count >= T then
32:         REPORT(min)                          ▷ Report key
33:     else
34:         if level < nlevels and IS_AGED(age, level) then   ▷ If key is aged at level move it to the next level
35:             l_age ← levels[level + 1].age
36:             levels[level + 1].INSERT(min, count, l_age)   ▷ key gets the current age of the level
37:         else
38:             levels[level].INSERT(min, count, age)         ▷ Else insert the key at level with aggregate count
39:     return min
```

In the time-stretch LERT, each level is split into $c = 1 + \lceil 1/\alpha \rceil$ 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 $\lceil \log c \rceil$ 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. Before a flush, the age of each level involved in the flush is incremented. The age of a level wraps 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 the same as the level's age, then the item has survived c flushes on that level, and is therefore eligible to flush. When an item is inserted in a level, its age is set to the level's age. However, if the level already has an instance of the item, then we just increment the count of the existing instance whatever its age.

We follow a fixed schedule for flushes. We trigger a flush after every group of $(\frac{\alpha}{\alpha+1})M$ stream observations. Every r^ith flush, level i flushes to level $i + 1$. That is, after every $r - 1$ flushes to level $i \geq 1$, level $i + 1$ is involved in the next (rth) flush. To determine the number of levels involved in a flush, we maintain a counter per level for the number of times level ℓ has been involved in a flush from $\ell - 1$ since its last flush to $\ell + 1$. Algorithm 1 shows the pseudocode for the flush schedule of the levels in a time-stretch LERT.

Note that only "eligible items" are flushed down a level during these flush operations, in particular, items that have aged enough to be in the last bin at a level—or equivalently, items whose age is equal to the level's age.

Consolidating item-counts during a flush is implemented as a k-way merge sort. We first aggregate the count of an item across all k 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. Algorithm 2 shows the pseudocode for flushing items in a time-stretch LERT. We use $T = \phi N$ to denote the reporting threshold in the implementation.

Summary. By allowing a little delay, we can solve the timely event-detection problem at the same asymptotic cost as simply indexing our data [12, 14, 16, 25–27].

Recall that in the online solution the increments and decrements of the MG algorithm determined the flushes from one level to the other. In contrast, these flushing decisions in the time-stretch solution are based entirely on the age of the items. The MG style count estimates came essentially for free from the size and cascading nature of the levels. Thus, we get different reporting guarantees depending on whether we flush based on age or count.

Our experimental results for 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. [17] 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 $\theta = 1 + 1/\alpha$ [1]. They show that for Zipfian distributions with $\alpha > 1$ (power-law distributions with $\theta \leq 2$), the MG algorithm can solve the approximate heavy-hitter problem with error ε 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. The error bound ε^{α} is in fact stronger as it can be applied to the tail frequency of the stream, rather than the whole stream. In particular, if c_i is the true count of item i, and $\tilde{c}_i$ is the estimate, then on zipfian distributions with $1/\varepsilon$ space, $c_i - \tilde{c}_i \leq \varepsilon^{\alpha} N_{\text{tail}}$, where N_{tail} is the sum of counts of all keys except the top-$(1/\varepsilon)$ most frequent keys [17]. Our Misra-Gries LERT data structure based on the MG algorithm automatically inherits these improved bounds.

We give improved results for power-law streams with exponent $\theta > 2$, a range which is representative of power-law distributions observed in practice [55]. In Section 5.1, we study the *exact* TED problem and design algorithms tailored for such a distribution. In Section 5.2, we present a data structure that has improved I/O performance and guarantees a count-dependent bounded delay.

Preliminaries. We use the continuous power-law definition [55]: 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]

$$1 = \int_1^\infty p(x)\,dx = Z \int_1^\infty x^{-\theta}\,dx = \frac{Z}{\theta - 1}\left[\frac{-1}{x^{\theta-1}}\right]_1^\infty = \frac{Z}{\theta - 1}.$$

Thus, $Z = (\theta - 1)$.[4]

We use the cumulative distribution of a power law:

$$\text{Prob}\ (x > c) = \int_{j=c}^\infty (\theta - 1)x^{-\theta}dx = \frac{1}{c^{\theta-1}}. \tag{2}$$

For our analysis, we assume that the input stream S is constructed offline as follows. Let U denote the number of distinct keys in the stream S. The count for each key is drawn independently from a power-law distribution. Then the instances of the keys in S are ordered arbitrarily. That is, we do not make any assumptions on the arrival order of keys. Next, we analyze some properties of the input stream.

Lemma 6. *In the input stream with U distict keys, where the count of each key is drawn independently from a power-law distribution with $\theta > 2$, the following holds with high probability with respect to U:*

(1) the number of keys with count greater than c is $\frac{U}{c^{\theta-1}}$;

(2) the size of the stream $N = \left(\frac{\theta-1}{\theta-2}\right)U$.

Proof. Let x_{ic} denote the indicator random variable which is 1 if key i has count greater than c and 0 otherwise. Let $X_c = \sum_{i=1}^U x_{ic}$. Then $\mathbb{E}[X_c] = \mathbb{E}[\sum_{i=1}^U x_{ic}] = \sum_{i=1}^U \mathbb{E}[x_{ic}] = \sum_{i=1}^U \text{Prob}\ (x_{ic}) = \sum_{i=1}^U \frac{1}{c^{\theta-1}} = \frac{U}{c^{\theta-1}}$. This also holds with high probability with respect to U using Chernoff bounds. This proves (1) in the above lemma.

Next, let y_{ic} be the random variable denoting that key i has count c. Let $Y_c = \int_{i=1}^U y_{ic}$ and $Y = \int_{c=1}^\infty Y_c dc$. Then $\mathbb{E}[Y] = \int_{c=1}^\infty \mathbb{E}[Y_c]dc = \int_{c=1}^\infty \int_{i=1}^U \mathbb{E}[y_{ic}]dc = \int_{c=1}^\infty \int_{i=1}^U c \cdot \text{Prob}\ (y_{ic})dc = U\int_{c=1}^\infty c(\theta - 1)c^{-\theta}dc = U\frac{\theta-1}{\theta-2}[-c^{2-\theta}]_{c=1}^\infty = U\frac{\theta-1}{\theta-2}$.

The result holds with high probability with respect to U using a Chernoff bound argument. □

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 merge*** and ***immediate-reporting query***. Finally we analyze its correctness and I/O performance.

Data structure layout. Similar to the data structures in the previous sections, the immediate-report LERT consists of a cascade of tables, where M is the size of the table in RAM. There are $L = \log_r(N/M)$ levels on disk, where N is the size of the stream. 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 and is based on the level's size. In particular, each level i in the immediate-report LERT has a ***level threshold*** τ_i for $1 \le i \le L$, $(\tau_1 \ge \tau_2 \ge \ldots \ge \tau_L)$, where τ_i indicates the maximum count of a key that can be stored on level i.

[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 $\theta > 1$.

[4] In principle, one could have power-law distributions with $\theta < 1$, but these distributions cannot be normalized and are not common [55].

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 the τ_i's based on the input stream's 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, we find the smallest level $j > i$ that has enough empty space to hold all items from levels $0, 1, \ldots, j$.
- We aggregate the count of each item k on levels $0, \ldots, j$, resulting in a consolidated count c_k^j.
- If $c_k^j \geq \phi N$, (and we have not reported it before[5]) we report k. Otherwise, we distribute instances of k in a bottom-up fashion on levels j to 0, while maintaining the threshold invariant. 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 $0 \leq y \leq j-1$.

In the above algorithm, notice that items can end up in higher levels (compared to the level they were before), which is why we call this operation a shuffle-merge instead of a merge. Also, observe 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 $\sum_{i=\ell+1}^{L} \tau_i$.

To maintain efficient shuffle merges, the number of packed items at a level should not occupy more than a constant fraction of the size of the level. In Lemma 7, we show that given a power-law stream with exponent θ, we can set the thresholds based on θ so as to satisfy this requirement.

Lemma 7. *Let the counts 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 at level i is at most $\frac{\theta-2}{\theta-1}$ times the size of level i.*

Proof. We prove by induction on the number of levels. We start at level $L-1$. An item is packed at level $L-1$ if its count is greater than $\tau_L = r^{\frac{1}{\theta-1}}$. By Lemma 6 (1), there are $U/\tau_L^{\theta-1} = U/r$ such items. By Lemma 6 (2), the size of the stream $N = \left(\frac{\theta-1}{\theta-2}\right)U$. The size of level $L-1$ is $N/r = \left(\frac{\theta-1}{\theta-2}\right)\frac{U}{r}$. Thus, number of items packed at level $L-1$ is $\frac{\theta-2}{\theta-1}$ times the size of level $L-1$.

Suppose the lemma holds for level $i+1$. We show that it holds for level i. An item is packed at level $i+1$ if its count is greater than $\sum_{\ell=i+2}^{L} \tau_\ell$. Using Lemma 6 (1), and the induction hypothesis, the expected number of such items is

$$\frac{U}{\left(\sum_{\ell=i+1}^{L} \tau_\ell\right)^{\theta-1}} < \frac{U}{{\tau_{i+2}}^{\theta-1}} \leq \left(\frac{\theta-2}{\theta-1}\right)\frac{N}{r^{L-i-1}}. \tag{3}$$

Finally, an item is packed at level i if its count is greater than $\sum_{\ell=i+1}^{L} \tau_\ell$. Using Lemma 6 (1) and Inequality (3), the expected number of items packed at level i is

$$\begin{aligned}
&\frac{U}{\left(\sum_{\ell=i+1}^{L} \tau_\ell\right)^{\theta-1}} < \frac{U}{\left({\tau_{i+1}}^{\theta-1}\right)} = \frac{U}{\left(r^{\frac{1}{\theta-1}} \cdot \tau_{i+2}\right)^{\theta-1}} \\
&= \frac{1}{r} \cdot \frac{U}{\left({\tau_{i+2}}^{\theta-1}\right)} \leq \frac{1}{r}\left(\frac{\theta-2}{\theta-1}\right)\frac{N}{r^{L-i-1}} = \left(\frac{\theta-2}{\theta-1}\right)\frac{N}{r^{L-i}}.
\end{aligned}$$

□

Immediate reporting. As soon as the count of an item k in RAM reaches a threshold of $\phi N - \sum_{i=1}^{L} \tau_i$, the data structure triggers an ***immediate-reporting query***, which sweeps all L levels, consolidates

[5]Each reported item is stored in a separate table in RAM to avoid duplicate reporting of events.

the counts of k at all levels into RAM and reports if the consolidated count reaches threshold $T = \phi N$. Reported items are remembered, so that each event is reported exactly once.

Analysis. Next, we prove correctness of the immediate-report LERT and analyze its I/O complexity. We set $r = e$, which minimizes the insertion cost (in Theorem 9).

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

Lemma 8. *Let S be a stream of size N where the item counts follow a power-law distribution with exponent $\theta > 2$. The immediate-report LERT solves the TED problem with immediate reporting on S with high probability.*

Proof. Let $\tilde{c}_i$ denote the count estimate of key i in RAM in the immediate-report LERT. Because of the threshold invariant at most $\sum_{\ell=1}^{L} \tau_\ell$ instances of a key can be stored on disk at any time.

Suppose $\tilde{c}_i$, the count in RAM for key i, is incremented to the search threshold $T_s = \phi N - \sum_{\ell=1}^{L} \tau_\ell$ at time t. This triggers an immediate-report query. The counts from all levels of the disk are added to $\tilde{c}_i$ to give an accurate count c_i. Because of the threshold invariants, we have $c_i \leq \phi N - \sum_{\ell=1}^{L} \tau_\ell + \sum_{\ell=1}^{L} \tau_\ell = \phi N$. If $c_i = \phi N$, then we report item i at time t, exactly as its count reaches the reporting threshold. Otherwise, the system sets a bit to indicate the count includes all occurences of the key in the data structure; when this (accurate) count c_i reaches ϕN, it is reported immediately. □

Next, we analyze the I/O complexity of the immediate-report LERT. Similar to Section 3.2, we assume the levels of the immediate-report LERT are implemented as a cascade filter [15].

Theorem 9. *Let S be a stream of size N where the item counts follow a power-law distribution with exponent $\theta > 2$. Then the immediate-report LERT can solve the TED problem on S w.h.p. for thresholds $\phi N > \gamma$, where $\gamma = \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1} \cdot \left(\frac{N}{M}\right)^{1/(\theta-1)}$. The amortized I/O complexity of the data structure is $O\left(\left(\frac{1}{B} + \frac{1}{(\phi N-\gamma)^{\theta-1}}\right) \log \frac{N}{M}\right)$ per stream item.*

Proof. During a shuffle-merge at level i, the items that are not packed are flushed down to level $i+1$ incurring an I/O cost of $O(r^{i+1}M)$. We can charge this to $r^i M(1 - \frac{\theta-2}{\theta-1})$ unpacked items that get flushed down, and the amortized cost is $O(r(\theta-1)/B)$. This can happen at most L time and thus the amortized insert cost is $O(rL(\theta-1)/B)$. This cost is minimized at $r = e$.

We perform at most one query each time an item's count in RAM reaches $c = \phi N - \sum_{\ell=1}^{L} \tau_\ell$. We upper bound the total number of items in the stream that have count at least c, given that $r = e$.

$$\sum_{\ell=1}^{L} \tau_\ell = \sum_{\ell=1}^{L} e^{\ell/(\theta-1)} = e^{1/(\theta-1)} \left(\frac{e^{L/(\theta-1)} - 1}{e^{1/(\theta-1)} - 1}\right) \leq \frac{e^{(L+1)/(\theta-1)}}{e^{1/(\theta-1)} - 1} = \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)} - 1} \cdot \left(\frac{N}{M}\right)^{1/(\theta-1)} = \gamma$$

Thus, $c = \phi N - \sum_{\ell=1}^{L} \tau_\ell \geq \phi N - \gamma$. The total number of items in the stream with count at least c is $\frac{U}{c^{\theta-1}}$ by Lemma 6 (1). Using the lower bound on c, we get that $\frac{U}{c^{\theta-1}} \leq \frac{U}{(\phi N-\gamma)^{\theta-1}}$.

A query in a cascade filter costs $O(L)$ I/Os as it requires one I/O per level. Thus, the overall amortized I/O complexity of the queries over N elements is $O\left(\frac{UL}{N(\phi N-\gamma)^{\theta-1}}\right) = O\left(\left(\frac{(\theta-2)L}{(\theta-1)(\phi N-\gamma)^{\theta-1}}\right)\right.$.

Putting it all together, substituting $L = \log_e \frac{N}{M}$ and ignoring multiplicative factors of θ, we conclude that the amortized I/O complexity of the data structure is $O\left(\left(\frac{1}{B} + \frac{1}{(\phi N-\gamma)^{\theta-1}}\right) \log \frac{N}{M}\right)$. □

Remark 2. *The relationship between the reporting threshold ϕN and the power-law exponent θ identified in Theorem 9 for the immediate-report LERT is a generalization of the relationship presented in [58], which provides a weaker bound.*

Supporting smaller reporting thresholds for power-law streams. The immediate-report LERT—tailored for power-law streams—lets us support smaller reporting-thresholds ϕN compared to the Misra-Gries LERT for immediate reporting (in particular, the reporting threshold in Corollary 1). To see why this is true, notice that γ the lower bound on ϕN in Theorem 9, consists of two parts multiplied together: the first term $\frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1}$ depends only on θ, and for the range-of-interest $2 < \theta < 3$, this term is a small constant. Specifically, for $\theta < 2.96$, $\frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1} < 2.5$. The second term $\left(\frac{N}{M}\right)^{1/(\theta-1)}$ decreases exponentially as θ increases. Conversely, the reporting threshold ϕN that Misra-Gries LERT can support must be at least as large as $\frac{N}{M}$.

Thus, under reasonable conditions on N, M and θ, the immediate-report LERT can support smaller reporting thresholds for immediate reporting than previous data structures. For example, when $\theta > 2 + 1/(\log_t(N/M))$, where $t = \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1}$, we have $\gamma = \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1} \cdot \left(\frac{N}{M}\right)^{1/(\theta-1)} < \frac{N}{M}$.

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 a TED 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 (including reporting during 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 give a key a count of ϕN quickly, with unfortunate shuffle-merge timing moving the maximum number of occurences to disk before the RAM count hits ϕN. It could take much longer to get from the ϕNth occurrence to the $(1+\omega)\phi N$th occurrence.

Theorem 10. *Let S be a stream of size N where the item counts follow a power-law distribution with exponent $\theta > 2$, and let parameters ϕ, ω be such that $\phi N \cdot \omega > \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1}\left(\frac{N}{M}\right)^{\frac{1}{\theta-1}}$. Then the count-stretch LERT solves the TED problem on S w.h.p. with count stretch $1 + \omega$ at amortized I/O cost $O(\frac{1}{B}\log\frac{N}{M})$ per stream item.*

Proof. The amortized I/O complexity of the count-stretch LERT follows from the insertion cost of Theorem 9, without the expensive immediate-reporting queries. Recall that the insertion cost is minimized by setting $r = e$.

For a count stretch of $1 + \omega$ it is sufficient to show that when an item hits a count of ϕN in RAM, there are at most $\omega\phi N$ occurrences of that item stored in the lower levels of the data structure on disk.

By the threshold invariant of the count-stretch LERT, we can bound the total occurrences of an item in levels $1, 2, \ldots L$ on disk as $\sum_{i=1}^{L} \tau_i$. Below, we show that this quantity must be at most $\omega\phi N$.

For $r = e$, we can upper bound this sum as follows:

$$\sum_{\ell=1}^{L} \tau_\ell = \sum_{\ell=1}^{L} e^{\ell/(\theta-1)} = e^{1/(\theta-1)}\left(\frac{e^{L/(\theta-1)}-1}{e^{1/(\theta-1)}-1}\right) \leq \frac{e^{(L+1)/(\theta-1)}}{e^{1/(\theta-1)}-1} = \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1} \cdot \tau_1$$

Since $\tau_1 = \left(\frac{N}{M}\right)^{1/(\theta-1)}$, it follows that there can be at most $\frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1}\left(\frac{N}{M}\right)^{1/(\theta-1)}$ occurrences of an item stored on disk at any time. Thus, when the count estimate of an item in RAM reaches ϕN, its true count is at most $\phi N + \frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1}\left(\frac{N}{M}\right)^{1/(\theta-1)} < \phi N(1+\omega)$. □

Remark 3. *For power-law exponent $2 < \theta < 3$, a range that is typically observed in practice [1, 17, 24, 31, 55], the term $\frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1}$ in Theorem 10 is a small constant. For example, for $\theta < 2.96$, $\frac{e^{1/(\theta-1)}}{e^{1/(\theta-1)}-1} < 2.5$, and thus, the count-stretch LERT can support parameters ϕ, ω such that $\phi N \cdot \omega > 2.5\left(\frac{N}{M}\right)^{\frac{1}{\theta-1}}$.*

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 the thresholds to ensure that no level of the data structure has too many packed items. In particular, to satisfy Lemma 7, for $\theta < 3$, it is sufficient to ensure that the number of items packed at any level i does not exceed $\frac{\theta-2}{\theta-1} < \frac{1}{2}$ 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-1} to the minimum value such that the number of keys packed at level j 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 3.1), 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.

5.3 Implementation of 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 time-stretch LERT, each level is an exact counting quotient filter [57]. In the count-stretch LERT, in addition to the count of each key, we store a few additional bits to mark whether an item has its absolute count at a level (its aggregate count across all the levels).

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. Algorithm 3 shows the pseudocode for the shuffle-merge in a count-stretch LERT.

Optimization. We also implement an optimization in the count-stretch LERT that further reduces I/O costs by following a "greedy" flushing schedule instead of a fixed schedule. This is based on the observation that unlike time stretch, the count stretch does not depend on the number of observations in the stream. Therefore, we do not need to perform shuffle merges at regular intervals. We only invoke a shuffle-merge if it is needed, i.e., when the RAM is at capacity. The greedy flushing optimization is implemented as an additional input flag which can be turned on or off.

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. This is the case for streams such as the one from Firehose, where counts have a power-law distribution. The greedy shuffle-merge schedule avoids unnecessary I/Os that a fixed schedule would incur during shuffle-merges.

Algorithm 3 Count-stretch LERT shuffle-merge

```
1:  procedure SHUFFLE_MERGE(nlevels, τ[])                ▷ τ[] is an array of all τ values for levels
2:      elem ← −∞
3:      while true do
4:          elem ← SHUFFLE_MERGE_STEP(nlevels, τ, elem)
5:          if elem = ∞ then
6:              break
7:
8:  procedure SHUFFLE_MERGE_STEP(nlevels, τ[], prev)   ▷ Aggregate count of the smallest key across nlevels and distribute
9:      min ← ∞
10:     next ← []                                        ▷ an array of {key, age, level, count}
11:     for i in 0 ...nlevels do
12:         next[i] ← levels[i].SUCC(prev)               ▷ SUCC returns the smallest key larger than prev in level i
13:         if min > next[i] then
14:             min ← next[i]
15:     if min = ∞ then
16:         return ∞
17:     count ← 0
18:     for i in 0 ...nlevels do
19:         if min = next[i] then
20:             count ← count + next[i].count
21:             levels[i].DELETE(min)                    ▷ DELETE remove key from level i
22:     if count >= T then
23:         REPORT(min)                                  ▷ Report key
24:     else
25:         for i in nlevels ...0 do                     ▷ This is a reverse loop
26:             if count ≥ τ[i] then
27:                 levels[i].INSERT(min, τ[i])          ▷ Insert key in level i with count τ[i]
28:                 count ← count −τ[i]
29:             else if count > 0 then
30:                 levels[i].INSERT(min, count)         ▷ Insert key in level i with count count
31:             else
32:                 break
33:     return min
```

As explained in Section 5.1, 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 aggregate the counts. If the aggregate count in RAM and on disk is T we report the item. Otherwise we insert the aggregate count in RAM and set a bit, ***the absolute bit***, that indicates that all the counts for the item have 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 are garbage collected during the next shuffle merge.

6 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 7.

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

Proof. The count stretch of an item in a count-stretch LERT depends only upon the item's final count when it is reported. This final count is independent of the rest of stream. In the deamortized count-stretch LERT, all observations of an item go to a single cone, and each cone independently provides the same count stretch as the amortized count-stretch LERT for items mapped to that cone. □

Lemma 12. *There exists an input stream for which the deamortized time-stretch LERT provides no global time stretch guarantee.*

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 13. *Consider a random stream where each arriving 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.*

Proof. Suppose each item maps to cone i with probability η_i. Consider a key k that maps to cone i with its first appearance at index I_0 and its Tth occurence at index I_T. Let $L_k = I_T - I_0$. The time-stretch LERT without cones will report k by time (index) $T_D = T_0 + (1 + \alpha)L_k$. In the deamortized version, cone i receives $\eta_i L_k$ items between indices I_0 and I_T in expectation. So it will report k when another $\alpha\eta_i L_k$ items arrive at cone i. But cone i should receive that many items in the αL_k items after I_T. Thus we expect cone i to report k at time T_D. A similar argument holds when the stream is a random permutation of a finite stream with $\eta_i N$ elements from cone i. □

7 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. [57].

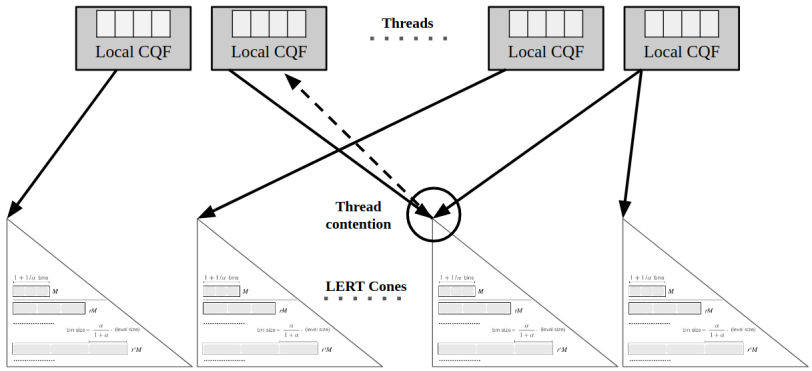


Fig. 2. A depiction of multi-threading with cones in a LERT.

We assign a small local insertion buffer to each thread. See Figure 2. 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. It then processes each observation in the packet 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, the thread dumps the items in the buffer into theirrespective cones. When dumping a buffer, the threads wait for the locks.

If a thread 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. This is because threads do not wait to acquire a lock on the cone for every insertion and continue to make progress. Also, item counts are consolidated in local buffers. Thus during the buffer dump, only one insertion for each item is required instead of multiple insertions for each instance of the same item. Our method scales well with increasing number of insertion threads even for streams with skewed distributions. We show this empirically in Section 8.8.

Using readers-writer locks at the cone level allows multiple threads to 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.

7.1 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 ith 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 each pull a chunk (usually 1024) of observations simultaneously. Therefore, multi-threading adds an extra delay to the timeliness guarantees of the time-stretch LERT and extra counts to the guarantees of the count-stretch LERT. We analyze this empirically in Section 8.4.

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 14. *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. Because the maximum count of an item in a thread's local buffer is $\frac{T}{P}$, for P threads the maximum count for any item is $\frac{T}{P} \times P = T$. An individual cone with count-stretch guarantee ω will report an item when it holds at most $(1+\omega)T$ instances of that item. Thus the maximum number of instances in the system at the time of the report is $(2+\omega)T$. □

Time stretch. It is harder to provide a time-stretch guarantee with multiple threads compared to the count-stretch guarantee. This is because time stretch depends on the arrival distribution of other items in the stream, while count stretch 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 a 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, dumping contents of the local buffer in cones, and performing a flush/shuffle-merge on the cone. As above, we constrain how long a thread can go before dumping its local buffer. Every thread has to dump its 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.

Observation 1. *In a multi-threaded time-stretch LERT in which each thread dumps its local buffer every t time steps, we guarantee that an item s is reported in at most $\alpha F_s + t$ additional time steps (after the item-count reaches T), where F_s is the flow time of s.*

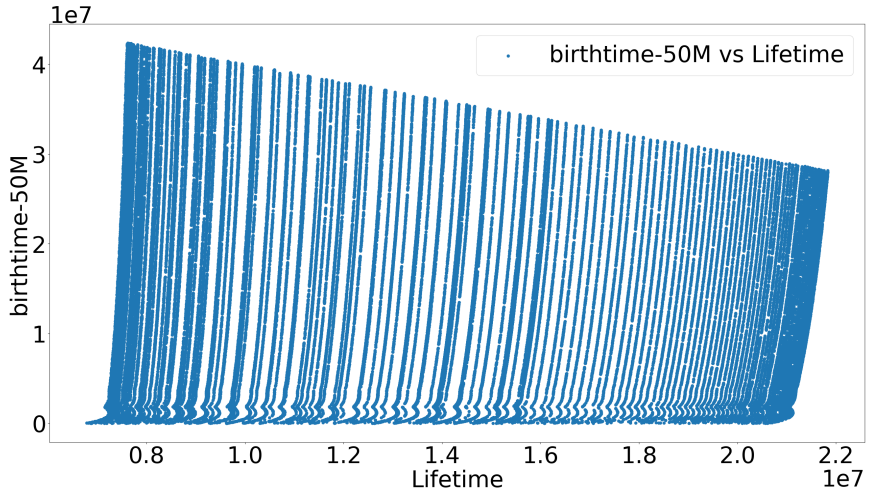


Fig. 3. Birthtime vs. the lifetime of each reportable item in the active-set generator dataset consisting of 50M observations.

8 EVALUATION

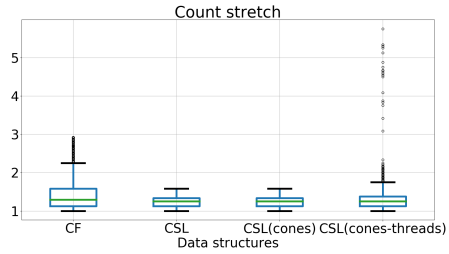
In this section, we evaluate our implementations of the time-stretch LERT (TSL), count-stretch LERT (CSL), and immediate-report LERT (IRL) for timeliness, robustness to input distributions, I/O performance, insertion throughput, and scalability with multiple threads. Our implementation is publicly available at https://github.com/splatlab/lerts.

We compare our implementations against Bender et al.'s cascade filter [15] 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.

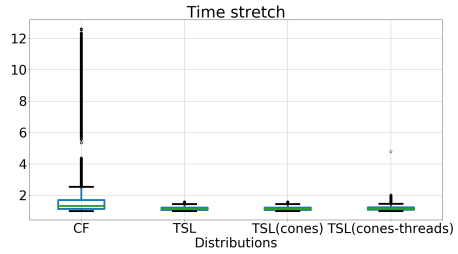
We also evaluate an implementation of the Misra-Gries data structure as a baseline for in-memory insertion throughput. We implement the Misra-Gries data structure with an exact counting data structure (counting quotient filter) to forbid false positives. This gives an upper bound on the insertion throughput 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:

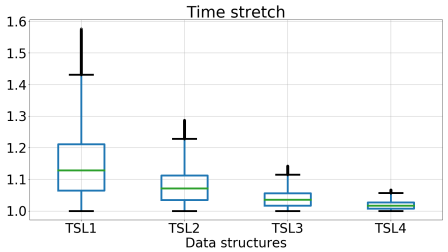
(1) How does the empirical timeliness of reported items compare to the theoretical bounds?
(2) How robust is the time-stretch LERT to different input distributions?
(3) How does deamortization and multi-threading affect the empirical timeliness of reported items?
(4) How does the buffering strategy affect 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?



(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.

Fig. 4. 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) $\boldsymbol{\alpha = 1}$, 2 (TSL2) $\boldsymbol{\alpha = 0.33}$, 3 (TSL3) $\boldsymbol{\alpha = 0.14}$, and 4 (TSL4) $\boldsymbol{\alpha = 0.06}$.

8.1 Experimental setup

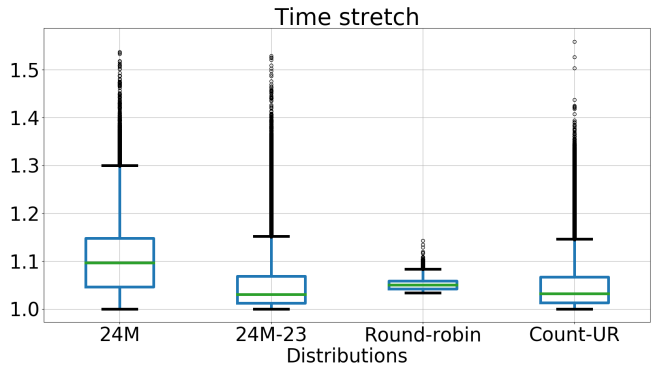
In this section, we describe how we designed experiments to answer the questions above and describe our workloads,

Our experiments fall into two categories: validation experiments and scalability experiments. The validation experiments require an offline analysis of the dataset to compute the lifetime and measure the stretch of every key to perform the validation. We use smaller datasets (64 million) for the validation experiments. For scalability experiments, we use bigger datasets (4 billion).

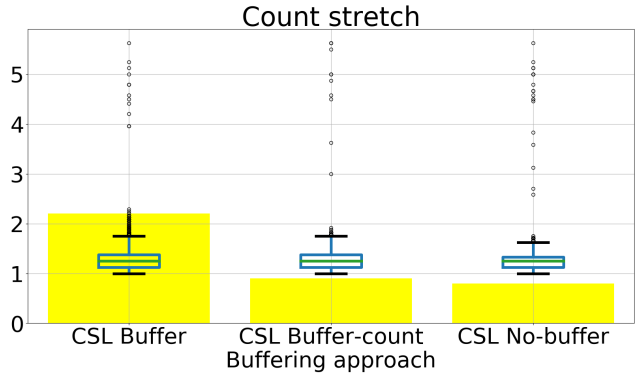
Workload. Firehose [5] is a suite of benchmarks simulating a network-event monitoring workload. A Firehose benchmark consists of a *generator* that feeds keys to the *analytic*, being benchmarked. The analytic must detect and report each key that has 24 observations.

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

Figure 3 shows the distribution of birthtime (the index of the first occurrence of an item) vs. the lifetime (number of observations between the first and the T-th occurrence) of items in the stream from active-set generator. The stream contains 50M observations and the active-set size is 1M.



(a) Distribution of time stretch for different distributions. These distributions are described in 8.1.

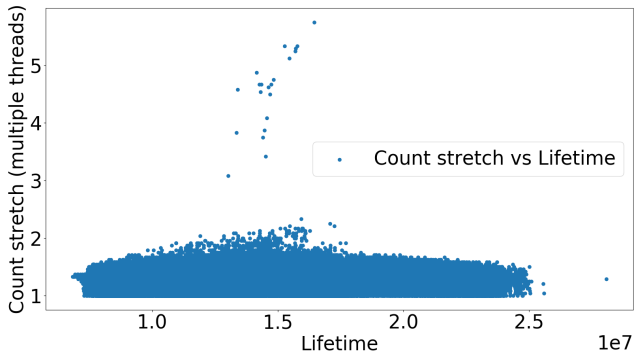


(b) Distribution of count stretch with different buffering strategies. Bars show the average insertion throughput (Million insertions/sec) for each buffering strategy. Average insertion throughput when no-buffer is used is 2.7× lower compared to when buffers are used.

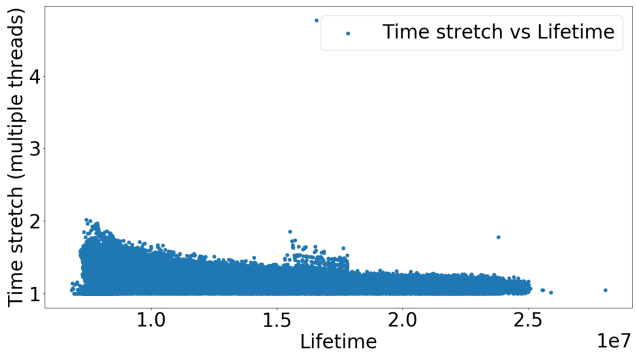
Fig. 5. Data structure configuration: RAM level: 8388608 slots in the CQF, levels: 4, growth factor: 4, level thresholds for on-disk level: (2, 4, 8), cones: 8, threads: 8, number of observations: 512M.

The longest lifetime is ≈ 22M. Whenever a new item is added to the active set it is assigned a count value from the set of counts based on the power-law distribution. Therefore, we see bands of items that have similar lifetime but are born at different times throughout the stream. The lifetime of items in these bands tend to increase slightly as the items are born later in the stream due to different selection probabilities of items from the active set. In all of our experiments we have used dataset from the active-set generator unless noted otherwise.

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.



(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.

Fig. 6. 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.

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 empirical stretch validation, 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

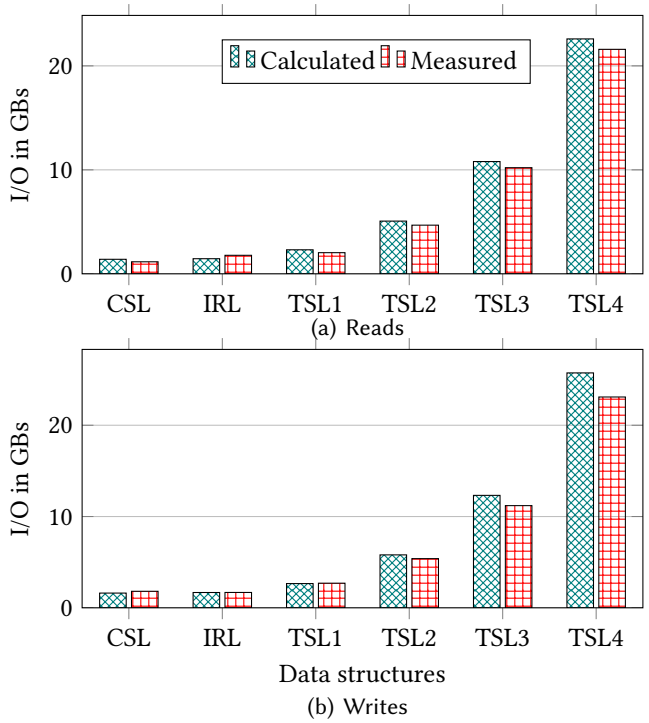


Fig. 7. Total I/O performed by the count-stretch, time-stretch and immediate report LERT. Data structure configuration: RAM level: 4194304 slots in the CQF, levels: 3, growth factor: 4, number of observations: 64M. Immediate-report LERT (IRL).

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.

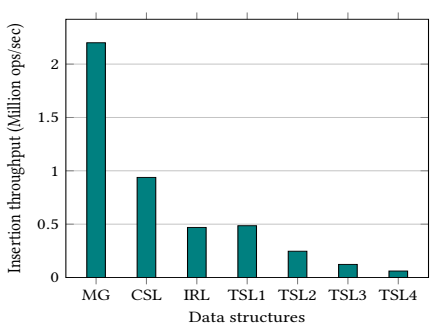
For all the experiments, we use a reporting threshold of 24 since it is the default in the Firehose benchmarking suite.

8.2 Timely reporting

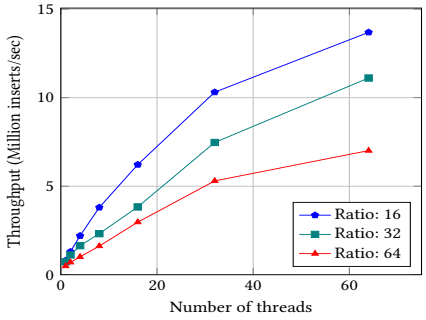
Cascade filter. Figures 4a and 4b show the distribution of count stretch and time stretch of reported items in the cascade filter. The cascade filter's maximum count-stretch is 3.0 and maximum time stretch is > 12, much higher than any single-threaded count-stretch or time-stretch LERT.

Count-stretch LERT. Figure 4a 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 4b 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 4c 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.



(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.

Fig. 8. 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.

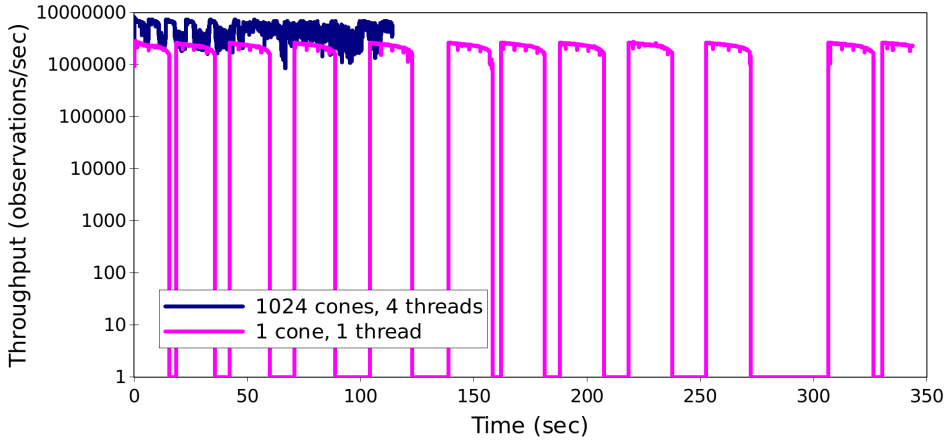


Fig. 9. Instantaneous throughput of the count-stretch LERT with 1 cone and 1 thread and 1024 cones and 4 threads. Same configuration as Figure 4a.

8.3 Robustness with input distributions

Figure 5a 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.

8.4 Effect of deamortization/threading

Figures 4a and 4b 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 8.1) 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 6).

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 6 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.

8.5 Effect of buffering

Figure 5b 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.

8.6 I/O performance and throughput

Figure 7 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 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 8a 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.

8.7 Instantaneous throughput

Figure 9 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.

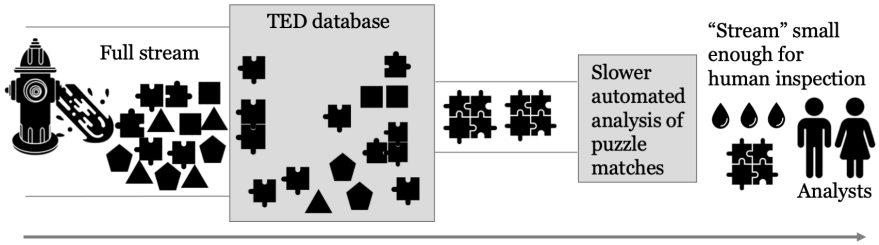


Fig. 10. The analysis pipeline that motivates our TED problem solution. Analysts associate a multi-piece pattern, represented by the 4-piece puzzle, to a high-consequence event. The pieces arrive slowly over time, mixed with innocent traffic in a high-throughput "firehose" stream. Our database stores many partial matches to the pattern reporting all instances of the pattern. There still may be a fair number of matches, which are pared down by an automated system to a small number (essentially droplets compared to the original stream) of matches worthy of human inspection.

8.8 Scaling with multiple threads

Figure 8b 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.

9 MOTIVATING NATIONAL SECURITY APPLICATION

In this section, we describe the more complex national-security setting that motivates our modeling constraints. Firehose [4, 5] is a clean benchmark that captures the fundamental elements of this setting. The TED problem in this paper in turn distills the most difficult part of the Firehose benchmark. Therefore our solutions have direct line of sight to important national-security applications.

An ideal solution for TED would have 1) no false negatives, 2) no false positives, 3) immediate reporting of a stream element that upon arrival hits the reporting threshold, and 4) speed sufficient to keep up with real sensor data streams. To better allow (1) and (4), in this paper we relax (2) and (3). Our algorithms limit false positives to keys that are "close" to reportable and bound reporting delay by either time or count. Our use case explains why we can tolerate these relaxations. It also explains why we can not relax the no-false-negative requirement. This critical aspect of the model means we cannot consider sampling-based or randomized algorithms for finding reportable items, since these can miss events.

We are motivated by monitoring systems for national security [4, 5], where experts associate special patterns in a cyberstream to rare, high-consequence real-life events. These patterns are formed by a small number of "puzzle pieces," as shown in Figure 10. Each piece is associated with a key such as an IP address or a hostname. The pieces arrive over time. When an entire puzzle associated with a particular key is complete, this is an event, which should be reported as soon as the final puzzle piece falls into place. In Figure 10, the first stage is like our TED problem algorithm, except that it must store puzzle pieces with each key rather than a count and the reporting trigger is a complete puzzle, not a count threshold.

There can still be a fair number of matches to this special pattern, most of which are still not the critically bad event. This might overwhelm a human analyst, who would then not use the system. However, automated tools, shown in the second stage of Figure 10, can pare these down to the few events worthy of analyst attention.

The first stage filter, like our TED problem solution, must struggle to handle a massively large, fast stream. It is reasonable to allow a few false positives in the first stage to improve its speed. The second stage can screen out almost all of these false positives as long as the stream is significantly reduced. The second stage is a slower, more careful tool which cannot keep up with the initial stream. This second tool cannot, however, repair false negatives since anything the first filter misses is gone forever. So the first tool cannot drop any matches to the pattern. Experts have gone to great effort to find a pattern that is a good filter for the high-consequence events. We do not allow false negatives because the high-consequence events that match this carefully crafted pattern can and must be detected.

Each of these patterns are small with respect to the stream size, so the detection algorithm must be scalable, that is, must be able to support a small threshold T. The consequences of missing an event (false negative) are so severe that it is not reasonable to risk facing those consequences just to save a little space. Thus we must save all partial patterns, motivating our use of external memory.

The ability to tolerate a reporting delay depends upon how much lead time the search pattern gives before possible damage. There will be some additional delay from the second-stage testing. Reports are still "better late than never." Even if some damage has occurred, the system operators still have significantly more information than they would have if they had received no report.

The DoD Firehose benchmark captures the essence of this setting [5]. In Firehose, the input stream has (key,value) pairs. When a key is seen for the 24th time, the system must return a function of the associated 24 values. The most difficult part of this is determining when the 24th instance of a key arrives. Thus, like Firehose, the TED problem captures the essence of the motivating application.

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 1947789, CCF 1725543, CSR 1763680, CCF 1716252, CCF 1617618, CNS 1938709, CCF 2106827, CCF 1715777, AitF 1637458, 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. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

This research is also funded in part by the Advanced Scientific Computing Research (ASCR) program within the Office of Science of the DOE, and resources of the NERSC supported by the Office of Science of the DOE under contract number DE-AC02-05CH11231, and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Image attributions for Figure 10: Fire Hydrant by Claire Jones, skill magic stream by Maxicons, puzzle pieces by Iconika, puzzle pieces by studiographic, water Drop by Aldiki Gustiyan Putra, and man and woman by Alice Design; all icons from the Noun Project (https://nounproject.com).

REFERENCES

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

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

[3] 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.

[4] Karl Anderson. 2016. FireHose Benchmarking Streaming Architectures. https://www.clsac.org/uploads/5/0/6/3/50633811/anderson-clsac-2016.pdf. Slides from talk at Chesapeake Large Scale Analytics Conference, Accessed: 2021-7-9.

[5] Karl Anderson and Steve Plimpton. 2013. FireHose Streaming Benchmarks. https://github.com/stream-benchmarking/firehose. Accessed: 2018-12-11.

[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] Tim Bartrand, Walter Grayman, and Terra Haxton. 2017. *Drinking water treatment source water early warning system state of the science review.* Technical Report EPA/600/R-17/405.

[9] Ran Ben-Basat, Gil Einziger, Roy Friedman, and Yaron Kassner. 2016. Heavy hitters in streams and sliding windows. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[10] 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.

[11] 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.

[12] 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.

[13] 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-trees and Write-Optimization. *login; magazine* 40, 5 (2015).

[14] 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^ε-Trees and Write-Optimization. *:login; magazine* 40, 5 (October 2015), 22–28.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] Robert S Boyer and J Strother Moore. 1991. MJRTY—a fast majority vote algorithm. In *Automated Reasoning*. Springer, 105–117.

[22] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, Jelani Nelson, Zhengyu Wang, and David P Woodruff. 2017. BPTree: an ℓ_2 heavy hitters algorithm using constant memory. In *PODS '17: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. ACM, 361–376.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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

[46] 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.

[47] 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.

[48] 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

[49] 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.

[50] 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.

[51] 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*.

[52] 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.

[53] Katsiaryna Mirylenka, Graham Cormode, Themis Palpanas, and Divesh Srivastava. 2015. Conditional heavy hitters: detecting interesting correlations in data streams. *The VLDB Journal* 24, 3 (2015), 395–414.

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

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

[56] 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.

[57] 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

[58] 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*. 1431–1446.

[59] 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

[60] Daniel Ting. 2018. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*. 1129–1140.

[61] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. 2005. New Streaming Algorithms for Fast Detection of Superspreaders. *Network and Distributed Systems Security Symposium (NDSS)*.

[62] 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

[63] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. 2019. HeavyKeeper: An Accurate Algorithm for Finding Top-k Elephant Flows. *IEEE/ACM Transactions on Networking* 27, 5 (2019), 1845–1858.

[64] Yu Zhang, BinXing Fang, and YongZheng Zhang. 2010. Identifying heavy hitters in high-speed network monitoring. *Science China Information Sciences* 53, 3 (2010), 659–676.