FORTY YEARS OF FREQUENT ITEMS

JELANI NELSON

ABSTRACT

We survey the last 40 years of algorithm development for finding frequent items in data streams, a line of work which surprisingly wound up developing new tools in information theory, pseudorandomness, chaining methods for bounding suprema of stochastic processes, and spectral graph theory.

MATHEMATICS SUBJECT CLASSIFICATION 2020

Primary 68Q99; Secondary 68W20

KEYWORDS

Streaming, sketching, heavy hitters, differential privacy

INTERNATIONAL CONGRESS OF MATHEMATICIANS
2022 JULY 6—14

Proc. Int. Cong. Math. 2022, Vol. 6, pp. 4872–4896
DOI 10.4171/ICM2022/154

In many big data applications, data continuously arrives in a streaming fashion, for example, the constant stream of queries to a search engine, purchases from online vendors, or posts to social media. Such applications gave rise to the popularity of so-called *streaming algorithms*, which process such data on the fly as it arrives to later answer some queries of interest, with such algorithms often times using memory *sublinear* in the data seen so that most data is forgotten at query time. Aside from minimizing memory consumption, such sublinear memory algorithms also have the advantage of potentially being faster, since the working memory of the algorithm can fit in the faster CPU cache instead of RAM or disk, or minimize communication in distributed environments where the stream of data is sharded to many different servers for processing, which must then communicate intermediate results (or their memory footprints) to then reconstruct query answers to the aggregate data.

One of the oldest and most well-studied problems in the streaming literature is that of finding frequent items in data streams. This line of work began with an algorithm of Boyer and Moore, originally published as a technical report in 1981 and later republished a decade later [9], and still continues into the present. The results along the way have led to the development of new tools in information theory, pseudorandomness, chaining methods for bounding suprema of stochastic processes, and spectral graph theory. In this survey, we discuss some of the progress on this problem over the last 40 years.

1. THE EARLY WORK

We henceforth assume that all items in the data stream, which is finite, come from some finite universe $\mathcal{U} = [n] := \{1, \dots, n\}$. We also define the frequency histogram $f \in \mathbb{R}^n$, where f_i denotes the number of times item i was seen in the stream. Thus, seeing i in the stream corresponds to the update "$f_i \leftarrow f_i + 1$," where f is initially the zero vector. The frequent items problem then asks us to report the i such that f_i is "large" at the end of the stream.

As mentioned, the first algorithm for finding frequent items in data streams was the so-called MJRTY algorithm, discovered by Boyer and Moore in 1981 [9]. The largeness criteria used by their algorithm is that i is frequent iff $f_i > \ell/2$, where ℓ is the stream's length. That is, the MJRTY algorithm should report any i which appears a strict majority of the time (and if no such item exists, the algorithm's output is allowed to be arbitrary). Clearly, at most one majority item can possibly exist. The algorithm uses $O(1)$ memory[1] and is simple to both describe and analyze. The algorithm at any point in time stores only two things in memory: a candidate $i \in [n] \cup \{\perp\}$ for the majority element, and a counter value C which is initialized to 0. For each item j seen in the stream, if $i = j$ then C is

[1] Here we measure memory in *machine words*, where a word is a unit of memory large enough to hold the name of an item ($\lceil \log_2 n \rceil$ bits) as well as the largest frequency of any item ($\lceil \log_2 \|f\|_\infty \rceil$ bits). Whenever measuring memory in bits instead, we write so explicitly.

incremented, else if $i \neq j$ then C is decremented. If C becomes nonpositive, then i is set to j and C is set to 1. Pseudocode is given in Figure 1.

```
MJRTY:
initialize().
        1.  $i \leftarrow \perp$
        2.  $C \leftarrow 0$

update($j$). // process item j in stream
        1.  if $i = j$:
                $C \leftarrow C + 1$
        2.  else:
                $C \leftarrow \max\{0, C - 1\}$
        3.      if $C = 0$:
                    $i \leftarrow j$
                    $C \leftarrow 1$

query().
        1.  return $i$
```

```
Frequent:
initialize($k$).
        1.  $i_1, \dots, i_{k-1} \leftarrow \perp$
        2.  $C_1, \dots, C_{k-1} \leftarrow 0$

update($j$). // process item j in stream
        1.  if $\exists r$ such that $i_r = j$:
                $C_r \leftarrow C_r + 1$
        2.  else:
                for $r = 1, \dots, k - 1$:
                    $C_r \leftarrow \max\{0, C_r - 1\}$
        3.      if $\exists\ r$ such that $C_r = 0$:
                    pick such $r$ arbitrarily
                    $i_r \leftarrow j$
                    $C_r \leftarrow 1$

query().
        1.  return $\{i_1, \dots, i_{k-1}\} \backslash \{\perp\}$
```

FIGURE 1
Pseudocode for MJRTY and Frequent.

Theorem 1.1. *If there exists some $i^* \in [n]$ such that $f_{i^*} > \ell/2$, then at the time of query we must have $i = i^*$.*

Not long after the development of the MJRTY algorithm, Misra and Gries developed the generalized Frequent algorithm [32], which outputs a list $L \subset [n]$ such that (1) $|L| < k$, and (2) if i is k-frequent then $i \in L$; see the pseudocode in Figure 1. Here k is a parameter that is given at the time of initialization, and we say an item is k-frequent if $f_i > \ell/k$. That is, whereas MJRTY must report any item that appears strictly more than half the time in the stream, Frequent must report any item that appears strictly more than a $1/k$ fraction of the time. The MJRTY problem thus solves the special case $k = 2$. Rather than store a single candidate frequent item i, Frequent stores $k - 1$ candidate items $i_1, \dots, i_{k-1}$ together with counters $C_1, \dots, C_{k-1}$ (observe that the number of k-frequent items is at most $k - 1$). When a stream item j matches one of these candidate items, its corresponding counter is incremented. Otherwise, *all* counters are decremented and some arbitrary candidate with counter zero (if one exists) is replaced with the new item j and its counter is reset to 1. From the description and pseudocode, the memory consumption of Frequent is $O(k)$, which is clearly optimal since $\Omega(k)$ memory is required since there can be up to $k - 1$ frequent items and just writing down their names would take $\Omega(k)$ memory.

An alternative (randomized) algorithm is to, of course, sample: if an item appears often, then we expect it to also appear often in a substream obtained by sampling m uniformly random updates without replacement. Such a sample can be maintained in $O(m)$ memory on the fly as the stream is being updated using a fairly simple technique known as *reservoir sampling* [37], which we do not discuss in detail here. Unfortunately, it is not too hard to show that to identify all frequent items in the sense of [32], one must take $m = \Omega(k^2 \log k)$, which yields significantly worse memory consumption than the $O(k)$ memory of the Frequent algorithm, while also being randomized with a chance of error rather than providing the deterministic guarantees enjoyed by Frequent.

2. PROBLEM REFORMULATIONS AND MORE GENERAL ALGORITHMS

Rather than finding k-frequent items as defined in Section 1, one may aspire to identify a more natural set of items: the "top k" items by frequency, i.e., the k indices i with the largest f_i values (breaking ties arbitrarily). Unfortunately, we will see in Section 3 that such a task is impossible using memory sublinear in n. Instead, we try to approximate the top k set as follows. We say item i is (k, p)-*tail frequent* if

$$f_i^p > \frac{1}{k} \sum_{j=k+1}^{n} (f_j^*)^p := \frac{1}{k} \|f_{\mathrm{tail}(k)}\|_p^p.$$

Here x^* denotes the decreasing rearrangement of a vector x, i.e., x with its entries permuted so that $|x_1^*| \geq |x_2^*| \geq \cdots \geq |x_n^*|$; $x_{\mathrm{tail}(k)}$ denotes x but with its k largest entries (in magnitude) zeroed out; ties are broken arbitrarily. Similarly, we define $x_{\mathrm{head}(k)} := x - x_{\mathrm{tail}(k)}$. One sees that the number of items i which can be (k, p)-tail frequent is less than $2k$: every index in the head could be frequent, and the number of tail indices which are frequent must be strictly less than k.

Definition 2.1. For integer $k \geq 2$ and real $p \geq 1$, a (randomized) streaming algorithm is said to solve the (k, p)-*tail frequent problem* with failure probability $\delta \in (0, 1)$ if at query time it outputs a list $L \subset [n]$ such that (1) $|L| = O(k)$, and (2) with probability at least $1 - \delta$, L contains every (k, p)-tail frequent item.

When k is understood by context or not particularly relevant to the point of discussion, we sometimes refer to the (k, p)-tail frequent problem as the ℓ_p *tail heavy hitters problem*, or even more simply, the ℓ_p *heavy hitters problem*. When discussing the nontail version, we say the ℓ_p *nontail heavy hitters problem*.

It turns out the Frequent algorithm of Section 1 in fact solves the $(k, 1)$-tail frequent problem [7], i.e., it finds items that are not just more than a $1/k$ fraction of $\|f\|_1$, but even of $\|f_{\mathrm{tail}(k)}\|_1$ (with failure probability 0 in fact, as it is a deterministic algorithm). An interesting fact about the formulation of the frequent items problem in Definition 2.1 is that the notion provides a hierarchy of approximation to the actual top k problem, up to potentially changing k by a small constant factor.

Lemma 2.2. *For $p > q \geq 1$ and a frequency vector $f \in \mathbb{R}^n$, if i is (k, q)-tail frequent for f then it is also $(2k, p)$-tail frequent for f.*

Proof. Item i being (k,q)-tail frequent is equivalent to $f_i > \frac{1}{k^{1/q}}\|f_{\mathrm{tail}(k)}\|_q$. We must thus show that

$$\frac{1}{k^{1/q}}\|f_{\mathrm{tail}(k)}\|_q \ge \frac{1}{(2k)^{1/p}}\|f_{\mathrm{tail}(2k)}\|_p.$$

Define $B_j := \{jk+1,\dots,j(k+1)\}$ for $j = 0,1,\dots,n/k-1$ (we can assume that n is divisible by k without loss of generality by padding it with 0 entries, which does not affect the subsequent argument). Then

$$\begin{aligned}
\frac{1}{(2k)^{1/p}}\|f_{\mathrm{tail}(2k)}\|_p &= \left(\frac{1}{2k}\sum_{j=2}^{n/k-1}\|f^*_{B_j}\|_p^p\right)^{1/p}\\
&\le \left(\frac{1}{2k}\sum_{j=2}^{n/k-1}k\cdot\|f_{B^*_j}\|_\infty^p\right)^{1/p}\\
&\le \left(\frac{1}{2k}\sum_{j=2}^{n/k-1}k\cdot\left(\frac{\|f^*_{B_{j-1}}\|_q^q}{k}\right)^{p/q}\right)^{1/p}\\
&= \frac{1}{2k^{1/q}}\left(\sum_{j=1}^{n/k-2}\left(\|f^*_{B_{j-1}}\|_q^q\right)^{p/q}\right)^{1/p}\\
&\le \frac{1}{2k^{1/q}}\left(\sum_{j=1}^{n/k-2}\|f^*_{B_{j-1}}\|_q^q\right)^{1/q} \qquad \left(\|z\|_p \le \|z\|_q \text{ since } p > q\right)\\
&\le \frac{1}{2k^{1/q}}\|f_{\mathrm{tail}(k)}\|_q\\
&\le \frac{1}{k^{1/q}}\|f_{\mathrm{tail}(k)}\|_q. \qquad ■
\end{aligned}$$

Thus, up to changing k by a factor of at most 2, an algorithm which solves the p-version of the problem is strictly stronger than one for the q-version for $p \ge q$, in the sense that it is guaranteed to find at least as many items in its output list L. It is also not hard to show via Hölder's inequality that for any fixed x, the value p can be taken large enough (but finite) so that L is guaranteed to contain every element in the top k, regardless of how small its actual frequency is. Thus for some large but finite p the problem is essentially the top k problem, and we can gradually relax the problem (i.e., make it easier) by making p smaller and smaller. What then is the largest value of p for which the problem is algorithmically tractable in small memory? As we will see in Section 3, $p > 2$ requires $n^{\Omega(1)}$ memory. Meanwhile, Charikar, Chen, and Farach-Colton developed the CountSketch algorithm for the case $p = 2$ [14], using space $O(k \log n)$, which we now discuss in Section 2.1.

2.1. ℓ_2 heavy hitters and the turnstile model

We show a diagram representing the CountSketch data structure in Figure 2. Memory stores BR counters $C_{r,b}$ for $r \in [R]$, $b \in [B]$, all initialized to zero. We also pick R random functions $h_1,\dots,h_R : [n] \to [B]$ and another R random functions $\sigma_1,\dots,\sigma_R : [n] \to \{-1,1\}$. The functions are drawn independently, and each such function is drawn uniformly at random from the set of all functions mapping $[n]$ to its respective range. Note that just storing these

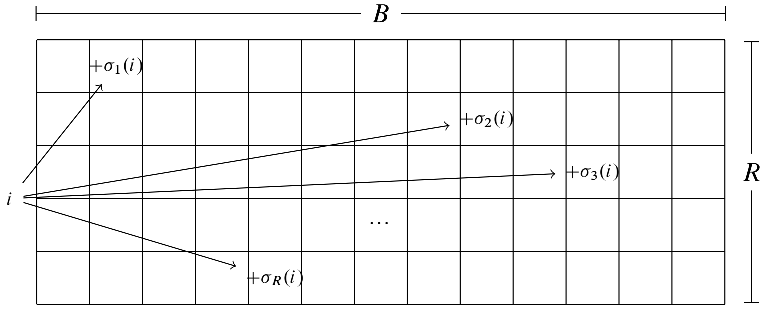

FIGURE 2
Diagram showing an update to CountSketch, when seeing item i in the stream.

functions would require an exorbitant amount of memory (more memory than simply storing the frequency histogram f explicitly in memory); we ignore the cost of storing these functions for now and address this issue later in Section 2.2. When seeing item i in the stream, for $r = 1, 2, \dots, R$ we perform the update $C_{r,h_r(i)} \leftarrow C_{r,h_r(i)} + \sigma_r(i)$. Thus at the end of the stream, each $C_{r,j}$ will equal $\sum_{i:h_r(i)=j} \sigma_r(i) f_i$. At query time, any f_i can then be estimated as $\tilde{f}_i := \text{median}\{\sigma_r(i)C_{r,h_r(i)}\}_{r=1}^R$.

Although we are ultimately interested in answering queries for the list of frequent items, we state two different types of queries the CountSketch can answer:

- `point_query`(i) : return a value $\tilde{f}_i$ in $[f_i - \frac{1}{\sqrt{k}}\|f_{\text{tail}(k)}\|_2, f_i + \frac{1}{\sqrt{k}}\|f_{\text{tail}(k)}\|_2]$.
- `frequent()` : return a list $L \subset [n]$ such that (1) $|L| = O(k)$, and (2) if i is $(k, 2)$-tail frequent then $i \in L$.

The CountSketch has randomized correctness guarantees: for any query, there is some (tunably small) probability that its output is not correct. As mentioned above, to answer `point_query`(i) we return $\tilde{f}_i := \text{median}\{\sigma_r(i)C_{r,h_r(i)}\}_{r=1}^R$. To answer `frequent()`, we return the $2k$ coordinates i with the largest $|\tilde{f}_i|$ values. Below we show that this algorithm is correct with large probability.

Lemma 2.3. *For $B \geq 6k$, for any $1 \leq i \leq n$,*

$$\mathbb{P}\left(|\tilde{f}_i - f_i| > \sqrt{\frac{6}{B}}\|f_{tail(k)}\|_2\right) \leq \exp(-R/16).$$

Proof. Write $\tilde{f}_{r,i} = \sigma_r(i)C_{r,h_r(i)}$. Let $H \subset [n]$ denote the locations of the largest k entries of f in magnitude so that $f_{\text{head}(k)} = f_H$. Let $\mathcal{E}_r$ be the event that $h_r(i) \notin h_r(H\backslash\{i\})$. Consider also the random variable $Z_r := \sum_{\substack{j \neq i \\ j \notin H}} \mathbb{1}\{h_r(j) = h_r(i)\}\sigma_r(j) f_j$ and let $\mathcal{E}'_r$ denote the event that $|Z_r| \leq \sqrt{6/B}\|f_{\text{tail}(k)}\|_2$.

Then by Markov's inequality,

$$\mathbb{P}(\neg\mathcal{E}_r) = \mathbb{P}\left(\left|\left(H\backslash\{i\}\right) \cap h_r^{-1}(i)\right| \geq 1\right) \leq \frac{k}{B} \leq \frac{1}{6}.$$

Also, $\mathbb{E}\, Z_r^2 \le \|f_{\text{tail}(k)}\|_2^2/B$, and thus

$$\mathbb{P}(\neg\mathcal{E}_r') = \mathbb{P}\left(|Z_r| > \sqrt{\frac{6}{B}}\|f_{\text{tail}(k)}\|_2\right) = \mathbb{P}\left(Z_r^2 > \frac{6}{B}\|f_{\text{tail}(k)}\|_2^2\right) < \frac{1}{6},$$

also by Markov's inequality. Thus by a union bound $\mathbb{P}(\mathcal{E}_r \wedge \mathcal{E}_r') > 2/3$. Note that when $\mathcal{E}_r \wedge \mathcal{E}_r'$ occurs, we necessarily have $|\tilde{f}_{r,i} - f_i| \le \sqrt{\frac{6}{B}}\|f_{\text{tail(k)}}\|_2$. We just showed that in expectation this fails to occur for fewer than $R/3$ values of r. Thus by the Chernoff–Hoeffding bound,

$$\mathbb{P}\left(\left|\left\{r : |\tilde{f}_{r,i} - f_i| > \sqrt{\frac{6}{B}}\|f_{\text{tail(k)}}\|_2\right\}\right| \ge R/2\right) \le \exp\left(-R\frac{(1/6)^2}{2(1/3)(2/3)}\right) = \exp(-R/16),$$

which implies the claim since $\tilde{f}_i$ is the median of the $\tilde{f}_{r,i}$ values over all $r \in [R]$. ■

Lemma 2.3 implies the following corollary by setting $B = 6k$, $R = \lceil 16\ln(1/\delta)\rceil$. The query time follows since the median of T numbers can be found in linear time $O(T)$ [8].

Corollary 2.4. *For any $\delta \in (0,1)$ and $k \ge 1$, there is an algorithm for answering a single call to `point_query` with $(k,2)$-tail error and failure probability δ using memory $O(k\log(1/\delta))$ with update time $\Theta(\log(1/\delta))$ and query time $\Theta(\log(1/\delta))$.*

A simple algorithmic reduction, which we now describe, shows how to obtain an algorithm to answer `frequent` queries in a black box way given an algorithm that solves `point_query`.

Theorem 2.5. *The CountSketch data structure with parameters $B = 54k$ and $R = \lceil 16\ln(n/\delta)\rceil$ provides a solution to the ℓ_2 heavy hitters problem with failure probability δ. The memory usage is $O(k\log(n/\delta))$, the update time is $O(\log(n/\delta))$, and the query time is $O(n\log(n/\delta))$. The output list L has size at most $18k$.*

Proof. We use the CountSketch to answer `point_query(`i`)` for every $i \in [n]$ to obtain $\tilde{f} = (\tilde{f}_i)_{i=1}^n$. We then define L to be the largest $18k$ entries of $\tilde{f}$ in magnitude (ties broken arbitrarily). We now analyze correctness. We show that correctness is guaranteed when we condition on the event $\|\tilde{f} - f\|_\infty \le \frac{1}{3\sqrt{k}}\|f_{\text{tail}(k)}\|_2$, which happens with probability at least $1-\delta$ by Lemma 2.3 and a union bound over all $i \in [n]$. Now conditioned on this event, consider some $(k,2)$-tail frequent item i; we must show that i is in L. Note that if i' is not even $(9k,2)$-tail frequent, then necessarily $\tilde{f}_{i'} < \tilde{f}_i$. This is because $\tilde{f}_j = f_j \pm \|\tilde{f} - f\|_\infty = f_j \pm \frac{1}{3\sqrt{k}}\|f_{\text{tail}(k)}\|_2$ for any j. Thus, the only items that could appear more frequent than an actual frequent item are the $(9k,2)$-tail frequent items, but since there are fewer than $18k$ such items the claim is proven. ■

Not only does the CountSketch solve the ℓ_2 tail heavy hitters problem, but it does so in a more general streaming model known as the *turnstile model*. In this model, each stream update is an (i,Δ) pair for $i \in [n]$ and $\Delta \in \mathbb{R}$ (Δ may even be negative). Such an update causes the change $f_i \leftarrow f_i + \Delta$. The previous model discussed implicitly took $\Delta = 1$ always. The definition of a $(k,2)$-tail frequent item is then similar as before except that we take absolute values: i is such a frequent item if $|f_i| > \frac{1}{\sqrt{k}}\|f_{\text{tail}(k)}\|_2$.

2.2. A digression on pseudorandomness

As mentioned in Section 2.1, the CountSketch makes use of independently chosen, uniformly random functions $h_1, \dots, h_R : [n] \to [B]$ and $\sigma_1, \dots, \sigma_R : [n] \to \{-1, 1\}$. Naively storing such functions would require $\Omega(nR)$ memory, whereas the frequent items problem already admits a trivial $O(n)$ memory solution by simply storing the frequency vector f in memory explicitly. We remedy this issue by not storing perfectly random functions, but rather functions that are *pseudorandom*.

Definition 2.6. Given integer $n \geq 1$ and a finite range M, a *hash family* is simply a collection $\mathcal{H}$ of functions mapping $[n]$ to M. For integer $k \geq 1$, we say a hash family is *k-wise independent* if for all distinct $x_1, \dots, x_k \in [n]$ and all (possibly not distinct) $y_1, \dots, y_k \in M$,

$$\mathop{\mathbb{P}}_{h \in \mathcal{H}} \left(\bigwedge_{t=1}^{k} h(x_t) = y_t \right) = \frac{1}{|M|^k},$$

where h is chosen uniformly at random from $\mathcal{H}$. That is, the distribution of $(h(x_t))_{t=1}^k$ is uniform for any choice of k distinct values $x_t \in [n]$. Similarly $\mathcal{H}$ is *ϵ-almost k-wise independent* if the distribution of $(h(x_t))_{t=1}^k$ is ϵ-close to uniform in total variation distance for any choice of k distinct x_t.

The benefit of Definition 2.6 is that a uniformly random function from a hash family $\mathcal{H}$ can be specified using only $\lceil \log_2 |\mathcal{H}| \rceil$ bits. Thus whereas the σ_r from Section 2.1 are drawn uniformly from the set $\mathcal{H}_{[n],\{-1,1\}}$ of *all* functions from $[n]$ to $\{-1, 1\}$, requiring $\log_2 \lceil H_{n,\{-1,1\}} \rceil = n$ bits each (and even worse for the h_r), we could hope that (1) picking these hash functions from k-wise independent hash families instead still guarantees correctness of CountSketch, and (2) for small k there are k-wise independent hash families that are significantly smaller than the set of all functions mapping $[n]$ to some range. Item (2) is indeed true: if n and m are powers of 2, for example, Carter and Wegman showed that k-wise independent hash families exist mapping $\{0, 1, \dots, n-1\}$ to $\{0, 1, \dots, m-1\}$ of size only $N := \max\{n, m\}^{O(k)}$ [38], thus requiring only $O(k \log N)$ bits to specify a random function from the family. For example, one can consider the family

$$\mathcal{H}_{k,\mathrm{poly}} = \left\{ h(x) = \left(\sum_{i=0}^{k-1} a_i x^i \right) \bmod m : a_0, \dots, a_{k-1} \in \mathbb{F}_N \right\},$$

where the arithmetic in computing $h(x)$ is done over $\mathbb{F}_N$, and the "mod m" simply identifies $\mathbb{F}_N$ with $\{0, 1\}^{\log_2 N}$ then projects to the least significant $\log_2 m$ bits. That is, $\mathcal{H}_{k,\mathrm{poly}}$ is the set of less than degree k polynomials over $\mathbb{F}_N$ (with a mod operation after evaluation).

It can be shown that the analysis of the CountSketch in Section 2.1 only requires the h_r, σ_r to be drawn from 2-wise independent hash families, thus requiring only $O(R \log n)$ bits (i.e., $O(R)$ machine words) of memory to store all hash functions combined. Essentially, this is because the analysis of the data structure only depends on first and second moment calculations of linear forms, which are fully determined by 2-wise independence of the hash functions (note we can round B up to the nearest power of 2).

Note: The construction of $\mathcal{H}_{k,\mathrm{poly}}$ does not actually require that n, m be powers of 2, but rather can be any prime powers. In practice, evaluation of the hash function is fastest when they are primes (and not prime powers) to allow for faster arithmetic over the finite field. The domain size n can simply be rounded up to the nearest prime. If the range size m is not a prime power, often in the analysis one can make do with ϵ-almost k-wise independence (as defined above) instead of exact k-wise independence; to achieve this, one can simply pick a prime $p > mk/\epsilon$ and pick polynomials over $\mathbb{F}_{\max\{n,p\}}$ then output the evaluation of any polynomial mod m. The number of bits to specify h is then $O(k \log(nm/\epsilon))$.

3. IMPOSSIBILITY RESULTS

So far we have discussed how to obtain and analyze algorithms for the frequent items problems, which provides an upper bound on the minimum memory required to solve the problem. In this section we focus on *lower bounds*, i.e., proving that any correct algorithm requires at least some amount of memory.

3.1. ℓ_p heavy hitters for $p > 2$

Whereas the dependence on n in the memory of CountSketch is logarithmic, it turns out that any solution to ℓ_p heavy hitters requires memory that grows polynomially with n when $p > 2$ [3]. The source of this lower bound is via reduction from a problem in *communication complexity* [29]. In the simplest communication complexity setting, there are two parties Alice and Bob. Alice receives an input $x \in \mathcal{X}$, and Bob receives $y \in \mathcal{Y}$, and they also both know some function $f : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$. They would like to communicate back and forth (if Alice speaks first then she sends a message to Bob, who in response sends a message to Alice, who then sends a message to Bob, etc.) until some player is certain of the answer and outputs $f(x, y)$. A trivial solution is for Alice to simply send her input to Bob explicitly, taking $\lceil \log_2 |\mathcal{X}| \rceil$ bits of communication; similarly, Bob can send his input to Alice using $\lceil \log_2 |\mathcal{Y}| \rceil$ bits. The question is whether it is possible to devise a communication protocol whose total communication, that is, the sum of the lengths of all messages sent, is smaller.

In the proof of the memory lower bound for the ℓ_p heavy hitters problem [3], we imagine there are not just two parties Alice and Bob, but rather $t \geq 2$ parties $P_1, P_2, \ldots, P_t$. Each P_i receives an input from the same domain $\mathcal{X}_i$, which is the power set of $[n]$; that is, the input to P_i is some subset $S_i \subseteq [n]$. The model of communication considered is that P_i sends a message to P_{i+1}, in sequential order starting from $i = 1$, and P_t must then output its guess of the function evaluation; this model is referred to as *one-way communication*, since the players only speak once each, to the next player in turn, and there is no back-and-forth conversation. The relevant function considered to show hardness of the frequent items problem is the following partial function known as *set disjointness*:

$$\textsc{Disj}_{n,t}(S_1, S_2, \ldots, S_t) := \begin{cases} 1, & \forall i \neq j, S_i \cap S_j = \emptyset, \\ 0, & \exists x \in [n] : \forall i \neq j, S_i \cap S_j = \{x\}. \end{cases}$$

Note $\textsc{Disj}_{n,t}$ is partial since it is not defined when the pairwise intersections are not all equal (or contain more than one item). Furthermore, we will be concerned with the *randomized complexity* of the problem, in which we imagine all players share knowledge of an infinite sequence of uniform random bits, for free without any communication. P_t then need only be correct with some failure probability of at most δ, where $\delta \in (0,1)$ is a parameter known to all parties at the beginning of the communication game. We refer to the minimum number of total bits (sum of message lengths sent by all players) required for a one-way randomized communication protocol to solve any input to $\textsc{Disj}_{n,t}$ (on the subdomain where it is defined) with failure probability at most $\delta \in (0,1)$ as $\mathbf{R}_{\delta}^{\rightarrow,\text{pub}}(\textsc{Disj}_{n,t})$ ("pub" signifies that the randomness is public, and "$\rightarrow$" signifies that we only consider one-way communication protocols). The following theorem is due to [3,13] (see also [24,25], with a more recent lower bound in [27] for a slightly modified problem but which implies new lower bounds for heavy hitters and other problems).

Theorem 3.1. *There exists universal $\delta_0 > 0$ such that $\forall 1 \le t \le n$, $\mathbf{R}_{\delta_0}^{\rightarrow,pub}(\textsc{Disj}_{n,t}) = \Omega(n/t)$.*

Corollary 3.2. *There exists universal constant $\delta_0 > 0$ such that for $p > 2$, any randomized streaming algorithm solving the $(2,p)$-tail heavy hitters problem with failure probability at most δ_0 must use at least $\Omega(n^{1-2/p})$ bits of memory.*

Proof. Suppose there exists an algorithm $\mathcal{A}$ using at most s bits of memory which solves the $(2,p)$-tail heavy hitters problem with failure probability at most δ_0, which is the same δ_0 from the statement of Theorem 3.1. We use such $\mathcal{A}$ to define an efficient communication protocol for $\textsc{Disj}_{n,t}$ as follows for $t = \lceil (2n^{1/p}) \rceil + 1$. P_1 initializes the algorithm, then feeds $\mathcal{A}$ the stream consisting of all elements of S_1. They then take the memory state of $\mathcal{A}$, which is simply an element of $\{0,1\}^s$, and send this state to P_2. P_2 can then continue running $\mathcal{A}$ from where it left off, and feed it as input a stream consisting of all elements of S_2, etc., for each of the first $t-1$ players. After feeding $S_1, \dots, S_{t-1}$ to $\mathcal{A}$, the $(t-1)$st player then queries $\mathcal{A}$ to obtain a list L of size $O(1)$ which contains all the $(2,p)$-tail frequent items. They then send $L \cap S_{t-1}$ to P_t, using $O(|L| \cdot \log n) = O(\log n)$ bits, who then outputs 1 iff $L \cap S_t = \emptyset$.

Note if there exists an x in the intersection of all S_i, then after processing $S_1, \dots, S_{t-1}$, x is $(2,p)$-tail frequent (in fact is it even non-tail frequent) since $f_x^p \ge 2n$ yet $\|f_{\text{tail}(k)}\|_p^p < n$, and thus x will be included in L with probability at least $1 - \delta_0$. Meanwhile, if all S_i have pairwise empty intersection, then the protocol will output 1 with probability 1. Thus we have given a correct protocol for $\textsc{Disj}_{n,t}$ where each player communicates at most $\max\{s, O(\log n)\}$ bits. Note that Theorem 3.1 measures total communication, and thus by an averaging argument, there must be some particular player who sends $\Omega(n/t^2)$ bits. Thus $\max\{s, O(\log n)\} = \Omega(n/t^2)$, which implies the desired lower bound on s. ■

4. STATE-OF-THE-ART ALGORITHMS

4.1. Insertion-only streams: the BPTree

Recall that in insertion-only streams, the frequency vector $f \in \mathbb{R}^n$ is updated at each time step by incrementing a particular coordinate, "$f_i \leftarrow f_i + 1$." The CountSketch provides a solution to ℓ_2 heavy hitters under such updates with high probability using memory $O(k \log n)$. In this section we outline the state-of-the-art algorithm, BPTree [10] (following the CountSieve data structure of earlier work [11]), which solves the same problem using $O(k \log k)$ memory. It is an open problem as to whether $O(k)$ memory is achievable. As this article is meant to be a survey of many results, we provide only informal arguments and not rigorous proofs. We also specifically focus on the case of failure probability $\delta < 1/3$, say (for general δ, the BPTree uses space $O(k \log(k/\delta))$).

The main approach of the BPTree (and of the earlier CountSieve) is to reduce from the problem of finding $O(k)$ frequent items to that of finding a single item that is *superheavy*.

Definition 4.1. Given a frequency vector $f \in \mathbb{R}^n$ and $C > 1$, $i \in [n]$ is C*-superheavy* if

$$f_i^2 > C \cdot \sum_{i' \neq i} f_{i'}^2.$$

One should have in mind C being a large constant, e.g., 10^5. Then, a superheavy item is one that not only contributes a noticeable fraction of the frequency vector's energy (in an ℓ_2 sense), but rather contributes almost everything.

The reduction. We show a reduction that if we have a space-S randomized algorithm $\mathcal{A}$ that identifies a C-superheavy item if one exists with probability at least $9/10$ (where C is e.g., 10^5), then we can use $\mathcal{A}$ in a black box manner to solve ℓ_2 heavy hitters with failure probability $1/3$ using space $O(S \cdot k \log k)$ (we then ultimately design such $\mathcal{A}$ with $S = O(1)$).

Now we sketch the details of the reduction. Imagine instantiating B independent copies $\mathcal{A}_1, \ldots, \mathcal{A}_B$ of algorithm $\mathcal{A}$ for B equal to some large constant times k (the constant depends on C). We also pick a hash function $h : [n] \to [B]$ at random from a 2-wise independent hash family as described in Section 2.2. An update to i is then fed to the algorithm $\mathcal{A}_{h(i)}$. It is not hard to show that if i is a $(k, 2)$-tail frequent item, then with probability at least $9/10$, over the randomness of h, i will be C-superheavy in the projected frequency vector $f_{h^{-1}(h(i))}$, and thus $\mathcal{A}_{h(i)}$ will report i with probability at least $(9/10)^2 > 4/5$. Thus in expectation, we recover 80% of the $(k, 2)$-tail frequent items. To recover them all with high probability, we repeat this basic scheme $\Theta(\log k)$ times, so that overall the total number of instantiations of $\mathcal{A}$ is $B \log k = \Theta(k \log k)$. We then return

$$L = \big\{ i \in [n] : \text{at least } k/2 \text{ of the } Bk \text{ instantiations of } \mathcal{A} \text{ output } i \big\}.$$

Then $|L| = O(k)$ simply by counting, and a Chernoff–Hoeffding bound implies that any frequent item is contained in L with probability at least $1 - 1/\operatorname{poly}(k)$; thus all are contained with probability $1 - 1/\operatorname{poly}(k)$ by a union bound.

Finding a superheavy item. The remaining task is then to identify a superheavy item in a stream with large constant probability; if one does not exist, the algorithm is allowed to behave arbitrarily. Suppose the superheavy item is $i^* \in [n]$, which we wish to learn.

Before we describe the algorithm, we first need the concept of a tracker.

Definition 4.2. Let $F : \mathbb{N}^n \to \mathbb{R}_{\geq 0}$ be a function mapping frequency vectors to nonnegative reals. Let $f^{(0)}, f^{(1)}, \dots, f^{(\ell)} \in \mathbb{N}^n$ be the evolution of a frequency vector throughout a stream, where $f^{(t)}$ is the frequency vector after seeing the first t stream updates (where $f^{(0)} = 0$). An algorithm $\mathcal{A}$ is a *weak tracker* for F with failure probability δ and error ε if after every time step t it outputs some $\tilde{F}_t \in \mathbb{R}$ such that

$$\mathbb{P}\Big(\exists t \in [\ell],\ \big|F(f^{(t)}) - \tilde{F}_t\big| > \varepsilon \sup_{q \in [\ell]} F(f^{(q)})\Big) \leq \delta,$$

where the probability is taken over the internal randomness used by $\mathcal{A}$.

We omit the proof of the following theorem from [10]; it primarily follows from the chaining arguments of [28], but slightly modified to take bounded independence into account.

Theorem 4.3. *Let $F_2(f) = \|f\|_2^2$, and consider the algorithm $\mathcal{B}$ which stores $\Pi \in \{-1, 1\}^{m \times n}$ with Rademacher entries drawn from an 8-wise independent family for $m \geq c/\varepsilon^2$ for some sufficiently large constant c, and which provides estimates $\tilde{F}_t = \|\Pi f^{(t)}\|_2^2/m$. Then $\mathcal{B}$ is a weak tracker for F_2 with failure probability $1/10$ and error ε. Its memory usage is $O(1/\varepsilon^2)$.*

Henceforth, for ease of exposition we assume we exactly know $Q^2 := \|f^{(\ell)}\|_2^2$ before the stream even starts, where ℓ is the stream's length (the subsequent arguments can all be slightly modified if we only know Q up to a constant factor). In reality, we do not actually know Q (we do not know the future!), but this issue is circumvented in the following way. We run a weak tracker $\mathcal{B}$ for F_2 as in Theorem 4.3 with error $\varepsilon = 1/3$. We make 10 guesses in parallel that Q^2 is approximately 2^j for $j = b + 0, b + 1, \dots, b + 9$, say, for $b = 0$. For each of these guesses independently, we run an algorithm $\mathcal{A}$ for finding the superheavy item using $S = O(1)$ memory which assumes that the corresponding guess for Q^2 was correct (up to a factor of two). Conditioned on the weak tracker succeeding, when it first reports $\tilde{F}_t \geq 2^1$, then we are certain that $Q^2 > 1$ (which is 2^0). More generally, when it first reports $\tilde{F}_t \geq 2^{b+1}$, then we are certain $Q^2 > 2^b$. In this case, we can terminate the copy of $\mathcal{A}$ which assumed $Q^2 \approx 2^b$, increment b, then start a new copy of $\mathcal{A}$ (recycling the memory from the terminated algorithm) that assumes $Q^2 \approx 2^{b+9}$ for the new value of b. The key observations are that (1) since we only operate on 10 guesses in parallel at any given time, our overall memory usage is still $O(S)$, and (2) when we instantiate a new copy of $\mathcal{A}$, if that new copy corresponds to the (approximately) correct guess of Q, then it is not hard to show that the prefix of the stream it missed processing only contained a very small constant fraction of the number of occurrences of the superheavy item and that the item must thus still be superheavy (with a slight adjustment to the constant C) in the remaining suffix of the stream.

The idea behind the algorithm $\mathcal{A}$ assuming we know Q exactly is then as follows. Knowing $Q = \|f^{(\ell)}\|_2$ exactly is equivalent to approximately knowing $Q' = (f^{(\ell)})_{i^*}$ since $(f^{(\ell)})_{i^*} \approx \|f^{(\ell)}\|_2$ due to superheaviness. We write i^* expanded in base-2 as $i^*_T i^*_{T-1} \cdots i^*_0$ for $T = \lfloor \log_2 n \rfloor$ and aim to learn these bits one at a time, starting from i_0 then moving from the least to most significant bit. The strategy to learn i^*_0 is as follows: first, we initialize two counters B_0, B_1, each to zero. We also pick a random function $\sigma : [n] \to \{-1, 1\}$ from a 4-wise independent family as described in Section 2.2. When we see i in the stream, we simply add $\sigma(i)$ to B_{i_0} (where i_j here denotes the jth least significant bit in the base-2 representation of i). We wait until the first time t that $|B_r^{(t)}| \ge Q/10$ for some $r \in \{0, 1\}$, and at that moment we declare that we have learned $i^*_0 = r$. We must iterate in some fashion to learn the remaining bits, but before we describe that, let us first get a sense for why this approach is reasonable. Consider the values of the two counters at some time t:

$$B^{(t)}_{i^*_0} = \sigma(i^*) f^{(t)}_{i^*} + \underbrace{\sum_{\substack{i \ne i^* \\ i_0 = i^*_0}} \sigma(i) f^{(t)}_i}_{\alpha}, \quad B_{1-i^*_0} = \underbrace{\sum_{i_0 = 1 - i^*_0} \sigma(i) f^{(t)}_i}_{\beta}.$$

The variances of α, β are each at most $\|f^{(t)}_{[n]\setminus\{i^*\}}\|_2^2$, which is far less than $(f^{(\ell)}_{i^*})^2 \approx Q^2$ by superheaviness. Thus we expect $|\alpha|, |\beta| \ll |f^{(\ell)}_{i^*}|$ at any fixed point in time with large probability, by the second moment method. But not only do we expect this inequality to hold at any fixed point in time, but with large probability it turns out to hold at *all* points in time, simultaneously. This fact follows by the following lemma, which can be proven by applying a Dudley-type chaining argument using limited independence (see Section 4.1.1).

Lemma 4.4. *Let $0 = y^{(0)}, y^{(1)}, \dots, y^{(T)} \in \mathbb{R}^n$ be the evolution of a frequency vector in an insertion-only stream. Let $\sigma \in \{-1, 1\}^n$ be drawn from a 4-wise independent family. Then*

$$\mathop{\mathbb{E}}_{\sigma} \sup_{0 \le t \le T} \left|\langle \sigma, y^{(t)} \rangle\right| = O\left(\|y^{(T)}\|_2\right).$$

Remark 4.5. Consider a random walk on the integers, starting at 0, where at every time step one decrements with probability $1/2$ and increments with probability $1/2$. Let the position of this random walk at time t be $x(t)$. Then one can model the evolution of this position in the following way: consider the stream $1, 2, 3, \dots, T$. This stream yields the sequence of frequency vectors

$$y^{(t)} = (\underbrace{1, 1, \dots, 1}_{t \text{ entries}}, 0, \dots, 0)^\top, \tag{4.1}$$

and then $x(t) = \langle \sigma, y^{(t)} \rangle$ for $\sigma \in \{-1, 1\}^T$ uniformly at random. Lemma 4.4 then implies $\mathbb{E} \sup_{1 \le t \le T} |x(t)| = O(\sqrt{T})$, which follows from the Lévy–Ottaviani maximal inequality (see, e.g., [30, PROPOSITION 1.1.1]). Lemma 4.4 generalizes this maximal inequality in two ways: (1) the entries of σ do not need to be independent, but rather only 4-wise independent, and (2) the lemma generalizes to the arbitrary evolution of the $y^{(t)}$ vectors where each $y^{(t+1)} - y^{(t)}$ can be any standard basis vector.

With Lemma 4.4 in hand, by Markov's inequality $|\alpha|, |\beta| \ll Q/10$ at all points in time with large constant probability. Thus, with large constant probability, (1) we will never have $|\beta| \geq Q/10$ and thus never declare some $r \neq i_0^*$, and (2) at the moment in time that we have seen the $\lceil Q/9 \rceil$th occurrence of i^* in the stream, $|\alpha|$ will be sufficiently small in magnitude that $|B_r^{(t)}| = f_{i^*}^{(t)} \pm |\alpha|$ will be at least $Q/10$. How then though do we learn *all* the bits of i^*? One we learn i_0^*, we could reset B_0, B_1 to 0 again and restart a similar process to learn i_1^*, but the argument given so far requires us to see potentially $\lceil Q/9 \rceil$ occurrences of i^* to learn a single bit of its binary representation. Thus we could only learn at most 9 of its bits this way, whereas we need to learn $\log_2 n$ bits! The idea to overcome this is to first pick a random permutation π on $[n]$ (it is possible to do this pseudorandomly as well so that π can be represented using only $O(1)$ words of memory; we omit the details). Then for every update we see in the stream, we feed $\pi(i)$ to $\mathcal{A}$ instead of i (then at the end of the entire algorithm, we apply π^{-1} to the superheavy index founded to recover i^*). This permutation has the effect that the ℓ_2^2 energy of the vector is randomly spread (in expectation). Then, after we have learned $r_{j-1}r_{j-2}\cdots r_0 = \pi(i^*)_{j-1}\pi(i^*)_{j-2}\cdots\pi(i^*)_0$ and are trying to learn $\pi(i^*)_j$, for every index i in the stream we simply ignore i (and do not feed into to $\mathcal{A}$) unless i is consistent with the bits we have learned so far, i.e., $\pi(i)_{j-1}\pi(i)_{j-2}\cdots\pi(i)_0 = r_{j-1}r_{j-2}\cdots r_0$. Intuitively this makes sense: if these bits do not match that of i^* then surely i cannot be i^*, so feeding it into $\mathcal{A}$ can only contribute to the noise α, β. Since the coordinates are randomly permuted and the energy from the nonsuperheavy item is randomly spread, by dropping a $1/2^j$ fraction of coordinates (other than i^*), the effect is that i^* only becomes *heavier* in the projected frequency vector that remains, at a geometric rate as j increases. This means that we no longer need to see $\approx Q/9$ occurrences of i^* to learn the next bit, but rather can get away with seeing a geometrically smaller number of occurrences! If we iterate in this way, then eventually there will be a unique consistent i in the remaining part of the stream (possibly because we learned all the bits of $\pi(i^*)$), and this i must be i^*. This concludes the description of $\mathcal{A}$.

4.1.1. A brief introduction to chaining arguments

We here sketch the proof of Theorem 4.4. We will be considering *Rademacher processes* defined as follows. Given a collection of vectors X, we can define a collection of random variables $(Z_x)_{x\in X}$ by $Z_x := \langle \sigma, x\rangle$, where $\sigma \in \{-1,1\}^n$ is a vector of independent Rademachers. We now study methods for upper bounding $W(X) := \mathbb{E}\sup_{x\in X}|Z_x|$; in what remains, we assume X is a subset of the unit sphere S^{n-1}.

When reading the subsequent bounds, a good example to keep in mind is the special case of Lemma 4.4 related to a random walk on the integers of length n. That is, we define $y_t = (\sum_{i=1}^t e_i)/\sqrt{n}$ (similarly as in (4.1) but normalized to lie in the unit Euclidean ball) and $Y = \{y_t\}_{t=1}^n$. Here e_i denotes the ith standard basis vector. Then we know $W(Y) = O(1)$ by Levy's maximal inequality. In the case of independent Rademachers this can be proven simply via a simple reflection argument, but since our aim is to prove this bound even when the Rademachers are only 4-wise independent, we develop another approach.

Union bound. The first bound is via a union bound over all $x \in X$:

$$\begin{aligned} W(X) &= \int_0^\infty \mathbb{P}\Big(\sup_x |Z_x| > u\Big) \mathrm{d}u \\ &= \int_0^{u^*} \mathbb{P}\Big(\sup_x |Z_x| > u\Big) \mathrm{d}u + \int_{u^*}^\infty \mathbb{P}\Big(\sup_x |Z_x| > u\Big) \mathrm{d}u \\ &\le u^* + \sum_{x \in X} \int_{u^*}^\infty \mathbb{P}\big(|Z_x| > u\big) \mathrm{d}u \quad \text{(union bound)} \\ &\lesssim u^* + |X| e^{-(u^*)^2/2} \\ &\lesssim \sqrt{\log |X|} \quad \big(\text{choose } u^* = \Theta\big(\sqrt{\log |X|}\big)\big). \end{aligned} \tag{4.2}$$

Thus in the case of Y we obtain the bound $W(Y) = O(\sqrt{\log n})$, which is not sharp.

ϵ-net. Let $\mathcal{N}(X, \ell_2, \epsilon)$ denote the minimum number of ℓ_2 balls of radius ϵ required to cover X (the *covering number*), and let X' be the set of centers in such a minimum covering. Any such covering is called an ϵ-*net*, and X' is thus an ϵ-net of optimum (i.e., minimum) size. Then for any $x \in X$ let $x' \in X'$ be defined as the closest point in X' to x in ℓ_2 distance. Then

$$\begin{aligned} \mathbb{E} \sup_{x \in X} \big|\langle \sigma, x \rangle\big| &= \mathbb{E} \sup_{x \in X} \big|\big\langle \sigma, x' + (x - x')\big\rangle\big| \\ &\le W(X') + \mathbb{E} \sup_{x \in X} \big|\big\langle \sigma, x - x' \big\rangle\big| \\ &\lesssim \log^{1/2} \mathcal{N}(X, \ell_2, \epsilon) + \epsilon \cdot \|\sigma\|_2 \quad \text{((4.2) and Cauchy–Schwarz)} \\ &\lesssim \log^{1/2} \mathcal{N}(X, \ell_2, \epsilon) + \epsilon \sqrt{n}. \end{aligned}$$

Note one can take $\epsilon = 0$ and recover (4.2). In the case of Y, an optimal ϵ-net is $Y' = \{y_{\lfloor k\epsilon^2 n \rfloor}\}_{k=1}^{\lfloor 1/\epsilon^2 \rfloor}$, and so $\log^{1/2} \mathcal{N}(Y, \ell_2, \epsilon) = \Theta(\sqrt{\log(1/\epsilon)})$. The bound is asymptotically optimized by taking $\epsilon = \Theta(1/\sqrt{n})$, which yields the same suboptimal bound $W(X) = O(\sqrt{\log n})$ as above.

Dudley's inequality. Dudley iterates the ϵ-net approach by taking a sequence of ϵ-nets $X^0, X^2, X^3, \ldots$ where X^j is a 2^{-j}-net. Letting $x(j)$ denote the closest point in X^j to x, one can write $x = x(0) + \sum_{j=1}^\infty (x(j) - x(j-1))$. Then, taking $X^0 = \{0\}$,

$$\begin{aligned} \mathbb{E} \sup_{x \in X} \big|\langle \sigma, x \rangle\big| &= \mathbb{E} \sup_{x \in X} \Big|\big\langle \sigma, x(0) \big\rangle + \sum_{j=1}^\infty \langle \sigma, x(j) - x(j-1) \rangle\Big| \\ &\le \sum_{j=1}^\infty \mathbb{E} \sup_{x \in X} \big|\big\langle \sigma, x(j) - x(j-1) \big\rangle\big| \\ &\lesssim \sum_{j=1}^\infty \frac{1}{2^j} \cdot \log^{1/2}\big(\mathcal{N}\big(X, \ell_2, 2^{-j}\big) \cdot \mathcal{N}\big(X, \ell_2, 2^{-(j-1)}\big)\big) \\ &\lesssim \sum_{j=1}^\infty \frac{1}{2^j} \cdot \log^{1/2} \mathcal{N}\big(X, \ell_2, 2^{-j}\big). \end{aligned}$$

In the case of Y, the above sum is $\sum_j \sqrt{j}/2^j = O(1)$, which is correct. How can we deal with the issue though that in our case σ only has 4-wise independent entries? The key observation is that the appearance of the "$\log^{1/2}$" function in Dudley's inequality arises as the inverse of the gaussian tail of $\langle \sigma, x\rangle$ (Khintchine's inequality). If σ only has $2k$-wise independent entries, then we still have *some* tail bound from applying Markov inequality on the $(2k)$th moment (it is important that we only look at even moments, since then $|\langle \sigma, x\rangle|^{2k} = \langle \sigma, x\rangle^{2k}$, whose expectation is determined by bounded independence via expansion into a sum of monomials). This tail leads to a converging sum for $k = 2$, and hence 4-wise independence suffices. Interestingly, it was shown that 4-wise independence is necessary; Narayanan constructed a distribution over 3-wise independent Rademachers for which the conclusion of Levy's maximal inequality fails to hold [33].

Remark 4.6. Dudley's inequality is not sharp, as can be seen, for example, by taking $X = \ell_1^n$. Then $W(X) = 1$, but a calculation reveals Dudley's bound yields only $W(X) = O(\log^{3/2} n)$. A sharp approach that is correct for any X when σ is replaced by a gaussian vector is given by Fernique [22], with an asymptotically matching lower bound by Talagrand [35]. In the Rademacher case as discussed here, an upper bound was observed by Talagrand with a conjectured matching lower bound (the so-called "Bernoulli Conjecture"); that lower bound was eventually proven by Bednorz and Latała [6].

4.2. General turnstile streams: the ExpanderSketch

While the BPTree of Section 4.1 achieves an improved memory bound of $O(k \log k)$ to solve the ℓ_2 heavy hitters problem, it only works in the insertion-only model. Recall the more general *turnstile model* is one in which each update in the stream consists of a pair (i, Δ), which triggers the change $f_i \leftarrow f_i + \Delta$ (where $\Delta \in \mathbb{R}$ may even be negative). Unfortunately, a lower bound of $\Omega(k \log n)$ memory is known to hold in the turnstile model, even for the ℓ_1 heavy hitters problem [26], showing that the memory usage of CountSketch is asymptotically optimal in this more general model.

What then is there left to study in the general turnstile model? Memory turns out to not be the only resource we care about, but rather we should judge the quality of algorithms based on at least four measures of efficiency:

1. **Memory:** as already discussed.
2. **Update time:** How much time does it take the algorithm to process a new update in the stream?
3. **Query time:** At the end of the stream, when queried how long does it take the algorithm to produce the list L of frequent items?
4. **Failure probability:** Fixing the above three quantities, the lower the failure probability, the better.

Using $O(k \log n)$ memory, examining the proof of Theorem 2.5 reveals the CountSketch has update time $\Theta(\log n)$, failure probability $O(1/n^c)$ for arbitrarily large constant c (by increasing the constant in the big-Oh of the memory bound), and query time $\Theta(n \log n)$. It is this query time that we wish to improve: the output L is of size at most k, yet it takes time more than linear in the universe size n to find the items in this list! Can we obtain an algorithm that has the same asymptotic memory, update time, and failure probability as the CountSketch but with much better query time? The answer is "yes," and this is achieved by the ExpanderSketch [31], following prior work which had shown to improve the query time but at the expense of increased memory and update time [16,17].

Theorem 4.7. *There is an algorithm for the ℓ_2 heavy hitters problem, the ExpanderSketch, which uses $O(k \log n)$ memory, and which has update time $O(\log n)$, query time $O(k \cdot \mathrm{poly}(\log n))$, and failure probability $1/n^c$ for a constant $c > 0$ that can be made arbitrarily large.*

We do not prove the theorem here but rather just give an overview of the main ideas. The main idea is to reduce to the case of small n, so that the CountSketch can then be used after the reduction. The idea behind the algorithm can be broken down into two steps. We describe Step 2 only in the case of the easier ℓ_1 heavy hitters problem, and in the so-called *strict turnstile model*, where we are promised that $f_i \geq 0$ for all i at query time. The ideas can be extended to the general turnstile model, and for the ℓ_2 version of the problem, but the details are a bit more technical and so we do not discuss them here; the simplified setting we discuss here is sufficient to highlight most of the main ideas.

Step 1. We first reduce to the case of small k: more specifically, to the case $k = O(\log n)$. This is accomplished by defining $B := \lceil k/\log n \rceil$ and picking a hash function $h : [n] \to [B]$ at random from a $\Theta(\log n)$-wise independent family. A simple argument based on Bernstein's inequality implies that any k-frequent item in the original stream will be $O(\log n)$-frequent in the projected vector $f_{h^{-1}(h(i))}$. Thus, by running a frequent items data structure $\mathcal{A}_j$ for each $j \in [B]$, we can recover the full list L as the union of the L_j returned by each $\mathcal{A}_j$.

Step 2. Due to Step 1, we can now assume that $k = O(\log n)$, and we show here how to implement each $\mathcal{A}_j$. As mentioned above, we focus only on the strict turnstile model, and for the ℓ_1 heavy hitters problem. As mentioned, the main idea is to reduce the universe size n, which we accomplish as follows. For each update (i, Δ) in the stream, we view i in base-b for $b = \mathrm{poly}(\log n)$. In this base, i has $t = O(\log_b n) = O(\log n / \log\log n)$ digits $i_{t-1} i_{t-2} \cdots i_0$. We instantiate t independent CountSketch data structures $\mathsf{CS}_0, \dots, \mathsf{CS}_{t-1}$.[2] We would like to then feed the update (i_j, Δ) to CS_j for each $0 \leq j < t$. The reasoning is that if i is k-frequent, then i_j will be k-frequent from the viewpoint of CS_j for each j. This is because all the mass from i contributes to the frequency of i_j, plus other indices in $[n]$ with the same base-b digit in the jth position can only contribute more (this is where the strict turnstile assumption comes in, since otherwise other such indices might have frequencies with opposing sign and cause cancellation). Thus, we would like to query each CM_j to obtain i_j as frequent, then simply concatenate these digits. Note that since each CS_j only operates over a frequency histogram of dimension $b = \mathrm{poly}(\log n)$, its query time is a fast $O(b \log b) = \mathrm{poly}(\log n)$.

2 Though CountSketch solves the ℓ_2 version of the problem, we know by Lemma 2.2 that it must also solve the ℓ_p version for any $p \leq 2$ (specifically, it solves the ℓ_1 version)

There are two main issues with the above scheme. The first, and easiest to fix, is the following: recall that the CS_j are randomized data structures, and so CS_j may fail to report i_j with probability as large as $1/b^c$. It is possible to show that with probability $1/n^c$, at most 1% of the CS_j data structures fail, which means we may miss 1% of the digits of a heavy hitter i. This is easily fixed though using *error-correcting codes*. For our purposes, an error-correcting code is simply a collection $\mathcal{C}$ of at least n vectors in $[b]^{O(t)}$ such that the pairwise Hamming distances between vectors in $\mathcal{C}$ is large. In that way, given some $x \in C$ then corrupting 1% of its entries in an arbitrary way, there is a unique way to "decode" that corrupted vector to recover x. Since $|\mathcal{C}| \geq n$, there is an injection (which we call the "encoding") $\mathsf{Enc} : [n] \to C$. Then when we process update (i, Δ) in the stream, we first compute $i' = \mathsf{Enc}(i)$ and run the above scheme on i', which is represented by $t' = O(t)$ digits in base-b. With high probability we will recover 99% of these digits from the CS_j, which we can then error-correct uniquely to recover i' fully, at which point we can invert the injection Enc to recover i. Codes which let us recover from such errors with linear time encoding, error-correction, and decoding exist [34], which can be used here.

The more serious issue though is that there may not be just one heavy hitter, but up to $k = O(\log n)$ of them; that is, CS_j will not only output a single i_j, but rather a list L_j of $O(\log n)$ elements in $\{0, \dots, b-1\}$. The question is then: how do we now disentangle these lists to know which digits in different lists L_j correspond to the same $i \in [n]$? Note the number of possible combinations is $\prod_j |L_j|$, which can be as big as $k^{t'} = \mathrm{poly}(n)$. We now discuss the approach to overcoming this issue.

First, consider the following simple idea which does not quite work: pick hash functions $h_1, \dots, h_{t'} : [n] \to [r]$ independently from a 2-wise independent hash family for $r = \log^c n$, for some large constant $c > 0$. Then when processing the update (i, Δ), rather than feeding $(\mathsf{Enc}(i)_j, \Delta)$ to CS_j, we instead feed the update $(h_j(i) \circ h_{j+1}(i) \circ \mathsf{Enc}(i)_j, \Delta)$, where $\circ$ denotes concatenation of objects. Note that from the perspective of CS_j it is receiving updates that index into a vector of length $2^{b+2r} = \mathrm{poly}(\log n)$, so its query time is still fast. The main intuition is that since the range of each h_j is $r \gg k^2$, with good probability (1) the set of frequent items are mapped injectively by h_j and thus have a unique "name" $h_j(i)$ in block j. Furthermore, since $b \gg k$, one can show that (2) the total amount of infrequent item mass that collides with i under h_j is small with large probability. We also still have that (3) CM_j succeeds with large probability. We say that any block $j \in [t']$ satisfying (1) through (3) is a "good" block.

Suppose all blocks are good. Then for each j, we first perform a filtering step on L_j: if two different returned elements have the same $h_j(i)$ values, we remove the one with the smaller estimated frequency. After this filtering, if (1)–(3) hold then L_j contains every frequent item and no items whose $h_j(i)$ values collide with any frequent item. We can then create a graph G on the vertex set $[t'] \times [2^r]$. Recall that an element of L_j will be a concatenation of three strings α, β, γ (ideally, α is a name $h_j(i)$, $\beta = h_{j+1}(i)$, and $\gamma = \mathsf{Enc}(i)_j$); each such element adds an edge in G from vertex (j, α) to $(j+1, \beta)$. Then, in the ideal situation that all blocks are good, G will contain a collection of at most k disjoint paths: one for each frequent item. We can thus recover all the frequent items by finding the connected components of G to recover these paths, concatenating the γ values along each component path to obtain a codeword, then decoding the codeword to recover the corresponding frequent item name in $[n]$.

Of course, life is not so simple: if we want success probability $1 - 1/\operatorname{poly}(n)$, then we can only condition on 99% of the blocks $j \in [t']$ being good, not them *all* being good. In such a case, however, we lose each frequent item corresponding to a path connected component of length t'. Specifically, every roughly 100 vertices along the path on average, we expect to hit a bad block j, which might cause us to miss seeing the edge from block j to $j + 1$. Bad blocks might also introduce spurious edges between path fragments corresponding to different heavy hitters. Thus, it may be not be possible to extract the vertex-disjoint paths, one corresponding to each heavy hitter, from the graph G we end up seeing after these corruptions. The main issue is that the errors introduced into G hide the underlying connected components (paths) that we were hoping to find. This final obstacle is overcome by borrowing an idea from [23], but with a more sophisticated disentangling algorithm. Specifically, rather than represent each heavy hitter by a path, we represent it by a base graph H which is *robust*, in that if one deletes a small fraction of edges within H and also attaches a small number of edges to H from outside parts of G, it is still possible to identify most of H inside G. Intuitively such an H should be tightly connected internally, and in fact a clique would serve this purpose. We will want an H with constant degree though, so rather we use a *constant degree expander*. Specifically, say H has vertex set $[t']$ and is regular with degree D, and let each vertices neighbors be ordered arbitrarily. Let $\Gamma(j)_r$ be the rth neighbor of $j \in [t']$ according to H. Then now when receiving an update (i, Δ) in the stream, we feed $(h_j(i) \circ h_{\Gamma(j)_1}(i) \circ \cdots \circ h_{\Gamma(j)_D}(i) \circ \mathsf{Enc}(i)_j, \Delta)$ to CS_j for each $j \in [t']$. Thus when we recover each L_j, we hope to not only recover the random name of a heavy hitter i in block j, but also its random name in every block adjacent to j according to the expander H. In this way, we ideally recover each heavy hitter as an expander connected component in H. However, due to bad blocks each such component may be slightly corrupted as mentioned above, with some internal edges missing, and some spurious edges leading outside the component. This is overcome by developing a spectral-based graph clustering algorithm, based upon the graph version of Cheeger's inequality [2,18], to recover most of the original components; we omit the details.

5. FREQUENT ITEMS WITH PRIVACY CONSTRAINTS

A new model that is increasingly gaining relevance comes from the following example. Suppose a company makes mobile devices and wishes to train better spellcheckers and autocomplete features for its messaging software. To this end, it wishes to train machine learning models based on words that its customers are texting to their contacts. The device manufacturer could accomplish this by monitoring all its customers' activity, embedding code in its messaging software which reports all text messages back to the company. Such behavior is of course problematic, as it violates most users' expectation of privacy and could even be illegal in some countries.

One way around this issue is to use *differential privacy* [19], and specifically the so-called *local model* model for differential privacy (see also [40] for so-called federated analytics approaches). Given some database $D = \{x_1, \dots, x_n\}$ and a randomized algorithm $\mathcal{M}(D)$ for releasing information, we say the algorithm is *ε-differentially private* if for all possible outputs M and for all "adjacent" D, D',

$$\mathbb{P}\big(\mathcal{M}(D) = M\big) \le e^{\varepsilon} \cdot \mathbb{P}\big(\mathcal{M}(D') = M\big),$$

where two databases are said to be adjacent if D, D' differ on exactly one data item x (either x is in one but not the other, or the metadata associated with x is altered between the two). An example to keep in mind is a hospital storing a database of patient records, with public health analysts wishing to query that data for their own research. Then the hospital is (possibly even legally) bound to maintain privacy of patients, which is at odds with the public health utility from obtaining that information. On the one hand, the hospital could release exact answers to all queries or even release the entire database (i.e., $\mathcal{M}(D) = D$), which would allow the analysts to determine the answer to any query they would like and thus provide them with optimal utility but at the expense of no privacy; on the other hand, the hospital could release $\mathcal{M}(D) = \perp$ (or a random string independent of D) which provides perfect privacy ($\varepsilon = 0$) but zero utility.

In the local model that is relevant for the original example with mobile devices, there is not one central server that owns the entire database, but rather the data is distributed across all devices (each device knows the words it communicated). Thus each individual device i will run its own algorithm $\mathcal{M}_i$ to decide a randomized message to send a central server (being run by the device manufacturer). Unlike the example of the hospital, in this scenario the central server is untrusted. As one might imagine for the case of training spellcheck or autocomplete software, it would be useful to know popular words, i.e., *frequent* words, that are being typed on the devices; indeed, a patent even exists on precisely such an approach [36]. One can then devise differentially private algorithms that allow efficient procedures for solving `point_query` and `frequent` in this model. One wishes for solutions which (1) trade off utility and privacy as efficiently as possible, (2) require low communication per device, and (3) require low processing time from the central server to answer queries given all the randomized messages it received. Finding solutions to these problems has been a very active area of research in the last several years [1,4,5,12,15,20,21,39]. We do not attempt to describe the latest and most efficient solutions in-depth, but rather we describe just two simple solutions for `point_query` to give a reader of the flavor of how such private algorithms look.

Below, we again assume the universe is $[n]$, and d denotes the number of devices.

Randomized response. The idea here is simple: if a device holds $x \in [n]$, then they send x to the server with some probability p, and otherwise they send a uniformly random $x \in [n] \backslash \{x\}$. Picking $p = \frac{e^{\varepsilon}}{e^{\varepsilon}+n-1}$ ensures that ε-differential privacy is satisfied.

To then produce an unbiased estimator $\tilde{f}_x$ of f_x to answer `point_query(x)`, the central server uses an estimator of the form

$$\tilde{f}_x = \sum_{i=1}^{d} \bigl(\alpha \cdot \mathbb{1}\{m_i = x\} + \beta\bigr), \tag{5.1}$$

for some $\alpha, \beta \in \mathbb{R}$, where m_i is the message sent by device i. Also, $\mathbb{1}\{\mathcal{E}\}$ is the indicator random variable for event $\mathcal{E}$. For the above estimator to be unbiased, we must have that each summand has expectation 1 when $x_i = x$ and has expectation 0 when $x_i \neq x$. Taking expectations, we thus obtain the following two linear constraints:

$$\alpha \cdot \frac{e^{\varepsilon}}{e^{\varepsilon} + U - 1} + \beta = 1 \text{ and}$$
$$\alpha \cdot \frac{1}{e^{\varepsilon} + U - 1} + \beta = 0.$$

Solving this system of two linear equations with two unknowns gives

$$\alpha = \frac{e^{\varepsilon} + n - 1}{e^{\varepsilon} - 1}, \quad \beta = -\frac{1}{e^{\varepsilon} - 1}.$$

One can also compute the variance (which is our proxy for "utility") and find

$$\mathrm{Var}[\tilde{f}_x] = \frac{e^{\varepsilon} + n - 2}{(e^{\varepsilon} - 1)^2} d + \frac{n - 2}{e^{\varepsilon} - 1} f_x.$$

The message length from each device in this protocol is $b = \lceil \log_2 n \rceil$. The query time for the server to obtain $\tilde{f}_x$ for all x is $\Theta(d + n)$. Note the dependence of the variance on n is linear, which can be quite large.

RAPPOR. We describe a simplified version of the RAPPOR scheme [20]. There are two versions depending on the specific privacy guarantees desired. In so-called *deletion privacy*, a device should be able to opt out from sending its data without the server knowing it opted out, in which case it will send a message based on some dummy input "x^*." In *replacement privacy*, we want privacy in the sense that what the server receives should be nearly indistinguishable if that device had been replaced by some other device holding some other data x. We focus only on deletion privacy, as the scheme is slightly simpler to present in this way, and for that we use the *symmetric* version of RAPPOR. In this scheme, a device holding x maps it to the standard basis vector $e_x \in \mathbb{R}^n$, also known as the *one-hot encoding* of x. The device then flips each bit of e_x independently with probability p to form its message M, which it then sends to the server (in asymmetric RAPPOR, the probabilities of flipping 0 to 1 versus 1 to 0 are different). One would expect the variance of a resulting estimator to monotonically increase as p increases from 0 to $1/2$, so the goal is to make p as small as possible while preserving privacy. To ensure privacy, we must ensure that the message that is sent has roughly the same probability for any device even if it chooses to "delete" its input and replace it with some canonical dummy input x^*.

A device that opts out of sharing its data at all will pretend that it holds $x^* = \perp$ (resulting in the vector $e_{x^*} = \vec{0}$). Consider $x \in [n]$; we must ensure that for any message M

$$e^{-\varepsilon} \cdot \mathbb{P}(M|x^*) \leq \mathbb{P}(M|x) \leq e^{\varepsilon} \cdot \mathbb{P}(M|x^*).$$

For these two inputs, the message M is either obtained by flipping bits independently in e_x or in e_{x^*}. The only index for which the resulting bit in M has differing probabilities is the index x, differing by a factor of $(1-p)/p = 1/p - 1$. Since this quantity must be at least e^ε, the smallest we can set p is $p = 1/(e^\varepsilon + 1)$.

We must now determine an unbiased estimator for the server to estimate f_x in answering `point_query(x)`. If device i sends message $m_i \in \{0,1\}^n$, we will use an estimator of the form

$$\tilde{f}_x = \sum_{i=1}^{d} \big(\alpha \cdot \mathbb{1}\{(m_i)_x = 1\} + \beta\big). \tag{5.2}$$

We focus on each summand and again after taking expectations have two linear constraints that arise from the cases $x_i = x$ and $x_i \neq x$. These constraints are

$$\alpha(1-p) + \beta = 1 \text{ and}$$
$$\alpha p + \beta = 0.$$

Some calculation then yields the solution

$$\alpha = \frac{1}{1-2p}, \quad \beta = -\frac{p}{1-2p}.$$

To compute $\mathrm{Var}[\tilde{f}_x]$, we again have independence of summands, where summands with $x_i = x$ each contribute some value A, and those with $x_i \neq x$ contribute B. The total variance is then $A \cdot f_x + B \cdot (d - f_x)$. Some computation then yields

$$\mathrm{Var}[\tilde{f}_x] = \frac{p(1-p)}{(1-2p)^2} d + \frac{p^2}{(1-2p)^2} f_x.$$

Unlike the case of Randomized Response, we do not have such a large dependence on n in the variance, and thus the utility is far superior especially for large n. The message length in this protocol is $b = n$, and the query time for the server to obtain $\tilde{f}_x$ for all x is $\Theta(dn)$; both are much worse than Randomized Response. A recent scheme of Feldman and Talwar provides a slight variant of RAPPOR which significantly reduces the message length to $O(\log n)$ bits [21].

ACKNOWLEDGMENTS

We thank Ravi Kumar, Pasin Manurangsi, Huy Lê Nguyễn, and Uri Stemmer for providing helpful feedback on a previous draft of this article.

FUNDING

The author is partially supported by NSF award CCF-1951384, ONR grant N00014-18-1-2562, ONR DORECG award N00014-17-1-2127, and a Google Faculty Research Award.

REFERENCES

[1] J. Acharya, Z. Sun, and H. Zhang, Hadamard response: estimating distributions privately, efficiently, and with little communication. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 1120–1129, 2019.

[2] N. Alon and V. D. Milman, λ_1, isoperimetric inequalities for graphs, and superconcentrators. *J. Combin. Theory Ser. B* **38** (1985), no. 1, 73–88.

[3] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar, An information statistics approach to data stream and communication complexity. *J. Comput. System Sci.* **68** (2004), no. 4, 702–732.

[4] R. Bassily, K. Nissim, U. Stemmer, and A. Thakurta, Practical locally private heavy hitters. *J. Mach. Learn. Res.* **21** (2020), 16:1–16:42.

[5] R. Bassily and A. D. Smith, Local, private, efficient protocols for succinct histograms. In *Proceedings of the 47th Annual ACM on Symposium on Theory of Computing (STOC)*, pp. 127–135, ACM, 2015.

[6] W. Bednorz and R. Latała, On the boundedness of Bernoulli processes. *Ann. of Math.* **3** (2014), no. 180, 1167–1203.

[7] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss, Space-optimal heavy hitters with strong error bounds. *ACM Trans. Database Syst.* **35** (2010), no. 4, 26:1–26:28.

[8] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, Linear time bounds for median computations. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 119–124, ACM, 1972.

[9] R. S. Boyer and J. S. Moore, MJRTY: A fast majority vote algorithm. In *Automated reasoning: Essays in honor of Woody Bledsoe*, pp. 105–118, Springer, 1991.

[10] V. Braverman, S. R. Chestnut, N. Ivkin, J. Nelson, Z. Wang, and D. P. Woodruff, Bptree: An ℓ_2 heavy hitters algorithm using constant memory. In *Proceedings of the 36th ACM SIGMOD–SIGACT–SIGAI Symposium on Principles of Database Systems (PODS)*, pp. 361–376, ACM, 2017.

[11] V. Braverman, S. R. Chestnut, N. Ivkin, and D. P. Woodruff, Beating CountSketch for heavy hitters in insertion streams. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pp. 740–753, ACM, 2016.

[12] M. Bun, J. Nelson, and U. Stemmer, Heavy hitters and the structure of local privacy. *ACM Trans. Algorithms* **15** (2019), no. 4, 51:1–51:40.

[13] A. Chakrabarti, S. Khot, and X. Sun, Near-optimal lower bounds on the multiparty communication complexity of set disjointness. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity (CCC)*, pp. 107–117, IEEE Computer Society, 2003.

[14] M. Charikar, K. C. Chen, and M. Farach-Colton, Finding frequent items in data streams. *Theoret. Comput. Sci.* **312** (2004), no. 1, 3–15.

[15] W. Chen, P. Kairouz, and A. Özgür, Breaking the communication-privacy-accuracy trilemma. In *Proceedings of the 33rd Annual Conference on Advances in Neural Information Processing Systems (NeurIPS)*, Neural Information Processing Systems Foundation, Inc., 2020.

[16] G. Cormode and M. Hadjieleftheriou, Finding frequent items in data streams. *Proc. VLDB Endow.* **1** (2008), no. 2, 1530–1541.

[17] G. Cormode and S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications. *J. Algorithms* **55** (2005), no. 1, 58–75.

[18] J. Dodziuk, Difference equations, isoperimetric inequality and transience of certain random walks. *Trans. Amer. Math. Soc.* **284** (1985), 787–794.

[19] C. Dwork, F. McSherry, K. Nissim, and A. D. Smith, Calibrating noise to sensitivity in private data analysis. In *Proceedings of the 3rd Theory of Cryptography Conference (TCC)*, pp. 265–284, Springer, 2006.

[20] Ú. Erlingsson, V. Pihur, and A. Korolova, RAPPOR: randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1054–1067, ACM, 2014.

[21] V. Feldman and K. Talwar, Lossless compression of efficient private local randomizers. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, pp. 3208–3219, Proceedings of Machine Learning Research, 2021.

[22] X. Fernique, Regularité des trajectoires des fonctions aléatoires gaussiennes. *Lecture Notes in Math.* **480** (1975), 1–96.

[23] A. C. Gilbert, Y. Li, E. Porat, and M. J. Strauss, For-all sparse recovery in near-optimal time. In *Proceeedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pp. 538–550, Springer, 2014.

[24] A. Gronemeier, Asymptotically optimal lower bounds on the NIH-multi-party information complexity of the and-function and disjointness. In *Proceedings of the 26th International Symposium on Theoretical Aspects of Computer Science*, pp. 505–516, Leibniz-Zentrum für Informatik, 2009.

[25] T. S. Jayram, Hellinger strikes back: A note on the multi-party information complexity of AND. In *Proceedings of the 13th International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, pp. 562–573, Springer, 2009.

[26] H. Jowhari, M. Saglam, and G. Tardos, Tight bounds for L_p samplers, finding duplicates in streams, and related problems. In *Proceedings of the 30th ACM SIGMOD–SIGACT–SIGART Symposium on Principles of Database Systems (PODS)*, pp. 49–58, ACM, 2011.

[27] A. Kamath, E. Price, and D. P. Woodruff, A simple proof of a new set disjointness with applications to data streams. In *Proceedings of the 36th Computational Complexity Conference (CCC)*, pp. 37:1–37:24, Leibniz-Zentrum für Informatik, 2021.

[28] F. Krahmer, S. Mendelson, and H. Rauhut, Suprema of chaos processes and the restricted isometry property. *Comm. Pure Appl. Math.* **67** (2014), no. 11, 1877–1904.

[29] E. Kushilevitz and N. Nisan, *Communication complexity*. Cambridge University Press, 1997.

[30] S. Kwapień and W. A. Woyczyński, *Random series and stochastic integrals: single and multiple*. Birkhäuser, 1992.

[31] K. G. Larsen, J. Nelson, H. L. Nguyen, and M. Thorup, Heavy hitters via cluster-preserving clustering. In *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 61–70, IEEE Computer Society, 2016.

[32] J. Misra and D. Gries, Finding repeated elements. *Sci. Comput. Program.* **2** (1982), no. 2, 143–152.

[33] S. Narayanan, 3-wise independent random walks can be slightly unbounded. *Random Structures Algorithms* (to appear), (2021).

[34] D. A. Spielman, Linear-time encodable and decodable error-correcting codes. *IEEE Trans. Inf. Theory* **42** (1996), no. 6, 1723–1731.

[35] M. Talagrand, Regularity of gaussian processes. *Acta Math.* **159** (1987), 99–149.

[36] A. Thakurta, A. Vyrros, U. Vaishampayan, G. Kapoor, J. Freudiger, V. Sridhar, and D. Davidson, *Learning new words*. 2017, URL https://www.google.com/patents/US9594741, US Patent 9,594,741

[37] J. S. Vitter, Random sampling with a reservoir. *ACM Trans. Math. Software* **11** (1985), no. 1, 37–57.

[38] M. N. Wegman and L. Carter, New classes and applications of hash functions. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 175–182, IEEE Computer Society, 1979.

[39] H. Wu and A. Wirth, Locally differentially private frequency estimation. 2021, CoRR, arXiv:2106.07815.

[40] W. Zhu, P. Kairouz, B. McMahan, H. Sun, and W. Li, Federated heavy hitters discovery with differential privacy. In *Proceedings of the 23rd International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 3837–3847, Proceedings of Machine Learning Research, 2020.

JELANI NELSON

Soda Hall 633, Berkeley, CA, USA 94720-1776, minilek@berkeley.edu