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 $\alpha$-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 is defined by $L_p = (\sum_i n_i^p)^{1/p}$, where $n_i$ is

\footnote{Throughout the paper we use the term “$L_p$ norm” to indicate the $L_p$ norm of the frequency vector, i.e., the $p$th root of the $p$th 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 \( n_1 = \frac{1}{\sqrt{N}} \) 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 \( \gamma L_p \) with memory \( \text{poly}(\gamma^{-1}, \log N, \log u) \), 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** (\((\gamma, \epsilon)\)-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 \( \Omega(1^{1-2/p}) \) 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 \( \phi_k \) is the frequency of the \( k \)th 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 \( \phi_k \), 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 \( \gamma L_2 \) times in the sliding window model, using \( \text{poly}(\gamma^{-1}, \log N, \log u) \) memory.

**Our Results.** We give the first polylogarithmic algorithm for finding an \( \epsilon \)-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 \( \gamma \)-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 \((\gamma, \epsilon)\)-approximation of the \( L_2 \)-frequent-elements, with probability at least \( 1 - \delta \) and memory \( \text{poly}(\epsilon^{-1}, \gamma^{-1}, \log N, \log \delta^{-1}) \).

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, 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 \( \alpha \)-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 \( \alpha \)-rarity, introduced by Datar and Muthukrishnan [20], is that of finding the fraction of elements that appear exactly \( \alpha \) 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 $\epsilon, \alpha, \delta$. At the bit level, their algorithm requires $O(\alpha \cdot \epsilon^{-3} \log \delta^{-1} \log N(\log N + \log u))$ bits for $\alpha$-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 $(\epsilon, \delta)$-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 $\delta$ 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 $f$ denoted $\hat{f}$ will be denoted as $\hat{f}$.

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 $\text{poly}(\log \frac{1}{\epsilon}, \log \log \frac{1}{\delta}, \log \log N, \log \log u)$.

2.2 Smooth histograms

Recently, Braverman and Ostrovsky [9] showed that a function $f$ can be $\epsilon$-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 $(\alpha, \beta)$-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 $(\alpha, \beta)$-smooth $f$ can be calculated (or $(\epsilon, \delta)$-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 $\tilde{O}(\frac{1}{\epsilon^3} \log N(g(\epsilon, \frac{\beta}{\log N}) + \frac{1}{\epsilon} \log N))$ bits [9].

The key idea is to construct a “smooth-histogram”, a structure that contains estimations on $O(\frac{1}{\epsilon} \log N)$-suffixes of the stream, $A_1 \supseteq_r A_2 \supseteq_r \ldots \supseteq_r A_\epsilon^\alpha \log(n)$. 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 $O(\frac{1}{\epsilon} \log N)$ 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** ((\( \gamma, \epsilon \))-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,

\[
n_i \geq \gamma L_1 = \gamma \sum_j n_j \implies n_i^2 \geq \gamma^2 \left( \sum_j n_j \right)^2 \geq \gamma^2 \sum_j n_j^2 = (\gamma L_2)^2,
\]

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 \( \epsilon \)-approximation (in the unbounded stream model) for the following top-frequent approximation problem.

**Definition 3.3** ((\( k, \epsilon \))-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 \( \phi_k \) 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 \( O \left( \left( k + \frac{1}{(\epsilon \gamma)^2} \right) \log \frac{4}{\delta} \right) \), 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 \( \epsilon' \).

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 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 \((\epsilon, \frac{1}{\epsilon})\)-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 \((\gamma, \epsilon)\)-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 \((\frac{\gamma}{2}, \frac{\epsilon}{2})\)-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 \(k = \frac{1}{\epsilon^2} + 1\) most frequent elements, by running \((\frac{\gamma}{2}, \frac{\epsilon}{2})\)-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 smooth-histogram. Let \(q_1, \ldots, q_k \in \{1, \ldots, w\}\) 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 \(\hat{n}_i > \frac{1}{1+\epsilon} \hat{L}_2\).

**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 \((1 + \epsilon)\hat{L}_2(W)\); in addition there exists some \(\alpha\) such that \((1 - \alpha)\hat{L}_2(A_1) \leq \hat{L}_2(W) \leq L_2(A_1)\). In our case the inequality is satisfied for \(\alpha = \epsilon/2\) (see Theorem 3 and Definition 3 in [9]). Any element \(j\) with frequency \(n_j(W) > (1 + \epsilon)\hat{L}_2(W)\) satisfies

\[
n_j(A_1) \geq n_j(W) \geq (1 + \epsilon)\gamma L_2(W) \geq (1 + \epsilon)(1 - \epsilon/2)\gamma L_2(A_1),
\]

and will be added to the output list in Step 2, since Proposition A.3 guarantees that any element \(i\) such that \(n_i(A_1) > (1 + \epsilon/4)\gamma L_2(A_1)\) 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 \(\hat{L}_2 < (1 + \epsilon)\hat{L}_2(W)\) 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 \((1 - \epsilon c)\gamma L_2(W)\), 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,

\[
n_i(W)^2 + \sum_{j \neq i} n_j(W)^2 > n_i(A_1)^2 + \sum_{j \neq i} n_j(A_1)^2 - \epsilon L_2(A_1)^2
\]

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

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,

\[
n_i(A_1) \geq \hat{n}_i(A_1) - \frac{\epsilon}{8} \gamma L_2(A_1) \geq \left(\frac{1 - \epsilon}{1 + \epsilon} - \frac{\epsilon}{8}\right) \gamma L_2(A_1) > (1 - 3\epsilon) \gamma L_2(A_1),
\]

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

\[\square\]

**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 on each of the buckets. According to [9] (corollary 5), maintaining a smooth-histogram for $L_2$ can be done with memory

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

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

As for the second part, recall that one instance of COUNTSKETCH requires a memory of

\[O\left(\frac{1}{\epsilon^2} \log N \log \frac{N}{\delta}\right)\]

(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 $(1 - \alpha)L_2(A_1) \leq L_2(W) \leq N$. The number of COUNTSKETCH instances is bounded by the number of buckets, $O\left(\frac{1}{\epsilon^2} \log N\right)$ [9], which leads to a total memory bound of

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

### 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 $\text{poly}(\epsilon^{-1}, \gamma^{-1}, \log N, \log \delta^{-1})$ 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 $\alpha$-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 $\text{DEC}(A)$ as the number of distinct elements in the stream $A$, i.e., $\text{DEC}(A) = |A|$. The function $\text{DEC}$ is an $(\epsilon, \epsilon)$-smooth-function, for every $0 \leq \epsilon \leq 1$. 


The time per hash calculation is bounded by
\[ O(t_{in}) \] by using the following lemma, which is useful tool.

Definition 4.2 (min-hash). Let \( \pi_i \) be a family of permutations over \([u]\) = \{1, \ldots, u\}. For a subset \( A \subseteq [u] \) define \( h_i \) to be the minimal permuted value of \( \pi_i \) 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 \( a \in A \),
\[
\Pr[h_i(A) = \pi_i(a)] = \frac{1}{|A|}.
\]

The family \{\( h_i \)\} is called \( \epsilon \)-approximated min-wise independent hash functions (or \( \epsilon \)-min-hash) if for any subset \( A \subseteq [u] \) and \( a \in A \),
\[
\Pr[h_i(A) = \pi_i(a)] \in \left[\frac{1}{|A|}(1 \pm \epsilon)\right].
\]

A specific construction of \( \epsilon \)-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 \( \epsilon' \)-min-hash function \( h_i \), it holds that
\[
\Pr[h_i(A = h_i(W)] = \frac{|A \cap W|}{|A \cup W|} \pm \epsilon'.
\]

4.2 A semi-smooth estimation of \( \alpha \)-rarity

In the following section we present an algorithm that estimates the \( \alpha \)-rarity of a stream (in the sliding window model), i.e., the ratio of elements that appear exactly \( \alpha \) 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 \( \alpha \)-rarity in each bucket.

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

Our algorithm follows the method used by [20] to estimate \( \alpha \)-rarity in the unbounded model. The estimation is based on the fact that the \( \alpha \)-rarity is equal to the portion of min-hash functions that their min-value appears exactly \( \alpha \) 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 $\alpha$ 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 $\alpha$-rarity of the window by calculating the fraction of min-hash functions of which the appropriate min-value $x_i$ appears exactly $\alpha$ times within the window. Due to feasibility reasons we use approximated min-wise hashes, and prove that this estimation is an $\epsilon$-approximation of the $\alpha$-rarity of the current window (up to a pre-specified additive precision). The semi-smooth algorithm $\text{ApproxRarity}$ for $\alpha$-rarity is defined in Fig. 2.

### ApproxRarity($\epsilon, \delta$)

1. Randomly choose $k \frac{1}{\epsilon}$-min-hash functions $h_1, h_2, \ldots, h_k$.
2. Maintain an $(\epsilon, \frac{\delta}{k})$-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 $\alpha$ active elements in $L_i(A_1)$, i.e. the ratio
   \[ \hat{\rho}_\alpha = \left| \{i \text{ s.t. } L_i(A_1) \text{ consists exactly } \alpha \text{ active elements} \} \right| / k. \]

**Fig. 2.** Semi-smooth algorithm for $\alpha$-rarity

The $\text{ApproxRarity}$ algorithm provides an $(\epsilon, \delta)$-approximation for the $\alpha$-rarity problem, up to an additive error of $\epsilon$. As proven by Datar et al. [20], the ratio of min-hash functions that have exactly $\alpha$ active elements in the window is an estimation of $\rho_\alpha$. 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 $(\epsilon, \delta)$-approximation for the $\alpha$-rarity problem, up to an additive precision.

**Proof.** For the sake of simplicity we treat the multisets $A_1, W, \text{ etc.}$, as sets. Let $R_\alpha$ be the set of elements which are $\alpha$-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 $\Pr[L_i(A_1) \text{ consists of exactly } \alpha \text{ active elements}]$, 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 $\alpha$ 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 $\alpha$, 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 $L_i(A_1)$ counts the number of occurrences of $x_i$ in the bucket, and since $x_i \in R_\alpha$, it appears exactly $\alpha$ 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\(^\text{a}\) that $(1 - \epsilon)|A_1| \leq |W| \leq |A_1|$, thus

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

\(^\text{a}\) Actually, it guarantees even a better bound, specifically, $(1 - \frac{\epsilon}{2})|A_1| \leq |W| \leq |A_1|.$
Therefore, estimating the ratio $\rho_a$ using $k$ hash functions results with a value $(1 + \epsilon)\rho_a \pm \frac{\epsilon}{2}$ up to some additive error $\epsilon'$ determined by $k$. Finally, using Chernoff’s inequality we can bound the additive error so that $\epsilon' < \frac{\epsilon}{2}$, except for probability $\frac{\epsilon}{2}$. In order to achieve the desired precision we require $k = \Omega\left(\frac{1}{\epsilon} \log \frac{1}{\delta}\right)$, and the estimation satisfies

$$\hat{\rho}_a \in (1 \pm \epsilon)\rho_a \pm \epsilon,$$

except for probability at most $\delta$. 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 $O((\log u + \frac{1}{\epsilon}) \log N \log \frac{1}{\delta} + \frac{1}{\epsilon} \log^2 N)$: $k$ seeds for the $\frac{1}{\epsilon}$-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 $i \in [k]$:

$$O([\log u + \alpha \log N] \frac{1}{\epsilon} \log N).$$

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 $O\left(\frac{1}{\epsilon^2} \alpha \log^2 N\right)$ bits, with $k = \Omega\left(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon}\right)$, and the time complexity is $O\left(\frac{\alpha}{\epsilon^2} \log N\right)$ calculations per element, suppressing $\text{poly}(\log \frac{1}{\epsilon}, \log \log N)$ 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 $\epsilon$-min hash functions and let

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

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 $\delta$ 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 $k = \Omega\left(\frac{1}{\epsilon^2} \log \frac{1}{\epsilon}\right)$. 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 $(\epsilon, \delta)$-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 $0 \leq |A_X \setminus W_X| \leq \frac{\epsilon'}{1-\epsilon'}|W_X|$, which follows from the guarantee of the smooth-histogram that $(1 - \epsilon')|A_X| \leq |W_X| \leq |A_X|$ (and same for $A_Y$ and $W_Y$). Using elementary set operations,
ApproxSimilarity($\epsilon, \delta$)

1. Randomly choose $k \epsilon'$-min-hashes, $h_1, \ldots, h_k$. The constant $\epsilon'$ will be specified later, as a function of the desired precision $\epsilon$.
2. For each stream ($X$ and $Y$) maintain an $(\epsilon', \frac{\epsilon}{2})$-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.,

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

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

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

$$|W_X \cup W_Y| \leq |A_X \cup A_Y| \leq |W_X \cup W_Y| + \frac{\epsilon'}{1-\epsilon'}|W_X| + \frac{\epsilon'}{1-\epsilon'}|W_Y|$$

$$\leq |W_X \cup W_Y| + 2 \frac{\epsilon'}{1-\epsilon'}|W_X \cup W_Y|$$

$$= \frac{1 + \epsilon'}{1-\epsilon'}|W_X \cup W_Y|.$$ 

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

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

and

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

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 $k = \Omega(\frac{1}{\epsilon^2 \log \frac{1}{\epsilon}})$, 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: $O(\log u + \frac{1}{\epsilon^2} \log N \log \frac{1}{\epsilon} + \frac{1}{\epsilon^2} \log^2 N)$; $k$ seeds for $\epsilon/2$-min-hash functions: $O(k \log \frac{1}{\epsilon} \log u)$; Keeping the hash value for each $h_i$: $O(k \frac{1}{\epsilon} \log N \log u)$.

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 $k = \Omega(\frac{1}{\epsilon^2 \log \frac{1}{\epsilon}})$ and assuming $\log u = O(\log N)$, we achieve a memory bound of $O(k \frac{1}{\epsilon} \log^2 N)$, with $O(k \frac{1}{\epsilon} \log N)$ calculations per element, suppressing $\textit{poly}(\log \frac{1}{\epsilon}, \log \log N)$ 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 $\alpha$-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 N66001-1-2-4014. 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

37. Open problems in data streams and related topics. IITK Workshop on Algorithms for Data Streams ’06 (2006), compiled and edited by Andrew McGregor
Appendix

A The CountSketch Algorithm

In this section we describe the CountSketch 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 \(t = O(\log \frac{n}{b})\) and \(b \geq \max(8k, 256\frac{1}{\epsilon^2\phi^2})\) then the algorithm outputs any element with frequency at least \((1 + \epsilon)\phi_k\), except with probability \(\delta\). \(\phi_k\) is the frequency of the \(k\)-th heavy element, and \(L_2\) is the \(L_2\)-frequency norm of the entire \((n\text{-element})\) 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 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 \(\gamma L_2\).

Proof. Assume that there are \(m\) elements with frequency higher than \(\gamma L_2\). It follows that \(L_2 = (\sum_{j=1}^{m} n_j^2)^{1/2} \geq \sqrt{m} \cdot \gamma L_2\). 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 \(\gamma L_2\) or more.

However, CountSketch 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 \(S\),

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

where \(\hat{n}_i\) is the approximated frequency of \(i\) calculated by CountSketch, 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 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 is bounded by \(O((k + b)\log \frac{|S|}{\delta})\) [13], which in our case gives \(O(\frac{1}{\gamma^2}\log \frac{|S|}{\delta})\).