Stream Frequency over Interval Queries *

Ran Ben Basat
Harvard University
ran@seas.harvard.edu

Roy Friedman
CS Technion
roy@cs.technion.ac.il

Rana Shahout
CS Technion
ranas@cs.technion.ac.il

ABSTRACT

Stream frequency measurements are fundamental in many data stream applications such as financial data trackers, intrusion-detection systems, and network monitoring. Typically, recent data items are more relevant than old ones, a notion we can capture through a *sliding window* abstraction. This paper considers a generalized sliding window model that supports stream frequency queries over an interval given at *query time*. This enables drill-down queries, in which we can examine the behavior of the system in finer and finer granularities. For this model, we asymptotically improve the space bounds of existing work, reduce the update and query time to a constant, and provide deterministic solutions. When evaluated over real Internet packet traces, our fastest algorithm processes items 90–250 times faster, serves queries at least 730 times quicker and consumes at least 40% less space than the best known method.

PVLDB Reference Format:
Ran Ben Basat, Roy Friedman, Rana Shahout. Stream Frequency Over Interval Queries. *PVLDB*, 12(4): 433-445, 2018.
DOI: https://doi.org/10.14778/3297753.3297762

1 Introduction

High-performance stream processing is essential for many applications such as financial data trackers, intrusion-detection systems, network monitoring, and sensor networks. Such applications require algorithms that are both time and space efficient to cope with high-speed data streams. Space efficiency is needed, due to the memory hierarchy structure, to enable cache residency and to avoid page swapping. This residency is vital for obtaining good performance, even when the theoretical computational cost is small (e.g., constant time algorithms may be inefficient if they access the DRAM for each element). To that end, stream processing algorithms often build compact approximate *sketches* (synopses) of the input streams.

*This work is partially supported by ISF grant #1505/16, Technion HPI research school, Zuckerman Institute and the Technion Hiroshi Fujiwara cyber security research center.

Proceedings of the VLDB Endowment, Vol. 12, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3297753.3297762

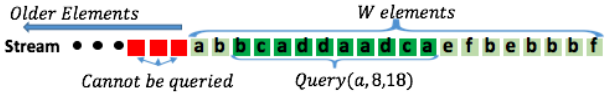


Figure 1: We process items and support frequency queries within an interval specified at query time. While the traditional sliding window model can answer queries for a *fixed window*, our approach allows us to consider *any* interval that is contained within the last W items. In this example, we ask about the frequency of the item a within the interval $[8, 18]$. If we allow an additive error of 2, the answer to this query should be in the range $[4, 6]$.

Recent items are often more relevant than old ones, which requires an aging mechanism for the sketches. Many applications realize this by tracking the stream's items over a *sliding window*. That is, the sliding window model [18] considers only a window of the most recent items in the stream, while older ones do not affect the quantity we wish to estimate. Indeed, the problem of maintaining different types of sliding window statistics was extensively studied [4, 8, 18, 33, 27].

Yet, sometimes the window of interest may not be known a priori or they may be multiple interesting windows [17]. Further, the ability to perform drill-down queries, in which we examine the behavior of the system in finer and finer granularity may also be beneficial, especially for security applications. For example, this enables detecting when precisely a particular anomaly has started and who was involved in it [20]. Additional applications for this capability include identifying the sources of flash crowd effects and pinpointing the cause-effect relation surrounding a surge in demand on an e-commerce website [26].

In this work, we study a model that allows the user to specify an interval of interest at query time. This extends traditional sliding windows that only consider fixed sized windows. As depicted in Figure 1, a sub-interval of a maximal window is passed as a parameter for each query, and the goal of the algorithm is to reply correspondingly. Naturally, one could maintain an instance of a sliding window algorithm for each possible interval within the maximal sliding window. Alas, this is both computationally and space inefficient. Hence, the challenge is to devise efficient solutions.

This same model was previously explored in [33], which based their solution on *exponential histograms* [18]. However, as we elaborate below, their solution is both memory wasteful and computationally inefficient. Further, they only provide probabilistic guarantees.

Table 1: Comparison of the algorithms proposed in the paper with ECM and WCSS (that solves the simpler problem of fixed-size windows). ACC_k can be instantiated for any $k \in \mathbb{N}$.

Algorithm	Space	Update Time	Query Time	Comments
WCSS [8]	$O(\epsilon^{-1}\log(W\lvert\mathcal{U}\rvert))$	$O(1)$	$O(1)$	Only supports fixed-size window queries.
ECM [33]	$O(\epsilon^{-2}\log W\log\delta^{-1})$	$O(\log\delta^{-1})$	$O(\epsilon^{-1}\log W\log\delta^{-1})$	Only provides probabilistic guarantees.
RAW	$O(\epsilon^{-2}\log(W\lvert\mathcal{U}\rvert))$	$O(\epsilon^{-1})$	$O(1)$	Uses prior art (WCSS) as a black box.
ACC_k	$O\Big(\epsilon^{-1}\log(W\lvert\mathcal{U}\rvert) + k\epsilon^{-(1+1/k)}\log\epsilon^{-1}\Big)$	$O(k+\epsilon^{-2}/W)$	$O(k)$	Constant time operations for $k = O(1) \wedge \epsilon = \Omega(W^{-1/2})$.
HIT	$O(\epsilon^{-1}(\log(W\lvert\mathcal{U}\rvert) + \log^2\epsilon^{-1}))$	$O(1 + (\epsilon^{-1}\cdot\log\epsilon^{-1})/W)$	$O(\log\epsilon^{-1})$	Optimal space when $\log^2\epsilon^{-1} = O(\log(W\lvert\mathcal{U}\rvert))$, $O(1)$ time updates when $\epsilon = \Omega\left(\frac{\log W}{W}\right)$.

Contributions

Our work focuses on the problem of estimating frequencies over an ad-hoc interval given at query time. We start by introducing a formal definition of this generalized estimation problem nicknamed (W,ϵ)-IntervalFrequency.

To systematically explore the problem, we first present a naïve strawman algorithm (RAW), which uses multiple instances of a state-of-the-art fixed window algorithm. In such an approach, an interval query is satisfied by querying the instances that are closest to the beginning and end of the interval and then subtracting their results. This algorithm is memory wasteful and its update time is slow, but it serves as a baseline for comparing our more sophisticated solutions. Interestingly, RAW achieves constant query time while the previously published ECM algorithm [33] answers queries in $O(\epsilon^{-1}\log W\log\delta^{-1})$, where W is the maximal window size and δ is the probability of failure. Additionally, it requires about the same amount of memory and is deterministic while ECM has an error probability.

While developing our advanced algorithms, we discovered that both intrinsically solve a common problem that we nickname n-Interval. Hence, our next contribution is in identifying and formally defining the n-Interval problem and showing a reduction from n-Interval to the (W,ϵ)-IntervalFrequency problem. This makes our algorithms shorter, simpler, and easier to prove, analyze and implement.

Our algorithms, nicknamed *HIT* and ACC_k (to be precise, $\{ACC_k\}_{k\geq 1}$ is a family of algorithms), process items in constant time (under reasonable assumptions on the error target) – asymptotically faster than RAW. HIT is asymptotically memory optimal while serving queries in logarithmic time. Conversely, ACC_k answers queries in constant time and incurs a sub-quadratic space overhead.

We present formal correctness proofs as well as space and runtime analysis. We summarize our solutions' asymptotic performance in Table 1.

Our next contribution is a performance evaluation study of our various algorithms along with (*i*) ECM-Sketch [33], the previously suggested solution for interval queries and (*ii*) the state-of-the-art *fixed window* algorithm (WCSS) [8], which serves as a best case reference point since it solves a more straightforward problem. We use on real-world packet traces from Internet backbone routers, from a university datacenter, and from a university's border router. Overall, our methods (HIT and ACC_k) process items 75–2000 times faster and consume at least 20 times less space than the naive approach (RAW) while requiring a similar amount of memory as the state-of-the-art *fixed size window* algorithm (WCSS). Compared to the previously known solution to this problem (ECM-Sketch [33]), all our advanced algorithms are both faster and more space efficient. In particular, our fastest algorithm, ACC_1, processes items 90–250 times faster than ECM-Sketch, serves queries at least 730 times quicker and consumes at least 40% less space.

Last, we extend our results to time-based intervals, heavy hitters [31, 12], hierarchical heavy hitters [15, 21], and for detecting traffic volume heavy-hitters [9], i.e., when counting each flow's total traffic rather than item count. We also discuss applying our algorithms in a distributed settings, in which measurements are recorded independently by multiple sites (e.g., multiple routers), and the goal is to obtain a global network analysis.

Paper roadmap We briefly survey related work in Section 2. We state the formal model and problem statement in Section 3. Our naïve algorithm RAW is described in Section 4. We present the auxiliary n-Interval problem, which both our advanced algorithms solve and has a simple reduction to the (W,ϵ)-IntervalFrequency problem, in Section 5. The improved algorithms, *HIT* and ACC_k, are then described in Section 6. The performance evaluation of our algorithms and their comparison to ECM-Sketch and WCSS is detailed in Section 7. Section 8 discusses extensions of our work. Finally, we conclude with a discussion in Section 9.

2 Related Work

Count Sketch [13] and Count Min Sketch [16] are perhaps the two most widely used sketches for maintaining item's frequency estimation over a stream. The problem of estimating item frequencies over sliding windows was first studied in [4]. For estimating frequency within a $W\epsilon$ additive error over a W sized window, their algorithm requires $O(\epsilon^{-1}\log^2\epsilon^{-1}\log W)$ bits. This was then reduced to the optimal $O(\epsilon^{-1}\log W)$ bits [27]. In [23], Hung and Ting improved the update time to $O(1)$ while being able to find all heavy hitters in the optimal $O(\epsilon^{-1})$ time. Finally, the WCSS algorithm presented in [8] also estimates item frequencies in constant time. While some of these works also considered a variant in which the window can expand and shrink when processing updates [4, 27], its size was increased/decreased by one at each update, and cannot be specified at query time.

The most relevant paper that solves the same problem as our work is [33], who was the first to explore heavy hitters interval queries. They introduced a sketching technique with probabilistic accuracy guarantees called *Exponential Count-Min sketch* (ECM-Sketch). ECM-Sketch combines *Count-Min sketch*' structure [16] with *Exponential Histograms* [18]. Count-Min sketch is composed of a set of d hash functions, and a 2-dimensional array of counters of width w and depth d. To add an item x of value v_x, Count-Min sketch increases the counters located at $CM[j,h_j(x)]$ by v_x, for $1 \leq j \leq d$. Point query for an item q is done by getting the minimum value of the corresponding cells.

Exponential Histograms [18] allow tracking of metrics over a sliding window to within a multiplicative error. Specifically, they allow one to estimate the number of 1's in a sliding window of a binary stream. To that end, they utilize a sequence of *buckets* such that each bucket stores the timestamp of the oldest 1 in the bucket. When a new element arrives, a new bucket is created for it; to save space, the histogram may merge older buckets. While the amortized update complexity is $O(1)$, some arriving elements may trigger an $O(\log W)$-long cascade of bucket merges.

ECM-Sketch replaces each Count-Min counter with an Exponential Histogram. Adding an item x to the structure is analogous to the case of the regular Count-Min sketch. For each of the histograms $CM[\mathrm{j}, h_j(x)]$, where $1 \leq j \leq d$, the item is registered with time/count of its arrival and all expired information is removed from the Exponential Histogram. To query item x in range r, each of the corresponding d histograms $E(\mathrm{j}, h_j(x, r))$, where $1 \leq j \leq d$, computes the given query range. The estimate value for the frequency of x is $\min_{j=1,\ldots,d} E(j, h_j(x), r)$. While the Exponential Histogram counters estimate the counts within a multiplicative error, their combination with the Count-Min sketch changes the error guarantee to additive.

An alternative approach for these interval queries was proposed in [17]. Their solution uses hCount [24], a sketch algorithm which is essentially identical to the Count-Min sketch. Unlike the ECM-Sketch, which uses a matrix of Exponential Histograms, [17] uses a sequence of $\log(W/b)$ *buckets* each of which is associated with an hCount instance. The smallest bucket is of size b while the size of the i'th bucket is $b \cdot 2^{i-1}$. When queried, [17] finds the buckets closest to the interval and queries the hCount instances. The paper does not provide any formal accuracy guarantees but shows that it has reasonable accuracy in practice. It seems that the memory used is $O(\log(W/b) \cdot \epsilon^{-1} \log \delta^{-1} \log W)$ bits while the actual error has two components: (i) an error of up to $b + W/4$ in the *time axis* (when the queried interval is not fully aligned with the buckets); and (ii) an error of up to $W\epsilon$, with probability $1 - \delta$, due to the hCount instance used for the queried buckets.

In *other* domains, ad-hoc window queries were proposed and investigated. That is, the algorithm assumes a predetermined upper bound on the window size W, and the user could specify the actual window size $w \leq W$ at query time. This model was studied for quantiles [28] and summing [7].

The problem of identifying the frequent items in a data stream, known as *heavy hitters*, dates back to the 80's [31]. There, Misra and Gries (MG) proposed a space optimal algorithm for computing an $N\epsilon$ additive approximation for the frequency of elements in an N-sized stream. Their algorithm had a runtime of $O(\log \epsilon^{-1})$, which was improved to a constant [19, 25]. Later, the Space Saving (SS) algorithm was proposed [30] and shown to be empirically superior to prior art (see also [14, 29]). Surprisingly, Agarwal et al. recently showed that MG and SS are isomorphic [2], in the sense that from a k-counters MG data structure one can compute the estimate that a $k+1$ SS algorithm would produce.

The problem of *hierarchical heavy hitters*, which has important security and anomaly detection applications [32], was previously addressed with the SS algorithm [32]. To estimate the number of packets that originate from a specific network (rather than a single IP source), it maintains several separate SS instances, each dedicated to measuring different network sizes (e.g., networks with 2-bytes net ids are tracked separately than those with 3-bytes, etc.). When a packet arrives, all possible prefixes are computed and each is fed into the relevant SS instance. Recently, it was shown that randomization techniques can drive the update complexity down to a constant [6, 10].

Table 2: List of Symbols

Symbol	Meaning
$\mathcal{S}$	the data stream
$\mathcal{U}$	the universe of elements
W	the maximal window size
f_x^w	the frequency of element x within the last w elements of $\mathcal{S}$
$\widehat{f_x^w}$	an estimation of f_x^w
$f_x^{i,j}$	the frequency of element x between the i^{th} and j^{th} most recent elements of $\mathcal{S}$
$\widehat{f_x^{i,j}}$	an estimation of $f_x^{i,j}$
ϵ	estimation accuracy parameter
δ	probability of failure
n	number of blocks in a frame ($6/\epsilon$)
N	max sum of blocks' cardinalities ($12/\epsilon$) within a window

3 Preliminaries

Given a *universe* $\mathcal{U}$, a stream $\mathcal{S} = x_1, x_2, \ldots \in \mathcal{U}^*$ is a sequence of universe elements. We denote by $W \in \mathbb{N}$ the *maximal window size*; that is, we consider algorithms that answer queries for an interval contained with the last W elements window. The actual value of W is application dependent. For example, a network operator that wishes to monitor up to a minute of traffic of a major backbone link may need W of tens of millions of packets [22]. Given an element $x \in \mathcal{U}$ and an integer $0 \leq w \leq W$, the *w-frequency*, denoted f_x^w, is the number of times x appears within the last w elements of $\mathcal{S}$. For integers $i \leq j \leq W$, we further denote by $f_x^{i,j} \triangleq f_x^j - f_x^i$ the frequency of x between the i^{th} and j^{th} most recent elements of $\mathcal{S}$.

We seek algorithms that support the following operations:

- **ADD(x)**: given an element $x \in \mathcal{U}$, append x to $\mathcal{S}$.
- **IntervalFrequencyQuery(x, i, j)**: given $x \in \mathcal{U}$ and indices $i \leq j \leq W$, return an estimate $\widehat{f_x^{i,j}}$ of $f_x^{i,j}$.

We now formalize the required guarantees.

Definition 1. *An algorithm solves* (W, ϵ)*-IntervalFrequency if given any* IntervalFrequencyQuery(x, i, j) *it satisfies*

$$f_x^{i,j} \leq \widehat{f_x^{i,j}} \leq f_x^{i,j} + W\epsilon.$$

For simplicity of presentation, we assume that $W\epsilon/12$ and ϵ^{-1} are integers. For ease of reference, Table 2 includes a summary of basic notations used in this work.

Space Saving: as we use the Space Saving (SS) algorithm [30] in our reduction in Section 5.2, we overview it here. SS maintains a set of $1/\epsilon$ counters, each has an associated element and a value. When an item arrives, SS first checks if it has a counter. If so, the counter is incremented; otherwise, SS allocates the item with a minimal-valued counter. For example, assume that the smallest counter was associated with x and had a value of 4; if y arrives and has no counter, it will take over x's counter and increment its value to 5 (leaving x without a counter). When queried for the frequency of a flow, we return the value of its counter if it has one, or the minimal counter's value otherwise. If we

denote the overall number of insertions by Z, then we have that the sum of counters equals Z, and the minimal counter is at most $Z\epsilon$. This ensures that the error in the SS estimate is at most $Z\epsilon$. An important observation is that once a counter reached a value of $Z\epsilon$ it is no longer the minimum throughout the rest of the measurement.

4 Strawman Algorithm

Here, we present the simple Redundant Approximate Windows (RAW) algorithm that uses several instances of a black box algorithm $\mathbb{A}(w,\epsilon)$ for solving the frequency estimation problem over a fixed W-sized window. That is, we assume that $\mathbb{A}(w,\epsilon)$ supports the ADD(x) operation and upon QUERY(x) produces an estimation $\widehat{f_x^w}$ that satisfies: $f_x^w \leq \widehat{f_x^w} \leq f_x^w + w\epsilon$. We note that the WCSS algorithm [8] solves this problem using $O(\epsilon^{-1})$ counters and in $O(1)$ time for updates and queries. Both its runtime and space are optimal.[1]

Specifically, we maintain $4\epsilon^{-1}$ separate solutions denoted $A_1, \ldots A_{4\epsilon^{-1}}$, where each A_ℓ is an $\mathbb{A}(\ell \cdot W\epsilon/4, \epsilon/4)$ instance. We perform the ADD(x) operation simply by invoking the operation A_ℓ.ADD(x) for $\ell = 1, \ldots, 4\epsilon^{-1}$. When given an INTERVALFREQUENCYQUERY(x,i,j), we return

$$\widehat{f_x^{i,j}} \triangleq A_{\lceil j/(W\epsilon/4)\rceil}.\text{QUERY}(x) - A_{\lfloor i/(W\epsilon/4)\rfloor}.\text{QUERY}(x) + W\epsilon/4.$$

We now state the correctness of RAW. Due to lack of space, we defer the proof to the full version of the paper [11].

Theorem 1. *Using WCSS as the black box algorithm $\mathbb{A}$, RAW requires $O(\epsilon^{-2}(\log W + \log|\mathcal{U}|))$ bits, performs updates in $O(\epsilon^{-1})$ time and answers queries in constant time.*

5 Block Interval Frequency

In this section, we formally define an auxiliary problem, nicknamed n-Interval, show a reduction to the (W,ϵ)- IntervalFrequency problem, and rigorously analyze the reduction's cost. Our motivation lies in the fact that the suggested algorithms in Section 6 both intrinsically solve the n-Interval auxiliary problem. It also has the benefit that any improved reduction between these problems would improve both algorithms. In n-Interval, the arriving elements are inserted into $O(W\epsilon)$-sized "blocks" and we are required to compute exact interval frequencies *within the blocks*. Doing so simplifies the presentation and analysis of the algorithms in Section 6, in which we propose algorithms that improve over RAW in both space and update time. The two algorithms, HIT and ACC_k present a space-time tradeoff while achieving asymptotic reductions over RAW.

5.1 The Block Interval Frequency Problem

Here, instead of frequency, we consider items' *block frequency*. Namely, for some $x \in \mathcal{U}$, we define its window block frequency g_x^n as the number of blocks x appears in within the last n blocks. For integers $i \leq j \leq n$, we define $g_x^{i,j} \triangleq g_x^j - g_x^i$. Block algorithms support three operations:

- **ADD(x)**: given an element $x \in \mathcal{U}$, add it to the stream.
- **ENDBLOCK()**: a new empty block is inserted into the window, and the oldest one leaves.

[1] A lower bound of matching asymptotic complexity appears in [25], even for non-window solutions.

Table 3: Variables used by the Algorithm 1.

fo	the offset within the current frame.
$\mathcal{A}$	an algorithm that solves $(6/\epsilon)$-Interval.
SS	a Space Saving instance with $\lceil 6\epsilon^{-1}\rceil$ counters.
s	the size of blocks (fixed at $s \triangleq W\epsilon/6$).

- **IntervalQuery(x,i,j)**: given an element $x \in \mathcal{U}$ and indices $i \leq j \leq n$, compute $g_x^{i,j}$ (without error).

We define the n-Interval as $(W, \epsilon = 0)$-IntervalFrequency. That is, we say that an algorithm solves the n-Interval problem if given an INTERVALQUERY(x,i,j) it is able to compute the *exact* answer for any $i \leq j \leq n$ and $x \in \mathcal{U}$.

For analyzing the memory requirements of algorithms solving this problem, we denote by N the sum of cardinalities of the blocks in the n-sized window. An example of this setting is given in Figure 2.

5.2 A Reduction to (W,ϵ)-IntervalFrequency

We show a reduction from the n-Interval problem to (W,ϵ)-IntervalFrequency. To that end, we assume that $\mathcal{A}$ is an algorithm that solves the n-Interval problem for $n \triangleq 6\epsilon^{-1}$.

Our reduction relies on the observation that by applying such $\mathcal{A}$ on a data structure maintained by counter-based algorithm such as Space Saving [30], we can compute interval queries and not only fixed window size frequency estimations. The setup of the reduction is illustrated in Figure 3. We break the stream into W sized *frames*, which are further divided into *blocks* of size $O(W\epsilon)$. We employ a Space Saving [30, 8] instance to track element frequencies within each frame; it supports two methods: ADD(x) – adds element x to the stream and QUERY(x) – reports the frequency estimation of element x with tight guarantees on the error.

Whenever a counter reaches an integer multiple of the block size, we add its associated flow's identifier to the most recent block of $\mathcal{A}$. When a frame ends, we *flush* the Space Saving instance and reset all of its counters. We note that an implementation that supports constant time flush operations was suggested in [8]. Also, the max sum of block's cardinalities within a window (overlapping up to 2 frames) is $N = 12/\epsilon$. Finally, we reduce each INTERVALFREQUENCYQUERY to an INTERVALQUERY by computing the indices of the blocks in which the interval starts and ends. The variables of the reduction algorithm are described in Table 3 and its pseudocode appears in Algorithm 1.

Algorithm 1 From Blocks to Approximate Frequencies

```
    Initialization: fo ← 0, s ≜ Wε/6,   initialize   A, SS(ε/6).
1:  function ADD(x)
2:      fo ← (fo + 1) mod W
3:      SS.Add(x)
4:      if SS.Query(x) mod s = 0 then
5:          A.Add(x)
6:      if fo mod s = 0 then
7:          A.EndBlock()
8:      if fo = 0 then
9:          SS.Flush()
10: function IntervalFrequencyQuery(x, i, j)
11:     return s · (A.IntervalQuery(x, ⌈i/s⌉, ⌊j/s⌋) + 2)
```

5.3 Theoretical Analysis

Given a query INTERVALFREQUENCYQUERY(x,i,j), we are required to estimate $f_x^{i,j} = f_x^j - f_x^i$. Our estimator is $\widehat{f_x^{i,j}} = \mathcal{A}.IntervalQuery(x, \lceil i/(W\epsilon/6)\rceil, \lfloor j/(W\epsilon/6)\rfloor) + W\epsilon/3$. Intuitively, we query $\mathcal{A}$ for the block frequency of x in the

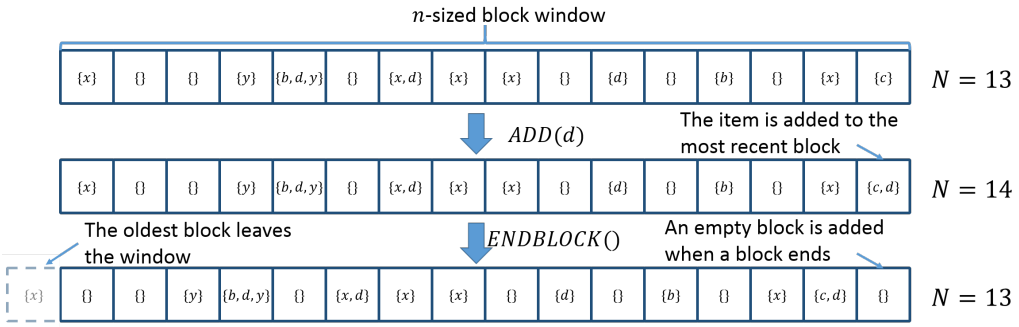


Figure 2: The block stream setting. Here, after the EndBlock, x appears in two blocks out of the last 9 and thus $g_x^9 = 2$.

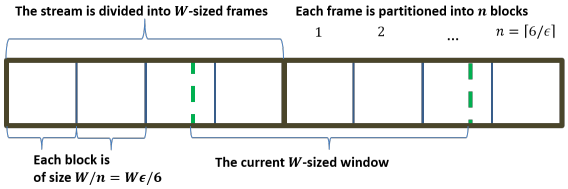


Figure 3: The stream is logically divided into intervals of size W called *frames* and each frame is logically partitioned into n equal-sized *blocks*. The window of interest is also of size W, and overlaps with at most 2 frames and $n+1$ blocks.

minimal sequence of blocks that contain interval i, j. Every time x's counter reaches an integer multiple of the block size, the condition in Line 4 is satisfied and the block frequency of x, as tracked by $\mathcal{A}$, increases by 1. Thus, multiplying the block frequency by $s \triangleq W\epsilon/6$ allows us to approximate x's frequency in the original stream.

There are several sources of estimation error: First, we do not have a counter for each element but rather a Space Saving instance in which counters are shared. Next, unless the counter of an item reaches an integer multiple of s, we do not add it to the block stream. Additionally, the queried interval might not be aligned with the blocks. Finally, when a frame ends, we flush the counters and thus lose the frequency counts of elements that are not recorded in the block stream. With these sources of error in mind, we prove the correctness of our algorithm.

Theorem 2. *Let $\mathcal{A}$ be an algorithm for the $6\epsilon^{-1}$-Interval problem. Then Algorithm 1 solves (W, ϵ)-IntervalFrequency.*

Proof. We begin by noticing that once an element's counter reaches $s = W\epsilon/6$, it will stay associated with the element until the end of the frame. This follows directly from the Space Saving algorithm, which only disassociates elements whose counter is minimal among all counters(see the SS overview in Section 3). Recall that the number of elements in a frame is W and that the Space Saving instance is allocated with $\lceil 6\epsilon^{-1} \rceil$ counters. Since the sum of counters always equals the number of elements processed, any counter that reaches a value of s will never be minimal. Thus, once an element was added to a block (Line 5), its block frequency within the frame is increased by one for every s subsequent arrivals. This means that an item might be added to a block while appearing just once in the stream, but this gives an overestimation of at most $s-1$. As the queried intervals can overlap with two frames, this can happen at most twice, which imposes an overestimation error of no more than $2s$.

Our next error source is the fact that the queried interval may begin and end anywhere within a block. By considering the blocks that contain i and j, regardless of their offset, we incur another overestimation error of at most $2s$.

We have two sources of underestimation error, where items frequency is lower than s times its block frequency. The first is the count we lose when flushing the Space Saving instance. Since we record every multiple of s in the block stream, a frequency of at most $s-1$ is lost due to the flush. Second, in the current frame, the residual frequency of an item (i.e., the appearances that have not been recorded in the block stream) may be at most $s-1$. We make up for these by adding $2s$ to the estimation (Line 11). As we have covered all error sources, the total error is smaller than $6s \leq W\epsilon$. □

Reducing the Error Above, we used a block size of $s = W\epsilon/6$, which can be reduced to $W\epsilon/5$ as follows: One of the error sources in Theorem 2 is the fact that the queried interval i, j may begin and end in the middle of a block and we always consider the entire blocks that contain i and j. We can optimize this by considering i's and j's offsets *within* the relevant blocks, and including the block's frequency only if the offset crosses half the size of the block. This incurs an overestimation error of at most s instead of $2s$, allows blocks of size $W\epsilon/5$ and reduces the number of blocks to $n = 5/\epsilon$.

6 Improved Algorithms

6.1 Approximate Cumulative Count (ACC)

We present a family of algorithms for solving the n-Interval problem. Approximate Cumulative Count (ACC) of level k, denoted ACC_k, aims to compute the interval frequencies while accessing at most k hash tables for updates and $2k+1$ for queries. To reduce clutter, we assume in this section that $n^{1/k} \in \mathbb{N}$; this assumption can be omitted with the necessary adjustments while incurring a $1 + o(1)$ multiplicative space overhead. This family presents a space-time trade off — the larger k is, ACC_k takes less space but is also slower.

The ACC algorithms break the block stream into consecutive *frames* of size n (the maximal window size). That is, blocks $B_1, B_2, \ldots B_n$ are in the first frame, $B_{n+1}, \ldots B_{2n}$ in the second frame and so on. Notice that any n-sized window intersects with at most two frames. Within each frame, ACC algorithms use a hierarchical structure of tables that enables it to compute an item's block frequency in $O(1)$ time.

ACC_1 and ACC_2 are illustrated in Figure 4 and are explained below. The simplest and fastest algorithm, ACC_1, computes for each block a frequency table that tracks how many times each item has arrived *from the beginning of the frame*. For example, the table for block $5n+7$ (for $n > 7$) will contain an entry for each item that is a member of at least one of $B_{5n+1}, \ldots B_{5n+7}$. The key is the item identifier,

Table 4: Variables used by ACC_k algorithm.

$BlockSize$	the number of segments from a level that consist of a next-level segment.
$Tables[\ell, idx]$	used for tracking block frequencies. Each table is identified with a level ℓ and the index of the last block in its segments.
$incTables[\ell]$	tables for incomplete segments.
$ghostTables[\ell]$	tables for leaving segments.
$offset$	The offset within the current frame.

and the value is its block frequency from the frame's start. This way, we can compute any block interval frequency by querying at most 3 tables. Within a frame, we can compute any interval by subtracting the queried item's block frequency at the beginning of the interval from its block frequency at the end. If the interval spans across two frames, we make one additional query for reaching the beginning of the frame, in total we query at most 3 tables.

ACC_2 saves space at the expense of additional table accesses. Tables now have "levels", such that each table is either in $level_0$ or $level_1$. The core idea is that ACC_1 is somewhat wasteful as it may create $O(n)$ table entries for each item, as it appears in all tables within the frame after its arrival. Instead, we "break" each frame into $\sqrt{n}$ sized *segments*. At the end of each segment, we keep a single $level_1$ table that counts item frequencies from the beginning of the frame. Since we can use these tables just as in ACC_1, we are left with computing the queried item frequency *within a segment*. This is achieved with a $level_0$ table, which we maintain for each block. Alas, unlike the $level_1$ tables, $level_0$ tables only keep the block frequency counts from the *beginning of the segment* the block belongs to. Thus, each appearance of an item (within a specific block) can appear on all $\sqrt{n}$ $level_1$ tables, but on at most $\sqrt{n}$ $level_0$ tables and this reduce space consumption. Compared with ACC_1, ACC_2 reduces the overall number of table entries from $O(N \cdot n)$ to $O(N \cdot \sqrt{n})$.

For an interval $[i, j]$, let $block_i$ and $block_j$ be the block numbers of i and j respectively. To answer any interval frequency query of item x, we consider two cases: If $block_i$ and $block_j$ are in the same frame, we access $block_i$'s and $block_j$'s tables to get x's frequency from the beginning of the frame till $block_i$, $block_j$ and subtract the results, line 18 . If $block_i$ and $block_j$ are in different frames, we consider x's frequency in the j blocks within the current frame by accessing $block_j$'s tables, plus its frequency within the last i blocks of the previous frame. To do so, we compute x's frequency from the beginning of the previous frame, lines 19 - 22. A corner case that arises is that the $level_1$ table that includes $block_j$ may have already left the table. We solve it by maintaining *ghostTables* for leaving segments: for $1 \le \ell \le k$, $ghostTables[\ell]$ contains the table of last leaving block that has a table at $level_\ell$, line 10. Hence, we can subtract the corresponding *ghostTables* entries as well.

Next, we generalize this to arbitrary k values. In ACC_k, we have k levels of tables and segments. We consider each block to be in its own $level_0$ segment and maintain a $level_0$ table for it. Inductively, each $level_\ell$ segment (for $1 \le \ell \le k$) consists of $n^{1/k}$ $level_{\ell-1}$ segments. That is, each $level_1$ segment contains $n^{1/k}$ blocks, $level_2$ segments each consists of $n^{1/k}$ $level_1$ segments for a total of $n^{2/k}$ blocks, etc. As each item may now appear in at most $n^{1/k}$ tables of each level, we get that the overall number of table entries is $O(N \cdot k \cdot n^{1/k})$. To avoid lengthy computations at the end of each segment,

Algorithm 2 ACC_k

```
    Init: offset ← 1, d ≜ n^{1/k},
 1:       initialize    Tables, incTables, ghostTables
 2: function ADD(x)
 3:     for ℓ ∈ 0, 1, ..., k − 1 do        ▷ Update all incomplete tables
 4:         incTables[ℓ](x) += 1
 5: function ENDBLOCK()
 6:     ℓ ← 0
 7:     while ((ℓ < k − 1) ∧ offset mod d^{ℓ+1} = 0) do
                                          ▷ A Level-ℓ block has ended
 8:         empty incTables[ℓ], ghostTables[ℓ]      ▷ Delete all entries
 9:         ℓ ← ℓ + 1
10:     ghostTables[ℓ] ← Tables[ℓ, offset]
11:     Tables[ℓ, offset] ← incTables[ℓ]                ▷ Copy Table
12:     if offset = n then                               ▷ New frame
13:         empty incTables[k − 1], ghostTables[k − 1]
14:     offset ← 1 + (offset mod n)
15: function WINQUERY(x, w)        ▷ Frequency in the last w blocks
16:     cFreq ← incTables[0](x) + Σ_{ℓ=1}^{log_d offset} Tables[ℓ, d^ℓ ⌊offset/d^ℓ⌋](x)
17:     if w ≤ offset + 1 then
18:         return cFreq − Σ_{ℓ=0}^{log_d offset+1−w} Tables[ℓ, d^ℓ ⌊(offset+1−w)/d^ℓ⌋](x)
19:     B ← n + offset + 1 − w
20:     L ← max{ℓ : d^ℓ ⌊B/d^ℓ⌋ ≥ offset + 1}
21:     preW ← Σ_{ℓ=0}^{L} Tables[ℓ, d^ℓ ⌊B/d^ℓ⌋](x) + Σ_{ℓ=L+1}^{k−1} ghostTables[ℓ](x)
22:     return cFreq + Tables[k − 1, n](x) − preW
23: function INTERVALQUERY(x, i, j)
24:     if i = 0 then
25:         return WINQUERY(x, j)
26:     return WINQUERY(x, j) - WINQUERY(x, i)
```

we maintain k additional "incomplete" tables that contain the cumulative counts for segments that already started, but not all of their blocks have ended yet. A pseudo-code of the ACC_k algorithm appears in Algorithm 2.

6.1.1 Analysis

The following theorem bounds the memory consumption of the ACC algorithms.

Theorem 3. *Denote the sum of cardinalities of the last n blocks by N. Algorithm 2 requires $O(Nn^{\frac{1}{k}} k \log(n|\mathcal{U}|))$ space.*

Proof. Each item may appears in at most $n^{1/k}$ tables of each level (because, as described before, each $level_\ell$ segment (for $1 \le \ell \le k$) consists of $n^{1/k}$ $level_{\ell-1}$ segments). ACC_k algorithm has k levels of tables, thus each item may appear in total at most $kn^{1/k}$ tables. Each table entry consists of an $O(\log|\mathcal{U}|)$-bits identifier and a counter of $O(\log n)$ bits. Thus, the space of each item is $O(n^{\frac{1}{k}} k \log(n|\mathcal{U}|))$; hence, for N items, ACC_k requires $O(Nn^{\frac{1}{k}} k \log(n|\mathcal{U}|))$ bits. □

Theorem 4. *Algorithm 2 solves the n-Interval problem.*

Proof Sketch. We need to prove that upon an INTERVALQUERY (x, i, j) query, for any $i \le j \le n$ and $x \in \mathcal{U}$, ACC_k is able to compute the *exact* answer. Notice that in handling queries in Algorithm 2, we split the computation in two: The first case is when $i = 0$, Line 24, this means that the end of the interval is also the last block, and thus we only need to return the frequency in the last j block in the window, as calculated by WINQUERY(x, j). Otherwise, we subtract the frequency that is calculated WINQUERY(x, i) from the result of WINQUERY(x, j). Hence, we need to show that the frequency calculated by WINQUERY(x, w) is correct.

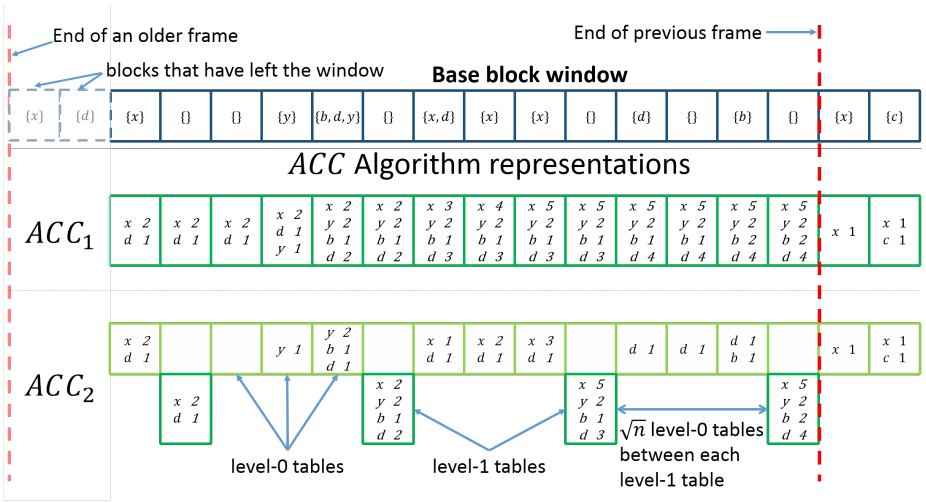


Figure 4: Illustration of the ACC_1 and ACC_2 algorithms. ACC_1 has only $level_0$ tables that track how many times each item has arrived from the beginning of the frame, while ACC_2 has two levels of tables. Each $level_1$ table in ACC_2 tracks the frequencies from the beginning of the frame, while $level_0$ tables aggregate the data from the previous $level_1$ table.

As is evident from the code in Lines 1–13, *incTables* store the frequency of items within the current block, while *Tables* store the frequencies of completed blocks from the beginning of their frame. Consider the case where the entire interval is within the current frame. In this case, the frequency of an item in the last w blocks can be calculated as its frequency in the current block (using *incTables*) plus its frequency in the preceding blocks, as is done in Line 16, and stored in *cFreq*. Notice that to reduce query time, we access the highest level containing this information. However, since *Tables* store the frequency from the beginning of the frame, we need to subtract from *cFreq* the item's frequency in prior blocks, which is done in Line 18 (again, by accessing the highest level tables that include this data).

The second case is when the given interval crosses into the previous frame. In this case, we need to add to *cFreq* the frequency of the blocks that are included in the previous frame. Once again, we need to query the table holding the frequency in the last relevant block of that frame and subtract from the result the frequency in the preceding tables. As some of these tables might be beyond an entire window limit, their information might be stored in *ghostTables* rather than *Tables*. This is handled in Lines 19–22. □

6.2 Hierarchical Interval Tree (*HIT*)

Hierarchical Interval Tree, denoted *HIT*, tracks flow frequencies using a hierarchical tree structure in which each node stores the partial frequency of its sub-tree. Precisely, the levels of the tree are defined as follows: $level_0$ includes frequency tables, one for each block of the stream, that track how many times each item arrived *within the corresponding block*. Tables at $level_\ell$ of $block_i$ track how many times each item has arrived *between* $block_{i-2^\ell+1}$ *and* $block_i$, where $0 < \ell \leq trailing\_zeros(i)$, line 8. That is, these tables contain partial queries results for each item and track item's multiplicity from the previous same level block. Hence, each level contains tables for half the blocks of the previous level, and thus each $block_i$ has tables in $trailing\_zeros(i)$ levels; we assume that the number of trailing zeros can be computed efficiently with the *ctz* machine instruction in modern CPUs. An illustration of the algorithm appears in Figure 5.

For example, consider $block_9$, $block_{10}$, $block_{11}$ and $block_{12}$ in Figure 5. During $block_9$, items x and d arrive; x also arrives in $block_{10}$ and $block_{11}$, while there are no items arrivals in $block_{12}$. So the tables of $block_{12}$ will be as follow: $level_0$ table is empty because there is no items arrival within $block_{12}$. $level_1$ table tracks items arrival between $block_{11}$ and $block_{12}$; its content will be item x with count 1. $level_2$ table counts the item arrival between $block_9$ and $block_{12}$, so it will contain item x three times ($block_9$, $block_{10}$ and $block_{11}$) and d once (in $block_9$). Note that each table at $level_{\ell+1}$ merges two $level_\ell$ frequency tables.

We can compute any interval frequency by using the hierarchical tree tables. While this can be done using linear scan, the higher levels of the tree are designed to allow efficient time computation by using the stored partial queries.

Notice that some of the partial queries results stored in the higher levels may be invalid. For example, in case a new block is added, the oldest one departs the window, so the content of tables that refer to the departing block become invalid. We solve this problem by choosing the levels to use such that we only consider valid tables. Let $block_i$ and $block_j$ be the block numbers of the first interval index and the second one. Here, we scan backward from $block_j$ to $block_i$, greedily using the highest possible level at each point, line 16. This minimizes the number of needed steps. If $block_j > block_i$, all tables along the way are valid. In this case, we only need $\log_2(block_j - block_i + 1)$ value look-ups. Otherwise, we choose $level_0$ tables between blocks 1 and $block_j$, so we need $\log_2(block_j + 1)$ value look-ups, and then another $\log_2(n - block_i + 1)$ look-ups for querying the remaining interval. Overall, our computation takes at most $2 \log n$ steps.

We use an incremental table for incomplete blocks in each we increment an element x's entry for any ADD($\boldsymbol{x}$) operation, line 2. The pseudo code of the algorithms appears in Algorithm 3 and Table 5 contains a list of the used variables.

6.2.1 Analysis

Theorem 5. *Algorithm 3 solves the n-Interval problem.*

The proof is simple and technical; it appears in the full version of this paper [11].

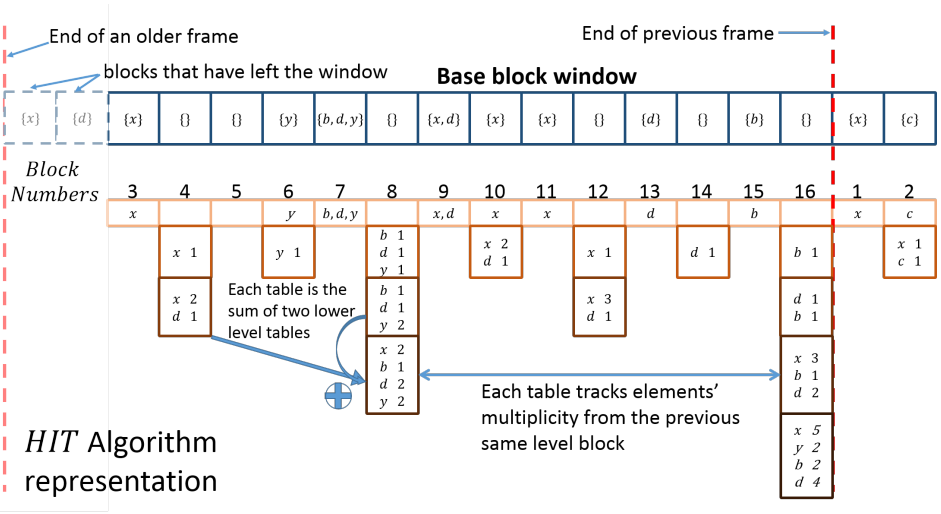


Figure 5: *HIT* algorithms, first level tables track how many times each item arrived within the corresponding block. At $level_\ell$, tables of $block_i$ track how many times each item has arrived *between* $block_{i-2^\ell+1}$ *and* $block_i$. For example, item b arrives once at $block_7$, so $block_7$ $level_0$ table contains b with count 1, $block_8$ $level_1$ table tracks how many times each item arrived between $block_7$ and $block_8$, so it contains b with count 1 and $level_2$ table for $block_8$, track how many times each item arrived between $block_5$ and $block_8$, and as well contains b once. Table at $level_{\ell+1}$ merges two $level_\ell$ frequency tables. For example, $block_8$ third level table merges second level tables of $block_8$ and $block_8$.

Algorithm 3 *HIT*

```
    Initialization: offset ← 0, initialize    Tables, incTable.
1:  function ADD(x)
2:      incTable(x)+ = 1    ▷ Update the incomplete block's tables
3:  function ENDBLOCK()
4:      offset ← (offset + 1)  mod n
5:      Tables[0, idx] ← incTable
6:      empty  incTable                          ▷ Delete all entries.
7:      for ℓ ∈ 1, . . . , ctz(offset) do
8:          Tables[ℓ, offset] = Tables[ℓ − 1, offset]
                                  + Tables[ℓ − 1, offset − 2^ℓ + 1]
9:  function INTERVALQUERY(x, i, j)
10:     last = (offset − j)  mod n   ▷ The most recent block's index
11:     first = (offset − i)  mod n      ▷ The oldest queried block's
12:     b ← first
13:     count ← 0
14:     d ← 1 + (first − last  mod n)
15:     while d > 0 do
16:         level ← min(ctz(b), ⌊log d⌋)
17:         count ← count + Tables[level, b](x)
18:         d ← d - 2^level
19:         b ← b - 2^level
20:         if b = 0 then
21:             b ← k
22:     return count
```

Theorem 6. *Denote the sum of cardinalities of the last n blocks by N. Algorithm 3 requires* $O(N \log n \log(n|\mathcal{U}|))$ *space.*

Proof. As described above, each element's appearance may reflect in $O(\log n)$ tables. Every table entry takes $O(\log |\mathcal{U}|)$ bits for the key and another $O(\log n)$ for the value, and thus the overall space is $O(N \log n \log(n|\mathcal{U}|))$. □

Optimizations In the full version [11] we provide optimizations that reduce HIT's space to $O(N(\log|\mathcal{U}| + \log n \log N))$ and that of ACC_k to $O(N(\log|\mathcal{U}| + n^{\frac{1}{k}} k \log N))$. We also discuss how to *deamortize* the update process to get the worst-case time complexity equivalent to the amortized analysis.

Table 5: Variables used by *HIT* algorithm.

$BlockSize$	Number of blocks from a level that consist of a next-level block.
$Tables[\ell, idx]$	used for tracking block frequencies. Each table is identified with a level ℓ and the index of the last block in its block.
$incTable$	A table for the most recent, incomplete, block.
$offset$	The offset within the current frame.

7 Evaluation

We developed a C++ prototype of all algorithms described in this work: *HIT*, *RAW*, and instantiations of the ACC_k protocols for $k = 1, 2, 4, 8$. Here, the *HIT* and ACC_k algorithms are implemented using Space Saving [30] as a building block. Besides, we also implemented *ECM-Sketch* [33] (a.k.a *ECM*) in C++ for comparison because the authors' code is in Java. *ECM* was configured for error probability $\delta = 0.01\%$. As Table 1 shows, δ affects the space and performance of ECM. Specifically, the memory, update time, and query time, all logarithmically depend on δ^{-1}. While the actual value of δ is application dependent, for performing a drill-down query (which translates into muliple interval queries), one may need δ to be quite small so that the overall error probability will be acceptable. Figure 8 shows *ECM* space consumption and update time as functions of δ. As expected, as δ increases, update and query operations become faster and *ECM* consumes less space but the overall error probability is higher. We also compared with the *WCSS* algorithm [8] as a general baseline since it is the state of the art for the more straightforward problem of a *fixed* sliding window. Here again, we implemented *WCSS* in C++ as its authors implemented it in Java. For each algorithm, we evaluated the speed of executing INTERVALFREQUENCYQUERY (x, i, j) and ADD(x) operations, as well as its memory requirements.

The evaluation was performed on an Intel(R) 3.20GHz Xeon(R) CPU E5-2667 v4 running Linux with kernel 4.4.0-71. Each data point in all runtime measurements is shown as a 95% confidence interval of 10 runs. Our evaluation includes a *Backbone* dataset collected during 2016 from the

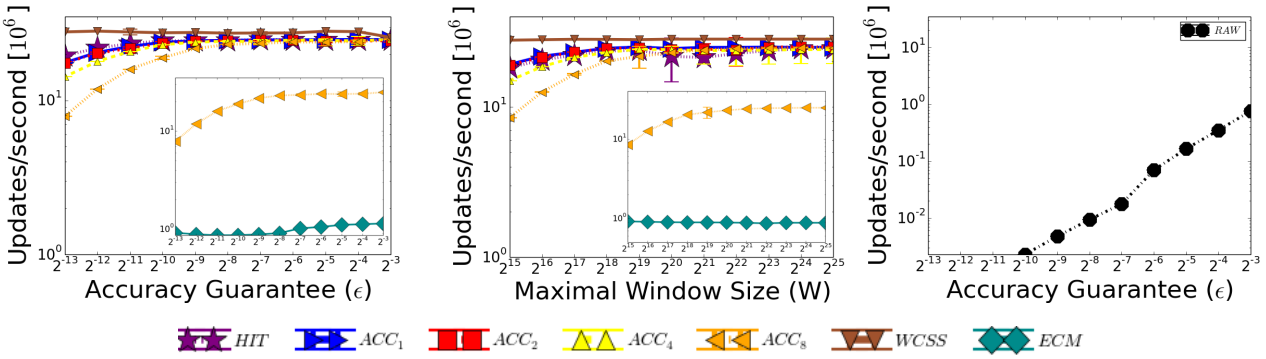


Figure 6: Update operation runtime comparison as a function of the accuracy guarantee (ϵ) and the maximum window size (W). Since the update speed of RAW is significantly slower than the other algorithms, we have placed it in a separate graph and we managed to run *RAW* only up to $\epsilon = 2^{-10}$ due to its memory consumption limitation. The graphs plot contains a subplot which compares *ECM* with our slowest algorithm, ACC_8 in this case. We see that *ECM* is much slower than ACC_8. We also compared our algorithms with *WCSS* which is the state of the art for the simpler problem of a *fixed* sliding window.

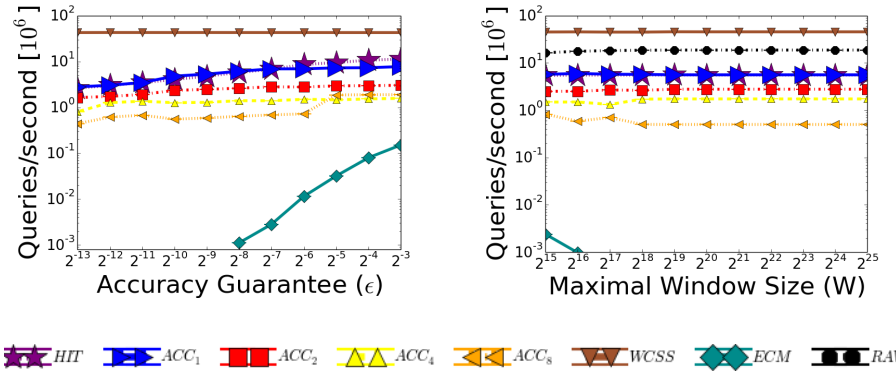


Figure 7: Query operation runtime comparison as a function of the accuracy guarantee (ϵ) and the maximum window size (W). When varying ϵ, again, we managed to run *RAW* only up to $\epsilon = 2^{-10}$ due to its memory consumption limitation. For graphs exploring varying W, we ran *ECM* only up to 2^{16} due to time limitation. We compared our algorithms with *WCSS* since it is the state of the art for the simpler problem of a *fixed* sliding window but it can only answer queries with fixed window size.

backbone router 'equinix-chicago' [22]. In the full version [11] we show these graphs with two additional packet traces (a data-center and an edge router) with very similar results.

7.1 Update Speed Comparison

Figure 6 compares the update speed. We start by exploring the trade-off of ϵ parameter with a fixed maximal window of size $W = 2^{20}$. Then, we explain the trade-off of window size parameter with a fixed $\epsilon = 2^{-8}$.

7.1.1 Effect of ϵ on Update Time

Throughout, as ϵ decreases, more tables must be updated on every overflowed element. Thus, update operations become slower when ϵ decreases. As depicted, *HIT* update performance is close to ACC_1 and ACC_2. As k increases, there are more tables to update on every overflowed element, so the performance decreases. This difference becomes especially noticeable with small ϵ values.

Recall that every update operation in *RAW* means $4\epsilon^{-1}$ add operations, one for every $4\epsilon^{-1}$ instances of the $\mathbb{A}(\cdot, \epsilon/4)$ algorithm, which is *WCSS* in our implementation. So, as ϵ decreases, update operations take more time. Since the update speed of RAW is orders of magnitudes slower than the other algorithms, we have placed it in a separate graph in which we managed to run this only for $\epsilon \geq 2^{-10}$ due to space limitation on the server. This echoes Table 1, which presents the analytical performance summary of the different algorithms. Among our algorithms depicted in Figure 6, the slowest one is ACC_8, as can be seen in the inner graphs, even ACC_8 processes items 57-210 times faster than *ECM*.

7.1.2 Effect of Window Size on Update Time

Figure 6 shows also the effect of window size when ϵ is fixed to 2^{-8}. All algorithms perform better when the window size is larger as this means fewer blocks and table accesses. The ACC_k algorithms get slower as k increases as they need to update more tables. Again, we compared the most inefficient algorithm ACC_8 with *ECM* in the inner graphs; ACC_8 processes items 50-218 times faster than *ECM* for the given ϵ values.

7.2 Query Speed Comparison

For query speed comparison, we chose random intervals, each of size 1% of the total window's size. We begin the evaluation by exploring the impact of the ϵ parameter with a fixed window of size 2^{20}. Then, we explain the trade-off of the window size parameter with fixed $\epsilon = 2^{-8}$. The performance of the improved algorithms is compared with the existing work, *ECM*, and *WCSS*, recall that *WCSS* can only answer queries with fixed window size.

7.2.1 Effect of ϵ on Query Time

As shown in Figure 7, *RAW* is the fastest as each interval query is translated to two *WCSS* queries. We managed to run *RAW* only up to $\epsilon = 2^{-10}$ due to its memory consumption limitation (see section 7.3).

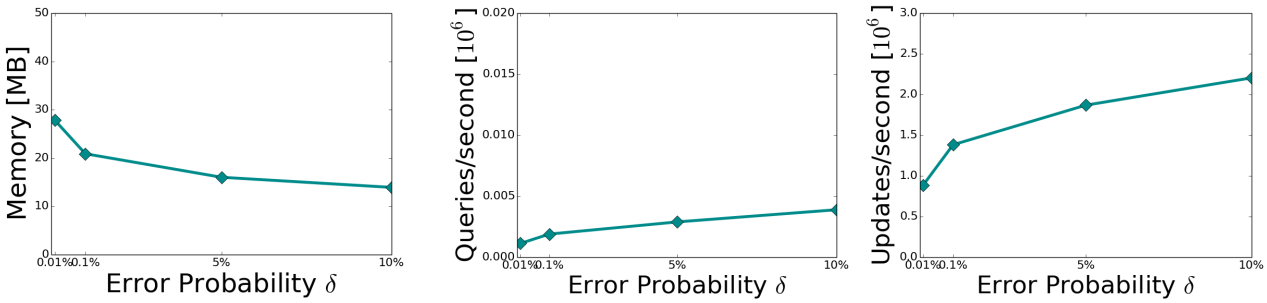
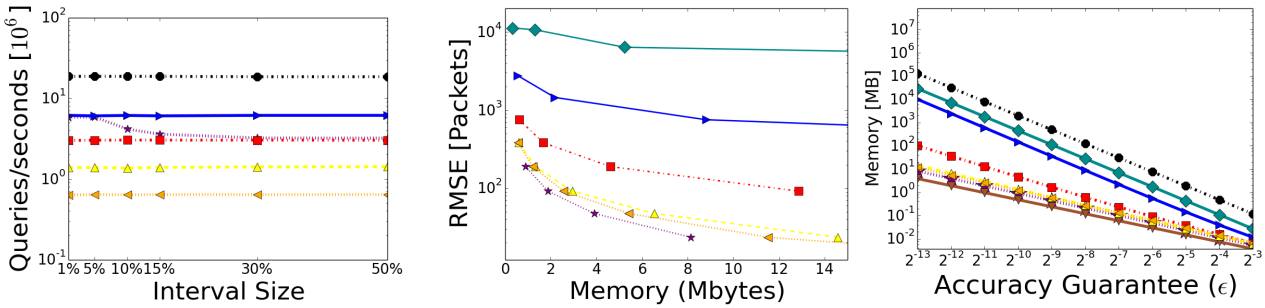


Figure 8: *ECM* space and performance comparison as functions of the error probability δ using *Backbone* dataset, $\epsilon = 2^{-8}$ and window of size 2^{20}. Note that the y-axes of these graphs is in linear scales.



(a) Vary Interval sizes **(b)** Observed Error **(c)** Memory Consumption

Figure 9: (a) Query operation runtime comparison as a function of interval size (b) Root Mean Square Error comparison as a function of required memory and maximum window of size 2^{20} (c) Algorithms space comparison as a function of the accuracy guarantee (ϵ).

HIT computes any block interval frequency by using the hierarchical tree tables, greedily choosing the highest possible level each time. For decreasing ϵ values, the blocks numbers increases, so the queried interval crosses more blocks and accesses more tables. Consequently, we got slower interval query operations.

For the ACC_k algorithms, for increasing k values we get fewer queries per seconds as we read more tables on average. For example, ACC_1 computes any block frequency by querying at most 3 tables, while ACC_2 does the same by accessing no more than 5 tables as explained in Section 6.1. Query operations runtime depends also on the interval itself; there are "good" intervals in which the corresponding blocks have table at level $k-1$, so one table access for each is sufficient. Therefore, we chose random intervals for every query. As ϵ value decreases, block sizes become smaller and the number of tables grow. In this case, not all the tables fit in memory and we experienced paging that causes lower query performance. Recall that while *WCSS* is the fastest, it solves the much simpler problem of a fixed window size and only serves as a best case reference point. *ECM* answers queries in a very inefficient way compared to our algorithms. We only run *ECM* up to $\epsilon = 2^{-8}$ due to time limitation. As expected, its performance decreases for decreasing ϵ values.

7.2.2 *Effect of Window Size on Query Time*

As mentioned before, we evaluated queries by choosing random intervals of size 1% of window's size, when ϵ is fixed to 2^{-8}. Figure 7 shows that all algorithms' query performance is not very sensitive to the window size. This is because the number of tables accessed depends on the ratio between the interval and window sizes. We ran *ECM* only up to 2^{16} due to time limitation. The performances of our algorithms are orders of magnitudes better than *ECM* also in this case.

7.3 Memory Consumption Comparison

Figure 9c shows the space consumed by our algorithms as well as *ECM* for a given ϵ value. As seen, the smaller ϵ gets, all algorithms consume more space. We can see that *ECM* is more compact than *RAW* but consumes more space than the others. As mentioned before, RAW maintains $4\epsilon^{-1}$ separate *WCSS* instances so its space consumption is the largest. For the ACC_k algorithms, as k increases, the overall number of tables entries for overflowed elements decreases resulting is better space consumption. So there is a trade-off between the speed and required spaces by adjusting the parameter k , using Figure 9c and figures 6,7 can help choosing k parameter according to the desired speed and memory consumption. Yet, *ECM* consumes more space than ACC_1 which has the highest memory consumption among the ACC_k algorithms family. As shown, the memory consumption of ACC_8 is close to *HIT*. Yet, *HIT* is the most efficient among the algorithms that solve n-Interval because its data structure is the most compact but its query performance affected by the interval size as explained in section 7.4 so for larges interval sizes we may prefer *ACC* algorithm over *HIT*. Recall that while *WCSS* is the most compact algorithm in term of space, it solves the much simpler problem of a fixed window size and only serves as a best case reference point.

7.4 Interval Size Comparison

Figure 9a shows query runtime performance of our algorithms as a function of interval size. The query operation performance was measured with random intervals of varying

sizes: 1%, 5%, 10%, 15%, 30%, or 50% of the total window's size while fixing $\epsilon = 2^{-8}$ and $W = 2^{20}$.

As expected, the query performance of *HIT* gets slower as the size of the interval gets larger, because the queried interval crosses more blocks and accesses more tables. In contrast, query performance of ACC_k algorithms is not affected by interval size as ACC_k algorithms consider only the edges of the queried interval. That is, when the edges are i and j, they compute the frequency of the given item from the beginning of the frame till $block_i$ and $block_j$ and subtract the results. *RAW* algorithm query performance is not affected by interval size since it is translated to two *WCSS* queries regardless of interval size. *ECM* algorithm is not included in the graph since it is orders of magnitudes slower than our algorithms so we will not see the difference between them (see figure 7). *ECM* query performance is affected by interval size since its query operation depends on Exponential Histograms [18] query. As interval size gets larger, Exponential Histograms scans a larger sequence of *buckets* and as a result, *ECM* query gets slower.

In conclusion, when the interval size is big, we would prefer to choose ACC_1 over *HIT* when there is sufficient memory. We expect that as data rates and volumes get higher, one would use smaller ϵ values, making ACC_k increasingly more attractive also for larger k values.

7.5 Root-Mean-Square Error Comparison

Figure 9b shows the empirical Root Mean Square Error (*RMSE*) in correlation with the required memory for ACC_k algorithms, *HIT* and *ECM* with window of size 2^{20}. The observed errors are lower than the user-selected value ϵ. Since ACC_k algorithms and *HIT* solve the same n-Interval instance, their empirical error is equal for same ϵ values. The difference between the algorithms comes from the memory requirements which differ for the same n value. In general, a lower space consumption required for a specific ϵ value translates into better empirical error. For example, ACC_1 consumes more memory than ACC_2 for the same ϵ. Thus, for a given memory budget, ACC_2 is more accurate than ACC_1 and *HIT* is more accurate than both. *ECM* was measured with error probability $\delta = 0.01\%$ which led to large memory consumption relative to *HIT* and ACC_k algorithms; as a result its empirical error higher than others but yet lower than the theoretical value.

8 Extensions and Applications

Here, we briefly discuss how our solutions can be applied to temporal queries, weighted stream, distributed stream, heavy hitters, and hierarchical heavy hitters (HHH).

8.1 Time Based Intervals

In this section, we describe how to extend our algorithms for supporting time intervals. The idea is that sometimes what matters is the flow frequencies in a time interval rather than during a item-count interval. For example, if we want to allow a user to make 100 queries/sec to an API, we need to measure the number of times this user has accessed the system during the last second. In such a setting, we consider a *timed stream* $\mathcal{S} = \langle x_1, t_1\rangle, \langle x_2, t_2\rangle, \ldots \in (\mathcal{U} \times N)^*$. Here, each item has an integer valued timestamp and we assume that the items arrive in order, *i.e.*, $t_1 \leq t_2 \leq \ldots$.

We also assume that the number of elements that arrive in a single time-frame is bounded by $R \in \mathbb{N}$. In practice, this is a reasonable assumption; for example, if we perform the measurement over a 1Gbps link, and each item must be of size of at least 64 bytes for its headers, then we can set $R \triangleq 10^9/(8 \cdot 64) = 2M$ [items / second]. We denote by h_x^t the frequency of x within the last t timestamps. That is, if the query time is T, then $h_x^t \triangleq |\{\langle x, t_i\rangle \in \mathcal{S} : t_i \geq T - t\}|$. Similarly, we define the frequency within a time window as $h_x^{i,j} \triangleq h_x^i - h_x^j$. The time based interval algorithms goal is then to answer the following queries:

- **TimeIntervalFrequencyQuery(x, i, j)**: given an element $x \in \mathcal{U}$ and indices $j \leq i \leq W$, return an estimate $\widehat{h_x^{i,j}}$ of $h_x^{i,j}$.

Finally, if an algorithm's error is at most an ϵ fraction of the overall possible traffic in T time, we say that it solves the $(\mathrm{T}, R, \epsilon)$-TimeIntervalFrequency problem. That is, its estimation needs to satisfy

$$\forall j \leq i \leq \mathrm{T} : h_x^{i,j} \leq \widehat{h_x^{i,j}} \leq h_x^{i,j} + \mathrm{T} \cdot R \cdot \epsilon.$$

Our construction has two parts: We maintain a $(\mathrm{T} \cdot R, \epsilon/2)$-IntervalFrequency solution in addition to a data structure that translates time intervals into item intervals. For this, we use Ben Basat's *Sliding Ranker* (SR) algorithm [7] that can compute a sliding window sum over an integer stream, where the size of the window is given at query time. SR has parameters $\langle R, \mathfrak{W}, \Delta\rangle$; it processes a stream in $\{0, 1, \ldots, R\}$ such that upon a query for some $i \leq \mathfrak{W}$, it computes a Δ-additive approximation for the sum of the last i elements. Every timestamp, we feed the *number of items* that arrived into an SR with parameters $\langle R, \mathrm{T}, \mathrm{T} \cdot R\epsilon/2\rangle$. Given a time-interval query x, i, j, we use the SR for computing the number of items sent since time i and from time j. We then use these estimations to query the IntervalFrequency instance for the estimated item-interval. Since the SR and IntervalFrequency each has an error of $\mathrm{T} \cdot R\epsilon/2$, we satisfy the error guarantee. The memory consumption of SR for $\Delta = \Theta(R\mathfrak{W})$ is just $O(R\mathfrak{W}\Delta + \log \mathfrak{W}) = O(\epsilon^{-1} + \log \mathfrak{W})$ bits.

8.2 Supporting Heavy-Hitters

We now show how one can use the described algorithms to support heavy hitters queries over a given interval. Denote by $HH_\theta^{i,j} \triangleq \left\{x \in \mathcal{U} : f_x^{i,j} \geq \theta \cdot (j - i)\right\}$ the set of heavy hitters items that appeared at least a θ fraction of the queried interval for given integers $i \leq j \leq W$ and a real number $\theta \in [0, 1]$. **IntervalHeavyHittersQuery(θ, i, j)** operation returns an estimate $\widehat{HH_\theta^{i,j}} \subseteq \mathcal{U}$ that approximates $HH_\theta^{i,j}$ given indices $i \leq j \leq W$. The algorithms solves **IntervalHeavyHittersQuery(θ, i, j)** and guarantees

$$HH_\theta^{i,j} \subseteq \widehat{HH_\theta^{i,j}} \subseteq \left\{x \in \mathcal{U} : f_x^{i,j} \geq \theta \cdot (j - i) - W\epsilon\right\}.$$

That is, the estimated set must contain all elements that appear at least a θ fraction of the interval and must not have any members whose frequency is lower than $\theta \cdot (j - i) - W\epsilon$. Given that the described algorithms solve the (W, ϵ)-IntervalFrequency problem, by the following observation they also solve the (W, ϵ)-IntervalHeavyHitters problem.

Observation 7. *Any algorithm $\mathbb{A}$ that solves (W, ϵ)-IntervalFrequency can answer an* IntervalHeavyHittersQuery *by returning $\widehat{HH_\theta^{i,j}} \triangleq \{x \in \mathcal{U} : \widehat{f_x^{i,j}} \geq \theta \cdot (j - i)\}$.*

Specifically, all algorithms presented in this paper can compute the set $\widehat{HH_\theta^{i,j}}$ in time $O(\epsilon^{-1})$ without iterating over all universe elements. Thus, we note that all proposed algorithms can efficiently compute the $\widehat{HH_\theta^{i,j}}$ suggested above.

8.3 Hierarchical Heavy Hitters

Next, we describe how our algorithms can be used for answering interval HHH queries (see [32] for formal definitions). In [32], Mitzenmacher et al. proposed combining their approach (which originally utilized Space Saving [30]) with sliding window algorithms such as [23, 8] to solve HHH on sliding windows. However, such an approach yields a fixed window size algorithm. By replacing the underlying black box algorithm by our interval query solutions, we get an algorithm that solves HHH on interval queries. This is also orthogonal to the other approaches; a combination of the extensions proposed in this chapter would allow finding HHH over time-based intervals, finding distributed HHH (see the following section), or finding HHH in terms of traffic volume (Section 8.5).

8.4 The Distributed Model

We now consider applying our algorithms in distributed settings. Here, multiple streams are received at various sites $S_1,\ldots,S_r$ ($r > 1$) and each site maintains its own instance of the chosen algorithm, e.g., ACC_k, HIT, etc. Obtaining a global view of the system's status requires merging data structures from all individual sites. The common way of serving such queries is to have all individual sites transmit a copy of their data structures to a central controller C, which merges them into a global data structure. This can be done either periodically assuming synchronized clocks between the sites, or in a coordinated manner initiated periodically by C. Queries are forwarded to the controller that computes the reply based on its merged data-structure. This model is communication efficient when queries are frequent, since queries are served directly by the controller and the rate in which the distributed sites need to communicate with the controller can be lower than the query rate.

In contrast, when queries are not as frequent, the above solution is inefficient, since the sites needlessly update the controller. To that end, by applying the time based intervals adaptation, our solution enables the reverse model. That is, given a range query, it is directly propagated to each of the r distributed sites. Each site returns its locally computed portion and all replies are then merged into a global one. The reason why time based intervals are needed is that individual streams might arrive at different rates to the various sites. Hence, it is meaningless to merge the results of queries on an item based window or range. For this reason, the above approach cannot be applied to item based sketch algorithms.

Last, as mentioned before, our algorithms provide an ϵ error guarantee. Hence, when each of the sites runs its independent instance, the overall error guarantee of the distributed model becomes $r\epsilon$. Another way of looking at this is that since the space requirement is inversely proportional to the error guarantee, the space requirement for a given error grows with r. Since usually r is a small constant, for most systems this is acceptable.

8.5 Supporting Traffic Volume Heavy-Hitters

It is often desired to find the heavy hitters in terms of traffic volume. That is, consider a stream in which its item has a *size* and we wish to find the flows that account for most of the bandwidth in a given interval. Formally, we consider a *weighted stream* $\mathcal{S} = \langle x_1, \mathfrak{w}_1\rangle, \langle x_2, \mathfrak{w}_2\rangle, \ldots \in (\mathcal{U} \times \{1, 2, \ldots, M\})^*$ and define a flow's volume as the sum of sizes for items that belong to it.

Intuitively, to address this problem we can add the weight of the item in Line 3 of Algorithm 1, and change the condition of Line 4 to consider whether the current estimation exceeds a new multiple of $s \cdot M$. The Space Saving algorithm [30] can find weighted heavy hitters over a stream with $O(\log \epsilon^{-1})$ update time [12]. Recent breakthroughs [9, 5, 3] improve this runtime to a constant. Thus, we can solve the interval volume estimation and (weighted) heavy hitters problems with the same asymptotic complexity as the unweighted variants and with an error of at most $WM\epsilon$. This is a generalization of the result of [5] that finds weighted heavy hitters over fixed size windows.

9 Discussion

In this paper, we studied the problems of flow frequency estimation over intervals that are passed at query time. Such capabilities can be useful when one wishes to maintain the above statistics over multiple sliding windows and for performing drill-down queries, e.g., for root cause analysis of network anomalies.

We presented formal definitions of these generalized problems and explored three alternative solutions: a naive approach (RAW) and more sophisticated solutions called HIT and ACC_k. Both HIT and ACC_k process updates in $O(1)$, but differ in their space vs. query time tradeoff: HIT is asymptotically memory optimal but answers queries in logarithmic time whereas ACC_k processes queries in $O(1)$ but consumes more space. Moreover, HIT interval queries performance is affected by interval size: as interval size gets larger, query gets slower. In contrast, the ACC_k algorithms are not affected by interval size. In fact, HIT's space requirement is similar to the memory requirement of the state of the art algorithm that can only cope with fixed size windows. Hence, HIT is adequate when space is tight or the intervals are small while ACC_k is suitable for real time query processing. Both our advanced algorithms are faster and more space efficient than ECM [33], the previously known solution for interval queries. This is true both asymptotically and in measurements over real-world traces, in which we demonstrated orders of magnitude runtime improvements as well as at least 40% memory reductions for similar estimation errors. Our approach can be applied to additional related problems. For example, we showed in Section 8 how to adapt our algorithms to answer queries over time based intervals as well as to identifying heavy hitters. This can be further generalized to the *hierarchical heavy hitters* (HHH) problem [32], which is useful in detecting distributed denial of service attacks (DDoS). In the latter, one can replace the Space Saving instances employed by [32] with HIT or ACC_k to detect HHH over query intervals!

Code Availability: All code is available online [1].

Acknowledgements: We are grateful for many helpful comments and observations made by Dimitrios Kaliakmanis.

10 References

[1] Open source code. https://github.com/r4n4sh/IFQ.

[2] P. K. Agarwal, G. Cormode, Z. Huang, J. M. Phillips, Z. Wei, and K. Yi. Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4):26, 2013.

[3] D. Anderson, P. Bevan, K. Lang, E. Liberty, L. Rhodes, and J. Thaler. A high-performance algorithm for identifying frequent items in data streams. In *Proceedings of the 2017 Internet Measurement Conference*, pages 268–282. ACM, 2017.

[4] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 286–296. ACM, 2004.

[5] R. B. Basat, G. Einziger, and R. Friedman. Fast flow volume estimation. *Pervasive and Mobile Computing*, 2018.

[6] R. B. Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Volumetric hierarchical heavy hitters. *Network*, 13:23, 2018.

[7] R. Ben-Basat. Succinct approximate rank queries. CoRR/1704.07710, 2017.

[8] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Heavy hitters in streams and sliding windows. In *INFOCOM*, pages 1–9, 2016.

[9] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner. Optimal elephant flow detection. In *IEEE INFOCOM*, 2017.

[10] R. Ben Basat, G. Einziger, R. Friedman, M. C. Luizelli, and E. Waisbard. Constant time updates in hierarchical heavy hitters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 127–140. ACM, 2017.

[11] R. Ben-Basat, R. Friedman, and R. Shahout. Stream frequency over interval queries. *CoRR*, abs/1804.10740, 2018.

[12] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems (TODS)*, 35(4):26, 2010.

[13] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[14] G. Cormode and M. Hadjieleftheriou. Methods for Finding Frequent Items in Data Streams. *J. VLDB*, 19(1):3–20, 2010.

[15] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Finding Hierarchical Heavy Hitters in Streaming Data. *ACM Trans. Knowl. Discov. Data*, 1(4):2:1–2:48, 2008.

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

[17] M. Dallachiesa and T. Palpanas. Identifying streaming frequent items in ad hoc time windows. *Data & Knowledge Engineering*, 87:66–90, 2013.

[18] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[19] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*, pages 348–360. Springer, 2002.

[20] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *ACM SIGCOMM*, pages 323–336, 2002.

[21] J. Hershberger, N. Shrivastava, S. Suri, and C. D. Tóth. Space Complexity of Hierarchical Heavy Hitters in Multi-dimensional Data Streams. In *ACM PODS*, pages 338–347, 2005.

[22] P. Hick. CAIDA Anonymized Internet Trace, equinix-chicago, 2016.

[23] R. Y. Hung, L.-K. Lee, and H.-F. Ting. Finding frequent items over sliding windows with constant update time. *Information Processing Letters*, 110(7):257–260, 2010.

[24] C. Jin, W. Qian, C. Sha, J. X. Yu, and A. Zhou. Dynamically maintaining frequent items over a data stream. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 287–294. ACM, 2003.

[25] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions Database Systems*, 2003.

[26] R. Kohavi and F. Provost. Applications of Data Mining to Electronic Commerce. *Data Mining and Knowledge Discovery*, (5):5–10, 2001.

[27] L.-K. Lee and H. Ting. A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 290–297. ACM, 2006.

[28] X. Lin, H. Lu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *null*, page 362. IEEE, 2004.

[29] N. Manerikar and T. Palpanas. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering*, 68(4):415–430, 2009.

[30] A. Metwally, D. Agrawal, and A. El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory*, pages 398–412. Springer, 2005.

[31] J. Misra and D. Gries. Finding repeated elements. Technical report, 1982.

[32] M. Mitzenmacher, T. Steinke, and J. Thaler. Hierarchical heavy hitters with the space saving algorithm. In *2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 160–174. SIAM, 2012.

[33] O. Papapetrou, M. Garofalakis, and A. Deligiannakis. Sketching distributed sliding-window data streams. *The VLDB JournalThe International Journal on Very Large Data Bases*, 24(3):345–368, 2015.