High-Probability List-Recovery, and Applications to Heavy Hitters

Dean Doron
Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Mary Wootters
Departments of Computer Science and Electrical Engineering, Stanford University, CA, USA

Abstract

An error correcting code $\mathcal{C}\colon \Sigma^k \to \Sigma^n$ is efficiently list-recoverable from input list size ℓ if for any sets $\mathcal{L}_1, \ldots, \mathcal{L}_n \subseteq \Sigma$ of size at most ℓ, one can efficiently recover the list $\mathcal{L} = \{x \in \Sigma^k : \forall j \in [n], \mathcal{C}(x)_j \in \mathcal{L}_j\}$. While list-recovery has been well-studied in error correcting codes, all known constructions with "efficient" algorithms are not efficient in the parameter ℓ. In this work, motivated by applications in algorithm design and pseudorandomness, we study list-recovery with the goal of obtaining a good dependence on ℓ. We make a step towards this goal by obtaining it in the weaker case where we allow a *randomized* encoding map and a small failure probability, and where the input lists are derived from unions of codewords. As an application of our construction, we give a data structure for the heavy hitters problem in the strict turnstile model that, for some parameter regimes, obtains stronger guarantees than known constructions.

2012 ACM Subject Classification Theory of computation → Error-correcting codes; Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Pseudorandomness and derandomization

Keywords and phrases List recoverable codes, Heavy Hitters, high-dimensional expanders

Digital Object Identifier 10.4230/LIPIcs.ICALP.2022.55

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://eccc.weizmann.ac.il/report/2020/162/` [11]

Funding *Dean Doron*: The work was done at Stanford, supported by NSF award CCF-1763311.
Mary Wootters: Supported by NSF award CCF-1844628 and NSF-BSF award CCF-1814629.

Acknowledgements We would like to thank Jelani Nelson and Amnon Ta-Shma for helpful conversations. We thank Mahdi Cheraghchi, Venkat Guruswami, and Badih Ghazi for pointing out relevant related work. We also thank anonymous reviewers for helpful comments and for pointing out related works.

1 Introduction

Let $\mathcal{C}\colon \Sigma^k \to \Sigma^n$ be an *error correcting code*. We say that $\mathcal{C}$ is (efficiently) *list-recoverable*[1] from list-size ℓ with output list-size L if, for any lists $\mathcal{L}_1, \ldots, \mathcal{L}_n \subseteq \Sigma$ with $|\mathcal{L}_i| \leq \ell$ for all i, there is an (efficient) algorithm to recover the list

$$\mathcal{L} = \{x \in \Sigma^k : \forall i \in [n], \mathcal{C}(x)_i \in \mathcal{L}_i\},$$

and $|\mathcal{L}| \leq L$. List recovery has historically been studied in the context of *list-decodable* codes, where it has been used as a tool to obtain efficient list-decoding algorithms (see, e.g., [17, 16, 19, 27, 22]). However, even though efficient list-recovery algorithms have been

[1] In this paper we focus on *zero-error* list-recovery, which is the definition given here. Other works focus on the more general problem of list-recovery from errors, in which $\mathcal{C}(x)_i$ needs to be in $\mathcal{L}_i$ only for some fraction of the i-s.

developed, all of them have a poor dependence on the parameter ℓ. For example, Hemenway, Ron-Zewi, and Wootters [22] presents near-linear-time (in n) list-recovery algorithms, but the output list $\mathcal{L}$ has size doubly-exponential in ℓ.

In this work, we are motivated by the following goal (which we do not fully achieve):

▶ **Goal 1.** *For $\ell \geq 2$, design a family of codes $\mathcal{C} \colon \Sigma^k \to \Sigma^n$ so that:*

1. *$\mathcal{C}$ can be encoded in time $O(n)$;*
2. *The* rate *k/n of the code is a constant (independent of n **and** ℓ);*
3. *The* alphabet size *$|\Sigma|$ is polynomial in ℓ (and independent of n);*
4. *The code $\mathcal{C}$ can be list-recovered in time $O(n \cdot \ell)$ (linear in both n **and** ℓ), with output list size $|\mathcal{L}| = O(\ell)$.*

To the best of our knowledge, this goal is open even if we allow the output list size $|\mathcal{L}|$ and the running time to depend polynomially on ℓ, rather than linearly.

Goal 1 is desirable for several reasons. First, it represents a bottleneck in our understanding of algorithmic coding theory, and it seems likely that achieving it would involve developing new techniques that would be useful elsewhere. Second, list-recovery with reasonable dependence on ℓ is related to questions in pseudorandomness, where the the parameter ℓ is often very large (see our discussion in Section 1.2). Third, as we explore in this paper, obtaining Goal 1 has applications in algorithm design, in particular to algorithms for heavy hitters.

Probabilistic list-recovery with good dependence on ℓ

In this work, we make progress on Goal 1 by achieving a relaxed version where the encoding map $\mathcal{C} \colon \Sigma^k \to \Sigma^n$ is allowed to be *randomized*, and where the input lists are generated from unions of codewords; we must succeed with high probability over the randomness in $\mathcal{C}$. In particular, our main result implies the following theorem.

▶ **Theorem 2** (informal; weaker than main result)**.** *For all $\ell > 0$, there is a randomized encoding map $\mathcal{C} \colon \Sigma^k \to \Sigma^n$ so that*

1. *$\mathcal{C}$ can be encoded in time $O(n)$;*
2. *The rate of $\mathcal{C}$, k/n, is a constant independent of ℓ and n;*
3. *The alphabet size $|\Sigma|$ is polynomial in ℓ (and independent of n);*
4. *For any list $x^{(1)}, x^{(2)}, \ldots, x^{(\ell)} \in \Sigma^k$, there is an algorithm that runs in time $O(n\ell \operatorname{polylog} \ell)$ that has the following guarantee. With probability at least $1 - o(1)$ over the randomness of $\mathcal{C}$, given the lists $\mathcal{L}_i = \{\mathcal{C}(x^{(j)})_i : j \in [\ell]\}$, the algorithm returns a list $\mathcal{L}$ so that $x^{(i)} \in \mathcal{L}$ for all i, and so that $|\mathcal{L}| = O(\ell)$.*

This statement is weaker than our main result because in fact our result still holds even if a random subset of the lists $\mathcal{L}_i$ in Item 4 are erased, and moreover the result still holds when some of the lists $\mathcal{L}_i$ in Item 4 contain some extra "distractor" symbols that occur according to any sufficiently "nice" distribution. We defer the formal statement of our list-recovery guarantee to Section 2.

Our code is essentially an *expander code* with *aggregated symbols*. That is, we begin with an expander code $\mathcal{C}_0 \colon \Sigma_0^k \to \Sigma_0^n$, as in [39], and we aggregate together the symbols as in [1]. (We discuss this construction in more detail below.) Our recovery algorithm uses ideas from previous algorithms, propagating information around the underlying expander graph given some advice. What makes our work different are the facts that (a) we leverage the randomness of $\mathcal{C}$ and a small failure probability, and (b) our underlying expander graph comes

from a *high-dimensional expander*.[2] In particular, using the randomness in $\mathcal{C}$ we are able to obtain an algorithm with running time nearly-linear in ℓ, and using a high-dimensional expander we are able to boost our success probability to a level appropriate for an application to heavy hitters, which we discuss next.

Motivation from Heavy Hitters

One of the reasons we are interested in Goal 1 is because of the potential algorithmic applications of such a code. To illustrate this potential, we work out an application of our construction to the *heavy hitters* problem. **We emphasize that our focus is on the parameter regime where N is very large, specifically $\log N \gg \mathbf{poly}(1/\varepsilon)$.** In particular, we are interested in optimizing the dependence on N, rather than on ε.

The set-up is as follows. We are given a stream of updates $(x^{(i)}, \Delta^{(i)})$, for $x^{(i)}$ in some universe $\mathcal{U}$ of size N, and $\Delta^{(i)} \in \mathbb{R}$. For all m, x, we assume that $f(x) \triangleq \sum_{j\in[m]} \Delta^{(j)} \cdot \mathbf{1}_{x^{(j)}=x} > 0$. we think of $f(x)$ as the "frequency" of item x. The Δ-s are updates: we may add or remove some quantity of each item x, provided that $f(x)$ never becomes negative. This is called the *strict turnstile model.* The goal is to maintain a small data structure (a "sketch") so that, after m (efficient) updates $(x^{(i)}, \Delta^{(i)})$, we can (efficiently) query the data structure to return a list of *ε-heavy hitters.* That is, we would like to recover a list $\mathcal{L}$ of size at most $O(1/\varepsilon)$ that contains all $x \in \mathcal{U}$ so that $f(x) \geq \varepsilon \cdot \|f\|_1 \triangleq \sum_{x\in\mathcal{U}} f(x)$.

The beautiful *Count-Min Sketch* (CMS) data structure of Cormode and Muthukrishnan [7] gives a solution to this problem. It uses optimal space $O(\varepsilon^{-1} \log N)$ and has update time $O(\log N)$. However, the query time to return all $O(1/\varepsilon)$ heavy hitters is large, $O(N \log N)$ (essentially, one performs a point query for each $x \in \mathcal{U}$ to see if it is a heavy hitter). The work [7] showed how to alleviate this with a so-called "dyadic trick," bringing the query time to $O(\log^2 N)$ at the cost of an extra $\log N$ factor in both the space and update time.[3] (See Table 1 for a summary of the parameters in these and other works).

The starting point for our work is the work of Larsen, Nelson, Nguyễn and Thorup [29]. That work studied a much more general problem – heavy hitters for all ℓ_p norms in the general turnstile model – but for the special case of the ℓ_1 norm and the strict turnstile model, they were able to get a nearly optimal algorithm, with the same space and update time complexity as the original CMS, but with query time $O(\varepsilon^{-1} \log^{1+\gamma} N)$ for any constant $\gamma > 0$. That work highlighted a connection to list-recovery (see [29, Section C]; a similar connection is also present in earlier works on group testing and compressed sensing, for example [24, 35, 36, 13, 12]), which is one of our motivations to study Goal 1.

The approach of [29] was the following (we have modified the description to be more explicitly coding-theoretic). To perform an update on an item $x \in \mathcal{U}$, encode it as $\mathcal{C}(x) \in \Sigma^n$ with our (randomized) encoding function. Then insert each symbol $\mathcal{C}(x)_j$ into n different ε-heavy hitters data structures that work on universe Σ (this could be a small CMS sketch, or something else). To query all of the heavy hitters, we first query each smaller data structure to find a list $\mathcal{L}_j$. Notice that since $|\Sigma| \ll |\mathcal{U}|$, it does not matter that the query algorithm for the small data structures is slow. Now, we do list-recovery on the lists $\mathcal{L}_j$ to recover a list $\mathcal{L}$ that contains all of the heavy hitters.[4]

[2] We note that the construction of Dinur et al. [9] is similar to ours, also using an ABNNR-style [1] symbol aggregation with a high-dimensional expander. However, in that work they have a more ambitious goal – list-decoding with no randomness in the encoder – but in return the parameters are not close to those in Goal 1.

[3] See also the work by Cormode and Hadjieleftheriou [6] who consider a generalization of the dyadic trick that trades off between the query time and the overhead in update time and space.

[4] Provided that the output $\mathcal{L}$ of the list-recovery algorithm is not too large, we can use an additional large CMS data structure to efficiently do point queries on each item $x \in \mathcal{L}$, pruning it down to $O(1/\varepsilon)$.

Table 1 Some relevant results on ε-heavy hitters in the strict turnstile model where the universe has size N, for $\log N \gg \mathrm{poly}(1/\varepsilon)$. We consider schemes with failure probability $\delta \geq 1/\mathrm{poly}(N)$; see the discussion in Section 1.3 for smaller failure probability where the works marked with $\star$ shine. The $\tilde{O}$ notation hides $\log\log(N)$ factors and $\log(1/\varepsilon)$ factors. Above, c is a constant independent of N and ε, and γ is any constant larger than 0. Unfortunately, the failure probability for our algorithm is only $N^{-\mathrm{poly}(\varepsilon)}$, rather that N^{-c} for some constant c. By repeating our algorithm $\mathrm{poly}(1/\varepsilon)$ times we can boost the success probability to N^{-c}. We note that each of Space, Update, Query time for [7] (with the dyadic trick) and [31] can be multiplied by ε^c if one replaces the failure probability with $N^{-\varepsilon^c}$ and the results from [29, Theorem 9] remain the same for that larger failure probability.

Reference	Space	Update	Query	Failure probability
[7]	$O\left(\frac{\log N}{\varepsilon}\right)$	$O(\log N)$	$O(N \log N)$	N^{-c}
[7] ("dyadic trick")	$O\left(\frac{\log^2 N}{\varepsilon}\right)$	$O(\log^2 N)$	$O\left(\frac{\log^2 N}{\varepsilon}\right)$	N^{-c}
[29]	$O\left(\frac{\log N}{\varepsilon}\right)$	$O(\log N)$	$O\left(\frac{\log^{1+\gamma} N}{\varepsilon}\right)$	N^{-c}
[31]*	$\widetilde{O}\left(\log^2 N\right)$	$\widetilde{O}\left(\varepsilon \log^2 N\right)$	$\frac{1}{\varepsilon}\,\mathrm{poly}(\log N)$	N^{-c}
[5]*	$O\left(\frac{\log N}{\varepsilon}\right)$	$\widetilde{O}(\log N)$	$\frac{1}{\varepsilon}\,\mathrm{poly}(\log N)$	0
This work	$O\left(\frac{\log N}{\varepsilon}\right)$	$O(\log N)$	$O\left(\frac{\log N}{\varepsilon}\right)$	$N^{-\mathrm{poly}(\varepsilon)}$
This work	$O\left(\frac{\log N}{\varepsilon^c}\right)$	$O\left(\frac{\log N}{\varepsilon^c}\right)$	$O\left(\frac{\log N}{\varepsilon^c}\right)$	N^{-c}

However, as Goal 1 remains open, [29] did not use a list-recoverable code to obtain their results. Instead, they (like us) took advantage of the fact that the lists $\mathcal{L}_j$ can be viewed as random variables over the randomness in the encoding map $\mathcal{C}$, and then use a construction based on "cluster-preserving clustering" to solve the problem. While in some sense this construction must be a list-recoverable code for randomized input lists, it is not clear (to us) how to extract a natural code out of it: the work [29] took the perspective of graph clustering, rather than coding theory. In contrast, our code is very natural in the context of coding theory, as it is simply an expander code with aggregated symbols (albeit using a high-dimensional expander for the underlying graph).

As an example of the utility of our construction, we plug our randomized list-recoverable code (as in Theorem 2) into the framework of [29]. This gives us an algorithm for heavy hitters that, in some parameter regimes, even slightly outperforms that of [29]. When ε is constant and N is growing, we are able to improve the query time from $O(\log^{1+\gamma} N)$ to $O(\log N)$. In particular, we prove the following theorem. (See Table 1 for a comparison to other work when $\log N \gg \mathrm{poly}(1/\varepsilon)$).

▶ **Theorem 3** (informal; see Theorem 5.11 in the full version)**.** *There is a data structure that solves the heavy hitters problem in the strict turnstile model, that uses space $O(\varepsilon^{-1}\log N)$, update time $O(\log N)$, and query time $O(\varepsilon^{-1}\log N\,\mathrm{polylog}(1/\varepsilon))$, with failure probability $\delta = N^{-\Theta(\varepsilon^3)}$, as long as $\varepsilon \geq (\log N)^{-\Omega(1)}$.*

By repeating this data structure $O(\varepsilon^{-3})$ times, we obtain a data structure that takes space $O(\varepsilon^{-4}\log N)$, update time $O(\varepsilon^{-3}\log N)$ and query time $O(\varepsilon^{-4}\log N\,\mathrm{polylog}(1/\varepsilon))$, with failure probability $\delta = N^{-c}$.

Our algorithm has the added property that a successful $\mathcal{L}$ of size $O(1/\varepsilon)$ not only contains all the true heavy hitters, but also does not contain "false-positives", in the sense that each $x \in \mathcal{L}$ satisfies, say, $f(x) \geq \frac{\varepsilon}{4}\|f\|_1$. This property also applies to most previous heavy hitters algorithms.

Contributions

To summarize, our main contributions are the following.

1. **A code with probabilistic list-recovery.** We give a natural code construction that achieves a probabilistic version of Goal 1, as per Theorem 2. Our code construction leverages recent progress in high-dimensional expanders in order to succeed with high probability. We hope that our construction and techniques may be used in the the future to make further progress on Goal 1.
2. **Proof of concept: application to heavy hitters.** As an illustration of the utility of our construction – and as an proof-of-concept meant to encourage study of Goal 1 – we obtain a new data structure for ε-heavy hitters in the strict turnstile model. Our data structure has slightly stronger guarantees than existing constructions for failure probability $1/\operatorname{poly}(N)$ when ε is constant and the universe size N is growing (although it is outperformed by previous work when ε is small compared to $1/\log(N)$).

1.1 Construction Overview

In this section, we give a brief overview of our probabilistically list-recoverable code. We use this code to solve the ε-heavy-hitters problem following the paradigm described above, by using small heavy-hitters sketches for each symbol of the (randomized) encoding $\mathcal{C}(x)$ of $x \in \mathcal{U}$.

At a high level, we construct our code $\mathcal{C}\colon \Sigma_0^k \to \Sigma^{n'}$ as follows. We start with some base code $\mathcal{C}_0\colon \Sigma_0^k \to \Sigma_0^n$, as well as a bipartite expander graph $G = (R, L, E)$, where $L = [n]$ and $R = [n']$, for some $n' = O(n)$.[5] We will need $\mathcal{C}_0$ and G to have specific properties, which we will come to below. For $x \in \Sigma_0^k$, we generate the encoding $\mathcal{C}(x)$ as follows. For $j \in [n']$, the encoded symbol $\mathcal{C}(x)_j$ will be gotten as the concatenation of the symbols $\mathcal{C}_0(x)_i$ for $i \in \Gamma_G(j)$, where $\Gamma_G(j)$ denotes the neighbors of j in the graph G. This sort of "aggregation along an expander" technique, introduced in [1], has become a standard distance amplification technique in error correcting codes. Because of the concatenation, our final alphabet Σ will be $\Sigma = \Sigma_0^{m_2}$.

To perform list recovery, we will start with a small piece of "advice," and then recover the (hopefully unique) message x consistent with that advice. We will generate our final list $\mathcal{L}$ by iterating over all possible values of the advice. Towards this end, we will choose some coordinate $j \in [n']$ for which $\mathcal{L}_j$ is not erased, and some $\sigma^\star \in \mathcal{L}_j$ as our guess for $\mathcal{C}_0(x)|_{\Gamma_G(j)}$ to act as our advice. Given this advice $\sigma^\star$, we wish to keep propagating information until we obtain enough coordinates of $\mathcal{C}_0$ that would allow us to uniquely determine x; this amounts to decoding the code $\mathcal{C}_0$ from erasures.

In the exposition below, we start with a naive attempt to do this propagation, and build up the properties that we will need $\mathcal{C}_0$ and G to satisfy as we refine it. Our construction is depicted in Figure 1.

A naive attempt

Our first attempt (which will not work) is the following. Let $j \in [n']$ be as above, so we assume that we are given as advice the m_2 symbols $\mathcal{C}_0(x)|_{\Gamma_G(j)}$; our goal is to recover (a hopefully unique) x given this advice and given the input lists $\mathcal{L}_{j'}$ for $j' \in [n']$. Choose some coordinate $j' \in [n']$ such that $\Gamma_G(j) \cap \Gamma_G(j') \neq \emptyset$. As we already know the symbols in the

[5] Using the notation of the full version, $\Sigma_0 = \mathbb{F}_q$, $\Sigma = \mathbb{F}_q^{m_2}$ for some constant m_2, and $n' = |V_2| = O(n)$.

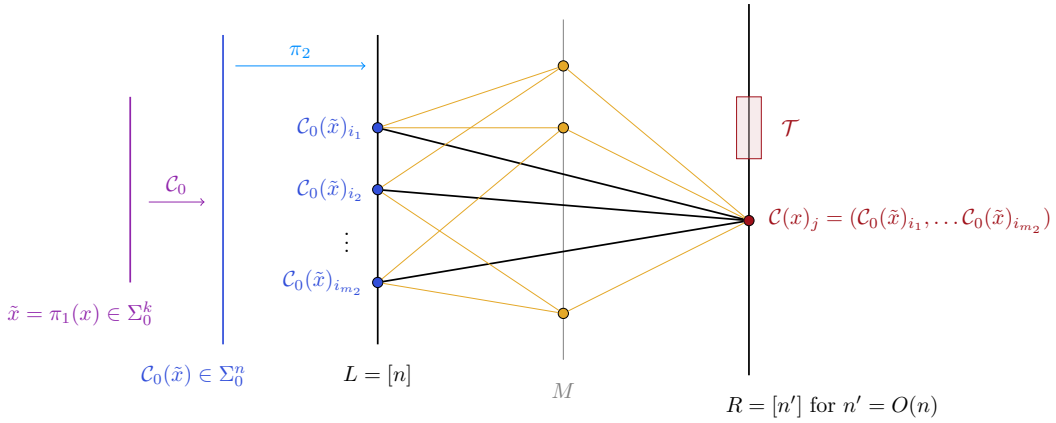

Figure 1 Illustration of our construction. The coordinates of the inner code $\mathcal{C}_0$ live on the vertices of L. The final code $\mathcal{C}$ consists of symbols aggregated by vertices in R. The randomness in the encoding comes from the permuations π_1 and π_2, which scramble the messages in Σ^k and the coordinates in $[n]$, respectively. We use the vertices in $\mathcal{T} \subseteq R$ to define parity checks that partially define the code $\mathcal{C}_0$. The "middle layer" M is not used in the definition of the code, but is a necessary auxiliary structure for our recovery algorithm.

coordinates indexed by $\Gamma_G(j)$, this gives us partial information about $\mathcal{C}(x)_{j'}$ in the form of $|\Gamma_G(j) \cap \Gamma_G(j')|$ elements of Σ_0 in known locations. One can hope that this information would be enough to pinpoint a specific entry in the list $\mathcal{L}_{j'}$, allowing us to recover all symbols of $\mathcal{C}_0(x)$ in the coordinates indexed by $\Gamma_G(j')$, and keep going in the same manner until enough information is propagated.

Clearly, when we have no guarantee on the input lists $\mathcal{L}_i$, this approach fails miserably, as it may be the case that $\mathcal{L}_{j'}$ contains numerous elements in $\Sigma_0^{m_2}$ that agree in some of the m_2 locations, and the information coming from our advice for j will not uniquely pin down an element of $\mathcal{L}_{j'}$. However, note that for a *completely random* input list $\mathcal{L}_{j'}$, such an attempt would be successful with probability at least $1 - |\mathcal{L}_{j'}| / |\Sigma_0|$, and we could set the parameters in such a way that $|\mathcal{L}_{j'}| \ll |\Sigma_0|$. That is, in this case it would become reasonably likely that the choice of $\sigma^\star \in \mathcal{L}_j$ would uniquely pin down an element $\sigma \in \mathcal{L}_{j'}$, allowing us to propagate information to another vertex in the graph. The hope is that we could propagate this information throughout the graph, using the fact that G is an expander to guarantee that most vertices will be determined. Of course, the problem with this is that we do not want to assume that the input lists are completely random, but this leads us to our next attempt, where we inject randomness into the encoding procedure.

Injecting randomness

While we won't get completely random lists $\mathcal{L}_j$ as we might have wanted for the naive attempt, we can make the input lists randomized via a randomized encoding. More specifically, our base code $\mathcal{C}_0$ will be deterministic, and to apply $\mathcal{C}$ we will make use of two permutations: a permutation π_1 acting on the universe $\mathcal{U}$ and a permutation π_2 acting on $[n]$. More formally, given $x \in \Sigma_0^k$, we first apply $\pi_1(x)$ and apply the encoding $\mathcal{C}_0$ to $\pi_1(x)$. Next, we permute the coordinates of the outcome according to π_2. Finally, we aggregate symbols according to G, yielding $\mathcal{C}(x) \in \Sigma^{n'}$. Roughly speaking, the first permutation – which will be pairwise independent – will make $\mathcal{C}_0(x)$ uniformly distributed over the code's image, even conditioned the value of $\mathcal{C}_0(x')$ for some $x' \neq x$. The second permutation will make sure that querying

any particular symbol $\mathcal{C}_0(x)_j$ symbol will behave like sampling a uniformly random symbol in $\mathcal{C}_0(x)$, and even more strongly, combined with π_1 it will behave like a random sampling from a nearly uniform distribution over Σ_0.

Analyzing the permutation-aided construction carefully, we are able to show that indeed, with probability roughly $1-\eta$ for $\eta \approx |\mathcal{L}_{j'}|/\sqrt{|\Sigma_0|}$, we can pinpoint a single list element of $\mathcal{L}_{j'}$. One conceptual observation that will help us establish that result is the fact that the distribution of symbols in most codewords of a high-rate code is close to uniform, and indeed we will need the rate of $\mathcal{C}_0$ to be very high (see Section 3.3 of the full version). We leave the more technical details to Section 5 of the full version.

Although promising, this approach is still problematic. We start with $m_2 = O(1)$ symbols that we know, and at each iteration the set of revealed coordinates grows by a small constant factor, using the expansion properties of G. As initially our sets are of constant size, we cannot hope for success probability much greater than $1-\eta$ for the initial propagation steps. A failure probability of η, even if we disregard the need for a problematic union bound over all propagation steps, is far too large for us, and in particular for our application to heavy hitters. The problem described here is common to various expander-based techniques, and in this work we resolve it by choosing G to be a special expander graph that comes from a high-dimensional expander, and by choosing $\mathcal{C}_0$ to be a suitable *Tanner code*. We discuss these modifications next.

Using high-dimensional expanders to get a good head start

We resolve the issue described above – that we cannot possibly get a good failure probability if we start with only a few known symbols – by using techniques from high-dimensional expanders. Suppose that, starting with only the advice $\sigma^\star$ for m_2 symbols of $\mathcal{C}_0(x)$, we could deterministically identify a large subset $\mathcal{T} \subseteq [n']$ for which we know all symbols of $\mathcal{C}_0(x)$ indexed by $\Gamma_G(\mathcal{T})$. This way, concentration bounds can kick in, and hopefully each propagation step would be successful with probability roughly $\eta^{|\mathcal{T}|}$, provided we can get a enough independence between query attempts at the same propagation step. We defer the independence issue to the full version (this ends up following from the amount of independence we have in our permutations π_1 and π_2), and concentrate on obtaining such a $\mathcal{T}$.

Recall that we work over the bipartite expander graph $G = (R = [n'], L = [n], E)$. We will construct G', a tripartite extension of G, with an added middle layer M, $|M| = O(n)$, having the following property. Identify each vertex j of R with a subset $\Gamma_G(j) \subset [n]$ of cardinally m_2 in the natural way. Each vertex in M is identified with a subset $S \subset [n]$ of cardinality m_1, for $1 < m_1 < m_2$, such that S is connected to all its m_1 elements on the left, and to all its supersets on the right. More specifically, each vertex j in R will be connected to all $\binom{m_2}{m_1}$ subsets of $\Gamma_G(j)$ in M. (See Figure 1 for an illustration.)

We will choose the code $\mathcal{C}_0$ to be a *Tanner code* with respect to the structure of the graph G. That is, as before, we associate the n symbols of a codeword $\mathcal{C}_0(x)$ with the left hand vertices L of G, and we define $\mathcal{C}_0$ so that a codeword $\mathcal{C}_0(x)$ is a labeling of L so that to following property holds: For every j in an appropriate subset $\mathcal{T} \subset R$, the labels on the vertices of $\Gamma_G(j)$ form a codeword in some error correcting code $\mathcal{C}_{00}$ of length m_2 with good distance; in particular, given any m_1 symbols of $\mathcal{C}_{00}(x')$ for some x', we can recover all of $\mathcal{C}_{00}(x')$.

The reason to choose $\mathcal{C}_0$ like this is the following. Say we know that j and j' are in the set $\mathcal{T}$, and that they have a common neighbor in M. This implies that $|\Gamma_G(j) \cap \Gamma_G(j')| \geq m_1$, since there is some set of size m_1 that both of those sets contain. In particular, by our choice of $\mathcal{C}_{00}$, once we know the symbols of $\Gamma_G(j)$, we can *deterministically* reveal all symbols of

$\Gamma_G(j')$ by decoding $\mathcal{C}_{00}$. Then we can continue this process until we recover the symbols in $\Gamma_G(j)$ for all $j \in \mathcal{T}$. By counting constraints, it turns out that we can choose $\mathcal{T}$ to be large and still have a high-rate code $\mathcal{C}_0$. This gives us our set $\mathcal{T}$ so that we can deterministically fill in the symbols of $\Gamma_G(\mathcal{T})$ to use as a head start and increase our success probability.

How do we construct such a tripartite graph, that on the one hand has not too many vertices in R and M (i.e., $R = O(n)$ and $M = O(n)$), but on the other hand has favorable intersection and expansion properties? This is where *high-dimensional expanders* enter the picture, and indeed the tripartite graph comes from an $(m_2 - 1)$-dimensional simplicial complex (see the full version for the formal definitions). A similar object was used by Dinur et al. [9] as a *double sampler*, and in Dikstein et al. [8] as a *multilayer agreement sampler*. We note that the construction of [9] is quite similar to ours, as they also use the symbol-aggregation technique of Alon et al. [1]; the main difference in the construction is that we use a very specific inner code $\mathcal{C}_0$ that uses the structure of G as part of its parity checks, while the work of [9] chooses $\mathcal{C}_0$ to be an arbitrary code with good distance.

In our actual construction, the code $\mathcal{C}_0$ is a bit more involved, and its constraints arise both from the special subset $\mathcal{T}$ of R and from an *additional* bipartite expander. Each of the two types of constraints is helpful for a different aspect of our algorithm. Roughly speaking, the constraints that come from $\mathcal{T} \subseteq R$ help us as described above (filling in the set $\Gamma_G(\mathcal{T})$ to get a head start). The other constraints are there to ensure that the final code $\mathcal{C}_0$ has good enough distance to allow for the final unique decoding. All in all, we are able to achieve a set $\mathcal{T}$ that has size about $|\mathcal{T}| \approx \text{poly}(\varepsilon) \cdot n$. We remark that this is the point where we don't quite get the failure probability that we want, resulting in a sub-optimal dependence on ε for our application to heavy hitters: we want failure probability $\exp(-n)$ (we will choose n logarithmic in N, so this would be $\text{poly}(1/N)$), and we end up with failure probability $\exp(-|\mathcal{T}|) = \exp(-\text{poly}(\varepsilon)n)$.

There are plenty of details that are swept under the rug in the description above, including implementation details needed to keep the recovery algorithm linear-time. We give the recovery algorithm in detail in the full version of the paper. We present our list-recovery algorithm in the context of a query algorithm for heavy hitters, since for our analysis we want to focus on the distribution of input lists that arises from the heavy hitters example, and it is easiest to present everything together. In particular, the input lists do not arise simply from the union of ℓ codewords $\mathcal{C}(x)$, but (a) may be erased if the corresponding small data structure failed, and (b) may contain extraneous symbols that arise from items $x^{(i)}$ that appear in the stream that are not heavy hitters.

1.2 Motivating Goal 1 from Pseudorandomness

In this section, we briefly explain why Goal 1 – and in particular, getting a good dependence on the parameter ℓ – is of interest in pseudorandomness. There is a tight connection between error correcting codes and fundamental constructions in pseudorandomness, notably the equivalence between (strong) seeded extractors and list-decodable codes [41, 40]. It turns out that list recovery can also play a prominent role in the study of related objects from extractor theory. In *seeded condensers*, first studied in [38], the goal is to "improve" the quality of a random source $\boldsymbol{X}$ using few additional random bits. A bit more formally, given a random variable $\boldsymbol{X} \sim \{0,1\}^n$ with min-entropy k, a condenser $\mathsf{Cond}\colon \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ is such that $\mathsf{Cond}(\boldsymbol{X}, U_d)$ has min-entropy k', where we want the entropy rate to improve, namely, $\frac{k'}{m} \gg \frac{k}{n}$, and to maintain a small entropy gap $m - k'$. (For the formal definition,

see, e.g., [18].) List recoverable codes in the *errors* model[6] give seeded condensers, and vice versa. More specifically, the input and output entropies k and k' are almost in one-to-one correspondence with the (logarithm of the) output and input list sizes, $\log|\mathcal{L}|$ and $\log \ell$ (for the precise statement, see [10]). Thus, to get meaningful condensers from list-recoverable codes, the dependence between L and ℓ needs to be good, in all regime of parameters, and in particular handle ℓ that grows arbitrarily with the message length. In fact, the best list-recoverable code in this regime is the (folded) Parvaresh-Vardy code [18], giving $|\mathcal{L}| \approx \ell$.[7] The connection between condensers and list-recoverable codes was recently utilized in the *computational* setting to construct nearly-optimal pseudorandom generators for polynomial-sized circuits [10].

The model of *zero-error list recovery*, described in Goal 1 (when $|\mathcal{L}|$ depends nicely on ℓ and ℓ can be arbitrary), has applications to pseudorandomness too. A (strong) *disperser* is a function $\mathsf{Disp}\colon \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ such that for any random variable $\boldsymbol{X} \sim \{0,1\}^n$ with sufficient min-entropy, the support of $\mathsf{Disp}(\boldsymbol{X}, U_d)$ is large. Such dispersers have found several applications, and are tightly connected to open problems in expander graphs. It is not hard to show, and we do so in Appendix B of the full version, that dispersers, in some parameter regime, are equivalent to zero-error list-recoverable codes. We are not aware of this equivalence being stated elsewhere. For completeness, we note that dispersers in another parameter regime give rise to erasure list-decodable codes [3].

Finally, observe that in order to get good pseudorandomness primitives from list-recoverable codes, efficient recovery is not an issue, and all that is needed is an efficient *encoding*.

Even though a probabilistic guarantee as in Theorem 2 does not immediately yield improved pseudorandom objects, it is our hope that our progress on Goal 1 is a first step towards achieving that goal, which would imply improved dispersers.

1.3 Related Work

Algorithmic List-Recovery

List-recovery was originally introduced as an avenue towards *list-decoding*, where the goal is, given a vector $z \in \Sigma^n$, to recover the list $\mathcal{L}$ of all messages $x \in \Sigma^k$ so that $\mathcal{C}(x)$ is sufficiently close to z in Hamming distance. For example, the celebrated list-decoding algorithm of Guruswami and Sudan for Reed-Solomon codes [17] is in fact a list-recovery algorithm. However, the Guruswami-Sudan algorithm stops working at the so-called *Johnson bound*, which in the context of list-recovery means that the rate k/n of the code can be at most $1/\ell$. Since the Guruswami-Sudan algorithm, there has been a great deal of work, mostly based on algebraic constructions, aimed at surpassing the Johnson bound for list-decoding and list-recovery. In particular, the works [16, 19, 27, 28] show variations of Reed-Solomon codes, like folded RS codes and multiplicity codes, can be efficiently list-decoded and list-recovered beyond the Johnson bound. For list-recovery, these constructions are able to obtain rate $k/n = \Omega(1)$, but unfortunately the size of the lists $\mathcal{L}$ returned (and in particular the running time of the algorithm that returns that list) is at least quasipolynomial in ℓ [16, 28], and sometimes exponential in ℓ. Moreover, those constructions naturally have large alphabet sizes, polynomial in n. In order to reduce the alphabet size, constructions using algebraic

[6] In the errors model, we are given $\mathcal{L}_1, \ldots, \mathcal{L}_n \subseteq \Sigma$ with $|\mathcal{L}_i| \leq \ell$ for all i, and we require the list $\mathcal{L} = \left\{x \in \Sigma^k : \Pr_{i\in[n]}[\mathcal{C}_i(x) \in \mathcal{L}_i] \geq 1-\gamma\right\}$ to be small, for some error parameter γ.

[7] Note, however, that the rate of the code in [18] is only $k^{-\Omega(1)}$ for k being the message length.

geometry codes have been used (e.g. [20, 21, 15]), although these works still have parameters with an exponential dependence on ℓ. Moreover, all of the works mentioned above have polynomial – and not linear – time recovery algorithms. Using expander-based techniques (e.g. that of Alon et al. [2]), these algorithms can be improved to near-linear time in n (e.g., as in [22]), but at the cost of increasing the dependence on ℓ to doubly-exponential.

In addition to algebraic constructions, there have also been a few constructions of purely graph-based codes, which are more similar to our constructions. The work of Guruswami and Indyk [14] gives a linear-time algorithm for list-recovery of graph-based codes, which does even better in the setting of *mixture-recovery* (similar to the setting that we study here) where the input lists are generated from unions of codewords. That work achieves output list size $|\mathcal{L}|$ exactly equal to ℓ, but has rate $O(1/\ell)$ and the alphabet size is exponential in ℓ. The work of Hemenway and Wootters [23] gives an $O(n)$-time algorithm for list-recovering graph-based codes (the expander codes of [39, 42], with an appropriate inner code); these can have high rate (close to 1), but unfortunately the dependence on ℓ in other parameters is *quadruply*-exponential.

The work of Dinur, Harsha, Kaufman, Livni-Navon and Ta-Shma [9], which directly inspired our work, used *double-samplers* derived from high-dimensional expanders, combined with an expander-based symbol aggregation technique of ABNNR that we also use [1]. The goal of that work was to give an efficient list-decoding algorithm for *any* code that follows the ABNNR construction. This is much more general that what we are aiming to do (since we get to carefully design our code before applying the ABNNR construction), and also the goal is different (list-decoding in the worst case, rather than randomized list-recovery). That work is able to get efficient (polynomial-time) algorithms, but when one tries to turn their algorithm into a list-recovery algorithm in the most direct way, the parameters are not close to those in Goal 1; in particular, the algorithm is only $\text{poly}(n)$-time, and the dependence on ℓ is again exponential. It is not clear (to us) how to use the approach of [9] to achieve Goal 1.

We also mention a recent work of Dikstein, Dinur, and Harsha [8] that suggests an approach for constructing locally testable codes. In particular, as in our construction they also use the underlying graph (an *agreement expander* coming from a high-dimensional expander) both for symbol manipulation and for defining the parity checks. However, their goal is quite different than ours: they obtain locally testable codes via lifting a set of "smaller" locally testable codes, extending the natural Tanner tests.

Heavy Hitters

The first work with provable guarantees for the heavy hitters problem was by Misra and Gries [32], which applied to the *cash register* model where each of the updates $\Delta^{(i)}$ are equal to 1. We work in the more general strict turnstile model described above. For the strict turnstile model, the Count-Min Sketch data structure of [7] above already gets good results, and the best current results for the parameter regime we are motivated by (in particular, with failure probability $1/\text{poly}(N)$, and where $\log N \gg \text{poly}(1/\varepsilon)$) are those of [29] described above. It is known [25] that $\Omega(\varepsilon^{-1} \log N)$ words of memory are required for this setting, and thus the space used by these works are optimal.

We next mention three works that study heavy hitters when the failure probability is extremely small (or zero) [31, 5, 33]. Relative to our work, these works achieve – as with [29] – a better dependence on ε but worse dependence on N; however, these works can additionally get away with extremely small or even zero failure probability. In [31], Li et al. modify the Count-Min Sketch by looking at different hash functions, and they present a data

structure with failure probability δ with space $\widetilde{O}\left(\log(\varepsilon N)\left(\varepsilon^{-1}+\log(1/\delta)\right)\right)$, update time $\widetilde{O}(\log^2(1/\varepsilon)\log(\varepsilon N)\left(1+\varepsilon\log(1/\delta)\right)$, and query time $\widetilde{O}(\varepsilon^{-1}\log^2(1/\varepsilon)\log(\varepsilon N)\log(1/\delta))$. For $\delta = N^{-c}$ and $\log N \gg \text{poly}(1/\varepsilon)$, this gives the parameters stated in Table 1. However, when δ is much smaller – for example, $\delta = N^{-\Omega(1/\varepsilon)}$ – this gives better results than the works previously discussed, and in particular implies a result that is *uniform* over all sets of heavy hitters by union bounding over the $N^{O(1/\varepsilon)}$ choices for such sets. In [5], Cheraghchi and Nakos give a randomized construction of a data structure that also solves the heavy hitters problem uniformly over all streams $x^{(1)}, x^{(2)}, \ldots$ (that is, with error probability *zero* assuming that the data structure was constructed correctly). This scheme uses space $O(\varepsilon^{-1}\log(N\varepsilon))$, has update time $\tilde{O}(\log^2(1/\varepsilon)\log(\varepsilon N))$, and query time $\varepsilon^{-1}\,\text{polylog}(N)$.[8] That work actually provides solutions to several problems, not just heavy hitters, via a construction of *list-disjunct matrices*. Finally, we mention the work of Nelson et al. [33], which gives a fully deterministic construction of a data structure for heavy hitters (and more generally for ℓ_∞/ℓ_1 sparse recovery) with zero error probability; the space and query time is $O(\varepsilon^{-2}\,\text{poly}\log(N))$, and the update time is $O(\varepsilon^{-1}\,\text{poly}\log(n))$.[9]

We note that there are algorithms that achieve $O(\log N)$ update and query time for constant ε, but with only a constant failure probability. For example, such an algorithm is given in the full version of [29] (see [30, Theorem 10]).

One can generalize to the *general turnstile* model, where there is no guarantee that $f(x)$ is positive at each point in the stream, and one can generalize to ℓ_p-heavy hitters, where the goal is to return all x so that $|f(x)| \geq \varepsilon\|f\|_p$. There has been a great deal of work along both of these lines; see [29] and the references therein. In particular, for ℓ_p heavy hitters in the general turnstile model, the work [29] gives a data structure with space $O(\varepsilon^{-p}\log N)$, update time $O(\log N)$, and query complexity $\varepsilon^{-p}\,\text{polylog}(n)$.

We briefly discuss the approach of [29], in order to illustrate the differences between their approach and ours. While that work inspired the list-recovery approach we take, and they also use error correcting codes and expander graphs, the construction itself is quite different. That work takes the perspective of *graph clustering*. In more detail, their sketch can output a graph in which each heavy hitter is represented by a well-connected cluster in the graph. They then develop a clustering algorithm that can recover the clusters, and hence the heavy hitters. In order to make the connection to graph clustering, they first encode x with an error correcting code $\mathcal{C}_0$ as we do; but they only need this code to have good distance, as they do not go down the list-recovery route. Then they break $\mathcal{C}_0(x)$ up into n' chunks. Before putting the j-th of these chunks into the j-th smaller data structure, they append it with tags $h_j(x)$ and $\{h_{\Gamma(j)_i}(x)\}$, where the h_j are hash functions and Γ is the adjacency function for an expander graph G. Thus, the j-th chunk of $\mathcal{C}_0(x)$ is essentially connected by edges in G to the other chunks of $\mathcal{C}_0(x)$, and in particular the chunks of $\mathcal{C}_0(x)$ form a cluster that can be recovered by a clustering algorithm.

We note that [4] was also inspired by [29], and builds on their approach to develop differentially private heavy-hitters algorithms. In fact, that work even casts the scheme of [29] as a list-recovery scheme, in a relaxed definition of list-recovery that is different

[8] We note that here the guarantee is to return a list $\mathcal{L}$ of size $O(1/\varepsilon)$ containing all the true heavy hitters, although in both [31] and [5], the list is allowed to contain elements with frequencies $f(x) \ll \varepsilon\|f\|_1$, while most of the heavy-hitters work surveyed above, including ours, does not have such false positives.

[9] We note that in [33], the query time to recover the list of the heavy hitters is $\Omega(N)$ and the space involves a single factor of $\log(N)$, but the "dyadic trick" can be used to obtain the bounds mentioned above.

from our relaxed version in Theorem 2. In particular, their notion of list-recovery will not handle input lists $\mathcal{L}_1, \ldots, \mathcal{L}_n$ that are generated by any ℓ distinct messages, as we handle in Theorem 2.[10]

Algorithmic applications of list-recovery

Our work is inspired by the use of list-recovery in [29], but there is a rich history of using list-recoverable codes in similar algorithmic applications. One example is *group testing*, where the goal is to identify d "positive" items out of a universe of size N, given tests of the form $\bigvee_{i \in I} \mathbf{1}[i \text{ is positive}]$ for subsets $I \subset [N]$. A classic construction of Kautz and Singleton [26] reduces this question to the question of list-recovery. This connection, and elaborations on it, has been exploited in several works, which aim to both minimize the number of tests and to develop sublinear-time algorithms to recover the set of positive items [24, 35].

A second example, even closer to our work, is in *compressed sensing*. In compressed sensing, the goal is to approximately recover an approximately sparse vector $v \in \mathbb{R}^N$ given linear measurements Av for some $A \in \mathbb{R}^{t \times N}$. The heavy-hitters problem is closely related, as a (linear) solution to the heavy hitters problem can approximately recover the support of v. List-recoverable codes have been used in the context of compressed sensing in a similar way as it was used in [29]: associate each $i \in [N]$ with a message, and encode it with a list-recoverable code to get a codeword $\mathcal{C}(i) = (c_1, \ldots, c_n) \in \Sigma^n$. Then reduce the compressed sensing problem to n smaller instances of the same problem for vectors of length $|\Sigma|$: for each $j \in [n]$, we have a vector $w^{(j)}$ indexed by Σ so that the entry $w^{(j)}_\sigma$ is obtained by aggregating all of the coordinates v_i of v so that $\mathcal{C}(i)_j = \sigma$. Now we can either recurse or solve these smaller problems in another way. Previous works [36, 13, 12] have observed that a good list-recoverable code (e.g., satisfying Goal 1) would solve this problem. However, they ran into the same issue that we did, namely that we do not know of any such codes. Instead, they either used sub-optimal codes or developed work-arounds, as we describe below.

The work of Ngo et al. [36] was, to the best of our knowledge, the first to apply list-recovery in compressed sensing. We mention two results in that work that use a framework quite similar to that of [29] (and thus to ours), making explicit use of (sub-optimal) list-recoverable codes. The first result is based on the list-recoverability of Reed-Solomon codes. As RS codes do not achieve Goal 1, this results in a sub-optimal number of measurements, but is nice and simple. The second is based on Parvaresh-Vardy (PV) codes. PV codes have good rate and output list-size, but unfortunately the alphabet size is very large. To get around this, [36] (inspired by the work [35] on group testing mentioned above) considered a code constructed by repeatedly concatenating PV codes with themselves. This does not lead to a code that achieves Goal 1 – the rate depends on ℓ, and either the alphabet size or the rate must depend on n – but they are able to make these dependencies not too bad. This leads to schemes with near-optimal number of measurements t, although the schemes only work for non-negative signals. Further, since PV codes do not have near-linear-time algorithms, the recovery algorithm runs in time $\text{poly}(t)$ rather than near-linear in t.

[10] In a bit more detail, the notion of list-recovery in [4] allows for $\mathcal{L}_1, \ldots, \mathcal{L}_n$ to be generated by ℓ messages $x^{(1)}, \ldots, x^{(\ell)}$, *provided* that the messages lie in distinct "buckets," according to any fixed bucketing of the message space. The choice of the code may depend on the bucketing. In this language, the result of [29] (see [4, Theorem 3.6]) says that it is possible to obtain a code with constant rate, output list size $L = O(\ell)$, and alphabet size that is polynomial in the number of buckets, and a polynomial-time decoding algorithm. If the number of buckets is $\text{poly}(\ell)$, the alphabet size is also $\text{poly}(\ell)$, as we would hope, but as the number of buckets grows (to approach the general case with $|\Sigma|^k$ buckets of size one, where there is no "bucketing" restriction) the alphabet size grows accordingly.

The work of Gilbert et al. [13] follows a similar outline, using the Loomis-Whitney-based codes of [34]. For $d > 0$ some integer parameter, these are codes $\mathcal{C}\colon [N] \to [N^{1/d}]^{d-1}$ are $(\ell, \ell^{d/(d-1)})$-list-recoverable in time $O(\ell^{d/(d-1)} \log N)$. In terms of the desiderata of Goal 1, this does give near-linear-time recovery with good dependence on ℓ; however the alphabet size is huge, growing exponentially in the message length. In [13], they deal with this by applying the scheme mentioned above recursively until the alphabet size becomes manageable. As a result, they are able to get a nearly optimal number of measurements, with a recovery time that depends polynomially (but not linearly) on $\log N$, and with an extremely small error probability, smaller than $1/\operatorname{poly}(N)$.

We also mention [12], which uses list-recoverable codes (PV codes) in a more complicated way to achieve a near-optimal compressed sensing algorithm in the uniform ("forall") model. They also treat the indices i as messages and encode them with a list-recoverable code, but they develop more machinery – using an expander to add linking information between the symbols for example – in order to reduce to the list-recovery problem.

1.4 Open Questions and Future Work

In this work we have made progress towards Goal 1 by constructing a *randomized* code that supports, with high probability, linear-time list-recovery from certain lists. This was enough for our application to heavy hitters, but many open questions still remain.

1. The most obvious open question is to fully attain Goal 1. In addition to furthering our knowledge in algorithmic coding theory, it seems likely that attaining Goal 1 (or the techniques used to do it), would have other applications in algorithm design, as well as in pseudorandomness (as per Section 1.2).
2. While we are able to use techniques from high-dimensional expansion to obtain a failure probability of $N^{-\Omega(\varepsilon^3)}$ (in the setting of ε-heavy hitters), we would like a failure probability of $N^{-\Omega(1)}$.
3. In this paper we studied only *zero-error* list-recovery (or, more accurately, list-recovery from a small fraction of erasures). While this question is interesting and challenging on its own, one can ask about extending our results to list-recovery from errors. In particular, this might lead to improved heavy-hitters schemes in the general turnstile model.
4. We motivated our "probabilistic list-recovery" model by an application to heavy hitters. However, we hope that there are many other algorithmic applications for such a model and for our construction. Indeed, there are several algorithmic applications of list-recovery mentioned in Section 1.3 (e.g., [24, 35, 36, 13]) that explicitly use list-recoverable codes and would be improved by codes that achieve Goal 1. It is our hope that some of these applications could also be improved by better constructions of the probabilistically list-recoverable codes that we study here. As one example, if one could obtain Theorem 2 with $|\Sigma| = \widetilde{O}(\ell)$ (rather than polynomial in ℓ), then by the construction of Kautz and Singleton mentioned above [26] this would yield optimal constructions of probabilistic group testing matrices with sublinear-time decoding, matching a recent result of [37] in a black-box way.

2 Randomized List Recovery

Inspecting our main theorem's proof (in particular, Section 5 in the full version), we can extract a list recovery result for our (randomized) code $\mathcal{C}$ that tolerates a small fraction of erasures. Our randomized encoding can handle input lists that come from a union of

codewords $\{\boldsymbol{\mathcal{C}}(x) : x \in \mathcal{L}_0\}$ for some $\mathcal{L}_0 \subseteq \mathbb{F}_q^k$; this is what we stated in Theorem 2. Moreover, our algorithm can also handle some extra "distractor symbols," provided that those symbols are randomized and unlikely to collide with the symbols that come from $\mathcal{L}_0$. In order to state this formally, we first give a definition that captures the sort of input lists that our algorithm can handle.

▶ **Definition 4.** *Let $\boldsymbol{\mathcal{C}} \colon \mathbb{F}_q^k \to \Sigma^{n'}$ be a randomized encoding, for $\Sigma = \mathbb{F}_q^b$. Consider a randomized function of $\mathcal{C} \sim \boldsymbol{\mathcal{C}}$ and a set of messages $\mathcal{L}_0 \subseteq \mathbb{F}_q^k$, that outputs lists $\mathcal{L}_1, \ldots, \mathcal{L}_{n'} \subseteq \Sigma$. We say that such a function is (t, η)-*nice *w.r.t. $\boldsymbol{\mathcal{C}}$ if the following holds for all $\mathcal{L}_0 \subseteq \mathbb{F}_q^k$ (note that the lists $\mathcal{L}_i$ can depend on $\mathcal{L}_0$):*

1. *For any $i \in [n']$,*
   - *with probability at least $1 - \eta$, $|\mathcal{L}_i| = O(|\mathcal{L}_0|)$, and,*
   - *with probability 1, $\boldsymbol{\mathcal{C}}(x)_i \in \mathcal{L}_i$ for all $x \in \mathcal{L}_0$.*
2. *For any $x \in \mathcal{L}_0$ and $i \in [n']$, with probability at least $1 - \eta$ it holds that $(\boldsymbol{\mathcal{C}}(x)_i)_j \neq \sigma_j$ for every $j \in [b]$ and $\sigma \in \mathcal{L}_i \setminus \{\boldsymbol{\mathcal{C}}(y)_i : y \in \mathcal{L}_0\}$.*

Furthermore, we require that the above properties should hold t-wise independently across the lists. Namely, for any $x \in \mathcal{L}_0$, whether (1) and (2) hold for some $i \in [n']$ is independent of whether it holds for any $t - 1$ other values of $i' \in [n']$.

To illustrate this definition, we give a few examples.

▶ **Example 5** (lists from a union of codewords)**.** The simplest example of a nice distribution is the function that gives

$$\mathcal{L}_i = \{\boldsymbol{\mathcal{C}}(x)_i : x \in \mathcal{L}_0\}.$$

That is, the lists $\mathcal{L}_i$ are just given by the union of the codewords in $\mathcal{L}_0$. To see that this is $(\eta = 0, t = n')$-nice, observe that both (1) and (2) hold deterministically, with probability 1. Indeed, (1) holds by construction, and (2) holds because there are no $\sigma \in \mathcal{L}_i \setminus \{\boldsymbol{\mathcal{C}}(y)_i : y \in \mathcal{L}_0\}$, so the condition is trivial.

▶ **Example 6** (lists with random distractor symbols)**.** Another natural example of a nice distribution is the example above, with some uniformly random extra "distractor" symbols. That is,

$$\mathcal{L}_i = \{\boldsymbol{\mathcal{C}}(x)_i : x \in \mathcal{L}_0\} \cup \{\sigma_{i,h} : h \in [r]\}$$

where $r > 0$ is some parameter and where $\sigma_{i,h}$ are i.i.d. and uniform in Σ. Again, this satisfies item (1) deterministically, provided that $r = O(|\mathcal{L}_0|)$. For (2), we can compute the probability of a collision between the distractor symbols $\{\sigma_{i,j} : j \in [r]\}$ and a given codeword $\boldsymbol{\mathcal{C}}(x)$ for $x \in \mathcal{L}_0$:

$$\begin{aligned}\Pr\left[(\boldsymbol{\mathcal{C}}(x)_i)_j \neq \sigma_j \ \forall j \in [b], \sigma \in \{\sigma_{i,h} : h \in [r]\}\right] &= \left(1 - \frac{1}{q}\right)^{br} \\ &\leq \exp(-br/q).\end{aligned}$$

In particular, when $q \gg br$, this is $1 - O\left(\frac{br}{q}\right)$. Thus, this distribution is (η, t)-nice where $\eta = O(br/q)$ and $t = n$.

Finally, we note that the distribution of distractor symbols that arises in our heavy hitters application is also nice for the code $\mathcal{C}$ that we use. The first point of Item (1) holds because the lists $\mathcal{L}_i$ can only become too large if the inner InnerHH fails and includes items that are not $\varepsilon/4$-heavy hitters. The second point of Item (1) holds because our instantiation of InnerHH has only one-sided error. Item (2) holds even for any $\sigma \in \mathcal{L}_i \setminus \{\mathcal{C}(x)_i\}$, which follows from Lemma 5.4 in the full version.

With this definition in place, we can now state our main theorem for list-recovery. Theorem 7 generalizes Theorem 2, because it allows for input lists with some extra "distractor" symbols, as per Definition 4.

▶ **Theorem 7.** *There exist constants $c > 1$ and $\gamma \in (0, 1)$ such that the following holds for any positive integers k and $\ell \leq k^\gamma$. There exists a randomized encoding $\mathcal{C} \colon \mathbb{F}_q^k \to \Sigma^{n'}$, for $q = \mathrm{poly}(\ell)$, $\Sigma = \mathbb{F}_q^{O(1)}$ and $n' = \Theta(k)$, and a randomized list recovery algorithm $\mathcal{A}$ running in time $\ell^c \cdot k$, with the following guarantee.*

For some constant $\eta < 1$, and an integer $t = \frac{k}{\mathrm{poly}(\ell)}$, for any list of messages $\mathcal{L}_0 \subseteq \mathbb{F}_q^k$ of size ℓ, and any distribution over input lists $\mathcal{L}_1, \ldots, \mathcal{L}_{n'}$ to $\mathcal{A}$ which are randomized functions of $\mathcal{C}$ and $\mathcal{L}_0$ and are (t, η)-nice w.r.t. $\mathcal{C}$, the list recovery algorithm $\mathcal{A}$, with probability $1 - \ell^{-\Omega(k)}$ (over the randomness of the encoding and the lists), outputs $\mathcal{L} \subseteq \mathbb{F}_q^k$ of size $O(\ell)$ such that $\mathcal{L}_0 \subseteq \mathcal{L}$. Furthermore, the encoding time of $\mathcal{C}$ is $O(k \log \ell)$, with a preprocessing step which takes $\mathrm{poly}(k)$ time.

We stress that unlike in standard state-of-the-art efficient list recovery algorithms, here we have a good dependence on ℓ, namely $q = \mathrm{poly}(\ell)$ and $|\mathcal{L}| = O(\ell)$.

We hope that Theorem 7 will find more applications. As discussed in Section 1.3, there are many algorithmic applications of list-recovery in the literature, and several previous applications of list-recovery have ended up with sub-optimal parameters due to the unavailability of codes that achieve Goal 1. It seems possible that Theorem 7 (or a further improvement on our techniques) could lead to improved results in (non-uniform or "for-each") group testing or compressed sensing.

References

1. Noga Alon, Jehoshua Bruck, Joseph Naor, Moni Naor, and Ron M. Roth. Construction of asymptotically good low-rate error-correcting codes through pseudo-random graphs. *IEEE Transactions on information theory*, 38(2):509–516, 1992.
2. Noga Alon, Jeff Edmonds, and Michael Luby. Linear time erasure codes with nearly optimal recovery. In *36th Annual Symposium on Foundations of Computer Science (FOCS 1995)*, pages 512–519. IEEE, 1995.
3. Avraham Ben-Aroya, Dean Doron, and Amnon Ta-Shma. Near-optimal erasure list-decodable codes. In *35th Computational Complexity Conference (CCC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
4. Mark Bun, Jelani Nelson, and Uri Stemmer. Heavy hitters and the structure of local privacy. *ACM Transactions on Algorithms (TALG)*, 15(4):1–40, 2019.
5. Mahdi Cheraghchi and Vasileios Nakos. Combinatorial group testing and sparse recovery schemes with near-optimal decoding time. In *61st Annual Symposium on Foundations of Computer Science (FOCS 2020)*. IEEE, 2020. To appear.
6. Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 19(1):3–20, 2010.
7. Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

8 Yotam Dikstein, Irit Dinur, Prahladh Harsha, and Noga Ron-Zewi. Locally testable codes via high-dimensional expanders. *arXiv preprint*, 2020. `arXiv:2005.01045`.

9 Irit Dinur, Prahladh Harsha, Tali Kaufman, Inbal Livni Navon, and Amnon Ta-Shma. List decoding with double samplers. In *ACM-SIAM 38th Annual Symposium on Discrete Algorithms (SODA 2019)*, pages 2134–2153. SIAM, 2019.

10 Dean Doron, Dana Moshkovitz, Justin Oh, and David Zuckerman. Nearly optimal pseudorandomness from hardness. In *61st Annual Symposium on Foundations of Computer Science (FOCS 2020)*, pages 1057–1068. IEEE, 2020.

11 Dean Doron and Mary Wootters. High-probability list-recovery, and applications to heavy hitters. In *Electronic Colloquium on Computational Complexity (ECCC)*, 2021. Manuscript.

12 Anna C. Gilbert, Yi Li, Ely Porat, and Martin J. Strauss. For-all sparse recovery in near-optimal time. *ACM Transactions on Algorithms (TALG)*, 13(3):1–26, 2017.

13 Anna C. Gilbert, Hung Q. Ngo, Ely Porat, Atri Rudra, and Martin J. Strauss. ℓ_2/ℓ_2-foreach sparse recovery with low risk. In *International Colloquium on Automata, Languages, and Programming (ICALP 2013)*, pages 461–472. Springer, 2013.

14 Venkatesan Guruswami and Piotr Indyk. Linear-time list decoding in error-free settings. In *International Colloquium on Automata, Languages, and Programming (ICALP 2004)*, pages 695–707. Springer, 2004.

15 Venkatesan Guruswami and Swastik Kopparty. Explicit subspace designs. *Combinatorica*, 36(2):161–185, 2016.

16 Venkatesan Guruswami and Atri Rudra. Explicit codes achieving list decoding capacity: Error-correction with optimal redundancy. *IEEE Transactions on Information Theory*, 54(1):135–150, 2008.

17 Venkatesan Guruswami and Madhu Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. In *39th Annual Symposium on Foundations of Computer Science (FOCS 1998)*, pages 28–37. IEEE, 1998.

18 Venkatesan Guruswami, Christopher Umans, and Salil Vadhan. Unbalanced expanders and randomness extractors from parvaresh–vardy codes. *Journal of the ACM (JACM)*, 56(4):1–34, 2009.

19 Venkatesan Guruswami and Carol Wang. Linear-algebraic list decoding for variants of reed–solomon codes. *IEEE Transactions on Information Theory*, 59(6):3257–3268, 2013.

20 Venkatesan Guruswami and Chaoping Xing. Folded codes from function field towers and improved optimal rate list decoding. In *44th Annual Symposium on Theory of Computing (STOC 2012)*, pages 339–350. ACM, 2012.

21 Venkatesan Guruswami and Chaoping Xing. List decoding Reed-Solomon, algebraic-geometric, and Gabidulin subcodes up to the Singleton bound. In *45th Annual Symposium on Theory of Computing (STOC 2012)*, pages 843–852. ACM, 2013.

22 Brett Hemenway, Noga Ron-Zewi, and Mary Wootters. Local list recovery of high-rate tensor codes and applications. *SIAM Journal on Computing*, pages FOCS17–157, 2019.

23 Brett Hemenway and Mary Wootters. Linear-time list recovery of high-rate expander codes. *Information and Computation*, 261:202–218, 2018.

24 Piotr Indyk, Hung Q. Ngo, and Atri Rudra. Efficiently decodable non-adaptive group testing. In *ACM-SIAM 21st Annual Symposium on Discrete Algorithms (SODA 2010)*, pages 1126–1142. SIAM, 2010.

25 Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight bounds for ℓ_p samplers, finding duplicates in streams, and related problems. In *ACM SIGMOD-SIGACT-SIGART 30th Annual Symposium on Principles of Database Systems*, pages 49–58, 2011.

26 William Kautz and Roy Singleton. Nonrandom binary superimposed codes. *IEEE Transactions on Information Theory*, 10(4):363–377, 1964.

27 Swastik Kopparty. List-decoding multiplicity codes. *Theory of Computing*, 11(1):149–182, 2015.

28 Swastik Kopparty, Noga Ron-Zewi, Shubhangi Saraf, and Mary Wootters. Improved decoding of folded Reed-Solomon and multiplicity codes. In *59th Annual Symposium on Foundations of Computer Science (FOCS 2018)*, pages 212–223. IEEE, 2018.

29 Kasper Green Larsen, Jelani Nelson, Huy L. Nguyễn, and Mikkel Thorup. Heavy hitters via cluster-preserving clustering. In *57th Annual Symposium on Foundations of Computer Science (FOCS 2016)*, pages 61–70. IEEE, 2016.

30 Kasper Green Larsen, Jelani Nelson, Huy L. Nguyễn, and Mikkel Thorup. Heavy hitters via cluster-preserving clustering. arxiv:1604.01357 [cs.DS], 2016.

31 Yi Li, Vasileios Nakos, and David P. Woodruff. On low-risk heavy hitters and sparse recovery schemes. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2018)*, volume 116, pages 19:1–19:13. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2018.

32 Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

33 Jelani Nelson, Huy L. Nguyễn, and David P. Woodruff. On deterministic sketching and streaming for sparse recovery and norm estimation. *Linear Algebra and its Applications*, 441:152–167, 2014.

34 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018.

35 Hung Q. Ngo, Ely Porat, and Atri Rudra. Efficiently decodable error-correcting list disjunct matrices and applications. In *International Colloquium on Automata, Languages, and Programming (ICALP 2011)*, pages 557–568. Springer, 2011.

36 Hung Q. Ngo, Ely Porat, and Atri Rudra. Efficiently decodable compressed sensing by list-recoverable codes and recursion. In *29th Annual Symposium on Theoretical Aspects of Computer Science (STACS 2012)*, volume 14, pages 230–241. LIPIcs, 2012.

37 Eric Price and Jonathan Scarlett. A fast binary splitting approach to non-adaptive group testing. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

38 Ran Raz and Omer Reingold. On recycling the randomness of states in space bounded computation. In *31st Annual Symposium on Theory of Computing (STOC 1999)*, pages 159–168, 1999.

39 Michael Sipser and Daniel A. Spielman. Expander codes. *IEEE Transactions on Information Theory*, 42(6):1710–1722, 1996.

40 Amnon Ta-Shma and David Zuckerman. Extractor codes. *IEEE Transactions on Information Theory*, 50(12):3015–3025, 2004.

41 Luca Trevisan. Extractors and pseudorandom generators. *Journal of the ACM (JACM)*, 48(4):860–879, 2001.

42 Gillés Zémor. On expander codes. *IEEE Transactions on Information Theory*, 47(2):835–837, 2001.