Near-Optimal Hypergraph Sparsification in Insertion-Only and Bounded-Deletion Streams

Sanjeev Khanna
School of Engineering and Applied Sciences, University of Pennsylvania, Philadelphia, PA, USA

Aaron Putterman
School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA

Madhu Sudan
School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA

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.[1]

2012 ACM Subject Classification Theory of computation → Sketching and sampling

Keywords and phrases Sparsification, sketching, hypergraphs

Digital Object Identifier 10.4230/LIPIcs.ICALP.2025.108

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2504.16321`

Funding *Sanjeev Khanna*: Supported in part by NSF award CCF-2402284 and AFOSR award FA9550-25-1-0107.
Aaron Putterman: Supported in part by the Simons Investigator Awards of Madhu Sudan and Salil

[1] Find a full version of the paper at `https://arxiv.org/abs/2504.16321`.

Vadhan, NSF Award CCF 2152413 and AFOSR award FA9550-25-1-0112.
Madhu Sudan: Supported in part by a Simons Investigator Award, NSF Award CCF 2152413 and AFOSR award FA9550-25-1-0112.

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 [3], to clustering and machine learning [18, 19], and to modeling transistors and other circuitry elements in circuit design [2, 14]. 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 [5] 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 [6, 5, 12].

Most recently, the work of Khanna, Putterman, and Sudan [12] 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 $\widetilde{O}(nr\log(m)/\epsilon^2)$[2] bits for computing $(1 \pm \epsilon)$ sparsifiers in dynamic streams, and also established a nearly-matching lower bound of $\Omega(nr\log(m))$ bits,

[2] 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 $\log(m)$.

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 [10, 9], and moreover, can be constructed in $\widetilde{O}(nr/\epsilon^2)$ bits of space. Thus, one interpretation of the work of [12] 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 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.** *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 $1-1/\mathrm{poly}(n)$.*

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 [12].

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 2.** *For $k\geq 1$, there is a k-bounded deletion streaming algorithm requiring $\widetilde{O}(nr\log(k)/\epsilon^2)$[3] 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 $1-1/\mathrm{poly}(n)$.*

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.

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

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 [8] in the bounded-deletion streaming model, we show that this space complexity is essentially *optimal*:

▶ **Theorem 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.*

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.

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 [13, 17, 5, 10, 9, 7, 15, 11], the work of Quanrud [16] 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. [16] 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 [16] introduces to measure *strength* in hypergraphs is the notion of *k-cuts* in hypergraphs (and here we adopt the language used by [12]):

▶ **Definition 4.** *For any $k \in [2..n]$, a k-cut in a hypergraph is defined by a k-partition of the vertices, say, $V_1, \dots V_k$. 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 $E[V_1, \dots V_k]$.*

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:*

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

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 [16] established the following result regarding normalized and un-normalized k-cuts:

▶ **Theorem 5** ([16])**.** *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 $k \in [2, \dots n]$) 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 [4] approach for creating $\widetilde{O}(n/\epsilon^2)$ size graph sparsifiers, Quanrud [16] 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 $V_1, \dots V_k$), their strengths are determined recursively (within their respective induced subgraphs) using the same scheme. This allows Quanrud [16] 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 6. [16] 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 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 8 ([12]). 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 $H/(V_1, \dots V_k)$, where we use $H/(V_1, \dots V_k)$ to denote the hypergraph where $V_1, \dots V_k$ 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 [12] recovers low-strength hyperedges generically.

1.2.3 The Work of Khanna, Putterman, and Sudan [12]

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 [4, 1, 6, 12]):

Algorithm 1 SimpleSparsification(H, ϵ).

```
Let H_0 = H, let C be a sufficiently large constant.
for i = 0, 1, ... log(m) do
    Let F_i be all hyperedges in H_i of strength ≤ 2C log(n)/ε^2.
    Store F_i.
    Let H_{i+1} be hyperedges in (H_i − F_i) sampled at rate 1/2.
end
return ∪_i 2^i · F_i.
```

This approach is exactly what was used by [12] 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 [12]:

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 [12] 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 [12] 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 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 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 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 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 [12] 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 [12] (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 6 [16]. 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 [12] 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 [12]: 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 [12], 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 [12] 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 [8]).
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 [12] (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 [8] on bounding the complexity of this problem in the bounded-deletion setting. We defer a proof to the full version of the paper.

References

1 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. doi:10.1137/1.9781611973099.40.

2 Charles J Alpert and Andrew B Kahng. Recent directions in netlist partitioning: a survey. *Integration*, 19(1):1–81, 1995. doi:10.1016/0167-9260(95)00008-4.

3 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. doi:10.1145/3015144.

4 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. doi:10.1145/237814.237827.

5 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. doi:10.1109/FOCS46700.2020.00015.

6 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. doi:10.1145/2745754.2745763.

7 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. doi:10.1145/3564246.3585136.

8 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. doi:10.1145/3196959.3196986.

9 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. doi:10.1109/FOCS52979.2021.00114.

10 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. doi:10.1145/3406325.3451061.

11 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.

12 Sanjeev Khanna, Aaron L Putterman, and Madhu Sudan. Near-optimal size linear sketches for hypergraph cut sparsifiers. *arXiv preprint arXiv:2407.03934*, 2024.

13 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. doi:10.1145/2688073.2688093.

14 Eugene L. Lawler. Cutsets and partitions of hypergraphs. *Networks*, 3(3):275–285, 1973. doi:10.1002/NET.3230030306.

15 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. doi:10.1145/3564246.3585165.

16 Kent Quanrud. *Quotient sparsification for submodular functions*, pages 5209–5248. SIAM, 2024. doi:10.1137/1.9781611977912.187.

17 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. doi:10.1137/1.9781611975482.159.

18 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. URL: https://proceedings.neurips.cc/paper/2019/hash/1efa39bcaec6f3900149160693694536-Abstract.html.

19 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. URL: https://proceedings.neurips.cc/paper/2006/hash/dff8e9c2ac33381546d96deea9922999-Abstract.html.