Near-optimal Hypergraph Sparsification in Insertion-only and Bounded-deletion Streams

Sanjeev Khanna* Aaron (Louie) Putterman† Madhu Sudan‡

September 14, 2025

Abstract

We study the problem of constructing hypergraph cut sparsifiers in the streaming model where a hypergraph on n vertices is revealed either via an arbitrary sequence of hyperedge insertions alone (*insertion-only* streaming model) or via an arbitrary sequence of hyperedge insertions and deletions (*dynamic* streaming model). For any $\epsilon \in (0,1)$, a $(1 \pm \epsilon)$ hypergraph cut-sparsifier of a hypergraph H is a reweighted subgraph H' whose cut values approximate those of H to within a $(1 \pm \epsilon)$ factor. Prior work shows that in the static setting, one can construct a $(1\pm\epsilon)$ hypergraph cut-sparsifier using $\tilde{O}(nr/\epsilon^2)$ bits of space [Chen-Khanna-Nagda FOCS 2020], and in the setting of dynamic streams using $\tilde{O}(nr \log m/\epsilon^2)$ bits of space [Khanna-Putterman-Sudan FOCS 2024]; here the $\tilde{O}$ notation hides terms that are polylogarithmic in n, and we use m to denote the total number of hyperedges in the hypergraph. Up until now, the best known space complexity for insertion-only streams has been the same as that for the dynamic streams. This naturally poses the question of understanding the complexity of hypergraph sparsification in insertion-only streams.

Perhaps surprisingly, in this work we show that in *insertion-only* streams, a $(1 \pm \epsilon)$ cut-sparsifier can be computed in $\tilde{O}(nr/\epsilon^2)$ bits of space, *matching the complexity* of the static setting. As a consequence, this also establishes an $\Omega(\log m)$ factor separation between the space complexity of hypergraph cut sparsification in insertion-only streams and dynamic streams, as the latter is provably known to require $\Omega(nr \log m)$ bits of space. To better explain this gap, we then show a more general result: namely, if the stream has at most k hyperedge deletions then $\tilde{O}(nr \log k/\epsilon^2)$ bits of space suffice for hypergraph cut sparsification. Thus the space complexity smoothly interpolates between the insertion-only regime ($k = 0$) and the fully dynamic regime ($k = m$). Our algorithmic results are driven by a key technical insight: once sufficiently many hyperedges have been inserted into the stream (relative to the number of allowed deletions), we can significantly reduce the underlying hypergraph by size by *irrevocably* contracting large subsets of vertices.

Finally, we complement this result with an essentially matching lower bound of $\Omega(nr \log(k/n))$ bits, thus providing essentially a tight characterization of the space complexity for hypergraph cut-sparsification across a spectrum of streaming models.

*School of Engineering and Applied Sciences, University of Pennsylvania, Philadelphia, PA. Email: sanjeev@cis.upenn.edu. Supported in part by NSF award CCF-2402284 and AFOSR award FA9550-25-1-0107.

†School of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts, USA. Supported in part by the Simons Investigator Awards of Madhu Sudan and Salil Vadhan, NSF Award CCF 2152413 and AFOSR award FA9550-25-1-0112. Email: aputterman@g.harvard.edu.

‡School of Engineering and Applied Sciences, Harvard University, Cambridge, Massachusetts, USA. Supported in part by a Simons Investigator Award, NSF Award CCF 2152413 and AFOSR award FA9550-25-1-0112. Email: madhu@cs.harvard.edu.

Contents

1 Introduction

In this work, we continue the line of study on designing *hypergraph cut sparsifiers* in the streaming model.

Recall that a hypergraph, denoted by $H = (V, E)$, is given by a set of hyperedges E where each hyperedge $e \in E$ is an arbitrary subset of the vertices $e \subseteq V$. In the *streaming model*, the hyperedges are revealed in a sequence of steps, where each step consists of either an insertion of a new hyperedge, or the deletion of an existing hyperedge from the hypergraph. At the end of the stream, our goal is to return a *sparsifier* of the resulting hypergraph. Recall that for a hypergraph H, and $\epsilon \in (0, 1)$, a $(1 \pm \epsilon)$ hypergraph cut-sparsifier of H is a *re-weighted sub-hypergraph* $\hat{H}$ such that *for every* cut $S \subseteq V$,

$$\text{cut}_{\hat{H}}(S) \in (1 \pm \epsilon)\text{cut}_H(S).$$

Here, $\text{cut}_H(S)$ denotes the total weight of hyperedges that cross from S to $\bar{S}$ (i.e., the set of hyperedges $\{e \in E : e \cap S \neq \emptyset \wedge e \cap \bar{S} \neq \emptyset\}$). Cuts in hypergraphs capture a variety of applications, ranging from scientific computing on sparse matrices [BDKS16], to clustering and machine learning [YNY+19, ZHS06], and to modeling transistors and other circuitry elements in circuit design [AK95, Law73]. In each of these applications, the ability to sparsify the hypergraph while still preserving cut-sizes is a key building block, as these dramatically decrease the memory footprint (and complexity) of the networks being optimized. In fact, while an arbitrary hypergraph may have as many as 2^n distinct hyperedges, the work of Chen, Khanna, and Nagda [CKN20] showed that $(1 \pm \epsilon)$ hypergraph cut-sparsifiers can be constructed using only $\widetilde{O}(n/\epsilon^2)$ re-weighted hyperedges, a potentially exponential size saving in the bit complexity of the object.

This dramatic reduction in the complexity of the hypergraph while still preserving key properties has also prompted a line of research into the feasibility of sparsifying hypergraphs in the *streaming* model of computation. Here, the hyperedges are presented one at a time, with each hyperedge either being inserted into or deleted from the hypergraph constructed so far. In the streaming model, the goal is to compute some function of the hypergraph (in our case, to construct a *sparsifier* of the hypergraph) after the final update in the stream has arrived, while using as few bits of memory as possible in each intermediate step of processing the stream. As mentioned above, computing sparsifiers of a hypergraph in a stream is valuable as it immediately yields itself to streaming algorithms for any problem that relies *only* on a sparsifier (for instance, computing cut sizes, flows, many clustering objectives, and more), and as such has seen study in several papers [GMT15, CKN20, KPS24b].

Most recently, the work of Khanna, Putterman, and Sudan [KPS24b] studied the space complexity of of hypergraph cut-sparsification in the setting of *dynamic streams* where a hypergraph is revealed via an arbitrary sequence of hyperedge insertions and deletions. They showed an upper bound of $\widetilde{O}(nr\log(m)/\epsilon^2)$[1] bits for computing $(1 \pm \epsilon)$ sparsifiers in dynamic streams, and also established a nearly-matching lower bound of $\Omega(nr\log(m))$ bits, where n is the number of vertices, r is the maximum size of any hyperedge, and m is the number of hyperedges. In particular, because dynamic streaming is a strictly harder setting than insertion-only streams, this also implies an $\widetilde{O}(nr\log(m)/\epsilon^2)$ bit upper bound for the complexity of constructing hypergraph cut-sparsifiers in the insertion-only model.

For comparison, in the static setting (i.e., when the algorithm has unrestricted random-access to the underlying hypergraph), it is known that hypergraph sparsifiers require $\Omega(nr)$ bits to represent [KKTY21b, KKTY21a], and moreover, can be constructed in $\widetilde{O}(nr/\epsilon^2)$ bits of space. Thus, one

[1] Here $\widetilde{O}(\cdot)$ hides polylog$(\cdot)$ factors. Importantly, in hypergraphs m may be as large as 2^n, and thus $\log(m)$ can potentially be as large n. Hence $\widetilde{O}(\cdot)$ does not hide factors of $\log(m)$.

interpretation of the work of [KPS24b] is that hypergraph sparsifiers can be constructed in dynamic streams at the cost of only a $\log(m)$ times increase in the space complexity (ignoring $\text{polylog}(n)$ factors), as compared to the static setting. However, this highlights two natural questions:

1. What is the space complexity of constructing hypergraph sparsifiers in insertion-only streams? Is the complexity essentially same as in the static setting, or can it be as large as the dynamic setting?

2. More generally, how does the space complexity of hypergraph sparsification change as a function of the number of hyperedge deletions? Does it smoothly interpolate between the space complexity needed for insertion-only stream (no deletions) and the dynamic setting (unrestricted number of deletions)?

1.1 Our Contributions

As our first contribution, we provide an answer to the first question above regarding the complexity of constructing sparsifiers in insertion-only streams:

Theorem 1.1. *There is an insertion-only streaming algorithm requiring $\widetilde{O}(nr/\epsilon^2)$ bits of space which creates a $(1\pm\epsilon)$ cut-sparsifier for a hypergraph on n vertices and hyperedges of arity $\leq r$, with probability $1-1/\text{poly}(n)$.*

Specifically, this improves over the prior state of the art algorithms for constructing hypergraph sparsifiers in insertion-only streams by a factor of $\log(m)$ where m denotes an upper bound on the number of hyperedges. This implies that (perhaps surprisingly) the complexity of the insertion-only setting *mirrors* that of the static sparsification setting, and thus there is also a strict separation between the space complexity of the insertion-only and dynamic streaming settings, as the latter is known to require $\Omega(nr\log m)$ bits of space [KPS24b].

Because of this large separation, it is natural to ask if there is some other parameter which governs the space complexity of sparsifying hypergraphs in streams. As our second contribution, we show that this is indeed the case: if we parameterize the dynamic streaming setting by the maximum number of allowed hyperedge *deletions*, then the space complexity smoothly interpolates between the insertion-only setting and the unrestricted dynamic setting:

Theorem 1.2. *For $k\geq 1$, there is a k-bounded deletion streaming algorithm requiring $\widetilde{O}(nr\log(k)/\epsilon^2)$*[2] *bits of space which creates a $(1\pm\epsilon)$ cut-sparsifier for a hypergraph on n vertices and hyperedges of arity $\leq r$, with probability $1-1/\text{poly}(n)$.*

When m is the maximum number of hyperedges in the stream, then the number of deletions is effectively bounded by m. Thus, the dynamic streaming setting is effectively the case when $k=m$, and the above theorem captures the space complexity in this setting. Likewise, as the number of deletions decreases and approaches 0, the setting approaches the insertion-only setting, and the above theorem explains exactly the space savings that are achieved.

Finally, building off of the prior work of Jayaram and Woodruff [JW18] in the bounded-deletion streaming model, we show that this space complexity is essentially *optimal*:

Theorem 1.3. *Any streaming algorithm for k-bounded deletion streams, which for hypergraphs on n vertices, of arity $r\leq n/2+1$, produces a $(1\pm\epsilon)$ cut-sparsifier for $\epsilon<1$, must use $\Omega(nr\log(k/n))$ bits of space.*

[2] Technically, when $k=1$, the term should be $\max(1,\log(k))$.

In summary, this provides a complete picture of the space complexity of producing hypergraph sparsifiers in the streaming setting: as the number of deletions k increases from 0 to m, the space complexity grows by a factor of $\log(k)$ over the space complexity of the static sparsification regime, leading to a smooth phase transition in the space complexity of these algorithms. In the following subsection, we explain more of the techniques that go into these results.

Concurrent Work: In concurrent work, Cohen-Addad, Woodruff, Xie, and Zhou [CAWXZ25] study hypergraph spectral sparsification (a strengthening of cut-sparsification) in the insertion-only regime and likewise show that the factor of $\log(m)$ can be shaved-off.

1.2 Technical Overview

1.2.1 Importance Sampling for Hypergraph Cut Sparsification

To start, let us recap how we create hypergraph sparsifiers in the *static* setting. After an extensive line of works studying hypergraph sparsification [KK15,SY19,CKN20,KKTY21b,KKTY21a,JLS23, Lee23, KPS24a], the work of Quanrud [Qua24] provided the simplest lens through which one can build hypergraph sparsifiers. Roughly speaking, given a hypergraph $H = (V, E)$, each hyperedge $e \in E$ is assigned a value λ_e which is called its *strength*. The strength of a hyperedge is intuitively a measure of the (un)importance of a hyperedge; the smaller the strength of the hyperedge e, this means e is crossing smaller cuts in the hypergraph, and so we are more likely to need to keep e, while if the strength is larger, then there are many other hyperedges which cross the same cuts as e, and thus it is not as necessary for us to keep e. [Qua24] showed that for a specific definition of strength, sampling each hyperedge (independently) at rate roughly $p_e \geq \log(n)/(\epsilon^2\lambda_e)$ (ignoring constant factors), and assigning weight $1/p_e$ to the surviving hyperedges then yields a $(1 \pm \epsilon)$ sparsifier with high probability.

In fact, this procedure lends itself to a simple iterative algorithm for designing sparsifiers: starting with the original hypergraph H, we recover all of the hyperedges in H whose strength is smaller than $\lambda \approx \log(n)/\epsilon^2$, and denote these hyperedges by $T^{(1)}$. Then, it must necessarily be the case that all hyperedges in $H - T^{(1)}$ have *large strength*, and so we can afford to sample these hyperedges at rate $1/2$ (denote this sampled hypergraph by $H^{(1)}$). By the same analysis as above, it turns out that we can show that $T^{(1)} \cup 2 \cdot H^{(1)}$ will be a $(1 \pm \epsilon)$ sparsifier of H with high probability. Now, it remains only to *re-sparsify* $H^{(1)}$, which we can do by repeating the same procedure. Thus, after $\log(m)$ levels of this procedure (where m is the starting number of hyperedges), we can recover a sparsifier of our original hypergraph.

1.2.2 Formal Definitions of Strength

However, to continue our discussion, we will require the formal definition of strength, as well as some auxiliary facts about strength in hypergraphs. The key notion that [Qua24] introduces to measure *strength* in hypergraphs is the notion of *k-cuts* in hypergraphs (and here we adopt the language used by [KPS24b]):

Definition 1.4. For any $k \in [2..n]$, a k-cut in a hypergraph is defined by a k-partition of the vertices, say, $V_1, \ldots V_k$. The *un-normalized size of a k-cut* in an unweighted hypergraph is the number of hyperedges that are not completely contained in any single V_i (we refer to these as the crossing hyperedges), denoted by $E[V_1, \ldots V_k]$.

The *normalized size of a k-cut* in a hypergraph is its un-normalized size divided by $k-1$. We will often use $\mathbf{\Phi}(H)$ to denote the minimum normalized k-cut, defined formally as follows:

$$\mathbf{\Phi}(H) = \min_{k \in [2..n]} \min_{V_1, \cup \cdots \cup V_k = V} \frac{|E[V_1, \ldots V_k]|}{k-1}.$$

Note that when we generically refer to a k-cut, this refers any choice of $k \in [2..n]$. That is, we are not restricting ourselves to a single choice of k, but instead allowing ourselves to range over any partition of the vertex set into any number of parts.

The work of [Qua24] established the following result regarding normalized and un-normalized k-cuts:

Theorem 1.5. *[Qua24] Let H be a hypergraph, with associated minimum normalized k-cut size $\mathbf{\Phi}(H)$. Then for any $t \in \mathbb{Z}^+$, and $k \in [2..n]$, there are at most $n^{O(t)}$ un-normalized k-cuts of size $\leq t \cdot \mathbf{\Phi}(H)$.*

A direct consequence of the above is that in order to preserve all k-cuts (again, simultaneously for every $k \in [2, \ldots n]$) in a hypergraph H to a factor $(1 \pm \epsilon)$, it suffices to sample each hyperedge at rate $p \geq \frac{C \log(n)}{\epsilon^2 \mathbf{\Phi}(H)}$, and re-weight each sampled hyperedge by $1/p$.

Similar to Benczúr and Karger's [BK96] approach for creating $\widetilde{O}(n/\epsilon^2)$ size graph sparsifiers, Quanrud [Qua24] next uses this notion to define *k-cut strengths* for each hyperedge. To do this, we fix a minimum normalized k-cut, with $V_1, V_2, ..., V_k$ denoting the partition of the vertices created by this cut. For any hyperedge crossing this minimum normalized k-cut, we define its strength to be exactly $\mathbf{\Phi}(H)$. For the remaining hyperedges (i.e, those which are completely contained within the components $V_1, \ldots V_k$), their strengths are determined recursively (within their respective induced subgraphs) using the same scheme. This allows Quanrud [Qua24] to calculate sampling rates of hyperedges, which when sampled, approximately preserve the size of every k-cut (for all $k \in [2, n]$). Note that just as in the graph setting, the reciprocal sum of strengths is bounded, which allows for convenient bounds on the number of low strength hyperedges:

Claim 1.6. *[Qua24] Let $H = (V, E)$ be a hypergraph on n vertices. Then,*

$$\sum_{e \in E} \frac{1}{\lambda_e} = n - 1.$$

However, the power of the strength definition extends beyond just identifying sampling rates of hyperedges, and can also be used to identify *sets of vertices* which can be contracted away. In particular, we can define the notion of the *strength of a component*, as we do below:

Definition 1.7. For a subset of vertices $S \subseteq V$, we say that the *strength of S in H* is $\lambda_S = \min_{e \in H[S]} \lambda_e$. That is, when we look at the induced subgraph from looking at S, λ_S is the minimum strength of any edge in this induced subgraph.

We will take advantage of the following fact when working with these "contracted" versions of hypergraphs:

Claim 1.8. *[[KPS24b]] Let H be a hypergraph, and let $V_1, \ldots V_k$ be a set of connected components of strength $> \kappa$. Then, the hyperedges of strength $\leq \kappa$ in H are exactly those hyperedges of strength $\leq \kappa$ in $H/(V_1, \ldots V_k)$, where we use $H/(V_1, \ldots V_k)$ to denote the hypergraph where $V_1, \ldots V_k$ have each been* contracted *to their own super-vertices.*

This final claim will be very important for our algorithm. In particular, it implies that once certain components have become sufficiently strongly connected, we no longer have to worry about recovering low-strength hyperedges *within* the components, and can instead focus only on recovering hyperedges that *cross between* such components. However, before diving more into the details of this approach, we provide a more detailed re-cap of how prior work [KPS24b] recovers low-strength hyperedges generically.

1.2.3 The Work of [KPS24b]

With this formal notion of strength now in hand, we can formally present the algorithm discussed in the previous subsection for sparsification (essentially the framework of [BK96, AGM12, GMT15, KPS24b]):

Algorithm 1: SimpleSparsification(H, ϵ)

Let $H_0 = H$, let C be a sufficiently large constant.
for $i = 0, 1, \ldots \log(m)$ **do**
  Let F_i be all hyperedges in H_i of strength $\leq 2C\log(n)/\epsilon^2$.
  Store F_i.
  Let H_{i+1} be hyperedges in $(H_i - F_i)$ sampled at rate $1/2$.
end
return $\cup_i 2^i \cdot F_i$.

This approach is exactly what was used by [KPS24b] when designing their hypergraph sparsifiers for dynamic streams.

Indeed, their primary contribution was a *linear sketch* which can be used to exactly recover these low-strength hyperedges at each level of the algorithm. Recall that a linear sketch is simply a set of linear measurements of the hypergraph H and thus is *directly implementable* as a dynamic streaming algorithm on hypergraphs, as both insertions and deletions can be modeled as linear updates. Formally, when a hypergraph on n vertices is viewed as a vector in $\{0,1\}^{2^n}$, then a linear measurement of size s is obtained by mutliplying a (possibly random) $s \times 2^n$ matrix with this vector. Inserting a hyperedge is simply a $+1$ update in the coordinate corresponding to the hyperedge, and a deletion is simply a -1.

With this established, we can now summarize the space complexity resulting from the linear sketching implementation of [KPS24b]:

1. At each of the $\log(m)$ levels of sampling in the above algorithm, the linear sketch Section 1.2.1 can be used to recover the low strength edges. This is accomplished by storing $\log(m)$ independent copies of a sketch for recovering low-strength hyperedges (one copy at each level).

2. Within each individual sketch (at a fixed level of the sampling process), there is a specific linear sketch stored for the neighborhood of the n vertices.

3. Each such vertex neighborhood sketch requires space $\widetilde{O}(r/\epsilon^2)$ bits.

In total then, this yields a linear sketch (and hence a dynamic streaming algorithm) which stores $\widetilde{O}(\log(m) \cdot n \cdot r/\epsilon^2)$ bits. Once this sketch has been stored, the algorithm can simply iteratively recover the low strength edges at each of the $\log(m)$ levels of sampling, thereby recovering a sparsifier of the original hypergraph.

1.2.4 Optimizing the Complexity in Insertion-only Streams

At first glance, it may seem that none of these parameters from the work of [KPS24b] can be optimized in the insertion-only setting: indeed, there are m hyperedges in the hypergraph initially, and thus any iterative procedure will require $\Omega(\log(m))$ levels before exhausting the hypergraph if the sampling rate is $1/2$. Likewise, the n vertices are fixed, and the linear sketch designed by [KPS24b] requires storing the linear sketches for each vertex. Lastly, the complexity of the sketches for each vertex cannot hope to be improved, as simply recovering a single hyperedge yields $\Omega(r)$ bits of information. Thus, it may seem that one cannot build on top of this framework while achieving a space complexity that beats $O(nr\log(m))$ bits of space.

However, our first theorem shows that by cleverly merging and contracting vertices as hyperedges are inserted, we can in fact improve the space complexity. Perhaps counterintuitively, our optimization actually comes from *decreasing the number of vertices* (the parameter n above).

To illustrate, let us consider a sequence of hyperedge insertions, and let us suppose that at some point in time, a large polynomial number of hyperedges have been inserted (say, some n^{1002} hyperedges). As one might expect, if so many hyperedges have been inserted, there will naturally emerge certain components in the hypergraph which are very strongly connected. More formally, if we revisit Claim 1.6, we can observe that the number of hyperedges of strength $< n^{1000}$, *must be* less than n^{1001}. This implies that among the n^{1002} hyperedges which have been inserted, the *vast majority* are high strength hyperedges, and thus also define many high strength components.

As a consequence, after these hyperedges are inserted, there will be components $C \subseteq V$ for which all of the hyperedges in the component C are *high-strength hyperedges*, and therefore do not need to be recovered in order to perform sampling by Claim 1.8. Algorithmically, because we are dealing with an *insertion-only* stream, once such a component C becomes a high-strength component, it will forever remain a high-strength component, and thus, as per Claim 1.8, we can effectively contract this component away.

Thus, our algorithmic plan is principled: we will estimate the strength of components as hyperedges are inserted (separately for each level of the $\log(m)$ levels of sampling in the hypergraph), and whenever a component gets sufficiently large strength, we *irrevocably contract* the component away to a single super-vertex. We do this for the hypergraphs at each of the $\log(m)$ levels of sampling. Thus, there are two key points that must be shown:

1. We must show that contracting these vertices saves space in our sketch.
2. We must show that we can estimate the strength in $\widetilde{O}(nr)$ bits of space in the (insertion-only) streaming setting.

In what follows, we explain how we achieve both of these goals.

1.2.5 Saving Space by Contracting Vertices

First, we show that we can save space by contracting vertices. Let us consider the top level of sampling in the hypergraph. As mentioned above, as hyperedges are inserted, there will naturally become certain strongly connected components. So, let us denote one such component by C. Now, once this component is strong, it remains strong, and so we can be sure that we do not need to recover any hyperedges *within* the component, as per Claim 1.8. So, instead of storing the individual linear sketches $\mathcal{S}_v$ for the vertices $v \in C$, we instead *add the sketches together*, yielding $\mathcal{S}_C = \sum_{v\in C} \mathcal{S}_v$. Note that this operation is not reversible, and we are in fact *losing information* about the hypergraph when we perform this addition. However, this is the same reason that we will save some space: indeed, if there are k strong components, we end up storing only the linear

sketches of [KPS24b] for the k super-vertices, leading to a total space usage of $\widetilde{O}(kr/\epsilon^2)$ bits in a single level, as opposed to the $\widetilde{O}(nr/\epsilon^2)$ bits in [KPS24b] (note there is no $\log(m)$ here as we are only looking at the top level of sampling for now).

Unfortunately, this analysis alone is not sufficient for us to claim any space savings. Indeed, it is possible that (for instance) in the top level of sampling, there are *no* strong components. It follows then that we cannot add together any linear sketches, and must instead pay $\widetilde{O}(nr/\epsilon^2)$ at this level of sampling. However, this is where we now use a key bound on the number of low-strength edges Claim 1.6 [Qua24]. If there are no components of strength say, greater than n^{1000}, *then there must be fewer than* n^{1001} *hyperedges remaining.* It follows then that instead of storing this sketch of size $\widetilde{O}(nr/\epsilon^2)$ bits at each of the $\log(m)$ levels of sampling, we must only store it at $\log(n^{1001}) = O(\log(n))$ levels of sampling, thereby replacing this $\log(m)$ factor with a $\log(n)$ factor.

This argument as we have presented it though is far from general and must be extrapolated to work on all instances. In particular, it is possible that at different levels of sampling, the strong components are different, and thus there is no single set of strong components that we can look at. To address this, we make the observation that the strong components form a laminar family. That is to say, the strong components at the $i+1$st level of sampling are a refinement of the strong components at the ith level of sampling. Thus, across all $\log(m)$ levels of sampling, the number of distinct strong components that appear is bounded by $O(n)$. Likewise, because any component of strength $\geq n^{1000}$ is merged away, by the same logic as above, each strong component must have $\leq n^{1001}$ incident hyperedges (i.e., hyperedges which touch this component, as well as some other component). Thus, each component that appears will only have a non-empty neighborhood for $O(\log(n))$ levels of sampling (after which point we do not need to store anything - as the sketch of an empty neighborhood is empty).

In summary then, for each of the $O(n)$ strong component that appears, we store the sketch from [KPS24b] of size $\widetilde{O}(r/\epsilon^2)$ for $O(\log(n))$ different levels of sampling. This then yields the desired complexity of $\widetilde{O}(nr/\epsilon^2)$ bits of space for the final algorithm.

1.2.6 Identifying Strong Components

Finally, we show that we can approximately find the strong components in the hypergraph in an insertion-only stream. Fortunately, the foundation for this algorithm was presented in [KPS24b]: let us consider again the hypergraph at the top level of sampling, which we denote by H. Simultaneously, we consider an auxiliary hypergraph $\hat{H}$ which is the result of sampling the hyperedges of H at rate $1/n^{1000}$.

As shown in [KPS24b], it turns out the connected components in $\hat{H}$ exactly correspond to the strong components in H. Thus, in the insertion-only streaming model, this admits an exceedingly simple implementation: as hyperedges arrive, we sample them at rate $1/n^{1000}$, and keep track of the components. Then, whenever a new component is formed, we simply add together the corresponding linear sketches as discussed in the previous section (i.e., merging those vertices together). Note that storing the set of connected components can be done in $\widetilde{O}(n)$ bits of space, and thus across $\log(m)$ levels of sampling, requires only $\widetilde{O}(n\log(m))$ bits. Because $m \leq n^r$, this then yields the desired space complexity. These ideas then suffice for the insertion-only implementation.

1.2.7 Generalizing to Bounded-Deletions and Remarks

The key observation for generalizing the bounded deletion setting is that once a component has strength $k+n^{1000}$, then after any sequence of k deletions, the strength of the component remains *at least* n^{1000}. Thus, instead of adding samplers together after the components reach strength n^{1000},

we instead add samplers together once the strength reaches $k+n^{1000}$. Observe then that the $\log(k)$ term is a natural artifact: each strong component can have a non-empty neighborhood of incident hyperedges for $\log(k+n)$ levels of sampling, as opposed to simply $O(\log(n))$ levels of sampling, and this yields the final complexity.

We now finish with some remarks:

1. The primary work-horse of [KPS24b] is a linear sketch known as an ℓ_0-sampler, and it is tempting to simply try to replace the ℓ_0-samplers via linear sketching in their paper with an ℓ_0-sampler specifically for insertion-only streams, thereby saving space. However, such a replacement would necessitate an entirely new analysis: even though the stream itself may not have deletions, the process of recovering hyperedges, merging vertices together, and more *all rely* on the ability to linearly add together ℓ_0-samplers (which causes deletions). While the aforementioned approach may work, it would not build on the existing framework. The same goes for the bounded deletion setting, where it is tempting to use ℓ_0-samplers defined for bounded-deletion streams (as discussed in [JW18]).

2. There are several subtleties that arise in the analysis regarding the estimation of strong components, as this will never be an *exact* decomposition of the graph. We instead provide upper and lower bounds on the strength of the components and remaining hyperedges, and use this to facilitate our analysis. We defer a more complete description to the technical sections below.

3. Likewise, in the bounded-deletion setting, there is considerable difficulty in optimizing the dependence on k to *not be* $\log^2(k)$. Roughly speaking, this is because the k dependence tries to show up in both (1) the number of levels of sampling in which a strong component has a non-empty neighborhood (a $\log(k)$ factor as described above) and (2) the support size of the ℓ_0-samplers that are needed when using the sketch of [KPS24b] (another factor of $\log(k)$). We use a more refined analysis along with a *second* round of component merging to bypass this other factor of $\log(k)$.

4. The lower bound follows from the augmented index problem along with an argument from [JW18] on bounding the complexity of this problem in the bounded-deletion setting. We omit a full discussion here, as the analysis presented in Section 5 is concise.

1.3 Organization

In Section 2 we introduce more formal definitions of our model and statements of prior work. In Section 3, we prove Theorem 1.1, in Section 4 we prove Theorem 1.2, and finally in Section 5, we present Theorem 1.3.

2 Preliminaries

2.1 Definitions

First, we recap the definition of a hypergraph, and the notion of a hypergraph cut-sparsifier.

Definition 2.1. A hypergraph $H = (V, E)$ is given by a vertex set V of size n, and a set of hyperedges E where each $e \in E$ is an arbitrary subset of V. For a hyperedge e, we say that the *arity* of the hyperedge is $|e|$. The arity of the hypergraph is $\max_{e\in E} |e|$. The *support size* of the hypergraph is the number of distinct hyperedges, which is denoted by m.

Occasionally, a hypergraph may be given by a triple (V, E, w) where w is a weight function $E : \mathbb{R}^{\geq 0}$. When a weight function is omitted, the implication is that every hyperedge has weight 1.

Given a hypergraph, we can also define the notion of *cuts*:

Definition 2.2. For a hypergraph $H = (V, E, w)$ and a subset $S \subseteq V$, we say that the hyperedges cut by S are

$$\text{cut}_H(S) = \{e \in E : e \cap S \neq \emptyset \wedge e \cap \bar{S} \neq \emptyset\}.$$

The weight of the cut is then given by

$$|\text{cut}_H(S)| = \sum_{e \in \text{cut}_H(S)} w_e.$$

We are now ready to define a cut-sparsifier:

Definition 2.3. For a hypergraph H, and a parameter $\epsilon \in (0, 1)$, a $(1 \pm \epsilon)$ cut-sparsifier of H is a hypergraph $\widetilde{H}$ such that (simultaneously) for every $S \subseteq V$:

$$|\text{cut}_{\widetilde{H}}(S)| \in (1 \pm \epsilon)|\text{cut}_H(S)|.$$

In this work, we will be particularly interested in the streams of hyperedges:

Definition 2.4. A stream of hyperedges is given by a sequence $(e_1, \delta_1), (e_2, \delta_2), \ldots (e_i, \delta_i), \ldots$, where each $e \subseteq V$, and each $\delta_i \in \{\pm 1\}$. A pair $(e, -1)$ corresponds to a hyperedge being deleted, and a pair $(e, 1)$ corresponds to a hyperedge being inserted. The number of deletions in the stream is given by the number of tuples where $\delta_i = -1$.

A stream is a k-deletion stream if there are at most k deletions occurring in the stream. If $k = 0$, we refer to this as an insertion-only stream.

One useful tool for working with streams is the notion of a linear sketch:

Definition 2.5. A linear sketch of a hypergraph of arity r is specified by a (possibly randomized) sketching matrix $M \in \mathbb{Z}^{s \times \binom{n}{\leq r}}$. For a hypergraph H, the sketch is given by representing H as a vector in $\mathbf{1}_H \in \{0, 1\}^{\binom{n}{\leq r}}$ such that $(\mathbf{1}_H)_e = \mathbf{1}[e \in H]$, and then multiplying $M\mathbf{1}_H$.

The number of entries in the sketch is then s, though the number of bits required to represent the sketch may be larger (since the values can be arbitrary integers).

2.2 Useful Notions from Prior Work

Now, we recap some useful results from prior work. In particular, we start with the definition of *strength* in hypergraphs:

Definition 2.6. For a hypergraph $H = (V, E)$, the *minimum normalized k-cut* is defined to be

$$\min_{k \in [n]} \min_{V_1 \cup V_2 \cup \cdots \cup V_k = V} \frac{|E[V_1, \ldots V_k]|}{k - 1}.$$

$|E[V_1, \ldots V_k]|$ refers to the number of edges which cross between (any subset) of $V_1, \ldots V_k$. This is a generalization of the notion of a 2-cut in a graph, which is traditionally used to create cut sparsifiers in ordinary graphs. Further, note that $V_1, \ldots V_k$ form a partition of V. As mentioned in the introduction, we will often use the following to denote the minimum normalized k-cut:

$$\Phi(H) = \min_{k \in [n]} \min_{V_1, \cup \cdots \cup V_k = V} \frac{|E[V_1, \ldots V_k]|}{k - 1}.$$

We also refer later to *un-normalized k-cuts*, which is simply $|E[V_1, \dots V_k]|$, for some partition $V_1, \dots V_k$ of V.

Now, to define strength, we iteratively use the notion of the minimum k-cut.

Definition 2.7. Given a hypergraph $H = (V, E)$, let $\Phi(H)$ be the value of the minimum normalized k-cut, and let $V_1, \dots V_k$ be the components achieving this minimum. For every edge $e \in E[V_1, \dots V_k]$, we say that $\lambda_e = \Phi(H)$. Now, note that every remaining edge is contained entirely in one of $V_1, \dots V_k$. For these remaining edges, we define their strength to be the strength inside of their respective component.

Remark 2.8. Note that the strengths assigned via the preceding definition are non-decreasing. Indeed if the minimum normalized k-cut has value ϕ and splits a graph into components $V_1, \dots V_k$, it must be the case that the minimum normalized k-cuts in each $H[V_i]$ are $\geq \phi$, as otherwise one could create an even smaller original normalized k-cut by further splitting the component V_i. See [Qua24, KPS24b] for a longer discussion.

We will also refer to the strength of a component.

Definition 2.9. For a subset of vertices $S \subseteq V$, we say that the *strength of S in H* is $\lambda_S = \min_{e \in H[S]} \lambda_e$. That is, when we look at the induced subgraph from looking at S, λ_S is the minimum strength of any edge in this induced subgraph.

Definition 2.10. For a hypergraph H and partition $V_1, \dots V_k$ of the vertex set, let $H/(V_1, \dots V_k)$ denote the hypergraph obtained by contracting all vertices in each V_i to a single vertex. For a hyperedge $e \in H$, we say that the corresponding version of $e \in H/(V_1, \dots V_k)$ (denoted by $e/(V_1, \dots V_k)$) is incident on a super-vertex corresponding to V_i if there exists $v \in V_i$ such that $v \in e$.

We will take advantage of the following fact when working with these "contracted" versions of hypergraphs:

Claim 2.11. *Let H be a hypergraph, and let $V_1, \dots V_k$ be a set of connected components of strength $> \kappa$. Then, the hyperedges of strength $\leq \kappa$ in H are exactly those hyperedges of strength $\leq \kappa$ in $H/(V_1, \dots V_k)$.*

Now, we are ready to summarize some important results from [KPS24b]:

Theorem 2.12. *[[KPS24b]] There is a linear sketch of size $\widetilde{O}(nr\kappa \log(m))$ bits, which for arbitrary hypergraphs H on n vertices, $\leq m$ hyperedges, of arity $\leq r$, recovers exactly the hyperedes of H with strength $\leq \kappa$. The linear sketch is composed of n separate linear sketches of size $\widetilde{O}(r\kappa \log(m))$ for each vertex $v \in V$.*

An important building block in the above theorem is the following linear sketch:

Theorem 2.13. *[[GMT15]] There is a linear sketch of size $\widetilde{O}(nr \log(m))$ bits, which for arbitrary hypergraphs H on n vertices, $\leq m$ hyperedges, of arity $\leq r$, recovers exactly the connected components of H with high probability. The linear sketch is composed of $\log(n)$ ℓ_0 samplers initialized for each vertex $v \in V$.*

The following algorithm has appeared in many sparsification settings, and yields a simple procedure for sparsifying hypergraphs:

Algorithm 2: SparsifyHypergraph(H, ϵ)

Let $i = 0$, and for all $j : H_j = H$.
while H_i *is not empty* **do**
  Let F_i be all hyperedges in H_i of strength $\leq C \log(n)/\epsilon^2$, for a large constant C.
  Let $H_{\text{intermediate}} = H_i - F_i$.
  Subsample the hyperedges of $H_{\text{intermediate}}$ at rate $1/2$.
  Set $H_{i+1} = H_{\text{intermediate}}$.
  $i \leftarrow i + 1$.
end
return $\widetilde{H} = \bigcup_i 2^i \cdot F_i$.

A basic analysis shows that the above algorithm yields a $(1 \pm O(\epsilon \cdot \log(m)))$-sparsifier of H with probability $1 - 1/\text{poly}(n)$. This is essentially due to composing $\log(m)$ levels of sparsifiers, and naively combining their error. Thus, in order to get a $(1 \pm \epsilon)$ sparsifier, one would need to instead invoke the algorithm with a parameter $\epsilon' = \epsilon/\log(m)$, which leads to extra factors of $\log(m)$ in the sparsifier size (as one must recover more hyperedges in each level).

The work of [KPS24b] showed the following general fact about error-accumulation when sparsifying hypergraphs:

Theorem 2.14. *[[KPS24b]] For any choice of $\epsilon' \leq \epsilon/\log^2(n/\epsilon)$, SparsifyHypergraph$(H, \epsilon')$ returns a $(1 \pm \epsilon)$ hypergraph cut-sparsifier to H with probability $1 - 1/\text{poly}(n)$.*

Now, we recap a useful auxiliary algorithm that we will use:

Algorithm 3: FindStrongComponents(H)

$\widetilde{H}^{(0)} = H$.
for $i = 1, \ldots r \log(n)$ **do**
  Let $\text{CC}^{(i)}$ be the connected component structure of $\widetilde{H}^{(i)}$.
  Let $\widetilde{H}^{(i+1)} = \widetilde{H}^{(i)}$ with edges sampled at rate $1/2$.
end
return $\{\text{CC}^{(i)} : i \in [r \log(n)]\}$

We have the following claims from [KPS24b]:

Claim 2.15. *Let H be a hypergraph, and let $i \in [r \log(n)]$. Let H_i be hypergraph resulting from Algorithm 2 at the ith level of sampling. Then, the components $\text{CC}^{(i+20\log(n))}$ from Algorithm 3 are components of strength $\geq n^{15}$ in H_i with probability $1 - 2^{-\Omega(n)}$.*

Claim 2.16. *Let H be a hypergraph, and let $i \in [r \log(n)]$. Let H_i be hypergraph resulting from Algorithm 2 at the ith level of sampling. Then, any hyperedge of strength $\geq n^{100}$ in H_i is completely contained in some component $S \in \text{CC}^{(i+20\log(n))}$ from Algorithm 3 with probability $1 - 2^{-\Omega(n)}$.*

As a corollary, we have that if we merge each component in $\text{CC}^{(i+20\log(n))}$ into a super-vertex in H_i, then the number of crossing hyperedges in H_i is bounded.

Corollary 2.17. *Let H be a hypergraph, and let $i \in [r \log(n)]$. Let H_i be hypergraph resulting from Algorithm 2 at the ith level of sampling. Then, the hypergraph $H_i/\text{CC}^{(i+20\log(n))}$ has $\leq n^{101}$ hyperedges, and any hyperedge of strength $\leq n^{15}$ in H_i is still in $H_i/\text{CC}^{(i+20\log(n))}$.*

Proof. TOPROVE 0 □

This final corollary is what allows [KPS24b] to essentially replace the $\log(m)$ dependence in their linear sketch with a $\log(n)$ factor, as the linear sketch is only ever opened on the *contracted* version of the hypergraph, where there are now fewer hyperedges. However, because they carefully contracted, they are also guaranteed that the low-strength edges in the contracted version of the hypergraph are *the same* as the low-strength edges in the original version. Thus, their linear sketch succeeds at recovering from the original hypergraph, despite working on a hypergraph with many fewer hyperedges.

However, as shown by [KPS24b], this approach can only ever get down to $nr\log(m)$ bits of space (as indeed this is optimal for a linear sketch). In this work, we instead want to get rid of this final $\log(m)$ factor when we move to the *insertion-only* regime. With this, we now have the necessary context for deriving our results. We delve into this in the following section.

3 Building Sparsifiers in Insertion-Only Streams

In this section, we will show the following:

Theorem 3.1. *There is an insertion-only streaming algorithm requiring $\widetilde{O}(nr/\epsilon^2)$ bits of space which creates a $(1\pm\epsilon)$ cut-sparsifier for a hypergraph on n vertices and hyperedges of arity $\leq r$, with probabiltiy $1-1/\mathrm{poly}(n)$.*

Remark 3.2. Observe that the above theorem has *no* dependence on the length of the stream. In particular the number of hyperedges m can be potentially as large as $\binom{n}{\leq r}$. Nevertheless, the above algorithm always uses $\widetilde{O}(nr/\epsilon^2)$ bits of space, which nearly-matches the best possible space used in a static sparsifier.

3.1 Insertion-Only Improvement

With the "strong component" contractions introduced in the previous section, this effectively reduces the support of the linear sketch used to recover low-strength hyperedges to $\mathrm{poly}(n)$, meaning that we are storing only $\widetilde{O}(nr/\epsilon^2)$ bits for each of the low-strength recovery sketches. However, there is one final optimization that we can perform in the insertion-only regime.

First, let us recall how exactly the linear sketch for recovering low-strength hyperedges from [KPS24b] works: for each of the n vertices, the linear sketch stores ℓ_0 samplers for the neighborhood of the vertex (under different rates of "fingerprinting"). To get the ℓ_0 samplers for an arbitrary contracted component, one must only *add together* the ℓ_0 samplers of the constituent vertices for that component. However, there is one key limitation in the dynamic streaming (linear sketching) setting. Although we may have information about the strong components at some time step of the algorithm, we must still maintain individual samplers for each vertex. Because the stream can *remove* already inserted hyperedges, it is possible that what was once a strong connected component $S\subseteq V$ at some timestep t is no longer a strong connected component at some timestep t'. Once the component is no longer strong, we must store ℓ_0 samplers for the individual vertices it contains. This forces us to store $\Omega(n)$ ℓ_0 samplers at each level of the sparsification algorithm, hence leading to the $\Omega(nr\log(m))$ bit complexity lower-bound.

However, in the insertion-only streaming model we can actually bypass this issue. Indeed, once a hyperedge is inserted, it will never be deleted. In particular, this means that as the algorithm progresses and more hyperedges are inserted, our estimates of strong components $\mathrm{CC}^{(i)}$ is "monotonically" contracting. That is, our strong components are only ever being merged together, never being split apart. Thus, once we identify a strong component at some level of the algorithm, we

can be sure that we do not need to store individual ℓ_0 samplers for the constituent vertices, and can instead simply store a single ℓ_0 sampler for the entire component as a whole. We detail this new streaming algorithm below:

Algorithm 4: StreamingAlgorithm(H, ϵ))

To start, initialize $\mathrm{CC}^{(i)} = [n]$ for each i.
Initialize the sketch of Theorem 2.12 for each vertex $v \in V$, for each $i \in [r \log(n)]$ using $\kappa = \frac{C \log^5(n/\epsilon)}{\epsilon^2}$. Denote the sketch for vertex v, level i, by $\mathcal{S}_{v,i}$.
for $(e, 1)$ *arriving in the stream* **do**
    for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
        Add e to every linear sketch $\mathcal{S}_{C,i} : C \in \mathrm{CC}^{(i)}$.
        Update $\mathrm{CC}^{(i)}$ to merge any components connected by e.
        If any components $C_1, \ldots C_\ell$ are merged together in $\mathrm{CC}^{(i)}$ to create a larger strong component $C' = C_1 \cup C_2 \cup \ldots C_\ell$, set $\mathcal{S}_{C',i-20\log(n)} = \mathcal{S}_{C_1,i-20\log(n)} + \mathcal{S}_{C_2,i-20\log(n)} + \cdots + \mathcal{S}_{C_\ell,i-20\log(n)}$, and delete $\mathcal{S}_{C_1,i-20\log(n)}, \ldots \mathcal{S}_{C_\ell,i-20\log(n)}$.
        With probability 1/2; end **for**, otherwise, continue.
    end
end

Note that once we have built the sketch, we can also use it to easily recover a sparsifier:

Algorithm 5: RecoverSparsifier(H, ϵ))

for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
    Open the linear sketches $\mathcal{S}_{C,i} : C \in \mathrm{CC}^{(i)}$ in accordance with Theorem 2.12.
    Denote the recovered hyperedges by F_i.
    Remove these recovered hyperedges from sketches at levels $j > i$.
end
return $\bigcup_i 2^i \cdot F_i$.

In the next section, we analyze the above algorithm (both space and accuracy).

3.2 Analysis of Algorithm 4 and Algorithm 5

First, we focus on the space usage of Algorithm 4. We have the following claim:

Claim 3.3. *Fix a time step t of the stream, and let $\mathrm{CC}^{(i)} : i \in [r \log(n)]$ denote the estimates of the strong connected components (across all levels of sampling). Then,*

$$|\{S \subseteq V : \exists i \in [r \log(n)] : S \in \mathrm{CC}^{(i)}\}| \leq O(n).$$

Proof. TOPROVE 1 □

Note that this claim is not enough to give us anything substantial as is. Indeed, for each component we will store $\mathrm{polylog}(n)/\epsilon^2$ ℓ_0-samplers, across each of the levels of sampling in which the component appears. In the worst case, a component will be present for $\Omega(\log(m))$ levels of sampling, and therefore require $\Omega(\log(m)\mathrm{polylog}(n)/\epsilon^2)$ ℓ_0-samplers. Across the $O(n)$ components which appear, this would not offer us any substantial savings in space.

Our key observation is actually in the fact that the components we see *can only have a non-zero neighborhood for $O(\log(n))$ rounds of sampling.* Thus, even though we may store many ℓ_0 samplers

for a given component, *most* of these ℓ_0-samplers are empty, and can therefore be stored with a single bit. We formalize this claim below:

Claim 3.4. *Fix a timestep t. Let S be a component which appears in $\mathrm{CC}^{(i)}$ for some $i \in [r\log(n)]$. Then, there are only $O(\log(n))$ levels of sampling in Algorithm 4 where the neighborhood of S is non-empty with probability $1 - 1/n^{99}$.*

Proof. TOPROVE 2 □

Now, if the neighborhood of S is empty, then to store ℓ_0 samplers for this neighborhood is entirely trivial. Using this, we can bound the total space required by Algorithm 4.

Claim 3.5. *At any timestep t, the total space required by Algorithm 4 is $\widetilde{O}(nr/\epsilon^2)$ bits with probability $1 - 1/\mathrm{poly}(n)$.*

Proof. TOPROVE 3 □

Finally, we provide a proof of correctness; namely that the above algorithm does successfully create a $(1 \pm \epsilon)$ cut-sparsifier with high probability. We use this auxiliary claim:

Claim 3.6. *Let H_i denote the hypergraph at level i of the recovery of Algorithm 5. Then, with probability $1 - 1/\mathrm{poly}(n)$, opening the sketches $\mathcal{S}_{C,i} : C \in \mathrm{CC}^{(i+20\log(n))}$ recovers exactly the hyperedges in H_i of strength $\leq \kappa = \frac{C\log^5(n/\epsilon)}{\epsilon^2}$.*

Proof. TOPROVE 4 □

Now, we are able to conclude the proof of the accuracy:

Claim 3.7. *For a hypergraph H of arity $\leq r$ given through an insertion-only stream, Algorithm 5 recovers a $(1 \pm \epsilon)$ cut-sparsifier of H with high probability.*

Proof. TOPROVE 5 □

We then conclude this section:

Proof. TOPROVE 6 □

In the next section, we discuss how a natural generalization of these ideas can be used to handle *bounded deletion streams* without venturing entirely into the linear sketching regime.

4 Bounded-deletion Streaming Algorithms

In this section, we will show the following:

Theorem 4.1. *For $k \geq 1$, there is a k-bounded deletion algorithm requiring $\widetilde{O}(nr\log(k)/\epsilon^2)$ bits of space which creates a $(1 \pm \epsilon)$ cut-sparsifier for a hypergraph on n vertices and hyperedges of arity $\leq r$, with probability $1 - 1/\mathrm{poly}(n)$.*

Remark 4.2. One way to interpret the above result is as a smooth interpolation between the insertion-only and linear sketching modes. Linear sketching corresponds with the setting where *every* hyperedge can be deleted, leading to $k = m$. On the other hand, insertion-only streams correspond with when $k = 0$.

We present the algorithm for achieving this in the next subsection.

4.1 Bounded-deletion Strong Component Implementation

Let us recall Algorithm 4. The key improvement in this algorithm was in observing that once a component has been designated as *strongly connected*, this component will never cease to be strongly connected. This is because in that setting, the stream only allows insertions. Clearly, in the current setting, such an argument is not valid, as even with bounded deletions, it is certainly possible that a component will become less strong. However, the algorithm can be extended in a simple way which solves this short-coming: indeed, instead of merging components once their strength reaches some $\text{poly}(n)$ level, we instead merge them when the strength reaches $\text{poly}(k \cdot n)$. For instance, if a component has strength $(kn)^2$, then even after some k deletions, the strength of this component will remain large. We formalize this below:

Claim 4.3. *Let H be a hypergraph, and let $C \subseteq V$ denote some set of vertices in H. Let $T \subseteq E$ denote some set of $\leq k$ hyperedges to be deleted. Then,*

$$\lambda_{H-T}(C) \geq \lambda_H(C) - k.$$

That is, the strength of C in $H - T$ decreases by at most k compared to the strength of C in H.

Proof. TOPROVE 7 □

Now, we are ready to present the construction of our bounded-deletion sketch. We first represent the estimation of strong components first, as a separate algorithm:

Algorithm 6: FindStrongComponents(H)

```
H̃^(0) = H.
for i = 1, ... r log(n) do
    Let CC^(i) be the connected component structure of H̃^(i).
    Let H̃^(i+1) = H̃^(i) with edges sampled at rate 1/2.
end
return {CC^(i) : i ∈ [r log(n)]}
```

Naively, if we wanted to build a sparsification scheme resilient to k deletions, we could merge components in H_i in correspondence with the components $\text{CC}^{(i+20\log(n)+2\log(k))}$. This way, because the components would have strength $\text{poly}(nk)$, by Claim 4.3, they would remain strong even after k deletions. While this does yield a non-trivial result, it does not directly solve our problem. Indeed, the analog of Claim 3.4 would imply that a component only has a non-empty neighborhood for $O(\log(n)+\log(k))$ levels of sampling. However, we must *also* pay this factor of $\log(k)$ in the support size of the hypergraph as per Theorem 2.12 (as m is now as large as $\text{poly}(k)$). This would lead to a dependence of $\log(k)^2$, which is too large for us.

Instead, we augment Algorithm 6 with a linear-sketch based version, which provides refined information about the connected components. We state this algorithm below:

Algorithm 7: BoundedDeletionStrongComponents(H, ϵ))

To start, initialize $\mathrm{CC}^{(i)} = [n]$ for each i.
Initialize a connected component linear sketch for each vertex $v \in V$, for each $i \in [r \log(n)]$ as per Theorem 2.13 for support size $m \leq \mathrm{poly}(n)$. Denote the sketch for vertex v, level i, by $\mathcal{S}^{(\mathrm{conn})}_{v,i}$.
for $(e, \delta_e = \pm 1))$ *arriving in the stream* **do**
  for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
    Update $\mathrm{CC}^{(i)}$ to merge any components connected by e.
    If any components $C_1, \ldots C_\ell$ are merged together in $\mathrm{CC}^{(i)}$ to create a larger strong component $C' = C_1 \cup C_2 \cup \ldots C_\ell$, set $\mathcal{S}^{(\mathrm{conn})}_{C',i-20\log(nk)} = \mathcal{S}^{(\mathrm{conn})}_{C_1,i-20\log(nk)} + \mathcal{S}^{(\mathrm{conn})}_{C_2,i-20\log(nk)} + \cdots + \mathcal{S}^{(\mathrm{conn})}_{C_\ell,i-20\log(nk)}$, and delete $\mathcal{S}^{(\mathrm{conn})}_{C_1,i-20\log(nk)}, \cdots \mathcal{S}^{(\mathrm{conn})}_{C_\ell,i-20\log(nk)}$.
    With probability $1/2$; end **for**, otherwise, continue.
  end
  for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
    Add $\delta_e \cdot e$ to every linear sketch $\mathcal{S}^{(\mathrm{conn})}_{C,i} : C \in \mathrm{CC}^{(i+20\log(nk))}$.
    With probability $1/2$; end **for**, otherwise, continue.
  end
end

Now, we show how to recover strong components from this sketch:

Algorithm 8: RecoverComponents(H, ϵ))

for $i \in \{r \log(n), r \log(n) - 1, \ldots 1, 0\}$ **do**
  Open the linear sketches $\mathcal{S}^{(\mathrm{conn})}_{C,i}$ in accordance with Theorem 2.13.
  Denote the resulting connected components by $\widetilde{\mathrm{CC}}^{(i)}$.
  Merge the ℓ_0-samplers at level $i-1$ in accordance with the components $\widetilde{\mathrm{CC}}^{(i)}$.
end
return $\{\widetilde{\mathrm{CC}}^{(i)} : i \in [r \log(n)]\}$.

There are several claims we would like to show about this algorithm. First, we would like to bound the space required by the implementation:

Claim 4.4. *Fix a timestep t. Let S be a component which appears in $\mathrm{CC}^{(i)}$ for some $i \in [r \log(n)]$. Then, there are only $O(\log(nk))$ levels of sampling in Algorithm 7 where the neighborhood of S is non-empty with probability $1 - 1/n^{99}$.*

Proof. TOPROVE 8 □

Now, if the neighborhood of S is empty, then to store ℓ_0 samplers for this neighborhood is entirely trivial. Using this, we can bound the total space required by Algorithm 4.

Claim 4.5. *At any timestep t, the total space required by Algorithm 7 is $\widetilde{O}(nr \log(k)/\epsilon^2)$ bits with probability $1 - 1/\mathrm{poly}(n)$.*

Proof. TOPROVE 9 □

First, we observe that we have the following:

Claim 4.6. *Let H be a fixed hypergraph, and let $i \in [r\log(n)]$. Let H_i be hypergraph resulting from Algorithm 2 at the ith level of sampling. Then, the hypergraph $H_i/\mathrm{CC}^{(i+20\log(nk))}$ has $\leq (nk)^{101}$ hyperedges, and any hyperedge of strength $\leq (nk)^{15}$ in H_i is still in $H_i/\mathrm{CC}^{(i+20\log(nk))}$.*

Proof. TOPROVE 10 □

Next, using this we immediately obtain the following:

Claim 4.7. *Let H be a hypergraph resulting from a k-bounded deletion stream, and let $i \in [r\log(n)]$. Let H_i be hypergraph resulting from Algorithm 2 at the ith level of sampling. Then, the hypergraph $H_i/\mathrm{CC}^{(i+20\log(nk))}$ has $\leq (nk)^{101}$ hyperedges, and any hyperedge of strength $\leq (nk)^{15} - k$ in H_i is still in $H_i/\mathrm{CC}^{(i+20\log(nk))}$.*

Proof. TOPROVE 11 □

Note that in Algorithm 8, there is component information coming from two sources: on the one hand, we are using the pre-processing phase to roughly identify strong components on the order of $\mathrm{poly}(nk)$. During recovery time, we recover the connectivity information *from the bottom-up*, starting with the most aggressive sampling rate. This is because if the hypergraph is connected at a much smaller sampling rate, then it will also be connected at a slightly higher sampling rate. The reason we do this recovery process in reverse is so that we can ensure that (after merging components) the number of hyperedges in the hypergraph is bounded.

Before proceeding, we first introduce a piece of notation: in Algorithm 7, let H_i' refer to the sequence of hypergraphs that are constructed by sub-sampling in Lines 9-13. We show the following claim:

Claim 4.8. *With probability $1 - 1/\mathrm{poly}(n)$, the components $\{\widetilde{\mathrm{CC}}^{(i)} : i \in [r\log(n)]\}$, as returned by Algorithm 8, are exactly the connected components of H_i'.*

Proof. TOPROVE 12 □

Now, we can prove the following:

Claim 4.9. *Let H be a hypergraph, and let $i \in [r\log(n)]$. Let H_i be hypergraph resulting from Algorithm 2 at the ith level of sampling. Then, with high probability over the output $\{\widetilde{\mathrm{CC}}^{(i)} : i \in [r\log(n)]\}$ from Algorithm 8, the hypergraph $H_i/\widetilde{\mathrm{CC}}^{(i+20\log(n))}$ has $\leq n^{101}$ hyperedges, and any hyperedge of strength $\leq n^{15}$ in H_i is still in $H_i/\widetilde{\mathrm{CC}}^{(i+20\log(n))}$.*

Proof. TOPROVE 13 □

Note that the key benefit here is that the number of hyperedges no longer has a dependence on k.

4.2 Bounded-deletion Sparsification Implementation

Now that we have established some properties about the strong-component implementation, we are ready to proceed to the actual implementation of our sparsification scheme. We present the algorithm below:

Algorithm 9: StreamingAlgorithm(H, ϵ))

To start, initialize $\mathrm{CC}^{(i)} = [n]$ for each i.
Initialize the sketch of Theorem 2.12 for each vertex $v \in V$, for each $i \in [r \log(n)]$ using $\kappa = \frac{C \log^5(n/\epsilon)}{\epsilon^2}$, $m = \mathrm{poly}(n)$. Denote the sketch for vertex v, level i, by $\mathcal{S}_{v,i}$.
for $(e, \delta_e = \pm 1))$ *arriving in the stream* **do**
  for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
    Update $\mathrm{CC}^{(i)}$ to merge any components connected by e.
    If any components $C_1, \ldots C_\ell$ are merged together in $\mathrm{CC}^{(i)}$ to create a larger strong component $C' = C_1 \cup C_2 \cup \ldots C_\ell$, set $\mathcal{S}_{C',i-20\log(n)} = \mathcal{S}_{C_1,i-20\log(n)} + \mathcal{S}_{C_2,i-20\log(n)} + \cdots + \mathcal{S}_{C_\ell,i-20\log(n)}$, and delete $\mathcal{S}_{C_1,i-20\log(n)}, \ldots \mathcal{S}_{C_\ell,i-20\log(n)}$.
    With probability $1/2$; end **for**, otherwise, continue.
  end
  for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
    Add $\delta_e \cdot e$ to every linear sketch $\mathcal{S}^{(\mathrm{conn})}_{C,i} : C \in \mathrm{CC}^{(i+20\log(nk))}$.
    With probability $1/2$; end **for**, otherwise, continue.
  end
  for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
    Add $e \cdot \delta_e$ to every linear sketch $\mathcal{S}_{C,i} : C \in \mathrm{CC}^{(i+20\log(n))}$.
    With probability $1/2$; end **for**, otherwise, continue.
  end
end

We also present the recovery algorithm that we use to find the $(1 \pm \epsilon)$ cut-sparsifier:

Algorithm 10: RecoverComponents(H, ϵ))

for $i \in \{r \log(n), r \log(n) - 1, \ldots 1, 0\}$ **do**
  Open the linear sketches $\mathcal{S}^{(\mathrm{conn})}_{C,i}$ in accordance with Theorem 2.13.
  Denote the resulting connected components by $\widetilde{\mathrm{CC}}^{(i)}$.
  Merge the ℓ_0-samplers at level $i-1$ in accordance with the components $\widetilde{\mathrm{CC}}^{(i)}$.
end
for $i \in \{0, 1, \ldots r \log(n)\}$ **do**
  Open the linear sketches $\mathcal{S}_{C,i}$ in accordance with Theorem 2.12.
  Denote the recovered hyperedges by F_i.
  Remove these recovered hyperedges from sketches at levels $j > i$.
end
return $\bigcup_i 2^i \cdot F_i$.

4.3 Bounded-deletion Sparsification Analysis

As before, we first bound the space required by these algorithms:

Claim 4.10. *At any timestep t, the total space required by Algorithm 9 is $\widetilde{O}(nr \log(k)/\epsilon^2)$ bits with probability $1 - 1/\mathrm{poly}(n)$.*

Proof. TOPROVE 14 □

Finally, we provide a proof of correctness; namely that the above algorithm does successfully create a $(1 \pm \epsilon)$ cut-sparsifier with high probability. We use this auxiliary claim:

Claim 4.11. *Let H_i denote the hypergraph at level i of the recovery of Algorithm 10. Then, with probability $1 - 1/\text{poly}(n)$, opening the sketches $\mathcal{S}_{C,i} : C \in \widetilde{\text{CC}}^{(i+20\log(n))}$ recovers exactly the hyperedges in H_i of strength $\leq \kappa = \frac{C\log^5(n/\epsilon)}{\epsilon^2}$.*

Proof. TOPROVE 15 □

Now, we are able to conclude the proof of the accuracy:

Claim 4.12. *For a hypergraph H of arity $\leq r$ given through a k-bounded deletion stream, Algorithm 5 recovers a $(1 \pm \epsilon)$ cut-sparsifier of H with high probability.*

Proof. TOPROVE 16 □

We then conclude this section:

Proof. TOPROVE 17 □

In the next section, we complement this algorithm with a nearly-matching lower bound.

5 Lower Bounds for Creating Sparsifiers

First, we recall the Augmented Index problem (denoted $\mathbf{IND}_N$):

1. Alice is given a vector $y \in \{0,1\}^N$, and sends a message M to Bob.
2. Bob is given an index $i^* \in [N]$, $y_{i^*+1}, \ldots y_N$, and the message M, and must output the bit y_{i^*} with probability $\geq 2/3$.

The following is known regarding $\mathbf{IND}_N$:

Fact 5.1. [CK11] Any protocol which solves $\mathbf{IND}_N$ must use $|M| = \Omega(n)$.

In particular, the proof of the above fact is entirely information theoretic (in fact, [CK11] shows that a specific quantity called the *information cost* is lower bounded). Thus, we immediately have the following:

Definition 5.2. Let ℓ, N be integers. We define the ℓ-Augmented index problem $\ell - \mathbf{IND}_N$ to be ℓ simultaneous instances of the augmented index problem:

1. Alice is given ℓ vectors $y^{(1)}, \ldots y^{(\ell)} \in \{0,1\}^N$, and sends a message M to Bob.
2. Bob is given ℓ indices $i_1^*, \ldots i_\ell^* \in [N]$. Bob is also given the message M, and for each $j \in [\ell]$, Bob is given $y^{(j)}_{i_j^*+1}, \ldots y^{(j)}_N$. Bob must output $y^{(j)}_{i_j^*}$ for each $j \in [\ell]$ with probability $\geq 2/3$.

Naturally, we then have the following:

Corollary 5.3. *Any protocol which solves $\ell - \boldsymbol{IND}_N$ must use $|M| = \Omega(n\ell)$.*[3]

Now, we introduce the notion of a support-sampler for a bounded deletion stream:

[3]Technically, we must bound the information cost, but this follows exactly from [CK11].

Definition 5.4. Let $f \in \{0,1\}^N$ be constructed by a k-bounded deletion stream, where each update of the stream specifies $f_i + 1$ or $f_i - 1$. A one-pass support sampler is any algorithm that gets one pass through the stream and must output any $i \in [N]$ such that $f_i \neq 0$.

Next, we recall the following theorem from [JW18]:

Theorem 5.5 (Theorem 20 of [JW18])**.** *Any one-pass support sampler which outputs an arbitrary $i \in [N]$ such that $f_i \neq 0$ with probability $\geq 2/3$ can be used to solve $\boldsymbol{IND}_p$, for some $p = \Omega(\log(N/k)\log(k))$.*

With this, we can easily derive a corollary for the case of solving ℓ-**IND**:

Corollary 5.6. *Any streaming algorithm which (simultaneously) performs support-sampling on ℓ streams, each on a universe of N items, with $\leq k$-deletions, and overall success probability $\geq 2/3$, must use $\Omega(\ell \log(N/k)\log(k))$ bits of memory.*

Proof. TOPROVE 18 □

Finally, we are ready to conclude:

Theorem 5.7. *Any streaming algorithm for k-bounded deletion streams, which for hypergraphs on n vertices, of arity $r \leq n/2+1$, produces a $(1 \pm \epsilon)$ cut-sparsifier for $\epsilon < 1$, must use $\Omega(nr \log(k/n))$ bits of space.*

Proof. TOPROVE 19 □

References

[AGM12] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 459–467. SIAM, 2012.

[AK95] Charles J Alpert and Andrew B Kahng. Recent directions in netlist partitioning: a survey. *Integration*, 19(1):1–81, 1995.

[BDKS16] Grey Ballard, Alex Druinsky, Nicholas Knight, and Oded Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Trans. Parallel Comput.*, 3(3):18:1–18:34, 2016.

[BK96] András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 47–55. ACM, 1996.

[CAWXZ25] Vincent Cohen-Addad, David Woodruff, Shenghao Xie, and Samson Zhou. Nearly optimal graph and hypergraph sparsification for insertion-only data streams. *Private Communication*, 2025.

[CK11] Amit Chakrabarti and Ranganath Kondapally. Everywhere-tight information cost tradeoffs for augmented index. In Leslie Ann Goldberg, Klaus Jansen, R. Ravi, and

José D. P. Rolim, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 14th International Workshop, APPROX 2011, and 15th International Workshop, RANDOM 2011, Princeton, NJ, USA, August 17-19, 2011. Proceedings*, volume 6845 of *Lecture Notes in Computer Science*, pages 448–459. Springer, 2011.

[CKN20] Yu Chen, Sanjeev Khanna, and Ansh Nagda. Near-linear size hypergraph cut sparsifiers. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 61–72. IEEE, 2020.

[GMT15] Sudipto Guha, Andrew McGregor, and David Tench. Vertex and hyperedge connectivity in dynamic graph streams. In Tova Milo and Diego Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 241–247. ACM, 2015.

[JLS23] Arun Jambulapati, Yang P. Liu, and Aaron Sidford. Chaining, group leverage score overestimates, and fast spectral hypergraph sparsification. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 196–206. ACM, 2023.

[JW18] Rajesh Jayaram and David P. Woodruff. Data streams with bounded deletions. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 341–354. ACM, 2018.

[KK15] Dmitry Kogan and Robert Krauthgamer. Sketching cuts in graphs and hypergraphs. In Tim Roughgarden, editor, *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science, ITCS 2015, Rehovot, Israel, January 11-13, 2015*, pages 367–376. ACM, 2015.

[KKTY21a] Michael Kapralov, Robert Krauthgamer, Jakab Tardos, and Yuichi Yoshida. Spectral hypergraph sparsifiers of nearly linear size. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*, pages 1159–1170. IEEE, 2021.

[KKTY21b] Michael Kapralov, Robert Krauthgamer, Jakab Tardos, and Yuichi Yoshida. Towards tight bounds for spectral sparsification of hypergraphs. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 598–611. ACM, 2021.

[KPS24a] Sanjeev Khanna, Aaron Putterman, and Madhu Sudan. Code sparsification and its applications. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 5145–5168. SIAM, 2024.

[KPS24b] Sanjeev Khanna, Aaron Putterman, and Madhu Sudan. Near-optimal size linear sketches for hypergraph cut sparsifiers. In *2024 IEEE 65th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1669–1706. IEEE, 2024.

[Law73] Eugene L. Lawler. Cutsets and partitions of hypergraphs. *Networks*, 3(3):275–285, 1973.

[Lee23] James R. Lee. Spectral hypergraph sparsification via chaining. In Barna Saha and Rocco A. Servedio, editors, *Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023*, pages 207–218. ACM, 2023.

[Qua24] Kent Quanrud. *Quotient sparsification for submodular functions*, pages 5209–5248. SIAM, 2024.

[SY19] Tasuku Soma and Yuichi Yoshida. Spectral sparsification of hypergraphs. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2570–2581. SIAM, 2019.

[YNY+19] Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha P. Talukdar. Hypergcn: A new method for training graph convolutional networks on hypergraphs. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 1509–1520, 2019.

[ZHS06] Dengyong Zhou, Jiayuan Huang, and Bernhard Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In Bernhard Schölkopf, John C. Platt, and Thomas Hofmann, editors, *Advances in Neural Information Processing Systems 19, Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 4-7, 2006*, pages 1601–1608. MIT Press, 2006.