DYNAMIC SPECTRAL SPARSIFICATION: THEORY AND PRACTICE

Anonymous authors
Paper under double-blind review

ABSTRACT

We study the problem of compressing an undirected, weighted graph by replacing it with a sparser representative, called a spectral sparsifier, while preserving its Laplacian spectrum to a specified precision. Spectral sparsifiers are a fundamental primitive in numerical linear algebra, with applications ranging from fast solvers for Laplacian systems to computing commute times and semi-supervised learning on graph data. In this paper, we focus on the dynamic setting, where the graph undergoes edge updates such as insertions or deletions, and the goal is to maintain a spectral sparsifier as efficiently as possible. We introduce a simple algorithmic framework for dynamic spectral sparsification based on maintaining a hierarchy of clusterings of graphs into low-diameter clusters, known as low-diameter decompositions. Our algorithm maintains a sparsifier of nearly optimal size $O(n \log^3 n \log\log n/\varepsilon^2)$, supports updates in $O(\log^6 n \log\log n)$ amortized time, and achieves an amortized recourse of only $O(1)$. In each of these three performance measures, our theoretical guarantees improve those of Abraham, Durfee, Koutis, Krinninger, and Peng [FOCS '16]. On the practical side, we evaluate our framework on both real-world and synthetic datasets, where our algorithm shows competitive performance against natural baselines.

1 INTRODUCTION

Spectral sparsification is a central concept in the analysis of graph-structured data, which entails compressing an undirected graph into a sparser graph representative while approximately preserving its Laplacian spectrum. It underpins many applications, ranging from Laplacian system solvers (Spielman & Teng, 2014; Koutis et al., 2010; Jambulapati & Sidford, 2025) to computing commute times (Chandra et al., 1996) to semi-supervised learning (Kyng et al., 2015; Calandriello et al., 2018; Zhu et al., 2019). More broadly, it is one of the core tools underpinning the so-called "Laplacian paradigm", a powerful algorithmic framework that combines insights from algorithm design and spectral graph theory to efficiently process massive graphs. As a result, sparsification has been extensively studied due to its broad applicability and theoretical significance. In particular, starting with the seminal work of Spielman & Teng (2011), several fast algorithms (Spielman & Srivastava, 2008; Koutis et al., 2012; 2016) are known for computing spectral sparsifiers.

While these algorithms are competitive in static contexts, modern analysis of graph-based data involves handling frequently changing data, such as edge insertions and deletions. This necessitates algorithms that efficiently maintain high-quality sparsifiers over time. Therefore, spectral sparsification has been extensively studied across various computational models, including online algorithms (Cohen et al., 2020; Braverman et al., 2020; Goranci et al., 2026), streaming algorithms (Kelner & Levin, 2013; Ahn et al., 2013; Kapralov et al., 2020), dynamic algorithms (Abraham et al., 2016; Kyng et al., 2017; Bernstein et al., 2022), and more. In the dynamic algorithm model, the goal is to process edge updates in real time while ensuring efficient update time, tight approximation, and a compact sparsifier. Another equally important requirement that has gained increasing attention is ensuring sparsifier stability, typically measured via recourse (the number of changes in the sparsifier per update). Optimizing these performance measures for dynamic graph sparsification and its variants has become a focus of recent research (Abraham et al., 2016; Kyng et al., 2017; Bernstein et al., 2022; Forster et al., 2026; Goranci & Momeni, 2025; Khanna et al., 2025; Zhao, 2026).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Despite significant progress in understanding spectral sparsification in the dynamic setting, optimizing for the best poly-logarithmic overheads in both update time and sparsifier size remains underexplored. Moreover, the recourse setting has not been studied explicitly[1] and the practical implications of existing theoretical algorithms remain poorly understood. This paper makes significant progress on both fronts, designing dynamic spectral sparsifiers that are *theoretically* more efficient, stable, and compact, and showing that they also achieve competitive performance in *practice*.

Problem Definition. Let $G = (V, E, w)$ be an undirected, weighted graph, where $n = |V|$ and $m = |E|$. Let L_G be the Laplacian matrix associated with G. For any $\varepsilon \in (0, 1)$, a weighted graph $H = (V, E', w')$ is a $(1 \pm \varepsilon)$-approximate spectral sparsifier of G iff $(1 - \varepsilon)L_G \preceq L_H \preceq (1 + \varepsilon)L_G$, where $\preceq$ stands for Löwner ordering. Let $|E'|$ denote the *size* of sparsifier H.

In the dynamic setting, the underlying graph G evolves through a sequence of *updates*, where each update either inserts or deletes an edge. Thus, while the vertex set remains fixed, the edge set changes over time. The goal is to *explicitly* maintain a $(1 \pm \epsilon)$-approximate spectral sparsifier H after every update. The *recourse* of the algorithm is the number of edge updates made to H in response to a single update to G. The time the algorithm takes to process an update to G is called its *update time*. For simplicity, we assume that all edge weights in G are polynomially bounded in n[2]. We wish to design dynamic algorithms that simultaneously achieve strong approximation guarantees, small sparsifier size, low update time, and low recourse.

Our Result. We present a simple and efficient algorithm for the dynamic spectral sparsification problem. Formally,

Theorem 1.1. *There exists a randomized algorithm maintaining a $(1 \pm \epsilon)$-approximate spectral sparsifier of a weighted undirected graph on n vertices with polynomially bounded edge weights undergoing edge insertions and deletions in $O(\log^6 n \log\log n)$ expected amortized update time. The output sparsifier is of size $O(n \log^3 n \log\log n/\varepsilon^2)$, undergoes expected amortized $O(1)$ updates per update to the input graph and the algorithm is correct with high probability against an oblivious adversary.*

The state-of-the-art dynamic spectral sparsification algorithm of (Abraham et al., 2016), together with the improved analysis of (Kyng et al., 2017), achieves a $(1 \pm \varepsilon)$ approximation with sparsifier size $O(n \log^7 n/\varepsilon^2)$ and update time $O(\log^7 n/\varepsilon^2)$. Although these works do not study recourse explicitly, their algorithms incur recourse at least $\Omega(\log n)$. Beyond improving the logarithmic overheads in both sparsifier size and update time, which is particularly important for understanding the practical scalability of dynamic sparsifier constructions, our algorithm from Theorem 1.1 offers two further advantages: (i) its update time is independent of the precision parameter ε, and (ii) it achieves constant recourse.

Dynamic spectral sparsification has been repeatedly used as a key subroutine for adjacent problems at the intersection of data structures and spectral graph theory, including dynamic all-pairs effective resistance estimation and dynamic spectral vertex sparsification (Durfee et al., 2019; Chen et al., 2020; Gao et al., 2021; van den Brand et al., 2022). Consequently, plugging the guarantees of Theorem 1.1 into these frameworks directly yields improved logarithmic overheads in the running times of these algorithms.

Experiments. We perform a systematic empirical study of dynamic spectral sparsification algorithms on both real-world and synthetic data sets. Compared to the baseline, our algorithm shows strong performance in two key measures: it (i) produces smaller sparsifiers at the same certified approximation precision and (ii) achieves faster runtime per edge update. We attribute these gains in part to the simplicity of our approach, which is based on a hierarchy of low-diameter decompositions. As a baseline, we implement an optimized version of the dynamic spectral sparsification algorithm of (Abraham et al., 2016). To the best of our knowledge, this is the first practical implementation of this theoretical algorithm.

[1] If we maintain an explicit sparsifier, then the recourse can always be upper-bounded by the update time, but not vice-versa

[2] This allows us to state the running times and the sparsifier size independent of the ratio between the largest and smallest edge weight.

Prior work investigating dynamic spectral sparsification in practice remains limited. DyGrass (Yuan et al., 2025) does not provide theoretical approximation guarantees and operates in a regime where the observed approximation error is roughly 20 times larger than in our experiments, although it produces smaller sparsifiers. Zhu et al. (Zhu et al., 2019) study spectral sparsification in the fault-tolerant setting based on graph spanners, which is more restrictive than the fully dynamic setting considered here. Moreover, their empirical evaluation does not directly measure the approximation quality of the maintained sparsifier.

1.1 Technical Overview

(Spielman & Srivastava, 2008) showed that independently sampling each edge $e \in E$ with probability $p_e = \min\{1, \Theta(\varepsilon^{-2} w_e R_e \log n)\}$ and assigning each sampled edge weight w_e/p_e yields a spectral sparsifier, where R_e denotes the effective resistance of e. We build on the static spectral sparsification algorithm of Baumann & Zöcklein (2026) relying on this sampling technique, estimating effective resistances as follows. For each of $\widetilde{O}(1)$ dyadic sampling rates $\pi_i = 2^{1-i}$, their algorithm independently samples $\widetilde{O}(1)$ subgraphs, retaining each edge e with probability $\min\{1, w_e\pi_i\}$. It then constructs a low-diameter decomposition of each sampled subgraph. The effective resistance of an edge is then derived from the smallest sampling rate π_i, for which its endpoints belong to the same cluster in a sufficiently large constant fraction of the corresponding sampled subgraphs.

We implement this framework under edge deletions using the decremental low-diameter decomposition algorithm of Forster & Goranci (2019), which allows us to maintain the clusterings in all sampled subgraphs efficiently. Their algorithm guarantees that each vertex changes clusters only $\widetilde{O}(1)$ times throughout a deletion sequence. We maintain the edge-wise effective resistance estimates by scanning incident edges whenever a vertex changes clusters in any of the maintained decompositions. We ensure that the output sparsifier has amortized recourse $O(1)$ by using the largest observed effective resistance estimate for each edge. In turn this guarantees that sampling probabilities monotonically increase and edges enter the sparsifier at most once.

To incorporate edge insertions, we use the bucketing scheme of Abraham et al. (2016). We maintain $O(\log n)$ buckets with geometrically increasing capacities and sparsify each bucket separately. Insertions first enter the smallest bucket. Once a bucket reaches its insertion quota, its surviving edges are transferred to the next larger bucket. Whenever a bucket receives a batch of edge insertions, we reinitialize the decremental algorithm on its contents. A bucket of capacity 2^i is rebuilt only once per $\Theta(2^i)$ insertions, and between consecutive rebuilds its contents undergo only deletions. This means the algorithm has to pay the update time cost of the decremental implementation $O(\log n)$ times to maintain the fully dynamic structure. In order to obtain the $O(1)$ amortized recourse bound we rely on a more fine-grained decremental recourse bound (see Claim 4.4).

To further optimize sparsifier size, we re-sparsify the output with the same algorithm. The difference in the second layer of sparsification, is that during the bucketing scheme the smallest bucket is of size $\Theta_\epsilon(n)$ and its contents are not sparsified. As the input to the second sparsifier is of size $O_\epsilon(n \operatorname{poly}\log n)$ this reduces the number of buckets to $O(\log\log n)$. In turn the second sparsification reduces the final sparsifier size by an $O(\log n/\log\log n)$ factor.

2 Preliminaries

Throughout this paper we consider weighted simple graphs $G = (V, E, w)$ where $|V| = n$, $|E| = m$, and $W = \max_{e\in E} w_e$. Since rescaling all edge weights does not affect spectral approximation, we assume $w_e \geq 1$ for every $e \in E$. For $S \subseteq V$, let $G[S]$ denote the subgraph induced by S. For an unweighted graph F, let $\operatorname{dist}_F(u, v)$ denote the hop-distance between u and v, with value ∞ when they are disconnected. For a partition $\mathcal{X}$ of V, let $\mathcal{X}(v)$ denote the cluster containing v. An edge (u, v) is cut by $\mathcal{X}$ if $\mathcal{X}(u) \neq \mathcal{X}(v)$.

The *Laplacian* of G is $L_G = \sum_{e=(u,v)\in E} w_e(\mathbb{1}_u - \mathbb{1}_v)(\mathbb{1}_u - \mathbb{1}_v)^\mathsf{T}$. For symmetric matrices A and B, we write $A \preceq B$ when $B - A$ is positive semidefinite. The *effective resistance* of $e = (u, v)$ is $R_e = (\mathbb{1}_u - \mathbb{1}_v)^\mathsf{T} L_G^+ (\mathbb{1}_u - \mathbb{1}_v)$, and $w_e R_e$ is its *leverage score*.

Definition 2.1. *A weighted graph H on V is a $(1 \pm \varepsilon)$-approximate spectral sparsifier of G for $\varepsilon > 0$ if*

$$(1 - \varepsilon)L_G \preceq L_H \preceq (1 + \varepsilon)L_G.$$

For $\eta \in [0, 1]$, let G_η denote the unweighted random graph obtained by retaining every edge e independently with probability $\min\{w_e\eta, 1\}$. For an edge $e = (u, v)$ and $\kappa \geq 1$, define

$$p_\kappa(e, \eta) = \mathbb{P}[\mathrm{dist}_{G_\eta}(u, v) > \kappa], \qquad q_\kappa(e) = \max\{\eta \in [0, 1] : p_\kappa(e, \eta) \geq 1/2\}.$$

The quantity $q_\kappa(e)$ is the *κ-robust connectivity* of e.

Lemma 2.2 (Kapralov & Panigrahy, 2012; Baumann & Zöcklein, 2026)**.** *For every edge e, integer $\kappa \geq 4$, and $\pi \in (0, 1]$,*

$$R_e \leq 2\kappa q_\kappa(e), \qquad \sum_{e \in E} w_e q_\kappa(e) \leq 4n^{1+O(1/\kappa)}, \qquad q_\kappa(e) \geq \frac{1}{4mW}.$$

Moreover, $q_\kappa(e) \leq \pi/4$ implies $p_\kappa(e, \pi) \leq 1/4$.

Definition 2.3. *Let $G = (V, E)$ be an unweighted graph. A randomized partition $\mathcal{X}$ of V is a (β, D) low-diameter decomposition, or LDD, if*

- *every cluster $C \in \mathcal{X}$ has strong diameter at most D, meaning $\mathrm{dist}_{G[C]}(u, v) \leq D$ for all $u, v \in C$;*
- *$\mathbb{P}[\mathcal{X}(u) \neq \mathcal{X}(v)] \leq \beta$ for every edge $(u, v) \in E$.*

The edge-wise cut guarantee and a union bound imply the following path-wise guarantee.

Observation 2.4. *Let $\mathcal{X}$ be a (β, D) LDD of G. If $\mathrm{dist}_G(u, v) \leq k$, then*

$$\mathbb{P}[\mathcal{X}(u) = \mathcal{X}(v)] \geq 1 - k\beta.$$

In particular, this probability is at least $3/4$ when $\beta \leq 1/(4k)$.

Theorem 2.5 (Spielman & Srivastava, 2008)**.** *Let $0 < \delta < 1$ and suppose $\widehat{R}_e \geq R_e$ for every edge of $G = (V, E, w)$. Sample each edge independently with probability*

$$p_e = \min\{1, 10^4\varepsilon^{-2} w_e \widehat{R}_e \ln(n/\delta)\},$$

and give sampled edge e weight w_e/p_e. The resulting graph is a $(1 \pm \varepsilon)$-approximate spectral sparsifier of G with probability at least $1 - \delta$.

3 Static Spectral Sparsification via Low-Diameter Decompositions

This section is devoted to presenting an implementation of the static spectral sparsification framework of Baumann & Zöcklein (2026). In Section 4 we present our decremental implementation of the same algorithm and in Section 5 we show how this can be extended into the fully dynamic setting. Let $G = (V, E, w)$ be nonempty, let $M \geq m$ be an upper bound on its number of edges, and let $n^{-1/2} < \varepsilon \leq 1$; for smaller ε, the simple graph G has $m \leq n^2 \leq n\varepsilon^{-2}$ edges and may be returned unchanged. Let $\mathsf{LDD}(G, \beta)$ stand for an algorithm returning a $(\beta, D_{\mathsf{LDD}}(n, \beta))$-LDD of G. For the algorithm of (Forster & Goranci, 2019), we have $D_{\mathsf{LDD}}(n, \beta) = O(\beta^{-1} \log n)$; we keep its parameters symbolic. The algorithm pseudo-code is presented in Algorithm 1, Appendix A.

Assume that M is some upper bound on m. The algorithm considers $L = O(\log(MW))$ scales $\pi_i = 2^{1-i}$, for $i \in [L]$. For each scale π_i, the algorithm samples $\lambda = O(\log(nML))$ graphs, denoted by G_i^j for $j \in [\lambda]$, from the distribution G_{π_i}. Recall that in G_{π_i}, every edge $e \in E$ is sampled independently with probability $\min\{w_e\pi_i, 1\}$. The algorithm then constructs a low-diameter decomposition of every sampled graph using LDD, denoted by $\mathcal{X}_i^j$.

The algorithm sets $\kappa = \Theta(\log n)$ and $D = \max\{\kappa, D_{\mathsf{LDD}}(n, \beta)\}$. The goal of the algorithm is to find a robust connectivity estimate $\widehat{q}_e$ for each edge $e \in E$ satisfying $q_D(e) \leq \widehat{q}_e = O(q_\kappa(e))$. In order to find $\widehat{q}_e$ for each scale $\pi_i \mid i \in [L]$ the algorithm calculates $Z_i(e)$ the fraction of LDD-s of that scale where the endpoints of e fall in the same cluster. The algorithm then finds c_e the *critical scale* of e, which stands for the largest index i for which $Z_i(e) \geq 17/32$, and sets $\widehat{q}_e = \pi_{c_e}$.

The algorithm then uses $\widehat{R}_e = 2D\widehat{q}_e$ as the effective resistance estimate of edge e. Edge e is added to the resulting sparsifier H with probability $p_e = \Theta(\min\{1, \epsilon^{-2} w_e \widehat{R}_e \ln n\})$ and weight w_e/p_e which defines a suitable sparsifier by Theorem 2.5.

We will defer much of the formal analysis of correctness of the algorithm to Section A as it is re-stating the results of (Baumann & Zöcklein, 2026). Here we will just state the following two claims, their formal proof is deferred to the same section, which we will later rely on when proving the correctness of our dynamic implementations. The first claim shows how the critical scale c_e of an edge can be used to bound its robust connectivity.

Claim 3.1. *At the end of a run of Algorithm 1, with high probability, the following hold simultaneously for every edge $e \in E$ and scale $i \in [L]$:*

$$q_\kappa(e) \leq \pi_i/4 \implies Z_i(e) \geq 17/32, \qquad q_D(e) \geq \pi_i \implies Z_i(e) < 17/32.$$

Furthermore, if any edge e satisfies the above inequalities then $q_D(e) \leq \pi_{c_e} < 8q_\kappa(e)$.

Claim 3.1 implies that with high probability $q_D(e) \leq \pi_{c_e} = \widehat{q}_e \leq 8q_\kappa(e)$ for all $e \in E$. The following claim states that if the robust connectivity estimates satisfy these bounds, the sampling step results in a suitable sparsifier.

Claim 3.2. *Assume that at the end of the run of Algorithm 1 estimates $\widehat{q}_e$ satisfy $q_D(e) \leq \widehat{q}_e \leq 8q_\kappa(e)$ for all $e \in E$. Then the output graph H is a $(1 \pm \epsilon)$-approximate spectral sparsifier of G of size $O(nD\epsilon^{-2} \log n)$ with high probability.*

Setting LDD to be the algorithm of (Forster & Goranci, 2019) into, we have $D = O(\log^2 n)$ implying that H is of size $O(n\varepsilon^{-2} \log^3 n)$. We will only analyze the running time of the algorithm in the dynamic setting.

4 Spectral Sparsification Under Edge Deletions

We use the following decremental LDD algorithm as a black box. The stated total-time bound uses the faster ES-tree based implementation noted in (Forster & Goranci, 2019).

Lemma 4.1 (Forster & Goranci, 2019)**.** *For every $0 < \beta < 1$, there is a randomized decremental algorithm that maintains a $(\beta, O(\beta^{-1} \log n))$-LDD of an initially m-edge graph in expected total update time $O(m\beta^{-1} \log^2 n)$ against an oblivious adversary. Moreover, every vertex changes clusters $O(\beta^{-1} \log^2 n)$ times in expectation over the entire deletion sequence.*

The pseudo-code of the decremental implementation appears in Algorithm 2, deferred to Appendix B. A high level description of the algorithm is as follows. Let $G_0 = (V, E_0, w)$ be the initial graph and $m_0 = |E_0|$. At initialization, the decremental algorithm runs the static spectral sparsifier Algorithm 1 with $M = m_0$ using the algorithm of Lemma 4.1 to maintain the LDD-s. Throughout the deletion sequence parameters M, the initial weight bound W, κ, D, L, λ and all scales remain fixed. Similarly, the algorithm fixes the sampled sub-graphs G_i^j at initialization.

The sparsifier itself is sampled based on sampling probabilities p_e. In order to maintain p_e, the algorithm needs to maintain the critical scale c_e of e. This is done greedily: whenever due to a deletion in any of the LDD-s maintained by the algorithm a vertex v changes clusters, the algorithm updates the $Z_i(e)$ values of all edges e incident on v in the input graph. Based on this information, the algorithm can update the critical scale c_e.

Slightly deviating from the static algorithm, the decremental implementation defines the robust connectivity estimate $\widehat{q}_e$ of e to be $\pi_{c_e^*}$, where c_e^* stands for the *smallest* c_e value ever observed for edge e. Once the algorithm establishes the new $\widehat{q}_e$ values, it updates sampling probabilities p_e analogously to the static implementation and updates the output sampled sparsifier if necessary.

By Lemma 4.1 a vertex may change its clustering in a specific maintained LDD at most $\tilde{O}(1)$ times in expectation across the update sequence. As the algorithm maintains $\tilde{O}(1)$ scales and $\tilde{O}(1)$ sampled sub-graphs for each scale, the maintenance of $Z_i(e)$ requires in expectation at most $\tilde{O}(1)$ total time. Summing across all edges, the total update time of the algorithm is $\tilde{O}(m)$. The exact logarithmic

parameters of the running time bound are stated in the following claim, its proof is deferred to Appendix B.

Claim 4.2. *For $m_0 \geq 1$, the expected total time of algorithm 2 is*

$$O\left(m_0 \log(n \log(nW)) \log(nW) \log^3(n) \log\log(nW)\right),$$

or $O(m_0 \log^5(n) \log\log n)$ if $W = \text{poly}(n)$.

The following claim shows that the robust connectivity estimates $\pi_{c_e^*}$ maintained by the decremental implementation satisfy the same approximation properties as the estimates π_{c_e} used by the static Algorithm 1.

Claim 4.3. *Throughout a run of Algorithm 2 for any edge $e \in E$ with high probability,*

$$q_D(e) \leq \pi_{c_e^*} \leq 8 q_\kappa(e).$$

Proof. Assume that the high probability event of Claim 3.1 holds for all edges throughout the input sequence. Fix an edge $e \in E$ present in the input graph at a given time. By Claim 3.1 we have that $q_D(e) \leq \pi_{c_e} \leq \pi_{c_e^*}$, establishing the lower bound.

To observe the upper bound first note that, by definition, $q_\kappa(e)$ is monotonically increasing throughout the deletion sequence. Let $q_\kappa(e)^*$ stand for the value of $q_\kappa(e)$ when the minimal critical scale value c_e^* was first observed. Then by Claim 3.1 we must have that $\pi_{c_e^*} \leq 8 q_\kappa(e)^* \leq 8 q_\kappa(e)$.

□

The correctness of the decremental implementation follows from Claim 4.3 and Claim 3.2. We now explain why the decremental implementation deviates in the definition of the robust connectivity estimates from the static algorithm. Through always focusing on the minimal critical scale ever observed for each edge, we ensure that the robust connectivity estimates are monotonically increasing over time. In turn, this guarantees that the probability that an edge is ever added to the sparsifier is also monotonically increasing over time.

In order to decide if an edge is placed in the output based on its sampling probability p_e, the algorithm only needs to draw a uniform random value $x_e \in [0, 1]$ at initialization and compare x_e with p_e whenever the latter value increases. Thus, an edge may ever enter and leave the output sparsifier at most once. Note that between these two events its weight in the output may also undergo updates, which we count as a separate recourse event. This monotonicity allows us to derive the following fine-grained recourse bound, its proof is deferred to Appendix B.

Claim 4.4. *Let $S = n\varepsilon^{-2} \log^3 n$. During a run of Algorithm 2 the total recourse of the output sparsifier H during the first d deletions to G is at most $O(\min\{m_0, S\} + d)$ in expectation (including initialization).*

Claim 4.4 gives us slightly more than an $O(1)$ amortized recourse bound over m deletions, if the deletion sequence is short. This will prove useful when we turn the algorithm fully dynamic. The following theorem summarizes the guarantees of our decremental sparsifier.

Theorem 4.5. *There exists a dynamic algorithm which with high probability maintains a $(1 \pm \epsilon)$-approximate spectral sparsifier of size $S = O(n\epsilon^{-2} \log^3 n)$ of a graph on n vertices and initially m edges with maximum edge weight W undergoing edge deletions in expected total update time $O\left(m \log(n \log(nW)) \log(nW) \log^3(n) \log\log(nW)\right)$. The algorithm functions against an oblivious adversary and the maintained sparsifier undergoes $O(S + d)$ recourse during the first d deletions processed by algorithm (including initialization).*

5 Fully Dynamic Implementation

We now turn the decremental algorithm of section 4 into a fully dynamic one. We cannot use the fully dynamic LDD of (Forster & Goranci, 2019) directly: its insertion handling is lazy, so a newly inserted edge may be cut with probability larger than β. This is incompatible with the path-wise argument

used in claim 3.1. Instead, we will transform the decremental sparsifier algorithm of Theorem 4.5 into a fully dynamic one using the reduction of (Abraham et al., 2016).

Lemma 4.17 of (Abraham et al., 2016) already offers a black-box tool for making this transformation. However, their analysis does not account for recourse. We will now explain how their fully dynamic bucketing scheme yields a fully dynamic algorithm with $O(1)$ amortized recourse when combined with our fine-grained recourse analysis of Claim 4.4. We then show, that through re-sparsifying the output we can further reduce the size of the resulting sparsifier by an $O(\log n/\log\log n)$ factor at no cost in terms of asymptotic update time or recourse.

The fully dynamic implementation crucially relies on the composability of spectral sparsifiers. Partition the edges of graph G into k graphs $G_1, \dots G_k$ and let H_i be a $(1\pm\epsilon)$-approximate spectral sparsifier of G_i. Then $H = \cup_{i\in[k]} H_i$ serves as a $(1\pm\epsilon)$-spectral sparsifier of G. This holds due to the additivity of graph Laplacians:

$$(1-\varepsilon)L_G = \sum_{i=1}^{k}(1-\varepsilon)L_{G_i} \preceq \sum_{i=1}^{k} L_{H_i} = L_H \preceq \sum_{i=1}^{k}(1+\varepsilon)L_{G_i} = (1+\varepsilon)L_G.$$

(Abraham et al., 2016) uses a bucketing scheme to incorporate edge insertions. We implement a similar approach as follows. For simplicity, assume that the fully dynamic input graph is initially empty. Every $\Theta(n^2)$ update we reset the algorithm. Let $S = O(n\varepsilon^{-2}\log^3 n)$ stand for the size of the decremental sparsifier of Theorem 4.5. The algorithm creates buckets $\mathcal{B}_i \mid i \in [O(\log n)]$ of edges such that bucket i has capacity $2^i S$ and the largest bucket has capacity $\Theta(n^2)$.

The edges in $\mathcal{B}_1$ are placed in the output directly. All other buckets are sparsified using the decremental Algorithm 2 before being placed in the output. Inserted edges are initially placed in $\mathcal{B}_1$. Whenever a bucket $\mathcal{B}_i$ reaches its capacity in terms of inserted edges (not accounting for deletions) its contents are deleted and its live edges are inserted into $\mathcal{B}_{i+1}$. If a batch of edges is inserted into a bucket $\mathcal{B}_i \mid i > 1$ the algorithm re-initializes the decremental sparsifier on its contents. Between two re-initializations the contents of any such bucket only undergo edge deletions.

Bucket $\mathcal{B}_i$ is re-initialized every $\Theta(S2^i)$ updates. As it is responsible for $O(S2^i)$ edges, its amortized update time is the same as Theorem 4.5's. This means that the overhead in update time for the fully dynamic implementation compared to the decremental one is $O(\log n)$, the number of buckets. Similarly, as each bucket contributes $O(S)$ to output sparsifier size, the output of the fully dynamic algorithm is of size $O(S\log n)$.

It remains to argue why the amortized recourse of the output remains $O(1)$ across the $O(\log n)$ buckets. By Theorem 4.5 the output of bucket i undergoes $O(S+d)$ recourse throughout d deletions to its contents. As each edge is always contained in a unique bucket, when deleted, it only contributes to one $O(d)$ term in the recourse bounds across the buckets. As bucket $\mathcal{B}_i$ is rebuilt every $2^i S$ updates, its $O(S)$ recourse term contributes only amortized $O(1/2^i)$ to the output recourse. This means that the amortized recourse across all buckets remains $O(1)$. The following lemma states the properties of the resulting dynamic sparsifier; its formal proof is deferred to Appendix C.1.

Lemma 5.1. *There exists an algorithm which maintains a $(1\pm\epsilon)$-approximate spectral sparsifier of size $O(n\epsilon^{-2}\log^4 n)$ of a graph undergoing edge insertions and deletions with expected amortized update time $O\left(\log(n\log(nW))\log(nW)\log^4(n)\log\log(nW)\right)$ (or $O(\log^6 n\log\log n)$ is $W = \mathrm{poly}(n)$). The algorithms output is correct against an oblivious adversary with high probability and the output sparsifier undergoes expected amortized $O(1)$ recourse per update to the input graph.*

Our final Theorem 1.1 claims a sparsifier that is an $O(\log n/\log\log n)$ factor smaller than that of Lemma 5.1. We achieve this improvement as follows. We first sparsify the input graph using Lemma 5.1. The resulting sparsifier is of size $S = O(n\varepsilon^{-2}\log^4 n)$ and has amortized recourse $O(1)$. We introduce a second layer of sparsification using the decremental algorithm and the bucketing scheme of Lemma 5.1 with a slight twist. We set the capacity of the first bucket to be $\Theta(n\varepsilon^{-2}\log^3 n)$ and the capacity of the largest bucket to be S. In turn, we are only creating $O(\log\log n)$ buckets. This reduces the output size by an $O(\log n/\log\log n)$ factor. This subtle improvement is formally described in Appendix C.2.

6 EXPERIMENTAL EVALUATION

In this section, we demonstrate three claims: our algorithm retains fewer edges than the baseline at equal certified quality, it requires less time per operation, and it is effective on both real-world and synthetic graphs.

6.1 SETUP

To certify quality across all algorithms, we use a single instrument. It computes the condition number $\kappa = \lambda_{\max}/\lambda_{\min}$ of the pencil (L_H, L_G). We use κ instead of the raw deviation $\max\{\lambda_{\max} - 1,\, 1 - \lambda_{\min}\}$ because k is invariant to global rescaling. A setting meets a target κ^* when, over all its certified states (every checkpoint of every graph), at least 95% have $\kappa \le \kappa^*$ and none exceeds $1.15\,\kappa^*$. Our targets are $\kappa \in \{3, 5, 10\}$. We measure κ at mid-stream checkpoints. Appendix F fully details the instrument.

For the experiments, we use ER (Erdős & Rényi, 1959), SBM (Holland et al., 1983), R-MAT (Chakrabarti et al., 2004), and real-world graphs. Three streams evaluate different scenarios: *insert-only*, *teardown* (deletions only after build), and *fully dynamic* (interleaved insertions and deletions). A single machine runs everything (dual Intel Xeon Gold 6130, 251 GiB RAM). Both algorithms are written in C++20 and built at `-O2`. Timing runs are single-threaded and run one at a time on the otherwise idle machine.

6.2 DATASETS

For the synthetic graph families, the grid spans $n \in \{1{,}000, 4{,}000, 16{,}000, 64{,}000\}$ and densities $m/n \in \{30, 50, 80, 150, 250\}$. Edge weights are unit or drawn log-uniformly on $[1, W)$.

We evaluate on the dense k-core (Seidman, 1983; Batagelj & Zaveršnik, 2003) of six real-world networks (Table 1). We select k to guarantee at least 30 edges per vertex, matching our synthetic grid.

Table 1: Real-world networks and their dense k-cores. Five come from SNAP (Leskovec & Krevl, 2014): sx-askubuntu and sx-superuser (Paranjape et al., 2017), loc-Gowalla (Cho et al., 2011), ego-Facebook (McAuley & Leskovec, 2012) and web-NotreDame (Albert et al., 1999); YeastNet v3 comes from its own database (Kim et al., 2014). Source sizes are undirected and deduplicated with timestamps dropped, so they can differ from the dataset pages.

	source			k-core		
network	n	m	k	n	m	m/n
sx-askubuntu	157,222	455,691	40	511	20,897	40.9
sx-superuser	192,409	714,570	54	601	31,883	53.0
loc-Gowalla	196,591	950,327	45	546	23,913	43.8
ego-Facebook	4,039	88,234	30	1,224	55,714	45.5
web-NotreDame	325,729	1,090,108	72	1,367	107,526	78.7
YeastNet v3	5,808	362,421	81	1,981	179,041	90.4

6.3 IMPLEMENTATION AND TUNING

In practice, both algorithms follow the theoretical structure but use different constants to achieve better performance. Every deviation either preserves the theoretical bounds or is explicitly marked as empirical (Appendix D).

We tuned the baseline using two main parameters: the bundle count $t \in \{1, 2, 3\}$ and the sampling probability p. In our runs, the baseline stops compressing and returns its input graph whenever the density falls below 10t edges per vertex per weight class.

Our algorithm is tuned by sweeping only one parameter: the prefactor P, which scales the sampling probabilities. By tuning this configuration, our algorithm achieves strong results for both compression and speed.

6.4 RESULTS

Compression at certified quality. Our algorithm consistently achieves better compression ratios at comparable precision goals for all the synthetic graph families tested as shown in Figure 1.

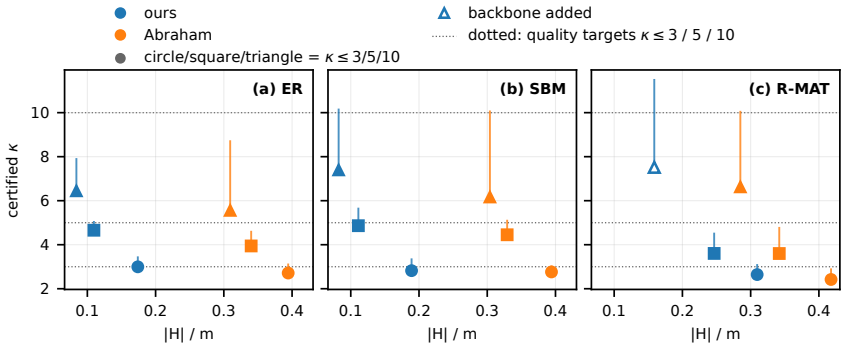

Figure 1: Certified quality versus the fraction of edges kept across three targets on the synthetic families. Markers show the 95th percentile of certified κ. Whiskers reach the worst state. The hollow marker on R-MAT is our algorithm with the connectivity backbone enabled (Appendix D).

Update cost. Our algorithm is faster in all comparisons shown in Figure 2. We measure a median speedup of $3.19\times$ on insertions, $3.01\times$ on fully dynamic streams, and $1.65\times$ on teardown. The same grid at $m/n = 80$ is in Appendix G.

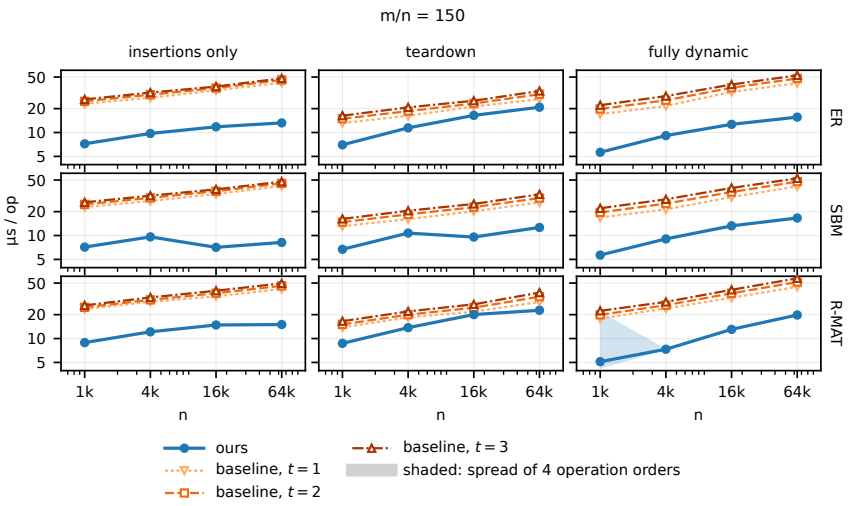

Figure 2: Cost per operation against graph size at $m/n = 150$.

Real graphs. On real-world networks, our algorithm achieves better compression in 46 out of 54 tested scenarios, winning all comparisons in the fully dynamic stream (Fig 3).

Other results. Appendix G details several additional findings. As the edge-weight ratio widens from 1:1 to 12:1, the baseline's sparsifier size grows by a median factor of 1.90, while our algorithm remains entirely flat (Fig 5). Using eight threads accelerates insertions by $1.52\times$ at $n = 4{,}000$ while guaranteeing bit-identical outputs (Fig 6). Scaling each bucket's sampling constant by its peak share of the live graph, rather than using a flat constant, reduces sparsifier size at matched certified quality by 17.8% in simple counter mode and 29.1% under two-tier consolidation (Table 3). Finally, we also compare our algorithm with the static sparsifier GRASS (Fig 7) and the dynamic heuristic dyGRASS (Fig 8).

Reproducibility Statement

Code for our algorithm and for our implementation of the baseline is available at
`https://anonymous.4open.science/r/LDD_full_dynamic_sparsifier-6EBC/`
and
`https://anonymous.4open.science/r/Abraham_fully_dynamic_sparsifier/`.

References

Ittai Abraham, David Durfee, Ioannis Koutis, Sebastian Krinninger, and Richard Peng. On fully dynamic graph sparsifiers. In *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science*, pp. 335–344, 2016. Full version available at arXiv:1604.02094.

Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pp. 1–10, 2013.

Réka Albert, Hawoong Jeong, and Albert-László Barabási. Diameter of the World-Wide Web. *Nature*, 401(6749):130–131, 1999.

Vladimir Batagelj and Matjaž Zaveršnik. An $O(m)$ algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

Yves Baumann and Gernot Zöcklein. Parallel spectral graph sparsification via low diameter decompositions. In *Proceedings of the 38th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 75–86, 2026.

Aaron Bernstein, Jan van den Brand, Maximilian Probst Gutenberg, Danupon Nanongkai, Thatchaphol Saranurak, Aaron Sidford, and He Sun. Fully-dynamic graph sparsifiers against an adaptive adversary. In *Proceedings of the 49th International Colloquium on Automata, Languages, and Programming*, pp. 20:1–20:20, 2022.

Vladimir Braverman, Petros Drineas, Cameron Musco, Christopher Musco, Jalaj Upadhyay, David P. Woodruff, and Samson Zhou. Near optimal linear algebra in the online and sliding window models. In *Proceedings of the 61st Annual IEEE Symposium on Foundations of Computer Science*, pp. 517–528, 2020.

Daniele Calandriello, Ioannis Koutis, Alessandro Lazaric, and Michal Valko. Improved large-scale graph learning through ridge spectral sparsification. In *Proceedings of the 35th International Conference on Machine Learning*, pp. 688–697, 2018.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the 4th SIAM International Conference on Data Mining*, pp. 442–446, 2004.

Ashok K. Chandra, Prabhakar Raghavan, Walter L. Ruzzo, Roman Smolensky, and Prasoon Tiwari. The electrical resistance of a graph captures its commute and cover times. *Computational Complexity*, 6(4):312–340, 1996.

Li Chen, Gramoz Goranci, Monika Henzinger, Richard Peng, and Thatchaphol Saranurak. Fast dynamic cuts, distances and effective resistances via vertex sparsifiers. In *Proceedings of the 61st Annual IEEE Symposium on Foundations of Computer Science*, pp. 1135–1146, 2020.

Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1082–1090, 2011.

Michael B. Cohen, Cameron Musco, and Jakub Pachocki. Online row sampling. *Theory of Computing*, 16(15):1–25, 2020.

David Durfee, Yu Gao, Gramoz Goranci, and Richard Peng. Fully dynamic spectral vertex sparsifiers and applications. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pp. 914–925, 2019.

Paul Erdős and Alfréd Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6: 290–297, 1959.

Zhuo Feng. GRASS: Graph spectral sparsification leveraging scalable spectral perturbation analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4944–4957, 2020.

Sebastian Forster and Gramoz Goranci. Dynamic low-stretch trees via dynamic low-diameter decompositions. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pp. 377–388, 2019. Full version available at arXiv:1804.04928.

Sebastian Forster, Gramoz Goranci, and Ali Momeni. Fully dynamic spectral sparsification for directed hypergraphs. In *Proceedings of the 43rd International Symposium on Theoretical Aspects of Computer Science*, pp. 38:1–38:20, 2026.

Yu Gao, Yang P. Liu, and Richard Peng. Fully dynamic electrical flows: Sparse maxflow faster than Goldberg-Rao. In *Proceedings of the 62nd Annual IEEE Symposium on Foundations of Computer Science*, pp. 516–527, 2021.

Gramoz Goranci and Ali Momeni. Fully dynamic spectral sparsification of hypergraphs. *arXiv preprint arXiv:2502.01421*, 2025.

Gramoz Goranci, Rasmus Kyng, Maximilian Probst Gutenberg, Yibin Zhao, and Gernot Zöcklein. An online sparsification algorithm from the book. *arXiv preprint arXiv:2607.23098*, 2026.

Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137, 1983.

Arun Jambulapati and Aaron Sidford. Ultrasparse ultrasparsifiers and faster Laplacian system solvers. *ACM Transactions on Algorithms*, 21(3), 2025.

Michael Kapralov and Rina Panigrahy. Spectral sparsification via random spanners. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pp. 393–398, 2012.

Michael Kapralov, Aida Mousavifar, Cameron Musco, Christopher Musco, Navid Nouri, Aaron Sidford, and Jakab Tardos. Fast and space efficient spectral sparsification in dynamic streams. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms*, pp. 1814–1833, 2020.

Jonathan A. Kelner and Alex Levin. Spectral sparsification in the semi-streaming setting. *Theory of Computing Systems*, 53(2):243–262, 2013.

Sanjeev Khanna, Huan Li, and Aaron Putterman. Near-optimal linear sketches and fully-dynamic algorithms for hypergraph spectral sparsification. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, pp. 1190–1200, 2025.

Hanhae Kim, Junha Shin, Eiru Kim, Hyojin Kim, Sohyun Hwang, Jung Eun Shim, and Insuk Lee. YeastNet v3: a public database of data-specific and integrated functional gene networks for Saccharomyces cerevisiae. *Nucleic Acids Research*, 42(D1):D731–D736, 2014.

Ioannis Koutis, Gary L. Miller, and Richard Peng. Approaching optimality for solving SDD linear systems. In *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science*, pp. 235–244, 2010.

Ioannis Koutis, Alex Levin, and Richard Peng. Improved spectral sparsification and numerical algorithms for SDD matrices. In *Proceedings of the 29th Symposium on Theoretical Aspects of Computer Science*, pp. 266–277, 2012.

Ioannis Koutis, Alex Levin, and Richard Peng. Faster spectral sparsification and numerical algorithms for SDD matrices. *ACM Transactions on Algorithms*, 12(2), 2016.

Rasmus Kyng, Anup Rao, Sushant Sachdeva, and Daniel A. Spielman. Algorithms for Lipschitz learning on graphs. In *Proceedings of the 28th Conference on Learning Theory*, pp. 1190–1223, 2015.

Rasmus Kyng, Jakub Pachocki, Richard Peng, and Sushant Sachdeva. A framework for analyzing resparsification algorithms. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 2032–2043, 2017.

Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, 2014.

Julian McAuley and Jure Leskovec. Learning to discover social circles in ego networks. In *Advances in Neural Information Processing Systems 25*, pp. 539–547, 2012.

Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

Ashwin Paranjape, Austin R. Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the 10th ACM International Conference on Web Search and Data Mining*, pp. 601–610, 2017.

Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

Daniel A Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pp. 563–568, 2008.

Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.

Daniel A. Spielman and Shang-Hua Teng. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM Journal on Matrix Analysis and Applications*, 35(3):835–885, 2014.

Jan van den Brand, Yu Gao, Arun Jambulapati, Yin Tat Lee, Yang P. Liu, Richard Peng, and Aaron Sidford. Faster maxflow via improved dynamic spectral vertex sparsifiers. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pp. 543–556, 2022.

Yihang Yuan, Ali Aghdaei, and Zhuo Feng. dyGRASS: Dynamic spectral graph sparsification via localized random walks on GPUs. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2025.

Yibin Zhao. Fully dynamic spectral and cut sparsifiers for directed graphs. In *Proceedings of the 53rd International Colloquium on Automata, Languages, and Programming*, 2026.

Chunjiang Zhu, Sabine Storandt, Kam-Yiu Lam, Song Han, and Jinbo Bi. Improved dynamic graph learning through fault-tolerant sparsification. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 7624–7633, 2019.

A STATIC ALGORITHM

The following pseudo-code describes our implementation of the static spectral sparsification framework of Baumann & Zöcklein (2026). The function $\mathsf{LDD}(G, \beta)$ is assumed to return a (β, D)-LDD of G.

Algorithm 1: Static Spectral Sparsification

Input: $G = (V, E, w)$, ε, $M \geq m$
Output: A weighted $(1 \pm \varepsilon)$-spectral sparsifier H

$\kappa \leftarrow \max\{4, \lceil \log_2 n \rceil\}$, $\beta \leftarrow 1/(4\kappa)$, $D \leftarrow \max\{\kappa, \lceil D_{\mathsf{LDD}}(n, \beta) \rceil\}$;
$L \leftarrow \lceil 10 \log(MW) \rceil$, $\pi_i \leftarrow 2^{1-i}$ for $i \in [L]$, $\lambda \leftarrow \lceil 10^7 \ln(nML) \rceil$;
for $i \in [L]$ **do**
  for $j \in [\lambda]$ **do**
    Sample $G_i^j \sim G_{\pi_i}$;
    $\mathcal{X}_i^j \leftarrow \mathsf{LDD}(G_i^j, \beta)$;
$H \leftarrow (V, \emptyset)$;
foreach $e = (u, v) \in E$ **do**
  for $i \in [L]$ **do**
    for $j \in [\lambda]$ **do**
      $Y_i^j(e) \leftarrow \mathbf{1}\{\mathcal{X}_i^j(u) = \mathcal{X}_i^j(v)\}$;
    $Z_i(e) \leftarrow \lambda^{-1} \sum_{j=1}^{\lambda} Y_i^j(e)$;
  $A_e \leftarrow \{i \in [L] : Z_i(e) \geq 17/32\}$;
  $c_e \leftarrow \max(A_e \cup \{1\})$;
  $\widehat{q}_e \leftarrow \pi_{c_e}$;
  $\widehat{R}_e \leftarrow 2D\widehat{q}_e$;
  $p_e \leftarrow \min\{1, 10^6 \varepsilon^{-2} w_e \widehat{R}_e \ln n\}$;
  Draw $x_e \sim \mathrm{Unif}[0, 1]$;
  if $x_e \leq p_e$ **then**
    Add e to H with weight w_e / p_e
return H;

To establish the high probability correctness of the algorithm we use the following form of the Chernoff bound (Mitzenmacher & Upfal, 2005).

Theorem A.1. *Let $X_1, \ldots, X_\lambda$ be independent Bernoulli random variables, let $Z = \lambda^{-1} \sum_{j=1}^{\lambda} X_j$, and let $\mu = \mathbb{E}[Z]$. For every $0 \leq \varepsilon \leq 1$,*

$$\mathbb{P}[Z \leq (1 - \varepsilon)\mu] \leq \exp(-\varepsilon^2 \lambda \mu / 2).$$

Moreover, for every $c \geq \mu$ and $0 \leq \varepsilon \leq 1$,

$$\mathbb{P}[Z \geq (1 + \varepsilon)c] \leq \exp(-\varepsilon^2 \lambda c / 3).$$

The following claim, re-stated for sake of convenience, shows that the critical scale c_e of each edge can be used to approximate its robust connectivity.

Claim 3.1. *At the end of a run of Algorithm 1, with high probability, the following hold simultaneously for every edge $e \in E$ and scale $i \in [L]$:*

$$q_\kappa(e) \leq \pi_i/4 \implies Z_i(e) \geq 17/32, \qquad q_D(e) \geq \pi_i \implies Z_i(e) < 17/32.$$

Furthermore, if any edge e satisfies the above inequalities then $q_D(e) \leq \pi_{c_e} < 8 q_\kappa(e)$.

Proof. Fix $e = (u, v)$ and $i \in [L]$. To observe the first part of the claim, first assume $q_\kappa(e) \leq \pi_i/4$. By Lemma 2.2 this implies $p_\kappa(e, \pi_i) \leq 1/4$. If $\mathrm{dist}_{G_i^j}(u, v) \leq \kappa$, by Observation 2.4 an LDD with edge cut probability β puts (u, v) in the same cluster with probability at least $1 - \kappa\beta$. As the algorithm selects $\beta = 1/(4\kappa)$ this probability is at least $3/4$. Therefore, for any graph $G_i^j \mid j \in [\lambda]$ sampled at scale i we have that

$$\mathbb{E}[Y_i^j] = \mathbb{P}[Y_i^j(e) = 1] \geq \frac{3}{4} \cdot (1 - \mathbb{P}[\text{dist}_{G_i^j}(u, v) > \kappa]) = \frac{3}{4} \cdot (1 - p_\kappa(e, \pi_i)) \geq \frac{9}{16}.$$

The algorithm choose the threshold $17/32 = (1 - 1/18)(9/16)$. As $Z_i(e) = \lambda^{-1} \sum_{j \in [\lambda]} Y_i^j$, by an application of Chernoff's bound (Theorem A.1) we can conclude

$$\mathbb{P}[Z_i(e) < 17/32] \leq \exp\left(-\frac{(1/18)^2(9/16)}{2}\lambda\right) \leq e^{-\lambda/10^4}.$$

If $q_D(e) \geq \pi_i$, then by definition $p_D(e, \pi_i) \geq 1/2$. For a specific sample graph G_i^j two vertices in the same cluster under $\mathcal{X}_i^j$ can have distance at most D in G_i^j. In turn, this implies that $\mathbb{E}[Y_i^j] \leq 1/2$ for all $j \in [\lambda]$. Since $17/32 = (1 + 1/16)(1/2)$, similarly by Chernoff's

$$\mathbb{P}[Z_i(e) \geq 17/32] \leq \exp\left(-\frac{(1/16)^2(1/2)}{3}\lambda\right) \leq e^{-\lambda/10^4}.$$

There are at most ML edge-scale pairs. By the definition of λ, each bad event has probability at most $(nML)^{-10^3}$. A union bound over both events for every pair gives total probability at most n^{-20}.

We now turn to proving the inequalities over π_{c_e}. For the lower bound, if $A_e = \emptyset$, then $c_e = 1$ and $\pi_{c_e} = 1 \geq q_D(e)$. Otherwise, $c_e \in A_e$, so $Z_{c_e}(e) \geq 17/32$. The second implication of Claim 3.1 therefore gives $q_D(e) < \pi_{c_e}$.

For the upper bound, if $c_e = L$, then the choice of L, the inequality $M \geq m$, and Lemma 2.2 give $\pi_{c_e} = \pi_L \leq 1/(4MW) \leq q_\kappa(e)$. Otherwise, $c_e < L$ and the definition of c_e implies $c_e + 1 \notin A_e$. Thus $Z_{c_e+1}(e) < 17/32$, and the first implication of Claim 3.1 gives $q_\kappa(e) > \pi_{c_e+1}/4 = \pi_{c_e}/8$.

□

The following claim, re-stated for sake of convenience, shows that assuming the robust connectivity estimates $\widehat{q}_e$ are accurate Algorithm 1 returns a suitable sparsifier, thereby finishing the proof of correctness in combination with Claim 3.1.

Claim 3.2. *Assume that at the end of the run of Algorithm 1 estimates $\widehat{q}_e$ satisfy $q_D(e) \leq \widehat{q}_e \leq 8q_\kappa(e)$ for all $e \in E$. Then the output graph H is a $(1 \pm \epsilon)$-approximate spectral sparsifier of G of size $O(nD\epsilon^{-2} \log n)$ with high probability.*

Proof. Under the assumption, by lemma 2.2, $\widehat{R}_e = 2D\widehat{q}_e \geq 2Dq_D(e) \geq R_e$ for all $e \in E$. The final sampling probabilities are $p_e = \min\{1, 10^6\varepsilon^{-2}w_e\widehat{R}_e \ln n\}$. Thus, by theorem 2.5 the output H is a $(1 \pm \epsilon)$-approximate spectral sparsifier with high $(1 - 1/\operatorname{poly}(n))$ probability.

Moreover, $\kappa \geq \log_2 n$ implies $n^{O(1/\kappa)} = O(1)$, and hence

$$\sum_{e \in E} w_e\widehat{R}_e \leq 16D \sum_{e \in E} w_e q_\kappa(e) \leq 64Dn^{1+O(1/\kappa)} = O(Dn).$$

Consequently, $\mu = \sum_e p_e = O(nD\varepsilon^{-2} \ln n)$. By Chernoff's bound theorem A.1 we have that

$$\mathbb{P}[|E(H)| \geq 2 \cdot \max\{\mu, 10^2 \ln n\}] \leq e^{-(\max\{\mu, 10^2 \ln n\})/3} \leq n^{-30}.$$

Since $\max\{\mu, 10^2 \ln n\} = O(nD\varepsilon^{-2} \ln n)$ we have that $|H| = O(nD\epsilon^{-2} \log n)$ with high probability.

□

B Decremental Implementation

The following pseudo-code describes our decremental implementation. Let $G_0 = (V, E_0, w)$ be the initial graph and $m_0 = |E_0|$. We run algorithm 1 with the fixed edge bound $M = m_0$. Thus the number of scales does not change as edges are deleted. For every $i \in [L]$ and $j \in [\lambda]$, all edge-sampling defining G_i^j are drawn once at initialization, and $\mathcal{X}_i^j$ is maintained by the algorithm of lemma 4.1.

For every edge $e = (u, v)$ currently in the graph we explicitly maintain the variables $Y_i^j(e), Z_i(e), c_e, \widehat{q}_e, \widehat{R}_e$, and p_e. In addition, we also maintain the ordered set $A_e = \{i : Z_i(e) \geq 17/32\}$. Whenever a vertex changes clusters in one LDD copy, we scan its live incident edges in the original graph and update these variables, not only the edges of the sampled graph G_i^j.

Algorithm 2: Decremental Spectral Sparsification

Input: $G_0 = (V, E_0, w), \varepsilon$
$G \leftarrow G_0, m_0 \leftarrow |E_0|$;
Initialize H and the relevant variables with Algorithm 1 (G_0, ε, m_0) ;
$c_e^* \leftarrow c_e$ for all $e \in E$;
Delete(e);
Remove e from G and H ;
$F \leftarrow \emptyset$;
foreach (i, j) *such that* $e \in G_i^j$ **do**
  Remove e from G_i^j ;
  Update $\mathcal{X}_i^j$;
  foreach *Vertex* v *changing clusters in* $\mathcal{X}_i^j$ **do**
    foreach $f = (u, v)$ *incident on* v *in* G **do**
      $y \leftarrow \mathbf{1}\{\mathcal{X}_i^j(v) = \mathcal{X}_i^j(u)\}$;
      if $y \neq Y_i^j(f)$ **then**
        $Z_i(f) \leftarrow Z_i(f) + (y - Y_i^j(f))/\lambda$;
        $Y_i^j(f) \leftarrow y$;
        Update A_f according to the new $Z_i(f)$ value ;
        if *scale* i *enters or leaves* A_f **then**
          Update c_f ;
          $F \leftarrow F \cup \{f\}$;
foreach $e \in F$ **do**
  if $c_e < c_e^*$ **then**
    $c_e^* \leftarrow c_e$;
    $\widehat{q}_e \leftarrow \pi_{c_e^*}$;
    Update $\widehat{R}_e, p_e$ as in Algorithm 1;
    Keep e in H exactly when $x_e \leq p_e$, with weight w_e/p_e;

The crucial difference with respect to the static Algorithm 1 is that robust connectivity estimates $\widehat{q}_e$ are derived from minimal critical scales observed prior c_e^* instead of current values. Claim 4.3 shows this change still ensures that the estimates fit the requirements of Claim 3.2 implying the correctness of the output. We now turn to formally proving the total update time bound, which we re-state for sake of presentation.

Claim 4.2. *For $m_0 \geq 1$, the expected total time of algorithm 2 is*

$$O\left(m_0 \log(n \log(nW)) \log(nW) \log^3(n) \log\log(nW)\right),$$

or $O(m_0 \log^5(n) \log\log n)$ if $W = \mathrm{poly}(n)$.

Proof. Fix one copy $\mathcal{X}_i^j$ and let $N_i^j(v)$ be the number of times $v \in V$ changes clusters throughout the deletion sequence. The expected total work scanning live incident edges in this copy is

$$O\left(\sum_{v \in V} \deg_{G_0}(v) \mathbb{E}[N_i^j(v)]\right) = O(m_0 \beta^{-1} \log^2 n)$$

by Lemma 4.1. The LDD update work satisfies the same bound. Multiplying by λL copies and charging $O(\log L)$ for updating an ordered set A_e gives

$$O(m_0 \lambda L \beta^{-1} \log^2 n \log L).$$

Using a membership flag and a list for F, collecting affected edges and updating each running maximum adds only constant work per existing critical-index update. Initialization, including sampling, constructing the LDDs and computing the maintained variables, is covered by the same bound. Substituting λ, L and β, and using $M = m_0 \leq n^2$, gives the claimed bounds.

□

Finally, we turn to proving our fine-grained recourse bound of the decremental algorithm, re-stated for sake of convenience.

Claim 4.4. *Let $S = n\varepsilon^{-2} \log^3 n$. During a run of Algorithm 2 the total recourse of the output sparsifier H during the first d deletions to G is at most $O(\min\{m_0, S\} + d)$ in expectation (including initialization).*

Proof. Fix an edge $e \in E$. By definition, the algorithms estimate $\widehat{q}_e$ must be monotonically increasing over time. This implies that the probability p_e with which is retained in H is also monotonically increasing until its deletion. Let $p_{e,1}, p_{e,2}, \ldots, p_{e,k}$ be the monotonically increasing distinct sampling probabilities assigned to e during the first d deletions.

First consider the number of insertions into H during the first d deletions and initialization. Since p_e may only increase if c_e increases, whenever it does increase, it must at least double. Therefore, $\sum_{i \in [k]} p_{e,i} \leq 3p_{e,k}$. At initialization, e is present in the graph with probability at most $p_{e,1}$. Whenever p_e increases, e enters the graph or changes its weight if x_e is upper bounded by the current value of p_e, so p_e times in expectation. Therefore, e contributes in expectation

$$p_{e_1} + \sum_{i \in [k]} p_{e,i} + p_{e,k} \leq 5p_{e,k}$$

recourse to H through insertions. For any edge e let p_e^* stand for its p_e value after the d-th deletion. Let E_d stand for the set of edges deleted during the first d deletions. Then the total recourse from insertions in expectation is upper bounded by:

$$5 \sum_{e \in E} p_e^* \leq 5 \sum_{e \in E_d} 1 + 5 \sum_{e \in E_0 \setminus E_d} p_e^* = O(d + \min\{m_0, S\}),$$

where the last equality follows from the upper bound on the size of the sparsifier at any given time (Claim 3.2). The deletions of E_d contribute at most d to the recourse in addition.

□

C Deferred Proofs of Section 5

C.1 Proof of Lemma 5.1

For simplicity of notation, let $S = \lceil n\varepsilon^{-2} \log^3 n \rceil$, and write τ for the factor multiplying m_0 in Claim 4.2, that is assume a decremental instance initialized on m edges takes $O(m\tau)$ expected total time for the decremental algorithm.

We divide the update sequence into phases, each consisting of n^2 edge insertions. At the beginning of a phase, we place all currently present edges into a base bucket and initialize a decremental sparsifier on its contents. In addition, we maintain an initially empty unsparsified bucket and buckets $\mathcal{B}_0, \ldots, \mathcal{B}_K$ of capacities $S, 2S, \ldots, 2^K S$, where $K = \lceil \log_2(n^2/S) \rceil = O(\log n)$.

Every inserted edge first enters the unsparsified bucket. After this bucket has received S insertions, we transfer its surviving edges into the other buckets according to a binary-counter rule. Specifically, let

i be the smallest index of an unoccupied bucket. We combine the surviving edges of the unsparsified bucket and $\mathcal{B}_0, \ldots, \mathcal{B}_{i-1}$, place them into $\mathcal{B}_i$, and initialize a new decremental instance there. The old instances are discarded and their buckets become unoccupied. Each occupied $\mathcal{B}_i$ represents exactly $2^i b$ insertions, including insertions of edges that have subsequently been deleted. Thus bucket occupancy is determined by insertion counts rather than the number of surviving edges.

Every input deletion is handled by the unique bucket containing the deleted edge. If that bucket is sparsified, we pass the deletion to its decremental instance; otherwise, we remove the edge directly. After n^2 insertions, we discard all instances and start a new phase on the current graph. Discarding an instance removes its remaining output edges directly, without processing its surviving input edges as deletions. We represent a weight change of an output edge by one deletion and one insertion.

At every completed update, the buckets partition the current input edges. We output the unsparsified bucket together with the sparsifiers of the other buckets. The bucket schedule and the deletion sequence received by each instance depend only on the oblivious update sequence, and we use independent randomness for every instance. Consequently, Theorem 4.5 applies to every bucket. At any fixed update, a union bound over the $O(\log n)$ active instances and the composability property give a $(1 \pm \varepsilon)$-approximate spectral sparsifier with high probability. Its size is $O(S + S \log n) = O(n\varepsilon^{-2} \log^4 n)$.

We next bound recourse over an arbitrary prefix containing I insertions and D deletions, starting from a graph with m_0 edges. By Claim 4.4, an instance initialized on m edges and receiving d input deletions has expected recourse $O(\min\{m, S\} + d)$. This bound also includes discarding the instance: every output edge removed at disposal must have been inserted into that instance's output earlier, so disposal at most doubles its recourse. For accounting purposes, we also charge disposal of the instances still active at the end of the prefix.

Let N_i be the number of instances initialized in bucket $\mathcal{B}_i$ during the prefix. The binary-counter rule gives $N_i \le I/(2^i b)$, including across phase boundaries, since the insertion buckets start empty in every phase. Hence, the sum of their $O(S)$ recourse contributions is $O(I)$. The initial base instance contributes $O(\min\{m_0, S\})$ apart from input deletions, and there are at most I/n^2 subsequent base instances, each contributing $O(S)$. Every genuine input deletion contributes to the d-term of at most one instance. Finally, each edge entering the unsparsified bucket is inserted into its output once and removed once, so this bucket contributes $O(I)$ recourse. Altogether,

$$\mathbb{E}[\text{total recourse}] = O\left(\min\{m_0, S\} + \frac{IS}{n^2} + S\sum_{i=0}^{K} \frac{I}{2^i S} + I + D\right) = O(\min\{m_0, S\} + I + D).$$

This gives $O(1)$ expected amortized recourse, with initialization charged to the initial edges. We now turn to bounding the update time. Each instance in $\mathcal{B}_i$ is initialized on at most $2^i S$ edges and therefore takes $O(2^i S\tau)$ expected total time. Summing over its at most $I/(2^i S)$ instances gives $O(I\tau)$ per bucket. The initial base instance costs $O(m_0\tau)$, and subsequent base instances cost $O(n^2\tau)$ once per n^2 insertions. Maintaining the unsparsified bucket and transferring edges are covered by the same bounds. Therefore,

$$\mathbb{E}[\text{total time}] = O\left(m_0\tau + I\tau + \sum_{i=0}^{K} \frac{I}{2^i S}\, 2^i S\tau + I + D\right) = O(m_0\tau + (I + D)\tau \log n).$$

Substituting the value of τ from Claim 4.2 proves the claimed expected amortized update-time bound.

C.2 Proof of Theorem 1.1

Set $\eta = \varepsilon/3$, let $S = n\lceil \eta^{-2} \log^3 n \rceil$. If $S \ge n^2$, we maintain G itself. Otherwise, we first apply Lemma 5.1 with approximation parameter η, and denote its output by H_1. For a sufficiently large constant C, let $B = \min\{n^2, \lceil CS \log n \rceil\}$. At each fixed update, H_1 is an η-approximate spectral sparsifier of G with at most B edges with high probability.

We apply a second layer of sparsification to H_1, using independent randomness. After each update to G, we pass the changes in H_1 to the second layer, performing all edge deletions before insertions. A weight change is passed as a deletion followed by an insertion, meaning that H_1 is a fully dynamic graph undergoing $O(1)$ expected amortized updates per update to G.

The second layer uses the same base bucket, unsparsified bucket of capacity S, and binary buckets of capacities $2^i S$ as in the proof of Lemma 5.1 and its output sparsifier is H_2. Every decremental instance of the second layer uses approximation parameter η. We modify only the phase boundaries. If a phase starts with m input edges, set $P = \max\{B, m\}$ and end the phase after either P insertions or $\lceil P/2 \rceil$ deletions, whichever occurs first. At its end, discard the bucket instances and start a new phase on the current input graph.

This choice ensures that the number of buckets reflects the current input size, even if an earlier first-layer output exceeded its size bound. Let $F = (V, E_{H_1})$ be the current input graph of the second layer. During a phase with $P > B$, the phase started with exactly P edges and fewer than $P/2$ deletions have occurred. Hence $|E_{H_1}| > P/2$. After a phase restart, $P = \max\{B, |E_{H_1}|\}$ by definition. Consequently, whenever $|E_{H_1}| \leq B$, the current phase has $P \leq 2B$ and uses $O(1 + \log(B/S)) = O(\log \log n)$ binary buckets. In turn this implies that $|E_{H_2}| = O(n\varepsilon^{-2} \log^3 n \log \log n)$.

Due to the high probability correctness guarantee of Lemma 5.1 and the independent randomness used in the second layer of sparsification, with high probability at each fixed update, H_2 is an η-approximate spectral sparsifier of H_1. Since $(1-\eta)^2 \geq 1-\varepsilon$ and $(1+\eta)^2 \leq 1+\varepsilon$, it is a $(1 \pm \varepsilon)$-approximate spectral sparsifier of G.

The recourse accounting for the binary buckets is unchanged compared to the derivation in Lemma 5.1. An instance in a bucket of capacity $2^i S$ contributes $O(S + d)$ expected recourse, including disposal, and at most one such instance is created per $2^i S$ insertions. Summing their $O(S)$ contributions over the levels gives $O(1)$ per insertion, while each input deletion contributes to the d-term of exactly one instance. The unsparsified bucket also has constant amortized recourse.

Let $\tau(m, n, W)$ stand for the expected amortized update time bound of Theorem 4.5. Note that by construction edge weights of H_1 are in $O(\mathrm{poly}(n, W))$. The second layer maintains $O(\log \log n)$ buckets, all but one of them being sparsified by Theorem 4.5. Similarly to the proof of Lemma 5.1, a second-layer bucket of size N has its decremental sparsifier re-initialized every $O(N)$ updates to H_1. This implies that the expected amortized update time cost of maintaining the sparsifier of a bucket of size N of the second layer is $O(\tau(m, n, \mathrm{poly}(n, W)))$. Since the second layer has just $O(\log \log n)$ buckets this makes its update time asymptotically smaller than that of the first.

Remark C.1. *The reader might reasonably ask, why don't we introduce more than* 2*-layers of sparsification as the second seems to give an overall improvement of sparsifier size. Taken to the extreme, with* $O(\log^* n)$ *layers of sparsification, one can remove the* $\log \log n$ *term from the sparsifier size at the cost of adding a* $2^{O(\log^* n)}$ *term to the update time and recourse bound. We did not optimize the* $o(\log \log n)$ *terms for sake of simplicity.*

D Declared Deviations from the Published Algorithms

This appendix section lists every deviation we made from the theory to make the algorithms usable in practice.

What we did not change. The overall structure of the baseline from Abraham et al. (2016) and of our algorithm follows the theoretical framework. What we changed are some mechanisms and constants to improve them in practice. However, several critical constants described by the theory are unchanged. The acceptance threshold of 17/32, the constant inside the cut parameter, and the cluster radius rule $\kappa = \lceil \log_2 n \rceil$ are left untouched.

The Abraham baseline

The update counter (declared scope). The published algorithm restarts the insertion counter every n^2 operations. Since our tested streams never reach this number of operations, this mechanism was not implemented.

The tuned parameters (empirical). The theory uses a very large number of bundles t to make the proofs work. For our tested graph topologies and sizes, we use $t \in \{1, 2, 3\}$. Furthermore, we also tune the sampling probability p, which the paper fixes at $1/4$.

OUR ALGORITHM

Repetitions per scale (empirical). In theory, the number of independent decompositions per scale asked is very high. For our graph sizes, this value would be around 10^6. In practice, we use 8 repetitions. The acceptance criteria used for these repetitions use the same acceptance threshold as the theoretical algorithm.

The sampling constant (empirical). The formula used in the paper for the probability constant is $p_e = \min(1, 10^4 \varepsilon^{-2} w_e \hat{R}_e \ln(n/\delta))$. In the code, we used $p_e = \min(1, c\Lambda w_e \hat{R}_e / \varepsilon^2)$. The structure is the same, but we replaced the 10^4 with a tuned constant c (which we report rescaled as P). All the other variables in the sampling formula are implemented as the paper dictates.

The scale floor (empirical). Our algorithm computes scales from a certain floor derived from the median degree and mean weight of the block. The floor level decision is empirical, but the guarantees are not broken by this. In fact, an edge whose decision would have landed below the floor is kept in the final sparsifier, never dropped.

The adaptive prefactor (empirical). The fixed rule assigns the exact same sampling constant to every bucket. The adaptive rule scales each bucket's constant by its peak share of the live graph. For states with just a single bucket, the two rules are perfectly identical. We compare their performance at matched certified quality in Appendix G (Tables 3 and 4).

The re-sparsification layer (omitted). Our theoretical analysis introduces a second re-sparsification layer to shave a logarithmic factor off the final size. We drop this mechanism entirely in the practical engine, as empirical testing proved the extra overhead simply wasn't worth.

The insertion buffer (guarantee-preserving). The edges inside the buffer enter the sparsifier H at their original weight. The buffer's capacity is derived from the block size at which compression stops paying off, capped so that the buffer never holds more than a quarter of the expected edges. Since the buffer and our algorithm blocks contain disjoint sets of edges and they are both spectral sparsifiers, the theoretical guarantee is respected.

The connectivity backbone (guarantee-preserving, optional). Random sampling can sometimes drop the last edge of a low-degree vertex. This disconnects the sparsifier even if the underlying graph remains connected, blowing the certified κ up to infinity (Appendix F). This failure occurs on 0.56% of our certified states, always on skewed graphs. The backbone fixes this. Each block computes a maximum spanning forest of its edges, scored by the algorithm's own leverage estimates, and injects the forest into the sparsifier at true weight (without duplicating existing edges). Because keeping an edge with probability one strictly increases its sampling probability, this injection cannot weaken the theoretical sampling guarantees. The backbone is disabled by default.

ON BOTH ALGORITHMS

Two-tier consolidation (guarantee-preserving). By default, both algorithms split live edges into buckets using the binary counter from Abraham et al. (2016). To sparsify better, we want to have bigger buckets. Two-tier consolidation adds a single reserved bucket on top of the small ones. Whenever the edge count in the ordinary buckets hits a fixed multiple of the reserved bucket's size, a freshly built reserved bucket absorbs every live edge. Because every instance remains a valid sparsifier of the edges it holds, their union is still a valid sparsifier of the whole graph.

E CONFIGURATION OF BOTH ALGORITHMS

Table 2: Every setting that was searched for, per family, size, density and counter mode, at each of the three targets. P is our prefactor, reported for the adaptive rule, which is the configuration all results use. The baseline's columns give its bundle count and sampling probability. Cells not listed reuse a setting from this table rather than one searched on their own data. † our search stopped on the last setting that still met the criterion, so it sits at the edge of its own evidence.

				P (ours)			baseline: t, p		
Family	Mode	n	m/n	$\kappa \leq 3$	$\kappa \leq 5$	$\kappa \leq 10$	$\kappa \leq 3$	$\kappa \leq 5$	$\kappa \leq 10$
ER	binary	1,000	30	10.00	6.50	5.00	1, 0.14	1, 0.06	1, 0.04
ER	binary	1,000	150	14.75†	9.00	6.75	1, 0.34	1, 0.18	1, 0.10
ER	binary	4,000	30	9.25	6.75	5.50	1, 0.08	1, 0.04	1, 0.02
ER	binary	4,000	150	16.00	10.25	6.75	1, 0.28	1, 0.14	1, 0.08
ER	binary	16,000	30	9.50	7.00	5.75	1, 0.12	1, 0.06	1, 0.04
ER	binary	16,000	150	19.50	11.50	7.25	1, 0.40	1, 0.20	1, 0.08
ER	TwoTier	1,000	30	8.75	6.00	4.50	1, 0.24	1, 0.12	1, 0.04
ER	TwoTier	1,000	150	16.25	8.25	6.25	1, 0.34	1, 0.16	1, 0.10
ER	TwoTier	4,000	30	8.75†	6.75	5.25	1, 0.20	1, 0.10	1, 0.04
ER	TwoTier	4,000	150	18.00	9.50	5.75	1, 0.28	1, 0.14	1, 0.08
ER	TwoTier	16,000	30	10.25	7.25	5.50	1, 0.32	1, 0.14	1, 0.06
ER	TwoTier	16,000	150	19.50	10.75	6.50	1, 0.46	1, 0.24	1, 0.14
SBM	binary	1,000	30	8.50	6.75	4.50	1, 0.16	1, 0.06	1, 0.04
SBM	binary	1,000	150	16.00	9.00	6.50	1, 0.36	1, 0.16	1, 0.10
SBM	binary	4,000	30	9.25†	7.25	5.00†	1, 0.08	1, 0.04	1, 0.02
SBM	binary	4,000	150	17.25	10.25	5.75	1, 0.28	1, 0.16	1, 0.06
SBM	TwoTier	1,000	30	8.50	6.50	3.75	1, 0.26	1, 0.12	1, 0.04
SBM	TwoTier	1,000	150	16.00	8.50	5.50	1, 0.32	1, 0.18	1, 0.12
SBM	TwoTier	4,000	30	8.50	8.00	8.00	1, 0.26	1, 0.10	1, 0.04
SBM	TwoTier	4,000	150	17.25†	9.75	6.00	1, 0.28	1, 0.16	1, 0.08
R-MAT	binary	1,000	30	31.00	24.25	18.25	1, 0.66	1, 0.22	1, 0.10
R-MAT	binary	1,000	150	29.25	22.50	16.50	1, 0.48	1, 0.30	1, 0.14
R-MAT	binary	4,000	30	25.00	21.50	19.00	1, 0.66	1, 0.38	1, 0.14
R-MAT	binary	4,000	150	39.25	29.50	21.00	2, 0.22	2, 0.12	1, 0.18
R-MAT	TwoTier	1,000	30	21.50	19.00	16.50	2, 0.24	2, 0.10	2, 0.04
R-MAT	TwoTier	1,000	150	35.50	27.25	22.00	1, 0.50	1, 0.28	1, 0.14
R-MAT	TwoTier	4,000	30	24.00	24.00	20.00	2, 0.28	2, 0.12	2, 0.06
R-MAT	TwoTier	4,000	150	42.25	31.50	28.00	1, 0.60	2, 0.14	1, 0.16

F CERTIFICATION METHODOLOGY

This appendix presents the instrument used to measure the quality metrics reported in Section 6.

One instrument. The certifier is a single script, pinned by its hash, used for our two algorithms, GRASS, and dyGRASS.

What is computed. To extract the extreme eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$, we solve the generalized eigenvalue problem for the graph G and its sparsifier H. From these values, we derive the condition number $\kappa = \lambda_{\max}/\lambda_{\min}$.

The nullspace. The rows of the Laplacian matrix sum to zero, which means it is singular: every connected component of the graph produces a null eigenvalue. Since computing κ requires dividing by $\lambda_{\min}$, these zero eigenvalues would break the evaluation while giving no useful information. We identify these zero directions combinatorially from the connected components of G, and remove them

before solving. For traceability, we record the dimension of the reduced problem for each certified state.

Disconnected states. During a dynamic stream, mainly on skewed graphs, the sparsifier H can become disconnected while G is still connected. In these cases, the reported κ is ∞. To detect this, we compare the number of connected components of H and G. We do not hide this as a solver failure; the state is explicitly marked as disconnected.

Two solver paths. For graphs with fewer than 6000 vertices, we solve the reduced problem densely, which gives the exact extreme eigenvalues. This method scales cubically with the number of vertices, so for larger graphs the instrument uses an iterative solver (ARPACK, via SciPy) at a convergence tolerance of 10^{-11}. The dense path is exact to machine precision, while the iterative path is accurate to the convergence tolerance above. We tested the two paths against each other and found that they agree to within 2×10^{-14}.

When we certify. Quality is measured at several mid-stream checkpoints, not only on the fully built graph. At each checkpoint, the current H is certified against the current G. When two algorithms are compared, we certify at the same checkpoints, so both are evaluated on the same states.

G Additional Results

G.1 Real graphs

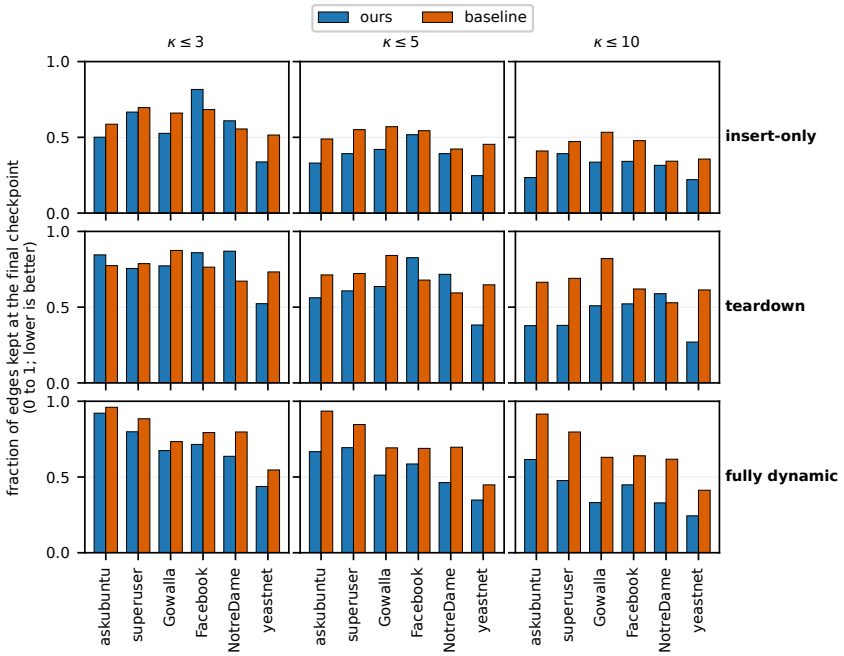

Figure 3: Fraction of live edges kept at the final checkpoint on the six dense k-cores. Both algorithms operate at their independently optimal setting to meet the quality goal.

G.2 Cost at the second density

Figure 4 repeats the timing grid of the main text at $m/n = 80$. The conclusion is the same: our algorithm is faster, delivering median speedups of $4.59\times$ on insertions, $2.57\times$ on fully dynamic, and $1.69\times$ on teardown.

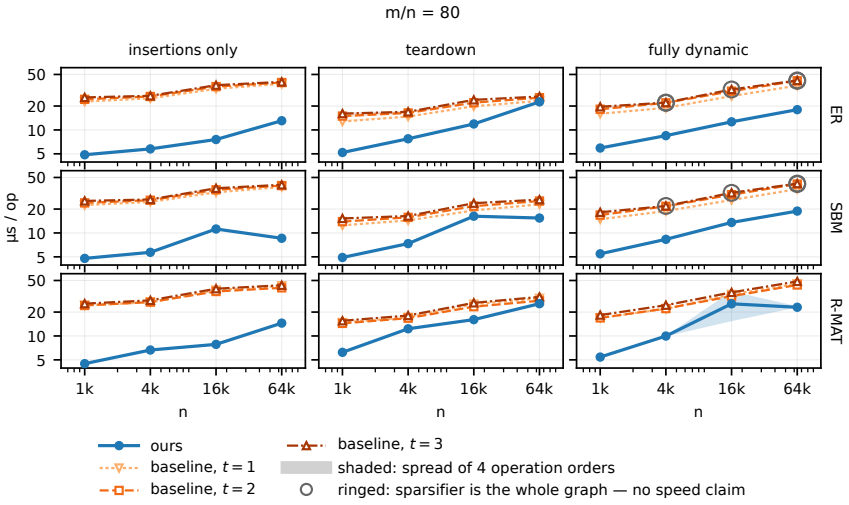

Figure 4: Cost per operation against graph size at $m/n = 80$.

G.3 Weight distribution

Figure 5 evaluates the edge-weight ratio across 1:1, 4:1, and 12:1. As the ratio widens, the baseline's sparsifier size grows from 0.31 to 0.58 edges per live edge. In contrast, our sparsifier remains stable, shifting slightly from 0.24 to 0.22. Both algorithms operate at their independently tuned settings for the $\kappa \leq 3$ target.

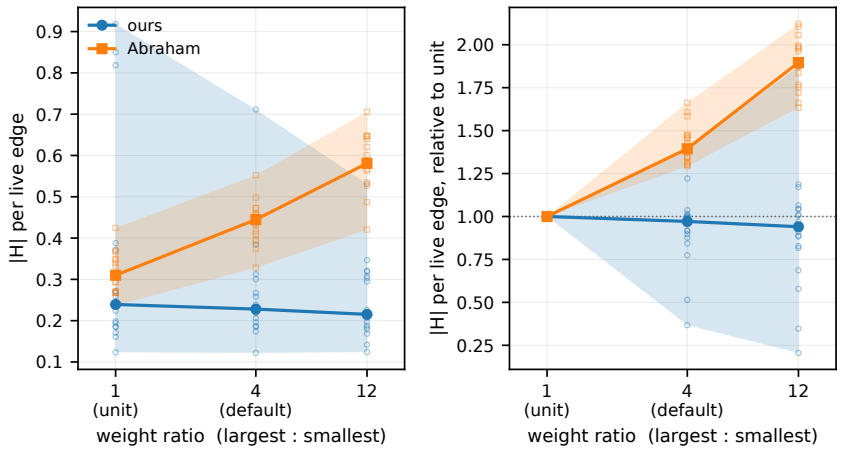

Figure 5: Sparsifier size per live edge against the edge-weight ratio, absolute (left) and relative to the unit-weight law (right).

G.4 Thread scaling

Because we derive random coins deterministically from the edge and block, our output is strictly independent of thread scheduling. Figure 6 tracks wall-time speedup across thread counts on ER graphs. At eight threads, insertions speed up by $1.52\times$ at $n = 4{,}000$ and $1.45\times$ at $n = 16{,}000$.

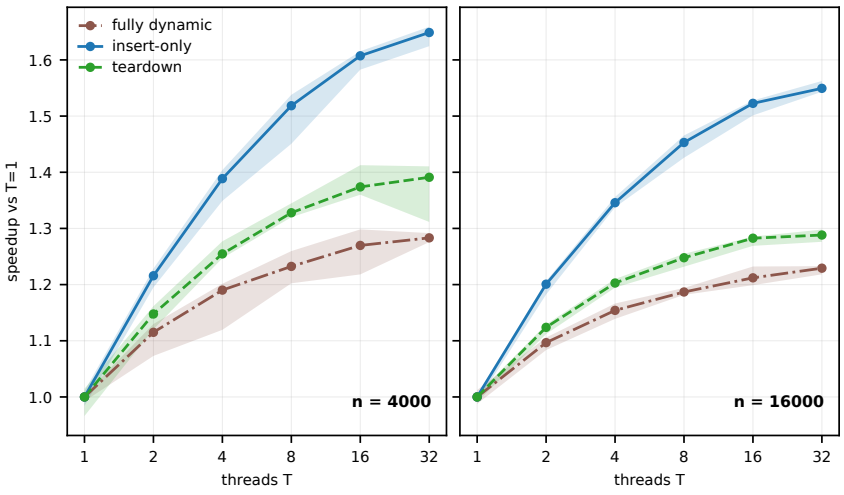

Figure 6: Speedup against thread count on ER graphs at $n = 4{,}000$ and $n = 16{,}000$.

G.5 The adaptive prefactor

The prefactor rule decides how we set the sampling constant for each bucket. The fixed rule gives every bucket the exact same constant. The adaptive rule scales the constant by the bucket's peak share of the live graph.

Table 3 compares the two approaches at matched certified quality. Table 4 lists every underlying pair. When a state holds only one bucket, the adaptive rule does nothing.

Table 3: Summary of the matched-quality comparison of the two prefactor rules. Each arm is tuned to its own frontier for the target in the same row, so the two are compared at equal certified quality rather than at equal settings.

Target	Mode	Pairs	Mean change	Median change
$\kappa \leq 3$	binary	7	-17.80%	-22.40%
$\kappa \leq 3$	TwoTier	4	-29.08%	-29.26%
$\kappa \leq 5$	binary	5	-18.79%	-22.88%
$\kappa \leq 5$	TwoTier	5	-26.91%	-27.66%
$\kappa \leq 10$	binary	5	-12.99%	-23.53%
$\kappa \leq 10$	TwoTier	4	-16.59%	-27.61%

Table 4: Every matched-quality pair behind Table 3. P is the tuned prefactor each arm's own search selected for that target and $|H|$ is the median sparsifier size over the states that frontier names.

Target	Family	Mode	n	P adaptive	P fixed	$\|H\|$ adaptive	$\|H\|$ fixed	Change
$\kappa \leq 3$	ER	binary	1,000	14.75	13.75	26,092.5	33,626.0	−22.40
$\kappa \leq 3$	ER	binary	4,000	16.00	15.50	114,018.5	147,989.0	−22.95
$\kappa \leq 3$	ER	binary	16,000	19.50	20.25	536,378.5	700,310.0	−23.41
$\kappa \leq 3$	R-MAT	binary	1,000	29.25	29.25	46,430.0	54,434.0	−14.70
$\kappa \leq 3$	R-MAT	binary	4,000	39.25	39.25	230,678.0	230,678.0	0.00
$\kappa \leq 3$	SBM	binary	1,000	16.00	14.00	28,366.5	34,292.5	−17.28
$\kappa \leq 3$	SBM	binary	4,000	17.25	17.25	119,658.5	157,181.5	−23.87
$\kappa \leq 3$	ER	TwoTier	1,000	16.25	16.00	32,966.0	47,378.0	−30.42
$\kappa \leq 3$	ER	TwoTier	4,000	18.00	18.00	143,715.5	206,997.5	−30.57
$\kappa \leq 3$	SBM	TwoTier	1,000	16.00	15.25	32,839.0	45,669.5	−28.09
$\kappa \leq 3$	SBM	TwoTier	4,000	17.25	16.25	138,759.0	190,659.0	−27.22
$\kappa \leq 5$	ER	binary	1,000	9.00	7.75	16,440.5	21,296.0	−22.80
$\kappa \leq 5$	ER	binary	4,000	10.25	9.25	74,856.0	98,994.5	−24.38
$\kappa \leq 5$	R-MAT	binary	1,000	22.50	16.00	36,978.5	36,698.5	+0.76
$\kappa \leq 5$	SBM	binary	1,000	9.00	8.00	16,672.0	22,121.5	−24.63
$\kappa \leq 5$	SBM	binary	4,000	10.25	9.00	73,823.0	95,721.5	−22.88
$\kappa \leq 5$	ER	TwoTier	1,000	8.25	8.00	17,507.5	27,326.0	−35.93
$\kappa \leq 5$	ER	TwoTier	4,000	9.50	7.25	110,294.5	152,468.5	−27.66
$\kappa \leq 5$	R-MAT	TwoTier	4,000	31.50	31.50	222,770.0	282,031.0	−21.01
$\kappa \leq 5$	SBM	TwoTier	1,000	8.50	7.50	18,467.0	26,368.5	−29.97
$\kappa \leq 5$	SBM	TwoTier	4,000	9.75	6.50	111,240.5	139,044.5	−20.00
$\kappa \leq 10$	ER	binary	1,000	6.75	4.50	12,592.0	13,707.0	−8.13
$\kappa \leq 10$	ER	binary	4,000	6.75	5.75	50,927.5	66,594.0	−23.53
$\kappa \leq 10$	R-MAT	binary	4,000	21.00	16.50	166,705.5	131,002.0	+27.25
$\kappa \leq 10$	SBM	binary	1,000	6.50	5.50	12,334.5	16,202.0	−23.87
$\kappa \leq 10$	SBM	binary	4,000	5.75	6.00	43,229.5	68,253.5	−36.66
$\kappa \leq 10$	ER	TwoTier	1,000	6.25	5.50	13,517.0	20,478.0	−33.99
$\kappa \leq 10$	ER	TwoTier	4,000	5.75	4.50	67,757.0	53,119.5	+27.56
$\kappa \leq 10$	R-MAT	TwoTier	4,000	28.00	28.00	208,880.5	265,140.0	−21.22
$\kappa \leq 10$	SBM	TwoTier	1,000	5.50	5.25	12,160.0	19,841.5	−38.71

G.6 Static and heuristic baselines

GRASS. GRASS (Feng, 2020) is a static algorithm. After any change, it rebuilds the sparsifier from scratch. Figure 7 pits one full GRASS rebuild against a single update from our algorithm across 71 matched graphs ($n = 1{,}000$ to $64{,}000$). A rebuild takes a median of 0.18 seconds at $n = 1{,}000$ and climbs to 19.7 seconds at $n = 64{,}000$. Our updates take just 29 to 38 microseconds.

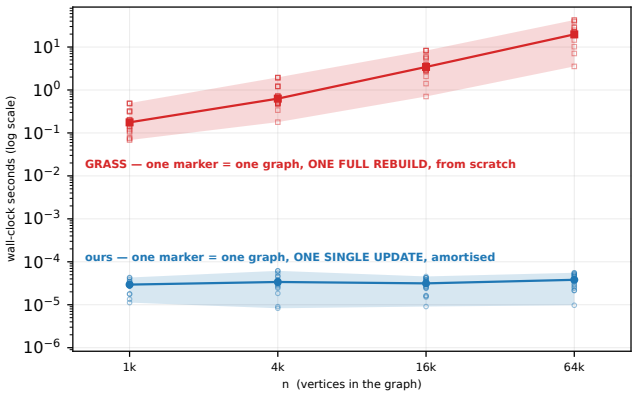

Figure 7: One full GRASS rebuild against one update of ours. The machine was not quiet during these runs

dyGRASS. dyGRASS (Yuan et al., 2025) offers no approximation guarantees, and targets a completely different operating point: tiny sparsifiers at loose quality. Figure 8 places both systems on the same graphs, measured by the same instrument. dyGRASS takes an extreme tradeoff: it keeps just 1.3 to 7.2% of the edges, but its certified κ blows up to between 70 and 154. None of its 60 states land inside our target band. Its dynamic runs only survived on the smallest ER and SBM graphs ($n = 1{,}000$). On larger or skewed graphs, its insertions simply overflowed its fixed memory arenas.

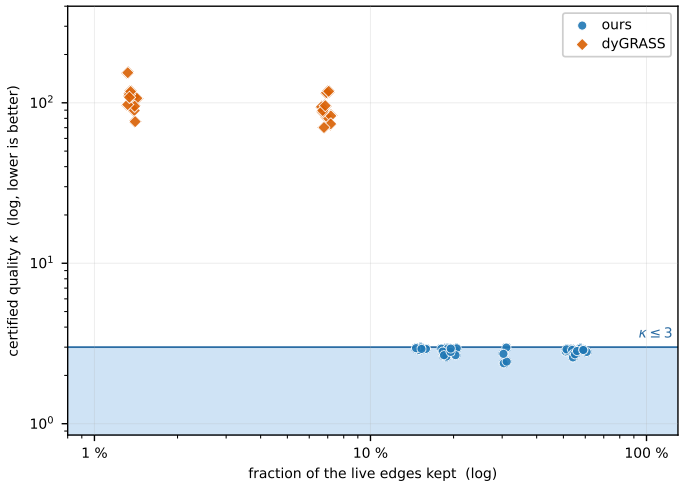

Figure 8: Certified quality against the fraction of live edges kept, one marker per cell, on the same graphs and the same instrument. The shaded region is $\kappa \leq 3$.

H LIMITATIONS

Certifying κ exactly is computationally expensive. We restrict certified quality claims to $n \leq 16{,}000$. We use synthetic streams even on timestamped real networks to maintain consistent evaluation conditions across the corpus. Real-world evaluations use dense k-cores because raw public networks are too sparse for meaningful compression. The prefactor P does not transfer across graph sizes and must be retuned for each n.

AI ASSISTANCE DISCLOSURE

Generative AI tools have been used for this paper. The source code of both algorithms was architected and designed by the authors. We used GitHub Copilot for code completion and Claude Code (Opus) to review the code, add comments, and look for potential errors. Claude Code was also used to write the benchmark scripts and the Python scripts used to generate the plots from our results, and to create an internal ledger system during the project development and benchmarking to keep track of everything smoothly. AI tools were also used to polish the grammar of the writing. All design decisions for the source code, benchmarks, and data analysis were made by the authors. AI-assisted code and text have been reviewed by the authors. We have not used AI to alter experimental results, and we take full responsibility for the content presented in this paper.