How to Catch L_2-Heavy-Hitters on Sliding Windows

Vladimir Braverman[1], Ran Gelles[2], and Rafail Ostrovsky[3]

[1] Department of Computer Science, Johns Hopkins University, vova@cs.jhu.edu.
[2] Department of Computer Science, University of California, Los Angeles, gelles@cs.ucla.edu.
[3] Department of Computer Science and Mathematics, University of California, Los Angeles, rafail@cs.ucla.edu.

Abstract. Finding heavy-elements (heavy-hitters) in streaming data is one of the central, and well-understood tasks. Despite the importance of this problem, when considering the *sliding windows* model of streaming (where elements eventually expire) the problem of finding L_2-heavy elements has remained completely open despite multiple papers and considerable success in finding L_1-heavy elements.
Since the L_2-heavy element problem doesn't satisfy certain conditions, existing methods for sliding windows algorithms, such as smooth histograms or exponential histograms are not directly applicable to it. In this paper, we develop the first polylogarithmic-memory algorithm for finding L_2-heavy elements in the sliding window model.
Our technique allows us not only to find L_2-heavy elements, but also heavy elements with respect to any L_p with $0 < p \leq 2$ on sliding windows. By this we completely "close the gap" and resolve the question of finding L_p-heavy elements in the sliding window model with polylogarithmic memory, since it is well known that for $p > 2$ this task is impossible.
We demonstrate a broader applicability of our method on two additional examples: we show how to obtain a sliding window approximation of the similarity of two streams, and of the fraction of elements that appear exactly a specified number of times within the window (the α-rarity problem). In these two illustrative examples of our method, we replace the current *expected* memory bounds with *worst case* bounds.

1 Introduction

A *data stream* S is an ordered multiset of elements $\{a_0, a_1, a_2 \ldots\}$ where each element $a_t \in \{1, \ldots, u\}$ arrives at time t. In the *sliding window model* we consider at each time $t \geq N$ the last N elements of the stream, i.e. the window $W = \{a_{t-(N-1)}, \ldots, a_t\}$. These elements are called *active*, whereas elements that arrived prior to the current window $\{a_i \mid 0 \leq i < t-(N-1)\}$ are *expired*. For $t < N$, the window consists of all the elements received so far, $\{a_0, \ldots, a_t\}$.

Usually, both u and N are considered to be extremely large so it is not applicable to save the entire stream (or even one entire window) in memory. The problem is to be able to calculate various characteristics about the window's elements using small amount of memory (usually, polylogarithmic in N and u). We refer the reader to the books of Muthukrishnan [39] and Aggarwal (ed.) [1] for extensive surveys on data stream models and algorithms.

One of the main open problems in data streams deals with the relations between the different streaming models [37], specifically between the unbounded stream model and the sliding window model. In this paper we provide another important step in clarifying the connection between these two models by showing that finding L_p-heavy hitters is just as doable on sliding windows as on the entire stream.

We focus on approximation-algorithms for certain statistical characteristics of the data streams, specifically, finding frequent elements. The problem of finding frequent elements in a stream is useful for many applications, such as network monitoring [42] and DoS prevention [23,18,4], and was extensively explored over the last decade (see [39,17] for a definition of the problem and a survey of existing solutions, as well as [13,36,26,32,16,3,19,44,27]).

We say that an element is *heavy* if it appears more times than a constant fraction of some L_p norm of the stream. Recall that for $p > 0$, the L_p norm of the frequency vector[4] is defined by $L_p = (\sum_i n_i^p)^{1/p}$, where n_i is

[4] Throughout the paper we use the term "L_p norm" to indicate the L_p norm of the frequency vector, i.e., the pth root of the pth frequency moment $F_p = \sum_i n_i^p$ [2], rather than the norm of the data itself.

the frequency of element $i \in [u]$, i.e., the number of times i appears in the window. Since different L_p can be considered, we obtain several different ways to define a "heavy" element. Generally speaking (as mentioned in [30]), when considering frequent elements (heavy-hitters) with respect to L_p, the higher p is, the better. Specifically, identifying frequent elements with respect to L_2 is better than L_1 since an L_1 algorithm can always be replaced with an L_2 algorithm, with less or equal memory consumption (but not vice versa).

Naturally, finding frequent elements with respect to the L_2 norm is a more difficult task (memory-wise) than the equivalent L_1 problem. To demonstrate this fact let us regard the following example: let S be a stream of size N, in which the element a_1 appears $\sqrt{N}$ times, while the rest of the elements $a_2, \ldots, a_{N-\sqrt{N}}$ appear exactly once in S. Say we wish to identify a_1 as an heavy element. Note that $n_1 = \frac{1}{\sqrt{N}} L_1$ while $n_1 = cL_2$, where c is a constant, lower bounded by $\frac{1}{\sqrt{2}}$. Therefore, as N grows, $n_1/L_1 \to 0$ goes to zero, while n_1/L_2 is bounded by a constant. If an algorithm finds elements which are heavier than γL_p with memory $poly(\gamma^{-1}, \log N, \log u)$, then for $p = 2$ we get a polylogarithmic memory, while for $p = 1$ the memory consumption is super-logarithmic.

We focus on solving the following L_2-heaviness problem:

Definition 1.1 ((γ, ϵ)-approximation of L_2-frequent elements). *For $0 < \epsilon, \gamma < 1$, output any element $i \in [u]$ such that $n_i > \gamma L_2$ and no element such that $n_i < (1-\epsilon)\gamma L_2$.*

The L_2 norm is the most powerful norm for which we can expect a polylogarithmic solution, for the frequent-elements problem. This is due to the known lower bound of $\Omega(u^{1-2/p})$ for calculating L_p over a stream [41,6].

There has been a lot of progress on the question of finding L_1-frequent elements, in the sliding window model [3,44,27], however those algorithms cannot be used to find L_2-frequent elements with an efficient memory. In 2002, Charikar, Chen and Farach-Colton [13] developed the CountSketch algorithm that can approximate the "top k" frequent-elements on *an unbounded stream*, where k is given as an input. Formally, their algorithm outputs only elements with frequency larger than $(1-\epsilon)\phi_k$, where ϕ_k is the frequency of the kth most frequent element in the stream, using memory proportional to $L_2^2/(\epsilon\phi_k)^2$. Since the "heaviness" in this case is relative to ϕ_k, and the memory is bounded by the fraction $L_2^2/(\epsilon\phi_k)^2$, Charikar et al.'s algorithm finds in fact heaviness in terms of the L_2 norm. A natural question is whether one can develop an algorithm for finding frequent-elements that appear at least γL_2 times in the *sliding window model*, using $poly(\gamma^{-1}, \log N, \log u)$ memory.

Our Results. We give the first polylogarithmic algorithm for finding an ϵ-approximation of the L_2-frequent elements in the sliding window model. Our algorithm is able to identify elements that appear within the window a number of times which is at least a γ-fraction of the L_2 norm of the window, up to a multiplicative factor of $(1-\epsilon)$. In addition, the algorithm guarantees to output *all* the elements with frequency at least $(1+\epsilon)\gamma L_2$.

Theorem 1.2. *There exists an efficient sliding window algorithm that outputs a (γ, ϵ)-approximation of the L_2-frequent-elements, with probability at least $1-\delta$ and memory $poly(\epsilon^{-1}, \gamma^{-1}, \log N, \log \delta^{-1})$.*

We note that the CountSketch algorithm works in the unbounded model and does not apply directly on sliding windows. Moreover, CountSketch solves a slightly different (yet related) problem, namely, the top-k problem, rather than the L_2 heaviness. To achieve our result on L_2 heavy hitters, we combine in a non-trivial way the scheme of Charikar et al. with a sliding-window approximation for L_2 as given by Braverman and Ostrovsky [9]. Variants of these techniques sufficient to derive similar results were known since 2002,[5] however no algorithm for L_2 heavy hitters was reported despite several papers on L_1 heavy hitters.

Our solution gives another step in the direction of making a connection between the unbounded and sliding window models, as it provides an answer for the very important question of heavy hitters in the sliding window model. The result joins the various solutions of finding L_1-heavy hitters in sliding windows [26,3,40,4,44,27,28], and can be used in various algorithms that require identifying L_2 heavy hitters,

[5] Indeed, we use the algorithm of Charikar et al. [13] that is known since 2002. Also, it is possible to replace (with some non-trivial effort) our smooth histogram method for L_2 computation with the algorithm of Datar, Gionis, Indyk and Motwani [21] for L_2 approximation.

such as [31,8] and others. More generally, our paper resolves the question of finding L_p-heavy elements on sliding windows for all values of p that allows small memory one-pass solutions (i.e. for $0 < p \leq 2$). By this we completely close the gap between the case of $p \leq 1$, solved by previous works, and the impossibility result for the case of $p > 2$.

A Broader Perspective. In fact, one can consider the tools we develop for the frequent elements problem as a general method that allows obtaining a sliding window solution out of an algorithm for the unbounded model, for a wide range of functions. We explain this concept in this section.

Many statistical properties were aggregated into families, and efficient algorithms were designed for those families. For instance, Datar, Gionis, Indyk and Motwani, in their seminal paper [21] showed that a sliding window estimation is easy to achieve for any function which is *weakly-additive* by using a data structure named *exponential histograms* [21]; for certain functions that decay with time, one can maintain time-decaying aggregates [15]; another data structure, named *smooth-histogram* [9] can be used in order to approximate an even larger set of functions, known as *smooth functions*. See [1] for a survey of synopsis construction.

In this paper we introduce a new concept which uses a smooth-histogram in order to perform sliding window approximation of *non-smooth* properties. Informally speaking, the main idea is to relate the non-smooth property f with some other, smooth[6], property g, such that changes in f are bounded by the changes in g. By maintaining a smooth-histogram for the smooth function g, we *partition* the stream into sets of sub-streams (buckets). Due to the properties of the smooth-histogram we can bound the error (of approximating g) for every sub-stream, and thus get an approximation of f. We use the term *semi-smooth* to describe these kinds of algorithms.

We demonstrate the above idea by showing a concrete efficient sliding window algorithm for the properties of rarity and similarity [20]; we stress that neither is smooth (see Section 4 for definitions of these problems). Although there already exist algorithms for these problems with *expected* polylogarithmic memory [20], our techniques improve these results and obtain a *worst case* memory consumption of essentially the same magnitude (up to a factor of $\log\log N$).

In addition to the properties of rarity and similarity, we believe that the tools we develop here can be used to build efficient sliding window approximations for many other (non-smooth) properties and provide a general new method for computing on sliding windows. Indeed, in a subsequent work Tirthapura and Woodruff [43] use our methods to compute various *correlated aggregations*. It is important to note that trying to build a smooth-histogram (or any other known sketch) directly to f will not preserve the required invariants, and the memory consumption might not be efficient.

Previous Works.
Frequent elements. Finding elements that appear many times in the stream ("heavy hitters") is a very central question and thus has been extensively studied both for the unbounded model [22,34,16,38] and for the sliding window model [3,40,44,27] as well as other variants such as the *offline stream* model [36], insertion and deletion model [19,32], finding heavy-distinct-hitter [4], etc. Reducing the processing time was done by [35] into $O(\frac{1}{\epsilon})$ and by [28] into $O(1)$.

Another problem which is related to finding the heavy hitters, is the top-k problem, namely, finding the k most frequent elements. As mentioned above, Charikar, Chen and Farach-Colton [13] provide an algorithm that finds the k most frequent elements in the unbounded model (up to a precision of $1 \pm \epsilon$). Golab, DeHaan, Demaine, López-Ortiz and Munro [26] solve this problem in the *jumping window* model.

Similarity and α-rarity. The similarity problem was defined in order to give a rough estimation of closeness between files over the web [11] (and independently in [14]). Later, it was shown how to use min-hash functions [29] in order to sample from the stream, and estimate the similarity of two streams.

The notion of α-rarity, introduced by Datar and Muthukrishnan [20], is that of finding the fraction of elements that appear exactly α times within the stream. This quantity can be seen as finding the fraction of elements with frequency within certain bounds.

[6] Of course, other kinds of aggregations can be used, however our focus is on smooth histograms.

The questions of rarity and similarity were analyzed, both for the unbounded stream and the sliding window models, by Datar and Muthukrishnan [20], achieving an expected memory bound of $O(\log N + \log u)$ words of space for a constant ϵ, α, δ. At the bit level, their algorithm requires $O(\alpha \cdot \epsilon^{-3} \log \delta^{-1} \log N(\log N + \log u))$ bits for α-rarity and $O(\epsilon^{-3} \log \delta^{-1} \log N(\log N + \log u))$ bits for similarity, with $1 - \delta$ being the probability of success[7].

2 Preliminaries

2.1 Notations

We say that an algorithm A_f is an (ϵ, δ)-approximation of a function f, if for any input S, $(1-\epsilon)f(S) \leq A_f(S) \leq (1+\epsilon)f(S)$, except with probability δ over A_f's coin tosses. We denote this relation as $A_f \in (1 \pm \epsilon)f$ for short. We denote an output of an approximation algorithm with a hat symbol, e.g., the estimator of f is denoted $\hat{f}$.

The set $\{1, 2, \ldots, n\}$ is usually denoted as $[n]$. If a stream B is a suffix of A, we denote $B \subseteq_r A$. For instance, let $A = \{q_1, q_2, \ldots, q_n\}$ then $B = \{q_{n_1}, q_{n_1+1}, \ldots, q_n\} \subseteq_r A$, for $1 \leq n_1 \leq n$. The notation $A \cup C$ denotes the concatenation of the stream $C = \{c_1, c_2, \ldots, c_m\}$ to the end of stream A, i.e., $A \cup C = \{q_1, q_2, \ldots, q_n, c_1, c_2, \ldots c_m\}$. The notation $|A|$ denotes the number of different elements in the stream A, that is the cardinality of the *set* induced by the multiset A. The size of the stream (i.e. of the multiset) A will be denoted as $\|A\|$, e.g., for the example above $\|A\| = n$.

We use the notation $\tilde{O}(\cdot)$ to indicate an asymptotic bound which suppresses terms of magnitude $poly(\log \frac{1}{\epsilon}, \log\log \frac{1}{\delta}, \log\log N, \log\log u)$.

2.2 Smooth histograms

Recently, Braverman and Ostrovsky [9] showed that a function f can be ϵ-approximated in the sliding window model, if f is a *smooth function*, and if it can be calculated (or approximated) in the unbounded stream model. Formally,

Definition 2.1. *A polynomial function f is (α, β)-smooth if it satisfies the following properties: (i) $f(A) \geq 0$; (ii) $f(A) \geq f(B)$ for $B \subseteq_r A$; and (iii) there exist $0 < \beta \leq \alpha < 1$ such that if $(1-\beta)f(A) \leq f(B)$ for $B \subseteq_r A$, then $(1-\alpha)f(A \cup C) \leq f(B \cup C)$ for any C.*

If an (α, β)-smooth f can be calculated (or (ϵ, δ)-approximated) on an unbounded stream with memory $g(\epsilon, \delta)$, then there exists an $(\alpha+\epsilon, \delta)$-estimation of f in the sliding window model using $O(\frac{1}{\beta} \log N(g(\epsilon, \frac{\delta\beta}{\log N}) + \log N))$ bits [9].

The key idea is to construct a "smooth-histogram", a structure that contains estimations on $O(\frac{1}{\beta} \log N)$-suffixes of the stream, $A_1 \supseteq_r A_2 \supseteq_r \ldots \supseteq_r A_{c\frac{1}{\beta}\log(n)}$. Each suffix A_i is called *a Bucket*. Each new element in the stream initiates a new bucket, however adjacent buckets with a close estimation value are removed (keeping only one representative). Since the function is "smooth", i.e., monotonic and slowly-changing, it is enough to save $O(\frac{1}{\beta} \log N)$ buckets in order to maintain a reasonable approximation of the window. At any given time, the current window W is between buckets A_1 and A_2, i.e. $A_1 \supseteq_r W \supseteq_r A_2$. Once the window "slides" and the first element of A_2 expires, we delete the bucket A_1 and renumber the indices so that A_2 becomes the new A_1, A_3 becomes the new A_2, etc. We use the estimated value of bucket A_1 to estimate the value of the current window. The relation between the value of f on the window and on the first bucket is given by $(1-\alpha)f(A_1) \leq f(A_2) \leq f(W) \leq f(A_1)$.

[7] These bounds are not explicitly stated in [20], but follow from the analysis (see Lemma 1 and Lemma 2 in [20]).

3 A Semi-Smooth Estimation of Frequent Elements

In this section we develop an efficient semi-smooth algorithm for finding elements that occur frequently within the window. Let n_i be *the frequency* of element $i \in \{1, \ldots, u\}$, i.e., the number of times i appears in the window. The *first frequency norm* and the *second frequency norm* of the window are defined by $L_1 = \sum_{i=1}^{u} n_i = N$ and $L_2 = \left(\sum_{i=1}^{u} n_i^2\right)^{\frac{1}{2}}$. In many previous works, (e.g., [16,3,39,44,27]) the task of finding heavy-elements is defined using the L_1 norm as follows,

Definition 3.1 ((γ, ϵ)-approximation of L_1-heavy hitters). *Output any element $i \in [u]$ such that $n_i \geq \gamma L_1$ and no element such that $n_i \leq (1-\epsilon)\gamma L_1$.*

Our notion of approximating *frequent elements* is given by Definition 1.1. An equivalent definition which we use in our proof is the following:

Definition 3.2. *For $0 < \epsilon, \gamma < 1$, output all elements $i \in [u]$ with frequency higher than $(1+\epsilon)\gamma L_2$, and do not output any element with frequency lower than $(1-\epsilon)\gamma L_2$.*

Observe that the L_2 approximation is stronger than the above L_1 definition. If an element is heavy in terms of L_1 norm, it is also heavy in terms of the L_2 norm,

$$n_i \geq \gamma L_1 = \gamma \sum_j n_j \quad \Longrightarrow \quad n_i^2 \geq \gamma^2 \Big(\sum_j n_j\Big)^2 \geq \gamma^2 \sum_j n_j^2 = (\gamma L_2)^2 \ ,$$

while the opposite direction does not apply in general.

In order to identify the frequent elements in the current window, use a variant of the CountSketch algorithm of Charikar et al. [13], which provides an ϵ-approximation (in the unbounded stream model) for the following top-frequent approximation problem.

Definition 3.3 ((k, ϵ)-top frequent approximation). *Output a list of k elements such that every element i in the output has a frequency $n_i > (1-\epsilon)\phi_k$, where ϕ_k is the frequency of the k-th most frequent element in the stream.*

The CountSketch algorithm guarantees that any element that satisfies $n_i > (1+\epsilon)\phi_k$, appears in the output. This algorithm runs on a stream of size n and succeeds with probability at least $1-\delta$, and memory complexity of $O\left(\left(k + \frac{1}{(\epsilon\gamma)^2}\right) \log \frac{n}{\delta}\right)$, for every $\delta > 0$, given that $\phi_k \geq \gamma L_2$.

Definition 3.3 and Definition 1.1 do not describe the same problem, yet they are strongly connected. In fact, our method allows solving the frequent elements problem under both definitions, however in this paper we focus on solving the L_2-frequent-elements problem, as defined by Definition 3.2. In order to do so, we use a variant of the CountSketch algorithm with specific parameters tailored for our problem (See full details in Appendix A). This variant outputs a list of elements, and is guaranteed to output every element with frequency at least $(1+\epsilon')\gamma L_2$ and no element of frequency less than $(1-\epsilon')\gamma L_2$, for an input parameter ϵ'.

We stress that CountSketch is not sufficient on its own to prove Theorem 1.2. The main reason is that this algorithm works in the unbounded stream model, rather than in the sliding window model. Another reason is that it must be tweaked in order not to output false positives. Our solution below makes a use of smooth-histograms to overcome these issues.

3.1 Semi-smooth algorithm for frequent elements approximation

We construct a smooth-histogram for the L_2 norm, and partition the stream into buckets accordingly. It is known that the L_2 property is a $(\epsilon, \frac{\epsilon^2}{2})$-smooth function [9]. Using the method of Charikar et al. [13], separately on each bucket, with a careful choice of parameters, we are able to approximate the (γ, ϵ)-frequent elements problem on a sliding window (Fig. 1).

Theorem 3.4. *The semi-smooth algorithm* **ApproxFreqElements** *(Fig. 1) is a $(\gamma, O(\epsilon))$-approximation of the L_2-frequent elements problem, with success probability at least $1-\delta$.*

ApproxFreqElements$(\gamma, \epsilon, \delta)$

1. Maintain an $(\frac{\epsilon}{2}, \frac{\delta}{2})$-estimation of the L_2 norm of the window, using a smooth-histogram.
2. For each bucket of the smooth-histogram, $A_1, A_2, \ldots$ maintain an approximated list of the $k = \frac{1}{\gamma^2} + 1$ most frequent elements, by running $(\gamma, \frac{\epsilon}{4}, \frac{\delta}{2})-$CountSketch$_b$.
(see CountSketch$_b$'s description in Appendix A).
3. Let $\hat{L}_2$ be the approximated value of the L_2 norm of the current window W, as given by the the smooth-histogram. Let $q_1, \ldots, q_k \in \{1, \ldots, u\}$ be the list of the k most heavy elements in A_1, along with $\hat{n}_1, \ldots, \hat{n}_k$ their estimated frequencies, as outputted by CountSketch$_b$.
4. Output any element q_i that satisfies $\hat{n}_i > \frac{1}{1+\epsilon}\gamma\hat{L}_2$.

Fig. 1. A semi-smooth algorithm for the frequent elements problem

Proof. Recall that the smooth-histogram data structure of the L_2 guarantees us an estimation $\hat{L}_2$ which is $(1 \pm \epsilon)L_2(W)$; in addition there exists some α such that $(1-\alpha)L_2(A_1) \le L_2(W) \le L_2(A_1)$. In our case the inequality is satisfied for $\alpha = \epsilon/2$ (see Theorem 3 and Definition 3 in [9]). Any element j with frequency $n_j(W) > (1+\epsilon)\gamma L_2(W)$ satisfies

$$n_j(A_1) \ge n_j(W) \ge (1+\epsilon)\gamma L_2(W) \ge (1+\epsilon)(1-\epsilon/2)\gamma L_2(A_1) \ ,$$

and will be added to the output list in Step 2, since Proposition A.3 guarantees that any element i such that $n_i(A_i) > (1+\epsilon/4)\gamma L_2(A_1)$ is identified by CountSketch$_b$ (assuming $\epsilon < \frac{1}{2}$).

In order to show that all of the required elements survive Step 4, we use Lemma A.2 to bound the estimated frequency $\hat{n}_i$ reported by CountSketch$_b$, and show it is above the required threshold. If $n_i(W) > (1+\epsilon)\gamma L_2(W)$ then

$$\hat{n}_i(A) > n_i(A) - \frac{\epsilon}{8}\gamma L_2(A) > n_i(W) - \frac{\epsilon}{2-\epsilon}L_2(W) > \left[1 + \epsilon - \frac{\epsilon}{2-\epsilon}\right]\gamma L_2(W) \ ,$$

recalling that $\hat{L}_2 < (1+\epsilon)L_2(W)$ implies that the element survives Step 4.

While we are guaranteed that all the $(1+\epsilon)\gamma L_2(W)$-frequent elements appear in the output list, it might contain other elements which are not heavy enough. We now prove that Step 4 eliminates any element of frequency less than $(1-c\epsilon)\gamma L_2(W)$, for a constant c.

Lemma 3.5. *If for an element i there exists some $\zeta > \sqrt{\epsilon}$ such that $n_i(A_1) > \zeta L_2(A_1)$, then there exist a constant $\xi > 0$ such that $n_i(W) > \xi L_2(W)$.*

Proof. By the properties of the smooth-histogram,

$$L_2(W)^2 > (1-\epsilon/2)^2 L_2(A_1)^2 > (1-\epsilon)L_2(A_1)^2$$
$$n_i(W)^2 + \sum_{j\neq i} n_j(W)^2 > n_i(A_1)^2 + \sum_{j\neq i} n_j(A_1)^2 - \epsilon L_2(A_1)^2$$
$$n_i(W)^2 > n_i(A_1)^2 - \epsilon L_2(A_1)^2 > (\zeta^2 - \epsilon)L_2(A_1)^2$$

and $n_i(W) > \xi L_2(W)$ for $\xi \le \sqrt{(\zeta^2 - \epsilon)}$. □

Suppose some element i survives Step 4, then $\hat{n}_i(A_1) > \frac{1}{1+\epsilon}\gamma\hat{L}_2 > \frac{1-\epsilon}{1+\epsilon}\gamma L_2(W)$. By Lemma A.2,

$$n_i(A_1) \ge \hat{n}_i(A_1) - \frac{\epsilon}{8}\gamma L_2(A_1) \ge \left(\frac{(1-\epsilon)(1-\frac{\epsilon}{2})}{1+\epsilon} - \frac{\epsilon}{8}\right)\gamma L_2(A_1) > (1-3\epsilon)\gamma L_2(A_1),$$

and by Lemma 3.5, $n_i(W) \ge \sqrt{1-7\epsilon} \cdot \gamma L_2(W)$. This proves that for small enough ϵ there exists some constant c such that the algorithm doesn't output any element with frequency lower than $(1-c\epsilon)\gamma L_2(W)$.

To conclude, except for probability $\delta/2$ we are able to partition the stream into L_2-smooth buckets, and except for probability $\delta/2$, the CountSketch$_b$ algorithm outputs a list which can be used to identify the frequent elements of the window. Using a union bound we conclude that the entire algorithm succeeds except with probability δ. This completes the proof of the theorem. □

Memory Usage. The memory usage of the protocol is composed of two parts: maintaining a $(\epsilon/2, \delta/2)$-smooth-histogram of L_2, and running CountSketch$_b$ on each of the buckets. According to [9] (corollary 5), maintaining a smooth-histogram for L_2 can be done with memory

$$O\left(\tfrac{1}{\epsilon^2}\log^2 N + \tfrac{1}{\epsilon^4}\log N \log\frac{\log N}{\delta\epsilon}\right)$$

for a relative error of $\epsilon/2 + \epsilon^2/8$, with success probability at least $1 - \delta/2$. For small enough ϵ we have $\epsilon/2 + \epsilon^2/8 < \epsilon$ as required.

As for the second part, recall that one instance of CountSketch$_b$ requires a memory of $O\big(\frac{1}{\epsilon^2\gamma^2}\log\frac{n}{\delta}\big)$ (see Appendix A), where n is the size of the input. In our case the maximal size of the input is the size of the first bucket, $\|A_1\|$. Note that $\log\|A_1\| = O(\log N)$ since $(1-\alpha)L_2(A_1) \le L_2(W) \le N$. The number of CountSketch$_b$ instances is bounded by the number of buckets, $O(\frac{1}{\epsilon^2}\log N)$ [9], which leads to a total memory bound of

$$O\left(\frac{1}{\gamma^2\epsilon^4}\log N \log\frac{N}{\delta} + \frac{1}{\epsilon^4}\log N \log\frac{1}{\epsilon}\right) .$$

3.2 Extensions to any L_p with $p < 2$

It is easy to see that the same method can be used in order to approximate L_p-heavy elements for any $0 < p < 2$, up to a $1 \pm \epsilon$ precision. The algorithms and analysis remain the same, except for using a smooth-histogram for the L_p norm, and changing the parameters by constants.

Theorem 3.6. *For any $p \in (0, 2]$, there exists a sliding window algorithm that outputs all the elements with frequency at least $(1+\epsilon)\gamma L_p$, and no element with frequency less then $(1-\epsilon)\gamma L_p$. The algorithm succeeds with probability at least $1-\delta$ and takes poly$(\epsilon^{-1}, \gamma^{-1}, \log N, \log\delta^{-1})$ memory.*

4 Estimation of Non-Smooth Properties Relativized to the Number of Distinct Elements

In this section we extend the method shown above and apply it to other non-smooth functions. In contrast to the smooth L_2 used above, in this section we use a different smooth function to partition the stream, namely the distinct elements count problem. This allows us to obtain efficient semi-smooth approximations for the (non-smooth) *similarity* and *α-rarity* tasks.

4.1 Preliminaries

We now show that counting the number of distinct elements in a stream is smooth. This allows us to partition the stream into a smooth-histogram structure, where each two adjacent buckets have approximately the same number of distinct elements.

Proposition 4.1. *Define $\mathrm{DEC}(A)$ as the number of distinct elements in the stream A, i.e., $\mathrm{DEC}(A) = |A|$. The function DEC is an (ϵ, ϵ)-smooth-function, for every $0 \le \epsilon \le 1$.*

Proof. Properties (i) and (ii) of Definition 2.1 follow directly from DEC's definition. As for property (iii), assume that $B \subseteq_r A$ and $(1-\epsilon)\,\mathrm{DEC}(A) \le \mathrm{DEC}(B)$, then

$$\begin{aligned}(1-\epsilon)\,\mathrm{DEC}(A \cup C) &= (1-\epsilon)\left[\mathrm{DEC}(A) + \mathrm{DEC}(C \setminus A)\right] \\ &\le \mathrm{DEC}(B) + (1-\epsilon)\,\mathrm{DEC}(C \setminus A) \\ &\le \mathrm{DEC}(B) + \mathrm{DEC}(C \setminus B) \\ &= \mathrm{DEC}(B \cup C),\end{aligned}$$

where "$A \setminus B$" represents the set of all the elements in A which are not in B. ☐

There have been many works on counting distinct elements in streams, initiated by Flajolet and Martin [24], and later improved by many others [2,25,7,5]. Recently, Kane, Nelson and Woodruff provided an optimal algorithm for (ϵ, δ)-approximating the number of distinct elements [33], using $O((\frac{1}{\epsilon^2} + \log u)\log\frac{1}{\delta})$ bits and $O(1)$ time. We use the method of Kane et al. in order to construct a smooth-histogram for the distinct elements count with memory $\tilde{O}\big((\log u + \frac{1}{\epsilon^2})\frac{1}{\epsilon}\log N \log\frac{1}{\delta} + \frac{1}{\epsilon}\log^2 N\big)$, suppressing $\log\log N$ and $\log\frac{1}{\epsilon}$ terms.

Another tool we use is *min-wise hash functions* [12,10], used in various algorithms in order to estimate different characteristics of data streams, especially the *similarity* of two streams [12]. Informally speaking, these functions have a meaning of uniformly sampling an element from the stream, which makes them a very useful tool.

Definition 4.2 (min-hash). *Let $\Pi = \{\pi_i\}$ be a family of permutations over $[u] = \{1, \dots, u\}$. For a subset $A \subseteq [u]$ define h_i to be the minimal permuted value of π_i over A, $h_i = \min_{a \in A} \pi_i(a)$. A family $\{h_i\}$ of such functions is called exact min-wise independent hash functions (or min-hash) if for any subset $A \subseteq [u]$ and $a \in A$,*

$$\Pr_i\left[h_i(A) = \pi_i(a)\right] = \frac{1}{|A|}.$$

The family $\{h_i\}$ is called ϵ-approximated min-wise independent hash functions (or ϵ-min-hash) if for any subset $A \subseteq [u]$ and $a \in A$,

$$\Pr_i\left[h_i(A) = \pi_i(a)\right] \in \frac{1}{|A|}(1 \pm \epsilon).$$

A specific construction of ϵ-min-hash functions was presented by Indyk [29], using only $O(\log\frac{1}{\epsilon}\log u)$ bits. The time per hash calculation is bounded by $O(\log\frac{1}{\epsilon})$. Min-hash functions can be used in order to estimate the similarity of two sets, by using the following lemma,

Lemma 4.3. ([10]. See also [20].) *For any two sets A and W and an ϵ'-min-hash function h_i, it holds that $\Pr_i\left[h_i(A) = h_i(W)\right] = \frac{|A \cap W|}{|A \cup W|} \pm \epsilon'$.*

4.2 A semi-smooth estimation of α-rarity

In the following section we present an algorithm that estimates the α-*rarity* of a stream (in the sliding window model), i.e., the ratio of elements that appear exactly α times in the window. The rarity property is known not to be smooth, yet by using a smooth-histogram for distinct elements count, we are able to partition the stream into $O(\frac{1}{\epsilon}\log N)$ buckets, and estimate the α-rarity in each bucket.

Definition 4.4. *An element x is* α-rare *if it appears exactly α times in the stream. The α-rarity measure, ρ_α, denotes the ratio of α-rare elements in the entire stream S, i.e., $\rho_\alpha = \frac{|\{x \mid x \text{ is } \alpha\text{-rare in } S\}|}{\mathrm{DEC}(S)}$.*

Our algorithm follows the method used by [20] to estimate α-rarity in the unbounded model. The estimation is based on the fact that the α-rarity is equal to the portion of min-hash functions that their min-value appears exactly α times in the stream.

However, in order to estimate rarity over sliding windows, one needs to estimate the ratio of min-hash functions of which the min-value appears exactly α times *within the window*. Our algorithm builds a smooth-histogram for DEC in order to partition the stream into buckets, such that each two consecutive buckets have approximately the same number of distinct elements. In addition, we sample the bucket using a min-wise hash, and count the $\alpha+1$ last occurrences of the sampled element x_i in the bucket. We estimate the α-rarity of the window by calculating the fraction of min-hash functions of which the appropriate min-value x_i appears exactly α times *within the window*. Due to feasibility reasons we use approximated min-wise hashes, and prove that this estimation is an ϵ-approximation of the α-rarity of the current window (up to a pre-specified additive precision). The *semi-smooth* algorithm **ApproxRarity** for α-rarity is defined in Fig. 2.

ApproxRarity(ϵ, δ)

1. Randomly choose k $\frac{\epsilon}{2}$-min-hash functions $h_1, h_2, \ldots, h_k$.
2. Maintain an $(\epsilon, \frac{\delta}{2})$-estimation of the number of distinct elements by building a smooth histogram.
3. For every bucket instance A_j of the smooth-histogram and for each one of the hash functions h_i, $i \in [k]$
   (a) maintain the value of the min-hash function h_i over the bucket, $h_i(A_j)$
   (b) maintain a list $L_i(A_j)$ of the most recent $\alpha+1$ occurrences of $h_i(A_j)$ in A_j
   (c) whenever the value $h_i(A_j)$ changes, re-initialize the list $L_i(A_j)$, and continue maintaining the occurrences of the new value $h_i(A_j)$.
4. Output $\hat{\rho}_\alpha$, the ratio of the min-hash functions h_i, which has exactly α *active* elements in $L_i(A_1)$, i.e. the ratio
$$\hat{\rho}_\alpha = |\{i \text{ s.t. } L_i(A_1) \text{ consists exactly } \alpha \text{ active elements}\}|/k \ .$$

Fig. 2. Semi-smooth algorithm for α-rarity

The **ApproxRarity** algorithm provides an (ϵ, δ)-approximation for the α-rarity problem, up to an additive error of ϵ. As proven by Datar et al. [20], the ratio of min-hash functions that have exactly α active elements in the window is an estimation of ρ_α. This is true even when using the min-value of the inclusive bucket A_1 rather than the min-value of the current windows W.

Theorem 4.5. *The semi-smooth algorithm (Fig. 2) is an* (ϵ, δ)*-approximation for the* α*-rarity problem, up to an additive precision.*

Proof. For the sake of simplicity we treat the multisets A_1, W, etc., as sets. Let R_α be the set of elements which are α-rare in the window W. Following Lemma 4.3, with $R_\alpha \subseteq A_1$,

$$\Pr[h_i(A_1) = h_i(R_\alpha)] = \frac{|R_\alpha \cap A_1|}{|R_\alpha \cup A_1|} \pm \frac{\epsilon}{2} = \frac{|R_\alpha|}{|A_1|} \pm \frac{\epsilon}{2}.$$

The algorithm outputs an approximation of $\Pr\left[L_i(A_1) \text{ consists of exactly } \alpha \text{ active elements}\right]$, which equals to $\Pr[h_i(A_1) = h_i(R_\alpha)]$, since $h_i(A_1) = h_i(R_\alpha)$ if and only if $L_i(A_1)$ consists of α active elements. Let x_i be the element which minimizes h_i on A_1, $h(x_i) = h(A_1)$. If the number of active elements in $L_i(A_1)$ is not α, then $x_i \notin R_\alpha$, thus $h(A_1) \neq h(R_\alpha)$. For the other direction, if $h_i(A_1) = h_i(R_\alpha)$ then $L_1(A_1)$ counts the number of occurrences of x_i in the bucket, and since $x_i \in R_\alpha$, it appears exactly α times within the window.

We build a smooth-histogram for DEC by using the algorithm of Kane et al. [33] as an approximation of DEC for the unbounded model (see Theorem 3 in [9]). The smooth-histogram guarantees[8] that $(1-\epsilon)|A_1| \leq |W| \leq |A_1|$, thus

$$\frac{|R_\alpha|}{|A_1|} \leq \frac{|R_\alpha|}{|W|} = \rho_\alpha \qquad , \qquad \frac{|R_\alpha|}{|A_1|} \geq (1-\epsilon)\frac{|R_\alpha|}{|W|} \geq (1-\epsilon)\rho_\alpha \ .$$

[8] Actually, it guarantees even a better bound, specifically, $(1-\frac{\epsilon}{2})|A_1| \leq |W| \leq |A_1|$.

Therefore, estimating the ratio ρ_α using k hash functions results with a value $(1 \pm \epsilon)\rho_\alpha \pm \frac{\epsilon}{2}$ up to some additive error ϵ' determined by k. Finally, using Chernoff's inequality we can bound the additive error so that $\epsilon' < \frac{\epsilon}{2}$, except for probability $\frac{\delta}{2}$. In order to achieve the desired precision we require $k = \Omega(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$, and the estimation satisfies

$$\hat{\rho}_\alpha \in (1 \pm \epsilon)\rho_\alpha \pm \epsilon ,$$

except for probability at most δ. This concludes the correctness of the algorithm. □

Memory Usage. The memory consumption of the **ApproxRarity** algorithm is as follows. Maintaining a smooth histogram for DEC is done using the method of Kane et al. [33] as the underlying algorithm for DEC in the unbounded model, with memory $\tilde{O}\big((\log u + \frac{1}{\epsilon^2})\frac{1}{\epsilon} \log N \log \frac{1}{\delta} + \frac{1}{\epsilon} \log^2 N\big)$; k seeds for the $\frac{\epsilon}{2}$-min-hash functions: $O(k \log \frac{1}{\epsilon} \log u)$; Saving a list L_i and a value h_i for each bucket A_j and for $i \in [k]$: $O([\log u + \alpha \log N] \frac{k}{\epsilon} \log N)$.

We note that this improves the *expected* memory bound of Datar et al. [20] into a *worst case* bound of the same magnitude (up to a $\log \log N$ term). In most of the practical cases $\log u$ and $\log N$ are very close, and we can assume that $\log u = O(\log N)$. In that case, the space complexity is $\tilde{O}\left(\frac{k}{\epsilon} \alpha \log^2 N\right)$ bits, with $k = \Omega(\frac{1}{\epsilon^2} \log \frac{1}{\delta})$, and the time complexity is $\tilde{O}\left(\frac{k\alpha}{\epsilon} \log N\right)$ calculations per element, suppressing $poly(\log \frac{1}{\epsilon}, \log \log N)$ terms.

4.3 A semi-smooth estimation of streams similarity

In this section we present an algorithm for calculating the *similarity* of two streams X and Y. As in the case of the rarity, the similarity property is known not to be smooth, however we are able to design a semi-smooth algorithm that estimates it. We maintain a smooth-histogram of the distinct elements count in order to partition each of the streams, and sample each bucket of this partition using a min-hash function. We compare the ratio of sample agreements in order to estimate the similarity of the two streams.

Definition 4.6. *The (Jaccard)* similarity *of two streams, X and Y is given by $S(X,Y) = \frac{|X \cap Y|}{|X \cup Y|}$.*

Recall that for two streams X and Y, a reasonable estimation of $S(X,Y)$ is given by the number of min-hash values they agree on [20]. In other words, let $h_1, h_2, \ldots, h_k$ be a family of ϵ-min hash functions and let

$$\hat{S}(X,Y) = |\{i \in [k] \text{ s.t. } h_i(X) = h_i(Y)\}| / k ,$$

then $\hat{S}(X,Y) \in (1 \pm \epsilon)S(X,Y) + \epsilon(1+p)$, with success probability at least $1 - \delta$, where p and δ are determined by k. Based on this fact, Datar et al. [20] showed an algorithm for estimating similarity in the sliding window model, that uses expected memory of $O(k(\log \frac{1}{\epsilon} + \log N))$ words with $k = \Omega(\frac{1}{\epsilon^3 p} \log \frac{1}{\delta})$. Using smooth-histograms, our algorithm reduces the expected memory bound into a worst-case bound. The semi-smooth algorithm **ApproxSimilarity** is rather straightforward and is given in Fig. 3.

Theorem 4.7. *The semi-smooth algorithm for estimating similarity (Fig. 3), is an (ϵ, δ)-approximation for the similarity problem, up to an additive precision.*

Proof. Following Lemma 4.3,

$$Pr[h_i(A_X) = h_i(A_Y)] = \frac{|A_X \cap A_Y|}{|A_X \cup A_Y|} \pm \epsilon'.$$

For convenience, once again we treat buckets A_X, A_Y, W_X, W_Y as sets. Notice that we can write $A_X = W_X \cup (A_X \setminus W_X)$ and that $0 \le |A_X \setminus W_X| \le \frac{\epsilon'}{1-\epsilon'}|W_X|$, which follows from the guarantee of the smooth-histogram that $(1-\epsilon')|A_X| \le |W_X| \le |A_X|$ (and same for A_Y and W_Y). Using elementary set operations,

ApproxSimilarity(ϵ, δ)

1. Randomly choose k ϵ'-min-hash functions, $h_1, \ldots, h_k$. The constant ϵ' will be specified later, as a function of the desired precision ϵ.
2. For each stream (X and Y) maintain an $(\epsilon', \frac{\delta}{2})$-estimation of the number of distinct elements by building a smooth histogram.
3. For each stream and for each bucket instance $A_1, A_2, \ldots$, separately calculate the values of each of the min-hash functions h_i, $i = 1 \ldots k$.
4. Let A_X (A_Y) be the first smooth-histogram bucket that includes the current window W_X (W_Y) of the stream X (Y). Output the ratio of hash-functions h_i which agree on the minimal value, i.e.,

$$\hat{\sigma}(W_X, W_Y) = |\{i \in [k] \text{ s.t. } h_i(A_X) = h_i(A_Y)\}| / k \ .$$

Fig. 3. A semi-smooth algorithm for estimating similarity

we can estimate $|W_X \cup W_Y|$ using $|A_X \cup A_Y|$,

$$\begin{aligned} |W_X \cup W_Y| \leq |A_X \cup A_Y| &\leq |W_X \cup W_Y| + \frac{\epsilon'}{1-\epsilon'}|W_X| + \frac{\epsilon'}{1-\epsilon'}|W_Y| \\ &\leq |W_X \cup W_Y| + 2\frac{\epsilon'}{1-\epsilon'}|W_X \cup W_Y| \\ &= \frac{1+\epsilon'}{1-\epsilon'}|W_X \cup W_Y| \ . \end{aligned}$$

In addition, any two sets S, Q always satisfy $\frac{|S \cap Q|}{|S \cup Q|} = \frac{|S|+|Q|}{|S \cup Q|} - 1$, thus the similarity estimation satisfies

$$\frac{|A_X \cap A_Y|}{|A_X \cup A_Y|} = \frac{|A_X| + |A_Y|}{|A_X \cup A_Y|} - 1 \leq \frac{\frac{1}{1-\epsilon'}|W_X| + \frac{1}{1-\epsilon'}|W_Y|}{|W_X \cup W_Y|} - 1 = \frac{1}{1-\epsilon'}\frac{|W_X \cap W_Y|}{|W_X \cup W_Y|} + \frac{\epsilon'}{1-\epsilon'}, \quad \text{and}$$

$$\frac{|A_X \cap A_Y|}{|A_X \cup A_Y|} \geq \frac{|W_X \cap W_Y|}{\frac{1+\epsilon'}{1-\epsilon'}|W_X \cup W_Y|} = \frac{1-\epsilon'}{1+\epsilon'}\frac{|W_X \cap W_Y|}{|W_X \cup W_Y|},$$

Finally, setting $\epsilon' \leq \epsilon/2$ gives an estimation $\hat{\sigma}(W_X, W_Y) \in (1 \pm \epsilon)S(W_X, W_Y) \pm \epsilon$, up to an additional additive error, which can be arbitrarily decreased using Chernoff's bound, by increasing k. Specifically, this additional error is bounded by $O(\epsilon)$ when $k = \Omega(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$, with success probability at least $1 - O(\delta)$. ☐

Memory Usage. Let us summarize the memory consumption of the **ApproxSimilarity** algorithm. Maintaining a smooth histogram for DEC: $\tilde{O}\big((\log u + \frac{1}{\epsilon^2})\frac{1}{\epsilon}\log N \log\frac{1}{\delta} + \frac{1}{\epsilon}\log^2 N\big)$; k seeds for $\epsilon/2$-min-hash functions: $O(k \log\frac{1}{\epsilon}\log u)$; Keeping the hash value for each h_i: $O\big(k\frac{1}{\epsilon}\log N \log u\big)$.

Our algorithm improves the currently known *expected* bound [20] into a *worst case* bound of the same magnitude (up to a $\log\log N$ term). Taking $k = \Omega(\frac{1}{\epsilon^2}\log\frac{1}{\delta})$ and assuming $\log u = O(\log N)$, we achieve a memory bound of $\tilde{O}\big(k\frac{1}{\epsilon}\log^2 N\big)$, with $\tilde{O}(k\frac{1}{\epsilon}\log N)$ calculations per element, suppressing $poly(\log\frac{1}{\epsilon}, \log\log N)$ elements.

5 Conclusions

We have shown the first polylogarithmic algorithm for identifying L_2 heavy-hitters up to $1 \pm \epsilon$ precision, over sliding windows. Our result supplies another insight about the relations between the unbounded and sliding window models, for the central question of heavy-hitters. As the L_p-heavy-hitters problem is more difficult for larger p, and for $p > 2$ there cannot exist a polylogarithmic solution, our algorithm provides a small-memory solution for the "strongest" L_p norm.

Although our main concern was the L_2 norm, the algorithm can easily be extended for any L_p with $0 < p \leq 2$. Moreover, a polylogarithmic approximation of the top-k problem in sliding window is immediate using our methods.

The tools shown in this paper can be applied to many other properties, if there exists a smooth function which is correlated to the target function. We have shown how to employ the same techniques in order to obtain an efficient sliding window algorithm for the similarity and α-rarity problems, with essentially the same memory consumption as the current state of the art, however, our bound applies for the *worst case* rather than holds only in expectation. We believe that our method can be used to improve the memory efficiency of many other sliding-window algorithms for non-smooth properties.

Acknowledgments

V.B. is supported in part by DARPA grant N660001-1-2-4014. R.O. is supported in part by NSF grants CNS-0830803; CCF-0916574; IIS-1065276; CCF-1016540; CNS-1118126; CNS-1136174; US-Israel BSF grant 2008411, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation Research Award. This material is also based upon work supported by the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-1-0392. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

1. Aggarwal, C.C.: Data streams: models and algorithms. Springer-Verlag New York Inc (2007)
2. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. Journal of Computer and System Sciences 58(1), 137 – 147 (1999)
3. Arasu, A., Manku, G.S.: Approximate counts and quantiles over sliding windows. In: PODS '04. pp. 286–296. ACM, New York, NY, USA (June 2004)
4. Bandi, N., Agrawal, D., Abbadi, A.E.: Fast algorithms for heavy distinct hitters using associative memories. Distributed Computing Systems, International Conference on p. 6 (June 2007)
5. Bar-Yossef, Z., Jayram, T., Kumar, R., Sivakumar, D., Trevisan, L.: Counting distinct elements in a data stream. Lecture Notes in Computer Science 2483, 1–10 (2002)
6. Bar-Yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D.: An information statistics approach to data stream and communication complexity. In: FOCS '02. pp. 209–218. IEEE Computer Society, Washington, DC, USA (2002)
7. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: SODA '02. pp. 623–632. Philadelphia, PA, USA (2002)
8. Bhuvanagiri, L., Ganguly, S., Kesh, D., Saha, C.: Simpler algorithm for estimating frequency moments of data streams. In: SODA '06. pp. 708–713. ACM, New York, NY, USA (2006)
9. Braverman, V., Ostrovsky, R.: Smooth histograms for sliding windows. In: FOCS '07. pp. 283–293. IEEE Computer Society (2007)
10. Broder, A.Z., Charikar, M., Frieze, A.M., Mitzenmacher, M.: Min-wise independent permutations. Journal of Computer and System Sciences 60(3), 630 – 659 (2000)
11. Broder, A.Z., Glassman, S.C., Manasse, M.S., Zweig, G.: Syntactic clustering of the web. Computer Networks and ISDN Systems 29(8-13), 1157 – 1166 (1997), papers from the Sixth International World Wide Web Conference
12. Broder, A.: On the resemblance and containment of documents. In: Compression and Complexity of Sequences 1997. Proceedings. pp. 21–29 (Jun 1997)
13. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. Automata, Languages and Programming pp. 784–784 (2002)
14. Cohen, E.: Size-estimation framework with applications to transitive closure and reachability,. Journal of Computer and System Sciences 55(3), 441 – 453 (1997)
15. Cohen, E., Strauss, M.J.: Maintaining time-decaying stream aggregates. Journal of Algorithms 59(1), 19 – 36 (2006)

16. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. LATIN 2004: Theoretical Informatics pp. 29–38 (April 2004)
17. Cormode, G., Hadjieleftheriou, M.: Finding frequent items in data streams. Proc. VLDB Endow. 1(2), 1530–1541 (september 2008)
18. Cormode, G., Korn, F., Muthukrishnan, S., Srivastava, D.: Finding hierarchical heavy hitters in data streams. In: VLDB '2003: Proceedings of the 29th international conference on Very large data bases. pp. 464–475. VLDB Endowment (september 2003)
19. Cormode, G., Muthukrishnan, S.: What's hot and what's not: tracking most frequent items dynamically. ACM Trans. Database Syst. 30(1), 249–278 (2005)
20. Datar, M., Muthukrishnan, S.: Estimating rarity and similarity over data stream windows. Lecture notes in computer science pp. 323–334 (2002)
21. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows: (extended abstract). In: SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms. pp. 635–644. Philadelphia, PA, USA (2002)
22. Demaine, E., López-Ortiz, A., Munro, J.: Frequency estimation of internet packet streams with limited space. In: Möhring, R., Raman, R. (eds.) Algorithms ESA 2002, LNCS, vol. 2461, pp. 11–20. Springer, Berlin (2002)
23. Estan, C., Varghese, G.: New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. ACM Trans. Comput. Syst. 21(3), 270–313 (august 2003)
24. Flajolet, P., Martin, G.N.: Probabilistic counting. FOCS '83 pp. 76–82 (1983)
25. Gibbons, P.B., Tirthapura, S.: Estimating simple functions on the union of data streams. In: SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures. pp. 281–291. ACM, New York, NY, USA (2001)
26. Golab, L., DeHaan, D., Demaine, E.D., López-Ortiz, A., Munro, J.I.: Identifying frequent items in sliding windows over on-line packet streams. In: IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement. pp. 173–178. ACM, New York, NY, USA (2003)
27. Hung, R., Ting, H.: Finding heavy hitters over the sliding window of a weighted data stream. LATIN 2008: Theoretical Informatics pp. 699–710 (April 2008)
28. Hung, R.Y., Lee, L.K., Ting, H.: Finding frequent items over sliding windows with constant update time. Information Processing Letters 110(7), 257 – 260 (march 2010)
29. Indyk, P.: A small approximately min-wise independent family of hash functions. In: SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms. pp. 454–456. Philadelphia, PA, USA (1999)
30. Indyk, P.: Heavy hitters and sparse approximations (2009), lecture notes. http://people.csail.mit.edu/indyk/Rice/lec4.pdf
31. Indyk, P., Woodruff, D.: Optimal approximations of the frequency moments of data streams. In: STOC '05. pp. 202–208. ACM, New York, NY, USA (2005)
32. Jin, C., Qian, W., Sha, C., Yu, J.X., Zhou, A.: Dynamically maintaining frequent items over a data stream. In: CIKM '03. pp. 287–294. ACM, New York, NY, USA (2003)
33. Kane, D.M., Nelson, J., Woodruff, D.P.: An optimal algorithm for the distinct elements problem. In: PODS '10. pp. 41–52. ACM, New York, NY, USA (2010)
34. Karp, R.M., Shenker, S., Papadimitriou, C.H.: A simple algorithm for finding frequent elements in streams and bags. ACM Trans. Database Syst. 28, 51–55 (March 2003)
35. Lee, L.K., Ting, H.F.: 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. pp. 290–297 (june 2006)
36. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: VLDB '02. pp. 346–357. VLDB Endowment (2002)
37. Open problems in data streams and related topics. IITK Workshop on Algrithms for Data Streams '06 (2006), compiled and edited by Andrew McGregor
38. Metwally, A., Agrawal, D., Abbadi, A.: Efficient computation of frequent and top-k elements in data streams. Database Theory-ICDT 2005 pp. 398–412 (2005)
39. Muthukrishnan, S.: Data streams: Algorithms and applications. Now Publishers Inc (2005)
40. Nie, G., Lu, Z.: Approximate frequency counts in sliding window over data stream. In: Canadian Conference on Electrical and Computer Engineering, 2005. pp. 2232 –2236 (May 2005)
41. Saks, M., Sun, X.: Space lower bounds for distance approximation in the data stream model. In: STOC '02. pp. 360–369. ACM, New York, NY, USA (2002)
42. Sen, S., Wang, J.: Analyzing peer-to-peer traffic across large networks. In: IMW '02: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment. pp. 137–150. ACM, New York, NY, USA (2002)

43. Tirthapura, S., Woodruff, D.P.: A general method for estimating correlated aggregates over a data stream. Data Engineering, International Conference on pp. 162–173 (2012)
44. Zhang, L., Guan, Y.: Frequency estimation over sliding windows. Data Engineering, International Conference on pp. 1385–1387 (April 2008)

Appendix

A The CountSketch$_b$ Algorithm

In this section we describe the CountSketch$_b$ algorithm and prove several of its properties. Let us sketch the details of the CountSketch algorithm as defined in [13]. CountSketch is defined by three parameters (t, b, k) such that the algorithm takes space $O(tb + k)$, and if $t = O(\log \frac{n}{\delta})$ and $b \geq \max(8k, 256\frac{L_2}{\epsilon^2 \phi_k^2})$ then the algorithm outputs any element with frequency at least $(1+\epsilon)\phi_k$, except with probability δ. ϕ_k is the frequency of the kth-heavy element, and L_2 is the L_2-frequency norm of the entire (n-element) stream. The algorithm works by computing, for each element i, an approximation $\hat{n}_i$ of its frequency. The scheme guarantees that with high probability, for every element i, $|\hat{n}_i - n_i| < 8\frac{L_2(S)}{\sqrt{b}}$ (see Lemma 4 in [13]).

For $0 < \epsilon', \gamma, \delta \leq 1$ define $(\gamma, \epsilon', \delta)$-CountSketch$_b$ as the algorithm CountSketch, setting $k = \frac{1}{\gamma^2} + 1$ and letting $b = \frac{256}{\gamma^2 \epsilon'^2}$ (the parameter t remains as in the original scheme). The choice of k follows from the following known fact.

Lemma A.1. *There are at most $\frac{1}{\gamma^2}$ elements with frequency higher than γL_2.*

Proof. Assume that there are m elements with frequency higher than γL_2. It follows that $L_2 = (\sum_{j=1}^{u} n_j^2)^{1/2} \geq \sqrt{m} \cdot \gamma L_2$. Clearly, $m \leq \frac{1}{\gamma^2}$. □

Setting $k = \frac{1}{\gamma^2} + 1$ ensures that the output list is large enough to contain all the elements with frequency γL_2 or more.

However, CountSketch$_b$ does not guarantee anymore to output all the elements with frequency higher than $(1+\epsilon')\phi_k$ and no element of frequency less than $(1-\epsilon')\phi_k$ (Lemma 5 of [13]), since the value of b might not satisfy the conditions of that lemma.

We can still follow the analysis of [13] and claim that the frequency approximation of each element is still bounded (Lemma 4 of [13]),

Lemma A.2. *With probability at least $1 - \delta$, for all elements $i \in [u]$ in the stream S,*

$$|\hat{n}_i - n_i| < 8\frac{L_2(S)}{\sqrt{b}} < \tfrac{1}{2}\gamma\epsilon' L_2(S)$$

where $\hat{n}_i$ is the approximated frequency of i calculated by CountSketch$_b$*, and n_i is the real frequency of the element i.*

The proof is immediate from [13]. The above lemma allows us to bound the frequencies of the outputted elements

Proposition A.3. *The $(\gamma, \epsilon', \delta)-$*CountSketch$_b$ *algorithm outputs all the elements whose frequency is at least $(1+\epsilon')\gamma L_2(S)$.*

Proof. An element is not in the output list only if there are (at least) k elements with higher approximated frequency. Due to Lemma A.2, any element i with frequency $n_i > (1+\epsilon')\gamma L_2(S)$ has an estimated frequency of at least $\hat{n}_i \geq (1+\frac{1}{2}\epsilon')\gamma L_2(S)$, so it can be replaced only by an element with frequency higher than $\gamma L_2(S)$, however, there are at most k elements with $n_i \geq \gamma L_2(S)$, specifically, at most $k-1$ elements other than i itself, which completes the proof. □

The memory consumption of CountSketch$_b$ is bounded by $O((k+b)\log\frac{|S|}{\delta}))$ [13], which in our case gives $O(\frac{1}{\gamma^2\epsilon'^2}\log\frac{|S|}{\delta})$.