

The Case for External Graph Sketching

Michael A. Bender

Stony Brook University and RelationalAI

Martín Farach-Colton

New York University

Riko Jacob

IT University of Copenhagen

Hanna Komlós

New York University

David Tench

Lawrence Berkeley National Laboratory

Evan T. West

Stony Brook University

Algorithms in the data stream model use $O(\text{polylog}(N))$ 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 $O(V \text{polylog}(V))$ 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 $O(V \text{polylog}(V))$ 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 $O(V \text{poly}(\log(V), \epsilon^{-1}, k))$ 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.

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 [28, 53] 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 [3–6, 10, 12, 21, 22, 24, 28, 32, 36, 37, 42, 44, 46, 47, 53, 54, 57], as well as computational geometric problems [13, 17, 25, 29, 35, 65], and hypergraph problems [11, 14, 26, 32, 33, 45, 48]. The semi-streaming model is elegant and captures something exciting about the structure of graph problems and their algorithms.

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 [42].

There is a corresponding need in industry and large-scale science to process massive, dynamic graphs. A recent survey by Sahu et al. [58] 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 [31, 56], large-scale clustering [27, 63, 64], and tracking social network communities that change as users add or delete friends [16, 40].

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 we illustrate this problem with a case study on the k -connectivity sketch of Ahn et al. [5], 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 appendix that due to a lower bound of Nelson and Yu [54], 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 [5]. 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.

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 [62]. 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 disk 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 updates and computing solutions to queries) in addition

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

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 [5, 12, 32, 36, 43], 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 detailed 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 [5, 12, 32, 36, 43] 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 ε -approximate min cut also outperform the best existing algorithms for these problems for some parameter settings (see Section 3 for details).

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 [28, 53] (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 and -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, i.e., $w(e) = 1$ for all updates and in these cases we omit $w(e)$ in the update notation. Let s_i denote the i th 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 [5, 12, 32, 36, 43] called vertex-based sketches.

DEFINITION 1. (VERTEX-BASED SKETCH [32]) *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 [62], 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 [20].

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 1, we summarize the space and I/O bounds for the external semi-streaming algorithms we present in this paper. All of our sketches are vertex-based (see Definition 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 2 compares these bounds to those of the 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 Table 3 in

Algorithm	Vertex Sketch Size	I/O Cost
Connected Comp.	$\phi_1 = O(\log^2 V)$	$O(\text{permute}(N) + \text{sort}(V\phi_1))$
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	$\phi_4 = O(\varepsilon^{-1} \log^2 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_4/M) + \text{sort}(V) \cdot \text{poly}(\varepsilon^{-1}, \log V))$
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	$\phi_6 = O(\varepsilon^{-2} \log^4 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_6/M) + \text{sort}(V) \cdot \text{poly}(\varepsilon^{-1}, \log V))$
ε -Cut Sparsifier	$\phi_7 = O(\varepsilon^{-2} \log^5 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_7/M) + \text{scan}(V^{2+o(1)}) \cdot \text{poly}(\varepsilon^{-1}, \log V))$
$2(1 + \varepsilon)$ -Appx Densest Subgraph	$\phi_8 = O(\varepsilon^{-2} \log^2 V)$	$O(\text{permute}(N) + \text{scan}(N\phi_8/M) + \text{sort}(V^2) \cdot \text{poly}(\varepsilon^{-1}, \log V))$

Table 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
$2(1 + \varepsilon)$ -Approximate Densest Subgraph*	Better	Same	First	First

Table 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.

Appendix C.

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 ($\text{permute}(N)$ vs. $\text{sort}(N)$) the I/O costs of the best known algorithms. For approx-

imate 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 ε -approximate min cut, our algorithm performs better than the best (exact) EM algorithm if $\varepsilon = \Omega\left(\max\left(M^{-1/2} \log^{-1/2} V, (V/E)^{1/4} \log \log^{1/4} V\right)\right)$.

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. [5] give a $O(V \log^2 V)$ -space sketching algorithm to solve semi-streaming connectivity and, later, Nelson and Yu [54] prove that this space cost is optimal. Subsequent work by Tench

et al. [61] 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 [5]:

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 2. *A linear measurement of a graph on n vertices is defined by a set of coefficients $\{ce : e \in \binom{V}{2}\}$. Given a graph $\mathcal{G} = (V, \mathcal{E})$, the evaluation of this measurement is defined as $\sum_{e \in \mathcal{E}} ce$. 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 ce is $\text{poly}(n)$.*

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. [42] 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 $\mathcal{S}^0(\mathcal{G}), \mathcal{S}^1(\mathcal{G}), \dots, \mathcal{S}^{V-1}(\mathcal{G})$ 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 $u \in V$. $\mathcal{S}(\mathcal{G})$ 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 $(A, V \setminus A)$. After the stream, the algorithm finds the connected components using Boruvka’s algorithm [55], 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 i th 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 for a detailed discussion.

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. [61] 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 $\mathcal{U} \subset V$ 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.

LEMMA 4.1. 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*.

Proof. EXTskETCHCC's sketch data structures and update buffers use $O(V \log^2(V))$ space. We also store at most $V \log^2 V$ pending updates from a batch on disk at a time.

Now we analyze the I/O cost of EXTskETCHCC's ingestion procedure. Let \mathcal{B} be a batch of updates. Collecting the updates for \mathcal{B} and writing them to disk costs $\text{scan}(V \log^2 V)$ I/Os in total. Permuting the updates costs $\text{permute}(V \log^2 V) = \min(V \log^2 V, \text{sort}(V \log^2 V))$ I/Os. Applying the updates to the sketches costs $O(\text{scan}(V \log^2 V))$ I/Os, which is dominated by the permute cost unless $N < V \log^2 V$.

There are $\frac{N}{2V \log^2 V}$ batches, so the overall I/O complexity of ingesting the stream is $O\left(\frac{N}{V \log^2 V} \text{permute}(V \log^2 V)\right) = O\left(\min\left(N, \frac{N}{B} \log_{M/B}((V/B) \log^2 V)\right)\right)$.

Finally, we have to address the case where $N < V \log^2 V$. In this case, still need to scan through the sketches after the permute operation, so the minimum I/O cost is $\text{scan}(V \log^2 V)$. \square

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 consisting of 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. [2] 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 $u \rightarrow v$, 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 $u \rightarrow v$. 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. [20] to compute these connected components. We then replace the merges in the list with merges of the form $u \rightarrow v'$, 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 naively 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 destina-

tion, 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.

LEMMA 4.2. *Once all stream updates have been processed, EXTSKETCHCC computes connected components using $O(\text{sort}(V \log^2(V)))$ I/Os.*

Proof. We analyze the I/O cost of each phase of a round separately, as the total cost is a constant factor greater than the cost of performing the first round. This is because the number of components we operate on, and thus the amount of I/Os we perform, decreases by a constant factor in each round.

To query the sketches in the first phase, each sketch must be read into RAM, queried, and the result appended to a list. This can be done with a single scan using $O(\text{scan}(V \log^2(V)))$ I/Os to produce a list of $O(V)$ edges.

Finding the supernodes corresponding to each edge in the merge list consists of a scan and a sort of data of size $O(V)$ per iteration, with $\alpha(V)$ iterations. This gives $O(\text{sort}(V)\alpha(V))$ I/Os in total to find the supernodes. Since the graph \mathcal{H} has $O(V)$ edges, computing its connected components takes $O(\text{sort}(V))$ I/Os [20]. Performing the queries in the merge list of size $O(V)$ in the union-find data structure takes $O(\text{sort}(V))$ I/Os [2].

In the third phase, if sketches are larger than the block size, the cost of summing the sketches is the scan cost of $O(V \log^2(V)/B)$.

If sketches are smaller than blocks, we sort the merge list in $O(\text{sort}(V))$ I/Os, then sort the sketches in the same order in $O(\text{sort}(V \log^2(V)))$ I/Os. The simultaneous scan through the sketches and merge list is a low order cost to the sort operations. Finally, sorting the sketches by their marked merge destinations costs $\text{scan}(V \log^2 V) + \text{sort}(V) = \text{scan}(V \log^2 V)$. I/Os, after which this round is completed with one additional scan.

The cost to run our entire query only a constant factor greater than the cost of these three phases or $O(\text{scan}(V \log^2 V)) + O(\text{sort}(V)\alpha(V)) + O(\text{sort}(V \log^2 V)) = O(\text{sort}(V \log^2(V)))$. \square

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 [20], and uses $\tilde{O}(V)$ less space.

COROLLARY 4.1. *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*

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

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 4.1, we have an I/O cost of $\text{scan}(V\phi)$ in the case that $N < V\phi$. \square

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 argue for 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

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

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

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

Proof. The first lower bound is justified by a reduction of sparse matrix multiply. Here we assume that the adversary can choose the numbers that are inserted into the sketches arbitrarily (by choosing an appropriate hash function). Let A be a sparse matrix with $2N$ columns and ρV rows, where each column has precisely one entry that is given in column major layout. Now, the above data structure can be used to compute Ax by scanning over x and A in its layout and inserting items. Assume the entry is in row i , calculate $k = i/\rho$ and $h = i \bmod \rho$. Create an edge that involves vertex k and let the entry of the vertex sketch in row h and in the first column be the only non-zero entry associated with this endpoint of the edge. Set the value of this entry to $a_{ij}x_j$. Each consecutive pair of these are leading to an add operation. The Finalize operation creates the sum of the basic sketches, from whom we can read out the vector y in the first columns of the vertex sketches. Hence, the lower bound of [15] applies and justifies the term.

The second line is a lower bound following from a volume consideration in Hong and Kung rounds: It is well known [15, 34] that there is no asymptotic penalty for assuming that an I/O program operates in rounds of M/B I/Os, where first the memory is loaded, then computation happens, and finally the memory is written to disk. This can simulate any I/O program with twice the memory size and at most twice the number of I/Os. In this normalized setting, it is clear what we mean by tracing the original atoms (including making copies),

and atoms that are part of the final output (including reduction/summation steps). Obviously, these traces must meet for output atoms that depend on input items, and the “transformation” from an input item to an output atom must happen in some Hong and Kung round. In each such round, there are at most M input items loaded, and at most M output atoms are written out, so at most M^2 different useful transformations can happen per round. There is a need for a total of at least $2N\rho$ transformations, leading to a lower bound of $2N \cdot \rho/M^2$ Hong and Kung rounds, which is equivalent to a lower bound of $M/B \cdot N \cdot \rho/M^2 = \text{scan}(N\rho/M)$ I/Os, as written in the second term. \square

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 upper and lower bounds. To almost close this gap, we can improve the upper bound by the following considerations.

LEMMA 5.1. *Assume there are ρ random variables $X_i \geq 1$ with $p[X_i = j] = (1/2)^j$. Then the probability $p[\exists i : X_i > j] \leq \min(1, \rho(1/2)^j)$*

Proof. By a union bound. \square

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 upper and lower bound. If $M = \Omega(\rho \log \log N)$, there is no asymptotic gap between upper and lower bound.

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 $(1 + \varepsilon)$ -approximating MST weight, all of which use $O(V \text{poly}(\log(V), \varepsilon^{-1}, k, r))$ space and $\text{vsketch}(N, V, \text{poly}(\log(V), \varepsilon^{-1}, k, r))$ I/Os.

Hypergraph connectivity. In followup work, Guha et al. [32] 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.

COROLLARY 5.1. *Given a hypergraph $\mathcal{G} = (V, \mathcal{E})$ 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.*

Proof. The algorithm is only slightly different than the algorithm for connected components on (non-hyper-) graphs. There are $O(V^r)$ possible edges in a hypergraph with maximum edge cardinality r , so we may encode all of them in a characteristic vector with length V^r . Then $\phi = r^2 \log^2 V$. [32]. Since each update needs to be applied to at most r vertex sketches, we make at most r copies of each update, and apply the ingestion procedure from Theorem 5.1 to a stream that is now of length rN . This gives an ingestion cost of $\text{vsketch}(rN, V, r^2 \log^2 V)$. The total space for the sketch is $O(r^2 V \log^2 V)$.

The procedure for computing a spanning hypergraph from the sketch again is similar to connectivity on graphs. The first phase of a round is identical, requiring a scan over the sketches for $\text{scan}(r^2 V \log^2(V))$ I/Os. Since a hypergraph edge may require up to $r - 1$ union-find updates, the cost of finding the components of each endpoint of each edge and merging them in the union find data structure is $\text{sort}(rV) \alpha(V)$. The mechanics of the third phase are unchanged, so the cost is $\text{sort}(r^2 V \log^2(V))$. So the total cost of the first round is $O(\text{sort}(r^2 V \log^2(V)))$. \square

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. [5] reduce determining whether a graph $\mathcal{G} = (V, \mathcal{E})$ is bipartite to computing the number of connected components of a graph $D(\mathcal{G}) = (V', \mathcal{E}')$ such that for each $v \in V$ we add $v_1, v_2 \in V'$ 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 4.1 and 4.2, give the following theorem:

THEOREM 5.3. *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. [5] show how to $(1 + \varepsilon)$ 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 $r = \log_{1+\varepsilon}(W)$ CC sketches and use the i th to sketch $\mathcal{G}_i = (\mathcal{V}, \mathcal{E}_i)$, where $\mathcal{E}_i = \{e \in \mathcal{E} : w(e) \leq (1 + \varepsilon)^i\}$, and $w(e)$ denotes the weight of edge e . They prove that

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

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

It suffices to find the number of connected components of each \mathcal{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 4.2. This gives the following theorem:

THEOREM 5.4. *There exists a $O(\varepsilon^{-1} V \log^2(V))$ -space, $O(\text{vsketch}(N, V, \varepsilon^{-1} \log^2 V) + \varepsilon^{-1} \text{sort}(V \log^2(V)))$ I/O algorithm which $(1 + \varepsilon)$ -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 4.2 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.

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. [5], which constructs a k-connectivity certificate $H = \bigcup_{i \in [k]} F_i$ where F_0, F_1, \dots, F_{k-1} are edge-disjoint span-

ning forests of \mathcal{G} . H has the property that it is k' -edge connected iff \mathcal{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} : $\mathcal{S}_0(\mathcal{G}), \mathcal{S}_1(\mathcal{G}), \dots, \mathcal{S}_{k-1}(\mathcal{G})$. 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: $\text{scan}(k^2 V \log^2 V)$. 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 $\mathcal{S}_j \forall j > i$. This reduces the I/O overhead from k to $\log k$. Finally, we apply an exact min cut algorithm due to Geissman and Gianinazzi [30] 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, $O(\text{vsketch}(N, V, k \log^2 V) + \text{k sketch}(V, k) + \text{sort}(kV \log^4(V)))$ I/O algorithm for testing k-edge connectivity that succeeds w.h.p., where*

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

Proof. Constructing $\mathcal{K}(\mathcal{G}) = \mathcal{S}_0(\mathcal{G}), \mathcal{S}_1(\mathcal{G}), \dots, \mathcal{S}_{k-1}(\mathcal{G})$ takes $O(\text{vsketch}(N, V, k \log^2 V))$ I/Os by Theorem 5.1. We will now analyze the cost of forming H from the sketches, and computing the minimum cut of H , in more detail.

Computing F_i from $\mathcal{S}_i(\mathcal{G} \setminus \bigcup_{j < i} F_j)$ takes $\text{sort}(V \log^2(V))$ I/Os for each $0 \leq i < k$ by Lemma 4.2. When computing F_0 , we already have $\mathcal{S}_0(\mathcal{G})$, so this is the only cost. Deleting the edges in F_0 from $\mathcal{S}_i(\mathcal{G})$ for all $0 < i < k$ takes $O(kV \log^2(V)/B)$ I/Os. Repeating this process to create each $\mathcal{S}_i(\mathcal{G} \setminus \bigcup_{j < i} F_j)$

therefore takes $O(k^2 V \log^2(V)/B)$ I/Os, but a more careful method allows us to reduce this cost when k is small enough that a vertex sketch fits in RAM. For the remainder of the proof, we assume $\phi = k \log^2(V) = o(M)$.

Let us focus on the cost of forming $\mathcal{S}_i(\mathcal{G} \setminus \bigcup_{j < i} F_j)$ for some i . Since we begin knowing $\mathcal{S}_i(\mathcal{G})$, this cost is exactly the cost of deleting $\bigcup_{j < i} F_j$, the edges in the previous $i - 1$ spanning forests, from $\mathcal{S}_i(\mathcal{G})$. For any $i < \log^2(V)$ the total cost to form $\mathcal{S}_i(\mathcal{G} \setminus \bigcup_{j < i} F_j)$ is less than $\text{scan}(V \log^2(V))$ because both $\mathcal{S}_i(\mathcal{G})$ and the list of deletions $\bigcup_{j < i} F_j$ have size $O(V \log^2(V))$.

Let $F'_i = \bigcup_{j=i \log^2(V)}^{(i+1) \log^2(V)-1} F_j$ for $0 \leq i < k/\log(V)$ and similarly let $\mathcal{S}'_i(\mathcal{G} \setminus \bigcup_{j < i} F'_j) = \bigcup_{j=i \log^2(V)}^{(i+1) \log^2(V)-1} \mathcal{S}_j(\mathcal{G} \setminus \bigcup_{j' < j} F'_{j'})$. The cost of applying deletions to sketches while producing F'_0 is at most $\eta/2$ where $\eta = \text{scan}(V \log^4(V))$ by Gauss summation. Similarly, the cost of the deletions for F'_1 is $3\eta/2$. Of this total, η is from deleting F'_0 from each of the $\log^2(V)$ new sketches and $\eta/2$ is from deleting the new spanning forests as they are built from all remaining sketches.

When we are ready to compute F'_i , let $\psi = \text{argmax}_j \{i = 0 \bmod 2^j\}$. We delete $\bigcup_{j=i-2^\psi}^{i-1} F'_j$ from $\bigcup_{i=2^\psi}^{i+2^\psi-1} \mathcal{S}'_i(\mathcal{G})$. For example, we begin computing F'_2 by deleting $F'_0 \cup F'_1$ both from $\mathcal{S}_2(\mathcal{G})$ and from $\mathcal{S}_3(\mathcal{G})$. Later, when we want to extract F'_3 from $\mathcal{S}_3(\mathcal{G})$, we need only delete F'_2 from it because the other deletions have already been done. Figure 1 summarizes this deletion schedule.

Now we can analyze the I/O cost for each deletion. For simplicity we will overestimate: Round up $k' = k/\log^2(V)$ to the next power of 2. Each F'_i pays an $\eta/2$ I/O cost for its local deletions, for a total of $k'\eta/2$. Now consider the block deletions. There is one block deletion of size $(k'\eta/2)$ for $F'_{k'/2}$, two deletions of size $(k'\eta/4)$ for $F'_{k'/4}$ and $F'_{3k'/4}$, etc. The total cost for these deletions is therefore

$$\begin{aligned} & \sum_{j < \log(k')} k' 2^j 2^{-(j+1)} \eta \\ &= \frac{k'\eta}{2} \log(k') \\ &= \eta \left(\frac{k}{2 \log^2(V)} \log \left(\frac{k}{\log^2(V)} \right) \right) \\ &= \frac{k}{2} (\log(k) - 2 \log \log(V)) \text{scan}(V \log^2(V)). \end{aligned}$$

Therefore the total I/O cost to produce H is $O(k \text{sort}(V \log^2(V)) + k \log(k) \text{scan}(V \log^2(V)))$.

Finally, Geissman and Gianinazzi provide a cache-oblivious algorithm for exact min cut which uses $O(\text{sort}(E) \log^4(V))$ I/Os [30]. We use this algorithm

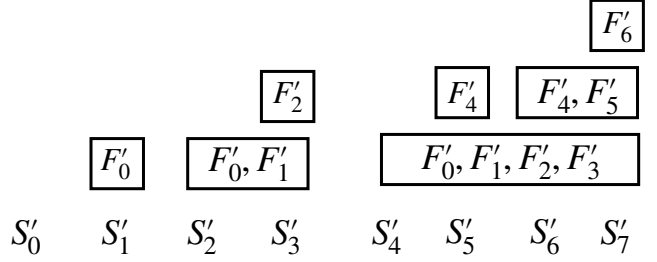


Figure 1: Deletion schedule for k -connectivity query procedure when $k = 8$. Each rectangle represents a block deletion operation over all the sketches it covers and the labels on the rectangle indicate the spanning forests which are deleted during that block deletion. For example, before querying $\mathcal{S}'_4(\mathcal{G})$ the four spanning forests $\bigcup_{i=0}^3 F_i$ are deleted from the four sketches $\bigcup_{i=4}^7 \mathcal{S}_i$, and before querying $\mathcal{S}'_5(\mathcal{G})$, only F'_4 is deleted.

to determine the edge connectivity of H . Since H has at most $k(V - 1)$ edges, the I/O complexity of this step is $O(\text{sort}(kV) \log^4(V))$. \square

Approximating the minimum cut. Ahn et al. [3] provide a $O(\varepsilon^{-2} V \log^4(V))$ -space single-pass streaming algorithm for $(1 + \varepsilon)$ -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 $k = O(\varepsilon^{-2} \log(V))$ -skeleton H_i using Ahn et al.'s k -connectivity algorithm [5]. The authors show that $\lambda(\mathcal{G}) \leq 2^j \lambda(H_j) \leq (1 + \varepsilon) \lambda(\mathcal{G})$ 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 $(S, V \setminus S)$ has weight no more than $(1 + \varepsilon) \lambda(\mathcal{G})$, it cannot be used to recover the set of edges across $(S, V \setminus S)$.

We can obtain an 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 [30] to find the minimum cuts of each (H_i) . This gives the following theorem.

THEOREM 6.2. *There exists an external semi-streaming algorithm for $(1 + \varepsilon)$ -approximating the minimum cut of*

\mathcal{G} w.h.p. that uses $O(\varepsilon^{-2}V \log^4(V))$ -space and

$$\begin{aligned} & O(\text{vsketch}(N, V, \varepsilon^{-2} \log^4 V) \\ & + \log \log V \cdot \text{k sketch}(V, \varepsilon^{-2} \log(V)) \\ & + \log^4(V) \log \log(V) \text{sort}(\varepsilon^{-2} V \log(V))) \end{aligned}$$

I/Os.

Proof. By Theorem 6.1 with $k = \varepsilon^{-2} \log V$, constructing each $\mathcal{K}(\mathcal{G}_i)$ takes $O(\varepsilon^{-2}V \log^3(V))$ space, and there are $O(\log V)$ subgraphs \mathcal{G}_i .

By Theorem 5.1 it costs $\text{vsketch}(N, V, \varepsilon^{-2} \log^4 V)$ I/Os to ingest the stream and compute $\mathcal{K}(\mathcal{G}_i)$, since the total sketch size per vertex is $\phi = \log^2 V \cdot \varepsilon^{-2} \log V$ in $\log V$.

Constructing H_i from the sketches for some i takes $O(\text{k sketch}(V, \varepsilon^{-2} \log(V)))$ I/Os, by Theorem 6.1. To find $\lambda(H_i)$ we apply Geissman and Gianinazzi's exact min cut algorithm which uses $O(\text{sort}(E) \log^4(V))$ I/Os [30]. Since each H_i has a maximum of $O(kV) = O(\varepsilon^{-2}V \log(V))$ edges, we can compute $\lambda(H_i)$ in $O(\log^4(V) \text{sort}(\varepsilon^{-2}V \log(V)))$ I/Os.

We perform a binary search through the certificates H_i to find $j = \min\{i : \lambda(H_i) < k\}$. Thus, the total I/O cost to compute $\lambda(\mathcal{G})$ from the minimum cut sketch is $O(\log \log V \cdot \text{k sketch}(V, \varepsilon^{-2} \log(V)) + \log^4(V) \log \log(V) \text{sort}(\varepsilon^{-2}V \log(V)))$. \square

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 $(1 + \varepsilon)$ 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 6.1. *There exists a*

$$\begin{aligned} & O(\text{sort}(E) + \log \log V \cdot \text{k sketch}(V, \varepsilon^{-2} \log(V)) \\ & + \log^4(V) \log \log(V) \text{sort}(\varepsilon^{-2} V \log(V))) \end{aligned}$$

-I/O external-memory algorithm that returns the edges of a cut $(S, \mathcal{V} \setminus S)$ that is at most $(1 + \varepsilon)$ 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 ε -**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 $\forall S \subset \mathcal{V}, (1 - \varepsilon)\lambda_S(\mathcal{H}) \leq \lambda_S(\mathcal{G}) \leq (1 + \varepsilon)\lambda_S(\mathcal{H})$.

Ahnet al. [3] 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 $k = O(\varepsilon^{-2} \log^2(V))$ -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 [39], 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 $\tilde{O}(V)$ factor.

THEOREM 6.3. *There exists a $O(\varepsilon^{-2}V \log^5(V))$ -space and $O(\text{vsketch}(N, V, \varepsilon^{-2} \log^5(V)) + \log \log(V)(\varepsilon^{-10}V^2 \log^{11}(V))^{1+o(1)}/B)$ -I/O algorithm for constructing a $(1 + \varepsilon)$ -cut sparsifier of graph \mathcal{G} w.h.p.*

Proof. By the bounds in Theorem 6.1, we can ingest the stream and construct the \mathcal{H}_i 's using $O(\text{vsketch}(N, V, \varepsilon^{-2} \log^5(V)) + \log(V) \text{k sketch}(V, \varepsilon^{-2} \log^2(V)))$ I/Os and $O(\varepsilon^{-2}V \log^5(V))$ space.

For each edge $e \in \bigcup_i \mathcal{H}_i$, to compute $j(e)$ we use an external-memory $(1 - \varepsilon')$ -approximation algorithm for maximum flow on undirected graphs with polynomially bounded edge weights due to Laxhuber [39]. On a graph with m edges, this algorithm runs in $O(\text{scan}(m^{1+1/\psi})\varepsilon'^{-3})$ I/Os for any constant $\psi \geq 1$. By setting $\varepsilon' = 1/2k = O(\varepsilon^2/\log^2(V))$, we ensure that this maximum flow algorithm returns the exact value of $\lambda_e(\mathcal{H}_i)$ provided that $\lambda_e(\mathcal{H}_i) \leq k$. And if $\lambda_e(\mathcal{H}_i) \geq k$, the maximum flow returned will also be $\geq k$. By performing a binary search over the \mathcal{H}_i s we can compute $j(e)$ for some e in $O(\log \log(V)\varepsilon^{-6} \log^6(V)(\varepsilon^{-2}V \log^2(V))^{1+o(1)}/B)$ I/Os. Since there are $O(\varepsilon^{-2}V \log^3(V))$ edges in $\bigcup_i \mathcal{H}_i$, the total I/O cost to compute $j(e)$ for all e is $O(\log \log(V)\varepsilon^{-8} \log^9(V)(\varepsilon^{-2}V^2 \log^2(V))^{1+o(1)}/B) = O(V^{2+o(1)}/B \cdot \text{poly}(\log V, \varepsilon^{-1}))$ I/Os.

Finally, once we have computed $j(e)$ for each e in $\bigcup_i \mathcal{H}_i$, we can sort the edges by their j values and determine whether $e \in \mathcal{H}_{j(e)}$ for all e in $O(\text{sort}(\varepsilon^{-2}V \log^2(V)) \text{ I/Os})$. \square

Once we have the cut sparsifier, we can use it to approximately answer s-t min cut queries with Laxhuber's max flow algorithm [39]. 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 6.2. *There exists an algorithm to find x different s - t min cuts on a graph \mathcal{G} w.h.p. using*

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

I/Os in the external-memory model.

Densest subgraph. McGregor et al. [44] give an algorithm for $(1 + \varepsilon)$ -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 $p = \frac{V \log V}{\varepsilon^2 E}$, 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 $(1 + \varepsilon)$ 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 $\Theta(\varepsilon^{-2}V)$ 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 i th bucket in the j th partition, randomly draw $X_{ij} \sim \text{Bin}(E_{ij}, p)$, 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 $4\varepsilon^2 E/V$ 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.

For computing the densest subgraph of \mathcal{H} , we use the algorithm of Charikar [19], as detailed in the proof of Theorem 6.4. This provides a 2-approximation to $d^*(\mathcal{H})$, resulting in a $2(1 + \varepsilon)$ -approximation overall.

THEOREM 6.4. *For a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\varepsilon > 0$ satisfying $\varepsilon^2 E/V = \Omega(\log V)$, there exists a $O(\varepsilon^{-2}V \log^3 V)$ -space and $O(\text{vsketch}(N, \varepsilon^{-2}V, \log^3 V) + V \text{sort}(\varepsilon^{-2}V \log^2 V))$ -I/O algorithm for $2(1 + \varepsilon)$ approximating the density of the densest subgraph of a graph \mathcal{G} w.h.p.*

Proof. By Proposition 2.10 in [12], our modified algorithm produces small buckets w.h.p, since the expected number of edges in each bucket is $\Omega(\log V)$. Therefore, this algorithm correctly solves the densest subgraph problem.

Our algorithm uses $O(\log^2 V)$ sketches for each of the $\Theta(\varepsilon^{-2}V)$ buckets, for $O(\varepsilon^{-2}V \log^3 V)$ total space. Each bucket must also maintain a counter for the number of edges that have been inserted (and not later deleted). This can be done with an additional $O(\log V)$ -space per bucket.

Since each edge update only affects the sketches for its bucket, we can essentially treat this as a vertex-based sketching algorithm, where each of the $\Theta(\varepsilon^{-2}V)$ buckets functions as a vertex. Therefore, we can ingest the stream in $\text{vsketch}(N, \varepsilon^{-2}V, \log^3 V)$ by Theorem 5.1.

In post processing, we must recover an edge from sketch one at a time, and delete them from subsequent sketches for the bucket. The $O(\log^2 V)$ sketches for a bucket fit in memory, so the deletion incurs no additional I/Os. Thus we can recover all sampled edges by scanning through the sketches once in $O(\text{scan}(\varepsilon^{-2}V \log^3 V))$ I/Os, which is dominated by the ingestion cost.

To compute the densest subgraph in \mathcal{H} , we use Charikar's greedy peeling algorithm [19], which gives a 2-approximation to the densest subgraph. 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.

We represent \mathcal{H} as an adjacency list on disk: The vertices are listed in increasing degree order. Each

vertex is followed by a list of its neighbors. This can be formed in $\text{sort}(\varepsilon^{-2}V \log^2 V)$ I/Os to sort the edges of \mathcal{H} . Remove the lowest degree node from the list, and delete it from its neighbors in $\text{scan}(\varepsilon^{-2}V \log^2 V)$ I/Os. Sort the list so that it is still in increasing degree order in $\text{sort}(\varepsilon^{-2}V \log^2 V)$ I/Os. Since this is repeated V times, the total I/O cost is $V \text{sort}(\varepsilon^{-2}V \log^2 V)$. \square

7 Related Work

Graph semi-streaming The graph semi-streaming literature includes both insert only streaming algorithms and fully dynamic (insert and delete) algorithms. The results we enumerate here are restricted to algorithms that require only a single pass over the edge stream. Fully dynamic semi-streaming algorithms include connectivity, bipartiteness testing, cut sparsification, spectral sparsification, approximate minimum spanning tree [43], approximate hierarchical clustering [1], vertex coloring [9], vertex cover [22], approximating the minimum cut, and approximating the number of sub-graphs isomorphic to a graph H [3]. Semi-streaming algorithms for insert only streams include matching, diameter, spanners [28], exact minimum spanning tree [43], and link prediction [41].

External-memory graph algorithms Chiang et al. [20] presented the first external-memory algorithms for various classical graph problems, including connected components, minimum spanning forest, list ranking, and Euler tour trees. Kumar and Schwabe [38] presented improved bounds for computing minimum spanning forests and gave an algorithm for single-source shortest paths in external memory. Arge et al. [8] further improved the bounds for minimum spanning forests, and Meyer and Zeh [51] showed improved bounds for single-source shortest paths. Arge [7] introduced the buffer tree, which is useful in a variety of graph problems. Munagala and Ranade [52] provided an external-memory breadth-first search algorithm which was later improved by Mehlhorn and Meyer [49]. Meyer [50] gave an algorithm for approximating the diameter of sparse graphs.

8 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. This means that 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] A. Agarwal, S. Khanna, H. Li, and P. Patil. Sub-linear algorithms for hierarchical clustering. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:3417–3430, 2022.

- [2] P. K. Agarwal, L. Arge, and K. Yi. I/o-efficient batched union-find and its applications to terrain analysis. In N. Amenta and O. Cheong, editors, *Proceedings of the 22nd Annual ACM Symposium on Computational Geometry (SoCG)*, pages 167–176. ACM, 2006.
- [3] 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.
- [4] 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, pages 2237–2246. JMLR.org, 2015.
- [5] 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)*, pages 459–467, 2012.
- [6] K. J. Ahn, S. Guha, and A. McGregor. Spectral sparsification in dynamic graph streams. In *International Workshop on Approximation Algorithms for Combinatorial Optimization (APPROX)*, volume 8096, pages 1–10, 2013.
- [7] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Workshop on Algorithms and Data Structures (WADS)*, pages 334–345, 1995.
- [8] L. Arge, G. S. Brodal, and L. Toma. On external-memory mst, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.
- [9] S. Assadi, Y. Chen, and S. Khanna. Sublinear algorithms for $(\delta + 1)$ vertex coloring. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 767–786. SIAM, 2019.
- [10] S. Assadi, S. Khanna, and Y. Li. Tight bounds for single-pass streaming complexity of the set cover problem. In *STOC*, pages 698–711. ACM, 2016.
- [11] 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*, pages 698–711, 2016.
- [12] S. Assadi and V. Shah. Tight bounds for vertex connectivity in dynamic streams. In *Symposium on Simplicity in Algorithms (SOSA)*, pages 213–227. SIAM, 2023.
- [13] A. Bagchi, A. Chaudhary, D. Eppstein, and M. T. Goodrich. Deterministic sampling and range counting in geometric data streams. In J. Snoeyink and J. Boissonnat, editors, *Proceedings of the 20th ACM Symposium on Computational Geometry*, Brooklyn, New York, USA, June 8-11, 2004, pages 144–151. ACM, 2004.
- [14] 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*, pages 13–23, 2017.
- [15] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems (TOCS)*, 934-962(47), 2010.
- [16] 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, pages 523–528, New York, NY, USA, 2006. Association for Computing Machinery.
- [17] A. Bishnu, A. Ghosh, G. Mishra, and S. Sen. On the streaming complexity of fundamental geometric problems. *CoRR*, abs/1803.06875, 2018.
- [18] W. D. C. Blog. Cpu bandwidth – the worrisome 2020 trend, nov 2021. Accessed: June 7, 2024.
- [19] M. Charikar. Greedy approximation algorithms for finding dense components in a graph. In K. Jansen and S. Khuller, editors, *Proceedings of the Third Annual Approximation Algorithms for Combinatorial Optimization (APPROX)*, volume 1913, pages 84–95. Springer, 2000.
- [20] 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)*, pages 139–149, 1995.
- [21] 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*, pages 1326–1344. SIAM, 2016.
- [22] 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)*, pages 1234–1251, 2014.
- [23] R. Clark. Largest ssd that you can buy, Aug 2023. Accessed: Jan 23, 2024.
- [24] M. S. Crouch, A. McGregor, and D. Stubbs. Dynamic graphs in the sliding-window model. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, pages 337–348, 2013.
- [25] A. Czumaj, S. H. Jiang, R. Krauthgamer, and P. Veselý. Streaming algorithms for geometric steiner forest. In M. Bojanczyk, E. Merelli, and D. P. Woodruff, editors, *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022, July 4-8, 2022, Paris, France*, volume 229 of *LIPIcs*, pages 47:1–47:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
- [26] Y. Emek and A. Rosén. Semi-streaming set cover. *ACM Transactions on Algorithms (TALG)*, 13(1):1–22, 2016.
- [27] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.
- [28] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and

- J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2):207–216, Dec. 2005.
- [29] G. Frahling and C. Sohler. Coresets in dynamic geometric data streams. In H. N. Gabow and R. Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22–24, 2005*, pages 209–217. ACM, 2005.
- [30] B. Geissmann and L. Gianinazzi. Cache oblivious minimum cut. In D. Fotakis, A. Pagourtzis, and V. T. Paschos, editors, *7th International Conference on Algorithms and Complexity (CIAM)*, pages 285–296. Springer International Publishing, 2017.
- [31] E. Georganas, R. Egan, S. Hofmeyr, E. Goltsman, B. Arndt, A. Tritt, A. Buluç, L. Olikek, and K. Yelick. Extreme scale de novo metagenome assembly. SC '18. IEEE Press, 2018.
- [32] 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)*, pages 241–247. ACM, 2015.
- [33] S. Har-Peled, P. Indyk, S. Mahabadi, and A. Vakilian. Towards Tight Bounds for the Streaming Set Cover Problem. *arXiv e-prints*, page arXiv:1509.00118, Aug. 2015.
- [34] J.-W. Hong and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on the Theory of Computation (STOC)*, pages 326–333, 1981.
- [35] P. Indyk. Streaming algorithms for geometric problems. In K. Lodaya and M. Mahajan, editors, *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, 24th International Conference, Chennai, India, December 16–18, 2004, Proceedings*, volume 3328 of *Lecture Notes in Computer Science*, pages 32–34. Springer, 2004.
- [36] 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)*, pages 561–570, 2014.
- [37] 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)*, pages 272–281. ACM, 2014.
- [38] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 169–176, 1996.
- [39] H. Laxhuber. A cache-efficient (1-epsilon)-approximation algorithm for the maximum flow problem on undirected graphs. Bachelor thesis, ETH Zurich, Zurich, 2021.
- [40] 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*, pages 596–604. Springer, 2014.
- [41] J. Li, K. Cheng, L. Wu, and H. Liu. Streaming link prediction on dynamic attributed networks. In *Proceedings of the 11th ACM International Conference on Web Search and Data Mining (WSDM)*, pages 369–377, 2018.
- [42] 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)*, page 174–183. Association for Computing Machinery, 2014.
- [43] A. McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.
- [44] A. McGregor, D. Tench, S. Vorotnikova, and H. T. Vu. Densest subgraph in dynamic graph streams. In G. F. Italiano, G. Pighizzini, and D. T. Sannella, editors, *Proceedings of the 40th Mathematical Foundations of Computer Science (MFCS)*, pages 472–482, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [45] 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.
- [46] 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)*, pages 401–411. ACM, 2016.
- [47] A. McGregor and H. T. Vu. Evaluating bayesian networks via data streams. In D. Xu, D. Du, and D. Du, editors, *Computing and Combinatorics*, pages 731–743, Cham, 2015. Springer International Publishing.
- [48] A. McGregor and H. T. Vu. Better streaming algorithms for the maximum coverage problem. *Theory of Computing Systems*, 63:1595–1619, 2019.
- [49] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In R. H. Möhring and R. Raman, editors, *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, volume 2461, pages 723–735. Springer, 2002.
- [50] U. Meyer. On trade-offs in external-memory diameter-approximation. In J. Gudmundsson, editor, *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 5124, pages 426–436. Springer, 2008.
- [51] U. Meyer and N. Zeh. I/o-efficient shortest path algorithms for undirected graphs with random or bounded edge lengths. *ACM Transactions on Algorithms (TALG)*, 8(3):22:1–22:28, 2012.
- [52] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, volume 99, pages 687–694, 1999.
- [53] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science (FnTs)*, 1(2):117–236, 2005.
- [54] J. Nelson and H. Yu. Optimal lower bounds for dis-

- tributed and streaming spanning forest computation. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1844–1860. SIAM, 2019.
- [55] 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(1–3):3–36, Apr. 2001.
 - [56] S. Nurk, D. Meleshko, A. Korobeynikov, and P. Pevzner. Metaspades: A new versatile metagenomic assembler. *Genome Research*, 27:gr.213959.116, 03 2017.
 - [57] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Information Processing Letters (IPL)*, 112(7):277–281, 2012.
 - [58] 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(4):420–431, Dec. 2017.
 - [59] F. Smith. Memory bandwidth napkin math, 2020. Accessed: June 7th, 2024.
 - [60] SolidIGM. Data center: D5-p5336, 2024. Accessed: Jan 23, 2024.
 - [61] 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, page 325–339, New York, NY, USA, 2022. Association for Computing Machinery.
 - [62] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33(2), 2001.
 - [63] 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(3):377–399, June 2019.
 - [64] 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(3):377–399, June 2019.
 - [65] D. P. Woodruff and T. Yasuda. High-dimensional geometric streaming in polynomial space. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 732–743. IEEE, 2022.

A Case Study: k-Connectivity

In this section, we expand on why graph sketching is infeasible using today’s hardware with a case study of k-connectivity.

Consider the following back-of-the-envelope calculation. Ahn et al.’s [3] sketch for testing k-connectivity has size $O(kV \log^3 V)$ bits. Assuming 128GB of RAM and $k = \log V$, the largest graph whose sketch fits in RAM has 114,000 vertices.²

However, for graphs of this size, sketching doesn’t even save space. The adjacency list of even a complete graph on 114,000 vertices has size at most 13GiB, while the k-connectivity sketch has size 128GiB—an order of magnitude larger. Despite the fact that the sketch is asymptotically smaller than an adjacency list, at this input size its high constant factors and the $\log^3 V$ term dominate the space cost.

Thus, the asymptotic advantages of sketching for semi-streaming algorithms [3, 5, 6, 12, 36, 37, 46, 57] so far have not translated into space savings in implementation. When might we expect empirical advantages?

These advantages are unlikely to arrive for a long time, unfortunately. For example, for the k-connectivity sketch to save space, we need graphs with at least $V = 33,000,000$ vertices and at this size, the k-connected-components sketch has size 118TiB. Further, the V and $\log V$ factors in the k-connectivity space complexity cannot be improved asymptotically due to a lower bound of Nelson and Yu [54]. So sketching will not save us space for this problem until we have RAM sizes in the hundreds of TiBs.

B Hardware Trends in Solid State Drives

The external semi-streaming model requires storing most data structures on disk rather than RAM. This approach helps because solid state drives (SSDs) are big enough to store sketches. High-performance enterprise grade SSDs can be bought in sizes of 64 TBs [60] and consumer grade SSDs reach up to 8 TB in size [23]. Many such disks can be used on a single computer. But the approach appears to be a step backwards because the underlying motivation of streaming and semi-streaming is that disks are too slow to support effective computation on graph data [43].

However, in the decades since the introduction of the streaming model, the cost of accessing disk has changed. It is no longer exactly true that disks are too slow—in fact they are fast enough if accessed in the right way. Notably, the sequential bandwidth of

²To compute the actual size of the sketch, including constants, we extrapolate from the sketch size in Tench et al.’s implementation [61] of Ahn et al.’s [5] connectivity sketch.

modern SSDs is very fast. In fact, SSD sequential bandwidth is already nearly as fast as RAM random-access bandwidth [59]—and SSD bandwidth is growing faster [18].

Unfortunately, most existing graph-sketching algorithms use techniques such as hashing to random locations [5, 12, 32, 36, 43], meaning that most accesses that the algorithm makes will be random accesses. Using disks the way most graph-sketching algorithms need is too slow because SSD *random-access bandwidth* is much slower than that of RAM [18]. The result is that an off-the-shelf sketching algorithm designed for RAM will run orders-of-magnitude more slowly on an SSD, making it useless for any reasonable application.

C Summary of Existing Graph Sketch and External-Memory Algorithms

We summarize the I/O cost of existing graph sketching and external-memory algorithms for the problems we study in this paper in Table 3.

D I/O Complexity of StreamingCC

In the streaming connectivity problem, stream updates are *fine-grained*: each update represents the insertion or deletion of a single edge. Since streams are ordered arbitrarily, even a short sequence of stream updates can be highly non-local, inducing changes throughout the graph. As a result, STREAMINGCC and similar graph streaming algorithms do not have good data locality in the worst case. This lack of locality can incur many I/Os and therefore reduce the ingestion rate if the sketches are stored on disk. The I/O cost can be high since ingesting each stream update (u, v, Δ) requires modifying two poly-logarithmic-sized vertex sketches, and can thus induce a poly-logarithmic number of I/Os.

OBSERVATION 1. *In the hybrid graph streaming setting with $M = o(V \log^2(V))$ RAM and $D = \Omega(V \log^2(V))$ disk, STREAMINGCC uses $\Omega(1)$ I/Os per edge update, and processing the entire stream of length N uses $\Omega(N) = \Omega(E)$ I/Os.*

Any sketching algorithm that scales out of core suffers severe performance degradation unless it amortizes the per-update overhead of accessing disk. Such an amortization is not straightforward, since sketching inherently makes use of hashing and as a result induces many random accesses, which are slow on persistent storage. Tench et al. [61] give an external-memory sketching algorithm for connected components which amortizes disk access costs, even for adversarial graph streams. The result is an algorithm that is both space-efficient and I/O-efficient. We restate below their results

on the space- and I/O- complexity of first reading in the stream, and then computing the connected components of the resulting graph.

LEMMA D.1. (RESTATED FROM [61] LEMMA 4)

The algorithm from [61] uses $O(V \log^2 V)$ space and $O(\text{sort}(N) + \text{scan}(V \log^2 V))$ I/Os to process the stream updates.

LEMMA D.2. (RESTATED FROM [61] LEMMA 5)

Once all stream updates have been processed, the algorithm from [61] computes connected components using $O((V/B) \log^2(V) + V\alpha(V))$ I/Os.

Algorithm	Standard Sketching I/O Cost	External Memory
Connected Components [5, 20, 61]	$O(\text{sort}(N) + V\alpha(V) + (N + V)\text{scan}(\log^2 V))$	$\text{sort}(N)$
Bipartiteness Testing [5, 20]	$O(N + V\alpha(V) + (N + V)\text{scan}(\log^2 V))$	$\text{sort}(N)$
Hypergraph Connectivity [32]	$O(N + V\alpha(V) + (N + V)\text{scan}(r \log^2 V))$	no nontrivial algorithm
ε -Approx. MST Weight [5, 20]	$O(N + \varepsilon^{-1}V\alpha(V) + (N + V)\text{scan}(\varepsilon^{-1} \log^2 V))$	$\text{sort}(N)$ (exact)
k -Connectivity [3, 30]	$O(N + kV\alpha(V) + (kN + k^2V)\text{scan}(\log^2 V))$	$O(\text{sort}(N) \log^4 V)$ (min-cut)
ε -Approx. Minimum Cut [3, 30]	$O(N + V\varepsilon^{-2} \log^2 V\alpha(V) + (N\varepsilon^{-2} \log^2 V + V\varepsilon^{-4} \log^3 V)\text{scan}(\log^2 V))$	$O(\text{sort}(N) \log^4 V)$ (exact)
ε -Cut Sparsifier [3]	$O(N + V\varepsilon^{-2} \log^3 V\alpha(V) + (N\varepsilon^{-2} \log^3 V + V\varepsilon^{-4} \log^4 V)\text{scan}(\log^2 V))$	no nontrivial algorithm
ε -Approx. Densest Subgraph [44]	$O(N \log V + (N + V)\text{scan}(\varepsilon^{-2} \log^2 V))$	no nontrivial algorithm

Table 3: Bounds for naively using known sketching algorithms in external memory and for the external memory algorithms that solve these problems. The input to all algorithms is a dynamic graph stream of length N on V vertices. Note that the sketching column indicates the I/Os required to sketch the input stream and to extract a sparser graph from the sketch that approximates the original property in question. We do not include the I/O cost of computing said property on the sparser graph. Also note that all existing EM algorithms for these problems have a space cost of $\Omega(N)$ because they need to store the entire graph. $\alpha(x)$ is the inverse Ackermann function.