The Case for External Graph Sketching*

Michael A. Bender† Martín Farach-Colton‡ Riko Jacob§ Hanna Komlós‡
David Tench¶ Evan T. West‖

Algorithms in the data stream model use $O(polylog(N))$ space to compute some property of an input of size N, and many of these algorithms are implemented and used in practice. However, sketching algorithms in the graph semi-streaming model use $O(Vpolylog(V))$ space for a V-vertex graph, and the fact that implementations of these algorithms are not used in the academic literature or in industrial applications may be because this space requirement is too large for RAM on today's hardware.

In this paper we introduce the external semi-streaming model, which addresses the aspects of the semi-streaming model that limit its practical impact. In this model, the input is in the form of a stream and $O(Vpolylog(V))$ space is available, but most of that space is accessible only via block I/O operations as in the external memory model. The goal in the external semi-streaming model is to simultaneously achieve small space and low I/O cost.

We present a general transformation from any vertex-based sketch algorithm to one which has a low sketching cost in the new model. We prove that this automatic transformation is tight or nearly (up to a $O(\log(V))$ factor) tight via an I/O lower bound for the task of sketching the input stream.

Using this transformation and other techniques, we present external semi-streaming algorithms for connectivity, bipartiteness testing, $(1+\epsilon)$-approximating MST weight, testing k-edge connectivity, $(1+\epsilon)$-approximating the minimum cut of a graph, computing ϵ-cut sparsifiers, and approximating the density of the densest subgraph. These algorithms all use $O(Vpoly(\log(V), \epsilon^{-1}, k)$ space. For many of these problems, our external semi-streaming algorithms outperform the state of the art algorithms in both the sketching and external-memory models.

*The full version of the paper can be accessed at https://arxiv.org/abs/2504.17563

†Stony Brook University and RelationalAI
‡New York University
§IT University of Copenhagen
¶Lawrence Berekeley National Laboratory
‖Stony Brook University

1 Introduction The streaming model has been widely successful in both the theory and systems literature for a variety of reasons. In this model, the input is presented as a arbitrarily-ordered stream of updates, and the challenge is to design algorithms that compute properties of the input in small (ideally polylogarithmic) space. On the theory side, it is an elegant and simplified model that allows for the development of interesting upper bounds without getting bogged down in the details of the hardware. Despite the relative simplicity of the model, it has also been successful in the systems literature because it does capture important aspects of real computers. Specifically, it captures the idea that caches are small and fast and streams of data arrive quickly and are too big to store. The streaming model succeeded because it hit a sweet spot between the elegance needed for theoretical results and capturing the right hardware constraints for designing software.

However, not all streaming problems can be solved in the streaming model. For example, most graph-theoretic problems have outputs that are by themselves too large to store in the polylogarithmically sized RAM specified by the streaming model. The graph semi-streaming model [21, 39] was introduced in order to bridge this gap. Specifically, in the semi-streaming model we assume that we have $O(V\text{polylog}V)$ space, where V denotes the number of vertices in the input graph. The input stream is a sequence of edge insertions (and possibly deletions).

The semi-streaming model has proven to be fertile soil for theoretical results for both upper and lower bounds. For example, there is a rich literature for addressing a long list of graph problems [4, 2, 40, 21, 39, 28, 25, 5, 15, 6, 36, 29, 34, 17, 37, 3, 43, 16, 32, 8], as well as computational geometric problems [27, 12, 18, 49, 9, 22], and hypergraph problems [25, 35, 19, 38, 7, 10, 26]. The semi-streaming model is elegant and captures something exciting about the structure of graph problems and their algorithms. We refer the reader to the full version of the paperfor a more detailed discussion of related work.

In the general case where the stream contains deletions, all known space-efficient algorithms are ***linear***

sketches. These algorithms generate a random linear projection of the input that can be stored in $O(V\text{polylog}V)$ space. Moreover, there is strong theoretical evidence that linear sketches are universal; i.e., that any space-efficient single-pass semi-streaming algorithm with constant success probability can be formulated as a linear sketch [32].

There is a corresponding need in industry and large-scale science to process massive, dynamic graphs. A recent survey by Sahu et al. [44] of industrial uses of graph algorithms indicates that a majority of industry respondents need to process massive (at minimum multi-billion-edge) graphs, and a majority work with graphs that change over time. Scientific applications include metagenome assembly [24, 42], large-scale clustering [47, 48, 20], and tracking social network communities that change as users add or delete friends [31, 11].

Despite its great theoretical success and a wealth of potential applications, the semi-streaming model has not yielded a corresponding applied literature. The reason is straightforward – for practical purposes $O(V\text{polylog}V)$ space is enormously larger than RAM and will remain so for the foreseeable future. In Appendix A in the full version of the paper, we illustrate this problem with a case study on the k-connectivity sketch of Ahn et al. [4], one of the simplest and smallest graph sketch algorithms. We show that the logarithmic and constant factors in the space complexity of this algorithm are large enough that it will not save space until we have RAM sizes in the hundreds of TBs. We note in the case study that due to a lower bound of Nelson and Yu [40], this asymptotic space complexity cannot be significantly improved. Additionally, the k-connectivity sketch is a subroutine for many other semi-streaming algorithms such as minimum cut, spectral sparsification, and minimum spanning tree [4]. This suggests that the problem illustrated in our case study is not an isolated one but rather a general limitation of many semi-streaming algorithms.

In this paper, we show that not all hope is lost. Our optimism is based on a reexamination of hardware trends. We notice that the bandwidth of high-speed storage systems is now so high that the cost of random access to RAM is comparable to the cost of sequential access to storage. On the other hand, such high-speed storage is expensive and limited in size. So one of the contributions of this paper is a modification of the semi-streaming model based on modern hardware. For a more detailed description of hardware that we use to reach these conclusions, see Appendix B in the full version of the paper.

This observation about bandwidths has an interesting algorithmic implication. Many if not most advanced semi-streaming algorithms use hashing to keep sketches in RAM, and therefore the random-access bandwidth of RAM is an upper bound on their performance.[1] So a hypothetical semi-streaming algorithm that made only sequential accesses could be run on storage at the same speed that traditional semi-streaming algorithms can be run on RAM.

QUESTION 1. *Is there a way to redesign a semi-streaming algorithm that uses random RAM accesses to perform the same computation using sequential accesses instead?*

This notion of algorithms limited to sequential accesses on disk is captured by the well-studied external-memory model [46]. It assumes RAM of size M and disk of unbounded size. Words in RAM can be accessed for free, but disk is accessed in blocks of size $B = o(M)$ and each block access costs one dick access (I/O). The goal is to minimize I/O cost.

So our hardware observations seem to reflect some aspects of both of these models: because semi-streaming algorithms are too large for modern RAM, we hope instead to run them on modern high-speed storage devices. The block-access constraint of the external-memory model captures the need to design algorithms that make sequential accesses for good performance. The space constraint of semi-streaming captures the fact that modern high-speed storage devices are large enough to store semi-streaming data structures but not the entire input graph.

In this paper, we formalize this combination of semi-streaming and external memory to provide a theoretical model that allows for interesting algorithmic development while also holding out the hope that more of the good ideas that have already been developed in the semi-streaming literature can find their way to practical relevance.

The external semi-streaming model. As in the semi-streaming model, graph updates in the form of edge insertions or deletions are received in a stream, and the total space available is $O(V\text{polylog}(V))$. However, there is an additional constraint on the *type* of memory available for computation: only $M = \Omega(\text{polylog}(V))$ and $M = o(V)$ RAM is available, as in the data stream model, and $D = O(V\text{polylog}(V))$ and $D = o(V^2)$ disk space is available. As in the external-memory model, a word in RAM is accessed at no cost, and disk is accessed in blocks of $B = o(M)$ words at a cost of a single I/O. The algorithmic challenge in the new model is to minimize the I/O complexity (of ingesting stream

[1] Sequential-access RAM is significantly faster (an order of magnitude) but existing sketch data structures do not perform updates sequentially.

updates and computing solutions to queries) in addition to satisfying the typical limited-space requirement of the data stream model.

The technical challenge. Of course, any existing semi-streaming algorithm can be run in the new model, in the sense that the data structures can be stored on disk. However, since most existing graph-sketching algorithms use techniques such as hashing to random locations [33, 4, 28, 25, 8], most accesses that the algorithm makes will be random accesses. So this approach falls short because the I/O cost is too high.

Let us get more specific about what a good algorithm in the new model would look like. Any graph sketch algorithm first compresses a large graph stream into a small sketch (we call this *sketching the input stream*), and then extracts an answer to the problem from the small sketch. We want an algorithm that completes both of these steps at low I/O cost and uses low space throughout. Specifically an ideal algorithm would have the following properties:

1. **Sketching cost.** The cost of sketching the input stream is at most the cost of sorting it. We choose sorting as our target complexity because it is a natural lower bound for most non-trivial external-memory problems.

2. **Extraction cost.** The cost of computing the desired property of the input graph from the sketch is no more than the cost of computing the property on a sparsified graph via the best existing external-memory algorithm.

3. **Space.** The space required is the same as the best existing graph sketch for the problem.

1.1 Overview of Results Our first result is a definition of the external semi-streaming model. We also present external semi-streaming graph sketch algorithms for classical problems: connectivity, hypergraph connectivity, minimum cut, cut sparsification, bipartiteness testing, minimum spanning tree, and densest subgraph. These algorithms meet or nearly meet the above list of properties that ideal external semi-streaming algorithms should have; see Section 3 for a discussion.

Moreover, we show how to transform a graph sketch algorithm that was not designed with external semi-streaming in mind into one that sketches the input stream with an I/O cost roughly equivalent to that of permuting the stream. This transformation does not increase the space cost. This transformation applies to a large [33, 4, 28, 25, 8] class of sketches which are called ***vertex-based sketches*** (see Section 2).

We complement these upper bounds with I/O lower bounds for sketching input streams in external memory via a reduction from sparse matrix-dense vector multiplication. For several of the problems we consider, the upper and lower bounds match.

The external semi-streaming algorithms we present in this paper match the space costs of the best existing semi-streaming algorithms for these problems, but have much lower I/O complexity. Interestingly, several of our external semi-streaming algorithms have I/O costs comparable to or better than existing external-memory algorithms for the same problems. There are two important consequences of these results.

Graph-sketching algorithms via external-memory techniques. Practical graph-sketching algorithms will have to use disk, but achieving I/O efficiency is not overly painful. By leveraging external-memory techniques, we design graph-sketching algorithms with low I/O cost. In fact, the I/O cost of most of these algorithms is competitive with the best existing external-memory algorithms even though our algorithms use limited space and only have stream access to the input. Even better, these results show that the algorithm designer who desires practical semi-streaming algorithms need not throw out existing techniques from the semi-streaming literature. They simply require additional work to make them efficient in the new model.

External-memory graph algorithms via graph sketching. Graph sketching is a fruitful technique for designing external-memory graph algorithms because one can exploit the data locality of sketches to minimize I/Os. For example, we present the first nontrivial external-memory algorithms for hypergraph connectivity, cut sparsification, and densest subgraph. Our algorithms for $k-$connectivity and $\varepsilon-$approximate min cut also outperform the best existing algorithms for these problems for some parameter settings (see Section 3 for details).

Roadmap Section 2 gives definitions and other preliminaries. We discuss our results in more detail in Section 3. We give an external semi-streaming algorithm for the connected components problem in Section 4. Section 5 gives a general transformation to efficiently process any vertex-based sketch algorithm in external memory, together with a corresponding lower bound and algorithms for other problems as corollaries. In Section 6, we provide external semi-streaming algorithms for some problems that require additional techniques for extracting the answer from the sketches. We conclude with a discussion in Section 7. In the interest of space, some parts of this paper refer the reader to the full version of the paper, namely the proofs of most lemmas/theorems, a discussion of related work, and the appendix sections. That version can be found at the following link: https://arxiv.org/abs/2504.17563.

2 Preliminaries

Graphs and hypergraphs. For graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ let $V = |\mathcal{V}|$ and $E = |\mathcal{E}|$. In this paper we consider only undirected graphs. For convenience, we number the nodes in the graph arbitrarily from 0 to $V-1$ and adopt the convention that the node ID of u is less than node ID of v for edge $e = (u, v)$. We refer to u as the ***left endpoint*** of e and v as the ***right endpoint*** of e. We sometimes consider weighted graphs where each edge has a weight $w(e) \geq 0$. We define the following notation for graph properties: let $\lambda(\mathcal{G})$ denote the minimum weight cut (or equivalently the ***edge connectivity***) of graph $\mathcal{G}$, and then $|\lambda(\mathcal{G})|$ denote the weight of that cut. Similarly, let $\lambda_{st}(\mathcal{G})$ denote the minimum weight $s-t$ cut in $\mathcal{G}$, let $\lambda_e(\mathcal{G}) = \lambda_{uv}$ denote the $u-v$ cut for edge $e = (u, v)$, and let $|\lambda_{st}(\mathcal{G})|$, $|\lambda_e(\mathcal{G})|$ denote their respective weights. For any $S \subset \mathcal{V}$ let $\lambda_S(\mathcal{G})$ denote the $(S, \mathcal{V} \setminus S)$ cut in (hyper)graph $\mathcal{G}$.

A hypergraph $\mathcal{G}$ is specified by a set of vertices $\mathcal{V}$ and a set $\mathcal{E}$ of subsets of $\mathcal{V}$ called hyperedges. We assume all hyperedges have cardinality at most r for some fixed r. Hypergraphs are a generalization of graphs, which correspond to the special case where each hyperedge has cardinality two. Let a spanning graph $T = (\mathcal{V}, \mathcal{E}_T)$ of a hypergraph be a subgraph such that $|\lambda_S(T)| \geq \min\{1, |\lambda_S(\mathcal{G})|\}$ for every $S \subset \mathcal{V}$.

Semi-streaming model. In the ***graph semi-streaming*** model [21, 39] (sometimes just called the ***graph-streaming*** model), an algorithm is presented with a ***stream*** ξ of updates (each an ***edge insertion*** or ***deletion***) where the length of the stream is N. Stream ξ defines an undirected input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The challenge in this model is to compute (perhaps approximately) some property of $\mathcal{G}$ given a single pass over ξ and at most $o(V^2)$ (and ideally $O(V\text{polylog}(V))$) words of memory. The model can be extended to hypergraphs as well and the formalism below applies to both settings.

Each stream update has the form $(e, \Delta, w(e))$ where $e = (u, v)$ for $u, v \in \mathcal{V}$, $u < v$, $\Delta \in \{-1, 1\}$ where 1 indicates an edge insertion, -1 indicates an edge deletion, and $w(e)$ denotes the weight of the edge. For most of the problems considered in this paper, the graph is unweighted, and in these cases we omit $w(e)$ in the update notation. Let s_i denote the ith element of ξ, and let ξ_i denote the first i elements of ξ. Let $\mathcal{E}_i$ be the edge set defined by ξ_i, i.e., those edges which have been inserted and not subsequently deleted by step i. The stream may only insert edge e at time i if $e \notin \mathcal{E}_{i-1}$, and may only delete edge e at time i if $e \in \mathcal{E}_{i-1}$.

Once every update in the graph stream has been processed, a single query is performed, to which the algorithm returns the computed property of the graph. The query procedure is performed using the $O(V\text{polylog}(V))$ memory retained by the algorithm at the conclusion of the stream.

Vertex-based Sketches. In this paper we present techniques that apply to a large family of graph sketch algorithms[33, 4, 28, 25, 8] called vertex-based sketches.

DEFINITION 2.1 (Vertex-based sketch[25]). *We say a linear measurement is local for vertex v if the measurement only depends on edges incident to v, i.e., $ce = 0$ for all edges that do not include v. We say a sketch is vertex-based if every linear measurement is local to some vertex.*

In other words, a vertex-based sketch is partitioned such that each part is mapped to a unique vertex in the graph, and each edge update only needs to be applied to the sketches associated with its endpoints.

External-memory model. In the ***external-memory (EM)*** model [46], memory is partitioned into RAM and disk. RAM has size M and disk has unbounded size. A word stored in RAM may be accessed at no cost, while disk is accessed in blocks of $B = o(M)$ words. We refer to a disk block read or write as an I/O, and the goal is to minimize the number of I/Os required for an algorithm.

We will use the shorthand notation $\text{scan}(N) = \Theta\left(\frac{N}{B}\right)$, $\text{sort}(N) = \Theta\left(\frac{N}{B}\log_{M/B}\left(\frac{N}{B}\right)\right)$, and $\text{permute}(N) = \min(N, \text{sort}(N))$ for the optimal I/Os to scan, sort, and permute data of size N in external memory, respectively [14].

External semi-streaming model. For convenience, we restate the definition of our external semi-streaming model here.

In the ***external semi-streaming model***, edge insertions or deletions arrive in a stream. An algorithm in this model has $M = \Omega(\text{polylog}(V)) = o(V)$ RAM available, and $D = O(V\text{polylog}(V)) = o(V^2)$ disk space. A word in RAM is accessed at no cost, and disk is accessed in blocks of $B = o(M)$ words at a cost of a single I/O.

3 Detailed Discussion of Results

In Table 3.1, we summarize the space and I/O bounds for the algorithms we present in this paper. All of our sketches are vertex based (see Definition 2.1) so for each algorithm A_i, its sketch is partitioned into exactly V equal-sized ***vertex sketches*** and we denote the size of a vertex sketch as ϕ_i. For most of these algorithms, the I/O cost is dominated by the cost of permuting the input stream, subject to mild assumptions. Specifically, this is true when N, the length of the stream, is greater than the size of the sketches, and $M = \Omega(\phi_i)$, i.e., a single vertex sketch fits in RAM.

Table 3.2 compares these bounds to those of the

Algorithm	Vertex Sketch Size	I/O Cost
Connected Comp.	$\phi_1 = O(\log^2 V)$	$O\left(\text{permute}(N) + \text{sort}(V\phi_1)\right)$
Bipartiteness Testing	$\phi_2 = O(\log^2 V)$	$O(\text{permute}(N) + \text{sort}(V\phi_2))$
Hypergraph Conn.	$\phi_3 = O(r^2 \log^2 V)$	$O(\text{permute}(rN) + \text{scan}(N\phi_3/M) + \text{sort}(V) \cdot \text{poly}(r, \log V))$
ϵ-Appx MST Weight	$\phi_4 = O(\epsilon^{-1} \log^2 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_4/M) + \text{sort}(V) \cdot \text{poly}(\epsilon^{-1}, \log V))$
k-Connectivity	$\phi_5 = O(k \log^2 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_5/M) + \text{sort}(V) \cdot \text{poly}(k, \log V))$
ϵ-Appx Min Cut	$\phi_6 = O(\epsilon^{-2} \log^4 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_6/M) + \text{sort}(V) \cdot \text{poly}(\epsilon^{-1}, \log V))$
ϵ-Cut Sparsifier	$\phi_7 = O(\epsilon^{-2} \log^5 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_7/M) + \text{scan}(V^{2+o(1)}) \cdot \text{poly}(\epsilon^{-1}, \log V))$
$2(1+\epsilon)$-Appx Densest Subgraph	$\phi_8 = O(\epsilon^{-2} \log^2 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_8/M) + \text{sort}(V^2) \cdot \text{poly}(\epsilon^{-1}, \log V))$

Table 3.1: Space and I/O bounds for our algorithms in words of space. N denotes the length of the stream, V denotes the number of vertices in the graph, and M denotes the size of RAM. To improve readability, we report the space of our algorithms in terms of ϕ, the size of a vertex sketch (all of the reported sketch algorithms except densest subgraph are vertex-based; see Section 5). The total space for algorithm i is $O(V\phi_i)$. For hypergraph connectivity, r denotes the maximum hyperedge cardinality.

Algorithm	Sketch		Ext. Mem.	
	I/O	Space	I/O	Space
Connected Components	Better	Same	Same	Better
Bipartiteness Testing	Better	Same	Same	Better
Hypergraph Connectivity	Better	Same	First	First
ϵ-Approximate MST Weight*	Better	Same	Worse	Better
k-Connectivity	Better	Same	Conditional	Better
ϵ-Approximate Minimum Cut	Better	$O(\log)$-factor worse	Conditional	Better
ϵ-Cut Sparsifier	Better	Same	First	First
$2(1+\epsilon)$-Approximate Densest Subgraph*	Better	Same	First	First

Table 3.2: Comparison of our external semi-streaming algorithms' space and I/O complexities to the best existing graph sketching and external-memory algorithms. For example, "Better" indicates that the external semi-streaming algorithm has a lower cost than the other algorithm, and "Worse" indicates that it has a higher cost. Note for MST weight we compute an approximation while the best EM algorithm solves it exactly, and for densest subgraph we compute a $2(1+\varepsilon)$-approximation while the existing sketch gives a $(1+\varepsilon)$-approximation.

best existing graph sketch and external-memory algorithms for the problems we study. To compare I/O costs against external memory graph algorithms which assume a static input graph, we treat them as having an insert-only input stream of length $N = E$. For reference, the full details of the space and I/O costs of these existing algorithms are summarized in Appendix C in the full version of the paper.

Comparison to existing graph sketches. Our external semi-streaming graph sketches always have significantly lower I/O costs than existing graph sketches for the same problems, and always match their space costs (with the exception of the cut sparsifier sketch, which uses a $\log V$ factor more space).

Comparison to existing external-memory algorithms. Our external semi-streaming graph sketches always use less space than existing external-memory algorithms except when $N = o(V\phi_i)$, i.e., when the graph is very sparse and the input stream is very short. Our algorithms for hypergraph connectivity, approximate densest subgraph, and cut sparsification are the first non-trivial external-memory algorithms for these problems to our knowledge. For connected components and bipartiteness testing, our sketches essentially match (permute(N) vs. sort(N)) the I/O costs of the best known algorithms. For approximate MST, our graph sketch has worse I/O performance than the best exact EM algorithm. For $k-$connectivity, our algorithm performs better than the best EM algorithm when $k = O(\min(M \log^3 V, E/V))$. For $\varepsilon-$approximate min cut, our algorithm performs better than the best (exact) EM algorithm if $\epsilon = \Omega\left(\max\left(M^{-1/2} \log^{-1/2} V, (V/E)^{1/4} \text{loglog}^{1/4} V\right)\right)$.

The upshot is that many of our external semi-streaming algorithms have I/O costs comparable to or better than existing external-memory algorithms.

4 An External Semi-Streaming Algorithm for Connectivity

We begin by considering the problem of computing the connected components problem in the external semi-streaming model. Ahn et al. [4] give a $O(V \log^2 V)$-space sketching algorithm to solve semi-streaming connectivity and, later, Nelson and Yu [40] prove that this space cost is optimal. Subsequent work by Tench et al. [45] presents a somewhat I/O-efficient version of the connectivity sketch, which was sufficient to achieve good performance when implemented, but falls short of our desired properties for a external semi-streaming algorithm. In this section we present a sketching algorithm which improves on the I/O cost of the Tench et al. algorithm, and actually matches the I/O cost of the best known external-memory connected components algorithm (assuming that $E = \Omega(V \log^2 V)$, that is, when the graph is not so sparse that sketching would save no space). Further, we show a lower bound in Section 5 for a large family of sketch algorithms that implies as a special case that our connectivity sketch is I/O-optimal.

4.1 Ahn et al.'s Connectivity Sketch

We begin by reviewing the dynamic semi-streaming connectivity algorithm of Ahn, Guha, and McGregor, which we refer to as StreamingCC [4]:

Theorem 4.1. *There exists an $O(V \log^2(V))$-space dynamic streaming algorithm for the connected components problem that succeeds with high probability (w.h.p.) in V.*

The algorithm in the above theorem is a linear-sketching algorithm:

Definition 4.2. *A linear measurement of a graph on n vertices is defined by a set of coefficients $\{c_e : e \in \binom{V}{2}\}$. Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the evaluation of this measurement is defined as $\sum_{e \in \mathcal{E}} c_e$. A sketch is a collection of (non-adaptive) linear measurements. The cardinality of this collection is referred to as the size of the sketch. We will assume that the magnitude of the coefficients c_e is poly(n).*

Nearly all known small-space algorithms for data stream problems whose input streams have both insertions and deletions are linear sketch algorithms. Further, Li et al. [32] show that the family of linear sketch algorithms are essentially universal for insert/delete data stream problems: for any space-optimal algorithm that succeeds with constant probability, there is an equivalent linear sketching algorithm that uses at most a logarithmic factor more space than optimal.

In this paper we elide many details of StreamingCC but make several necessary observations here. Let $\mathcal{S}(\mathcal{G})$ denote a connectivity sketch of graph $\mathcal{G}$. $\mathcal{S}(\mathcal{G})$ can be partitioned into V $O(\log^2(V))$-size data structures $\mathcal{S}^0(\mathcal{G}), \mathcal{S}^1(\mathcal{G}), \ldots \mathcal{S}^{V-1}(\mathcal{G})$ which have the property that edge update $e = (u, v, \Delta)$ induces changes only to $\mathcal{S}^u(\mathcal{G})$ and $\mathcal{S}^v(\mathcal{G})$. We call $\mathcal{S}^u(\mathcal{G})$ the ***vertex sketch of vertex u***. For a subset $A \subseteq V$, we let $\mathcal{S}^A(\mathcal{G}) = \bigcup_{u \in A} \mathcal{S}^u(\mathcal{G})$ denote the union of the sketches of the vertices in A.

Crucially, the sketch is linear; i.e., it has the property that $\mathcal{S}^u(\mathcal{G}) = \sum_{(v,w) \in \mathcal{E}} \mathcal{S}^u((v, w))$ for all $u \in \mathcal{V}$. $\mathcal{S}(\mathcal{G})$ is computed by keeping a running sum of the vertex sketches of each stream update.

$\mathcal{S}^u(\mathcal{G})$ may be queried to sample edges from the neighborhood of u, and $\mathcal{S}^A(\mathcal{G})$ may be queried to sample edges from the cut $(A, \mathcal{V} \setminus A)$. After the stream, the algorithm finds the connected components using Boruvka's algorithm [41], querying sketches to sample edges leaving each component.

Some algorithms in this paper make use of multiple connectivity sketches of the same graph, where each connectivity sketch is initialized with different random bits. We denote the ith connectivity sketch as $\mathcal{S}_i(\mathcal{G})$.

In the external semi-streaming model, StreamingCC has high I/O complexity. While Tench et al.'s modified algorithm gets better performance in practice, their I/O complexity can be improved. See Appendix D in the full version of the paper.

4.2 Connectivity: Sketching the Input Stream

We present a new algorithm called ExtSketchCC that computes the connected components of the graph defined by the input stream. We refine the stream-sketching technique of Tench et al. [45] and introduce a new query procedure. As a result, ExtSketchCC uses fewer I/Os to sketch the input stream and to compute connectivity from the sketch than either StreamingCC or Tench et al.'s algorithm.

First we describe how ExtSketchCC sketches the input stream. As stream updates come in, we process them in batches. For each batch, the updates are initially sent to disk after collecting B at a time. Once $O(V \log^2 V)$ updates have been collected, we empty the batch by applying all its updates to the corresponding vertex sketches. We repeat this process for every succeeding $O(V \log^2 V)$ updates until the stream terminates.

We now describe the batching and update procedure in more detail. We arbitrarily partition the vertices of the graph into ***vertex groups*** of cardinality $\max\{1, B/\log^2(V)\}$. Let $\mathcal{U} \subset \mathcal{V}$ denote a vertex group. We store $\mathcal{S}^{\mathcal{U}}(\mathcal{G})$, the vertex sketches associated with the vertices in $\mathcal{U}$, contiguously on disk. This allows $\mathcal{S}^{\mathcal{U}}(\mathcal{G})$ to be read into memory I/O efficiently: if vertex groups are of cardinality 1, then B is smaller than the size of a vertex sketch, and if each vertex group has cardinality

$B/\log^2(V) > 1$, then the sketches for the group have total size $\Theta(B)$.

For each vertex group, we have a corresponding ***update buffer***, which will collect the updates affecting that vertex group so that they can be processed efficiently. The update buffers are stored on disk in the same order as the vertex sketches. Since each edge update $e = (u, v)$ needs to be applied to both endpoints, before performing the following update procedure, we make a copy of each edge update, and mark one copy with the left endpoint u and one with the right endpoint v. Once a batch is full, we permute the updates into the update buffers corresponding to the marked (left or right) endpoints. If the update buffer belonging to any vertex group $\mathcal{U}$ fills up during the course of the permuting procedure, we immediately empty it by reading $\mathcal{S}^{\mathcal{U}}(\mathcal{G})$ into memory and applying the updates in the buffer. This ensures that all updates can be placed in their target update buffers without overflow. Once all elements have been permuted, we simultaneously scan through the update buffers and the sketches, applying each remaining update to the corresponding sketch.

The proof of the following lemma is deferred to the full version of the paper.

LEMMA 4.3. *ExtSketchCC's stream ingestion uses $O(V \log^2(V))$ space and $O\left(\min\left(N, \frac{N}{B}\log_{M/B}((V/B)\log^2 V)\right) + \text{scan}(V\log^2 V)\right)$ I/Os.*

4.3 Extracting the Components from the Sketch We now describe ExtSketchCC's procedure for computing connected components once all stream updates have been processed.

Our algorithm proceeds through $O(\log V)$ rounds each with three phases. In the first phase, an edge is recovered from the sketch of each current connected component. These edges make up a 'merge list'.

In the second phase, for each edge in the merge list, its endpoints are merged in a union–find data structure which keeps track of the current connected components. We use the union–find data structure of Agarwal et al. [1] for efficient batched computation. To obtain the best bounds from this data structure, we need the merge list to be free of redundant merges (i.e., two different edges that effectively merge the same components). To achieve this, we preprocess the merge list as follows. First, we find the supernodes corresponding to the endpoints of each edge in parallel: we sort edges by increasing node ID of their left endpoints. Then we simultaneously scan through this sorted edge list and the union–find data structure to get the parent of each left endpoint. Repeating this $\alpha(V)$ times gives the component of each left endpoint, where α denotes the inverse Ackermann function. Finally, we repeat these steps for the right endpoints of each edge. At the end of this phase we have a new list of $O(V)$ merges of the form $u \to v$, indicating that the sketch for component u should be merged into the sketch for component v. In order to remove redundant merges, we construct a graph $\mathcal{H}$ from this list, where each vertex corresponds to a supernode in the list, and an edge (u, v) for every merge $u \to v$. Connected components in $\mathcal{H}$ correspond to nodes that will all be merged together when all merges in the list are completed. Therefore, we run the external-memory connected components algorithm of Chiang et al. [14] to compute these connected components. We then replace the merges in the list with merges of the form $u \to v'$, where v' is the representative of the connected component of v. Finally, we run this list of merges through the union find data structure.

In the third phase, for each pair of connected components merged in phase 2, the corresponding sketches are summed. Summing the sketches of the merged components together naïvely is I/O efficient if $B = O(\log^2(V))$, since the disk reads and writes necessary for summing sketches are the size of a block or larger.

If $B = \omega(\log^2(V))$, that is if sketches are much smaller than the block size, then we need a more sophisticated merge procedure. Since the merges performed in each round of Borůvka are a function both of the input stream and of the randomness of the sketches, these merges induce random accesses to the sketches on disk if performed directly. In this case, this induces $O(1)$ I/Os per sketch merged for a total cost of $O(V)$ I/Os to perform all the sketch merges. However, this operation can be done more efficiently by sorting the merge list by merge source in node ID order and sorting the sketches in the same order. We then scan through the sketches, marking each sketch with its merge destination, and finally sort the sketches by these merge destinations. Now, because the sketches for each component are stored contiguously, we can perform all the merges with one more scan of the sketches.

The proof of the following lemma is deferred to the full version of the paper.

LEMMA 4.4. *Once all stream updates have been processed, ExtSketchCC computes connected components using $O(\text{sort}(V\log^2(V)))$ I/Os.*

ExtSketchCC can also be used as an external-memory connected components algorithm on a static graph. Provided the graph has enough edges, it matches the best known upper bound of $\text{sort}(E)$ I/Os for connected components [14], and uses $\widetilde{O}(V)$ less space.

COROLLARY 4.5. *When $E = \Omega(V\log^2(V))$, ExtSketchCC solves the connected components problem in*

$O\left(\min\left(E, \frac{E}{B}\log_{M/B}((V/B)\log^2 V)\right)\right)$ *I/Os.*

5 A General Transformation for External Semi-Streaming Graph Sketching In the prior section we showed that connectivity can be solved via sketching for essentially the same I/O cost as the best known (non-sketching) external-memory algorithm. This surprising fact is due in part to the fact that it is a vertex-based sketch, because each edge update only needs to be applied to the sketches associated with its endpoints. In this section we show how to, for any such algorithm, sketch the input stream using essentially the number of I/Os required to permute the input stream — without increasing the space cost. We then show that this I/O cost is optimal or nearly optimal (depending on the problem) via a lower bound based on a reduction to sparse matrix/dense vector multiplication.

THEOREM 5.1. *For any single-pass vertex-based sketch streaming algorithm, the input stream can be processed using*

$$\begin{aligned}\text{vsketch}(N, V, \phi) \coloneqq \\ O\left(\min\left(N, \frac{N}{B}\log_{M/B}(V\phi/B)\right)\right. \\ \left. + \text{scan}(N\phi/M) + \text{scan}(V\phi)\right)\end{aligned}$$

I/Os and total space $O(V\phi)$.

Proof. We follow the procedure described in Section 4.2, except we set the size of a batch to be $O(V\phi)$, vertex groups to have cardinality $\max\{1, B/\phi\}$ and the size of each update buffer to be ϕ.

In the case where $\phi = o(M)$, the result follows immediately. In the case where $\phi = \Omega(M)$, the procedure for applying the updates in an update buffer to a sketch is more complicated and expensive. (In this case, vertex groups will always be of size 1.)

Sorting each batch costs $\text{permute}(V\phi) = \min(V\phi, \text{sort}(V\phi))$ I/Os. Whenever the update buffer for vertex u fills, we apply the updates by holding the first $O(M)$ elements of the buffer in memory and scanning over $\mathcal{S}_u(\mathcal{G})$ in $O(M)$-size chunks, applying the updates to each chunk. This costs $\text{scan}(\phi)$ I/Os per $O(M)$ updates, and there are at most $V\phi/M$ such sets of updates per batch, for a total I/O cost to apply the updates of $O(\text{scan}(V\phi^2/M))$ per batch.

There are $N/(V\phi)$ batches, for a total ingestion I/O cost of $O\left(\frac{N}{V\phi}(\text{permute}(V\phi) + \text{scan}(V\phi^2/M))\right) = O\left(\min\left(N, N\log_{M/B}(V\phi/B)\right) + \text{scan}(N\phi/M)\right)$.

Finally, similarly to Lemma 4.3, we have an I/O cost of $\text{scan}(V\phi)$ in the case that $N < V\phi$. ∎

5.1 A Matching I/O Lower Bound Now we show an I/O lower bound for sketching the input stream for any vertex-based sketch algorithm. Depending on the problem and M, the lower bound either matches the upper bound exactly or has a $O(\log V)$ gap.

To prove a lower bound, it is useful to separate the sketching from the data-structural aspect, so that we can argue about the I/O complexity of the data-structure. Here, we focus on sketching algorithms such that

- The sketch is created from N edge updates (insert or delete) that are presented in arbitrary order to the data structure.
- Sketches work with vertex sketches of polylog many numbers that are treated as atoms of the I/O model. These atoms can only be added up (using associativity and commutativity).
- An edge contributes precisely to the two vertex sketches of its endpoints. An edge is treated as an I/O atom. The I/O algorithm can read out the (polylog many) number atoms from an edge atom at no cost in internal memory (transformation). The difference between insert and delete is not visible to the I/O data structure—it merely adds up all the components of the sketches. This can easily be used to implement deletions by canceling contributions. The data structure is not required to check if the multiplicity of an edge is 1. The transformation function is available to the I/O algorithm at no cost.
- All vertex sketches have the following two-dimensional sparsity structure: The sketch consists of numbers organized in ρ rows and $\tau = \Theta(\log N)$ columns. The contribution of a single edge satisfies
  - The first column of each row always has a non-zero entry.
  - Each row starts with non-zero entries followed by zero entries.
  - The probability of a row having k non-zero entries is $(1/2)^k$. This is truncated at τ, if the row should be longer.

Encapsulating the nature of hash functions defining the contributions of an edge to a sketch, the above "magic expansion" of an edge atom into many number atoms is justified: while it is deterministic, to the I/O algorithm everything looks as if it is completely random and has no structure that can be exploited for improved efficiency.

Add edge (u, v) The item is a single atom of the I/O model.

Finalize The algorithm produces a sorted list of vertex sketches, each having ρ rows and τ columns.

The parameters of this interface are N, the number of add (insert/delete) operations; ρ, the number of rows in a vertex sketch; τ, the number of columns in a vertex sketch; $\phi = \rho\tau$, the number of atoms in a vertex sketch; and V, the number of vertices.

The upper bound is

$$O\Big(\min\big(N, \frac{N}{B}\log_{M/B}(V\phi/B)\big) + \text{scan}(N\phi/M) + \text{scan}(V\phi)\Big).$$

THEOREM 5.2. *Assume there is an implementation of the above interface. There exists a sequence of N Add operations followed by a Finalize such that the following number of I/Os are necessary:*

$$\Omega\left(\min\left(N, \frac{N}{B}\log_{M/B}(V\phi/B)\right) + \text{scan}(N\rho/M)\right)$$

The proof of this theorem is deferred to the full version of the paper.

Observe that for the case of $M \geq \phi$ (and the scanning of the sketch not dominating the algorithm), we have asymptotically matching upper and lower bounds. This is the case for the previously stated assumption $M = \Omega(\text{polylog}(V))$ of our hybrid graph streaming setting. Otherwise, in an extended parameter range of the I/O data structure, with the results presented so far, there is a $\Theta(\log N)$ gap between the upper and lower bounds. To almost close this gap, we can improve the upper bound by the following considerations. The following lemma follows by a straightforward union bound.

LEMMA 5.3. *Assume there are ρ random variables $X_i \geq 1$ with $p[X_i = j] = (1/2)^j$. Then the probability $p[\exists i : X_i > j] \leq \min(1, \rho(1/2)^j)$*

Now we split the data structure for the sketches by columns, $\tau' \geq 1$ columns per data structure, chosen such that $\tau'\rho \leq M$ (if possible, otherwise set $\tau' = 1$). An edge is always inserted into the data structure for the first columns, and also in all the data structures where one of its rows has a non-zero entry in one of the columns of the data structure. Hence, the expected contribution of an edge to the input stream of the k-th data structures is $\min(1, \rho(1/2)^{k\tau'})$, i.e., 1 for $k\tau' < \log\rho$ and then geometrically decreasing. This improves the $\text{scan}(\frac{N\phi}{M})$ term in the upper bound to $\text{scan}(\frac{N\rho\log\log N}{M})$. As long as ρ is polylogarithmic in N this leads to a gap of $O(\log\log N)$ between the upper and lower bounds. If $M = \Omega(\rho\log\log N)$, there is no asymptotic gap between the upper and lower bounds.

5.2 More External Semi-Streaming Algorithms Theorem 5.1 immediately implies efficient external semi-streaming algorithms for hypergraph connectivity (for bounded hyperedge cardinality r), bipartiteness testing, and $(1+\epsilon)$-approximating MST weight, all of which use $O(V\text{poly}(\log(V), \epsilon^{-1}, k, r))$ space and $\text{vsketch}(N, V, \text{poly}(\log(V), \epsilon^{-1}, k, r))$ I/Os.

Hypergraph connectivity. In followup work, Guha et al. [25] show that by using a slightly different vector encoding, their connectivity result can be extended to hypergraphs at the cost of a multiplicative $O(r^2)$ increase in the size of the sketch, where r is the maximum hyperedge cardinality. The remainder of the algorithm is essentially unchanged. We defer the proof of this corollary to the full version of the paper.

COROLLARY 5.4. *Given a hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with maximum edge cardinality r, there exists a $O(\text{vsketch}(rN, V, r^2\log^2 V) + \text{sort}(r^2 V\log^2 V))$ -I/O algorithm which computes a spanning forest of $\mathcal{G}$ w.h.p. and uses $O(r^2 V\log^2(V))$ space.*

To our knowledge, this is the first nontrivial external-memory algorithm for hypergraph connectivity.

Bipartiteness testing. We present an algorithm for testing whether a graph is bipartite. The result is immediate: Ahn et al. [4] reduce determining whether a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is bipartite to computing the number of connected components of a graph $D(\mathcal{G}) = (\mathcal{V}', \mathcal{E}')$ such that for each $v \in \mathcal{V}$ we add $v_1, v_2 \in \mathcal{V}'$ and for each edge $(u, v) \in \mathcal{E}$, we add two edges (u_1, v_2) and (u_2, v_1). This, combined with Lemmas 4.3 and 4.4, give the following theorem:

THEOREM 5.5. *There exists a $O(V\log^2(V))$-space, $\text{vsketch}(N, V, \log^2 V)$- I/O algorithm for bipartiteness testing that succeeds w.h.p.*

Approximating minimum spanning tree weight. Ahn et al. [4] show how to $(1+\epsilon)$ approximate the weight of the minimum spanning tree (MST) of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ by using their connected components (CC) sketches. For edge weights in the range $[W]$, they create $r = \log_{1+\epsilon}(W)$ CC sketches and use the ith to sketch $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}_i)$, where $\mathcal{E}_i = \{e \in \mathcal{E} : w(e) \leq (1+\epsilon)^i\}$, and $w(e)$ denotes the weight of edge e. They prove that

$$w(T) \leq n - (1+\epsilon)^r + \sum_{i=0}^{r}\sigma_i cc(\mathcal{G}_i) \leq (1+\epsilon)w(T),$$

where T denotes the minimum spanning tree of G, $cc(\mathcal{G}_i)$ denotes the number of connected components in G_i, and $\sigma_i = (1+\epsilon)^i - (1+\epsilon)^{i-1}$.

It suffices to find the number of connected components of each G_i. Constructing the

$O(\log(V))$ CC sketches via Theorem 5.1 uses $O(\text{vsketch}(N, V, \varepsilon^{-1}\log^2 V))$ I/Os, and reconstructing the spanning forests from each sketch takes $\text{sort}(V\log^2(V))$ I/Os by Lemma 4.4. This gives the following theorem:

THEOREM 5.6. *There exists a* $O(\epsilon^{-1}V\log^2(V))$*-space,* $O(\text{vsketch}(N, V, \varepsilon^{-1}\log^2 V) + \varepsilon^{-1}\text{sort}(V\log^2(V)))$ *I/O algorithm which* $(1+\epsilon)$*-approximates minimum spanning tree weight w.h.p.*

6 More Extraction Techniques for External Semi-Streaming Algorithms The algorithms described in the previous section rely on both the general transformation described in Theorem 5.1 and the procedure described in Lemma 4.4 that computes a spanning forest from the sketches after stream ingestion. In general, while Theorem 5.1 provides a way to perform stream ingestion I/O efficiently on any single-pass vertex-based sketch algorithm, the challenge of how to minimize the extraction cost remains open. Most graph sketch algorithms have a post-stream procedure for *querying* their sketch data structures to produce a sparse graph which retains (perhaps approximately) some property of the graph defined by the stream. We now present some external semi-streaming algorithms that demonstrate how to minimize both the I/O cost of querying the sketch to produce a sparsifier, and then computing the answer from that sparsifier. The proofs of the theorems in this section are deferred to the full version of the paper.

Testing k-edge-connectivity. We consider the problem of testing k-connectivity of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, or equivalently, exactly computing the minimum cut $\lambda(\mathcal{G})$ if $\lambda(\mathcal{G}) \leq k$.

We make use of the solution of Ahn et al. [4], which constructs a k-connectivity certificate $H = \bigcup_{i\in[k]} F_i$ where $F_0, F_1, \ldots, F_{k-1}$ are edge-disjoint spanning forests of $\mathcal{G}$. H has the property that it is k'-edge connected iff G is k'-edge connected for all $k' \leq k$. They find each F_i by computing a connectivity sketch $\mathcal{S}_i(\mathcal{G} \setminus \bigcup_{j<i} F_j)$. This is done in a single pass over the stream: during the stream, they keep k different sketches of $\mathcal{G}$: $\mathcal{S}_0(\mathcal{G}), \mathcal{S}_1(\mathcal{G}), \ldots, \mathcal{S}_{k-1}(\mathcal{G})$. We use $\mathcal{K}(\mathcal{G})$ to denote the concatenation of these k connectivity sketches. After the stream, $\mathcal{S}_0(\mathcal{G})$ is used to find F_0 and the edges of F_0 are deleted from the remaining $k-1$ connectivity sketches. $\mathcal{S}_1(\mathcal{G} \setminus F_0)$ can now be used to get F_1, whose edges are subsequently deleted from the remaining $k-2$ sketches and so on.

The extraction step of the above algorithm must access $\mathcal{K}(\mathcal{G})$ at least once so it has a trivial lower I/O bound of $\text{scan}(kV\log^2 V)$. Performing the spanning forest deletions naively requires a k factor I/O overhead: $\text{scan}(k^2V\log^2 V)$. Recall that we would like our algorithm to have an extraction cost not much higher than the cost of computing the property on a sparse graph in external memory, so we must reduce this overhead. The primary challenge to doing so is that the edges found in each spanning forest induce deletions to subsequent spanning forests, which necessitate many random accesses with different deadlines. We solve this issue by scheduling deletions for each spanning forest F_i carefully so that the deletion cost can be amortized over the cost of querying later sketches $S_j \forall j > i$, as described in the proof in the full version of the paper. This reduces the I/O overhead from k to $\log k$. Finally, we apply an exact min cut algorithm due to Geissman and Gianinazzi [23] to compute the minimum cut of the union of the k forests. This gives the following theorem:

THEOREM 6.1. *There exists an* $O(kV\log^2(V))$*-space,* $O(\text{vsketch}(N, V, k\log^2 V) + \text{ksketch}(V, k) + \text{sort}(kV)\log^4(V))$ *-I/O algorithm for testing k-edge connectivity that succeeds w.h.p., where*

$$\text{ksketch}(V,k) = \begin{cases} \text{scan} & (k\log(k)V\log^2(V)), \\ & \textit{when } k\log^2(V) = o(M) \\ \text{scan} & (k^2V\log^2(V)), \textit{otherwise.} \end{cases}$$

Approximating the minimum cut. Ahn et al. [2] provide a $O(\epsilon^{-2}V\log^4(V))$-space single-pass streaming algorithm for $(1+\epsilon)$-approximating the minimum cut. We summarize it here.

Define $\mathcal{G}_0 = \mathcal{G}$ and form $\mathcal{G}_i \subset \mathcal{G}_{i-1}$ by deleting each edge in $\mathcal{G}_{i-1}$ independently with probability $1/2$ for each $i \in [O(\log(V))]$. For each $\mathcal{G}_i$, construct a $k = O(\epsilon^{-2}\log(V))$-skeleton H_i using Ahn et al.'s k-connectivity algorithm [4]. The authors show that $\lambda(\mathcal{G}) \leq 2^j\lambda(H_j) \leq (1+\epsilon)\lambda(\mathcal{G})$ for $j = \min\{i : \lambda(H_i) < k\}$ where $\lambda(D)$ denotes the minimum cut of graph D. Therefore, returning $2^j\lambda(H_j)$ gives the desired approximation to $\lambda(\mathcal{G})$. Note that while the algorithm returns a vertex set S such that the cut $(S, \mathcal{V}\setminus S)$ has weight no more than $(1+\epsilon)\lambda(\mathcal{G})$, it cannot be used to recover the set of edges across $(S, \mathcal{V}\setminus S)$.

We can obtain a external semi-streaming algorithm by applying Theorem 5.1 and Theorem 6.1 to the above sketch, and then applying the external-memory exact min cut algorithm of Geissman and Gianinazzi [23] to find the minimum cuts of each (H_i). This gives the following theorem.

THEOREM 6.2. *There exists a external semi-streaming algorithm for* $(1+\epsilon)$*-approximating the minimum cut of* $\mathcal{G}$ *w.h.p. that uses* $O(\epsilon^{-2}V\log^4(V))$*-*

space and

$$O(\text{vsketch}(N, V, \epsilon^{-2}\log^4 V) + \text{loglog}V \cdot \text{ksketch}(V, \epsilon^{-2}\log(V)) + \log^4(V)\text{loglog}(V)\text{sort}(\epsilon^{-2}V\log(V)))$$

I/Os.

Returning the Edges Crossing a Minimum Cut In the external-memory model, where we retain access to all the edges in $\mathcal{G}$, we can recover all of the edges in the approximate minimum cut returned by the above algorithm. Let $(S, \mathcal{V} \setminus S)$ denote the $(1+\epsilon)$ minimum cut returned. Sort the nodes in S in increasing node ID order. Similarly, sort the list of edges $\mathcal{E}$ in the input graph $\mathcal{G}$ in increasing node ID order of the left endpoint, that is, each edge $e = (u, v)$ is sorted in increasing node ID order of u. Scan through the list of nodes in S and the list of edges simultaneously, marking each edge e s.t. $u \in S$. Next, sort the edge list in increasing node ID order of right endpoint, and mark each edge e s.t. $v \in S$ similarly. Finally, scan through the edge list and return each edge such that exactly one of its endpoints is in S. This gives the following corollary to Theorem 6.2:

COROLLARY 6.3. *There exists a*

$$O(\text{sort}(E) + \text{loglog}V \cdot \text{ksketch}(V, \epsilon^{-2}\log(V)) + \log^4(V)\text{loglog}(V)\text{sort}(\epsilon^{-2}V\log(V)))$$

-I/O external-memory algorithm that returns the edges of a cut $(S, \mathcal{V}\setminus S)$ that is at most $(1+\epsilon)$ times the weight of the minimum cut w.h.p.

Cut sparsifiers. We turn our attention to approximating *any* cut value in the graph. Specifically, the task is to find a ***$\varepsilon-$cut sparsifier*** $\mathcal{H}$ of $\mathcal{G}$, that is, a weighted subgraph $\mathcal{H} = (\mathcal{V}, \mathcal{E}', w)$ is an ε-cut sparsifier for $\mathcal{G}$ if $\forall S \subset \mathcal{V}$, $(1-\epsilon)\lambda_S(\mathcal{H}) \leq \lambda_S(\mathcal{G}) \leq (1+\epsilon)\lambda_S(\mathcal{H})$.

Ahnet al. [2] provide a semi-streaming algorithm for constructing a cut sparsifier. As in the algorithm for approximating the minimum cut, define $\mathcal{G}_0 = \mathcal{G}$, and then graphs $\mathcal{G}_i \subset \mathcal{G}_{i-1}$ are formed by deleting each edge in $\mathcal{G}_{i-1}$ independently with probability $1/2$, for each $i \in [O(\log(V))]$. For each such i, construct $\mathcal{H}_i$, a $k = O(\epsilon^{-2}\log^2(V))$-connectivity certificate of $\mathcal{G}_i$. Then a post-processing step decides for each edge $e \in \bigcup_i \mathcal{H}_i$ whether to add e to the sparsifier $\mathcal{H}$, as follows. For each e, compute $j(e) = \min\{i : \lambda_e(\mathcal{H}_i) < k\}$. Then e is added to $\mathcal{H}$ with weight $2^{j(e)}$ if and only if $e \in \mathcal{H}_j$. $\mathcal{H}$ is returned as the desired cut sparsifier.

We need an I/O efficient way to compute λ_e for all the edges in each $\mathcal{H}_i$. We make use of Laxhuber et al.'s ϵ-approximate max flow algorithm [30], which has I/O cost that is proportional to $E^{1+o(1)}$. By using sketching to sparsify the graph while preserving scaled cut values, we reduce both the number of max flow computations as well as their individual cost by reducing the number of edges by a $\widetilde{O}(V)$ factor.

THEOREM 6.4. *There exists a $O(\epsilon^{-2}V\log^5(V))$-space and $O(\text{vsketch}(N, V, \varepsilon^{-2}\log^5(V)) + \log\log(V)(\epsilon^{-10}V^2\log^{11}(V))^{1+o(1)}/B)$-I/O algorithm for constructing a $(1+\epsilon)$-cut sparsifier of a graph $\mathcal{G}$ w.h.p.*

Once we have the cut sparsifier, we can use it to approximately answer s-t min cut queries with Laxhuber's max flow algorithm [30]. If we want a $(1+\varepsilon)$ approximation overall, we must set $\varepsilon' = \sqrt{1+\varepsilon} - 1 = O(\varepsilon^{1/2})$ for both the cut sparsifier algorithm and the max flow algorithm. This yields the following corollary:

COROLLARY 6.5. *There exists an algorithm to find x different $s-t$ min cuts on a graph $\mathcal{G}$ w.h.p. using*

$$O(\text{vsketch}(N, V, \varepsilon^{-4}\log^5(V)) + \log\log(V)\epsilon^{-20}\log^{11}V)\text{scan}(V^2) + x\varepsilon^{-6}\text{scan}(\varepsilon^{-4}V\log^3(V))$$

I/Os.

Densest subgraph. McGregor et al. [34] give an algorithm for $(1+\epsilon)$-approximating the density $d^*(\mathcal{G})$ of the densest subgraph of graph $\mathcal{G}$. The main idea is to create a subgraph $\mathcal{H}$, which subsamples each edge in $\mathcal{G}$ independently with probability $p = \frac{V\log V}{\epsilon^2 E}$, despite the fact that true value of E (and therefore p) is not known until the end of the stream. They show that $\frac{1}{p}d^*(\mathcal{H})$ approximates $d^*(\mathcal{G})$ to within a factor of $(1+\epsilon)$ with high probability. The density of the densest subgraph of $\mathcal{H}$ is computed by a black-box algorithm.

The following is performed $O(\log V)$ times independently in parallel. Before the stream, partition the potential edges of the graph into $\Theta(\epsilon^{-2}V)$ buckets using pairwise independent hash functions. Insert arriving edges into the $O(\log V)$ ℓ_0-sketches corresponding to their bucket.

In the post-processing step, compute p based on the final number of edges E that are present in the graph. Then simulate sampling each edge independently with probability p as follows. For the ith bucket in the jth partition, randomly draw $X_{ij} \sim Bin(E_{ij}, p)$, where E_{ij} is the number of edges present in the bucket at the end of the stream. Then, select X_{ij} edges uniformly without replacement by querying each of the first X_{ij} of the bucket's sketches in sequence to produce one edge each, where any queried edges are deleted from all subsequent sketches before querying the next sketch. This is performed only for buckets that are 'small', i.e.,

those that contain at most $4\epsilon^2 E/V$ edges. The parallel partitions are performed to ensure that every edge is in some small bucket with high probability. Finally, the union of the queried edges from makes up $\mathcal{H}$.

The above algorithm is not a vertex-based algorithm. However, we show below that it is possible to partition the edges once using a $\Theta(\log V)$-wise independent hash function such that every bucket is small with high probability. Now we maintain $O(\log^2 V)$ ℓ_0-sketches for each bucket. This ensures that edges only have to be added to the sketches for their one corresponding bucket, which can be stored contiguously. This allows us to apply Theorem 5.1.

To compute the densest subgraph in $\mathcal{H}$, we use Charikar's greedy peeling algorithm [13]. The algorithm iteratively removes the lowest degree vertex from the graph, as well as all incident edges, to produce a set of induced subgraphs down to a singleton vertex. The algorithm returns the densest of these subgraphs. This provides a 2-approximation to the densest subgraph, resulting in a $2(1+\epsilon)$-approximation overall.

THEOREM 6.6. *For a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\epsilon > 0$ with $\epsilon^2 E/V = \Omega(\log V)$, there exists a $O(\epsilon^{-2} V \log^3 V)$-space and $O(\text{vsketch}(N, \epsilon^{-2}V, \log^3 V) + V\,\text{sort}(\epsilon^{-2} V \log^2 V))$-I/O algorithm for $2(1+\epsilon)$ approximating the density of the densest subgraph of a $\mathcal{G}$ w.h.p.*

7 Conclusion In this paper we introduce the external semi-streaming model, which combines the stream input and limited space of the semi-streaming model with the block-access constraint of the external-memory model.

We present a general transformation from any vertex-based sketch algorithm in the semi-streaming model to one which a low sketching cost in the external semi-streaming model. We complement this transformation with a I/O lower bound for sketching the input stream. For some algorithms, these bounds are tight; for others there is a $O(\log V)$ gap.

We present several techniques for minimizing the extraction cost. We show how to I/O-efficiently extract many mutually edge-disjoint spanning forests from a k-connectivity, min cut, or cut sparsifier sketch. We also present new external-memory graph algorithms for densest subgraph and cut sparsification. These algorithms have low I/O complexity on sparse graphs and high I/O complexity on dense graphs, but because we sparsify the input graph via sketching, our result is an algorithm that has low I/O complexity on any graph regardless of density.

Putting these techniques together, we present external semi-streaming algorithms for connectivity, hypergraph connectivity, minimum cut, cut sparsification, bipartiteness testing, minimum spanning tree, and densest subgraph. For many of these problems, our external semi-streaming algorithms outperform the state of the art in sketching and external-memory graph algorithms.

The field of semi-streaming has had the problem that the algorithms developed in the model are generally too big for RAM and too random for SSD. This barrier prevents most graph sketch algorithms from running on today's hardware, making them useless for any reasonable application. External semi-streaming algorithms get around this barrier because they are small enough to store on SSD and they make mostly sequential accesses. So instead of having to wait decades for these algorithms to be useful, we may be able to make use of graph sketching on today's hardware.

Given the results in this paper, we believe that I/O complexity should be treated as a first-class citizen in the design and analysis of semi-streaming algorithms. The transformations in this paper for vertex-based sketches makes it more feasible, and in some cases even trivial, to design external semi-streaming algorithms. We also believe that sketching is a powerful technique for designing external-memory graph algorithms, even outside of a streaming setting.

Finally we note that the fields of external memory and semi-streaming have been parallel but separate ways of dealing with massive data. In this paper we show that each field can contribute to the other in important ways.

References

[1] P. K. AGARWAL, L. ARGE, AND K. YI, *I/o-efficient batched union-find and its applications to terrain analysis*, in Proceedings of the 22nd Annual ACM Symposium on Computational Geometry (SoCG), ACM, 2006, pp. 167–176, https://doi.org/10.1145/1137856.1137884.

[2] K. AHN, S. GUHA, AND A. MCGREGOR, *Graph sketches: Sparsification, spanners, and subgraphs*, in Proceedings of the 23rd ACM Symposium on Principles of Database Systems (PODS), 03 2012, https://doi.org/10.1145/2213556.2213560.

[3] K. J. AHN, G. CORMODE, S. GUHA, A. MCGREGOR, AND A. WIRTH, *Correlation clustering in data streams*, in Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37, ICML'15, JMLR.org, 2015, pp. 2237–2246.

[4] K. J. AHN, S. GUHA, AND A. MCGREGOR, *Analyzing graph structure via linear measurements*, in Proceedings of the 23rd Annual ACM-SIAM

Symposium on Discrete Algorithms (SODA), 2012, pp. 459–467.

[5] K. J. Ahn, S. Guha, and A. McGregor, *Spectral sparsification in dynamic graph streams*, in International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX), vol. 8096, 2013, pp. 1–10.

[6] S. Assadi, S. Khanna, and Y. Li, *Tight bounds for single-pass streaming complexity of the set cover problem*, in STOC, ACM, 2016, pp. 698–711.

[7] S. Assadi, S. Khanna, and Y. Li, *Tight bounds for single-pass streaming complexity of the set cover problem*, in Proceedings of the forty-eighth annual ACM symposium on Theory of Computing, 2016, pp. 698–711.

[8] S. Assadi and V. Shah, *Tight bounds for vertex connectivity in dynamic streams*, in Symposium on Simplicity in Algorithms (SOSA), SIAM, 2023, pp. 213–227.

[9] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich, *Deterministic sampling and range counting in geometric data streams*, in Proceedings of the 20th ACM Symposium on Computational Geometry (SoCG)„ ACM, 2004, pp. 144–151, https://doi.org/10.1145/997817.997842.

[10] M. Bateni, H. Esfandiari, and V. Mirrokni, *Almost optimal streaming algorithms for coverage problems*, in Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, 2017, pp. 13–23.

[11] T. Y. Berger-Wolf and J. Saia, *A framework for analysis of dynamic social networks*, in Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '06, New York, NY, USA, 2006, Association for Computing Machinery, pp. 523–528, https://doi.org/10.1145/1150402.1150462.

[12] A. Bishnu, A. Ghosh, G. Mishra, and S. Sen, *On the streaming complexity of fundamental geometric problems*, CoRR, abs/1803.06875 (2018), https://arxiv.org/abs/1803.06875.

[13] M. Charikar, *Greedy approximation algorithms for finding dense components in a graph*, in Proceedings of the Third Annual Approximation Algorithms for Combinatorial Optimization (APPROX), vol. 1913, Springer, 2000, pp. 84–95, https://doi.org/10.1007/3-540-44436-X_10.

[14] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter, *External-memory graph algorithms*, in Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1995, pp. 139–149.

[15] R. Chitnis, G. Cormode, H. Esfandiari, M. Hajiaghayi, A. McGregor, M. Monemizadeh, and S. Vorotnikova, *Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams*, in SODA, SIAM, 2016, pp. 1326–1344.

[16] R. Chitnis, G. Cormode, M. T. Hajiaghayi, and M. Monemizadeh, *Parameterized streaming: Maximal matching and vertex cover*, in Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2014, pp. 1234–1251.

[17] M. S. Crouch, A. McGregor, and D. Stubbs, *Dynamic graphs in the sliding-window model*, in Proceedings of the 21st Annual European Symposium on Algorithms (ESA), 2013, pp. 337–348, https://doi.org/10.1007/978-3-642-40450-4_29.

[18] A. Czumaj, S. H. Jiang, R. Krauthgamer, and P. Veselý, *Streaming algorithms for geometric steiner forest*, in 49th International Colloquium on Automata, Languages, and Programming, ICALP, vol. 229 of LIPIcs, 2022, pp. 47:1–47:20, https://doi.org/10.4230/LIPICS.ICALP.2022.47.

[19] Y. Emek and A. Rosén, *Semi-streaming set cover*, ACM Transactions on Algorithms (TALG), 13 (2016), pp. 1–22.

[20] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, *A density-based algorithm for discovering clusters in large spatial databases with noise*, AAAI Press, 1996, pp. 226–231.

[21] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, *On graph problems in a semi-streaming model*, Theoretical Computer Science, 348 (2005), pp. 207–216, https://doi.org/10.1016/j.tcs.2005.09.013.

[22] G. Frahling and C. Sohler, *Coresets in dynamic geometric data streams*, in Proceedings of the 37th Annual ACM Symposium on Theory of Computing, (STOC), ACM, 2005, pp. 209–217, https://doi.org/10.1145/1060590.1060622.

[23] B. Geissmann and L. Gianinazzi, *Cache oblivious minimum cut*, in 7th International Conference on Algorithms and Complexity (CIAM), 2017, pp. 285–296.

[24] E. Georganas, R. Egan, S. Hofmeyr, E. Goltsman, B. Arndt, A. Tritt, A. Buluç, L. Oliker, and K. Yelick, *Extreme scale de novo metagenome assembly*, SC '18, IEEE Press, 2018.

[25] S. Guha, A. McGregor, and D. Tench, *Vertex and hyperedge connectivity in dynamic graph streams*, in Proceedings of the 34th Annual ACM Symposium on Principles of Database Systems (PODS), ACM, 2015, pp. 241–247.

[26] S. Har-Peled, P. Indyk, S. Mahabadi, and A. Vakilian, *Towards Tight Bounds for the Streaming Set Cover Problem*, arXiv e-prints, (2015), arXiv:1509.00118, p. arXiv:1509.00118, https://arxiv.org/abs/1509.00118.

[27] P. Indyk, *Streaming algorithms for geometric problems*, in FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference Proceedings, vol. 3328, Springer, 2004, pp. 32–34, https://doi.org/10.1007/978-3-540-30538-5_3.

[28] M. Kapralov, Y. T. Lee, C. Musco, C. Musco, and A. Sidford, *Single pass spectral sparsification in dynamic streams*, in Proceedings of the 55th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2014, pp. 561–570.

[29] M. Kapralov and D. P. Woodruff, *Spanners and sparsifiers in dynamic streams*, in Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC), ACM, 2014, pp. 272–281.

[30] H. Laxhuber, *A cache-efficient (1-epsilon)-approximation algorithm for the maximum flow problem on undirected graphs*, bachelor thesis, ETH Zurich, Zurich, 2021, https://spcl.inf.ethz.ch/Publications/.pdf/2021_co_maxflow_thesis.pdf.

[31] W. Lee, J. J. Lee, and J. Kim, *Social network community detection using strongly connected components*, in Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2014, pp. 596–604.

[32] Y. Li, H. L. Nguyen, and D. P. Woodruff, *Turnstile streaming algorithms might as well be linear sketches*, in Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC), Association for Computing Machinery, 2014, p. 174–183, https://doi.org/10.1145/2591796.2591812.

[33] A. McGregor, *Graph stream algorithms: a survey*, ACM SIGMOD Record, 43 (2014), pp. 9–20.

[34] A. McGregor, D. Tench, S. Vorotnikova, and H. T. Vu, *Densest subgraph in dynamic graph streams*, in Proceedings of the 40th Mathematical Foundations of Computer Science (MFCS), Berlin, Heidelberg, 2015, pp. 472–482.

[35] A. McGregor, D. Tench, and H. T. Vu, *Maximum coverage in the data stream model: Parameterized and generalized*, in 24th International Conference on Database Theory (ICDT), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[36] A. McGregor, S. Vorotnikova, and H. T. Vu, *Better algorithms for counting triangles in data streams*, in Proceedings of the 35th Annual ACM Symposium on Principles of Database Systems (PODS), ACM, 2016, pp. 401–411.

[37] A. McGregor and H. T. Vu, *Evaluating bayesian networks via data streams*, in Computing and Combinatorics, Cham, 2015, pp. 731–743.

[38] A. McGregor and H. T. Vu, *Better streaming algorithms for the maximum coverage problem*, Theory of Computing Systems, 63 (2019), pp. 1595–1619.

[39] S. Muthukrishnan, *Data streams: Algorithms and applications*, Foundations and Trends® in Theoretical Computer Science (FnTs), 1 (2005), pp. 117–236, https://doi.org/10.1561/0400000002.

[40] J. Nelson and H. Yu, *Optimal lower bounds for distributed and streaming spanning forest computation*, in Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), SIAM, 2019, pp. 1844–1860.

[41] J. Nešetřil, E. Milková, and H. Nešetřilová, *Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history*, Discrete Mathematics, 233 (2001), pp. 3–36, https://doi.org/10.1016/S0012-365X(00)00224-7.

[42] S. Nurk, D. Meleshko, A. Korobeynikov, and P. Pevzner, *Metaspades: A new versatile metagenomic assembler*, Genome Research, 27 (2017), https://doi.org/10.1101/gr.213959.116.

[43] R. Pagh and C. E. Tsourakakis, *Colorful triangle counting and a mapreduce implementation*, Information Processing Letters (IPL), 112 (2012), pp. 277–281.

[44] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, *The ubiquity of large graphs and surprising challenges of graph processing*, Proc. VLDB Endow., 11 (2017), pp. 420–431, https://doi.org/10.1145/3186728.3164139.

[45] D. Tench, E. West, V. Zhang, M. A. Bender, A. Chowdhury, J. A. Dellas, M. Farach-Colton, T. Seip, and K. Zhang, *Graphzeppelin: Storage-friendly sketching for connected components on dynamic graph streams*, in Proceedings of the 2022 International Conference on Management of Data, SIGMOD, ACM, 2022, p. 325–339, https://doi.org/10.1145/3514221.3526146.

[46] J. S. Vitter, *External memory algorithms and data structures: dealing with massive data*, ACM Computing Surveys (CsUR), 33 (2001).

[47] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin, *Efficient structural graph clustering: An index-based approach*, The Very Large Data Base Endowment Journal (VLDB), 28 (2019), pp. 377–399, https://doi.org/10.1007/s00778-019-00541-4.

[48] D. Wen, L. Qin, Y. Zhang, L. Chang, and X. Lin, *Efficient structural graph clustering: An index-based approach*, The Very Large Data Base Endowment Journal (VLDB), 28 (2019), pp. 377–399, https://doi.org/10.1007/s00778-019-00541-4.

[49] D. P. Woodruff and T. Yasuda, *High-dimensional geometric streaming in polynomial space*, in 63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS, IEEE, 2022, pp. 732–743, https://doi.org/10.1109/FOCS54457.2022.00075.