Timely Reporting of Heavy Hitters using External Memory

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

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

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

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

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

Rob Johnson
robj@vmware.com
VMware Research

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

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

CCS CONCEPTS

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

KEYWORDS

Dictionary data structure; streaming algorithms; external-memory algorithms

ACM Reference Format:
Prashant Pandey, Shikha Singh, Michael A. Bender, Jonathan W. Berry, Martín Farach-Colton, Rob Johnson, Thomas M. Kroeger, and Cynthia A. Phillips. 2020. Timely Reporting of Heavy Hitters using External Memory. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3318464.3380598

1 EVALUATION

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

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

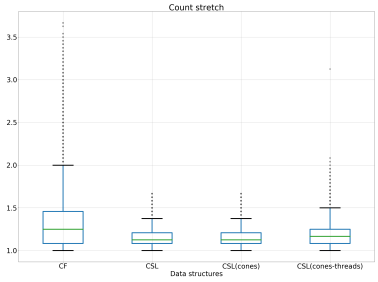
1.1 Experimental setup

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

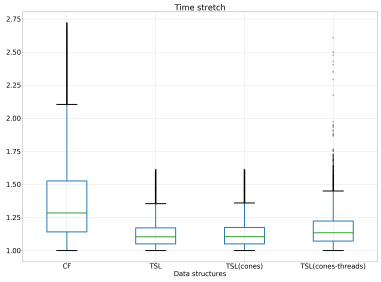
*Both authors contributed equally to this research.

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

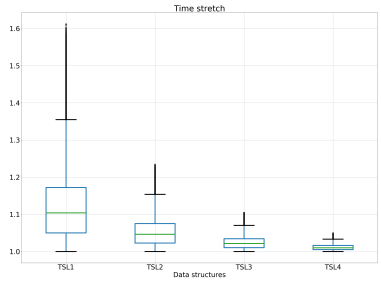
ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3380598



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



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



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

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

Workload. Firehose [**?**] 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.

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

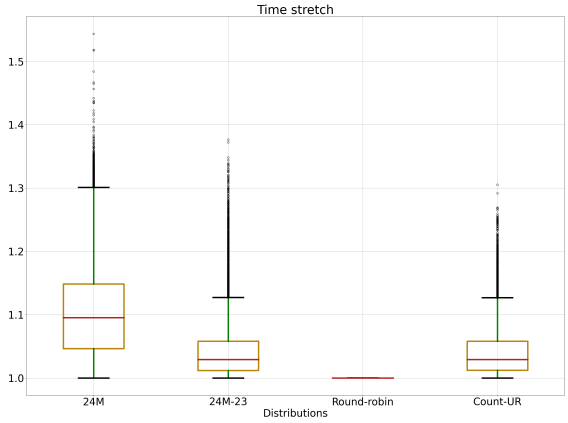
Reporting. During insertion, we record each reported item and the index in the stream at which it is reported by the data structure. We record by inserting the reported item in an exact CQF (anomaly CQF) and encoding the index as the count of the item in the anomaly CQF. We also use the anomaly CQF to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

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

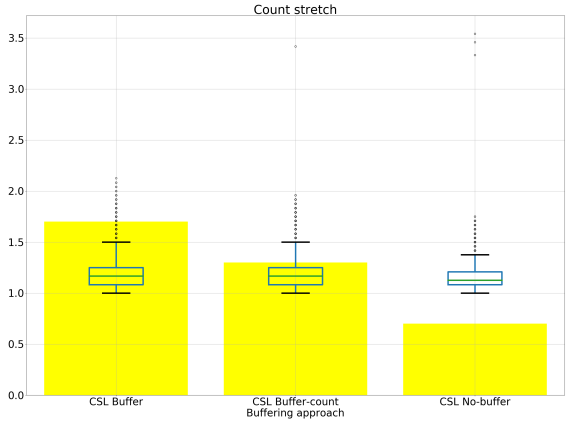
The time-stretch LERT upper bounds the reporting delay of an item based on its lifetime (i.e. time between its first and Tth instance). To validate the timeliness of the time-stretch LERT, we first perform an offline analysis of the stream and calculate the lifetime of each reportable item. Given a reporting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

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

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



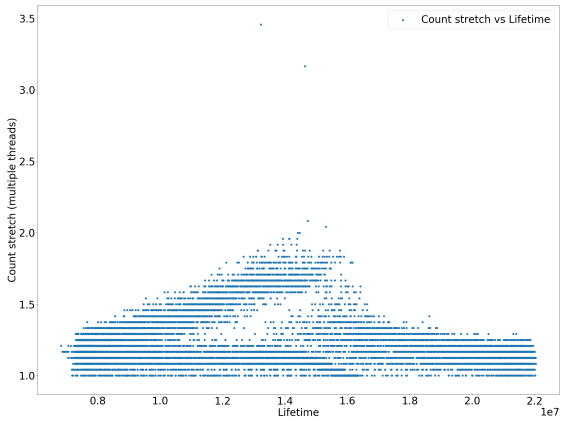
(a) Distribution of time stretch for different distributions. These distributions are described in Section 1.1.



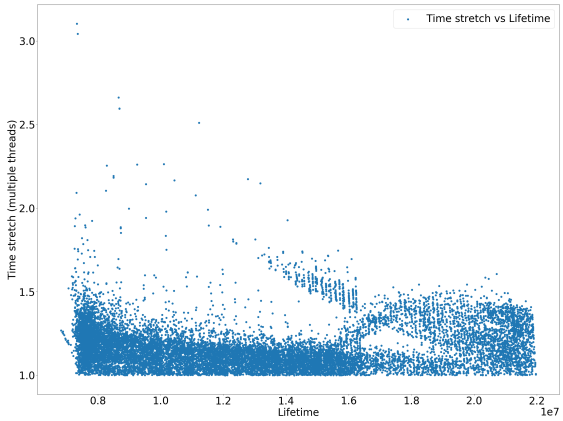
(b) Distribution of count stretch with different buffering strategies. Bars show the average insertion throughput (Million insertions/sec) for each buffering strategy. Average insertion throughput when no-buffer is used is 2.7× lower compared to when buffers are used.

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

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.



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



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

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

I/O performance. In our implementation of the time-stretch, count-stretch and immediate-report LERT, we allocate space for the data structure by mmap-ing each level (i.e., the CQF) to a file on SSD. To force the data structure to keep all levels except the first one on SSD we limit the RAM available to the insertion process using the "cgroups" utility in linux. We calculate the total RAM needed by the insertion process to only keep the first level in RAM by adding the size of the first level, the space used by the anomaly CQF to

record reported keys, the space used by thread-local buffers, and a small amount of extra space to read the stream sequentially from SSD. We then provision the RAM to the next power-of-two of the total sum.

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

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

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

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

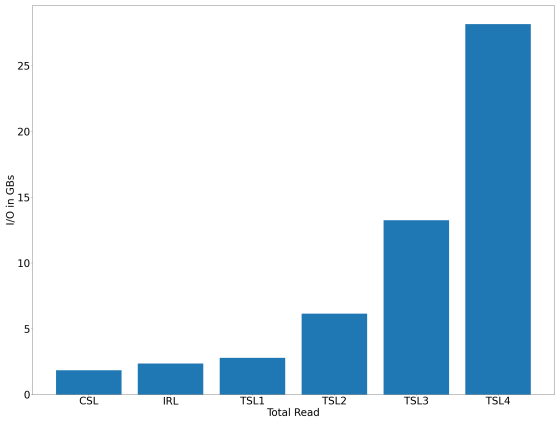
To evaluate scalability, we measure how data-structure throughput changes with increasing number of threads. We evaluate power-of-2 thread counts between 1 and 64.

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

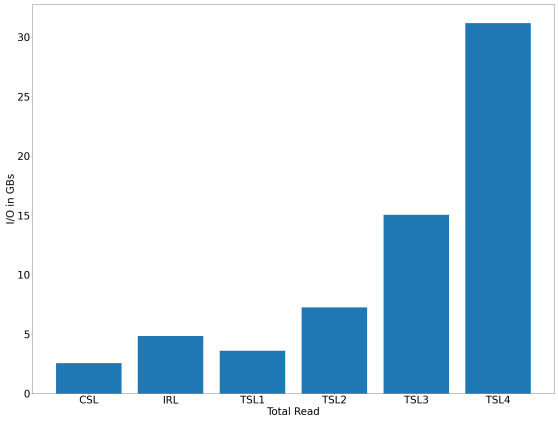
Instantaneous insertion throughput. We also evaluate the instantaneous throughput of the data structure when run using either a single cone and thread or multiple cones and threads. We approximate instantaneous throughput by calculating throughput (using system timestamps) every κ observations. In our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

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

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



(a) Reads I/O



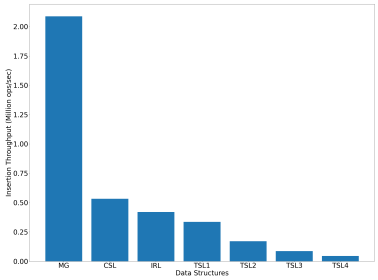
(b) Write I/O

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

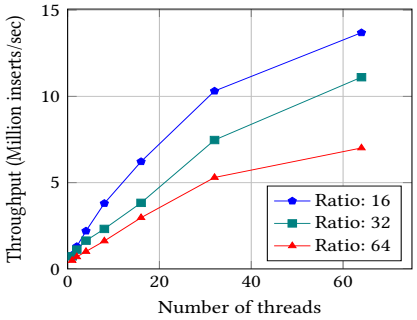
1.2 Timely reporting

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

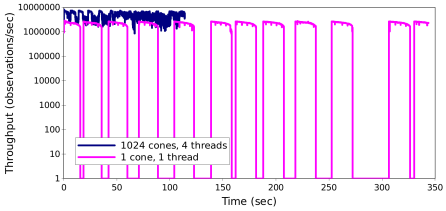
Count-stretch LERT. Figure 1a validates worst-case count stretch for the count-stretch LERT. The total on-disk count for an element is 14, so the maximum possible count when



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



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



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

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

reported is 38 (i.e., 24 + 14), for a maximum count stretch of 1.583. The maximum reported count stretch is 1.583.

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

1.3 Robustness with input distributions

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

1.4 Effect of deamortization/threading

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

Using 8 cones instead of one does not change the timeliness of any reported item. This is because the distribution of items in the stream is random (see Section 1.1) 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 **??**).

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

1.5 Effect of buffering

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

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

1.6 I/O performance and throughput

Section 1.1 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 5a shows insertion throughput using the same configuration and stream as the total-I/O experiments. The count-stretch LERT has the highest throughput because it performs the fewest I/Os. The immediate-report LERT has lower throughput because it performs extra random point queries. The time-stretch LERT throughput decreases as we add bins and decrease the stretch.

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

1.7 Instantaneous throughput

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

1.8 Scaling with multiple threads

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