DOI:10.1145/3339185

Heavy Hitters via Cluster-Preserving Clustering

By Kasper Green Larsen,* Jelani Nelson,† Huy L. Nguyễn,‡ and Mikkel Thorup§

Abstract

We develop a new algorithm for the turnstile heavy hitters problem in general turnstile streams, the ExpanderSketch, which finds the approximate top-k items in a universe of size n using the same asymptotic $O(k \log n)$ words of memory and $O(\log n)$ update time as the CountMin and CountSketch, but requiring only $O(k\ poly(\log n))$ time to answer queries instead of the $O(n \log n)$ time of the other two. The notion of "approximation" is the same ℓ_2 sense as the CountSketch, which given known lower bounds is the strongest guarantee one can achieve in sublinear memory.

Our main innovation is an efficient reduction from the heavy hitters problem to a clustering problem in which each heavy hitter is encoded as some form of noisy spectral cluster in a graph, and the goal is to identify every cluster. Since every heavy hitter must be found, correctness requires that every cluster be found. We thus need a "cluster-preserving clustering" algorithm that partitions the graph into pieces while finding every cluster. To do this we first apply standard spectral graph partitioning, and then we use some novel local search techniques to modify the cuts obtained so as to make sure that the original clusters are sufficiently preserved. Our clustering algorithm may be of broader interest beyond heavy hitters and streaming algorithms.

1. INTRODUCTION

Finding "frequent" or "top-k" items in a dataset is a common task in data mining. In the data streaming literature, this problem is typically referred to as the *heavy hitters problem*, which is as follows: a frequency vector $x \in \mathbb{R}^n$ is initialized to the zero vector, and we process a stream of updates `update`(i, Δ) for $\Delta \in \mathbb{R}$, with each such update causing the change $x_i \leftarrow x_i + \Delta$. The goal is to identify coordinates in x with large weight (in absolute value) while using limited memory. For example, i may index distinct Web surfers, and x_i could denote the number of times person i clicked a Web ad on a particular site; Metwally et al.[12] gave an application of then finding frequent ad clickers in Web advertising. Or in networking, $n = 2^{32}$ may denote the number of source IP addresses in IPv4, and x_i could be the number of packets sent by i on some link. One then may want to find sources with high link utilization in a network traffic monitoring application. In both these cases $\Delta = 1$ in every update. Situations with negative Δ may also arise; for example one may want to notice *changes* in trends. Imagine n is the number of words in some lexicon, and a search engine witnesses a stream of queries. One may spot trend shifts by identifying words that had a large change in frequency of being searched across two distinct time periods T_1 and T_2. If one processes all words in T_1 as $\Delta = +1$ updates and those in T_2 as $\Delta = -1$ updates, then x_i will equal the difference in frequency of queries for word i across these two time periods. Thus, finding the top k heavy indices in x could find newly trending words (or words that once were trending but no longer are).

Returning to technical definitions, we define item weights $w_i := f(x_i)$ based on a weighting function f: $(-\infty, \infty) \to [0, \infty)$. We then define W to be the sum of all the weights that is $W := \sum_{i=1}^{n} w_i$. Given some parameter k, a *k-heavy hitter* under this weighting function is an item i such that $w_i > W/k$. It is clear that k is an upper bound on the number of k-heavy hitters, and thus an algorithm that finds them all is solving some form of approximate top-k problem (it is approximate since we only require finding top-k items that are heavy enough, with respect to the weighting function under consideration). We will in fact study a harder problem known as the *tail heavy hitters problem*, for which we say an item is a *k-tail heavy hitter* if $w_i > W_{[\bar{k}]}/k$. Here $W_{[\bar{k}]}$ is the sum of all weights except for the top k. Note that the number of tail heavy hitters must be less than $2k$ (namely the actual top k, plus the fewer than k items in the tail which may satisfy $w_i > W_{[\bar{k}]}/k$). One specific goal for the algorithm then, which we require in this paper, is to at query time output a list $L \subset \{1, \ldots, n\}$ such that (1) L contains every k-tail heavy hitter, and (2) $|L| = O(k)$. All the algorithms discussed here can also be slightly modified to provide the guarantee that every item in L is at least a $2k$-tail heavy hitter, so that false positives in L are guaranteed to still be somewhat heavy.

The earliest literature on the heavy hitters problem focused on the case of *insertion-only streams*, in which case `update` (i, Δ) has $\Delta = 1$ for every update (in contrast with the *general turnstile model*, which allows arbitrary $\Delta \in \mathbb{R}$). This corresponds to seeing a stream of indices $i_1 i_2 \ldots i_m$ and wanting to find those items which occur frequently. The perhaps most well-studied form of the problem in insertion-only streams is $f(x_i) = x_i$. A simple solution then is sampling: if the stream is $i_1 i_2 \ldots i_m$, then sample t uniformly random indices $j_1, \ldots, j_t$ to create a new sampled stream $i_{j_1} \ldots i_{j_t}$. A straightforward analysis based on the

The preliminary version of this paper was published in IEEE FOCS 2016.

* Supported by Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation, grant DNRF84, a Villum Young Investigator Grant and an AUFF Starting Grant.

† Supported by NSF grant IIS-1447471 and CAREER award CCF-1350670, ONR Young Investigator award N00014-15-1-2388 and DORECG award N00014-17-1-2127, an Alfred P. Sloan Research Fellowship, and a Google Faculty Research Award.

‡ Supported by NSF CAREER award CCF-1750716.

§ Supported by his Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research and by his Investigator Grant 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM Foundation.

Chernoff-Hoeffding bound shows if one picks $t = \Omega(k^2 \log k)$ then with large probability one can solve (the non-tail version) of the problem by returning the top $O(k)$ frequency items in the sampled stream. A more clever algorithm from 1982 known as Frequent[13] though solves the problem for the same identity weighting function using only $O(k)$ words of memory, which is optimal. One then may wonder: what then is the motivation for considering other weighting functions f?

We now observe the following: consider the weighting function $f(x_i) = |x_i|^p$ for some fixed $p > 1$. The usual jargon for such f refers to the resulting problem as ℓ_p *heavy hitters* (or ℓ_p *tail heavy hitters* when one considers the tail version). If one assumes that item frequencies are distinct, or simply that if several items are tied for the kth largest then none of them are considered as being in the top k, then as $p \to \infty$ the top k items are all guaranteed to be k-tail heavy hitters. This implies that for large p a correct tail heavy hitter algorithm is required to not miss any item in the top k, which is a more stringent requirement and thus a harder problem to solve. To see this, suppose the kth item has frequency x_i and the $(k+1)$st has frequency $x_{i'} < x_i$.

Then we want that

$$w_i = x_i^p > \frac{n-k}{k} \cdot x_{i'}^p \geq W_{[k]} / k.$$

This inequality indeed holds for $p \to \infty$; specifically it suffices that $p > \log((n-k)/k) / \log(x_i / x_{i'})$. One then may expect that, due to larger p forcing a "more exact" solution to the top-k problem, solving ℓ_p tail heavy hitters should be harder as p grows. This is in fact the case: any solution to ℓ_p heavy hitters (even the non-tail version) requires $\Omega(n^{1-2/p})$ bits of memory.[2] It is furthermore also known, via a formal reduction, that solving larger p *is* strictly harder,[9] in that for $p > q$ a solution to ℓ_p tail heavy hitters in space S leads to a solution for ℓ_q in space $O(S)$, up to changing the parameter k by at most a factor of two. In light of these two results, for tail heavy hitters solving the case $p = 2$ is the best one could hope for while using memory that is subpolynomial in n. A solution to the case $p = 2$ was first given by the CountSketch, which uses $O(k \log n)$ words of memory. Most recently the BPTree was given using only $O(k \log k)$ words of memory,[3] improving the preceding CountSieve data structure using $O(k \log k \log \log n)$ words.[4]

Thus far we have described previous results in the insertion-only model, and we now turn back to the turnstile model. Considering both positive and negative Δ is important when one does not necessarily want the top k items in a single stream, but perhaps the top k items in terms of frequency *change* across two different streams (e.g., two time windows). For example, we may have two streams of query words to a search engine across two time windows and want to know which words had their frequencies change the most. If one interprets stream items in the first window as $\Delta = +1$ updates and those in the second window as corresponding to $\Delta = -1$ updates, then x_i for any word i will be the difference between the frequencies across the two windows, so that we are now attempting to solve an approximate top-k problem in this vector of frequency differences.

The complexity of the heavy hitters problem varies dramatically when one moves from the insertion-only model to the turnstile model. For example, it is clear that the first-mentioned algorithm of sampling the stream no longer works (consider for the example the stream which does `update(1, +1)` N times followed by `update(1, –1)` N times for large N, followed by `update(2, 1)`; only 2 is a heavy hitter, but a sampling algorithm will likely not notice). It turns out that the Frequent, BPTree, and CountSieve data structures also do not have versions that can operate in the turnstile model; in fact a lower bound of the work of Jowhari et al. [9] shows that any solution to ℓ_p heavy hitters for $p \geq 1$ in the turnstile model requires $\Omega(k \log n)$ words of memory, which is more than what is required by any of these three data structures. The CountSketch, however, does function correctly even in the turnstile model, and is asymptotically memory-optimal in this model due to the lower bound of the work of Jowhari et al.[9] Another popular algorithm in the turn-stile model, for the easier ℓ_1 tail heavy hitters problem, is the CountMin sketch, also using $O(k \log n)$ words of memory. The CountMin sketch though has the advantage that in the so-called *strict* turnstile model, when we are promised that at all times in the stream $x_i \geq 0$ for all i, the constants that appear in its space complexity are smaller than that of the CountSketch.

Given that the space complexity of the heavy hitters problem is resolved up to a constant factor in the turnstile model, what room for improvement is left? In fact, the quality of a data structure is not measured only along the axis of space complexity, but also update time, query time, and failure probability (all the algorithms mentioned thus far, even in the insertion-only model but with the exception of Frequent, are randomized and have a bounded failure probability of incorrectly answering a query). Using $k \log n$ words of memory the CountMin sketch and CountSketch each have $O(\log n)$ update time and failure probability $1/poly(n)$, which are both the best known, but they suffer from a query time of $\Theta(n \log n)$. That is, even though the final output list L only has size $O(k)$, the time to construct it is slightly superlinear in the size of the universe the items are drawn from. For $p = 1$ and only in the strict turnstile model, this was remedied by the so-called "dyadic trick" as shown in the work of Cormode and Muthukrishnan[7] and "hierarchical CountSketch" as shown in the work of Cormode and Hadjieleftheriou,[6] which each could trade off space and update time for improved query time; see Figure 1 for detailed bounds. None of these solutions though were able to simultaneously achieve the best of all worlds, namely (1) working in the general turnstile model, (2) achieving the ℓ_2 guarantee, (3) achieving the $O(k \log n)$ space and $O(\log n)$ update time of the CountSketch, and (4) achieving the $k\,poly(\log n)$ query time of the dyadic trick. That is, none were able to do so until the ExpanderSketch.

1.1. Our main contribution

We give the first data structure, the ExpanderSketch, for general turnstile ℓ_2 tail heavy hitters using optimal $O(k \log n)$ words of memory with $1/poly(n)$ failure probability, fast $O(\log n)$ update time, and fast $k\,poly(\log n)$ query time.

2. THE EXPANDERSKETCH

We here provide an overview of our algorithm, ExpanderSketch. Henceforth to avoid repetitiveness, we will often drop the "tail" in "tail heavy hitters"; all heavy hitters problems we consider henceforth are the tail version. Our first

Figure 1. Previous results for the turnstile ℓ_p heavy hitters problem, stated for failure probability $1/poly(n)$. The "general" column states whether the algorithm works in the general turnstile model (as opposed to only strict turnstile). Memory consumption is stated in machine words. The Hierarchical CountSketch query time could be made $k n^{\gamma}$ for arbitrarily small constant $\gamma > 0$. The constant c in the $\log^c n$ term in the ExpanderSketch query time can be made $3 + o(1)$ and most likely even $2 + o(1)$ using more sophisticated graph clustering subroutines, and in the strict turnstile model it can even be made $1 + o(1)$; see full paper.

Data structure	Memory	Update time	Query time	Guarantee	General?
CountMin[7]	$k \log n$	$\log n$	$n \log n$	ℓ_1	Y
CountMin with dyadic trick[7]	$k \log^2 n$	$\log^2 n$	$k \log^2 n$	ℓ_1	N
Hierarchical CountSketch[6]	$k \log n$	$\log n$	$k \cdot n^{.01}$	ℓ_1	N
CountSketch[5]	$k \log n$	$\log n$	$n \log n$	ℓ_2	Y
ExpanderSketch (this work)	$k \log n$	$\log n$	$k \log^c n$	ℓ_2	Y

step is a reduction from arbitrary k to several heavy hitters problems for which k is "small". In this reduction, we initialize data structures (to be designed) $D_1, \ldots, D_q$ for the k'-heavy hitters problem for $q = \max\{1, \Theta(k/\log n)\}$ and pick a hash function $h: [n] \to [q]$. When we see an update (i, Δ) in the stream, we feed that update only to $D_{h(i)}$. We can show that after the reduction, whp (i.e., with probability $1 - 1/poly(n)$) we are guaranteed each of the k-heavy hitters i in the original stream now appears in some substream updating a vector $x' \in \mathbb{R}^n$, and i is a k'-heavy hitter in x' for $k' = C \log n$. One then finds all the k'-heavy hitters in each substream then outputs their union as the final query result. For the remainder of this overview we thus focus on solving k-heavy hitters for $k < C \log n$, that is how to implement one of the D_j, so there are at most $O(\log n)$ heavy hitters. Our goal is to achieve failure probability $1/poly(n)$ with space $O(k \log n)$, update time $O(\log n)$, and query time $k\, poly(\log n) = poly(\log n)$.

There is an algorithm, the Hierarchical CountSketch (see Figure 1), which is almost ideal. It achieves $O(k \log n)$ space and $O(\log n)$ update time, but the query time is polynomial in the universe size instead of our desired polylogarithmic. Our main idea is to reduce to the universe size to polylogarithmic so that this solution then becomes viable for fast query. We accomplish this reduction while maintaining $O(\log n)$ update time and optimal space by reducing our heavy hitter problem into $m = \Theta(\log n/\log\log n)$ separate *partition heavy hitters problems*, which we now describe.

In the partition k-heavy hitters problem there is some partition $\mathcal{P} = \{S_1, \ldots, S_N\}$ of $[n]$, and it is presented in the form of an oracle $\mathcal{O}: [n] \to [N]$ such that for any $i \in [n]$, $\mathcal{O}(i)$ gives the $j \in [N]$ such that $i \in S_j$. In what follows the partitions will be random, with $\mathcal{O}_j$ depending on some random hash functions. Define a vector $y \in \mathbb{R}^N$ such that for each $j \in [N]$, $y_j = \|x_{S_j}\|_2$, where x_S is the projection of x onto a subset S of coordinates. The goal then is to solve the k-heavy hitters problem on y subject to streaming updates to x: we should output a list $L \subset [N]$, $|L| = O(k)$, containing all the k-heavy hitters of y. Our desired failure probability is $1/poly(N)$.

We remark at this point that the ℓ_1 version of partition heavy hitters in the strict turnstile model is simple to solve (and for ℓ_1 strict turnstile, our algorithm already provides improvements over previous work and is thus delving into). In particular, one can simply use a standard strict turnstile ℓ_1 heavy hitters algorithm for an N-dimensional vector and translate every `update`(i, Δ) to `update`$(\mathcal{O}(i), \Delta)$. This in effect treats y_j as $\sum_{i:\mathcal{O}(i)=j} x_i$, which is exactly $\|x_{\mathcal{O}^{-1}(j)}\|_1$ as desired in the case of strict turnstile streams. For details on the ℓ_2 version in the general turnstile model, we refer to the full version of our work. It suffices to say here that the Hierarchical CountSketch can be modified to solve even the partition heavy hitters problem, with the same space and time complexities as in Figure 1 (but with the n's all replaced with N's).

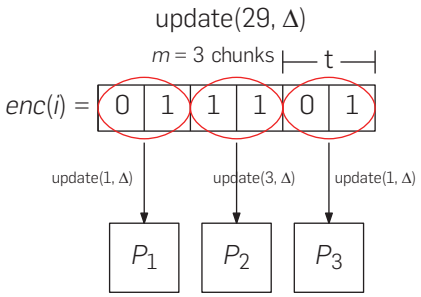
Now we explain how we make use of partition heavy hitters. We take the pedagogical approach of explaining a simple but flawed solution, then iteratively refine to fix flaws until we arrive at a working solution.

Take 1. Recall our overall plan: reduce the universe size so that (the partition heavy hitters version of the) Hierarchical CountSketch has fast query time. For each index $i \in [n]$ we can write i in base b (for some b yet to be determined) so that it has digit expansion $d_{m-1} d_{m-2} \cdots d_0$ with each $d_i \in \{0, \ldots, b-1\}$. Our algorithm instantiates m separate partition heavy hitter data structures $P_0, \ldots, P_{m-1}$ each with $N = b$ and where P_j has oracle $\mathcal{O}_j$ with $\mathcal{O}_j(i)$ mapping i to the jth digit in its base-b expansion (See Figure 2). If we choose $b = poly(\log n)$ (which we will do), then our query time per P_j is a fast $k \cdot b^{\gamma} = poly(\log n)$ (see Figure 1). Suppose for a moment that there was only one heavy hitter i and that, by a stroke of luck, none of the P_j's fail. Then since i is heavy, $\mathcal{O}_j(i)$ must be heavy from the perspective of P_j for each j (since $\mathcal{O}_j(i)$ receives all the mass from x_i, plus potentially even *more* mass from indices that have the same jth digit in base-b). Recovering i would then be simple: we query each P_j to obtain d_j, then we concatenate the digits d_j to obtain i.

There are of course two main problems: first, each P_j actually outputs a list L_j which can have up to $\Theta(\log n)$ "heavy digits" and not just 1, so it is not clear which digits to concatenate with which across the different j. The second problem is that the P_j's are randomized data structures that fail with probability $1/poly(b) = 1/poly(\log n)$, so even if we knew which digits to concatenate with which, some of those digits are likely to be wrong.

Take 2. We continue from where we left off in the previous take. The second issue, that some digits are likely to be wrong, is easy to fix. Specifically, for $b = poly(\log n)$ we have $m = \log_b n = O(\log n/\log\log n)$. For this setting of b, m, using that the failure probability of each P_j is $1/poly(b)$, a simple calculation shows that whp $1 - 1/poly(n)$, at most a small constant fraction of the P_j fail. Thus, for example, at most 1% of the digits d_j are wrong. This is then easy to fix: we do not write i in base b but rather write enc(i) in base b, where enc(i) is the encoding of i (treated as a $\log n$ bit string) into $T = O(\log n)$ bits by an error-correcting code with constant rate that can correct an $\Omega(1)$-fraction of errors. Such codes exist

Figure 2. Simplified version of final data structure. The update is $x_{29} \leftarrow x_{29} + \Delta$ with $m = 3$, $t = 2$ in this example. Each P_j is a b-tree operating on a partition of size 2^t.

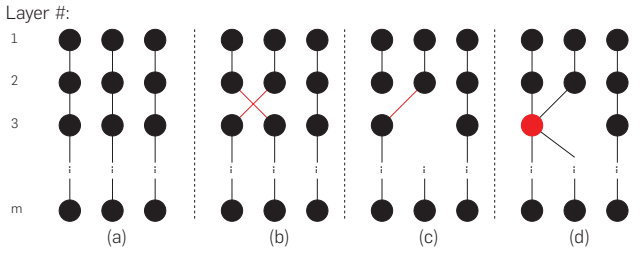


with linear time encoding and decoding.[16] Then even if we recover enc(i) with 1% of the digits being incorrect, we can decode to recover i exactly.

We now return to the first issue, which is the main complication: in general, there may be more than one heavy hitter (there may be up to $\Theta(\log n)$ of them). Thus, even if we performed wishful thinking and pretended that every P_j succeeded, and furthermore that every L_j contained exactly the jth digits of the encodings of heavy hitters and nothing else, it is not clear how to perform the concatenation. For example, if there are two heavy hitters with encoded indices 1100 and 0110 in binary that we then write in, say, base 4 (as 30 and 12), suppose the P_j correctly return $L_1 = \{3, 1\}$ and $L_2 = \{0, 2\}$. How would we then know which digits matched with which for concatenation? That is, are the heavy hitter encodings 30 and 12, or are they 32 and 10? Brute force trying all possibilities is too slow, since $m = \Theta(\log n/\log\log n)$ and each $|L_j|$ could be as big as $\Theta(\log n)$, yielding $(C \log n)^m = poly(n)$ possibilities. In fact this question is quite related to the problem of *list-recoverable codes*, but since no explicit codes are known to exist with the efficiency guarantees we desire, we proceed in a different direction.

To aid us in knowing which chunks to concatenate with which across the L_j, the attempt we describe now (which also does not quite work) is as follows. Define m pairwise independent hash functions $h_1, \ldots, h_m: [n] \to [poly(\log n)]$. Since there are $O(\log n)$ heavy hitters, any given h_j perfectly hashes them with decent probability. Now rather than partitioning according to $\mathcal{O}_j(i) = \text{enc}(i)_j$ (the jth digit of enc(i)), we imagine setting $\mathcal{O}_j(i) = h_j(i) \circ \$ \circ \text{enc}(i)_j \circ \$ \circ h_{j+1}(i)$ where $\circ$ denotes concatenation. Define an index $j \in [m]$ to be *good* if (a) P_j succeeds, (b) h_j perfectly hashes all heavy hitters $i \in [n]$, and (c) for each heavy hitter i, the total ℓ_2 weight from non-heavy hitters hashing to $h_j(i)$ is $o\left(\left(1/\sqrt{\log n}\right)\|x_{[\tilde{k}]}\|_2\right)$. A simple argument shows that whp a $1 - \epsilon$ fraction of the $j \in [m]$ are good, where ϵ can be made an arbitrarily small positive constant. Now let us perform some wishful thinking: if *all* $j \in [m]$ are good, and furthermore no non-heavy elements appear in L_j with the same h_j but different h_{j+1} evaluation as an actual heavy hitter, then the indices in L_j tell us which chunks to concatenate within L_{j+1}, so we can concatenate, decode, then be done. Unfortunately a small constant fraction of the $j \in [m]$ are not good, which prevents this scheme from working (see Figure 3). Indeed, in order to succeed in a query, for each heavy hitter we must correctly identify a large connected component of the vertices corresponding to that heavy hitter's path — that would correspond to containing a large fraction of the digits of the encoding, which would allow for decoding. Unfortunately, paths are not robust in having large connected component subgraphs remaining even for $O(1)$ bad levels.

Figure 3. Each vertex in row j corresponds to an element of L_j, that is the heavy hitter chunks out-put by P_j. When indices in $\mathcal{P}_j$ are partitioned by $h_j(i) \circ \text{enc}(i)_j \circ h_{j+1}(i)$, we connect chunks along paths. Case (a) is the ideal case, when all j are good. In (b) P_2 failed, producing a wrong output that triggered incorrect edge insertions. In (c) both P_2 and P_3 failed, triggering an incorrect edge and a missing vertex, respectively. In (d) two heavy hitters collided under h_3, causing their vertices to have the same name thereby giving the appearance of a merged vertex. Alternatively, light items masking as a heavy hitter might have appeared in L_3 with the same h_3 evaluation as a heavy hitter but different h_4 evaluation, causing the red vertex to have two outgoing edges to level 4.



Take 3. The above consideration motivates our final scheme, which uses an expander-based idea first proposed in the work of Gilbert et al.[8] in the context of "for all" ℓ_1/ℓ_1 sparse recovery, a problem in compressed sensing. Although our precise motivation for the next step is slightly different than in the work of Gilbert et al.,[8] and our query algorithm and our definition of "robustness" for a graph will be completely different, the idea of connecting chunks using expander graphs is very similar to an ingredient in that work. The idea is to replace the path in the last paragraph by a graph which is robust to a small fraction of edge insertions and deletions, still allowing the identification of a large connected component in such scenarios. Expander graphs will allow us to accomplish this. For us, "robust" will mean that over the randomness in our algorithm, whp each corrupted expander is still a spectral cluster (to be defined shortly). For,[8] robustness meant that each corrupted expander still contains an induced small-diameter subgraph (in fact an expander) on a small but constant fraction of the vertices, which allowed them a recovery procedure based on a shallow breadth-first search. They then feed the output of this breadth-first search into a recovery algorithm for an existing list-recoverable code (namely Parvaresh-Vardy codes). Due to suboptimality of known list-recoverable codes, such an approach would not allow us to obtain our optimal results.

2.1. An expander-based approach

Let F be an arbitrary D-regular connected graph on the vertex set $[m]$ for some $D = O(1)$. For $j \in [m]$, let $\Gamma(j) \subset [m]$ be the set of neighbors of vertex j. We partition $[n]$ according to $\mathcal{O}_j(i) = z(i)_j = h_j(i) \circ \$ \circ \text{enc}(i)_j \circ \$ \circ h_{\Gamma(j)_1} \circ \$ \cdots \$ \circ h_{\Gamma(j)_D}$ where

$\Gamma(j)_k$ is the kth neighbor of j in F. Given some such z, we say its *name* is the first $s = O(\log \log n)$ bits comprising the h_j portion of the concatenation. Now, we can imagine a graph G on the layered vertex set $V = [m] \times [2^s]$ with m layers. If L_j is the output of a heavy hitter query on P_j, we can view each element z of L_j as suggesting D edges to add to G, where each such z connects D vertices in various layers of V to the vertex in layer j corresponding to the name of z. The way we actually insert the edges is as follows. First, for each $j \in [m]$ we instantiate a *partition point query structure* Q_j with the same oracle as the P_j; this is a data structure which, given any $z \in [N]$, outputs a low-error estimate $\tilde{y}_z$ of y_z with failure probability $1/poly(N)$. We modify the definition of a level $j \in [m]$ being "good" earlier to say that Q_j must also succeed on queries to every $z \in L_j$. We point query every partition $z \in L_j$ to obtain an estimate $\tilde{y}_z$ approximating y_z. We then group all $z \in L_j$ by name, and within each group we remove all z from L_j except for the one with the largest $\tilde{y}_z$, breaking ties arbitrarily. This filtering step guarantees that the vertices in layer j have unique names, and furthermore, when j is good all vertices corresponding to heavy hitters appear in L_j and none of them are thrown out by this filtering. We then let G be the graph created by including the at most $(D/2)\cdot\sum_j |L_j|$ edges suggested by the z's across all L_j (we only include an edge if both endpoints suggest it). Note G will have many isolated vertices since only $m \cdot \max_j |L_j| = O(\log^2 n/\log \log n)$ edges are added, but the number of vertices in each layer is 2^s, which may be a large power of $\log n$. We let G be its restriction to the union of non-isolated vertices and vertices whose names match the hash value of a heavy hitter at the m different levels. This ensures G has $O(\log^2 n/\log \log n)$ vertices and edges. We call this G the *chunk graph*.

Now, the intuitive picture is that G *should* be the vertex-disjoint union of several copies of the expander F, one for each heavy hitter, plus other junk edges and vertices coming from other non-heavy hitters in the L_j. Due to certain bad levels j however, some expanders might be missing a small constant ϵ-fraction of their edges, and also the ϵm bad levels may cause spurious edges to connect these expanders to the rest of the graph. The key insight is as follows. Let W be the vertices of G corresponding to some particular heavy hitter, so that in the ideal case W would be a single connected component whose induced graph is F. What we can prove, even with ϵm bad levels, is that every heavy hitter's such vertices W forms an $O(\epsilon)$-*spectral cluster*.

Definition 1. An ϵ-*spectral cluster* in an undirected graph $G = (V, E)$ is a vertex set $W \subseteq V$ of any size satisfying the following two conditions: First, only an ϵ-fraction of the edges incident to W leave W, that is, $|\partial(W)| \leq \epsilon\, vol(W)$, where $vol(W)$ is the sum of edge degrees of vertices inside W. Second, given any subset A of W, let $r = vol(A)/vol(W)$ and $B = W\backslash A$. Then

$$\left|E\left(A,B\right)\right| \geq \left(r\left(1-r\right)-\epsilon\right) vol\left(W\right).$$

Note $r(1–r)\, vol(W)$ is the number of edges one would expect to see between A and B had W been a random graph with a prescribed degree distribution.

Roughly, the above means that (a) the cut separating W from the rest of G has $O(\epsilon)$ conductance (i.e., it is a very sparse cut), and (b) for any cut $(A, W\backslash A)$ within W, the number of edges crossing the cut is what is guaranteed from a spectral expander, minus $O(\epsilon) \cdot vol(W)$. Our task then reduces to finding all ϵ-spectral clusters in a given graph. We devise a scheme CutGrabClose that for each such cluster W, we are able to find a $(1 - O(\epsilon))$-fraction of its volume with at most $O(\epsilon)\cdot vol(W)$ erroneous volume from outside W. This suffices for decoding for ϵ a sufficiently small constant, since this means we find most vertices, that is chunks of the encoding, of the heavy hitter.

For the special case of ℓ_1 heavy hitters in the strict turnstile model, we are able to devise a much simpler query algorithm that works; see the full paper for details. For this special case we also in the full paper devise a space-optimal algorithm with $O(\log n)$ update time, whp success, and *expected* query time $O(k \log n)$ (though unfortunately the variance of the query time may be quite high).

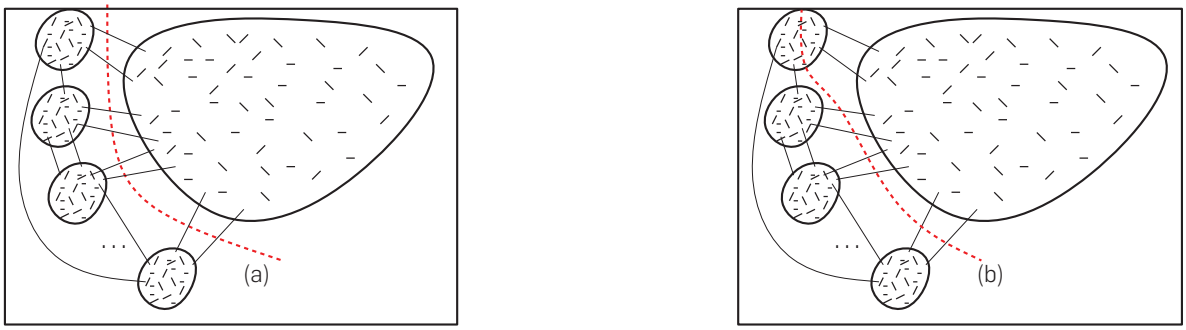
2.2. Cluster-preserving clustering

As mentioned, an ϵ-spectral cluster is a subset W of the vertices in a graph $G = (V, E)$ such that (1) $|\partial W| \leq \epsilon\, vol(W)$, and (2) for any $A \subsetneq W$ with $vol(A)/vol(W) = r$, $|E(A, W\backslash A)| \geq (r(1 - r) - \epsilon)\, vol(W)$. Item (2) means the number of edges crossing a cut within W is what you would expect from a random graph, up to $\epsilon\, vol(W)$. Our goal is to, given G, find a partition of V such that every ϵ-spectral cluster W in G matches some set of the partition up to $\epsilon\, vol(W)$ symmetric difference.

Our algorithm CutGrabClose is somewhat similar to the spectral clustering algorithm of (Kannan et al.,[10] Section 4), but with local search. That algorithm is simple: find a low-conductance cut (e.g., a Fiedler cut) to split G into two pieces, then recurse on both pieces. Details aside, Fiedler cuts are guaranteed by Cheeger's inequality to find a cut of conductance $O(\sqrt{\gamma})$ as long as a cut of conductance at most γ exists in the graph. The problem with this basic recursive approach is shown in Figure 4 (in particular cut (b)). Note that a cluster can be completely segmented after a few levels of recursion, so that a large portion of the cluster is never found.

Our approach is as follows. Like the above, we find a low-conductance cut then recurse on both sides. However, before recursing on both sides we make certain "improvements" to the cut. We say $A \subset V$ is *closed* in G if there is no vertex $v \in G\backslash A$ with at least 5/9ths of its neighbors in A. Our algorithm maintains that all recursive subcalls are to closed subsets in G as follows. Suppose we are calling CutGrabClose on some set A which we inductively know is closed in G. We first try to find a low-conductance cut within A. If we do not find one, we terminate and let A be one of the sets in the partition. Otherwise, if we cut A into $(S, \overline{S})$, then we close both S, $\overline{S}$ by finding vertices violating closure and simply moving them. It can be shown that if the $(S, \overline{S})$ cut had sufficiently low conductance, then these local moves can only improve conductance further. Now both S and $\overline{S}$ are closed in A (which by a transitivity lemma we show, implies they are closed in G as well). We then show that if (1) some set S is closed, and (2) S has much more than half the volume of some spectral cluster W (e.g., a 2/3rds fraction), then in fact S contains a $(1 - O(\epsilon))$-fraction of W. Thus after closing both S, $\overline{S}$, we have that S either: (a) has almost none of W, (b) has

Figure 4. Each small oval is a spectral cluster. They are well-connected internally, with sparse cuts to the outside. The large oval is the rest of the graph, which can look like anything. Cut (a) represents a good low-conductance cut, which makes much progress (cutting the graph in roughly half) while not losing any mass from any cluster. Cut (b) is also a low-conductance cut as long as the number of small ovals is large, since then cutting one cluster in half has negligible effect on the cut's conductance. However, (b) is problematic since recursing on both sides loses half of one cluster forever.



almost all of W, or (c) has roughly half of W (between 1/3 and 2/3, say). To fix the latter case, we then "grab" all vertices in $\overline{S}$ with some $\Omega(1)$-fraction, for example 1/6th, of their neighbors in S and simply move them all to S. Doing this some constant number of times implies S has much more than 2/3rds of W (and if S was in case (a), then we show it still has almost none of W). Then by doing another round of closure moves, one can ensure that both S, $\overline{S}$ are closed, and each of them has either an $O(\epsilon)$-fraction of W or a $(1 - O(\epsilon))$-fraction. It is worth noting that our algorithm can make use of *any* spectral cutting algorithm as a black box and not just Fiedler cuts, followed by our grab and closure steps. For example, algorithms from[14, 15] run in nearly linear time and either (1) report that no γ-conductance cut exists (in which case we could terminate), (2) find a *balanced* cut of conductance $O(\sqrt{\gamma})$ (where both sides have nearly equal volume), or (3) find an $O(\sqrt{\gamma})$ -conductance cut in which every $W \subset G$ with $vol(W) \leq (1/2)\ vol(G)$ and $\phi(W) \leq O(\gamma)$ has more than half its volume on the smaller side of the cut. Item (2), if it always occurred, would give a divide-and-conquer recurrence to yield nearly linear time for finding all clusters. It turns out item (3) though is even better! If the small side of the cut has half of every cluster W, then by grabs and closure moves we could ensure it is still small and has almost all of W, so we could recurse just on the smaller side.

As a result we end up completing the cluster preserving clustering in polynomial time in the graph size, which is polynomial in $poly(\log n)$, and this allows us to find the k heavy hitters with whp in $O(k\ poly(\log n))$ time. For a full description of the algorithm, the reader is referred to reference Larsen et al.[11]

Acknowledgments

We thank Noga Alon for pointing us to (Alon and Chung,[1] Lemma 2.3), Piotr Indyk for the reference,[8] Yi Li for answering several questions about,[8] Mary Wootters for making us aware of the formulation of the list-recovery problem in coding theory and its appearance in prior work in compressed sensing and group testing, and Fan Chung Graham and Olivia Simpson for useful conversations about graph partitioning algorithms.

References

1. Alon, N., Chung, F.R.K. Explicit construction of linear sized tolerant networks. *Discrete Math.* 72 (1988), 15–19.
2. Bar-Yossef, Z., Jayram, T.S., Kumar, R., Sivakumar, D. An information statistics approach to data stream and communication complexity. *J. Comput. Syst. Sci.* 68, 4 (2004), 702–732.
3. Braverman, V., Chestnut, S.R., Ivkin, N., Nelson, J., Wang, Z., Woodruff, D.P. BPTree: An ℓ_2 heavy hitters algorithm using constant memory. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)* (2017), ACM, Chicago, IL, 361–376.
4. Braverman, V., Chestnut, S.R., Ivkin, N., Woodruff, D.P. Beating CountSketch for heavy hitters in insertion streams. In *Proceedings of the 48th STOC* (2016), ACM, Cambridge, MA.
5. Charikar, M., Chen, K., Farach-Colton, M. Finding frequent items in data streams. *Theor. Comput. Sci.* 312, 1 (2004), 3–15.
6. Cormode, G., Hadjieleftheriou, M. Finding frequent items in data streams. *PVLDB* 1, 2 (2008), 1530–1541.
7. Cormode, G., Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* 55, 1 (2005), 58–75.
8. Gilbert, A.C., Li, Y., Porat, E., Strauss, M.J. For-all sparse recovery in near-optimal time. In *Proceedings of the 41st ICALP* (2014), Springer, Copenhagen, Denmark, 538–550.
9. Jowhari, H., Saglam, M., Tardos, G. Tight bounds for Lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the 30th PODS* (2011), ACM, Athens, Greece, 49–58.
10. Kannan, R., Vempala, S., Vetta, A. On clusterings: Good, bad and spectral. *J. ACM* 51, 3 (2004), 497–515.
11. Larsen, K.G., Nelson, J., Nguyễn, H.L., Thorup, M. Heavy hitters via cluster-preserving clustering. *CoRR*, abs/1511.01111 (2016).
12. Metwally, A., Agrawal, D., El Abbadi, A. Efficient computation of frequent and top-k elements in data streams, In *Proceedings of the 10th ICDT* (2005), Springer, Edinburgh, UK, 398–412.
13. Misra, J., Gries, D. Finding repeated elements. *Sci. Comput. Program* 2, 2 (1982), 143–152.
14. Orecchia, L., Sachdeva, S., Vishnoi, N.K. Approximating the exponential, the Lanczos method and an $\tilde{O}(m)$-time spectral algorithm for balanced separator. In *Proceedings of the 44th STOC* (2012), 1141–1160.
15. Orecchia, L., Vishnoi, N.K. Towards an SDP-based approach to spectral methods: A nearly-linear-time algorithm for graph partitioning and decomposition. In *Proceedings of the 22nd SODA* (2011), SIAM, San Francisco, CA, 532–545.
16. Spielman, D.A. Linear-time encodable and decodable error-correcting codes. *IEEE Trans. Information Theory* 42, 6 (1996), 1723–1731.

Kasper Green Larsen (larsen@cs.au.dk), Aarhus University, Aarhus, Denmark.

Jelani Nelson (minilek@seas.harvard.edu), Harvard University, Cambridge, MA, USA.

Huy L. Nguyễn (hu.nguyen@northeastern.edu), Northeastern University, Boston, MA, USA.

Mikkel Thorup (mthorup@di.ku.dk), University of Copenhagen, Denmark.